PiCore: A Rely-guarantee Framework for Concurrent Reactive Systems

Yongwang Zhao
zhaoyongwang@gmail.com, zhaoyw@buaa.edu.cn

January 12, 2020

Contents

theory *Heap*
imports *Main*
begin

0.1 References

definition *ref* = (*UNIV*::*nat set*)

typedef *ref* = *ref* **by** (*simp add*: *ref-def*)

code-datatype *Abs-ref*

lemma *finite-nat-ex-max*:
  assumes *fin*: *finite* (*N*::*nat set*)
  shows $\exists m. \forall n\in N. n < m$
using *fin*
proof (*induct*)
  case *empty*
  show *?case* **by** *auto*
next
  case (*insert k N*)
  have $\exists m. \forall n\in N. n < m$ **by** *fact*

```
  then obtain m where m-max: ∀ n∈N. n < m..
  show ∃ m. ∀ n∈insert k N. n < m
  proof (rule exI [where x=Suc (max k m)])
  qed (insert m-max, auto simp add: max-def)
qed

lemma infinite-nat: ¬finite (UNIV::nat set)
proof
  assume fin: finite (UNIV::nat set)
  then obtain m::nat where ∀ n∈UNIV. n < m
    by (rule finite-nat-ex-max [elim-format] ) auto
  moreover have m∈UNIV..
  ultimately show False by blast
qed

lemma infinite-ref [simp,intro]: ¬finite (UNIV::ref set)
proof
  assume finite (UNIV::ref set)
  hence finite (range Rep-ref)
    by simp
  moreover
  have range Rep-ref = ref
  proof
    show range Rep-ref ⊆ ref
      by (simp add: ref-def)
  next
    show ref ⊆ range Rep-ref
    proof
      fix x
      assume x: x ∈ ref
      show x ∈ range Rep-ref
        by (rule Rep-ref-induct) (auto simp add: ref-def)
    qed
  qed
  ultimately have finite ref
    by simp
  thus False
    by (simp add: ref-def infinite-nat)
qed

consts Null :: ref

definition new :: ref set ⇒ ref where
  new A = (SOME a. a ∉ {Null} ∪ A)
```

Constant *Null* can be defined later on. Conceptually *Null* and *new* are *fixes* of a locale with $finite\ A \Longrightarrow new\ A \notin A \cup \{Null\}$. But since definitions relative to a locale do not yet work in Isabelle2005 we use this workaround to avoid lots of parameters in definitions.

lemma *new-notin* [*simp*,*intro*]:
finite A $\Longrightarrow$ *new* (*A*) $\notin$ *A*
 apply (*unfold new-def*)
 apply (*rule someI2-ex*)
 apply (*fastforce intro*: *ex-new-if-finite*)
 apply *simp*
 done

lemma *new-not-Null* [*simp*,*intro*]:
 finite A $\Longrightarrow$ *new* (*A*) $\neq$ *Null*
 apply (*unfold new-def*)
 apply (*rule someI2-ex*)
 apply (*fastforce intro*: *ex-new-if-finite*)
 apply *simp*
done

end

1 Abstract Syntax of PiCore Language

theory *PiCore-Language*
imports *Main* **begin**

type-synonym (*'l*,*'s*,*'prog*) *event* = *'l* × (*'s set* × *'prog*)

definition *guard* :: (*'l*,*'s*,*'prog*) *event* ⇒ *'s set* **where**
 guard ev ≡ *fst* (*snd ev*)

definition *body* :: (*'l*,*'s*,*'prog*) *event* ⇒ *'prog* **where**
 body ev ≡ *snd* (*snd ev*)

datatype (*'l*,*'k*,*'s*,*'p*) *esys* =
 EAnon 'p
 | *EBasic* (*'l*,*'s*,*'p*) *event*
 | *EAtom* (*'l*,*'s*,*'p*) *event*
 | *ESeq* (*'l*,*'k*,*'s*,*'p*) *esys* (*'l*,*'k*,*'s*,*'p*) *esys* (- *NEXT* - [*81*,*81*] *80*)

 | *EChc* (*'l*,*'k*,*'s*,*'p*) *esys* (*'l*,*'k*,*'s*,*'p*) *esys* (- *OR* - [*81*,*81*] *80*)

 | *EJoin* (*'l*,*'k*,*'s*,*'p*) *esys* (*'l*,*'k*,*'s*,*'p*) *esys* (- ⋈ - [*81*,*81*] *80*)

 | *EWhile 's set* (*'l*,*'k*,*'s*,*'p*) *esys*

primrec *es-size* :: ‹(*'l*,*'k*,*'s*,*'p*) *esys* ⇒ *nat*› **where**
 ‹*es-size* (*EAnon* -) = *1*› |

‹*es-size* (*EBasic* -) = *1*› |
‹*es-size* (*EAtom* -) = *1*› |
‹*es-size* (*ESeq es1 es2*) = *Suc* (*es-size es1* + *es-size es2*)› |
‹*es-size* (*EChc es1 es2*) = *Suc* (*es-size es1* + *es-size es2*)› |
‹*es-size* (*EJoin es1 es2*) = *Suc* (*es-size es1* + *es-size es2*)› |
‹*es-size* (*EWhile* - *es*) = *Suc* (*es-size es*)›

type-synonym (*'l*,*'k*,*'s*,*'prog*) *paresys* = *'k* ⇒ (*'l*,*'k*,*'s*,*'prog*) *esys*

end

2 Small-step Operational Semantics of PiCore Language

theory *PiCore-Semantics*
imports *PiCore-Language*
begin

2.1 Datatypes for Semantics

datatype (*'l*,*'s*,*'prog*) *act* =
Cmd |
EvtEnt (*'l*,*'s*,*'prog*) *event* |
AtomEvt (*'l*,*'s*,*'prog*) *event*

record (*'l*,*'k*,*'s*,*'prog*) *actk* =
Act :: (*'l*,*'s*,*'prog*) *act*
K :: *'k*

abbreviation *mk-actk* :: (*'l*,*'s*,*'prog*) *act* ⇒ *'k* ⇒ (*'l*,*'k*,*'s*,*'prog*) *actk* (-♯- [*91*,*91*] *90*)
where *mk-actk a k* ≡ (|*Act*=*a*, *K*=*k*|)

lemma *actk-destruct*:
‹*a* = *Act a*♯*K a*› **by** *simp*

type-synonym (*'l*,*'k*,*'s*,*'prog*) *ectx* = *'k* ⇀ (*'l*,*'s*,*'prog*) *event*

type-synonym (*'s*,*'prog*) *pconf* = *'prog* × *'s*

type-synonym (*'s*,*'prog*) *pconfs* = (*'s*,*'prog*) *pconf list*

definition *getspc-p* :: (*'s*,*'prog*) *pconf* ⇒ *'prog* **where**
getspc-p conf ≡ *fst conf*

definition *gets-p* :: (*'s*,*'prog*) *pconf* ⇒ *'s* **where**
gets-p conf ≡ *snd conf*

type-synonym $('l,'k,'s,'prog)$ *esconf* = $('l,'k,'s,'prog)$ *esys* $\times$ $('s \times ('l,'k,'s,'prog)$ *ectx*)
type-synonym $('l,'k,'s,'prog)$ *pesconf* = $(('l,'k,'s,'prog)$ *paresys*) $\times$ $('s \times ('l,'k,'s,'prog)$ *ectx*)

locale *event* =
  fixes *ptran* :: *'Env* $\Rightarrow$ $(('s,'prog)$ *pconf* $\times$ $('s,'prog)$ *pconf*) *set*
  fixes *fin-com* :: *'prog*

  assumes *none-no-tran'*: ((*fin-com*, *s*),(P,t)) $\notin$ *ptran* Γ
  assumes *ptran-neq*: ((P, *s*),(P,t)) $\notin$ *ptran* Γ
begin

definition *ptran'* :: *'Env* $\Rightarrow$ $('s,'prog)$ *pconf* $\Rightarrow$ $('s,'prog)$ *pconf* $\Rightarrow$ *bool* (- $\vdash$ - $-c\rightarrow$ - [*81*,*81*] *80*)
  where $\Gamma \vdash P\ -c\rightarrow\ Q \equiv (P,Q) \in$ *ptran* Γ

declare *ptran'-def*[*simp*]

definition *ptrans* :: *'Env* $\Rightarrow$ $('s,'prog)$ *pconf* $\Rightarrow$ $('s,'prog)$ *pconf* $\Rightarrow$ *bool* (- $\vdash$ - $-c{*}\rightarrow$ - [*81*,*81*,*81*] *80*)
  where $\Gamma \vdash P\ -c{*}\rightarrow\ Q \equiv (P,Q) \in$(*ptran* Γ)^*

lemma *none-no-tran*: $\neg(\Gamma \vdash$ (*fin-com*,*s*) $-c\rightarrow$ $(P,t))$
  using *none-no-tran'* **by** *simp*

lemma *none-no-tran2*: $\neg(\Gamma \vdash$ (*fin-com*,*s*) $-c\rightarrow$ $Q)$
  using *none-no-tran* **by** (*metis prod.collapse*)

lemma *ptran-not-none*: $(\Gamma \vdash (Q,s)\ -c\rightarrow\ (P,t)) \Longrightarrow Q \neq$ *fin-com*
  using *none-no-tran* **apply** *simp* **by** *metis*

2.2 Semantics of Event Systems

abbreviation ‹*fin* ≡ *EAnon fin-com*›

inductive *estran-p* :: *'Env* $\Rightarrow$ $('l,'k,'s,'prog)$ *esconf* $\Rightarrow$ $('l,'k,'s,'prog)$ *actk* $\Rightarrow$ $('l,'k,'s,'prog)$ *esconf* $\Rightarrow$ *bool*
  (- $\vdash$ - $-es[-]\rightarrow$ - [*81*,*81*] *80*)
  where
    EAnon: $[\![\Gamma \vdash (P,\ s)\ -c\rightarrow\ (Q,\ t);\ Q \neq$ *fin-com* $]\!] \Longrightarrow$
        $\Gamma \vdash$ (*EAnon* P, s,x) $-es[Cmd\sharp k]\rightarrow$ (*EAnon* Q, t,x)
  | *EAnon-fin*: $[\![\ \Gamma \vdash (P,\ s)\ -c\rightarrow\ (Q,\ t);\ Q =$ *fin-com*; $y = x(k := None)\ ]\!] \Longrightarrow$
        $\Gamma \vdash$ (*EAnon* P, s,x) $-es[Cmd\sharp k]\rightarrow$ (*EAnon* Q, t, y)
  | *EBasic*: $[\![\ P =$ *body* e; $s \in$ *guard* e; $y = x(k{:=}Some\ e)\ ]\!] \Longrightarrow$

$\Gamma \vdash (EBasic\ e,\ s,x)\ -es[(EvtEnt\ e)\sharp k]\rightarrow\ ((EAnon\ P),\ s,y)$
| *EAtom*: $[\![\ P = body\ e;\ s \in guard\ e;\ \Gamma \vdash (P,s){-}c{*}{\rightarrow}(fin\text{-}com,t)\ ]\!] \Longrightarrow$
$\Gamma \vdash (EAtom\ e,\ s,x)\ -es[(AtomEvt\ e)\sharp k]\rightarrow\ (fin,\ t,x)$
| *ESeq*: $[\![\Gamma \vdash (es1,\ s,x)\ -es[a]\rightarrow(es1',\ t,y);\ es1' \neq fin\ ]\!] \Longrightarrow$
$\Gamma \vdash (ESeq\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (ESeq\ es1'\ es2,\ t,y)$
| *ESeq-fin*: $[\![\Gamma \vdash (es1,\ s,x)\ -es[a]\rightarrow(fin,\ t,y)]\!] \Longrightarrow$
$\Gamma \vdash (ESeq\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (es2,\ t,y)$

| *EChc1*: $\Gamma \vdash (es1,s,x)\ -es[a]\rightarrow\ (es1',t,y) \Longrightarrow$
$\Gamma \vdash (EChc\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (es1',\ t,y)$
| *EChc2*: $\Gamma \vdash (es2,s,x)\ -es[a]\rightarrow\ (es2',t,y) \Longrightarrow$
$\Gamma \vdash (EChc\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (es2',\ t,y)$

| *EJoin1*: $\Gamma \vdash (es1,s,x)\ -es[a]\rightarrow\ (es1',t,y) \Longrightarrow$
$\Gamma \vdash (EJoin\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (EJoin\ es1'\ es2,\ t,y)$
| *EJoin2*: $\Gamma \vdash (es2,s,x)\ -es[a]\rightarrow\ (es2',t,y) \Longrightarrow$
$\Gamma \vdash (EJoin\ es1\ es2,\ s,x)\ -es[a]\rightarrow\ (EJoin\ es1\ es2',\ t,y)$
| *EJoin-fin*: $\langle\Gamma \vdash (EJoin\ fin\ fin,\ s,x)\ -es[Cmd\sharp k]\rightarrow\ (fin,s,x)\rangle$
| *EWhileT*: $s{\in}b \Longrightarrow P \neq fin \Longrightarrow \Gamma \vdash (EWhile\ b\ P,\ s,x)\ -es[Cmd\sharp k]\rightarrow\ (ESeq\ P\ (EWhile\ b\ P),\ s,x)$
| *EWhileF*: $s{\notin}b \Longrightarrow \Gamma \vdash (EWhile\ b\ P,\ s,x)\ -es[Cmd\sharp k]\rightarrow\ (fin,\ s,x)$

primrec *Choice-height* :: $('l,'k,'s,'p)\ esys \Rightarrow nat$ **where**
Choice-height $(EAnon\ p) = 0$ |
Choice-height $(EBasic\ p) = 0$ |
Choice-height $(EAtom\ p) = 0$ |
Choice-height $(ESeq\ p\ q) = max\ (Choice\text{-}height\ p)\ (Choice\text{-}height\ q)$ |
Choice-height $(EChc\ p\ q) = Suc\ (max\ (Choice\text{-}height\ p)\ (Choice\text{-}height\ q))$ |
Choice-height $(EJoin\ p\ q) = max\ (Choice\text{-}height\ p)\ (Choice\text{-}height\ q)$ |
Choice-height $(EWhile\ \text{-}\ p) = Choice\text{-}height\ p$

primrec *Join-height* :: $('l,'k,'s,'p)\ esys \Rightarrow nat$ **where**
Join-height $(EAnon\ p) = 0$ |
Join-height $(EBasic\ p) = 0$ |
Join-height $(EAtom\ p) = 0$ |
Join-height $(ESeq\ p\ q) = max\ (Join\text{-}height\ p)\ (Join\text{-}height\ q)$ |
Join-height $(EChc\ p\ q) = max\ (Join\text{-}height\ p)\ (Join\text{-}height\ q)$ |
Join-height $(EJoin\ p\ q) = Suc\ (max\ (Join\text{-}height\ p)\ (Join\text{-}height\ q))$ |
Join-height $(EWhile\ \text{-}\ p) = Join\text{-}height\ p$

lemma *chcneq-specneq*: $Choice\text{-}height\ es1 \neq Choice\text{-}height\ es2 \Longrightarrow es1 \neq es2$
by *auto*

lemma *allneq-specneq*: $All\text{-}height\ es1 \neq All\text{-}height\ es2 \Longrightarrow es1 \neq es2$
by *auto*

inductive-cases *estran-from-basic-cases*: $\langle\Gamma \vdash (EBasic\ e,\ s)\ -es[a]\rightarrow\ (es,\ t)\rangle$

lemma *chc-hei-convg*: $\Gamma \vdash (es1,s)\ -es[a]\rightarrow\ (es2,t) \Longrightarrow Choice\text{-}height\ es1 \geq Choice\text{-}height$

es2
apply(*induct es1 arbitrary*: *es2 a s t*; *rule estran-p.cases*, *auto*)
by *fastforce*+

lemma *join-hei-convg*: $\Gamma \vdash (es1,s) -es[a]\rightarrow (es2,t) \Longrightarrow$ *Join-height es1* $\geq$ *Join-height es2*
apply (*induct es1 arbitrary*: *es2 a s t*; *rule estran-p.cases*, *auto*)
by *fastforce*+

lemma $\neg(\exists\, es2\ t\ a.\ \Gamma \vdash (es1,s) -es[a]\rightarrow (EChc\ es1\ es2,t))$
using *chc-hei-convg* **by** *fastforce*

lemma *seq-neq2*:
⟨*P NEXT Q* $\neq$ *Q*⟩
proof
assume ⟨*P NEXT Q* = *Q*⟩
then have ⟨*es-size* (*P NEXT Q*) = *es-size Q*⟩ **by** *simp*
then show *False* **by** *simp*
qed

lemma *join-neq1*: ⟨$P \bowtie Q \neq P$⟩ **by** (*induct P*) *auto*
lemma *join-neq2*: ⟨$P \bowtie Q \neq Q$⟩ **by** (*induct Q*) *auto*

lemma *spec-neq*: $\Gamma \vdash (es1,s,x) -es[a]\rightarrow (es2,t,y) \Longrightarrow es1 \neq es2$
proof(*induct es1 arbitrary*: *es2 s x t y a*)
case (*EAnon x*)
then show *?case* **apply**−
apply(*erule estran-p.cases*, *auto*) **using** *ptran-neq* **by** *simp*+
next
case (*EBasic x*)
then show *?case* **using** *estran-p.cases* **by** *fast*
next
case (*EAtom x*)
then show *?case* **using** *estran-p.cases* **by** *fast*
next
case (*ESeq es11 es12*)
then show *?case* **apply**−
apply(*erule estran-p.cases*, *auto*)
using *seq-neq2* **by** *blast*+
next
case (*EChc es11 es12*)
then show *?case* **apply**−
apply(*rule estran-p.cases*, *auto*)
proof−
assume ⟨$\Gamma \vdash$ (*es11*, *s*,*x*) $-es[a]\rightarrow$ (*es11 OR es12*, *t*,*y*)⟩
with *chc-hei-convg* **have** ⟨*Choice-height* (*es11 OR es12*) $\leq$ *Choice-height es11*⟩ **by** *blast*
then show *False* **by** *force*
next

```
    assume ‹Γ ⊢ (es12, s,x) −es[a]→ (es11 OR es12, t,y)›
    with chc-hei-convg have ‹Choice-height (es11 OR es12) ≤ Choice-height es12›
by blast
    then show False by force
  qed
next
  case (EJoin es11 es12)
  then show ?case apply−
    apply(rule estran-p.cases, auto)
    using join-neq2 apply blast
    apply blast.
next
  case EWhile
  then show ?case using estran-p.cases by fast
qed
```

2.3 Semantics of Parallel Event Systems

```
inductive
  pestran-p :: 'Env ⇒ ('l,'k,'s,'prog) pesconf ⇒ ('l,'k,'s,'prog) actk
                    ⇒ ('l,'k,'s,'prog) pesconf ⇒ bool  (- ⊢ - −pes[-]→ - [70,70] 60)
  where
    ParES: Γ ⊢ (pes k, s,x) −es[a♯k]→ (es', t,y) ⟹ Γ ⊢ (pes, s,x) −pes[a♯k]→
(pes(k:=es'), t,y)
```

2.4 Lemmas

2.4.1 Programs

```
lemma prog-not-eq-in-ctran-aux:
  assumes c: Γ ⊢ (P,s) −c→ (Q,t)
  shows P≠Q using c
  using ptran-neq apply simp apply auto
  done

lemma prog-not-eq-in-ctran [simp]: ¬ Γ ⊢ (P,s) −c→ (P,t)
  apply clarify using ptran-neq apply simp
  done
```

2.4.2 Event systems

```
lemma no-estran-to-self: ‹¬ Γ ⊢ (es, s,x) −es[a]→ (es, t,y)›
  using spec-neq by blast

lemma no-estran-from-fin:
  ‹¬ Γ ⊢ (EAnon fin-com, s) −es[a]→ c›
proof
  assume ‹Γ ⊢ (EAnon fin-com, s) −es[a]→ c›
  then show False
    apply(rule estran-p.cases, auto)
```

```
    using none-no-tran by simp+
qed

lemma no-pestran-to-self: ‹¬ Γ ⊢ (Ps, S) −pes[a]→ (Ps, T)›
proof(rule ccontr, simp)
  assume ‹Γ ⊢ (Ps, S) −pes[a]→ (Ps, T)›
  then show False
  proof(cases)
    case ParES
    then show ?thesis using no-estran-to-self
      by (metis fun-upd-same)
  qed
qed

definition ‹estran Γ ≡ {(c,c'). ∃ a. estran-p Γ c a c'}›
definition ‹pestran Γ ≡ {(c,c'). ∃ a k. pestran-p Γ c (a♯k) c'}›

lemma no-estran-to-self': ‹¬((P,S),(P,T))∈estran Γ›
  apply(simp add: estran-def)
  using no-estran-to-self surjective-pairing[of S] surjective-pairing[of T] by metis

lemma no-estran-to-self'': ‹fst c1 = fst c2 ⟹ (c1,c2)∉estran Γ›
  apply(subst surjective-pairing[of c1])
  apply(subst surjective-pairing[of c2])
  using no-estran-to-self' by metis

lemma no-pestran-to-self': ‹¬((P,s),(P,t))∈pestran Γ›
  apply(simp add: pestran-def)
  using no-pestran-to-self by blast

end

end
theory Computation imports Main begin

definition etran :: (('p×'s) × ('p×'s)) set where
  etran ≡ {(c,c'). fst c = fst c'}

declare etran-def[simp]

definition etran-p :: ‹('p×'s) ⇒ ('p×'s) ⇒ bool› (- −e→ - [81,81] 80)
  where ‹etran-p c c' ≡ (c,c') ∈ etran›

declare etran-p-def[simp]

inductive-set cpts :: ‹(('p×'s) × ('p×'s)) set ⇒ ('p×'s) list set›
  for tran :: (('p×'s) × ('p×'s)) set where
    CptsOne[intro]: [(P,s)] ∈ cpts tran |
    CptsEnv[intro]: (P,t)#cs ∈ cpts tran ⟹ (P,s)#(P,t)#cs ∈ cpts tran |
```

CptsComp: ⟦ $((P,s),(Q,t)) \in$ *tran*; $(Q,t)\#cs \in$ *cpts tran* ⟧ $\Longrightarrow (P,s)\#(Q,t)\#cs$ $\in$ *cpts tran*

lemma *cpts-snoc-env*:
  assumes *h*: *cpt* ∈ *cpts tran*
  assumes *tran*: ‹*last cpt* −*e*→ *c*›
  shows ‹*cpt*@[*c*] ∈ *cpts tran*›
  using *h tran*
proof(*induct*)
  case (*CptsOne P s*)
  then have ‹*fst c* = *P*› **by** *simp*
  then show *?case*
    apply(*subst surjective-pairing*[*of c*])
    apply(*erule ssubst*)
    apply *simp*
    apply(*rule CptsEnv*)
    apply(*rule cpts.CptsOne*)
    done
next
  case (*CptsEnv P t cs s*)
  then have ‹*last* ((*P*, *t*) # *cs*) −*e*→ *c*› **by** *simp*
  with *CptsEnv*(*2*) **have** ‹((*P*, *t*) # *cs*) @ [*c*] ∈ *cpts tran*› **by** *blast*
  then show *?case* **using** *cpts.CptsEnv* **by** *fastforce*
next
  case (*CptsComp P s Q t cs*)
  then have ‹((*Q*, *t*) # *cs*) @ [*c*] ∈ *cpts tran*› **by** *fastforce*
  with *CptsComp*(*1*) **show** *?case* **using** *cpts.CptsComp* **by** *fastforce*
qed

lemma *cpts-snoc-comp*:
  assumes *h*: *cpt* ∈ *cpts tran*
  assumes *tran*: ‹(*last cpt*, *c*) ∈ *tran*›
  shows ‹*cpt*@[*c*] ∈ *cpts tran*›
  using *h tran*
proof(*induct*)
  case (*CptsOne P s*)
  then show *?case* **apply** *simp*
    apply(*subst* (*asm*) *surjective-pairing*[*of c*])
    apply(*subst surjective-pairing*[*of c*])
    apply(*rule CptsComp*)
     apply *simp*
    apply(*rule cpts.CptsOne*)
    done
next
  case (*CptsEnv P t cs s*)
  then have ‹((*P*, *t*) # *cs*) @ [*c*] ∈ *cpts tran*› **by** *fastforce*
  then show *?case* **using** *cpts.CptsEnv* **by** *fastforce*
next
  case (*CptsComp P s Q t cs*)

```
  then have ‹((Q, t) # cs) @ [c] ∈ cpts tran› by fastforce
  with CptsComp(1) show ?case using cpts.CptsComp by fastforce
qed

lemma cpts-nonnil:
  assumes h: ‹cpt ∈ cpts tran›
  shows ‹cpt ≠ []›
  using h by (induct; simp)

lemma cpts-def': ‹cpt ∈ cpts tran ⟷ cpt ≠ [] ∧ (∀i. Suc i < length cpt ⟶
(cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i −e→ cpt!Suc i)›
proof
  assume cpt: ‹cpt ∈ cpts tran›
  show ‹cpt ≠ [] ∧ (∀i. Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i
−e→ cpt!Suc i)›
  proof
    show ‹cpt ≠ []› by (rule cpts-nonnil[OF cpt])
  next
   show ‹∀i. Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i −e→ cpt!Suc
i›
    proof
      fix i
      show ‹Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i −e→ cpt!Suc
i›
      proof
        assume i-lt: ‹Suc i < length cpt›
        show ‹(cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i −e→ cpt!Suc i›
          using cpt i-lt
        proof(induct arbitrary:i)
          case (CptsOne P s)
          then show ?case by simp
        next
          case (CptsEnv P t cs s)
          show ?case
          proof(cases i)
            case 0
            then show ?thesis apply−
              apply(rule disjI2)
              apply(erule ssubst)
              apply simp
              done
          next
            case (Suc i')
            then show ?thesis using CptsEnv(2)[of i'] CptsEnv(3) by force
          qed
        next
          case (CptsComp P s Q t cs)
          show ?case
          proof(cases i)
```

```
          case 0
          then show ?thesis apply−
            apply(rule disjI1)
            apply(erule ssubst)
            apply simp
            by (rule CptsComp(1))
        next
          case (Suc i′)
          then show ?thesis using CptsComp(3)[of i′] CptsComp(4) by force
        qed
      qed
    qed
  qed
qed
next
  assume h: ‹cpt ≠ [] ∧ (∀ i. Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ tran ∨
cpt!i −e→ cpt!Suc i)›
  from h have cpt-nonnil: ‹cpt ≠ []› by (rule conjunct1)
  from h have ct-et: ‹∀ i. Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ tran ∨ cpt!i
−e→ cpt!Suc i› by (rule conjunct2)
  show ‹cpt ∈ cpts tran› using cpt-nonnil ct-et
  proof(induct cpt)
    case Nil
    then show ?case by simp
  next
    case (Cons c cs)
    have IH: ‹cs ≠ [] ⟹ ∀ i. Suc i < length cs ⟶ (cs ! i, cs ! Suc i) ∈ tran ∨
cs ! i −e→ cs ! Suc i ⟹ cs ∈ cpts tran›
      by (rule Cons(1))
    have ct-et′: ‹∀ i. Suc i < length (c # cs) ⟶ ((c # cs) ! i, (c # cs) ! Suc i)
∈ tran ∨ (c # cs) ! i −e→ (c # cs) ! Suc i›
      by (rule Cons(3))
    show ?case
    proof(cases cs)
      case Nil
      then show ?thesis apply−
        apply(erule ssubst)
        apply(subst surjective-pairing[of c])
        by (rule CptsOne)
    next
      case (Cons c′ cs′)
      then have ‹cs ≠ []› by simp
      moreover have ‹∀ i. Suc i < length cs ⟶ (cs ! i, cs ! Suc i) ∈ tran ∨ cs !
i −e→ cs ! Suc i›
        using ct-et′ by auto
      ultimately have cs-cpts: ‹cs ∈ cpts tran› using IH by fast
      show ?thesis apply (rule ct-et′[THEN allE, of 0])
        apply(simp add: Cons)
      proof−
```

assume ‹$(c,\ c') \in tran \lor fst\ c = fst\ c'$›
then show ‹$c\ \#\ c'\ \#\ cs' \in cpts\ tran$›
proof
assume h: ‹$(c,\ c') \in tran$›
show ‹$c\ \#\ c'\ \#\ cs' \in cpts\ tran$›
apply(*subst surjective-pairing*[*of c*])
apply(*subst surjective-pairing*[*of c'*])
apply(*rule CptsComp*)
apply *simp*
apply (*rule h*)
using *cs-cpts* **by** (*simp add*: *Cons*)
next
assume h: ‹$fst\ c = fst\ c'$›
show ‹$c\ \#\ c'\ \#\ cs' \in cpts\ tran$›
apply(*subst surjective-pairing*[*of c*])
apply(*subst surjective-pairing*[*of c'*])
apply(*subst h*)
apply(*rule CptsEnv*)
apply *simp*
using *cs-cpts* **by** (*simp add*: *Cons*)
qed
qed
qed
qed
qed

lemma *cpts-tran*:
‹$cpt \in cpts\ tran \Longrightarrow$
$\forall i.\ Suc\ i < length\ cpt \longrightarrow$
$(cpt!i,\ cpt!Suc\ i) \in tran \lor cpt!i\ {-e\rightarrow}\ cpt!Suc\ i$›
using *cpts-def'* **by** *blast*

definition *cpts-from* :: ‹$(('p\times's) \times ('p\times's))\ set \Rightarrow ('p\times's) \Rightarrow ('p\times's)\ list\ set$›
where
$cpts\text{-}from\ tran\ c0 \equiv \{cpt.\ cpt \in cpts\ tran \land hd\ cpt = c0\}$

declare *cpts-from-def*[*simp*]

lemma *cpts-from-def'*:
$cpt \in cpts\text{-}from\ tran\ c0 \longleftrightarrow cpt \in cpts\ tran \land hd\ cpt = c0$ **by** *simp*

definition *cpts-from-ctran-only* :: ‹$(('p\times's) \times ('p\times's))\ set \Rightarrow ('p\times's) \Rightarrow ('p\times's)\ list\ set$› **where**
$cpts\text{-}from\text{-}ctran\text{-}only\ tran\ c0 \equiv \{cpt.\ cpt \in cpts\text{-}from\ tran\ c0 \land (\forall i.\ Suc\ i < length\ cpt \longrightarrow (cpt!i,\ cpt!Suc\ i) \in tran)\}$

lemma *cpts-tl'*:
assumes h: ‹$cpt \in cpts\ tran$›
and cpt: ‹$cpt = c0\#c1\#cs$›

```
  shows c1#cs ∈ cpts tran
  using h cpt apply− apply(erule cpts.cases, auto) done

lemma cpts-tl:
  ‹cpt ∈ cpts tran ⟹ tl cpt ≠ [] ⟹ tl cpt ∈ cpts tran›
  using cpts-tl′ by (metis cpts-nonnil list.exhaust-sel)

lemma cpts-from-tl:
  assumes h: ‹cpt ∈ cpts-from tran (P,s)›
    and cpt: ‹cpt = (P,s)#(P,t)#cs›
  shows (P,t)#cs ∈ cpts-from tran (P,t)
proof−
  from h have cpt ∈ cpts tran by simp
  with cpt show ?thesis apply− apply(erule cpts.cases, auto) done
qed

lemma cpts-drop:
  assumes h: cpt ∈ cpts tran
    and i: i < length cpt
  shows drop i cpt ∈ cpts tran
  using i
proof(induct i)
  case 0
  then show ?case using h by simp
next
  case (Suc i′)
  then show ?case
  proof−
    assume h1: ‹i′ < length cpt ⟹ drop i′ cpt ∈ cpts tran›
    assume h2: ‹(Suc i′) < length cpt›
    with h1 have ‹drop i′ cpt ∈ cpts tran› by fastforce
    let ?cpt′ = ‹drop i′ cpt›
    have ‹drop (Suc i′) cpt = tl ?cpt′›
      by (simp add: drop-Suc drop-tl)
    with h2 have ‹tl ?cpt′ ≠ []› by auto
    then show ‹drop (Suc i′) cpt ∈ cpts tran› using cpts-tl[of ?cpt′]
      by (simp add: ‹drop (Suc i′) cpt = tl (drop i′ cpt)› ‹drop i′ cpt ∈ cpts tran›
cpts-tl)
  qed
qed

lemma cpts-take′:
  assumes h: cpt ∈ cpts tran
  shows take (Suc i) cpt ∈ cpts tran
  using h
proof(induct i)
  case 0
  have [(fst (hd cpt), snd (hd cpt))] ∈ cpts tran using CptsOne by fast
  then show ?case
```

```
    using 0.prems cpts-def' by fastforce
next
  case (Suc i)
  then have cpt': ‹take (Suc i) cpt ∈ cpts tran› by blast
  let ?cpt' = take (Suc i) cpt
  show ?case
  proof(cases ‹Suc i < length cpt›)
    case True
    with cpts-drop have drop-i: ‹drop i cpt ∈ cpts tran›
      using Suc-lessD h by blast
    have ‹?cpt' @ [cpt!Suc i] ∈ cpts tran› using drop-i
    proof(cases)
      case (CptsOne P s)
      then show ?thesis using h
      by (metis Cons-nth-drop-Suc Suc-lessD True append.right-neutral append-eq-append-conv
append-take-drop-id list.simps(3) nth-via-drop take-Suc-conv-app-nth)
    next
      case (CptsEnv P t cs s)
      then show ?thesis apply−
        apply(rule cpts-snoc-env)
        apply(rule cpt')
      proof−
        assume h1: ‹drop i cpt = (P, s) # (P, t) # cs›
        assume h2: ‹(P, t) # cs ∈ cpts tran›
        from h1 h2 have ‹last (take (Suc i) cpt) = (P, s)›
            by (metis Suc-lessD True hd-drop-conv-nth list.sel(1) snoc-eq-iff-butlast
take-Suc-conv-app-nth)
        moreover from h1 h2 have cpt!Suc i = (P,t)
          by (metis Cons-nth-drop-Suc Suc-lessD True list.sel(1) list.sel(3))
        ultimately show ‹last (take (Suc i) cpt) −e→ cpt ! Suc i› by force
      qed
    next
      case (CptsComp P s Q t cs)
      then show ?thesis apply−
        apply(rule cpts-snoc-comp)
         apply(rule cpt')
      proof−
        assume h1: ‹drop i cpt = (P, s) # (Q, t) # cs›
        assume h2: ‹(Q, t) # cs ∈ cpts tran›
        assume h3: ‹((P, s), (Q, t)) ∈ tran›
        from h1 h2 have ‹last (take (Suc i) cpt) = (P, s)›
            by (metis Suc-lessD True hd-drop-conv-nth list.sel(1) snoc-eq-iff-butlast
take-Suc-conv-app-nth)
        moreover from h1 h2 have cpt!Suc i = (Q,t)
          by (metis Cons-nth-drop-Suc Suc-lessD True list.sel(1) list.sel(3))
         ultimately show ‹(last (take (Suc i) cpt), cpt ! Suc i) ∈ tran› using h3
by simp
       qed
    qed
```

```
    with True show ?thesis
      by (simp add: take-Suc-conv-app-nth)
  next
    case False
    then show ?thesis using cpt' by simp
  qed
qed

lemma cpts-take:
  assumes h: cpt ∈ cpts tran
  assumes i: i ≠ 0
  shows take i cpt ∈ cpts tran
proof−
  from i obtain i' where ‹i = Suc i'› using not0-implies-Suc by blast
  with h cpts-take' show ?thesis by blast
qed

lemma cpts-from-take:
  assumes h: cpt ∈ cpts-from tran c
  assumes i: i ≠ 0
  shows take i cpt ∈ cpts-from tran c
  apply simp
proof
  from h have cpt ∈ cpts tran by simp
  with i cpts-take show ‹take i cpt ∈ cpts tran› by blast
next
  from h have hd cpt = c by simp
  with i show ‹hd (take i cpt) = c› by simp
qed

type-synonym 'a tran = ‹'a × 'a›

lemma cpts-prepend:
  ‹[c0,c1]∈cpts tran ⟹ c1#cs ∈ cpts tran ⟹ c0#c1#cs ∈ cpts tran›
  apply(erule cpts.cases, auto)
  apply(rule CptsComp, auto)
  done

lemma all-etran-same-prog:
  assumes all-etran: ‹∀ i. Suc i < length cpt ⟶ cpt!i −e→ cpt!Suc i›
    and fst-hd-cpt: ‹fst (hd cpt) = P›
    and ‹cpt≠[]›
  shows ‹∀ i<length cpt. fst (cpt!i) = P›
proof
  fix i
  show ‹i < length cpt ⟶ fst (cpt ! i) = P›
  proof(induct i)
    case 0
    then show ?case
```

```
      apply(rule impI)
      apply(subst hd-conv-nth[THEN sym])
       apply(rule ‹cpt≠[]›)
      apply(rule fst-hd-cpt)
      done
  next
    case (Suc i)
    have 1: Suc i < length cpt ⟶ cpt ! i −e→ cpt ! Suc i
      by (rule all-etran[THEN spec[where x=i]])
    show ?case
    proof
      assume Suc-i-lt: ‹Suc i < length cpt›
      with 1 have ‹cpt ! i −e→ cpt ! Suc i› by blast
      moreover from Suc Suc-i-lt[THEN Suc-lessD] have ‹fst (cpt ! i) = P› by blast
      ultimately show ‹fst (cpt ! Suc i) = P› by simp
    qed
  qed
qed

lemma cpts-append-comp:
  ‹cs1 ∈ cpts tran ⟹ cs2 ∈ cpts tran ⟹ (last cs1, hd cs2) ∈ tran ⟹ cs1@cs2 ∈ cpts tran›
proof−
  assume c1: ‹cs1∈cpts tran›
  assume c2: ‹cs2∈cpts tran›
  assume tran: ‹(last cs1, hd cs2) ∈ tran›
  show ?thesis using c1 tran
  proof(induct)
    case (CptsOne P s)
    then show ?case
      apply simp
      apply(cases cs2)
      using cpts-nonnil c2 apply fast
      apply simp
      apply(rename-tac c cs)
      apply(subst surjective-pairing[of c])
      apply(rule CptsComp)
       apply simp
      using c2 by simp
  next
    case (CptsEnv P t cs s)
    then show ?case
      apply simp
      apply(rule cpts.CptsEnv)
      by simp
  next
    case (CptsComp P s Q t cs)
    then show ?case
```

```
      apply simp
      apply(rule cpts.CptsComp)
      apply blast
      by blast
  qed
qed

lemma cpts-append-env:
  assumes c1: ‹cs1∈cpts tran› and c2: ‹cs2∈cpts tran›
    and etran: ‹fst (last cs1) = fst (hd cs2)›
  shows ‹cs1@cs2 ∈ cpts tran›
  using c1 etran
proof(induct)
  case (CptsOne P s)
  then show ?case
    apply simp
    apply(subst hd-Cons-tl[OF cpts-nonnil[OF c2], symmetric]) back
    apply(subst surjective-pairing[of ‹hd cs2›]) back
    apply(rule CptsEnv)
    using hd-Cons-tl[OF cpts-nonnil[OF c2]] c2 by simp
next
  case (CptsEnv P t cs s)
  then show ?case
    apply simp
    apply(rule cpts.CptsEnv)
    by simp
next
  case (CptsComp P s Q t cs)
  then show ?case
    apply simp
    apply(rule cpts.CptsComp)
     apply blast
    by blast
qed

lemma cpts-remove-last:
  assumes ‹c#cs@[c'] ∈ cpts tran›
  shows ‹c#cs ∈ cpts tran›
proof–
  from assms cpts-def' have 1: ‹∀ i. Suc i < length (c#cs@[c']) ⟶ ((c#cs@[c'])
! i, (c#cs@[c']) ! Suc i) ∈ tran ∨ (c#cs@[c']) ! i −e→ (c#cs@[c']) ! Suc i› by
blast
   have ‹∀ i. Suc i < length (c#cs) ⟶ ((c#cs) ! i, (c#cs) ! Suc i) ∈ tran ∨
(c#cs) ! i −e→ (c#cs) ! Suc i› (is ‹∀ i. ?P i›)
  proof
    fix i
    show ‹?P i›
    proof
      assume Suc-i-lt: ‹Suc i < length (c # cs)›
```

show ‹((*c* # *cs*) ! *i*, (*c* # *cs*) ! *Suc i*) ∈ *tran* ∨ (*c* # *cs*) ! *i* −*e*→ (*c* # *cs*) ! *Suc i*›
using *1*[*THEN spec*[**where** *x*=*i*]] *Suc-i-lt*
by (*metis* (*no-types*, *hide-lams*) *Suc-lessD Suc-less-eq Suc-mono append-Cons length-Cons length-append-singleton nth-Cons-Suc nth-butlast snoc-eq-iff-butlast*)
qed
qed
then show *?thesis* **using** *cpts-def'* **by** *blast*
qed

lemma *cpts-append*:
assumes *a1*: ‹*cs*@[*c*] ∈ *cpts tran*›
and *a2*: ‹*c*#*cs'* ∈ *cpts tran*›
shows ‹*cs*@*c*#*cs'* ∈ *cpts tran*›
proof−
from *a1 cpts-def'* **have** *a1'*: ‹∀ *i*. *Suc i* < *length* (*cs*@[*c*]) ⟶ ((*cs*@[*c*]) ! *i*, (*cs*@[*c*]) ! *Suc i*) ∈ *tran* ∨ (*cs*@[*c*]) ! *i* −*e*→ (*cs*@[*c*]) ! *Suc i*› **by** *blast*
from *a2 cpts-def'* **have** *a2'*: ‹∀ *i*. *Suc i* < *length* (*c*#*cs'*) ⟶ ((*c*#*cs'*) ! *i*, (*c*#*cs'*) ! *Suc i*) ∈ *tran* ∨ (*c*#*cs'*) ! *i* −*e*→ (*c*#*cs'*) ! *Suc i*› **by** *blast*
have ‹∀ *i*. *Suc i* < *length* (*cs*@*c*#*cs'*) ⟶ ((*cs*@*c*#*cs'*) ! *i*, (*cs*@*c*#*cs'*) ! *Suc i*) ∈ *tran* ∨ (*cs*@*c*#*cs'*) ! *i* −*e*→ (*cs*@*c*#*cs'*) ! *Suc i*›
proof
fix *i*
show ‹*Suc i* < *length* (*cs*@*c*#*cs'*) ⟶ ((*cs*@*c*#*cs'*) ! *i*, (*cs*@*c*#*cs'*) ! *Suc i*) ∈ *tran* ∨ (*cs*@*c*#*cs'*) ! *i* −*e*→ (*cs*@*c*#*cs'*) ! *Suc i*›
proof
assume *Suc-i-lt*: ‹*Suc i* < *length* (*cs*@*c*#*cs'*)›
show ‹((*cs*@*c*#*cs'*) ! *i*, (*cs*@*c*#*cs'*) ! *Suc i*) ∈ *tran* ∨ (*cs*@*c*#*cs'*) ! *i* −*e*→ (*cs*@*c*#*cs'*) ! *Suc i*›
proof(*cases* ‹*Suc i* < *length* (*cs*@[*c*])›)
case *True*
with *a1'*[*THEN spec*[**where** *x*=*i*]] **show** *?thesis*
by (*metis Suc-less-eq length-append-singleton less-antisym nth-append nth-append-length*)
next
case *False*
with *a2'*[*THEN spec*[**where** *x*=*i* − *length cs*]] **show** *?thesis*

by (*smt Suc-diff-Suc Suc-i-lt Suc-lessD add-diff-cancel-left' diff-Suc-Suc diff-less-mono length-append length-append-singleton less-Suc-eq-le not-less-eq nth-append*)
qed
qed
qed
with *cpts-def'* **show** *?thesis* **by** *blast*
qed

end
theory *List-Lemmata* **imports** *Main* **begin**

lemma *last-take-Suc*:
  i < *length l* ⟹ *last* (*take* (*Suc i*) *l*) = *l*!*i*
  by (*simp add*: *take-Suc-conv-app-nth*)

lemma *list-eq*: (*length xs* = *length ys* ∧ (∀ *i*<*length xs*. *xs*!*i*=*ys*!*i*)) = (*xs*=*ys*)
  apply(*rule iffI*)
  apply *clarify*
  apply(*erule nth-equalityI*)
  apply *simp*+
  done

lemma *nth-tl*: ⟦ *ys*!*0*=*a*; *ys*≠[] ⟧ ⟹ *ys*=(*a*#(*tl ys*))
  by (*cases ys*) *simp-all*

lemma *nth-tl-if* [*rule-format*]: *ys*≠[] ⟶ *ys*!*0*=*a* ⟶ *P ys* ⟶ *P* (*a*#(*tl ys*))
  by (*induct ys*) *simp-all*

lemma *nth-tl-onlyif* [*rule-format*]: *ys*≠[] ⟶ *ys*!*0*=*a* ⟶ *P* (*a*#(*tl ys*)) ⟶ *P ys*
  by (*induct ys*) *simp-all*

lemma *drop-destruct*:
  ‹*Suc n* ≤ *length xs* ⟹ *drop n xs* = *hd* (*drop n xs*) # *drop* (*Suc n*) *xs*›
  by (*metis drop-Suc drop-eq-Nil hd-Cons-tl not-less-eq-eq tl-drop*)

lemma *drop-last*:
  ‹*xs* ≠ [] ⟹ *drop* (*length xs* − *1*) *xs* = [*last xs*]›
  by (*metis append-butlast-last-id append-eq-conv-conj length-butlast*)

end

3 Computations of PiCore Language

theory *PiCore-Computation*
  imports *PiCore-Semantics Computation List-Lemmata*
begin

type-synonym (′*l*,′*k*,′*s*,′*prog*) *escpt* = ‹((′*l*,′*k*,′*s*,′*prog*) *esconf*) *list*›

locale *event-comp* = *event ptran fin-com*
  for *ptran* :: ′*Env* ⇒ ((′*s*,′*prog*) *pconf* × (′*s*,′*prog*) *pconf*) *set*
    and *fin-com* :: ′*prog*

begin

inductive-cases *estran-from-anon-cases*: ‹Γ ⊢ (*EAnon p*, *S*) −*es*[*a*]→ *c*›

lemma *cpts-from-anon*:
  assumes *h*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAnon p0*, *s0*,*x0*)›

shows ‹∀ i. i < *length cpt* ⟶ (∃ p. *fst*(*cpt*!i) = *EAnon p*)›
proof
from h **have** *cpt-nonnil*: *cpt* ≠ [] **using** *cpts-nonnil* **by** *auto*
from h **have** *h1*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **by** *fastforce*
from h **have** *h2*: ‹*hd cpt* = (*EAnon p0*, *s0*,*x0*)› **by** *auto*
fix i
show ‹i < *length cpt* ⟶ (∃ p. *fst*(*cpt*!i) = *EAnon p*)›
proof
assume *i-lt*: ‹i < *length cpt*›
show ‹(∃ p. *fst*(*cpt*!i) = *EAnon p*)›
using *i-lt*
proof(*induct i*)
case *0*
from h **have** *hd cpt* = (*EAnon p0*, *s0*,*x0*) **by** *simp*
then show *?case* **using** *hd-conv-nth cpt-nonnil* **by** *fastforce*
next
case (*Suc i'*)
then obtain p **where** *fst-cpt-i'*: *fst*(*cpt*!*i'*) = (*EAnon p*) **by** *fastforce*
have ‹(*cpt*!*i'*, *cpt*!(*Suc i'*)) ∈ *estran* Γ ∨ *cpt*!*i'* −*e*→ *cpt*!(*Suc i'*)›
using *cpts-tran h1 Suc*(*2*) **by** *blast*
then show *?case*
proof
assume ‹(*cpt* ! *i'*, *cpt* ! *Suc i'*) ∈ *estran* Γ›
then show *?thesis*
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*subst*(*asm*) *surjective-pairing*[*of* ‹*cpt*!*i'*›])
apply(*subst*(*asm*) *fst-cpt-i'*)
apply(*erule estran-from-anon-cases*)
by *simp+*
next
assume ‹*cpt* ! *i'* −*e*→ *cpt* ! *Suc i'*›
then show *?thesis*
apply *simp*
using *fst-cpt-i'* **by** *metis*
qed
qed
qed
qed

lemma *cpts-from-anon'*:
assumes h: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAnon p0*, *s0*)›
shows ‹∀ i. i < *length cpt* ⟶ (∃ *p s x*. *cpt*!i = (*EAnon p*, *s*, *x*))›
using *cpts-from-anon* **by** (*metis h prod.collapse*)

primrec (*nonexhaustive*) *unlift-prog* **where**
‹*unlift-prog* (*EAnon p*) = *p*›

definition ‹*unlift-conf* ≡ λ(*p*,*s*,-). (*unlift-prog p*, *s*)›

definition *unlift-cpt* :: ‹(('l, 'k, 's, 'prog) esconf) list ⇒ ('prog × 's) list› **where**
‹unlift-cpt ≡ map unlift-conf›
declare *unlift-conf-def*[*simp*] *unlift-cpt-def*[*simp*]

definition *lift-conf* :: ('l,'k,'s,'prog) ectx ⇒ ('prog×'s) ⇒ (('l,'k,'s,'prog) esconf) **where**
‹lift-conf x ≡ λ(p,s). (EAnon p, s,x)›

declare *lift-conf-def*[*simp*]

lemma *lift-conf-def'*: ‹lift-conf x (p, s) = (EAnon p, s,x)› **by** *simp*

definition *lift-cpt* :: ('l,'k,'s,'prog) ectx ⇒ ('prog×'s) list ⇒ (('l,'k,'s,'prog) esconf) list **where**
‹lift-cpt x ≡ map (lift-conf x)›

declare *lift-cpt-def*[*simp*]

inductive-cases *estran-anon-to-anon-cases*: ‹Γ ⊢ (EAnon p, s,x) −es[a]→ (EAnon q, t,y)›

lemma *unlift-tran*: ‹((EAnon p, s,x), (EAnon q, t,x)) ∈ estran Γ ⟹ ((p,s),(q,t)) ∈ ptran Γ›
apply(*simp add*: *case-prod-unfold estran-def*)
apply(*erule exE*)
apply(*erule estran-anon-to-anon-cases*)
apply *simp+*
done

lemma *unlift-tran'*: ‹(lift-conf x c, lift-conf x c') ∈ estran Γ ⟹ (c, c') ∈ ptran Γ›
apply (*simp add*: *case-prod-unfold*)
apply(*subst surjective-pairing*[*of c*])
apply(*subst surjective-pairing*[*of c'*])
using *unlift-tran* **by** *fastforce*

lemma *cpt-unlift-aux*:
‹((EAnon p0, s0,x), Q, t,y) ∈ estran Γ ⟹ ∃ Q'. Q = EAnon Q' ∧ ((p0,s0),(Q',t)) ∈ ptran Γ›
by (*simp add*: *estran-def*, *erule exE*, *erule estran-p.cases*, *auto*)

lemma *ctran-or-etran*:
‹cpt ∈ cpts (estran Γ) ⟹
Suc i < length cpt ⟹
(cpt!i, cpt!Suc i) ∈ estran Γ ∧ (¬ cpt!i −e→ cpt!Suc i) ∨
(cpt!i −e→ cpt!Suc i) ∧ (cpt!i, cpt!Suc i) ∉ estran Γ›
proof−
assume *cpt*: ‹cpt ∈ cpts (estran Γ)›
assume *Suc-i-lt*: ‹Suc i < length cpt›

```
  from cpts-drop[OF cpt Suc-i-lt[THEN Suc-lessD]] have
    ‹drop i cpt ∈ cpts (estran Γ)› by assumption
  then show
    ‹(cpt!i, cpt!Suc i) ∈ estran Γ ∧ (¬ cpt!i −e→ cpt!Suc i) ∨
     (cpt!i −e→ cpt!Suc i) ∧ (cpt!i, cpt!Suc i) ∉ estran Γ›
  proof(cases)
    case (CptsOne P s)
    then have False
      by (metis (no-types, lifting) Cons-nth-drop-Suc Suc-i-lt Suc-lessD drop-eq-Nil
list.inject not-less)
    then show ?thesis by blast
  next
    case (CptsEnv P t cs s)
    from nth-via-drop[OF CptsEnv(1)] have ‹cpt!i = (P,s)› by assumption
    moreover from CptsEnv(1) have ‹cpt!Suc i = (P,t)›
      by (metis Suc-i-lt drop-Suc hd-drop-conv-nth list.sel(1) list.sel(3) tl-drop)
    ultimately show ?thesis
      by (simp add: no-estran-to-self′)
  next
    case (CptsComp P s Q t cs)
    from nth-via-drop[OF CptsComp(1)] have ‹cpt!i = (P,s)› by assumption
    moreover from CptsComp(1) have ‹cpt!Suc i = (Q,t)›
      by (metis Suc-i-lt drop-Suc hd-drop-conv-nth list.sel(1) list.sel(3) tl-drop)
    ultimately show ?thesis
      apply simp
      apply(rule disjI1)
      apply(rule conjI)
       apply(rule CptsComp(2))
      using CptsComp(2) no-estran-to-self′ by blast
  qed
qed

lemma ctran-or-etran-par:
  ‹cpt ∈ cpts (pestran Γ) ⟹
   Suc i < length cpt ⟹
   (cpt!i, cpt!Suc i) ∈ pestran Γ ∧ (¬ cpt!i −e→ cpt!Suc i) ∨
   (cpt!i −e→ cpt!Suc i) ∧ (cpt!i, cpt!Suc i) ∉ pestran Γ›
proof−
  assume cpt: ‹cpt ∈ cpts (pestran Γ)›
  assume Suc-i-lt: ‹Suc i < length cpt›
  from cpts-drop[OF cpt Suc-i-lt[THEN Suc-lessD]] have
    ‹drop i cpt ∈ cpts (pestran Γ)› by assumption
  then show
    ‹(cpt!i, cpt!Suc i) ∈ pestran Γ ∧ (¬ cpt!i −e→ cpt!Suc i) ∨
     (cpt!i −e→ cpt!Suc i) ∧ (cpt!i, cpt!Suc i) ∉ pestran Γ›
  proof(cases)
    case (CptsOne P s)
    then have False using Suc-i-lt
      by (metis Cons-nth-drop-Suc drop-Suc drop-tl list.sel(3) list.simps(3))
```

```
    then show ?thesis by blast
  next
    case (CptsEnv P t cs s)
    from nth-via-drop[OF CptsEnv(1)] have ‹cpt!i = (P,s)› by assumption
    moreover from CptsEnv(1) have ‹cpt!Suc i = (P,t)›
      by (metis Suc-i-lt drop-Suc hd-drop-conv-nth list.sel(1) list.sel(3) tl-drop)
    ultimately show ?thesis
      using no-pestran-to-self
      by (simp add: no-pestran-to-self')
  next
    case (CptsComp P s Q t cs)
    from nth-via-drop[OF CptsComp(1)] have ‹cpt!i = (P,s)› by assumption
    moreover from CptsComp(1) have ‹cpt!Suc i = (Q,t)›
      by (metis Suc-i-lt drop-Suc hd-drop-conv-nth list.sel(1) list.sel(3) tl-drop)
    ultimately show ?thesis
      apply simp
      apply(rule disjI1)
      apply(rule conjI)
       apply(rule CptsComp(2))
      using CptsComp(2) no-pestran-to-self' by blast
  qed
qed

abbreviation lift-seq Q P ≡ ESeq P Q
primrec lift-seq-esconf where lift-seq-esconf Q (P,s) = (lift-seq Q P, s)
abbreviation ‹lift-seq-cpt Q ≡ map (lift-seq-esconf Q)›
primrec lift-seq-esconf' where lift-seq-esconf' Q (P,s) = (if P = fin then (Q,s)
else (lift-seq Q P, s))
abbreviation ‹lift-seq-cpt' Q ≡ map (lift-seq-esconf' Q)›

lemma all-fin-after-fin:
  ‹(fin, s) # cs ∈ cpts (estran Γ) ⟹ ∀ c∈set cs. fst c = fin›
proof–
  obtain cpt where cpt: cpt = (fin, s)#cs by simp
  assume ‹(fin, s) # cs ∈ cpts (estran Γ)›
  with cpt have ‹cpt ∈ cpts (estran Γ)› by simp
  then show ?thesis using cpt
    apply (induct arbitrary: s cs)
      apply simp
  proof–
    fix P s t sa
    fix cs csa :: ‹('a,'k,'s,'prog) escpt›
    assume h: ‹⋀s csa. (P, t) # cs = (fin, s) # csa ⟹ ∀ c∈set csa. fst c = fin›
    assume eq: ‹(P, s) # (P, t) # cs = (fin, sa) # csa›
    then have P-fin: ‹P = fin› by simp
    with h have ‹∀ c∈set cs. fst c = fin› by blast
    moreover from eq P-fin have csa = (fin, t)#cs by fast
    ultimately show ‹∀ c∈set csa. fst c = fin› by simp
  next
```

```
    fix P Q :: ‹('a,'k,'s,'prog) esys›
    fix s t sa :: ‹'s ×('a,'k,'s,'prog) ectx›
    fix cs csa :: ‹('a,'k,'s,'prog) escpt›
    assume tran: ‹((P, s), Q, t) ∈ estran Γ›
    assume ‹(P, s) # (Q, t) # cs = (fin, sa) # csa›
    then have P-fin: ‹P = fin› by simp
    with tran have ‹((fin, s), (Q,t)) ∈ estran Γ› by simp
    then have False
      apply(simp add: estran-def)
      using no-estran-from-fin by fast
    then show ‹∀ c∈set csa. fst c = fin› by blast
  qed
qed

lemma lift-seq-cpt-partial:
  assumes ‹cpt∈cpts (estran Γ)›
    and ‹fst (last cpt) ≠ fin›
  shows ‹lift-seq-cpt Q cpt ∈ cpts (estran Γ)›
  using assms
proof(induct)
  case (CptsOne P s)
  show ?case by auto
next
  case (CptsEnv P t cs s)
  then show ?case by auto
next
  case (CptsComp P S Q1 T cs)
  from CptsComp(4) have 1: ‹fst (last ((Q1, T) # cs)) ≠ fin› by simp
  from CptsComp(3)[OF 1] have IH': ‹map (lift-seq-esconf Q) ((Q1, T) # cs) ∈
cpts (estran Γ)› .
  have ‹Q1≠fin›
  proof
    assume ‹Q1=fin›
    with all-fin-after-fin CptsComp(2) have ‹fst (last ((Q1, T) # cs)) = fin› by
fastforce
    with 1 show False by blast
  qed
  obtain s x where S: ‹S=(s,x)› by fastforce
  obtain t y where T: ‹T=(t,y)› by fastforce
  show ?case
    apply simp
    apply(rule cpts.CptsComp)
    apply(insert CptsComp(1))
     apply(simp add: estran-def) apply(erule exE) apply(rule exI)
     apply(simp add: S T)
    apply(erule ESeq)
     apply(rule ‹Q1≠fin›)
    using IH'[simplified] .
qed
```

```
lemma lift-seq-cpt:
  assumes ‹cpt∈cpts (estran Γ)›
    and ‹Γ ⊢ last cpt −es[a]→ (fin,t,y)›
  shows ‹lift-seq-cpt Q cpt @ [(Q,t,y)] ∈ cpts (estran Γ)›
  using assms
proof(induct)
  case (CptsOne P S)
  obtain s x where S: ‹S=(s,x)› by fastforce
  show ?case apply simp
    apply(rule CptsComp)
     apply (simp add: estran-def)
     apply(rule exI)
    apply(subst S)
    apply(rule ESeq-fin)
    using CptsOne S apply simp
    by (rule cpts.CptsOne)
next
  case (CptsEnv P T1 cs S)
  have ‹map (lift-seq-esconf Q) ((P, T1) # cs) @ [(Q, t,y)] ∈ cpts (estran Γ)›
    apply(rule CptsEnv(2))
    using CptsEnv(3) by fastforce
  then show ?case apply simp by (erule cpts.CptsEnv)
next
  case (CptsComp P S Q1 T1 cs)
  from CptsComp(1) have ctran: ‹∃ a. Γ ⊢ (P,S)−es[a]→(Q1,T1)›
    by (simp add: estran-def)
  have ‹Q1≠fin›
  proof
    assume ‹Q1=fin›
    with all-fin-after-fin CptsComp(2) have ‹∀ c∈set cs. fst c = fin› by fastforce
    with ‹Q1=fin› have ‹fst (last ((P, S) # (Q1, T1) # cs)) = fin› by simp
     with CptsComp(4) have ‹Γ ⊢ (fin, snd (last ((P, S) # (Q1, T1) # cs)))
−es[a]→ (fin, t,y)› using surjective-pairing by metis
    with no-estran-from-fin show False by blast
  qed
  obtain s x where S:‹S=(s,x)› by fastforce
  obtain t1 y1 where T1:‹T1=(t1,y1)› by fastforce
  have ‹map (lift-seq-esconf Q) ((Q1, T1) # cs) @ [(Q, t,y)] ∈ cpts (estran Γ)›
using CptsComp(3,4) by fastforce
  then show ?case apply simp apply(rule cpts.CptsComp)
    apply(simp add: estran-def) apply(insert ctran) apply(erule exE) apply(rule
exI)
    apply(simp add: S T1)
    apply(erule ESeq)
     apply(rule ‹Q1≠fin›)
    by assumption
qed
```

lemma *all-etran-from-fin*:
  assumes *cpt*: $cpt \in cpts\ (estran\ \Gamma)$
    and *cpt-eq*: $cpt = (fin,\ t)\ \#\ cs$
  shows ‹$\forall\, i.\ Suc\ i < length\ cpt \longrightarrow cpt!i\ -e\rightarrow\ cpt!Suc\ i$›
  using *cpt cpt-eq*
proof(*induct arbitrary*:*t cs*)
  case (*CptsOne P s*)
  then show *?case* **by** *simp*
next
  case (*CptsEnv P t1 cs1 s*)
  then have *et*: ‹$\forall\, i.\ Suc\ i < length\ ((P,\ t1)\ \#\ cs1) \longrightarrow ((P,\ t1)\ \#\ cs1)\ !\ i\ -e\rightarrow$ $((P,\ t1)\ \#\ cs1)\ !\ Suc\ i$› **by** *fast*
  show *?case*
  proof
    fix *i*
    show ‹$Suc\ i < length\ ((P,\ s)\ \#\ (P,\ t1)\ \#\ cs1) \longrightarrow ((P,\ s)\ \#\ (P,\ t1)\ \#\ cs1)$ $!\ i\ -e\rightarrow\ ((P,\ s)\ \#\ (P,\ t1)\ \#\ cs1)\ !\ Suc\ i$›
    proof(*cases i*)
      case *0*
      then show *?thesis* **by** *simp*
    next
      case (*Suc i′*)
      then show *?thesis* **using** *et* **by** *auto*
    qed
  qed
next
  case (*CptsComp P s Q t1 cs1*)
  then have ‹$((EAnon\ fin\text{-}com,\ t),\ Q,\ t1) \in estran\ \Gamma$› **by** *fast*
  then obtain *a* **where**
    ‹$\Gamma \vdash (EAnon\ fin\text{-}com,\ t)\ -es[a]\rightarrow\ (Q,\ t1)$› **using** *estran-def* **by** *blast*
  then have *False* **using** *no-estran-from-fin* **by** *blast*
  then show *?case* **by** *blast*
qed

lemma *no-ctran-from-fin*:
  assumes *cpt*: $cpt \in cpts\ (estran\ \Gamma)$
    and *cpt-eq*: $cpt = (fin,\ t)\ \#\ cs$
  shows ‹$\forall\, i.\ Suc\ i < length\ cpt \longrightarrow (cpt!i,\ cpt!Suc\ i) \notin estran\ \Gamma$›
proof
  fix *i*
  have *1*: ‹$\forall\, i.\ Suc\, i < length\ cpt \longrightarrow cpt!i\ -e\rightarrow\ cpt!Suc\ i$› **by** (*rule all-etran-from-fin*[*OF cpt cpt-eq*])
  show ‹$Suc\ i < length\ cpt \longrightarrow (cpt\ !\ i,\ cpt\ !\ Suc\ i) \notin estran\ \Gamma$›
  proof
    assume ‹$Suc\ i < length\ cpt$›
    with *1* **have** ‹$cpt!i\ -e\rightarrow\ cpt!Suc\ i$› **by** *blast*
    then show ‹$(cpt\ !\ i,\ cpt\ !\ Suc\ i) \notin estran\ \Gamma$›
      apply *simp*
      using *no-estran-to-self″* **by** *blast*

qed
qed

inductive-set *cpts-es-mod* **for** Γ **where**
CptsModOne[*intro*]: [(*P*,*s*,*x*)] ∈ *cpts-es-mod* Γ |
CptsModEnv[*intro*]: (*P*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟹ (*P*,*s*,*x*)#(*P*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ |
CptsModAnon: ⟦ Γ ⊢ (*P*, *s*) −*c*→ (*Q*, *t*); *Q* ≠ *fin-com*; (*EAnon Q*, *t*,*x*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EAnon P*, *s*,*x*)#(*EAnon Q*, *t*,*x*)#*cs* ∈ *cpts-es-mod* Γ |
CptsModAnon-fin: ⟦ Γ ⊢ (*P*, *s*) −*c*→ (*Q*, *t*); *Q* = *fin-com*; *y* = *x*(*k*:=*None*); (*EAnon Q*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EAnon P*, *s*,*x*)#(*EAnon Q*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ |
CptsModBasic: ‹⟦ *P* = *body e*; *s* ∈ *guard e*; *y*=*x*(*k*:=*Some e*); (*EAnon P*, *s*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EBasic e*, *s*,*x*)#(*EAnon P*, *s*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModAtom: ‹⟦ *P* = *body e*; *s* ∈ *guard e*; Γ ⊢ (*P*,*s*)−*c*∗→(*fin-com*,*t*); (*EAnon fin-com*, *t*,*x*)#*cs* ∈ *cpts-es-mod* Γ ⟧
⟹ (*EAtom e*, *s*,*x*)#(*EAnon fin-com*, *t*,*x*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModSeq: ‹Γ ⊢ (*P*,*s*,*x*)−*es*[*a*]→(*Q*,*t*,*y*) ⟹ *Q*≠*fin* ⟹ (*ESeq Q R*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟹ (*ESeq P R*, *s*,*x*)#(*ESeq Q R*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModSeq-fin: ‹Γ ⊢ (*P*,*s*,*x*)−*es*[*a*]→(*fin*,*t*,*y*) ⟹ (*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟹ (*P NEXT Q*, *s*,*x*)#(*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModChc1: ‹⟦ Γ ⊢ (*P*,*s*,*x*) −*es*[*a*]→ (*Q*,*t*,*y*); (*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EChc P R*, *s*,*x*)#(*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModChc2: ‹⟦ Γ ⊢ (*P*,*s*,*x*) −*es*[*a*]→ (*Q*,*t*,*y*); (*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EChc R P*, *s*,*x*)#(*Q*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |

CptsModJoin1: ‹⟦ Γ ⊢ (*P*,*s*,*x*) −*es*[*a*]→ (*Q*,*t*,*y*); (*EJoin Q R*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EJoin P R*, *s*,*x*)#(*EJoin Q R*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModJoin2: ‹⟦ Γ ⊢ (*P*,*s*,*x*) −*es*[*a*]→ (*Q*,*t*,*y*); (*EJoin R Q*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EJoin R P*, *s*,*x*)#(*EJoin R Q*, *t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModJoin-fin: ‹(*fin*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ ⟹ (*fin*⋈*fin*,*t*,*y*)#(*fin*,*t*,*y*)#*cs* ∈ *cpts-es-mod* Γ› |
CptsModWhileTMore: ‹⟦ *s*∈*b*; (*P*,*s*,*x*)#*cs* ∈ *cpts* (*estran* Γ); Γ ⊢ (*last* ((*P*,*s*,*x*)#*cs*)) −*es*[*a*]→ (*fin*,*t*,*y*); (*EWhile b P*, *t*,*y*)#*cs'* ∈ *cpts-es-mod* Γ ⟧
⟹ (*EWhile b P*, *s*,*x*) # *lift-seq-cpt* (*EWhile b P*) ((*P*,*s*,*x*)#*cs*) @ (*EWhile b P*, *t*,*y*) # *cs'* ∈ *cpts-es-mod* Γ› |
CptsModWhileTOnePartial: ‹⟦ *s*∈*b*; (*P*,*s*,*x*)#*cs* ∈ *cpts* (*estran* Γ); *fst* (*last* ((*P*,*s*,*x*)#*cs*)) ≠ *fin* ⟧ ⟹ (*EWhile b P*, *s*,*x*) # *lift-seq-cpt* (*EWhile b P*) ((*P*,*s*,*x*)#*cs*) ∈ *cpts-es-mod* Γ› |
CptsModWhileTOneFull: ‹⟦ *s*∈*b*; (*P*,*s*,*x*)#*cs* ∈ *cpts* (*estran* Γ); Γ ⊢ (*last* ((*P*,*s*,*x*)#*cs*))−*es*[*a*]→(*fin*,*t*,*y*); (*fin*,*t*,*y*)#*cs'* ∈ *cpts-es-mod* Γ ⟧ ⟹
(*EWhile b P*, *s*,*x*) # *lift-seq-cpt* (*EWhile b P*) ((*P*,*s*,*x*)#*cs*) @ *map* (λ(-,*s*,*x*). (*EWhile b P*, *s*,*x*)) ((*fin*,*t*,*y*)#*cs'*) ∈ *cpts-es-mod* Γ› |
CptsModWhileF: ‹⟦ *s*∉*b*; (*fin*, *s*,*x*)#*cs* ∈ *cpts-es-mod* Γ ⟧ ⟹ (*EWhile b P*, *s*,*x*)#(*fin*, *s*,*x*)#*cs* ∈ *cpts-es-mod* Γ›

definition ‹*all-seq Q cs* ≡ ∀ *c*∈*set cs*. ∃ *P*. *fst c* = *P NEXT Q*›

lemma *equiv-aux1*:

‹cs ∈ cpts (estran Γ) ⟹
hd cs = (P NEXT Q, s) ⟹
P ≠ fin ⟹
all-seq Q cs ⟹
∃ cs0. cs = lift-seq-cpt Q ((P, s) # cs0) ∧ (P,s)#cs0 ∈ cpts (estran Γ) ∧ fst (last ((P,s)#cs0)) ≠ fin›
proof−
assume cpt: ‹cs ∈ cpts (estran Γ)›
assume cs: ‹hd cs = (P NEXT Q, s)›
assume ‹P≠fin›
assume all-seq: ‹all-seq Q cs›
show ?thesis
using cpt cs ‹P≠fin› all-seq
proof(induct arbitrary: P s)
case (CptsOne P1 s1)
then show ?case **apply**−
apply(rule exI[**where** x=‹[]›])
apply simp
by (rule cpts.CptsOne)
next
case (CptsEnv P1 t cs s1)
from CptsEnv(3) **have** 1: ‹hd ((P1, t) # cs) = (P NEXT Q, t)› **by** simp
from ‹all-seq Q ((P1, s1) # (P1, t) # cs)› **have** 2: ‹all-seq Q ((P1, t) # cs)› **by** (simp add: all-seq-def)
from CptsEnv(3) **have** ‹s1=s› **by** simp
from CptsEnv(2)[OF 1 CptsEnv(4) 2] **obtain** cs0 **where**
‹(P1, t) # cs = map (lift-seq-esconf Q) ((P, t) # cs0) ∧ (P, t) # cs0 ∈ cpts (estran Γ) ∧ fst (last ((P, t) # cs0)) ≠ fin› **by** meson
then show ?case **apply**− **apply**(rule exI[**where** x=‹(P,t)#cs0›])
apply (simp add: ‹s1=s›)
apply(rule cpts.CptsEnv)
by blast
next
case (CptsComp P1 s1 Q1 t cs)
from CptsComp(6) **obtain** P′ **where** Q1: ‹Q1 = P′ NEXT Q› **by** (auto simp add: all-seq-def)
then have 1: ‹hd ((Q1, t) # cs) = (P′ NEXT Q, t)› **by** simp
from CptsComp(4) **have** P1: ‹P1=P NEXT Q› **and** ‹s1=s› **by** simp+
from CptsComp(1) P1 Q1 **have** ‹P′≠fin›
apply (simp add: estran-def)
apply(erule exE)
apply(erule estran-p.cases, auto)[]
using Q1 seq-neq2 **by** blast
from CptsComp(1) P1 Q1 **have** tran: ‹((P, s), P′, t) ∈ estran Γ›
apply(simp add: estran-def) **apply**(erule exE) **apply**(erule estran-p.cases, auto)[]
apply(rule exI) **apply** (simp add: ‹s1=s›)
using seq-neq2 **by** blast
from CptsComp(6) **have** 2: ‹all-seq Q ((Q1, t) # cs)› **by** (simp add: all-seq-def)

```
    from CptsComp(3)[OF 1 ‹P′≠fin› 2] obtain cs0 where
      ‹(Q1, t) # cs = map (lift-seq-esconf Q) ((P′, t) # cs0) ∧ (P′, t) # cs0 ∈
cpts (estran Γ) ∧ fst (last ((P′, t) # cs0)) ≠ fin› by meson
    then show ?case apply− apply(rule exI[where x=‹(P′,t)#cs0›])
      apply(rule conjI)
       apply (simp add: ‹s1=s› P1)
      apply(rule conjI)
       apply(rule cpts.CptsComp)
        apply(rule tran)
       apply blast
      by simp
  qed
qed

lemma split-seq-mod:
  assumes cpt: ‹cpt ∈ cpts-es-mod Γ›
    and hd-cpt: ‹hd cpt = (es1 NEXT es2, S0)›
    and not-all-seq: ‹¬ all-seq es2 cpt›
  shows
    ∃ i S′. cpt!i = (es2, S′) ∧
          i ≠ 0 ∧
          i < length cpt ∧
        (∃ cpt′. take i cpt = lift-seq-cpt es2 ((es1,S0)#cpt′) ∧ ((es1,S0)#cpt′)∈cpts
(estran Γ) ∧ (last ((es1,S0)#cpt′), (fin, S′))∈estran Γ) ∧
          all-seq es2 (take i cpt) ∧
          drop i cpt ∈ cpts-es-mod Γ
  using cpt hd-cpt not-all-seq
proof(induct arbitrary: es1 S0)
case (CptsModOne P S)
  then show ?case by (simp add: all-seq-def)
next
  case (CptsModEnv P t y cs s x)

  from CptsModEnv(3) have P-dest: ‹P = es1 NEXT es2› by simp
  from P-dest have 1: ‹(hd ((P, t, y) # cs)) = (es1 NEXT es2, t, y)› by simp
   from CptsModEnv(4) have 2: ‹¬ all-seq es2 ((P, t, y) # cs)› by (simp add:
all-seq-def)
  from CptsModEnv(2)[OF 1 2] obtain i S′ where
    ‹((P, t, y) # cs) ! i = (es2, S′) ∧
     i ≠ 0 ∧
     i < length ((P, t, y) # cs) ∧
     (∃ cpt′. take i ((P, t, y) # cs) = map (lift-seq-esconf es2) ((es1, t, y) # cpt′)
∧ (es1, t, y) # cpt′ ∈ cpts (estran Γ) ∧ (last ((es1, t, y) # cpt′), fin, S′) ∈ estran
Γ) ∧
     all-seq es2 (take i ((P, t, y) # cs)) ∧ drop i ((P, t, y) # cs) ∈ cpts-es-mod Γ›
    by meson
  then have
    p1: ‹((P, t, y) # cs) ! i = (es2, S′)› and
    p2: ‹i≠0› and
```

p3: ‹*i* < *length* ((*P*, *t*, *y*) # *cs*)› **and**
p4: ‹∃ *cpt'*. *take i* ((*P*, *t*, *y*) # *cs*) = *map* (*lift-seq-esconf es2*) ((*es1*, *t*, *y*) # *cpt'*) ∧ ((*es1*, *t*, *y*) # *cpt'*) ∈ *cpts* (*estran* Γ) ∧ (*last* ((*es1*, *t*, *y*) # *cpt'*), *fin*, *S'*) ∈ *estran* Γ› **and**
p5: ‹*all-seq es2* (*take i* ((*P*, *t*, *y*) # *cs*))› **and**
p6: ‹*drop i* ((*P*, *t*, *y*) # *cs*) ∈ *cpts-es-mod* Γ› **by** *argo*+
from *p4* **obtain** *cpt'* **where**
p4-1: ‹*take i* ((*P*, *t*, *y*) # *cs*) = *map* (*lift-seq-esconf es2*) ((*es1*, *t*, *y*) # *cpt'*)› **and**
p4-2: ‹((*es1*, *t*, *y*) # *cpt'*) ∈ *cpts* (*estran* Γ)› **and**
p4-3: ‹(*last* ((*es1*, *t*, *y*) # *cpt'*), *fin*, *S'*) ∈ *estran* Γ› **by** *meson*
show *?case*
apply(*rule exI*[**where** *x*=*Suc i*])
apply(*rule exI*[**where** *x*=*S'*])
apply(*rule conjI*)
using *p1* **apply** *simp*
apply(*rule conjI*) **apply** *simp*
apply(*rule conjI*) **using** *p3* **apply** *simp*
apply(*rule conjI*)
apply(*rule exI*[**where** *x*=‹(*es1*,*t*,*y*)#*cpt'*›])
apply(*rule conjI*)
using *p4-1 P-dest* **apply** *simp*
using *CptsModEnv*(*3*) **apply** *simp*
apply(*rule conjI*)
apply(*rule CptsEnv*)
using *p4-2* **apply** *fastforce*
using *p4-3* **apply** *fastforce*
using *p5 P-dest* **apply**(*simp add*: *all-seq-def*)
using *p6* **apply** *simp*.
next
case (*CptsModAnon*)
then show *?case* **by** *simp*
next
case (*CptsModAnon-fin*)
then show *?case* **by** *simp*
next
case (*CptsModBasic*)
then show *?case* **by** *simp*
next
case (*CptsModAtom*)
then show *?case* **by** *simp*
next
case (*CptsModSeq P s x a Q t y R cs*)
from *CptsModSeq*(*5*) **have** ‹*R*=*es2*› **by** *simp*
then have *1*: ‹(*hd* ((*Q NEXT R*, *t*,*y*) # *cs*)) = (*Q NEXT es2*, *t*,*y*)› **by** *simp*
from *CptsModSeq*(*6*) ‹*R*=*es2*› **have** *2*: ‹¬ *all-seq es2* ((*Q NEXT R*, *t*,*y*) # *cs*)› **by** (*simp add*: *all-seq-def*)
from *CptsModSeq*(*4*)[*OF 1 2*] **obtain** *i S'* **where**
‹((*Q NEXT R*, *t*, *y*) # *cs*) ! *i* = (*es2*, *S'*) ∧

$i \neq 0 \wedge$
$i < length$ ((Q NEXT R, t, y) # cs) ∧
(∃ cpt'. take i ((Q NEXT R, t, y) # cs) = map (lift-seq-esconf es2) ((Q, t, y) # cpt') ∧ (Q, t, y) # cpt' ∈ cpts (estran Γ) ∧ (last ((Q, t, y) # cpt'), fin, S') ∈ estran Γ) ∧
all-seq es2 (take i ((Q NEXT R, t, y) # cs)) ∧ drop i ((Q NEXT R, t, y) # cs) ∈ cpts-es-mod Γ› **by** *meson*
then have
p1: ‹((Q NEXT R, t,y) # cs) ! i = (es2, S')› **and**
p2: ‹i≠0› **and**
p3: ‹i < length ((Q NEXT R, t,y) # cs)› **and**
p4: ‹∃ cpt'. take i ((Q NEXT R, t,y) # cs) = map (lift-seq-esconf es2) ((Q, t,y) # cpt') ∧ ((Q, t,y) # cpt') ∈ cpts (estran Γ) ∧ (last ((Q, t,y) # cpt'), fin, S') ∈ estran Γ› **and**
p5: ‹all-seq es2 (take i ((Q NEXT R, t,y) # cs))› **and**
p6: ‹drop i ((Q NEXT R, t,y) # cs) ∈ cpts-es-mod Γ› **by** *argo+*
from *p4* **obtain** *cpt'* **where**
p4-1: ‹take i ((Q NEXT R, t,y) # cs) = map (lift-seq-esconf es2) ((Q, t,y) # cpt')› **and**
p4-2: ‹((Q, t,y) # cpt') ∈ cpts (estran Γ)› **and**
p4-3: ‹(last ((Q, t,y) # cpt'), fin, S') ∈ estran Γ› **by** *meson*
show *?case*
apply(*rule exI*[**where** *x=Suc i*])
apply(*rule exI*[**where** *x=S'*])
apply(*rule conjI*)
using *p1* **apply** *simp*
apply(*rule conjI*) **apply** *simp*
apply(*rule conjI*) **using** *p3* **apply** *simp*
apply(*rule conjI*)
apply(*rule exI*[**where** *x=‹(Q,t,y)#cpt'›*])
apply(*rule conjI*)
using *p4-1 CptsModSeq*(*5*) **apply** *simp*
apply(*rule conjI*)
apply(*rule CptsComp*)
using *CptsModSeq*(*1*,*5*) **apply** (*auto simp add*: *estran-def*)[]
using *p4-2* **apply** *simp*
using *p4-3* **apply** *simp*
using *p5* ‹R=es2› **apply**(*simp add*: *all-seq-def*)
using *p6* **by** *fastforce*
next
case (*CptsModSeq-fin P s x a t y Q cs*)
from *CptsModSeq-fin*(*4*) **have** ‹P=es1› ‹Q=es2› ‹(s,x)=S0› **by** *simp+*
show *?case*
apply(*rule exI*[**where** *x=1*])
apply(*rule exI*[**where** *x=‹(t,y)›*])
apply(*simp add*: *all-seq-def* ‹P=es1› ‹Q=es2› ‹(s,x)=S0›)
apply(*rule conjI*)
apply(*rule CptsOne*)
apply(*rule conjI*)

```
   using CptsModSeq-fin(1) ‹P=es1› ‹(s,x)=S0› apply (auto simp add: estran-def)[]
    using CptsModSeq-fin(2) ‹Q=es2› by simp
next
  case (CptsModChc1)
  then show ?case by simp
next
  case (CptsModChc2)
  then show ?case by simp
next
  case (CptsModJoin1)
  then show ?case by simp
next
  case (CptsModJoin2)
  then show ?case by simp
next
  case (CptsModJoin-fin)
  then show ?case by simp
next
  case (CptsModWhileTMore)
  then show ?case by simp
next
  case (CptsModWhileTOnePartial)
  then show ?case by simp
next
  case (CptsModWhileTOneFull)
  then show ?case by simp
next
  case (CptsModWhileF)
  then show ?case by simp
qed

lemma equiv-aux2:
  ‹∀ i<length cs. fst (cs!i) = P ⟹ (P,s)#cs ∈ cpts tran›
proof(induct cs arbitrary:s)
  case Nil
  show ?case by (rule CptsOne)
next
  case (Cons c cs)
  from Cons(2)[THEN spec[where x=0]] have ‹fst c = P› by simp
  show ?case apply(subst surjective-pairing[of c]) apply(subst ‹fst c = P›)
    apply(rule CptsEnv)
    apply(rule Cons(1))
    using Cons(2) by fastforce
qed

theorem cpts-es-mod-equiv:
  ‹cpts (estran Γ) = cpts-es-mod Γ›
proof
  show ‹cpts (estran Γ) ⊆ cpts-es-mod Γ›
```

proof
  fix *cpt*
  assume ‹*cpt* ∈ *cpts* (*estran* Γ)›
  then show ‹*cpt* ∈ *cpts-es-mod* Γ›
  proof(*induct*)
    case (*CptsOne P S*)
    obtain *s x* **where** ‹*S*=(*s*,*x*)› **by** *fastforce*
    from *CptsOne this CptsModOne* **show** *?case* **by** *fast*
  next
    case (*CptsEnv P T cs S*)
    obtain *s x* **where** *S*:‹*S*=(*s*,*x*)› **by** *fastforce*
    obtain *t y* **where** *T*:‹*T*=(*t*,*y*)› **by** *fastforce*
    show *?case* **using** *CptsModEnv estran-def S T CptsEnv* **by** *fast*
  next
    case (*CptsComp P S Q T cs*)
    from *CptsComp*(*1*) **obtain** *a* **where** *h*:
      ‹Γ ⊢ (*P*,*S*)−*es*[*a*]→(*Q*,*T*)› **using** *estran-def* **by** *blast*
    then show *?case*
    proof(*cases*)
      case (*EAnon*)
      then show *?thesis* **apply** *clarify*
        apply(*erule CptsModAnon*) **apply** *blast*
        using *CptsComp EAnon* **by** *blast*
    next
      case (*EAnon-fin*)
      then show *?thesis* **apply** *clarify*
        apply(*erule CptsModAnon-fin*) **apply** *blast+*
        using *CptsComp EAnon* **by** *blast*
    next
      case (*EBasic*)
      then show *?thesis* **apply** *clarify*
        apply(*rule CptsModBasic*, *auto*)
        using *CptsComp EBasic* **by** *simp*
    next
      case (*EAtom*)
      then show *?thesis* **apply** *clarify*
        apply(*rule CptsModAtom*) **using** *CptsComp* **by** *auto*
    next
      case (*ESeq*)
      then show *?thesis* **apply** *clarify*
        apply(*rule CptsModSeq*) **using** *CptsComp* **by** *auto*
    next
      case (*ESeq-fin*)
      then show *?thesis* **apply** *clarify*
        apply(*rule CptsModSeq-fin*) **using** *CptsComp* **by** *auto*
    next
      case (*EChc1*)
      then show *?thesis* **apply** *clarify*
        apply(*rule CptsModChc1*) **using** *CptsComp* **by** *auto*

next
case (*EChc2*)
then show *?thesis* **apply** *clarify*
apply(*rule CptsModChc2*) **using** *CptsComp* **by** *auto*
next
case (*EJoin1*)
then show *?thesis* **apply** *clarify*
apply(*rule CptsModJoin1*) **using** *CptsComp* **by** *auto*
next
case (*EJoin2*)
then show *?thesis* **apply** *clarify*
apply(*rule CptsModJoin2*) **using** *CptsComp* **by** *auto*
next
case *EJoin-fin*
then show *?thesis* **apply** *clarify*
apply(*rule CptsModJoin-fin*) **using** *CptsComp* **by** *auto*
next
case *EWhileF*
then show *?thesis* **apply** *clarify*
apply(*rule CptsModWhileF*) **using** *CptsComp* **by** *auto*
next
case (*EWhileT s b P1 x k*)
thm *CptsComp*

show *?thesis*
proof(*cases* ‹*all-seq* (*EWhile b P1*) ((*P1 NEXT EWhile b P1*, *T*) # *cs*)›)
case *True*
from *EWhileT*(*4*) **have** *1*: ‹*hd* ((*Q*, *T*) # *cs*) = (*P1 NEXT EWhile b P1*, *T*)› **by** *simp*
from *True EWhileT*(*4*) **have** *2*: ‹*all-seq* (*EWhile b P1*) ((*Q*, *T*) # *cs*)› **by** *simp*
from *equiv-aux1*[*OF CptsComp*(*2*) *1* ‹*P1*≠*fin*› *2*] **obtain** *cs0* **where**
3: ‹(*Q*, *T*) # *cs* = *map* (*lift-seq-esconf* (*EWhile b P1*)) ((*P1*, *T*) # *cs0*) ∧ (*P1*, *T*) # *cs0* ∈ *cpts* (*estran* Γ) ∧ *fst* (*last* ((*P1*, *T*) # *cs0*)) ≠ *fin*› **by** *meson*

then have *p3-1*: ‹(*Q*, *T*) # *cs* = *map* (*lift-seq-esconf* (*EWhile b P1*)) ((*P1*, *T*) # *cs0*)› **and**
p3-2: ‹(*P1*, *s*, *x*) # *cs0* ∈ *cpts* (*estran* Γ)› **and**
p3-3: ‹*fst* (*last* ((*P1*, *s*, *x*) # *cs0*)) ≠ *fin*› **using** ‹*T*=(*s*,*x*)› **by** *blast+*
from *CptsModWhileTOnePartial*[*OF* ‹*s*∈*b*› *p3-2 p3-3*]
have ‹(*EWhile b P1*, *s*,*x*) # *map* (*lift-seq-esconf* (*EWhile b P1*)) ((*P1*, *s*,*x*) # *cs0*) ∈ *cpts-es-mod* Γ› **.**
with *EWhileT 3* **show** *?thesis* **by** *simp*
next
case *False*
with *EWhileT*(*4*) **have** *not-all-seq*: ‹¬*all-seq* (*EWhile b P1*) ((*Q*,*T*)#*cs*)› **by** *simp*
from *EWhileT*(*4*) **have** ‹(*hd* ((*Q*, *T*) # *cs*)) = (*P1 NEXT EWhile b P1*, *T*)› **by** *simp*

from *split-seq-mod*[*OF CptsComp*(*3*) *this not-all-seq*] **obtain** *i S′* **where** *split*:
‹((Q, T) # cs) ! i = (EWhile b P1, S′) ∧
i ≠ 0 ∧
i < length ((Q, T) # cs) ∧
(∃ cpt′. take i ((Q, T) # cs) = map (lift-seq-esconf (EWhile b P1)) ((P1, T) # cpt′) ∧ (P1, T) # cpt′ ∈ cpts (estran Γ) ∧ (last ((P1, T) # cpt′), fin, S′) ∈ estran Γ) ∧
all-seq (EWhile b P1) (take i ((Q, T) # cs)) ∧ drop i ((Q, T) # cs) ∈ cpts-es-mod Γ›
by *blast*
then have *3*: ‹all-seq (EWhile b P1) (take i ((Q, T) # cs))›
and ‹i≠0›
and *i-lt*: ‹i < length ((Q, T) # cs)›
and *part2-cpt*: ‹drop i ((Q, T) # cs) ∈ cpts-es-mod Γ›
and *ex-cpt′*: ‹∃ cpt′. take i ((Q, T) # cs) = map (lift-seq-esconf (EWhile b P1)) ((P1, T) # cpt′) ∧ (P1, T) # cpt′ ∈ cpts (estran Γ) ∧ (last ((P1, T) # cpt′), fin, S′) ∈ estran Γ› **by** *blast+*
from *ex-cpt′* **obtain** *cpt′* **where** *cpt′1*: ‹take i ((Q, T) # cs) = map (lift-seq-esconf (EWhile b P1)) ((P1, T) # cpt′)› **and**
cpt′2: ‹((P1, s,x) # cpt′) ∈ cpts (estran Γ)› **and**
cpt′3: ‹(last ((P1, s,x) # cpt′), fin, S′) ∈ estran Γ› **using** ‹T=(s,x)› **by** *meson*
from *cpts-take*[*OF CptsComp*(*2*)] ‹i≠0› **have** *1*: ‹take i ((Q, T) # cs) ∈ cpts (estran Γ)› **by** *fast*
have *2*: ‹hd (take i ((Q, T) # cs)) = (P1 NEXT EWhile b P1, T)› **using** ‹i≠0› *EWhileT*(*4*) **by** *simp*
obtain *s′ x′* **where** *S′*: ‹S′=(s′,x′)› **by** *fastforce*
obtain *cs′* **where** *part2-eq*: ‹drop i ((Q, T) # cs) = (EWhile b P1, S′) # cs′›
proof
from *split* **have** ‹((Q, T) # cs) ! i = (EWhile b P1, S′)› **by** *argo*
with *i-lt* **show** ‹drop i ((Q, T) # cs) = (EWhile b P1, S′) # drop (Suc i) ((Q,T)#cs)›
using *Cons-nth-drop-Suc* **by** *metis*
qed
with *part2-cpt S′* **have** ‹(EWhile b P1, s′,x′) # cs′ ∈ cpts-es-mod Γ› **by** *argo*
from *cpt′3* **have** ‹∃ a. Γ ⊢ last ((P1, s,x) # cpt′) −es[a]→ (fin, S′)› **by** (*simp add*: *estran-def*)
then obtain *a* **where** ‹Γ ⊢ last ((P1, s,x) # cpt′) −es[a]→ (fin, s′, x′)› **using** *S′* **by** *meson*
from *CptsModWhileTMore*[*OF* ‹s∈b› *cpt′2*[*simplified*] *this* ‹(EWhile b P1, s′,x′) # cs′ ∈ cpts-es-mod Γ›] **have**
‹(EWhile b P1, s, x) # map (lift-seq-esconf (EWhile b P1)) ((P1, s, x) # cpt′) @ (EWhile b P1, s′, x′) # cs′ ∈ cpts-es-mod Γ› **.**
moreover have ‹(Q,T)#cs = map (lift-seq-esconf (EWhile b P1)) ((P1, T) # cpt′) @ (EWhile b P1, S′) # cs′›
using *cpt′1 part2-eq i-lt* **by** (*metis append-take-drop-id*)

```
        ultimately show ?thesis using EWhileT S' by argo
      qed
    qed
  qed
qed
next
  show ‹cpts-es-mod Γ ⊆ cpts (estran Γ)›
  proof
    fix cpt
    assume ‹cpt ∈ cpts-es-mod Γ›
    then show ‹cpt ∈ cpts (estran Γ)›
    proof(induct)
      case (CptsModOne)
      then show ?case by (rule CptsOne)
    next
      case (CptsModEnv)
      then show ?case using CptsEnv by fast
    next
      case (CptsModAnon P s Q t x cs)
      from CptsModAnon(1) have ‹((P,s),(Q,t))∈ptran Γ› by simp
      with CptsModAnon show ?case apply− apply(rule CptsComp, auto simp add: estran-def)
        apply(rule exI)
        apply(rule EAnon)
        apply simp+
        done
    next
      case (CptsModAnon-fin P s Q t y x k cs)
      from CptsModAnon-fin(1) have ‹((P,s),(Q,t))∈ptran Γ› by simp
      with CptsModAnon-fin show ?case apply− apply(rule CptsComp, auto simp add: estran-def)
        apply(rule exI)
        apply(rule EAnon-fin)
        by simp+
    next
      case (CptsModBasic)
      then show ?case apply− apply(rule CptsComp, auto simp add: estran-def, rule exI)
        apply(rule EBasic, auto) done
    next
      case (CptsModAtom)
      then show ?case apply− apply(rule CptsComp, auto simp add: estran-def, rule exI)
        apply(rule EAtom, auto) done
    next
      case (CptsModSeq)
      then show ?case apply− apply(rule CptsComp, auto simp add: estran-def, rule exI)
        apply(rule ESeq, auto) done
```

next
case *CptsModSeq-fin*
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule ESeq-fin*).
next
case (*CptsModChc1*)
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EChc1*, *auto*) **done**
next
case (*CptsModChc2*)
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EChc2*, *auto*) **done**
next
case (*CptsModJoin1*)
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EJoin1*, *auto*) **done**
next
case (*CptsModJoin2*)
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EJoin2*, *auto*) **done**
next
case *CptsModJoin-fin*
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EJoin-fin*).
next
case *CptsModWhileF*
then show *?case* **apply**− **apply**(*rule CptsComp*, *auto simp add*: *estran-def*, *rule exI*)
apply(*rule EWhileF*, *auto*) **done**
next
case (*CptsModWhileTMore s b P x cs a t y cs'*)
from *CptsModWhileTMore*(*2*,*3*) *all-fin-after-fin no-estran-from-fin* **have** ‹$P \neq fin$›
by (*metis last-in-set list.distinct*(*1*) *prod.collapse set-ConsD*)
have *1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*) @ (*EWhile b P*, *t*,*y*) # *cs'* ∈ *cpts* (*estran* Γ)›
proof−
from *lift-seq-cpt*[*OF* ‹(*P*, *s*,*x*) # *cs* ∈ *cpts* (*estran* Γ)› *CptsModWhileT-More*(*3*)]
have ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*) @ [(*EWhile b P*, *t*,*y*)] ∈ *cpts* (*estran* Γ)› **.**
then have *cpt-part1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*) ∈ *cpts* (*estran* Γ)›

apply *simp* **using** *cpts-remove-last* **by** *fast*
from *CptsModWhileTMore*(*3*)
have *tran*: ‹(*last* (*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*)), *hd* ((*EWhile b P*, *t*,*y*) # *cs'*)) ∈ *estran* Γ›
apply (*auto simp add*: *estran-def*)
apply(*rule exI*)
apply(*erule ESeq-fin*)
apply(*rule exI*)
apply(*subst last-map*)
apply *assumption*
apply(*simp add*: *lift-seq-esconf-def case-prod-unfold*)
apply(*subst surjective-pairing*[*of* ‹*snd* (*last cs*)›])
apply(*rule ESeq-fin*)
by *simp*
show *?thesis*
apply(*rule cpts-append-comp*)
apply(*rule cpt-part1*)
apply(*rule CptsModWhileTMore*(*5*))
apply(*rule tran*)
done
qed
show *?case*
apply *simp*
apply(*rule CptsComp*)
apply (*simp add*: *estran-def*)
apply(*rule exI*)
apply(*rule EWhileT*)
apply(*rule* ‹*s*∈*b*›)
apply(*rule* ‹*P*≠*fin*›)
using *1* **by** *fastforce*
next
case (*CptsModWhileTOnePartial s b P x cs*)
from *CptsModWhileTOnePartial*(*3*) *all-fin-after-fin* **have** ‹*P*≠*fin*›
by (*metis CptsModWhileTOnePartial.hyps*(*2*) *fst-conv last-in-set list.distinct*(*1*) *set-ConsD*)
from *lift-seq-cpt-partial*[*OF* ‹(*P*, *s*,*x*) # *cs* ∈ *cpts* (*estran* Γ)› ‹*fst* (*last* ((*P*, *s*,*x*) # *cs*)) ≠ *fin*›]
have *1*: ‹*lift-seq-cpt* (*EWhile b P*) ((*P*, *s*,*x*) # *cs*) ∈ *cpts* (*estran* Γ)› **.**
show *?case*
apply *simp*
apply(*rule CptsComp*)
apply (*simp add*: *estran-def*)
apply(*rule exI*)
apply(*rule EWhileT*)
apply(*rule* ‹*s*∈*b*›)
apply(*rule* ‹*P*≠*fin*›)
using *1* **by** *simp*
next
case (*CptsModWhileTOneFull s b P x cs a t y cs'*)

from *lift-seq-cpt*[*OF* ‹(*P*, *s*,*x*) # *cs* ∈ *cpts* (*estran* Γ)› ‹Γ ⊢ *last* ((*P*, *s*,*x*) # *cs*) −*es*[*a*]→ (*fin*, *t*,*y*)›]
have *1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*) @ [(*EWhile b P*, *t*,*y*)] ∈ *cpts* (*estran* Γ)› **.**
let *?map* = ‹*map* (λ(-, *s*,*x*). (*EWhile b P*, *s*,*x*)) *cs*′›
have *p*: ‹∀ *i*<*length ?map*. *fst* (*?map*!*i*) = *EWhile b P*› **by** (*simp add*: *case-prod-unfold*)
have *2*: ‹(*EWhile b P*, *t*,*y*) # *map* (λ(-, *s*,*x*). (*EWhile b P*, *s*,*x*)) *cs*′ ∈ *cpts* (*estran* Γ)›
using *equiv-aux2*[*OF p*] **.**
from *cpts-append*[*OF 1 2*] **have** *3*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*,*x*) # *cs*) @ (*EWhile b P*, *t*,*y*) # *map* (λ(-, *s*,*x*). (*EWhile b P*, *s*,*x*)) *cs*′ ∈ *cpts* (*estran* Γ)› **.**
from *CptsModWhileTOneFull*(*2*,*3*) *all-fin-after-fin no-estran-from-fin* **have** ‹*P*≠*fin*›
by (*metis last-in-set list.distinct*(*1*) *prod.collapse set-ConsD*)
show *?case*
apply *simp*
apply(*rule CptsComp*)
apply(*simp add*: *estran-def*) **apply** (*rule exI*) **apply**(*rule EWhileT*) **apply**(*rule* ‹*s*∈*b*›)
apply(*rule* ‹*P*≠*fin*›)
using *3*[*simplified*] **.**
qed
qed
qed

lemma *ctran-imp-not-etran*:
‹(*c1*,*c2*)∈*estran* Γ ⟹ ¬ *c1* −*e*→ *c2*›
apply (*simp add*: *estran-def*)
apply(*erule exE*)
using *no-estran-to-self* **by** (*metis prod.collapse*)

fun *split* :: ‹(′*l*,′*k*,′*s*,′*prog*) *escpt* ⇒ (′*l*,′*k*,′*s*,′*prog*) *escpt* × (′*l*,′*k*,′*s*,′*prog*) *escpt*›
where
‹*split* ((*P* ⋈ *Q*, *s*) # *rest*) = ((*P*,*s*) # *fst* (*split rest*), (*Q*,*s*) # *snd* (*split rest*))› |
‹*split* - = ([],[])›

inductive-cases *estran-all-cases*: ‹(*P* ⋈ *Q*, *s*) # (*R*, *t*) # *cs* ∈ *cpts-es-mod* Γ›

lemma *split-same-length*:
‹*length* (*fst* (*split cpt*)) = *length* (*snd* (*split cpt*))›
by (*induct cpt rule*: *split.induct*) *auto*

lemma *split-same-state1*:
‹*i* < *length* (*fst* (*split cpt*)) ⟹ *snd* (*fst* (*split cpt*) ! *i*) = *snd* (*cpt* ! *i*)›
apply (*induct cpt arbitrary*: *i rule*: *split.induct*, *auto*)

apply(*case-tac i*; *simp*)
done

lemma *split-same-state2*:
$\langle i < length\ (snd\ (split\ cpt)) \Longrightarrow snd\ (snd\ (split\ cpt)\ !\ i) = snd\ (cpt\ !\ i)\rangle$
apply (*induct cpt arbitrary*: *i rule*: *split.induct*, *auto*)
apply(*case-tac i*; *simp*)
done

lemma *split-length-le1*:
$\langle length\ (fst\ (split\ cpt)) \leq length\ cpt\rangle$
by (*induct cpt rule*: *split.induct*, *auto*)

lemma *split-length-le2*:
$\langle length\ (snd\ (split\ cpt)) \leq length\ cpt\rangle$
by (*induct cpt rule*: *split.induct*, *auto*)

lemma *all-neq1*[*simp*]: $\langle P \bowtie Q \neq P\rangle$
proof
assume $\langle P \bowtie Q = P\rangle$
then have $\langle es\text{-}size\ (P \bowtie Q) = es\text{-}size\ P\rangle$ **by** *simp*
then show *False* **by** *simp*
qed

lemma *all-neq2*[*simp*]: $\langle P \bowtie Q \neq Q\rangle$
proof
assume $\langle P \bowtie Q = Q\rangle$
then have $\langle es\text{-}size\ (P \bowtie Q) = es\text{-}size\ Q\rangle$ **by** *simp*
then show *False* **by** *simp*
qed

lemma *split-cpt-aux1*:
$\langle((P \bowtie Q,\ s0),\ fin,\ t) \in estran\ \Gamma \Longrightarrow P = fin \wedge Q = fin\rangle$
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*erule estran-p.cases*, *auto*)
done

lemma *split-cpt-aux3*:
$\langle((P \bowtie Q,\ s),\ (R,\ t)) \in estran\ \Gamma \Longrightarrow$
$R \neq fin \Longrightarrow$
$\exists P'\ Q'.\ R = P' \bowtie Q' \wedge (P = P' \wedge ((Q,s),(Q',t)) \in estran\ \Gamma \vee Q = Q' \wedge$
$((P,s),(P',t)) \in estran\ \Gamma)\rangle$
proof−
assume $\langle((P \bowtie Q,\ s),\ (R,\ t)) \in estran\ \Gamma\rangle$
with *estran-def* **obtain** *a* **where** *h*: $\langle\Gamma \vdash (P \bowtie Q,\ s) -es[a]\rightarrow (R,\ t)\rangle$ **by** *blast*
assume $\langle R \neq fin\rangle$
with *h* **show** *?thesis* **apply**− **by** (*erule estran-p.cases*, *auto simp add*: *estran-def*)
qed

```
lemma split-cpt:
  assumes cpt-from:
    ‹cpt ∈ cpts-from (estran Γ) (P ⋈ Q, s0)›
  shows
    ‹fst (split cpt) ∈ cpts-from (estran Γ) (P, s0) ∧
     snd (split cpt) ∈ cpts-from (estran Γ) (Q, s0)›
proof−
  from cpt-from have cpt: ‹cpt ∈ cpts (estran Γ)› and hd-cpt: ‹hd cpt = (P ⋈ Q,
s0)› by auto
  show ?thesis using cpt hd-cpt
  proof(induct arbitrary: P Q s0)
    case (CptsOne)
    then show ?case
      apply(simp add: split-def)
      apply(rule conjI; rule cpts.CptsOne)
      done
  next
    case (CptsEnv)
    then show ?case
      apply(simp add: split-def)
      apply(rule conjI; rule cpts.CptsEnv, simp)
      done
  next
    case (CptsComp P1 S Q1 T cs)
    show ?case
    proof(cases ‹Q1 = fin›)
      case True
      with CptsComp show ?thesis
        apply(simp add: split-def)
        apply(drule split-cpt-aux1)
        apply clarify
        apply(rule conjI; rule CptsOne)
        done
    next
      case False
      with CptsComp show ?thesis
        apply(simp add: split-def)
        apply(rule conjI)
         apply(drule split-cpt-aux3, assumption)
         apply clarify
         apply simp
         apply(erule disjE)
        apply simp
          apply(rule CptsEnv) using surjective-pairing apply metis
        apply clarify
         apply (rule cpts.CptsComp, assumption)
         apply simp
        using surjective-pairing apply metis
```

apply(*drule split-cpt-aux3*) **apply** *assumption*
apply *clarsimp*
apply(*erule disjE*)
apply *clarify*
apply(*rule cpts.CptsComp*, *assumption*)
using *surjective-pairing* **apply** *metis*
apply *clarify*
apply(*rule CptsEnv*)
using *surjective-pairing* **apply** *metis*
done
qed
qed
qed

lemma *estran-from-all-both-fin*:
‹$\Gamma \vdash$ (*fin* ⋈ *fin*, *s*) $-es[a]\rightarrow$ (*Q1*, *t*) $\Longrightarrow$ *Q1* = *fin*›
apply(*erule estran-p.cases*, *auto*)
using *no-estran-from-fin* **apply** *blast+*
done

lemma *estran-from-all*:
‹$\Gamma \vdash$ (*P* ⋈ *Q*, *s*) $-es[a]\rightarrow$ (*Q1*, *t*) $\Longrightarrow \neg$ (*P* = *fin* $\wedge$ *Q* = *fin*) $\Longrightarrow \exists P'\ Q'.$ *Q1* = P' ⋈ Q'›
by (*erule estran-p.cases*, *auto*)

lemma *all-fin-after-fin'*:
‹(*fin*, *s*) # *cs* $\in$ *cpts* (*estran* Γ) $\Longrightarrow$ *i* < *Suc* (*length cs*) $\Longrightarrow$ *fst* (((*fin*,*s*)#*cs*)!*i*) = *fin*›
apply(*cases i*) **apply** *simp*
using *all-fin-after-fin* **by** *fastforce*

lemma *all-fin-after-fin''*:
assumes *cpt*: ‹*cpt* $\in$ *cpts* (*estran* Γ)›
and *i-lt*: ‹*i* < *length cpt*›
and *fin*: ‹*fst* (*cpt*!*i*) = *fin*›
shows ‹$\forall j.\ j > i \longrightarrow j <$ *length cpt* $\longrightarrow$ *fst* (*cpt*!*j*) = *fin*›
proof(*auto*)

have ‹*drop i cpt* = *cpt*!*i* # *drop* (*Suc i*) *cpt*›
by (*simp add*: *Cons-nth-drop-Suc i-lt*)
then have ‹*drop i cpt* = (*fst* (*cpt*!*i*), *snd* (*cpt*!*i*)) # *drop* (*Suc i*) *cpt*›
using *surjective-pairing* **by** *simp*
with *fin* **have** *1*: ‹*drop i cpt* = (*fin*, *snd* (*cpt*!*i*)) # *drop* (*Suc i*) *cpt*› **by** *simp*

from *cpts-drop*[*OF cpt i-lt*] **have** ‹*drop i cpt* $\in$ *cpts* (*estran* Γ)› **.**
with *1* **have** *2*: ‹(*fin*, *snd* (*cpt*!*i*)) # *drop* (*Suc i*) *cpt* $\in$ *cpts* (*estran* Γ)› **by** *simp*

fix *j*
assume ‹*i* < *j*›

assume ‹j < *length cpt*›

have ‹$j-i$ < *Suc* (*length* (*drop* (*Suc i*) *cpt*))›
 by (*simp add*: *Suc-diff-Suc* ‹$i < j$› ‹j < *length cpt*› *diff-less-mono i-lt less-imp-le*)

from *all-fin-after-fin*′[*OF 2 this*] *1* **have** ‹*fst* (*drop i cpt* ! ($j-i$)) = *fin*› **by** *simp*

then show ‹*fst* (*cpt*!*j*) = *fin*›
 apply(*subst* (*asm*) *nth-drop*) **using** *i-lt* **apply** *linarith*
 using ‹$i<j$› **by** *simp*
qed

lemma *estran-from-fin-AND-fin*:
‹((*fin* ⋈ *fin*, *s*), *Q1*, *t*) ∈ *estran* Γ ⟹ *Q1* = *fin*›
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*erule estran-p.cases*, *auto*)
using *no-estran-from-fin* **by** *blast*+

lemma *split-etran-aux*:
‹*P1* = *P* ⋈ *Q* ⟹ ((*P1*,*s*),(*Q1*,*t*)) ∈ *estran* Γ ⟹ (*Q1*,*t*)#*cs* ∈ *cpts* (*estran* Γ) ⟹ *Suc i* < *length* ((*P1*, *s*) # (*Q1*, *t*) # *cs*) ⟹ *fst* (((*P1*, *s*) # (*Q1*, *t*) # *cs*) ! *Suc i*) ≠ *fin* ⟹ ∃ *P*′ *Q*′. *Q1* = *P*′ ⋈ *Q*′›
apply(*cases* ‹*P* = *fin* ∧ *Q* = *fin*›)
 apply *simp*
 apply(*drule estran-from-fin-AND-fin*)
 apply *simp*
using *all-fin-after-fin*′ **apply** *blast*
apply(*simp add*: *estran-def*)
apply(*erule exE*)
using *estran-from-all* **by** *blast*

lemma *split-etran*:
assumes *cpt*: *cpt* ∈ *cpts* (*estran* Γ)
assumes *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*›
assumes *Suc-i-lt*: *Suc i* < *length cpt*
assumes *etran*: *cpt*!*i* −*e*→ *cpt*!*Suc i*
assumes *not-fin*: ‹*fst* (*cpt*!*Suc i*) ≠ *fin*›
shows
 fst (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i* ∧
 snd (*split cpt*) ! *i* −*e*→ *snd* (*split cpt*) ! *Suc i*
using *cpt fst-hd-cpt Suc-i-lt etran not-fin*
proof(*induct arbitrary*:*P Q i*)
case (*CptsOne P s*)
then show *?case* **by** *simp*
next
case (*CptsEnv P1 t cs s*)
show *?case*
proof(*cases i*)

```
    case 0
    with CptsEnv show ?thesis by simp
  next
   case (Suc i')
    from CptsEnv(3) have 1:
      ‹fst (hd ((P1, t) # cs)) = P ⋈ Q› by simp
    then have P1-conv: ‹P1 = P ⋈ Q› by simp
    from Suc ‹Suc i < length ((P1, s) # (P1, t) # cs)› have 2: ‹Suc i' < length
((P1,t)#cs)› by simp
    from Suc ‹((P1, s) # (P1, t) # cs) ! i −e→ ((P1, s) # (P1, t) # cs) ! Suc
i› have 3:
      ‹((P1, t) # cs) ! i' −e→ ((P1, t) # cs) ! Suc i'› by simp
    from CptsEnv(6) Suc have 4: ‹fst (((P1, t) # cs) ! Suc i') ≠ fin› by simp
    have
      ‹fst (split ((P1, t) # cs)) ! i' −e→ fst (split ((P1, t) # cs)) ! Suc i' ∧
       snd (split ((P1, t) # cs)) ! i' −e→ snd (split ((P1, t) # cs)) ! Suc i'›
      by (rule CptsEnv(2)[OF 1 2 3 4])
    with Suc P1-conv show ?thesis by simp
  qed
next
  case (CptsComp P1 s Q1 t cs)
  show ?case
  proof(cases i)
    case 0
    with CptsComp show ?thesis using no-estran-to-self' by auto
  next
    case (Suc i')
    from CptsComp(4) have 1: ‹P1 = P ⋈ Q› by simp
     have ‹∃P' Q'. Q1 = P' ⋈ Q'› using split-etran-aux[OF 1 CptsComp(1)
CptsComp(2)] CptsComp(5,7) by force
    then obtain P' Q' where 2: ‹Q1 = P' ⋈ Q'› by blast
    from 2 have 3: ‹fst (hd ((Q1, t) # cs)) = P' ⋈ Q'› by simp
    from CptsComp(5) Suc have 4: ‹Suc i' < length ((Q1,t)#cs)› by simp
     from CptsComp(6) Suc have 5: ‹((Q1, t) # cs) ! i' −e→ ((Q1, t) # cs) !
Suc i'› by simp
    from CptsComp(7) Suc have 6: ‹fst (((Q1, t) # cs) ! Suc i') ≠ fin› by simp
    have
      ‹fst (split ((Q1, t) # cs)) ! i' −e→ fst (split ((Q1, t) # cs)) ! Suc i' ∧
       snd (split ((Q1, t) # cs)) ! i' −e→ snd (split ((Q1, t) # cs)) ! Suc i'›
      by (rule CptsComp(3)[OF 3 4 5 6])
    with Suc 1 show ?thesis by simp
  qed
qed

lemma all-join-aux:
  ‹(c1, c2) ∈ estran Γ ⟹
   fst c1 = P ⋈ Q ⟹
   fst c2 ≠ fin ⟹
   ∃P' Q'. fst c2 = P' ⋈ Q'›
```

```
  apply(simp add: estran-def, erule exE)
  apply(erule estran-p.cases, auto)
  done

lemma all-join:
  ‹cpt ∈ cpts (estran Γ) ⟹
   fst (hd cpt) = P ⋈ Q ⟹
   n < length cpt ⟹
   fst (cpt!n) ≠ fin ⟹
   ∀ i ≤ n. ∃ P′ Q′. fst (cpt!i) = P′ ⋈ Q′›
proof−
  assume cpt: ‹cpt ∈ cpts (estran Γ)›
  with cpts-nonnil have ‹cpt≠[]› by blast
  from cpt cpts-def′ have ct-or-et:
    ‹∀ i. Suc i < length cpt ⟶ (cpt!i, cpt!Suc i) ∈ estran Γ ∨ cpt!i −e⟶ cpt!Suc
i› by blast
  assume fst-hd-cpt: ‹fst (hd cpt) = P ⋈ Q›
  assume n-lt: ‹n < length cpt›
  assume not-fin: ‹fst (cpt!n) ≠ fin›
  show ‹∀ i ≤ n. ∃ P′ Q′. fst (cpt!i) = P′ ⋈ Q′›
  proof
    fix i
    show ‹i ≤ n ⟶ (∃ P′ Q′. fst (cpt!i) = P′ ⋈ Q′)›
    proof(induct i)
      case 0
      then show ?case
        apply(rule impI)
        apply(rule exI)+
        apply(subst hd-conv-nth[THEN sym])
        apply(rule ‹cpt≠[]›)
        apply(rule fst-hd-cpt)
        done
    next
      case (Suc i)
      show ?case
      proof
        assume Suc-i-le: ‹Suc i ≤ n›
        then have ‹i ≤ n› by simp
        with Suc obtain P′ Q′ where fst-cpt-i: ‹fst (cpt ! i) = P′ ⋈ Q′› by blast
        from Suc-i-le n-lt have Suc-i-lt: ‹Suc i < length cpt› by linarith
       have ‹Suc i < length cpt ⟶ (cpt ! i, cpt ! Suc i) ∈ estran Γ ∨ cpt ! i −e⟶
cpt ! Suc i›
          by (rule ct-or-et[THEN spec[where x=i]])
        with Suc-i-lt have ct-or-et′:
          ‹(cpt ! i, cpt ! Suc i) ∈ estran Γ ∨ cpt ! i −e⟶ cpt ! Suc i› by blast
        then show ‹∃ P′ Q′. fst (cpt ! Suc i) = P′ ⋈ Q′›
        proof
          assume ctran: ‹(cpt ! i, cpt ! Suc i) ∈ estran Γ›
          show ‹∃ P′ Q′. fst (cpt ! Suc i) = P′ ⋈ Q′›
```

proof(*cases* ‹*fst* (*cpt*!*Suc* *i*) = *fin*›)
case *True*
have *1*: ‹(*fin*, *snd* (*cpt*!*Suc* *i*)) # *drop* (*Suc* (*Suc* *i*)) *cpt* ∈ *cpts* (*estran* Γ)›
proof−
have *cpt-Suc-i*: ‹*cpt*!*Suc* *i* = (*fin*, *snd* (*cpt*!*Suc* *i*))›
apply(*subst True*[*THEN sym*]) **by** *simp*
moreover have ‹*drop* (*Suc* *i*) *cpt* ∈ *cpts* (*estran* Γ)› **by** (*rule cpts-drop*[*OF cpt Suc-i-lt*])
ultimately show *?thesis*
by (*simp add*: *Cons-nth-drop-Suc Suc-i-lt*)
qed
let *?cpt'* = ‹*drop* (*Suc* (*Suc* *i*)) *cpt*›
have ‹∀ *c*∈*set* *?cpt'*. *fst* *c* = *fin*› **by** (*rule all-fin-after-fin*[*OF 1*])
then have ‹∀ *j*<*length* *?cpt'*. *fst* (*?cpt'*!*j*) = *fin*› **using** *nth-mem* **by** *blast*
then have *all-fin*: ‹∀ *j*. *Suc* (*Suc* *i*) + *j* < *length* *cpt* ⟶ *fst* (*cpt*!(*Suc* (*Suc* *i*) + *j*)) = *fin*› **by** *auto*
have ‹*fst* (*cpt*!*n*) = *fin*›
proof(*cases* ‹*Suc* *i* = *n*›)
case *True*
then show *?thesis* **using** ‹*fst* (*cpt* ! *Suc* *i*) = *fin*› **by** *simp*
next
case *False*
with ‹*Suc* *i* ≤ *n*› **have** ‹*Suc* (*Suc* *i*) ≤ *n*› **by** *linarith*
then show *?thesis* **using** *all-fin n-lt le-Suc-ex* **by** *blast*
qed
with *not-fin* **have** *False* **by** *blast*
then show *?thesis* **by** *blast*
next
case *False*
from *Suc* ‹*i*≤*n*› **obtain** *P'* *Q'* **where** *1*: ‹*fst* (*cpt* ! *i*) = *P'* ⋈ *Q'*› **by** *blast*
show *?thesis* **by** (*rule all-join-aux*[*OF ctran 1 False*])
qed
next
assume *etran*: ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc* *i*›
then show ‹∃ *P'* *Q'*. *fst* (*cpt* ! *Suc* *i*) = *P'* ⋈ *Q'*›
apply *simp*
using *fst-cpt-i* **by** *metis*
qed
qed
qed
qed
qed

lemma *all-join-aux'*:
‹*fst* (*cpt* ! *m*) = *fin* ⟹ *length* (*fst* (*split* *cpt*)) ≤ *m* ∧ *length* (*snd* (*split* *cpt*)) ≤ *m*›
apply(*induct cpt arbitrary*:*m rule*:*split.induct*; *simp*)

```
  apply(case-tac m; simp)
  done

lemma all-join1:
  ‹∀ i < length (fst (split cpt)). ∃ P' Q'. fst (cpt!i) = P' ⋈ Q'›
  apply(induct cpt rule:split.induct, auto)
  apply(case-tac i; simp)
  done

lemma all-join2:
  ‹∀ i < length (snd (split cpt)). ∃ P' Q'. fst (cpt!i) = P' ⋈ Q'›
  apply(induct cpt rule:split.induct, auto)
  apply(case-tac i; simp)
  done

lemma split-length:
  ‹cpt ∈ cpts (estran Γ) ⟹
   fst (hd cpt) = P ⋈ Q ⟹
   Suc m < length cpt ⟹
   fst (cpt ! m) ≠ fin ⟹
   fst (cpt ! Suc m) = fin ⟹
   length (fst (split cpt)) = Suc m ∧ length (snd (split cpt)) = Suc m›
proof(induct cpt arbitrary: P Q m rule: split.induct; simp)
  fix P Q s Pa Qa m
  fix rest
  assume IH:
    ‹⋀P Q m.
     rest ∈ cpts (estran Γ) ⟹
     fst (hd rest) = P ⋈ Q ⟹
     Suc m < length rest ⟹ fst (rest ! m) ≠ fin ⟹ fst (rest ! Suc m) = fin ⟹
length (fst (split rest)) = Suc m ∧ length (snd (split rest)) = Suc m›
  assume a1: ‹(Pa ⋈ Qa, s) # rest ∈ cpts (estran Γ)›
  assume a2: ‹m < length rest›
  then have ‹rest≠[]› by fastforce
  from cpts-tl[OF a1] this have 1: ‹rest ∈ cpts (estran Γ)› by simp
  assume a3: ‹fst (((Pa ⋈ Qa, s) # rest) ! m) ≠ fin›
  from all-join[OF a1] a2 a3 have 2: ‹∀ i≤m. ∃ P' Q'. fst (((Pa ⋈ Qa, s) # rest)
! i) = P' ⋈ Q'›
    by (metis fstI length-Cons less-SucI list.sel(1))
  assume a4: ‹fst (rest ! m) = fin›
  show ‹length (fst (split rest)) = m ∧ length (snd (split rest)) = m›
  proof(cases ‹m=0›)
    case True
    with a4 have ‹fst (rest ! 0) = fin› by simp
    with hd-conv-nth[OF ‹rest≠[]›] have ‹fst (hd rest) = fin› by simp
    then obtain t where ‹hd rest = (fin,t)› using surjective-pairing by metis
    then have ‹rest = (fin,t) # tl rest› using hd-Cons-tl[OF ‹rest≠[]›] by simp
    then have ‹split rest = ([],[])› apply− apply(erule ssubst) by simp
    then show ?thesis using True by simp
```

next
case *False*
then have ‹$m \geq 1$› **by** *fastforce*
from *2*[*rule-format*, *of 1*, *OF this*] **obtain** P' Q' **where** ‹*fst* (((*Pa* ⋈ *Qa*, *s*) # *rest*) ! *1*) = P' ⋈ Q'› **by** *blast*
with *hd-conv-nth*[*OF* ‹*rest*≠[]›] **have** *fst-hd-rest*: ‹*fst* (*hd rest*) = P' ⋈ Q'› **by** *simp*
from *not0-implies-Suc*[*OF False*] **obtain** m' **where** m': ‹m = *Suc* m'› **by** *blast*
from *a2* m' **have** *Suc-m'-lt*: ‹*Suc* m' < *length rest*› **by** *simp*
from *a3* m' **have** *not-fin*: ‹*fst* (*rest* ! m') ≠ *fin*› **by** *simp*
from *a4* m' **have** *fin*: ‹*fst* (*rest* ! *Suc* m') = *fin*› **by** *simp*
from *IH*[*OF 1 fst-hd-rest Suc-m'-lt not-fin fin*] m' **show** *?thesis* **by** *simp*
qed
qed

lemma *split-prog1*:
‹*i* < *length* (*fst* (*split cpt*)) ⟹ *fst* (*cpt*!*i*) = *P* ⋈ *Q* ⟹ *fst* (*fst* (*split cpt*) ! *i*) = *P*›
apply(*induct cpt arbitrary*:*i rule*:*split.induct*, *auto*)
apply(*case-tac i*; *simp*)
done

lemma *split-prog2*:
‹*i* < *length* (*snd* (*split cpt*)) ⟹ *fst* (*cpt*!*i*) = *P* ⋈ *Q* ⟹ *fst* (*snd* (*split cpt*) ! *i*) = *Q*›
apply(*induct cpt arbitrary*:*i rule*:*split.induct*, *auto*)
apply(*case-tac i*; *simp*)
done

lemma *split-ctran-aux*:
‹((*P* ⋈ *Q*, *s*), P' ⋈ Q', *t*) ∈ *estran* Γ ⟹
((*P*, *s*), P', *t*) ∈ *estran* Γ ∧ *Q* = Q' ∨ ((*Q*, *s*), Q', *t*) ∈ *estran* Γ ∧ *P* = P'›
apply(*simp add*: *estran-def*, *erule exE*)
apply(*erule estran-p.cases*, *auto*)
done

lemma *split-ctran*:
assumes *cpt*: *cpt* ∈ *cpts* (*estran* Γ)
assumes *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*›
assumes *not-fin* : ‹*fst* (*cpt*!*Suc i*) ≠ *fin*›
assumes *Suc-i-lt*: *Suc i* < *length cpt*
assumes *ctran*: (*cpt*!*i*, *cpt*!*Suc i*) ∈ *estran* Γ
shows
‹(*fst* (*split cpt*) ! *i*, *fst* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *snd* (*split cpt*) ! *i* −*e*→ *snd* (*split cpt*) ! *Suc i* ∨
(*snd* (*split cpt*) ! *i*, *snd* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *fst* (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i*›
proof−
have *all-All'*: ‹∀ *j*≤*Suc i*. ∃ P' Q'. *fst* (*cpt* ! *j*) = P' ⋈ Q'› **by** (*rule all-join*[*OF*

cpt fst-hd-cpt Suc-i-lt not-fin])
  show *?thesis*
    using *cpt fst-hd-cpt Suc-i-lt ctran all-All'*
  proof(*induct arbitrary*:*P Q i*)
    case (*CptsOne P s*)
    then show *?case* **by** *simp*
  next
    case (*CptsEnv P1 t cs s*)
    from *CptsEnv*(*3*) **have** *1*: ‹*fst* (*hd* ((*P1*, *t*) # *cs*)) = *P* ⋈ *Q*› **by** *simp*
    show *?case*
    proof(*cases i*)
      case *0*
      with *CptsEnv* **show** *?thesis*
        apply (*simp add*: *split-def*)
        using *no-estran-to-self'* **by** *blast*
    next
      case (*Suc i'*)
      with *CptsEnv* **have**
        ‹(*fst* (*split* ((*P1*, *t*) # *cs*)) ! *i'*, *fst* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i'*) ∈ *estran* Γ ∧ *snd* (*split* ((*P1*, *t*) # *cs*)) ! *i'* −*e*→ *snd* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i'* ∨
        (*snd* (*split* ((*P1*, *t*) # *cs*)) ! *i'*, *snd* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i'*) ∈ *estran* Γ ∧ *fst* (*split* ((*P1*, *t*) # *cs*)) ! *i'* −*e*→ *fst* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i'*›
        by *fastforce*
      then show *?thesis* **using** *Suc 1* **by** *simp*
    qed
  next
    case (*CptsComp P1 s Q1 t cs*)
    from *CptsComp*(*7*)[*THEN spec*[**where** *x*=*1*]] **obtain** *P' Q'* **where** *Q1*: ‹*Q1* = *P'* ⋈ *Q'*› **by** *auto*
    show *?case*
    proof(*cases i*)
      case *0*
      with *Q1 CptsComp* **show** *?thesis*
        apply(*simp add*: *split-def*)
        using *split-ctran-aux* **by** *fast*
    next
      case (*Suc i'*)
      from *Q1* **have** *1*: ‹*fst* (*hd* ((*Q1*, *t*) # *cs*)) = *P'* ⋈ *Q'*› **by** *simp*
      from *CptsComp*(*5*) *Suc* **have** *2*: ‹*Suc i'* < *length* ((*Q1*, *t*) # *cs*)› **by** *simp*
      from *CptsComp*(*6*) *Suc* **have** *3*: ‹(((*Q1*, *t*) # *cs*) ! *i'*, ((*Q1*, *t*) # *cs*) ! *Suc i'*) ∈ *estran* Γ› **by** *simp*
      from *CptsComp*(*7*) *Suc* **have** *4*: ‹∀ *j*≤*Suc i'*. ∃ *P' Q'*. *fst* (((*Q1*, *t*) # *cs*) ! *j*) = *P'* ⋈ *Q'*› **by** *auto*
      have
        ‹(*fst* (*split* ((*Q1*, *t*) # *cs*)) ! *i'*, *fst* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i'*) ∈ *estran* Γ ∧ *snd* (*split* ((*Q1*, *t*) # *cs*)) ! *i'* −*e*→ *snd* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i'* ∨
        (*snd* (*split* ((*Q1*, *t*) # *cs*)) ! *i'*, *snd* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i'*) ∈ *estran* Γ ∧ *fst* (*split* ((*Q1*, *t*) # *cs*)) ! *i'* −*e*→ *fst* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i'*›
        by (*rule CptsComp*(*3*)[*OF 1 2 3 4*])

with *Suc CptsComp*(*4*) **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *etran-imp-not-ctran*:
⟨*c1* −*e*→ *c2* ⟹ ¬((*c1*,*c2*) ∈ *estran* Γ)⟩
using *no-estran-to-self* ′′ **by** *fastforce*

lemma *split-etran1-aux*:
⟨((*P* ′ ⋈ *Q*, *s*), *P* ′ ⋈ *Q*′, *t*) ∈ *estran* Γ ⟹ *P* = *P* ′ ⟹ ((*Q*, *s*), *Q*′, *t*) ∈ *estran* Γ⟩
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*erule estran-p.cases*, *auto*)
using *no-estran-to-self* **by** *blast*

lemma *split-etran1*:
assumes *cpt*: ⟨*cpt* ∈ *cpts* (*estran* Γ)⟩
and *fst-hd-cpt*: ⟨*fst* (*hd cpt*) = *P* ⋈ *Q*⟩
and *Suc-i-lt*: ⟨*Suc i* < *length cpt*⟩
and *not-fin*: ⟨*fst* (*cpt* ! *Suc i*) ≠ *fin*⟩
and *etran*: ⟨*fst* (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i*⟩
shows
⟨*cpt* ! *i* −*e*→ *cpt* ! *Suc i* ∨
(*snd* (*split cpt*) ! *i*, *snd* (*split cpt*) ! *Suc i*) ∈ *estran* Γ⟩
proof−
have *all-All*′: ⟨∀ *j*≤*Suc i*. ∃ *P* ′ *Q*′. *fst* (*cpt* ! *j*) = *P* ′ ⋈ *Q*′⟩
by (*rule all-join*[*OF cpt fst-hd-cpt Suc-i-lt not-fin*])
show *?thesis*
using *cpt fst-hd-cpt Suc-i-lt not-fin etran all-All*′
proof(*induct arbitrary*:*P Q i*)
case (*CptsOne P s*)
then show *?case* **by** *simp*
next
case (*CptsEnv P1 t cs s*)
show *?case*
proof(*cases i*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc i*′)
from *CptsEnv*(*3*) **have** *1*: ⟨*fst* (*hd* ((*P1*, *t*) # *cs*)) = *P* ⋈ *Q*⟩ **by** *simp*
then have *P1*: ⟨*P1* = *P* ⋈ *Q*⟩ **by** *simp*
from *CptsEnv*(*4*) *Suc* **have** *2*: ⟨*Suc i* ′ < *length* ((*P1*, *t*) # *cs*)⟩ **by** *simp*
from *CptsEnv*(*5*) *Suc* **have** *3*: ⟨*fst* (((*P1*, *t*) # *cs*) ! *Suc i*′) ≠ *fin*⟩ **by** *simp*
from *CptsEnv*(*6*) *Suc P1*
have *4*: ⟨*fst* (*split* ((*P1*, *t*) # *cs*)) ! *i* ′ −*e*→ *fst* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i*′⟩ **by** *simp*

from *CptsEnv*(*7*) *Suc* **have** *5*: ‹∀*j*≤*Suc i*′. ∃*P*′ *Q*′. *fst* (((*P1*, *t*) # *cs*) ! *j*) = *P*′ ⋈ *Q*′› **by** *auto*
from *CptsEnv*(*2*)[*OF 1 2 3 4 5*]
have ‹((*P1*, *t*) # *cs*) ! *i*′ −*e*→ ((*P1*, *t*) # *cs*) ! *Suc i*′ ∨ (*snd* (*split* ((*P1*, *t*) # *cs*)) ! *i*′, *snd* (*split* ((*P1*, *t*) # *cs*)) ! *Suc i*′) ∈ *estran* Γ› **.**
then show *?thesis* **using** *Suc P1* **by** *simp*
qed
next
case (*CptsComp P1 s Q1 t cs*)
from *CptsComp*(*4*) **have** *P1*: ‹*P1* = *P* ⋈ *Q*› **by** *simp*
from *CptsComp*(*8*)[*THEN spec*[**where** *x*=*1*]] **obtain** *P*′ *Q*′ **where** *Q1*: ‹*Q1* = *P*′ ⋈ *Q*′› **by** *auto*
show *?case*
proof(*cases i*)
case *0*
with *P1 Q1 CptsComp*(*1*) *CptsComp*(*7*) **show** *?thesis*
apply (*simp add*: *split-def*)
apply(*rule disjI2*)
apply(*erule split-etran1-aux*, *assumption*)
done
next
case (*Suc i*′)
have *1*: ‹*fst* (*hd* ((*Q1*, *t*) # *cs*)) = *P*′ ⋈ *Q*′› **using** *Q1* **by** *simp*
from *CptsComp*(*5*) *Suc* **have** *2*: ‹*Suc i*′ < *length* ((*Q1*, *t*) # *cs*)› **by** *simp*
from *CptsComp*(*6*) *Suc* **have** *3*: ‹*fst* (((*Q1*, *t*) # *cs*) ! *Suc i*′) ≠ *fin*› **by** *simp*
from *CptsComp*(*7*) *Suc P1* **have** *4*: ‹*fst* (*split* ((*Q1*, *t*) # *cs*)) ! *i*′ −*e*→ *fst* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i*′› **by** *simp*
from *CptsComp*(*8*) *Suc* **have** *5*: ‹∀*j*≤*Suc i*′. ∃*P*′ *Q*′. *fst* (((*Q1*, *t*) # *cs*) ! *j*) = *P*′ ⋈ *Q*′› **by** *auto*
from *CptsComp*(*3*)[*OF 1 2 3 4 5*]
have ‹((*Q1*, *t*) # *cs*) ! *i*′ −*e*→ ((*Q1*, *t*) # *cs*) ! *Suc i*′ ∨ (*snd* (*split* ((*Q1*, *t*) # *cs*)) ! *i*′, *snd* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i*′) ∈ *estran* Γ› **.**
then show *?thesis* **using** *Suc P1* **by** *simp*
qed
qed
qed

lemma *split-etran2-aux*:
‹((*P* ⋈ *Q*′, *s*), *P*′ ⋈ *Q*′, *t*) ∈ *estran* Γ ⟹ *Q* = *Q*′ ⟹ ((*P*, *s*), *P*′, *t*) ∈ *estran* Γ›
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*erule estran-p.cases*, *auto*)
using *no-estran-to-self* **by** *blast*

lemma *split-etran2*:
assumes *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)›
and *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*›
and *Suc-i-lt*: ‹*Suc i* < *length cpt*›

```
    and not-fin: ‹fst (cpt ! Suc i) ≠ fin›
    and etran: ‹snd (split cpt) ! i −e→ snd (split cpt) ! Suc i›
  shows
    ‹cpt ! i −e→ cpt ! Suc i ∨
     (fst (split cpt) ! i, fst (split cpt) ! Suc i) ∈ estran Γ›
proof−
  have all-All′: ‹∀ j≤Suc i. ∃ P′ Q′. fst (cpt ! j) = P′ ⋈ Q′›
    by (rule all-join[OF cpt fst-hd-cpt Suc-i-lt not-fin])
  show ?thesis
    using cpt fst-hd-cpt Suc-i-lt not-fin etran all-All′
  proof(induct arbitrary:P Q i)
    case (CptsOne P s)
    then show ?case by simp
  next
    case (CptsEnv P1 t cs s)
    show ?case
    proof(cases i)
      case 0
      then show ?thesis by simp
    next
      case (Suc i′)
      from CptsEnv(3) have 1: ‹fst (hd ((P1, t) # cs)) = P ⋈ Q› by simp
      then have P1: ‹P1 = P ⋈ Q› by simp
      from CptsEnv(4) Suc have 2: ‹Suc i′ < length ((P1, t) # cs)› by simp
      from CptsEnv(5) Suc have 3: ‹fst (((P1, t) # cs) ! Suc i′) ≠ fin› by simp
      from CptsEnv(6) Suc P1 have 4: ‹snd (split ((P1, t) # cs)) ! i′ −e→ snd
(split ((P1, t) # cs)) ! Suc i′› by simp
      from CptsEnv(7) Suc have 5: ‹∀ j≤Suc i′. ∃ P′ Q′. fst (((P1, t) # cs) ! j)
= P′ ⋈ Q′› by auto
      have ‹((P1, t) # cs) ! i′ −e→ ((P1, t) # cs) ! Suc i′ ∨ (fst (split ((P1, t)
# cs)) ! i′, fst (split ((P1, t) # cs)) ! Suc i′) ∈ estran Γ›
        by (rule CptsEnv(2)[OF 1 2 3 4 5])
      then show ?thesis using Suc P1 by simp
    qed
  next
    case (CptsComp P1 s Q1 t cs)
    from CptsComp(4) have P1: ‹P1 = P ⋈ Q› by simp
    from CptsComp(8)[THEN spec[where x=1]] obtain P′ Q′ where Q1: ‹Q1
= P′ ⋈ Q′› by auto
    show ?case
    proof(cases i)
      case 0
      with P1 Q1 CptsComp(1) CptsComp(7) show ?thesis
        apply (simp add: split-def)
        apply(rule disjI2)
        apply(erule split-etran2-aux, assumption)
        done
    next
      case (Suc i′)
```

have *1*: ‹*fst* (*hd* ((*Q1*, *t*) # *cs*)) = *P* ′ ⋈ *Q*′› **using** *Q1* **by** *simp*
from *CptsComp*(*5*) *Suc* **have** *2*: ‹*Suc i*′ < *length* ((*Q1*, *t*) # *cs*)› **by** *simp*
from *CptsComp*(*6*) *Suc* **have** *3*: ‹*fst* (((*Q1*, *t*) # *cs*) ! *Suc i*′) ≠ *fin*› **by** *simp*
from *CptsComp*(*7*) *Suc P1* **have** *4*: ‹*snd* (*split* ((*Q1*, *t*) # *cs*)) ! *i*′ −*e*→ *snd* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i*′› **by** *simp*
from *CptsComp*(*8*) *Suc* **have** *5*: ‹∀ *j*≤*Suc i*′. ∃ *P*′ *Q*′. *fst* (((*Q1*, *t*) # *cs*) ! *j*) = *P*′ ⋈ *Q*′› **by** *auto*
have ‹((*Q1*, *t*) # *cs*) ! *i*′ −*e*→ ((*Q1*, *t*) # *cs*) ! *Suc i*′ ∨ (*fst* (*split* ((*Q1*, *t*) # *cs*)) ! *i*′, *fst* (*split* ((*Q1*, *t*) # *cs*)) ! *Suc i*′) ∈ *estran* Γ›
by (*rule CptsComp*(*3*)[*OF 1 2 3 4 5*])
then show *?thesis* **using** *Suc P1* **by** *simp*
qed
qed
qed

lemma *split-ctran1-aux*:
‹*i* < *length* (*fst* (*split cpt*)) =⇒
fst (*cpt*!*i*) ≠ *fin*›
apply(*induct cpt arbitrary*: *i rule*: *split.induct*, *auto*)
apply(*case-tac i*; *simp*)
done

lemma *split-ctran1*:
‹*cpt* ∈ *cpts* (*estran* Γ) =⇒
fst (*hd cpt*) = *P* ⋈ *Q* =⇒
Suc i < *length* (*fst* (*split cpt*)) =⇒
(*fst* (*split cpt*) ! *i*, *fst* (*split cpt*) ! *Suc i*) ∈ *estran* Γ =⇒
(*cpt*!*i*, *cpt*!*Suc i*) ∈ *estran* Γ›
proof(*rule ccontr*)
assume *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)›
assume *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*›
assume *Suc-i-lt1*: ‹*Suc i* < *length* (*fst* (*split cpt*))›
with *split-length-le1*[*of cpt*]
have *Suc-i-lt*: ‹*Suc i* < *length cpt*› **by** *fastforce*
assume *ctran1*: ‹(*fst* (*split cpt*) ! *i*, *fst* (*split cpt*) ! *Suc i*) ∈ *estran* Γ›
assume ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∉ *estran* Γ›
with *ctran-or-etran*[*OF cpt Suc-i-lt*] **have** *etran*: ‹*cpt*!*i* −*e*→ *cpt*!*Suc i*› **by** *blast*
from *split-ctran1-aux*[*OF Suc-i-lt1*] **have** ‹*fst* (*cpt* ! *Suc i*) ≠ *fin*› **.**
from *split-etran*[*OF cpt fst-hd-cpt Suc-i-lt etran this*, *THEN conjunct1*] **have** ‹*fst* (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i*› **.**
with *ctran1 no-estran-to-self* ′′ **show** *False* **by** *fastforce*
qed

lemma *split-ctran2-aux*:
‹*i* < *length* (*snd* (*split cpt*)) =⇒
fst (*cpt*!*i*) ≠ *fin*›
apply(*induct cpt arbitrary*: *i rule*: *split.induct*, *auto*)
apply(*case-tac i*; *simp*)
done

lemma *split-ctran2*:
‹$cpt \in cpts\ (estran\ \Gamma) \Longrightarrow$
$fst\ (hd\ cpt) = P \bowtie Q \Longrightarrow$
$Suc\ i < length\ (snd\ (split\ cpt)) \Longrightarrow$
$(snd\ (split\ cpt)\ !\ i,\ snd\ (split\ cpt)\ !\ Suc\ i) \in estran\ \Gamma \Longrightarrow$
$(cpt!i,\ cpt!Suc\ i) \in estran\ \Gamma$›
proof(*rule ccontr*)
assume *cpt*: ‹$cpt \in cpts\ (estran\ \Gamma)$›
assume *fst-hd-cpt*: ‹$fst\ (hd\ cpt) = P \bowtie Q$›
assume *Suc-i-lt2*: ‹$Suc\ i < length\ (snd\ (split\ cpt))$›
with *split-length-le2*[*of cpt*]
have *Suc-i-lt*: ‹$Suc\ i < length\ cpt$› **by** *fastforce*
assume *ctran2*: ‹$(snd\ (split\ cpt)\ !\ i,\ snd\ (split\ cpt)\ !\ Suc\ i) \in estran\ \Gamma$›
assume ‹$(cpt\ !\ i,\ cpt\ !\ Suc\ i) \notin estran\ \Gamma$›
with *ctran-or-etran*[*OF cpt Suc-i-lt*] **have** *etran*: ‹$cpt!i\ -e\rightarrow\ cpt!Suc\ i$› **by** *blast*
from *split-ctran2-aux*[*OF Suc-i-lt2*] **have** ‹$fst\ (cpt\ !\ Suc\ i) \neq fin$› **.**
from *split-etran*[*OF cpt fst-hd-cpt Suc-i-lt etran this*, *THEN conjunct2*] **have** ‹$snd\ (split\ cpt)\ !\ i\ -e\rightarrow\ snd\ (split\ cpt)\ !\ Suc\ i$› **.**
with *ctran2 no-estran-to-self''* **show** *False* **by** *fastforce*
qed

lemma *no-fin-before-non-fin*:
assumes *cpt*: ‹$cpt \in cpts\ (estran\ \Gamma)$›
and *m-lt*: ‹$m < length\ cpt$›
and *m-not-fin*: $fst\ (cpt!m) \neq fin$
and ‹$i \le m$›
shows ‹$fst\ (cpt!i) \neq fin$›
proof(*rule ccontr*, *simp*)
assume *i-fin*: ‹$fst\ (cpt!i) = fin$›
from *m-lt* ‹$i \le m$› **have** *i-lt*: ‹$i < length\ cpt$› **by** *simp*
from *cpts-drop*[*OF cpt this*] **have** ‹$drop\ i\ cpt \in cpts\ (estran\ \Gamma)$› **by** *assumption*
have *1*: ‹$drop\ i\ cpt = (fin,\ snd\ (cpt!i))\ \#\ drop\ (Suc\ i)\ cpt$› **using** *i-fin i-lt*
by (*metis Cons-nth-drop-Suc surjective-pairing*)
from *cpts-drop*[*OF cpt i-lt*] **have** ‹$drop\ i\ cpt \in cpts\ (estran\ \Gamma)$› **by** *assumption*
with *1* **have** ‹$(fin,\ snd\ (cpt!i))\ \#\ drop\ (Suc\ i)\ cpt \in cpts\ (estran\ \Gamma)$› **by** *simp*
from *all-fin-after-fin*[*OF this*] **have** ‹$\forall\ c \in set\ (drop\ (Suc\ i)\ cpt).\ fst\ c = fin$› **by** *assumption*
then have ‹$\forall\ j < length\ (drop\ (Suc\ i)\ cpt).\ fst\ (drop\ (Suc\ i)\ cpt\ !\ j) = fin$› **using** *nth-mem* **by** *blast*
then have *2*: ‹$\forall\ j.\ Suc\ i + j < length\ cpt \longrightarrow fst\ (cpt\ !\ (Suc\ i + j)) = fin$› **by** *simp*
find-theorems *nth drop*
show *False*
proof(*cases* ‹$i=m$›)
case *True*
then show *False* **using** *m-not-fin i-fin* **by** *simp*
next
case *False*

with ‹$i \le m$› **have** ‹$i < m$› **by** *simp*
with *2 m-not-fin* **show** *False*
using *Suc-leI le-Suc-ex m-lt* **by** *blast*
qed
qed

lemma *no-estran-from-fin'*:
‹$(c1,\ c2) \in estran\ \Gamma \Longrightarrow fst\ c1 \neq fin$›
apply(*simp add*: *estran-def*)
apply(*subst* (*asm*) *surjective-pairing*[*of c1*])
using *no-estran-from-fin* **by** *metis*

3.1 Compositionality of the Semantics

3.1.1 Definition of the conjoin operator

definition *same-length* :: $('l,'k,'s,'prog)$ *pesconf list* $\Rightarrow$ $('k \Rightarrow ('l,'k,'s,'prog)$ *esconf list*$)$ $\Rightarrow$ *bool* **where**
$same\text{-}length\ c\ cs \equiv \forall k.\ length\ (cs\ k) = length\ c$

definition *same-state* :: $('l,'k,'s,'prog)$ *pesconf list* $\Rightarrow$ $('k \Rightarrow ('l,'k,'s,'prog)$ *esconf list*$)$ $\Rightarrow$ *bool* **where**
$same\text{-}state\ c\ cs \equiv \forall k\ j.\ j < length\ c \longrightarrow snd\ (c!j) = snd\ (cs\ k\ !\ j)$

definition *same-spec* :: $('l,'k,'s,'prog)$ *pesconf list* $\Rightarrow$ $('k \Rightarrow ('l,'k,'s,'prog)$ *esconf list*$)$ $\Rightarrow$ *bool* **where**
$same\text{-}spec\ c\ cs \equiv \forall k\ j.\ j < length\ c \longrightarrow fst\ (c!j)\ k = fst\ (cs\ k\ !\ j)$

definition *compat-tran* :: $('l,'k,'s,'prog)$ *pesconf list* $\Rightarrow$ $('k \Rightarrow ('l,'k,'s,'prog)$ *esconf list*$)$ $\Rightarrow$ *bool* **where**
$compat\text{-}tran\ c\ cs \equiv$
$\forall j.\ Suc\ j < length\ c \longrightarrow$
$((\exists t\ k\ \Gamma.\ (\Gamma \vdash c!j\ -pes[t\sharp k]\rightarrow\ c!Suc\ j)) \wedge$
$(\forall k\ t\ \Gamma.\ (\Gamma \vdash c!j\ -pes[t\sharp k]\rightarrow\ c!Suc\ j) \longrightarrow$
$(\Gamma \vdash cs\ k\ !\ j\ -es[t\sharp k]\rightarrow\ cs\ k\ !\ Suc\ j) \wedge (\forall k'.\ k' \neq k \longrightarrow (cs\ k'\ !\ j$
$-e\rightarrow\ cs\ k'\ !\ Suc\ j)))) \vee$
$(c!j\ -e\rightarrow\ c!Suc\ j \wedge (\forall k.\ cs\ k\ !\ j\ -e\rightarrow\ cs\ k\ !\ Suc\ j))$

definition *conjoin* :: $('l,'k,'s,'prog)$ *pesconf list* $\Rightarrow$ $('k \Rightarrow ('l,'k,'s,'prog)$ *esconf list*$)$ $\Rightarrow$ *bool* (- $\propto$ - [*65*,*65*] *64*) **where**
$c \propto cs \equiv (same\text{-}length\ c\ cs) \wedge (same\text{-}state\ c\ cs) \wedge (same\text{-}spec\ c\ cs) \wedge (compat\text{-}tran\ c\ cs)$

3.1.2 Properties of the conjoin operator

lemma *conjoin-ctran*:
assumes *conjoin*: ‹$pc \propto cs$›
assumes *Suc-i-lt*: ‹$Suc\ i < length\ pc$›
assumes *ctran*: ‹$\Gamma \vdash pc!i\ -pes[a\sharp k]\rightarrow\ pc!Suc\ i$›
shows

⟨$(\Gamma \vdash cs\ k\ !\ i\ -es[a\sharp k]\rightarrow\ cs\ k\ !\ Suc\ i) \wedge$
$(\forall k'.\ k' \neq k \longrightarrow (cs\ k'\ !\ i\ -e\rightarrow\ cs\ k'\ !\ Suc\ i))$⟩
proof−
from *conjoin* **have** ⟨*compat-tran pc cs*⟩ **using** *conjoin-def* **by** *blast*
then have
h: ⟨$\forall j.\ Suc\ j < length\ pc \longrightarrow$
$(\exists t\ k\ \Gamma.\ \Gamma \vdash pc\ !\ j\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ j) \wedge$
$(\forall k\ t\ \Gamma.\ (\Gamma \vdash pc\ !\ j\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ j) \longrightarrow (\Gamma \vdash cs\ k\ !\ j\ -es[t\sharp k]\rightarrow\ cs\ k\ !\ Suc\ j) \wedge (\forall k'.\ k' \neq k \longrightarrow fst\ (cs\ k'\ !\ j) = fst\ (cs\ k'\ !\ Suc\ j))) \vee$
$fst\ (pc\ !\ j) = fst\ (pc\ !\ Suc\ j) \wedge (\forall k.\ fst\ (cs\ k\ !\ j) = fst\ (cs\ k\ !\ Suc\ j))$⟩ **by** (*simp add*: *compat-tran-def*)
from *ctran* **have** ⟨$fst\ (pc\ !\ i) \neq fst\ (pc\ !\ Suc\ i)$⟩ **using** *no-pestran-to-self* **by** (*metis prod.collapse*)
with *h*[*rule-format*, *OF Suc-i-lt*] **have**
⟨$\forall k\ t\ \Gamma.\ (\Gamma \vdash pc\ !\ i\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ i) \longrightarrow (\Gamma \vdash cs\ k\ !\ i\ -es[t\sharp k]\rightarrow\ cs\ k\ !\ Suc\ i) \wedge (\forall k'.\ k' \neq k \longrightarrow fst\ (cs\ k'\ !\ i) = fst\ (cs\ k'\ !\ Suc\ i))$⟩
by *argo*
from *this*[*rule-format*, *OF ctran*] **show** *?thesis* **by** *fastforce*
qed

lemma *conjoin-etran*:
assumes *conjoin*: ⟨$pc \propto cs$⟩
assumes *Suc-i-lt*: ⟨$Suc\ i < length\ pc$⟩
assumes *etran*: ⟨$pc!i\ -e\rightarrow\ pc!Suc\ i$⟩
shows ⟨$\forall k.\ cs\ k\ !\ i\ -e\rightarrow\ cs\ k\ !\ Suc\ i$⟩
proof−
from *conjoin* **have** ⟨*compat-tran pc cs*⟩ **using** *conjoin-def* **by** *blast*
then have
⟨$\forall j.\ Suc\ j < length\ pc \longrightarrow$
$(\exists t\ k\ \Gamma.\ \Gamma \vdash pc\ !\ j\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ j) \wedge$
$(\forall k\ t\ \Gamma.\ (\Gamma \vdash pc\ !\ j\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ j) \longrightarrow (\Gamma \vdash cs\ k\ !\ j\ -es[t\sharp k]\rightarrow\ cs\ k\ !\ Suc\ j) \wedge (\forall k'.\ k' \neq k \longrightarrow fst\ (cs\ k'\ !\ j) = fst\ (cs\ k'\ !\ Suc\ j))) \vee$
$fst\ (pc\ !\ j) = fst\ (pc\ !\ Suc\ j) \wedge (\forall k.\ fst\ (cs\ k\ !\ j) = fst\ (cs\ k\ !\ Suc\ j))$⟩ **by** (*simp add*: *compat-tran-def*)
from *this*[*rule-format*, *OF Suc-i-lt*] **have** *h*:
⟨$(\exists t\ k\ \Gamma.\ \Gamma \vdash pc\ !\ i\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ i) \wedge$
$(\forall k\ t\ \Gamma.\ (\Gamma \vdash pc\ !\ i\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ i) \longrightarrow (\Gamma \vdash cs\ k\ !\ i\ -es[t\sharp k]\rightarrow\ cs\ k\ !\ Suc\ i) \wedge (\forall k'.\ k' \neq k \longrightarrow fst\ (cs\ k'\ !\ i) = fst\ (cs\ k'\ !\ Suc\ i))) \vee$
$fst\ (pc\ !\ i) = fst\ (pc\ !\ Suc\ i) \wedge (\forall k.\ fst\ (cs\ k\ !\ i) = fst\ (cs\ k\ !\ Suc\ i))$⟩ **by** *blast*
from *etran* **have** ⟨$\neg(\exists t\ k\ \Gamma.\ \Gamma \vdash pc\ !\ i\ -pes[t\sharp k]\rightarrow\ pc\ !\ Suc\ i)$⟩ **using** *no-pestran-to-self*
by (*metis* (*mono-tags*, *lifting*) *etran-def etran-p-def mem-Collect-eq prod.simps*(*2*) *surjective-pairing*)
with *h* **have** ⟨$\forall k.\ fst\ (cs\ k\ !\ i) = fst\ (cs\ k\ !\ Suc\ i)$⟩ **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *conjoin-cpt*:
assumes *pc*: ⟨$pc \in cpts\ (pestran\ \Gamma)$⟩
assumes *conjoin*: ⟨$pc \propto cs$⟩

```
  shows ‹cs k ∈ cpts (estran Γ)›
proof-
  from pc cpts-def'[of pc ‹pestran Γ›] have
    ‹pc ≠ []› and 1: ‹(∀ i. Suc i < length pc ⟶ (pc ! i, pc ! Suc i) ∈ pestran Γ ∨
pc ! i −e→ pc ! Suc i)›
    by auto
  from ‹pc≠[]› have ‹length pc ≠ 0› by simp
  then have ‹length (cs k) ≠ 0› using conjoin by (simp add: conjoin-def same-length-def)
  then have ‹cs k ≠ []› by simp
  moreover have ‹∀ i. Suc i < length (cs k) ⟶ (cs k ! i) −e→ (cs k ! Suc i) ∨
(cs k ! i, cs k ! Suc i) ∈ estran Γ›
  proof(rule allI, rule impI)
    fix i
    assume ‹Suc i < length (cs k)›
    then have Suc-i-lt: ‹Suc i < length pc› using conjoin conjoin-def same-length-def
by metis
    from 1[rule-format, OF this]
    have ctran-or-etran-par: ‹(pc ! i, pc ! Suc i) ∈ pestran Γ ∨ pc ! i −e→ pc !
Suc i› by assumption
    then show ‹cs k ! i −e→ cs k ! Suc i ∨ (cs k ! i, cs k ! Suc i) ∈ estran Γ›
    proof
      assume ‹(pc ! i, pc ! Suc i) ∈ pestran Γ›
      then have ‹∃ a k. Γ ⊢ pc!i −pes[a♯k]→ pc!Suc i› by (simp add: pestran-def)
      then obtain a k' where ‹Γ ⊢ pc!i −pes[a♯k']→ pc!Suc i› by blast
      from conjoin-ctran[OF conjoin Suc-i-lt this]
      have 2: ‹(Γ ⊢ cs k' ! i −es[a♯k']→ cs k' ! Suc i) ∧ (∀ k'a. k'a ≠ k' ⟶ cs
k'a ! i −e→ cs k'a ! Suc i)›
        by assumption
      show ?thesis
      proof(cases ‹k'=k›)
        case True
        then show ?thesis
          using 2 apply (simp add: estran-def)
          apply(rule disjI2)
          by auto
      next
        case False
        then show ?thesis using 2 by simp
      qed
    next
      assume ‹pc ! i −e→ pc ! Suc i›
      from conjoin-etran[OF conjoin Suc-i-lt this] show ?thesis
        apply-
        apply (rule disjI1)
        by blast
    qed
  qed
  ultimately show ‹cs k ∈ cpts (estran Γ)› using cpts-def' by blast
qed
```

lemma *conjoin-cpt′*:
  assumes *pc*: ‹*pc* ∈ *cpts-from* (*pestran* Γ) (*Ps*, *s0*)›
  assumes *conjoin*: ‹*pc* ∝ *cs*›
  shows ‹*cs k* ∈ *cpts-from* (*estran* Γ) (*Ps k*, *s0*)›
proof−
  from *pc* **have** *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)› **and** *hd-pc*: ‹*hd pc* = (*Ps*, *s0*)› **by** *auto*
  from *pc-cpt cpts-nonnil* **have** ‹*pc*≠[]› **by** *blast*
  have *ck-cpt*: ‹*cs k* ∈ *cpts* (*estran* Γ)› **using** *conjoin-cpt*[*OF pc-cpt conjoin*] **by** *assumption*
  moreover have ‹*hd* (*cs k*) = (*Ps k*, *s0*)›
  proof−
    from *ck-cpt cpts-nonnil* **have** ‹*cs k* ≠ []› **by** *blast*
    from *conjoin conjoin-def* **have** ‹*same-spec pc cs*› **and** ‹*same-state pc cs*› **by** *blast*+
    then show *?thesis* **using** *hd-pc* ‹*pc*≠[]› ‹*cs k* ≠ []›
      apply(*simp add*: *same-spec-def same-state-def hd-conv-nth*)
      apply(*erule allE*[**where** *x*=*k*])
      apply(*erule allE*[**where** *x*=*0*])
      apply *simp*
      by (*simp add*: *prod-eqI*)
  qed
  ultimately show *?thesis* **by** *auto*
qed

lemma *conjoin-same-length*:
  ‹*pc* ∝ *cs* ⟹ *length pc* = *length* (*cs k*)›
  by (*simp add*: *conjoin-def same-length-def*)

lemma *conjoin-same-spec*:
  ‹*pc* ∝ *cs* ⟹ ∀ *k i*. *i* < *length pc* ⟶ *fst* (*pc*!*i*) *k* = *fst* (*cs k* ! *i*)›
  by (*simp add*: *conjoin-def same-spec-def*)

lemma *conjoin-same-state*:
  ‹*pc* ∝ *cs* ⟹ ∀ *k i*. *i* < *length pc* ⟶ *snd* (*pc*!*i*) = *snd* (*cs k*!*i*)›
  by (*simp add*: *conjoin-def same-state-def*)

lemma *conjoin-all-etran*:
  assumes *conjoin*: ‹*pc* ∝ *cs*›
    and *Suc-i-lt*: ‹*Suc i* < *length pc*›
    and *all-etran*: ‹∀ *k*. *cs k* ! *i* −*e*→ *cs k* ! *Suc i*›
  shows ‹*pc*!*i* −*e*→ *pc*!*Suc i*›
proof−
  from *conjoin-same-spec*[*OF conjoin*]
  have *same-spec*: ‹∀ *k i*. *i* < *length pc* ⟶ *fst* (*pc* ! *i*) *k* = *fst* (*cs k* ! *i*)› **by** *assumption*
  from *same-spec*[*rule-format*, *OF Suc-i-lt*[*THEN Suc-lessD*]]
  have *eq1*: ‹∀ *k*. *fst* (*pc* ! *i*) *k* = *fst* (*cs k* ! *i*)› **by** *blast*

```
  from same-spec[rule-format, OF Suc-i-lt]
  have eq2: ‹∀ k. fst (pc ! Suc i) k = fst (cs k ! Suc i)› by blast
  have ‹∀ k. fst (pc!i) k = fst (pc!Suc i) k›
  proof
    fix k
    from eq1[THEN spec[where x=k]] have 1: ‹fst (pc ! i) k = fst (cs k ! i)› by
assumption
    from eq2[THEN spec[where x=k]] have 2: ‹fst (pc!Suc i) k = fst (cs k ! Suc
i)› by assumption
    from 1 2 all-etran[THEN spec[where x=k]]
    show ‹fst (pc!i) k = fst (pc!Suc i) k› by simp
  qed
  then have ‹fst (pc!i) = fst (pc!Suc i)› by blast
  then show ?thesis by simp
qed

lemma conjoin-etran-k:
  assumes pc: ‹pc ∈ cpts (pestran Γ)›
    and conjoin: ‹pc ∝ cs›
    and Suc-i-lt: ‹Suc i < length pc›
    and etran: ‹cs k!i −e→ cs k!Suc i›
  shows ‹(pc!i −e→ pc!Suc i) ∨ (∃ k'. k'≠k ∧ (cs k'!i, cs k'!Suc i) ∈ estran Γ)›
proof(rule ccontr, clarsimp)
  assume neq: ‹fst (pc ! i) ≠ fst (pc ! Suc i)›
  assume 1: ‹∀ k'. k' = k ∨ (cs k' ! i, cs k' ! Suc i) ∉ estran Γ›
  have ‹∀ k'. cs k' ! i −e→ cs k' ! Suc i›
  proof
    fix k'
    show ‹cs k' ! i −e→ cs k' ! Suc i›
    proof(cases ‹k=k'›)
      case True
      then show ?thesis using etran by blast
    next
      case False
      with 1 have not-ctran: ‹(cs k' ! i, cs k' ! Suc i) ∉ estran Γ› by fast
     from conjoin-same-length[OF conjoin] Suc-i-lt have Suc-i-lt': ‹Suc i < length
(cs k')› by simp
     from conjoin-cpt[OF pc conjoin] have ‹cs k' ∈ cpts (estran Γ)› by assumption
      from ctran-or-etran[OF this Suc-i-lt'] not-ctran
      show ?thesis by blast
    qed
  qed
  from conjoin-all-etran[OF conjoin Suc-i-lt this]
  have ‹fst (pc!i) = fst (pc!Suc i)› by simp
  with neq show False by blast
qed

end
```

end
theory *Validity* **imports** *Computation* **begin**

definition *assume* :: $'s$ *set* ⇒ $('s\times's)$ *set* ⇒ $('p\times's)$ *list set* **where**
assume pre rely ≡ {*cpt. snd* (*hd cpt*) ∈ *pre* ∧ (∀ *i. Suc i* < *length cpt* ⟶ (*cpt*!*i* −*e*→ *cpt*!(*Suc i*)) ⟶ (*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *rely*)}

definition *commit* :: $(('p\times's) \times ('p\times's))$ *set* ⇒ $'p$ *set* ⇒ $('s\times's)$ *set* ⇒ $'s$ *set* ⇒ $('p\times's)$ *list set* **where**
commit tran fin guar post ≡
{*cpt.* (∀ *i. Suc i* < *length cpt* ⟶ (*cpt*!*i*, *cpt*!(*Suc i*)) ∈ *tran* ⟶ (*snd* (*cpt*!*i*), *snd* (*cpt*!(*Suc i*))) ∈ *guar*) ∧
(*fst* (*last cpt*) ∈ *fin* ⟶ *snd* (*last cpt*) ∈ *post*)}

definition *validity* :: $(('p\times's) \times ('p\times's))$ *set* ⇒ $'p$ *set* ⇒ $'p$ ⇒ $'s$ *set* ⇒ $('s\times's)$ *set* ⇒ $('s\times's)$ *set* ⇒ $'s$ *set* ⇒ *bool* **where**
validity tran fin P pre rely guar post ≡ ∀ *s0*. *cpts-from tran* (*P*,*s0*) ∩ *assume pre rely* ⊆ *commit tran fin guar post*

declare *validity-def*[*simp*]

lemma *commit-Cons-env*:
‹∀ *P s t*. ((*P*,*s*),(*P*,*t*)) ∉ *tran* ⟹
(*P*,*t*)#*cpt* ∈ *commit tran fin guar post* ⟹
(*P*,*s*)#(*P*,*t*)#*cpt* ∈ *commit tran fin guar post*›
apply (*simp add*: *commit-def*)
apply *clarify*
apply(*case-tac i*, *auto*)
done

lemma *commit-Cons-comp*:
‹(*Q*,*t*)#*cpt* ∈ *commit tran fin guar post* ⟹
((*P*,*s*),(*Q*,*t*)) ∈ *tran* ⟹
(*s*,*t*)∈*guar* ⟹
(*P*,*s*)#(*Q*,*t*)#*cpt* ∈ *commit tran fin guar post*›
apply (*simp add*: *commit-def*)
apply *clarify*
apply(*case-tac i*, *auto*)
done

lemma *cpts-from-assume-take*:
assumes *h*: *cpt* ∈ *cpts-from tran c* ∩ *assume pre rely*
assumes *i*: *i* ≠ *0*
shows *take i cpt* ∈ *cpts-from tran c* ∩ *assume pre rely*
proof
from *h* **have** ‹*cpt* ∈ *cpts-from tran c*› **by** *blast*
with *i cpts-from-take* **show** ‹*take i cpt* ∈ *cpts-from tran c*› **by** *blast*
next
from *h* **have** ‹*cpt* ∈ *assume pre rely*› **by** *blast*

with i **show** ‹*take i cpt* ∈ *assume pre rely*› **by** (*simp add*: *assume-def*)
qed

lemma *assume-snoc*:
assumes *assume*: ‹*cpt* ∈ *assume pre rely*›
and *nonnil*: ‹*cpt* ≠ []›
and *tran*: ‹¬(*last cpt* −*e*→ *c*)›
shows ‹*cpt*@[*c*] ∈ *assume pre rely*›
using *assume nonnil* **apply** (*simp add*: *assume-def*)
proof
fix *i*
show ‹*i* < *length cpt* ⟶
fst ((*cpt* @ [*c*]) ! *i*) = *fst* ((*cpt* @ [*c*]) ! *Suc i*) ⟶ (*snd* ((*cpt* @ [*c*]) ! *i*),
snd ((*cpt* @ [*c*]) ! *Suc i*)) ∈ *rely*›
proof(*cases* ‹*Suc i* < *length cpt*›)
case *True*
then show *?thesis* **using** *assume nonnil*
apply (*simp add*: *assume-def*)
apply *clarify*
apply(*erule allE*[**where** *x*=*i*])
by (*simp add*: *nth-append*)
next
case *False*
then show *?thesis*
apply *clarsimp*
apply(*subgoal-tac Suc i* = *length cpt*)
apply *simp*
apply (*smt Suc-lessD append-eq-conv-conj etran-def etran-p-def hd-drop-conv-nth last-snoc length-append-singleton lessI mem-Collect-eq prod.simps*(*2*) *take-hd-drop tran*)
apply *simp*
done
qed
qed

lemma *commit-tl*:
‹(*P*,*s*)#(*Q*,*t*)#*cs* ∈ *commit tran fin guar post* ⟹
(*Q*,*t*)#*cs* ∈ *commit tran fin guar post*›
apply(*unfold commit-def*)
apply(*unfold mem-Collect-eq*)
apply *clarify*
apply(*rule conjI*)
apply *fastforce*
by *simp*

lemma *assume-appendD*:
‹(*P*,*s*)#*cs*@*cs*′ ∈ *assume pre rely* ⟹ (*P*,*s*)#*cs* ∈ *assume pre rely*›
apply(*auto simp add*: *assume-def*)
apply(*erule-tac x*=*i* **in** *allE*)

apply *auto*
apply (*metis append-Cons length-Cons lessI less-trans nth-append*)
by (*metis Suc-diff-1 Suc-lessD linorder-neqE-nat nth-Cons′ nth-append zero-order*(*3*))

lemma *assume-appendD2*:
‹*cs*@*cs′* ∈ *assume pre rely* ⟹ ∀ *i*. *Suc i* < *length cs′* ⟶ *cs′*!*i* −*e*→ *cs′*!*Suc i* ⟶ (*snd*(*cs′*!*i*), *snd*(*cs′*!*Suc i*))∈*rely*›
apply(*auto simp add*: *assume-def*)
apply(*erule-tac x*=‹*length cs*+*i*› **in** *allE*)
apply *simp*
by (*metis add-Suc-right nth-append-length-plus*)

lemma *commit-append*:
assumes *cmt1*: ‹*cs* ∈ *commit tran fin guar mid*›
and *guar*: ‹(*snd* (*last cs*), *snd c′*) ∈ *guar*›
and *cmt2*: ‹*c′*#*cs′* ∈ *commit tran fin guar post*›
shows ‹*cs*@*c′*#*cs′* ∈ *commit tran fin guar post*›
apply(*auto simp add*: *commit-def*)
using *cmt1* **apply**(*simp add*: *commit-def*)
using *guar* **apply** (*metis Suc-lessI append-Nil2 append-eq-conv-conj hd-drop-conv-nth nth-append nth-append-length snoc-eq-iff-butlast take-hd-drop*)
using *cmt2* **apply**(*simp add*: *commit-def*)
apply(*case-tac* ‹*Suc i* < *length cs*›)
using *cmt1* **apply**(*simp add*: *commit-def*) **apply** (*simp add*: *nth-append*)
apply(*case-tac* ‹*Suc i* = *length cs*›)
using *guar* **apply** (*metis Cons-nth-drop-Suc drop-eq-Nil id-take-nth-drop last.simps last-appendR le-refl lessI less-irrefl-nat less-le-trans nth-append nth-append-length*)
using *cmt2* **apply**(*simp add*: *commit-def*) **apply** (*simp add*: *nth-append*)
using *cmt2* **apply**(*simp add*: *commit-def*) **.**

lemma *assume-append*:
assumes *asm1*: ‹*cs* ∈ *assume pre rely*›
and *asm2*: ‹∀ *i*. *Suc i* < *length* (*c′*#*cs′*) ⟶ (*c′*#*cs′*)!*i* −*e*→ (*c′*#*cs′*)!*Suc i* ⟶ (*snd*((*c′*#*cs′*)!*i*), *snd*((*c′*#*cs′*)!*Suc i*)) ∈ *rely*›
and *rely*: ‹*last cs* −*e*→ *c′* ⟶ (*snd* (*last cs*), *snd c′*) ∈ *rely*›
and ‹*cs*≠[]›
shows ‹*cs*@*c′*#*cs′* ∈ *assume pre rely*›
using *asm1* ‹*cs*≠[]›
apply(*auto simp add*: *assume-def*)
apply(*case-tac* ‹*Suc i* < *length cs*›)
apply(*erule-tac x*=*i* **in** *allE*)
apply (*metis Suc-lessD append-eq-conv-conj nth-take*)
apply(*case-tac* ‹*Suc i* = *length cs*›)
apply *simp*
using *rely* **apply**(*simp add*: *last-conv-nth*) **apply** (*metis diff-Suc-Suc diff-zero lessI nth-append*)
subgoal for *i*
using *asm2*[*THEN spec*[**where** *x*=‹*i*−*length cs*›]] **by** (*simp add*: *nth-append*)
done

end

4 Rely-guarantee Validity of PiCore Computations

theory *PiCore-Validity*
imports *PiCore-Computation Validity*
begin

4.1 Definitions Correctness Formulas

record $('p,'s)$ *rgformula* =
  Com :: $'p$
  Pre :: $'s$ *set*
  Rely :: $('s \times 's)$ *set*
  Guar :: $('s \times 's)$ *set*
  Post :: $'s$ *set*

locale *event-validity* = *event-comp ptran fin-com*
for *ptran* :: $'Env \Rightarrow (('prog \times 's) \times 'prog \times 's)$ *set*
and *fin-com* :: $'prog$
+
fixes *prog-validity* :: $'Env \Rightarrow 'prog \Rightarrow 's\ set \Rightarrow ('s \times 's)\ set \Rightarrow ('s \times 's)\ set \Rightarrow 's\ set \Rightarrow bool$
  $(\text{-} \models \text{-}\ sat_p\ [\text{-}, \text{-}, \text{-}, \text{-}]\ [60,60,0,0,0,0]\ 45)$

assumes *prog-validity-def*: $\Gamma \models P\ sat_p\ [pre,\ rely,\ guar,\ post] \Longrightarrow validity\ (ptran\ \Gamma)\ \{fin\text{-}com\}\ P\ pre\ rely\ guar\ post$

begin

definition *lift-state-set* :: ‹$'s\ set \Rightarrow ('s \times 'a)\ set$› **where**
  ‹$lift\text{-}state\text{-}set\ P \equiv \{(s,x).s \in P\}$›

definition *lift-state-pair-set* :: ‹$('s \times 's)\ set \Rightarrow (('s \times 'a) \times ('s \times 'a))set$› **where**
  ‹$lift\text{-}state\text{-}pair\text{-}set\ P \equiv \{((s,x),(t,y)).\ (s,t) \in P\}$›

definition *es-validity* :: $'Env \Rightarrow ('l,'k,'s,'prog)\ esys \Rightarrow 's\ set \Rightarrow ('s \times 's)\ set \Rightarrow ('s \times 's)\ set \Rightarrow 's\ set \Rightarrow bool$
  $(\text{-} \models \text{-}\ sat_e\ [\text{-}, \text{-}, \text{-}, \text{-}]\ [60,0,0,0,0,0]\ 45)$ **where**
  $\Gamma \models es\ sat_e\ [pre,\ rely,\ guar,\ post] \equiv validity\ (estran\ \Gamma)\ \{fin\}\ es\ (lift\text{-}state\text{-}set\ pre)\ (lift\text{-}state\text{-}pair\text{-}set\ rely)\ (lift\text{-}state\text{-}pair\text{-}set\ guar)\ (lift\text{-}state\text{-}set\ post)$

declare *es-validity-def*[*simp*]

abbreviation ‹*par-fin* ≡ {*Ps*. ∀ *k*. *Ps k* = *fin*}›

abbreviation ‹*par-com prgf* ≡ λ*k*. *Com* (*prgf k*)›

definition *pes-validity* :: ‹′*Env* ⇒ (′*l*,′*k*,′*s*,′*prog*) *paresys* ⇒ ′*s set* ⇒ (′*s* × ′*s*) *set* ⇒ (′*s* × ′*s*) *set* ⇒ ′*s set* ⇒ *bool*›
(- ⊨ - SAT_e [-, -, -, -] [*60*,*0*,*0*,*0*,*0*,*0*] *45*) **where**
‹Γ ⊨ *Ps* SAT_e [*pre*, *rely*, *guar*, *post*] ≡ *validity* (*pestran* Γ) *par-fin Ps* (*lift-state-set pre*) (*lift-state-pair-set rely*) (*lift-state-pair-set guar*) (*lift-state-set post*)›

declare *pes-validity-def* [*simp*]

lemma *commit-Cons-env-p*:
‹(*P*,*t*)#*cpt* ∈ *commit* (*ptran* Γ) {*fin-com*} *guar post* ⟹ (*P*,*s*)#(*P*,*t*)#*cpt* ∈ *commit* (*ptran* Γ) {*fin-com*} *guar post*›
using *commit-Cons-env ptran-neq* **by** *metis*

lemma *commit-Cons-env-es*:
‹(*P*,*t*)#*cpt* ∈ *commit* (*estran* Γ) {*EAnon fin-com*} *guar post* ⟹ (*P*,*s*)#(*P*,*t*)#*cpt* ∈ *commit* (*estran* Γ) {*EAnon fin-com*} *guar post*›
using *commit-Cons-env no-estran-to-self* ′ **by** *metis*

lemma *cpt-from-ptran-star*:
assumes *h*: ‹Γ ⊢ (*P*, *s0*) −*c*∗→ (*fin-com*, *t*)›
shows ‹∃ *cpt*. *cpt* ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*) ∩ *assume* {*s0*} {} ∧ *last cpt* = (*fin-com*, *t*)›
proof−
from *h* **have** ‹((*P*,*s0*),(*fin-com*,*t*)) ∈(*ptran* Γ)ˆ∗› **by** (*simp add*: *ptrans-def*)
then show *?thesis*
proof(*induct*)
case *base*
show *?case*
proof
show ‹[(*P*,*s0*)] ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*) ∩ *assume* {*s0*} {} ∧ *last* [(*P*,*s0*)] = (*P*, *s0*)›
apply (*simp add*: *assume-def*)
apply(*rule CptsOne*)
done
qed
next
case (*step c c*′)
from *step*(*3*) **obtain** *cpt* **where** *cpt*: ‹*cpt* ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*) ∩ *assume* {*s0*} {} ∧ *last cpt* = *c*› **by** *blast*
with *step* **have** *tran*: ‹(*last cpt*, *c*′) ∈ *ptran* Γ› **by** *simp*
then have *prog-neq*: ‹*fst* (*last cpt*) ≠ *fst c*′› **using** *ptran-neq*
by (*metis prod.exhaust-sel*)
from *cpt* **have** *cpt1*:‹*cpt* ∈ *cpts* (*ptran* Γ)› **by** *simp*
then have *cpt-nonnil*: ‹*cpt* ≠ []› **using** *cpts-nonnil* **by** *blast*
show *?case*

proof
show ‹(*cpt*@[*c'*]) ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*) ∩ *assume* {*s0*} {} ∧ *last* (*cpt*@[*c'*]) = *c'*›
proof
show ‹*cpt* @ [*c'*] ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*) ∩ *assume* {*s0*} {}›
proof
from *cpt1 tran cpts-snoc-comp* **have** ‹*cpt*@[*c'*] ∈ *cpts* (*ptran* Γ)› **by** *blast*
moreover from *cpt* **have** ‹*hd* (*cpt*@[*c'*]) = (*P*, *s0*)›
using *cpt-nonnil* **by** *fastforce*
ultimately show ‹*cpt* @ [*c'*] ∈ *cpts-from* (*ptran* Γ) (*P*, *s0*)› **by** *fastforce*
next
from *cpt* **have** *assume*: ‹*cpt* ∈ *assume* {*s0*} {}› **by** *blast*
then have ‹*snd* (*hd cpt*) ∈ {*s0*}› **using** *assume-def* **by** *blast*
then have *1*: ‹*snd* (*hd* (*cpt*@[*c'*])) ∈ {*s0*}› **using** *cpt-nonnil*
by (*simp add*: *nth-append*)
from *assume* **have** *assume2*: ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt*!*i* −*e*→ *cpt*!(*Suc i*)) ⟶ (*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ {}›
by (*simp add*: *assume-def*)
have *2*: ‹∀ *i*. *Suc i* < *length* (*cpt*@[*c'*]) ⟶ ((*cpt*@[*c'*])!*i* −*e*→ (*cpt*@[*c'*])!(*Suc i*)) ⟶ (*snd* ((*cpt*@[*c'*])!*i*), *snd* ((*cpt*@[*c'*])!*Suc i*)) ∈ {}›
proof
fix *i*
show ‹*Suc i* < *length* (*cpt* @ [*c'*]) ⟶
(*cpt* @ [*c'*]) ! *i* −*e*→ (*cpt* @ [*c'*]) ! *Suc i* ⟶ (*snd* ((*cpt* @ [*c'*]) ! *i*), *snd* ((*cpt* @ [*c'*]) ! *Suc i*)) ∈ {}›
proof
assume *Suc-i*: ‹*Suc i* < *length* (*cpt* @ [*c'*])›
show ‹(*cpt* @ [*c'*]) ! *i* −*e*→ (*cpt* @ [*c'*]) ! *Suc i* ⟶ (*snd* ((*cpt* @ [*c'*]) ! *i*), *snd* ((*cpt* @ [*c'*]) ! *Suc i*)) ∈ {}›
proof(*cases* ‹*Suc i* < *length cpt*›)
case *True*
then show *?thesis* **using** *assume2*
by (*simp add*: *Suc-lessD nth-append*)
next
case *False*
with *Suc-i* **have** ‹*Suc i* = *length cpt*› **by** *fastforce*
then have *i*: *i* = *length cpt* − *1* **by** *fastforce*
find-theorems *last length ?x* − *1*
show *?thesis*
proof
have *eq1*: ‹(*cpt* @ [*c'*]) ! *i* = *last cpt*› **using** *i cpt-nonnil*
by (*simp add*: *last-conv-nth nth-append*)
have *eq2*: ‹(*cpt* @ [*c'*]) ! *Suc i* = *c'*› **using** *Suc-i*
by (*simp add*: ‹*Suc i* = *length cpt*›)
assume ‹(*cpt* @ [*c'*]) ! *i* −*e*→ (*cpt* @ [*c'*]) ! *Suc i*›
with *eq1 eq2* **have** ‹*last cpt* −*e*→ *c'*› **by** *simp*
with *prog-neq* **have** *False* **by** *simp*
then show ‹(*snd* ((*cpt* @ [*c'*]) ! *i*), *snd* ((*cpt* @ [*c'*]) ! *Suc i*)) ∈ {}› **by** *blast*

```
              qed
            qed
          qed
        qed
        from 1 2 assume-def show ‹cpt @ [c'] ∈ assume {s0} {}› by blast
      qed
    next
      show ‹last (cpt @ [c']) = c'› by simp
    qed
  qed
 qed
qed

end

end
```

5 The Rely-guarantee Proof System of PiCore and its Soundness

```
theory PiCore-Hoare
imports PiCore-Validity List-Lemmata
begin
```

5.1 Proof System for Programs

```
definition stable :: 'a set ⇒ ('a × 'a) set ⇒ bool where
  stable P R ≡ ∀ s s'. s ∈ P ⟶ (s, s') ∈ R ⟶ s' ∈ P
```

5.2 Rely-guarantee Condition

```
locale event-hoare = event-validity ptran fin-com prog-validity
for ptran :: 'Env ⇒ (('prog × 's) × 'prog × 's) set
and fin-com :: 'prog
and prog-validity :: 'Env ⇒ 'prog ⇒ 's set ⇒ ('s × 's) set ⇒ ('s × 's) set ⇒ 's
set ⇒ bool
                (- ⊨ - sat_p [-, -, -, -] [60,60,0,0,0,0] 45)
+
fixes rghoare-p :: 'Env ⇒ ['prog, 's set, ('s × 's) set, ('s × 's) set, 's set] ⇒ bool
    (- ⊢ - sat_p [-, -, -, -] [60,60,0,0,0,0] 45)
  assumes rgsound-p: Γ ⊢ P sat_p [pre, rely, guar, post] ⟹ Γ ⊨ P sat_p [pre, rely,
guar, post]
begin

lemma stable-lift:
  ‹stable P R ⟹ stable (lift-state-set P) (lift-state-pair-set R)›
  by (simp add: lift-state-set-def lift-state-pair-set-def stable-def)
```

5.3 Proof System for Events

lemma *estran-anon-inv*:
  assumes ‹((*EAnon p*,*s*,*x*), (*EAnon q*,*t*,*y*))∈*estran* Γ›
  shows ‹((*p*,*s*), (*q*,*t*))∈*ptran* Γ›
  using *assms* **apply**−
  apply(*simp add*: *estran-def*)
  apply(*erule exE*)
  apply(*erule estran-p.cases*, *auto*)
  done

lemma *unlift-cpt*:
  assumes ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAnon p0*, *s0*, *x0*)›
  shows ‹*unlift-cpt cpt* ∈ *cpts-from* (*ptran* Γ) (*p0*, *s0*)›
  using *assms*
proof(*auto*)
  assume *a1*: ‹*cpt* ∈ *cpts* (*estran* Γ)›
  assume *a2*: ‹*hd cpt* = (*EAnon p0*, *s0*, *x0*)›
  show ‹*map* (λ(*p*, *s*, -). (*unlift-prog p*, *s*)) *cpt* ∈ *cpts* (*ptran* Γ)›
    using *a1 a2*
  proof(*induct arbitrary*:*p0 s0 x0*)
    case (*CptsOne P s*)
    then show *?case* **by** *auto*
  next
    case (*CptsEnv P T cs S*)
    obtain *t y* **where** *T*: ‹*T*=(*t*,*y*)› **by** *fastforce*
    from *CptsEnv*(*3*) *T* **have** ‹*hd* ((*P*,*T*)#*cs*) = (*EAnon p0*, *t*, *y*)› **by** *simp*
    from *CptsEnv*(*2*)[*OF this*] **have** ‹*map* (λ*a*. *case a of* (*p*, *s*, -) ⇒ (*unlift-prog p*, *s*)) ((*P*, *T*) # *cs*) ∈ *cpts* (*ptran* Γ)› **.**
    then show *?case* **by** (*auto simp add*: *case-prod-unfold*)
  next
    case (*CptsComp P S Q T cs*)
    from *CptsComp*(*4*) **have** *P*: ‹*P* = *EAnon p0*› **by** *simp*
    obtain *q* **where** *ptran*: ‹((*p0*,*fst S*),(*q*,*fst T*))∈*ptran* Γ› **and** *Q*: ‹*Q* = *EAnon q*›
    proof−
      assume *a*: ‹⋀*q*. ((*p0*, *fst S*), *q*, *fst T*) ∈ *ptran* Γ ⟹ *Q* = *EAnon q* ⟹ *thesis*›
      show *thesis*
        using *CptsComp*(*1*) **apply**(*simp add*: *P estran-def*)
        apply(*erule exE*)
        apply(*erule estran-p.cases*, *auto*)
        apply(*rule a*) **apply** *simp*+
        by (*simp add*: *a*)
    qed
    obtain *t y* **where** *T*: ‹*T*=(*t*,*y*)› **by** *fastforce*
    have ‹*hd* ((*Q*, *T*) # *cs*) = (*EAnon q*, *t*, *y*)› **by** (*simp add*: *Q T*)
    from *CptsComp*(*3*)[*OF this*] **have** ∗: ‹*map* (λ*a*. *case a of* (*p*, *s*, *uu*-) ⇒ (*unlift-prog p*, *s*)) ((*Q*, *T*) # *cs*) ∈ *cpts* (*ptran* Γ)› **.**
    show *?case*

apply(*simp add*: *case-prod-unfold*)
apply(*rule cpts.CptsComp*)
using *ptran Q* **apply**(*simp add*: *P*)
using $*$ **by** (*simp add*: *case-prod-unfold*)
qed
next
assume *a1*: ‹*cpt* ∈ *cpts* (*estran* Γ)›
assume *a2*: ‹*hd cpt* = (*EAnon p0*, *s0*, *x0*)›
show ‹*hd* (*map* (λ(*p*, *s*, -). (*unlift-prog p*, *s*)) *cpt*) = (*p0*, *s0*)›
by (*simp add*: *hd-map*[*OF cpts-nonnil*[*OF a1*]] *case-prod-unfold a2*)
qed

theorem *Anon-sound*:
assumes *h*: ‹Γ ⊢ *p* sat_p [*pre*, *rely*, *guar*, *post*]›
shows ‹Γ ⊨ *EAnon p* sat_e [*pre*, *rely*, *guar*, *post*]›
proof−
from *h* **have** Γ ⊨ *p* sat_p [*pre*, *rely*, *guar*, *post*] **using** *rgsound-p* **by** *blast*
then have ‹*validity* (*ptran* Γ) {*fin-com*} *p pre rely guar post*› **using** *prog-validity-def* **by** *simp*
then have *p-valid*[*rule-format*]: ‹∀ *S0*. *cpts-from* (*ptran* Γ) (*p*,*S0*) ∩ *assume pre rely* ⊆ *commit* (*ptran* Γ) {*fin-com*} *guar post*› **using** *validity-def* **by** *fast*

let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›
have ‹∀ *S0*. *cpts-from* (*estran* Γ) (*EAnon p*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*EAnon fin-com*} *?guar ?post*›
proof
fix *S0*
show ‹*cpts-from* (*estran* Γ) (*EAnon p*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*EAnon fin-com*} *?guar ?post*›
proof
fix *cpt*
assume *h1*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAnon p*, *S0*) ∩ *assume ?pre ?rely*›
from *h1* **have** *cpt*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAnon p*, *S0*)› **by** *blast*
then have ‹*cpt* ∈ *cpts* (*estran* Γ)› **by** *simp*
from *h1* **have** *cpt-assume*: ‹*cpt* ∈ *assume ?pre ?rely*› **by** *blast*
have *cpt-unlift*: ‹*unlift-cpt cpt* ∈ *cpts-from* (*ptran* Γ) (*p*, *fst S0*) ∩ *assume pre rely*›
proof
show ‹*unlift-cpt cpt* ∈ *cpts-from* (*ptran* Γ) (*p*, *fst S0*)›
using *unlift-cpt cpt surjective-pairing* **by** *metis*
next
from *cpt-assume* **have** ‹*snd* (*hd* (*map* (λ(*p*, *s*, -). (*unlift-prog p*, *s*)) *cpt*)) ∈ *pre*›
by (*auto simp add*: *assume-def hd-map*[*OF cpts-nonnil*[*OF* ‹*cpt* ∈ *cpts* (*estran* Γ)›]] *case-prod-unfold lift-state-set-def*)
then show ‹*unlift-cpt cpt* ∈ *assume pre rely*›

```
        using h1
        apply(auto simp add: assume-def case-prod-unfold)
        apply(erule-tac x=i in allE)
        apply(simp add: lift-state-pair-set-def case-prod-unfold)
          by (metis (mono-tags, lifting) Suc-lessD cpt cpts-from-anon′ fst-conv
unlift-prog.simps)
    qed
   with p-valid have unlift-commit: ‹unlift-cpt cpt ∈ commit (ptran Γ) {fin-com}
guar post› by blast
    show cpt ∈ commit (estran Γ) {EAnon fin-com} ?guar ?post
    proof(auto simp add: commit-def)
      fix i
      assume a1: ‹Suc i < length cpt›
      assume estran: ‹(cpt ! i, cpt ! Suc i) ∈ estran Γ›
      from cpts-from-anon′[OF cpt, rule-format, OF a1[THEN Suc-lessD]]
      obtain p1 s1 x1 where 1: ‹cpt!i = (EAnon p1,s1,x1)› by blast
      from cpts-from-anon′[OF cpt, rule-format, OF a1]
      obtain p2 s2 x2 where 2: ‹cpt!Suc i = (EAnon p2,s2,x2)› by blast
      from estran have ‹((p1,s1), (p2,s2)) ∈ ptran Γ›
        using 1 2 estran-anon-inv by fastforce
      then have ‹(unlift-conf (cpt!i), unlift-conf (cpt!Suc i)) ∈ ptran Γ›
        by (simp add: 1 2)
          then have ‹(fst (snd (cpt!i)), fst (snd (cpt!Suc i))) ∈ guar› using
unlift-commit
        apply(simp add: commit-def case-prod-unfold)
        apply clarify
        apply(erule allE[where x=i])
        using a1 by blast
      then show ‹(snd (cpt ! i), snd (cpt ! Suc i)) ∈ lift-state-pair-set guar›
        by (simp add: lift-state-pair-set-def case-prod-unfold)
    next
      assume a1: ‹fst (last cpt) = fin›
      from cpt cpts-nonnil have ‹cpt≠[]› by auto
      have ‹fst (last (map (λp. (unlift-prog (fst p), fst (snd p))) cpt)) = fin-com›
        by (simp add: last-map[OF ‹cpt≠[]›] a1)
       then have ‹snd (last (map (λp. (unlift-prog (fst p), fst (snd p))) cpt)) ∈
post› using unlift-commit
        by (simp add: commit-def case-prod-unfold)
      then show ‹snd (last cpt) ∈ lift-state-set post›
        by (simp add: last-map[OF ‹cpt≠[]›] lift-state-set-def case-prod-unfold)
    qed
  qed
 qed
 then have ‹validity (estran Γ) {EAnon fin-com} (EAnon p) ?pre ?rely ?guar
?post›
   by (subst validity-def, assumption)
 then show ?thesis
   by (subst es-validity-def, assumption)
qed
```

type-synonym *'a tran* = ‹*'a* × *'a*›

inductive-cases *estran-from-basic*: ‹Γ ⊢ (*EBasic ev*, *s*) −*es*[*a*]→ (*es*, *t*)›

lemma *assume-tl-comp*:
‹(*P*, *s*) # (*P*, *t*) # *cs* ∈ *assume pre rely* ⟹
stable pre rely ⟹
(*P*, *t*) # *cs* ∈ *assume pre rely*›
apply (*simp add*: *assume-def*)
apply *clarify*
apply(*rule conjI*)
apply(*erule-tac x=0* **in** *allE*)
apply(*simp add*: *stable-def*)
apply *auto*
done

lemma *assume-tl-env*:
assumes ‹(*P*,*s*)#(*Q*,*s*)#*cs* ∈ *assume pre rely*›
shows ‹(*Q*,*s*)#*cs* ∈ *assume pre rely*›
using *assms*
apply(*clarsimp simp add*: *assume-def*)
apply(*erule-tac x*=‹*Suc i*› **in** *allE*)
by *auto*

lemma *Basic-sound*:
assumes *h*: ‹Γ ⊢ *body* (*ev*::(*'l*,*'s*,*'prog*)*event*) sat_p [*pre* ∩ *guard ev*, *rely*, *guar*, *post*]›
and *stable*: ‹*stable pre rely*›
and *guar-refl*: ‹∀ *s*. (*s*, *s*) ∈ *guar*›
shows ‹Γ ⊨ *EBasic ev* sat_e [*pre*, *rely*, *guar*, *post*]›
proof−
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›

from *stable* **have** *stable'*: ‹*stable ?pre ?rely*›
by (*simp add*: *lift-state-set-def lift-state-pair-set-def stable-def*)

from *h Anon-sound* **have**
‹Γ ⊨ *EAnon* (*body ev*) sat_e [*pre* ∩ *guard ev*, *rely*, *guar*, *post*]› **by** *blast*
then have *es-valid*:
‹∀ *S0*. *cpts-from* (*estran* Γ) (*EAnon* (*body ev*), *S0*) ∩ *assume* (*lift-state-set* (*pre* ∩ *guard ev*)) *?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
using *es-validity-def* **by** (*simp*)

have ‹∀ *S0*. *cpts-from* (*estran* Γ) (*EBasic ev*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›

proof
fix *S0*
show ‹*cpts-from* (*estran* Γ) (*EBasic ev*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *cpt*
assume *cpt*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EBasic ev*, *S0*) ∩ *assume ?pre ?rely*›
then have *cpt-nonnil*: ‹*cpt* ≠ []› **using** *cpts-nonnil* **by** *auto*
then have *cpt-Cons*: *cpt* = *hd cpt* # *tl cpt* **using** *hd-Cons-tl* **by** *simp*
let *?c0* = *hd cpt*
from *cpt* **have** *fst-c0*: *fst* (*hd cpt*) = *EBasic ev* **by** *auto*
from *cpt* **have** *cpt1*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EBasic ev*, *S0*)› **by** *blast*
then have *cpt1-1*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **using** *cpts-from-def* **by** *blast*
from *cpt* **have** *cpt-assume*: ‹*cpt* ∈ *assume ?pre ?rely*› **by** *blast*

show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
using *cpt1-1 cpt*
proof(*induct arbitrary*:*S0*)
case (*CptsOne P S*)
then have ‹(*P*,*S*) = (*EBasic ev*, *S0*)› **by** *simp*
then show *?case* **by** (*simp add*: *commit-def*)
next
case (*CptsEnv P T cs S*)
from *CptsEnv*(*3*) **have** *P-s*:
‹(*P*,*S*) = (*EBasic ev*, *S0*)› **by** *simp*
from *CptsEnv*(*3*) **have**
‹(*P*, *S*) # (*P*, *T*) # *cs* ∈ *assume ?pre ?rely*› **by** *blast*
with *assume-tl-comp stable′* **have** *assume′*:
‹(*P*,*T*)#*cs* ∈ *assume ?pre ?rely*› **by** *fast*
have ‹(*P*, *T*) # *cs* ∈ *cpts-from* (*estran* Γ) (*EBasic ev*, *T*)› **using** *CptsEnv*(*1*) *P-s* **by** *simp*
with *assume′* **have** ‹(*P*, *T*) # *cs* ∈ *cpts-from* (*estran* Γ) (*EBasic ev*, *T*) ∩ *assume ?pre ?rely*› **by** *blast*
with *CptsEnv*(*2*) **have** ‹(*P*, *T*) # *cs* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **by** *blast*
then show *?case* **using** *commit-Cons-env-es* **by** *blast*
next
case (*CptsComp P S Q T cs*)
obtain *s0 x0* **where** *S0*: ‹*S0*=(*s0*,*x0*)› **by** *fastforce*
obtain *s x* **where** *S*: ‹*S*=(*s*,*x*)› **by** *fastforce*
obtain *t y* **where** *T*: ‹*T*=(*t*,*y*)› **by** *fastforce*
from *CptsComp*(*4*) **have** *P-s*:
‹(*P*,*S*) = (*EBasic ev*, *S0*)› **by** *simp*
from *CptsComp*(*4*) **have**
‹(*P*, *S*) # (*Q*, *T*) # *cs* ∈ *assume ?pre ?rely*› **by** *blast*
then have *pre*:
‹*snd* (*hd* ((*P*,*S*)#(*Q*,*T*)#*cs*)) ∈ *?pre*›
and *rely*:

⟨∀ *i*. *Suc i* < *length* ((*P*,*S*)#(*Q*,*T*)#*cs*) ⟶
(((*P*,*S*)#(*Q*,*T*)#*cs*)!*i* −*e*→ ((*P*,*S*)#(*Q*,*T*)#*cs*)!(*Suc i*)) ⟶
(*snd* (((*P*,*S*)#(*Q*,*T*)#*cs*)!*i*), *snd* (((*P*,*S*)#(*Q*,*T*)#*cs*)!*Suc i*)) ∈ *?rely*⟩
using *assume-def* **by** *blast*+

from *pre* **have** ⟨*S* ∈ *?pre*⟩ **by** *simp*
then have ⟨*s*∈*pre*⟩ **by** (*simp add*: *lift-state-set-def S*)
from *CptsComp*(*1*) **have** ⟨∃ *a k*. Γ ⊢ (*P*,*S*) −*es*[*a*♯*k*]→ (*Q*,*T*)⟩
apply(*simp add*: *estran-def*)
apply(*erule exE*) **apply**(*rule-tac x*=⟨*Act a*⟩ **in** *exI*) **apply**(*rule-tac x*=⟨*K a*⟩ **in** *exI*)
apply(*subst*(*asm*) *actk-destruct*) **by** *assumption*
then obtain *a k* **where** ⟨Γ ⊢ (*P*,*S*) −*es*[*a*♯*k*]→ (*Q*,*T*)⟩ **by** *blast*
with *P-s* **have** *tran*: ⟨Γ ⊢ (*EBasic ev*, *S0*) −*es*[*a*♯*k*]→ (*Q*,*T*)⟩ **by** *simp*
then have *a*: ⟨*a* = *EvtEnt ev*⟩ **apply**− **apply**(*erule estran-from-basic*) **apply** *simp* **done**
from *tran* **have** *guard*: ⟨*s0* ∈ *guard ev*⟩ **apply**− **apply**(*erule estran-from-basic*) **apply** (*simp add*: *S0*) **done**
from *tran* **have** *s0*=*t* **apply**− **apply**(*erule estran-from-basic*) **using** *a guard* **apply** (*simp add*: *T S0*) **done**
with *P-s S S0* **have** *s*=*t* **by** *simp*
with *guar-refl* **have** *guar*: ⟨(*s*, *t*) ∈ *guar*⟩ **by** *simp*

have ⟨(*Q*,*T*)#*cs* ∈ *cpts-from* (*estran* Γ) (*EAnon* (*body ev*), *T*)⟩
proof−
have (*Q*,*T*)#*cs* ∈ *cpts* (*estran* Γ) **by** (*rule CptsComp*(*2*))
moreover have *Q* = *EAnon* (*body ev*) **using** *estran-from-basic* **using** *tran* **by** *blast*
ultimately show *?thesis* **by** *auto*
qed
moreover have ⟨(*Q*,*T*)#*cs* ∈ *assume* (*lift-state-set* (*pre* ∩ *guard ev*)) *?rely*⟩
proof−
have ⟨*fst* (*snd* (*hd* ((*Q*,*T*)#*cs*))) ∈ (*pre* ∩ *guard ev*)⟩
proof
show ⟨*fst* (*snd* (*hd* ((*Q*, *T*) # *cs*))) ∈ *pre*⟩ **using** ⟨*s*=*t*⟩ ⟨*s*∈*pre*⟩ *T* **by** *simp*
next
show ⟨*fst* (*snd* (*hd* ((*Q*, *T*) # *cs*))) ∈ *guard ev*⟩ **using** ⟨*s0*=*t*⟩ *guard T* **by** *fastforce*
qed
then have ⟨*snd* (*hd* ((*Q*,*T*)#*cs*)) ∈ *lift-state-set* (*pre* ∩ *guard ev*)⟩ **using** *lift-state-set-def* **by** *fastforce*
moreover have
⟨∀ *i*. *Suc i* < *length* ((*Q*,*T*)#*cs*) ⟶ (((*Q*,*T*)#*cs*)!*i* −*e*→ ((*Q*,*T*)#*cs*)!(*Suc i*)) ⟶ (*snd* (((*Q*,*T*)#*cs*)!*i*), *snd* (((*Q*,*T*)#*cs*)!*Suc i*)) ∈ *?rely*⟩
using *rely* **by** *auto*
ultimately show *?thesis* **using** *assume-def* **by** *blast*
qed
ultimately have ⟨(*Q*,*T*)#*cs* ∈ *cpts-from* (*estran* Γ) (*EAnon* (*body ev*), *T*)

∩ *assume* (*lift-state-set* (*pre* ∩ *guard ev*)) *?rely*› **by** *blast*
then have ‹(Q,T)#*cs* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **using** *es-valid* **by** *blast*
then show *?case* **using** *commit-Cons-comp CptsComp*(*1*) *guar S T lift-state-set-def lift-state-pair-set-def* **by** *fast*
qed
qed
qed
then show *?thesis* **by** *simp*
qed

inductive-cases *estran-from-atom*: ‹Γ ⊢ (*EAtom ev*, *s*) −*es*[*a*]→ (Q, *t*)›

lemma *estran-from-atom′*:
assumes *h*: ‹Γ ⊢ (*EAtom ev*, *s*,*x*) −*es*[*a*♯*k*]→ (Q, *t*,*y*)›
shows ‹*a* = *AtomEvt ev* ∧ *s* ∈ *guard ev* ∧ Γ ⊢ (*body ev*, *s*) −*c*∗→ (*fin-com*, *t*) ∧ Q = *EAnon fin-com*›
using *h estran-from-atom* **by** *blast*

lemma *last-sat-post*:
assumes *t*: ‹*t* ∈ *post*›
and *cpt*: *cpt* = (Q,*t*)#*cs*
and *etran*: ‹∀ *i*. *Suc i* < *length cpt* ⟶ *cpt*!*i* −*e*→ *cpt*!*Suc i*›
and *stable*: ‹*stable post rely*›
and *rely*: ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt*!*i* −*e*→ *cpt*!*Suc i*) ⟶ (*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *rely*›
shows ‹*snd* (*last cpt*) ∈ *post*›
proof−
from *etran rely* **have** *rely′*:
‹∀ *i*. *Suc i* < *length cpt* ⟶ (*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *rely*› **by** *auto*
show *?thesis* **using** *cpt rely′*
proof(*induct cs arbitrary*:*cpt rule*:*rev-induct*)
case *Nil*
then show *?case* **using** *t* **by** *simp*
next
case (*snoc x xs*)
have
‹∀ *i*. *Suc i* < *length* ((Q,*t*)#*xs*) ⟶ (*snd* (((Q,*t*)#*xs*) ! *i*), *snd* (((Q,*t*)#*xs*) ! *Suc i*)) ∈ *rely*›
proof
fix *i*
show ‹*Suc i* < *length* ((Q,*t*)#*xs*) ⟶ (*snd* (((Q,*t*)#*xs*) ! *i*), *snd* (((Q,*t*)#*xs*) ! *Suc i*)) ∈ *rely*›
proof
assume *Suc-i-lt*: ‹*Suc i* < *length* ((Q,*t*)#*xs*)›
then have *eq1*:
((Q,*t*)#*xs*)!*i* = *cpt*!*i* **using** *snoc*(*2*)
by (*metis Suc-lessD butlast.simps*(*2*) *nth-butlast snoc-eq-iff-butlast*)
from *Suc-i-lt snoc*(*2*) **have** *eq2*:

((*Q*,*t*)#*xs*)!*Suc i* = *cpt*!*Suc i*
by (*simp add*: *nth-append*)
have ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *rely*›
using *Suc-i-lt snoc.prems*(*1*) *snoc.prems*(*2*) **by** *auto*
then show ‹(*snd* (((*Q*,*t*)#*xs*) ! *i*), *snd* (((*Q*,*t*)#*xs*) ! *Suc i*)) ∈ *rely*› **using** *eq1 eq2* **by** *simp*
qed
qed
then have *last-post*: ‹*snd* (*last* ((*Q*, *t*) # *xs*)) ∈ *post*›
using *snoc.hyps* **by** *blast*
have ‹(*snd* (*last* ((*Q*,*t*)#*xs*)), *snd x*) ∈ *rely*› **using** *snoc*(*2*,*3*)
by (*metis List.nth-tl append-butlast-last-id append-is-Nil-conv butlast.simps*(*2*) *butlast-snoc length-Cons length-append-singleton lessI list.distinct*(*1*) *list.sel*(*3*) *nth-append-length nth-butlast*)
with *last-post stable*
have *snd x* ∈ *post* **by** (*simp add*: *stable-def*)
then show *?case* **using** *snoc*(*2*) **by** *simp*
qed
qed

lemma *Atom-sound*:
assumes *h*: ‹∀ *V*. Γ ⊢ *body* (*ev*::(*'l*,*'s*,*'prog*)*event*) sat_p [*pre* ∩ *guard ev* ∩ {*V*}, *Id*, *UNIV*, {*s*. (*V*,*s*)∈*guar*} ∩ *post*]›
and *stable-pre*: ‹*stable pre rely*›
and *stable-post*: ‹*stable post rely*›
shows ‹Γ ⊨ *EAtom ev* sat_e [*pre*, *rely*, *guar*, *post*]›
proof−
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›

from *stable-pre* **have** *stable-pre'*: ‹*stable ?pre ?rely*›
by (*simp add*: *lift-state-set-def lift-state-pair-set-def stable-def*)
from *stable-post* **have** *stable-post'*: ‹*stable ?post ?rely*›
by (*simp add*: *lift-state-set-def lift-state-pair-set-def stable-def*)

from *h rgsound-p* **have**
‹∀ *V*. Γ ⊨ (*body ev*) sat_p [*pre* ∩ *guard ev* ∩ {*V*}, *Id*, *UNIV*, {*s*. (*V*,*s*)∈*guar*} ∩ *post*]› **by** *blast*
then have *body-valid*:
‹∀ *V s0*. *cpts-from* (*ptran* Γ) ((*body ev*), *s0*) ∩ *assume* (*pre* ∩ *guard ev* ∩ {*V*}) *Id* ⊆ *commit* (*ptran* Γ) {*fin-com*} *UNIV* ({*s*. (*V*,*s*)∈*guar*} ∩ *post*)›
using *prog-validity-def* **by** (*meson validity-def*)

have ‹∀ *s0*. *cpts-from* (*estran* Γ) (*EAtom ev*, *s0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *S0*

show ‹*cpts-from* (*estran* Γ) (*EAtom ev*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *cpt*
assume *cpt*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAtom ev*, *S0*) ∩ *assume ?pre ?rely*›
then have *cpt1*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EAtom ev*, *S0*)› **by** *blast*
then have *cpt1-1*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **by** *simp*
from *cpt1* **have** *hd cpt* = (*EAtom ev*, *S0*) **by** *fastforce*
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
using *cpt1-1 cpt*
proof(*induct arbitrary*:*S0*)
case (*CptsOne P S*)
then show *?case* **by** (*simp add*: *commit-def*)
next
case (*CptsEnv P T cs S*)
have ‹(*P*, *T*) # *cs* ∈ *cpts-from* (*estran* Γ) (*EAtom ev*, *T*) ∩ *assume ?pre ?rely*›
proof
from *CptsEnv*(*3*) **have** ‹(*P*, *S*) # (*P*, *T*) # *cs* ∈ *cpts-from* (*estran* Γ) (*EAtom ev*, *S0*)› **by** *blast*
then show ‹(*P*, *T*) # *cs* ∈ *cpts-from* (*estran* Γ) (*EAtom ev*, *T*)›
using *CptsEnv.hyps*(*1*) **by** *auto*
next
from *CptsEnv*(*3*) **have** ‹(*P*, *S*) # (*P*, *T*) # *cs* ∈ *assume ?pre ?rely*› **by** *blast*
with *assume-tl-comp stable-pre′* **show** ‹(*P*, *T*) # *cs* ∈ *assume ?pre ?rely*› **by** *fast*
qed
then have ‹(*P*, *T*) # *cs* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **using** *CptsEnv*(*2*) **by** *blast*
then show *?case* **using** *commit-Cons-env-es* **by** *blast*
next
case (*CptsComp P S Q T cs*)
obtain *s0 x0* **where** *S0*: ‹*S0*=(*s0*,*x0*)› **by** *fastforce*
obtain *s x* **where** *S*: ‹*S*=(*s*,*x*)› **by** *fastforce*
obtain *t y* **where** *T*: ‹*T*=(*t*,*y*)› **by** *fastforce*
from *CptsComp*(*1*) **have** ‹∃ *a k*. Γ ⊢ (*P*,*S*) −*es*[*a*♯*k*]→ (*Q*,*T*)›
apply− **apply**(*simp add*: *estran-def*) **apply**(*erule exE*) **apply**(*rule-tac x*=‹*Act a*› **in** *exI*) **apply**(*rule-tac x*=‹*K a*› **in** *exI*)
apply(*subst* (*asm*) *actk-destruct*) **by** *assumption*
then obtain *a k* **where** Γ ⊢ (*P*,*S*) −*es*[*a*♯*k*]→ (*Q*,*T*) **by** *blast*
moreover from *CptsComp*(*4*) **have** *P-s*: (*P*,*S*) = (*EAtom ev*, *S0*) **by** *force*
ultimately have *tran*: ‹Γ ⊢ (*EAtom ev*, *S0*) −*es*[*a*♯*k*]→ (*Q*,*T*)› **by** *simp*
then have *tran-inv*:
a = *AtomEvt ev* ∧ *s0* ∈ *guard ev* ∧ Γ ⊢ (*body ev*, *s0*) −*c*∗→ (*fin-com*, *t*) ∧ *Q* = *EAnon fin-com*
using *estran-from-atom′ S0 T* **by** *fastforce*
from *tran-inv* **have** *Q*: ‹*Q* = *EAnon fin-com*› **by** *blast*

from *CptsComp(4)* **have** *assume*: ‹(*P*, *S*) # (*Q*, *T*) # *cs* ∈ *assume ?pre ?rely*› **by** *blast*
from *assume* **have** *assume1*: ‹*snd* (*hd* ((*P*,*S*)#(*Q*,*T*)#*cs*)) ∈ *?pre*› **using** *assume-def* **by** *blast*
then have ‹*S* ∈ *?pre*› **by** *simp*
then have ‹*s*∈*pre*› **by** (*simp add*: *lift-state-set-def S*)
then have ‹*s0* ∈ *pre*› **using** *P-s S0 S* **by** *simp*
have ‹*s0* ∈ *guard ev*› **using** *tran-inv* **by** *blast*
have ‹*S0* ∈ {*S0*}› **by** *simp*

from *assume* **have** *assume2*:
‹∀ *i*. *Suc i* < *length* ((*P*,*S*)#(*Q*,*T*)#*cs*) ⟶ (((*P*,*S*)#(*Q*,*T*)#*cs*)!*i* −*e*→ ((*P*,*S*)#(*Q*,*T*)#*cs*)!(*Suc i*)) ⟶ (*snd* (((*P*,*S*)#(*Q*,*T*)#*cs*)!*i*), *snd* (((*P*,*S*)#(*Q*,*T*)#*cs*)!*Suc i*)) ∈ *?rely*›
using *assume-def* **by** *blast*
then have *assume2-tl*:
‹∀ *i*. *Suc i* < *length* ((*Q*,*T*)#*cs*) ⟶ ((((*Q*,*T*)#*cs*)!*i* −*e*→ ((*Q*,*T*)#*cs*)!(*Suc i*)) ⟶ (*snd* ((((*Q*,*T*)#*cs*)!*i*), *snd* ((((*Q*,*T*)#*cs*)!*Suc i*)) ∈ *?rely*›
by *fastforce*
from *tran-inv* **have** ‹Γ ⊢ (*body ev*, *s0*) −*c*∗→ (*fin-com*, *t*)› **by** *blast*
with *cpt-from-ptran-star* **obtain** *pcpt* **where** *pcpt*:
‹*pcpt* ∈ *cpts-from* (*ptran* Γ) (*body ev*, *s0*) ∩ *assume* {*s0*} {} ∧ *last pcpt* = (*fin-com*, *t*)› **by** *blast*

from *pcpt* **have**
‹*pcpt* ∈ *assume* {*s0*} {}› **by** *blast*
with ‹*s0*∈*pre*› ‹*s0*∈*guard ev*› **have** ‹*pcpt* ∈ *assume* (*pre* ∩ *guard ev* ∩ {*s0*}) *Id*›
by (*simp add*: *assume-def*)
with *pcpt body-valid* **have** *pcpt-commit*:
‹*pcpt* ∈ *commit* (*ptran* Γ) {*fin-com*} *UNIV* ({*s*. (*s0*, *s*) ∈ *guar*} ∩ *post*)›
by *blast*
then have ‹*t* ∈ ({*s*. (*s0*, *s*) ∈ *guar*} ∩ *post*)›
by (*simp add*: *pcpt commit-def*)
with *P-s S0 S T* **have** ‹(*s*,*t*)∈*guar*› **by** *simp*
from *pcpt-commit* **have**
‹*fst* (*last pcpt*) = *fin-com* ⟶ *snd* (*last pcpt*) ∈ ({*s*. (*s0*, *s*) ∈ *guar*} ∩ *post*)›
by (*simp add*: *commit-def*)
with *pcpt* **have** *t*:
‹*t* ∈ ({*s*. (*s0*, *s*) ∈ *guar*} ∩ *post*)› **by** *force*

have *rest-etran*:
‹∀ *i*. *Suc i* < *length* ((*Q*,*T*)#*cs*) ⟶ ((*Q*,*T*)#*cs*)!*i* −*e*→ ((*Q*,*T*)#*cs*)!*Suc i*› **using** *all-etran-from-fin*
using *CptsComp.hyps(2) Q* **by** *blast*
from *rest-etran assume2-tl* **have** *rely*:
‹∀ *i*. *Suc i* < *length* ((*Q*,*T*)#*cs*) ⟶ (*snd* (((*Q*, *T*) # *cs*) ! *i*), *snd* (((*Q*,

```
T) # cs) ! Suc i)) ∈ ?rely⟩
     by blast
    have commit1:
       ⟨∀ i. Suc i < length ((P,S)#(Q,T)#cs) ⟶ (((P,S)#(Q,T)#cs)!i,
((P,S)#(Q,T)#cs)!(Suc i)) ∈ (estran Γ) ⟶ (snd (((P,S)#(Q,T)#cs)!i), snd
(((P,S)#(Q,T)#cs)!(Suc i))) ∈ ?guar⟩
    proof
     fix i
      show ⟨Suc i < length ((P,S)#(Q,T)#cs) ⟶ (((P,S)#(Q,T)#cs)!i,
((P,S)#(Q,T)#cs)!(Suc i)) ∈ (estran Γ) ⟶ (snd (((P,S)#(Q,T)#cs)!i), snd
(((P,S)#(Q,T)#cs)!(Suc i))) ∈ ?guar⟩
     proof
      assume ⟨Suc i < length ((P, S) # (Q, T) # cs)⟩
      show ⟨(((P, S) # (Q, T) # cs) ! i, ((P, S) # (Q, T) # cs) ! Suc i) ∈
(estran Γ) ⟶
  (snd (((P, S) # (Q, T) # cs) ! i), snd (((P, S) # (Q, T) # cs) ! Suc i)) ∈
?guar⟩
      proof(cases i)
       case 0
      then show ?thesis apply simp using ⟨(s,t)∈guar⟩ lift-state-pair-set-def
S T by blast
      next
       case (Suc i')
       then show ?thesis apply simp apply(subst Q)
        using no-ctran-from-fin
        using CptsComp.hyps(2) Q ⟨Suc i < length ((P, S) # (Q, T) # cs)⟩
        by (metis Suc-less-eq length-Cons nth-Cons-Suc)
      qed
     qed
    qed
    have commit2-aux:
     ⟨fst (last ((Q,T)#cs)) = fin ⟶ snd (last ((Q,T)#cs)) ∈ ?post⟩
    proof
     assume ⟨fst (last ((Q, T) # cs)) = fin⟩
     from t have 1: ⟨T∈?post⟩ using T by (simp add: lift-state-set-def)
     from last-sat-post[OF 1 refl rest-etran stable-post'] rely
     show ⟨snd (last ((Q, T) # cs)) ∈ ?post⟩ by blast
    qed
    then have commit2:
     ⟨fst (last ((P,S)#(Q,T)#cs)) = fin ⟶ snd (last ((P,S)#(Q,T)#cs)) ∈
?post⟩ by simp
    show ?case using commit1 commit2
     by (simp add: commit-def)
   qed
  qed
 qed
 then show ?thesis
  by (simp)
qed
```

theorem *conseq-sound*:
assumes *h*: ‹$\Gamma \models$ *es* sat_e [*pre'*, *rely'*, *guar'*, *post'*]›
and *pre*: *pre* $\subseteq$ *pre'*
and *rely*: *rely* $\subseteq$ *rely'*
and *guar*: *guar'* $\subseteq$ *guar*
and *post*: *post'* $\subseteq$ *post*
shows ‹$\Gamma \models$ *es* sat_e [*pre*, *rely*, *guar*, *post*]›
proof−
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›
let *?pre'* = ‹*lift-state-set pre'*›
let *?rely'* = ‹*lift-state-pair-set rely'*›
let *?guar'* = ‹*lift-state-pair-set guar'*›
let *?post'* = ‹*lift-state-set post'*›

from *h* **have**
valid: ‹$\forall$ *S0*. *cpts-from* (*estran* Γ) (*es*, *S0*) $\cap$ *assume ?pre' ?rely'* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar' ?post'*›
by *auto*
have ‹$\forall$ *S0*. *cpts-from* (*estran* Γ) (*es*, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *S0*
show ‹*cpts-from* (*estran* Γ) (*es*, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *cpt*
assume *cpt*: ‹*cpt* $\in$ *cpts-from* (*estran* Γ) (*es*, *S0*) $\cap$ *assume ?pre ?rely*›
then have *cpt1*: ‹*cpt* $\in$ *cpts-from* (*estran* Γ) (*es*, *S0*)› **by** *blast*
from *cpt* **have** *assume*: ‹*cpt* $\in$ *assume ?pre ?rely*› **by** *blast*
then have *assume'*: ‹*cpt* $\in$ *assume ?pre' ?rely'*›
apply(*simp add*: *assume-def lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
using *pre rely* **by** *auto*
from *cpt1 assume'* **have** ‹*cpt* $\in$ *cpts-from* (*estran* Γ) (*es*, *S0*) $\cap$ *assume ?pre' ?rely'*› **by** *blast*
with *valid* **have** *commit*: *cpt* $\in$ *commit* (*estran* Γ) {*fin*} *?guar' ?post'* **by** *blast*
then show ‹*cpt* $\in$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
apply(*simp add*: *commit-def lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
using *guar post* **by** *auto*
qed
qed
then have ‹*validity* (*estran* Γ) {*fin*} *es ?pre ?rely ?guar ?post*› **using** *validity-def* **by** *metis*
then show *?thesis* **using** *es-validity-def* **by** *simp*
qed

primrec (*nonexhaustive*) *unlift-seq* **where**
⟨*unlift-seq* (*ESeq P Q*) = *P*⟩

primrec *unlift-seq-esconf* **where**
⟨*unlift-seq-esconf* (*P*,*s*) = (*unlift-seq P*, *s*)⟩

abbreviation ⟨*unlift-seq-cpt* ≡ *map unlift-seq-esconf*⟩

lemma *split-seq*:
assumes *cpt*: ⟨*cpt* ∈ *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*)⟩
and *not-all-seq*: ⟨¬ *all-seq es2 cpt*⟩
shows
∃ *i S'*. *cpt*!*Suc i* = (*es2*, *S'*) ∧
Suc i < *length cpt* ∧
all-seq es2 (*take* (*Suc i*) *cpt*) ∧
unlift-seq-cpt (*take* (*Suc i*) *cpt*) @ [(*fin*,*S'*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∧
(*cpt*!*i*, *cpt*!*Suc i*)∈*estran* Γ ∧
(*unlift-seq-esconf* (*cpt*!*i*), (*fin*,*S'*))∈*estran* Γ
proof−
from *cpt* **have** *hd-cpt*: ⟨*hd cpt* = (*ESeq es1 es2*, *S0*)⟩ **by** *simp*
from *cpt* **have** ⟨*cpt* ∈ *cpts* (*estran* Γ)⟩ **by** *simp*
then have ⟨*cpt* ∈ *cpts-es-mod* Γ⟩ **using** *cpts-es-mod-equiv* **by** *blast*
then show *?thesis* **using** *hd-cpt not-all-seq*
proof(*induct arbitrary*:*S0 es1*)
case (*CptsModOne*)
then show *?case*
by (*simp add*: *all-seq-def*)
next
case (*CptsModEnv P t y cs s x*)
from *CptsModEnv*(*3*) **have** *1*: ⟨*hd* ((*P*,*t*,*y*)#*cs*) = (*es1 NEXT es2*, *t*,*y*)⟩ **by** *simp*
from *CptsModEnv*(*4*) **have** *2*: ⟨¬ *all-seq es2* ((*P*,*t*,*y*)#*cs*)⟩ **by** (*simp add*: *all-seq-def*)
from *CptsModEnv*(*2*)[*OF 1 2*] **obtain** *i S'* **where**
⟨((*P*, *t*, *y*) # *cs*) ! *Suc i* = (*es2*, *S'*) ∧
Suc i < *length* ((*P*, *t*, *y*) # *cs*) ∧
all-seq es2 (*take* (*Suc i*) ((*P*, *t*, *y*) # *cs*)) ∧
map unlift-seq-esconf (*take* (*Suc i*) ((*P*, *t*, *y*) # *cs*)) @ [(*fin*, *S'*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *t*, *y*) ∧ (((*P*, *t*, *y*) # *cs*) ! *i*, ((*P*, *t*, *y*) # *cs*) ! *Suc i*) ∈ *estran* Γ ∧ (*unlift-seq-esconf* (((*P*, *t*, *y*) # *cs*) ! *i*), *fin*, *S'*) ∈ *estran* Γ⟩
by *blast*
then show *?case* **apply**−
apply(*rule exI*[**where** *x*=*Suc i*])
apply (*simp add*: *all-seq-def*)
apply(*rule conjI*)
apply(*rule CptsEnv*)
apply *fastforce*

apply(*rule conjI*)
using *CptsModEnv*(*3*) **apply** *simp*
by *argo*
next
case (*CptsModAnon*)
then show *?case* **by** *simp*
next
case (*CptsModAnon-fin*)
then show *?case* **by** *simp*
next
case (*CptsModBasic*)
then show *?case* **by** *simp*
next
case (*CptsModAtom*)
then show *?case* **by** *simp*
next
case (*CptsModSeq P s x a Q t y R cs*)
from *CptsModSeq*(*5*) **have** ‹(*s*,*x*) = *S0*› **and** ‹*R*=*es2*› **and** ‹*P*=*es1*› **by** *simp*+
from *CptsModSeq*(*5*) **have** *1*: ‹*hd* ((*Q NEXT R*, *t*,*y*) # *cs*) = (*Q NEXT es2*, *t*,*y*)› **by** *simp*
from *CptsModSeq*(*6*) **have** *2*: ‹¬ *all-seq es2* ((*Q NEXT R*, *t*,*y*) # *cs*)› **by** (*simp add*: *all-seq-def*)
from *CptsModSeq*(*4*)[*OF 1 2*] **obtain** *i S*′ **where**
‹((*Q NEXT R*, *t*, *y*) # *cs*) ! *Suc i* = (*es2*, *S*′) ∧
Suc i < *length* ((*Q NEXT R*, *t*, *y*) # *cs*) ∧
all-seq es2 (*take* (*Suc i*) ((*Q NEXT R*, *t*, *y*) # *cs*)) ∧
map unlift-seq-esconf (*take* (*Suc i*) ((*Q NEXT R*, *t*, *y*) # *cs*)) @ [(*fin*, *S*′)] ∈ *cpts-from* (*estran* Γ) (*Q*, *t*, *y*) ∧
(((*Q NEXT R*, *t*, *y*) # *cs*) ! *i*, ((*Q NEXT R*, *t*, *y*) # *cs*) ! *Suc i*) ∈ *estran* Γ ∧
(*unlift-seq-esconf* (((*Q NEXT R*, *t*, *y*) # *cs*) ! *i*), *fin*, *S*′) ∈ *estran* Γ›
by *blast*
then show *?case* **apply**−
apply(*rule exI*[**where** *x*=*Suc i*])
apply(*simp add*: *all-seq-def*)
apply(*rule conjI*)
apply(*rule CptsComp*)
apply(*simp add*: *estran-def*; *rule exI*)
apply(*rule CptsModSeq*(*1*))
apply *fast*
apply(*rule conjI*)
apply(*rule* ‹*P*=*es1*›)
apply(*rule conjI*)
apply(*rule* ‹(*s*,*x*) = *S0*›)
by *argo*
next
case (*CptsModSeq-fin Q s x a t y cs cs*′)
then show *?case*
apply−

```
      apply(rule exI[where x=0])
      apply (simp add: all-seq-def)
      apply(rule conjI)
       apply(rule CptsComp)
        apply(simp add: estran-def; rule exI; assumption)
       apply(rule CptsOne)
      apply(rule conjI)
       apply(simp add: estran-def; rule exI)
      using ESeq-fin apply blast
      apply(simp add: estran-def)
      apply(rule exI)
      by assumption
  next
    case (CptsModChc1)
    then show ?case by simp
  next
    case (CptsModChc2)
    then show ?case by simp
  next
    case (CptsModJoin1)
    then show ?case by simp
  next
    case (CptsModJoin2)
    then show ?case by simp
  next
    case (CptsModJoin-fin)
    then show ?case by simp
  next
    case (CptsModWhileTOnePartial)
    then show ?case by simp
  next
    case (CptsModWhileTOneFull)
    then show ?case by simp
  next
    case (CptsModWhileTMore)
    then show ?case by simp
  next
    case (CptsModWhileF)
    then show ?case by simp
  qed
qed

lemma all-seq-unlift:
  assumes all-seq: all-seq Q cpt
    and h: ‹cpt ∈ cpts-from (estran Γ) (ESeq P Q, S0) ∩ assume pre rely›
  shows ‹unlift-seq-cpt cpt ∈ cpts-from (estran Γ) (P, S0) ∩ assume pre rely›
proof
  from h have h1:
    ‹cpt ∈ cpts-from (estran Γ) (ESeq P Q, S0)› by blast
```

then have *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **by** *simp*
with *cpts-es-mod-equiv* **have** *cpt-mod*: *cpt* ∈ *cpts-es-mod* Γ **by** *auto*
from *h1* **have** *hd-cpt*: ‹*hd cpt* = (*ESeq P Q*, *S0*)› **by** *simp*
show ‹*map unlift-seq-esconf cpt* ∈ *cpts-from* (*estran* Γ) (*P*, *S0*)› **using** *cpt-mod hd-cpt all-seq*
proof(*induct arbitrary*:*P S0*)
case (*CptsModOne P s*)
then show *?case* **apply** *simp* **apply**(*rule CptsOne*) **done**
next
case (*CptsModEnv P1 t y cs s x*)
from *CptsModEnv*(*3*) **have** ‹*hd* ((*P1*, *t*,*y*) # *cs*) = (*P NEXT Q*, *t*,*y*)› **by** *simp*
moreover from *CptsModEnv*(*4*) **have** ‹*all-seq Q* ((*P1*, *t*,*y*) # *cs*)›
apply− **apply**(*unfold all-seq-def*) **apply** *auto* **done**
ultimately have ‹*map unlift-seq-esconf* ((*P1*, *t*,*y*) # *cs*) ∈ *cpts-from* (*estran* Γ) (*P*, *t*,*y*)›
using *CptsModEnv*(*2*) **by** *blast*
moreover have (*s*,*x*)=*S0* **using** *CptsModEnv*(*3*) **by** *simp*
ultimately show *?case* **apply** *clarsimp* **apply**(*erule CptsEnv*) **done**
next
case (*CptsModAnon*)
then show *?case* **by** *simp*
next
case (*CptsModAnon-fin*)
then show *?case* **by** *simp*
next
case (*CptsModBasic*)
then show *?case* **by** *simp*
next
case (*CptsModAtom*)
then show *?case* **by** *simp*
next
case (*CptsModSeq P1 s x a Q1 t y R cs*)
from *CptsModSeq*(*5*) **have** ‹*hd* ((*Q1 NEXT R*, *t*,*y*) # *cs*) = (*Q1 NEXT Q*, *t*,*y*)› **by** *simp*
moreover from *CptsModSeq*(*6*) **have** ‹*all-seq Q* ((*Q1 NEXT R*, *t*,*y*) # *cs*)›
apply(*unfold all-seq-def*) **by** *auto*
ultimately have ‹*map unlift-seq-esconf* ((*Q1 NEXT R*, *t*,*y*) # *cs*) ∈ *cpts-from* (*estran* Γ) (*Q1*, *t*,*y*)›
using *CptsModSeq*(*4*) **by** *blast*
moreover from *CptsModSeq*(*5*) **have** (*s*,*x*)=*S0* **and** *P1*=*P* **by** *simp-all*
ultimately show *?case* **apply** (*simp add*: *estran-def*)
apply(*rule CptsComp*) **using** *CptsModSeq*(*1*) **by** *auto*
next
case (*CptsModSeq-fin*)
from *CptsModSeq-fin*(*5*) **have** *False*
apply(*auto simp add*: *all-seq-def*)
using *seq-neq2* **by** *metis*
then show *?case* **by** *blast*

```
  next
    case (CptsModChc1)
    then show ?case by simp
  next
    case (CptsModChc2)
    then show ?case by simp
  next
    case (CptsModJoin1)
    then show ?case by simp
  next
    case (CptsModJoin2)
    then show ?case by simp
  next
    case (CptsModJoin-fin)
    then show ?case by simp
  next
    case CptsModWhileTOnePartial
    then show ?case by simp
  next
    case CptsModWhileTOneFull
    then show ?case by simp
  next
    case CptsModWhileTMore
    then show ?case by simp
  next
    case CptsModWhileF
    then show ?case by simp
  qed
next
  from h have h2: cpt ∈ assume pre rely by blast
  then have a1: ‹snd (hd cpt) ∈ pre› by (simp add: assume-def)
  from h2 have a2:
    ‹∀ i. Suc i < length cpt ⟶
        fst ( (cpt ! i)) = fst ( (cpt ! Suc i)) ⟶
        (snd ( (cpt ! i)), snd ( (cpt ! Suc i))) ∈ rely› by (simp add: assume-def)
  from h have ‹cpt ∈ cpts (estran Γ)› by fastforce
  with cpts-nonnil have cpt-nonnil: cpt ≠ [] by blast
  show ‹map unlift-seq-esconf cpt ∈ assume pre rely›
    apply (simp add: assume-def)
  proof
    show ‹snd (hd (map unlift-seq-esconf cpt)) ∈ pre› using a1 cpt-nonnil
      by (metis eq-snd-iff hd-map unlift-seq-esconf.simps)
  next
    show ‹∀ i. Suc i < length cpt ⟶
        fst (unlift-seq-esconf (cpt ! i)) = fst (unlift-seq-esconf (cpt ! Suc i)) ⟶
         (snd (unlift-seq-esconf (cpt ! i)), snd (unlift-seq-esconf (cpt ! Suc i))) ∈
rely›
     using a2 by (metis Suc-lessD all-seq all-seq-def fst-conv nth-mem prod.collapse
snd-conv unlift-seq.simps unlift-seq-esconf.simps)
```

  qed
qed

lemma *cpts-from-assume-snoc-fin*:
  assumes *cpt*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P*, *S0*) ∩ *assume pre rely*›
    and *tran*: ‹(*last cpt*, (*fin*, *S1*)) ∈ (*estran* Γ)›
  shows ‹*cpt* @ [(*fin*, *S1*)] ∈ *cpts-from* (*estran* Γ) (*P*, *S0*) ∩ *assume pre rely*›
proof
  from *cpt* **have** *cpt-from*:
    ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P*,*S0*)› **by** *blast*
   with *cpts-snoc-comp tran cpts-from-def* **show** ‹*cpt* @ [(*fin*, *S1*)] ∈ *cpts-from* (*estran* Γ) (*P*, *S0*)›
    using *cpts-nonnil* **by** *fastforce*
next
  from *cpt* **have** *cpt-assume*:
    ‹*cpt* ∈ *assume pre rely*› **by** *blast*
  from *cpt* **have** *cpt-nonnil*:
    ‹*cpt* ≠ []› **using** *cpts-nonnil* **by** *fastforce*
  from *tran ctran-imp-not-etran* **have** *not-etran*:
    ‹¬ *last cpt* −*e*→ (*fin*, *S1*)› **by** *fast*
  show ‹*cpt* @ [(*fin*, *S1*)] ∈ *assume pre rely*›
    using *assume-snoc cpt-assume cpt-nonnil not-etran* **by** *blast*
qed

lemma *unlift-seq-estran*:
  assumes *all-seq*: ‹*all-seq Q cpt*›
    and *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)›
    and *i*: ‹*Suc i* < *length cpt*›
    and *tran*: ‹(*cpt*!*i*, *cpt*!*Suc i*) ∈ (*estran* Γ)›
  shows ‹(*unlift-seq-cpt cpt* ! *i*, *unlift-seq-cpt cpt* ! *Suc i*) ∈ (*estran* Γ)›
proof−
  let *?part* = ‹*drop i cpt*›
  from *i* **have** *i'*: ‹*i* < *length cpt*› **by** *simp*
  from *cpts-drop cpt i'* **have** ‹*?part* ∈ *cpts* (*estran* Γ)› **by** *blast*
  with *cpts-es-mod-equiv* **have** *part-cpt*: ‹*?part* ∈ *cpts-es-mod* Γ› **by** *blast*
  show *?thesis* **using** *part-cpt*
  proof(*cases*)
    case (*CptsModOne P s*)
    then show *?thesis* **using** *i*
      by (*metis Cons-nth-drop-Suc i' list.discI list.sel*(*3*))
  next
    case (*CptsModEnv P t y cs s x*)
    with *tran* **have** ‹((*P*,*s*,*x*),(*P*,*t*,*y*)) ∈ (*estran* Γ)›
      using *Cons-nth-drop-Suc i' nth-via-drop* **by** *fastforce*
    then have *False* **apply** (*simp add*: *estran-def*)
      using *no-estran-to-self* **by** *fast*
    then show *?thesis* **by** *blast*
  next
    case (*CptsModAnon*)

from *CptsModAnon*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModAnon-fin*)
from *CptsModAnon-fin*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModBasic*)
from *CptsModBasic*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModAtom*)
from *CptsModAtom*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModSeq P1 s x a Q1 t y R cs*)
then have *eq1*:
‹*map unlift-seq-esconf cpt* ! *i* = (*P1*,*s*,*x*)›
by (*simp add*: *i′ nth-via-drop*)
from *CptsModSeq* **have** *eq2*:
‹*map unlift-seq-esconf cpt* ! *Suc i* = (*Q1*,*t*,*y*)›
by (*metis Cons-nth-drop-Suc i i′ list.sel*(*1*) *list.sel*(*3*) *nth-map unlift-seq.simps unlift-seq-esconf.simps*)
from *CptsModSeq*(*2*) *eq1 eq2* **show** *?thesis*
apply(*unfold estran-def*) **by** *auto*
next
case (*CptsModSeq-fin*)
from *CptsModSeq-fin*(*1*) *all-seq all-seq-def* **obtain** *P2* **where** ‹*Q* = *P2 NEXT Q*›
by (*metis* (*no-types*, *lifting*) *Cons-nth-drop-Suc esys.inject*(*4*) *fst-conv i i′ list.inject nth-mem*)
then show *?thesis* **using** *seq-neq2* **by** *metis*
next
case (*CptsModChc1*)
from *CptsModChc1*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModChc2*)
from *CptsModChc2*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModJoin1*)
from *CptsModJoin1*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next
case (*CptsModJoin2*)
from *CptsModJoin2*(*1*) *all-seq all-seq-def* **show** *?thesis*
using *i′ nth-mem nth-via-drop* **by** *fastforce*
next

    case *CptsModJoin-fin*
    from *CptsModJoin-fin(1) all-seq all-seq-def* **show** *?thesis*
      using *i′ nth-mem nth-via-drop* **by** *fastforce*
  next
    case *CptsModWhileTOnePartial*
    with *all-seq all-seq-def* **show** *?thesis*
      using *i′ nth-mem nth-via-drop* **by** *fastforce*
  next
    case *CptsModWhileTOneFull*
    with *all-seq all-seq-def* **show** *?thesis*
      using *i′ nth-mem nth-via-drop* **by** *fastforce*
  next
    case *CptsModWhileTMore*
    with *all-seq all-seq-def* **show** *?thesis*
      using *i′ nth-mem nth-via-drop* **by** *fastforce*
  next
    case *CptsModWhileF*
    with *all-seq all-seq-def* **show** *?thesis*
      using *i′ nth-mem nth-via-drop* **by** *fastforce*
  qed
qed

lemma *fin-imp-not-all-seq*:
  assumes ‹*fst* (*last cpt*) = *fin*›
    and ‹*cpt* ≠ []›
  shows ‹¬ *all-seq Q cpt*›
  apply(*unfold all-seq-def*)
proof
  assume ‹∀ *c*∈*set cpt*. ∃ *P*. *fst c* = *P NEXT Q*›
  then obtain *P* **where** ‹*fst* (*last cpt*) = *P NEXT Q*›
    using *assms*(*2*) *last-in-set* **by** *blast*
  with *assms*(*1*) **show** *False* **by** *simp*
qed

lemma *all-seq-guar*:
  assumes *all-seq*: ‹*all-seq es2 cpt*›
     and *h1′*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*es1*, *s0*) ∩ *assume pre rely* ⊆ *commit* (*estran* Γ) {*fin*} *guar post*›
    and *cpt*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *s0*) ∩ *assume pre rely*›
  shows ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ (*estran* Γ) ⟶ (*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *guar*›
proof−
  let *?cpt′* = ‹*unlift-seq-cpt cpt*›
  from *all-seq-unlift*[*of es2 cpt* Γ *es1 s0 pre rely*] *all-seq cpt* **have** *cpt′*:
    ‹*?cpt′* ∈ *cpts-from* (*estran* Γ) (*es1*, *s0*) ∩ *assume pre rely*› **by** *blast*
  with *h1′* **have** ‹*?cpt′* ∈ *commit* (*estran* Γ) {*fin*} *guar post*› **by** *blast*
  then have *guar*:
    ‹∀ *i*. *Suc i* < *length ?cpt′* ⟶ (*?cpt′*!*i*, *?cpt′*!*Suc i*) ∈ (*estran* Γ) ⟶ (*snd* (*?cpt′*!*i*), *snd* (*?cpt′*!*Suc i*)) ∈ *guar*›

by (*simp add*: *commit-def*)
show *?thesis*
proof
fix *i*
from *guar* have *guar-i*: ‹*Suc i* < *length ?cpt'* ⟶ (*?cpt'*!*i*, *?cpt'*!*Suc i*) ∈ (*estran* Γ) ⟶ (*snd* (*?cpt'*!*i*), *snd* (*?cpt'*!*Suc i*)) ∈ *guar*› by *blast*
show ‹*Suc i* < *length cpt* ⟶ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ (*estran* Γ) ⟶ (*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *guar*› apply *clarify*
proof−
assume *i*: ‹*Suc i* < *length cpt*›
assume *tran*: ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∈ (*estran* Γ)›
from *cpt* have ‹*cpt* ∈ *cpts* (*estran* Γ)› by *force*
with *unlift-seq-estran*[*of es2 cpt* Γ *i*] *all-seq i tran* have *tran'*:
‹(*?cpt'*!*i*, *?cpt'*!*Suc i*) ∈ (*estran* Γ)› by *blast*
with *guar-i i* show ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *guar*›
by (*metis* (*no-types*, *lifting*) *Suc-lessD length-map nth-map prod.collapse sndI unlift-seq-esconf.simps*)
qed
qed
qed

lemma *part1-cpt-assume*:
assumes *split*:
‹*cpt*!*Suc i* = (*es2*, *S*) ∧
Suc i < *length cpt* ∧
all-seq es2 (*take* (*Suc i*) *cpt*) ∧
unlift-seq-cpt (*take* (*Suc i*) *cpt*) @ [(*fin*,*S*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∧
(*unlift-seq-esconf* (*cpt*!*i*), (*fin*,*S*))∈*estran* Γ›
and *h1'*:
‹∀ *S0*. *cpts-from* (*estran* Γ) (*es1*, *S0*) ∩ *assume pre rely* ⊆ *commit* (*estran* Γ) {*fin*} *guar mid*›
and *cpt*:
‹*cpt* ∈ *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*) ∩ *assume pre rely*›
shows ‹*unlift-seq-cpt* (*take* (*Suc i*) *cpt*)@[(*fin*,*S*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∩ *assume pre rely*›
proof−
let *?part1* = ‹*take* (*Suc i*) *cpt*›
let *?part2* = ‹*drop* (*Suc i*) *cpt*›
let *?part1'* = ‹*unlift-seq-cpt ?part1*›
let *?part1''* = ‹*?part1'*@[(*fin*,*S*)]›

show ‹*?part1''* ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∩ *assume pre rely*›
proof
show ‹*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*)›
using *split* by *blast*
next
from *cpt cpts-nonnil* have ‹*cpt*≠[]› by *auto*
then have ‹*take* (*Suc i*) *cpt* ≠ []› by *simp*

have *1*: ‹*snd* (*hd* (*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*))) ∈ *pre*›
apply(*simp add*: *hd-map*[*OF* ‹*take*(*Suc i*)*cpt*≠[]›])
using *cpt* **by** (*auto simp add*: *assume-def*)
show ‹*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)] ∈ *assume pre rely*›
apply(*auto simp add*: *assume-def*)
using *1* ‹*cpt*≠[]› **apply** *fastforce*
subgoal for *j*
proof(*cases j*=*i*)
case *True*
assume *contra*: ‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *j*) = *fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *Suc j*)›
from *split* **have** ‹*Suc i* < *length cpt*› **by** *argo*
have *1*: ‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *i*) ≠ *fin*›
proof−
from *split* **have** *tran*: ‹(*unlift-seq-esconf* (*cpt*!*i*), (*fin*,*S*))∈*estran* Γ› **by** *argo*
have ∗: ‹*i* < *length* (*take*(*Suc i*)*cpt*)›
by (*simp add*: ‹*Suc i* < *length cpt*›[*THEN Suc-lessD*])
have ‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *i*) ≠ *fin*›
apply(*simp add*: *nth-map*[*OF* ∗])
using *no-estran-from-fin'*[*OF tran*] **.**
then show *?thesis* **by** (*simp add*: ‹*Suc i* < *length cpt*›[*THEN Suc-lessD*] *nth-append*)
qed
have *2*: ‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *Suc i*) = *fin*›
using ‹*cpt*≠[]› ‹*Suc i* < *length cpt*›
by (*metis* (*no-types*, *lifting*) *Suc-leI Suc-lessD length-map length-take min.absorb2 nth-append-length prod.collapse prod.inject*)
from *contra* **have** *False* **using** *True 1 2* **by** *argo*
then show *?thesis* **by** *blast*
next
case *False*
assume *a2*: ‹*j*<*Suc i*›
with *False* **have** ‹*j*<*i*› **by** *simp*
from *split* **have** ‹*Suc i* < *length cpt*› **by** *argo*
from *split* **have** *all-seq*: ‹*all-seq es2* (*take* (*Suc i*) *cpt*)› **by** *argo*
have ∗: ‹*Suc j* < *length* (*take* (*Suc i*) *cpt*)›
using ‹*Suc i* < *length cpt*› ‹*j*<*i*› **by** *auto*
assume *a3*:
‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *j*) =
fst ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *Suc j*)›
then have
‹*fst* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *j*) =
fst ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *Suc j*)›
using ‹*j*<*i*› ‹*Suc i* < *length cpt*›
by (*smt Suc-lessD Suc-mono length-map length-take less-trans-Suc min-less-iff-conj nth-append*)

```
      then have ‹fst (unlift-seq-esconf (take (Suc i) cpt ! j)) = fst (unlift-seq-esconf
(take (Suc i) cpt ! Suc j))›
          by (simp add: nth-map[OF ∗] nth-map[OF ∗[THEN Suc-lessD]])
        then have ‹fst (cpt!j) = fst (cpt!Suc j)›
        proof−
        assume a: ‹fst (unlift-seq-esconf (take (Suc i) cpt ! j)) = fst (unlift-seq-esconf
(take (Suc i) cpt ! Suc j))›
          have 1: ‹take (Suc i) cpt ! j = cpt ! j›
            by (simp add: a2)
          have 2: ‹take (Suc i) cpt ! Suc j = cpt ! Suc j›
            by (simp add: ‹j<i›)
          obtain P1 S1 where 3: ‹cpt!j = (P1 NEXT es2, S1)›
            using all-seq apply(simp add: all-seq-def)
            by (metis ∗ 1 Suc-lessD nth-mem prod.collapse)
          obtain P2 S2 where 4: ‹cpt!Suc j = (P2 NEXT es2, S2)›
            using all-seq apply(simp add: all-seq-def)
            by (metis ∗ 2 nth-mem prod.collapse)
          from a have ‹fst (unlift-seq-esconf (cpt ! j)) = fst (unlift-seq-esconf (cpt
! Suc j))›
            by (simp add: 1 2)
          then show ?thesis by (simp add: 3 4)
        qed
        from cpt have ‹cpt ∈ assume pre rely› by blast
          then have ‹fst (cpt!j) = fst (cpt!Suc j) ⟹ (snd (cpt!j), snd (cpt!Suc
j))∈rely›
          apply(auto simp add: assume-def)
          apply(erule allE[where x=j])
          using ‹Suc i < length cpt› ‹j<i› by fastforce
        from this[OF ‹fst (cpt!j) = fst (cpt!Suc j)›]
          have ‹(snd ((map unlift-seq-esconf (take (Suc i) cpt)) ! j), snd ((map
unlift-seq-esconf (take (Suc i) cpt)) ! Suc j)) ∈ rely›
          apply(simp add: nth-map[OF ∗] nth-map[OF ∗[THEN Suc-lessD]])
          using ‹j<i› all-seq
         by (metis (no-types, lifting) Suc-mono a2 nth-take prod.collapse prod.inject
unlift-seq-esconf.simps)
        then show ?thesis
          by (metis (no-types, lifting) ∗ Suc-lessD length-map nth-append)
      qed
      done
  qed
qed

lemma part2-assume:
  assumes split:
    ‹cpt!Suc i = (es2, S) ∧
     Suc i < length cpt ∧
     all-seq es2 (take (Suc i) cpt) ∧
     unlift-seq-cpt (take (Suc i) cpt) @ [(fin,S)] ∈ cpts-from (estran Γ) (es1, S0) ∧
     (unlift-seq-esconf (cpt!i), (fin,S))∈estran Γ›
```

```
    and h1':
    ‹∀S0. cpts-from (estran Γ) (es1, S0) ∩ assume pre rely ⊆ commit (estran Γ)
{fin} guar mid›
    and cpt:
    ‹cpt ∈ cpts-from (estran Γ) (ESeq es1 es2, S0) ∩ assume pre rely›
  shows ‹drop (Suc i) cpt ∈ assume mid rely›
  apply(unfold assume-def)
  apply(subst mem-Collect-eq)
proof
  let ?part1 = ‹take (Suc i) cpt›
  let ?part2 = ‹drop (Suc i) cpt›
  let ?part1' = ‹unlift-seq-cpt ?part1›
  let ?part1'' = ‹?part1'@[(fin,S)]›

  have ‹?part1'' ∈ cpts-from (estran Γ) (es1, S0) ∩ assume pre rely›
    using part1-cpt-assume[OF split h1' cpt] .
  with h1' have ‹?part1'' ∈ commit (estran Γ) {fin} guar mid› by blast
  then have ‹S∈mid›
    by (auto simp add: commit-def)
  then show ‹snd (hd ?part2) ∈ mid›
    by (simp add: split hd-drop-conv-nth)
next
  let ?part2 = ‹drop (Suc i) cpt›
  from cpt have ‹cpt ∈ assume pre rely› by blast
  then have ‹∀j. Suc j < length cpt ⟶ cpt!j −e→ cpt!Suc j ⟶ (snd (cpt!j),
snd (cpt!Suc j)) ∈ rely› by (simp add: assume-def)
  then show ‹∀j. Suc j < length ?part2 ⟶ ?part2!j −e→ ?part2!Suc j ⟶ (snd
(?part2!j), snd (?part2!Suc j)) ∈ rely› by simp
qed

theorem Seq-sound:
  assumes h1:
    ‹Γ ⊨ es1 sat_e [pre, rely, guar, mid]›
  assumes h2:
    ‹Γ ⊨ es2 sat_e [mid, rely, guar, post]›
  shows
    ‹Γ ⊨ ESeq es1 es2 sat_e [pre, rely, guar, post]›
proof−
  let ?pre = ‹lift-state-set pre›
  let ?rely = ‹lift-state-pair-set rely›
  let ?guar = ‹lift-state-pair-set guar›
  let ?post = ‹lift-state-set post›
  let ?mid = ‹lift-state-set mid›

  from h1 have h1':
    ‹∀S0. cpts-from (estran Γ) (es1, S0) ∩ assume ?pre ?rely ⊆ commit (estran
Γ) {fin} ?guar ?mid›
    by (simp)
  from h2 have h2':
```

⟨∀ *S0*. *cpts-from* (*estran* Γ) (*es2*, *S0*) ∩ *assume ?mid ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*⟩
by (*simp*)

have ⟨∀ *S0*. *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*⟩
proof
fix *S0*
show ⟨*cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*⟩
proof
fix *cpt*
assume *cpt*: ⟨*cpt* ∈ *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*) ∩ *assume ?pre ?rely*⟩
from *cpt* **have** *cpt1*: ⟨*cpt* ∈ *cpts-from* (*estran* Γ) (*ESeq es1 es2*, *S0*)⟩ **by** *blast*
then have *cpt-cpts*: ⟨*cpt* ∈ *cpts* (*estran* Γ)⟩ **by** *simp*
then have ⟨*cpt* ≠ []⟩ **using** *cpts-nonnil* **by** *auto*
from *cpt* **have** *hd-cpt*: ⟨*hd cpt* = (*ESeq es1 es2*, *S0*)⟩ **by** *simp*
from *cpt* **have** *cpt-assume*: ⟨*cpt* ∈ *assume ?pre ?rely*⟩ **by** *blast*
show ⟨*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*⟩
apply (*simp add*: *commit-def*)
proof
show ⟨∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?guar*⟩
proof(*cases* ⟨*all-seq es2 cpt*⟩)
case *True*
with *all-seq-guar h1′ cpt* **show** *?thesis* **by** *blast*
next
case *False*
with *split-seq*[*OF cpt1*] **obtain** *i S* **where** *split*:
⟨*cpt* ! *Suc i* = (*es2*, *S*) ∧
Suc i < *length cpt* ∧
all-seq es2 (*take* (*Suc i*) *cpt*) ∧ *map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∧ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ∧ (*unlift-seq-esconf* (*cpt* ! *i*), *fin*, *S*) ∈ *estran* Γ⟩ **by** *blast*
let *?part1* = ⟨*take* (*Suc i*) *cpt*⟩
let *?part1′* = ⟨*unlift-seq-cpt ?part1*⟩
let *?part1″* = ⟨*?part1′* @ [(*fin*,*S*)]⟩
let *?part2* = ⟨*drop* (*Suc i*) *cpt*⟩
from *split* **have**
Suc-i-lt: ⟨*Suc i* < *length cpt*⟩ **and**
all-seq-part1: ⟨*all-seq es2 ?part1*⟩ **by** *argo+*
have *part1-cpt*:
⟨*?part1* ∈ *cpts-from* (*estran* Γ) (*es1 NEXT es2*, *S0*) ∩ *assume ?pre ?rely*⟩
using *cpts-from-assume-take*[*OF cpt*, *of* ⟨*Suc i*⟩] **by** *simp*
have *guar-part1*:
∀ *j*. *Suc j* < *length ?part1* ⟶ (*?part1*!*j*, *?part1*!*Suc j*)∈(*estran* Γ) ⟶ (*snd* (*?part1*!*j*), *snd* (*?part1*!*Suc j*))∈*?guar*

using *all-seq-guar all-seq-part1 h1′ part1-cpt* **by** *blast*
have *guar-part2*:
⟨∀ *j*. *Suc j* < *length ?part2* ⟶ (*?part2*!*j*, *?part2*!*Suc j*)∈(*estran* Γ) ⟶ (*snd* (*?part2*!*j*), *snd* (*?part2*!*Suc j*))∈*?guar*⟩
proof−
from *part2-assume*[*OF - h1′ cpt*] *split* **have** ⟨*?part2* ∈ *assume ?mid ?rely*⟩ **by** *blast*
moreover from *cpts-drop cpt cpts-from-def split* **have** *?part2* ∈ *cpts* (*estran* Γ) **by** *blast*
moreover from *split* **have** ⟨*hd ?part2* = (*es2*, *S*)⟩ **by** (*simp add*: *hd-conv-nth*)
ultimately have ⟨*?part2* ∈ *cpts-from* (*estran* Γ) (*es2*,*S*) ∩ *assume ?mid ?rely*⟩ **by** *fastforce*
with *h2′* **have** ⟨*?part2* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*⟩ **by** *blast*
then show *?thesis* **by** (*simp add*: *commit-def*)
qed
have *guar-tran*:
⟨(*snd* (*last ?part1*), *snd* (*hd ?part2*))∈*?guar*⟩
proof−
have ⟨(*snd* (*?part1″*!*i*), *snd* (*?part1″*!*Suc i*))∈*?guar*⟩
proof−
have *part1″-cpt-asm*: ⟨*?part1″* ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∩ *assume ?pre ?rely*⟩
using *part1-cpt-assume*[*of cpt i es2 S* Γ *es1 S0*, *OF - h1′ cpt*] *split* **by** *blast*
from *split* **have** *tran*: ⟨(*unlift-seq-esconf* (*cpt* ! *i*), *fin*, *S*) ∈ *estran* Γ⟩ **by** *argo*
have ⟨(*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *i* = (*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *i*⟩
using ⟨*Suc i* < *length cpt*⟩ **by** (*simp add*: *nth-append*)
moreover have ⟨(*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *i* = *unlift-seq-esconf* (*cpt* ! *i*)⟩
proof−
have ∗: ⟨*i* < *length* (*take* (*Suc i*) *cpt*)⟩ **using** ⟨*Suc i* < *length cpt*⟩ **by** *simp*
show *?thesis* **by** (*simp add*: *nth-map*[*OF* ∗])
qed
ultimately have *1*: ⟨(*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *i* = (*unlift-seq-esconf* (*cpt*!*i*))⟩ **by** *simp*
have *2*: ⟨(*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *Suc i* = (*fin*, *S*)⟩
using ⟨*Suc i* < *length cpt*⟩
by (*metis* (*no-types*, *lifting*) *length-map length-take min.absorb2 nat-less-le nth-append-length*)
from *tran* **have** *tran′*: ⟨((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *i*, (*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *Suc i*) ∈ *estran* Γ⟩
by (*simp add*: *1 2*)
from *h1′ part1″-cpt-asm* **have** ⟨*?part1″* ∈ *commit* (*estran* Γ) {*fin*}

(*lift-state-pair-set guar*) (*lift-state-set mid*)›
by *blast*
then show *?thesis*
apply(*auto simp add*: *commit-def*)
apply(*erule allE*[**where** *x*=*i*])
using ‹*Suc i* < *length cpt*› *tran′* **by** *linarith*
qed
moreover have ‹*snd* (*?part1″*!*i*) = *snd* (*last ?part1*)›
proof−
have *1*: ‹*snd* (*last* (*take* (*Suc i*) *cpt*)) = *snd* (*cpt*!*i*)› **using** *Suc-i-lt*
by (*simp add*: *last-take-Suc*)
have *2*: ‹*snd* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)]) ! *i*) = *snd* ((*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) ! *i*)›
using *Suc-i-lt*
by (*simp add*: *nth-append*)
have *3*: ‹*i* < *length* (*take* (*Suc i*) *cpt*)› **using** *Suc-i-lt* **by** *simp*
show *?thesis*
apply (*simp add*: *1 2 nth-map*[*OF 3*])
apply(*subst surjective-pairing*[*of* ‹*cpt*!*i*›])
apply(*subst unlift-seq-esconf.simps*)
by *simp*
qed
moreover have ‹*snd* (*?part1″*!*Suc i*) = *snd* (*hd ?part2*)›
proof−
have ‹*snd* (*?part1″*!*Suc i*) = *S*›
proof−
have ‹*length* (*map unlift-seq-esconf* (*take* (*Suc i*) *cpt*)) = *Suc i*› **using** *Suc-i-lt* **by** *simp*
then show *?thesis* **by** (*simp add*: *nth-via-drop*)
qed
moreover have ‹*snd* (*hd ?part2*) = *S*› **using** *split* **by** (*simp add*: *hd-conv-nth*)
ultimately show *?thesis* **by** *simp*
qed
ultimately show *?thesis* **by** *simp*
qed
show *?thesis*
proof
fix *j*
show ‹*Suc j* < *length cpt* ⟶ (*cpt* ! *j*, *cpt* ! *Suc j*) ∈ *estran* Γ ⟶ (*snd* (*cpt* ! *j*), *snd* (*cpt* ! *Suc j*)) ∈ *?guar*›
proof(*cases* ‹*j*<*i*›)
case *True*
then show *?thesis* **using** *guar-part1* **by** *simp*
next
case *False*
then show *?thesis*
proof(*cases* ‹*j*=*i*›)
case *True*

then show *?thesis* **using** *guar-tran*
by (*metis Suc-lessD hd-drop-conv-nth last-take-Suc*)
next
case *False*
with ‹¬*j*<*i*› **have** ‹*j*>*i*› **by** *simp*
then obtain *d* **where** ‹*Suc i* + *d* = *j*›
using *Suc-leI le-Suc-ex* **by** *blast*
then show *?thesis* **using** *guar-part2*[*THEN spec*, *of d*] **by** *simp*
qed
qed
qed
qed
next
show ‹*fst* (*last cpt*) = *fin* ⟶ *snd* (*last cpt*) ∈ *?post*›
proof
assume *fin*: ‹*fst* (*last cpt*) = *fin*›
then have
‹¬ *all-seq es2 cpt*›
using *fin-imp-not-all-seq* ‹*cpt*≠[]› **by** *blast*

with *split-seq*[*OF cpt1*] **obtain** *i S* **where** *split*:
‹*cpt* ! *Suc i* = (*es2*, *S*) ∧
Suc i < *length cpt* ∧
all-seq es2 (*take* (*Suc i*) *cpt*) ∧ *map unlift-seq-esconf* (*take* (*Suc i*) *cpt*) @ [(*fin*, *S*)] ∈ *cpts-from* (*estran* Γ) (*es1*, *S0*) ∧ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ∧ (*unlift-seq-esconf* (*cpt* ! *i*), *fin*, *S*) ∈ *estran* Γ› **by** *blast*
then have
cpt-Suc-i: ‹*cpt*!(*Suc i*) = (*es2*, *S*)› **and**
Suc-i-lt: ‹*Suc i* < *length cpt*› **and**
all-seq: ‹*all-seq es2* (*take* (*Suc i*) *cpt*)› **by** *argo+*
let *?part2* = ‹*drop* (*Suc i*) *cpt*›
from *cpt-Suc-i* **have** *hd-part2*:
‹*hd ?part2* = (*es2*, *S*)›
by (*simp add*: *Suc-i-lt hd-drop-conv-nth*)

have ‹*?part2* ∈ *cpts* (*estran* Γ)› **using** *cpts-drop Suc-i-lt cpt1* **by** *fastforce*
with *cpt-Suc-i* **have** ‹*?part2* ∈ *cpts-from* (*estran* Γ) (*es2*, *S*)›
using *hd-drop-conv-nth Suc-i-lt* **by** *fastforce*
moreover have ‹*?part2* ∈ *assume ?mid ?rely*›
using *part2-assume split h1′ cpt* **by** *blast*
ultimately have ‹*?part2* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **using** *h2′* **by** *blast*
then have *fst* (*last ?part2*) ∈ {*fin*} ⟶ *snd* (*last ?part2*) ∈ *?post*
by (*simp add*: *commit-def*)
moreover from *fin* **have** *fst* (*last ?part2*) = *fin* **using** *Suc-i-lt* **by** *fastforce*
ultimately have ‹*snd* (*last ?part2*) ∈ *?post*› **by** *blast*
then show ‹*snd* (*last cpt*) ∈ *?post*› **using** *Suc-i-lt* **by** *force*
qed
qed

qed
qed
then show *?thesis* **using** *es-validity-def validity-def*
by *metis*
qed

lemma *assume-choice1*:
⟨$(P\ OR\ R,\ S)\ \#\ (Q,\ T)\ \#\ cs \in assume\ pre\ rely \Longrightarrow$
$\Gamma \vdash (P,S)\ {-es[a]\rightarrow}\ (Q,T) \Longrightarrow$
$(P,S)\#(Q,T)\#cs \in assume\ pre\ rely$⟩
apply(*simp add*: *assume-def*)
apply *clarify*
apply(*case-tac i*)
prefer *2*
apply *fastforce*
apply *simp*
using *no-estran-to-self surjective-pairing* **by** *metis*

lemma *assume-choice2*:
⟨$(P\ OR\ R,\ S)\ \#\ (Q,\ T)\ \#\ cs \in assume\ pre\ rely \Longrightarrow$
$\Gamma \vdash (R,S)\ {-es[a]\rightarrow}\ (Q,T) \Longrightarrow$
$(R,S)\#(Q,T)\#cs \in assume\ pre\ rely$⟩
apply(*simp add*: *assume-def*)
apply *clarify*
apply(*case-tac i*)
prefer *2*
apply *fastforce*
apply *simp*
using *no-estran-to-self surjective-pairing* **by** *metis*

lemma *exists-least*:
⟨$P\ (n{::}nat) \Longrightarrow \exists\, m.\ P\ m \wedge (\forall\, i{<}m.\ \neg\ P\ i)$⟩
using *exists-least-iff* **by** *auto*

lemma *choice-sound-aux1*:
⟨$cpt' = map\ (\lambda(\text{-},\ s).\ (P,\ s))\ (take\ (Suc\ m)\ cpt)\ @\ drop\ (Suc\ m)\ cpt \Longrightarrow$
$Suc\ m < length\ cpt \Longrightarrow$
$\forall\, j{<}Suc\ m.\ fst\ (cpt'\ !\ j) = P$⟩
proof
fix *j*
assume *cpt'*: ⟨$cpt' = map\ (\lambda(\text{-},\ s).\ (P,\ s))\ (take\ (Suc\ m)\ cpt)\ @\ drop\ (Suc\ m)\ cpt$⟩
assume *Suc-m-lt*: ⟨$Suc\ m < length\ cpt$⟩
show ⟨$j{<}Suc\ m \longrightarrow fst(cpt'!j) = P$⟩
proof
assume ⟨$j{<}Suc\ m$⟩
with *cpt'* **have** ⟨$cpt'!j = map\ (\lambda(\text{-},\ s).\ (P,\ s))\ (take\ (Suc\ m)\ cpt)\ !\ j$⟩
by (*metis* (*mono-tags*, *lifting*) *Suc-m-lt length-map length-take less-trans min-less-iff-conj nth-append*)

then have ‹fst ($cpt'!j$) = fst (map (λ(-, s). (P, s)) ($take$ (Suc m) cpt) ! j)› **by** *simp*
moreover have ‹fst (map (λ(-, s). (P, s)) ($take$ (Suc m) cpt) ! j) = P› **using** ‹j<Suc m›
by (*simp add*: *Suc-leI Suc-lessD Suc-m-lt case-prod-unfold min.absorb2*)
ultimately show ‹fst($cpt'!j$) = P› **by** *simp*
qed
qed

theorem *Choice-sound*:
assumes *h1*:
‹$\Gamma \models P$ sat_e [pre, $rely$, $guar$, $post$]›
assumes *h2*:
‹$\Gamma \models Q$ sat_e [pre, $rely$, $guar$, $post$]›
shows
‹$\Gamma \models$ *EChc* P Q sat_e [pre, $rely$, $guar$, $post$]›
proof−
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›

from *h1* **have** *h1'*:
‹$\forall$ *S0*. *cpts-from* (*estran* Γ) (P, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
by (*simp*)
from *h2* **have** *h2'*:
‹$\forall$ *S0*. *cpts-from* (*estran* Γ) (Q, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
by (*simp*)
have ‹$\forall$ *S0*. *cpts-from* (*estran* Γ) (*EChc P Q*, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *S0*
show ‹*cpts-from* (*estran* Γ) (*EChc P Q*, *S0*) $\cap$ *assume ?pre ?rely* $\subseteq$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *cpt*
assume *cpt-from-assume*: ‹*cpt* $\in$ *cpts-from* (*estran* Γ) (*EChc P Q*, *S0*) $\cap$ *assume ?pre ?rely*›
then have *cpt*: ‹*cpt* $\in$ *cpts* (*estran* Γ)›
and *hd-cpt*: ‹*hd cpt* = (P *OR* Q, *S0*)›
and *fst-hd-cpt*: *fst* (*hd cpt*) = P *OR* Q
and *cpt-assume*: ‹*cpt* $\in$ *assume ?pre ?rely*› **by** *auto*
from *cpt cpts-nonnil* **have** ‹*cpt*$\neq$[]› **by** *auto*
show ‹*cpt* $\in$ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof(*cases* ‹$\forall$ i. *Suc* i < *length cpt* $\longrightarrow$ *cpt*!i −e→ *cpt*!*Suc* i›)
case *True*
then show *?thesis*

```
      apply(simp add: commit-def)
    proof
      assume ‹∀i. Suc i < length cpt ⟶ fst (cpt ! i) = fst (cpt ! Suc i)›
      then show
        ‹∀i. Suc i < length cpt ⟶ (cpt ! i, cpt ! Suc i) ∈ estran Γ ⟶
            (snd (cpt ! i), snd (cpt ! Suc i)) ∈ ?guar›
        using no-estran-to-self'' by blast
    next
      assume ‹∀i. Suc i < length cpt ⟶ fst (cpt ! i) = fst (cpt ! Suc i)›
      show ‹fst (last cpt) = fin ⟶ snd (last cpt) ∈ ?post›
      proof−
        have ‹∀i<length cpt. fst (cpt ! i) = P OR Q›
          by (rule all-etran-same-prog[OF True fst-hd-cpt ‹cpt≠[]›])
         then have ‹fst (last cpt) = P OR Q› using last-conv-nth ‹cpt≠[]› by force
        then show ?thesis by simp
      qed
    qed
  next
    case False
    then obtain i where 1: ‹Suc i < length cpt ∧ ¬ cpt ! i −e→ cpt ! Suc i› (is ?P i) by blast
    with exists-least[of ?P, OF 1] obtain m where 2: ‹?P m ∧ (∀i<m. ¬?P i)› by blast
     from 2 have Suc-m-lt: ‹Suc m < length cpt› and all-etran: ‹∀i<m. cpt!i −e→ cpt!Suc i› by simp-all
    from 2 have ‹¬ cpt!m −e→ cpt!Suc m› by blast
   then have ctran: ‹(cpt!m, cpt!Suc m) ∈ (estran Γ)› using ctran-or-etran[OF cpt Suc-m-lt] by simp
    have fst-cpt-m: ‹fst (cpt!m) = P OR Q›
    proof−
      let ?cpt = ‹take (Suc m) cpt›
     from Suc-m-lt all-etran have 1: ‹∀i. Suc i < length ?cpt ⟶ ?cpt!i −e→ ?cpt!Suc i› by simp
      from fst-hd-cpt have 2: ‹fst (hd ?cpt) = P OR Q› by simp
      from ‹cpt≠[]› have ‹?cpt ≠ []› by simp
       have ‹∀i<length (take (Suc m) cpt). fst (take (Suc m) cpt ! i) = P OR Q›
        by (rule all-etran-same-prog[OF 1 2 ‹?cpt≠[]›])
      then show ?thesis
        by (simp add: Suc-lessD Suc-m-lt)
    qed
    with ctran show ?thesis
      apply(subst (asm) estran-def)
      apply(subst (asm) mem-Collect-eq)
      apply(subst (asm) case-prod-unfold)
      apply(erule exE)
      apply(erule estran-p.cases, auto)
    proof−
```

```
fix s a P′ t
assume cpt-m: ‹cpt!m = (P OR Q, s)›
assume cpt-Suc-m: ‹cpt!Suc m = (P′, t)›
assume ctran-from-P: ‹Γ ⊢ (P, s) −es[a]→ (P′, t)›
obtain cpt′ where cpt′: ‹cpt′ = map (λ(-,s). (P, s)) (take (Suc m) cpt)
@ drop (Suc m) cpt› by simp
then have cpt′-m: ‹cpt′!m = (P, s)› using Suc-m-lt
  by (simp add: Suc-lessD cpt-m nth-append)
have len-eq: ‹length cpt′ = length cpt› using cpt′ by simp
have same-state: ‹∀ i<length cpt. snd (cpt′!i) = snd (cpt!i)› using cpt′
Suc-m-lt
  by (metis (mono-tags, lifting) append-take-drop-id length-map nth-append
nth-map prod.collapse prod.simps(2) snd-conv)
have ‹cpt′ ∈ cpts-from (estran Γ) (P,S0) ∩ assume ?pre ?rely›
proof
  show ‹cpt′ ∈ cpts-from (estran Γ) (P,S0)›
    apply(subst cpts-from-def′)
  proof
    show ‹cpt′ ∈ cpts (estran Γ)›
      apply(subst cpts-def′)
    proof
      show ‹cpt′≠[]› using cpt′ ‹cpt≠[]› by simp
    next
      show ‹∀ i. Suc i < length cpt′ ⟶ (cpt′ ! i, cpt′ ! Suc i) ∈ estran Γ
∨ cpt′ ! i −e→ cpt′ ! Suc i›
      proof
        fix i
        show ‹Suc i < length cpt′ ⟶ (cpt′ ! i, cpt′ ! Suc i) ∈ estran Γ ∨
cpt′ ! i −e→ cpt′ ! Suc i›
        proof
          assume Suc-i-lt: ‹Suc i < length cpt′›
          show ‹(cpt′ ! i, cpt′ ! Suc i) ∈ estran Γ ∨ cpt′ ! i −e→ cpt′ ! Suc
i›
          proof(cases ‹i<m›)
            case True
            have ‹∀ j < Suc m. fst(cpt′!j) = P› by (rule choice-sound-aux1[OF
cpt′ Suc-m-lt])
            then have all-etran′: ‹∀ j<m. cpt′!j −e→ cpt′!Suc j› by simp
            have ‹cpt′!i −e→ cpt′!Suc i› by (rule all-etran′[THEN spec[where
x=i], rule-format, OF True])
            then show ?thesis by blast
          next
            case False
            have eq-Suc-i: ‹cpt′!Suc i = cpt!Suc i› using cpt′ False Suc-m-lt
              by (metis (no-types, lifting) Suc-less-SucD append-take-drop-id
length-map length-take min-less-iff-conj nth-append)
            show ?thesis
            proof(cases ‹i=m›)
              case True
```

then show *?thesis*
apply *simp*
apply(*rule disjI1*)
using *cpt′-m eq-Suc-i cpt-Suc-m* **apply** (*simp add*: *estran-def*)
using *ctran-from-P* **by** *blast*
next
case *False*
with ‹¬ *i* < *m*› **have** ‹*m*<*i*› **by** *simp*
then have *eq-i*: ‹*cpt′*!*i* = *cpt*!*i*› **using** *cpt′ Suc-m-lt*
by (*metis* (*no-types*, *lifting*) ‹¬ *i* < *m*› *append-take-drop-id length-map length-take less-SucE min-less-iff-conj nth-append*)
from *cpt* **have** ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt*!*i*, *cpt*!*Suc i*)∈*estran* Γ ∨ (*cpt*!*i* −*e*→ *cpt*!*Suc i*)› **using** *cpts-def′* **by** *metis*
then show *?thesis* **using** *eq-i eq-Suc-i Suc-i-lt len-eq* **by** *simp*
qed
qed
qed
qed
qed
next
show ‹*hd cpt′* = (*P*, *S0*)› **using** *cpt′ hd-cpt*
by (*simp add*: ‹*cpt* ≠ []› *hd-map*)
qed
next
show ‹*cpt′* ∈ *assume ?pre ?rely*›
apply(*simp add*: *assume-def*)
proof
from *cpt′* **have** ‹*snd* (*hd cpt′*) = *snd* (*hd cpt*)›
by (*simp add*: ‹*cpt* ≠ []› *hd-cpt hd-map*)
then show ‹*snd* (*hd cpt′*) ∈ *?pre*›
using *cpt-assume* **by** (*simp add*: *assume-def*)
next
show ‹∀ *i*. *Suc i* < *length cpt′* ⟶ *fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof
fix *i*
show ‹*Suc i* < *length cpt′* ⟶ *fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof
assume ‹*Suc i* < *length cpt′*›
with *len-eq* **have** ‹*Suc i* < *length cpt*› **by** *simp*
show ‹*fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof(*cases* ‹*i*<*m*›)
case *True*
from *same-state* ‹*Suc i* < *length cpt′*› *len-eq* **have**
‹*snd* (*cpt′*!*i*) = *snd* (*cpt*!*i*)› **and** ‹*snd* (*cpt′*!*Suc i*) = *snd* (*cpt*!*Suc i*)› **by** *simp-all*
then show *?thesis*

using *cpt-assume* ‹*Suc i* < *length cpt*› *all-etran True* **by** (*auto simp add*: *assume-def*)
next
case *False*
have *eq-Suc-i*: ‹*cpt'*!*Suc i* = *cpt*!*Suc i*› **using** *cpt' False Suc-m-lt*
by (*metis* (*no-types*, *lifting*) *Suc-less-SucD append-take-drop-id length-map length-take min-less-iff-conj nth-append*)
show *?thesis*
proof(*cases* ‹*i*=*m*›)
case *True*
have ‹*fst* (*cpt'*!*i*) ≠ *fst* (*cpt'*!*Suc i*)› **using** *True eq-Suc-i cpt'-m cpt-Suc-m ctran-from-P no-estran-to-self surjective-pairing* **by** *metis*
then show *?thesis* **by** *blast*
next
case *False*
with ‹¬ *i* < *m*› **have** ‹*m*<*i*› **by** *simp*
then have *eq-i*: ‹*cpt'*!*i* = *cpt*!*i*› **using** *cpt' Suc-m-lt*
by (*metis* (*no-types*, *lifting*) ‹¬ *i* < *m*› *append-take-drop-id length-map length-take less-SucE min-less-iff-conj nth-append*)
from *eq-i eq-Suc-i cpt-assume* ‹*Suc i* < *length cpt*›
show *?thesis* **by** (*auto simp add*: *assume-def*)
qed
qed
qed
qed
qed
qed
with *h1'* **have** *cpt'-commit*: ‹*cpt'* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **by** *blast*
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
apply(*simp add*: *commit-def*)
proof
show ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?guar*›
(**is** ‹∀ *i*. *?P i*›)
proof
fix *i*
show ‹*?P i*›
proof(*cases i*<*m*)
case *True*
then show *?thesis*
apply *clarify*
apply(*insert all-etran*[*THEN spec*[**where** *x*=*i*]])
apply *auto*
using *no-estran-to-self''* **apply** *blast*
done
next
case *False*
have *eq-Suc-i*: ‹*cpt'*!*Suc i* = *cpt*!*Suc i*› **using** *cpt' False Suc-m-lt*

by (*metis* (*no-types*, *lifting*) *Suc-less-SucD append-take-drop-id length-map length-take min-less-iff-conj nth-append*)
show *?thesis*
proof(*cases i*=*m*)
case *True*
with *eq-Suc-i* **have** *eq-Suc-m*: ‹*cpt*′!*Suc m* = *cpt*!*Suc m*› **by** *simp*
have *snd-cpt-m-eq*: ‹*snd* (*cpt*!*m*) = *s*› **using** *cpt-m* **by** *simp*
from *True* **show** *?thesis* **using** *cpt*′*-commit*
apply(*simp add*: *commit-def*)
apply *clarify*
apply(*erule allE*[**where** *x*=*i*])
apply (*simp add*: *cpt*′*-m eq-Suc-m cpt-Suc-m estran-def snd-cpt-m-eq len-eq*)
using *ctran-from-P* **by** *blast*
next
case *False*
with ‹¬ *i* < *m*› **have** ‹*m*<*i*› **by** *simp*
then have *eq-i*: ‹*cpt*′!*i* = *cpt*!*i*› **using** *cpt*′ *Suc-m-lt*
by (*metis* (*no-types*, *lifting*) ‹¬ *i* < *m*› *append-take-drop-id length-map length-take less-SucE min-less-iff-conj nth-append*)
from *False* **show** *?thesis* **using** *cpt*′*-commit*
apply(*simp add*: *commit-def*)
apply *clarify*
apply(*erule allE*[**where** *x*=*i*])
apply(*simp add*: *eq-i eq-Suc-i len-eq*)
done
qed
qed
qed
next
have *eq-last*: ‹*last cpt* = *last cpt*′› **using** *cpt*′ *Suc-m-lt* **by** *simp*
show ‹*fst* (*last cpt*) = *fin* ⟶ *snd* (*last cpt*) ∈ *?post*›
using *cpt*′*-commit*
by (*simp add*: *commit-def eq-last*)
qed
next
fix *s a Q*′ *t*
assume *cpt-m*: ‹*cpt*!*m* = (*P OR Q*, *s*)›
assume *cpt-Suc-m*: ‹*cpt*!*Suc m* = (*Q*′, *t*)›
assume *ctran-from-Q*: ‹Γ ⊢ (*Q*, *s*) −*es*[*a*]→ (*Q*′, *t*)›
obtain *cpt*′ **where** *cpt*′: ‹*cpt*′ = *map* (λ(-,*s*). (*Q*, *s*)) (*take* (*Suc m*) *cpt*) @ *drop* (*Suc m*) *cpt*› **by** *simp*
then have *cpt*′*-m*: ‹*cpt*′!*m* = (*Q*, *s*)› **using** *Suc-m-lt*
by (*simp add*: *Suc-lessD cpt-m nth-append*)
have *len-eq*: ‹*length cpt*′ = *length cpt*› **using** *cpt*′ **by** *simp*
have *same-state*: ‹∀ *i*<*length cpt*. *snd* (*cpt*′!*i*) = *snd* (*cpt*!*i*)› **using** *cpt*′ *Suc-m-lt*
by (*metis* (*mono-tags*, *lifting*) *append-take-drop-id length-map nth-append nth-map prod.collapse prod.simps*(*2*) *snd-conv*)

have ‹*cpt′* ∈ *cpts-from* (*estran* Γ) (*Q*,*S0*) ∩ *assume ?pre ?rely*›
proof
show ‹*cpt′* ∈ *cpts-from* (*estran* Γ) (*Q*,*S0*)›
apply(*subst cpts-from-def′*)
proof
show ‹*cpt′* ∈ *cpts* (*estran* Γ)›
apply(*subst cpts-def′*)
proof
show ‹*cpt′*≠[]› **using** *cpt′* ‹*cpt*≠[]› **by** *simp*
next
show ‹∀ *i*. *Suc i* < *length cpt′* ⟶ (*cpt′* ! *i*, *cpt′* ! *Suc i*) ∈ *estran* Γ ∨ *cpt′* ! *i* −*e*→ *cpt′* ! *Suc i*›
proof
fix *i*
show ‹*Suc i* < *length cpt′* ⟶ (*cpt′* ! *i*, *cpt′* ! *Suc i*) ∈ *estran* Γ ∨ *cpt′* ! *i* −*e*→ *cpt′* ! *Suc i*›
proof
assume *Suc-i-lt*: ‹*Suc i* < *length cpt′*›
show ‹(*cpt′* ! *i*, *cpt′* ! *Suc i*) ∈ *estran* Γ ∨ *cpt′* ! *i* −*e*→ *cpt′* ! *Suc i*›
proof(*cases* ‹*i*<*m*›)
case *True*
have ‹∀ *j* < *Suc m*. *fst*(*cpt′*!*j*) = *Q*› **by** (*rule choice-sound-aux1*[*OF cpt′ Suc-m-lt*])
then have *all-etran′*: ‹∀ *j*<*m*. *cpt′*!*j* −*e*→ *cpt′*!*Suc j*› **by** *simp*
have ‹*cpt′*!*i* −*e*→ *cpt′*!*Suc i*› **by** (*rule all-etran′*[*THEN spec*[**where** *x*=*i*], *rule-format*, *OF True*])
then show *?thesis* **by** *blast*
next
case *False*
have *eq-Suc-i*: ‹*cpt′*!*Suc i* = *cpt*!*Suc i*› **using** *cpt′ False Suc-m-lt*
by (*metis* (*no-types*, *lifting*) *Suc-less-SucD append-take-drop-id length-map length-take min-less-iff-conj nth-append*)
show *?thesis*
proof(*cases* ‹*i*=*m*›)
case *True*
then show *?thesis*
apply *simp*
apply(*rule disjI1*)
using *cpt′-m eq-Suc-i cpt-Suc-m* **apply** (*simp add*: *estran-def*)
using *ctran-from-Q* **by** *blast*
next
case *False*
with ‹¬ *i* < *m*› **have** ‹*m*<*i*› **by** *simp*
then have *eq-i*: ‹*cpt′*!*i* = *cpt*!*i*› **using** *cpt′ Suc-m-lt*
by (*metis* (*no-types*, *lifting*) ‹¬ *i* < *m*› *append-take-drop-id length-map length-take less-SucE min-less-iff-conj nth-append*)
from *cpt* **have** ‹∀ *i*. *Suc i* < *length cpt* ⟶ (*cpt*!*i*, *cpt*!*Suc i*)∈*estran* Γ ∨ (*cpt*!*i* −*e*→ *cpt*!*Suc i*)› **using** *cpts-def′* **by** *metis*

then show *?thesis* **using** *eq-i eq-Suc-i Suc-i-lt len-eq* **by** *simp*
qed
qed
qed
qed
qed
next
show ‹*hd cpt′* = (*Q*, *S0*)› **using** *cpt′ hd-cpt*
by (*simp add*: ‹*cpt* ≠ []› *hd-map*)
qed
next
show ‹*cpt′* ∈ *assume ?pre ?rely*›
apply(*simp add*: *assume-def*)
proof
from *cpt′* **have** ‹*snd* (*hd cpt′*) = *snd* (*hd cpt*)›
by (*simp add*: ‹*cpt* ≠ []› *hd-cpt hd-map*)
then show ‹*snd* (*hd cpt′*) ∈ *?pre*›
using *cpt-assume* **by** (*simp add*: *assume-def*)
next
show ‹∀ *i*. *Suc i* < *length cpt′* ⟶ *fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof
fix *i*
show ‹*Suc i* < *length cpt′* ⟶ *fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof
assume ‹*Suc i* < *length cpt′*›
with *len-eq* **have** ‹*Suc i* < *length cpt*› **by** *simp*
show ‹*fst* (*cpt′* ! *i*) = *fst* (*cpt′* ! *Suc i*) ⟶ (*snd* (*cpt′* ! *i*), *snd* (*cpt′* ! *Suc i*)) ∈ *?rely*›
proof(*cases* ‹*i*<*m*›)
case *True*
from *same-state* ‹*Suc i* < *length cpt′*› *len-eq* **have**
‹*snd* (*cpt′*!*i*) = *snd* (*cpt*!*i*)› **and** ‹*snd* (*cpt′*!*Suc i*) = *snd* (*cpt*!*Suc i*)› **by** *simp-all*
then show *?thesis*
using *cpt-assume* ‹*Suc i* < *length cpt*› *all-etran True* **by** (*auto simp add*: *assume-def*)
next
case *False*
have *eq-Suc-i*: ‹*cpt′*!*Suc i* = *cpt*!*Suc i*› **using** *cpt′ False Suc-m-lt*
by (*metis* (*no-types*, *lifting*) *Suc-less-SucD append-take-drop-id length-map length-take min-less-iff-conj nth-append*)
show *?thesis*
proof(*cases* ‹*i*=*m*›)
case *True*
have ‹*fst* (*cpt′*!*i*) ≠ *fst* (*cpt′*!*Suc i*)› **using** *True eq-Suc-i cpt′-m cpt-Suc-m ctran-from-Q no-estran-to-self surjective-pairing* **by** *metis*
then show *?thesis* **by** *blast*

next
case *False*
with ‹¬ i < m› **have** ‹m<i› **by** *simp*
then have *eq-i*: ‹*cpt*''!i = *cpt*!i› **using** *cpt*′ *Suc-m-lt*
by (*metis* (*no-types*, *lifting*) ‹¬ i < m› *append-take-drop-id* *length-map length-take less-SucE min-less-iff-conj nth-append*)
from *eq-i eq-Suc-i cpt-assume* ‹*Suc* i < *length cpt*›
show *?thesis* **by** (*auto simp add*: *assume-def*)
qed
qed
qed
qed
qed
qed
with *h2*′ **have** *cpt*′*-commit*: ‹*cpt*′ ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*› **by** *blast*
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
apply(*simp add*: *commit-def*)
proof
show ‹∀ i. *Suc* i < *length cpt* ⟶ (*cpt* ! i, *cpt* ! *Suc* i) ∈ *estran* Γ ⟶ (*snd* (*cpt* ! i), *snd* (*cpt* ! *Suc* i)) ∈ *?guar*›
(**is** ‹∀ i. *?P* i›)
proof
fix i
show ‹*?P* i›
proof(*cases* i<m)
case *True*
then show *?thesis*
apply *clarify*
apply(*insert all-etran*[*THEN spec*[**where** x=i]])
apply *auto*
using *no-estran-to-self* ′′ **apply** *blast*
done
next
case *False*
have *eq-Suc-i*: ‹*cpt*''!*Suc* i = *cpt*!*Suc* i› **using** *cpt*′ *False Suc-m-lt*
by (*metis* (*no-types*, *lifting*) *Suc-less-SucD append-take-drop-id* *length-map length-take min-less-iff-conj nth-append*)
show *?thesis*
proof(*cases* i=m)
case *True*
with *eq-Suc-i* **have** *eq-Suc-m*: ‹*cpt*''!*Suc* m = *cpt*!*Suc* m› **by** *simp*
have *snd-cpt-m-eq*: ‹*snd* (*cpt*!m) = s› **using** *cpt-m* **by** *simp*
from *True* **show** *?thesis* **using** *cpt*′*-commit*
apply(*simp add*: *commit-def*)
apply *clarify*
apply(*erule allE*[**where** x=i])
apply (*simp add*: *cpt*′*-m eq-Suc-m cpt-Suc-m estran-def snd-cpt-m-eq* *len-eq*)

```
          using ctran-from-Q by blast
        next
          case False
          with ‹¬ i < m› have ‹m<i› by simp
          then have eq-i: ‹cpt'!i = cpt!i› using cpt' Suc-m-lt
              by (metis (no-types, lifting) ‹¬ i < m› append-take-drop-id
length-map length-take less-SucE min-less-iff-conj nth-append)
          from False show ?thesis using cpt'-commit
            apply(simp add: commit-def)
            apply clarify
            apply(erule allE[where x=i])
            apply(simp add: eq-i eq-Suc-i len-eq)
            done
        qed
      qed
    qed
  next
    have eq-last: ‹last cpt = last cpt'› using cpt' Suc-m-lt by simp
    show ‹fst (last cpt) = fin ⟶ snd (last cpt) ∈ ?post›
      using cpt'-commit
      by (simp add: commit-def eq-last)
  qed
 qed
qed
qed
qed
then show ?thesis by simp
qed
```

```
lemma join-sound-aux2:
  assumes cpt-from-assume: ‹cpt ∈ cpts-from (estran Γ) (P ⋈ Q, s0) ∩ assume
pre rely›
    and valid1: ‹∀s0. cpts-from (estran Γ) (P, s0) ∩ assume pre1 rely1 ⊆ commit
(estran Γ) {fin} guar1 post1›
    and valid2: ‹∀s0. cpts-from (estran Γ) (Q, s0) ∩ assume pre2 rely2 ⊆ commit
(estran Γ) {fin} guar2 post2›
    and pre: ‹pre ⊆ pre1 ∩ pre2›
    and rely1: ‹rely ∪ guar2 ⊆ rely1›
    and rely2: ‹rely ∪ guar1 ⊆ rely2›
  shows
    ‹∀i. Suc i < length (fst (split cpt)) ∧ Suc i < length (snd (split cpt)) ⟶
     ((fst (split cpt)!i, fst (split cpt)!Suc i) ∈ estran Γ ⟶ (snd (fst (split cpt)!i),
snd (fst (split cpt)!Suc i)) ∈ guar1) ∧
    ((snd (split cpt)!i, snd (split cpt)!Suc i) ∈ estran Γ ⟶ (snd (snd (split cpt)!i),
snd (snd (split cpt)!Suc i)) ∈ guar2)›
proof-
  let ?cpt1 = ‹fst (split cpt)›
```

let ?cpt2 = ‹snd (split cpt)›
have cpt1-from: ‹?cpt1 ∈ cpts-from (estran Γ) (P,s0)›
using cpt-from-assume split-cpt **by** blast
have cpt2-from: ‹?cpt2 ∈ cpts-from (estran Γ) (Q,s0)›
using cpt-from-assume split-cpt **by** blast
from cpt-from-assume **have** cpt-from: ‹cpt ∈ cpts-from (estran Γ) (P ⋈ Q, s0)›
and cpt-assume: cpt ∈ assume pre rely **by** auto
from cpt-from **have** cpt: ‹cpt ∈ cpts (estran Γ)› **and** fst-hd-cpt: ‹fst (hd cpt) = P ⋈ Q› **by** auto
from cpts-nonnil[OF cpt] **have** ‹cpt≠[]› **.**
show ?thesis
proof(rule ccontr, simp, erule exE)
fix k
assume
‹Suc k < length ?cpt1 ∧ Suc k < length ?cpt2 ∧
((?cpt1 ! k, ?cpt1 ! Suc k) ∈ estran Γ ∧ (snd (?cpt1 ! k), snd (?cpt1 ! Suc k)) ∉ guar1 ∨
(?cpt2 ! k, ?cpt2 ! Suc k) ∈ estran Γ ∧ (snd (?cpt2 ! k), snd (?cpt2 ! Suc k)) ∉ guar2)›
(**is** ?P k)
from exists-least[of ?P k, OF this] **obtain** m **where** ‹?P m ∧ (∀ i<m. ¬?P i)› **by** blast
then show False
proof(auto)
assume Suc-m-lt1: ‹Suc m < length ?cpt1›
assume Suc-m-lt2: ‹Suc m < length ?cpt2›
from Suc-m-lt1 split-length-le1[of cpt] **have** Suc-m-lt: ‹Suc m < length cpt› **by** simp
assume h:
‹∀ i<m. ((?cpt1 ! i, ?cpt1 ! Suc i) ∈ estran Γ ⟶ (snd (?cpt1 ! i), snd (?cpt1 ! Suc i)) ∈ guar1) ∧
((?cpt2 ! i, ?cpt2 ! Suc i) ∈ estran Γ ⟶ (snd (?cpt2 ! i), snd (?cpt2 ! Suc i)) ∈ guar2)›
assume ctran: ‹(?cpt1 ! m, ?cpt1 ! Suc m) ∈ estran Γ›
assume not-guar: ‹(snd (?cpt1 ! m), snd (?cpt1 ! Suc m)) ∉ guar1›
let ?cpt1' = ‹take (Suc (Suc m)) ?cpt1›
from cpt1-from **have** cpt1'-from: ‹?cpt1' ∈ cpts-from (estran Γ) (P,s0)›
by (metis Zero-not-Suc cpts-from-take)
then have cpt1': ‹?cpt1' ∈ cpts (estran Γ)› **by** simp
from ctran **have** ctran': ‹(?cpt1'!m, ?cpt1'!Suc m) ∈ estran Γ› **by** auto
from split-ctran1-aux[OF Suc-m-lt1]
have Suc-m-not-fin: ‹fst (cpt ! Suc m) ≠ fin› **.**
have ‹∀ i. Suc i < length ?cpt1' ⟶ ?cpt1'!i −e→ ?cpt1'!Suc i ⟶ (snd (?cpt1'!i), snd (?cpt1'!Suc i)) ∈ rely ∪ guar2›
proof
fix i
show ‹Suc i < length ?cpt1' ⟶ ?cpt1'!i −e→ ?cpt1'!Suc i ⟶ (snd (?cpt1'!i), snd (?cpt1'!Suc i)) ∈ rely ∪ guar2›
proof(rule impI, rule impI)

assume *Suc-i-lt'*: ‹*Suc i < length ?cpt1'*›
with *Suc-m-lt1* **have** ‹$i \leq m$› **by** *simp*
from *Suc-i-lt'* **have** *Suc-i-lt1*: ‹*Suc i < length ?cpt1*› **by** *simp*
with *split-same-length*[*of cpt*] **have** *Suc-i-lt2*: ‹*Suc i < length ?cpt2*› **by** *simp*
from *no-fin-before-non-fin*[*OF cpt Suc-m-lt Suc-m-not-fin*] ‹$i \leq m$›
have *Suc-i-not-fin*: ‹*fst* (*cpt*!*Suc i*) ≠ *fin*› **by** *fast*
from *Suc-i-lt' split-length-le1*[*of cpt*] **have** *Suc-i-lt*: ‹*Suc i < length cpt*› **by** *simp*
assume *etran'*: ‹*?cpt1'*!*i* −*e*→ *?cpt1'*!*Suc i*›
then have *etran*: ‹*?cpt1*!*i* −*e*→ *?cpt1*!*Suc i*› **using** *Suc-m-lt Suc-i-lt'* **by** (*simp add*: *split-def*)
show ‹(*snd* (*?cpt1'*!*i*), *snd* (*?cpt1'*!*Suc i*)) ∈ *rely* ∪ *guar2*›
proof−
from *split-etran1*[*OF cpt fst-hd-cpt Suc-i-lt Suc-i-not-fin etran*]
have ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i* ∨ (*?cpt2* ! *i*, *?cpt2* ! *Suc i*) ∈ *estran* Γ› **.**
then show *?thesis*
proof
assume *etran*: ‹*cpt*!*i* −*e*→ *cpt*!*Suc i*›
with *cpt-assume Suc-i-lt* **have** ‹(*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *rely*›
by (*simp add*: *assume-def*)
then have ‹(*snd* (*?cpt1*!*i*), *snd* (*?cpt1*!*Suc i*)) ∈ *rely*›
using *split-same-state1*[*OF Suc-i-lt1*] *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]] **by** *argo*
then have ‹(*snd* (*?cpt1'*!*i*), *snd* (*?cpt1'*!*Suc i*)) ∈ *rely*› **using** ‹$i \leq m$› **by** *simp*
then show ‹(*snd* (*?cpt1'*!*i*), *snd* (*?cpt1'*!*Suc i*)) ∈ *rely* ∪ *guar2*› **by** *simp*
next
assume *ctran2*: ‹(*?cpt2*!*i*, *?cpt2*!*Suc i*) ∈ *estran* Γ›
have ‹(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *guar2*›
proof(*cases* ‹$i=m$›)
case *True*
with *ctran etran ctran-imp-not-etran* **show** *?thesis* **by** *blast*
next
case *False*
with ‹$i \leq m$› **have** ‹$i<m$› **by** *linarith*
show *?thesis* **using** *ctran2 h*[*THEN spec*[**where** *x*=*i*], *rule-format*, *OF* ‹$i<m$›] **by** *blast*
qed
thm *split-same-state2*
then have ‹(*snd* (*cpt*!*i*), *snd*(*cpt*!*Suc i*)) ∈ *guar2*›
using *Suc-i-lt2* **by** (*simp add*: *split-same-state2*)
then have ‹(*snd* (*?cpt1*!*i*), *snd* (*?cpt1*!*Suc i*)) ∈ *guar2*›
using *split-same-state1*[*OF Suc-i-lt1*] *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]] **by** *argo*
then have ‹(*snd* (*?cpt1'*!*i*), *snd* (*?cpt1'*!*Suc i*)) ∈ *guar2*› **using** ‹$i \leq m$› **by** *simp*
then show ‹(*snd* (*?cpt1'*!*i*), *snd* (*?cpt1'*!*Suc i*)) ∈ *rely* ∪ *guar2*› **by**

simp

qed

qed

qed

qed

moreover have ‹*snd* (*hd ?cpt1′*) ∈ *pre*›

proof−

have ‹*snd* (*hd cpt*) ∈ *pre*› **using** *cpt-assume* **by** (*simp add*: *assume-def*)

then have ‹*snd* (*hd ?cpt1*) ∈ *pre*› **using** *split-same-state1*

by (*metis* ‹*cpt* ≠ []› *cpt1′ cpts-def′ hd-conv-nth length-greater-0-conv take-eq-Nil*)

then show *?thesis* **by** *simp*

qed

ultimately have ‹*?cpt1′* ∈ *assume pre1 rely1*› **using** *rely1 pre*

by (*auto simp add*: *assume-def*)

with *cpt1′-from pre* **have** ‹*?cpt1′* ∈ *cpts-from* (*estran* Γ) (*P*,*s0*) ∩ *assume pre1 rely1*› **by** *blast*

with *valid1* **have** ‹*?cpt1′* ∈ *commit* (*estran* Γ) {*fin*} *guar1 post1*› **by** *blast*

then have ‹(*snd* (*?cpt1′* ! *m*), *snd* (*?cpt1′* ! *Suc m*)) ∈ *guar1*›

apply(*simp add*: *commit-def*)

apply *clarify*

apply(*erule allE*[**where** *x*=*m*])

using *Suc-m-lt1 ctran′* **by** *simp*

with *not-guar Suc-m-lt* **show** *False* **by** (*simp add*: *Suc-m-lt Suc-lessD*)

next

assume *Suc-m-lt1*: ‹*Suc m* < *length ?cpt1*›

assume *Suc-m-lt2*: ‹*Suc m* < *length ?cpt2*›

from *Suc-m-lt1 split-length-le1*[*of cpt*] **have** *Suc-m-lt*: ‹*Suc m* < *length cpt*› **by** *simp*

assume *h*:

‹∀ *i*<*m*. ((*?cpt1* ! *i*, *?cpt1* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*?cpt1* ! *i*), *snd* (*?cpt1* ! *Suc i*)) ∈ *guar1*) ∧

((*?cpt2* ! *i*, *?cpt2* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*?cpt2* ! *i*), *snd* (*?cpt2* ! *Suc i*)) ∈ *guar2*)›

assume *ctran*: ‹(*?cpt2* ! *m*, *?cpt2* ! *Suc m*) ∈ *estran* Γ›

assume *not-guar*: ‹(*snd* (*?cpt2* ! *m*), *snd* (*?cpt2* ! *Suc m*)) ∉ *guar2*›

let *?cpt2′* = ‹*take* (*Suc* (*Suc m*)) *?cpt2*›

from *cpt2-from* **have** *cpt2′-from*: ‹*?cpt2′* ∈ *cpts-from* (*estran* Γ) (*Q*,*s0*)›

by (*metis Zero-not-Suc cpts-from-take*)

then have *cpt2′*: ‹*?cpt2′* ∈ *cpts* (*estran* Γ)› **by** *simp*

from *ctran* **have** *ctran′*: ‹(*?cpt2′*!*m*, *?cpt2′*!*Suc m*) ∈ *estran* Γ› **by** *fastforce*

from *split-ctran2-aux*[*OF Suc-m-lt2*]

have *Suc-m-not-fin*: ‹*fst* (*cpt* ! *Suc m*) ≠ *fin*› **.**

have ‹∀ *i*. *Suc i* < *length ?cpt2′* ⟶ *?cpt2′*!*i* −*e*→ *?cpt2′*!*Suc i* ⟶ (*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely* ∪ *guar1*›

proof

fix *i*

show ‹*Suc i* < *length ?cpt2′* ⟶ *?cpt2′*!*i* −*e*→ *?cpt2′*!*Suc i* ⟶ (*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely* ∪ *guar1*›

proof(*rule impI*, *rule impI*)
assume *Suc-i-lt′*: ‹*Suc i* < *length ?cpt2′*›
with *Suc-m-lt* **have** ‹*i*≤*m*› **by** *simp*
from *Suc-i-lt′* **have** *Suc-i-lt2*: ‹*Suc i* < *length ?cpt2*› **by** *simp*
with *split-same-length*[*of cpt*] **have** *Suc-i-lt1*: ‹*Suc i* < *length ?cpt1*› **by** *simp*
from *no-fin-before-non-fin*[*OF cpt Suc-m-lt Suc-m-not-fin*] ‹*i*≤*m*›**have** *Suc-i-not-fin*: ‹*fst* (*cpt*!*Suc i*) ≠ *fin*› **by** *fast*
from *Suc-i-lt′ split-length-le2*[*of cpt*] **have** *Suc-i-lt*: ‹*Suc i* < *length cpt*› **by** *simp*
assume *etran′*: ‹*?cpt2′*!*i* −*e*→ *?cpt2′*!*Suc i*›
then have *etran*: ‹*?cpt2*!*i* −*e*→ *?cpt2*!*Suc i*› **using** *Suc-m-lt Suc-i-lt′* **by** (*simp add*: *split-def*)
show ‹(*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely* ∪ *guar1*›
proof−
have ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i* ∨ (*?cpt1* ! *i*, *?cpt1* ! *Suc i*) ∈ *estran* Γ›
by (*rule split-etran2*[*OF cpt fst-hd-cpt Suc-i-lt Suc-i-not-fin etran*])
then show *?thesis*
proof
assume *etran*: ‹*cpt*!*i* −*e*→ *cpt*!*Suc i*›
with *cpt-assume Suc-i-lt* **have** ‹(*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *rely*›
by (*simp add*: *assume-def*)
then have ‹(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *rely*›
using *split-same-state2*[*OF Suc-i-lt2*] *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]] **by** *argo*
then have ‹(*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely*› **using** ‹*i*≤*m*› **by** *simp*
then show ‹(*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely* ∪ *guar1*› **by** *simp*
next
assume *ctran1*: ‹(*?cpt1*!*i*, *?cpt1*!*Suc i*) ∈ *estran* Γ›
then have ‹(*snd* (*?cpt1*!*i*), *snd* (*?cpt1*!*Suc i*)) ∈ *guar1*›
proof(*cases* ‹*i*=*m*›)
case *True*
with *ctran etran ctran-imp-not-etran* **show** *?thesis* **by** *blast*
next
case *False*
with ‹*i*≤*m*› **have** ‹*i*<*m*› **by** *simp*
show *?thesis* **using** *ctran1 h*[*THEN spec*[**where** *x*=*i*], *rule-format*, *OF* ‹*i*<*m*›] **by** *blast*
qed
then have ‹(*snd* (*cpt*!*i*), *snd*(*cpt*!*Suc i*)) ∈ *guar1*›
using *Suc-i-lt1* **by** (*simp add*: *split-same-state1*)
then have ‹(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *guar1*›
using *split-same-state2*[*OF Suc-i-lt2*] *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]] **by** *argo*
then have ‹(*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *guar1*› **using** ‹*i*≤*m*› **by** *simp*
then show ‹(*snd* (*?cpt2′*!*i*), *snd* (*?cpt2′*!*Suc i*)) ∈ *rely* ∪ *guar1*› **by**

```
simp
          qed
        qed
      qed
    qed
    moreover have ‹snd (hd ?cpt2') ∈ pre›
    proof-
      have ‹snd (hd cpt) ∈ pre› using cpt-assume by (simp add: assume-def)
      then have ‹snd (hd ?cpt2) ∈ pre› using split-same-state2
           by (metis ‹cpt ≠ []› cpt2' cpts-def' hd-conv-nth length-greater-0-conv
take-eq-Nil)
      then show ?thesis by simp
    qed
    ultimately have ‹?cpt2' ∈ assume pre2 rely2› using rely2 pre
      by (auto simp add: assume-def)
    with cpt2'-from have ‹?cpt2' ∈ cpts-from (estran Γ) (Q,s0) ∩ assume pre2
rely2› by blast
    with valid2 have ‹?cpt2' ∈ commit (estran Γ) {fin} guar2 post2› by blast
    then have ‹(snd (?cpt2' ! m), snd (?cpt2' ! Suc m)) ∈ guar2›
      apply(simp add: commit-def)
      apply clarify
      apply(erule allE[where x=m])
      using Suc-m-lt2 ctran' by simp
    with not-guar Suc-m-lt show False by (simp add: Suc-m-lt Suc-lessD)
  qed
 qed
qed

lemma join-sound-aux3a:
 ‹(c1, c2) ∈ estran Γ ⟹ ∃P' Q'. fst c1 = P' ⋈ Q' ⟹ fst c2 = fin ⟹ ∀s.
(s,s)∈guar ⟹ (snd c1, snd c2) ∈ guar›
 apply(subst (asm) surjective-pairing[of c1])
 apply(subst (asm) surjective-pairing[of c2])
 apply(erule exE, erule exE)
 apply(simp add: estran-def)
 apply(erule exE)
 apply(erule estran-p.cases, auto)
 done


lemma split-assume-pre:
 assumes cpt: cpt ∈ cpts (estran Γ)
 assumes fst-hd-cpt: fst (hd cpt) = P ⋈ Q
 assumes cpt-assume: cpt ∈ assume pre rely
 shows
   snd (hd (fst (split cpt))) ∈ pre ∧
    snd (hd (snd (split cpt))) ∈ pre
proof-
```

from *cpt-assume* **have** *pre*: ‹*snd* (*hd cpt*) ∈ *pre*› **using** *assume-def* **by** *blast*
from *cpt cpts-nonnil* **have** *cpt*≠[] **by** *blast*
from *pre hd-conv-nth*[*OF* ‹*cpt*≠[]›] **have** ‹*snd* (*cpt*!*0*) ∈ *pre*› **by** *simp*
obtain *s* **where** *hd-cpt-conv*: ‹*hd cpt* = (*P* ⋈ *Q*, *s*)› **using** *fst-hd-cpt surjective-pairing* **by** *metis*
from ‹*cpt*≠[]› **have** *1*:
‹*snd* (*fst* (*split cpt*)!*0*) ∈ *pre*›
apply−
apply(*subst hd-Cons-tl*[*symmetric*, *of cpt*]) **apply** *assumption*
using *pre hd-cpt-conv* **by** *auto*
from ‹*cpt*≠[]› **have** *2*:
‹*snd* (*snd* (*split cpt*)!*0*) ∈ *pre*›
apply−
apply(*subst hd-Cons-tl*[*symmetric*, *of cpt*]) **apply** *assumption*
using *pre hd-cpt-conv* **by** *auto*
from *cpt fst-hd-cpt* **have** ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *snd* (*hd cpt*))›
using *cpts-from-def'* **by** (*metis surjective-pairing*)
from *split-cpt*[*OF this*] **have** *cpt1*:
fst (*split cpt*) ∈ *cpts* (*estran* Γ)
and *cpt2*:
snd (*split cpt*) ∈ *cpts* (*estran* Γ) **by** *auto*
from *cpt1 cpts-nonnil* **have** *cpt1-nonnil*: ‹*fst*(*split cpt*)≠[]› **by** *blast*
from *cpt2 cpts-nonnil* **have** *cpt2-nonnil*: ‹*snd*(*split cpt*)≠[]› **by** *blast*
from *1 2 hd-conv-nth*[*OF cpt1-nonnil*] *hd-conv-nth*[*OF cpt2-nonnil*] **show** *?thesis* **by** *simp*
qed

lemma *join-sound-aux3-1*:
‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume pre rely* ⟹
∀ *s0*. *cpts-from* (*estran* Γ) (*P*, *s0*) ∩ *assume pre1 rely1* ⊆ *commit* (*estran* Γ) {*fin*} *guar1 post1* ⟹
∀ *s0*. *cpts-from* (*estran* Γ) (*Q*, *s0*) ∩ *assume pre2 rely2* ⊆ *commit* (*estran* Γ) {*fin*} *guar2 post2* ⟹
pre ⊆ *pre1* ∩ *pre2* ⟹
rely ∪ *guar2* ⊆ *rely1* ⟹
rely ∪ *guar1* ⊆ *rely2* ⟹
Suc i < *length* (*fst* (*split cpt*)) ⟹
fst (*split cpt*)!*i* −*e*→ *fst* (*split cpt*)!*Suc i* ⟹
(*snd* (*fst* (*split cpt*)!*i*), *snd* (*fst* (*split cpt*)!*Suc i*)) ∈ *rely* ∪ *guar2*›
proof−
assume *cpt-from-assume*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume pre rely*›
then have *cpt-from*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*)›
and *cpt-assume*: ‹*cpt* ∈ *assume pre rely*›
and ‹*cpt*≠[]› **apply** *auto* **using** *cpts-nonnil* **by** *blast*
from *cpt-from* **have** *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **and** *hd-cpt*: ‹*hd cpt* = (*P* ⋈ *Q*, *s0*)› **by** *auto*
from *hd-cpt* **have** *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*› **by** *simp*
assume *valid1*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*P*, *s0*) ∩ *assume pre1 rely1* ⊆ *commit*

(*estran* Γ) {*fin*} *guar1 post1*›
assume *valid2*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*Q*, *s0*) ∩ *assume pre2 rely2* ⊆ *commit* (*estran* Γ) {*fin*} *guar2 post2*›
assume *pre*: ‹*pre* ⊆ *pre1* ∩ *pre2*›
assume *rely1*: ‹*rely* ∪ *guar2* ⊆ *rely1*›
assume *rely2*: ‹*rely* ∪ *guar1* ⊆ *rely2*›
let *?cpt1* = ‹*fst* (*split cpt*)›
let *?cpt2* = ‹*snd* (*split cpt*)›
assume *Suc-i-lt1*: ‹*Suc i* < *length ?cpt1*›
from *Suc-i-lt1 split-same-length* **have** *Suc-i-lt2*: ‹*Suc i* < *length ?cpt2*› **by** *metis*
from *Suc-i-lt1 split-length-le1*[*of cpt*] **have** *Suc-i-lt*: ‹*Suc i* < *length cpt*› **by** *simp*
assume *etran1*: ‹*?cpt1*!*i* −*e*→ *?cpt1*!*Suc i*›
from *split-cpt*[*OF cpt-from*, *THEN conjunct1*] **have** *cpt1-from*: ‹*?cpt1* ∈ *cpts-from* (*estran* Γ) (*P*, *s0*)› **.**
from *split-cpt*[*OF cpt-from*, *THEN conjunct2*] **have** *cpt2-from*: ‹*?cpt2* ∈ *cpts-from* (*estran* Γ) (*Q*, *s0*)› **.**
from *cpt1-from* **have** *cpt1*: ‹*?cpt1* ∈ *cpts* (*estran* Γ)› **by** *auto*
from *cpt2-from* **have** *cpt2*: ‹*?cpt2* ∈ *cpts* (*estran* Γ)› **by** *auto*
from *cpts-nonnil*[*OF cpt1*] **have** ‹*?cpt1*≠[]› **.**
from *cpts-nonnil*[*OF cpt2*] **have** ‹*?cpt2*≠[]› **.**
from *ctran-or-etran*[*OF cpt Suc-i-lt*]
show ‹(*snd* (*?cpt1*!*i*), *snd*(*?cpt1*!*Suc i*)) ∈ *rely* ∪ *guar2*›
proof
assume *ctran-no-etran*: ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ∧ ¬ *cpt* ! *i* −*e*→ *cpt* ! *Suc i*›
from *split-ctran1-aux*[*OF Suc-i-lt1*] **have** *Suc-i-not-fin*: ‹*fst* (*cpt* ! *Suc i*) ≠ *fin*› **.**
from *split-ctran*[*OF cpt fst-hd-cpt Suc-i-not-fin Suc-i-lt ctran-no-etran*[*THEN conjunct1*]] **show** *?thesis*
proof
assume ‹(*fst* (*split cpt*) ! *i*, *fst* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *snd* (*split cpt*) ! *i* −*e*→ *snd* (*split cpt*) ! *Suc i*›
with *ctran-or-etran*[*OF cpt1 Suc-i-lt1*] *etran1* **have** *False* **by** *blast*
then show *?thesis* **by** *blast*
next
assume ‹(*snd* (*split cpt*) ! *i*, *snd* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *fst* (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i*›
from *join-sound-aux2*[*OF cpt-from-assume valid1 valid2 pre rely1 rely2*, *rule-format*, *OF conjI*[*OF Suc-i-lt1 Suc-i-lt2*], *THEN conjunct2*, *rule-format*, *OF this*[*THEN conjunct1*]]
have ‹(*snd* (*snd* (*split cpt*) ! *i*), *snd* (*snd* (*split cpt*) ! *Suc i*)) ∈ *guar2*› **.**
with *split-same-state1*[*OF Suc-i-lt1*] *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]] *split-same-state2*[*OF Suc-i-lt2*] *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]]
have ‹(*snd* (*fst* (*split cpt*) ! *i*), *snd* (*fst* (*split cpt*) ! *Suc i*)) ∈ *guar2*› **by** *simp*
then show *?thesis* **by** *blast*
qed
next
assume ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i* ∧ (*cpt* ! *i*, *cpt* ! *Suc i*) ∉ *estran* Γ›

from *this*[*THEN conjunct1*] *cpt-assume* **have** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *rely*›
apply(*auto simp add*: *assume-def*)
apply(*erule allE*[**where** *x*=*i*])
using *Suc-i-lt* **by** *blast*
with *split-same-state1*[*OF Suc-i-lt1*] *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]]
have ‹(*snd* (*?cpt1*!*i*), *snd* (*?cpt1*!*Suc i*)) ∈ *rely*› **by** *simp*
then show *?thesis* **by** *blast*
qed
qed

lemma *join-sound-aux3-2*:
‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume pre rely* ⟹
∀ *s0*. *cpts-from* (*estran* Γ) (*P*, *s0*) ∩ *assume pre1 rely1* ⊆ *commit* (*estran* Γ) {*fin*} *guar1 post1* ⟹
∀ *s0*. *cpts-from* (*estran* Γ) (*Q*, *s0*) ∩ *assume pre2 rely2* ⊆ *commit* (*estran* Γ) {*fin*} *guar2 post2* ⟹
pre ⊆ *pre1* ∩ *pre2* ⟹
rely ∪ *guar2* ⊆ *rely1* ⟹
rely ∪ *guar1* ⊆ *rely2* ⟹
Suc i < *length* (*snd* (*split cpt*)) ⟹
snd (*split cpt*)!*i* −*e*→ *snd* (*split cpt*)!*Suc i* ⟹
(*snd* (*snd* (*split cpt*)!*i*), *snd* (*snd* (*split cpt*)!*Suc i*)) ∈ *rely* ∪ *guar1*›
proof−
assume *cpt-from-assume*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume pre rely*›
then have *cpt-from*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*)›
and *cpt-assume*: ‹*cpt* ∈ *assume pre rely*›
and ‹*cpt*≠[]› **apply** *auto* **using** *cpts-nonnil* **by** *blast*
from *cpt-from* **have** *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **and** *hd-cpt*: ‹*hd cpt* = (*P* ⋈ *Q*, *s0*)› **by** *auto*
from *hd-cpt* **have** *fst-hd-cpt*: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*› **by** *simp*
assume *valid1*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*P*, *s0*) ∩ *assume pre1 rely1* ⊆ *commit* (*estran* Γ) {*fin*} *guar1 post1*›
assume *valid2*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*Q*, *s0*) ∩ *assume pre2 rely2* ⊆ *commit* (*estran* Γ) {*fin*} *guar2 post2*›
assume *pre*: ‹*pre* ⊆ *pre1* ∩ *pre2*›
assume *rely1*: ‹*rely* ∪ *guar2* ⊆ *rely1*›
assume *rely2*: ‹*rely* ∪ *guar1* ⊆ *rely2*›
let *?cpt1* = ‹*fst* (*split cpt*)›
let *?cpt2* = ‹*snd* (*split cpt*)›
assume *Suc-i-lt2*: ‹*Suc i* < *length ?cpt2*›
from *Suc-i-lt2 split-same-length* **have** *Suc-i-lt1*: ‹*Suc i* < *length ?cpt1*› **by** *metis*
from *Suc-i-lt2 split-length-le2*[*of cpt*] **have** *Suc-i-lt*: ‹*Suc i* < *length cpt*› **by** *simp*
assume *etran2*: ‹*?cpt2*!*i* −*e*→ *?cpt2*!*Suc i*›
from *split-cpt*[*OF cpt-from*, *THEN conjunct1*] **have** *cpt1-from*: ‹*?cpt1* ∈ *cpts-from* (*estran* Γ) (*P*, *s0*)› **.**
from *split-cpt*[*OF cpt-from*, *THEN conjunct2*] **have** *cpt2-from*: ‹*?cpt2* ∈ *cpts-from* (*estran* Γ) (*Q*, *s0*)› **.**

from *cpt1-from* **have** *cpt1*: ‹*?cpt1* ∈ *cpts* (*estran* Γ)› **by** *auto*
from *cpt2-from* **have** *cpt2*: ‹*?cpt2* ∈ *cpts* (*estran* Γ)› **by** *auto*
from *cpts-nonnil*[*OF cpt1*] **have** ‹*?cpt1*≠[]› **.**
from *cpts-nonnil*[*OF cpt2*] **have** ‹*?cpt2*≠[]› **.**
from *ctran-or-etran*[*OF cpt Suc-i-lt*]
show ‹(*snd* (*?cpt2*!*i*), *snd*(*?cpt2*!*Suc i*)) ∈ *rely* ∪ *guar1*›
proof
assume *ctran-no-etran*: ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ ∧ ¬ *cpt* ! *i* −*e*→ *cpt* ! *Suc i*›
from *split-ctran1-aux*[*OF Suc-i-lt1*] **have** *Suc-i-not-fin*: ‹*fst* (*cpt* ! *Suc i*) ≠ *fin*› **.**
from *split-ctran*[*OF cpt fst-hd-cpt Suc-i-not-fin Suc-i-lt ctran-no-etran*[*THEN conjunct1*]] **show** *?thesis*
proof
assume ‹(*fst* (*split cpt*) ! *i*, *fst* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *snd* (*split cpt*) ! *i* −*e*→ *snd* (*split cpt*) ! *Suc i*›
from *join-sound-aux2*[*OF cpt-from-assume valid1 valid2 pre rely1 rely2*, *rule-format*, *OF conjI*[*OF Suc-i-lt1 Suc-i-lt2*], *THEN conjunct1*, *rule-format*, *OF this*[*THEN conjunct1*]]
have ‹(*snd* (*fst* (*split cpt*) ! *i*), *snd* (*fst* (*split cpt*) ! *Suc i*)) ∈ *guar1*› **.**
with *split-same-state1*[*OF Suc-i-lt1*] *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]] *split-same-state2*[*OF Suc-i-lt2*] *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]]
have ‹(*snd* (*snd* (*split cpt*) ! *i*), *snd* (*snd* (*split cpt*) ! *Suc i*)) ∈ *guar1*› **by** *simp*
then show *?thesis* **by** *blast*
next
assume ‹(*snd* (*split cpt*) ! *i*, *snd* (*split cpt*) ! *Suc i*) ∈ *estran* Γ ∧ *fst* (*split cpt*) ! *i* −*e*→ *fst* (*split cpt*) ! *Suc i*›
with *ctran-or-etran*[*OF cpt2 Suc-i-lt2*] *etran2* **have** *False* **by** *blast*
then show *?thesis* **by** *blast*
qed
next
assume ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i* ∧ (*cpt* ! *i*, *cpt* ! *Suc i*) ∉ *estran* Γ›
from *this*[*THEN conjunct1*] *cpt-assume* **have** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *rely*›
apply(*auto simp add*: *assume-def*)
apply(*erule allE*[**where** *x*=*i*])
using *Suc-i-lt* **by** *blast*
with *split-same-state2*[*OF Suc-i-lt2*] *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]]
have ‹(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *rely*› **by** *simp*
then show *?thesis* **by** *blast*
qed
qed

lemma *split-cpt-nonnil*:
‹*cpt* ≠ [] ⟹ *fst* (*hd cpt*) = *P* ⋈ *Q* ⟹ *fst* (*split cpt*) ≠ [] ∧ *snd* (*split cpt*) ≠ []›
apply(*rule conjI*)
apply(*subst hd-Cons-tl*[*of cpt*, *symmetric*]) **apply** *assumption*

apply(*subst surjective-pairing*[*of* ‹*hd cpt*›])
apply *simp*
apply(*subst hd-Cons-tl*[*of cpt*, *symmetric*]) **apply** *assumption*
apply(*subst surjective-pairing*[*of* ‹*hd cpt*›])
apply *simp*
done

lemma *join-sound-aux5*:
‹$cpt \in cpts\text{-}from\ (estran\ \Gamma)\ (P \bowtie Q,\ S0) \cap assume\ pre\ rely \Longrightarrow$
$\forall S0.\ cpts\text{-}from\ (estran\ \Gamma)\ (P,\ S0) \cap assume\ pre1\ rely1 \subseteq commit\ (estran\ \Gamma)$ $\{fin\}\ guar1\ post1 \Longrightarrow$
$\forall S0.\ cpts\text{-}from\ (estran\ \Gamma)\ (Q,\ S0) \cap assume\ pre2\ rely2 \subseteq commit\ (estran\ \Gamma)$ $\{fin\}\ guar2\ post2 \Longrightarrow$
$pre \subseteq pre1 \cap pre2 \Longrightarrow$
$rely \cup guar2 \subseteq rely1 \Longrightarrow$
$rely \cup guar1 \subseteq rely2 \Longrightarrow$
$fst\ (last\ cpt) \in \{fin\} \longrightarrow snd\ (last\ cpt) \in post1 \cap post2$›
proof−
assume *cpt-from-assume*: ‹$cpt \in cpts\text{-}from\ (estran\ \Gamma)\ (P \bowtie Q,\ S0) \cap assume\ pre\ rely$›
then have *cpt*: ‹$cpt \in cpts\ (estran\ \Gamma)$›
and *fst-hd-cpt*: ‹$fst\ (hd\ cpt) = P \bowtie Q$›
and *cpt-assume*: ‹$cpt \in assume\ pre\ rely$›
and *cpt-from*: ‹$cpt \in cpts\text{-}from\ (estran\ \Gamma)\ (P \bowtie Q,\ S0)$›
by *auto*
assume *valid1*: ‹$\forall S0.\ cpts\text{-}from\ (estran\ \Gamma)\ (P,\ S0) \cap assume\ pre1\ rely1 \subseteq commit\ (estran\ \Gamma)\ \{fin\}\ guar1\ post1$›
assume *valid2*: ‹$\forall S0.\ cpts\text{-}from\ (estran\ \Gamma)\ (Q,\ S0) \cap assume\ pre2\ rely2 \subseteq commit\ (estran\ \Gamma)\ \{fin\}\ guar2\ post2$›
assume *pre*: ‹$pre \subseteq pre1 \cap pre2$›
assume *rely1*: ‹$rely \cup guar2 \subseteq rely1$›
assume *rely2*: ‹$rely \cup guar1 \subseteq rely2$›
let *?cpt1* = ‹*fst* (*split cpt*)›
let *?cpt2* = ‹*snd* (*split cpt*)›
from *cpts-nonnil*[*OF cpt*] **have** ‹$cpt \neq []$› **.**
from *split-cpt-nonnil*[*OF* ‹$cpt \neq []$› *fst-hd-cpt*, *THEN conjunct1*] **have** ‹$?cpt1 \neq []$› **.**
from *split-cpt-nonnil*[*OF* ‹$cpt \neq []$› *fst-hd-cpt*, *THEN conjunct2*] **have** ‹$?cpt2 \neq []$› **.**
show *?thesis*
proof(*cases* ‹$fst\ (last\ cpt) = fin$›)
case *True*
with *last-conv-nth*[*OF* ‹$cpt \neq []$›] **have** ‹$fst\ (cpt\ !\ (length\ cpt - 1)) = fin$› **by** *simp*
from *exists-least*[**where** P=‹$\lambda i.\ fst\ (cpt!i) = fin$›, *OF this*]
obtain *m* **where** *m*: ‹$fst\ (cpt\ !\ m) = fin \wedge (\forall i{<}m.\ fst\ (cpt\ !\ i) \neq fin)$› **by** *blast*
note *m-fin* = *m*[*THEN conjunct1*]
have ‹$m \neq 0$›

apply(*rule ccontr*)
apply(*insert m*)
apply(*insert* ‹*fst* (*hd cpt*) = *P* ⋈ *Q*›)
apply(*subst* (*asm*) *hd-conv-nth*) **apply**(*rule* ‹*cpt*≠[]›)
apply *simp*
done
then obtain *m′* **where** *m′*: ‹*m* = *Suc m′*› **using** *not0-implies-Suc* **by** *blast*
have *m-lt*: ‹*m* < *length cpt*›
proof(*rule ccontr*)
assume *h*: ‹¬ *m* < *length cpt*›
from *m*[*THEN conjunct2*] **have** ‹∀ *i*<*m*. *fst* (*cpt* ! *i*) ≠ *fin*› **.**
then have ‹*fst* (*cpt* ! (*length cpt* − *1*)) ≠ *fin*›
apply−
apply(*erule allE*[**where** *x*=‹*length cpt* − *1*›])
using *h* **by** (*metis* ‹*cpt* ≠ []› *diff-less length-greater-0-conv less-imp-diff-less linorder-neqE-nat zero-less-one*)
with *last-conv-nth*[*OF* ‹*cpt*≠[]›] **have** ‹*fst* (*last cpt*) ≠ *fin*› **by** *simp*
with ‹*fst* (*last cpt*) = *fin*› **show** *False* **by** *blast*
qed
with *m′* **have** *Suc-m′-lt*: ‹*Suc m′* < *length cpt*› **by** *simp*
from *m m′* **have** *m1*: ‹*fst* (*cpt* ! *Suc m′*) = *fin* ∧ (∀ *i*<*Suc m′*. *fst* (*cpt* ! *i*) ≠ *fin*)› **by** *simp*
from *m1*[*THEN conjunct1*] **obtain** *s* **where** *cpt-Suc-m′*: ‹*cpt*!*Suc m′* = (*fin*, *s*)› **using** *surjective-pairing* **by** *metis*
from *m1* **have** *m′-not-fin*: ‹*fst* (*cpt*!*m′*) ≠ *fin*›
apply *clarify*
apply(*erule allE*[**where** *x*=*m′*])
by *fast*
have ‹*fst* (*cpt*!*m′*) = *fin* ⋈ *fin*›
proof−
from *ctran-or-etran*[*OF cpt Suc-m′-lt*]
have ‹(*cpt* ! *m′*, *cpt* ! *Suc m′*) ∈ *estran* Γ ∧ ¬ *cpt* ! *m′* −*e*→ *cpt* ! *Suc m′* ∨ *cpt* ! *m′* −*e*→ *cpt* ! *Suc m′* ∧ (*cpt* ! *m′*, *cpt* ! *Suc m′*) ∉ *estran* Γ› **.**
moreover have ‹¬ *cpt* ! *m′* −*e*→ *cpt* ! *Suc m′*›
proof(*rule ccontr*, *simp*)
assume *h*: ‹*fst* (*cpt* ! *m′*) = *fst* (*cpt* ! *Suc m′*)›
from *m1*[*THEN conjunct1*] *m′-not-fin h* **show** *False* **by** *simp*
qed
ultimately have *ctran*: ‹(*cpt* ! *m′*, *cpt* ! *Suc m′*) ∈ *estran* Γ› **by** *blast*
with *cpt-Suc-m′* **show** *?thesis*
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*insert all-join*[*OF cpt fst-hd-cpt Suc-m′-lt*[*THEN Suc-lessD*] *m′-not-fin*, *rule-format*, *of m′*])
apply(*erule estran-p.cases*, *auto*)
done
qed
have ‹*length ?cpt1* = *m* ∧ *length ?cpt2* = *m*›
using *split-length*[*OF cpt fst-hd-cpt Suc-m′-lt m′-not-fin m1*[*THEN conjunct1*]]

m' **by** *simp*
then have ‹*length ?cpt1* = *m*› **and** ‹*length ?cpt2* = *m*› **by** *auto*

from ‹*length ?cpt1* = *m*› *m-lt* **have** *cpt1-shorter*: ‹*length ?cpt1* < *length cpt*› **by** *simp*
from ‹*length ?cpt2* = *m*› *m-lt* **have** *cpt2-shorter*: ‹*length ?cpt2* < *length cpt*› **by** *simp*

have ‹m' < *length ?cpt1*› **using** ‹*length ?cpt1* = *m*› m' **by** *simp*
from *split-prog1*[*OF this* ‹*fst* (*cpt*!m') = *fin* ⋈ *fin*›]
have ‹*fst* (*fst* (*split cpt*) ! m') = *fin*› **.**
moreover have ‹*last ?cpt1* = *?cpt1* ! m'›
apply(*subst last-conv-nth*[*OF* ‹*?cpt1*≠[]›])
using m' ‹*length ?cpt1* = *m*› **by** *simp*
ultimately have ‹*fst* (*last* (*fst* (*split cpt*))) = *fin*› **by** *simp*

have ‹m' < *length ?cpt2*› **using** ‹*length ?cpt2* = *m*› m' **by** *simp*
from *split-prog2*[*OF this* ‹*fst* (*cpt*!m') = *fin* ⋈ *fin*›]
have ‹*fst* (*snd* (*split cpt*) ! m') = *fin*› **.**
moreover have ‹*last ?cpt2* = *?cpt2* ! m'›
apply(*subst last-conv-nth*[*OF* ‹*?cpt2*≠[]›])
using m' ‹*length ?cpt2* = *m*› **by** *simp*
ultimately have ‹*fst* (*last* (*snd* (*split cpt*))) = *fin*› **by** *simp*

let *?cpt1'* = ‹*?cpt1* @ *drop* (*Suc m*) *cpt*›
let *?cpt2'* = ‹*?cpt2* @ *drop* (*Suc m*) *cpt*›

from *split-cpt*[*OF cpt-from*, *THEN conjunct1*, *simplified*, *THEN conjunct2*]
have ‹*hd* (*fst* (*split cpt*)) = (*P*, *S0*)› **.**
with *hd-Cons-tl*[*OF* ‹*?cpt1*≠[]›]
have ‹*?cpt1* = (*P*,*S0*) # *tl ?cpt1*› **by** *simp*
from *split-cpt*[*OF cpt-from*, *THEN conjunct2*, *simplified*, *THEN conjunct2*]
have ‹*hd* (*snd* (*split cpt*)) = (*Q*, *S0*)› **.**
with *hd-Cons-tl*[*OF* ‹*?cpt2*≠[]›]
have ‹*?cpt2* = (*Q*,*S0*) # *tl ?cpt2*› **by** *simp*

have *cpt'-from*: ‹*?cpt1'* ∈ *cpts-from* (*estran* Γ) (*P*,*S0*) ∧ *?cpt2'* ∈ *cpts-from* (*estran* Γ) (*Q*,*S0*)›
proof(*cases* ‹*Suc m* < *length cpt*›)
case *True*
then have ‹*m* < *length cpt*› **by** *simp*
have ‹*m* < *Suc m*› **by** *simp*
from *all-fin-after-fin''*[*OF cpt* ‹*m* < *length cpt*› *m-fin*, *rule-format*, *OF* ‹*m* < *Suc m*› *True*]
have ‹*fst* (*cpt* ! *Suc m*) = *fin*› **.**
then have ‹*fst* (*hd* (*drop* (*Suc m*) *cpt*)) = *fin*› **by** (*simp add*: *True hd-drop-conv-nth*)
show *?thesis*
apply *auto*
apply(*rule cpts-append-env*)

```
        using split-cpt cpt-from-assume apply fastforce
            apply(rule cpts-drop[OF cpt True])
           apply(simp add: ‹fst (last (fst (split cpt))) = fin› ‹fst (hd (drop (Suc m)
cpt)) = fin›)
          apply(subst ‹?cpt1 = (P,S0) # tl (fst (split cpt))›)
          apply simp
         apply(rule cpts-append-env)
        using split-cpt cpt-from-assume apply fastforce
          apply(rule cpts-drop[OF cpt True])
         apply(simp add: ‹fst (last (snd (split cpt))) = fin› ‹fst (hd (drop (Suc m)
cpt)) = fin›)
        apply(subst ‹?cpt2 = (Q,S0) # tl ?cpt2›)
        apply simp
        done
    next
      case False
      then have ‹length cpt ≤ Suc m› by simp
      from drop-all[OF this]
      show ?thesis
        apply auto
        using split-cpt cpt-from-assume apply fastforce
          apply(rule ‹hd (fst (split cpt)) = (P, S0)›)
        using split-cpt cpt-from-assume apply fastforce
        apply(rule ‹hd (snd (split cpt)) = (Q, S0)›)
        done
    qed

    from cpt-from[simplified, THEN conjunct2] have ‹hd cpt = (P ⋈ Q, S0)› .
    have ‹S0 ∈ pre›
      using cpt-assume apply(simp add: assume-def)
      apply(drule conjunct1)
      by (simp add: ‹hd cpt = (P ⋈ Q, S0)›)
    have cpt'-assume: ‹?cpt1' ∈ assume pre1 rely1 ∧ ?cpt2' ∈ assume pre2 rely2›
    proof(auto simp add: assume-def)
      show ‹snd (hd (fst (split cpt) @ drop (Suc m) cpt)) ∈ pre1›
        apply(subst ‹?cpt1 = (P,S0) # tl ?cpt1›)
        apply simp
        using ‹S0∈pre› pre by blast
    next
      fix i
      assume ‹Suc i < length ?cpt1 + (length cpt − Suc m)›
       with ‹length ?cpt1 = m› Suc-leI[OF m-lt] have ‹Suc (Suc i) < length cpt›
by linarith
      then have ‹Suc i < length cpt› by simp
      assume ‹fst (?cpt1'!i) = fst (?cpt1'!Suc i)›
      show ‹(snd (?cpt1'!i), snd (?cpt1'!Suc i)) ∈ rely1›
      proof(cases ‹Suc i < length ?cpt1›)
        case True
        from True have ‹?cpt1'!i = ?cpt1!i›
```

```
          by (simp add: Suc-lessD nth-append)
        from True have ‹?cpt1'!Suc i = ?cpt1!Suc i›
          by (simp add: nth-append)
        from ‹fst (?cpt1'!i) = fst (?cpt1'!Suc i)› ‹?cpt1'!i = ?cpt1!i› ‹?cpt1'!Suc i
= ?cpt1!Suc i›
        have ‹?cpt1!i −e→ ?cpt1!Suc i› by simp
        have ‹(snd (fst (split cpt) ! i), snd (fst (split cpt) ! Suc i)) ∈ rely1›
          using join-sound-aux3-1[OF cpt-from-assume valid1 valid2 pre rely1 rely2
True ‹?cpt1!i −e→ ?cpt1!Suc i›] rely1 by blast
        then show ?thesis
          by (simp add: ‹?cpt1'!i = ?cpt1!i› ‹?cpt1'!Suc i = ?cpt1!Suc i›)
      next
        case False
        then have Suc-i-ge: ‹Suc i ≥ length ?cpt1› by simp
        show ?thesis
        proof(cases ‹Suc i = length ?cpt1›)
          case True
          then have ‹i < length ?cpt1› by linarith
          from cpt1-shorter True have ‹Suc i < length cpt› by simp
          from True ‹length ?cpt1 = m› have ‹Suc i = m› by simp
          with m' have ‹i = m'› by simp
          with ‹fst (cpt!m') = fin ⋈ fin› have ‹fst (cpt!i) = fin ⋈ fin› by simp
          from ‹Suc i < length ?cpt1 + (length cpt − Suc m)› ‹Suc i = m› ‹length
?cpt1 = m›
          have ‹Suc m < length cpt› by simp
          from ‹Suc i = m› m-fin have ‹fst (cpt!Suc i) = fin› by simp
          have conv1: ‹snd (?cpt1' ! i) = snd (cpt ! Suc i)›
          proof−
               have ‹snd (?cpt1'!i) = snd (?cpt1!i)› using True by (simp add:
nth-append)
            moreover have ‹snd (?cpt1!i) = snd (cpt!i)›
              using split-same-state1[OF ‹i < length ?cpt1›] .
            moreover have ‹snd (cpt!i) = snd (cpt!Suc i)›
            proof−
              from ctran-or-etran[OF cpt ‹Suc i < length cpt›] ‹fst (cpt!i) = fin ⋈
fin› ‹fst (cpt!Suc i) = fin›
              have ‹(cpt ! i, cpt ! Suc i) ∈ estran Γ› by fastforce
              then show ?thesis
                apply(subst (asm) surjective-pairing[of ‹cpt!i›])
                apply(subst (asm) surjective-pairing[of ‹cpt!Suc i›])
                   apply(simp add: ‹fst (cpt!i) = fin ⋈ fin› ‹fst (cpt!Suc i) = fin›
estran-def)
                apply(erule exE)
                apply(erule estran-p.cases, auto)
                done
            qed
            ultimately show ?thesis by simp
          qed
          have conv2: ‹snd (?cpt1' ! Suc i) = snd (cpt ! Suc (Suc i))›
```

apply(*simp add*: *nth-append True*)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
apply(*simp add*: ‹*length ?cpt1* = *m*›)
done
have ‹(*snd* (*cpt* ! *Suc i*), *snd* (*cpt* ! *Suc* (*Suc i*))) ∈ *rely*›
proof−
have ‹*m*<*Suc m*› **by** *simp*
from *all-fin-after-fin*′′[*OF cpt m-lt m-fin*, *rule-format*, *OF this* ‹*Suc m* < *length cpt*›]
have *Suc-m-fin*: ‹*fst* (*cpt* ! *Suc m*) = *fin*› **.**
from *cpt-assume* **show** *?thesis*
apply(*simp add*: *assume-def*)
apply(*drule conjunct2*)
apply(*erule allE*[**where** *x*=*m*])
using ‹*Suc m* < *length cpt*› *m-fin Suc-m-fin* ‹*Suc i* = *m*› **by** *argo*
qed
then show *?thesis*
apply(*simp add*: *conv1 conv2*) **using** *rely1* **by** *blast*
next
case *False*
with *Suc-i-ge* **have** *Suc-i-gt*: ‹*Suc i* > *length ?cpt1*› **by** *linarith*
with ‹*length ?cpt1* = *m*› **have** ‹¬ *i* < *m*› **by** *simp*
then have ‹*m* < *Suc i*› **by** *simp*
then have ‹*m* < *Suc* (*Suc i*)› **by** *simp*
have *conv1*: ‹*?cpt1*′!*i* = *cpt*!*Suc i*›
apply(*simp add*: *nth-append Suc-i-gt* ‹*length ?cpt1* = *m*› ‹¬ *i* < *m*›)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
using ‹¬*i*<*m*› **by** *simp*
have *conv2*: ‹*?cpt1*′!*Suc i* = *cpt*!*Suc*(*Suc i*)›
using *Suc-i-gt* **apply**(*simp add*: *nth-append*)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
by (*simp add*: ‹*length ?cpt1* = *m*›)
from *all-fin-after-fin*′′[*OF cpt m-lt m-fin*, *rule-format*, *OF* ‹*m* < *Suc i*› ‹*Suc i* < *length cpt*›]
have ‹*fst* (*cpt* ! *Suc i*) = *fin*› **.**
from *all-fin-after-fin*′′[*OF cpt m-lt m-fin*, *rule-format*, *OF* ‹*m* < *Suc* (*Suc i*)› ‹*Suc* (*Suc i*) < *length cpt*›]
have ‹*fst* (*cpt* ! *Suc* (*Suc i*)) = *fin*› **.**
from *cpt-assume* **show** *?thesis*
apply(*simp add*: *assume-def conv1 conv2*)
apply(*drule conjunct2*)
apply(*erule allE*[**where** *x*=‹*Suc i*›])
using ‹*Suc* (*Suc i*) < *length cpt*› ‹*fst* (*cpt* ! *Suc i*) = *fin*› ‹*fst* (*cpt* ! *Suc* (*Suc i*)) = *fin*› *rely1* **by** *auto*
qed
qed
next
show ‹*snd* (*hd* (*snd* (*split cpt*) @ *drop* (*Suc m*) *cpt*)) ∈ *pre2*›
apply(*subst* ‹*?cpt2* = (*Q*,*S0*) # *tl ?cpt2*›)

```
      apply simp
      using ‹S0∈pre› pre by blast
  next
    fix i
    assume ‹Suc i < length ?cpt2 + (length cpt − Suc m)›
     with ‹length ?cpt2 = m› Suc-leI[OF m-lt] have ‹Suc (Suc i) < length cpt›
by linarith
    then have ‹Suc i < length cpt› by simp
    assume ‹fst (?cpt2'!i) = fst (?cpt2'!Suc i)›
    show ‹(snd (?cpt2'!i), snd (?cpt2'!Suc i)) ∈ rely2›
    proof(cases ‹Suc i < length ?cpt2›)
      case True
      from True have conv1: ‹?cpt2'!i = ?cpt2!i›
        by (simp add: Suc-lessD nth-append)
      from True have conv2: ‹?cpt2'!Suc i = ?cpt2!Suc i›
        by (simp add: nth-append)
      from ‹fst (?cpt2'!i) = fst (?cpt2'!Suc i)› conv1 conv2
      have ‹?cpt2!i −e→ ?cpt2!Suc i› by simp
      have ‹(snd (snd (split cpt) ! i), snd (snd (split cpt) ! Suc i)) ∈ rely2›
        using join-sound-aux3-2[OF cpt-from-assume valid1 valid2 pre rely1 rely2
True ‹?cpt2!i −e→ ?cpt2!Suc i›] rely2 by blast
      then show ?thesis
        by (simp add: conv1 conv2)
    next
      case False
      then have Suc-i-ge: ‹Suc i ≥ length ?cpt2› by simp
      show ?thesis
      proof(cases ‹Suc i = length ?cpt2›)
        case True
        then have ‹i < length ?cpt2› by linarith
        from cpt2-shorter True have ‹Suc i < length cpt› by simp
        from True ‹length ?cpt2 = m› have ‹Suc i = m› by simp
        with m' have ‹i = m'› by simp
        with ‹fst (cpt!m') = fin ⋈ fin› have ‹fst (cpt!i) = fin ⋈ fin› by simp
        from ‹Suc i < length ?cpt2 + (length cpt − Suc m)› ‹Suc i = m› ‹length
?cpt2 = m›
        have ‹Suc m < length cpt› by simp
        from ‹Suc i = m› m-fin have ‹fst (cpt!Suc i) = fin› by simp
        have conv1: ‹snd (?cpt2' ! i) = snd (cpt ! Suc i)›
        proof−
              have ‹snd (?cpt2'!i) = snd (?cpt2!i)› using True by (simp add:
nth-append)
          moreover have ‹snd (?cpt2!i) = snd (cpt!i)›
            using split-same-state2[OF ‹i < length ?cpt2›] .
          moreover have ‹snd (cpt!i) = snd (cpt!Suc i)›
          proof−
            from ctran-or-etran[OF cpt ‹Suc i < length cpt›] ‹fst (cpt!i) = fin ⋈
fin› ‹fst (cpt!Suc i) = fin›
            have ‹(cpt ! i, cpt ! Suc i) ∈ estran Γ› by fastforce
```

then show *?thesis*
apply(*subst* (*asm*) *surjective-pairing*[*of* ‹*cpt*!*i*›])
apply(*subst* (*asm*) *surjective-pairing*[*of* ‹*cpt*!*Suc i*›])
apply(*simp add*: ‹*fst* (*cpt*!*i*) = *fin* ⋈ *fin*› ‹*fst* (*cpt*!*Suc i*) = *fin*› *estran-def*)
apply(*erule exE*)
apply(*erule estran-p.cases*, *auto*)
done
qed
ultimately show *?thesis* **by** *simp*
qed
have *conv2*: ‹*snd* (*?cpt2′* ! *Suc i*) = *snd* (*cpt* ! *Suc* (*Suc i*))›
apply(*simp add*: *nth-append True*)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
apply(*simp add*: ‹*length ?cpt2* = *m*›)
done
have ‹(*snd* (*cpt* ! *Suc i*), *snd* (*cpt* ! *Suc* (*Suc i*))) ∈ *rely*›
proof−
have ‹*m*<*Suc m*› **by** *simp*
from *all-fin-after-fin″*[*OF cpt m-lt m-fin*, *rule-format*, *OF this* ‹*Suc m* < *length cpt*›]
have *Suc-m-fin*: ‹*fst* (*cpt* ! *Suc m*) = *fin*› **.**
from *cpt-assume* **show** *?thesis*
apply(*simp add*: *assume-def*)
apply(*drule conjunct2*)
apply(*erule allE*[**where** *x*=*m*])
using ‹*Suc m* < *length cpt*› *m-fin Suc-m-fin* ‹*Suc i* = *m*› **by** *argo*
qed
then show *?thesis*
apply(*simp add*: *conv1 conv2*) **using** *rely2* **by** *blast*
next
case *False*
with *Suc-i-ge* **have** *Suc-i-gt*: ‹*Suc i* > *length ?cpt2*› **by** *linarith*
with ‹*length ?cpt2* = *m*› **have** ‹¬ *i* < *m*› **by** *simp*
then have ‹*m* < *Suc i*› **by** *simp*
then have ‹*m* < *Suc* (*Suc i*)› **by** *simp*
have *conv1*: ‹*?cpt2′*!*i* = *cpt*!*Suc i*›
apply(*simp add*: *nth-append Suc-i-gt* ‹*length ?cpt2* = *m*› ‹¬ *i* < *m*›)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
using ‹¬*i*<*m*› **by** *simp*
have *conv2*: ‹*?cpt2′*!*Suc i* = *cpt*!*Suc*(*Suc i*)›
using *Suc-i-gt* **apply**(*simp add*: *nth-append*)
apply(*subst nth-drop*) **apply**(*rule Suc-leI*[*OF m-lt*])
by (*simp add*: ‹*length ?cpt2* = *m*›)
from *all-fin-after-fin″*[*OF cpt m-lt m-fin*, *rule-format*, *OF* ‹*m* < *Suc i*› ‹*Suc i* < *length cpt*›]
have ‹*fst* (*cpt* ! *Suc i*) = *fin*› **.**
from *all-fin-after-fin″*[*OF cpt m-lt m-fin*, *rule-format*, *OF* ‹*m* < *Suc* (*Suc i*)› ‹*Suc* (*Suc i*) < *length cpt*›]

have ‹*fst* (*cpt* ! *Suc* (*Suc* *i*)) = *fin*› **.**
from *cpt-assume* **show** *?thesis*
apply(*simp add*: *assume-def conv1 conv2*)
apply(*drule conjunct2*)
apply(*erule allE*[**where** *x*=‹*Suc i*›])
using ‹*Suc* (*Suc i*) < *length cpt*› ‹*fst* (*cpt* ! *Suc i*) = *fin*› ‹*fst* (*cpt* ! *Suc* (*Suc i*)) = *fin*› *rely2* **by** *auto*
qed
qed
qed

from *cpt′-from cpt′-assume valid1 valid2*
have
commit1: ‹*?cpt1′* ∈ *commit* (*estran* Γ) {*fin*} *guar1 post1*› **and**
commit2: ‹*?cpt2′* ∈ *commit* (*estran* Γ) {*fin*} *guar2 post2*› **by** *blast+*

from *ctran-or-etran*[*OF cpt Suc-m′-lt*] ‹*fst* (*cpt*!*m′*) = *fin* ⋈ *fin*› ‹*fst* (*cpt*!*Suc* *m′*) = *fin*›
have ‹(*cpt* ! *m′*, *cpt* ! *Suc m′*) ∈ *estran* Γ› **by** *fastforce*
then have ‹*snd* (*cpt*!*m′*) = *snd* (*cpt*!*m*)›
apply(*subst* ‹*m* = *Suc m′*›)
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*insert* ‹*fst* (*cpt*!*m′*) = *fin* ⋈ *fin*›)
apply(*insert* ‹*fst* (*cpt*!*Suc m′*) = *fin*›)
apply(*erule estran-p.cases*, *auto*)
done
have *last-conv1*: ‹*last ?cpt1′* = *last cpt*›
proof(*cases* ‹*Suc m* = *length cpt*›)
case *True*
then have ‹*m* = *length cpt* − *1*› **by** *linarith*
have ‹*snd* (*last ?cpt1*) = *snd* (*cpt* ! *m′*)›
apply(*simp add*: ‹*last ?cpt1* = *?cpt1* ! *m′*›)
by (*rule split-same-state1*[*OF* ‹*m′* < *length ?cpt1*›])
moreover have ‹*cpt*!*m* = *last cpt*›
apply(*subst last-conv-nth*[*OF* ‹*cpt*≠[]›])
using ‹*m* = *length cpt* − *1*› **by** *simp*
ultimately have ‹*snd* (*last ?cpt1*) = *snd* (*last cpt*)› **using** ‹*snd* (*cpt*!*m′*) = *snd* (*cpt*!*m*)› **by** *argo*
with ‹*fst* (*last ?cpt1*) = *fin*› ‹*fst* (*last cpt*) = *fin*› **show** *?thesis*
apply(*simp add*: *True*)
using *surjective-pairing* **by** *metis*
next
case *False*
with ‹*m* < *length cpt*› **have** ‹*Suc m* < *length cpt*› **by** *linarith*
then show *?thesis* **by** *simp*
qed

have *last-conv2*: ‹*last ?cpt2′* = *last cpt*›

proof(*cases* ‹*Suc m* = *length cpt*›)
case *True*
then have ‹*m* = *length cpt* − *1*› **by** *linarith*
have ‹*snd* (*last ?cpt2*) = *snd* (*cpt* ! *m*′)›
apply(*simp add*: ‹*last ?cpt2* = *?cpt2* ! *m*′›)
by (*rule split-same-state2*[*OF* ‹*m*′ < *length ?cpt2*›])
moreover have ‹*cpt*!*m* = *last cpt*›
apply(*subst last-conv-nth*[*OF* ‹*cpt*≠[]›])
using ‹*m* = *length cpt* − *1*› **by** *simp*
ultimately have ‹*snd* (*last ?cpt2*) = *snd* (*last cpt*)› **using** ‹*snd* (*cpt*!*m*′) = *snd* (*cpt*!*m*)› **by** *argo*
with ‹*fst* (*last ?cpt2*) = *fin*› ‹*fst* (*last cpt*) = *fin*› **show** *?thesis*
apply(*simp add*: *True*)
using *surjective-pairing* **by** *metis*
next
case *False*
with ‹*m* < *length cpt*› **have** ‹*Suc m* < *length cpt*› **by** *linarith*
then show *?thesis* **by** *simp*
qed

from *commit1 commit2*
show *?thesis* **apply**(*simp add*: *commit-def*)
apply(*drule conjunct2*)
apply(*drule conjunct2*)
using *last-conv1 last-conv2* **by** *argo*
next
case *False*
have ‹*?cpt1* ∈ *cpts-from* (*estran* Γ) (*P*,*S0*)› **using** *cpt-from-assume split-cpt* **by** *blast*
moreover have ‹*?cpt1* ∈ *assume pre1 rely1*›
proof(*auto simp add*: *assume-def*)
from *split-assume-pre*[*OF cpt fst-hd-cpt cpt-assume*, *THEN conjunct1*] *pre*
show ‹*snd* (*hd* (*fst* (*split cpt*))) ∈ *pre1*› **by** *blast*
next
fix *i*
assume *etran*: ‹*fst* (*fst* (*split cpt*) ! *i*) = *fst* (*fst* (*split cpt*) ! *Suc i*)›
assume *Suc-i-lt1*: ‹*Suc i* < *length* (*fst* (*split cpt*))›
from *join-sound-aux3-1*[*OF cpt-from-assume valid1 valid2 pre rely1 rely2 Suc-i-lt1*] *etran*
have ‹(*snd* (*fst* (*split cpt*) ! *i*), *snd* (*fst* (*split cpt*) ! *Suc i*)) ∈ *rely* ∪ *guar2*› **by** *force*
then show ‹(*snd* (*fst* (*split cpt*) ! *i*), *snd* (*fst* (*split cpt*) ! *Suc i*)) ∈ *rely1*› **using** *rely1* **by** *blast*
qed
ultimately have *cpt1-commit*: ‹*?cpt1* ∈ *commit* (*estran* Γ) {*fin*} *guar1 post1*› **using** *valid1* **by** *blast*
have ‹*?cpt2* ∈ *cpts-from* (*estran* Γ) (*Q*,*S0*)› **using** *cpt-from-assume split-cpt* **by** *blast*
moreover have ‹*?cpt2* ∈ *assume pre2 rely2*›

proof(*auto simp add*: *assume-def*)
show ⟨*snd* (*hd* (*snd* (*split cpt*))) ∈ *pre2*⟩
using *split-assume-pre*[*OF cpt fst-hd-cpt cpt-assume*] *pre* **by** *blast*
next
fix *i*
assume *etran*: ⟨*fst* (*?cpt2*!*i*) = *fst* (*?cpt2*!*Suc i*)⟩
assume *Suc-i-lt2*: ⟨*Suc i* < *length ?cpt2*⟩
from *join-sound-aux3-2*[*OF cpt-from-assume valid1 valid2 pre rely1 rely2 Suc-i-lt2*] *etran*
have ⟨(*snd* (*snd* (*split cpt*) ! *i*), *snd* (*snd* (*split cpt*) ! *Suc i*)) ∈ *rely* ∪ *guar1*⟩ **by** *force*
then show ⟨(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *rely2*⟩ **using** *rely2* **by** *blast*
qed
ultimately have *cpt2-commit*: ⟨*?cpt2* ∈ *commit* (*estran* Γ) {*fin*} *guar2 post2*⟩ **using** *valid2* **by** *blast*
from *cpt1-commit commit-def* **have**
⟨*fst* (*last ?cpt1*) ∈ {*fin*} ⟶ *snd* (*last ?cpt1*) ∈ *post1*⟩ **by** *fastforce*
moreover from *cpt2-commit commit-def* **have**
⟨*fst* (*last ?cpt2*) ∈ {*fin*} ⟶ *snd* (*last ?cpt2*) ∈ *post2*⟩ **by** *fastforce*
ultimately show ⟨*fst* (*last cpt*) ∈ {*fin*} ⟶ *snd* (*last cpt*) ∈ *post1* ∩ *post2*⟩
using *False* **by** *blast*
qed
qed

lemma *split-length-gt*:
assumes *cpt*: ⟨*cpt* ∈ *cpts* (*estran* Γ)⟩
and *fst-hd-cpt*: ⟨*fst* (*hd cpt*) = *P* ⋈ *Q*⟩
and *i-lt*: ⟨*i* < *length cpt*⟩
and *not-fin*: ⟨*fst* (*cpt*!*i*) ≠ *fin*⟩
shows ⟨*length* (*fst* (*split cpt*)) > *i* ∧ *length* (*snd* (*split cpt*)) > *i*⟩
proof−
from *all-join*[*OF cpt fst-hd-cpt i-lt not-fin*]
have *1*: ⟨∀ *ia*≤*i*. ∃ *P′ Q′*. *fst* (*cpt* ! *ia*) = *P′* ⋈ *Q′*⟩ **.**
from *cpt fst-hd-cpt i-lt not-fin 1*
show *?thesis*
proof(*induct cpt arbitrary*:*P Q i rule*:*split.induct*; *simp*; *case-tac ia*; *simp*)
fix *s Pa Qa ia nat*
fix *rest*
assume *IH*:
⟨⋀*P Q i*.
rest ∈ *cpts* (*estran* Γ) ⟹
fst (*hd rest*) = *P* ⋈ *Q* ⟹
i < *length rest* ⟹
fst (*rest* ! *i*) ≠ *fin* ⟹
∀ *ia*≤*i*. ∃ *P′ Q′*. *fst* (*rest* ! *ia*) = *P′* ⋈ *Q′* ⟹
i < *length* (*fst* (*split rest*)) ∧ *i* < *length* (*snd* (*split rest*))⟩
assume *a1*: ⟨(*Pa* ⋈ *Qa*, *s*) # *rest* ∈ *cpts* (*estran* Γ)⟩
assume *a2*: ⟨*nat* < *length rest*⟩
assume *a3*: ⟨*fst* (*rest* ! *nat*) ≠ *fin*⟩

assume *a4*: ‹∀ *ia*≤*Suc nat*. ∃ *P′ Q′*. *fst* (((*Pa* ⋈ *Qa*, *s*) # *rest*) ! *ia*) = *P′* ⋈ *Q′*›
from *a2* **have** *rest*≠[] **by** *fastforce*
from *cpts-tl*[*OF a1*, *simplified*, *OF* ‹*rest*≠[]›] **have** *1*: ‹*rest* ∈ *cpts* (*estran* Γ)› **.**
from *a4* **have** *5*: ‹∀ *ia*≤*nat*. ∃ *P′ Q′*. *fst* (*rest* ! *ia*) = *P′* ⋈ *Q′*› **by** *auto*
from *a4*[*THEN spec*[**where** *x*=*1*]] **have** ‹∃ *P′ Q′*. *fst* (((*Pa* ⋈ *Qa*, *s*) # *rest*) ! *1*) = *P′* ⋈ *Q′*› **by** *force*
then have ‹∃ *P′ Q′*. *fst* (*hd rest*) = *P′* ⋈ *Q′*›
apply *simp*
apply(*subst hd-conv-nth*) **apply**(*rule* ‹*rest*≠[]›) **apply** *assumption* **done**
then obtain *P′ Q′* **where** *2*: ‹*fst* (*hd rest*) = *P′* ⋈ *Q′*› **by** *blast*
from *IH*[*OF 1 2 a2 a3 5*]
show ‹*nat* < *length* (*fst* (*split rest*)) ∧ *nat* < *length* (*snd* (*split rest*))› **.**
qed
qed

lemma *Join-sound-aux*:
assumes *h1*:
‹Γ ⊨ *P sat*$_e$ [*pre1*, *rely1*, *guar1*, *post1*]›
assumes *h2*:
‹Γ ⊨ *Q sat*$_e$ [*pre2*, *rely2*, *guar2*, *post2*]›
and *rely1*: ‹*rely* ∪ *guar2* ⊆ *rely1*›
and *rely2*: ‹*rely* ∪ *guar1* ⊆ *rely2*›
and *guar-refl*: ‹∀ *s*. (*s*,*s*)∈*guar*›
and *guar*: ‹*guar1* ∪ *guar2* ⊆ *guar*›
shows
‹Γ ⊨ *EJoin P Q sat*$_e$ [*pre1* ∩ *pre2*, *rely*, *guar*, *post1* ∩ *post2*]›
using *h1 h2*
proof(*unfold es-validity-def validity-def*)
let *?pre1* = ‹*lift-state-set pre1*›
let *?pre2* = ‹*lift-state-set pre2*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?rely1* = ‹*lift-state-pair-set rely1*›
let *?rely2* = ‹*lift-state-pair-set rely2*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?guar1* = ‹*lift-state-pair-set guar1*›
let *?guar2* = ‹*lift-state-pair-set guar2*›
let *?post1* = ‹*lift-state-set post1*›
let *?post2* = ‹*lift-state-set post2*›
let *?inter-pre* = ‹*lift-state-set* (*pre1*∩*pre2*)›
let *?inter-post* = ‹*lift-state-set* (*post1*∩*post2*)›

have *rely1′*: ‹*?rely* ∪ *?guar2* ⊆ *?rely1*›
apply *standard*
apply(*simp add*: *lift-state-pair-set-def case-prod-unfold*)
using *rely1* **by** *blast*
have *rely2′*: ‹*?rely* ∪ *?guar1* ⊆ *?rely2*›

apply *standard*
apply(*simp add*: *lift-state-pair-set-def case-prod-unfold*)
using *rely2* **by** *blast*
have *guar-refl'*: ‹∀ *S*. (*S*,*S*)∈*?guar*› **using** *guar-refl lift-state-pair-set-def* **by** *blast*
have *guar'*: ‹*?guar1* ∪ *?guar2* ⊆ *?guar*›
apply *standard*
apply(*simp add*: *lift-state-pair-set-def case-prod-unfold*)
using *guar* **by** *blast*

assume *h1'*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*P*, *s0*) ∩ *assume ?pre1 ?rely1* ⊆ *commit* (*estran* Γ) {*fin*} *?guar1 ?post1*›
assume *h2'*: ‹∀ *s0*. *cpts-from* (*estran* Γ) (*Q*, *s0*) ∩ *assume ?pre2 ?rely2* ⊆ *commit* (*estran* Γ) {*fin*} *?guar2 ?post2*›
show ‹∀ *s0*. *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume ?inter-pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?inter-post*›
proof
fix *s0*
show ‹*cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume ?inter-pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?inter-post*›
proof
fix *cpt*
assume *cpt-from-assume*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*) ∩ *assume ?inter-pre ?rely*›
then have
cpt-from: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*P* ⋈ *Q*, *s0*)› **and**
cpt: ‹*cpt* ∈ *cpts* (*estran* Γ)› **and**
fst-hd-cpt: ‹*fst* (*hd cpt*) = *P* ⋈ *Q*› **and**
cpt-assume: ‹*cpt* ∈ *assume ?inter-pre ?rely*› **by** *auto*
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?inter-post*›
proof−
let *?cpt1* = ‹*fst* (*split cpt*)›
let *?cpt2* = ‹*snd* (*split cpt*)›
from *split-cpt*[*OF cpt-from*, *THEN conjunct1*] **have** *?cpt1* ∈ *cpts-from* (*estran* Γ) (*P*, *s0*) **.**
then have ‹*?cpt1*≠[]› **using** *cpts-nonnil* **by** *auto*
from *split-cpt*[*OF cpt-from*, *THEN conjunct2*] **have** *?cpt2* ∈ *cpts-from* (*estran* Γ) (*Q*, *s0*) **.**
then have ‹*?cpt2*≠[]› **using** *cpts-nonnil* **by** *auto*
from *cpts-nonnil*[*OF cpt*] **have** ‹*cpt*≠[]› **.**
from *join-sound-aux2*[*OF cpt-from-assume h1' h2' - rely1' rely2'*]
have *2*:
‹∀ *i*. *Suc i* < *length ?cpt1* ∧ *Suc i* < *length ?cpt2* ⟶
((*?cpt1* ! *i*, *?cpt1* ! *Suc i*) ∈ *estran* Γ ⟶
(*snd* (*?cpt1* ! *i*), *snd* (*?cpt1* ! *Suc i*)) ∈ *?guar1*) ∧
((*?cpt2* ! *i*, *?cpt2* ! *Suc i*) ∈ *estran* Γ ⟶
(*snd* (*?cpt2* ! *i*), *snd* (*?cpt2* ! *Suc i*)) ∈ *?guar2*)› **unfolding** *lift-state-set-def* **by** *blast*
show *?thesis* **using** *cpt-from-assume*
proof(*auto simp add*: *assume-def commit-def*)

fix *i*
assume *Suc-i-lt*: ‹*Suc i* < *length cpt*›
assume *ctran*: ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ›
show ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?guar*›
proof(*cases* ‹*fst* (*cpt*!*Suc i*) = *fin*›)
case *True*
have ‹*fst* (*cpt* ! *i*) ≠ *fin*› **by** (*rule no-estran-from-fin'*[*OF ctran*])
from *all-join*[*OF cpt fst-hd-cpt Suc-i-lt*[*THEN Suc-lessD*] *this*, *THEN spec*[**where** *x*=*i*]] **have**
‹∃ *P' Q'*. *fst* (*cpt* ! *i*) = *P'* ⋈ *Q'*› **by** *simp*
from *join-sound-aux3a*[*OF ctran this True guar-refl'*] **show** *?thesis* **.**
next
case *False*
from *split-length-gt*[*OF cpt fst-hd-cpt Suc-i-lt False*]
have
Suc-i-lt1: ‹*Suc i* < *length ?cpt1*› **and**
Suc-i-lt2: ‹*Suc i* < *length ?cpt2*› **by** *auto*
from *split-ctran*[*OF cpt fst-hd-cpt False Suc-i-lt ctran*] **have**
(*?cpt1*!*i*, *?cpt1*!*Suc i*) ∈ *estran* Γ ∨
(*?cpt2*!*i*, *?cpt2*!*Suc i*) ∈ *estran* Γ **by** *fast*
then show *?thesis*
proof
assume ‹(*?cpt1* ! *i*, *?cpt1* ! *Suc i*) ∈ *estran* Γ›
with *2 Suc-i-lt1 Suc-i-lt2* **have** ‹(*snd* (*?cpt1*!*i*), *snd* (*?cpt1*!*Suc i*)) ∈ *?guar1*› **by** *blast*
with *split-same-state1*[*OF Suc-i-lt1*[*THEN Suc-lessD*]] *split-same-state1*[*OF Suc-i-lt1*]
have ‹(*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *?guar1*› **by** *argo*
with *guar'* **show** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?guar*› **by** *blast*
next
assume ‹(*?cpt2* ! *i*, *?cpt2* ! *Suc i*) ∈ *estran* Γ›
with *2 Suc-i-lt1 Suc-i-lt2* **have** ‹(*snd* (*?cpt2*!*i*), *snd* (*?cpt2*!*Suc i*)) ∈ *?guar2*› **by** *blast*
with *split-same-state2*[*OF Suc-i-lt2*[*THEN Suc-lessD*]] *split-same-state2*[*OF Suc-i-lt2*]
have ‹(*snd* (*cpt*!*i*), *snd* (*cpt*!*Suc i*)) ∈ *?guar2*› **by** *argo*
with *guar'* **show** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?guar*› **by** *blast*
qed
qed
next
have *1*: ‹*fst* (*last cpt*) = *fin* ⟹ *snd* (*last cpt*) ∈ *?post1*›
using *join-sound-aux5*[*OF cpt-from-assume h1' h2' - rely1' rely2'*] **unfolding** *lift-state-set-def* **by** *fastforce*
have *2*: ‹*fst* (*last cpt*) = *fin* ⟹ *snd* (*last cpt*) ∈ *?post2*›
using *join-sound-aux5*[*OF cpt-from-assume h1' h2' - rely1' rely2'*] **unfolding** *lift-state-set-def* **by** *fastforce*
from *1 2*
show ‹*fst* (*last cpt*) = *fin* ⟹ *snd* (*last cpt*) ∈ *lift-state-set* (*post1* ∩ *post2*)›

```
          by (simp add: lift-state-set-def case-prod-unfold)
        qed
      qed
    qed
  qed
qed

lemma post-after-fin:
  ‹(fin, s) # cs ∈ cpts (estran Γ) ⟹
   (fin, s) # cs ∈ assume pre rely ⟹
   s ∈ post ⟹
   stable post rely ⟹
   snd (last ((fin, s) # cs)) ∈ post›
proof−
  assume 1: ‹(fin, s) # cs ∈ cpts (estran Γ)›
  assume asm: ‹(fin, s) # cs ∈ assume pre rely›
  assume ‹s ∈ post›
  assume stable: ‹stable post rely›
  obtain cpt where cpt: ‹cpt = (fin, s) # cs› by simp
  with asm have ‹cpt ∈ assume pre rely› by simp
  have all-etran: ‹∀ i. Suc i < length cpt ⟶ cpt!i −e→ cpt!Suc i›
    apply(rule allI)
    apply(case-tac i; simp)
    using cpt all-fin-after-fin[OF 1] by simp+
  from asm have all-rely: ‹∀ i. Suc i < length cpt ⟶ (snd (cpt!i), snd (cpt!Suc
i))∈rely›
    apply (auto simp add: assume-def)
    using all-etran by (simp add: cpt)
  from cpt have fst-hd-cpt: ‹fst (hd cpt) = fin› by simp
  have aux: ‹∀ i. i < length cpt ⟶ snd (cpt!i) ∈ post›
    apply(rule allI)
    apply(induct-tac i)
    using cpt apply simp apply (rule ‹s∈post›)
    apply clarify
  proof−
    fix n
    assume h: ‹n < length cpt ⟶ snd (cpt ! n) ∈ post›
    assume lt: ‹Suc n < length cpt›
    with h have ‹snd (cpt!n) ∈ post› by fastforce
   moreover have ‹(snd (cpt!n), snd(cpt!Suc n))∈rely› using all-rely lt by simp
    ultimately show ‹snd (cpt!Suc n) ∈ post› using stable stable-def by fast
  qed
  then have ‹snd (last cpt) ∈ post›
    apply(subst last-conv-nth)
    using cpt apply simp
    using aux[THEN spec[where x=‹length cpt−1›]] cpt by force
  then show ?thesis using cpt by simp
qed
```

lemma *unlift-seq-assume*:
‹*map* (*lift-seq-esconf Q*) ((*P*, *s*) # *cs*) ∈ *assume pre rely* ⟹ (*P*,*s*)#*cs* ∈ *assume pre rely*›
apply(*auto simp add*: *assume-def lift-seq-esconf-def case-prod-unfold*)
apply(*erule-tac x*=*i* **in** *allE*)
apply *auto*
apply (*metis* (*no-types*, *lifting*) *Suc-diff-1 Suc-lessD fst-conv linorder-neqE-nat nth-Cons′ nth-map zero-order*(*3*))
by (*metis* (*no-types*, *lifting*) *Suc-diff-1 Suc-lessD linorder-neqE-nat nth-Cons′ nth-map snd-conv zero-order*(*3*))

lemma *lift-seq-commit-aux*:
‹((*P NEXT Q*, *S*), *fst c NEXT Q*, *snd c*) ∈ *estran* Γ ⟹ ((*P*, *S*), *c*) ∈ *estran* Γ›
apply(*simp add*: *estran-def*, *erule exE*)
apply(*erule estran-p.cases*, *auto*)
using *surjective-pairing* **apply** *metis*
using *seq-neq2* **by** *fast*

lemma *nth-length-last*:
‹((*P*, *s*) # *cs* @ *cs′*) ! *length cs* = *last* ((*P*, *s*) # *cs*)›
by (*induct cs*) *auto*

lemma *while-sound-aux1*:
‹(*Q*,*t*)#*cs′* ∈ *commit* (*estran* Γ) {*fin*} *guar post* ⟹
(*P*,*s*)#*cs* ∈ *commit* (*estran* Γ) {*f*} *guar p* ⟹
(*last* ((*P*,*s*)#*cs*), (*Q*,*t*)) ∈ *estran* Γ ⟹
snd (*last* ((*P*,*s*)#*cs*)) = *t* ⟹
∀ *s*. (*s*,*s*)∈*guar* ⟹
(*P*,*s*) # *cs* @ (*Q*,*t*) # *cs′* ∈ *commit* (*estran* Γ) {*fin*} *guar post*›
proof−
assume *commit2*: ‹(*Q*,*t*)#*cs′* ∈ *commit* (*estran* Γ) {*fin*} *guar post*›
assume *commit1*: ‹(*P*,*s*)#*cs* ∈ *commit* (*estran* Γ) {*f*} *guar p*›
assume *tran*: ‹(*last* ((*P*,*s*)#*cs*), (*Q*,*t*)) ∈ *estran* Γ›
assume *last-state1*: ‹*snd* (*last* ((*P*,*s*)#*cs*)) = *t*›
assume *guar-refl*: ‹∀ *s*. (*s*,*s*)∈*guar*›
show ‹(*P*,*s*) # *cs* @ (*Q*,*t*) # *cs′* ∈ *commit* (*estran* Γ) {*fin*} *guar post*›
apply(*auto simp add*: *commit-def*)
apply(*case-tac* ‹*i*<*length cs*›)
apply *simp*
using *commit1* **apply**(*simp add*: *commit-def*)
apply *clarify*
apply(*erule-tac x*=*i* **in** *allE*)
apply (*smt append-is-Nil-conv butlast.simps*(*2*) *butlast-snoc length-Cons less-SucI nth-butlast*)
apply(*subgoal-tac* ‹*i* = *length cs*›)
prefer *2*
apply *linarith*

```
      apply(thin-tac ‹i < Suc (length cs)›)
      apply(thin-tac ‹¬ i < length cs›)
      apply simp
      apply(thin-tac ‹i = length cs›)
    apply(unfold nth-length-last)
    using tran last-state1 guar-refl apply simp using guar-refl apply blast
    using commit2 apply(simp add: commit-def)
      apply(case-tac ‹i<length cs›)
       apply simp
    using commit1 apply(simp add: commit-def)
    apply clarify
      apply(erule-tac x=i in allE)
    apply (metis (no-types, lifting) Suc-diff-1 Suc-lessD linorder-neqE-nat nth-Cons'
nth-append zero-order(3))
     apply(case-tac ‹i = length cs›)
      apply simp
    apply(unfold nth-length-last)
    using tran last-state1 guar-refl apply simp using guar-refl apply blast
      apply(subgoal-tac ‹i > length cs›)
       prefer 2
       apply linarith
    apply(thin-tac ‹¬ i < length cs›)
     apply(thin-tac ‹i ≠ length cs›)
     apply(case-tac i; simp)
    apply(rename-tac i')
    using commit2 apply(simp add: commit-def)
     apply(subgoal-tac ‹∃j. i' = length cs + j›)
      prefer 2
    using le-Suc-ex apply simp
    apply(erule exE)
     apply simp
    apply clarify
     apply(erule-tac x=j in allE)
   apply (metis (no-types, hide-lams) add-Suc-right nth-Cons-Suc nth-append-length-plus)
    using commit2 apply(simp add: commit-def)
    done
qed

lemma while-sound-aux2:
  assumes ‹stable post rely›
    and ‹s ∈ post›
    and ‹∀i. Suc i < length ((P,s)#cs) ⟶ ((P,s)#cs)!i −e→ ((P,s)#cs)!Suc i›
    and ‹∀i. Suc i < length ((P,s)#cs) ⟶ ((P,s)#cs)!i −e→ ((P,s)#cs)!Suc i
⟶ (snd(((P,s)#cs)!i), snd(((P,s)#cs)!Suc i))∈rely›
  shows ‹snd (last ((P,s)#cs)) ∈ post›
  using assms(2−4)
proof(induct cs arbitrary:P s)
  case Nil
  then show ?case by simp
```

next
  case (*Cons c cs*)
  obtain P' s' **where** *c*: ‹$c=(P',s')$› **by** *fastforce*
  have *1*: ‹$s'\in post$›
  proof−
    have *rely*: ‹$(s,s')\in rely$›
      using *Cons*(*3*)[*THEN spec*[**where** *x*=*0*]] *Cons*(*4*)[*THEN spec*[**where** *x*=*0*]] *c*
      by (*simp add*: *assume-def*)
    show *?thesis* **using** *assms*(*1*) ‹$s\in post$› *rely*
      by (*simp add*: *stable-def*)
  qed
  from *Cons*(*3*) *c*
  have *2*: ‹$\forall i.$ *Suc i* < *length* $((P', s')$ # *cs*) $\longrightarrow$ $((P', s')$ # *cs*) ! *i* −*e*→ $((P', s')$ # *cs*) ! *Suc i*› **by** *fastforce*
  from *Cons*(*4*) *c*
  have *3*: ‹$\forall i.$ *Suc i* < *length* $((P', s')$ # *cs*) $\longrightarrow$ $((P', s')$ # *cs*) ! *i* −*e*→ $((P', s')$ # *cs*) ! *Suc i* $\longrightarrow$ (*snd* $(((P', s')$ # *cs*) ! *i*), *snd* $(((P',s')$ # *cs*) ! *Suc i*)) ∈ *rely*› **by** *fastforce*
  show *?case* **using** *Cons*(*1*)[*OF 1 2 3*] *c* **by** *fastforce*
qed

lemma *seq-tran-inv*:
  assumes ‹$((P\ NEXT\ Q,S), (P'\ NEXT\ Q,T))\in estran\ \Gamma$›
    shows ‹$((P,S), (P',T))\in estran\ \Gamma$›
  using *assms*
  apply (*simp add*: *estran-def*)
  apply(*erule exE*) **apply**(*rule exI*) **apply**(*erule estran-p.cases*, *auto*)
  using *seq-neq2* **by** *blast*

lemma *seq-tran-inv-fin*:
  assumes ‹$((P\ NEXT\ Q,S), (Q,T))\in estran\ \Gamma$›
  shows ‹$((P,S), (fin,T))\in estran\ \Gamma$›
  using *assms*
  apply (*simp add*: *estran-def*)
  apply(*erule exE*) **apply**(*rule exI*) **apply**(*erule estran-p.cases*, *auto*)
  using *seq-neq2*[*symmetric*] **by** *blast*

lemma *lift-seq-commit*:
  assumes ‹$cpt \in commit\ (estran\ \Gamma)\ \{fin\}\ guar\ post$›
    and ‹$cpt\neq[]$›
  shows ‹*map* (*lift-seq-esconf Q*) $cpt \in commit\ (estran\ \Gamma)\ \{fin\}\ guar\ post$›
  using *assms*(*1*)
  apply(*simp add*: *commit-def lift-seq-esconf-def case-prod-unfold*)
  apply(*rule conjI*)
   apply(*rule allI*)
  apply *clarify*
  apply(*erule-tac x*=*i* **in** *allE*)
   apply(*drule seq-tran-inv*)

apply *force*
apply *clarify*
by (*simp add*: *last-map*[*OF* ‹*cpt*≠[]›])

lemma *while-sound-aux3*:
assumes ‹*cs* ∈ *commit* (*estran* Γ) {*fin*} *guar post*›
and ‹*cs*≠[]›
shows ‹*map* (*lift-seq-esconf Q*) *cs* ∈ *commit* (*estran* Γ) {*fin*} *guar post'*›
using *assms*
apply(*auto simp add*: *commit-def lift-seq-esconf-def case-prod-unfold*)
subgoal for *i*
proof−
assume *a*: ‹∀ *i*. *Suc i* < *length cs* ⟶ (*cs* ! *i*, *cs* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*cs* ! *i*), *snd* (*cs* ! *Suc i*)) ∈ *guar*›
assume *1*: ‹*Suc i* < *length cs*›
assume ‹((*fst* (*cs* ! *i*) *NEXT Q*, *snd* (*cs* ! *i*)), *fst* (*cs* ! *Suc i*) *NEXT Q*, *snd* (*cs* ! *Suc i*)) ∈ *estran* Γ›
then have *2*: ‹(*cs* ! *i*, *cs* ! *Suc i*) ∈ *estran* Γ› **using** *seq-tran-inv surjective-pairing* **by** *metis*
from *a*[*rule-format*, *OF 1 2*] **show** *?thesis* **.**
qed
subgoal
proof−
assume *1*: ‹*fst* (*last cs*) ≠ *fin*›
assume *2*: ‹*fst* (*last* (*map* (λ*uu*. (*fst uu NEXT Q*, *snd uu*)) *cs*)) = *fin*›
from *1 2* **have** *False*
by (*metis* (*no-types*, *lifting*) *esys.distinct*(*5*) *fst-conv last-map list.simps*(*8*))
then show *?thesis* **by** *blast*
qed
subgoal for *i*
proof−
assume *a*: ‹∀ *i*. *Suc i* < *length cs* ⟶ (*cs* ! *i*, *cs* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*cs* ! *i*), *snd* (*cs* ! *Suc i*)) ∈ *guar*›
assume *1*: ‹*Suc i* < *length cs*›
assume ‹((*fst* (*cs* ! *i*) *NEXT Q*, *snd* (*cs* ! *i*)), *fst* (*cs* ! *Suc i*) *NEXT Q*, *snd* (*cs* ! *Suc i*)) ∈ *estran* Γ›
then have *2*: ‹(*cs* ! *i*, *cs* ! *Suc i*) ∈ *estran* Γ› **using** *seq-tran-inv surjective-pairing* **by** *metis*
from *a*[*rule-format*, *OF 1 2*] **show** *?thesis* **.**
qed
subgoal
proof−
assume ‹*fst* (*last* (*map* (λ*uu*. (*fst uu NEXT Q*, *snd uu*)) *cs*)) = *fin*›
with ‹*cs*≠[]› **have** *False* **by** (*simp add*: *last-conv-nth*)
then show *?thesis* **by** *blast*
qed
.

lemma *no-fin-in-unfinished*:

assumes ‹cpt ∈ cpts (estran Γ)›
and ‹Γ ⊢ last cpt −es[a]→ c›
shows ‹∀ i. i<length cpt ⟶ fst (cpt!i) ≠ fin›
proof(rule allI, rule impI)
fix i
assume ‹i<length cpt›
show ‹fst (cpt!i) ≠ fin›
proof
assume fin: ‹fst (cpt!i) = fin›
let ?cpt = ‹drop i cpt›
have drop-cpt: ‹?cpt ∈ cpts (estran Γ)› **using** cpts-drop[OF assms(1) ‹i<length cpt›] .
obtain S **where** ‹cpt!i = (fin,S)› **using** surjective-pairing fin **by** metis
have drop-cpt-dest: ‹drop i cpt = (fin,S) # tl (drop i cpt)›
using ‹i<length cpt› ‹cpt!i = (fin,S)›
by (metis cpts-def′ drop-cpt hd-Cons-tl hd-drop-conv-nth)
have ‹(fin,S) # tl (drop i cpt) ∈ cpts (estran Γ)› **using** drop-cpt drop-cpt-dest **by** argo
from all-fin-after-fin[OF this] **have** ‹fst (last cpt) = fin›
by (metis (no-types, lifting) ‹cpt ! i = (fin, S)› ‹i < length cpt› drop-cpt-dest fin last-ConsL last-ConsR last-drop last-in-set)
with assms(2) no-estran-from-fin **show** False
by (metis prod.collapse)
qed
qed

lemma while-sound-aux:
assumes ‹cpt ∈ cpts-es-mod Γ›
and ‹preL = lift-state-set pre›
and ‹relyL = lift-state-pair-set rely›
and ‹guarL = lift-state-pair-set guar›
and ‹postL = lift-state-set post›
and ‹pre ∩ − b ⊆ post›
and ‹∀ S0. cpts-from (estran Γ) (P,S0) ∩ assume (lift-state-set (pre∩b)) relyL ⊆ commit (estran Γ) {fin} guarL preL›
and ‹∀ s. (s, s) ∈ guar›
and ‹stable pre rely›
and ‹stable post rely›
shows ‹∀ S cs. cpt = (EWhile b P, S)#cs ⟶ cpt ∈ assume preL relyL ⟶ cpt ∈ commit (estran Γ) {fin} guarL postL›
using assms
proof(induct)
case (CptsModOne P s x)
then show ?case **by** (simp add: commit-def)
next
case (CptsModEnv P t y cs s x)
have 1: ‹∀ P s t. ((P, s), P, t) ∉ estran Γ› **using** no-estran-to-self′ **by** blast
have 2: ‹stable preL relyL› **using** stable-lift[OF ‹stable pre rely›] CptsModEnv(3,4) **by** simp

```
  show ?case
    apply clarify
    apply(rule commit-Cons-env)
     apply(rule 1)
    apply(insert CptsModEnv(2)[OF CptsModEnv(3−11)])
    apply clarify
    apply(erule allE[where x=‹(t,y)›])
    apply(erule allE[where x=cs])
    apply(drule assume-tl-comp[OF - 2])
    by blast
next
  case (CptsModAnon P s Q t x cs)
  then show ?case by simp
next
  case (CptsModAnon-fin P s Q t x cs)
  then show ?case by simp
next
  case (CptsModBasic P e s y x k cs)
  then show ?case by simp
next
  case (CptsModAtom P e s t x cs)
  then show ?case by simp
next
  case (CptsModSeq P s x a Q t y R cs)
  then show ?case by simp
next
  case (CptsModSeq-fin P s x a t y Q cs)
  then show ?case by simp
next
  case (CptsModChc1 P s x a Q t y cs R)
  then show ?case by simp
next
  case (CptsModChc2 P s x a Q t y cs R)
  then show ?case by simp
next
  case (CptsModJoin1 P s x a Q t y R cs)
  then show ?case by simp
next
  case (CptsModJoin2 P s x a Q t y R cs)
  then show ?case by simp
next
  case (CptsModJoin-fin t y cs)
  then show ?case by simp
next
  case (CptsModWhileTMore s b1 P1 x cs a t y cs')
  show ?case
  proof(rule allI, rule allI, clarify)
    assume ‹P1=P› ‹b1=b›
     assume a: ‹(EWhile b P, s, x) # map (lift-seq-esconf (EWhile b P)) ((P, s,
```

x) # *cs*) @ (*EWhile b P*, *t*, *y*) # *cs'* ∈ *assume preL relyL*›

let *?part1* = ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*)›
have *part2-assume*: ‹(*EWhile b P*, *t*, *y*) # *cs'* ∈ *assume preL relyL*›
proof(*simp add*: *assume-def*, *rule conjI*)
  let *?c* = ‹(*P1*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]›
  have ‹*?c* ∈ *cpts-from* (*estran* Γ) (*P1*,*s*,*x*) ∩ *assume* (*lift-state-set* (*pre*∩*b*)) *relyL*›
  proof
    show ‹(*P1*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)] ∈ *cpts-from* (*estran* Γ) (*P1*, *s*, *x*)›
    proof(*simp*)
      from *CptsModWhileTMore*(*3*) **have** *tran*: ‹(*last* ((*P1*, *s*, *x*) # *cs*), (*fin*, *t*, *y*)) ∈ *estran* Γ›
        apply(*simp only*: *estran-def*) **by** *blast*
      from *cpts-snoc-comp*[*OF CptsModWhileTMore*(*2*) *tran*]
      show ‹*?c* ∈ *cpts* (*estran* Γ)› **by** *simp*
    qed
  next
    from *a*
    show ‹(*P1*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)] ∈ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
    proof(*auto simp add*: *assume-def*)
      assume ‹(*s*, *x*) ∈ *preL*›
      then show ‹(*s*, *x*) ∈ *lift-state-set* (*pre* ∩ *b*)›
        using ‹*preL* = *lift-state-set pre*› ‹*s*∈*b1*›
        by (*simp add*: *lift-state-set-def* ‹*b1*=*b*›)
    next
      fix *i*
      assume *a2*[*rule-format*]: ‹∀ *i*<*Suc* (*Suc* (*length cs* + *length cs'*)).
          fst (((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs'*) ! *i*) =
        fst (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs'*) ! *i*) ⟶
          (*snd* (((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs'*) ! *i*),
          snd (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs'*) ! *i*)) ∈ *relyL*›
      let *?j* = ‹*Suc i*›
      assume *i-lt*: ‹*i* < *Suc* (*length cs*)›
      assume *etran*: ‹*fst* (((*P1*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*) = *fst* ((*cs* @ [(*fin*, *t*, *y*)]) ! *i*)›
      show ‹(*snd* (((*P1*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*), *snd* ((*cs* @ [(*fin*, *t*, *y*)]) ! *i*)) ∈ *relyL*›
      proof(*cases* ‹*i*=*length cs*›)
        case *True*
        from *CptsModWhileTMore*(*3*) **have** *ctran*: ‹(*last* ((*P1*, *s*, *x*) # *cs*), (*fin*,

```
t, y)) ∈ estran Γ›
            apply(simp only: estran-def) by blast
          have 1: ‹((P1, s, x) # cs @ [(fin, t, y)]) ! i = last ((P1,s,x)#cs)› using
True by (simp add: nth-length-last)
           have 2: ‹(cs @ [(fin, t, y)]) ! i = (fin, t, y)› using True by (simp add:
nth-append)
           from ctran-imp-not-etran[OF ctran] etran 1 2 have False by force
           then show ?thesis by blast
         next
           case False
           with i-lt have ‹i<length cs› by simp
           have
             ‹fst (map (lift-seq-esconf (EWhile b P)) ((P,s,x)#cs) ! i) =
              fst (map (lift-seq-esconf (EWhile b P)) cs ! i)›
           proof−
             have ∗: ‹i < length ((P1,s,x)#cs)› using ‹i<length cs› by simp
             have ∗∗: ‹i < length ((P,s,x)#cs)› using ‹i<length cs› by simp
             have ‹(((P1, s, x) # cs) @ [(fin, t, y)]) ! i = ((P1,s,x)#cs) ! i›
               using ∗ apply(simp only: nth-append) by simp
             then have eq1: ‹((P1, s, x) # cs @ [(fin, t, y)]) ! i = ((P1,s,x)#cs)
! i› by simp
             have eq2: ‹(cs @ [(fin, t, y)]) ! i = cs!i›
               using ‹i<length cs› by (simp add: nth-append)
             show ?thesis
               apply(simp only: nth-map[OF ∗∗] nth-map[OF ‹i<length cs›])
           using etran apply(simp add: eq1 eq2 lift-seq-esconf-def case-prod-unfold)
               using ‹P1=P› by simp
           qed
           then have
             ‹fst ((map (lift-seq-esconf (EWhile b P)) ((P,s,x)#cs) @ (EWhile b P,
t, y) # cs') ! i) =
               fst ((map (lift-seq-esconf (EWhile b P)) cs @ (EWhile b P, t, y) #
cs') ! i)›
             by (metis (no-types, lifting) One-nat-def ‹i < length cs› add.commute
i-lt length-map list.size(4) nth-append plus-1-eq-Suc)
           then have 2:
               ‹fst (((EWhile b P, s, x) # (P NEXT EWhile b P, s, x) # map
(lift-seq-esconf (EWhile b P)) cs @ (EWhile b P, t, y) # cs') ! ?j) =
              fst (((P NEXT EWhile b P, s, x) # map (lift-seq-esconf (EWhile b
P)) cs @ (EWhile b P, t, y) # cs') ! ?j)›
             by simp
            have 1: ‹?j < Suc (Suc (length cs + length cs'))› using ‹i<length cs›
by simp
           from a2[OF 1 2] have rely:
               ‹(snd (((EWhile b P, s, x) # (P NEXT EWhile b P, s, x) # map
(lift-seq-esconf (EWhile b P)) cs @ (EWhile b P, t, y) # cs') ! Suc i),
   snd (((P NEXT EWhile b P, s, x) # map (lift-seq-esconf (EWhile b P)) cs @
(EWhile b P, t, y) # cs') ! Suc i))
```

```
∈ relyL› .
        have eq1: ‹snd (((EWhile b P, s, x) # (P NEXT EWhile b P, s, x) #
map (lift-seq-esconf (EWhile b P)) cs @ (EWhile b P, t, y) # cs') ! Suc i) =
snd (((P1, s, x) # cs @ [(fin, t, y)]) ! i)›
        proof-
          have **: ‹i < length ((P,s,x)#cs)› using ‹i<length cs› by simp
          have ‹snd ((map (lift-seq-esconf (EWhile b P)) ((P,s,x)#cs)) ! i) =
snd (((P1, s, x) # cs) ! i)›
            apply(subst nth-map[OF **])
            by (simp add: lift-seq-esconf-def case-prod-unfold ‹P1=P›)
          then have ‹snd ((map (lift-seq-esconf (EWhile b P)) ((P,s,x)#cs) @
((EWhile b P, t, y) # cs')) ! i) = snd ((((P1, s, x) # cs)@[(fin,t,y)]) ! i)›
            apply-
            apply(subst nth-append) apply(subst nth-append)
            using ‹i<length cs› by simp
          then show ?thesis by simp
        qed
        have eq2: ‹snd (((P NEXT EWhile b P, s, x) # map (lift-seq-esconf
(EWhile b P)) cs @ (EWhile b P, t, y) # cs') ! Suc i) =
snd ((cs @ [(fin, t, y)]) ! i)›
        proof-
          have ‹snd ((map (lift-seq-esconf (EWhile b P)) cs) ! i) = snd (cs ! i)›
            apply(subst nth-map[OF ‹i<length cs›])
            by (simp add: lift-seq-esconf-def case-prod-unfold ‹P1=P›)
          then have ‹snd ((map (lift-seq-esconf (EWhile b P)) cs @ ((EWhile b
P, t, y) # cs')) ! i) = snd ((cs@[(fin,t,y)]) ! i)›
            apply-
            apply(subst nth-append) apply(subst nth-append)
            using ‹i<length cs› by simp
          then show ?thesis by simp
        qed
        from rely show ?thesis by (simp only: eq1 eq2)
      qed
    qed
  qed
  with CptsModWhileTMore(11) ‹P1=P› have ‹?c ∈ commit (estran Γ) {fin}
guarL preL› by blast
  then show ‹(t,y)∈preL› by (simp add: commit-def)
next
  show ‹∀i<length cs'. fst (((EWhile b P, t, y) # cs') ! i) = fst (cs' ! i) ⟶
(snd (((EWhile b P, t, y) # cs') ! i), snd (cs' ! i)) ∈ relyL›
    apply(rule allI)
    using a apply(auto simp add: assume-def)
    apply(erule-tac x=‹Suc(Suc(length cs)) + i› in allE)
    subgoal for i
    proof-
      assume h[rule-format]:
        ‹Suc (Suc (length cs)) + i < Suc (Suc (length cs + length cs')) ⟶
  fst (((EWhile b P, s, x) # (P NEXT EWhile b P, s, x) # map (lift-seq-esconf
```

(*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)) =
fst (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)) ⟶
(*snd* (((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)),
snd (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*))) ∈ *relyL*›

assume *i-lt*: ‹*i* < *length cs′*›
assume *etran*: ‹*fst* (((*EWhile b P*, *t*, *y*) # *cs′*) ! *i*) = *fst* (*cs′* ! *i*)›
have *eq1*:
‹((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*) =
((*EWhile b P*, *t*, *y*) # *cs′*) ! *i*›
by (*metis* (*no-types*, *lifting*) *Cons-eq-appendI One-nat-def add.commute length-map list.size*(*4*) *nth-append-length-plus plus-1-eq-Suc*)
have *eq2*:
‹((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*) =
cs′!*i*›
by (*metis* (*no-types*, *lifting*) *Cons-eq-appendI One-nat-def add.commute add-Suc-shift length-map list.size*(*4*) *nth-Cons-Suc nth-append-length-plus plus-1-eq-Suc*)
from *i-lt* **have** *i-lt′*: ‹*Suc* (*Suc* (*length cs*)) + *i* < *Suc* (*Suc* (*length cs* + *length cs′*))› **by** *simp*
from *etran* **have** *etran′*:
‹*fst* (((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)) =
fst (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*))›
using *eq1 eq2* **by** *simp*
from *h*[*OF i-lt′ etran′*] **have**
‹(*snd* (((*EWhile b P*, *s*, *x*) # (*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)),
snd (((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *cs′*) ! (*Suc* (*Suc* (*length cs*)) + *i*)))
∈ *relyL*› **.**
then show *?thesis*
using *eq1 eq2* **by** *simp*
qed
done
qed
show ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) @ (*EWhile b P*, *t*, *y*) # *cs′* ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
proof−
from *CptsModWhileTMore*(*5*)[*OF CptsModWhileTMore*(*6*−*14*), *rule-format*, *of* ‹(*t*,*y*)› *cs′*] ‹*P1*=*P*› ‹*b1*=*b*› *part2-assume*
have *part2-commit*: ‹(*EWhile b P*, *t*, *y*) # *cs′* ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*› **by** *simp*

have *part1-commit*: ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*›
proof−
have *1*: ‹(*P*,*s*,*x*)#*cs* ∈ *cpts-from* (*estran* Γ) (*P*,*s*,*x*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
proof
show ‹(*P*, *s*, *x*) # *cs* ∈ *cpts-from* (*estran* Γ) (*P*, *s*, *x*)›
proof(*simp*)
show ‹(*P*,*s*,*x*)#*cs* ∈ *cpts* (*estran* Γ)›
using *CptsModWhileTMore*(*2*) ‹*P1*=*P*› **by** *simp*
qed
next
from *assume-tl-env*[*OF a*[*simplified*]] *assume-appendD*
have ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*› **by** *simp*
from *unlift-seq-assume*[*OF this*] **have** ‹(*P*, *s*, *x*) # *cs* ∈ *assume preL relyL*› .
then show ‹(*P*, *s*, *x*) # *cs* ∈ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*› **using** ‹*s*∈*b1*›
by (*auto simp add*: *assume-def lift-state-set-def* ‹*preL* = *lift-state-set pre*› ‹*b1*=*b*›)
qed
from ‹∀ *s*. (*s*, *s*) ∈ *guar*› ‹*guarL* = *lift-state-pair-set guar*› **have** ‹∀ *S*. (*S*,*S*)∈*guarL*›
using *lift-state-pair-set-def* **by** *blast*
from *CptsModWhileTMore*(*11*) *1* **have** ‹(*P*, *s*, *x*) # *cs* ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*› **by** *blast*
from *lift-seq-commit*[*OF this*]
have *2*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*› **by** *blast*
have ‹*P*≠*fin*›
proof
assume ‹*P*=*fin*›
with ‹*P1*=*P*› *CptsModWhileTMore*(*2*) **have** ‹(*fin*, *s*, *x*) # *cs* ∈ *cpts* (*estran* Γ)› **by** *simp*
from *all-fin-after-fin*[*OF this*] **have** ‹*fst* (*last* ((*fin*,*s*,*x*)#*cs*)) = *fin*› **by** *simp*
with *CptsModWhileTMore*(*3*) *no-estran-from-fin* **show** *False*
by (*metis* ‹*P* = *fin*› ‹*P1* = *P*› *prod.collapse*)
qed
show *?thesis*
apply *simp*
apply(*rule commit-Cons-comp*)
apply(*rule 2*[*simplified*])
apply(*simp add*: *estran-def*)
apply(*rule exI*)
apply(*rule EWhileT*)
using ‹*s*∈*b1*› **apply**(*simp add*: ‹*b1*=*b*›)
apply(*rule* ‹*P*≠*fin*›)

using ‹∀ *S*. (*S*,*S*)∈*guarL*› **by** *blast*
qed
have *guar*: ‹(*snd* (*last* ((*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*))), *snd* (*EWhile b P*, *t*, *y*)) ∈ *guarL*›
proof−
from *CptsModWhileTMore*(*3*)
have *tran*: ‹(*last* ((*P1*, *s*, *x*) # *cs*), (*fin*, *t*, *y*)) ∈ *estran* Γ›
apply(*simp only*: *estran-def*) **by** *blast*
thm *CptsModWhileTMore*
have *1*: ‹(*P*,*s*,*x*)#*cs*@[(*fin*,*t*,*y*)] ∈ *cpts-from* (*estran* Γ) (*P*,*s*,*x*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
proof
show ‹(*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)] ∈ *cpts-from* (*estran* Γ) (*P*, *s*, *x*)›
proof(*simp*)
show ‹(*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)] ∈ *cpts* (*estran* Γ)›
using *CptsModWhileTMore*(*2*) **apply**(*auto simp add*: ‹*P1*=*P*› *cpts-def'*)
apply(*erule-tac x*=*i* **in** *allE*)
apply(*case-tac* ‹*i*=*length cs*›; *simp*)
using *tran* ‹*P1*=*P*› **apply**(*simp add*: *nth-length-last*)
by (*metis* (*no-types*, *lifting*) *Cons-eq-appendI One-nat-def add.commute less-antisym list.size*(*4*) *nth-append plus-1-eq-Suc*)
qed
next
have *1*: ‹*fst* (((*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *length cs*) ≠ *fst* ((*cs* @ [(*fin*, *t*, *y*)]) ! *length cs*)›
apply(*subst append-Cons*[*symmetric*])
apply(*subst nth-append*)
apply *simp*
using *no-fin-in-unfinished*[*OF CptsModWhileTMore*(*2*,*3*)] ‹*P1*=*P*› **by** *simp*
from *a* **have** ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) @ (*EWhile b P*, *t*, *y*) # *cs'* ∈ *assume preL relyL*›
using *assume-tl-env* **by** *fastforce*
then have ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*›
using *assume-appendD* **by** *fastforce*
then have ‹((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*›
using *unlift-seq-assume* **by** *fast*
then show ‹(*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)] ∈ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
apply(*auto simp add*: *assume-def*)
using ‹*s*∈*b1*› **apply**(*simp add*: *lift-state-set-def* ‹*preL* = *lift-state-set pre*› ‹*b1*=*b*›)
apply(*case-tac* ‹*i*=*length cs*›)
using *1* **apply** *blast*
apply(*erule-tac x*=*i* **in** *allE*)
apply(*subst append-Cons*[*symmetric*])
apply(*subst nth-append*) **apply**(*subst nth-append*)
apply *simp*

apply(*subst*(*asm*) *append-Cons*[*symmetric*])
apply(*subst*(*asm*) *nth-append*) **apply**(*subst*(*asm*) *nth-append*)
apply *simp*
done
qed
with *CptsModWhileTMore*(*11*) **have** ‹(*P*,*s*,*x*)#*cs*@[(*fin*,*t*,*y*)] ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*› **by** *blast*
then show *?thesis*
apply(*auto simp add*: *commit-def*)
using *tran* ‹*P1*=*P*› **apply** *simp*
apply(*erule allE*[**where** *x*=‹*length cs*›])
using *tran* **by** (*simp add*: *nth-append last-map lift-seq-esconf-def case-prod-unfold last-conv-nth*)
qed
have ‹((*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*)) @ (*EWhile b P*, *t*, *y*) # *cs′* ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
using *commit-append*[*OF part1-commit guar part2-commit*] **.**
then show *?thesis* **by** *simp*
qed
qed
next
case (*CptsModWhileTOnePartial s b1 P1 x cs*)
have *guar-refl′*: ‹∀ *S*. (*S*,*S*)∈*guarL*›
using ‹∀ *s*. (*s*,*s*)∈*guar*› ‹*guarL* = *lift-state-pair-set guar*› *lift-state-pair-set-def* **by** *auto*
show *?case*
proof(*rule allI*, *rule allI*, *clarify*)
assume ‹*P1*=*P*› ‹*b1*=*b*›
assume *a*: ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*›
have *1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
proof−
have ‹((*P*, *s*, *x*) # *cs*) ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*›
proof−
have ‹((*P*, *s*, *x*) # *cs*) ∈ *cpts-from* (*estran* Γ) (*P*, *s*, *x*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
proof
show ‹(*P*, *s*, *x*) # *cs* ∈ *cpts-from* (*estran* Γ) (*P*, *s*, *x*)› **using** ‹(*P1*, *s*, *x*) # *cs* ∈ *cpts* (*estran* Γ)› ‹*P1*=*P*› **by** *simp*
next
show ‹(*P*, *s*, *x*) # *cs* ∈ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
proof−
from *a* **have** ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*›
by (*auto simp add*: *assume-def*)
from *unlift-seq-assume*[*OF this*] **have** ‹((*P*, *s*, *x*) # *cs*) ∈ *assume preL relyL*› **.**
then show *?thesis*

proof(*auto simp add*: *assume-def lift-state-set-def* ‹*preL* = *lift-state-set pre*›)
show ‹*s*∈*b*› **using** ‹*s*∈*b1*› ‹*b1*=*b*› **by** *simp*
qed
qed
qed
with ‹ ∀ *S0*. *cpts-from* (*estran* Γ) (*P*, *S0*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL* ⊆ *commit* (*estran* Γ) {*fin*} *guarL preL*›
show *?thesis* **by** *blast*
qed
then show *?thesis* **using** *while-sound-aux3* **by** *blast*
qed
show ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
apply(*auto simp add*: *commit-def*)
using *guar-refl′* **apply** *blast*
apply(*case-tac i*; *simp*)
using *guar-refl′* **apply** *blast*
using *1* **apply**(*simp add*: *commit-def*)
apply(*simp add*: *last-conv-nth lift-seq-esconf-def case-prod-unfold*) **.**
qed
next
case (*CptsModWhileTOneFull s b1 P1 x cs a t y cs′*)
have *guar-refl′*: ‹∀ *S*. (*S*,*S*)∈*guarL*›
using ‹∀ *s*. (*s*,*s*)∈*guar*› ‹*guarL* = *lift-state-pair-set guar*› *lift-state-pair-set-def* **by** *auto*
show *?case*
proof(*rule allI*, *rule allI*, *clarify*)
assume ‹*P1*=*P*› ‹*b1*=*b*›
assume *a*: ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) @ *map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs′*) ∈ *assume preL relyL*›
have *1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) @ *map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs′*)
∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
proof−
have *1*: ‹((*P*, *s*, *x*) # *cs*) @ ((*fin*, *t*, *y*) # *cs′*) ∈ *commit* (*estran* Γ) {*fin*} *guarL preL*›
proof−
let *?c* = ‹((*P*, *s*, *x*) # *cs*) @ ((*fin*, *t*, *y*) # *cs′*)›
have ‹*?c* ∈ *cpts-from* (*estran* Γ) (*P*,*s*,*x*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
proof
show ‹((*P*, *s*, *x*) # *cs*) @ (*fin*, *t*, *y*) # *cs′* ∈ *cpts-from* (*estran* Γ) (*P*, *s*, *x*)›
proof(*simp*)
note *part1* = *CptsModWhileTOneFull*(*2*)
from *CptsModWhileTOneFull*(*4*) *cpts-es-mod-equiv*
have *part2*: ‹(*fin*, *t*, *y*) # *cs′* ∈ *cpts* (*estran* Γ)› **by** *blast*

from *CptsModWhileTOneFull*(*3*)
have *tran*: ‹(*last* ((*P1*, *s*, *x*) # *cs*), (*fin*, *t*, *y*)) ∈ *estran* Γ›
apply(*subst estran-def*) **by** *blast*
show ‹(*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs′* ∈ *cpts* (*estran* Γ)›
using *cpts-append-comp*[*OF part1 part2*] *tran* ‹*P1*=*P*› **by** *force*
qed
next
from *assume-appendD*[*OF assume-tl-env*[*OF a*[*simplified*]]]
have ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*) ∈ *assume preL relyL*› **by** *simp*
from *unlift-seq-assume*[*OF this*] **have** *part1*: ‹(*P*, *s*, *x*) # *cs* ∈ *assume preL relyL*› **.**
have *part2*: ‹∀ *i*. *Suc i* < *length* ((*fin*,*t*,*y*)#*cs′*) ⟶ (*snd* (((*fin*,*t*,*y*)#*cs′*)!*i*), *snd* (((*fin*,*t*,*y*)#*cs′*)!*Suc i*)) ∈ *relyL*›
proof−
from *CptsModWhileTOneFull*(*4*) *cpts-es-mod-equiv*
have *part2-cpt*: ‹(*fin*, *t*, *y*) # *cs′* ∈ *cpts* (*estran* Γ)› **by** *blast*
let *?c2* = ‹*map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs′*)›
from *assume-appendD2*[*OF a*[*simplified append-Cons*[*symmetric*]]]
have *1*: ‹∀ *i*. *Suc i* < *length ?c2* ⟶ (*snd* (*?c2*!*i*), *snd* (*?c2*!*Suc i*))∈*relyL*›
apply(*auto simp add*: *assume-def case-prod-unfold*)
apply(*erule-tac x*=*i* **in** *allE*)
by (*simp add*: *nth-Cons′*)
show *?thesis*
proof(*rule allI*, *rule impI*)
fix *i*
assume *a1*: ‹*Suc i* < *length* ((*fin*, *t*, *y*) # *cs′*)›
then have ‹*i*<*length cs′*› **by** *simp*
from *1* **have** ‹∀ *i*. *i* < *length cs′* ⟶
(*snd* (*map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs′*) ! *i*), *snd* (*map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs′*) ! *Suc i*)) ∈ *relyL*›
by *simp*
from *this*[*rule-format*, *OF* ‹*i*<*length cs′*›]
show ‹(*snd* (((*fin*, *t*, *y*) # *cs′*) ! *i*), *snd* (((*fin*, *t*, *y*) # *cs′*) ! *Suc i*)) ∈ *relyL*›
apply(*simp only*: *nth-map*[*OF* ‹*i*<*length cs′*›] *nth-map*[*OF a1*[*THEN Suc-lessD*]] *nth-map*[*OF a1*] *case-prod-unfold*)
by *simp*
qed
qed
from *CptsModWhileTOneFull*(*3*)
have *tran*: ‹(*last* ((*P1*, *s*, *x*) # *cs*), (*fin*, *t*, *y*)) ∈ *estran* Γ›
apply(*subst estran-def*) **by** *blast*
from *assume-append*[*OF part1*] *part2 ctran-imp-not-etran*[*OF tran*[*simplified* ‹*P1*=*P*›]]
have ‹((*P*, *s*, *x*) # *cs*) @ (*fin*,*t*,*y*) # *cs′* ∈ *assume preL relyL*› **by** *blast*
then show ‹((*P*, *s*, *x*) # *cs*) @ (*fin*, *t*, *y*) # *cs′* ∈ *assume* (*lift-state-set* (*pre* ∩ *b*)) *relyL*›
using ‹*s*∈*b1*› **by** (*simp add*: *assume-def lift-state-set-def* ‹*preL* =

lift-state-set pre› ‹*b1*=*b*›)
qed
with *CptsModWhileTOneFull*(*11*) **show** *?thesis* **by** *blast*
qed
show *?thesis*
apply(*auto simp add*: *commit-def*)
using *1* **apply**(*simp add*: *commit-def*)
apply *clarify*
apply(*erule-tac x*=*i* **in** *allE*)
subgoal for *i*
proof−
assume *a*: ‹*i* < *Suc* (*length cs*) ⟶ (((*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*, (*cs* @ [(*fin*, *t*, *y*)]) ! *i*) ∈ *estran* Γ ⟶ (*snd* (((*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*), *snd* ((*cs* @ [(*fin*, *t*, *y*)]) ! *i*)) ∈ *guarL*›
assume *1*: ‹*i* < *Suc* (*length cs*)›
assume *a3*: ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]) ! *i*, (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]) ! *i*)
∈ *estran* Γ›
have *2*: ‹(((*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*, (*cs* @ [(*fin*, *t*, *y*)]) ! *i*) ∈ *estran* Γ›
proof−
from *a3* **have** *a3'*: ‹((*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*) @ [(*EWhile b P*, *t*, *y*)]) ! *i*, (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]) ! *i*)
∈ *estran* Γ› **by** *simp*
have *eq1*:
‹(*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*) @ [(*EWhile b P*, *t*, *y*)]) ! *i* =
(*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*)) ! *i*›
using *1* **by** (*simp add*: *nth-append del*: *list.map*)
show *?thesis*
proof(*cases* ‹*i*=*length cs*›)
case *True*
let *?c* = ‹((*P*, *s*, *x*) # *cs*) ! *length cs*›
from *a3'* **show** *?thesis*
apply(*simp add*: *eq1 nth-append True del*: *list.map*)
apply(*subst append-Cons*[*symmetric*])
apply(*simp add*: *nth-append del*: *append-Cons*)
apply(*simp add*: *lift-seq-esconf-def case-prod-unfold*)
apply(*simp add*: *estran-def*)
apply(*erule exE*)
apply(*rule exI*)
apply(*erule estran-p.cases*, *auto*)[]
apply(*subst surjective-pairing*[*of ?c*])
by *auto*
next
case *False*
with ‹*i*<*Suc* (*length cs*)› **have** ‹*i* < *length cs*› **by** *simp*

have *eq2*:
‹(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]) ! *i* = (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) ! *i*›
using ‹*i*<*length cs*› **by** (*simp add*: *nth-append*)
from *a3'* **show** *?thesis*
using ‹*i*<*length cs*› **apply**(*simp add*: *eq1 eq2 nth-append del*: *list.map*)
apply(*subst append-Cons*[*symmetric*])
apply(*simp add*: *nth-append del*: *append-Cons*)
apply(*simp add*: *lift-seq-esconf-def case-prod-unfold*)
using *seq-tran-inv* **by** *fastforce*
qed
qed
from *a*[*rule-format*, *OF 1 2*] **have**
‹(*snd* (((*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]) ! *i*), *snd* ((*cs* @ [(*fin*, *t*, *y*)]) ! *i*)) ∈ *guarL*› **.**
then have
‹(((*s*,*x*) # *map snd cs* @ [(*t*,*y*)])!*i*, (*map snd cs* @ [(*t*,*y*)])!*i*) ∈ *guarL*›
using *1 nth-map*[*of i* ‹(*P*, *s*, *x*) # *cs* @ [(*fin*, *t*, *y*)]› *snd*] *nth-map*[*of i* ‹*cs* @ [(*fin*, *t*, *y*)]› *snd*] **by** *simp*
then have
‹(((*s*,*x*) # *map snd* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) @ [(*t*,*y*)])!*i*, (*map snd* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) @ [(*t*,*y*)])!*i*) ∈ *guarL*›
proof−
assume *a*: ‹(((*s*, *x*) # *map snd cs* @ [(*t*, *y*)]) ! *i*, (*map snd cs* @ [(*t*, *y*)]) ! *i*) ∈ *guarL*›
have *aux*[*rule-format*]: ‹∀ *f*. *map* (*snd* ∘ (λ*uu*. (*f uu*, *snd uu*))) *cs* = *map snd cs*› **by** *simp*
from *a* **show** *?thesis* **by** (*simp add*: *lift-seq-esconf-def case-prod-unfold aux*)
qed
then show *?thesis*
using *1 nth-map*[*of i* ‹(*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]› *snd*]
nth-map[*of i* ‹*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ [(*EWhile b P*, *t*, *y*)]› *snd*]
by *simp*
qed
using *1* **apply**(*simp add*: *commit-def*)
apply *clarify*
apply(*erule-tac x*=*i* **in** *allE*)
subgoal for *i*
proof−
assume *a*: ‹*i* < *Suc* (*length cs* + *length cs'*) ⟶ (((*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*, (*cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*) ∈ *estran* Γ ⟶
(*snd* (((*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*), *snd* ((*cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*)) ∈ *guarL*›
assume *1*: ‹*i* < *Suc* (*length cs* + *length cs'*)›
assume ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs'*) ! *i*,

(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*)
∈ *estran* Γ›
then have *2*: ‹(((*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs′*) ! *i*, (*cs* @ (*fin*, *t*, *y*) # *cs′*) ! *i*) ∈ *estran* Γ›
apply(*cases* ‹*i* < *length cs*›; *simp*)
subgoal
proof−
assume *a1*: ‹*i* < *length cs*›
assume *a2*: ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*,
(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*)
∈ *estran* Γ›
have *aux*[*rule-format*]: ‹∀ *x xs y ys*. *i* < *length xs* ⟶ (*x*#*xs*@*y*#*ys*)!*i* = (*x*#*xs*)!*i*›
by (*metis add-diff-cancel-left′ less-SucI less-Suc-eq-0-disj nth-Cons′ nth-append plus-1-eq-Suc*)
from *a1* **have** *a1′*: ‹*i* < *length* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*)› **by** *simp*
have *a2′*: ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs*)!*i*, (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*)!*i*) ∈ *estran* Γ›
proof−
have *1*: ‹((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*
=
((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) ! *i*› **using** *aux*[*OF a1′*] **.**
have *2*: ‹(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i* =
map (*lift-seq-esconf* (*EWhile b P*)) *cs* ! *i*› **using** *a1′* **by** (*simp add*: *nth-append*)
from *a2* **show** *?thesis* **by** (*simp add*: *1 2*)
qed
thm *seq-tran-inv*
have ‹(((*P*, *s*, *x*) # *cs*) ! *i*, *cs* ! *i*) ∈ *estran* Γ›
proof−
from *a2′* **have** *a2″*: ‹((*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*)) ! *i*, *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* ! *i*) ∈ *estran* Γ› **by** *simp*
obtain *P1 S1* **where** *1*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*) ! *i* = (*P1 NEXT EWhile b P*, *S1*)›
proof−
assume *a*: ‹⋀*P1 S1*. *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) ! *i* = (*P1 NEXT EWhile b P*, *S1*) ⟹ *thesis*›
have *a1′*: ‹*i* < *length* ((*P*,*s*,*x*)#*cs*)› **using** *a1* **by** *auto*
show *thesis* **apply**(*rule a*) **apply**(*subst nth-map*[*OF a1′*]) **by** (*simp add*: *lift-seq-esconf-def case-prod-unfold*)
qed
obtain *P2 S2* **where** *2*: ‹*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* ! *i* = (*P2 NEXT EWhile b P*, *S2*)›

proof−
assume *a*: ‹⋀*P2 S2*. *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* ! *i* = (*P2 NEXT EWhile b P*, *S2*) ⟹ *thesis*›
show *thesis* **apply**(*rule a*) **apply**(*subst nth-map*[*OF a1*]) **by** (*simp add*: *lift-seq-esconf-def case-prod-unfold*)
qed
have *tran*: ‹((*P1*,*S1*),(*P2*,*S2*)) ∈ *estran* Γ› **using** *seq-tran-inv a2*″ *1 2* **by** *metis*
have *aux*[*rule-format*]: ‹∀ *Q P S cs i*. *map* (*lift-seq-esconf Q*) *cs* ! *i* = (*P NEXT Q*,*S*) ⟶ *i* < *length cs* ⟶ *cs*!*i* = (*P*,*S*)›
apply(*rule allI*)+ **apply** *clarify* **apply**(*simp add*: *lift-seq-esconf-def case-prod-unfold nth-map*[*OF a1*])
using *surjective-pairing* **by** *metis*
have *3*: ‹((*P*, *s*, *x*) # *cs*) ! *i* = (*P1*,*S1*)› **using** *aux*[*OF 1*] *a1* **by** *auto*
have *4*: ‹*cs*!*i* = (*P2*,*S2*)› **using** *aux*[*OF 2 a1*] **.**
show *?thesis* **using** *tran 3 4* **by** *argo*
qed
moreover have ‹((*P*, *s*, *x*) # *cs*) ! *i* = (((*P*, *s*, *x*) # *cs*) @ (*fin*, *t*, *y*) # *cs*′) ! *i*› **using** *a1* **by** (*simp add*: *aux*)
moreover have ‹(*cs* @ (*fin*, *t*, *y*) # *cs*′) ! *i* = *cs*!*i*› **using** *a1* **by** (*simp add*: *nth-append*)
ultimately show *?thesis* **by** *simp*
qed
apply(*cases* ‹*i* = *length cs*›; *simp*)
subgoal
proof−
assume *a*: ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs*′) ! *length cs*,
(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs*′) ! *length cs*)
∈ *estran* Γ›
have *1*: ‹((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs*′) ! *length cs* = ((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) ! *length cs*›
by (*metis append-Nil2 length-map nth-length-last*)
have *2*: ‹(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs*′) ! *length cs* =
(*EWhile b P*, *t*, *y*)›
by (*metis* (*no-types*, *lifting*) *map-eq-imp-length-eq map-ident nth-append-length*)
from *a* **have** *a*′: ‹(((*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) ! *length cs*, (*EWhile b P*, *t*, *y*)) ∈ *estran* Γ›
by (*simp add*: *1 2*)
obtain *P1 S1* **where** *3*: ‹(*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*)) ! *length cs* = (*P1 NEXT EWhile b P*,*S1*)›
proof−

assume *a*: ‹⋀*P1 S1*. (*map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*,*s*,*x*)#*cs*)) ! *length cs* = (*P1 NEXT EWhile b P*, *S1*) ⟹ *thesis*›
have *1*: ‹*length cs* < *length* ((*P*,*s*,*x*)#*cs*)› **by** *simp*
show *thesis* **apply**(*rule a*) **apply**(*subst nth-map*[*OF 1*]) **by** (*simp add*: *lift-seq-esconf-def case-prod-unfold*)
qed
from *a′ seq-tran-inv-fin 3* **have** ‹((*P1 NEXT EWhile b P*,*S1*),(*EWhile b P*,*t*,*y*))∈*estran* Γ› **by** *auto*
moreover have ‹((*P*,*s*,*x*)#*cs*) ! *length cs* = (*P1*,*S1*)›
proof−
have ∗: ‹*length cs* < *length* ((*P*,*s*,*x*)#*cs*)› **by** *simp*
show *?thesis* **using** *3*
apply(*simp only*: *lift-seq-esconf-def case-prod-unfold*)
apply(*subst* (*asm*) *nth-map*[*OF* ∗])
by *auto*
qed
moreover have ‹((*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs′*) ! *length cs* = ((*P*, *s*, *x*) # *cs*) ! *length cs*›
by (*metis append-Nil2 nth-length-last*)
ultimately show *?thesis* **using** *seq-tran-inv-fin* **by** *metis*
qed
subgoal
proof−
assume *a1*: ‹¬ *i* < *length cs*›
assume *a2*: ‹((*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *Suc 0*),
(*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*)
∈ *estran* Γ›
assume *a3*: ‹*i* ≠ *length cs*›
from *a1 a3* **have** ‹*i*>*length cs*› **by** *simp*
have *1*: ‹((*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *Suc 0*)) =
((*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *Suc 0* − *length cs*)›
by (*metis* (*no-types*, *lifting*) *Suc-pred* ‹*length cs* < *i*› *a1 length-map less-Suc-eq-0-disj less-antisym nth-append*)
have *2*: ‹((*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! *i*) =
((*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *length cs*)›
by (*simp add*: *a1 nth-append*)
from *a2* **have** *a2′*: ‹((((*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *Suc 0* − *length cs*)), (((*EWhile b P*, *t*, *y*) # *map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs′*) ! (*i* − *length cs*))) ∈ *estran* Γ›
by (*simp add*: *1 2*)
note *i-lt* = ‹*i* < *Suc* (*length cs* + *length cs′*)›
obtain *S1* **where** *3*: ‹((*map* (λ(-, *y*). (*EWhile b P*, *y*)) ((*fin*,*t*,*y*)#*cs′*)) ! (*i* − *Suc 0* − *length cs*)) = (*EWhile b P*, *S1*)›
proof−

assume *a*: ‹⋀*S1*. *map* (λ(-, *y*). (*EWhile b P*, *y*)) ((*fin*, *t*, *y*) # *cs'*) ! (*i* − *Suc 0* − *length cs*) = (*EWhile b P*, *S1*) ⟹ *thesis*›
have ∗: ‹*i* − *Suc 0* − *length cs* < *length* ((*fin*,*t*,*y*)#*cs'*)› **using** *i-lt* **by** *simp*
show *thesis* **apply**(*rule a*) **apply**(*subst nth-map*[*OF* ∗]) **by** (*simp add*: *case-prod-unfold*)
qed
obtain *S2* **where** *4*: ‹(*map* (λ(-, *y*). (*EWhile b P*, *y*)) ((*fin*,*t*,*y*)#*cs'*)) ! (*i* − *length cs*) = (*EWhile b P*, *S2*)›
proof−
assume *a*: ‹⋀*S2*. (*map* (λ(-, *y*). (*EWhile b P*, *y*)) ((*fin*, *t*, *y*) # *cs'*)) ! (*i* − *length cs*) = (*EWhile b P*, *S2*) ⟹ *thesis*›
have ∗: ‹*i* − *length cs* < *length* ((*fin*,*t*,*y*)#*cs'*)› **using** *i-lt* **by** *simp*
show *thesis* **apply**(*rule a*) **apply**(*subst nth-map*[*OF* ∗]) **by** (*simp add*: *case-prod-unfold*)
qed
from *no-estran-to-self' a2' 3 4* **have** *False* **by** *fastforce*
then show *?thesis* **by** (*rule FalseE*)
qed
done
from *a*[*rule-format*, *OF 1 2*] **have** ‹(*snd* (((*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*), *snd* ((*cs* @ (*fin*, *t*, *y*) # *cs'*) ! *i*)) ∈ *guarL*› **.**

then have
‹(((*s*,*x*) # *map snd cs* @ (*t*,*y*) # *map snd cs'*)!*i*, (*map snd cs* @ (*t*,*y*) # *map snd cs'*)!*i*) ∈ *guarL*›
using *1 nth-map*[*of i* ‹(*P*, *s*, *x*) # *cs* @ (*fin*, *t*, *y*) # *cs'*› *snd*] *nth-map*[*of i* ‹*cs* @ (*fin*, *t*, *y*) # *cs'*› *snd*] **by** *simp*
then have
‹(((*s*,*x*) # *map snd* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) @ (*t*,*y*) # *map snd* (*map* (λ(-,*S*). (*EWhile b P*, *S*)) *cs'*))!*i*, (*map snd* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) @ (*t*,*y*) # *map snd* (*map* (λ(-,*S*). (*EWhile b P*, *S*)) *cs'*))!*i*) ∈ *guarL*›
proof−
assume ‹(((*s*,*x*) # *map snd cs* @ (*t*,*y*) # *map snd cs'*)!*i*, (*map snd cs* @ (*t*,*y*) # *map snd cs'*)!*i*) ∈ *guarL*›
moreover have ‹*map snd* (*map* (*lift-seq-esconf* (*EWhile b P*)) *cs*) = *map snd cs*› **by** *auto*
moreover have ‹*map snd* (*map* (λ(-, *S*). (*EWhile b P*, *S*)) *cs'*) = *map snd cs'*› **by** *auto*
ultimately show *?thesis* **by** *metis*
qed
then show *?thesis*
using *1 nth-map*[*of i* ‹(*P NEXT EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-,*S*). (*EWhile b P*, *S*)) *cs'*› *snd*]
nth-map[*of i* ‹*map* (*lift-seq-esconf* (*EWhile b P*)) *cs* @ (*EWhile b P*, *t*, *y*) # *map* (λ(-,*S*). (*EWhile b P*, *S*)) *cs'*› *snd*]
by *simp*
qed

apply(*rule FalseE*) **by** (*simp add*: *last-conv-nth case-prod-unfold*)
qed
show ‹(*EWhile b P*, *s*, *x*) # *map* (*lift-seq-esconf* (*EWhile b P*)) ((*P*, *s*, *x*) # *cs*) @ *map* (λ(-, *s*, *x*). (*EWhile b P*, *s*, *x*)) ((*fin*, *t*, *y*) # *cs'*)
∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
apply(*auto simp add*: *commit-def*)
apply(*case-tac i*; *simp*)
using *guar-refl'* **apply** *blast*
using *1* **apply**(*simp add*: *commit-def*)
apply(*case-tac i*; *simp*)
using *1* **apply**(*simp add*: *commit-def*)
using *guar-refl'* **apply** *blast*
using *1* **apply**(*simp add*: *commit-def*)
subgoal
proof−
assume ‹*cs'*≠[]› ‹*fst* (*last* (*map* (λ(-, *y*). (*EWhile b P*, *y*)) *cs'*)) = *fin*›
then have *False* **by** (*simp add*: *last-conv-nth case-prod-unfold*)
then show *?thesis* **by** *blast*
qed.
qed
next
case (*CptsModWhileF s b1 x cs P1*)
have *cpt*: ‹((*fin*, *s*, *x*) # *cs*) ∈ *cpts* (*estran* Γ)› **using** ‹((*fin*, *s*, *x*) # *cs*) ∈ *cpts-es-mod* Γ› *cpts-es-mod-equiv* **by** *blast*

show *?case*
proof(*rule allI*, *rule allI*, *clarify*)
assume ‹*P1*=*P*› ‹*b1*=*b*›
assume *a*: ‹(*EWhile b P*, *s*, *x*) # (*fin*, *s*, *x*) # *cs* ∈ *assume preL relyL*›
then have ‹*s*∈*pre*› **by** (*simp add*: *assume-def lift-state-set-def* ‹*preL* = *lift-state-set pre*›)

show ‹(*EWhile b P*, *s*, *x*) # (*fin*, *s*, *x*) # *cs* ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
proof−
have *1*: ‹(*fin*, *s*, *x*) # *cs* ∈ *commit* (*estran* Γ) {*fin*} *guarL postL*›
proof−
have *1*: ‹(*s*,*x*)∈*postL*›
proof−
have ‹*s*∈*post*› **using** ‹*s*∈*pre*› ‹*pre*∩−*b*⊆*post*› ‹*s*∉*b1*› ‹*b1*=*b*› **by** *blast*
then show *?thesis* **using** ‹*postL* = *lift-state-set post*› **by** (*simp add*: *lift-state-set-def*)
qed
have *guar-refl'*: ‹∀ *S*. (*S*,*S*)∈*guarL*›
using ‹∀ *s*. (*s*,*s*)∈*guar*› ‹*guarL* = *lift-state-pair-set guar*› *lift-state-pair-set-def* **by** *auto*
have *all-etran*: ‹∀ *i*. *Suc i* < *length* ((*fin*, *s*, *x*) # *cs*) ⟶ ((*fin*, *s*, *x*) # *cs*) ! *i* −*e*→ ((*fin*, *s*, *x*) # *cs*) ! *Suc i*›
using *all-etran-from-fin*[*OF cpt*] **by** *blast*

```
      show ?thesis
      proof(auto simp add: commit-def 1)
        fix i
        assume ‹i<length cs›
        assume a: ‹(((fin, s, x) # cs) ! i, cs ! i) ∈ estran Γ›
        have False
        proof−
          from ctran-or-etran[OF cpt] ‹i<length cs› a all-etran
          show False by simp
        qed
        then show ‹(snd (((fin, s, x) # cs) ! i), snd (cs ! i)) ∈ guarL› by blast
      next
        assume ‹cs≠[]›
        thm while-sound-aux2
        show ‹snd (last cs) ∈ postL›
        proof−
         have 1: ‹stable postL relyL› using ‹stable post rely› ‹postL = lift-state-set
post› ‹relyL = lift-state-pair-set rely›
            by (simp add: lift-state-set-def lift-state-pair-set-def stable-def)
          have 2: ‹∀ i. Suc i < length ((fin, s, x) # cs) ⟶
   ((fin, s, x) # cs) ! i −e→ ((fin, s, x) # cs) ! Suc i ⟶ (snd (((fin, s, x) #
cs) ! i), snd (((fin, s, x) # cs) ! Suc i)) ∈ relyL›
            using a
            apply(simp add: assume-def)
            apply(rule allI)
            apply(erule conjE)
            apply(erule-tac x=‹Suc i› in allE)
            by simp
          have ‹snd (last ((fin, s, x) # cs)) ∈ postL› using while-sound-aux2[OF
1 ‹(s,x)∈postL› all-etran 2] .
          then show ?thesis using ‹cs≠[]› by simp
        qed
      qed
    qed
    have 2: ‹((EWhile b P, s, x), (fin, s, x)) ∈ estran Γ›
      apply(simp add: estran-def)
      apply(rule exI)
      apply(rule EWhileF)
      using ‹s∉b1› ‹b1=b› by simp
      from ‹∀ s. (s, s) ∈ guar› ‹guarL = lift-state-pair-set guar› have 3: ‹∀ S.
(S,S)∈guarL›
      using lift-state-pair-set-def by auto
    from commit-Cons-comp[OF 1 2 3[rule-format]] show ?thesis .
  qed
 qed
qed


theorem While-sound:
```

‹⟦ *stable pre rely*; (*pre* ∩ −*b*) ⊆ *post*; *stable post rely*;
Γ ⊨ *P* sat_e [*pre* ∩ *b*, *rely*, *guar*, *pre*]; ∀ *s*. (*s*,*s*)∈*guar* ⟧ ⟹
Γ ⊨ *EWhile b P* sat_e [*pre*, *rely*, *guar*, *post*]›
apply(*unfold es-validity-def validity-def*)
proof−
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›

assume *stable-pre*: ‹*stable pre rely*›
assume *pre-post*: ‹*pre* ∩ −*b* ⊆ *post*›
assume *stable-post*: ‹*stable post rely*›
assume *P-valid*: ‹∀ *S0*. *cpts-from* (*estran* Γ) (*P*, *S0*) ∩ *assume* (*lift-state-set* (*pre* ∩ *b*)) *?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?pre*›
assume *guar-refl*: ‹∀ *s*. (*s*,*s*)∈*guar*›
show ‹∀ *S0*. *cpts-from* (*estran* Γ) (*EWhile b P*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *S0*
show ‹*cpts-from* (*estran* Γ) (*EWhile b P*, *S0*) ∩ *assume ?pre ?rely* ⊆ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
proof
fix *cpt*
assume *cpt-from-assume*: ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EWhile b P*, *S0*) ∩ *assume ?pre ?rely*›
then have *cpt*:
‹*cpt* ∈ *cpts* (*estran* Γ)› **and** *cpt-assume*:
‹*cpt* ∈ *assume ?pre ?rely*› **by** *auto*
from *cpt-from-assume* **have** ‹*cpt* ∈ *cpts-from* (*estran* Γ) (*EWhile b P*, *S0*)› **by** *blast*
then have ‹*hd cpt* = (*EWhile b P*, *S0*)› **by** *simp*
moreover from *cpt cpts-nonnil* **have** ‹*cpt*≠[]› **by** *blast*
ultimately obtain *cs* **where** *1*: ‹*cpt* = (*EWhile b P*, *S0*) # *cs*› **by** (*metis hd-Cons-tl*)
from *cpt cpts-es-mod-equiv* **have** *cpt-mod*:
‹*cpt* ∈ *cpts-es-mod* Γ› **by** *blast*
obtain *preL* :: ‹('*s* × ('*a*,'*b*,'*s*,'*prog*) *ectx*) *set*› **where** *preL*: ‹*preL* = *?pre*› **by** *simp*
obtain *relyL* :: ‹('*s* × ('*a*,'*b*,'*s*,'*prog*) *ectx*) *tran set*› **where** *relyL*: ‹*relyL* = *?rely*› **by** *simp*
obtain *guarL* :: ‹('*s* × ('*a*,'*b*,'*s*,'*prog*) *ectx*) *tran set*› **where** *guarL*: ‹*guarL* = *?guar*› **by** *simp*
obtain *postL* :: ‹('*s* × ('*a*,'*b*,'*s*,'*prog*) *ectx*) *set*› **where** *postL*: ‹*postL* = *?post*› **by** *simp*
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?guar ?post*›
using *while-sound-aux*[*OF cpt-mod preL relyL guarL postL pre-post - guar-refl stable-pre stable-post*, *THEN spec*[**where** *x*=*S0*], *THEN spec*[**where** *x*=*cs*], *rule-format*] *P-valid 1 cpt-assume preL relyL guarL postL* **by** *blast*

qed
qed
qed

lemma *lift-seq-assume*:
⟨$cs \neq [] \Longrightarrow cs \in assume\ pre\ rely \longleftrightarrow lift\text{-}seq\text{-}cpt\ P\ cs \in assume\ pre\ rely$⟩
by (*auto simp add*: *assume-def lift-seq-esconf-def case-prod-unfold hd-map*)

inductive *rghoare-es* :: $'Env \Rightarrow [('l,'k,'s,'prog)\ esys,\ 's\ set,\ ('s \times 's)\ set,\ ('s \times 's)\ set,\ 's\ set] \Rightarrow bool$
($- \vdash -\ sat_e\ [-, -, -, -]\ [60,60,0,0,0,0]\ 45$)
where
Evt-Anon: $\Gamma \vdash P\ sat_p\ [pre, rely, guar, post] \Longrightarrow \Gamma \vdash EAnon\ P\ sat_e\ [pre, rely, guar, post]$

| *Evt-Basic*: $[\![\Gamma \vdash body\ ev\ sat_p\ [pre \cap (guard\ ev), rely, guar, post];$
$stable\ pre\ rely; \forall s.\ (s, s) \in guar]\!] \Longrightarrow \Gamma \vdash EBasic\ ev\ sat_e\ [pre, rely, guar, post]$

| *Evt-Atom*:
⟨$[\![\forall V.\ \Gamma \vdash body\ ev\ sat_p\ [pre \cap guard\ ev \cap \{V\}, Id, UNIV, \{s.\ (V,s) \in guar\} \cap post];$
$stable\ pre\ rely; stable\ post\ rely\ ]\!] \Longrightarrow$
$\Gamma \vdash EAtom\ ev\ sat_e\ [pre, rely, guar, post]$⟩

| *Evt-Seq*:

⟨$[\![\Gamma \vdash es1\ sat_e\ [pre, rely, guar, mid]; \Gamma \vdash es2\ sat_e\ [mid, rely, guar, post]\ ]\!] \Longrightarrow$
$\Gamma \vdash ESeq\ es1\ es2\ sat_e\ [pre, rely, guar, post]$⟩

| *Evt-conseq*: $[\![\ pre \subseteq pre'; rely \subseteq rely'; guar' \subseteq guar; post' \subseteq post;$
$\Gamma \vdash ev\ sat_e\ [pre', rely', guar', post']\ ]\!]$
$\Longrightarrow \Gamma \vdash ev\ sat_e\ [pre, rely, guar, post]$

| *Evt-Choice*:
⟨$\Gamma \vdash P\ sat_e\ [pre, rely, guar, post] \Longrightarrow$
$\Gamma \vdash Q\ sat_e\ [pre, rely, guar, post] \Longrightarrow$
$\Gamma \vdash P\ OR\ Q\ sat_e\ [pre, rely, guar, post]$⟩

| *Evt-Join*:
⟨$\Gamma \vdash P\ sat_e\ [pre1, rely1, guar1, post1] \Longrightarrow$
$\Gamma \vdash Q\ sat_e\ [pre2, rely2, guar2, post2] \Longrightarrow$
$pre \subseteq pre1 \cap pre2 \Longrightarrow$
$rely \cup guar2 \subseteq rely1 \Longrightarrow$
$rely \cup guar1 \subseteq rely2 \Longrightarrow$
$\forall s.\ (s,s) \in guar \Longrightarrow$
$guar1 \cup guar2 \subseteq guar \Longrightarrow$
$post1 \cap post2 \subseteq post \Longrightarrow$
$\Gamma \vdash EJoin\ P\ Q\ sat_e\ [pre, rely, guar, post]$⟩

| *Evt-While*:
‹⟦ *stable pre rely*; (*pre* ∩ −*b*) ⊆ *post*; *stable post rely*;
Γ ⊢ *P* sat_e [*pre* ∩ *b*, *rely*, *guar*, *pre*]; ∀ *s*. (*s*,*s*)∈*guar* ⟧ ⟹
Γ ⊢ *EWhile b P* sat_e [*pre*, *rely*, *guar*, *post*]›

theorem *rghoare-es-sound*:
assumes *h*: Γ ⊢ *es* sat_e [*pre*, *rely*, *guar*, *post*]
shows Γ ⊨ *es* sat_e [*pre*, *rely*, *guar*, *post*]
using *h*
proof(*induct*)
case (*Evt-Anon* Γ *P pre rely guar post*)
then show *?case* **by**(*rule Anon-sound*)
next
case (*Evt-Basic* Γ *ev pre rely guar post*)
then show *?case* **using** *Basic-sound* **by** *blast*
next
case (*Evt-Atom* Γ *ev pre guar post rely*)
then show *?case* **using** *Atom-sound* **by** *blast*
next
case (*Evt-Seq* Γ *es1 pre rely guar mid es2 post*)
then show *?case* **using** *Seq-sound* **by** *blast*
next
case (*Evt-conseq pre pre′ rely rely′ guar′ guar post′ post* Γ *ev*)
then show *?case* **using** *conseq-sound* **by** *blast*
next
case *Evt-Choice*
then show *?case* **using** *Choice-sound* **by** *blast*
next
case (*Evt-Join* Γ *P pre1 rely1 guar1 post1 Q pre2 rely2 guar2 post2 pre rely guar post*)
then show *?case* **apply**−
apply(*rule conseq-sound*[*of* Γ - ‹*pre1*∩*pre2*› *rely guar* ‹*post1*∩*post2*›])
using *Join-sound-aux* **apply** *blast*
by *auto*
next
case *Evt-While*
then show *?case* **using** *While-sound* **by** *blast*
qed

inductive *rghoare-pes* :: [*′Env*, *′k* ⇒ ((*′l*,*′k*,*′s*,*′prog*)*esys*,*′s*) *rgformula*, *′s set*, (*′s* × *′s*) *set*, (*′s* × *′s*) *set*, *′s set*] ⇒ *bool*
(- ⊢ - SAT_e [-, -, -, -] [*60*,*0*,*0*,*0*,*0*,*0*] *45*)
where
Par:
⟦ ∀ *k*. Γ ⊢ *Com* (*prgf k*) sat_e [*Pre* (*prgf k*), *Rely* (*prgf k*), *Guar* (*prgf k*), *Post* (*prgf k*)];
∀ *k*. *pre* ⊆ *Pre* (*prgf k*);

∀ *k*. *rely* ⊆ *Rely* (*prgf k*);
∀ *k j*. *j*≠*k* ⟶ *Guar* (*prgf j*) ⊆ *Rely* (*prgf k*);
∀ *k*. *Guar* (*prgf k*) ⊆ *guar*;
(⋂ *k*. (*Post* (*prgf k*))) ⊆ *post* ⟧ ⟹
Γ ⊢ *prgf* SAT_e [*pre*, *rely*, *guar*, *post*]

lemma *Par-conseq*:
⟦ *pre* ⊆ *pre'*; *rely* ⊆ *rely'*; *guar'* ⊆ *guar*; *post'* ⊆ *post*;
Γ ⊢ *prgf* SAT_e [*pre'*, *rely'*, *guar'*, *post'*] ⟧ ⟹
Γ ⊢ *prgf* SAT_e [*pre*, *rely*, *guar*, *post*]
apply(*erule rghoare-pes.cases*, *auto*)
apply(*rule Par*)
apply *auto*
by *blast*+

lemma *par-sound-aux2*:
assumes *pc*: ‹*pc* ∈ *cpts-from* (*pestran* Γ) ((λ*k*. *Com* (*prgf k*)), *S0*) ∩ *assume pre rely*›
and *valid*: ‹∀ *k S0*. *cpts-from* (*estran* Γ) (*Com* (*prgf k*), *S0*) ∩ *assume pre* (*Rely* (*prgf k*)) ⊆ *commit* (*estran* Γ) {*fin*} (*Guar* (*prgf k*)) (*Post* (*prgf k*))›
and *rely1*: ‹∀ *k*. *rely* ⊆ *Rely* (*prgf k*)›
and *rely2*: ‹∀ *k k'*. *k'* ≠ *k* ⟶ *Guar* (*prgf k'*) ⊆ *Rely* (*prgf k*)›
and *guar*: ‹∀ *k*. *Guar* (*prgf k*) ⊆ *guar*›
and *conjoin*: ‹*pc* ∝ *cs*›
shows
‹∀ *i k*. *Suc i* < *length pc* ⟶ (*cs k* ! *i*, *cs k* ! *Suc i*) ∈ *estran* Γ ⟶ (*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Guar* (*prgf k*)›
proof(*rule ccontr*, *simp*, *erule exE*)
from *pc* **have** *pc-cpts-from*: ‹*pc* ∈ *cpts-from* (*pestran* Γ) ((λ*k*. *Com* (*prgf k*)), *S0*)› **by** *blast*
then have *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)› **by** *simp*
from *pc* **have** *pc-assume*: ‹*pc* ∈ *assume pre rely*› **by** *blast*
fix *l*
assume ‹*Suc l* < *length pc* ∧ (∃ *k*. (*cs k* ! *l*, *cs k* ! *Suc l*) ∈ *estran* Γ ∧ (*snd* (*cs k* ! *l*), *snd* (*cs k* ! *Suc l*)) ∉ *Guar* (*prgf k*))›
(**is** ‹*?P l*›)
from *exists-least*[*of ?P*, *OF this*] **obtain** *m* **where** *contra*:
‹(*Suc m* < *length pc* ∧ (∃ *k*. (*cs k* ! *m*, *cs k* ! *Suc m*) ∈ *estran* Γ ∧ (*snd* (*cs k* ! *m*), *snd* (*cs k* ! *Suc m*)) ∉ *Guar* (*prgf k*))) ∧
(∀ *i*<*m*. ¬ (*Suc i* < *length pc* ∧ (∃ *k*. (*cs k* ! *i*, *cs k* ! *Suc i*) ∈ *estran* Γ ∧ (*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∉ *Guar* (*prgf k*))))›
by *blast*
then have *Suc-m-lt*: ‹*Suc m* < *length pc*› **by** *argo*
from *contra* **obtain** *k* **where** ‹(*cs k* ! *m*, *cs k* ! *Suc m*) ∈ *estran* Γ ∧ (*snd* (*cs k* ! *m*), *snd* (*cs k* ! *Suc m*)) ∉ *Guar* (*prgf k*)›
by *blast*
then have *ctran*: ‹(*cs k* ! *m*, *cs k* ! *Suc m*) ∈ *estran* Γ› **and** *not-guar*: ‹(*snd* (*cs k* ! *m*), *snd* (*cs k* ! *Suc m*)) ∉ *Guar* (*prgf k*)›
by *auto*

from *contra* **have** ‹∀ i<m. ¬ (Suc i < length pc ∧ (∃ k. (cs k ! i, cs k ! Suc i) ∈ estran Γ ∧ (snd (cs k ! i), snd (cs k ! Suc i)) ∉ Guar (prgf k)))›
  by *argo*
then have *forall-i-lt-m*: ‹∀ i<m. Suc i < length pc ⟶ (∀ k. (cs k ! i, cs k ! Suc i) ∈ estran Γ ⟶ (snd (cs k ! i), snd (cs k ! Suc i)) ∈ Guar (prgf k))›
  by *simp*
from *Suc-m-lt* **have** ‹Suc m < length (cs k)› **using** *conjoin*
  by (*simp add*: *conjoin-def same-length-def*)
let *?c* = ‹take (Suc (Suc m)) (cs k)›
have ‹cs k ∈ cpts-from (estran Γ) (Com (prgf k), S0)› **using** *conjoin-cpt'*[*OF pc-cpts-from conjoin*] **.**
then have *c-from*: ‹?c ∈ cpts-from (estran Γ) (Com (prgf k), S0)›
  by (*metis Zero-not-Suc cpts-from-take*)
have ‹∀ i. Suc i < length ?c ⟶ ?c!i −e→ ?c!Suc i ⟶ (snd (?c!i), snd (?c!Suc i)) ∈ rely ∪ (⋃j∈{j. j ≠ k}. Guar (prgf j))›
proof(*rule allI*, *rule impI*, *rule impI*)
  fix *i*
  assume *Suc-i-lt'*: ‹Suc i < length ?c›
  then have ‹i≤m› **using** *Suc-m-lt* **by** *simp*
  then have *Suc-i-lt*: ‹Suc i < length pc› **using** *Suc-m-lt* **by** *simp*
  assume *etran'*: ‹?c!i −e→ ?c!Suc i›
  then have *etran*: ‹cs k!i −e→ cs k!Suc i› **using** ‹i≤m› **by** *simp*
  from *conjoin-etran-k*[*OF pc-cpt conjoin Suc-i-lt etran*]
  have ‹(pc!i −e→ pc!Suc i) ∨ (∃ k'. k'≠k ∧ (cs k'!i, cs k'!Suc i) ∈ estran Γ)› **.**
  then show ‹(snd (?c!i), snd (?c!Suc i)) ∈ rely ∪ (⋃j∈{j. j ≠ k}. Guar (prgf j))›
  proof
    assume ‹pc!i −e→ pc!Suc i›
    then have ‹(snd (pc!i), snd (pc!Suc i)) ∈ rely› **using** *pc-assume Suc-i-lt*
      by (*simp add*: *assume-def*)
    then have ‹(snd (cs k!i), snd (cs k!Suc i)) ∈ rely› **using** *conjoin Suc-i-lt*
      by (*simp add*: *conjoin-def same-state-def*)
    then have ‹(snd (?c!i), snd (?c!Suc i)) ∈ rely› **using** ‹i≤m› **by** *simp*
    then show ‹(snd (?c!i), snd (?c!Suc i)) ∈ rely ∪ (⋃j∈{j. j ≠ k}. Guar (prgf j))› **by** *blast*
  next
    assume ‹∃ k'. k' ≠ k ∧ (cs k' ! i, cs k' ! Suc i) ∈ estran Γ›
    then obtain *k'* **where** *k'*: ‹k' ≠ k ∧ (cs k' ! i, cs k' ! Suc i) ∈ estran Γ› **by** *blast*
    then have *ctran-k'*: ‹(cs k' ! i, cs k' ! Suc i) ∈ estran Γ› **by** *argo*
    have ‹(snd (cs k'!i), snd (cs k'!Suc i)) ∈ Guar (prgf k')›
    proof(*cases i=m*)
      case *True*
      with *ctran etran ctran-imp-not-etran* **show** *?thesis* **by** *blast*
    next
      case *False*
      with ‹i≤m› **have** ‹i<m› **by** *linarith*
      with *forall-i-lt-m Suc-i-lt ctran-k'* **show** *?thesis* **by** *blast*
    qed

then have ‹(snd (cs k!i), snd (cs k!Suc i)) ∈ Guar (prgf k')› using conjoin Suc-i-lt
by (simp add: conjoin-def same-state-def)
then have ‹(snd (?c!i), snd (?c!Suc i)) ∈ Guar (prgf k')› using ‹i≤m› by fastforce
then show ‹(snd (?c!i), snd (?c!Suc i)) ∈ rely ∪ (⋃j∈{j. j ≠ k}. Guar (prgf j))›
using k' by blast
qed
qed
moreover have ‹snd (hd ?c) ∈ pre›
proof−
from pc-cpt cpts-nonnil have ‹pc≠[]› by blast
then have length pc ≠ 0 by simp
then have ‹length (cs k) ≠ 0› using conjoin by (simp add: conjoin-def same-length-def)
then have ‹cs k ≠ []› by simp
have ‹snd (hd pc) ∈ pre› using pc-assume by (simp add: assume-def)
then have ‹snd (pc!0) ∈ pre› by (simp add: hd-conv-nth ‹pc≠[]›)
then have ‹snd (cs k ! 0) ∈ pre› using conjoin
by (simp add: conjoin-def same-state-def ‹pc ≠ []›)
then have ‹snd (hd (cs k)) ∈ pre› by (simp add: hd-conv-nth ‹cs k≠[]›)
then show ‹snd (hd ?c) ∈ pre› by simp
qed
ultimately have ‹?c ∈ assume pre (Rely (prgf k))› using rely1 rely2
apply(auto simp add: assume-def) by blast
with c-from have ‹?c ∈ cpts-from (estran Γ) (Com (prgf k), S0) ∩ assume pre (Rely (prgf k))› by blast
with valid have ‹?c ∈ commit (estran Γ) {fin} (Guar (prgf k)) (Post (prgf k))› by blast
then have ‹(snd (?c!m), snd (?c!Suc m)) ∈ Guar (prgf k)›
apply(simp add: commit-def)
apply clarify
apply(erule allE[where x=m])
using ctran ‹Suc m < length (cs k)› by blast
with not-guar ‹Suc m < length (cs k)› show False by simp
qed

lemma par-sound-aux3:
assumes pc: ‹pc ∈ cpts-from (pestran Γ) ((λk. Com (prgf k)), s0) ∩ assume pre rely›
and valid: ‹∀k s0. cpts-from (estran Γ) (Com (prgf k), s0) ∩ assume pre (Rely (prgf k)) ⊆ commit (estran Γ) {fin} (Guar (prgf k)) (Post (prgf k))›
and rely1: ‹∀k. rely ⊆ Rely (prgf k)›
and rely2: ‹∀k k'. k' ≠ k ⟶ Guar (prgf k') ⊆ Rely (prgf k)›
and guar: ‹∀k. Guar (prgf k) ⊆ guar›
and conjoin: ‹pc ∝ cs›
and Suc-i-lt: ‹Suc i < length pc›
and etran: ‹(cs k ! i −e→ cs k ! Suc i)›

shows ‹(*snd* (*cs k*!*i*), *snd* (*cs k*!*Suc i*)) ∈ *Rely* (*prgf k*)›
proof−

from *pc* **have** *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)› **by** *fastforce*
from *conjoin-etran-k*[*OF pc-cpt conjoin Suc-i-lt etran*]
have ‹*pc* ! *i* −*e*→ *pc* ! *Suc i* ∨ (∃ *k'*. *k'* ≠ *k* ∧ (*cs k'* ! *i*, *cs k'* ! *Suc i*) ∈ *estran* Γ)› **.**
then show *?thesis*
proof
assume ‹*pc* ! *i* −*e*→ *pc* ! *Suc i*›
moreover from *pc* **have** ‹*pc* ∈ *assume pre rely*› **by** *blast*
ultimately have ‹(*snd* (*pc*!*i*), *snd* (*pc*!*Suc i*)) ∈ *rely*› **using** *Suc-i-lt*
by (*simp add*: *assume-def*)
with *conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*[*THEN Suc-lessD*]] *conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*] *rely1*
show ‹(*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Rely* (*prgf k*)›
by *auto*
next
assume ‹∃ *k'*. *k'* ≠ *k* ∧ (*cs k'* ! *i*, *cs k'* ! *Suc i*) ∈ *estran* Γ›
then obtain *k''* **where** *k''*: ‹*k''* ≠ *k* ∧ (*cs k''* ! *i*, *cs k''* ! *Suc i*) ∈ *estran* Γ› **by** *blast*
then have ‹(*cs k''* ! *i*, *cs k''* ! *Suc i*) ∈ *estran* Γ› **by** (*rule conjunct2*)
from *par-sound-aux2*[*OF pc valid rely1 rely2 guar conjoin*, *rule-format*, *OF Suc-i-lt*, *OF this*]
have *1*: ‹(*snd* (*cs k''* ! *i*), *snd* (*cs k''* ! *Suc i*)) ∈ *Guar* (*prgf k''*)› **.**
show ‹(*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Rely* (*prgf k*)›
proof−
from *1 conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*[*THEN Suc-lessD*]] *conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*]
have ‹(*snd* (*pc* ! *i*), *snd* (*pc* ! *Suc i*)) ∈ *Guar* (*prgf k''*)› **by** *simp*
with *conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*[*THEN Suc-lessD*]] *conjoin-same-state*[*OF conjoin*, *rule-format*, *OF Suc-i-lt*]
have ‹(*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Guar* (*prgf k''*)› **by** *simp*
moreover from *k''* **have** ‹*k''*≠*k*› **by** (*rule conjunct1*)
ultimately show *?thesis* **using** *rely2*[*rule-format*, *OF* ‹*k''*≠*k*›] **by** *blast*
qed
qed
qed

lemma *par-sound-aux5*:
assumes *pc*: ‹*pc* ∈ *cpts-from* (*pestran* Γ) ((λ*k*. *Com* (*prgf k*)), *s0*) ∩ *assume pre rely*›
and *valid*: ‹∀ *k s0*. *cpts-from* (*estran* Γ) (*Com* (*prgf k*), *s0*) ∩ *assume pre* (*Rely* (*prgf k*)) ⊆ *commit* (*estran* Γ) {*fin*} (*Guar* (*prgf k*)) (*Post* (*prgf k*))›
and *rely1*: ‹∀ *k*. *rely* ⊆ *Rely* (*prgf k*)›
and *rely2*: ‹∀ *k k'*. *k'* ≠ *k* ⟶ *Guar* (*prgf k'*) ⊆ *Rely* (*prgf k*)›
and *guar*: ‹∀ *k*. *Guar* (*prgf k*) ⊆ *guar*›
and *conjoin*: ‹*pc* ∝ *cs*›
and *fin*: ‹*fst* (*last pc*) ∈ *par-fin*›

shows ‹*snd* (*last pc*) ∈ (⋂*k*. *Post* (*prgf k*))›
proof−
have ‹∀*k*. *cs k* ∈ *cpts-from* (*estran* Γ) (*Com* (*prgf k*), *s0*) ∩ *assume pre* (*Rely* (*prgf k*))›
proof
fix *k*
show ‹*cs k* ∈ *cpts-from* (*estran* Γ) (*Com* (*prgf k*), *s0*) ∩ *assume pre* (*Rely* (*prgf k*))›
proof
from *pc* **have** *pc*′: ‹*pc* ∈ *cpts-from* (*pestran* Γ) ((λ*k*. *Com* (*prgf k*)), *s0*)› **by** *blast*
show ‹*cs k* ∈ *cpts-from* (*estran* Γ) (*Com* (*prgf k*), *s0*)›
using *conjoin-cpt*′[*OF pc*′ *conjoin*] **.**
next
show ‹*cs k* ∈ *assume pre* (*Rely* (*prgf k*))›
proof(*auto simp add*: *assume-def*)
from *pc* **have** *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)› **by** *simp*
from *pc* **have** *pc-assume*: ‹*pc* ∈ *assume pre rely*› **by** *blast*
from *pc-cpt cpts-nonnil* **have** ‹*pc*≠[]› **by** *blast*
then have *length pc* ≠ *0* **by** *simp*
then have ‹*length* (*cs k*) ≠ *0*› **using** *conjoin* **by** (*simp add*: *conjoin-def same-length-def*)
then have ‹*cs k* ≠ []› **by** *simp*
have ‹*snd* (*hd pc*) ∈ *pre*› **using** *pc-assume* **by** (*simp add*: *assume-def*)
then have ‹*snd* (*pc*!*0*) ∈ *pre*› **by** (*simp add*: *hd-conv-nth* ‹*pc*≠[]›)
then have ‹*snd* (*cs k* ! *0*) ∈ *pre*› **using** *conjoin*
by (*simp add*: *conjoin-def same-state-def* ‹*pc* ≠ []›)
then show ‹*snd* (*hd* (*cs k*)) ∈ *pre*› **by** (*simp add*: *hd-conv-nth* ‹*cs k*≠[]›)
next
fix *i*
show ‹*Suc i* < *length* (*cs k*) ⟹ *fst* (*cs k* ! *i*) = *fst* (*cs k* ! *Suc i*) ⟹ (*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Rely* (*prgf k*)›
proof−
assume ‹*Suc i* < *length* (*cs k*)›
with *conjoin-same-length*[*OF conjoin*] **have** ‹*Suc i* < *length pc*› **by** *simp*
assume ‹*fst* (*cs k* ! *i*) = *fst* (*cs k* ! *Suc i*)›
then have *etran*: ‹(*cs k* ! *i*) −*e*→ (*cs k* ! *Suc i*)› **by** *simp*
show ‹(*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *Rely* (*prgf k*)›
using *par-sound-aux3*[*OF pc valid rely1 rely2 guar conjoin* ‹*Suc i* < *length pc*› *etran*] **.**
qed
qed
qed
qed
with *valid* **have** *commit*: ‹∀*k*. *cs k* ∈ *commit* (*estran* Γ) {*fin*} (*Guar* (*prgf k*)) (*Post* (*prgf k*))› **by** *blast*
from *pc* **have** *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)› **by** *fastforce*
with *cpts-nonnil* **have** ‹*pc*≠[]› **by** *blast*
have ‹∀*k*. *fst* (*last* (*cs k*)) = *fin*›

```
  proof
    fix k
    from conjoin-cpt[OF pc-cpt conjoin] have ‹cs k ∈ cpts (estran Γ)› .
    with cpts-nonnil have ‹cs k ≠ []› by blast
    from fin have ‹∀k. fst (last pc) k = fin› by blast
   moreover have ‹fst (last pc) k = fst (last (cs k))› using conjoin-same-spec[OF
conjoin]
      apply(subst last-conv-nth)
       apply(rule ‹pc≠[]›)
      apply(subst last-conv-nth)
       apply(rule ‹cs k≠[]›)
      apply(subst conjoin-same-length[OF conjoin, of k])
      apply(erule allE[where x=k])
      apply(erule allE[where x=‹length (cs k) − 1›])
      apply(subst (asm) conjoin-same-length[OF conjoin, of k])
      using ‹cs k ≠ []› by force
     ultimately show ‹fst (last (cs k)) = fin› using fin conjoin-same-spec[OF
conjoin] by simp
  qed
  then have ‹∀k. snd (last (cs k)) ∈ Post (prgf k)› using commit
    by (simp add: commit-def)
  moreover have ‹∀k. snd (last (cs k)) = snd (last pc)›
  proof
    fix k
    from conjoin-cpt[OF pc-cpt conjoin] have ‹cs k ∈ cpts (estran Γ)› .
    with cpts-nonnil have ‹cs k ≠ []› by blast
    show ‹snd (last (cs k)) = snd (last pc)› using conjoin-same-state[OF conjoin]
      apply−
      apply(subst last-conv-nth)
       apply(rule ‹cs k ≠ []›)
      apply(subst last-conv-nth)
       apply(rule ‹pc ≠ []›)
      apply(subst conjoin-same-length[OF conjoin, of k])
      apply(erule allE[where x=k])
      apply(erule allE[where x=‹length (cs k) − 1›])
      apply(subst (asm) conjoin-same-length[OF conjoin, of k])
      using ‹cs k ≠ []› by force
  qed
  ultimately show ?thesis by fastforce
qed

definition ‹split-par pc ≡ λk. map (λ(Ps,s). (Ps k, s)) pc›

lemma split-par-conjoin:
  ‹pc ∈ cpts (pestran Γ) ⟹ pc ∝ split-par pc›
proof(unfold conjoin-def, auto)
  show ‹same-length pc (split-par pc)›
    by (simp add: same-length-def split-par-def)
next
```

```
  show ‹same-state pc (split-par pc)›
    by (simp add: same-state-def split-par-def case-prod-unfold)
next
  show ‹same-spec pc (split-par pc)›
    by (simp add: same-spec-def split-par-def case-prod-unfold)
next
  assume ‹pc ∈ cpts (pestran Γ)›
  then show ‹compat-tran pc (split-par pc)›
  proof(auto simp add: compat-tran-def split-par-def case-prod-unfold)
    fix j
    assume cpt: ‹pc ∈ cpts (pestran Γ)›
    assume Suc-j-lt: ‹Suc j < length pc›
    assume not-etran: ‹fst (pc ! j) ≠ fst (pc ! Suc j)›
    from ctran-or-etran-par[OF cpt Suc-j-lt] not-etran
    have ‹(pc ! j, pc ! Suc j) ∈ pestran Γ› by fastforce
    then show ‹∃ t k Γ. Γ ⊢ pc ! j −pes[t♯k]→ pc ! Suc j›
      by (auto simp add: pestran-def)
  next
    fix j k t Γ'
    assume ctran: ‹Γ' ⊢ pc ! j −pes[t♯k]→ pc ! Suc j›
    then show ‹Γ' ⊢ (fst (pc ! j) k, snd (pc ! j)) −es[t♯k]→ (fst (pc ! Suc j) k,
snd (pc ! Suc j))›
      apply−
      by (erule pestran-p.cases, auto)
  next
    fix j k t Γ' k'
    assume ‹Γ' ⊢ pc ! j −pes[t♯k]→ pc ! Suc j›
    moreover assume ‹k' ≠ k›
    ultimately show ‹fst (pc ! j) k' = fst (pc ! Suc j) k'›
      apply−
      by (erule pestran-p.cases, auto)
  next
    fix j k
    assume cpt: ‹pc ∈ cpts (pestran Γ)›
    assume Suc-j-lt: ‹Suc j < length pc›
    assume ‹fst (pc ! j) k ≠ fst (pc ! Suc j) k›
    then have ‹fst (pc!j) ≠ fst (pc!Suc j)› by force
    with ctran-or-etran-par[OF cpt Suc-j-lt] have ‹(pc ! j, pc ! Suc j) ∈ pestran Γ›
by fastforce
    then show ‹∃ t k Γ. Γ ⊢ pc ! j −pes[t♯k]→ pc ! Suc j› by (auto simp add:
pestran-def)
  next
    fix j k ka t Γ'
    assume ‹Γ' ⊢ pc ! j −pes[t♯ka]→ pc ! Suc j›
    then show ‹Γ' ⊢ (fst (pc ! j) ka, snd (pc ! j)) −es[t♯ka]→ (fst (pc ! Suc j) ka,
snd (pc ! Suc j))›
      apply−
      by (erule pestran-p.cases, auto)
  next
```

fix j k ka t Γ' k'
assume ‹$\Gamma' \vdash pc ! j -pes[t\sharp ka]\rightarrow pc ! Suc j$›
moreover assume ‹$k' \neq ka$›
ultimately show ‹$fst (pc ! j) k' = fst (pc ! Suc j) k'$›
apply−
by (*erule pestran-p.cases*, *auto*)
qed
qed

theorem *par-sound*:
assumes *h*: ‹$\forall k. \Gamma \vdash Com (prgf k) sat_e [Pre (prgf k), Rely (prgf k), Guar (prgf k), Post (prgf k)]$›
assumes *pre*: ‹$\forall k. pre \subseteq Pre (prgf k)$›
assumes *rely1*: ‹$\forall k. rely \subseteq Rely (prgf k)$›
assumes *rely2*: ‹$\forall k j. j \neq k \longrightarrow Guar (prgf j) \subseteq Rely (prgf k)$›
assumes *guar*: ‹$\forall k. Guar (prgf k) \subseteq guar$›
assumes *post*: ‹$(\bigcap k. Post (prgf k)) \subseteq post$›
shows
‹$\Gamma \models par\text{-}com\ prgf\ SAT_e [pre, rely, guar, post]$›
proof(*simp*)
let *?pre* = ‹*lift-state-set pre*›
let *?rely* = ‹*lift-state-pair-set rely*›
let *?guar* = ‹*lift-state-pair-set guar*›
let *?post* = ‹*lift-state-set post*›
obtain *prgf'* :: ‹$'a \Rightarrow (('b, 'a, 's, 'prog)\ esys, 's \times ('a \Rightarrow ('b \times 's\ set \times 'prog)\ option))\ rgformula$›
where *prgf'-def*: ‹$prgf' = (\lambda k.$ ⦇ $Com = Com (prgf k)$, $Pre = lift\text{-}state\text{-}set (Pre (prgf k))$, $Rely = lift\text{-}state\text{-}pair\text{-}set (Rely (prgf k))$,
$Guar = lift\text{-}state\text{-}pair\text{-}set (Guar (prgf k))$, $Post = lift\text{-}state\text{-}set (Post (prgf k))$ ⦈$)$›
by *simp*

from *rely1* **have** *rely1'*: ‹$\forall k. lift\text{-}state\text{-}pair\text{-}set\ rely \subseteq lift\text{-}state\text{-}pair\text{-}set (Rely (prgf k))$›
apply(*simp add*: *lift-state-pair-set-def*) **by** *blast*
from *rely2* **have** *rely2'*: ‹$\forall k\ k'. k' \neq k \longrightarrow lift\text{-}state\text{-}pair\text{-}set (Guar (prgf k')) \subseteq lift\text{-}state\text{-}pair\text{-}set (Rely (prgf k))$›
apply(*simp add*: *lift-state-pair-set-def*) **by** *blast*
from *guar* **have** *guar'*: ‹$\forall k. lift\text{-}state\text{-}pair\text{-}set (Guar (prgf k)) \subseteq ?guar$›
apply(*simp add*: *lift-state-pair-set-def*) **by** *blast*
from *post* **have** *post'*: ‹$\bigcap (lift\text{-}state\text{-}set\ ` (Post\ ` (prgf\ ` UNIV))) \subseteq ?post$›
apply(*simp add*: *lift-state-set-def*) **by** *fast*

have *valid*: ‹$\forall k\ s0. cpts\text{-}from (estran \Gamma) (Com (prgf k), s0) \cap assume\ ?pre (lift\text{-}state\text{-}pair\text{-}set (Rely (prgf k))) \subseteq commit (estran \Gamma) \{fin\} (lift\text{-}state\text{-}pair\text{-}set (Guar (prgf k))) (lift\text{-}state\text{-}set (Post (prgf k)))$›
proof
fix k
from *rghoare-es-sound*[*OF h*[*rule-format*, *of k*]] *pre*[*rule-format*, *of k*]
show ‹$\forall s0. cpts\text{-}from (estran \Gamma) (Com (prgf k), s0) \cap assume\ ?pre (lift\text{-}state\text{-}pair\text{-}set$

(*Rely* (*prgf k*))) ⊆ *commit* (*estran* Γ) {*fin*} (*lift-state-pair-set* (*Guar* (*prgf k*)))
(*lift-state-set* (*Post* (*prgf k*)))›
by (*auto simp add*: *assume-def lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
qed
show ‹∀ *s0 x0*. {*cpt* ∈ *cpts* (*pestran* Γ). *hd cpt* = (*par-com prgf*, *s0*, *x0*)} ∩
assume ?pre ?rely ⊆ *commit* (*pestran* Γ) *par-fin ?guar ?post*›
proof(*rule allI*, *rule allI*)
fix *s0*
fix *x0*
show ‹{*cpt* ∈ *cpts* (*pestran* Γ). *hd cpt* = (*par-com prgf*, *s0*, *x0*)} ∩ *assume*
?pre ?rely ⊆ *commit* (*pestran* Γ) *par-fin ?guar ?post*›
proof(*auto*)
fix *pc*
assume *hd-pc*: ‹*hd pc* = (*par-com prgf*, *s0*, *x0*)›
assume *pc-cpt*: ‹*pc* ∈ *cpts* (*pestran* Γ)›
assume *pc-assume*: ‹*pc* ∈ *assume ?pre ?rely*›
from *hd-pc pc-cpt pc-assume*
have *pc*: ‹*pc* ∈ *cpts-from* (*pestran* Γ) (*par-com prgf*, *s0*, *x0*) ∩ *assume ?pre*
?rely› **by** *simp*
obtain *cs* **where** ‹*cs* = *split-par pc*› **by** *simp*
with *split-par-conjoin*[*OF pc-cpt*] **have** *conjoin*: ‹*pc* ∝ *cs*› **by** *simp*
show ‹*pc* ∈ *commit* (*pestran* Γ) *par-fin ?guar ?post*›
proof(*auto simp add*: *commit-def*)
fix *i*
assume *Suc-i-lt*: ‹*Suc i* < *length pc*›
assume ‹(*pc*!*i*, *pc*!*Suc i*) ∈ *pestran* Γ›
then obtain *a k* **where** ‹Γ ⊢ *pc* ! *i* −*pes*[*a*♯*k*]→ *pc* ! *Suc i*› **by** (*auto simp*
add: *pestran-def*)
then show ‹(*snd* (*pc* ! *i*), *snd* (*pc* ! *Suc i*)) ∈ *?guar*› **apply** −
proof(*erule pestran-p.cases*, *auto*)
fix *pes s x es′ t y*
assume *eq1*: ‹*pc* ! *i* = (*pes*, *s*, *x*)›
assume *eq2*: ‹*pc* ! *Suc i* = (*pes*(*k* := *es′*), *t*, *y*)›
have *eq1s*: ‹*snd* (*cs k* ! *i*) = (*s*,*x*)› **using** *conjoin-same-state*[*OF conjoin*,
rule-format, *OF Suc-i-lt*[*THEN Suc-lessD*], *of k*] *eq1*
by *simp*
have *eq2s*: ‹*snd* (*cs k* ! *Suc i*) = (*t*,*y*)› **using** *conjoin-same-state*[*OF*
conjoin, *rule-format*, *OF Suc-i-lt*, *of k*] *eq2*
by *simp*
have *eq1p*: ‹*fst* (*cs k* ! *i*) = *pes k*› **using** *conjoin-same-spec*[*OF conjoin*,
rule-format, *OF Suc-i-lt*[*THEN Suc-lessD*], *of k*] *eq1*
by *simp*
have *eq2p*: ‹*fst* (*cs k* ! *Suc i*) = *es′*› **using** *conjoin-same-spec*[*OF conjoin*,
rule-format, *OF Suc-i-lt*, *of k*] *eq2*
by *simp*
assume ‹Γ ⊢ (*pes k*, *s*, *x*) −*es*[*a*♯*k*]→ (*es′*, *t*, *y*)›
with *eq1s eq2s eq1p eq2p*
have ‹Γ ⊢ (*fst* (*cs k* ! *i*), *snd* (*cs k* ! *i*)) −*es*[*a*♯*k*]→ (*fst* (*cs k* ! *Suc i*), *snd*
(*cs k* ! *Suc i*))› **by** *simp*

then have *estran*: ‹(*cs k*!*i*, *cs k*!*Suc i*)∈*estran* Γ› **by** (*auto simp add*: *estran-def*)
from *par-sound-aux2*[*of pc* Γ *prgf'*, *simplified prgf'-def rgformula.simps*, *OF pc valid rely1' rely2' guar' conjoin*, *rule-format*, *of i k*, *OF Suc-i-lt estran*]
have ‹(*snd* (*cs k* ! *i*), *snd* (*cs k* ! *Suc i*)) ∈ *lift-state-pair-set* (*Guar* (*prgf k*))› **.**
with *eq1s eq2s* **have** ‹((*s*,*x*),(*t*,*y*)) ∈ *lift-state-pair-set* (*Guar* (*prgf k*))› **by** *simp*
with *guar'* **show** ‹((*s*, *x*), *t*, *y*) ∈ *lift-state-pair-set guar*› **by** *blast*
qed
next
assume ‹∀ *k*. *fst* (*last pc*) *k* = *fin*›
then have *fin*: ‹*fst* (*last pc*) ∈ *par-fin*› **by** *fast*
from *par-sound-aux5*[*of pc* Γ *prgf'*, *simplified prgf'-def rgformula.simps*, *OF pc valid rely1' rely2' guar' conjoin fin*] *post'*
show ‹*snd* (*last pc*) ∈ *lift-state-set post*› **by** *blast*
qed
qed
qed
qed

theorem *rghoare-pes-sound*:
assumes *h*: ‹Γ ⊢ *prgf* SAT_e [*pre*, *rely*, *guar*, *post*]›
shows ‹Γ ⊨ *par-com prgf* SAT_e [*pre*, *rely*, *guar*, *post*]›
using *h*
proof(*cases*)
case *Par*
then show *?thesis* **using** *par-sound* **by** *blast*
qed

definition *Evt-sat-RG* :: *'Env* ⇒ ((*'l*, *'k*, *'s*, *'prog*) *esys*, *'s*) *rgformula* ⇒ *bool* (- ⊢ - [*60*,*60*] *61*)
where Γ ⊢ *rg* ≡ Γ ⊢ *Com rg* sat_e [*Pre rg*, *Rely rg*, *Guar rg*, *Post rg*]

end

end

6 Rely-guarantee-based Safety Reasoning

theory *PiCore-RG-Invariant*
imports *PiCore-Hoare*
begin

type-synonym *'s invariant* = *'s* ⇒ *bool*

context *event-hoare*
begin

definition *invariant-presv-pares*::'*Env* ⇒ '*s invariant* ⇒ ('*l*,'*k*,'*s*,'*prog*) *paresys* ⇒ '*s set* ⇒ ('*s* × '*s*) *set* ⇒ *bool*
where *invariant-presv-pares* Γ *invar pares init R* ≡
∀ *s0 x0 pesl*. *s0*∈*init* ∧ *pesl* ∈ (*cpts-from* (*pestran* Γ) (*pares*, *s0*, *x0*) ∩ *assume* (*lift-state-set init*) (*lift-state-pair-set R*))
⟶ (∀ *i*<*length pesl*. *invar* (*fst* (*snd* (*pesl*!*i*))))

definition *invariant-presv-pares2*::'*Env* ⇒ '*s invariant* ⇒ ('*l*,'*k*,'*s*,'*prog*) *paresys* ⇒ '*s set* ⇒ ('*s* × '*s*) *set* ⇒ *bool*
where *invariant-presv-pares2* Γ *invar pares init R* ≡
∀ *s0 x0 pesl*. *pesl* ∈ (*cpts-from* (*pestran* Γ) (*pares*, *s0*, *x0*) ∩ *assume* (*lift-state-set init*) (*lift-state-pair-set R*))
⟶ (∀ *i*<*length pesl*. *invar* (*fst* (*snd* (*pesl*!*i*))))

lemma *invariant-presv-pares* Γ *invar pares init R* = *invariant-presv-pares2* Γ *invar pares init R*
by (*auto simp add*:*invariant-presv-pares-def invariant-presv-pares2-def assume-def lift-state-set-def*)

theorem *invariant-theorem*:
assumes *parsys-sat-rg*: Γ ⊢ *pesf* SAT_e [*init*, *R*, *G*, *pst*]
and *stb-rely*: *stable* (*Collect invar*) *R*
and *stb-guar*: *stable* (*Collect invar*) *G*
and *init-in-invar*: *init* ⊆ (*Collect invar*)
shows *invariant-presv-pares* Γ *invar* (*par-com pesf*) *init R*
proof −
let *?init* = ‹*lift-state-set init*›
let *?R* = ‹*lift-state-pair-set R*›
let *?G* = ‹*lift-state-pair-set G*›
let *?pst* = ‹*lift-state-set pst*›
from *parsys-sat-rg* **have** Γ ⊨ *par-com pesf* SAT_e [*init*, *R*, *G*, *pst*] **using** *rghoare-pes-sound* **by** *fast*
hence *cpts-pes*: ∀ *s*. (*cpts-from* (*pestran* Γ) (*par-com pesf*, *s*)) ∩ *assume ?init ?R* ⊆ *commit* (*pestran* Γ) *par-fin ?G ?pst* **by** *simp*
show *?thesis*
proof −
{
fix *s0 x0 pesl*
assume *a0*: *s0*∈*init*
and *a1*: *pesl*∈*cpts-from* (*pestran* Γ) (*par-com pesf*, *s0*, *x0*) ∩ *assume ?init ?R*
from *a1* **have** *a3*: *pesl*!*0* = (*par-com pesf*, *s0*, *x0*) ∧ *pesl*∈*cpts* (*pestran* Γ) **using** *hd-conv-nth cpts-nonnil* **by** *force*
from *a1 cpts-pes* **have** *pesl-in-comm*: *pesl* ∈ *commit* (*pestran* Γ) *par-fin ?G ?pst* **by** *auto*
{
fix *i*
assume *b0*: *i*<*length pesl*
then have *fst* (*snd* (*pesl*!*i*)) ∈ (*Collect invar*)

proof(*induct i*)
case *0*
with *a3* **have** *snd* (*pesl*!*0*) = (*s0*,*x0*) **by** *simp*
with *a0 init-in-invar* **show** *?case* **by** *auto*
next
case (*Suc ni*)
assume *c0*: *ni* < *length pesl* ⟹ *fst* (*snd* (*pesl* ! *ni*)) ∈ (*Collect invar*)
and *c1*: *Suc ni* < *length pesl*
then have *c2*: *fst* (*snd* (*pesl* ! *ni*)) ∈ (*Collect invar*) **by** *auto*
from *c1* **have** *c3*: *ni* < *length pesl* **by** *simp*
with *c0* **have** *c4*: *fst* (*snd* (*pesl* ! *ni*)) ∈ (*Collect invar*) **by** *simp*
from *a3 c1* **have** *pesl* ! *ni* −*e*→ *pesl* ! *Suc ni* ∨ (*pesl* ! *ni*, *pesl* ! *Suc ni*) ∈ *pestran* Γ
using *ctran-or-etran-par* **by** *blast*
then show *?case*
proof
assume *d0*: *pesl* ! *ni* −*e*→ *pesl* ! *Suc ni*
then show *?thesis* **using** *c3 c4 a1 c1 stb-rely* **by**(*simp add*:*assume-def stable-def lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
next
assume (*pesl* ! *ni*, *pesl* ! *Suc ni*) ∈ *pestran* Γ
then obtain *et* **where** *d0*: Γ ⊢ *pesl* ! *ni* −*pes*[*et*]→ *pesl* ! *Suc ni* **by** (*auto simp add*: *pestran-def*)
then show *?thesis* **using** *c3 c4 c1 pesl-in-comm stb-guar*
apply(*simp add*:*commit-def stable-def lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
using ‹(*pesl* ! *ni*, *pesl* ! *Suc ni*) ∈ *pestran* Γ› **by** *blast*
qed
qed
}
}
then show *?thesis* **using** *invariant-presv-pares-def* **by** *blast*
qed
qed

end

end

7 Extending SIMP language with new proof rules

theory *SIMP-plus*
imports *HOL−Hoare-Parallel.RG-Hoare*
begin

7.1 new proof rules

inductive *rghoare-p* :: [*′a com option*, *′a set*, (*′a* × *′a*) *set*, (*′a* × *′a*) *set*, *′a set*] ⇒ *bool*

($\vdash_I$ - sat_p [-, -, -, -] [60,0,0,0,0] 45)
where

Basic: $[\![$ $pre \subseteq \{s.\ f\ s \in post\}$; $\{(s,t).\ s \in pre \wedge (t{=}f\ s)\} \subseteq guar$;
$stable\ pre\ rely$; $stable\ post\ rely$ $]\!]$
$\Longrightarrow \vdash_I Some\ (Basic\ f)\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *Seq*: $[\![\vdash_I Some\ P\ sat_p\ [pre,\ rely,\ guar,\ mid]; \vdash_I Some\ Q\ sat_p\ [mid,\ rely,\ guar,\ post]]\!]$
$\Longrightarrow \vdash_I Some\ (Seq\ P\ Q)\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *Cond*: $[\![$ $stable\ pre\ rely$; $\vdash_I Some\ P1\ sat_p\ [pre \cap b,\ rely,\ guar,\ post]$;
$\vdash_I Some\ P2\ sat_p\ [pre \cap -b,\ rely,\ guar,\ post]$; $\forall s.\ (s,s){\in}guar$ $]\!]$
$\Longrightarrow \vdash_I Some\ (Cond\ b\ P1\ P2)\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *While*: $[\![$ $stable\ pre\ rely$; $(pre \cap -b) \subseteq post$; $stable\ post\ rely$;
$\vdash_I Some\ P\ sat_p\ [pre \cap b,\ rely,\ guar,\ pre]$; $\forall s.\ (s,s){\in}guar$ $]\!]$
$\Longrightarrow \vdash_I Some\ (While\ b\ P)\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *Await*: $[\![$ $stable\ pre\ rely$; $stable\ post\ rely$;
$\forall V. \vdash_I Some\ P\ sat_p\ [pre \cap b \cap \{V\},\ \{(s,\ t).\ s = t\},$
$UNIV,\ \{s.\ (V,\ s) \in guar\} \cap post]$ $]\!]$
$\Longrightarrow \vdash_I Some\ (Await\ b\ P)\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *None-hoare*: $[\![$ $stable\ pre\ rely$; $pre \subseteq post$ $]\!]$ $\Longrightarrow \vdash_I None\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *Conseq*: $[\![$ $pre \subseteq pre'$; $rely \subseteq rely'$; $guar' \subseteq guar$; $post' \subseteq post$;
$\vdash_I P\ sat_p\ [pre',\ rely',\ guar',\ post']$ $]\!]$
$\Longrightarrow \vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post]$

| *Unprecond*: $[\![\vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post]; \vdash_I P\ sat_p\ [pre',\ rely,\ guar,\ post]]\!]$
$\Longrightarrow \vdash_I P\ sat_p\ [pre \cup pre',\ rely,\ guar,\ post]$

| *Intpostcond*: $[\![\vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post]; \vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post']]\!]$
$\Longrightarrow \vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post \cap post']$

| *Allprecond*: $\forall v{\in}U. \vdash_I P\ sat_p\ [\{v\},\ rely,\ guar,\ post]$
$\Longrightarrow \vdash_I P\ sat_p\ [U,\ rely,\ guar,\ post]$

| *Emptyprecond*: $\vdash_I P\ sat_p\ [\{\},\ rely,\ guar,\ post]$

definition *prog-validity* :: $'a\ com\ option \Rightarrow 'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set \Rightarrow 'a\ set \Rightarrow bool$
($\models_I$ - sat_p [-, -, -, -] [60,0,0,0,0] 45) **where**
$\models_I P\ sat_p\ [pre,\ rely,\ guar,\ post] \equiv$
$\forall s.\ cp\ P\ s \cap assum(pre,\ rely) \subseteq comm(guar,\ post)$

7.2 lemmas of SIMP

lemma *etran-or-ctran2-disjI3*:
$[\![x{\in}cptn;\ Suc\ i{<}length\ x;\ \neg\ x!i\ -c\rightarrow\ x!Suc\ i]\!] \Longrightarrow x!i\ -e\rightarrow\ x!Suc\ i$
apply(*induct x arbitrary*:*i*)
 apply *simp*
apply *clarify*
apply(*rule cptn.cases*)
  apply *simp+*
  using *less-Suc-eq-0-disj etran.intros* **apply** *force*
  apply(*case-tac i*,*simp*)
  by *simp*

lemma *stable-id*: *stable P Id*
  unfolding *stable-def Id-def* **by** *auto*

lemma *stable-id2*: *stable P* $\{(s,t).\ s = t\}$
  unfolding *stable-def* **by** *auto*

lemma *stable-int2*: *stable s r* $\Longrightarrow$ *stable t r* $\Longrightarrow$ *stable* $(s \cap t)$ *r*
  by (*metis* (*full-types*) *IntD1 IntD2 IntI stable-def*)

lemma *stable-int3*: *stable k r* $\Longrightarrow$ *stable s r* $\Longrightarrow$ *stable t r* $\Longrightarrow$ *stable* $(k \cap s \cap t)$
r
  by (*metis* (*full-types*) *IntD1 IntD2 IntI stable-def*)

lemma *stable-un2*: *stable s r* $\Longrightarrow$ *stable t r* $\Longrightarrow$ *stable* $(s \cup t)$ *r*
  by (*simp add*: *stable-def*)

lemma *Seq2*: $[\![\vdash_I$ *Some P* sat_p [*pre*, *rely*, *guar*, *mida*]; *mida* $\subseteq$ *midb*; $\vdash_I$ *Some Q*
sat_p [*midb*, *rely*, *guar*, *post*] $]\!]$
  $\Longrightarrow \vdash_I$ *Some* (*Seq P Q*) sat_p [*pre*, *rely*, *guar*, *post*]
  using *Seq*[*of P pre rely guar mida Q post*]
        Conseq[*of mida midb rely rely guar guar post post*]
  by *blast*

7.3 Soundness of the Rule of Consequence

lemma *Conseq-sound*:
  $[\![$*pre* $\subseteq$ *pre'*; *rely* $\subseteq$ *rely'*; *guar'* $\subseteq$ *guar*; *post'* $\subseteq$ *post*;
  $\models_I$ *P* sat_p [*pre'*, *rely'*, *guar'*, *post'*]$]\!]$
  $\Longrightarrow \models_I$ *P* sat_p [*pre*, *rely*, *guar*, *post*]
apply(*simp add*:*prog-validity-def assum-def comm-def*)
apply *clarify*
apply(*erule-tac x*=*s* **in** *allE*)
apply(*drule-tac c*=*x* **in** *subsetD*)
 apply *force*

apply *force*
done

7.4 Soundness of the Rule of Unprecond

lemma *Unprecond-sound*:
  assumes *p0*: $\models_I$ *P* sat_p [*pre*, *rely*, *guar*, *post*]
    and *p1*: $\models_I$ *P* sat_p [pre', *rely*, *guar*, *post*]
   shows $\models_I$ *P* sat_p [$pre \cup pre'$, *rely*, *guar*, *post*]
proof −
{
  fix *s c*
  assume $c \in cp\ P\ s \cap assum(pre \cup pre', rely)$
  hence *a1*: $c \in cp\ P\ s$ **and**
        a2: $c \in assum(pre \cup pre', rely)$ **by** *auto*
  hence $c \in assum(pre, rely) \vee c \in assum(pre', rely)$
    by (*metis* (*no-types*, *lifting*) *CollectD CollectI Un-iff assum-def prod.simps*(*2*))
  hence $c \in comm(guar, post)$
    proof
      assume $c \in assum\ (pre, rely)$
      with *p0 a1* **show** $c \in comm\ (guar, post)$
        unfolding *prog-validity-def* **by** *auto*
    next
      assume $c \in assum\ (pre', rely)$
      with *p1 a1* **show** $c \in comm\ (guar, post)$
        unfolding *prog-validity-def* **by** *auto*
    qed
}
then show *?thesis* **unfolding** *prog-validity-def* **by** *auto*
qed

7.5 Soundness of the Rule of Intpostcond

lemma *Intpostcond-sound*:
  assumes *p0*: $\models_I$ *P* sat_p [*pre*, *rely*, *guar*, *post*]
    and *p1*: $\models_I$ *P* sat_p [*pre*, *rely*, *guar*, $post'$]
   shows $\models_I$ *P* sat_p [*pre*, *rely*, *guar*, $post \cap post'$]
proof −
{
  fix *s c*
  assume *a0*: $c \in cp\ P\ s \cap assum(pre, rely)$
  with *p0* **have** $c \in comm(guar, post)$ **unfolding** *prog-validity-def* **by** *auto*
  moreover
  from *a0 p1* **have** $c \in comm(guar, post')$ **unfolding** *prog-validity-def* **by** *auto*
  ultimately have $c \in comm(guar, post \cap post')$
    by (*simp add*: *comm-def*)
}
then show *?thesis* **unfolding** *prog-validity-def* **by** *auto*
qed

7.6 Soundness of the Rule of Allprecond

lemma *Allprecond-sound*:
  assumes *p1*: $\forall v\in U.\ \models_I P\ sat_p\ [\{v\},\ rely,\ guar,\ post]$
    shows $\models_I P\ sat_p\ [U,\ rely,\ guar,\ post]$
proof −
{
  fix *s c*
  assume *a0*: $c \in cp\ P\ s \cap assum(U,\ rely)$
  then obtain *x* **where** *a1*: $x\in U \wedge snd\ (c!0) = x$
    by (*metis* (*no-types*, *lifting*) *CollectD IntD2 assum-def prod.simps*(*2*))

  with *p1* **have** $\models_I P\ sat_p\ [\{x\},\ rely,\ guar,\ post]$ **by** *simp*
  hence *a2*: $\forall s.\ cp\ P\ s \cap assum(\{x\},\ rely) \subseteq comm(guar,\ post)$ **unfolding** *prog-validity-def* **by** *simp*

  from *a0* **have** $c \in assum(U,\ rely)$ **by** *simp*
  hence $snd\ (c!0) \in U \wedge (\forall i.\ Suc\ i<length\ c \longrightarrow$
        $c!i\ -e\rightarrow\ c!(Suc\ i) \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in rely)$ **by** (*simp add*:*assum-def*)
  with *a1* **have** $snd\ (c!0) \in \{x\} \wedge (\forall i.\ Suc\ i<length\ c \longrightarrow$
        $c!i\ -e\rightarrow\ c!(Suc\ i) \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in rely)$ **by** *simp*

  hence $c\in assum(\{x\},\ rely)$ **by** (*simp add*:*assum-def*)
  with *a0 a2* **have** $c \in comm(guar,\ post)$ **by** *auto*
}
then show *?thesis* **using** *prog-validity-def* **by** *blast*
qed

7.7 Soundness of the Rule of Emptyprecond

lemma *Emptyprecond-sound*: $\models_I P\ sat_p\ [\{\},\ rely,\ guar,\ post]$
unfolding *prog-validity-def* **by**(*simp add*:*assum-def*)

7.8 Soundness of None rule

lemma *none-all-none*: $c!0=(None,s) \wedge c \in cptn \Longrightarrow \forall i<length\ c.\ fst\ (c\ !\ i) = None$
proof(*induct c arbitrary*:*s*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a c*)
  assume *p1*: $\bigwedge s.\ c\ !\ 0 = (None,\ s) \wedge c \in cptn \Longrightarrow \forall i<length\ c.\ fst\ (c\ !\ i) = None$
    and *p2*: $(a\ \#\ c)\ !\ 0 = (None,\ s) \wedge a\ \#\ c \in cptn$
  hence *a0*: $a = (None,s)$ **by** *simp*
  thus *?case*
    proof(*cases* $c = []$)
      case *True*

with *a0* **show** *?thesis* **by** *auto*
next
case *False*
assume *b0*: $c \neq []$
with *p2* **have** *c-cpts*: $c \in cptn$ **using** *tl-in-cptn* **by** *fast*
from *b0* **obtain** c' **and** *b* **where** *bc'*: $c = b \# c'$
using *list.exhaust* **by** *blast*
from *a0* **have** $\neg\ a\ -c\rightarrow\ b$ **by**(*force elim*: *ctran.cases*)
with *p2* **have** $a\ -e\rightarrow\ b$ **using** *bc'* *etran-or-ctran2-disjI3*[*of a#c 0*] **by** *auto*
hence $fst\ b = None$ **using** *etran.cases*
by (*metis a0 prod.collapse*)
with *p1* *bc'* *c-cpts* **have** $\forall i<length\ c.\ fst\ (c\ !\ i) = None$
by (*metis nth-Cons-0 prod.collapse*)
with *a0* **show** *?thesis*
by (*simp add*: *nth-Cons'*)
qed

qed

lemma *None-sound-h*: $\forall x.\ x \in pre \longrightarrow (\forall y.\ (x, y) \in rely \longrightarrow y \in pre) \Longrightarrow$
$pre \subseteq post \Longrightarrow$
$snd\ (c\ !\ 0) \in pre \Longrightarrow$
$c \neq [] \Longrightarrow \forall i.\ Suc\ i < length\ c \longrightarrow (snd\ (c\ !\ i),\ snd\ (c\ !\ Suc\ i)) \in rely$
$\Longrightarrow i < length\ c \Longrightarrow snd\ (c\ !\ i) \in pre$
apply(*induct i*) **by** *auto*

lemma *None-sound*:
$[\![$ *stable pre rely*; $pre \subseteq post$ $]\!]$
$\Longrightarrow \models_I None\ sat_p\ [pre,\ rely,\ guar,\ post]$
proof –
assume *p0*: *stable pre rely*
and *p2*: $pre \subseteq post$
{
fix *s c*
assume *a0*: $c \in cp\ None\ s \cap assum(pre,\ rely)$
hence *c1*: $c!0=(None,s) \wedge c \in cptn$ **by** (*simp add*:*cp-def*)
from *a0* **have** *c2*: $snd\ (c!0) \in pre \wedge (\forall i.\ Suc\ i<length\ c \longrightarrow$
$c!i\ -e\rightarrow\ c!(Suc\ i) \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in rely)$
by (*simp add*:*assum-def*)
from *c1* **have** *c-ne-empty*: $c \neq []$
by *auto*
from *c1* **have** *c-all-none*: $\forall i<length\ c.\ fst\ (c\ !\ i) = None$ **using** *none-all-none*
by *fast*

{
fix *i*
assume *suci*: $Suc\ i<length\ c$
and *cc*: $c!i\ -c\rightarrow\ c!(Suc\ i)$

from *suci c-all-none* **have** $c!i\ {-e\rightarrow}\ c!(Suc\ i)$
by (*metis Suc-lessD etran.intros prod.collapse*)
with *cc* **have**$(snd\ (c!i),\ snd\ (c!Suc\ i)) \in guar$
using *c1 etran-or-ctran2-disjI1 suci* **by** *auto*
}
moreover
{
assume *last-none*: $fst\ (last\ c) = None$
from *c2 c-all-none* **have** $\forall i.\ Suc\ i{<}length\ c \longrightarrow (snd\ (c!i),\ snd\ (c!Suc\ i)) \in rely$
by (*metis Suc-lessD etran.intros prod.collapse*)
with *p0 p2 c2 c-ne-empty* **have** $\forall i.\ i < length\ c \longrightarrow snd\ (c\ !\ i) \in pre$
apply(*simp add*: *stable-def*) **apply** *clarify* **using** *None-sound-h* **by** *blast*
with *p2 c-ne-empty* **have** $snd\ (last\ c) \in post$
using *One-nat-def c-ne-empty last-conv-nth* **by** *force*
}
ultimately have $c \in comm(guar,\ post)$ **by** (*simp add*:*comm-def*)
}
thus $\models_I None\ sat_p\ [pre,\ rely,\ guar,\ post]$ **using** *prog-validity-def* **by** *blast*
qed

7.9 Soundness of the Await rule

lemma *Await-sound*:
$[\![stable\ pre\ rely;\ stable\ post\ rely;$
$\forall V.\ \vdash_I Some\ P\ sat_p\ [pre \cap b \cap \{s.\ s = V\},\ \{(s,\ t).\ s = t\},$
$UNIV,\ \{s.\ (V,\ s) \in guar\} \cap post] \wedge$
$\models_I Some\ P\ sat_p\ [pre \cap b \cap \{s.\ s = V\},\ \{(s,\ t).\ s = t\},$
$UNIV,\ \{s.\ (V,\ s) \in guar\} \cap post]\ ]\!]$
$\Longrightarrow\ \models_I Some\ (Await\ b\ P)\ sat_p\ [pre,\ rely,\ guar,\ post]$
apply(*unfold prog-validity-def*)
apply *clarify*
apply(*simp add*:*comm-def*)
apply(*rule conjI*)
apply *clarify*
apply(*simp add*:*cp-def assum-def*)
apply *clarify*
apply(*frule-tac j=0* **and** *k=i* **and** *p=pre* **in** *stability*,*simp-all*)
apply(*erule-tac x=ia* **in** *allE*,*simp*)
apply(*subgoal-tac* $x\in cp\ (Some(Await\ b\ P))\ s$)
apply(*erule-tac i=i* **in** *unique-ctran-Await*,*force*,*simp-all*)
apply(*simp add*:*cp-def*)

apply(*erule ctran.cases*,*simp-all*)
apply(*drule Star-imp-cptn*)
apply *clarify*
apply(*erule-tac x=sa* **in** *allE*)
apply *clarify*
apply(*erule-tac x=sa* **in** *allE*)

```
 apply(drule-tac c=l in subsetD)
  apply (simp add:cp-def)
  apply clarify
  apply(erule-tac x=ia and P=λi. H i ⟶ (J i, I i)∈ctran for H J I in allE,simp)
  apply(erule etranE,simp)
 apply simp
apply clarify
apply(simp add:cp-def)
apply clarify
apply(frule-tac i=length x - 1 in exists-ctran-Await-None,force)
  apply (case-tac x,simp+)
 apply(rule last-fst-esp,simp add:last-length)
 apply(case-tac x, simp+)
apply clarify
apply(simp add:assum-def)
apply clarify
apply(frule-tac j=0 and k=j and p=pre in stability,simp-all)
  apply(erule-tac x=i in allE,simp)
 apply(erule-tac i=j in unique-ctran-Await,force,simp-all)
apply(case-tac x!j)
apply clarify
apply simp
apply(drule-tac s=Some (Await b P) in sym,simp)
apply(case-tac x!Suc j,simp)
apply(rule ctran.cases,simp)
apply(simp-all)
apply(drule Star-imp-cptn)
apply clarify
apply(erule-tac x=sa in allE)
apply clarify
apply(erule-tac x=sa in allE)
apply(drule-tac c=l in subsetD)
 apply (simp add:cp-def)
 apply clarify
 apply(erule-tac x=i and P=λi. H i ⟶ (J i, I i)∈ctran for H J I in allE,simp)
 apply(erule etranE,simp)
apply simp
apply clarify
apply(frule-tac j=Suc j and k=length x - 1 and p=post in stability,simp-all)
 apply(case-tac x,simp+)
 apply(erule-tac x=i in allE)
apply(erule-tac i=j in unique-ctran-Await,force,simp-all)
 apply arith+
apply(case-tac x)
apply(simp add:last-length)+
done
```

theorem *rgsound-p*:
  $\vdash_I P\ sat_p\ [pre,\ rely,\ guar,\ post] \Longrightarrow \models_I P\ sat_p\ [pre,\ rely,\ guar,\ post]$

apply(*erule rghoare-p.induct*)
using *RG-Hoare.Basic-sound* **apply**(*simp add*:*prog-validity-def com-validity-def*)
apply *blast*
using *RG-Hoare.Seq-sound* **apply**(*simp add*:*prog-validity-def com-validity-def*) **apply** *blast*
using *RG-Hoare.Cond-sound* **apply**(*simp add*:*prog-validity-def com-validity-def*)
apply *blast*
using *RG-Hoare.While-sound* **apply**(*simp add*:*prog-validity-def com-validity-def*)
apply *blast*
using *Await-sound* **apply** *fastforce*
apply(*force elim*:*None-sound*)
apply(*erule Conseq-sound*,*simp+*)
apply(*erule Unprecond-sound*,*simp+*)
apply(*erule Intpostcond-sound*,*simp+*)
using *Allprecond-sound* **apply** *force*
using *Emptyprecond-sound* **apply** *force*
done

end

8 Rely-guarantee-based Safety Reasoning

theory *PiCore-ext*
imports *PiCore-Hoare*
begin

definition *list-of-set aset* ≡ (*SOME l. set l* = *aset*)

lemma *set-of-list-of-set*:
assumes *fin*: *finite aset*
shows *set* (*list-of-set aset*) = *aset*
proof(*simp add*: *list-of-set-def*)
from *fin* **obtain** *l* **where** *set l* = *aset* **using** *finite-list* **by** *auto*
then show *set* (*SOME l. set l* = *aset*) = *aset*
by (*metis* (*mono-tags*, *lifting*) *some-eq-ex*)
qed

context *event-hoare*
begin

fun *OR-list* :: ('l,'k,'s,'prog) *esys list* ⇒ ('l,'k,'s,'prog) *esys* **where**
OR-list [*a*] = *a* |
OR-list (*a*#*b*#*ax*) = *a OR* (*OR-list* (*b*#*ax*)) |
OR-list [] = *fin*

lemma *OR-list* [*a*] = *a* **by** *auto*
lemma *OR-list* [*a*,*b*] = *a OR b* **by** *auto*
lemma *OR-list* [*a*,*b*,*c*] = *a OR* (*b OR c*) **by** *auto*

lemma *Evt-OR-list*:
  $ess \neq [] \Longrightarrow \forall\, i{<}length\ ess.\ \Gamma \vdash (ess!i)\ sat_e\ [pre,\ rely,\ guar,\ post]$
  $\Longrightarrow \Gamma \vdash (OR\text{-}list\ ess)\ sat_e\ [pre,\ rely,\ guar,\ post]$
  apply(*induct ess*) **apply** *simp*
  apply(*case-tac ess=[]*) **apply** *auto[1]*
  by (*metis Evt-Choice OR-list.simps(2) length-Cons less-Suc-eq-0-disj list.exhaust nth-Cons-0 nth-Cons-Suc*)

fun $AND\text{-}list :: ('l,'k,'s,'prog)\ esys\ list \Rightarrow ('l,'k,'s,'prog)\ esys$ **where**
  $AND\text{-}list\ [a] = a\ |$
  $AND\text{-}list\ (a\#b\#ax) = a \bowtie (AND\text{-}list\ (b\#ax))\ |$
  $AND\text{-}list\ [] = fin$

lemma $AND\text{-}list\ [a] = a$ **by** *auto*
lemma $AND\text{-}list\ [a,b] = a \bowtie b$ **by** *auto*
lemma $AND\text{-}list\ [a,b,c] = a \bowtie (b \bowtie c)$ **by** *auto*

lemma *Int-list-lm*: $P\ a \cap (\bigcap i{<}length\ ess.\ P\ (ess\ !\ i)) = (\bigcap i{<}length\ (a\ \#\ ess).\ P\ ((a\ \#\ ess)\ !\ i))$
  apply(*induct ess*) **apply** *auto[1]*
  apply(*rule subset-antisym*)
  apply *auto[1]* **apply** (*metis lessThan-iff less-Suc-eq-0-disj nth-Cons-0 nth-Cons-Suc*)
  apply *auto*
  by (*metis Suc-leI le-imp-less-Suc lessThan-iff nth-Cons-Suc*)

lemma *Evt-AND-list*:
  $ess \neq [] \Longrightarrow$
  $\forall\, i{<}length\ ess.\ \Gamma \vdash Com\ (ess!i)\ sat_e\ [Pre\ (ess!i),\ Rely\ (ess!i),\ Guar\ (ess!i),\ Post\ (ess!i)] \Longrightarrow$
  $\forall\, i{<}length\ ess.\ \forall\, s.\ (s,s){\in}Guar\ (ess!i) \Longrightarrow$
  $\forall\, i\ j.\ i < length\ ess \wedge j < length\ ess \wedge i \neq j \longrightarrow Guar\ (ess!i) \subseteq Rely\ (ess!j) \Longrightarrow$
  $\Gamma \vdash (AND\text{-}list\ (map\ Com\ ess))\ sat_e\ [\bigcap i{<}length\ ess.\ Pre\ (ess!i),\ \bigcap i{<}length\ ess.\ Rely\ (ess!i),$
  $\bigcup i{<}length\ ess.\ Guar\ (ess!i),\ \bigcap i{<}length\ ess.\ Post\ (ess!i)]$
  apply(*induct ess*) **apply** *simp*
  apply(*case-tac ess=[]*) **apply** *auto[1]*
proof−
  fix *a ess*

  assume *a0*: $ess \neq [] \Longrightarrow$
  $\forall\, i{<}length\ ess.\ \Gamma \vdash Com\ (ess\ !\ i)\ sat_e\ [Pre\ (ess\ !\ i),\ Rely\ (ess\ !\ i),\ Guar\ (ess\ !\ i),\ Post\ (ess\ !\ i)] \Longrightarrow$
  $\forall\, i{<}length\ ess.\ \forall\, s.\ (s,\ s) \in Guar\ (ess\ !\ i) \Longrightarrow$
  $\forall\, i\ j.\ i < length\ ess \wedge j < length\ ess \wedge i \neq j \longrightarrow Guar\ (ess\ !\ i) \subseteq Rely\ (ess\ !\ j) \Longrightarrow$

$\Gamma \vdash$ *AND-list* (*map Com ess*) sat_e [$\bigcap i{<}length\ ess.\ Pre\ (ess\ !\ i)$, $\bigcap i{<}length\ ess.\ Rely\ (ess\ !\ i)$,
$\bigcup i{<}length\ ess.\ Guar\ (ess\ !\ i)$, $\bigcap i{<}length\ ess.\ Post\ (ess\ !\ i)$]
and *a1*: $a \# ess \neq []$
and *a2*: $\forall i{<}length\ (a \# ess).\ \Gamma \vdash Com\ ((a \# ess)\ !\ i)\ sat_e\ [Pre\ ((a \# ess)\ !\ i),$
$Rely\ ((a \# ess)\ !\ i),\ Guar\ ((a \# ess)\ !\ i),\ Post\ ((a \# ess)\ !\ i)]$
and *a3*: $\forall i{<}length\ (a \# ess).\ \forall s.\ (s, s) \in Guar\ ((a \# ess)\ !\ i)$
and *a4*: $\forall i\ j.\ i < length\ (a \# ess) \wedge j < length\ (a \# ess) \wedge i \neq j$
$\longrightarrow Guar\ ((a \# ess)\ !\ i) \subseteq Rely\ ((a \# ess)\ !\ j)$
and *a5*: $ess \neq []$
let $?pre = \bigcap i{<}length\ ess.\ Pre\ (ess\ !\ i)$
let $?rely = \bigcap i{<}length\ ess.\ Rely\ (ess\ !\ i)$
let $?guar = \bigcup i{<}length\ ess.\ Guar\ (ess\ !\ i)$
let $?post = \bigcap i{<}length\ ess.\ Post\ (ess\ !\ i)$
let $?pre' = \bigcap i{<}length\ (a \# ess).\ Pre\ ((a \# ess)\ !\ i)$
let $?rely' = \bigcap i{<}length\ (a \# ess).\ Rely\ ((a \# ess)\ !\ i)$
let $?guar' = \bigcup i{<}length\ (a \# ess).\ Guar\ ((a \# ess)\ !\ i)$
let $?post' = \bigcap i{<}length\ (a \# ess).\ Post\ ((a \# ess)\ !\ i)$

from *a2* **have** *a6*: $\forall i{<}length\ ess.\ \Gamma \vdash Com\ (ess\ !\ i)\ sat_e\ [Pre\ (ess\ !\ i),\ Rely\ (ess\ !\ i),\ Guar\ (ess\ !\ i),\ Post\ (ess\ !\ i)]$
by *auto*
moreover
from *a3* **have** *a7*: $\forall i{<}length\ ess.\ \forall s.\ (s, s) \in Guar\ (ess\ !\ i)$ **by** *auto*
moreover
from *a4* **have** *a8*: $\forall i\ j.\ i < length\ ess \wedge j < length\ ess \wedge i \neq j \longrightarrow Guar\ (ess\ !\ i) \subseteq Rely\ (ess\ !\ j)$
by *fastforce*
ultimately have *b1*: $\Gamma \vdash$ *AND-list* (*map Com ess*) sat_e [*?pre*, *?rely*, *?guar*, *?post*]
using *a0 a5* **by** *auto*
have *b2*: *AND-list* (*map Com* ($a \# ess$)) = *Com a* $\bowtie$ *AND-list* (*map Com ess*)
by (*metis* (*no-types*, *hide-lams*) *AND-list.simps*(*2*) *a5 list.exhaust list.simps*(*9*))
from *a2* **have** *b3*: $\Gamma \vdash Com\ a\ sat_e\ [Pre\ a,\ Rely\ a,\ Guar\ a,\ Post\ a]$
by *fastforce*
have *b4*: $\Gamma \vdash$ *AND-list* (*map Com ess*) sat_e [*?pre'*, *?rely*, *?guar*, *?post*]
apply(*rule Evt-conseq*[*of ?pre' ?pre ?rely ?rely ?guar ?guar ?post ?post*])
apply *fastforce* **using** *b1* **by** *simp+*
have *b5*: $\Gamma \vdash Com\ a\ sat_e$ [*?pre'*, *Rely a*, *Guar a*, *Post a*]
apply(*rule Evt-conseq*[*of ?pre' Pre a Rely a Rely a Guar a Guar a Post a Post a*])
apply *fastforce*
using *b3* **by** *simp+*
show $\Gamma \vdash$ *AND-list* (*map Com* ($a \# ess$)) sat_e [*?pre'*, *?rely'*, *?guar'*, *?post'*]
apply(*rule subst*[**where** *t=AND-list* (*map Com* ($a \# ess$)) **and** *s= Com a* $\bowtie$ *AND-list* (*map Com ess*)])
using *b2* **apply** *simp*
apply(*rule subst*[**where** *s=Post a* $\cap$ *?post* **and** *t=?post'*])

prefer *2*
apply(*rule Evt-Join*[*of* Γ *Com a ?pre′ Rely a Guar a Post a AND-list* (*map Com ess*)
?pre′ ?rely ?guar ?post ?pre′ ?rely′ ?guar′])
using *b5* **apply** *fast*
using *b4* **apply** *fast*
apply *blast*
apply(*rule Un-least*) **apply** *fastforce* **apply** *clarsimp* **using** *a4*
apply (*smt Suc-mono a1 drop-Suc-Cons hd-drop-conv-nth length-Cons length-greater-0-conv nat.simps*(*3*) *nth-Cons-0 set-mp*)
apply(*rule Un-least*) **apply** *fastforce* **apply** *clarsimp* **using** *a4*
apply (*smt Suc-mono a1 drop-Suc-Cons hd-drop-conv-nth length-Cons length-greater-0-conv nat.simps*(*3*) *nth-Cons-0 set-mp*)
using *a3* **apply** *force* **using** *a3 a5 a7* **apply** *auto*[*1*]
apply *auto*[*1*]
using *Int-list-lm* **by** *metis*
qed

lemma *Evt-AND-list2*:
$ess \neq [] \Longrightarrow$
$\forall i{<}length\ ess.\ \Gamma \vdash Com\ (ess!i)\ sat_e\ [Pre\ (ess!i),\ Rely\ (ess!i),\ Guar\ (ess!i),\ Post\ (ess!i)] \Longrightarrow$
$\forall i{<}length\ ess.\ \forall s.\ (s,s){\in}Guar\ (ess!i) \Longrightarrow$
$\forall i{<}length\ ess.\ P \subseteq Pre\ (ess!i) \Longrightarrow$
$\forall i{<}length\ ess.\ Guar\ (ess!i) \subseteq G \Longrightarrow$
$\forall i{<}length\ ess.\ R \subseteq Rely\ (ess!i) \Longrightarrow$
$\forall i\ j.\ i{<}length\ ess \wedge j < length\ ess \wedge i \neq j \longrightarrow Guar\ (ess!i) \subseteq Rely\ (ess!j) \Longrightarrow$
$\forall i{<}length\ ess.\ Post\ (ess!i) \subseteq Q \Longrightarrow$
$\Gamma \vdash (AND\text{-}list\ (map\ Com\ ess))\ sat_e\ [P,\ R,\ G,\ Q]$

apply(*rule Evt-conseq*[*of* $P\ \bigcap i{<}length\ ess.\ Pre\ (ess!i)$
$R\ \bigcap i{<}length\ ess.\ Rely\ (ess!i)$
$\bigcup i{<}length\ ess.\ Guar\ (ess!i)\ G$
$\bigcap i{<}length\ ess.\ Post\ (ess!i)\ Q$
Γ *AND-list* (*map Com ess*)])
apply *fast* **apply** *fast* **apply** *fast* **apply** *fastforce*
using *Evt-AND-list* **by** *metis*

definition ⟨*react-sys l* ≡ *EWhile UNIV* (*OR-list l*)⟩

lemma *fin-sat*:
⟨*stable P R* $\Longrightarrow \Gamma \models fin\ sat_e\ [P,\ R,\ G,\ P]$⟩
proof(*simp*, *rule allI*, *rule allI*, *standard*)
let *?P* = ⟨*lift-state-set P*⟩
let *?R* = ⟨*lift-state-pair-set R*⟩
let *?G* = ⟨*lift-state-pair-set G*⟩

fix *s0 x0*

fix *cpt*
assume *stable*: ‹*stable P R*›
assume ‹*cpt* ∈ {*cpt* ∈ *cpts* (*estran* Γ). *hd cpt* = (*fin*, *s0*, *x0*)} ∩ *assume ?P ?R*›

then have *cpt*: ‹*cpt* ∈ *cpts* (*estran* Γ)› **and** *hd-cpt*: ‹*hd cpt* = (*fin*, *s0*, *x0*)› **and** *cpt-assume*: ‹*cpt* ∈ *assume ?P ?R*› **by** *auto*
from *cpts-nonnil*[*OF cpt*] **have** ‹*cpt*≠[]› **.**
from *hd-cpt* ‹*cpt*≠[]› **obtain** *cs* **where** *cpt-Cons*: ‹*cpt* = (*fin*, *s0*, *x0*) # *cs*› **by** (*metis hd-Cons-tl*)
from *all-etran-from-fin*[*OF cpt cpt-Cons*] **have** *all-etran*: ‹∀ *i*. *Suc i* < *length cpt* ⟶ *cpt* ! *i* −*e*→ *cpt* ! *Suc i*› **.**
show ‹*cpt* ∈ *commit* (*estran* Γ) {*fin*} *?G ?P*›
proof(*auto simp add*: *commit-def*)
  fix *i*
  assume *Suc-i-lt*: ‹*Suc i* < *length cpt*›
  assume *ctran*: ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∈ *estran* Γ›
  from *all-etran*[*rule-format*, *OF Suc-i-lt*] **have** ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i*› **.**
  from *etran-imp-not-ctran*[*OF this*] **have** ‹(*cpt* ! *i*, *cpt* ! *Suc i*) ∉ *estran* Γ› **.**
  with *ctran* **show** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?G*› **by** *blast*
next
  assume ‹*fst* (*last cpt*) = *fin*›
  have ‹∀ *i*<*length cpt*. *snd* (*cpt*!*i*) ∈ *?P*›
  proof(*auto*)
    fix *i*
    assume *i-lt*: ‹*i* < *length cpt*›
    show ‹*snd* (*cpt* ! *i*) ∈ *?P*›
      using *i-lt*
    proof(*induct i*)
      case *0*
      then show *?case*
        apply(*subst hd-conv-nth*[*symmetric*])
         apply(*rule* ‹*cpt*≠[]›)
        using *cpt-assume* **by** (*simp add*: *assume-def*)
    next
      case (*Suc i*)
      then show *?case*
      proof−
        assume *1*: ‹*i* < *length cpt* ⟹ *snd* (*cpt* ! *i*) ∈ *?P*›
        assume *Suc-i-lt*: ‹*Suc i* < *length cpt*›
        with *1* **have** ‹*snd* (*cpt* ! *i*) ∈ *?P*› **by** *simp*
        from *all-etran*[*rule-format*, *OF Suc-i-lt*] **have** ‹*cpt* ! *i* −*e*→ *cpt* ! *Suc i*› **.**
        with *cpt-assume* **have** ‹(*snd* (*cpt* ! *i*), *snd* (*cpt* ! *Suc i*)) ∈ *?R*›
          apply(*auto simp add*: *assume-def*)
          using *Suc-i-lt* **by** *blast*
        with *stable* **show** ‹*snd* (*cpt* ! *Suc i*) ∈ *?P*›
          apply(*simp add*: *stable-def*)
         using ‹*snd* (*cpt* ! *i*) ∈ *?P*› **by** (*simp add*: *lift-state-set-def lift-state-pair-set-def case-prod-unfold*)
      qed

qed
qed
then show ⟨*snd* (*last cpt*) ∈ *?P*⟩ **using** ⟨*cpt*≠[]⟩
apply−
apply(*subst last-conv-nth*)
apply *assumption*
by *simp*
qed
qed

lemma *Evt-react-list*:
⟨⟦ ∀ *i*<*length* (*rgfs*::((′*l*,′*k*,′*s*,′*prog*) *esys*,′*s*) *rgformula list*). Γ ⊢ *Com* (*rgfs*!*i*) sat_e [*Pre* (*rgfs*!*i*), *Rely* (*rgfs*!*i*), *Guar* (*rgfs*!*i*), *Post* (*rgfs*!*i*)] ∧
pre ⊆ *Pre* (*rgfs*!*i*) ∧ *rely* ⊆ *Rely* (*rgfs*!*i*) ∧
Guar (*rgfs*!*i*) ⊆ *guar* ∧
Post (*rgfs*!*i*) ⊆ *pre*; *rgfs* ≠ [];
stable pre rely; ∀ *s*. (*s*, *s*)∈*guar* ⟧ ⟹
Γ ⊢ *react-sys* (*map Com rgfs*) sat_e [*pre*, *rely*, *guar*, *pre*]⟩
apply (*unfold react-sys-def*)
apply (*rule Evt-While*)
apply *assumption*
apply *fast*
apply *assumption*
apply (*simp add*: *list-of-set-def*)
apply(*rule Evt-OR-list*)
apply *simp*
apply *simp*
apply(*rule allI*)
apply(*rule impI*)
apply(*rule-tac pre*′=⟨*Pre* (*rgfs*!*i*)⟩ **and** *rely*′=⟨*Rely* (*rgfs*!*i*)⟩ **and** *guar*′=⟨*Guar* (*rgfs*!*i*)⟩ **and** *post*′=⟨*Post* (*rgfs*!*i*)⟩ **in** *Evt-conseq*)
apply *simp*+
done

lemma *Evt-react-set*:
⟨⟦ ∀ *rgf* ∈ (*rgfs*::((′*l*,′*k*,′*s*,′*prog*) *esys*,′*s*) *rgformula set*). Γ ⊢ *Com rgf* sat_e [*Pre rgf*, *Rely rgf*, *Guar rgf*, *Post rgf*] ∧
pre ⊆ *Pre rgf* ∧ *rely* ⊆ *Rely rgf* ∧
Guar rgf ⊆ *guar* ∧
Post rgf ⊆ *pre*; *rgfs* ≠ {}; *finite rgfs*;
stable pre rely; ∀ *s*. (*s*, *s*)∈*guar* ⟧ ⟹
Γ ⊢ *react-sys* (*map Com* (*list-of-set rgfs*)) sat_e [*pre*, *rely*, *guar*, *pre*]⟩
apply(*rule Evt-react-list*)
apply(*simp add*: *list-of-set-def*)
apply (*smt finite-list nth-mem tfl-some*)
apply(*simp add*: *list-of-set-def*)
apply (*metis* (*mono-tags*, *lifting*) *empty-set finite-list tfl-some*)
apply *assumption*
apply *assumption*

done

lemma *Evt-react-set'*:
‹⟦ ∀ *rgf* ∈ (*rgfs*::(('*l*,'*k*,'*s*,'*prog*) *esys*,'*s*) *rgformula set*). Γ ⊢ *Com rgf* sat_e [*Pre rgf*, *Rely rgf*, *Guar rgf*, *Post rgf*] ∧
pre ⊆ *Pre rgf* ∧ *rely* ⊆ *Rely rgf* ∧
Guar rgf ⊆ *guar* ∧
Post rgf ⊆ *pre*; *rgfs* ≠ {}; *finite rgfs*;
stable pre rely; ∀ *s*. (*s*, *s*)∈*guar*; *pre* ⊆ *post* ⟧ ⟹
Γ ⊢ *react-sys* (*map Com* (*list-of-set rgfs*)) sat_e [*pre*, *rely*, *guar*, *post*]›
apply(*subgoal-tac* ‹Γ ⊢ *react-sys* (*map Com* (*list-of-set rgfs*)) sat_e [*pre*, *rely*, *guar*, *pre*]›)
using *Evt-conseq* **apply** *blast*
using *Evt-react-set* **apply** *blast*
done

end

end

9 Integrating the SIMP language into Picore

theory *picore-SIMP*
imports *../picore/PiCore-RG-Invariant SIMP-plus ../picore/PiCore-ext*
begin

abbreviation *ptranI* :: '*Env* ⇒ ('*a conf* × '*a conf*) *set*
where *ptranI* Γ ≡ *ctran*

abbreviation *prog-validityI* :: '*Env* ⇒ ('*a com*) *option* ⇒ '*a set* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ⇒ '*a set* ⇒ *bool*
where *prog-validityI* Γ *P* ≡ *prog-validity P*

abbreviation *rghoare-pI* :: '*Env* ⇒ [('*a com*) *option*, '*a set*, ('*a* × '*a*) *set*, ('*a* × '*a*) *set*, '*a set*] ⇒ *bool*
(- $\vdash_I$ - sat_p [-, -, -, -] [*60*,*0*,*0*,*0*,*0*] *45*)
where *rghoare-pI* Γ ≡ *rghoare-p*

lemma *none-no-tranI'*: ((*Q*, *s*),(*P*,*t*)) ∈ *ptranI* Γ ⟹ *Q* ≠ *None*
apply (*simp*) **apply**(*rule ctran.cases*)
by *simp*+

lemma *none-no-tranI*: ((*None*, *s*),(*P*,*t*)) ∉ *ptranI* Γ
using *none-no-tranI'*
by *fast*

lemma *ptran-neqI*: ((*P*, *s*),(*P*,*t*)) ∉ *ptranI* Γ
by (*simp*)

lemma *eventI*: ‹*event ptranI None*›
  apply (*rule event.intro*)
  apply(*rule none-no-tranI*)
  apply(*rule ptran-neqI*)
  done

interpretation *event ptranI None*
  by(*rule eventI*)

lemma *event-compI*: ‹*event-comp ptranI None*›
  apply(*rule event-comp.intro*)
  by(*rule eventI*)

interpretation *event-comp ptranI None*
  by(*rule event-compI*)

lemma *rgsound-pI*: *rghoare-pI* Γ *P pre rely guar post* $\Longrightarrow$ *prog-validityI* Γ *P pre rely guar post*
  using *rgsound-p* **by** *blast*

lemma *cptn-equiv*: ‹*cptn* = *cpts ctran*›
proof
  show ‹*cptn* $\subseteq$ *cpts ctran*›
  proof
    fix *cpt*
    assume ‹*cpt* $\in$ *cptn*›
    then show ‹*cpt* $\in$ *cpts ctran*›
    proof(*induct*, *auto*)
      fix *P s Q t xs*
      assume ‹(P, s) $-c\rightarrow$ (Q, t)›
      moreover assume ‹(Q, t) # *xs* $\in$ *cpts ctran*›
      ultimately show ‹(P, s) # (Q, t) # *xs* $\in$ *cpts ctran*›
        by (*rule CptsComp*)
    qed
  qed
next
  show ‹*cpts ctran* $\subseteq$ *cptn*›
  proof
    fix *cpt*
    assume ‹*cpt* $\in$ *cpts ctran*›
    then show ‹*cpt* $\in$ *cptn*›
    proof(*induct*)
      case (*CptsOne P s*)
      then show *?case* **by** (*rule CptnOne*)
    next
      case (*CptsEnv P t cs s*)
      then show *?case* **using** *CptnEnv* **by** *fast*
    next
      case (*CptsComp P s Q t cs*)

then show *?case*
apply −
apply(*rule CptnComp*, *assumption*+)
done
qed
qed
qed

lemma *etran-equiv-aux*: ‹(P,s) −e→ (Q,t) = (P,s) −e→ (Q,t)›
apply *auto*
apply(*erule etran.cases*, *auto*)
apply(*rule Env*)
done

lemma *etran-equiv*: ‹*c1* −*e*→ *c2* = *c1* −*e*→ *c2*›
using *etran-equiv-aux surjective-pairing* **by** *metis*

lemma *cp-inter-assum-equiv*: ‹*cp P s* ∩ *assum* (*pre*, *rely*) = {*cpt* ∈ *cpts ctran*. *hd cpt* = (*P*, *s*)} ∩ *assume pre rely*›
proof
show ‹*cp P s* ∩ *assum* (*pre*, *rely*) ⊆ {*cpt* ∈ *cpts ctran*. *hd cpt* = (*P*, *s*)} ∩ *assume pre rely*›
proof
fix *cpt*
assume ‹*cpt* ∈ *cp P s* ∩ *assum* (*pre*, *rely*)›
then show ‹*cpt* ∈ {*cpt* ∈ *cpts ctran*. *hd cpt* = (*P*, *s*)} ∩ *assume pre rely*›
apply(*auto simp add*: *cp-def cptn-equiv assum-def assume-def etran-equiv*)
by (*simp add*: *hd-conv-nth cpts-nonnil*)+
qed
next
show ‹{*cpt* ∈ *cpts ctran*. *hd cpt* = (*P*, *s*)} ∩ *assume pre rely* ⊆ *cp P s* ∩ *assum* (*pre*, *rely*)›
proof
fix *cpt*
assume ‹*cpt* ∈ {*cpt* ∈ *cpts ctran*. *hd cpt* = (*P*, *s*)} ∩ *assume pre rely*›
then show ‹*cpt* ∈ *cp P s* ∩ *assum* (*pre*, *rely*)›
apply(*auto simp add*: *cp-def cptn-equiv assum-def assume-def etran-equiv*)
by (*simp add*: *hd-conv-nth cpts-nonnil*)+
qed
qed

lemma *comm-equiv*: ‹*comm* (*guar*, *post*) = *commit ctran* {*None*} *guar post*›
by (*simp add*: *comm-def commit-def*)

lemma *prog-validity-defI*: ‹$\models_I$ *P* sat_p [*pre*, *rely*, *guar*, *post*] ⟹ *validity ctran* {*None*} *P pre rely guar post*›
by (*simp add*: *prog-validity-def cp-inter-assum-equiv comm-equiv*)

interpretation *event-hoare ptranI None prog-validityI rghoare-pI*

apply(*rule event-hoare.intro*)
 apply(*rule event-validity.intro*)
  apply(*rule event-compI*)
 apply(*rule event-validity-axioms.intro*)
 apply(*erule prog-validity-defI*)
apply(*rule event-hoare-axioms.intro*)
using *rgsound-pI* **by** *blast*

end

10 Concrete Syntax of PiCore-SIMP

theory *picore-SIMP-Syntax*
imports *picore-SIMP*

begin

syntax
  -quote :: $'b \Rightarrow ('s \Rightarrow 'b)$ ((«-») [*0*] *1000*)
  -antiquote :: $('s \Rightarrow 'b) \Rightarrow 'b$ (´- [*1000*] *1000*)
  -Assert :: $'s \Rightarrow 's$ *set* ((⦃-⦄) [*0*] *1000*)

translations
  ⦃*b*⦄ ⇀ *CONST Collect* «*b*»

parse-translation ‹
  let
    fun quote-tr [*t*] = *Syntax-Trans.quote-tr* @{*syntax-const -antiquote*} *t*
      | *quote-tr ts* = *raise TERM* (*quote-tr*, *ts*);
  in [(@{*syntax-const -quote*}, *K quote-tr*)] *end*
›

definition *Skip* :: *'s com* (*SKIP*)
  where *SKIP* ≡ *Basic id*

notation *Seq* ((-;;/ -) [*60*,*61*] *60*)

syntax
  -Assign :: *idt* ⇒ *'b* ⇒ *'s com* ((´- :=/ -) [*70*, *65*] *61*)
  -Cond :: *'s bexp* ⇒ *'s com* ⇒ *'s com* ⇒ *'s com* ((*0IF -/ THEN -/ ELSE -/FI*) [*0*, *0*, *0*] *61*)
  -Cond2 :: *'s bexp* ⇒ *'s com* ⇒ *'s com* ((*0IF - THEN - FI*) [*0*,*0*] *62*)
  -While :: *'s bexp* ⇒ *'s com* ⇒ *'s com* ((*0WHILE - /DO - /OD*) [*0*, *0*] *61*)
  -Await :: *'s bexp* ⇒ *'s com* ⇒ *'s com* ((*0AWAIT - /THEN /- /END*)

[0,0] 61)
  -Atom :: 's com ⇒ 's com ((0ATOMIC - END) 61)
  -Wait :: 's bexp ⇒ 's com ((0WAIT - END) 61)
  -For :: 's com ⇒ 's bexp ⇒ 's com ⇒ 's com ⇒ 's com ((0FOR -;/ -;/ -/ DO -/ ROF))
  -Event :: ['a, 'a, 'a] ⇒ ('l,'s,'s com option) event ((EVENT - WHEN - THEN - END) [0,0,0] 61)
  -Event2 :: ['a, 'a] ⇒ ('l,'s,'s com option) event ((EVENT - THEN - END) [0,0] 61)
  -Event-a :: ['a, 'a, 'a] ⇒ ('l,'s,'s com option) event (($EVENT_A$ - WHEN - THEN - END) [0,0,0] 61)
  -Event-a2 :: ['a, 'a] ⇒ ('l,'s,'s com option) event (($EVENT_A$ - THEN - END) [0,0] 61)

translations
  ´x := a ⇀ CONST Basic ≪´(-update-name x (λ-. a))≫
  IF b THEN c1 ELSE c2 FI ⇀ CONST Cond {|b|} c1 c2
  IF b THEN c FI ⇌ IF b THEN c ELSE SKIP FI
  WHILE b DO c OD ⇀ CONST While {|b|} c
  AWAIT b THEN c END ⇌ CONST Await {|b|} c

  ATOMIC c END ⇌ AWAIT CONST True THEN c END
  WAIT b END ⇌ AWAIT b THEN SKIP END
  FOR a; b; c DO p ROF ⇀ a;; WHILE b DO p;;c OD
  EVENT l WHEN g THEN bd END ⇀ CONST EBasic (l, {|g|}, CONST Some bd)
  EVENT l THEN bd END ⇌ EVENT l WHEN CONST True THEN bd END
  $EVENT_A$ l WHEN g THEN bd END ⇀ CONST EAtom (l, {|g|}, CONST Some bd)
  $EVENT_A$ l THEN bd END ⇌ $EVENT_A$ l WHEN CONST True THEN bd END

Translations for variables before and after a transition:

syntax
  -before :: id ⇒ 'a (ᵒ-)
  -after :: id ⇒ 'a (ᵃ-)

translations
  ᵒx ⇌ x ´CONST fst
  ᵃx ⇌ x ´CONST snd

print-translation ‹
```
  let
    fun quote-tr' f (t :: ts) =
          Term.list-comb (f $ Syntax-Trans.quote-tr' @{syntax-const -antiquote} t, ts)
      | quote-tr' - - = raise Match;

    val assert-tr' = quote-tr' (Syntax.const @{syntax-const -Assert});
```

```
  fun bexp-tr' name ((Const (@{const-syntax Collect}, -) $ t) :: ts) =
        quote-tr' (Syntax.const name) (t :: ts)
    | bexp-tr' - - = raise Match;

  fun assign-tr' (Abs (x, -, f $ k $ Bound 0) :: ts) =
      quote-tr' (Syntax.const @{syntax-const -Assign} $ Syntax-Trans.update-name-tr'
f)
          (Abs (x, dummyT, Syntax-Trans.const-abs-tr' k) :: ts)
    | assign-tr' - = raise Match;
 in
  [(@{const-syntax Collect}, K assert-tr'),
   (@{const-syntax Basic}, K assign-tr'),
   (@{const-syntax Cond}, K (bexp-tr' @{syntax-const -Cond})),
   (@{const-syntax While}, K (bexp-tr' @{syntax-const -While}))]
 end
›
```

lemma *colltrue-eq-univ*[*simp*]: ⦃*True*⦄ = *UNIV* **by** *auto*

end
theory *aux-lemma*
imports *Main*
begin

lemma *mod-div-self*: $(a::nat)\ mod\ b = 0 \Longrightarrow (a\ div\ b) * b = a$
by *auto*

lemma *mod-div-mult*: $(a::nat)\ mod\ b = 0 \Longrightarrow a\ div\ b \leq (c - 1) \Longrightarrow a \leq c * b - b$
 apply(*subgoal-tac* $a \leq (c - 1) * b$)
 apply (*simp add*: *left-diff-distrib'*)
 by *fastforce*

lemma *mod0-div-self*: $(a::nat)\ mod\ b = 0 \Longrightarrow b * (a\ div\ b) = a$ **by** *auto*

lemma *m-mod-div*: $n\ mod\ x = 0 \Longrightarrow (m::nat) * n\ div\ x = m * (n\ div\ x)$
 by *auto*

lemma *pow-mod-0*: $x \geq y \Longrightarrow (m::nat)\ \hat{}\ x\ mod\ m\ \hat{}\ y = 0$
 by (*simp add*: *le-imp-power-dvd*)

lemma *ge-pow-mod-0*: $(x::nat) > y \Longrightarrow 4 * n * (4::nat)\ \hat{}\ x\ mod\ 4\ \hat{}\ y = 0$
 by (*metis less-imp-le-nat mod-mod-trivial mod-mult-right-eq mult-0-right pow-mod-0*)

lemma *div2-eq-minus*: $x \neq 0 \wedge m \geq n \Longrightarrow (x::nat)\ \hat{}\ m\ div\ x\ \hat{}\ n = x\ \hat{}\ (m - n)$
 by (*metis add-diff-cancel-left' div-mult-self1-is-m gr0I le-Suc-ex power-add power-not-zero*)

lemma *pow-lt-mod0*: $(n::nat) > 0 \wedge (x::nat) > y \Longrightarrow (n\ \hat{}\ x\ div\ n\ \hat{}\ y)\ mod\ n =$

0
  by (*simp add*: *div2-eq-minus*)

lemma *mod-div-gt*:
$(m::nat) < n \Longrightarrow n \; mod \; x = 0 \Longrightarrow m \; div \; x < n \; div \; x$
  by (*simp add*: *less-mult-imp-div-less mod-div-self*)

lemma *div2-eq-divmul*: $(a::nat) \; div \; b \; div \; c = a \; div \; (b * c)$
  by (*simp add*: *Divides.div-mult2-eq*)

lemma *addr-in-div*:
$(addr::nat) \in \{j2 * M \; ..< (Suc \; j2) * M\} \Longrightarrow addr \; div \; M = j2$
  by (*simp add*: *div-nat-eqI mult.commute*)

lemma *divn-mult-n*: $x > 0 \Longrightarrow (n::nat) = m \; div \; x * x \Longrightarrow (if \; m \; mod \; x = 0 \; then \; m = n \; else \; n < m \wedge m < n + x \wedge n \; mod \; x = 0)$
  apply *auto*
  apply (*metis div-mult-mod-eq less-add-same-cancel1*)
  by (*metis add-le-cancel-left div-mult-mod-eq mod-less-divisor not-less*)

lemma *mod-minus-0*:
$(m::nat) \leq n \wedge 0 < m \Longrightarrow a * (4::nat) \; \hat{} \; n \; mod \; 4 \; \hat{} \; (n - m) = 0$
by (*metis diff-le-self mod-mult-right-eq mod-mult-self2-is-0 mult-0 mult-0-right pow-mod-0*)

lemma *mod-minus-div-4*:
$(m::nat) \leq n \wedge 0 < m \Longrightarrow a * (4::nat) \; \hat{} \; n \; div \; 4 \; \hat{} \; (n - m) \; mod \; 4 = 0$
by (*metis add.left-neutral add-lessD1 diff-less m-mod-div mod-0 mod-mult-right-eq mult-0-right nat-less-le pow-lt-mod0 pow-mod-0 zero-less-numeral*)

lemma *modn0-xy-n*: $(n::nat) > 0 \Longrightarrow x \; mod \; n = 0 \Longrightarrow y \; mod \; n = 0 \Longrightarrow x < y \Longrightarrow x + n \leq y$
  by (*metis Nat.le-diff-conv2 add.commute add.left-neutral add-diff-cancel-left'
    le-less less-imp-add-positive mod-add-left-eq mod-less not-less*)

lemma *divn-multn-addn-le*: $(n::nat) > 0 \Longrightarrow y \; mod \; n = 0 \Longrightarrow x < y \Longrightarrow x \; div \; n * n + n \leq y$
  using *divn-mult-n*[*of n x div n * n x*] *modn0-xy-n*
  apply(*case-tac x mod n = 0*)
    apply(*rule subst*[**where** *s=x* **and** *t=x div n * n*]) **apply** *metis*
    by *auto*

lemma *div-in-suc*: $y > 0 \Longrightarrow n = (x::nat) \; div \; y \Longrightarrow x \in \{n * y \; ..< Suc \; n * y\}$
  by (*simp add*: *dividend-less-div-times*)

lemma *int1-eq*:$P \cap \{V\} \neq \{\} \Longrightarrow P \cap \{V\} = \{V\}$ **by** *auto*

lemma *int1-belong*: $P \cap \{V\} = \{V\} \Longrightarrow V \in P$ **by** *auto*

lemma *two-int-one*: $P \cap \{V\} \cap \{Va\} \neq \{\} \Longrightarrow V = Va \wedge \{V\} = P \cap \{V\} \cap \{Va\}$ **by** *auto*

end
theory *List-aux*
imports *aux-lemma*
begin

primrec *list-updates* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a* ⇒ *'a list* **where**
  list-updates [] *i1 i2 v* = [] |
  list-updates (*x#xs*) *i1 i2 v* =
    (*case i1 of 0* ⇒ (*if i2 > 0 then v # list-updates xs 0* (*i2 − 1*) *v else* (*v#xs*)) |
      Suc j ⇒ (*if i2 > j then x # list-updates xs j* (*i2 − 1*) *v else* (*x#xs*)))

value *list-updates* [*1::nat,2,3,4,5*] *9 0 6*

lemma *length-list-update2* [*simp*]: *length* (*list-updates l i1 i2 v*) = *length l*
  apply(*induct l arbitrary*: *i1 i2 v*)
    apply *simp*
    apply(*case-tac i1*)
      apply(*case-tac i2*) **apply** *simp+*
  done

lemma *list-updates-eq* [*simp*]: ⟦*i1* ≤ *i*; *i* ≤ *i2*; *i2* < *length l*⟧ ⟹ (*list-updates l i1 i2 v*)!*i* = *v*
  apply(*induct l arbitrary*: *i i1 i2 v*)
    apply *simp*
    apply(*case-tac i1*) **apply** *auto*
      apply(*case-tac i2*) **apply** *simp*
    by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-less-SucD diff-Suc-1 le-SucE le-zero-eq not-less-eq-eq nth-Cons' zero-induct*)

lemma *list-updates-neq* [*simp*]: *i* < *i1* ∨ *i* > *i2* ⟹ (*list-updates l i1 i2 v*)!*i* = *l*!*i*
  apply(*induct l arbitrary*: *i i1 i2 v*)
    apply *simp*
    apply(*case-tac i1*) **apply** *simp*
    apply(*case-tac i2*) **apply** *simp* **apply**(*case-tac i*) **apply** *simp+*
  done

lemma *list-updates-beyond*[*simp*]: *i1* ≥ *length l* ⟹ (*list-updates l i1 i2 v*) = *l*
  apply(*induct l arbitrary*: *i1 i2 v*)
    apply *simp* **apply**(*case-tac i1*) **by** *auto*

lemma *list-updates-beyond2*[*simp*]: *i2* < *i1* ⟹ (*list-updates l i1 i2 v*) = *l*
  apply(*induct l arbitrary*: *i1 i2 v*)
    apply *simp* **apply**(*case-tac i1*) **by** *auto*

lemma *list-updates-nonempty*[*simp*]: (*list-updates l i1 i2 v*) = [] ⟷ *l* = []

by (*metis length-greater-0-conv length-list-update2*)

lemma *list-updates-same-conv*:
$i1 < length\ l \wedge i2 < length\ l \Longrightarrow ((list\text{-}updates\ l\ i1\ i2\ v) = l) = (\forall\, i.\ i \geq i1 \wedge i \leq i2 \longrightarrow l\ !\ i = v)$
apply(*induct l arbitrary*: *i1 i2 v*)
apply *simp*
apply(*case-tac* $i1 \leq i2$) **apply**(*rule iffI*)
apply (*metis list-updates-eq*)
apply (*smt length-list-update2 list-updates-eq list-updates-neq not-le-imp-less nth-equalityI*)
by (*metis* (*mono-tags*, *lifting*) *list-updates-beyond2 list-updates-eq not-le-imp-less*)

lemma *list-updates-append1*:
$i2 < length\ l \Longrightarrow list\text{-}updates\ (l\ @\ t)\ i1\ i2\ v = list\text{-}updates\ l\ i1\ i2\ v\ @\ t$
apply(*induct l arbitrary*: *i1 i2 v*)
apply *simp*
apply(*case-tac* $i1 \leq i2$)
apply(*case-tac i1*) **apply** *simp*
apply(*case-tac i2*) **apply** *simp* **apply** *auto*[*1*]
by (*metis list-updates-beyond2 not-less*)

primrec *list-updates-fstn* :: *'a list* ⇒ *nat* ⇒ *'a* ⇒ *'a list* **where**
list-updates-fstn [] *n v* = [] |
list-updates-fstn (*x#xs*) *n v* =
(*case n of 0* ⇒ *x#xs* | *Suc m* ⇒ *v#list-updates-fstn xs m v*)

primrec *list-updates-n* :: *'a list* ⇒ *nat* ⇒ *nat* ⇒ *'a* ⇒ *'a list* **where**
list-updates-n [] *i n v* = [] |
list-updates-n (*x#xs*) *i n v* =
(*case i of 0* ⇒ *list-updates-fstn* (*x#xs*) *n v* | *Suc j* ⇒ *x#list-updates-n xs j n v*)

value *list-updates-n* [*1*::*nat*,*2*,*3*,*4*,*5*] *0 9 6*

lemma *length-list-update-fstn* [*simp*]: *length* (*list-updates-fstn l n v*) = *length l*
apply(*induct l arbitrary*: *n v*)
apply *simp* **apply**(*case-tac n*) **apply** *simp+*
done

lemma *length-list-update-n* [*simp*]: *length* (*list-updates-n l i n v*) = *length l*
apply(*induct l arbitrary*: *i n v*)
apply *simp*
apply(*case-tac i*)
apply(*case-tac n*) **apply** *simp+*
done

lemma *list-updates-fstn-eq* [*simp*]: $[\![i < length\ l;\ i < n]\!] \Longrightarrow$ (*list-updates-fstn l n*

$v)!i = v$
apply(*induct l arbitrary*: *i n v*) **apply** *simp*
apply(*case-tac i*)
apply(*case-tac n*) **apply** *simp+*
apply(*case-tac n*) **apply** *simp+*
done

lemma *list-updates-n-eq* [*simp*]: $[\![i \le j; j < length\ l; j < i + n]\!] \Longrightarrow$ (*list-updates-n* *l i n v*)$!j = v$
apply(*induct l arbitrary*: *i j n v*) **apply** *simp*
apply(*case-tac i*) **apply** *auto*
apply(*case-tac n*) **apply** *auto*
using *less-Suc-eq-0-disj* **by** *auto*

lemma *list-updates-fst0* [*simp*]: *list-updates-fstn l 0 v* = *l*
apply(*induct l arbitrary*: *v*) **by** *simp+*

lemma *list-updates-0* [*simp*]: *list-updates-n l i 0 v* = *l*
apply(*induct l arbitrary*: *i v*) **apply** *simp* **apply**(*case-tac i*) **apply** *simp+*
done

lemma *list-updates-fstn-neq* [*simp*]: $j \ge n \Longrightarrow$ (*list-updates-fstn l n v*)$!j = l!j$
apply(*induct l arbitrary*: *j n v*) **apply** *simp*
apply(*case-tac n*) **apply** *simp+*
done

lemma *list-updates-n-neq* [*simp*]: $j < i \lor j \ge i + n \Longrightarrow$ (*list-updates-n l i n v*)$!j = l!j$
apply(*induct l arbitrary*: *i j n v*) **apply** *simp*
apply(*case-tac i*) **apply**(*case-tac n*) **apply** *simp+*
apply(*case-tac n*) **apply** *simp* **apply**(*case-tac j*) **apply** *simp* **apply** *auto*
done

lemma *list-updates-n-beyond*[*simp*]: $i \ge length\ l \Longrightarrow$ (*list-updates-n l i n v*) = *l*
apply(*induct l arbitrary*: *i n v*)
apply *simp* **apply**(*case-tac i*) **by** *auto*

lemma *lst-udptn-set-eq*: $n > 0 \Longrightarrow$ *list-updates-n* (*lst*[*jj* := *TAG*]) (*jj div n* ∗ *n*) *n TAG1* =
list-updates-n lst (*jj div n* ∗ *n*) *n TAG1*
apply(*rule nth-equalityI*) **apply** *simp*
apply(*case-tac i* = *jj*)
apply(*subgoal-tac i* ≥ *jj div n* ∗ *n*) **prefer** *2* **apply** (*metis divn-mult-n less-or-eq-imp-le*)
apply(*subgoal-tac i* < *jj div n* ∗ *n* + *n*) **prefer** *2*
apply (*metis* (*no-types*) *add.commute dividend-less-div-times*)
apply *simp*
by (*metis length-list-update length-list-update-n list-updates-n-eq list-updates-n-neq not-less nth-list-update-neq*)

thm *list-updates-n.simps*
lemma *list-updates-n-simps2*: *list-updates-n* (*a*#*lst*) (*Suc ii*) *m v* = *a* # *list-updates-n lst ii m v*
by *fastforce*

lemma *list-updates-n-simps2'*: $ii > 0 \Longrightarrow$ *list-updates-n* (*a*#*lst*) *ii m v* = *a* # *list-updates-n lst* (*ii* − *1*) *m v*
using *list-updates-n-simps2*[*of a lst ii* − *1 m v*] **by** *force*

lemma *lst-updt1-eq-upd*: *list-updates-n lst ii 1 v* = *lst*[*ii* := *v*]
  apply(*induct lst arbitrary*: *ii*) **apply** *simp*
  apply(*case-tac ii* = *0*) **apply** *simp*
    using *list-updates-n-simps2'*
    by (*metis One-nat-def Suc-pred list-update-code*(*3*) *neq0-conv*)

lemma *list-neq-udpt-neq*:
$\forall$ *i*<*length l*. *l* ! *i* ≠ *P* $\Longrightarrow$
 l' = *list-updates-n l s n Q* $\Longrightarrow$
 P ≠ *Q* $\Longrightarrow$
 $\forall$ *i*<*length l'*. *l'* ! *i* ≠ *P*
apply(*induct l' arbitrary*:*l*) **apply** *simp*
 by (*metis le-neq-implies-less length-list-update-n list-updates-n-eq list-updates-n-neq nat-le-linear*)

lemma *lst-updts-eq-updts-updt*:
1 ≤ *ii* $\Longrightarrow$
  (*list-updates-n lst st* (*ii* − *1*) *TAG*) [*st* + *ii* − *1* := *TAG*] =
  list-updates-n lst st ii TAG
apply(*rule nth-equalityI*)
  apply *simp*
  apply *clarsimp* **apply**(*rename-tac ia*)
    apply(*case-tac ia* < *st*) **using** *list-updates-n-neq* **apply** *simp*
    apply(*case-tac ia* ≥ *st* + *ii*) **using** *list-updates-n-neq* **apply** *simp*
    apply(*case-tac ia* < *st* + *ii* − *1*) **using** *list-updates-n-eq* **apply** *simp*
    apply(*subgoal-tac ia* = *st* + *ii* − *1*) **prefer** *2*
      apply *force*
    apply(*subgoal-tac length lst* = *length* (*list-updates-n lst st ii TAG*))
      prefer *2* **apply** *simp*
    apply(*subgoal-tac length lst* = *length* (*list-updates-n lst st* (*ii* − *1*) *TAG*))
      prefer *2* **using** *length-list-update-n* **apply** *metis*
    apply(*case-tac ia* ≥ *length lst*) **apply** *linarith*
      apply(*subgoal-tac* (*list-updates-n lst st* (*ii* − *1*) *TAG*) [*st* + *ii* − *1* := *TAG*] ! *ia* = *TAG*) **prefer** *2*
        apply (*metis nth-list-update-eq*)
      apply(*subgoal-tac list-updates-n lst st ii TAG* ! *ia* = *TAG*) **prefer** *2*

apply (*meson list-updates-n-eq not-less*)
using *One-nat-def* **by** *presburger*

primrec *removes* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where *removes* [] *l* = *l* |
removes (*x*#*xs*) *l* = *removes xs* (*remove1 x l*)

lemma *removes-distinct* [*simp*]: *distinct l* ⟹ *distinct* (*removes rs l*)
apply(*induct rs arbitrary*:*l*) **by** *auto*

lemma *removes-length* [*simp*]: ⟦*set rs* ⊆ *set l*; *distinct l*; *distinct rs* ⟧
⟹ *length rs* + *length* (*removes rs l*) = *length l*
apply(*induct rs arbitrary*:*l*)
apply *simp* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred distinct-remove1*
in-set-remove1 length-pos-if-in-set length-remove1 subset-eq)

lemma *removes-empty* [*simp*]: *removes rs* [] = []
apply(*induct rs*) **by** *simp*+

lemma *removes-subs1* [*simp*]: *set* (*removes rs l*) ⊆ *set l*
apply(*induct rs arbitrary*: *l*) **apply** *simp* **apply** *simp*
apply(*subgoal-tac set* (*remove1 a l*) ⊆ *set l*) **apply** *auto*[*1*]
by (*simp add*: *set-remove1-subset*)

lemma *removes-subs2* [*simp*]: *distinct l* ⟹ *set* (*removes* (*a*#*rs*) *l*) ⊆ *set* (*removes rs l*)
apply *simp*
apply(*induct rs arbitrary*: *l a*)
apply *auto* **by** (*metis* (*full-types*) *distinct-remove1 remove1-commute set-mp*)

lemma *removes-nin* [*simp*]: ⟦*x* ∈ *set rs*; *distinct l*⟧ ⟹ *x* ∉ *set* (*removes rs l*)
apply(*induct rs arbitrary*:*l x*)
apply *simp*
apply *simp* **apply** *auto*
by (*metis DiffE contra-subsetD removes-subs1 set-remove1-eq singletonI*)

lemma *rmvs-empty*: *a* ∈ *set es* ⟹ *removes es* [*a*] = []
apply(*induct es*) **apply** *simp* **apply** *auto*
done

lemma *rmvs-unchg*: *a* ∉ *set es* ⟹ *removes es* [*a*] = [*a*]
apply(*induct es*) **apply** *simp* **apply** *auto*
done

lemma *rmvs-onemore-same*:
distinct lst ⟹ *e* ∉ *set lst* ⟹ *removes* (*es*@[*e*]) *lst* = *removes es lst*

apply(*induct es arbitrary*:*lst*)
apply (*simp add*: *remove1-idem*)
apply *auto*
done

lemma *rmvs-rev*: *removes* (*es*@[*e*]) *lst* = *remove1 e* (*removes es lst*)
apply(*induct es arbitrary*:*lst*) **apply** *simp* **apply** *auto*
done

definition *inserts xs l* ≡ *l* @ *xs*

lemma *inserts-set-un*: *set* (*inserts xs l*) = *set xs* ∪ *set l*
 by (*simp add*: *inserts-def sup-commute*)

lemma *inserts-emp1*: *set* (*inserts xs* []) = *set xs*
 using *inserts-set-un*[*of xs* []] **by** *auto*

lemma *inserts-emp2*: *set* (*inserts* [] *l*) = *set l*
 using *inserts-set-un*[*of* [] *l*] **by** *auto*

lemma *list-updt-samelen*: *length l* = *length* (*l*[*jj* := *a*]) **by** *simp*

lemma *list-nhd-in-tl-set*: *el* ∈ *set l* ⟹ *el* ≠ *hd l* ⟹ *el* ∈ *set* (*tl l*)
 by (*metis empty-iff empty-set list.exhaust-sel set-ConsD*)

lemma *dist-hd-nin-tl*: *distinct l* ⟹ *a*∈*set* (*tl l*) ⟹ *a* ≠ *hd l*
 by (*metis distinct.simps*(*2*) *equals0D list.collapse set-empty tl-Nil*)

end

theory *mem-spec*
imports *Main Heap* ../../*adapter-SIMP*/*picore-SIMP* ../../*adapter-SIMP*/*picore-SIMP-Syntax*
List-aux
begin

11 data types and state

typedecl *Thread*

typedef *mempool-ref* = *ref* **by** (*simp add*: *ref-def*)

we define memory address as nat

type-synonym *mem-ref* = *nat*

abbreviation *NULL* ≡ *0* :: *nat*

we have a thread scheduler, thread has 3 types. BLOCKED means a thread is waiting for memory and is in wait queue

datatype *Thread-State-Type* = *READY* | *RUNNING* | *BLOCKED*

a memory block: a ref to a memor pool, a level index and a block index in this level, a start address "data". max number of levels is n_level of a memory pool. So @level should be ¡ n_levels. The number of blocks at level 0 is n_max. the max number of blocks at level n is $n\_max * 4^n$. the block index should less then this number.

record *Mem-block* = *pool* :: *mempool-ref*
level :: *nat*
block :: *nat*
data :: *mem-ref*

BlockState defines the bit info in bitmap. We uses different types, while not 0 or 1 in this design. Then the blockstate could be implemented as 0 or 1, with additional information.

basic states of memory block are ALLOCATED, FREE, DIVIDED and NOEXIST. The levels of bitmap is actually a quad-tree of BlockState. ALLOCATED: the block is allocated to a thread FREE: the block is free DIVIDED: the block is divided, which means is was splited to 4 subblocks NOEXIST: the block is not exist

ALLOCACTED and FREE blocks are the leaf blocks of the quad-tree. DIVIDED blocks are inner nodes of the quad-tree. Otherwise is NOEXIST.

we also introduce FREEING and ALLOCATING state to avoid a case that a FREEING block may be allocated by other threads and a ALLOCATING block may be freed by other threads. In OS implementation, the allocating/freeing block is an inner block of alloc/free services, and other threads will not manipulate them. they are used to indicate state of the block which are going to be merged during freeing a block, and the block which is going to be split during allocating a block.

we may remove FREEING/ALLOCATING state later by revising alloc and free syscalls to avoid allocate or free blocks in freeing_node and allocating_node.

datatype *BlockState* = *ALLOCATED* | *FREE* | *DIVIDED* | *NOEXIST* | *FREEING* | *ALLOCATING*

data stucture at each level, a bitmap and a free block list

record *Mem-pool-lvl* =
bits :: *BlockState list*
free-list :: *mem-ref list*

a memory pool is actually a forest of @n_max numbers of blocks with size of @max_sz. A block may be split to 4 sub-blocks and so on, at most for @n_levels times. Thus, each block may be split as a quad-tree. a memory pool maintains a big memory block, where @buf is the start address of the memory block. The size of a memory pool is @n_max * @max_sz. @max_sz has a constraint. a small block at last level (level index is @n_levels - 1) should be aligned by 4 bits, i.e. the size of block at last level should be 4*n (n ¿ 0). Here, we dont demand 4^n, which is a special case of 4*n. Thus, @max_sz should be $4 * n * 4^n\_levels$.

@levels maintain the information at each level including a bitmap and a free block list. @wait_q is a list of threads, which is blocked on this memory pool.

record *Mem-pool* = *buf* :: *mem-ref*
max-sz :: *nat*
n-max :: *nat*
n-levels :: *nat*

levels :: *Mem-pool-lvl list*
wait-q :: *Thread list*

The state of memory management consists of thread state, memory pools, and local variables of each thread. In monocore OSs, there is only one currently executing thread @cur, where None means the scheduler has not choose a thread. @tick save a time for the system. @mem_pools maintains the refs of all memory pools. @mem_pool_info shows the detailed information of each memory pool by its ref. we assume that all memory pools are shared by all threads. This is the most relaxed case. The case that some memory pool is only shared by a set of thread is just a special case. Other fields are local vars of each thread used in alloc/free syscalls.

for each thread, we use freeing_node to maintain the freeing node in free syscall. when free a block, we set it to FREEING, and check if its other 3 partner blocks are also free. If so, we set the 4 blocks to NOEXIST and set their parent block to FREEING, and so on. until that other 3 partner blocks are not all free, then set the FREEING block to FREE. This design avoids the FREEING node is allocated by other threads.

we use allocating_node to maintain the allocating node in alloc syscall. when alloc a block, we find a free block at the nearest upper level, and set it to ALLOCATING. if size of the block is too big, we split it into 4 child blocks. We set the first child block to ALLOCATING and other 3 blocks to FREE, and so on. until that the size of block is suitable, then set the ALLOCATING block to ALLOCATED. This design avoids the ALLOCATING node is freed by other threads.

record *State* =

cur :: *Thread option*
tick :: *nat*
thd-state :: *Thread* ⇒ *Thread-State-Type*

mem-pools :: *mempool-ref set*

mem-pool-info :: *mempool-ref* ⇒ *Mem-pool*

i :: *Thread* ⇒ *nat*
j :: *Thread* ⇒ *nat*
ret :: *Thread* ⇒ *int*
endt :: *Thread* ⇒ *nat*
rf :: *Thread* ⇒ *bool*
tmout :: *Thread* ⇒ *int*
lsizes :: *Thread* ⇒ *nat list*
alloc-l :: *Thread* ⇒ *int*
free-l :: *Thread* ⇒ *int*
from-l :: *Thread* ⇒ *int*
blk :: *Thread* ⇒ *mem-ref*
nodev :: *Thread* ⇒ *mem-ref*
bn :: *Thread* ⇒ *nat*
lbn :: *Thread* ⇒ *nat*
lsz :: *Thread* ⇒ *nat*
block2 :: *Thread* ⇒ *mem-ref*
free-block-r :: *Thread* ⇒ *bool*
alloc-lsize-r :: *Thread* ⇒ *bool*
lvl :: *Thread* ⇒ *nat*
bb :: *Thread* ⇒ *nat*
block-pt :: *Thread* ⇒ *mem-ref*
th :: *Thread* ⇒ *Thread*
need-resched :: *Thread* ⇒ *bool*
mempoolalloc-ret :: *Thread* ⇒ *Mem-block option*

freeing-node :: *Thread* ⇒ *Mem-block option*
allocating-node :: *Thread* ⇒ *Mem-block option*

12 specification of events

12.1 data types

Since Zephyr uses fine-grained locks for shared memory pools, interleaving among scheduling, syscalls (alloc, free), and clock tick are allowed. Thus, we use 3 event systems to model scheduling, syscalls from threads, and clock tick. Then the whole system is the parallel composition of the three event systems. Actually, we have 1 scheduler, 1 timer, and n threads.

datatype *Core* = $\mathcal{S}$ | $\mathcal{T}$ *Thread* | *Timer* labels for different events

datatype $EL = ScheduleE \mid TickE \mid Mem\text{-}pool\text{-}allocE \mid Mem\text{-}pool\text{-}freeE \mid Mem\text{-}pool\text{-}defineE$

data types for event parameters

datatype $Parameter = Thread\ Thread \mid MPRef\ mempool\text{-}ref \mid MRef\ mem\text{-}ref \mid Block\ Mem\text{-}block \mid Natural\ nat \mid Integer\ int$

type-synonym $EventLabel = EL \times (Parameter\ list \times Core)$

definition $get\text{-}evt\text{-}label :: EL \Rightarrow Parameter\ list \Rightarrow Core \Rightarrow EventLabel$ (- - ⇛ - [30,30,30] 20)
 where $get\text{-}evt\text{-}label\ el\ ps\ k \equiv (el,(ps,k))$

define the waiting mode for alloc. FOREVER means that if allocating fails, the thread will wait forever until allocating succeed. NOWAIT means that if allocating fails, alloc syscall return error immediately. otherwise n ¿ 0, means the thread will wait for a timeout n.

abbreviation $FOREVER \equiv (-1)::int$
abbreviation $NOWAIT \equiv 0::int$

return CODE for alloc and free syscalls. free syscall always succeed, so it returns OK. alloc syscall may succeed (OK), timeout (ETIMEOUT), fails(ENOMEM), fails due to request too large size (ESIZEERR).

EAGAIN is an inner flag of alloc syscall. After it finds an available block for request, the block may be allocated immediately by other threads. In such a case, alloc will provide EAGAIN and try to allocate again.

We introduce ESIZEERR for Zephyr to avoid a dead loop. We introduce ETIMEOUT for Zephyr for robustness.

abbreviation $EAGAIN \equiv (-2)::int$
abbreviation $ENOMEM \equiv (-3)::int$
abbreviation $ESIZEERR \equiv (-4)::int$
abbreviation $OK \equiv 0 :: int$
abbreviation $ETIMEOUT \equiv (-1) :: int$

due to fine-grained lock used by Zephyr, we use a command for each atomic statement in free/alloc syscalls. the statements of syscalls from a thread t can only be executed when t is the currently executing thread by the scheduler. We use the AWAIT statement to represent this semantics.

definition $stm :: Thread \Rightarrow State\ com \Rightarrow State\ com$ (- ▶ - [0,0] 21)
where $stm\ t\ p = AWAIT\ {}'cur = Some\ t\ THEN\ p\ END$

12.2 aux definitions for events

definition $ALIGN4 :: nat \Rightarrow nat$
where $ALIGN4\ n \equiv ((n + 3)\ div\ 4) * 4$

lemma *align40*: $n\ mod\ 4 = 0 \Longrightarrow ALIGN4\ n = n$
  unfolding *ALIGN4-def* **by** *auto*

lemma *align41*: $n\ mod\ 4 = 1 \Longrightarrow ALIGN4\ n = n + 3$
  unfolding *ALIGN4-def*
proof −
  assume $n\ mod\ 4 = 1$
  then have $(n + 3)\ mod\ 4 = 0$
    by *presburger*
  then show $(n + 3)\ div\ 4 * 4 = n + 3$
    by *fastforce*
qed

lemma *align42*: $n\ mod\ 4 = 2 \Longrightarrow ALIGN4\ n = n + 2$
  unfolding *ALIGN4-def*
proof −
  assume $n\ mod\ 4 = 2$
  then have $(n + 2)\ mod\ 4 = 0$
    using *mod-add-left-eq* **by** *presburger*
  then show $(n + 3)\ div\ 4 * 4 = n + 2$
    by *fastforce*
qed

lemma *align43*: $n\ mod\ 4 = 3 \Longrightarrow ALIGN4\ n = n + 1$
  unfolding *ALIGN4-def*
proof −
  assume $n\ mod\ 4 = 3$
  then have $(n + 1)\ mod\ 4 = 0$
    using *mod-add-left-eq* **by** *presburger*
  then show $(n + 3)\ div\ 4 * 4 = n + 1$
    by *fastforce*
qed

lemma *align-mod0*: $ALIGN4\ n\ mod\ 4 = 0$
  unfolding *ALIGN4-def* **by** *simp*

lemma *align4-gt*: $ALIGN4\ n \geq n \wedge ALIGN4\ n \leq n + 3$
  apply(*case-tac* $n\ mod\ 4 = 0$)
    using *align40* **apply** *simp*
  apply(*case-tac* $n\ mod\ 4 = 1$)
    using *align41* **apply** *simp*
  apply(*case-tac* $n\ mod\ 4 = 2$)
    using *align42* **apply** *simp*
  apply(*case-tac* $n\ mod\ 4 = 3$)
    using *align43* **apply** *simp*
  by *auto*

lemma *align2-eq-align*: $ALIGN4\ (ALIGN4\ n) = ALIGN4\ n$
  unfolding *ALIGN4-def* **by** *auto*

Zephyr uses two events: reschedule for free and swap for alloc for context switch

definition *reschedule* :: *State com*
where *reschedule* ≡
  ´thd-state := *´thd-state*(*the ´cur* := *READY*);;
  ´cur := *Some* (*SOME t*. *´thd-state t* = *READY*);;
  ´thd-state := *´thd-state*(*the ´cur* := *RUNNING*)

definition *swap* :: *State com*
where *swap* ≡
  IF (∃ *t*. *´thd-state t* = *READY*) *THEN*
    ´cur := *Some* (*SOME t*. *´thd-state t* = *READY*);;
    ´thd-state := *´thd-state*(*the ´cur* := *RUNNING*)
  ELSE
    ´cur := *None*
  FI

definition *block-num* :: *Mem-pool* ⇒ *mem-ref* ⇒ *nat* ⇒ *nat*
where *block-num p bl sz* ≡ (*bl* − (*buf p*)) *div sz*

definition *clear-free-bit* :: (*mempool-ref* ⇒ *Mem-pool*) ⇒ *mempool-ref* ⇒ *nat* ⇒ *nat* ⇒ (*mempool-ref* ⇒ *Mem-pool*)
where *clear-free-bit mp-info p l b* ≡
    mp-info (*p* := (*mp-info p*) (|*levels* := (*levels* (*mp-info p*))
        [*l* := ((*levels* (*mp-info p*)) ! *l*) (|*bits* := (*bits* ((*levels* (*mp-info p*)) ! *l*)) [*b* := *ALLOCATED*]|)] |))

definition *set-bit* :: (*mempool-ref* ⇒ *Mem-pool*) ⇒ *mempool-ref* ⇒ *nat* ⇒ *nat* ⇒ *BlockState* ⇒ (*mempool-ref* ⇒ *Mem-pool*)
where *set-bit mp-info p l b st* ≡
    mp-info (*p* := (*mp-info p*) (|*levels* := (*levels* (*mp-info p*))
        [*l* := ((*levels* (*mp-info p*)) ! *l*) (|*bits* := (*bits* ((*levels* (*mp-info p*)) ! *l*)) [*b* := *st*]|)] |))

abbreviation *set-bit-free mp-info p l b* ≡ *set-bit mp-info p l b FREE*
abbreviation *set-bit-alloc mp-info p l b* ≡ *set-bit mp-info p l b ALLOCATED*
abbreviation *set-bit-divide mp-info p l b* ≡ *set-bit mp-info p l b DIVIDED*
abbreviation *set-bit-noexist mp-info p l b* ≡ *set-bit mp-info p l b NOEXIST*
abbreviation *set-bit-freeing mp-info p l b* ≡ *set-bit mp-info p l b FREEING*
abbreviation *set-bit-allocating mp-info p l b* ≡ *set-bit mp-info p l b ALLOCATING*

definition *set-bit-s* :: *State* ⇒ *mempool-ref* ⇒ *nat* ⇒ *nat* ⇒ *BlockState* ⇒ *State*
where *set-bit-s s p l b st* ≡
    s(|*mem-pool-info* := *set-bit* (*mem-pool-info s*) *p l b st* |)

lemma *set-bit-prev-len*:
length (*bits* (*levels* (*mp-info p*) ! *l*)) = *length* (*bits* (*levels* ((*set-bit mp-info p l b flg*) *p*) ! *l*))
  apply(*simp add*:*set-bit-def*)

using *list-updt-samelen*
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.surjective* *Mem-pool-lvl.update-convs*(*1*) *list-update-beyond not-less nth-list-update-eq*)

lemma *set-bit-prev-len2*:
$l \neq t \Longrightarrow$ *length* (*bits* (*levels* (*mp-info p*) ! *l*)) = *length* (*bits* (*levels* ((*set-bit mp-info p t b flg*) *p*) ! *l*))
by(*simp add*:*set-bit-def*)

abbreviation *get-bit* :: (*mempool-ref* $\Rightarrow$ *Mem-pool*) $\Rightarrow$ *mempool-ref* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *BlockState*
where *get-bit mp-info p l b* $\equiv$ (*bits* ((*levels* (*mp-info p*)) ! *l*)) ! *b*

abbreviation *get-bit-s* :: *State* $\Rightarrow$ *mempool-ref* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *BlockState*
where *get-bit-s s p l b* $\equiv$ *get-bit* (*mem-pool-info s*) *p l b*

lemma *set-bit-get-bit-eq*:
$l <$ *length* (*levels* (*mp-info p*)) $\Longrightarrow$
$b <$ *length* (*bits* (*levels* (*mp-info p*) ! *l*)) $\Longrightarrow$
mp-info2 = *set-bit mp-info p l b st* $\Longrightarrow$
get-bit mp-info2 p l b = *st*
by (*simp add*:*set-bit-def*)

lemma *set-bit-get-bit-eq2*:
$l <$ *length* (*levels* ((*mem-pool-info Va*) *p*)) $\Longrightarrow$
$b <$ *length* (*bits* (*levels* ((*mem-pool-info Va*) *p*) ! *l*)) $\Longrightarrow$
get-bit-s (*Va*(| *mem-pool-info* := *set-bit* (*mem-pool-info Va*) *p l b st*|)) *p l b* = *st*
using *set-bit-get-bit-eq*
[*of l* (*mem-pool-info Va*) *p b set-bit* (*mem-pool-info Va*) *p l b st st*]
by *simp*

lemma *set-bit-get-bit-neq*:
$p \neq p1 \vee l \neq l1 \vee b \neq b1 \Longrightarrow$
mp-info2 = *set-bit mp-info p l b st* $\Longrightarrow$
get-bit mp-info2 p1 l1 b1 = *get-bit mp-info p1 l1 b1*
apply(*simp add*:*set-bit-def*) **apply** *auto*
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.surjective* *Mem-pool-lvl.update-convs*(*1*) *list-update-beyond not-less nth-list-update-eq* *nth-list-update-neq*)

lemma *set-bit-get-bit-neq2*:
$p \neq p1 \vee l \neq l1 \vee b \neq b1 \Longrightarrow$
get-bit-s (*Va*(| *mem-pool-info* := *set-bit* (*mem-pool-info Va*) *p l b st*|)) *p1 l1 b1*
= *get-bit-s Va p1 l1 b1*
using *set-bit-get-bit-neq*
[*of p p1 l l1 b b1 set-bit* (*mem-pool-info Va*) *p l b st mem-pool-info Va*]
by *simp*

definition *buf-size* :: *Mem-pool* ⇒ *nat*
where *buf-size m* ≡ *n-max m* ∗ *max-sz m*

definition *block-fits* :: *Mem-pool* ⇒ *mem-ref* ⇒ *nat* ⇒ *bool*
where *block-fits p b bsz* ≡ *b* + *bsz* < *buf-size p* + *buf p* + *1*

definition *block-ptr* :: *Mem-pool* ⇒ *nat* ⇒ *nat* ⇒ *mem-ref*
where *block-ptr p lsize b* ≡ *buf p* + *lsize* ∗ *b*

definition *partner-bits* :: *Mem-pool* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *partner-bits p l b* ≡ *let bits* = *bits* (*levels p* ! *l*);
a = (*b div 4*) ∗ *4 in*
bits!*a* = *FREE* ∧ *bits*!(*a*+*1*) = *FREE* ∧ *bits*!(*a*+*2*) = *FREE* ∧ *bits*!(*a*+*3*) = *FREE*

lemma *partbits-div4*: *a div 4* = *b div 4* ⟹ *partner-bits p l a* = *partner-bits p l b*
by(*simp add:partner-bits-def*)

abbreviation *noexist-bits* :: *Mem-pool* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *noexist-bits mp ii jj* ≡ (*bits* (*levels mp* ! *ii*)) ! *jj* = *NOEXIST*
∧ (*bits* (*levels mp* ! *ii*)) ! (*jj* + *1*) = *NOEXIST*
∧ (*bits* (*levels mp* ! *ii*)) ! (*jj* + *2*) = *NOEXIST*
∧ (*bits* (*levels mp* ! *ii*)) ! (*jj* + *3*) = *NOEXIST*

definition *level-empty* :: *Mem-pool* ⇒ *nat* ⇒ *bool*
where *level-empty p n* ≡ *free-list* (*levels p*!*n*) = []

definition *head-free-list* :: *Mem-pool* ⇒ *nat* ⇒ *mem-ref*
where *head-free-list p l* ≡ *hd* (*free-list* ((*levels p*) ! *l*))

definition *rmhead-free-list* :: *Mem-pool* ⇒ *nat* ⇒ *Mem-pool*
where *rmhead-free-list p l* ≡
p(|*levels* := (*levels p*)
[*l* := ((*levels p*) ! *l*) (|*free-list* := *tl* (*free-list* ((*levels p*) ! *l*))|)] |)

definition *remove-free-list* :: *Mem-pool* ⇒ *nat* ⇒ *mem-ref* ⇒ *Mem-pool*
where *remove-free-list p l b* ≡
p(|*levels* := (*levels p*)
[*l* := ((*levels p*) ! *l*) (|*free-list* := *remove1 b* (*free-list* ((*levels p*) ! *l*))|)] |)

definition *append-free-list* :: *Mem-pool* ⇒ *nat* ⇒ *mem-ref* ⇒ *Mem-pool*
where *append-free-list p l b* ≡
p(|*levels* := (*levels p*)
[*l* := ((*levels p*) ! *l*) (|*free-list* := (*free-list* ((*levels p*) ! *l*)) @ [*b*]|)] |)

definition *in-free-list* :: *mem-ref* ⇒ *mem-ref list* ⇒ *bool*
where *in-free-list v fl* ≡ (∃ *i*<*length fl*. *fl*!*i* = *v*)

12.3 specification of events

lemma *timeout-lm*: (*timeout* = *FOREVER* ∨ *timeout* = *NOWAIT* ∨ *timeout* > *0*) = (*timeout* ≥ −*1*)
  by *auto*

definition *Mem-pool-alloc* :: *Thread* ⇒ *mempool-ref* ⇒ *nat* ⇒ *int* ⇒ (*EventLabel*, ′*a*, *State*, *State com option*) *esys*
where *Mem-pool-alloc t p sz timeout* =
  EVENT Mem-pool-allocE [*MPRef p*, *Natural sz*, *Integer timeout*] ⇛ (𝒯 *t*)
  WHEN
    p ∈ ´*mem-pools*
    ~~(* ∧ ´cur = Some t*) (* t is the current thread *) (** this condition is not stable on rely condition **)~~
    ∧ *timeout* ≥ −*1* ~~(* equv to (timeout = FOREVER ∨ timeout = NOWAIT ∨ timeout > 0) *)~~
    ~~(* ∧ p ∈ ´pools-of-thread t *) (* the mem pool p is shared in the thread t *)~~
  THEN
    (*t* ▶ ´*tmout* := ´*tmout*(*t* := *timeout*));;

    (*t* ▶ ´*endt* := ´*endt*(*t* := *0*));;
    (*t* ▶ *IF timeout* > *0 THEN*
        ´*endt* := ´*endt*(*t* := ´*tick* + *nat timeout*)
      FI);;
    (*t* ▶ ´*mempoolalloc-ret* := ´*mempoolalloc-ret* (*t* := *None*));;
    (*t* ▶ ´*ret* := ´*ret*(*t* := *ESIZEERR*));;
    (*t* ▶ ´*rf* := ´*rf*(*t* := *False*));;
    WHILE ¬ (´*rf t*) *DO*

    ~~(* ===== start: ret = pool-alloc(p, block, size); ============================== *)~~
      ~~(*(t ▶ ´lsizes := ´lsizes(t := [])));*)~~
      (*t* ▶ ´*blk* := ´*blk*(*t* := *NULL*));;
      (*t* ▶ ´*alloc-lsize-r* := ´*alloc-lsize-r*(*t* := *False*));;
      (*t* ▶ ´*alloc-l* := ´*alloc-l*(*t* := −*1*));;
      (*t* ▶ ´*free-l* := ´*free-l*(*t* := −*1*));;
      (*t* ▶ ´*lsizes* := ´*lsizes*(*t* := [*ALIGN4* (*max-sz* (´*mem-pool-info p*))]));;
      (*t* ▶ ´*i* := ´*i*(*t* := *0*));;
      WHILE ´*i t* < *n-levels* (´*mem-pool-info p*) ∧ ¬ ´*alloc-lsize-r t DO*
        IF ´*i t* > *0 THEN*
          (*t* ▶ ´*lsizes* := ´*lsizes*(*t* := ´*lsizes t* @ [*ALIGN4* (´*lsizes t* ! (´*i t* − *1*) *div 4*)]))
        FI;;
        IF ´*lsizes t* ! ´*i t* < *sz THEN*
          (*t* ▶ ´*alloc-lsize-r* := ´*alloc-lsize-r*(*t* := *True*))
        ELSE
          (*t* ▶ ´*alloc-l* := ´*alloc-l*(*t* := *int* (´*i t*)));;
          IF ¬ *level-empty* (´*mem-pool-info p*) (´*i t*) *THEN*
            (*t* ▶ ´*free-l* := ´*free-l*(*t* := *int* (´*i t*)))
          FI;;

(t ► ´i := ´i(t := ´i t + 1))
FI
OD;;

IF ´alloc-l t < 0 THEN
(t ► ´ret := ´ret(t := ESIZEERR))
ELSE
IF ´free-l t < 0 THEN
~~(* block->data = NULL; *)~~
(t ► ´ret := ´ret(t := ENOMEM))
ELSE
~~(* ##### start: blk = alloc-block(p, free-l, lsizes[free-l]); *)~~
(t ► ATOMIC
~~(* ##### start: block = sys-dlist-get(&p->levels[l].free-list); *)~~
IF level-empty (´mem-pool-info p) (nat (´free-l t)) THEN
´blk := ´blk(t := NULL)
ELSE
´blk := ´blk(t := head-free-list (´mem-pool-info p) (nat (´free-l t)));;

~~(* sys-dlist-remove(node); *)~~
´mem-pool-info := ´mem-pool-info (p := rmhead-free-list (´mem-pool-info p) (nat (´free-l t)))

FI;;
~~(* ##### end: block = sys-dlist-get(&p->levels[l].free-list); *)~~

IF ´blk t ≠ NULL THEN
~~(* clear-free-bit(p, l, block-num(p, block, lsz)); *)~~
´mem-pool-info := set-bit-allocating ´mem-pool-info p (nat (´free-l t))
(block-num (´mem-pool-info p) (´blk t) ((´lsizes t)!(nat (´free-l t))));;
~~(* set the allocating node info of the thread *)~~
´allocating-node := ´allocating-node (t := Some (|pool = p, level = nat (´free-l t),
block = (block-num (´mem-pool-info p) (´blk t) ((´lsizes t)!(nat (´free-l t)))), data = ´blk t |))
FI
END);;
~~(* ##### end: blk = alloc-block(p, free-l, lsizes[free-l]); *)~~

IF ´blk t = NULL THEN
(t ► ´ret := ´ret (t := EAGAIN))
ELSE

FOR (t ► ´from-l := ´from-l(t := ´free-l t));
~~(* level-empty (´mem-pool-info p) (nat (´alloc-l t)) \ *)~~ ´from-l t < ´alloc-l t;
~~(********** we remove the FOR termination condition "level-empty" to remove a concurrency BUG here ***********)~~

(t ► ´from-l := ´from-l(t := ´from-l t + 1)) DO

~~(* ###### start: blk = break-block(p, blk, from-l, lsizes); *)~~
(t ► ATOMIC
´bn := ´bn (t := block-num (´mem-pool-info p) (´blk t) ((´lsizes t)!(nat (´from-l t))));;

´mem-pool-info := set-bit-divide ´mem-pool-info p (nat (´from-l t)) (´bn t);;

´mem-pool-info := set-bit-allocating ´mem-pool-info p (nat (´from-l t + 1)) (4 * ´bn t);;

~~(* set the allocating node info of the thread *)~~
´allocating-node := ´allocating-node (t := Some (|pool = p, level = nat (´from-l t + 1),
block = 4 * ´bn t, data = ´blk t |));;

FOR ´i := ´i (t := 1);
´i t < 4;
´i := ´i (t := ´i t + 1) DO
´lbn := ´lbn (t := 4 * ´bn t + ´i t);;
´lsz := ´lsz (t := (´lsizes t) ! (nat (´from-l t + 1)));;
´block2 := ´block2(t := ´lsz t * ´i t + ´blk t);;

~~(* set-free-bit(p, l + 1, lbn); *)~~
´mem-pool-info := set-bit-free ´mem-pool-info p (nat (´from-l t + 1)) (´lbn t);;

IF block-fits (´mem-pool-info p) (´block2 t) (´lsz t) THEN

~~(* sys-dlist-append(&p->levels[l + 1].free-list, block2); *)~~
´mem-pool-info := ´mem-pool-info (p :=
append-free-list (´mem-pool-info p) (nat (´from-l t + 1)) (´block2 t))
FI
ROF

END)
~~(* ###### end: blk = break-block(p, blk, from-l, lsizes); *)~~

ROF;;

~~(* finally set the node from allocating to allocated and remove the allocating node info of the thread *)~~
(t ► ´mem-pool-info := set-bit-alloc ´mem-pool-info p (nat (´alloc-l t))
(block-num (´mem-pool-info p) (´blk t) ((´lsizes t)!(nat (´alloc-l t))));;
´allocating-node := ´allocating-node (t := None)

);;

(t ► ´mempoolalloc-ret := ´mempoolalloc-ret (t :=
Some (|pool = p, level = nat (´alloc-l t),
block = block-num (´mem-pool-info p) (´blk t) ((´lsizes t)!(nat (´alloc-l t))),
data = ´blk t |)));;

(t ► ´ret := ´ret (t := OK))
FI
FI
FI;;
~~(* ###### end: ret = pool-alloc(p, block, size); ########################## *)~~

~~(* IF ´ret t = 0 ∨ timeout = NOWAIT ∨ ´ret t = EAGAIN ∨ ´ret t = ENOMEM THEN *)~~ ~~(***** we change the IF condition to remove a functional BUG here *****)~~
IF ´ret t = OK ∨ timeout = NOWAIT ∨ ´ret t = ESIZEERR THEN
(t ► ´rf := ´rf(t := True));;
IF ´ret t = EAGAIN THEN ~~(*EAGAIN should not export to users*)~~
(t ► ´ret := ´ret(t := ENOMEM))
FI
ELSE
IF ´ret t = EAGAIN THEN SKIP
ELSE
(t ► ATOMIC

~~(* pend-current-thread(&p->wait-q, timeout); *)~~
´thd-state := ´thd-state(the ´cur := BLOCKED);;
~~(* cur := None;; *)~~
´mem-pool-info := ´mem-pool-info(p := ´mem-pool-info p(|wait-q := wait-q (´mem-pool-info p) @ [the ´cur] |));;

~~(* Swap(key); *)~~
swap

END);;

IF ´tmout t ≠ FOREVER THEN
(t ► ´tmout := ´tmout (t := int (´endt t) − int ´tick));;
IF ´tmout t < 0 THEN
(t ► ´rf := ´rf(t := True));;
(t ► ´ret := ´ret (t := ETIMEOUT))
FI
FI
FI
FI
OD

END

definition *Mem-pool-free* :: *Thread* ⇒ *Mem-block* ⇒ (*EventLabel*, *'a*, *State*, *State com option*) *esys*
where *Mem-pool-free t b* =
  EVENT Mem-pool-freeE [*Block b*] ⤃ (𝒯 *t*)
  WHEN
    pool b ∈ ´*mem-pools*
    ∧ *level b* < *length* (*levels* (´*mem-pool-info* (*pool b*)))
    ∧ *block b* < *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*))!(*level b*)))
   ∧ *data b* = *block-ptr* (´*mem-pool-info* (*pool b*)) ((*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ^ (*level b*))) (*block b*)
   ~~(* ∧ (bits ((levels (´mem-pool-info (pool b)))!(level b)))!(block b) = ALLOCATED ∧ ´cur = Some t *) (* t is the current thread *) (* ∧ pool b ∈ ´pools-of-thread t *) (* the mem pool is shared in the thread t *)~~
  THEN
    ~~(* here we set the bit to FREEING, so that other thread cannot mem-pool-free the same block. it also requires that it can only free ALLOCATED block *)~~
    (*t* ▶ *AWAIT* (*bits* ((*levels* (´*mem-pool-info* (*pool b*))) ! (*level b*))) ! (*block b*) = *ALLOCATED THEN*
            ´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (*level b*) (*block b*);;
            ´*freeing-node* := ´*freeing-node* (*t* := *Some b*) ~~(* set the freeing node of current thread *)~~
         END);;

    (*t* ▶ ´*need-resched* := ´*need-resched*(*t* := *False*));;
    ~~(* (t ▶ ´lsizes := ´lsizes(t := []));; *)~~
    (*t* ▶ ´*lsizes* := ´*lsizes*(*t* := [*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))]));;
    FOR (*t* ▶ ´*i* := ´*i*(*t* := *1*));
        ´*i t* ≤ *level b*;
        (*t* ▶ ´*i* := ´*i*(*t* := ´*i t* + *1*)) *DO*
      (*t* ▶ ´*lsizes* := ´*lsizes*(*t* := ´*lsizes t* @ [*ALIGN4* (´*lsizes t* ! (´*i t* − *1*) *div 4*)]))
    ROF;;

    ~~(* ### start: free-block(get-pool(block->id.pool), block->id.level, lsizes, block->id.block); *)~~
    (*t* ▶ ´*free-block-r* := ´*free-block-r* (*t* := *True*));;
    (*t* ▶ ´*bn* := ´*bn* (*t* := *block b*));;
    (*t* ▶ ´*lvl* := ´*lvl* (*t* := *level b*));;

    WHILE ´*free-block-r t DO*
      (*t* ▶ ´*lsz* := ´*lsz* (*t* := ´*lsizes t* ! (´*lvl t*)));;
      (*t* ▶ ´*blk* := ´*blk* (*t* := *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bn t*)));;

      (*t* ▶ *ATOMIC*

´mem-pool-info := set-bit-free ´mem-pool-info (pool b) (´lvl t) (´bn t);;
´freeing-node := ´freeing-node (t := None);; ~~(* remove the freeing node info of the thread *)~~

IF ´lvl t > 0 ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t) THEN
FOR ´i := ´i(t := 0);
´i t < 4;
´i := ´i(t := ´i t + 1) DO
´bb := ´bb (t := (´bn t div 4) * 4 + ´i t);;
~~(*(t ▶ ´mem-pool-info := clear-free-bit ´mem-pool-info (pool b) (´lvl t) (´bb t));;*)~~
´mem-pool-info := set-bit-noexist ´mem-pool-info (pool b) (´lvl t) (´bb t);;
´block-pt := ´block-pt (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bb t));;
IF ´bn t ≠ ´bb t ∧ block-fits (´mem-pool-info (pool b))
(´block-pt t)
(´lsz t) THEN

~~(* sys-dlist-remove(block-ptr(p, lsz, b)); *)~~
´mem-pool-info := ´mem-pool-info ((pool b) :=
remove-free-list (´mem-pool-info (pool b)) (´lvl t) (´block-pt t))
FI
ROF;;

(
~~(* ´j := ´j(t := ´lvl t);; (* use lbn and j to store the previous lvl and bn, or can not give the post condition *) ´lbn := ´lbn (t := ´bn t);; (* since the lbn and j are not used in M-pool-free *) ´lvl := ´lvl (t := ´j t − 1);; ´bn := ´bn (t := ´lbn t div 4);;*)~~
´lvl := ´lvl (t := ´lvl t − 1);;
´bn := ´bn (t := ´bn t div 4);;
~~(* we add this statement. set the parent node from divided to freeing *)~~
´mem-pool-info := set-bit-freeing ´mem-pool-info (pool b) (´lvl t) (´bn t);;
~~(* ´freeing-node := ´freeing-node (t := Some (|pool = (pool b), level = (´lvl t), block = (´bn t), data = block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bn t) |))*)~~
´freeing-node := ´freeing-node (t := Some (|pool = (pool b), level = (´lvl t),
block = (´bn t),
data = block-ptr (´mem-pool-info (pool b))
(((ALIGN4 (max-sz (´mem-pool-info (pool b)))) div (4 ^ (´lvl t))))
(´bn t) |))
)

ELSE
IF block-fits (´mem-pool-info (pool b)) (´blk t) (´lsz t) THEN

```
          (* sys-dlist-append(&p->levels[level].free-list, block); *)
          ´mem-pool-info := ´mem-pool-info ((pool b) :=
                  append-free-list (´mem-pool-info (pool b)) (´lvl t) (´blk t) )
        FI;;

        ´free-block-r := ´free-block-r (t := False)
      FI

    END)

  OD;;
 (* #### end: free-block(get-pool(block->id.pool), block->id.level, sizes, block->id.block);
*)

  (t ► ATOMIC

    WHILE wait-q (´mem-pool-info (pool b)) ≠ [] DO
      ´th := ´th (t := hd (wait-q (´mem-pool-info (pool b))));;
      (* unpend-thread(th); *)
      ´mem-pool-info := ´mem-pool-info (pool b := ´mem-pool-info (pool b)
              (|wait-q := tl (wait-q (´mem-pool-info (pool b)))|));;
      (* ready-thread(th); *)
      ´thd-state := ´thd-state (´th t := READY);;
      ´need-resched := ´need-resched(t := True)
    OD;;

    IF ´need-resched t THEN
      reschedule
    FI
  END)
END

definition Schedule :: Thread ⇒ (EventLabel, 'a, State, State com option) esys
where Schedule t ≡
  EVENT ScheduleE [Thread t] ⇛ 𝒮
  THEN
    AWAIT ´thd-state t = READY THEN (* only schedule the READY threads
*)
      IF (´cur ≠ None) THEN
        ´thd-state := ´thd-state(the (´cur) := READY);;
        ´cur := None
      FI;;
      ´cur := Some t;;
      ´thd-state := ´thd-state(t := RUNNING)
    END
  END

definition Tick :: (EventLabel, 'a, State, State com option) esys
where Tick ≡
```

EVENT TickE [] $\Rrightarrow$ *Timer*
THEN
  ´tick := *´tick* + *1*
END

term *Evt-sat-RG*
end

theory *invariant*
imports *mem-spec HOL−Eisbach.Eisbach-Tools*
begin

this theory defines the invariant and its lemmas.

13 invariants

13.1 defs of invariants

we consider multi-threaded execution on mono-core. A thread is the currently executing thread iff it is in RUNNING state.

definition *inv-cur* :: *State* $\Rightarrow$ *bool*
where $\mathit{inv\text{-}cur}\ s \equiv \forall t.\ \mathit{cur}\ s = \mathit{Some}\ t \longleftrightarrow \mathit{thd\text{-}state}\ s\ t = \mathit{RUNNING}$

abbreviation *dist-list* :: *'a list* $\Rightarrow$ *bool*
where $\mathit{dist\text{-}list}\ l \equiv \forall i\ j.\ i < \mathit{length}\ l \wedge j < \mathit{length}\ l \wedge i \neq j \longrightarrow l!i \neq l!j$

the relation of thread state and wait queue. here we dont consider other modules of zephyr, so blocked thread is in wait que of mem pools.

definition *inv-thd-waitq* :: *State* $\Rightarrow$ *bool*
where $\mathit{inv\text{-}thd\text{-}waitq}\ s \equiv$
$(\forall p{\in}\mathit{mem\text{-}pools}\ s.\ \forall t{\in}\ \mathit{set}\ (\mathit{wait\text{-}q}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p)).\ \mathit{thd\text{-}state}\ s\ t = \mathit{BLOCKED})$
~~(* thread in waitq is BLOCKED *)~~
$\wedge\ (\forall t.\ \mathit{thd\text{-}state}\ s\ t = \mathit{BLOCKED} \longrightarrow (\exists p{\in}\mathit{mem\text{-}pools}\ s.\ t \in \mathit{set}\ (\mathit{wait\text{-}q}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p))))$
~~(* BLOCKED thread is in a waitq *)~~
$\wedge\ (\forall p{\in}\mathit{mem\text{-}pools}\ s.\ \mathit{dist\text{-}list}\ (\mathit{wait\text{-}q}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p)))$
~~(* threads in a waitq are different with each other. which means a thread could not waiting for the same pool two times *)~~
$\wedge\ (\forall p\ q.\ p{\in}\mathit{mem\text{-}pools}\ s \wedge q{\in}\mathit{mem\text{-}pools}\ s \wedge p \neq q \longrightarrow (\nexists t.\ t \in \mathit{set}\ (\mathit{wait\text{-}q}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p))$
$\wedge\ t{\in}\ \mathit{set}\ (\mathit{wait\text{-}q}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ q))))$

invariant of configuration of memory pools. its actually a well-formed property for memory configuration. (1) the max size (the size of top-level (level 0) block) is 4^{n_levels} times of block size of the lowest level. 4 * n means that the block size of the lowest level is alignd with 4. (2) the block number at level 0 (n_max) ¿ 0, and the max number of levels is n_levels ¿ 0 (3) n_level is equal to the length of levels list. (4) the length of bitmap list at each level is equal to the block number at the same level. Thus, bitmap saves a complete quad-tree with height of n_levels. A real memory pool is a top subtree of the complete tree. bits of subnodes of a leaf node (ALLOCATED, FREE, ALLOCATING, FREEING) is NOEXIST.

abbreviation *inv-mempool-info-mp* :: *State* ⇒ *mempool-ref* ⇒ *bool*
where *inv-mempool-info-mp s p* ≡
  let mp = mem-pool-info s p in
    buf mp ≠ *NULL* ∧ (∃ *n*>*0*. *max-sz mp* = (*4* ∗ *n*) ∗ (*4* ^ *n-levels mp*))
   ∧ *n-max mp* > *0* ∧ *n-levels mp* > *0*
   ∧ *n-levels mp* = *length* (*levels mp*)
   ∧ (∀ *i*<*length* (*levels mp*). *length* (*bits* (*levels mp* ! *i*)) = (*n-max mp*) ∗ *4* ^ *i*)

definition *inv-mempool-info* :: *State* ⇒ *bool*
where *inv-mempool-info s* ≡ ∀ *p*∈*mem-pools s*. *inv-mempool-info-mp s p*

lemma *inv-max-sz-gt0*: *inv-mempool-info s* ⟹ ∀ *p*∈*mem-pools s*. *let mp* = *mem-pool-info s p in max-sz mp* > *0*
  unfolding *inv-mempool-info-def* **using** *neq0-conv* **by** *fastforce*

invariant between bitmap and free block list at each level. (1) bit of a block is FREE, iff its start address is in free list. the start address is buf mp + j * (max_sz mp div (4^i)), the start address of the mempool + block size at this level * block index (2) start address of blocks in free list is valid, i.e. it is the start address of some block (index n), where n is in the range of block index at the level (3) start address of blocks in free list are different with each other.

abbreviation *inv-bitmap-freelist-mp* :: *State* ⇒ *mempool-ref* ⇒ *bool*
where *inv-bitmap-freelist-mp s p* ≡
    let mp = mem-pool-info s p in
     ∀ *i* < *length* (*levels mp*).
      let bts = bits (*levels mp* ! *i*);
        fl = free-list (*levels mp* ! *i*) *in*
      (∀ *j* < *length bts*. *bts* ! *j* = *FREE* ⟷ *buf mp* + *j* ∗ (*max-sz mp div* (*4* ^ *i*)) ∈ *set fl*)
      ~~(∗ the block corresponding to a free bit iff it is in freelist ∗)~~
      ∧ (∀ *j* < *length fl*. (∃ *n*. *n* < *n-max mp* ∗ (*4* ^ *i*) ∧ *fl* ! *j* = *buf mp* + *n* ∗ (*max-sz mp div* (*4* ^ *i*))))
      ~~(∗ pointers in freelist are head address of blocks ∗)~~
      ∧ *distinct fl* ~~(∗(∀ k j. k < length fl ∧ j < length fl ⟶ fl!k ≠ fl!j ⟶ k ≠ j) ∗)~~
      ~~(∗ pointers in freelist are different with each other ∗)~~

definition *inv-bitmap-freelist* :: *State* ⇒ *bool*

where *inv-bitmap-freelist s* $\equiv$
$\forall p \in mem\text{-}pools\ s.\ inv\text{-}bitmap\text{-}freelist\text{-}mp\ s\ p$

this invariant represents that a memory pools is forest of valid quad-trees of blocks. parent node of a leaf node (ALLOCATED, FREE, ALLOCATING, FREEING) is an inner node (DIVIDED). parent node of an inner node (DIVIDED) is also a DIVIDED node. child nodes of a NOEXIST node is also NOEXIST nodes. parent node of a NOEXIST node should not be DIVIDE nodes (may be NOEXIST, ALLOCATED, FREE, ALLOCATING, FREEING)

abbreviation *inv-bitmap-mp* :: $State \Rightarrow mempool\text{-}ref \Rightarrow bool$
where *inv-bitmap-mp s p* $\equiv$
$let\ mp = mem\text{-}pool\text{-}info\ s\ p\ in$
$\forall i < length\ (levels\ mp).$
$let\ bts = bits\ (levels\ mp\ !\ i)\ in$
$(\forall j < length\ bts.$
$(bts\ !\ j = FREE \lor bts\ !\ j = FREEING \lor bts\ !\ j = ALLOCATED$
$\lor bts\ !\ j = ALLOCATING \longrightarrow$
$(i > 0 \longrightarrow (bits\ (levels\ mp\ !\ (i - 1)))\ !\ (j\ div\ 4) = DIVIDED)$
$\land (i < length\ (levels\ mp) - 1 \longrightarrow noexist\text{-}bits\ mp\ (i+1)\ (j*4)$
$))$
$\land (bts\ !\ j = DIVIDED \longrightarrow i > 0 \longrightarrow (bits\ (levels\ mp\ !\ (i - 1)))\ !$
$(j\ div\ 4) = DIVIDED)$
$\land (bts\ !\ j = NOEXIST \longrightarrow i < length\ (levels\ mp) - 1$
$\longrightarrow noexist\text{-}bits\ mp\ (i+1)\ (j*4))$
$\land (bts\ !\ j = NOEXIST \land i > 0 \longrightarrow (bits\ (levels\ mp\ !\ (i - 1)))\ !\ (j$
$div\ 4) \neq DIVIDED)\ )$

definition *inv-bitmap* :: $State \Rightarrow bool$
where *inv-bitmap s* $\equiv$
$\forall p \in mem\text{-}pools\ s.\ inv\text{-}bitmap\text{-}mp\ s\ p$

due to the rule of merge as possible, there should not exist a node with 4 FREE child blocks. In free syscall, 4 free child blocks should be merged to a bigger block.

abbreviation *inv-bitmap-not4free-mp* :: $State \Rightarrow mempool\text{-}ref \Rightarrow bool$
where *inv-bitmap-not4free-mp s p* $\equiv$
$let\ mp = mem\text{-}pool\text{-}info\ s\ p\ in$
$\forall i < length\ (levels\ mp).$
$let\ bts = bits\ (levels\ mp\ !\ i)\ in$
$(\forall j < length\ bts.\ i > 0 \longrightarrow \neg\ partner\text{-}bits\ mp\ i\ j)$

definition *inv-bitmap-not4free* :: $State \Rightarrow bool$
where *inv-bitmap-not4free s* $\equiv$
$\forall p \in mem\text{-}pools\ s.\ inv\text{-}bitmap\text{-}not4free\text{-}mp\ s\ p$

blocks at level 0 should not be NOEXIST. If so, the memory pool does not exist. We only allow real memory pools.

definition *inv-bitmap0* :: *State* ⇒ *bool*
where *inv-bitmap0 s* ≡
∀ p∈mem-pools s. let bits0 = bits (levels (mem-pool-info s p) ! 0) in ∀ i<length bits0. bits0 ! i ≠ NOEXIST

blocks at last level (n_level - 1) should not be split again, thus should not be DIVIDED

definition *inv-bitmapn* :: *State* ⇒ *bool*
where *inv-bitmapn s* ≡
∀ p∈mem-pools s. let bitsn = bits ((levels (mem-pool-info s p) ! (length (levels (mem-pool-info s p)) − 1)))
in ∀ i<length bitsn. bitsn ! i ≠ DIVIDED

definition *mem-block-addr-valid* :: *State* ⇒ *Mem-block* ⇒ *bool*
where *mem-block-addr-valid s b* ≡
data b = buf (mem-pool-info s (pool b)) + (block b) ∗ ((max-sz (mem-pool-info s (pool b))) div (4 ^ (level b)))

invariants between FREEING/ALLOCATING blocks and freeing/allocating_node variables.

definition *inv-aux-vars* :: *State* ⇒ *bool*
where *inv-aux-vars s* ≡
(∀ t n. freeing-node s t = Some n ⟶ get-bit (mem-pool-info s) (pool n) (level n) (block n) = FREEING)
~~(* freeing node is state of FREEING *)~~
∧ (∀ n. get-bit (mem-pool-info s) (pool n) (level n) (block n) = FREEING ∧ mem-block-addr-valid s n
⟶ (∃ t. freeing-node s t = Some n))
~~(* node of state of FREEING is freeing *)~~
∧ (∀ t n. allocating-node s t = Some n ⟶ get-bit (mem-pool-info s) (pool n) (level n) (block n) = ALLOCATING)
~~(* freeing node is state of FREEING *)~~
∧ (∀ n. get-bit (mem-pool-info s) (pool n) (level n) (block n) = ALLOCATING ∧ mem-block-addr-valid s n
⟶ (∃ t. allocating-node s t = Some n))
~~(* node of state of FREEING is freeing *)~~
∧ (∀ t1 t2 n1 n2. t1 ≠ t2 ∧ freeing-node s t1 = Some n1 ∧ freeing-node s t2 = Some n2
⟶ ¬(pool n1 = pool n2 ∧ level n1 = level n2 ∧ block n1 = block n2))
~~(*here we only consider the pool, level, and block, not the first addr of the block *)~~
~~(* freeing nodes are different each other *)~~
∧ (∀ t1 t2 n1 n2. t1 ≠ t2 ∧ allocating-node s t1 = Some n1 ∧ allocating-node s t2 = Some n2
⟶ ¬(pool n1 = pool n2 ∧ level n1 = level n2 ∧ block n1 = block n2))
~~(* allocating node are different each other *)~~

$\wedge$ ($\forall$ *t1 t2 n1 n2. allocating-node s t1 = Some n1* $\wedge$ *freeing-node s t2 = Some n2*
$\longrightarrow \neg$(*pool n1 = pool n2* $\wedge$ *level n1 = level n2* $\wedge$ *block n1 = block n2*))

definition *inv* :: *State* $\Rightarrow$ *bool*
where *inv s* $\equiv$ *inv-cur s* $\wedge$ *inv-thd-waitq s* $\wedge$ *inv-mempool-info s*
$\wedge$ *inv-bitmap-freelist s* $\wedge$ *inv-bitmap s* $\wedge$ *inv-aux-vars s*
$\wedge$ *inv-bitmap0 s* $\wedge$ *inv-bitmapn s* $\wedge$ *inv-bitmap-not4free s*

method *simp-inv* = (*simp add:inv-def inv-bitmap-def inv-bitmap-freelist-def*
inv-mempool-info-def inv-thd-waitq-def inv-cur-def inv-aux-vars-def
inv-bitmap0-def inv-bitmapn-def
inv-bitmap-not4free-def mem-block-addr-valid-def)

method *unfold-inv* = (*unfold inv-def inv-bitmap-def inv-bitmap-freelist-def*
inv-mempool-info-def inv-thd-waitq-def inv-cur-def inv-aux-vars-def
inv-bitmap0-def inv-bitmapn-def
inv-bitmap-not4free-def mem-block-addr-valid-def)[*1*]

lemma *inv-imp-fl-lt0*:
inv Va $\Longrightarrow$
$\forall$ *p*$\in$*mem-pools Va.*
let mp = mem-pool-info Va p in
$\forall$ *i < length (levels mp).*
$\forall$ *j < length (free-list (levels mp ! i)). free-list (levels mp ! i) ! j > 0*
apply(*simp add:inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply(*simp add:Let-def*) **apply** *clarsimp*
by *fastforce*

13.2 initial state s_0

we dont consider mem_pool_init, only define $s0$ to show the state after memory pool initialization.

axiomatization *s0*::*State* **where**
s0a1: *cur s0 = None* **and**
s0a2: *tick s0 = 0* **and**
s0a3: *thd-state s0* = (λ*t. READY*) **and**
s0a5: *mem-pools s0* $\neq$ {} **and**
s0a7: $\forall$ *p*$\in$*mem-pools s0. wait-q (mem-pool-info s0 p)* = [] **and**
s0a6: $\forall$ *p*$\in$*mem-pools s0. let mp = mem-pool-info s0 p in*
buf mp > 0 $\wedge$ ($\exists$ *n>0. max-sz mp = (4 * n) * (4 ^ n-levels mp)*)
$\wedge$ *n-max mp > 0* $\wedge$ *n-levels mp > 1*
$\wedge$ *n-levels mp = length (levels mp)* **and**
s0a8: $\forall$ *p*$\in$*mem-pools s0.* ~~(* defines level 1 to n *)~~
let mp = mem-pool-info s0 p in

∀ i. i > 0 ∧ i < length (levels mp) ⟶
length (bits (levels mp ! i)) = n-max mp * 4 ˆ i
∧ (∀ j<length (bits (levels mp ! i)). bits (levels mp ! i) ! j = NOEXIST)
∧ free-list (levels mp ! i) = [] **and**
s0a9: ∀ p∈mem-pools s0. ~~(* defines the level0 *)~~
let mp = mem-pool-info s0 p;
lv0 = (levels mp)!0 in
length (bits lv0) = n-max mp
∧ length (free-list lv0) = n-max mp
∧ (∀ i<length (bits lv0). (bits lv0)!i = FREE)
∧ (∀ i<length (free-list lv0). (free-list lv0) ! i = (buf mp) + i * max-sz mp)
∧ distinct (free-list lv0) **and**
s0a4: freeing-node s0 = Map.empty **and**
s0a10: allocating-node s0 = Map.empty **and**
s0a11: ∄ n. get-bit-s s0 (pool n) (level n) (block n) = FREEING **and**
s0a12: ∄ n. get-bit-s s0 (pool n) (level n) (block n) = ALLOCATING

lemma s0-max-sz-gt0: ∀ p∈mem-pools s0. let mp = mem-pool-info s0 p in max-sz mp > 0
using s0a6 zero-less-power **by** fastforce

lemma s0-inv-cur: inv-cur s0
by (simp add:inv-cur-def s0a1 s0a3)

lemma s0-inv-thdwaitq: inv-thd-waitq s0
by (simp add: inv-thd-waitq-def s0a7 s0a3)

lemma s0-inv-mempool-info: inv-mempool-info s0
apply (simp add: inv-mempool-info-def Let-def) **apply** clarsimp
apply(rule conjI) **apply** (metis neq0-conv s0a6)
apply(rule conjI) **apply** (meson s0a6)
apply(rule conjI) **apply** (meson s0a6)
apply(rule conjI) **using** neq0-conv s0a6 **apply** fastforce
apply(rule conjI) **apply** (meson s0a6)
by (metis One-nat-def mult-numeral-1-right neq0-conv numeral-1-eq-Suc-0 power.simps(1) s0a8 s0a9)

lemma s0-inv-bitmap-freelist: inv-bitmap-freelist s0
apply(simp add:inv-bitmap-freelist-def)
apply(simp add: Let-def) **apply** clarsimp
apply(case-tac i = 0)
apply(rule conjI) **apply** clarsimp **apply** (metis nth-mem s0a9)
apply(rule conjI) **apply** clarsimp **apply** (metis s0a9)
apply (meson s0a9)

apply(rule conjI) **apply** clarsimp

apply(*subgoal-tac n-levels* (*mem-pool-info s0 p*) = *length* (*levels* (*mem-pool-info s0 p*)))
prefer *2* **apply** (*meson s0a6*)
apply(*subgoal-tac get-bit-s s0 p i j* ≠ *FREE*)
prefer *2* **apply** (*metis BlockState.distinct*(*13*) *s0a8*)
apply(*subgoal-tac set* (*free-list* (*levels* (*mem-pool-info s0 p*) ! *i*)) = {})
prefer *2* **apply** (*metis all-not-in-conv in-set-conv-nth length-greater-0-conv neq0-conv not-less-zero s0a8*)
apply *simp*

apply(*rule conjI*) **apply** *clarsimp*
apply (*metis length-greater-0-conv neq0-conv not-less-zero s0a8*)
apply (*metis distinct-conv-nth length-0-conv neq0-conv not-less-zero s0a8*)
done

lemma *s0-inv-bitmap*: *inv-bitmap s0*
apply(*simp add*: *inv-bitmap-def*)
apply(*simp add*: *Let-def*) **apply** *clarsimp*
apply(*case-tac i* = *0*)
apply *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def partner-bits-def*)

apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)

apply(*case-tac i* = *1*) **apply** *clarsimp* **using** *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def*)
apply(*subgoal-tac i* > *1*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac get-bit-s s0 p* (*i* − *Suc NULL*) (*j div 4*) = *NOEXIST*)
prefer *2* **using** *s0a8* **apply**(*simp add*: *Let-def*)
apply(*subgoal-tac j div 4* < *length* (*bits* (*levels* (*mem-pool-info s0 p*) ! (*i* − *1*))))
prefer *2* **using** *s0a6* **apply**(*simp add*:*Let-def*)
apply(*subgoal-tac n-max* (*mem-pool-info s0 p*) > *0*)
prefer *2* **using** *s0a6* **apply**(*simp add*:*Let-def*)
apply(*simp add*: *power-eq-if*)
apply *auto*[*1*]
apply *simp*

done

lemma *s0-inv-bitmap-not4free*: *inv-bitmap-not4free s0*
apply(*simp add*: *inv-bitmap-not4free-def*)
apply(*simp add*: *Let-def*) **apply** *clarsimp*
using *s0a6 s0a8 s0a9* **apply**(*simp add*:*Let-def partner-bits-def*)
done

lemma *s0-inv-aux-vars*: *inv-aux-vars s0*
  apply(*simp add*: *inv-aux-vars-def Let-def*)
  apply(*rule conjI*) **apply** (*simp add*: *s0a4*)
  apply(*rule conjI*) **apply** *clarify* **using** *s0a11* **apply** *auto*[*1*]
  apply(*rule conjI*) **apply** (*simp add*: *s0a10*)
  apply(*rule conjI*) **apply** *clarify* **using** *s0a12* **apply** *auto*[*1*]
  apply(*rule conjI*) **apply** (*simp add*: *s0a4*)
  apply(*rule conjI*) **apply** (*simp add*: *s0a10*)
  apply (*simp add*: *s0a4 s0a10*)
done

lemma *s0-inv-bitmap-freelist0*: *inv-bitmap0 s0*
  apply(*simp add*: *inv-bitmap0-def Let-def*)
  using *s0a9* **apply**(*simp add*:*Let-def*)
done

lemma *s0-inv-bitmap-freelistn*: *inv-bitmapn s0*
  apply(*simp add*: *inv-bitmapn-def Let-def*)
  using *s0a8* **apply**(*simp add*:*Let-def*) **apply** *clarify*
  apply(*subgoal-tac get-bit-s s0 p* (*length* (*levels* (*mem-pool-info s0 p*)) − *Suc 0*) *i* = *NOEXIST*)
    prefer *2* **apply**(*subgoal-tac length* (*levels* (*mem-pool-info s0 p*)) > *0*)
      prefer *2* **using** *s0a6* **apply**(*simp add*:*Let-def*) **apply** *auto*[*1*]
    using *s0a6* **apply**(*simp add*:*Let-def*) **apply** *auto*[*1*]
 apply *simp*
done

lemma *s0-inv*: *inv s0*
  apply(*unfold inv-def*)
  apply(*rule conjI*) **using** *s0-inv-cur* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-thdwaitq* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-mempool-info* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-bitmap-freelist* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-bitmap* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-aux-vars* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-bitmap-freelist0* **apply** *fast*
  apply(*rule conjI*) **using** *s0-inv-bitmap-freelistn* **apply** *fast*
  using *s0-inv-bitmap-not4free* **apply** *fast*
done

13.3 lemmas of invariants

lemma *inv-bitmap-presv-setbit-0*:
¬ (*x* = *l* ∧ *y* = *b*) ⟹
    Vb = *Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
        (*p* := *mem-pool-info Va p*
          ⦇*levels* := (*levels* (*mem-pool-info Va p*))
              [*l* := (*levels* (*mem-pool-info Va p*) ! *l*)⦇*bits* := (*bits* (*levels*

(*mem-pool-info Va p*) ! *l*))[*b* := *st*]⦈]⦈))⦈ ⟹
get-bit-s Va p x y= *get-bit-s Vb p x y*
apply *simp* **by** (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
Mem-pool-lvl.surjective list-update-beyond not-less nth-list-update-eq
nth-list-update-neq)

lemma *inv-bitmap-presv-setbit*:
inv-bitmap Va ⟹
get-bit-s Va p l b = *FREE* ∨ *get-bit-s Va p l b* = *FREEING* ∨ *get-bit-s Va p l b*
= *ALLOCATED*
∨ *get-bit-s Va p l b* = *ALLOCATING* ⟹
st = *FREE* ∨ *st* = *FREEING* ∨ *st* = *ALLOCATED* ∨ *st* = *ALLOCATING*
⟹
Vb = *set-bit-s Va p l b st* ⟹
inv-bitmap Vb
apply(*simp add*:*inv-bitmap-def*) **apply**(*simp add*:*set-bit-s-def set-bit-def*)

apply(*simp add*:*Let-def*) **apply** *clarify* **apply**(*rename-tac ii jj*)
apply(*subgoal-tac p*∈*mem-pools Va*) **prefer** *2* **apply**(*simp add*:*set-bit-s-def set-bit-def*)
apply(*subgoal-tac jj* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply**(*simp add*:*set-bit-s-def set-bit-def*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
Mem-pool-lvl.surjective list-updt-samelen nth-list-update-eq nth-list-update-neq)

apply(*rule conjI*) **apply** *clarify* **apply**(*rule conjI*) **apply** *clarify*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) =
DIVIDED)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective*
One-nat-def nth-list-update-eq nth-list-update-neq)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* − *1*))) ! (*jj div 4*))
prefer *2* **apply**(*case-tac ii* − *1* = *l* ∧ *jj div 4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply *clarify* **apply**(*rule conjI*)

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*) = *NOEX-*
IST)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective*
nth-list-update-eq nth-list-update-neq)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

```
apply(rule conjI)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 1) =
NOEXIST)
 prefer 2 apply (smt Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4) Mem-pool-lvl.surjective
   Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 1)
                = (bits (levels (mem-pool-info Vb p) ! Suc ii)) ! (jj * 4 + 1))
  prefer 2 apply(case-tac Suc ii = l ∧ jj * 4 + 1 = b) apply simp using
inv-bitmap-presv-setbit-0 apply metis
apply simp

apply(rule conjI)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 2) =
NOEXIST)
  prefer 2
 apply (smt Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4) Mem-pool-lvl.surjective
       add-2-eq-Suc' nth-list-update-eq nth-list-update-neq)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 2)
                = (bits (levels (mem-pool-info Vb p) ! Suc ii)) ! (jj * 4 + 2))
  prefer 2 apply(case-tac Suc ii = l ∧ jj * 4 + 2 = b) apply simp using
inv-bitmap-presv-setbit-0 apply metis
apply simp

apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 3) =
NOEXIST)
  prefer 2
 apply (smt Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4) Mem-pool-lvl.surjective
       add-2-eq-Suc' nth-list-update-eq nth-list-update-neq)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! Suc ii)) ! (jj * 4 + 3)
                = (bits (levels (mem-pool-info Vb p) ! Suc ii)) ! (jj * 4 + 3))
  prefer 2 apply(case-tac Suc ii = l ∧ jj * 4 + 3 = b) apply simp using
inv-bitmap-presv-setbit-0 apply metis
apply simp


apply(rule conjI) apply clarify apply(rule conjI) apply clarify

apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! (ii − 1))) ! (jj div 4) =
DIVIDED)
 prefer 2 apply (smt Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4) Mem-pool-lvl.surjective
              One-nat-def nth-list-update-eq nth-list-update-neq)
apply(subgoal-tac (bits (levels (mem-pool-info Va p) ! (ii − 1))) ! (jj div 4)
                = (bits (levels (mem-pool-info Vb p) ! (ii − 1))) ! (jj div 4))
 prefer 2 apply(case-tac ii − 1 = l ∧ jj div 4 = b) apply simp using inv-bitmap-presv-setbit-0
apply simp
apply simp

apply clarify apply(rule conjI)
```

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*) = *NOEX-IST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *1* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *2* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *3* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*) **apply** *clarify* **apply**(*rule conjI*) **apply** *clarify*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) =

DIVIDED)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective One-nat-def nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* − *1*))) ! (*jj div 4*))
prefer *2* **apply**(*case-tac ii* − *1* = *l* ∧ *jj div 4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply *clarify* **apply**(*rule conjI*)

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*) = (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*) = (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *1* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*) = (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *2* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *3* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*) **apply** *clarify* **apply**(*rule conjI*) **apply** *clarify*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective One-nat-def nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* − *1*))) ! (*jj div 4*))
prefer *2* **apply**(*case-tac ii* − *1* = *l* ∧ *jj div 4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply *clarify* **apply**(*rule conjI*)

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *1*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *1* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*)

= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *2*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *2* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*) = *NOEXIST*)
prefer *2*
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective add-2-eq-Suc′ nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! *Suc ii*)) ! (*jj* ∗ *4* + *3*))
prefer *2* **apply**(*case-tac Suc ii* = *l* ∧ *jj* ∗ *4* + *3* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0* **apply** *metis*
apply *simp*

apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective One-nat-def nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* − *1*))) ! (*jj div 4*))
prefer *2* **apply**(*case-tac ii* − *1* = *l* ∧ *jj div 4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply(*rule conjI*)

apply *clarify*
apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* + *1*))) ! (*jj* ∗ *4*))
prefer *2* **apply**(*case-tac ii* + *1* = *l* ∧ *jj* ∗ *4* = *b*) **apply** *simp* **using** *inv-bitmap-presv-setbit-0*
apply *simp*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *1*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective*

Nat.add-0-right One-nat-def add-Suc-right nth-list-update-eq nth-list-update-neq)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *1*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *1*))
prefer *2* **apply**(*case-tac ii* + *1* = *l* ∧ *jj* ∗ *4* + *1* = *b*) **apply** *auto*[*1*] **using** *inv-bitmap-presv-setbit-0* **apply** *simp*
apply *simp*

apply(*rule conjI*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *2*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-2-eq-Suc′ add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *2*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *2*))
prefer *2* **apply**(*case-tac ii* + *1* = *l* ∧ *jj* ∗ *4* + *2* = *b*) **apply** *auto*[*1*] **using** *inv-bitmap-presv-setbit-0* **apply** *simp*
apply *simp*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *3*) = *NOEXIST*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Nat.add-0-right One-nat-def add-2-eq-Suc′ add-Suc-right nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *3*)
= (*bits* (*levels* (*mem-pool-info Vb p*) ! (*ii* + *1*))) ! (*jj* ∗ *4* + *3*))
prefer *2* **apply**(*case-tac ii* + *1* = *l* ∧ *jj* ∗ *4* + *3* = *b*) **apply** *auto*[*1*] **using** *inv-bitmap-presv-setbit-0* **apply** *simp*
apply *simp*

apply *clarify*

apply(*subgoal-tac bits* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj* = *NOEXIST*)
prefer *2* **apply**(*case-tac ii* = *l* ∧ *jj* = *b*) **apply** *auto*[*1*] **using** *inv-bitmap-presv-setbit-0* **apply** *simp*
apply(*subgoal-tac bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*)) ! (*jj div 4*) ≠ *DIVIDED*)
prefer *2* **apply** (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective One-nat-def nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*)
= (*bits* ((*levels* (*mem-pool-info Va p*))
[*l* := (*levels* (*mem-pool-info Va p*) ! *l*)(|*bits* := (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))[*b* := *st*]|)] !
(*ii* − *1*))) ! (*jj div 4*)) **prefer** *2*
apply(*case-tac ii* − *1* = *l* ∧ *jj div 4* = *b*) **apply** *clarsimp*
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *Suc NULL*)))[*jj div 4* := *st*] ! (*jj div 4*) = *st*)

prefer *2* **apply** (*metis list-update-beyond not-less nth-list-update-eq*)
apply *simp*
using *inv-bitmap-presv-setbit-0* **apply** *simp*
apply *clarsimp*

done

lemma *inv-bitmap-freelist-fl-bnum-in*:
inv-bitmap-freelist Va $\Longrightarrow$
inv-mempool-info Va $\Longrightarrow$
p∈*mem-pools Va* $\Longrightarrow$
ii < *length* (*levels* (*mem-pool-info Va p*)) $\Longrightarrow$
jj < *length* (*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) $\Longrightarrow$
block-num (*mem-pool-info Va p*)
((*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) ! *jj*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
apply(*simp add:inv-bitmap-freelist-def inv-mempool-info-def block-num-def Let-def*)
apply(*subgoal-tac* ∃ *n*. *n* < *n-max* (*mem-pool-info Va p*) ∗ (*4* ^ *ii*) ∧ *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj*
= *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*))
prefer *2* **apply** *blast*
apply(*subgoal-tac free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj* ≥ *buf* (*mem-pool-info Va p*))
prefer *2* **apply** *linarith*
using *nonzero-mult-div-cancel-right* **by** *force*

lemma *inv-bitmap-freelist-fl-FREE*:
inv-bitmap-freelist Va $\Longrightarrow$
inv-mempool-info Va $\Longrightarrow$
p∈*mem-pools Va* $\Longrightarrow$
ii < *length* (*levels* (*mem-pool-info Va p*)) $\Longrightarrow$
jj < *length* (*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) $\Longrightarrow$
get-bit-s Va p ii (*block-num* (*mem-pool-info Va p*)
((*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) ! *jj*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*)) = *FREE*
apply(*simp add:inv-bitmap-freelist-def inv-mempool-info-def block-num-def Let-def*)
apply(*subgoal-tac* ∃ *n*. *n* < *n-max* (*mem-pool-info Va p*) ∗ (*4* ^ *ii*) ∧ *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj*
= *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*))
prefer *2* **apply** *blast*
by (*metis add-diff-cancel-left' div-0 mult-0-right neq0-conv nonzero-mult-div-cancel-right not-less-zero nth-mem*)

lemma *inv-buf-le-fl*:

inv-bitmap-freelist Va $\Longrightarrow$
  inv-mempool-info Va $\Longrightarrow$
  p∈*mem-pools Va* $\Longrightarrow$
  ii < *length* (*levels* (*mem-pool-info Va p*)) $\Longrightarrow$
  jj < *length* (*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) $\Longrightarrow$
  buf (*mem-pool-info Va p*) ≤ (*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) ! *jj*
apply(*simp add*:*inv-bitmap-freelist-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac* ∃ *n*. *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj*
        = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ^
ii))
  prefer *2* **apply** *blast*
by *linarith*

lemma *inv-fl-mod-sz0*:
inv-bitmap-freelist Va $\Longrightarrow$
  inv-mempool-info Va $\Longrightarrow$
  p∈*mem-pools Va* $\Longrightarrow$
  ii < *length* (*levels* (*mem-pool-info Va p*)) $\Longrightarrow$
  jj < *length* (*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) $\Longrightarrow$
  ((*free-list* ((*levels* (*mem-pool-info Va p*)) ! *ii*)) ! *jj* − *buf* (*mem-pool-info Va p*))
mod
        (*max-sz* (*mem-pool-info Va p*) *div* *4* ^ *ii*) = *0*
apply(*simp add*:*inv-bitmap-freelist-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac* ∃ *n*. *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *jj*
        = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ^
ii))
  prefer *2* **apply** *blast*
by *force*

lemma *sameinfo-inv-bitmap-mp*:
mem-pool-info Va p = *mem-pool-info Vb p* $\Longrightarrow$ *inv-bitmap-mp Va p* = *inv-bitmap-mp*
Vb p
apply(*simp add*: *Let-def*)
done

lemma *sameinfo-inv-bitmap-freelist-mp*:
mem-pool-info Va p = *mem-pool-info Vb p* $\Longrightarrow$ *inv-bitmap-freelist-mp Va p* =
inv-bitmap-freelist-mp Vb p
apply(*simp add*: *Let-def*)
done

lemma *inv-bitmap-presv-mpls-mpi*:
  mem-pools Va = *mem-pools Vb* $\Longrightarrow$
   mem-pool-info Va = *mem-pool-info Vb* $\Longrightarrow$
   inv-bitmap Va $\Longrightarrow$
   inv-bitmap Vb
by(*simp add*:*inv-bitmap-def Let-def*)

lemma *inv-bitmap-presv-mpls-mpi2*:
  mem-pools Va = *mem-pools Vb* $\Longrightarrow$
   ($\forall$ *p*. *length* (*levels* (*mem-pool-info Va p*)) = *length* (*levels* (*mem-pool-info Vb p*))) $\Longrightarrow$
   ($\forall$ *p ii*. *ii* < *length* (*levels* (*mem-pool-info Va p*))
      $\longrightarrow$ *bits* (*levels* (*mem-pool-info Va p*) ! *ii*) = *bits* (*levels* (*mem-pool-info Vb p*) ! *ii*)) $\Longrightarrow$
   inv-bitmap Va $\Longrightarrow$
   inv-bitmap Vb
by (*simp add*: *inv-bitmap-def Let-def*)

lemma *inv-bitmap-freeing2free*:
inv-bitmap-mp V p $\Longrightarrow$
  $\exists$ *lv bl*. *bits* (*levels* (*mem-pool-info V p*) ! *lv*) ! *bl* = *FREEING*
       $\wedge$ *bits* (*levels* (*mem-pool-info V2 p*) ! *lv*) = (*bits* (*levels* (*mem-pool-info V p*) ! *lv*)) [*bl* := *FREE*]
       $\wedge$ ($\forall$ *lv'*. *lv* $\neq$ *lv'* $\longrightarrow$ *bits* (*levels* (*mem-pool-info V2 p*) ! *lv'*) = *bits* (*levels* (*mem-pool-info V p*) ! *lv'*)) $\Longrightarrow$
  length (*levels* (*mem-pool-info V p*)) = *length* (*levels* (*mem-pool-info V2 p*))
  $\Longrightarrow$ *inv-bitmap-mp V2 p*
apply(*simp add*:*Let-def*) **apply** *clarify*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V p*) ! *i*)) = *length* (*bits* (*levels* (*mem-pool-info V2 p*) ! *i*)))
  prefer *2* **apply**(*case-tac i* = *lv*) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *clarify*
  apply(*rule conjI*) **apply** *clarify*
    apply(*case-tac i* = *lv* $\wedge$ *j* = *bl*) **apply** *clarsimp*
      apply(*subgoal-tac get-bit-s V2 p i j* = *get-bit-s V p i j*)
        prefer *2* **apply**(*case-tac i* = *lv*) **apply** *clarsimp* **apply** *presburger*
    apply (*metis BlockState.distinct*(*21*) *nth-list-update-neq*)

  apply *clarify*
    apply(*case-tac i* = *lv* $\wedge$ *j* = *bl*) **apply** *clarsimp*
      apply(*subgoal-tac get-bit-s V2 p i j* = *get-bit-s V p i j*)
        prefer *2* **apply**(*case-tac i* = *lv*) **apply** *clarsimp* **apply** *presburger*
  apply (*smt BlockState.distinct*(*25*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
  apply(*case-tac i* = *lv* $\wedge$ *j* = *bl*) **apply** *clarsimp*
    apply(*subgoal-tac get-bit-s V2 p i j* = *get-bit-s V p i j*)
      prefer *2* **apply**(*case-tac i* = *lv*) **apply** *clarsimp* **apply** *presburger*
  apply (*smt BlockState.distinct*(*21*) *BlockState.distinct*(*25*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
  apply(*case-tac i* = *lv* $\wedge$ *j* = *bl*) **apply** *clarsimp*
    apply(*subgoal-tac get-bit-s V2 p i j* = *get-bit-s V p i j*)
      prefer *2* **apply**(*case-tac i* = *lv*) **apply** *clarsimp* **apply** *presburger*

apply (*smt BlockState.distinct*(*21*) *BlockState.distinct*(*25*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*21*) *BlockState.distinct*(*25*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*21*) *BlockState.distinct*(*25*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*21*) *BlockState.distinct*(*25*) *nth-list-update-neq*)

apply *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*11*) *list-update-beyond not-less nth-list-update-eq nth-list-update-neq*)

done

lemma *inv-bitmap-allocating2allocate*:
inv-bitmap-mp V p $\Longrightarrow$
$\exists$ *lv bl*. *bits* (*levels* (*mem-pool-info V p*) ! *lv*) ! *bl* = *ALLOCATING*
$\wedge$ *bits* (*levels* (*mem-pool-info V2 p*) ! *lv*) = (*bits* (*levels* (*mem-pool-info V p*) ! *lv*)) [*bl* := *ALLOCATED*]
$\wedge$ ($\forall$ *lv'*. *lv* $\neq$ *lv'* $\longrightarrow$ *bits* (*levels* (*mem-pool-info V2 p*) ! *lv'*) = *bits* (*levels* (*mem-pool-info V p*) ! *lv'*)) $\Longrightarrow$
length (*levels* (*mem-pool-info V p*)) = *length* (*levels* (*mem-pool-info V2 p*))
$\Longrightarrow$ *inv-bitmap-mp V2 p*
apply(*simp add*:*Let-def*) **apply** *clarify*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V p*) ! *i*)) = *length* (*bits* (*levels* (*mem-pool-info V2 p*) ! *i*)))
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*metis BlockState.distinct*(*23*) *nth-list-update-neq*)

apply *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*27*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*23*) *BlockState.distinct*(*27*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*23*) *BlockState.distinct*(*27*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*23*) *BlockState.distinct*(*27*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*23*) *BlockState.distinct*(*27*) *nth-list-update-neq*)

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*23*) *BlockState.distinct*(*27*) *nth-list-update-neq*)

apply *clarify*
apply(*case-tac* $i = lv \wedge j = bl$) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V2 p i j = get-bit-s V p i j*)
prefer *2* **apply**(*case-tac* $i = lv$) **apply** *clarsimp* **apply** *presburger*
apply (*smt BlockState.distinct*(*3*) *list-update-beyond not-less nth-list-update-eq nth-list-update-neq*)
done

lemma *inv-bitmap-freelist-presv-setbit-notfree-h*:
$\neg (x = lv \wedge y = bkn) \Longrightarrow$
Vb = set-bit-s V p lv bkn st $\Longrightarrow$

get-bit-s V p x y= get-bit-s Vb p x y
apply(*simp add*:*set-bit-s-def set-bit-def*)
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
Mem-pool-lvl.surjective list-update-beyond not-less nth-list-update-eq nth-list-update-neq)

lemma *inv-bitmap-freelist-presv-setbit-notfree*:
$p \in$ *mem-pools V* $\Longrightarrow$
inv-mempool-info V $\wedge$ *inv-aux-vars V* $\wedge$ *inv-bitmap-freelist V* $\Longrightarrow$
st $\neq$ *FREE* $\Longrightarrow$
get-bit-s V p lv bkn $\neq$ *FREE* $\Longrightarrow$
inv-bitmap-freelist (*set-bit-s V p lv bkn st*)
apply(*simp add*:*inv-bitmap-freelist-def*) **apply**(*simp add*:*set-bit-s-def set-bit-def*)
apply(*simp add*:*Let-def*)
apply *clarsimp* **apply**(*rename-tac ii*)
apply(*rule conjI*)
apply *clarsimp* **apply**(*rename-tac jj*)
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V p*))
[*lv* := (*levels* (*mem-pool-info V p*) ! *lv*)(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *lv*))[*bkn* := *st*]|)] ! *ii*))
= *length* (*bits* (*levels* (*mem-pool-info V p*) ! *ii*))) **prefer** *2*
apply(*case-tac ii* = *lv*) **apply** *fastforce* **apply** *fastforce*
apply(*case-tac ii* = *lv* $\wedge$ *jj* = *bkn*)
using *inv-bitmap-freelist-presv-setbit-notfree-h* **apply** *force*
apply(*subgoal-tac* (*bits* ((*levels* (*mem-pool-info V p*))[*lv* := (*levels* (*mem-pool-info V p*) ! *lv*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *lv*))[*bkn* := *st*]|)] ! *ii*)) ! *jj* =
(*bits* (*levels* (*mem-pool-info V p*) ! *ii*)) ! *jj*) **prefer** *2*
apply(*case-tac ii* $\neq$ *lv*) **apply** *fastforce*
apply(*case-tac jj* $\neq$ *bkn*) **apply** *fastforce*
apply *fastforce*
apply(*subgoal-tac free-list* ((*levels* (*mem-pool-info V p*))[*lv* := (*levels* (*mem-pool-info V p*) ! *lv*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *lv*))[*bkn* := *st*]|)] ! *ii*)
= *free-list* (*levels* (*mem-pool-info V p*) ! *ii*)) **prefer** *2*
apply(*case-tac ii* $\neq$ *lv*) **apply** *fastforce* **apply** *fastforce*
apply *auto*[*1*]

apply(*rule conjI*)
apply *clarsimp* **apply**(*rename-tac jj*)
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V p*))
[*lv* := (*levels* (*mem-pool-info V p*) ! *lv*)(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *lv*))[*bkn* := *st*]|)] ! *ii*))
= *length* (*bits* (*levels* (*mem-pool-info V p*) ! *ii*))) **prefer** *2*
apply(*case-tac ii* = *lv*) **apply** *fastforce* **apply** *fastforce*
apply(*subgoal-tac free-list* ((*levels* (*mem-pool-info V p*))[*lv* := (*levels* (*mem-pool-info V p*) ! *lv*)

(|*bits* := (*bits* (*levels* (*mem-pool-info* *V p*) ! *lv*))[*bkn* := *st*]|)] !
ii)
= *free-list* (*levels* (*mem-pool-info* *V p*) ! *ii*)) **prefer** *2*
apply(*case-tac ii* ≠ *lv*) **apply** *fastforce* **apply** *fastforce*
apply *auto*[*1*]

apply(*subgoal-tac free-list* ((*levels* (*mem-pool-info* *V p*))[*lv* := (*levels* (*mem-pool-info* *V p*) ! *lv*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *V p*) ! *lv*))[*bkn* := *st*]|)] !
ii)
= *free-list* (*levels* (*mem-pool-info* *V p*) ! *ii*)) **prefer** *2*
apply(*case-tac ii* ≠ *lv*) **apply** *fastforce* **apply** *fastforce*
apply *auto*[*1*]
done

end

theory *memory-cover*
imports *invariant*
begin

14 partition of memory addresses of a pool

declare [[*smt-timeout* = *300*]]

this theory shows that all memory blocks are a COVER of address space of a memory pool. A COVER means blocks are disjoint and continuous. It means that for any memory address of a memory pool, the address is in the address range of only one block.

Due to algorithm, address range of each block is implicitly derived. address range of a block at level *ii* with block index *jj* at this level is jj * (max_sz mp div ($4^i i$)) ..¡ Suc jj * (max_sz mp div ($4^i i$)).

abbreviation *addr-in-block mp addr ii jj* ≡
ii < *length* (*levels mp*) ∧ *jj* < *length* (*bits* (*levels mp* ! *ii*))
∧ (*bits* (*levels mp* ! *ii*) ! *jj* = *FREE* ∨ *bits* (*levels mp* ! *ii*) ! *jj* = *FREEING* ∨
bits (*levels mp* ! *ii*) ! *jj* = *ALLOCATED* ∨ *bits* (*levels mp* ! *ii*) ! *jj* = *ALLOCATING*)
∧ *addr* ∈ {*jj* ∗ (*max-sz mp div* (*4* ˆ *ii*)) ..< *Suc jj* ∗ (*max-sz mp div* (*4* ˆ *ii*))}

abbreviation *mem-cover-mp* :: *State* ⇒ *mempool-ref* ⇒ *bool*
where *mem-cover-mp s p* ≡
let mp = *mem-pool-info s p in* (∀ *addr* < *n-max mp* ∗ *max-sz mp*. (∃!(*i*,*j*). *addr-in-block mp addr i j*))

definition *mem-cover* :: *State* ⇒ *bool*

where *mem-cover s* ≡ ∀ *p*∈*mem-pools s*. *mem-cover-mp s p*

lemma *split-div-lemma*:
  assumes *0* < *n*
  shows *n* ∗ *q* ≤ *m* ∧ *m* < *n* ∗ *Suc q* ⟷ *q* = ((*m*::*nat*) *div n*) (**is** *?lhs* ⟷ *?rhs*)
proof
  assume *?rhs*
  with *minus-mod-eq-mult-div* [*symmetric*] **have** *nq*: *n* ∗ *q* = *m* − (*m mod n*) **by** *simp*
  then have *A*: *n* ∗ *q* ≤ *m* **by** *simp*
  have *n* − (*m mod n*) > *0* **using** *mod-less-divisor assms* **by** *auto*
  then have *m* < *m* + (*n* − (*m mod n*)) **by** *simp*
  then have *m* < *n* + (*m* − (*m mod n*)) **by** *simp*
  with *nq* **have** *m* < *n* + *n* ∗ *q* **by** *simp*
  then have *B*: *m* < *n* ∗ *Suc q* **by** *simp*
  from *A B* **show** *?lhs* **..**
next
  assume *P*: *?lhs*
  then show *?rhs*
    using *div-nat-eqI* **by** *blast*
qed

lemma *align-up-ge-low*:
sz1 > *0* ⟹ *sz2* > *sz1* ⟹ *sz2 mod sz1* = *0* ⟹ (*addr*::*nat*) *div sz2* ∗ *sz2* + *sz2* ≥ *addr div sz1* ∗ *sz1* + *sz1*
  apply(*subgoal-tac* ∃ *n*>*0*. *sz2* = *n* ∗ *sz1*) **prefer** *2* **apply** *auto*[*1*]
  apply(*rule subst*[**where** *s*=*addr* − *addr mod sz1* **and** *t*=*addr div sz1* ∗ *sz1*])
    using *minus-mod-eq-div-mult* **apply** *auto*[*1*]
  apply(*rule subst*[**where** *s*=*addr* − *addr mod sz2* **and** *t*=*addr div sz2* ∗ *sz2*])
    using *minus-mod-eq-div-mult* **apply** *auto*[*1*]
  apply(*subgoal-tac sz1* − *addr mod sz1* ≤ *sz2* − *addr mod sz2*)
    apply(*subgoal-tac addr mod sz1* < *sz1*) **prefer** *2* **apply** *simp*
    apply(*subgoal-tac addr mod sz2* < *sz2*) **prefer** *2* **apply** *simp*
    apply *simp*

    apply *clarsimp*
    apply(*case-tac addr mod* (*sz1* ∗ *n*) ≥ *sz1* ∗ (*n* − *1*))
      apply(*subgoal-tac addr mod* (*sz1* ∗ *n*) = *sz1* ∗ (*n* − *1*) + *addr mod sz1*)
        prefer *2* **using** *Suc-lessD Suc-pred′ mod-less-divisor*
              mult-div-mod-eq nat-0-less-mult-iff mod-mult-self4 split-div-lemma
          apply (*metis mod-mult2-eq*)
      apply *clarsimp*
   apply (*metis* (*no-types*, *lifting*) *Nat.diff-diff-right One-nat-def Suc-lessD add-diff-cancel-left′*
         le-add1 less-numeral-extra(*3*) *less-or-eq-imp-le mult.commute mult-eq-if*)
   by (*metis* (*no-types*, *lifting*) *Nat.le-diff-conv2 Suc-lessD add-mono-thms-linordered-semiring*(*1*)
         diff-le-self less-numeral-extra(*3*) *mod-le-divisor mult.commute mult-eq-if*
mult-pos-pos nat-le-linear)

lemma *addr-exist-block-h1-1*:
li < *ii* ⟹ *ii* < *nl* ⟹ (*4*::*nat*) ^ *nl div 4* ^ *ii* < *4* ^ *nl div 4* ^ *li*
  apply(*rule subst*[**where** *s*=*4* ^ (*nl* − *ii*) **and** *t*=*4* ^ *nl div 4* ^ *ii*])
    apply (*simp add*: *div2-eq-minus*)
  apply(*rule subst*[**where** *s*=*4* ^ (*nl* − *li*) **and** *t*=*4* ^ *nl div 4* ^ *li*])
    apply (*simp add*: *div2-eq-minus*)
  apply *auto*
done

lemma *mod-time*: (*x*::*nat*) *mod m* = *0* ⟹ *n* ∗ *x mod* (*n* ∗ *m*) = *0*
by *simp*

lemma *addr-exist-block-h1*:
li < *ii* ⟹
  ∃ *n*>*0*. *msz* = (*4* ∗ *n*) ∗ (*4* ^ *nl*) ⟹
  ii < *nl* ⟹
  Suc (*addr div* (*msz div 4* ^ *ii*)) ∗ (*msz div 4* ^ *ii*)
  ≤ *Suc* (*addr div* (*msz div 4* ^ *ii*) *div 4* ^ (*ii* − *li*)) ∗ (*msz div 4* ^ *li*)
apply(*rule subst*[**where** *s*=(*addr div* (*msz div 4* ^ *ii*)) ∗ (*msz div 4* ^ *ii*) + (*msz div 4* ^ *ii*)
  and *t*=*Suc* (*addr div* (*msz div 4* ^ *ii*)) ∗ (*msz div 4* ^ *ii*)]) **apply** *auto*[*1*]
apply(*rule subst*[**where** *s*=(*addr div* (*msz div 4* ^ *ii*) *div 4* ^ (*ii* − *li*)) ∗ (*msz div 4* ^ *li*) + (*msz div 4* ^ *li*)
  and *t*=*Suc* (*addr div* (*msz div 4* ^ *ii*) *div 4* ^ (*ii* − *li*)) ∗ (*msz div 4* ^ *li*)]) **apply** *auto*[*1*]
apply(*rule subst*[**where** *s*=*addr div* (*msz div 4* ^ *li*) **and** *t*=*addr div* (*msz div 4* ^ *ii*) *div 4* ^ (*ii* − *li*)])
  apply(*rule subst*[**where** *s*=*addr div* (*msz div 4* ^ *ii* ∗ *4* ^ (*ii* − *li*)) **and** *t*=*addr div* (*msz div 4* ^ *ii*) *div 4* ^ (*ii* − *li*)])
    using *div2-eq-divmul*[*of addr msz div 4* ^ *ii 4* ^ (*ii* − *li*)] **apply** *simp*
  apply(*rule subst*[**where** *s*=*msz div 4* ^ *li* **and** *t*=*msz div 4* ^ *ii* ∗ *4* ^ (*ii* − *li*)])
    apply(*subgoal-tac msz mod 4* ^ *ii* = *0*) **prefer** *2*
      using *ge-pow-mod-0* **apply** *auto*[*1*]
    apply (*smt add-diff-inverse-nat less-imp-le-nat mod-div-self mult.left-commute nonzero-mult-div-cancel-left not-less power-add power-not-zero rel-simps*(*76*))
  apply *fast*

apply(*rule align-up-ge-low*[*of msz div 4* ^ *ii msz div 4* ^ *li addr*])
 apply (*metis ge-pow-mod-0 mod-div-self nat-0-less-mult-iff zero-less-numeral zero-less-power*)
  apply *clarsimp* **apply**(*subgoal-tac 4* ^ *nl div 4* ^ *ii* < *4* ^ *nl div 4* ^ *li*)
    prefer *2* **using** *addr-exist-block-h1-1*[*of li ii nl*] **apply** *simp*
    using *m-mod-div pow-mod-0* **apply** *auto*[*1*]
  apply *clarsimp* **using** *mod-time*[*of 4* ^ *nl div 4* ^ *li 4* ^ *nl div 4* ^ *ii*]
   by (*smt less-imp-add-positive mod-div-self mod-mult-self1-is-0 mult.left-commute nonzero-mult-div-cancel-left power-add power-not-zero zero-neq-numeral*)

lemma *divornoe-imp-div-noe-neigh*:
$\forall$ *li≤ii. get-bit-s s p li (jj div 4 ^ (ii − li)) = DIVIDED* $\vee$ *get-bit-s s p li (jj div 4 ^ (ii − li)) = NOEXIST* $\Longrightarrow$
get-bit-s s p NULL (jj div 4 ^ ii) = DIVIDED $\Longrightarrow$
get-bit-s s p ii jj = NOEXIST $\Longrightarrow$
ii > 0 $\Longrightarrow$
$\exists$ *n. n > 0* $\wedge$ *n≤ii* $\wedge$ *get-bit-s s p (n−1) (jj div 4 ^ (ii − (n−1))) = DIVIDED* $\wedge$
get-bit-s s p n (jj div 4 ^ (ii − n)) = NOEXIST
apply(*induction ii arbitrary: jj*)
apply *simp*

apply(*case-tac get-bit-s s p ii (jj div 4) = DIVIDED*)
apply *auto[1]*

apply(*subgoal-tac get-bit-s s p ii (jj div 4) = NOEXIST*)
prefer *2*
apply (*metis One-nat-def Suc-diff-Suc diff-self-eq-0 lessI less-imp-le-nat power-one-right*)

apply(*case-tac ii = 0*) **apply** *auto[1]*

apply(*subgoal-tac* $\forall$ *li≤ii. get-bit-s s p li ((jj div 4) div 4 ^ (ii − li)) = DIVIDED*
$\vee$ *get-bit-s s p li ((jj div 4) div 4 ^ (ii − li)) = NOEXIST*)
prefer *2* **apply** *clarsimp*
apply (*metis Suc-diff-le div-mult2-eq le-SucI power-Suc*)

apply(*subgoal-tac* $\exists$ *n>NULL. n ≤ ii* $\wedge$
get-bit-s s p (n − 1) ((jj div 4) div 4 ^ (ii − (n − 1))) = DIVIDED $\wedge$
get-bit-s s p n ((jj div 4) div 4 ^ (ii − n)) = NOEXIST)
prefer *2* **apply** (*simp add: Divides.div-mult2-eq*)
proof −
fix *iia :: nat* **and** *jja :: nat*
assume $\exists$ *n>NULL. n ≤ iia* $\wedge$ *get-bit-s s p (n − 1) (jja div 4 div 4 ^ (iia − (n − 1))) = DIVIDED*
$\wedge$ *get-bit-s s p n (jja div 4 div 4 ^ (iia − n)) = NOEXIST*
then obtain *nn :: nat* **where**
f1: NULL < nn $\wedge$ *nn ≤ iia* $\wedge$ *get-bit-s s p nn (jja div 4 div 4 ^ (iia − nn)) = NOEXIST*
$\wedge$ *get-bit-s s p (nn − 1) (jja div 4 div 4 ^ (iia − (nn − 1))) = DIVIDED*
by *meson*
then have *f2: get-bit-s s p nn (jja div 4 ^ Suc (iia − nn)) = NOEXIST*
by (*metis (no-types) div-mult2-eq semiring-normalization-rules(27)*)
have *f3: get-bit-s s p (nn − 1) (jja div 4 ^ Suc (Suc (iia − nn))) = DIVIDED*
using *f1* **by** (*metis (no-types) Suc-diff-eq-diff-pred Suc-diff-le div-mult2-eq semiring-normalization-rules(27)*)
have *nn ≤ iia* $\wedge$ *NULL < nn*
using *f1* **by** *meson*
then show $\exists$ *n>NULL. n ≤ Suc iia* $\wedge$ *get-bit-s s p (n − 1) (jja div 4 ^ (Suc iia*

− (n − 1))) = DIVIDED
∧ get-bit-s s p n (jja div 4 ^ (Suc iia − n)) = NOEXIST
using *f3 f2 Suc-diff-le le-Suc-eq* **by** *auto*
qed

lemma *addr-exist-block*:
assumes
p2: *inv-bitmap0 s* **and**
p3: *inv-bitmap s* **and**
p6: *inv-mempool-info s* **and**
p4: *p* ∈ *mem-pools s* **and**
p7: *inv-bitmapn s* **and**
p5: *addr* < *n-max* (*mem-pool-info s p*) ∗ *max-sz* (*mem-pool-info s p*)
shows ∃ *i j*. *addr-in-block* (*mem-pool-info s p*) *addr i j*
proof −
obtain *ii* **where** *ii*: *ii* = *length* (*levels* (*mem-pool-info s p*)) − *1* **by** *auto*
obtain *jj* **where** *jj*: *jj* = *addr div* (*max-sz* (*mem-pool-info s p*) *div* (*4* ^ *ii*)) **by** *auto*
have *bits-len-nmax*: ∀ *i*<*length* (*levels* (*mem-pool-info s p*)). *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i*)) = (*n-max* (*mem-pool-info s p*)) ∗ *4* ^ *i*
using *p6 p4* **by**(*simp add:inv-mempool-info-def Let-def*)
have *maxsz*: ∃ *n*>*0*. *max-sz* (*mem-pool-info s p*) = (*4* ∗ *n*) ∗ (*4* ^ *n-levels* (*mem-pool-info s p*))
using *p4 p6* **apply**(*simp add:inv-mempool-info-def Let-def*) **by** *auto*
have *nl-eq-len*: *n-levels* (*mem-pool-info s p*) = *length* (*levels* (*mem-pool-info s p*))
using *p4 p6* **by**(*simp add:inv-mempool-info-def Let-def*)
from *ii* **have** *ii-len*: *ii* < *length* (*levels* (*mem-pool-info s p*))
by (*metis diff-less inv-mempool-info-def length-greater-0-conv p4 p6 rel-simps*(*68*))
from *ii p6* **have** *blk-ii*: *max-sz* (*mem-pool-info s p*) *div 4* ^ *ii* > *0*
by (*smt Euclidean-Division.div-eq-0-iff divisors-zero gr0I ii-len less-imp-le-nat m-mod-div maxsz mod-if nl-eq-len pow-mod-0 power-not-zero zero-neq-numeral*)
hence *addr-ran*: *addr* ∈ {*jj* ∗ (*max-sz* (*mem-pool-info s p*) *div* (*4* ^ *ii*)) ..< *Suc jj* ∗ (*max-sz* (*mem-pool-info s p*) *div* (*4* ^ *ii*))}
using *jj div-in-suc*[*of max-sz* (*mem-pool-info s p*) *div 4* ^ *ii jj addr*] **by** *blast*
have *jj-lt-maxdiv4ii*: *jj* < *n-max* (*mem-pool-info s p*) ∗ *4* ^ *ii*
apply(*rule subst*[**where** *s*=*addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *ii*) **and** *t*=*jj*]) **using** *jj* **apply** *fast*
apply(*rule subst*[**where** *s*=*n-max* (*mem-pool-info s p*) ∗ *max-sz* (*mem-pool-info s p*) *div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *ii*)
and *t*=*n-max* (*mem-pool-info s p*) ∗ *4* ^ *ii*]) **using** *ii-len maxsz*
apply (*metis* (*no-types*, *lifting*) *blk-ii ge-pow-mod-0 inv-mempool-info-def m-mod-div mod-div-self mod-mult-self1-is-0 neq0-conv nonzero-mult-div-cancel-left p4 p6*)
apply(*rule mod-div-gt*[*of addr n-max* (*mem-pool-info s p*) ∗ *max-sz* (*mem-pool-info s p*)
max-sz (*mem-pool-info s p*) *div 4* ^ *ii*]) **using** *p5* **apply** *fast*

using *maxsz nl-eq-len*
apply (*metis ge-pow-mod-0 ii-len mod-div-self mod-mult-right-eq mod-mult-self1-is-0 mult-0-right*)
done
have *lvlii-eq-last*: *levels* (*mem-pool-info s p*) ! *ii* = *last* (*levels* (*mem-pool-info s p*))
apply(*subgoal-tac length* (*levels* (*mem-pool-info s p*)) > *0*)
prefer *2* **using** *p4 p6 ii jj-lt-maxdiv4ii p4 p6 ii-len* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
using *ii* **apply** *clarsimp*
by (*simp add*: *last-conv-nth*)
have *jj-lt-len-lstbits*: *jj* < *length* (*bits* (*last* (*levels* (*mem-pool-info s p*))))
using *ii jj-lt-maxdiv4ii p4 p6 ii-len* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info s p*) ! *ii*)) = *n-max* (*mem-pool-info s p*) ∗ *4* ˆ *ii*)
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac levels* (*mem-pool-info s p*) ! *ii* = *last* (*levels* (*mem-pool-info s p*)))
prefer *2* **apply**(*subgoal-tac length* (*levels* (*mem-pool-info s p*)) > *0*)
prefer *2* **using** *p4 p6* **apply**(*simp add*:*inv-mempool-info-def Let-def*) **apply** *clarsimp*
apply (*simp add*: *last-conv-nth*)
by *fastforce*

have ∃ *li*≤*ii*. *addr-in-block* (*mem-pool-info s p*) *addr li* (*jj div 4* ˆ (*ii* − *li*))
proof −
{
assume *asm*: ¬ (∃ *li*≤*ii*. *addr-in-block* (*mem-pool-info s p*) *addr li* (*jj div 4* ˆ (*ii* − *li*)))

from *asm* **have** ∀ *li*≤*ii*. ¬ *addr-in-block* (*mem-pool-info s p*) *addr li* (*jj div 4* ˆ (*ii* − *li*)) **by** *fast*
moreover
from *ii* **have** *ii-len*: *ii* < *length* (*levels* (*mem-pool-info s p*))
by (*metis diff-less inv-mempool-info-def length-greater-0-conv p4 p6 rel-simps*(*68*))
moreover
have ∀ *li*≤*ii*. *addr* ∈ {*jj div 4* ˆ (*ii* − *li*) ∗ (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *li*)..<
Suc (*jj div 4* ˆ (*ii* − *li*)) ∗ (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *li*)}
apply(*subgoal-tac* ∃ *n*>*0*. *max-sz* (*mem-pool-info s p*) = (*4* ∗ *n*) ∗ (*4* ˆ *n-levels* (*mem-pool-info s p*)))
prefer *2* **using** *p4 p6* **apply**(*simp add*:*inv-mempool-info-def Let-def*) **apply** *auto*[*1*]
apply(*subgoal-tac n-levels* (*mem-pool-info s p*) = *length* (*levels* (*mem-pool-info s p*)) ∧
length (*levels* (*mem-pool-info s p*)) > *0*)
prefer *2* **using** *p4 p6* **apply**(*simp add*:*inv-mempool-info-def Let-def*)

apply *auto*[*1*]
apply *clarify* **apply** *auto*
apply(*subgoal-tac jj* $*$ (*max-sz* (*mem-pool-info s p*) *div 4* ^ *ii*)
$\geq$ *jj div 4* ^ (*ii* $-$ *li*) $*$ (*max-sz* (*mem-pool-info s p*) *div 4* ^ *li*))
prefer *2* **apply**(*case-tac li* = *ii*) **apply** *auto*[*1*]
using *Divides.div-mult2-eq Groups.mult-ac*(*2*) *blk-ii add-diff-inverse-nat calculation*(*2*)
div-mult-self-is-m divisors-zero ge-pow-mod-0 mod-div-self neq0-conv not-less power-add
semiring-normalization-rules(*17*) *split-div-lemma zero-less-numeral zero-less-power*
apply (*smt div-mult-self1-is-m nat-mult-le-cancel-disj*)
using *addr-ran* **apply** *auto*[*1*]

apply(*subgoal-tac Suc jj* $*$ (*max-sz* (*mem-pool-info s p*) *div* (*4* ^ *ii*))
$\leq$ *Suc* (*jj div 4* ^ (*ii* $-$ *li*)) $*$ (*max-sz* (*mem-pool-info s p*) *div 4* ^ *li*))
prefer *2* **apply**(*case-tac li* = *ii*) **apply** *simp*
apply(*rule subst*[**where** *s*=*addr div* (*max-sz* (*mem-pool-info s p*) *div* (*4* ^ *ii*)) **and** *t*=*jj*]) **using** *jj* **apply** *fast*
using *addr-exist-block-h1*[*of - ii max-sz* (*mem-pool-info s p*) *n-levels* (*mem-pool-info s p*) *addr*]
ii-len nl-eq-len maxsz **apply** *fastforce*
using *addr-ran ii-len* **apply** *auto*[*1*]
done
moreover
have *li-len*: $\forall$ *li*$\leq$*ii*. *jj div 4* ^ (*ii* $-$ *li*) < *length* (*bits* (*levels* (*mem-pool-info s p*) ! *li*))
apply *clarsimp*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info s p*) ! *li*)) = (*n-max* (*mem-pool-info s p*)) $*$ *4* ^ *li*)
prefer *2* **using** *p4 p6 ii-len* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
using *jj maxsz nl-eq-len jj-lt-maxdiv4ii Divides.div-mult2-eq add-diff-cancel-left*$'$ *blk-ii div-eq-0-iff gr-implies-not0*
le-Suc-ex less-not-refl2 mult.commute mult.left-commute mult-0 mult-is-0 p5 power-add
by (*smt not-less*)
ultimately have $\forall$ *li*$\leq$*ii*. $\neg$ (*get-bit-s s p li* (*jj div 4* ^ (*ii* $-$ *li*)) = *FREE* $\vee$
get-bit-s s p li (*jj div 4* ^ (*ii* $-$ *li*)) = *FREEING* $\vee$
get-bit-s s p li (*jj div 4* ^ (*ii* $-$ *li*)) = *ALLOCATED* $\vee$ *get-bit-s s p li* (*jj div 4* ^ (*ii* $-$ *li*)) = *ALLOCATING*)
by *auto*
hence *all-dv-ne*: $\forall$ *li*$\leq$*ii*. *get-bit-s s p li* (*jj div 4* ^ (*ii* $-$ *li*)) = *DIVIDED* $\vee$ *get-bit-s s p li* (*jj div 4* ^ (*ii* $-$ *li*)) = *NOEXIST*
using *BlockState.exhaust* **by** *blast*
moreover
have *bit-lvl0*: *get-bit-s s p 0* (*jj div 4* ^ *ii*) = *DIVIDED* **using** *all-dv-ne p2 p4* **apply**(*simp add*:*inv-bitmap0-def Let-def*)
using *li-len* **by** *fastforce*

moreover
have *bit-lvln*: *get-bit-s s p ii jj* = *NOEXIST*
using *all-dv-ne p4 p7* **apply**(*simp add*:*inv-bitmapn-def inv-bitmap-not4free-def Let-def*)
using *jj-lt-len-lstbits ii lvlii-eq-last*
by (*metis One-nat-def diff-self-eq-0 div-by-Suc-0 eq-imp-le power-0*)
ultimately have $\exists n.\ n > 0 \wedge n \leq ii \wedge$ *get-bit-s s p* $(n-1)$ $(jj\ div\ 4\ \hat{}\ (ii - (n-1))) = DIVIDED \wedge$
get-bit-s s p n $(jj\ div\ 4\ \hat{}\ (ii - n)) = NOEXIST$
using *divornoe-imp-div-noe-neigh*[*of ii s p jj*] **by** *fastforce*

then obtain *n* **where** $n > 0 \wedge n \leq ii \wedge$ *get-bit-s s p* $(n-1)$ $(jj\ div\ 4\ \hat{}\ (ii - (n-1))) = DIVIDED \wedge$
get-bit-s s p n $(jj\ div\ 4\ \hat{}\ (ii - n)) = NOEXIST$ **by** *auto*
moreover
with *p3* **have** *get-bit-s s p* $(n - Suc\ NULL)$ $(jj\ div\ 4\ \hat{}\ (ii - (n - Suc\ NULL))) \neq DIVIDED$
apply(*simp add*:*inv-bitmap-def Let-def*)
using *Divides.div-mult2-eq One-nat-def Suc-diff-eq-diff-pred Suc-pred diff-Suc-eq-diff-pred diff-commute ii less-Suc-eq-le li-len p4 power-minus-mult zero-less-diff*
by (*smt le-imp-less-Suc zero-le-numeral*)
ultimately have *False* **by** *simp*

} **thus** *?thesis* **by** *auto*
qed

thus *?thesis* **by** *auto*
qed

lemma *div-imp-up-alldiv*:
$\forall$ *i1 j1 j2*. *inv-bitmap s* $\wedge$ *inv-bitmap0 s* $\wedge$
inv-mempool-info s $\wedge$
$p \in$ *mem-pools s* $\wedge$
$i1 <$ *length* (*levels* (*mem-pool-info s p*)) $\wedge$
$j1 <$ *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i1*)) $\wedge$
$i2 <$ *length* (*levels* (*mem-pool-info s p*)) $\wedge$
$j2 <$ *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i2*)) $\wedge$
get-bit-s s p i2 j2 = *DIVIDED* $\wedge$
$i1 < i2$ $\wedge$
$j1 = j2\ div\ 4\ \hat{}\ (i2 - i1) \longrightarrow$
get-bit-s s p i1 j1 = *DIVIDED*
apply(*induct i2*)
apply *simp*

apply *clarsimp*
apply(*case-tac i1* = *i2*)
apply *clarsimp* **apply**(*simp add*:*inv-bitmap-def Let-def*)

apply *fastforce*

apply(*subgoal-tac i1 < i2*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac get-bit-s s p i2* (*j2 div 4*) = *DIVIDED*) **prefer** *2*
apply(*simp add*:*inv-bitmap-def Let-def*) **apply** *fastforce*
apply(*subgoal-tac get-bit-s s p i1* ((*j2 div 4*) *div 4* ˆ (*i2* − *i1*)) = *DIVIDED*)
prefer *2*
apply(*subgoal-tac* (*j2 div 4*) *div 4* ˆ (*i2* − *i1*) < *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i1*)))
prefer *2* **apply** (*simp add*: *Divides.div-mult2-eq Suc-diff-le*)
apply(*subgoal-tac j2 div 4* < *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i2*)))
prefer *2* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
apply *fastforce*
apply(*subgoal-tac j2 div 4 div 4* ˆ (*i2* − *i1*) = *j2 div 4* ˆ (*Suc i2* − *i1*))
prefer *2*
apply (*metis Suc-diff-le div-mult2-eq less-or-eq-imp-le power-Suc*)
apply *fastforce*
done

lemma *block-imp-up-alldiv*:
inv-bitmap s ⟹ *inv-bitmap0 s* ⟹
inv-mempool-info s ⟹
p ∈ *mem-pools s* ⟹
i1 < *length* (*levels* (*mem-pool-info s p*)) ⟹
j1 < *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i1*)) ⟹
i2 < *length* (*levels* (*mem-pool-info s p*)) ⟹
j2 < *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i2*)) ⟹
(*get-bit-s s p i2 j2* = *FREE* ∨
get-bit-s s p i2 j2 = *FREEING* ∨ *get-bit-s s p i2 j2* = *ALLOCATED* ∨ *get-bit-s s p i2 j2* = *ALLOCATING*) ⟹
i1 < *i2* ⟹
j1 = *j2 div 4* ˆ (*i2* − *i1*) ⟹
get-bit-s s p i1 j1 = *DIVIDED*
apply(*subgoal-tac get-bit-s s p* (*i2* − *1*) (*j2 div 4*) = *DIVIDED*)
prefer *2* **apply**(*simp add*:*inv-bitmap-def Let-def*)
apply (*metis neq0-conv not-less-zero*)
apply(*case-tac i1* = *i2* − *1*)
apply *simp*

apply *clarsimp*
apply(*rule div-imp-up-alldiv*[*rule-format*,*of s p i1 j2 div 4* ˆ (*i2* − *i1*) *i2* − *1 j2 div 4*])
apply *clarsimp*
apply(*rule conjI*) **apply** *simp*
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def Let-def*)
using *One-nat-def div-eq-0-iff gr-implies-not0 nat-0-less-mult-iff*
apply (*metis* (*no-types*, *lifting*) *less-mult-imp-div-less nat-neq-iff power-minus-mult semiring-normalization-rules*(*17*))
using *Divides.div-mult2-eq Suc-diff-Suc Suc-pred linorder-neqE-nat not-less-eq*

not-less-zero power-Suc
  by (*metis not-less*)

lemma *addr-in-same-block*:
inv-bitmap0 s $\Longrightarrow$ *inv-bitmap s* $\Longrightarrow$ *inv-mempool-info s* $\Longrightarrow$
 p $\in$ *mem-pools s* $\Longrightarrow$ *addr* $<$ *n-max* (*mem-pool-info s p*) $*$ *max-sz* (*mem-pool-info s p*) $\Longrightarrow$
 addr-in-block (*mem-pool-info s p*) *addr i1 j1* $\Longrightarrow$
 addr-in-block (*mem-pool-info s p*) *addr i2 j2* $\Longrightarrow$
 i1 = *i2* $\wedge$ *j1* = *j2*
apply(*case-tac i1* = *i2*)
  apply(*rule conjI*) **apply** *fast*
  apply *clarsimp*
  apply(*case-tac j1* $<$ *j2*)
    apply (*smt Groups.mult-ac*(*2*) *mult-Suc-right nat-0-less-mult-iff neq0-conv not-le split-div-lemma*)
  apply(*case-tac j1* $>$ *j2*)
    apply (*smt Groups.mult-ac*(*2*) *mult-Suc-right nat-0-less-mult-iff neq0-conv not-le split-div-lemma*)
  apply *simp*

apply(*subgoal-tac* $\exists$ *n*$>$*0*. *max-sz* (*mem-pool-info s p*) = (*4* $*$ *n*) $*$ (*4* ^ *n-levels* (*mem-pool-info s p*)))
  prefer *2* **apply**(*simp add*:*inv-mempool-info-def Let-def*) **apply** *metis*
apply(*subgoal-tac length* (*levels* (*mem-pool-info s p*)) = *n-levels* (*mem-pool-info s p*))
  prefer *2* **apply**(*simp add*:*inv-mempool-info-def Let-def*)

apply(*case-tac i1* $<$ *i2*)
apply(*subgoal-tac addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *i1*) = *j1*)
  prefer *2* **using** *addr-in-div*[*of addr j1 max-sz* (*mem-pool-info s p*) *div 4* ^ *i1*] **apply** *simp*
apply(*subgoal-tac addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *i2*) = *j2*)
  prefer *2* **using** *addr-in-div*[*of addr j2 max-sz* (*mem-pool-info s p*) *div 4* ^ *i2*] **apply** *simp*

apply(*subgoal-tac j1* = *j2 div* (*4* ^ (*i2* $-$ *i1*))) **prefer** *2*
  apply(*rule subst*[**where** *s*=*addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *i2*) *div 4* ^ (*i2* $-$ *i1*) **and** *t*=*j2 div 4* ^ (*i2* $-$ *i1*)])
    apply *fast*
  apply(*rule subst*[**where** *s*=*addr div* ((*max-sz* (*mem-pool-info s p*) *div 4* ^ *i2*) $*$ *4* ^ (*i2* $-$ *i1*))
                and *t*=*addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ^ *i2*) *div 4* ^ (*i2* $-$ *i1*)])
    using *div2-eq-divmul*[*of addr max-sz* (*mem-pool-info s p*) *div 4* ^ *i2 4* ^ (*i2* $-$ *i1*)] **apply** *simp*
  apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info s p*) *div 4* ^ *i1* **and**
                    t=*max-sz* (*mem-pool-info s p*) *div 4* ^ *i2* $*$ *4* ^ (*i2* $-$ *i1*)])

apply(*subgoal-tac max-sz* (*mem-pool-info s p*) *mod* (*4* ˆ *i1*) = *0*)
prefer *2* **apply** (*metis ge-pow-mod-0*)
apply(*subgoal-tac max-sz* (*mem-pool-info s p*) *mod* (*4* ˆ *i2*) = *0*)
prefer *2* **apply** (*metis ge-pow-mod-0*)
apply(*smt add-diff-inverse-nat div2-eq-minus less-imp-le-nat m-mod-div minus-div-mult-eq-mod minus-mult-div-eq-mod mod-div-self mod-mult-self2-is-0 not-less power-add zero-neq-numeral*)
apply *fast*

apply(*subgoal-tac get-bit-s s p i1 j1* = *DIVIDED*)
prefer *2* **using** *block-imp-up-alldiv*[*of s p i1 j1 i2 j2*] **apply** *fast*
apply *auto*[*1*]

apply(*case-tac i1* > *i2*)
apply(*subgoal-tac addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1*) = *j1*)
prefer *2* **using** *addr-in-div*[*of addr j1 max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1*]
apply *simp*
apply(*subgoal-tac addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i2*) = *j2*)
prefer *2* **using** *addr-in-div*[*of addr j2 max-sz* (*mem-pool-info s p*) *div 4* ˆ *i2*]
apply *simp*

apply(*subgoal-tac j2* = *j1 div* (*4* ˆ (*i1* − *i2*))) **prefer** *2*
apply(*rule subst*[**where** *s*=*addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1*) *div 4* ˆ (*i1* − *i2*) **and** *t*=*j1 div 4* ˆ (*i1* − *i2*)])
apply *fast*
apply(*rule subst*[**where** *s*=*addr div* ((*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1*) ∗ *4* ˆ (*i1* − *i2*))
and *t*=*addr div* (*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1*) *div 4* ˆ (*i1* − *i2*)])
using *div2-eq-divmul*[*of addr max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1 4* ˆ (*i1* − *i2*)] **apply** *simp*
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i2* **and**
t=*max-sz* (*mem-pool-info s p*) *div 4* ˆ *i1* ∗ *4* ˆ (*i1* − *i2*)])
apply(*subgoal-tac max-sz* (*mem-pool-info s p*) *mod* (*4* ˆ *i1*) = *0*)
prefer *2* **apply** (*metis ge-pow-mod-0*)
apply(*subgoal-tac max-sz* (*mem-pool-info s p*) *mod* (*4* ˆ *i2*) = *0*)
prefer *2* **apply** (*metis ge-pow-mod-0*)
apply(*smt add-diff-inverse-nat div2-eq-minus less-imp-le-nat m-mod-div minus-div-mult-eq-mod minus-mult-div-eq-mod mod-div-self mod-mult-self2-is-0 not-less power-add zero-neq-numeral*)
apply *fast*

apply(*subgoal-tac get-bit-s s p i2 j2* = *DIVIDED*)
prefer *2* **using** *block-imp-up-alldiv*[*of s p i2 j2 i1 j1*] **apply** *fast*
apply *auto*[*1*]

apply *auto*
done

lemma *inv-impl-mem-cover'*:
inv-mempool-info s $\Longrightarrow$
inv-bitmap0 s $\Longrightarrow$ *inv-bitmap s* $\Longrightarrow$ *inv-bitmapn s* $\Longrightarrow$ *mem-cover s*
apply(*simp add*: *mem-cover-def Let-def*)
apply *clarify*
apply(*rule ex-ex1I*)
apply *clarsimp* **using** *addr-exist-block*[*of s*] **apply** *fastforce*
apply *clarsimp* **using** *addr-in-same-block*[*of s*] **apply** *force*
done

lemma *inv-impl-mem-cover*: *inv s* $\Longrightarrow$ *mem-cover s*
apply(*simp add*:*inv-def*)
using *inv-impl-mem-cover'* **apply** *fast*
done

abbreviation *divide-noexist-cont'* :: *State* $\Rightarrow$ *mempool-ref* $\Rightarrow$ *bool*
where *divide-noexist-cont' s p* $\equiv$
let mp = mem-pool-info s p in
$\forall$ *i* < *length* (*levels mp*).
let bts = bits (*levels mp* ! *i*) *in*
($\forall$ *j* < *length bts*. (*bts* ! *j* = *DIVIDED* $\longrightarrow$ *i* > *0* $\longrightarrow$ (*bits* (*levels mp* ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
$\wedge$ (*bts* ! *j* = *NOEXIST* $\longrightarrow$ *i* < *length* (*levels mp*) − *1* $\longrightarrow$ *noexist-bits mp* (*i+1*) (*j*∗*4*)))

definition *divide-noexist-cont* :: *State* $\Rightarrow$ *bool*
where *divide-noexist-cont s* $\equiv$
$\forall$ *p*$\in$*mem-pools s*. *divide-noexist-cont' s p*

end

theory *rg-cond*
imports *mem-spec invariant*
begin

15 Rely-guarantee condition of events

15.1 defs of rely-guarantee conditions

definition *lvars-nochange* :: *Thread* $\Rightarrow$ *State* $\Rightarrow$ *State* $\Rightarrow$ *bool*
where *lvars-nochange t r s* $\equiv$
i r t = i s t $\wedge$ *j r t = j s t* $\wedge$ *ret r t = ret s t*
$\wedge$ *endt r t = endt s t* $\wedge$ *rf r t = rf s t* $\wedge$ *tmout r t = tmout s t*
$\wedge$ *lsizes r t = lsizes s t* $\wedge$ *alloc-l r t = alloc-l s t* $\wedge$ *free-l r t = free-l s t*
$\wedge$ *from-l r t = from-l s t* $\wedge$ *blk r t = blk s t* $\wedge$ *nodev r t = nodev s t*
$\wedge$ *bn r t = bn s t* $\wedge$ *lbn r t = lbn s t* $\wedge$ *lsz r t = lsz s t* $\wedge$ *block2 r t = block2 s t*
$\wedge$ *free-block-r r t = free-block-r s t* $\wedge$ *alloc-lsize-r r t = alloc-lsize-r s t* $\wedge$ *lvl r t = lvl s t* $\wedge$ *bb r t = bb s t*

$\wedge$ *block-pt r t* = *block-pt s t* $\wedge$ *th r t* = *th s t* $\wedge$ *need-resched r t* = *need-resched s t*
$\wedge$ *mempoolalloc-ret r t* = *mempoolalloc-ret s t*
$\wedge$ *freeing-node r t* = *freeing-node s t* $\wedge$ *allocating-node r t* = *allocating-node s t*

definition *lvars-nochange-rel* :: *Thread* $\Rightarrow$ (*State* $\times$ *State*) *set*
where *lvars-nochange-rel t* $\equiv$ {(*s*,*r*). *lvars-nochange t s r*}

definition *lvars-nochange-4all* :: (*State* $\times$ *State*) *set*
where *lvars-nochange-4all* $\equiv$ {(*s*,*r*). $\forall$ *t*. *lvars-nochange t s r*}

definition *lvars-nochange1* :: *Thread* $\Rightarrow$ *State* $\Rightarrow$ *State* $\Rightarrow$ *bool*
where *lvars-nochange1 t r s* $\equiv$ *freeing-node r t* = *freeing-node s t* $\wedge$ *allocating-node r t* = *allocating-node s t*

definition *lvars-nochange1-rel* :: *Thread* $\Rightarrow$ (*State* $\times$ *State*) *set*
where *lvars-nochange1-rel t* $\equiv$ {(*s*,*r*). *lvars-nochange1 t s r*}

definition *lvars-nochange1-4all* :: (*State* $\times$ *State*) *set*
where *lvars-nochange1-4all* $\equiv$ {(*s*,*r*). $\forall$ *t*. *lvars-nochange1 t s r*}

lemma *lvars-nochange-trans*:
lvars-nochange t x y $\Longrightarrow$ *lvars-nochange t y z* $\Longrightarrow$ *lvars-nochange t x z*
apply(*simp add*:*lvars-nochange-def*)
done

lemma *lvars-nochange-sym*:
lvars-nochange t x y $\Longrightarrow$ *lvars-nochange t y x*
apply(*simp add*:*lvars-nochange-def*)
done

lemma *lvars-nochange-refl*:
lvars-nochange t x x
apply(*simp add*:*lvars-nochange-def*)
done

lemma *lvars-nc-nc1*: *lvars-nochange t r s* $\Longrightarrow$ *lvars-nochange1 t r s*
unfolding *lvars-nochange-def lvars-nochange1-def* **by** *simp*

lemma *lv-noch-all1*: (*s*,*r*)$\in$*lvars-nochange-4all*
$\Longrightarrow$ (*s*,*r*)$\in$*lvars-nochange-rel t* $\wedge$ ($\forall$ *t'*. *t'* $\neq$ *t* $\longrightarrow$ (*s*,*r*)$\in$*lvars-nochange-rel t'*)
unfolding *lvars-nochange-4all-def lvars-nochange-rel-def* **by** *auto*

lemma *lv-noch-all2*: (*s*,*r*)$\in$*lvars-nochange-rel t* $\wedge$ ($\forall$ *t'*. *t'* $\neq$ *t* $\longrightarrow$ *lvars-nochange t' s r*)
$\Longrightarrow$ (*s*,*r*)$\in$*lvars-nochange-4all*
unfolding *lvars-nochange-4all-def lvars-nochange-rel-def* **by** *auto*

definition *gvars-nochange* :: *State* $\Rightarrow$ *State* $\Rightarrow$ *bool*

where *gvars-nochange s r* ≡ *cur r* = *cur s* ∧ *tick r* = *tick s* ∧ *thd-state r* = *thd-state s*
∧ *mem-pools r* = *mem-pools s* ∧ *mem-pool-info r* = *mem-pool-info s*

definition *gvars-nochange-rel* :: (*State* × *State*) *set*
where *gvars-nochange-rel* ≡ {(*s*,*r*). *gvars-nochange s r*}

definition *gvars-conf* :: *State* ⇒ *State* ⇒ *bool*
where *gvars-conf s r* ≡
  mem-pools r = *mem-pools s*
   ∧ (∀ *p*. *buf* (*mem-pool-info s p*) = *buf* (*mem-pool-info r p*)
     ∧ *max-sz* (*mem-pool-info s p*) = *max-sz* (*mem-pool-info r p*)
     ∧ *n-max* (*mem-pool-info s p*) = *n-max* (*mem-pool-info r p*)
     ∧ *n-levels* (*mem-pool-info s p*) = *n-levels* (*mem-pool-info r p*)
     ∧ *length* (*levels* (*mem-pool-info s p*)) = *length* (*levels* (*mem-pool-info r p*))
     ∧ (∀ *i*. *length* (*bits* (*levels* (*mem-pool-info s p*) ! *i*))
       = *length* (*bits* (*levels* (*mem-pool-info r p*) ! *i*))))

definition *gvars-conf-stable* :: (*State* × *State*) *set*
where *gvars-conf-stable* ≡ {(*s*,*r*). *gvars-conf s r*}

definition *inv-sta-rely* :: (*State* × *State*) *set*
where *inv-sta-rely* ≡ {(*s*,*r*). *inv s* ⟶ *inv r*}

definition *inv-sta-guar* :: (*State* × *State*) *set*
where *inv-sta-guar* ≡ {(*s*,*r*). *inv s* ⟶ *inv r*}

lemma *glnochange-inv0*:
  (*a*, *b*) ∈ *lvars-nochange1-4all* ⟹ *cur a* = *cur b* ⟹
   thd-state a = *thd-state b* ⟹ *mem-pools a* = *mem-pools b* ⟹
   mem-pool-info a = *mem-pool-info b* ⟹ *inv a* ⟹ *inv b*
 apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def inv-def*)
 apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*) **apply** *auto*[*1*]
  apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*)
  apply(*rule conjI*) **apply**(*simp add*: *inv-aux-vars-def mem-block-addr-valid-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
  apply(*simp add*:*inv-bitmap-not4free-def*)
done

lemma *glnochange-inv1*:
  (*a*, *b*) ∈ *lvars-nochange-4all* ⟹ *cur a* = *cur b* ⟹
   thd-state a = *thd-state b* ⟹ *mem-pools a* = *mem-pools b* ⟹
   mem-pool-info a = *mem-pool-info b* ⟹ *inv a* ⟹ *inv b*

apply(*simp add*:*lvars-nochange-4all-def lvars-nochange-def*)
using *glnochange-inv0*
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
by *metis*

lemma *glnochange-inv*:
$\mathit{inv}\ a \Longrightarrow \forall t'.\ t' \neq t1 \longrightarrow \mathit{lvars\text{-}nochange}\ t'\ a\ b$
$\Longrightarrow \mathit{gvars\text{-}nochange}\ a\ b \Longrightarrow \mathit{lvars\text{-}nochange}\ t1\ a\ b \Longrightarrow \mathit{inv}\ b$
apply(*subgoal-tac* $(a, b) \in$ *lvars-nochange-4all*)
apply(*simp add*: *gvars-nochange-def*)
using *glnochange-inv1* **apply** *auto*
using *lv-noch-all2*[*of a b t1*] **apply** *auto*[*1*]
by(*simp add*: *lvars-nochange-rel-def*)

definition *Schedule-rely* :: (*State* $\times$ *State*) *set*
where *Schedule-rely* $\equiv \{(s,r).\ \mathit{inv}\ s \longrightarrow \mathit{inv}\ r\} \cup \mathit{Id}$

definition *Schedule-guar* :: (*State* $\times$ *State*) *set*
where *Schedule-guar* $\equiv$
(~~(*{(ºcur = Some t ⟶ (ºcur ≠ None ⟶ ªthd-state = (ºthd-state (the (ºcur) := READY))(t := RUNNING)) ∧ ªcur = Some t) ∧ (ºcur = None ⟶ ªthd-state = ºthd-state (t := RUNNING)) ∧ ªcur = Some t) ∧ (ºcur = Some t ⟶ ªthd-state = ºthd-state ∧ ºcur = ªcur)}*)~~
$\{(s,r).\ \mathit{inv}\ s \longrightarrow \mathit{inv}\ r\}$
$\cap\ \{\!|\,{}^{\mathrm{o}}\mathit{tick} = {}^{\mathrm{a}}\mathit{tick} \wedge {}^{\mathrm{o}}\mathit{mem\text{-}pools} = {}^{\mathrm{a}}\mathit{mem\text{-}pools} \wedge {}^{\mathrm{o}}\mathit{mem\text{-}pool\text{-}info} = {}^{\mathrm{a}}\mathit{mem\text{-}pool\text{-}info}\,|\!\}$
$\cap\ (\bigcap t.\ \mathit{lvars\text{-}nochange\text{-}rel}\ t)) \cup \mathit{Id}$

definition
Schedule-RGCond $t \equiv (\!|$ *PiCore-Validity.rgformula.Com* = *Schedule* t, *Pre* = $\{s.\ \mathit{inv}\ s\}$, *Rely* = *Schedule-rely*, *Guar* = *Schedule-guar*, *Post* = $\{s.\ \mathit{inv}\ s\}$ $|\!)$

definition *Tick-rely* :: (*State* $\times$ *State*) *set*
where *Tick-rely* $\equiv \{\!|\,{}^{\mathrm{o}}\mathit{tick} = {}^{\mathrm{a}}\mathit{tick}\,|\!\} \cup \mathit{Id}$

definition *Tick-guar* :: (*State* $\times$ *State*) *set*
where *Tick-guar* $\equiv (\{\!|\,{}^{\mathrm{a}}\mathit{tick} = {}^{\mathrm{o}}\mathit{tick} + 1 \wedge {}^{\mathrm{o}}\mathit{cur} = {}^{\mathrm{a}}\mathit{cur} \wedge {}^{\mathrm{o}}\mathit{thd\text{-}state} = {}^{\mathrm{a}}\mathit{thd\text{-}state}$
$\wedge\ {}^{\mathrm{o}}\mathit{mem\text{-}pools} = {}^{\mathrm{a}}\mathit{mem\text{-}pools} \wedge {}^{\mathrm{o}}\mathit{mem\text{-}pool\text{-}info} = {}^{\mathrm{a}}\mathit{mem\text{-}pool\text{-}info}\,|\!\}$
$\cap\ (\bigcap t.\ \mathit{lvars\text{-}nochange\text{-}rel}\ t)) \cup \mathit{Id}$

definition *Tick-RGCond*
where *Tick-RGCond* $\equiv (\!|$ *PiCore-Validity.rgformula.Com* = *Tick*,
Pre = $\{\!|\,\mathit{True}\,|\!\}$, *Rely* = *Tick-rely*, *Guar* = *Tick-guar*, *Post* = $\{\!|\,\mathit{True}\,|\!\}$ $|\!)$

abbreviation *alloc-blk-valid* :: *State* $\Rightarrow$ *mempool-ref* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *mem-ref* $\Rightarrow$ *bool*
where *alloc-blk-valid s p lv bnum blkaddr*
$\equiv (\mathit{blkaddr} = \mathit{buf}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p) + \mathit{bnum} * ((\mathit{max\text{-}sz}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p))\ \mathit{div}\ (4\ \hat{}\ \mathit{lv}))$
$\wedge\ \mathit{bnum} < \mathit{n\text{-}max}\ (\mathit{mem\text{-}pool\text{-}info}\ s\ p) * (4\ \hat{}\ \mathit{lv}))$

abbreviation *alloc-memblk-data-valid* :: *State* ⇒ *mempool-ref* ⇒ *Mem-block* ⇒ *bool*
where *alloc-memblk-data-valid s p mb* ≡ *alloc-blk-valid s p* (*level mb*) (*block mb*) (*data mb*)

definition *alloc-memblk-valid* :: *State* ⇒ *mempool-ref* ⇒ *nat* ⇒ *Mem-block* ⇒ *bool*
where *alloc-memblk-valid s p sz mb* ≡
    p = *pool mb* ∧ *p* ∈ *mem-pools s*
    ∧ *sz* ≤ (*max-sz* (*mem-pool-info s p*)) *div* (*4* ˆ (*level mb*)) ~~(* block size of level mb + 1 < sz ≤ block size of level mb *)~~
    ∧ (*level mb* < *n-levels* (*mem-pool-info s p*) − *1* ⟶ *sz* > (*max-sz* (*mem-pool-info s p*)) *div* (*4* ˆ (*level mb* + *1*)))
    ∧ *alloc-memblk-data-valid s p mb*

abbreviation *Mem-pool-alloc-pre* :: *Thread* ⇒ *State set*
where *Mem-pool-alloc-pre t* ≡ {*s*. *inv s* ∧ *allocating-node s t* = *None* ∧ *freeing-node s t* = *None*}

definition *Mem-pool-alloc-rely* :: *Thread* ⇒ (*State* × *State*) *set*
where *Mem-pool-alloc-rely t* ≡
  ((*lvars-nochange-rel t* ∩ *gvars-conf-stable*
   ∩ {(*s*,*r*). *inv s* ⟶ *inv r*}
   ∩ {(*s*,*r*).(*cur s* = *Some t* ⟶ *mem-pool-info s* = *mem-pool-info r*
        ∧ (∀ *t'*. *t'* ≠ *t* ⟶ *lvars-nochange t' s r*))}) ∪ *Id*)

definition *Mem-pool-alloc-guar* :: *Thread* ⇒ (*State* × *State*) *set*
where *Mem-pool-alloc-guar t* ≡
    ((*gvars-conf-stable* ∩
     {(*s*,*r*). (*cur s* ≠ *Some t* ⟶ *gvars-nochange s r* ∧ *lvars-nochange t s r*)
        ∧ (*cur s* = *Some t* ⟶ *inv s* ⟶ *inv r*)
        ∧ (∀ *t'*. *t'* ≠ *t* ⟶ *lvars-nochange t' s r*) }
     ∩ {|ᵒ*tick* = ᵃ*tick*|}) ∪ *Id*)

definition *Mem-pool-alloc-post* :: *Thread* ⇒ *mempool-ref* ⇒ *nat* ⇒ *int* ⇒ *State set*
where *Mem-pool-alloc-post t p sz timeout* ≡
  {*s*. *inv s* ∧ *allocating-node s t* = *None* ∧ *freeing-node s t* = *None*
    ∧ (*timeout* = *FOREVER* ⟶ (*ret s t* = *ESIZEERR* ∧ *mempoolalloc-ret s t* = *None*
        ∨ *ret s t* = *OK* ∧ (∃ *mblk*. *mempoolalloc-ret s t* = *Some mblk* ∧ *alloc-memblk-valid s p sz mblk*)))
    ∧ (*timeout* = *NOWAIT* ⟶ ((*ret s t* = *ENOMEM* ∨ *ret s t* = *ESIZEERR*) ∧ *mempoolalloc-ret s t* = *None*)
        ∨ (*ret s t* = *OK* ∧ (∃ *mblk*. *mempoolalloc-ret s t* = *Some*

$mblk \wedge alloc\text{-}memblk\text{-}valid\ s\ p\ sz\ mblk)))$
$\wedge\ (timeout > 0 \longrightarrow ((ret\ s\ t = ETIMEOUT \vee ret\ s\ t = ESIZEERR) \wedge mempoolalloc\text{-}ret\ s\ t = None)$
$\vee\ (ret\ s\ t = OK \wedge (\exists\, mblk.\ mempoolalloc\text{-}ret\ s\ t = Some\ mblk \wedge alloc\text{-}memblk\text{-}valid\ s\ p\ sz\ mblk))))\}$

definition *Mem-pool-alloc-RGCond*
where *Mem-pool-alloc-RGCond t p sz timeout* $\equiv$
$(\!|$ *PiCore-Validity.rgformula.Com* $=$ *Mem-pool-alloc t p sz timeout*,
Pre $=$ *Mem-pool-alloc-pre t*,
Rely $=$ *Mem-pool-alloc-rely t*,
Guar $=$ *Mem-pool-alloc-guar t*,
Post $=$ *Mem-pool-alloc-post t p sz timeout* $|\!)$

abbreviation *Mem-pool-free-pre* :: *Thread* $\Rightarrow$ *State set*
where *Mem-pool-free-pre t* $\equiv \{s.\ inv\ s \wedge allocating\text{-}node\ s\ t = None \wedge freeing\text{-}node\ s\ t = None\}$

definition *Mem-pool-free-rely* :: *Thread* $\Rightarrow$ (*State* $\times$ *State*) *set*
where *Mem-pool-free-rely t* $\equiv$
$((lvars\text{-}nochange\text{-}rel\ t \cap gvars\text{-}conf\text{-}stable$
$\cap\ \{(s,r).\ inv\ s \longrightarrow inv\ r\}$
$\cap\ \{(s,r).(cur\ s = Some\ t \longrightarrow mem\text{-}pool\text{-}info\ s = mem\text{-}pool\text{-}info\ r$
$\wedge\ (\forall\, t'.\ t' \neq t \longrightarrow lvars\text{-}nochange\ t'\ s\ r))\}) \cup Id)$

definition *Mem-pool-free-guar* :: *Thread* $\Rightarrow$ (*State* $\times$ *State*) *set*
where *Mem-pool-free-guar t* $\equiv$
$((gvars\text{-}conf\text{-}stable\ \cap$
$\{(s,r).\ (cur\ s \neq Some\ t \longrightarrow gvars\text{-}nochange\ s\ r \wedge lvars\text{-}nochange\ t\ s\ r)$
$\wedge\ (cur\ s = Some\ t \longrightarrow inv\ s \longrightarrow inv\ r)$
$\wedge\ (\forall\, t'.\ t' \neq t \longrightarrow lvars\text{-}nochange\ t'\ s\ r)\ \}$
$\cap\ \{\!|{}^{o}tick = {}^{a}tick|\!\}) \cup Id)$

definition *Mem-pool-free-post* :: *Thread* $\Rightarrow$ *State set*
where *Mem-pool-free-post t* $\equiv \{s.\ inv\ s \wedge allocating\text{-}node\ s\ t = None \wedge freeing\text{-}node\ s\ t = None\}$

definition *Mem-pool-free-RGCond*
where *Mem-pool-free-RGCond t b* $\equiv$
$(\!|$ *PiCore-Validity.rgformula.Com* $=$ *Mem-pool-free t b*,
Pre $=$ *Mem-pool-free-pre t*,
Rely $=$ *Mem-pool-free-rely t*,
Guar $=$ *Mem-pool-free-guar t*,
Post $=$ *Mem-pool-free-post t* $|\!)$

15.2 stability, subset relations of rely-guarantee conditions

lemma *stable-inv-free-rely*:

$(s,r) \in$ *Mem-pool-free-rely t* $\Longrightarrow$ *inv s* $\Longrightarrow$ *inv r*
apply (*simp add:Mem-pool-free-rely-def*)
apply(*case-tac cur s = Some t*) **apply** *simp*
apply(*subgoal-tac* $(s, r) \in$ *lvars-nochange-4all*)
apply(*simp add:lvars-nochange-4all-def lvars-nochange-def*)
apply(*simp add:inv-def*) **unfolding** *gvars-conf-stable-def gvars-conf-def*
apply(*rule conjI*) **apply**(*simp add:inv-cur-def*) **apply** *auto*[*1*] **apply** *metis*
apply(*simp add:lvars-nochange-4all-def lvars-nochange-rel-def*)
apply *auto*[*1*] **apply**(*simp add:lvars-nochange-def*)
apply *auto*
done

lemma *stable-inv-free-rely1*: *stable* $\{\!\!\{ \acute{}inv \}\!\!\}$ (*Mem-pool-free-rely t*)
using *stable-inv-free-rely* **unfolding** *stable-def* **by** *auto*

lemma *stable-inv-alloc-rely*:
$(s,r) \in$ *Mem-pool-alloc-rely t* $\Longrightarrow$ *inv s* $\Longrightarrow$ *inv r*
apply(*subgoal-tac Mem-pool-alloc-rely t = Mem-pool-free-rely t*)
using *stable-inv-free-rely* **apply** *simp*
by (*simp add:Mem-pool-alloc-rely-def Mem-pool-free-rely-def*)

lemma *stable-inv-alloc-rely1*: *stable* $\{\!\!\{ \acute{}inv \}\!\!\}$ (*Mem-pool-alloc-rely t*)
using *stable-inv-alloc-rely* **unfolding** *stable-def* **by** *auto*

lemma *stable-inv-sched-rely*:
$(s,r) \in$ *Schedule-rely* $\Longrightarrow$ *inv s* $\Longrightarrow$ *inv r*
apply (*simp add:Schedule-rely-def*) **by** *auto*

lemma *stable-inv-sched-rely1*: *stable* $\{\!\!\{ \acute{}inv \}\!\!\}$ *Schedule-rely*
using *stable-inv-sched-rely* **unfolding** *stable-def* **by** *auto*

lemma *free-guar-stb-inv*: *stable* $\{\!\!\{ \acute{}inv \}\!\!\}$ (*Mem-pool-free-guar t*)
proof –
{
fix *x*
assume *a0*: *inv x*
{
fix *y*
assume *b0*: $(x,y) \in$ *Mem-pool-free-guar t*
hence $(x,y) \in \{(s,r).$ (*cur s* $\neq$ *Some t* $\longrightarrow$ *gvars-nochange s r* $\wedge$ *lvars-nochange t s r*)
$\wedge$ (*cur s = Some t* $\longrightarrow$ *inv s* $\longrightarrow$ *inv r*)
$\wedge$ $(\forall t'.\ t' \neq t \longrightarrow$ *lvars-nochange t' s r*)$\}$
unfolding *Mem-pool-free-guar-def gvars-nochange-def lvars-nochange-def* **by** *auto*
hence (*cur x* $\neq$ *Some t* $\longrightarrow$ *gvars-nochange x y* $\wedge$ *lvars-nochange t x y*)
$\wedge$ (*cur x = Some t* $\longrightarrow$ *inv x* $\longrightarrow$ *inv y*)
$\wedge$ $(\forall t'.\ t' \neq t \longrightarrow$ *lvars-nochange t' x y*) **by** *simp*
hence *inv y*

apply(*case-tac cur x ≠ Some t*)
apply (*simp add*: *gvars-nochange-def lvars-nochange-def*) **using** *a0* **apply** *clarify*
apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **using** *inv-bitmap-freelist-def* **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-aux-vars-def*)
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*mem-block-addr-valid-def*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*mem-block-addr-valid-def*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply *metis*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def*)
using *a0* **by** *auto*
}
}
then show *?thesis* **by** (*simp add*:*stable-def*)
qed

lemma *alloc-guar-stb-inv*: *stable* {|´*inv*|} (*Mem-pool-alloc-guar t*)
apply(*subgoal-tac Mem-pool-alloc-guar t = Mem-pool-free-guar t*)
using *free-guar-stb-inv* **apply** *simp*
by (*simp add*:*Mem-pool-alloc-guar-def Mem-pool-free-guar-def*)

lemma *sched-guar-stb-inv*:
(*s*,*r*)∈*Schedule-guar* ⟹ *inv s* ⟹ *inv r*
apply(*simp add*:*Schedule-guar-def*)
apply(*erule disjE*) **by** *auto*

lemma *tick-guar-stb-inv*:
(*s*,*r*)∈*Tick-guar* ⟹ *inv s* ⟹ *inv r*
apply(*simp add*:*Tick-guar-def*) **apply**(*erule disjE*)
using *glnochange-inv0 lvars-nc-nc1*
unfolding *lvars-nochange1-4all-def lvars-nochange-rel-def* **apply** *auto*[*1*] **apply** *blast*
by *auto*

lemma *stable-equiv*: ‹*PiCore-Hoare.stable = RG-Hoare.stable*›
by (*unfold PiCore-Hoare.stable-def RG-Hoare.stable-def*) *auto*

lemma *mem-pool-alloc-pre-stb*: *stable* (*Mem-pool-alloc-pre t*) (*Mem-pool-alloc-rely t*)

apply(*rule subst*[**where** *t*={|´*inv* ∧ ´*allocating-node t* = *None* ∧ ´*freeing-node t* = *None*|}
and *s*={|´*inv*|} ∩ {|´*allocating-node t* = *None* ∧ ´*freeing-node t* = *None*|}])
apply *auto*[*1*]
apply(*rule stable-int2*) **apply** (*simp add*: *stable-inv-alloc-rely1*)
apply(*simp add*:*stable-def Mem-pool-alloc-rely-def gvars-conf-stable-def lvars-nochange-rel-def lvars-nochange-def*)
done

lemma *mp-alloc-post-stb*: *stable* (*Mem-pool-alloc-post t p sz timeout*) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-alloc-rely-def Mem-pool-alloc-post-def*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def*) **unfolding** *gvars-conf-def* **apply** *metis*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x* = *y*)
apply *simp* **apply** *clarify*
apply(*simp add*:*alloc-memblk-valid-def gvars-conf-def gvars-conf-stable-def*)
done

lemma *mem-pool-free-pre-stb*: *stable* (*Mem-pool-free-pre t*) (*Mem-pool-free-rely t*)
apply(*rule subst*[**where** *t*={|´*inv* ∧ ´*allocating-node t* = *None* ∧ ´*freeing-node t* = *None*|}
and *s*={|´*inv*|} ∩ {|´*allocating-node t* = *None* ∧ ´*freeing-node t* = *None*|}])
apply *auto*[*1*]
apply(*rule stable-int2*) **apply** (*simp add*: *stable-inv-free-rely1*)
apply(*simp add*:*stable-def Mem-pool-free-rely-def gvars-conf-stable-def lvars-nochange-rel-def lvars-nochange-def*)

done

lemma *mem-pool-free-post-stb*: *stable* (*Mem-pool-free-post t*) (*Mem-pool-free-rely t*)
using *mem-pool-free-pre-stb* **apply**(*simp add*:*Mem-pool-free-post-def*)
done

lemma *allocguar-in-allocrely*: *t1* ≠ *t2* ⟹ *Mem-pool-alloc-guar t1* ⊆ *Mem-pool-alloc-rely t2*
apply *clarify*
proof −
fix *a b*
assume *p0*: *t1* ≠ *t2*
and *p1*: (*a*, *b*) ∈ *Mem-pool-alloc-guar t1*
hence *p2*: (*a*, *b*) ∈ *gvars-conf-stable*
∧ (*cur a* ≠ *Some t1* ⟶ *gvars-nochange a b* ∧ *lvars-nochange t1 a b*)
∧ (*cur a* = *Some t1* ⟶ *inv a* ⟶ *inv b*)

$\wedge$ ($\forall$ t'. $t' \neq$ *t1* $\longrightarrow$ *lvars-nochange* t' *a b*)
$\wedge$ *tick a* = *tick b* $\vee$ *a* = *b*
unfolding *Mem-pool-alloc-guar-def* **by** *auto*

from *p0 p2* **have**
(a, b) $\in$ *lvars-nochange-rel t2* $\wedge$ (a, b) $\in$ *gvars-conf-stable*
$\wedge$ (*inv a* $\longrightarrow$ *inv b*)
$\wedge$ (*cur a* = *Some t2* $\longrightarrow$ *mem-pool-info a* = *mem-pool-info b*
$\wedge$ ($\forall$ t'. $t' \neq$ *t2* $\longrightarrow$ *lvars-nochange* t' *a b*))
$\vee$ *a* = *b*
apply *clarify*
apply(*rule conjI*) **apply**(*simp add*:*lvars-nochange-rel-def*)
apply(*rule conjI*) **apply** *simp*
apply(*rule conjI*) **apply** *clarify* **using** *glnochange-inv* **apply** *auto*[*1*]
apply *clarify*
apply(*rule conjI*) **apply**(*simp add*:*gvars-nochange-def*)
by *auto*

thus (a, b) $\in$ *Mem-pool-alloc-rely t2* **unfolding** *Mem-pool-alloc-rely-def* **by** *simp*
qed

lemma *schedguar-in-allocrely*: *Schedule-guar* $\subseteq$ *Mem-pool-alloc-rely t2*
apply *clarify*
proof −
fix *a b*
assume *p0*: (a, b) $\in$ *Schedule-guar*
hence *p1*: (*inv a* $\longrightarrow$ *inv b*) $\wedge$ *tick a* = *tick b* $\wedge$ *mem-pools a* = *mem-pools b* $\wedge$ *mem-pool-info a* = *mem-pool-info b*
$\wedge$ (a,b)$\in$($\bigcap$ t. *lvars-nochange-rel t*) $\vee$ *a* = *b*
by(*simp add*:*Schedule-guar-def*)

hence (a, b) $\in$ *lvars-nochange-rel t2* $\wedge$ (a, b) $\in$ *gvars-conf-stable*
$\wedge$ (*inv a* $\longrightarrow$ *inv b*)
$\wedge$ (*cur a* = *Some t2* $\longrightarrow$ *mem-pool-info a* = *mem-pool-info b*
$\wedge$ ($\forall$ t'. $t' \neq$ *t2* $\longrightarrow$ *lvars-nochange* t' *a b*))
$\vee$ *a* = *b*
apply *clarify*
apply(*rule conjI*) **apply**(*simp add*:*lvars-nochange-rel-def*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply** *clarify* **apply** *clarify*
by(*simp add*:*lvars-nochange-rel-def*)

thus (a, b) $\in$ *Mem-pool-alloc-rely t2* **by**(*simp add*:*Mem-pool-alloc-rely-def*)
qed

lemma *schedguar-in-tickrely*: *Schedule-guar* $\subseteq$ *Tick-rely*
apply(*simp add*:*Schedule-guar-def Tick-rely-def*)
by *auto*

lemma *allocguar-in-tickrely*: *Mem-pool-alloc-guar t* $\subseteq$ *Tick-rely*
  apply(*simp add*:*Mem-pool-alloc-guar-def Tick-rely-def*)
  by *auto*

lemma *tickguar-in-allocrely*: *Tick-guar* $\subseteq$ *Mem-pool-alloc-rely t*
  apply *clarify*
  proof −
  fix *a b*
  assume *p0*: $(a,\ b) \in$ *Tick-guar*
  hence *p1*: *tick b* = *tick a* + *1* $\wedge$ *cur a* = *cur b* $\wedge$ *thd-state a* = *thd-state b*
    $\wedge$ *mem-pools a* = *mem-pools b* $\wedge$ *mem-pool-info a* = *mem-pool-info b*
    $\wedge$ $(a,b)\in(\bigcap t.$ *lvars-nochange-rel t*$)$ $\vee$ $a = b$
    by(*simp add*:*Tick-guar-def*)

  hence $(a,\ b) \in$ *lvars-nochange-rel t* $\wedge$ $(a,\ b) \in$ *gvars-conf-stable*
    $\wedge$ (*inv a* $\longrightarrow$ *inv b*)
    $\wedge$ (*cur a* = *Some t* $\longrightarrow$ *mem-pool-info a* = *mem-pool-info b*
      $\wedge$ $(\forall t'.\ t' \neq t \longrightarrow$ *lvars-nochange* t' *a b*))
    $\vee$ $a = b$
      apply *clarify*
      apply(*rule conjI*) **apply**(*simp add*:*lvars-nochange-rel-def*)
      apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
    apply(*rule conjI*) **using** *glnochange-inv0 lvars-nc-nc1* **unfolding** *lvars-nochange-rel-def lvars-nochange1-4all-def*
      apply *auto*[*1*] **apply** *blast*
      by *auto*

  thus $(a,\ b) \in$ *Mem-pool-alloc-rely t* **by**(*simp add*:*Mem-pool-alloc-rely-def*)
qed

lemma *allocguar-in-schedrely*: *Mem-pool-alloc-guar t* $\subseteq$ *Schedule-rely*
  apply(*simp add*:*Mem-pool-alloc-guar-def Schedule-rely-def*)
  apply *clarify*
  apply(*case-tac cur a* = *Some t*)
    apply *simp*
    apply *clarify*
    using *glnochange-inv* **by** *auto*

lemma *tickguar-in-schedrely*: *Tick-guar* $\subseteq$ *Schedule-rely*
  apply *clarify*
  proof −
    fix *a b*
    assume *p0*: $(a,\ b) \in$ *Tick-guar*
    thus $(a,\ b) \in$ *Schedule-rely*
      apply(*simp add*:*Tick-guar-def Schedule-rely-def*) **apply** *auto*
    using *glnochange-inv1* **by**(*simp add*:*lvars-nochange-4all-def lvars-nochange-rel-def*)
  qed

end

theory *func-cor-lemma*
imports *rg-cond*
begin

declare [[*smt-timeout* = *300*]]

16 some lemmas for functional correctness by rely guarantee proof

lemma *inv-mempool-info-maxsz-mod4*:
inv-mempool-info s $\Longrightarrow$ $\forall$ *p*$\in$*mem-pools s*. *max-sz* (*mem-pool-info s p*) *mod 4* = *0*
unfolding *inv-mempool-info-def*
by (*metis mod-mult-left-eq mod-mult-self1-is-0 mod-mult-self2-is-0 mult-0*)

lemma *inv-mempool-info-maxsz-align4*:
inv-mempool-info s $\Longrightarrow$ $\forall$ *p*$\in$*mem-pools s*. *ALIGN4* (*max-sz* (*mem-pool-info s p*)) = *max-sz* (*mem-pool-info s p*)
using *inv-mempool-info-maxsz-mod4 align40* **by** *simp*

lemma *inv-maxsz-align4*:
inv s $\Longrightarrow$ $\forall$ *p*$\in$*mem-pools s*. *ALIGN4* (*max-sz* (*mem-pool-info s p*)) = *max-sz* (*mem-pool-info s p*)
unfolding *inv-def* **using** *inv-mempool-info-maxsz-align4* **by** *simp*

lemma *lsizes-mod4*:
assumes *p0*: *inv V*
and *p1*: $\forall$ *ii*<*length ls*. *ls* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info V p*)) *div 4* ^ *ii*
and *p2*: *length ls* $\le$ *length* (*levels* (*mem-pool-info V p*))
and *p3*: *p* $\in$ *mem-pools V*
shows $\forall$ *ii*<*length ls*. (*ls* ! *ii*) *mod 4* = *0*
proof −
{
fix *ii*
assume *a0*: *ii* < *length ls*
from *p0 p3* **have** $\exists$ *n*>*0*. *max-sz* (*mem-pool-info V p*) = (*4* ∗ *n*) ∗ (*4* ^ (*length* (*levels* (*mem-pool-info V p*))))
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*) **by** *auto*
then obtain *n* **where** *n* > *0* ∧ *max-sz* (*mem-pool-info V p*) = (*4* ∗ *n*) ∗ (*4* ^ (*length* (*levels* (*mem-pool-info V p*)))) **by** *auto*
hence *a1*: *n* > *0* ∧ *max-sz* (*mem-pool-info V p*) = *n* ∗ (*4* ^ (*length* (*levels*

(*mem-pool-info V p*)) + *1*)) **by** *auto*

hence *ALIGN4* (*max-sz* (*mem-pool-info V p*)) = *max-sz* (*mem-pool-info V p*)
using *align40* **by** *auto*
with *a0 p1* **have** *a2*: *ls* ! *ii* = *max-sz* (*mem-pool-info V p*) *div* *4* ˆ *ii* **by** *auto*
with *a1* **have** *ls* ! *ii* = *n* ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1*)) *div* *4* ˆ *ii* **by** *simp*
moreover
from *a0 p2* **have** (*4*::*nat*) ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1*) *mod* *4* ˆ *ii* = *0*
using *pow-mod-0*[*of ii length* (*levels* (*mem-pool-info V p*)) + *1 4*] **by** *auto*
ultimately have *a3*: *ls* ! *ii* = *n* ∗ ((*4* ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1*)) *div* *4* ˆ *ii*)
using *m-mod-div* **by** *auto*

from *a0 p2* **have** *4* ≠ *NULL* ∧ *ii* ≤ *length* (*levels* (*mem-pool-info V p*)) + *1*
by *linarith*
hence ((*4*::*nat*) ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1*)) *div* *4* ˆ *ii*
= *4* ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1* − *ii*)
using *div2-eq-minus*[*of 4 ii* (*length* (*levels* (*mem-pool-info V p*)) + *1*)] **by** *simp*
hence *n* ∗ (((*4*::*nat*) ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1*)) *div* *4* ˆ *ii*)
= *n* ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1* − *ii*)) **by** *auto*
with *a3* **have** *ls* ! *ii* = *n* ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V p*)) + *1* − *ii*)) **by** *auto*
with *a0 p2* **have** *ls* ! *ii mod 4* = *0* **by** *auto*
}
then show *?thesis* **by** *auto*
qed

lemma *gvars-conf-stb-inv-mpinf*: (*x*,*y*)∈ *gvars-conf-stable* ⟹ *inv-mempool-info y* ⟹ *inv-mempool-info x*
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def inv-mempool-info-def*)
apply *clarify*
apply(*rule conjI*) **apply** *metis* **apply**(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis* **apply**(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis* **apply** *metis*
done

lemma *ref-byblkn-self*:
R ≥ *buf* (*mem-pool-info Va p*) ⟹
(*R* − *buf* (*mem-pool-info Va p*)) *mod sz* = *0* ⟹
buf (*mem-pool-info Va p*) + *block-num* (*mem-pool-info Va p*) *R sz* ∗ *sz* = *R*
apply(*simp add*:*block-num-def*)
apply(*rule subst*[**where** *t*=(*R* − *buf* (*mem-pool-info Va p*)) *div sz* ∗ *sz* **and** *s*=*R* − *buf* (*mem-pool-info Va p*)])
by *auto*

lemma *partnerbits-udptn-notbit-partbits*:
$\forall jj<length\ lst.\ \neg\ (let\ a = (jj\ div\ 4) * 4\ in$
$lst!a = TAG \wedge lst!(a+1) = TAG \wedge lst!(a+2) = TAG \wedge lst!(a+3) = TAG) \Longrightarrow$
$TAG \neq TAG2 \Longrightarrow lst' = list\text{-}updates\text{-}n\ lst\ ii\ m\ TAG2 \Longrightarrow$
$\forall jj<length\ lst'.\ \neg\ (let\ a = (jj\ div\ 4) * 4\ in$
$lst'!a = TAG \wedge lst'!(a+1) = TAG \wedge lst'!(a+2) = TAG \wedge lst'!(a+3) = TAG)$
apply(*unfold Let-def*) **apply**(*rule allI*, *rule impI*)
apply(*case-tac* $lst'\ !\ (jj\ div\ 4 * 4) = TAG \wedge lst'\ !\ (jj\ div\ 4 * 4 + 1) = TAG$
$\wedge\ lst'\ !\ (jj\ div\ 4 * 4 + 2) = TAG \wedge lst'\ !\ (jj\ div\ 4 * 4 + 3) = TAG$)
apply(*subgoal-tac length lst = length lst'*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac* $\neg\ (lst\ !\ (jj\ div\ 4 * 4) = TAG \wedge lst\ !\ (jj\ div\ 4 * 4 + 1) = TAG$
$\wedge\ lst\ !\ (jj\ div\ 4 * 4 + 2) = TAG \wedge lst\ !\ (jj\ div\ 4 * 4 + 3) = TAG)$)
prefer *2* **apply** *presburger*
apply(*case-tac* $jj\ div\ 4 * 4 + 3 < ii$) **using** *list-updates-n-neq*
apply (*smt One-nat-def add.right-neutral add-Suc-right add-lessD1 numeral-Bit1 numeral-One one-add-one plus-nat.simps*(*2*))
apply(*case-tac* $jj\ div\ 4 * 4 \geq ii + m$) **using** *list-updates-n-neq* **apply** (*smt le-add1 le-trans*)
using *list-updates-eq* **apply** (*smt One-nat-def Suc-leI add.right-neutral add-Suc-right add-lessD1*
div-mult-mod-eq le-less-trans list-updates-n-beyond list-updates-n-eq list-updates-n-neq not-le numeral-Bit1 numeral-One one-add-one)
by *assumption*

end

theory *func-cor-other*
imports *func-cor-lemma*
begin

17 Functional correctness of Schedule

lemma *Schedule-satRG-h1*:
$\Gamma \vdash_I Some\ (IF\ \exists y.\ ´cur = Some\ y\ THEN\ ´thd\text{-}state := ´thd\text{-}state(the\ ´cur := READY);;\ Basic\ (cur\text{-}update\ Map.empty)\ FI;;$
$Basic\ (cur\text{-}update\ (\lambda\text{-}.\ Some\ t));;$
$´thd\text{-}state := ´thd\text{-}state$
$(t := RUNNING))\ sat_p\ [\{\!|´inv|\!\} \cap \{\!|´thd\text{-}state\ t = READY|\!\} \cap$
$\{V\}, \{(s, t).\ s = t\}, UNIV, \{\!|´(Pair\ V) \in Schedule\text{-}guar|\!\} \cap \{\!|´inv|\!\}]$
apply(*case-tac* $\{\!|´inv|\!\} \cap \{\!|´thd\text{-}state\ t = READY|\!\} \cap \{V\} = \{\}$)
using *Emptyprecond* **apply** *auto*[*1*]

apply *simp*
apply(*case-tac* $\exists y.\ cur\ V = Some\ y$)

apply(*rule Seq*[**where** *mid* = $\{V(\!|thd\text{-}state := (thd\text{-}state\ V)(the\ (cur\ V) := READY)|\!)(\!|cur := None|\!)(\!|cur := Some\ t|\!)\}$])
apply(*rule Seq*[**where** *mid* = $\{V(\!|thd\text{-}state := (thd\text{-}state\ V)(the\ (cur\ V) := READY)|\!)(\!|cur := None|\!)\}$])
apply(*rule Cond*)
apply(*simp add*:*stable-def*)
apply(*rule Seq*[**where** *mid* = $\{V(\!|thd\text{-}state := (thd\text{-}state\ V)(the\ (cur\ V) := READY)|\!)\}$])
apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*simp add*:*Skip-def*) **apply**(*rule Basic*) **apply**(*simp add*:*stable-def*)+

apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+

apply(*rule Basic*)
apply(*simp add*:*Schedule-guar-def*)
apply(*subgoal-tac* $inv\ (V(\!|cur := Some\ t,\ thd\text{-}state := (thd\text{-}state\ V)(the\ (cur\ V) := READY,\ t := RUNNING)|\!)) \wedge$
$(\forall x.\ (V,\ V(\!|cur := Some\ t,\ thd\text{-}state := (thd\text{-}state\ V)(the\ (cur\ V) := READY,\ t := RUNNING)|\!)) \in lvars\text{-}nochange\text{-}rel\ x)$)
apply *simp*
apply(*rule conjI*) **apply**(*simp add*:*inv-def*) **apply** *clarify*
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*) **apply** *force*
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def inv-cur-def*)
apply (*metis Thread-State-Type.distinct*(*3*) *Thread-State-Type.distinct*(*6*))
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-aux-vars-def mem-block-addr-valid-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def*)
apply *auto*[*1*] **using** *lvars-nochange-rel-def lvars-nochange-def* **apply** *simp*
apply(*simp add*: *stable-def*)+

apply(*rule Seq*[**where** *mid* = $\{V(\!|cur := Some\ t|\!)\}$])
apply(*rule Seq*[**where** *mid* = $\{V\}$])
apply(*rule Cond*)
apply(*simp add*:*stable-def*)
apply(*rule Seq*[**where** *mid* = $\{\}$])

apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*simp add*:*Skip-def*) **apply**(*rule Basic*) **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply(*simp add*:*Schedule-guar-def*)
apply(*subgoal-tac inv* (V⦇*cur* := *Some t*, *thd-state* := (*thd-state V*)(t := *RUNNING*)⦈) ∧
(∀ x. (V, V⦇*cur* := *Some t*, *thd-state* := (*thd-state V*)(t := *RUNNING*)⦈) ∈ *lvars-nochange-rel x*))
apply *simp*
apply(*rule conjI*) **apply**(*simp add*:*inv-def*) **apply** *clarify*
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-aux-vars-def mem-block-addr-valid-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def*)

apply *auto*[*1*] **using** *lvars-nochange-rel-def lvars-nochange-def* **apply** *simp*
apply(*simp add*:*stable-def*)+
done

lemma *Schedule-satRG*: *Evt-sat-RG* Γ (*Schedule-RGCond t*)
apply(*simp add*:*Evt-sat-RG-def*)
apply (*simp add*: *Schedule-def Schedule-RGCond-def*)
apply(*rule Evt-Basic*)
apply(*simp add*:*body-def guard-def*)
apply(*rule Await*)
using *stable-inv-sched-rely1* **apply** *simp* **using** *stable-inv-sched-rely1* **apply** *simp*
using *Schedule-satRG-h1* **apply** *simp*

apply (*simp add*: *stable-equiv stable-inv-sched-rely1*)

by(*simp add*: *Schedule-guar-def*)

18 Functional correctness of Tick

lemma *Tick-satRG*: *Evt-sat-RG* Γ *Tick-RGCond*

apply(*simp add*:*Evt-sat-RG-def*)
apply (*simp add*: *Tick-def Tick-RGCond-def Tick-rely-def Tick-guar-def*)
apply(*rule Evt-Basic*)
apply(*simp add*:*body-def guard-def*)
apply(*rule Basic*)
apply *simp*
using *lvars-nochange-rel-def lvars-nochange-def* **apply** *simp* **apply** *auto*[*1*]
apply(*simp add*:*stable-def*)+
apply(*simp add*: *PiCore-Hoare.stable-def*) **apply** *auto*[*1*]
done

end

19 Lemmas of Picore-SIMP

theory *picore-SIMP-lemma*
imports *picore-SIMP-Syntax picore-SIMP*

begin

lemma *id-belong*[*simp*]: $Id \subseteq \{\!|{}^{a}x = {}^{o}x|\!\}$
by (*simp add*: *Collect-mono Id-fstsnd-eq*)

lemma *allpre-eq-pre*: $(\forall v \in U.\ \vdash_I P\ sat_p\ [\{v\},\ rely,\ guar,\ post]) \longleftrightarrow \vdash_I P\ sat_p\ [U,\ rely,\ guar,\ post]$
apply *auto* **using** *Allprecond* **apply** *blast*
using *Conseq*[*of - - rely rely guar guar post post P*] **by** *auto*

lemma *sat-pre-imp-allinpre*: $\vdash_I P\ sat_p\ [U,\ rely,\ guar,\ post] \Longrightarrow v \in U \Longrightarrow \vdash_I P\ sat_p\ [\{v\},\ rely,\ guar,\ post]$
using *Conseq*[*of - - rely rely guar guar post post P*] **by** *auto*

lemma *stable-int-col2*: $stable\ \{\!|s|\!\}\ r \Longrightarrow stable\ \{\!|t|\!\}\ r \Longrightarrow stable\ \{\!|s \wedge t|\!\}\ r$
by *auto*

lemma *stable-int-col3*: $stable\ \{\!|k|\!\}\ r \Longrightarrow stable\ \{\!|s|\!\}\ r \Longrightarrow stable\ \{\!|t|\!\}\ r \Longrightarrow stable\ \{\!|k \wedge s \wedge t|\!\}\ r$
by *auto*

lemma *stable-int-col4*: $stable\ \{\!|m|\!\}\ r \Longrightarrow stable\ \{\!|k|\!\}\ r \Longrightarrow stable\ \{\!|s|\!\}\ r \Longrightarrow stable\ \{\!|t|\!\}\ r \Longrightarrow stable\ \{\!|m \wedge k \wedge s \wedge t|\!\}\ r$
by *auto*

lemma *stable-int-col5*: $stable\ \{\!|q|\!\}\ r \Longrightarrow stable\ \{\!|m|\!\}\ r \Longrightarrow stable\ \{\!|k|\!\}\ r \Longrightarrow stable\ \{\!|s|\!\}\ r \Longrightarrow stable\ \{\!|t|\!\}\ r \Longrightarrow stable\ \{\!|q \wedge m \wedge k \wedge s \wedge t|\!\}\ r$
by *auto*

lemma *stable-un2*: $stable\ s\ r \Longrightarrow stable\ t\ r \Longrightarrow stable\ (s \cup t)\ r$

by (*simp add*: *stable-def*)

lemma *stable-un-R*: *stable s r* $\Longrightarrow$ *stable s r'* $\Longrightarrow$ *stable s* $(r \cup r')$
by (*meson UnE stable-def*)

lemma *stable-un-S*: $\forall t$. *stable s* $(P\ t) \Longrightarrow$ *stable s* $(\bigcup t.\ P\ t)$
apply(*simp add*:*stable-def*) **by** *auto*

lemma *stable-un-S2*: $\forall t\ x$. *stable s* $(P\ t\ x) \Longrightarrow$ *stable s* $(\bigcup t\ x.\ P\ t\ x)$
apply(*simp add*:*stable-def*) **by** *auto*

lemma *pairv-IntI*:
$y \in \{\!|\,\acute{}(Pair\ V) \in A|\!\} \Longrightarrow y \in \{\!|\,\acute{}(Pair\ V) \in B|\!\} \Longrightarrow y \in \{\!|\,\acute{}(Pair\ V) \in A \cap B|\!\}$
by *auto*

lemma *pairv-rId*:
$y \in \{\!|\,\acute{}(Pair\ V) \in A|\!\} \Longrightarrow y \in \{\!|\,\acute{}(Pair\ V) \in A \cup Id|\!\}$
by *auto*

end

theory *func-cor-mempoolfree*
imports *func-cor-lemma ../../adapter-SIMP/picore-SIMP-lemma*
begin

20 Functional correctness of $k\_mem\_pool\_free$

20.1 intermediate conditions and their stable to rely cond

abbreviation *mp-free-precond1-ext t b* $\equiv$
{|*pool b* $\in$ ´*mem-pools* $\wedge$ *level b* $<$ *length* (*levels* (´*mem-pool-info* (*pool b*)))
$\wedge$ *block b* $<$ *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*))!(*level b*)))
$\wedge$ *data b* = *block-ptr* (´*mem-pool-info* (*pool b*)) ((*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ˆ (*level b*))) (*block b*)|}

abbreviation *mp-free-precond1 t b* $\equiv$
Mem-pool-free-pre t $\cap$ *mp-free-precond1-ext t b*

lemma *mp-free-precond1-ext-stb*: *stable* (*mp-free-precond1-ext t b*) (*Mem-pool-free-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*rule conjI*) **apply**(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def*) **unfolding** *gvars-conf-def* **apply** *metis*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def*) **unfolding** *gvars-conf-def* **apply** *metis*

apply(*simp add*:*block-ptr-def*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
done

lemma *mp-free-precond1-stb* : *stable* (*mp-free-precond1 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
apply(*simp add*:*mem-pool-free-pre-stb*)
apply(*simp add*:*mp-free-precond1-ext-stb*)
done

abbreviation *mp-free-precond1-0 t b* ≡
{*s. inv s* ∧ *allocating-node s t* = *None*} ∩ *mp-free-precond1-ext t b*

lemma *mp-free-precond1-0-stb*: *stable* (*mp-free-precond1-0 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
apply(*rule subst*[**where** *t*=⦃´*inv* ∧ ´*allocating-node t* = *None*⦄
and *s*=⦃´*inv*⦄ ∩ ⦃´*allocating-node t* = *None*⦄])
apply *force*
apply(*rule stable-int2*)
apply(*simp add*:*stable-inv-free-rely1*)
apply(*simp add*:*stable-def Mem-pool-free-rely-def*)
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*)
apply(*simp add*:*mp-free-precond1-ext-stb*)
done

abbreviation *mp-free-precond2-ext t b* ≡ ⦃´*freeing-node t* = *Some b*⦄

abbreviation *mp-free-precond2 t b* ≡
mp-free-precond1-0 t b ∩ *mp-free-precond2-ext t b*

lemma *mp-free-precond2-ext-stb*: *stable* (*mp-free-precond2-ext t b*) (*Mem-pool-free-rely t*)

apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def*)
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-free-precond2-stb*: *stable* (*mp-free-precond2 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)

apply(*simp add*:*mp-free-precond1-0-stb*)
apply(*simp add*:*mp-free-precond2-ext-stb*)
done

abbreviation *mp-free-precond3-ext t b* ≡ {|´*need-resched t* = *False*|}

abbreviation *mp-free-precond3 t b* ≡ (*mp-free-precond2 t b*) ∩ *mp-free-precond3-ext t b*

lemma *mp-free-precond3-ext-stb* : *stable* (*mp-free-precond3-ext t b*) (*Mem-pool-free-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *auto*

lemma *mp-free-precond3-stb* : *stable* (*mp-free-precond3 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond2-stb* **apply** *simp*
using *mp-free-precond3-ext-stb* **apply** *simp*
done

abbreviation *mp-free-precond4-ext t b* ≡ {|´*lsizes t* = [*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))]|}

abbreviation *mp-free-precond4 t b* ≡
mp-free-precond3 t b ∩ *mp-free-precond4-ext t b*

lemma *mp-free-precond4-ext-stb*:
stable (*mp-free-precond4-ext t b*) (*Mem-pool-free-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def ALIGN4-def*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*case-tac x* = *y*) **apply** *simp*
apply *clarify* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
done

lemma *mp-free-precond4-stb* : *stable* (*mp-free-precond4 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond3-stb* **apply** *simp*
using *mp-free-precond4-ext-stb* **apply** *blast*
done

abbreviation *mp-free-precond4-0-ext t b* ≡
{|(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ^ *ii*))
∧ *length* (´*lsizes t*) > *0*|}

abbreviation *mp-free-precond4-0 t b* ≡ *mp-free-precond3 t b* ∩ *mp-free-precond4-0-ext t b*

lemma *mp-free-precond4-0-ext-stb*:
stable (*mp-free-precond4-0-ext t b*) (*Mem-pool-free-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def ALIGN4-def*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*case-tac* $x = y$) **apply** *simp*
apply *clarify* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
done

lemma *mp-free-precond4-0-stb* : *stable* (*mp-free-precond4-0 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond3-stb* **apply** *simp*
using *mp-free-precond4-0-ext-stb* **apply** *blast*
done

abbreviation *mp-free-precond4-1 t b* ≡
mp-free-precond4-0 t b ∩ ⦃*length* (´*lsizes t*) = ´*i t*⦄

lemma *mp-free-precond4-1-stb* : *stable* (*mp-free-precond4-1 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond4-0-stb* **apply** *simp*
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)+
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
done

abbreviation *mp-free-precond4-2 t b* ≡
mp-free-precond4-1 t b ∩ ⦃´*i t* ≤ *level b* ⦄

lemma *mp-free-precond4-2-stb* : *stable* (*mp-free-precond4-2 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond4-1-stb* **apply** *simp*
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)+
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *smt*

abbreviation *mp-free-precond4-3 t b* ≡
mp-free-precond4-0 t b ∩ (⦃´*i t* ≤ *level b* ⦄ ∩ ⦃*length* (´*lsizes t*) = ´*i t* + *1*⦄)

lemma *mp-free-precond4-3-stb* : *stable* (*mp-free-precond4-3 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond4-0-stb* **apply** *simp*

apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)+
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *smt*

abbreviation *mp-free-precond5-ext t b* ≡
⦃(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ^ *ii*))
∧ *length* (´*lsizes t*) > *level b*⦄

abbreviation *mp-free-precond5 t b* ≡ *mp-free-precond3 t b* ∩ *mp-free-precond5-ext t b*
term *mp-free-precond5 t b*

lemma *mp-free-precond5-ext-stb*:
stable (*mp-free-precond5-ext t b*) (*Mem-pool-free-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def ALIGN4-def*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*case-tac x* = *y*) **apply** *simp*
apply *clarify* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
done

lemma *mp-free-precond5-stb* : *stable* (*mp-free-precond5 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond3-stb* **apply** *simp*
using *mp-free-precond5-ext-stb* **apply** *blast*
done

abbreviation *mp-free-precond6 t b* ≡
mp-free-precond5 t b ∩ ⦃´*free-block-r t* = *True*⦄

lemma *mp-free-precond6-stb* : *stable* (*mp-free-precond6 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond5-stb* **apply** *simp*
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)+
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *auto*

abbreviation *mp-free-precond7 t b* ≡
mp-free-precond6 t b ∩ ⦃´*bn t* = *block b*⦄

lemma *mp-free-precond7-stb* : *stable* (*mp-free-precond7 t b*) (*Mem-pool-free-rely t*)
apply(*rule stable-int2*)
using *mp-free-precond6-stb* **apply** *simp*

apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)+
apply(*rule impI*)
apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *smt*

abbreviation *mp-free-precond8 t b* ≡
mp-free-precond1-0 t b ∩ ⦃*level b* < *length* (´*lsizes t*)
∧ (∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ^ *ii*))
∧ ´*bn t* < *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*))!(´*lvl t*)))
∧ ´*bn t* = (*block b*) *div* (*4* ^ (*level b* − ´*lvl t*))
∧ ´*lvl t* ≤ *level b*
∧ (´*free-block-r t* ⟶
(∃ *blk*. ´*freeing-node t* = *Some blk* ∧ *pool blk* = *pool b* ∧ *level blk* = ´*lvl t* ∧ *block blk* = ´*bn t*)
∧ ´*alloc-memblk-data-valid* (*pool b*) (*the* (´*freeing-node t*)))
∧ (¬ ´*free-block-r t* ⟶ ´*freeing-node t* = *None*)
~~(* ((*if* ´*freeing-node t* ≠ *None then* ´*lvl t* + *1 else 0*) > *0* ⟷~~
~~´*free-block-r t*) *) (* *this cond is implied by upper conds* *)~~ ⦄

abbreviation *mp-free-precond8-inv t b* α ≡
mp-free-precond8 t b ∩ ⦃ α = (*if* ´*freeing-node t* ≠ *None then* ´*lvl t* + *1 else 0*)
⦄

lemma *inv-*α*gt0-imp-looppre*:
mp-free-precond8-inv t b α ∩ ⦃α > *0*⦄ ⊆ *mp-free-precond8 t b* ∩ ⦃´*free-block-r t*⦄
by *auto*

lemma *looppre-imp-exist-*α*gt0*:
x ∈ *mp-free-precond8 t b* ∩ ⦃´*free-block-r t*⦄ ⟹ ∃ α. *x* ∈ *mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄
by *clarsimp*

lemma *x* ∈ *mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄ ⟹ *x* ∈ *mp-free-precond8 t b* ∩ ⦃´*free-block-r t*⦄
using *inv-*α*gt0-imp-looppre*[*of t b* α]
subsetI[*of mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄
mp-free-precond8 t b ∩ ⦃´*free-block-r t*⦄]
by *blast*

lemma *loopbody-sat-invterm-imp-inv-post*:
Γ $\vdash_I$ *P* sat_p [*mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄, *rely*, *guar*, *mp-free-precond8-inv t b* (α − *1*)]
⟹ Γ $\vdash_I$ *P* sat_p [*mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄, *rely*, *guar*,*mp-free-precond8 t b*]

using *Conseq* [*of mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$ *mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$
rely rely guar guar mp-free-precond8-inv t b $(\alpha - 1)$
mp-free-precond8 t b P] **by** *blast*

lemma *stm8-inv-imp-prepost*:
$(\forall \alpha.\ \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$, *rely*, *guar*, *mp-free-precond8-inv t b* $(\alpha - 1)$])
$\Longrightarrow \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8 t b* $\cap$ {´*free-block-r t*}, *rely*, *guar*,*mp-free-precond8 t b*]

apply(*rule subst*[**where** *s*=$\forall v.\ v\in$*mp-free-precond8 t b* $\cap$ {´*free-block-r t*} $\longrightarrow$
$\Gamma \vdash_I P\ sat_p$ [$\{v\}$, *rely*, *guar*,*mp-free-precond8 t b*] **and**
t=$\Gamma \vdash_I P\ sat_p$ [*mp-free-precond8 t b* $\cap$ {´*free-block-r t*}, *rely*, *guar*,*mp-free-precond8 t b*]])
using *allpre-eq-pre*[*of mp-free-precond8 t b* $\cap$ {´*free-block-r t*}
P rely guar mp-free-precond8 t b] **apply** *blast*

apply(*rule allI*) **apply**(*rule impI*)
apply(*subgoal-tac* $\exists \alpha.\ v \in$ *mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$)
prefer *2* **using** *looppre-imp-exist-*α*gt0* **apply** *blast*

apply(*erule exE*)

using *sat-pre-imp-allinpre*[*of P - rely guar mp-free-precond8 t b*]
loopbody-sat-invterm-imp-inv-post **apply** *blast*
done

lemma *loopbody-sat-invterm-imp-inv-post2*:
$\exists \beta<\alpha.\ \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$, *rely*, *guar*, *mp-free-precond8-inv t b* β]
$\Longrightarrow \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$, *rely*, *guar*,*mp-free-precond8 t b*]
using *Conseq* [*of mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$ *mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$
rely rely guar guar mp-free-precond8-inv t b -
mp-free-precond8 t b P] **by** *blast*

lemma *stm8-inv-imp-prepost2*:
$(\forall \alpha.\ \exists \beta<\alpha.\ \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\alpha > 0\}$, *rely*, *guar*, *mp-free-precond8-inv t b* β])
$\Longrightarrow \Gamma \vdash_I P\ sat_p$ [*mp-free-precond8 t b* $\cap$ {´*free-block-r t*}, *rely*, *guar*,*mp-free-precond8 t b*]

apply(*rule subst*[**where** *s*=$\forall v.\ v\in$*mp-free-precond8 t b* $\cap$ {´*free-block-r t*} $\longrightarrow$
$\Gamma \vdash_I P\ sat_p$ [$\{v\}$, *rely*, *guar*,*mp-free-precond8 t b*] **and**
t=$\Gamma \vdash_I P\ sat_p$ [*mp-free-precond8 t b* $\cap$ {´*free-block-r t*}, *rely*, *guar*,*mp-free-precond8*

t b]])
  using *allpre-eq-pre*[*of mp-free-precond8 t b* ∩ ⦃´*free-block-r t*⦄
                          P rely guar mp-free-precond8 t b] **apply** *blast*

apply(*rule allI*) **apply**(*rule impI*)
apply(*subgoal-tac* ∃α. *v* ∈ *mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄)
  prefer *2* **using** *looppre-imp-exist-αgt0* **apply** *blast*

apply(*erule exE*)

 using *sat-pre-imp-allinpre*[*of P - rely guar mp-free-precond8 t b*]
    loopbody-sat-invterm-imp-inv-post **apply** *blast*
done

lemma *stm8-loopinv0*: *mp-free-precond8-inv t b 0* ⊆ ⦃¬ ´*free-block-r t*⦄
by *auto*

lemma *stm8-loopinv-α*: α > *0* ⟹ *mp-free-precond8-inv t b* α ⊆ ⦃´*free-block-r t*⦄
by *auto*

lemma *inv-αeq0-eq-looppre*:
mp-free-precond8-inv t b 0 = *mp-free-precond8 t b* ∩ ⦃¬ ´*free-block-r t*⦄
by *auto*

term *mp-free-precond8 t b*

lemma *alloc-memblk-data-valid-stb-free*:
  alloc-memblk-data-valid x (*pool b*) (*the* (*freeing-node x t*)) ⟹
     (*x*, *y*) ∈ *lvars-nochange-rel t* ⟹
     (*x*, *y*) ∈ *gvars-conf-stable* ⟹
    alloc-memblk-data-valid y (*pool b*) (*the* (*freeing-node y t*))
  apply(*subgoal-tac blk x t* = *blk y t*)
    prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply(*subgoal-tac buf* (*mem-pool-info x* (*pool b*)) = *buf* (*mem-pool-info y* (*pool b*)))
    prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
  apply(*subgoal-tac lsizes x t* = *lsizes y t*)
    prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply(*subgoal-tac free-l x t* = *free-l y t*)
    prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply(*subgoal-tac max-sz* (*mem-pool-info x* (*pool b*)) = *max-sz* (*mem-pool-info y* (*pool b*)))
    prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
  apply(*subgoal-tac freeing-node x t* = *freeing-node y t*)
    prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply (*simp add*: *gvars-conf-def gvars-conf-stable-def*)
done

lemma *mp-free-precond8-stb* : *stable* (*mp-free-precond8 t b*) (*Mem-pool-free-rely t*)

apply(*rule stable-int2*) **apply**(*rule stable-int2*)

apply(*simp add*:*stable-def*)
apply *clarify*
apply(*rule conjI*)
using *stable-inv-free-rely* **apply** *blast*
apply(*simp add*: *Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*

apply(*simp add*:*stable-def*)
apply *clarify*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
apply(*simp add*: *block-ptr-def ALIGN4-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*Mem-pool-free-rely-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*

apply(*simp add*: *Mem-pool-free-rely-def stable-def*)
apply *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*rule conjI*) **apply**(*simp add*: *ALIGN4-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply**(*simp add*: *ALIGN4-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*) **apply** *metis*
apply(*simp add*: *ALIGN4-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)

apply *clarify* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply *clarify*

done

lemma *mp-free-precond8-inv-stb* : *stable* (*mp-free-precond8-inv t b* α) (*Mem-pool-free-rely*

t)
  apply(*rule stable-int2*)
  using *mp-free-precond8-stb* **apply** *fast*
  apply(*unfold stable-def*) **apply** *clarify*

  apply(*subgoal-tac lvl x t = lvl y t*) **prefer** *2*
   apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
  apply(*subgoal-tac freeing-node x t = freeing-node y t*) **prefer** *2*
   apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
by *simp*

lemma *mp-free-precond8-inv-presv-rely*:
s∈*mp-free-precond8-inv t b* $\alpha \Longrightarrow$ (*s*,*r*)∈*Mem-pool-free-rely t* $\Longrightarrow \exists \beta \leq \alpha$. *r*∈*mp-free-precond8-inv t b* β
apply(*rule exI*[**where** *x*=α])
apply(*rule conjI*) **apply** *fast*
using *mp-free-precond8-inv-stb*[*of t b* α] **apply**(*unfold stable-def*) **apply** *blast*
done

abbreviation *mp-free-precond8-1 t b* $\alpha \equiv$
  mp-free-precond8-inv t b α ∩ {|α > *0*|}

lemma *mp-free-precond8-1-imp-free-block-r*:
mp-free-precond8-1 t b α ⊆ {|´*free-block-r t*|}
  using *stm8-loopinv-*α **by** *blast*

lemma *mp-free-precond8-1-stb* : *stable* (*mp-free-precond8-1 t b* α) (*Mem-pool-free-rely t*)
  apply(*rule stable-int2*)
  using *mp-free-precond8-inv-stb* **apply** *blast*
  apply(*simp add*:*stable-def*)
done

abbreviation *mp-free-precond8-1' t b* ≡
  mp-free-precond8 t b ∩ {|´*free-block-r t*|}

lemma *mp-free-precond8-1'-stb* : *stable* (*mp-free-precond8-1' t b*) (*Mem-pool-free-rely t*)
  apply(*rule stable-int2*)
  using *mp-free-precond8-stb* **apply** *blast*
  apply(*simp add*:*stable-def*) **apply** *clarify*
 apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *smt*

abbreviation *mp-free-precond8-2 t b* $\alpha \equiv$

mp-free-precond8-1 t b α ∩ {| ´*lsz t* = ´*lsizes t* ! (´*lvl t*) |}

lemma *mp-free-precond8-2-stb* : *stable* (*mp-free-precond8-2 t b α*) (*Mem-pool-free-rely t*)
  apply(*rule stable-int2*)
  using *mp-free-precond8-1-stb* **apply** *blast*
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*simp add*:*Mem-pool-free-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by *smt*

abbreviation *mp-free-precond8-3 t b α* ≡
  mp-free-precond8-2 t b α ∩ {| ´*blk t* = *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bn t*) |}

lemma *mp-free-precond8-3-stb* : *stable* (*mp-free-precond8-3 t b α*) (*Mem-pool-free-rely t*)
  apply(*rule stable-int2*)
  using *mp-free-precond8-2-stb* **apply** *blast*
  apply(*simp add*:*stable-def block-ptr-def Mem-pool-free-rely-def*) **apply** *clarify*
  apply(*case-tac x* = *y*) **apply** *simp* **apply** *clarsimp*
  apply(*subgoal-tac blk x t* = *blk y t*)
  apply(*subgoal-tac lsz x t* = *lsz y t*)
  apply(*subgoal-tac bn x t* = *bn y t*)
  apply(*subgoal-tac buf* (*mem-pool-info x* (*pool b*)) = *buf* (*mem-pool-info y* (*pool b*)))
  apply *simp*
  apply(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
  apply(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
  apply(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*) **apply** *metis*
done

abbreviation *mp-free-precond9 t b* ≡ *mp-free-precond1 t b*
term *mp-free-precond1 t b*

lemma *mp-free-precond9-stb* : *stable* (*mp-free-precond9 t b*) (*Mem-pool-free-rely t*)
  using *mp-free-precond1-stb* **apply** *auto*[*1*]
  done

20.2 proof of each statement

lemma *mempool-free-stm1-inv-mempool-info*:
  inv-mempool-info Va ∧ *inv-bitmap-freelist Va* ⟹
    block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
    level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
    pool b ∈ *mem-pools Va* ⟹
    get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
    inv-mempool-info
     (*Va*(|*mem-pool-info* := (*mem-pool-info Va*)

(*pool b* := *mem-pool-info Va* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)]|)),
freeing-node := *freeing-node Va*(*t* ↦ *b*)|))
apply(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*) **apply** (*simp add*:*Let-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*) **apply** (*simp add*:*Let-def*)
apply(*rule allI*) **apply**(*rule impI*)
apply(*subgoal-tac* (∀ *i*<*length* (*levels* (*mem-pool-info Va* (*pool b*))).
length (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*)) = *n-max* (*mem-pool-info Va* (*pool b*)) ∗ *4* ˆ *i*))
apply(*case-tac i* = *level b*)
apply *auto*[*1*] **apply** *auto*[*1*]
apply(*simp add*:*Let-def*)
done

lemma *mempool-free-stm1-inv-bitmap-freelist*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* ∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
inv-bitmap-freelist
(*Va*(|*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)]|)),
freeing-node := *freeing-node Va*(*t* ↦ *b*)|))
apply(*simp add*:*inv-bitmap-freelist-def*)
apply(*rule allI*) **apply**(*rule impI*)+ **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*] **apply** *fastforce*
apply(*case-tac i* ≠ *level b*) **apply** *auto*[*1*]
apply(*case-tac j* ≠ *block b*) **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*rule conjI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac i* ≠ *level b*) **apply** *auto*[*1*]
apply(*case-tac j* ≠ *block b*) **apply** *auto*[*1*]

apply *auto*[*1*]
apply(*simp add*:*distinct-def*)
apply(*case-tac i* = *level b*) **apply** *auto*[*1*]
apply *auto*[*1*]
done

lemma *mempool-free-stm1-inv-bitmap*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* ∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
inv-bitmap
(*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈]⦈),
freeing-node := *freeing-node Va*(*t* ↦ *b*)⦈)
apply(*simp add*:*inv-bitmap-def*)
apply(*rule allI*) **apply**(*simp add*:*Let-def*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*rule conjI*) **apply**(*rule impI*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac i* − *1* = *level b* ∧ *j div 4* = *block b*)
apply (*metis* (*no-types*, *lifting*) *BlockState*.*distinct*(*3*) *One-nat-def Suc-pred lessI nat-neq-iff nth-list-update-neq*)
apply(*rule impI*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈] ! (*i* − *Suc NULL*)) ! (*j div 4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*i* − *Suc NULL*)) ! (*j div 4*))
apply (*metis Mem-pool-lvl*.*cases Mem-pool-lvl*.*simps*(*1*) *Mem-pool-lvl*.*simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl*.*cases Mem-pool-lvl*.*simps*(*1*) *Mem-pool-lvl*.*simps*(*4*) *One-nat-def nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl*.*cases Mem-pool-lvl*.*simps*(*1*) *Mem-pool-lvl*.*simps*(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule impI*)
apply(*rule conjI*)
apply(*case-tac i = level b ∧ j = block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i = level b ∧ j ∗ 4 = block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i = level b ∧ j = block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i = level b ∧ Suc* (*j* ∗ *4*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i = level b ∧ j = block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i = level b ∧ Suc* (*Suc* (*j* ∗ *4*)) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*

= *bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *i*) ! *j*)
apply(*subgoal-tac* *bits* ((*levels* (*mem-pool-info* *Va* (*pool* *b*)))
[*level* *b* := (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*))[*block* *b*
:= *FREEING*]|)] ! (*Suc* *i*)) ! *Suc* (*Suc* (*j* ∗ *4*))
= *bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! (*Suc* *i*)) ! *Suc* (*Suc* (*j* ∗
4)))
apply (*metis* *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update *nth-list-update-eq* *nth-list-update-neq*)
apply (*metis* *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq *nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq* *nth-list-update-neq*)

apply(*case-tac* *i* = *level* *b* ∧ *j* = *block* *b*) **apply** *auto*[*1*]
apply(*case-tac* *Suc* *i* = *level* *b* ∧ (*j* ∗ *4* + *3*) = *block* *b*)
apply (*metis* *BlockState.distinct*(*5*) *less-Suc-eq* *nth-list-update-neq* *order-less-irrefl*)
apply(*subgoal-tac* *bits* ((*levels* (*mem-pool-info* *Va* (*pool* *b*)))
[*level* *b* := (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*))[*block* *b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *i*) ! *j*)
apply(*subgoal-tac* *bits* ((*levels* (*mem-pool-info* *Va* (*pool* *b*)))
[*level* *b* := (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*))[*block* *b*
:= *FREEING*]|)] ! (*Suc* *i*)) ! (*j* ∗ *4* + *3*)
= *bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! (*Suc* *i*)) ! (*j* ∗ *4* + *3*))
apply (*metis* *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update *nth-list-update-eq* *nth-list-update-neq*)
apply (*metis* *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq *nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases* *Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq* *nth-list-update-neq*)

apply(*rule* *conjI*) **apply**(*rule* *impI*)
apply(*rule* *conjI*) **apply**(*rule* *impI*)
apply(*case-tac* *i* = *level* *b* ∧ *j* = *block* *b*) **apply** *auto*[*1*]
apply(*case-tac* *i* − *1* = *level* *b* ∧ *j* *div* *4* = *block* *b*)
apply (*metis* (*no-types*, *lifting*) *BlockState.distinct*(*3*) *One-nat-def* *Suc-pred*
lessI *nat-neq-iff* *nth-list-update-neq*)
apply(*subgoal-tac* *bits* ((*levels* (*mem-pool-info* *Va* (*pool* *b*)))
[*level* *b* := (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*))[*block* *b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *i*) ! *j*)
apply(*subgoal-tac* *bits* ((*levels* (*mem-pool-info* *Va* (*pool* *b*)))
[*level* *b* := (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *Va* (*pool* *b*)) ! *level* *b*))[*block* *b*
:= *FREEING*]|)] ! (*i* − *Suc* *NULL*)) ! (*j* *div* *4*)

= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*i* − *Suc NULL*)) ! (*j* *div 4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *One-nat-def nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule impI*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *j* ∗ *4* = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*j* ∗ *4*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule conjI*)
apply(*case-tac i = level b* ∧ *j = block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i = level b* ∧ *Suc* (*Suc* (*j* ∗ *4*)) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*))
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*)))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*case-tac i = level b* ∧ *j = block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i = level b* ∧ (*j* ∗ *4* + *3*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4* + *3*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4* + *3*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule conjI*) **apply**(*rule impI*)
apply(*rule conjI*) **apply**(*rule impI*)
apply(*case-tac i = level b* ∧ *j = block b*) **apply** *auto*[*1*]
apply(*case-tac i* − *1* = *level b* ∧ *j div 4* = *block b*)
apply (*metis* (*no-types*, *lifting*) *BlockState.distinct*(*3*) *One-nat-def Suc-pred lessI nat-neq-iff nth-list-update-neq*)

apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*i* − *Suc NULL*)) ! (*j div 4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*i* − *Suc NULL*)) ! (*j*
div 4))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
One-nat-def nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule impI*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *j* ∗ *4* = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*j* ∗ *4*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)

(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*Suc* (*j* ∗ *4*)) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*))
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*)))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ (*j* ∗ *4* + *3*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4* + *3*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4* + *3*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)

Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule conjI*) **apply**(*rule impI*)
apply(*rule conjI*) **apply**(*rule impI*)
apply(*case-tac* $i = $ *level* $b \wedge j = $ *block* b) **apply** *auto*[*1*]
apply(*case-tac* $i - 1 = $ *level* $b \wedge j$ *div* $4 = $ *block* b)
apply (*metis* (*no-types*, *lifting*) *BlockState.distinct*(*3*) *One-nat-def Suc-pred lessI nat-neq-iff nth-list-update-neq*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! ($i -$ *Suc NULL*)) ! (*j div 4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! ($i -$ *Suc NULL*)) ! (*j div 4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
One-nat-def nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule impI*)
apply(*rule conjI*)
apply(*case-tac* $i = $ *level* $b \wedge j = $ *block* b) **apply** *auto*[*1*]
apply(*case-tac Suc* $i = $ *level* $b \wedge j * 4 = $ *block* b)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)] ! (*Suc i*)) ! ($j * 4$)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! ($j * 4$))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac* $i = $ *level* $b \wedge j = $ *block* b) **apply** *auto*[*1*]

apply(*case-tac Suc i = level b* ∧ *Suc* (*j* ∗ *4*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*Suc* (*j* ∗ *4*)) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*))
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗
4)))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ (*j* ∗ *4* + *3*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))

[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*Suc i*)) ! (*j* ∗ *4* + *3*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4* + *3*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule conjI*)
apply(*rule impI*)+
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac i* − *1* = *level b* ∧ *j div 4* = *block b*)
apply (*metis* (*no-types*, *lifting*) *BlockState.distinct*(*3*) *One-nat-def Suc-pred
lessI nat-neq-iff nth-list-update-neq*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! (*i* − *Suc NULL*)) ! (*j div 4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*i* − *Suc NULL*)) ! (*j div
4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
length-list-update nth-list-update-eq nth-list-update-neq)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*)
One-nat-def nth-list-update-eq nth-list-update-neq)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*)
Mem-pool-lvl.simps(*4*) *nth-list-update-eq nth-list-update-neq*)

apply(*rule conjI*)
apply(*rule impI*)+
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *j* ∗ *4* = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]|)] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*

:= *FREEING*]⦈] ! (*Suc i*)) ! (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*j* ∗ *4*) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]⦈] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]⦈] ! (*Suc i*)) ! *Suc* (*j* ∗ *4*)
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*j* ∗ *4*))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply(*rule conjI*)
apply(*case-tac i* = *level b* ∧ *j* = *block b*) **apply** *auto*[*1*]
apply(*case-tac Suc i* = *level b* ∧ *Suc* (*Suc* (*j* ∗ *4*)) = *block b*)
apply (*metis BlockState.distinct*(*5*) *less-Suc-eq nth-list-update-neq order-less-irrefl*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]⦈] ! *i*) ! *j*
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *i*) ! *j*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b*
:= *FREEING*]⦈] ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗ *4*))
= *bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! (*Suc i*)) ! *Suc* (*Suc* (*j* ∗
4)))
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *length-list-update nth-list-update-eq nth-list-update-neq*)
apply (*metis Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)
apply (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.cases Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *nth-list-update-eq nth-list-update-neq*)

```
    apply(case-tac i = level b ∧ j = block b) apply auto[1]
    apply(case-tac Suc i = level b ∧ (j ∗ 4 + 3) = block b)
   apply (metis BlockState.distinct(5) less-Suc-eq nth-list-update-neq order-less-irrefl)
    apply(subgoal-tac bits ((levels (mem-pool-info Va (pool b)))
              [level b := (levels (mem-pool-info Va (pool b)) ! level b)
                 (|bits := (bits (levels (mem-pool-info Va (pool b)) ! level b))[block b
:= FREEING]|)] ! i) ! j
               = bits (levels (mem-pool-info Va (pool b)) ! i) ! j)
    apply(subgoal-tac bits ((levels (mem-pool-info Va (pool b)))
              [level b := (levels (mem-pool-info Va (pool b)) ! level b)
                 (|bits := (bits (levels (mem-pool-info Va (pool b)) ! level b))[block b
:= FREEING]|)] ! (Suc i)) ! (j ∗ 4 + 3)
               = bits (levels (mem-pool-info Va (pool b)) ! (Suc i)) ! (j ∗ 4 + 3))
   apply (metis Mem-pool-lvl.cases Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4)
         length-list-update nth-list-update-eq nth-list-update-neq)
   apply (metis Mem-pool-lvl.cases Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4)
          nth-list-update-eq nth-list-update-neq)
    apply (metis (no-types, lifting) Mem-pool-lvl.cases Mem-pool-lvl.simps(1)
         Mem-pool-lvl.simps(4) nth-list-update-eq nth-list-update-neq)

  apply(rule impI)+
    apply(case-tac i = level b ∧ j = block b) apply auto[1]
    apply(case-tac i − 1 = level b ∧ j div 4 = block b) apply auto[1]
    apply(subgoal-tac bits ((levels (mem-pool-info Va (pool b)))
              [level b := (levels (mem-pool-info Va (pool b)) ! level b)
                 (|bits := (bits (levels (mem-pool-info Va (pool b)) ! level b))[block b
:= FREEING]|)] ! i) ! j
               = bits (levels (mem-pool-info Va (pool b)) ! i) ! j)
    prefer 2 apply (metis (no-types, lifting) Mem-pool-lvl.cases Mem-pool-lvl.simps(1)
         Mem-pool-lvl.simps(4) nth-list-update-eq nth-list-update-neq)
    apply(subgoal-tac bits ((levels (mem-pool-info Va (pool b)))
              [level b := (levels (mem-pool-info Va (pool b)) ! level b)
                 (|bits := (bits (levels (mem-pool-info Va (pool b)) ! level b))[block b
:= FREEING]|)] ! (i − 1)) ! (j div 4)
               = bits (levels (mem-pool-info Va (pool b)) ! (i − 1)) ! (j div 4))
    prefer 2 apply (metis Mem-pool-lvl.cases Mem-pool-lvl.simps(1) Mem-pool-lvl.simps(4)
          nth-list-update-eq nth-list-update-neq)
    apply(subgoal-tac bits (levels (mem-pool-info Va (pool b)) ! (i − Suc NULL))
! (j div 4) ≠ DIVIDED)
     prefer 2 apply(subgoal-tac length (bits ((levels (mem-pool-info Va (pool b)))
                         [level b := (levels (mem-pool-info Va (pool b)) ! level b)
                            (|bits := (bits (levels (mem-pool-info Va (pool b)) ! level
b))[block b := FREEING]|)] !
                         i)) = length (bits (levels (mem-pool-info Va (pool b)) ! i)))
       prefer 2 apply(case-tac i = level b)
        apply auto[1] apply auto[1]
       apply simp
```

apply *simp*
done

lemma *mempool-free-stm1-inv-auxvars*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* ∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
data b = *block-ptr* (*mem-pool-info Va* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info Va* (*pool b*))) *div 4* ^ *level b*) (*block b*) ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
allocating-node Va t = *None* ⟹
freeing-node Va t = *None* ⟹
inv-aux-vars
(*Va*(|*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)]|)),
freeing-node := *freeing-node Va*(*t* ↦ *b*)|))
apply(*unfold inv-aux-vars-def*)
apply(*rule conjI*)
apply *clarify*
apply(*case-tac ta* = *t*) **apply** *auto*[*1*]
apply(*subgoal-tac* ¬(*pool n* = *pool b* ∧ *level n* = *level b* ∧ *block n* = *block b*))
apply(*subgoal-tac freeing-node*
(*Va*(|*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)]|)),
freeing-node := *freeing-node Va*(*t* ↦ *b*)|)) *ta* = *freeing-node Va ta*)
apply(*subgoal-tac get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *FREEING*)
apply(*subgoal-tac get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) =
get-bit
(*mem-pool-info*
(*Va*(|*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
(|*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]|)]|)),
freeing-node := *freeing-node Va*(*t* ↦ *b*)|)))
(*pool n*) (*level n*) (*block n*))
apply *auto*[*1*]

```
      apply(case-tac ¬ pool n = pool b) apply simp
      apply(case-tac ¬ level n = level b) apply simp
      apply(case-tac ¬ block n = block b) apply simp apply simp
    apply auto[1] apply auto[1]

    apply(subgoal-tac freeing-node Va ta = Some n) prefer 2 apply auto[1]
    apply fastforce

  apply(rule conjI)
    apply clarify
    apply(case-tac ¬(pool n = pool b ∧ level n = level b ∧ block n = block b))
       apply(subgoal-tac get-bit (mem-pool-info Va) (pool n) (level n) (block n) =
FREEING)
        prefer 2 apply auto[1]
          using set-bit-def set-bit-get-bit-neq apply auto[1]
          using set-bit-def set-bit-get-bit-neq apply auto[1]
      apply(subgoal-tac mem-block-addr-valid Va n)
        prefer 2 using mem-block-addr-valid-def apply auto[1]
      apply(subgoal-tac ∃ t'. t' ≠ t ∧ freeing-node Va t' = Some n)
        prefer 2 apply (metis option.discI)
      apply auto[1]

      apply(subgoal-tac data b = data n)
      prefer 2 apply(simp add:block-ptr-def mem-block-addr-valid-def inv-mempool-info-maxsz-align4)
        apply auto[1]

  apply(rule conjI)
    apply clarify
    apply(case-tac ta = t) apply auto[1]
    apply(subgoal-tac allocating-node Va ta = Some n)
      prefer 2 apply auto[1]
    apply(subgoal-tac get-bit-s Va (pool n) (level n) (block n) = ALLOCATING)
      prefer 2 apply auto[1]
    apply(case-tac ¬(pool n = pool b ∧ level n = level b ∧ block n = block b))
      apply(case-tac ¬ pool n = pool b) apply simp
      apply(case-tac ¬ level n = level b) apply force
      apply(case-tac ¬ block n = block b) apply force apply simp
    apply fastforce

  apply(rule conjI)
    apply clarify
    apply(case-tac ¬(pool n = pool b ∧ level n = level b ∧ block n = block b))
       apply(subgoal-tac get-bit (mem-pool-info Va) (pool n) (level n) (block n) =
ALLOCATING)
        prefer 2 apply auto[1]
          using set-bit-def set-bit-get-bit-neq apply auto[1]
          using set-bit-def set-bit-get-bit-neq apply auto[1]
      apply(subgoal-tac mem-block-addr-valid Va n)
        prefer 2 using mem-block-addr-valid-def apply auto[1]
```

apply(*subgoal-tac* $\exists t'. t' \neq t \wedge$ *allocating-node Va* $t' =$ *Some n*)
  prefer *2* **apply** (*metis option.discI*)
 apply *auto*[*1*]

 apply(*subgoal-tac data b = data n*)
 prefer *2* **apply**(*simp add:block-ptr-def mem-block-addr-valid-def inv-mempool-info-maxsz-align4*)
  apply *auto*[*1*]

apply(*rule conjI*)
 apply *clarify*
 apply(*case-tac t1* $\neq$ *t* $\wedge$ *t2* $\neq$ *t*)
  apply *auto*[*1*]
  apply(*case-tac t1 = t*)
   apply *clarify*
   apply(*subgoal-tac freeing-node Va t2 = Some n2*)
    prefer *2* **apply** *auto*[*1*]
   apply(*subgoal-tac b = n1*)
    prefer *2* **apply** *auto*[*1*]
   apply *simp*

  apply(*case-tac t2 = t*)
   apply *clarify*
   apply(*subgoal-tac freeing-node Va t1 = Some n1*)
    prefer *2* **apply** *auto*[*1*]
   apply(*subgoal-tac b = n2*)
    prefer *2* **apply** *auto*[*1*]
   apply *fastforce*

  apply *simp*

apply(*rule conjI*)
 apply *clarify*
 apply(*case-tac t1* $\neq$ *t* $\wedge$ *t2* $\neq$ *t*)
  apply *auto*[*1*]
  apply(*case-tac t1 = t*)
   apply *clarify*
   apply(*subgoal-tac freeing-node Va t2 = Some n2*)
    prefer *2* **apply** *auto*[*1*]
   apply(*subgoal-tac b = n1*)
    prefer *2* **apply** *auto*[*1*]
   apply *simp*

  apply(*case-tac t2 = t*)
   apply *clarify*
   apply(*subgoal-tac freeing-node Va t1 = Some n1*)
    prefer *2* **apply** *auto*[*1*]
   apply(*subgoal-tac b = n2*)
    prefer *2* **apply** *auto*[*1*]
   apply *fastforce*

apply *simp*

apply *clarify*
apply(*case-tac* $t1 \neq t \wedge t2 \neq t$)
apply *auto*[*1*]
apply(*case-tac* $t1 = t$)
apply *clarify*
apply(*subgoal-tac freeing-node Va t2 = Some n2*)
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac b = n1*)
prefer *2* **apply** *auto*[*1*]
apply *simp*

apply(*case-tac* $t2 = t$)
apply *clarify*
apply(*subgoal-tac allocating-node Va t1 = Some n1*)
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac b = n2*)
prefer *2* **apply** *auto*[*1*]
apply *fastforce*

apply *simp*
done

lemma *mempool-free-stm1-inv-lvl0*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ∧ *inv-bitmap0 Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
data b = *block-ptr* (*mem-pool-info Va* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info Va* (*pool b*))) *div 4* ^ *level b*) (*block b*) ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
allocating-node Va t = *None* ⟹
freeing-node Va t = *None* ⟹
inv-bitmap0
(*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈]⦈),
freeing-node := *freeing-node Va*(*t* ↦ *b*)⦈)
apply(*simp add*: *inv-bitmap0-def Let-def*)
apply *clarsimp*
apply(*case-tac level b* = *0*)
apply(*case-tac block b* = *i*) **apply** *auto*[*1*] **apply** *simp*

apply *simp*
done

lemma *mempool-free-stm1-inv-lvln*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ∧ *inv-bitmapn Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
data b = *block-ptr* (*mem-pool-info Va* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info Va* (*pool b*))) *div 4* ^ *level b*) (*block b*) ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
allocating-node Va t = *None* ⟹
freeing-node Va t = *None* ⟹
inv-bitmapn
(*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈]⦈),
freeing-node := *freeing-node Va*(*t* ↦ *b*)⦈)
apply(*simp add*: *inv-bitmapn-def Let-def*)
apply *clarsimp*
apply(*case-tac level b* = *length* (*levels* (*mem-pool-info Va* (*pool b*))) − *1*)
apply(*case-tac block b* = *i*) **apply** *auto*[*1*] **apply** *simp*
apply *simp*
done

lemma *mempool-free-stm1-inv-lvls-not4free*:
inv-cur Va ∧ *inv-thd-waitq Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
∧ *inv-bitmap Va* ∧ *inv-aux-vars Va* ∧ *inv-bitmap-not4free Va* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
pool b ∈ *mem-pools Va* ⟹
data b = *block-ptr* (*mem-pool-info Va* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info Va* (*pool b*))) *div 4* ^ *level b*) (*block b*) ⟹
get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
allocating-node Va t = *None* ⟹
freeing-node Va t = *None* ⟹
inv-bitmap-not4free
(*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
(*pool b* := *mem-pool-info Va* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
[*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈]⦈),
freeing-node := *freeing-node Va*(*t* ↦ *b*)⦈)
apply(*simp add*: *inv-bitmap-not4free-def Let-def partner-bits-def*)

apply *clarsimp*
apply(*case-tac level b = i*) **prefer** *2* **apply** *auto*[*1*]
  apply(*case-tac block b = j div 4 ∗ 4*) **apply** *auto*[*1*]
  apply(*case-tac block b = j div 4 ∗ 4 + 1*) **apply** *auto*[*1*]
  apply(*case-tac block b = j div 4 ∗ 4 + 2*) **apply** *auto*[*1*]
  apply(*case-tac block b = j div 4 ∗ 4 + 3*) **apply** *auto*[*1*]

  apply *simp*
done

lemma *mempool-free-smt1-inv*:
  inv Va ⟹
    block b < *length* (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)) ⟹
    level b < *length* (*levels* (*mem-pool-info Va* (*pool b*))) ⟹
    pool b ∈ *mem-pools Va* ⟹
   data b = *block-ptr* (*mem-pool-info Va* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info Va* (*pool b*))) *div 4* ^ *level b*) (*block b*) ⟹
    get-bit (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*) = *ALLOCATED* ⟹
    allocating-node Va t = *None* ⟹
    freeing-node Va t = *None* ⟹
    inv (*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
         (*pool b* := *mem-pool-info Va* (*pool b*)
           ⦇*levels* := (*levels* (*mem-pool-info Va* (*pool b*)))
             [*level b* := (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*)
               ⦇*bits* := (*bits* (*levels* (*mem-pool-info Va* (*pool b*)) ! *level b*))[*block b* := *FREEING*]⦈]⦈),
         freeing-node := *freeing-node Va*(*t* ↦ *b*)⦈)
  apply(*simp add*:*inv-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def Mem-pool-free-guar-def*)
  apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*)
      apply(*rule conjI*) **apply** *clarify* **apply** *metis*
      apply *clarify* **apply** *metis*
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-mempool-info* **apply** *auto*[*1*]
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-bitmap-freelist* **apply** *auto*[*1*]
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-bitmap* **apply** *auto*[*1*]
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-auxvars* **apply** *auto*[*1*]
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-lvl0* **apply** *auto*[*1*]
  apply(*rule conjI*) **using** *mempool-free-stm1-inv-lvln* **apply** *auto*[*1*]
          using *mempool-free-stm1-inv-lvls-not4free* **apply** *auto*[*1*]
done

lemma *mempool-free-stm1-h1*:
  Mem-pool-free-pre t ∩
      ⦃*pool b* ∈ ´*mem-pools* ∧
      level b < *length* (*levels* (´*mem-pool-info* (*pool b*))) ∧
      block b < *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*)) ! *level b*)) ∧
      data b =
       block-ptr (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info*

(*pool b*))) *div 4* ˆ *level b*) (*block b*)⦄ ∩
⦃´*cur* = *Some t*⦄ ∩
{*Va*} ∩
⦃´*get-bit-s* (*pool b*) (*level b*) (*block b*) = *ALLOCATED*⦄ ∩
{*Va*} ≠
{} ⟹
Γ ⊢$_I$ *Some* (´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (*level b*) (*block b*);;
´*freeing-node* := ´*freeing-node*(*t* ↦
b)) *sat*$_p$ [*Mem-pool-free-pre t* ∩
⦃*pool b* ∈ ´*mem-pools* ∧
level b < *length* (*levels* (´*mem-pool-info* (*pool b*))) ∧
block b < *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*)) ! *level b*)) ∧
data b =
block-ptr (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ˆ *level b*)
(*block b*)⦄ ∩
⦃´*cur* = *Some t*⦄ ∩
{*Va*} ∩
⦃´*get-bit-s* (*pool b*) (*level b*) (*block b*) = *ALLOCATED*⦄ ∩
{*Va*}, {(*x*, *y*).
x = *y*}, *UNIV*, ⦃´(*Pair Va*) ∈ *Mem-pool-free-guar t*⦄ ∩
(⦃´*invariant.inv* ∧ ´*allocating-node t* = *None*⦄ ∩
⦃*pool b* ∈ ´*mem-pools* ∧
level b < *length* (*levels* (´*mem-pool-info* (*pool b*))) ∧
block b < *length* (*bits* (*levels* (´*mem-pool-info* (*pool b*)) ! *level b*)) ∧
data b =
block-ptr (´*mem-pool-info* (*pool b*))
(*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ˆ *level b*) (*block b*)⦄ ∩
mp-free-precond2-ext t b)]
apply *clarsimp*
apply(*rule Seq*[**where** *mid*={*Va*⦇*mem-pool-info* := *set-bit-freeing* (*mem-pool-info Va*) (*pool b*) (*level b*) (*block b*)⦈}])
apply(*rule Basic*)
apply *auto*[*1*] **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply(*simp add*: *set-bit-def*)
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*) **apply** *auto*[*1*]
apply(*case-tac i* = *level b*) **apply** *auto*[*1*] **apply** *auto*[*1*]

apply(*rule conjI*)

using *mempool-free-smt1-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*)

apply(*rule conjI*)
using *mempool-free-smt1-inv* **apply** *auto*[*1*]
apply(*simp add*:*block-ptr-def*)

apply(*simp add*:*stable-def*)+
done

lemma *mempool-free-stm1*:
$\Gamma \vdash_I$ *Some* (*t* ► *AWAIT bits* (*levels* (´*mem-pool-info* (*pool b*)) ! *level b*) ! *block b* = *ALLOCATED THEN*
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (*level b*) (*block b*);;
´*freeing-node* := ´*freeing-node* (*t* := *Some b*)
END) sat_p
[*mp-free-precond1 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond2 t b*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)

using *mp-free-precond1-stb* **apply** *auto*[*1*]
using *mp-free-precond2-stb* **apply** *auto*[*1*]

apply(*rule allI*)
apply(*rule Await*)
apply(*simp add*:*stable-def*) **apply**(*auto simp add*:*stable-def*)
apply(*case-tac V* ≠ *Va*) **apply** *auto*[*1*] **using** *Emptyprecond* **apply** *blast*
apply *clarsimp*
apply(*case-tac mp-free-precond1 t b* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*Va*} ∩
⦃*get-bit* ´*mem-pool-info* (*pool b*) (*level b*) (*block b*) = *ALLOCATED*⦄ ∩
{*Va*} = {})
apply *simp* **using** *Emptyprecond* **apply** *auto*[*1*]
using *mempool-free-stm1-h1* **apply** *force*
done

lemma *mempool-free-stm2*:
$\Gamma \vdash_I$ *Some* (*t* ► ´*need-resched* := ´*need-resched*(*t* := *False*)) sat_p
[*mp-free-precond2 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond3 t b*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-free-precond2-stb* **apply** *simp*
using *mp-free-precond3-stb* **apply** *simp*

apply *clarify*

apply(*rule Basic*)
apply(*case-tac mp-free-precond2 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*]
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*, *V*(|*need-resched* := (*need-resched V*)(*t* := *False*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarify* **apply**(*simp add*: *lvars-nochange-def*)

apply(*subgoal-tac* (*V*, *V*(|*need-resched* := (*need-resched V*)(*t* := *False*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply(*simp add*:*stable-def*)+

done

lemma *mempool-free-stm3*:
$\Gamma \vdash_I$ *Some* (*t* ► ´*lsizes* := ´*lsizes*(*t* := [*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))])) sat_p
[*mp-free-precond3 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond4 t b*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-free-precond3-stb* **apply** *simp*
using *mp-free-precond4-stb* **apply** *simp*

apply *clarify*
apply(*rule Basic*)
apply(*case-tac mp-free-precond3 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*]
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*, *V*(|*lsizes* := (*lsizes V*)(*t* := [*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))])|))
∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarify* **apply**(*simp add*: *lvars-nochange-def*)
apply(*subgoal-tac* (*V*, *V*(|*lsizes* := (*lsizes V*)(*t* := [*ALIGN4* (*max-sz* (*mem-pool-info*

V (*pool* b)))])⦈)
$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

done

lemma *mempool-free-stm41-h1-1*: $(n::nat) > 0 \wedge (x::nat)$ *mod* $y = 0 \Longrightarrow n * x$ *mod* $y = 0$
by *auto*

lemma *mempool-free-stm41-h1*:
assumes *p1*: *i V t* $\leq$ *level b*
and *p2*: *length* (*lsizes V t*) = *i V t*
and *p3*: *inv V*
and *p4*: $\forall$ *ii*<*i V t*. *lsizes V t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *ii*
and *p5*: *lsizes V t* $\neq$ []
and *p6*: *pool b* $\in$ *mem-pools V*
and *p7*: *level b* < *length* (*levels* (*mem-pool-info V* (*pool b*)))
and *p8*: *block b* < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*))
and *p9*: *ii* = *i V t*
shows (*lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)]) ! *ii*
= *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *ii*
proof −
from *p2 p9* **have** *a0*: (*lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *1*) *div 4*)]) ! *ii*
= *ALIGN4* (*lsizes V t* ! (*i V t* − *1*) *div 4*)
by (*metis nth-append-length*)

from *p2 p4 p5* **have** *lsizes V t* ! (*i V t* − *1*) *div 4* = *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ (*i V t* − *1*) *div 4*
by (*metis One-nat-def diff-less-mono2 diff-zero length-greater-0-conv zero-less-Suc*)
hence *a1*: *lsizes V t* ! (*i V t* − *1*) *div 4* = *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ (*i V t*)
by (*metis One-nat-def Suc-pred div-mult2-eq length-greater-0-conv p2 p5 plus-1-eq-Suc power-add power-commutes power-one-right*)

from *p6 p3* **have** $\exists$ *n*>*0*. *max-sz* (*mem-pool-info V* (*pool b*))
= (*4* ∗ *n*) ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V* (*pool b*)))))
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*) **by** *auto*
then obtain *n* **where** *n* > *0* $\wedge$ *max-sz* (*mem-pool-info V* (*pool b*))
= (*4* ∗ *n*) ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V* (*pool b*)))))
by *auto*
hence *a2*: *n* > *0* $\wedge$ *max-sz* (*mem-pool-info V* (*pool b*))
= *n* ∗ (*4* ˆ (*length* (*levels* (*mem-pool-info V* (*pool b*))) + *1*)) **by** *auto*
hence *max-sz* (*mem-pool-info V* (*pool b*)) *mod 4* = *0* **by** *simp*
hence *a3*: *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) = *max-sz* (*mem-pool-info*

V (*pool b*))
  using *align40* **by** *auto*
 with *a1* **have** *a4*: *lsizes V t* ! (*i V t* − *1*) *div 4* = *max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ (*i V t*) **by** *simp*

 from *p1 p2 p7 a2* **have** (*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *i V t*) *mod 4* = *0*
  apply(*subgoal-tac 4* ˆ (*length* (*levels* (*mem-pool-info V* (*pool b*))) + *1*) *div 4* ˆ *i V t mod 4* = *NULL*)
   prefer *2* **using** *pow-lt-mod0*[*of 4 i V t length* (*levels* (*mem-pool-info V* (*pool b*)))+*1*] **apply** *auto*[*1*]
  apply *simp* **using** *mempool-free-stm41-h1-1*
   [*of n 4* ∗ *4* ˆ *length* (*levels* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *i V t 4*]
  using *m-mod-div mempool-free-stm41-h1-1 pow-mod-0* **by** *force*
 with *a0 a1 a3 a4 p9* **show** *?thesis* **using** *align40* **by** *simp*
qed

lemma *mempool-free-stm41*:
 $\Gamma \vdash_I$ *Some* (´*lsizes* := ´*lsizes*
   (*t* := ´*lsizes t* @ [*ALIGN4* (´*lsizes t* ! (´*i t* − *1*) *div 4*)]))
  sat_p [*mp-free-precond4-2 t b* ∩ {|´*cur* = *Some t*|} ∩ {*V*}, {(*s*, *t*). *s* = *t*}, *UNIV*,
   {|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ (*mp-free-precond2 t b* ∩ {|¬ ´*need-resched t*|} ∩
   {|(∀ *ii*<*length* (´*lsizes t*).
    ´*lsizes t* ! *ii* = *ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)) *div 4* ˆ *ii*) ∧ ´*lsizes t* ≠ []|} ∩
   ({|´*i t* ≤ *level b*|} ∩ {|*length* (´*lsizes t*) = *Suc* (´*i t*)|}))]
 apply(*rule Basic*)
 apply(*case-tac mp-free-precond4-2 t b* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
  apply *simp* **apply** *clarify* **apply** *auto*[*1*]
  apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
  apply(*rule disjI1*)
  apply(*rule conjI*)
  apply(*subgoal-tac* (*V*,*V*(|*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])|))∈*lvars-nochange1-4all*)
   using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
   apply(*simp add*:*lvars-nochange-def*)
  apply(*subgoal-tac* (*V*,*V*(|*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])|))∈*lvars-nochange1-4all*)
   using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

   apply(*case-tac ii* < *i V t*) **apply** (*simp add*: *nth-append*)
   apply(*case-tac ii* = *i V t*)
   using *mempool-free-stm41-h1* **apply** *metis*
   apply *simp*
 by (*simp add*:*stable-def*)+

lemma *mempool-free-stm4*:
 $\Gamma \vdash_I$ *Some* (*FOR* ($t \blacktriangleright$ ´*i* := ´*i*(t := *1*));
 ´*i* $t \leq$ *level* b;
 ($t \blacktriangleright$ ´*i* := ´*i*(t := ´*i* t + *1*)) *DO*
 ($t \blacktriangleright$ ´*lsizes* := ´*lsizes*(t := ´*lsizes* t @ [*ALIGN4* (´*lsizes* t ! (´*i* t − *1*) *div* *4*)]))
 ROF) sat_p [*mp-free-precond4* t b, *Mem-pool-free-rely* t, *Mem-pool-free-guar* t, *mp-free-precond5* t b]
 apply(*rule Seq*[**where** *mid*=*mp-free-precond4-1* t b])

 apply(*simp add*:*stm-def*)
 apply(*rule Await*)
 using *mp-free-precond4-stb* **apply** *simp*
 using *mp-free-precond4-1-stb* **apply** *simp*
 apply(*rule allI*)
 apply(*rule Basic*)
 apply(*case-tac mp-free-precond4* t b $\cap$ {|´*cur* = *Some* t|} $\cap$ {*V*} = {})
 apply *auto*[*1*]
 apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply *auto*[*1*]
 apply(*subgoal-tac* (*V*,*V*(|*i* := (*i* *V*)(t := *Suc NULL*)|))∈*lvars-nochange1-4all*)
 using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
 apply(*simp add*:*lvars-nochange-def*)
 apply(*subgoal-tac* (*V*,*V*(|*i* := (*i* *V*)(t := *Suc NULL*)|))∈*lvars-nochange1-4all*)
 using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
 apply(*simp add*:*stable-def*)+

 apply(*rule While*)
 using *mp-free-precond4-1-stb* **apply** *simp*
 apply *auto*[*1*]
 using *mp-free-precond5-stb* **apply** *simp*

 apply(*rule Seq*[**where** *mid*=*mp-free-precond4-3* t b])

 apply(*simp add*:*stm-def*)
 apply(*rule Await*)
 using *mp-free-precond4-2-stb* **apply** *simp*
 using *mp-free-precond4-3-stb* **apply** *simp*
 apply(*rule allI*)
 using *mempool-free-stm41* **apply** *simp*

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-free-precond4-3-stb* **apply** *simp*
using *mp-free-precond4-1-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-free-precond4-3 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*] **apply** *clarify* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*) **apply** *auto*[*1*]
apply(*subgoal-tac* (*V*,*V*(|*i* := (*i V*)(*t* := *Suc* (*i V t*))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* (*V*,*V*(|*i* := (*i V*)(*t* := *Suc* (*i V t*))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+ **apply**(*simp add*: *Mem-pool-free-guar-def Id-def*)

done

lemma *mempool-free-stm5*:
$\Gamma \vdash_I$ *Some* (*t* ► ´*free-block-r* := ´*free-block-r* (*t* := *True*))
sat$_p$ [*mp-free-precond5 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond6 t b*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-free-precond5-stb* **apply** *simp*
using *mp-free-precond6-stb* **apply** *simp*

apply *clarify*
apply(*rule Basic*)
apply(*case-tac mp-free-precond5 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*]
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*,*V*(|*free-block-r* := (*free-block-r V*)(*t* := *True*)|)) $\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarify* **apply**(*simp add*: *lvars-nochange-def*)
apply(*subgoal-tac* (*V*,*V*(|*free-block-r* := (*free-block-r V*)(*t* := *True*)|)) $\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply(*simp add*:*stable-def*)+
done

lemma *mempool-free-stm6*:
$\Gamma \vdash_I$ *Some* (t ▶ ´*bn* := ´*bn* (t := *block b*))
sat_p [*mp-free-precond6 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond7 t b*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-free-precond6-stb* **apply** *simp*
using *mp-free-precond7-stb* **apply** *simp*

apply *clarify*
apply(*rule Basic*)
apply(*case-tac mp-free-precond6 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {V} = {})
apply *auto*[*1*]
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* (V, V(|*bn* := (*bn V*)(t := *block b*)|)) $\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarify* **apply**(*simp add*: *lvars-nochange-def*)
apply(*subgoal-tac* (V, V(|*bn* := (*bn V*)(t := *block b*)|)) $\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+
done

lemma *mempool-free-stm7*:
$\Gamma \vdash_I$ *Some* (t ▶ ´*lvl* := ´*lvl* (t := *level b*))
sat_p [*mp-free-precond7 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond8 t b*]
apply(*unfold stm-def*)
apply(*rule Await*)
using *mp-free-precond7-stb* **apply** *simp*
using *mp-free-precond8-stb*[*of t b*] **apply** *fast*

apply *clarify*
apply(*rule Basic*)
apply(*case-tac mp-free-precond7 t b* $\cap$ {|´*cur* = *Some t*|} $\cap$ {V} = {})
apply *auto*[*1*]
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)

apply(*rule conjI*)
apply(*subgoal-tac* $(V, V(\!|lvl := (lvl\ V)(t := level\ b)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add:lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarify* **apply**(*simp add: lvars-nochange-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V, V(\!|lvl := (lvl\ V)(t := level\ b)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add:lvars-nochange1-4all-def lvars-nochange1-def*)

apply *auto*[*1*]
apply (*simp add: block-ptr-def inv-maxsz-align4*)
apply (*metis inv-mempool-info-def inv-def*)
apply *simp*
apply(*simp add:stable-def*)
using *stable-id2* **apply** *metis*
done

20.3 statement 8

abbreviation *free-stm8-precond1 Va t b* $\equiv$ *Va*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info Va*) (*pool b*) (*lvl Va t*) (*bn Va t*)|)
abbreviation *free-stm8-precond2 Va t b* $\equiv$ (*free-stm8-precond1 Va t b*)(|*freeing-node* := (*freeing-node Va*)(t := *None*)|)
abbreviation *free-stm8-loopinv1 Va t b* $\equiv$
{V. *let minf0* = (*mem-pool-info Va*)(*pool b*);
lvl0 = (*levels minf0*) ! (*lvl Va t*);
minf1 = (*mem-pool-info V*)(*pool b*);
lvl1 = (*levels minf1*) ! (*lvl Va t*) *in*
cur V = *cur Va* $\wedge$ *tick V* = *tick Va* $\wedge$ *thd-state V* = *thd-state Va* $\wedge$ (V,*Va*)$\in$*gvars-conf-stable*
$\wedge$ ($\forall p$. $p \neq$ *pool b* $\longrightarrow$ *mem-pool-info V p* = *mem-pool-info Va p*)
$\wedge$ ($\forall j$. $j \neq$ *lvl Va t* $\longrightarrow$ (*levels minf0*)!j = (*levels minf1*)!j)
$\wedge$ (*bits lvl1* = *list-updates-n* (*bits lvl0*) ((*bn Va t div 4*) $*$ *4*) (*i V t*) *NOEXIST*)
$\wedge$ (*free-list lvl1* = *removes* (*map* (λ*ii*. *block-ptr minf0* (*lsz Va t*) ((*bn Va t div 4*) $*$ *4* + *ii*)) [*0*..<(*i V t*)]) (*free-list lvl0*))
$\wedge$ (*wait-q minf0* = *wait-q minf1*)
$\wedge$ ($\forall t'$. $t' \neq t$ $\longrightarrow$ *lvars-nochange* t' *V Va*)
$\wedge$ *freeing-node Va t* = *freeing-node V t* $\wedge$ *allocating-node Va t* = *allocating-node V t* $\wedge$ *free-block-r Va t* = *free-block-r V t*
$\wedge$ *bn Va t* = *bn V t* $\wedge$ *lvl Va t* = *lvl V t* $\wedge$ *lsz Va t* = *lsz V t* $\wedge$ *lsizes Va t* = *lsizes V t*
$\wedge$ *i V t* $\leq$ *4*}

lemma *V-free-stm8-loopinv1*: *i V t* = *0* $\Longrightarrow$ $V \in$ *free-stm8-loopinv1 V t b*
by(*simp add:Let-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)

abbreviation *free-stm8-precond3 Va t b* $\equiv$ *free-stm8-loopinv1* (*free-stm8-precond2*

Va t b) *t b*
abbreviation *free-stm8-precond4 Va t b* ≡ *free-stm8-precond3 Va t b* ∩ ⦃´*i t* = *4*⦄

abbreviation *free-stm8-precond30 Va t b* ≡ *free-stm8-precond3 Va t b* ∩ ⦃´*i t* < *4*⦄
abbreviation *free-stm8-precond31 V t b* ≡ *V*⦇*bb* := (*bb V*) (*t*:=(*bn V t div 4*) ∗ *4* + *i V t*)⦈
abbreviation *free-stm8-precond32 V t b* ≡
  let minf = *mem-pool-info V* (*pool b*) *in*
    V⦇*mem-pool-info*:= (*mem-pool-info V*) (*pool b* := *minf* ⦇*levels* := (*levels minf*)
      [*lvl V t* := ((*levels minf*) ! (*lvl V t*)) ⦇*bits* := (*bits* ((*levels minf*) ! (*lvl V t*)))
[*bb V t* := *NOEXIST*]⦈] ⦈)⦈
abbreviation *free-stm8-precond33 V t b* ≡
  V⦇*block-pt* := (*block-pt V*) (*t*:=*block-ptr* (*mem-pool-info V* (*pool b*)) (*lsz V t*) (*bb V t*))⦈
abbreviation *free-stm8-precond34 V t b* ≡
  let minf = *mem-pool-info V* (*pool b*) *in*
    V⦇*mem-pool-info*:= (*mem-pool-info V*) (*pool b* := *minf* ⦇*levels* := (*levels minf*)
      [*lvl V t* := ((*levels minf*) ! (*lvl V t*)) ⦇*free-list* := *remove1* (*block-pt V t*) (*free-list* ((*levels minf*) ! (*lvl V t*)))⦈] ⦈)⦈

lemma *mempool-free-stm8-atombody-h1*:
{*free-stm8-precond1 V t b*} ⊆ ⦃´(*freeing-node-update* (λ-. ´*freeing-node*(*t* := *None*))) ∈ {*free-stm8-precond2 V t b*}⦄
  by *fastforce*

lemma *block-fits0-h1*: *maxsz mod mm* = *0* ⟹ *aa* < *nmax* ∗ *mm* ⟹
    maxsz div mm ∗ *aa* + *maxsz div mm* < *Suc* (*nmax* ∗ *maxsz*)
  apply(*subgoal-tac maxsz div mm* ∗ *aa* ≤ *maxsz div mm* ∗ (*nmax* ∗ *mm* − *1*))
    prefer *2* **apply** *auto*[*1*]

  by (*smt Groups.add-ac*(*2*) *Groups.mult-ac*(*2*) *Groups.mult-ac*(*3*) *One-nat-def Suc-leI distrib-left*
      le-imp-less-Suc mod-div-self mult.right-neutral mult-0-right mult-Suc-right mult-less-cancel2 mult-zero-right not-le plus-nat.simps(*2*))

lemma *block-fits0-h2*: (*lvlt*::*nat*) > *0* ⟹ *lvlt* ≤ *lvlb* ⟹ *ivt* < (*4*::*nat*) ⟹ *blockb* < *nmax* ∗ *4* ˆ *lvlb* ⟹
    blockb div 4 ˆ (*lvlb* − *lvlt*) *div 4* ∗ *4* + *ivt* < *nmax* ∗ *4* ˆ *lvlt*
  apply(*subgoal-tac nmax* > *0*) **prefer** *2* **using** *mult-not-zero* **apply** *fastforce*
  apply(*subgoal-tac blockb div 4* ˆ (*lvlb* − *lvlt*) < (*nmax* ∗ *4* ˆ *lvlt*)) **prefer** *2*
    apply(*subgoal-tac blockb* < *nmax* ∗ *4* ˆ (*lvlt* + (*lvlb* − *lvlt*)) ∧ *nmax* ∗ *4* ˆ *lvlt* ≠ *0*) **prefer** *2* **apply** *simp*
    apply(*subgoal-tac* ⋀*n na nb*. ¬ *n* < *na* ∗ *nb* ∨ *n div na* < *nb* ∨ *nb* = *NULL*)
      prefer *2*
  apply (*simp add*: *less-mult-imp-div-less mult.commute*)

apply (*metis mult.commute mult.left-commute power-add*)
apply(*subgoal-tac blockb div 4 ^ (lvlb − lvlt) div 4 ∗ 4 + 4 ≤ nmax ∗ 4 ^ lvlt*)
prefer *2* **apply**(*subgoal-tac* ⋀*x. x < nmax ∗ 4 ^ lvlt* ⟶ *x div 4 ∗ 4 + 4 ≤ nmax ∗ 4 ^ lvlt*)
prefer *2* **apply**(*case-tac x mod 4 = 0*) **apply** *auto*[*1*] **apply**(*rule modn0-xy-n*) **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*] **apply**(*rule divn-multn-addn-le*) **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*
done

lemma *block-fits0*:
V ∈ mp-free-precond8-3 t b α ∩ ⦃´cur = Some t⦄ ⟹
vt ∈ free-stm8-precond3 V t b ∩ ⦃´i t < 4⦄ ⟹
{free-stm8-precond2 V t b} ∩ ⦃0 < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)⦄ ≠ {} ⟹
free-stm8-precond33 (free-stm8-precond32 (free-stm8-precond31 vt t b) t b) t b
∈ ⦃block-fits (´mem-pool-info (pool b)) (´block-pt t) (´lsz t)⦄
apply(*unfold block-fits-def block-ptr-def buf-size-def*) **apply** *clarsimp*
apply(*rule subst*[**where** *s=lsz vt* **and** *t=lsz (let minf = mem-pool-info vt (pool b)*
in vt⦇bb := (bb vt)(t := bn vt t div 4 ∗ 4 + i vt t), mem-pool-info := (mem-pool-info vt)
(pool b := minf⦇levels := (levels minf)[lvl vt t := (levels minf ! lvl vt t)
⦇bits := (bits (levels minf ! lvl vt t))[bn vt t div 4 ∗ 4 + i vt t := NOEXIST]⦈]⦈)⦈)])
apply(*simp add:Let-def*)
apply(*rule subst*[**where** *s=bn vt t div 4 ∗ 4 + i vt t* **and** *t=bb (let minf = mem-pool-info vt (pool b)*
in vt⦇bb := (bb vt)(t := bn vt t div 4 ∗ 4 + i vt t), mem-pool-info := (mem-pool-info vt)
(pool b := minf⦇levels := (levels minf)[lvl vt t := (levels minf ! lvl vt t)
⦇bits := (bits (levels minf ! lvl vt t))[bn vt t div 4 ∗ 4 + i vt t := NOEXIST]⦈]⦈)⦈) t])
apply(*simp add:Let-def*)
apply(*rule subst*[**where** *s=n-max (mem-pool-info vt (pool b))* **and** *t=n-max (mem-pool-info*
(let minf = mem-pool-info vt (pool b)
in vt⦇bb := (bb vt)(t := bn vt t div 4 ∗ 4 + i vt t),
mem-pool-info := (mem-pool-info vt)
(pool b := minf ⦇levels := (levels minf)[lvl vt t := (levels minf ! lvl vt t)
⦇bits := (bits (levels minf ! lvl vt t))[bn vt t div 4 ∗ 4 + i vt t := NOEXIST]⦈]⦈)⦈)
(pool b))])
apply(*simp add:Let-def*)

apply(*rule subst*[**where** *s=max-sz* (*mem-pool-info vt* (*pool b*)) **and** *t=max-sz* (*mem-pool-info*
(*let minf* = *mem-pool-info vt* (*pool b*)
in vt⦇*bb* := (*bb vt*)(*t* := *bn vt t div 4* ∗ *4* + *i vt t*),
mem-pool-info := (*mem-pool-info vt*)
(*pool b* := *minf* ⦇*levels* := (*levels minf*)[*lvl vt t* := (*levels minf* ! *lvl vt t*)
⦇*bits* := (*bits* (*levels minf* ! *lvl vt t*))[*bn vt t div 4* ∗ *4* + *i vt t* := *NOEXIST*]⦈]⦈)⦈)
(*pool b*))])
apply(*simp add*:*Let-def*)

apply(*rule subst*[**where** *s=n-max* (*mem-pool-info V* (*pool b*)) **and** *t=n-max* (*mem-pool-info vt* (*pool b*))])
apply(*simp add*:*Let-def set-bit-def gvars-conf-stable-def gvars-conf-def*)

apply(*rule subst*[**where** *s=max-sz* (*mem-pool-info V* (*pool b*)) **and** *t=max-sz* (*mem-pool-info vt* (*pool b*))])
apply(*simp add*:*Let-def set-bit-def gvars-conf-stable-def gvars-conf-def*)
apply(*rule subst*[**where** *s=ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t* **and** *t=lsz vt t*])
apply(*simp add*:*Let-def*) **apply** *metis*
apply(*rule subst*[**where** *s=block b div 4* ˆ (*level b* − *lvl V t*) **and** *t=bn vt t*])
apply(*simp add*:*Let-def*) **apply** *metis*
apply(*rule subst*[**where** *s=max-sz* (*mem-pool-info V* (*pool b*)) **and** *t=ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))])
apply(*simp add*: *inv-def*) **using** *inv-mempool-info-maxsz-align4*[*rule-format*,*of V pool b*] **apply** *metis*

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*))) ! *level b*)) = (*n-max* (*mem-pool-info V* (*pool b*))) ∗ *4* ˆ (*level b*))
prefer *2* **apply**(*simp add*: *inv-def inv-mempool-info-def Let-def*)

apply(*subgoal-tac max-sz* (*mem-pool-info V* (*pool b*)) *mod 4* ˆ *lvl V t* = *0*)
prefer *2* **apply**(*subgoal-tac* ∃ *n. max-sz* (*mem-pool-info V* (*pool b*)) = (*4* ∗ *n*) ∗ (*4* ˆ *n-levels* (*mem-pool-info V* (*pool b*))))
prefer *2* **apply**(*simp add*:*inv-def*) **using** *inv-mempool-info-def*[*rule-format*, *of V*] **apply** *meson*
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*))) = *n-levels* (*mem-pool-info V* (*pool b*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def*) **apply** *metis*
apply(*simp add*: *inv-def inv-mempool-info-def*)
using *ge-pow-mod-0*[*of lvl V t n-levels* (*mem-pool-info V* (*pool b*))]
apply (*metis add-diff-inverse-nat add-lessD1 ge-pow-mod-0 le-antisym nat-less-le*)

apply(*subgoal-tac block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4* + *i vt t* < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *lvl V t*)
prefer *2* **apply**(*rule block-fits0-h2*[*of lvl V t level b i vt t block b n-max*

(*mem-pool-info V* (*pool b*))])
  apply *blast* **apply** *blast* **apply** *blast* **apply** *linarith*

 apply(*rule block-fits0-h1*[*of max-sz* (*mem-pool-info V* (*pool b*)) *4* ^ *lvl V t*
  block b div 4 ^ (*level b* − *lvl V t*) *div 4* ∗ *4* + *i vt t n-max* (*mem-pool-info V* (*pool b*))]])
  apply *blast* **apply** *blast*
done

lemma *block-fits1*:
 V ∈ *mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} =⇒
 vt ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* < *4*|} =⇒
 {*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} =⇒
 {*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*} ∩
      − {|*block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*)|} = {}
 using *block-fits0*[*of V t b* α *vt*] **apply** *fast*
done

lemma *mempool-free-stm8-set4partbits-while-one-h1*:
 ¬ *bn* (*free-stm8-precond33* (*free-stm8-precond32* (*vt*(|*bb* := (*bb vt*)(*t* := *bn vt t div 4* ∗ *4* + *i vt t*)|)) *t b*) *t b*) *t* ≠
   bb (*free-stm8-precond33* (*free-stm8-precond32* (*vt*(|*bb* := (*bb vt*)(*t* := *bn vt t div 4* ∗ *4* + *i vt t*)|)) *t b*) *t b*) *t* =⇒
  {*free-stm8-precond33* (*free-stm8-precond32* (*vt*(|*bb* := (*bb vt*)(*t* := *bn vt t div 4* ∗ *4* + *i vt t*)|)) *t b*) *t b*}
   ⊆ {|´*id* ∈ {*let vv* = *free-stm8-precond33* (*free-stm8-precond32* (*vt*(|*bb* := (*bb vt*)(*t* := *bn vt t div 4* ∗ *4* + *i vt t*)|)) *t b*) *t b*
      in if bn vv t = *bb vv t then vv else free-stm8-precond34 vv t b*}|}
 by(*simp add*:*Let-def*)

lemma *mempool-free-stm8-set4partbits-while-one-isuc-h1-1*:
∀ *p*. (∀ *i*. *length* (*bits* (*levels* (*mem-pool-info vt p*) ! *i*)) =
    length (*bits* (*levels* (*if p* = *pool b*
         then mem-pool-info V (*pool b*)
           (|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
            [*lvl vt t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*)
              (|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]|)]|)
         else mem-pool-info V p) !
       i))) =⇒
 ∀ *j*. *j* ≠ *lvl vt t* −→ *levels* (*mem-pool-info V* (*pool b*)) ! *j* = *levels* (*mem-pool-info vt* (*pool b*)) ! *j* =⇒
 length (*levels* (*mem-pool-info V* (*pool b*)))=*length* (*levels* (*mem-pool-info vt* (*pool b*))) =⇒
 length (*bits* ((*levels* (*mem-pool-info vt* (*pool b*)))

[*lvl vt t* := (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*)
(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*)
NOEXIST)
[*bn vt t* := *NOEXIST*]|)] !
ia)) =
length (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl vt t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]|)] !
ia))

apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply(*case-tac ia* = *lvl vt t*) **apply** *auto*[*1*]
apply (*metis* (*no-types*, *lifting*) *nth-list-update-neq*)
by (*smt list-eq-iff-nth-eq list-update-beyond not-less nth-list-update-neq*)

lemma *mempool-free-stm8-set4partbits-while-one-isuc-h1-2*:
¬ *free-block-r vt t* ⟶ *freeing-node V t* = *None* ⟹
free-block-r V t = *free-block-r vt t* ⟹
α = (*if* ∃ *y*. *freeing-node V t* = *Some y then lvl V t* + *1 else NULL*) ⟹
block b div 4 ˆ (*level b* − *lvl vt t*) = *bn vt t* ⟹
V ∈ (*if NULL* < (*if* ∃ *y*. *freeing-node V t* = *Some y then lvl V t* + *1 else NULL*) *then UNIV else* {}) ⟹
free-block-r V t
by *force*

lemma *mempool-free-stm8-set4partbits-while-one-isuc-h2*:
inv V ⟹
pool b ∈ *mem-pools V* ⟹
lvl vt t < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
bn vt t < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*)) ⟹
get-bit-s V (*pool b*) (*lvl vt t*) (*bn vt t*) ≠ *FREE* ⟹
buf (*mem-pool-info V* (*pool b*)) + *max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl vt t* ∗ *bn vt t*
∉ *set* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))
apply(*simp add*:*inv-def inv-bitmap-freelist-def Let-def*)
apply(*rule subst*[**where** *t*=*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl vt t* ∗ *bn vt t* **and**
s=*bn vt t* ∗ (*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl vt t*)])
apply *simp* **apply** *simp*
done

lemma *mempool-free-stm8-set4partbits-while-one-isuc-h1-3*:
∀ *p*. (∀ *i*. *length* (*bits* (*levels* (*mem-pool-info vt p*) ! *i*)) =
length (*bits* (*levels* (*if p* = *pool b*

then mem-pool-info V (pool b)
(|levels := (levels (mem-pool-info V (pool b)))
[lvl vt t := (levels (mem-pool-info V (pool b)) ! lvl vt t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl vt t))[bn vt t := FREE]|)]|)
else mem-pool-info V p) !
i))) ⟹
∀ j. j ≠ lvl vt t ⟶ levels (mem-pool-info V (pool b)) ! j = levels (mem-pool-info vt (pool b)) ! j ⟹
length (levels (mem-pool-info V (pool b)))=length (levels (mem-pool-info vt (pool b))) ⟹
length (bits ((levels (mem-pool-info vt (pool b)))
[lvl vt t := (levels (mem-pool-info vt (pool b)) ! lvl vt t)
(|bits := (list-updates-n ((bits (levels (mem-pool-info V (pool b)) ! lvl vt t))[bn vt t := FREE]) (bn vt t div 4 ∗ 4) (i vt t)
NOEXIST)
[bn vt t div 4 ∗ 4 + i vt t := NOEXIST],
free-list :=
remove1 (block-ptr
(mem-pool-info vt (pool b)
(|levels := (levels (mem-pool-info vt (pool b)))
[lvl vt t := (levels (mem-pool-info vt (pool b)) ! lvl vt t)
(|bits := (list-updates-n ((bits (levels (mem-pool-info V (pool b)) ! lvl vt t))[bn vt t := FREE])
(bn vt t div 4 ∗ 4) (i vt t) NOEXIST)
[bn vt t div 4 ∗ 4 + i vt t := NOEXIST]|)]|))
(lsz vt t) (bn vt t div 4 ∗ 4 + i vt t))
(removes (map (λii. block-ptr
(mem-pool-info V (pool b)
(|levels := (levels (mem-pool-info V (pool b)))
[lvl vt t := (levels (mem-pool-info V (pool b)) ! lvl vt t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl vt t))[bn vt t := FREE]|)]|))
(lsz vt t) (bn vt t div 4 ∗ 4 + ii))
[NULL..<i vt t])
(free-list (levels (mem-pool-info V (pool b)) ! lvl vt t)))|)] !
ia)) =
length (bits ((levels (mem-pool-info V (pool b)))
[lvl vt t := (levels (mem-pool-info V (pool b)) ! lvl vt t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl vt t))[bn vt t := FREE]|)] !
ia))

apply(*case-tac ia < length (levels (mem-pool-info V (pool b)))*)
apply(*case-tac ia = lvl vt t*) **apply** *auto[1]*
apply (*metis (no-types, lifting) nth-list-update-neq*)

by (*smt list-eq-iff-nth-eq list-update-beyond not-less nth-list-update-neq*)

lemma *mempool-free-stm8-set4partbits-while-one-isuc-h1*:
V ∈ *mp-free-precond8-3 t b α* ∩ {| ´*cur* = *Some t* |} ⟹
vt ∈ *free-stm8-precond3 V t b* ∩ {| ´*i t* < *4* |} ⟹
{*free-stm8-precond2 V t b*} ∩ {| *NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*) |} ≠ {} ⟹
{*let vv* = *free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b in*
if bn vv t = *bb vv t then vv else free-stm8-precond34 vv t b*}
⊆ {*s*. *s*(| *i* := (*i s*) (*t* := *Suc* (*i s t*)) |) ∈ *free-stm8-precond3 V t b*}
apply(*simp add*:*Let-def set-bit-def*)

apply(*rule conjI*)
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply *clarsimp*
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*)))=*length* (*levels* (*mem-pool-info vt* (*pool b*))))
prefer *2* **apply** *simp*
using *mempool-free-stm8-set4partbits-while-one-isuc-h1-1* **apply** *blast*
apply(*rule conjI*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*)))=*length* (*levels* (*mem-pool-info vt* (*pool b*))))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)

apply(*rule subst*[**where** *s*=(*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*) *NOEXIST*)
[*bn vt t* := *NOEXIST*] **and** *t*=*bits* ((*levels* (*mem-pool-info vt* (*pool b*)))
[*lvl vt t* := (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*)
(| *bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*) *NOEXIST*)
[*bn vt t* := *NOEXIST*] |)] !
lvl vt t)]) **apply** *auto*[*1*]
using *lst-updts-eq-updts-updt*[*of Suc* (*i vt t*) (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]
bn vt t div 4 ∗ *4 NOEXIST*] **apply** *auto*[*1*]
apply(*rule conjI*)
apply(*simp add*:*block-ptr-def*)
apply(*rule subst*[**where** *s*=*free-list* (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*) **and**
t=*free-list* ((*levels* (*mem-pool-info vt* (*pool b*)))
[*lvl vt t* := (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*)
(| *bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*) *NOEXIST*)
[*bn vt t* := *NOEXIST*] |)] ! *lvl vt t*)])
apply(*case-tac lvl vt t* < *length* (*levels* (*mem-pool-info vt* (*pool b*)))) **apply**

auto[*1*] **apply** *auto*[*1*]
apply(*subgoal-tac removes* (*map* (*λii. buf* (*mem-pool-info V* (*pool b*)) + *lsz vt t* ∗ (*bn vt t div 4* ∗ *4* + *ii*)) [*NULL..*<*i vt t*] @
[*buf* (*mem-pool-info V* (*pool b*)) + *lsz vt t* ∗ *bn vt t*])
(*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*)) =
removes (*map* (*λii. buf* (*mem-pool-info V* (*pool b*)) + *lsz vt t* ∗ (*bn vt t div 4* ∗ *4* + *ii*)) [*NULL..*<*i vt t*])
(*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))) **apply** *metis*
apply(*rule rmvs-onemore-same*)
apply(*simp add*:*inv-def inv-bitmap-freelist-def Let-def*)
apply(*subgoal-tac get-bit* (*mem-pool-info V*) (*pool b*) (*lvl vt t*) (*bn vt t*) = *FREEING*) **prefer** *2*
apply(*subgoal-tac free-block-r V t*) **prefer** *2*
using *mempool-free-stm8-set4partbits-while-one-isuc-h1-2* **apply** *blast*
apply(*subgoal-tac* ∃ *blk. freeing-node V t* = *Some blk* ∧ *pool blk* = *pool b* ∧ *level blk* = *lvl vt t* ∧ *block blk* = *bn vt t*)
prefer *2* **apply** *fast*
apply(*simp add*:*inv-def inv-aux-vars-def*) **apply** *metis*
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl vt t* **and** *t*=*lsz vt t*])
using *inv-maxsz-align4*[*rule-format*, *of V pool b*] **apply** *force*
apply(*subgoal-tac get-bit* (*mem-pool-info V*) (*pool b*) (*lvl vt t*) (*bn vt t*) ≠ *FREE*) **prefer** *2* **apply** *auto*[*1*]
apply(*subgoal-tac lvl vt t* < *length* (*levels* (*mem-pool-info V* (*pool b*)))) **prefer** *2* **apply** *force*
using *mempool-free-stm8-set4partbits-while-one-isuc-h2* **apply** *blast*

apply *clarsimp* **apply**(*simp add*:*block-ptr-def lvars-nochange-def*)

apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply *clarsimp*
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*)))=*length* (*levels* (*mem-pool-info vt* (*pool b*))))
prefer *2* **apply** *simp*
using *mempool-free-stm8-set4partbits-while-one-isuc-h1-3* **apply** *blast*
apply(*rule conjI*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*)))=*length* (*levels* (*mem-pool-info vt* (*pool b*))))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*rule subst*[**where** *s*=(*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*) *NOEXIST*)
[*bn vt t div 4* ∗ *4* + *i vt t* := *NOEXIST*] **and** *t*=*bits* ((*levels* (*mem-pool-info vt* (*pool b*)))
[*lvl vt t* :=
((*levels* (*mem-pool-info vt* (*pool b*)))
[*lvl vt t* := (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*)

(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *vt* *t*))[*bn* *vt* *t* := *FREE*]) (*bn* *vt* *t* *div* *4* * *4*) (*i* *vt* *t*) *NOEXIST*)
[*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t* := *NOEXIST*]|)] !
lvl *vt* *t*)
(|*free-list* :=
remove1 (*block-ptr*
(*mem-pool-info* *vt* (*pool* *b*)
(|*levels* := (*levels* (*mem-pool-info* *vt* (*pool* *b*)))
[*lvl* *vt* *t* := (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*)
(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *vt* *t*))[*bn* *vt* *t* := *FREE*]) (*bn* *vt* *t* *div* *4* * *4*) (*i* *vt* *t*)
NOEXIST)
[*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t* := *NOEXIST*]|)]|))
(*lsz* *vt* *t*) (*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t*))
(*free-list*
((*levels* (*mem-pool-info* *vt* (*pool* *b*)))
[*lvl* *vt* *t* := (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*)
(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *vt* *t*))[*bn* *vt* *t* := *FREE*]) (*bn* *vt* *t* *div* *4* * *4*) (*i* *vt* *t*) *NOEXIST*)
[*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t* := *NOEXIST*]|)] !
lvl *vt* *t*))|)] !
lvl *vt* *t*)]) **apply** *auto*[*1*]
using *lst-updts-eq-updts-updt*[*of Suc* (*i* *vt* *t*) (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *vt* *t*))[*bn* *vt* *t* := *FREE*]
bn *vt* *t* *div* *4* * *4* *NOEXIST*] **apply** *auto*[*1*]

apply(*rule conjI*)
apply(*simp add*:*block-ptr-def*)
apply(*subgoal-tac lvl vt t* < *length* (*levels* (*mem-pool-info* *vt* (*pool* *b*)))) **prefer** *2*
apply(*subgoal-tac length* (*levels* (*mem-pool-info* *V* (*pool* *b*)))=*length* (*levels* (*mem-pool-info* *vt* (*pool* *b*))))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply *force*
apply(*rule subst*[**where** *s*=*free-list* (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*) **and**
t=*free-list* ((*levels* (*mem-pool-info* *vt* (*pool* *b*)))
[*lvl* *vt* *t* := (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*)
(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *vt* *t*))[*bn* *vt* *t* := *FREE*]) (*bn* *vt* *t* *div* *4* * *4*) (*i* *vt* *t*) *NOEXIST*)
[*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t* := *NOEXIST*]|)] ! *lvl* *vt* *t*)])
apply *auto*[*1*]
apply(*rule subst*[**where** *s*=*remove1* (*buf* (*mem-pool-info* *vt* (*pool* *b*)) + *lsz* *vt* *t* * (*bn* *vt* *t* *div* *4* * *4* + *i* *vt* *t*))
(*free-list* (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*)) **and**
t=*free-list* ((*levels* (*mem-pool-info* *vt* (*pool* *b*)))
[*lvl* *vt* *t* :=
((*levels* (*mem-pool-info* *vt* (*pool* *b*)))
[*lvl* *vt* *t* := (*levels* (*mem-pool-info* *vt* (*pool* *b*)) ! *lvl* *vt* *t*)

(|*bits* := (*list-updates-n* ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*))[*bn vt t* := *FREE*]) (*bn vt t div 4* ∗ *4*) (*i vt t*) *NOEXIST*)
[*bn vt t div 4* ∗ *4* + *i vt t* := *NOEXIST*]|)] !
lvl vt t)
(|*free-list* :=
remove1 (*buf* (*mem-pool-info vt* (*pool b*)) + *lsz vt t* ∗ (*bn vt t div 4* ∗ *4* + *i vt t*))
(*free-list* (*levels* (*mem-pool-info vt* (*pool b*)) ! *lvl vt t*))|)] !
lvl vt t)])
apply *auto*[*1*]
apply(*subgoal-tac buf* (*mem-pool-info vt* (*pool b*)) = *buf* (*mem-pool-info V* (*pool b*))) **prefer** *2*
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
using *rmvs-rev*[*of* (*map* (λ*ii*. *buf* (*mem-pool-info V* (*pool b*)) + *lsz vt t* ∗ (*bn vt t div 4* ∗ *4* + *ii*)) [*NULL*..<*i vt t*])
buf (*mem-pool-info V* (*pool b*)) + *lsz vt t* ∗ (*bn vt t div 4* ∗ *4* + *i vt t*)
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl vt t*)] **apply** *simp*

apply *clarsimp* **apply**(*simp add*:*block-ptr-def lvars-nochange-def*)
done

lemma *mempool-free-stm8-set4partbits-while-one-isuc*:
V = *Va* ⟹
V ∈ *mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} ⟹
vt ∈ *free-stm8-precond3 Va t b* ∩ {|´*i t* < *4*|} ⟹
{*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} ⟹
Γ $\vdash_I$ *Some* (´*i* := ´*i*(*t* := *Suc* (´*i t*)))
sat_p [{*let vv* = *free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b in*
if bn vv t = *bb vv t then vv else free-stm8-precond34 vv t b*}, {(*s*, *t*). *s* = *t*}, *UNIV*, *free-stm8-precond3 Va t b*]
apply(*rule Basic*)
defer *1*
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

using *mempool-free-stm8-set4partbits-while-one-isuc-h1*[*of Va t b α vt*] **apply** *fast*
done

lemma *mempool-free-stm8-set4partbits-while-one*:
V = *Va* ⟹
V ∈ *mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} ⟹
vt ∈ *free-stm8-precond3 Va t b* ∩ {|´*i t* < *4*|} ⟹
{*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} ⟹

$\Gamma \vdash_I$ *Some* (´*bb* := ´*bb*(*t* := ´*bn t div 4* ∗ *4* + ´*i t*);;
´*mem-pool-info* := *set-bit-noexist* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bb t*);;
´*block-pt* := ´*block-pt* (*t* := *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bb t*));;
IF ´*bn t* ≠ ´*bb t* ∧ *block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *remove-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*block-pt t*))
FI;;
´*i* := ´*i*(*t* := *Suc* (´*i t*)))
sat_p [{*vt*}, {(*s*, *t*). *s* = *t*}, *UNIV*, *free-stm8-precond3 Va t b*]
apply(*rule Seq*[**where** *mid*={*let vv* = *free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b in*
if bn vv t = *bb vv t then vv else free-stm8-precond34 vv t b*}])
apply(*rule Seq*[**where** *mid*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*}])
apply(*rule Seq*[**where** *mid*={*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*}])
apply(*rule Seq*[**where** *mid*={*free-stm8-precond31 vt t b*}])

apply(*rule Basic*)
apply *fast* **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply(*simp add*:*Let-def set-bit-def*)
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply *fast* **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Cond*)
using *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply(*simp add*:*Let-def remove-free-list-def*) **apply** *auto*[*1*]
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*case-tac bn* (*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*) *t*
≠ *bb* (*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*) *t*)
apply(*rule subst*[**where** *s*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31

vt t b) *t b*) *t b*} ∩
− {|*block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*)|}
and *t*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*} ∩
− {|´*bn t* ≠ ´*bb t* ∧ *block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*)|}]) **apply** *fast*
apply(*rule subst*[**where** *s*={} **and** *t*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*} ∩
− {|*block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*)|}])
using *block-fits1*[*of V t b α vt*] **apply** *fast*
using *Emptyprecond* **apply** *fast*

apply(*rule subst*[**where** *s*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*}
and *t*={*free-stm8-precond33* (*free-stm8-precond32* (*free-stm8-precond31 vt t b*) *t b*) *t b*} ∩
− {|´*bn t* ≠ ´*bb t* ∧ *block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*)|}])
apply *fast*

apply(*unfold Skip-def*)
apply(*rule Basic*)
using *mempool-free-stm8-set4partbits-while-one-h1* **apply** *fast*
apply *fast*
using *stable-id2* **apply** *fast*
using *stable-id2* **apply** *fast*

apply *fast*

using *mempool-free-stm8-set4partbits-while-one-isuc*[*of V Va t b α vt*] **apply** *fast*
done

lemma *mempool-free-stm8-set4partbits-while*:
V = *Va* ⟹
V∈*mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} ⟹
{*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} ⟹
$\Gamma \vdash_I$ *Some*(´*bb* := ´*bb*(*t* := ´*bn t div 4* ∗ *4* + ´*i t*);;
´*mem-pool-info* := *set-bit-noexist* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bb t*);;
´*block-pt* := ´*block-pt* (*t* := *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bb t*));;
IF ´*bn t* ≠ ´*bb t* ∧ *block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *remove-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*block-pt t*))
FI;;
´*i* := ´*i*(*t* := *Suc* (´*i t*)))

sat$_p$ [*free-stm8-precond3 Va t b* ∩ {|´*i t* < *4*|}, {(*s*, *t*). *s* = *t*}, *UNIV*, *free-stm8-precond3 Va t b*]
using *mempool-free-stm8-set4partbits-while-one*[*of V Va t b α*]
Allprecond[**where** *U*=*free-stm8-precond3 Va t b* ∩ {|´*i t* < *4*|} **and**
P=*Some* (´*bb* := ´*bb*(*t* := ´*bn t div 4* ∗ *4* + ´*i t*);;
´*mem-pool-info* := *set-bit-noexist* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bb t*);;
´*block-pt* := ´*block-pt* (*t* := *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bb t*));;
IF ´*bn t* ≠ ´*bb t* ∧ *block-fits* (´*mem-pool-info* (*pool b*)) (´*block-pt t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *remove-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*block-pt t*))
FI;;
´*i* := ´*i*(*t* := *Suc* (´*i t*))) **and**
rely={(*x*, *y*). *x* = *y*} **and**
guar= *UNIV* **and** *post*= *free-stm8-precond3 Va t b*]
apply *meson*
done

term *free-stm8-precond3 Va t b*

abbreviation *free-stm8-atombody-rest-cond1 V t b* ≡ *V*(|*lvl* := (*lvl V*)(*t* := *lvl V t* − *1*)|)
abbreviation *free-stm8-atombody-rest-cond2 V t b* ≡ *V*(|*bn* := (*bn V*)(*t* := *bn V t div 4*)|)
abbreviation *free-stm8-atombody-rest-cond3 V t b* ≡
let minf = *mem-pool-info V* (*pool b*) *in*
V(|*mem-pool-info*:= (*mem-pool-info V*) (*pool b* := *minf* (|*levels* := (*levels minf*)
[*lvl V t* := ((*levels minf*) ! (*lvl V t*)) (|*bits* := (*bits* ((*levels minf*) ! (*lvl V t*))) [*bn V t* := *FREEING*]|)] |))|)

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-cur*:
V∈*mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} ⟹
~~(* {free-stm8-precond2 V t b} ∩ {NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)} ≠ {} ⟹ *)~~
V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* ⟹
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)|))|) ⟹
inv-cur y
apply(*rule subst*[**where** *s*=*inv-cur x* **and** *t*=*inv-cur y*])
apply(*simp add*:*block-ptr-def inv-cur-def*)

apply(*simp add*:*Let-def inv-def inv-cur-def*)
apply(*subgoal-tac thd-state V t = RUNNING*) **prefer** *2* **apply** *fast*
apply *clarsimp*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-thd-waitq*:
  V∈*mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} ⟹
  ~~(* {free-stm8-precond2 V t b} ∩ {NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)} # {} ⟹ *)~~
  V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} ⟹
  x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* ⟹
  y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
                    data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ^ *lvl x t*) (*bn x t*)|))|) ⟹
  inv-thd-waitq y
apply(*rule subst*[**where** *s*=*inv-thd-waitq x* **and** *t*=*inv-thd-waitq y*])
apply(*simp add*:*block-ptr-def inv-thd-waitq-def*)

apply(*simp add*:*Let-def inv-def inv-thd-waitq-def*)
apply(*simp add*:*set-bit-def*)
apply(*subgoal-tac mem-pools V = mem-pools V2*)
  prefer *2* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)

apply(*rule conjI*)
  apply *clarify* **apply** *metis*

apply(*rule conjI*)
  apply *clarify* **apply** *metis*

apply(*rule conjI*)
  apply *clarify* **apply** *metis*
apply *metis*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h1*:
∀ *p*. *buf* (*mem-pool-info V2 p*) =
    buf (*if p* = *pool b*
      then mem-pool-info V (*pool b*)
        (|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
          [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
      else mem-pool-info V p) ∧
    max-sz (*mem-pool-info V2 p*) =
    max-sz (*if p* = *pool b*
      then mem-pool-info V (*pool b*)

(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block*
b div 4 ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p) ∧
n-max (*mem-pool-info V2 p*) =
n-max (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block*
b div 4 ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p) ∧
n-levels (*mem-pool-info V2 p*) =
n-levels (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V*
t))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p) ∧
length (*levels* (*mem-pool-info V2 p*)) =
length (*levels* (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V*
t))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p)) ∧
(∀ *i*. *length* (*bits* (*levels* (*mem-pool-info V2 p*) ! *i*)) =
length (*bits* (*levels* (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) !
lvl V t)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool*
b)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p) !
i))) ⟹
ia < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)) = *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*))
apply *auto*
apply(*case-tac lvl V t* = *ia*) **apply** *auto*[*1*] **apply** *auto*[*1*]
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h2*:
ia < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹

length (bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
ia)) = length (bits (levels (mem-pool-info V2 (pool b)) ! ia))
apply(*case-tac lvl V2 t − Suc NULL = ia*)
apply(*case-tac ia < length (levels (mem-pool-info V2 (pool b))))* **apply** *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h3*:
mem-pools V = mem-pools V2 ⟹
p ∈ mem-pools V2 ⟹
∀ p∈mem-pools V2.
NULL < buf (mem-pool-info V p) ∧
(∃ n>NULL. max-sz (mem-pool-info V p) = 4 ∗ n ∗ 4 ˆ n-levels (mem-pool-info V p)) ∧
NULL < n-max (mem-pool-info V p) ∧
NULL < n-levels (mem-pool-info V p) ∧
n-levels (mem-pool-info V p) = length (levels (mem-pool-info V p)) ∧
(∀ i<length (levels (mem-pool-info V p)). length (bits (levels (mem-pool-info V p) ! i)) = n-max (mem-pool-info V p) ∗ 4 ˆ i) ⟹
mem-pools V = mem-pools V2 ⟹
pool b ∈ mem-pools V2 ⟹ levels (mem-pool-info V p) ≠ []
apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h1-1′*:
∀ j. j ≠ lvl V t ⟶
(levels (mem-pool-info V (pool b)))
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ˆ (level b − lvl V t) := FREE]|)] !
j =
levels (mem-pool-info V2 (pool b)) ! j ⟹
bits (levels (mem-pool-info V2 (pool b)) ! lvl V t) =
list-updates-n
(bits ((levels (mem-pool-info V (pool b)))
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ˆ (level b − lvl V t) := FREE]|)] !
lvl V t))
(block b div 4 ˆ (level b − lvl V t) div 4 ∗ 4) 4 NOEXIST ⟹
length (bits (levels (mem-pool-info V (pool b)) ! ia)) =
length (bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !

ia))
apply(*rule subst*[**where** *s=length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*))!*ia*))
and *t=length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc 0* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*)))[*bn V2 t div 4* := *FREEING*]|)] !
ia))])
apply(*case-tac ia* = *lvl V2 t* − *Suc 0*)
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V2* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]

apply(*case-tac ia* = *lvl V t*)
apply(*subgoal-tac length* (*list-updates-n*
(*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia))
(*block b div 4* ^ (*level b* − *lvl V t*) *div 4* ∗ *4*) *4 NOEXIST*) = *length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)))
prefer *2* **using** *length-list-update-n* **apply** *fast*
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*)
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*) **apply**(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply(*subgoal-tac* (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !

    ia = *levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)
  prefer *2* **apply** *fast*
 apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h1-1*:
∀ *j*. *j* ≠ *lvl V t* ⟶
   levels (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) ! *j*
   = *levels* (*mem-pool-info V2* (*pool b*)) ! *j* ⟹
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) =
list-updates-n (*bits* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*)
  (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) ! *lvl V t*))
 (*block b div 4* ^ (*level b* − *lvl V t*) *div 4* ∗ *4*) *4 NOEXIST* ⟹
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)) =
length (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
      [*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
       ⦇*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]⦈] !
      ia))
apply(*rule subst*[**where** *s*=*length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*))!*ia*)) **and** *t*=*length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
        [*lvl V2 t* − *Suc 0* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*))
         ⦇*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*)))[*bn V2 t div 4* := *FREEING*]⦈] !
        ia))])
 apply(*case-tac ia* = *lvl V2 t* − *Suc 0*)
  apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V2* (*pool b*))))
   apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]

apply(*simp add*:*set-bit-def*)
apply(*case-tac ia* = *lvl V t*)
 apply(*subgoal-tac length* (*list-updates-n*
   (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
     [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
      ⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] !
     ia))
   (*block b div 4* ^ (*level b* − *lvl V t*) *div 4* ∗ *4*) *4 NOEXIST*) = *length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
     [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
      ⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] !
     ia)))
  prefer *2* **using** *length-list-update-n* **apply** *fast*
 apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
      [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)

(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*)
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*) **apply**(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply(*subgoal-tac* (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
ia = *levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)
prefer *2* **apply** *auto*[*1*]
apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info*:
V ∈ *mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ⟹
~~(* {free-stm8-precond2 V t b} ∩ {NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)} ≠ {} ⟹ *)~~
V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* ⟹
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ^ *lvl x t*) (*bn x t*)|))|) ⟹
inv-mempool-info y
apply(*rule subst*[**where** *s*=*inv-mempool-info x* **and** *t*=*inv-mempool-info y*])
apply(*simp add*:*block-ptr-def inv-mempool-info-def*)

apply(*simp add*:*Let-def inv-def inv-mempool-info-def*)
apply(*simp add*:*set-bit-def*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)

apply(*subgoal-tac mem-pools V* = *mem-pools V2*)
prefer *2* **apply** *simp*

apply *clarify*

apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*))) > *0*) **prefer** *2*
apply *metis* **apply** *fast*
apply *clarify*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*))
= *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)))
prefer *2* **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h1*
apply *blast*
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) !
(*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* −
Suc NULL)))
[*bn V2 t div 4* := *FREEING*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)))
prefer *2* **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h2*
apply *blast*
apply *metis*

apply *clarify*
apply(*rule conjI*)
apply *metis*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h3* **apply** *blast*
done

lemma *free-stm8-atombody-rest-one-finalstm-VV2-len*:
∀ *p*. *length* (*levels* (*mem-pool-info V2 p*)) =
length (*levels* (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) !
lvl V t))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p)) ⟹
length (*levels* (*mem-pool-info V p*)) = *length* (*levels* (*mem-pool-info V2 p*))
by *auto*

lemma *free-stm8-atombody-rest-one-finalstm-bits-len*:
lvl V t = *lvl V2 t* ⟹
p = *pool b* ⟹
length (*bits* (*levels* (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)

(|*bits* := (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V* *t*))
[*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) := *FREE*]|)]|)
else *mem-pool-info* *V* *p*) !
(*lvl* *V2* *t*))) = *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
apply(*case-tac* *lvl* *V2* *t* < *length* (*levels* (*mem-pool-info* *V* (*pool* *b*))))
apply *auto*
done

lemma *free-stm8-atombody-rest-one-finalstm-ltlen*:
lvl *V2* *t* > *0* ⟹
lvl *V2* *t* = *lvl* *V* *t* ⟹
length (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
= (*n-max* (*mem-pool-info* *V* (*pool* *b*))) ∗ *4* ^ *lvl* *V2* *t* ⟹
block *b* *div* *4* ^ (*level* *b* − *lvl* *V2* *t*) < *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*)) ⟹
block *b* *div* *4* ^ (*level* *b* − *lvl* *V2* *t*) *div* *4* ∗ *4* + *4*
≤ *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
apply(*rule* *divn-multn-addn-le*[*of* *4* *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
block *b* *div* *4* ^ (*level* *b* − *lvl* *V2* *t*)])
apply *simp* **apply** *simp* **apply** *simp*
done

lemma *free-stm8-atombody-rest-one-finalstm-jj*:
lvl *V2* *t* > *0* ⟹
lvl *V2* *t* = *lvl* *V* *t* ⟹
length (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
= (*n-max* (*mem-pool-info* *V* (*pool* *b*))) ∗ *4* ^ *lvl* *V2* *t* ⟹
block *b* *div* *4* ^ (*level* *b* − *lvl* *V2* *t*) < *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*)) ⟹
jj ∈ {*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) *div* *4* ∗ *4* ..<
block *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) *div* *4* ∗ *4* + *4*} ⟹
jj < *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*))
apply *clarsimp*
apply(*subgoal-tac* *block* *b* *div* *4* ^ (*level* *b* − *lvl* *V2* *t*) *div* *4* ∗ *4* + *4*
≤ *length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V2* *t*)))
prefer *2*
apply(*rule* *free-stm8-atombody-rest-one-finalstm-ltlen*)
apply *simp*+
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap′-h1*:
bits (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! *lvl* *V* *t*) =
list-updates-n
(*bits* ((*levels* (*mem-pool-info* *V* (*pool* *b*)))
[*lvl* *V* *t* := (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V* *t*)

(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ^ (level b − lvl V t) := FREE]|)] !
lvl V t))
(block b div 4 ^ (level b − lvl V t) div 4 * 4) (i V2 t) NOEXIST ⟹
i V2 t = 4 ⟹
level b < length (levels (mem-pool-info V (pool b))) ⟹
lvl V t = lvl V2 t ⟹
lvl V2 t ≤ level b ⟹
ia = lvl V t ⟹
ia > 0 ⟹
p = pool b ⟹
block b div 4 ^ (level b − lvl V2 t) < length (bits (levels (mem-pool-info V (pool b)) ! lvl V2 t)) ⟹
length (bits (levels (mem-pool-info V (pool b)) ! lvl V2 t))
= (n-max (mem-pool-info V (pool b))) * 4 ^ lvl V2 t ⟹
length (levels (mem-pool-info V (pool b)))=length (levels (mem-pool-info V2 (pool b))) ⟹
jj ∈ {block b div 4 ^ (level b − lvl V t) div 4 * 4 ..<
block b div 4 ^ (level b − lvl V t) div 4 * 4 + 4} ⟹
get-bit-s (V2(|mem-pool-info := (mem-pool-info V2)
(pool b := mem-pool-info V2 (pool b)
(|levels := (levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))
[bn V2 t div 4 := FREEING]|)]|))|)) p ia jj = NOEXIST
apply(*rule subst*[**where** *s=get-bit-s V2 p ia jj*])
apply(*subgoal-tac list-updates-n*
(bits ((levels (mem-pool-info V (pool b)))
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ^ (level b − lvl V t) := FREE]|)] !
lvl V t))
(block b div 4 ^ (level b − lvl V t) div 4 * 4) (i V2 t) NOEXIST !
jj =
NOEXIST) **prefer** *2*
apply(*rule list-updates-n-eq*[*of block b div 4 ^ (level b − lvl V t) div 4 * 4 jj*
bits ((levels (mem-pool-info V (pool b)))
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ^ (level b − lvl V t) := FREE]|)] !
lvl V t) i V2 t NOEXIST])
apply *fastforce*
apply(*rule subst*[**where** *s=length (bits (levels (mem-pool-info V (pool b)) ! lvl V t))*
and *t=length (bits ((levels (mem-pool-info V (pool b)))*
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))

[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] !
lvl V t))]) **apply** *force*
apply(*rule free-stm8-atombody-rest-one-finalstm-jj*)
apply *fast* **apply** *fast* **apply** *presburger* **apply** *argo*
apply *fast* **apply** *force*

apply *argo*
by *fastforce*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap′-h2*:
get-bit-s
(*V2*⦇*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]⦈]⦈)⦈)
p ia jj =
FREE ∨
get-bit-s
(*V2*⦇*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]⦈]⦈)⦈)
p ia jj =
FREEING ∨
get-bit-s
(*V2*⦇*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]⦈]⦈)⦈)
p ia jj =
ALLOCATED ∨
get-bit-s
(*V2*⦇*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
⦇*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl*

V2 t − Suc NULL))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]|)]|))|))
p ia jj =
ALLOCATING ⟹
get-bit-s
(*V2*(|*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]|)]|))|))
p ia jj =
NOEXIST ⟹
get-bit-s
(*V2*(|*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]|)]|))|))
p (*ia* − *1*) (*jj div 4*) =
DIVIDED
by *force*

axiomatization where *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap*:
V∈*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ⟹
~~(* {free-stm8-precond2 V t b} ∩ {NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)} ≠ {} ⟹ *)~~
V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* ⟹
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ^ *lvl x t*) (*bn x t*)|))|) ⟹
inv-bitmap y

axiomatization where *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap-freelist*:
V∈*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ⟹
~~(* {free-stm8-precond2 V t b} ∩ {NULL < ´lvl t ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)} ≠ {} ⟹ *)~~

V2 ∈ *free-stm8-precond3 V t b* ∩ {´*i t* = *4*} ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* ⟹
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)|)|) ⟹
inv-bitmap-freelist y

lemma *mempool-free-stm8-atombody-rest-one-finalstm-len-bits1*:
∀ *j*. *j* ≠ *lvl V t* ⟶ *levels* (*mem-pool-info V* (*pool b*)) ! *j* = *levels* (*mem-pool-info V2* (*pool b*)) ! *j* ⟹
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) =
list-updates-n
(*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)] !
lvl V t))
(*block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4*) (*i V2 t*) *NOEXIST* ⟹
(*i V2 t*) = *4* ⟹
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
= *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
apply(*rule subst*[**where** *s*=*length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
(*lvl V2 t* − *Suc NULL*))) **and** *t*=*length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))])
using *mempool-free-stm8-atombody-rest-one-finalstm-h1-1*′[*of V t b V2 lvl V2 t* − *Suc NULL*] **apply** *auto*[*1*]
apply(*case-tac lvl V2 t* − *Suc NULL* < *length* (*levels* (*mem-pool-info V2* (*pool b*))))
apply *auto*
done

lemma *lm11*:
lvl V2 t ≤ *level b* ∧ *level b* > *0* ∧ *level b* < *length* (*levels* (*mem-pool-info V* (*pool b*))) ∧
0 < *lvl V2 t* ⟹
block b < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b* ⟹
block b div 4 ˆ (*level b* − *lvl V2 t*) = *bn V2 t* ⟹
bn V2 t div 4 < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ (*lvl V2 t* − *Suc NULL*)
apply(*rule subst*[**where** *s*=*block b div 4* ˆ (*level b* − *lvl V2 t*) *div 4* **and** *t*=*bn V2 t div 4*])
apply *simp*

apply(*rule subst*[**where** *s=n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b div 4* ˆ (*level b* − *lvl V2 t*) *div 4*
and *t=n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ (*lvl V2 t* − *Suc NULL*)])
apply (*smt Groups.mult-ac*(*2*) *Groups.mult-ac*(*3*) *One-nat-def add-diff-cancel-left′ div-mult-self1-is-m*
le-Suc-ex power-add power-minus-mult zero-less-numeral zero-less-power)

apply(*subgoal-tac n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b mod 4* ˆ (*level b* − *lvl V2 t*) = *0*)
prefer *2* **using** *mod-minus-0*[*of lvl V2 t level b n-max* (*mem-pool-info V* (*pool b*))] **apply** *fast*

apply(*subgoal-tac block b div 4* ˆ (*level b* − *lvl V2 t*) < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b div 4* ˆ (*level b* − *lvl V2 t*))
prefer *2* **using** *mod-div-gt*[*of block b n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b 4* ˆ (*level b* − *lvl V2 t*)]
apply *fast*
apply(*subgoal-tac n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b div 4* ˆ (*level b* − *lvl V2 t*) *mod 4* = *0*)
prefer *2* **using** *mod-minus-div-4*[*of lvl V2 t level b n-max* (*mem-pool-info V* (*pool b*))] **apply** *fast*

using *mod-div-gt*[*of block b div 4* ˆ (*level b* − *lvl V2 t*)
n-max (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *level b div 4* ˆ (*level b* − *lvl V2 t*) *4*]
apply *fast*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-aux-vars-h2*:
pool b ∈ *mem-pools V* ⟹
inv-mempool-info V ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b div 4 ˆ (*level b* − *lvl V t*) = *bn V2 t* ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ⟹
lvl V t = *lvl V2 t* ⟹
lvl V2 t ≤ *level b* ⟹
0 < *lvl V2 t* ⟹
length (*levels* (*mem-pool-info V* (*pool b*))) = *length* (*levels* (*mem-pool-info V2* (*pool b*))) ⟹
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
= *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))) ⟹
bits ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !

(*lvl V2 t* − *Suc NULL*)) !
(*bn V2 t div 4*) =
FREEING
apply(*subgoal-tac lvl V2 t* − *Suc 0* < *length* (*levels* (*mem-pool-info V2* (*pool b*))))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac bn V2 t div 4* < *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))))
prefer *2*

apply(*subgoal-tac level b* > *0*) **prefer** *2* **apply** *auto*[*1*]
apply(*subgoal-tac n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ (*lvl V2 t* − *Suc NULL*)
= *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))))
prefer *2* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
apply (*metis inv-mempool-info-def lm11*)
apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-len-lvls*:
(*V2*, *V*(|*mem-pool-info* := (*mem-pool-info V*)
(*pool b* := *mem-pool-info V* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V2 t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V2 t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V2 t*))[*bn V2 t* := *FREE*]|)]|)),
freeing-node := (*freeing-node V*)(*t* := *None*)|))
∈ *gvars-conf-stable* =⇒
length (*levels* (*mem-pool-info V2* (*pool b*))) = *length* (*levels* (*mem-pool-info V* (*pool b*)))
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
done

axiomatization where *mempool-free-stm8-atombody-rest-one-finalstm-inv-aux-vars*:
V∈*mp-free-precond8-3 t b α* ∩ {|´*cur* = *Some t*|} =⇒
{*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} =⇒
V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} =⇒
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* =⇒
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)|))|) =⇒
inv-aux-vars y

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0-case1-h1*:
NULL < length (levels (mem-pool-info V2 (pool b))) ⟹
(bits (levels (mem-pool-info V2 (pool b)) ! NULL))[ia := FREEING] =
bits ((levels (mem-pool-info V2 (pool b)))
[NULL := (levels (mem-pool-info V2 (pool b)) ! NULL)
⦇bits := (bits (levels (mem-pool-info V2 (pool b)) ! NULL))[ia := FREEING]⦈] !
NULL)
by *auto*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0-case1*:
pool b ∈ mem-pools V2 ⟹
inv V ⟹
NULL < lvl V2 t ⟹
pool b ∈ mem-pools V ⟹
~~(*(V2, V⦇mem-pool-info := set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t), freeing-node := (freeing-node V)(t := None)⦈) ∈ gvars-conf-stable ⟹ ∀ p. p ≠ pool b ⟶ mem-pool-info V2 p = set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t) p ⟹*)~~
∀ j. j ≠ lvl V2 t ⟶
levels (set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t) (pool b)) ! j =
levels (mem-pool-info V2 (pool b)) ! j ⟹
lvl V2 t ≤ level b ⟹
lvl V t = lvl V2 t ⟹
∀ i<length (bits (levels (mem-pool-info V (pool b)) ! NULL)). get-bit-s V (pool b) NULL i ≠ NOEXIST ⟹
ia < length (bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
⦇bits := (bits (levels (mem-pool-info V2 (pool b)) !
(lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]⦈] !
NULL)) ⟹
bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
⦇bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]⦈] !
NULL) !
ia =
NOEXIST ⟹
False
apply(*simp add:set-bit-def*)
apply(*subgoal-tac* levels (mem-pool-info V (pool b)) ! (lvl V2 t − 1)
= levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − 1)) **prefer** 2
apply(*subgoal-tac* (levels (mem-pool-info V (pool b)))
[lvl V2 t := (levels (mem-pool-info V (pool b)) ! lvl V2 t)
⦇bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V2 t))[bn V2 t := FREE]⦈] ! (lvl V2 t − 1)

= *levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *1*)) **prefer** *2* **apply** *auto*[*1*]
apply *auto*[*1*]
apply *auto*[*1*]

apply(*case-tac lvl V2 t* − *Suc 0* = *0*)
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *0*))
= *length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *0*))) **prefer** *2* **apply** *auto*[*1*]
apply(*subgoal-tac length* (*levels* (*mem-pool-info V2* (*pool b*)))>*0*) **prefer** *2* **apply** *auto*[*1*]
apply(*subgoal-tac ia* < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *0*))) **prefer** *2*
apply(*rule subst*[**where** *s*=*length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*)) **and**
t=*length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *NULL*))])
apply *auto*[*1*]
apply(*rule subst*[**where** *t*=*length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*)) **and**
s=*length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL))])
apply *auto*[*1*]
apply *auto*[*1*]

apply(*case-tac ia* = *bn V2 t div 4*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL) ! *ia* = *FREEING*) **prefer** *2*
apply(*rule subst*[**where** *s*=*0* **and** *t*=*lvl V2 t* − *Suc 0*]) **apply** *metis*
apply(*rule subst*[**where** *s*=*ia* **and** *t*=*bn V2 t div 4*]) **apply** *metis*
apply(*rule subst*[**where** *s*=(*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*))[*ia* := *FREEING*] **and**
t=*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*)(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*))
[*ia* := *FREEING*]|)] ! *NULL*)]) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply (*metis BlockState.distinct*(*25*))

apply(*subgoal-tac bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t*

− *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL) ! *ia* ≠ *NOEXIST*) **prefer** *2*
apply(*rule subst*[**where** *s=0* **and** *t=lvl V2 t* − *Suc 0*]) **apply** *metis*
apply(*rule subst*[**where** *s*=(*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*))[*bn V2 t div 4* := *FREEING*] **and**
t=bits ((*levels* (*mem-pool-info V2* (*pool b*)))
[*NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*)(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*))
[*bn V2 t div 4* := *FREEING*]|)] ! *NULL*)]) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *fast*

apply(*subgoal-tac bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL) ! *ia* ≠ *NOEXIST*) **prefer** *2*
apply(*rule subst*[**where** *s=levels* (*mem-pool-info V2* (*pool b*)) ! *NULL* **and** *t*=(*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL]) **apply** *simp*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *NULL*)) = *length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
NULL))) **prefer** *2* **apply** *simp*
apply *presburger*
apply *fast*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0-case2*:
p ∈ *mem-pools V2* ⟹
inv V ⟹
NULL < *lvl V2 t* ⟹
pool b ∈ *mem-pools V* ⟹
(*V2*, *V*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*),
freeing-node := (*freeing-node V*)(*t* := *None*)|))
∈ *gvars-conf-stable* ⟹
∀ *p*. *p* ≠ *pool b* ⟶ *mem-pool-info V2 p* = *set-bit-free* (*mem-pool-info V*) (*pool*

b) (lvl V2 t) (bn V2 t) p $\Longrightarrow$
  level b < length (lsizes V2 t) $\Longrightarrow$
  p $\neq$ pool b $\Longrightarrow$
  ia < length (bits (levels (set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t) p) ! NULL)) $\Longrightarrow$
  get-bit (set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t)) p NULL ia = NOEXIST $\Longrightarrow$ False
apply(simp add:set-bit-def)
apply(subgoal-tac $\forall$ i<length (bits (levels (mem-pool-info V p) ! 0)). (bits (levels (mem-pool-info V p) ! 0)) ! i $\neq$ NOEXIST)
   prefer 2 **apply**(subgoal-tac mem-pools V2 = mem-pools V) **prefer** 2 **apply**(simp add:gvars-conf-stable-def gvars-conf-def)
  apply(simp add:inv-def inv-bitmap0-def Let-def)
apply auto
done

lemma mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0:
  V$\in$mp-free-precond8-3 t b α $\cap$ {|´cur = Some t|} $\Longrightarrow$
   {free-stm8-precond2 V t b} $\cap$ {|NULL < ´lvl t $\wedge$ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t)|} $\neq$ {} $\Longrightarrow$
  V2 $\in$ free-stm8-precond3 V t b $\cap$ {|´i t = 4|} $\Longrightarrow$
  x = free-stm8-atombody-rest-cond3 (V2(|lvl := (lvl V2)(t := lvl V2 t − 1), bn := (bn V2)(t := bn V2 t div 4)|)) t b $\Longrightarrow$
  y = x(|freeing-node := (freeing-node x) (t := Some (|pool = pool b, level = lvl x t, block = bn x t,
      data = block-ptr (mem-pool-info x (pool b)) (ALIGN4 (max-sz (mem-pool-info x (pool b))) div 4 ^ lvl x t) (bn x t)|))|) $\Longrightarrow$
  inv-bitmap0 y
apply(simp add:inv-bitmap0-def Let-def)
apply clarify

apply(rule conjI)
  apply clarsimp
  apply(subgoal-tac $\forall$ i<length (bits (levels (mem-pool-info V (pool b)) ! 0)).
                    (bits (levels (mem-pool-info V (pool b)) ! 0)) ! i $\neq$ NOEXIST)
  prefer 2 **apply**(simp add:inv-def inv-bitmap0-def Let-def)
  apply(subgoal-tac levels (mem-pool-info V (pool b)) ! (lvl V2 t − 1)
                    = levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − 1)) **prefer** 2
    apply(subgoal-tac levels (set-bit-free (mem-pool-info V) (pool b) (lvl V2 t) (bn V2 t) (pool b)) ! (lvl V2 t − 1)
                    = levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − 1)) **prefer** 2 **apply** auto[1]
    apply(simp add:set-bit-def)

  using mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0-case1 **apply** blast

  apply clarsimp
  using mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0-case2 **apply** blast
done

term *mp-free-precond8-3 t b* α
term *free-stm8-precond2 V t b*
term *free-stm8-precond3 V t b*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln-case1*:
pool b $\in$ *mem-pools V2* $\Longrightarrow$
inv V $\Longrightarrow$
NULL < *lvl V2 t* $\Longrightarrow$
pool b $\in$ *mem-pools V* $\Longrightarrow$
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) $\Longrightarrow$
(*V2*, *V*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*),
freeing-node := (*freeing-node V*)(*t* := *None*)|))
$\in$ *gvars-conf-stable* $\Longrightarrow$
$\forall$ *j*. *j* $\neq$ *lvl V2 t* $\longrightarrow$
levels (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) ! *j* =
levels (*mem-pool-info V2* (*pool b*)) ! *j* $\Longrightarrow$
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V2 t*) =
list-updates-n (*bits* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) ! *lvl V2 t*))
(*bn V2 t div 4* $*$ *4*) *4 NOEXIST* $\Longrightarrow$
lvl V2 t $\leq$ *level b* $\Longrightarrow$
ia < *length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* $-$ *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]|)] !
(*length* (*levels* (*mem-pool-info V2* (*pool b*))) $-$ *Suc NULL*))) $\Longrightarrow$
bits ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* $-$ *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
(*length* (*levels* (*mem-pool-info V2* (*pool b*))) $-$ *Suc NULL*)) ! *ia* = *DIVIDED* $\Longrightarrow$
False
apply(*simp add*:*set-bit-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V2* (*pool b*)))
= *length* (*levels* (*mem-pool-info V* (*pool b*)))) **prefer** *2*
using *mempool-free-stm8-atombody-rest-one-finalstm-len-lvls* **apply** *blast*

apply(*subgoal-tac let bitsn* = *bits* ((*levels* (*mem-pool-info V* (*pool b*)) ! (*length* (*levels* (*mem-pool-info V* (*pool b*))) $-$ *1*)))
in $\forall$ *i*<*length bitsn*. *bitsn* ! *i* $\neq$ *DIVIDED*) **prefer** *2*
apply(*simp add*:*inv-def inv-bitmapn-def*)

apply(*case-tac lvl V2 t = length (levels (mem-pool-info V2 (pool b))) − Suc 0*)
apply(*subgoal-tac bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
(length (levels (mem-pool-info V2 (pool b))) − Suc NULL)) ! ia ≠ DIVIDED*)
apply *auto*[*1*]
apply(*rule subst*[**where** *s=bits (levels (mem-pool-info V2 (pool b)) ! (length (levels (mem-pool-info V2 (pool b))) − Suc NULL))*
and *t=bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
(length (levels (mem-pool-info V2 (pool b))) − Suc NULL))*])
apply *auto*[*1*]
apply(*unfold Let-def*)[*1*]
apply(*subgoal-tac* ∀ *i<length (bits (levels (mem-pool-info V2 (pool b)) ! lvl V2 t)). get-bit-s V2 (pool b) (lvl V2 t) i ≠ DIVIDED*)
apply *auto*[*1*]

apply(*rule list-neq-udpt-neq*[*of bits (levels (mem-pool-info V (pool b)) ! lvl V2 t) DIVIDED
bits (levels (mem-pool-info V2 (pool b)) ! lvl V2 t) (bn V2 t div 4 ∗ 4) 4 NOEXIST*])
apply *auto*[*1*]
using *lst-udptn-set-eq*[*of 4 bits (levels (mem-pool-info V (pool b)) ! lvl V2 t) bn V2 t FREE NOEXIST*]
apply *auto*[*1*]
apply *blast*

apply(*subgoal-tac bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
(length (levels (mem-pool-info V2 (pool b))) − Suc NULL)) ! ia ≠ DIVIDED*)
apply *fast*
apply(*rule subst*[**where** *s=levels (mem-pool-info V2 (pool b)) !
(length (levels (mem-pool-info V2 (pool b))) − Suc NULL)* **and** *t=(levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
(length (levels (mem-pool-info V2 (pool b))) − Suc NULL)*])

apply *auto*[*1*]
apply(*unfold Let-def*)[*1*]
apply(*subgoal-tac levels* (*mem-pool-info V* (*pool b*)) ! (*length* (*levels* (*mem-pool-info V* (*pool b*))) − *1*) =
levels (*mem-pool-info V2* (*pool b*)) ! (*length* (*levels* (*mem-pool-info V* (*pool b*))) − *1*))
prefer *2* **apply** (*metis One-nat-def*)
by (*metis One-nat-def Suc-diff-1 inv-mempool-info-def invariant.inv-def not-less nth-list-update-neq*)

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln-case2*:
$p \in$ *mem-pools V2* $\Longrightarrow$
inv V $\Longrightarrow$
NULL < *lvl V2 t* $\Longrightarrow$
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) $\Longrightarrow$
(*V2*, *V*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*),
freeing-node := (*freeing-node V*)(*t* := *None*)|))
$\in$ *gvars-conf-stable* $\Longrightarrow$
$\forall$ *p*. *p* $\neq$ *pool b* $\longrightarrow$ *mem-pool-info V2 p* = *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p* $\Longrightarrow$
lvl V2 t $\leq$ *level b* $\Longrightarrow$
p $\neq$ *pool b* $\Longrightarrow$
ia < *length* (*bits* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*) !
(*length* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*)) − *Suc NULL*))) $\Longrightarrow$
get-bit (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*)) *p*
(*length* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*)) − *Suc NULL*) *ia* =
DIVIDED $\Longrightarrow$
False
apply(*simp add*:*set-bit-def*)
apply(*subgoal-tac* $\forall$ *i*<*length* (*bits* ((*levels* (*mem-pool-info V p*) ! (*length* (*levels* (*mem-pool-info V p*)) − *1*)))).
bits ((*levels* (*mem-pool-info V p*) ! (*length* (*levels* (*mem-pool-info V p*)) − *1*))) ! *i* $\neq$ *DIVIDED*)
prefer *2* **apply**(*subgoal-tac mem-pools V2* = *mem-pools V*) **prefer** *2* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*inv-def inv-bitmapn-def Let-def*)
apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln*:
V$\in$*mp-free-precond8-3 t b* $\alpha \cap$ {|´*cur* = *Some t*|} $\Longrightarrow$
{*free-stm8-precond2 V t b*} $\cap$ {|*NULL* < ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} $\neq$ {} $\Longrightarrow$
V2 $\in$ *free-stm8-precond3 V t b* $\cap$ {|´*i t* = *4*|} $\Longrightarrow$

x = free-stm8-atombody-rest-cond3 (V2(|lvl := (lvl V2)(t := lvl V2 t − 1), bn := (bn V2)(t := bn V2 t div 4)|)) t b $\Longrightarrow$
y = x(|freeing-node := (freeing-node x) (t := Some (|pool = pool b, level = lvl x t, block = bn x t,
data = block-ptr (mem-pool-info x (pool b)) (ALIGN4 (max-sz (mem-pool-info x (pool b))) div 4 ^ lvl x t) (bn x t)|))|) $\Longrightarrow$
inv-bitmapn y
apply(*simp add:inv-bitmapn-def Let-def*)
apply *clarify*

apply(*rule conjI*)
apply *clarsimp*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln-case1* **apply** *blast*

apply *clarsimp*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln-case2* **apply** *blast*

done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h1*:
lvl V2 t − Suc NULL = ia $\Longrightarrow$
lvl V2 t − Suc NULL < length (levels (mem-pool-info V2 (pool b))) $\Longrightarrow$
(bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING] =
bits (levels (mem-pool-info V2 (pool b)
(|levels := (levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)]|)) !
ia)
by *simp*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h2*:
ia < length (levels (mem-pool-info V2 (pool b))) $\Longrightarrow$
jj < length (bits ((levels (mem-pool-info V2 (pool b)))
[lvl V2 t − Suc NULL := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]|)] !
ia)) $\Longrightarrow$
NULL < ia $\Longrightarrow$
length (levels (mem-pool-info V2 (pool b))) = length (levels (mem-pool-info V (pool b))) $\Longrightarrow$
length (bits (levels (mem-pool-info V2 (pool b)) ! ia)) = length (bits (levels (mem-pool-info V (pool b)) ! ia)) $\Longrightarrow$
$\forall$ *jj<length (bits (levels (mem-pool-info V2 (pool b)) ! ia)).*
$\neg$ *(let bits = bits (levels (mem-pool-info V (pool b)) ! ia); a = jj div 4 ∗ 4*

in bits ! a = FREE ∧ bits ! (a + 1) = FREE ∧ bits ! (a + 2) = FREE ∧ bits ! (a + 3) = FREE) ⟹
lvl V2 t − Suc NULL = ia ⟹
levels (mem-pool-info V (pool b)) ! (lvl V2 t − Suc NULL) = levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL) ⟹
¬ *(let bits = (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]; a = jj div 4 ∗ 4*
in bits ! a = FREE ∧ bits ! (a + 1) = FREE ∧ bits ! (a + 2) = FREE ∧ bits ! (a + 3) = FREE)
apply(*unfold Let-def*)
apply(*rule subst*[**where** *s=list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING* **and**
t=(bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL)))[bn V2 t div 4 := FREEING]])
using *lst-updt1-eq-upd* **apply** *fast*
apply(*subgoal-tac length (list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING)*
= length (bits (levels (mem-pool-info V2 (pool b)) ! ia)))
prefer *2* **using** *length-list-update-n* **apply** *fast*
apply(*subgoal-tac* ∀*jj<length (list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING).*
¬ *(let a = jj div 4 ∗ 4*
in list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING ! a = FREE ∧
list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING ! (a + 1) = FREE ∧
list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING ! (a + 2) = FREE ∧
list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING ! (a + 3) = FREE))
prefer *2*
apply(*rule partnerbits-udptn-notbit-partbits*[*of bits (levels (mem-pool-info V2 (pool b)) ! ia) FREE FREEING*
list-updates-n (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc NULL))) (bn V2 t div 4) 1 FREEING (bn V2 t div 4) 1])
apply(*unfold Let-def*)[*1*] **apply** *metis*
apply *blast*
apply *fast*
apply(*unfold Let-def*)
apply(*subgoal-tac length (bits ((levels (mem-pool-info V2 (pool b)))*
[lvl V2 t − Suc 0 := (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc 0))
(|bits := (bits (levels (mem-pool-info V2 (pool b)) ! (lvl V2 t − Suc 0)))[bn V2 t div 4 := FREEING]|)] !
ia)) = length (bits (levels (mem-pool-info V2 (pool b)) ! ia)))
prefer *2*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h2* **apply** *fast*
by *metis*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h3*:
lvl V2 t $-$ *Suc NULL* $\neq$ *ia* $\Longrightarrow$
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*) =
bits (*levels* (*mem-pool-info V2* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* $-$ *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)]|)) !
ia)
by *auto*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1*:
pool b $\in$ *mem-pools V2* $\Longrightarrow$
inv V $\Longrightarrow$
NULL $<$ *lvl V2 t* $\Longrightarrow$
partner-bits (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) (*lvl V2 t*) (*bn V2 t*) $\Longrightarrow$
pool b $\in$ *mem-pools V* $\Longrightarrow$
level b $<$ *length* (*levels* (*mem-pool-info V* (*pool b*))) $\Longrightarrow$
(*V2*, *V*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*),
freeing-node := (*freeing-node V*)(*t* := *None*)|))
$\in$ *gvars-conf-stable* $\Longrightarrow$
block b $<$ *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) $\Longrightarrow$
$\forall$ *j*. *j* $\neq$ *lvl V2 t* $\longrightarrow$
levels (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) ! *j* =
levels (*mem-pool-info V2* (*pool b*)) ! *j* $\Longrightarrow$
level b $<$ *length* (*lsizes V2 t*) $\Longrightarrow$
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V2 t*) =
list-updates-n (*bits* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) ! *lvl V2 t*))
(*bn V2 t div 4* $*$ *4*) *4* *NOEXIST* $\Longrightarrow$
lvl V2 t $\leq$ *level b* $\Longrightarrow$
ia $<$ *length* (*levels* (*mem-pool-info V2* (*pool b*))) $\Longrightarrow$
jj $<$ *length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* $-$ *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* $-$ *Suc NULL*)))
[*bn V2 t div 4* := *FREEING*]|)] !
ia)) $\Longrightarrow$
NULL $<$ *ia* $\Longrightarrow$
partner-bits
(*mem-pool-info V2* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))

[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)]|))
ia jj ⟹
False
apply(*simp add:set-bit-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V2* (*pool b*)))
= *length* (*levels* (*mem-pool-info V* (*pool b*)))) **prefer** *2*
using *mempool-free-stm8-atombody-rest-one-finalstm-len-lvls* **apply** *blast*

apply(*subgoal-tac* ¬ *partner-bits* (*mem-pool-info V2* (*pool b*) (|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)]|))
ia jj) **apply** *fast*

apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac lvl V2 t* = *ia*) **apply**(*simp add:set-bit-def*) **apply** *presburger*
apply(*subgoal-tac* ∀ *jj*<*length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*)). ¬ *partner-bits* (*mem-pool-info V* (*pool b*)) *ia jj*)
prefer *2* **apply**(*simp add:inv-def inv-bitmap-not4free-def Let-def*)

apply(*case-tac lvl V2 t* = *ia*)
apply(*rule subst*[**where** *s*=*partner-bits* (*mem-pool-info V2* (*pool b*)) *ia jj* **and** *t*=*partner-bits* (*mem-pool-info V2* (*pool b*) (|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)]|))
ia jj]) **apply**(*simp add:partner-bits-def Let-def*)

apply(*subgoal-tac bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V2 t*) =
list-updates-n (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V2 t*)) (*bn V2 t div 4* ∗ *4*) *4 NOEXIST*) **prefer** *2*
using *lst-udptn-set-eq*[*of 4 bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V2 t*) *bn V2 t FREE NOEXIST*] **apply** *simp*

apply(*unfold partner-bits-def*)[*1*]
apply(*subgoal-tac* ¬ (*let a* = *jj div 4* ∗ *4*
in list-updates-n (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)) (*bn V2 t div 4* ∗ *4*) *4 NOEXIST* ! *a* = *FREE* ∧
list-updates-n (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)) (*bn V2 t div 4* ∗ *4*) *4 NOEXIST* ! (*a* + *1*) = *FREE* ∧

list-updates-n (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ia*)) (*bn* *V2* *t* *div* *4* ∗ *4*) *4* *NOEXIST* ! (*a* + *2*) = *FREE* ∧
list-updates-n (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ia*)) (*bn* *V2* *t* *div* *4* ∗ *4*) *4* *NOEXIST* ! (*a* + *3*) = *FREE*))
prefer *2*
apply(*rule partnerbits-udptn-notbit-partbits*[*rule-format*, *of bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ia*) *FREE* *NOEXIST*
list-updates-n (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ia*)) (*bn* *V2* *t* *div* *4* ∗ *4*) *4* *NOEXIST*
bn *V2* *t* *div* *4* ∗ *4* *4* *jj*])
apply(*unfold Let-def*)[*1*] **apply** *presburger*
apply *blast* **apply** *fast* **apply** *force*
apply(*unfold Let-def*)[*1*] **apply** *presburger*

apply(*case-tac lvl* *V2* *t* − *Suc* *0* = *ia*)
apply(*unfold partner-bits-def*)
apply(*rule subst*[**where** *s*=(*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *0*)))[*bn* *V2* *t* *div* *4* := *FREEING*]
and *t*=*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)
(|*levels* := (*levels* (*mem-pool-info* *V2* (*pool* *b*)))
[*lvl* *V2* *t* − *Suc* *0* := (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *0*))
(|*bits* := (*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *0*)))[*bn* *V2* *t* *div* *4* := *FREEING*]|)]|)) !
ia)])
apply(*subgoal-tac lvl* *V2* *t* − *Suc* *NULL* < *length* (*levels* (*mem-pool-info* *V2* (*pool* *b*)))) **prefer** *2* **apply** *blast*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h1* **apply** *blast*

apply(*subgoal-tac levels* (*mem-pool-info* *V* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *0*) = *levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *0*))
prefer *2* **apply** *presburger*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h2* **apply** *blast*

apply(*rule subst*[**where** *s*=*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! *ia*) **and**
t=*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)
(|*levels* := (*levels* (*mem-pool-info* *V2* (*pool* *b*)))
[*lvl* *V2* *t* − *Suc* *NULL* := (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info* *V2* (*pool* *b*)) ! (*lvl* *V2* *t* − *Suc* *NULL*)))[*bn* *V2* *t* *div* *4* := *FREEING*]|)]|)) !
ia)])
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1-h3* **ap-**

ply *blast*

apply(*rule subst*[**where** *s=levels* (*mem-pool-info V* (*pool b*)) ! *ia* **and** *t=levels* (*mem-pool-info V2* (*pool b*)) ! *ia*])
  apply *metis*
apply(*unfold Let-def*)

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
  [*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
  (|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
  ia)) =
  length (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *ia*))) **prefer** *2*
 using *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info-h2* **apply** *blast*
by *metis*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case2*:
p ∈ *mem-pools V2* ⟹
  inv V ⟹
  NULL < *lvl V2 t* ⟹
  partner-bits (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) (*pool b*)) (*lvl V2 t*) (*bn V2 t*) ⟹
  pool b ∈ *mem-pools V* ⟹
  level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
  (*V2*, *V*(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*),
  freeing-node := (*freeing-node V*)(*t* := *None*)|))
  ∈ *gvars-conf-stable* ⟹
  lvl V2 t ≤ *level b* ⟹
  ∀ *p*. *p* ≠ *pool b* ⟶ *mem-pool-info V2 p* = *set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p* ⟹
  p ≠ *pool b* ⟹
  ia < *length* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*)) ⟹
  jj < *length* (*bits* (*levels* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*) ! *ia*)) ⟹
  NULL < *ia* ⟹ *partner-bits* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V2 t*) (*bn V2 t*) *p*) *ia jj* ⟹ *False*
apply(*simp add*:*set-bit-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info V2* (*pool b*)))
  = *length* (*levels* (*mem-pool-info V* (*pool b*)))) **prefer** *2*
  using *mempool-free-stm8-atombody-rest-one-finalstm-len-lvls* **apply** *blast*

apply(*subgoal-tac* ¬ *partner-bits* (*mem-pool-info V p*) *ia jj*) **apply** *fast*

apply(*subgoal-tac p* ∈ *mem-pools V*) **prefer** *2* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)

apply(*subgoal-tac* $\forall$ *jj*<*length* (*bits* (*levels* (*mem-pool-info V p*) ! *ia*)). $\neg$ *partner-bits* (*mem-pool-info V p*) *ia jj*)
prefer *2* **apply**(*simp add*:*inv-def inv-bitmap-not4free-def Let-def*)

by *blast*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free*:
$V \in$*mp-free-precond8-3 t b* $\alpha \cap \{\!|$ ´*cur* = *Some t* $|\!\} \Longrightarrow$
$\{$*free-stm8-precond2 V t b*$\} \cap \{\!|$*NULL* < ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)$|\!\} \neq \{\} \Longrightarrow$
V2 $\in$ *free-stm8-precond3 V t b* $\cap \{\!|$´*i t* = *4*$|\!\} \Longrightarrow$
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* $\Longrightarrow$
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ^ *lvl x t*) (*bn x t*)|))|) $\Longrightarrow$
inv-bitmap-not4free y
apply(*simp add*:*inv-bitmap-not4free-def Let-def*)
apply *clarify*
apply(*rule conjI*)
apply *clarsimp*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case1* **apply** *blast*

apply *clarsimp*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free-case2* **apply** *blast*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv'*:
$V \in$*mp-free-precond8-3 t b* $\alpha \cap \{\!|$ ´*cur* = *Some t* $|\!\} \Longrightarrow$
$\{$*free-stm8-precond2 V t b*$\} \cap \{\!|$*NULL* < ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)$|\!\} \neq \{\} \Longrightarrow$
V2 $\in$ *free-stm8-precond3 V t b* $\cap \{\!|$´*i t* = *4*$|\!\} \Longrightarrow$
x = *free-stm8-atombody-rest-cond3* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)|)) *t b* $\Longrightarrow$
y = *x*(|*freeing-node* := (*freeing-node x*) (*t* := *Some* (|*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ^ *lvl x t*) (*bn x t*)|))|) $\Longrightarrow$
inv y
apply(*rule subst*[**where** *s*=*inv-cur y* $\wedge$ *inv-thd-waitq y* $\wedge$ *inv-mempool-info y*
$\wedge$ *inv-bitmap-freelist y* $\wedge$ *inv-bitmap y* $\wedge$ *inv-aux-vars y*
$\wedge$ *inv-bitmap0 y* $\wedge$ *inv-bitmapn y* $\wedge$ *inv-bitmap-not4free y* **and** *t*=*inv y*])
using *inv-def*[*of y*] **apply** *fast*

apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-cur*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-thd-waitq*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-mempool-info*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap-freelist*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-bitmap*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-aux-vars*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvl0*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvln*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*
using *mempool-free-stm8-atombody-rest-one-finalstm-inv-lvls-not4free*[*of* $V\ t\ b\ \alpha\ V2\ x\ y$] **apply** *fast*

done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-inv*:
$V \in mp\text{-}free\text{-}precond8\text{-}3\ t\ b\ \alpha \cap \{\!\!\{\'cur = Some\ t\}\!\!\} \Longrightarrow$
$\{free\text{-}stm8\text{-}precond2\ V\ t\ b\} \cap \{\!\!\{NULL < \'lvl\ t \wedge partner\text{-}bits\ (\'mem\text{-}pool\text{-}info\ (pool\ b))\ (\'lvl\ t)\ (\'bn\ t)\}\!\!\} \neq \{\} \Longrightarrow$
$V2 \in free\text{-}stm8\text{-}precond3\ V\ t\ b \cap \{\!\!\{\'i\ t = 4\}\!\!\} \Longrightarrow$
$x = free\text{-}stm8\text{-}atombody\text{-}rest\text{-}cond3\ (V2(\!|lvl := (lvl\ V2)(t := lvl\ V2\ t - 1),\ bn := (bn\ V2)(t := bn\ V2\ t\ div\ 4)|\!))\ t\ b \Longrightarrow$
$x(\!|freeing\text{-}node := (freeing\text{-}node\ x)\ (t := Some\ (\!|pool = pool\ b,\ level = lvl\ x\ t,\ block = bn\ x\ t,$
$data = block\text{-}ptr\ (mem\text{-}pool\text{-}info\ x\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ x\ (pool\ b)))\ div\ 4\ \hat{}\ lvl\ x\ t)\ (bn\ x\ t)|\!))|\!)$
$\in \{\!\!\{\'inv\}\!\!\}$
using *mempool-free-stm8-atombody-rest-one-finalstm-inv'*[*of* $V\ t\ b\ \alpha\ V2\ x$
$x(\!|freeing\text{-}node := (freeing\text{-}node\ x)\ (t := Some\ (\!|pool = pool\ b,\ level = lvl\ x\ t,\ block = bn\ x\ t,$
$data = block\text{-}ptr\ (mem\text{-}pool\text{-}info\ x\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ x\ (pool\ b)))\ div\ 4\ \hat{}\ lvl\ x\ t)\ (bn\ x\ t)|\!))|\!)$] **apply** *fast*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h2*:
$V \in mp\text{-}free\text{-}precond8\text{-}3\ t\ b\ \alpha \cap \{\!\!\{\'cur = Some\ t\}\!\!\} \Longrightarrow$
$\{free\text{-}stm8\text{-}precond2\ V\ t\ b\} \cap \{\!\!\{NULL < \'lvl\ t \wedge partner\text{-}bits\ (\'mem\text{-}pool\text{-}info\ (pool\ b))\ (\'lvl\ t)\ (\'bn\ t)\}\!\!\} \neq \{\} \Longrightarrow$
$V2 \in free\text{-}stm8\text{-}precond3\ V\ t\ b \cap \{\!\!\{\'i\ t = 4\}\!\!\} \Longrightarrow$
$x = free\text{-}stm8\text{-}atombody\text{-}rest\text{-}cond3\ (V2(\!|lvl := (lvl\ V2)(t := lvl\ V2\ t - 1),\ bn := (bn\ V2)(t := bn\ V2\ t\ div\ 4)|\!))\ t\ b \Longrightarrow$
$x(\!|freeing\text{-}node := (freeing\text{-}node\ x)\ (t := Some\ (\!|pool = pool\ b,\ level = lvl\ x\ t,$

block = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)⦈))⦈
∈ ⦃´*allocating-node t* = *None*⦄
by (*simp add*:*Let-def block-ptr-def*)

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h1-2*:
V∈*mp-free-precond8-3 t b α* ∩ ⦃´*cur* = *Some t*⦄ ⟹
{*free-stm8-precond2 V t b*} ∩ ⦃*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)⦄ ≠ {} ⟹
V2 ∈ *free-stm8-precond3 V t b* ∩ ⦃´*i t* = *4*⦄ ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*⦇*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)⦈) *t b* ⟹
y = *x*⦇*freeing-node* := *freeing-node x*(*t* ↦
⦇*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)⦈)⦈ ⟹
y ∈ ⦃´(*Pair V*)
∈ {(*s*, *r*). (*cur s* ≠ *Some t* ⟶ *gvars-nochange s r* ∧ *lvars-nochange t s r*) ∧
(*cur s* = *Some t* ⟶ *invariant.inv s* ⟶ *invariant.inv r*) ∧ (∀ *t*′. *t*′ ≠ *t* ⟶ *lvars-nochange t*′ *s r*)}⦄
apply(*subgoal-tac* (*cur V* ≠ *Some t* ⟶ *gvars-nochange V y* ∧ *lvars-nochange t V y*) ∧
(*cur V* = *Some t* ⟶ *invariant.inv V* ⟶ *invariant.inv y*) ∧ (∀ *t*′. *t*′ ≠ *t* ⟶ *lvars-nochange t*′ *V y*))
prefer *2*
apply(*rule conjI*)
apply(*subgoal-tac cur V* = *Some t*) **prefer** *2* **apply** *fast* **apply** *fast*
apply(*rule conjI*)
apply(*rule impI*)+ **using** *mempool-free-stm8-atombody-rest-one-finalstm-inv*′[*of V t b α V2 x y*] **apply** *fast*
apply(*rule allI*) **apply**(*rule impI*) **apply**(*simp add*:*lvars-nochange-def Let-def*)

apply *fast*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h1-h1*:
∀ *j*. *j* ≠ *lvl V t* ⟶ *levels* (*mem-pool-info V* (*pool b*)) ! *j* = *levels* (*mem-pool-info V2* (*pool b*)) ! *j* ⟹
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) =
list-updates-n ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*])
(*block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4*) *4 NOEXIST* ⟹
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)) =
length (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))

(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
ia))
apply(*rule subst*[**where** *s*=*length* (*bits* (*levels* (*mem-pool-info V2* (*pool b*))!*ia*)) **and** *t*=*length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc 0* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc 0*)))[*bn V2 t div 4* := *FREEING*]|)] !
ia))])
apply(*case-tac ia* = *lvl V2 t* − *Suc 0*)
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V2* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]

apply(*case-tac ia* = *lvl V t*)
apply(*subgoal-tac length* (*list-updates-n*
(*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)] !
ia))
(*block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4*) *4 NOEXIST*) = *length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)))
prefer *2* **using** *length-list-update-n* **apply** *fast*
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*)
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)] !
ia)) = *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ia*)))
prefer *2* **apply**(*case-tac ia* = *lvl V t*) **apply**(*case-tac ia* < *length* (*levels* (*mem-pool-info V* (*pool b*))))
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h1*:
V∈*mp-free-precond8-3 t b α* ∩ ⦃´*cur* = *Some t*⦄ ⟹
{*free-stm8-precond2 V t b*} ∩ ⦃*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)⦄ ≠ {} ⟹
V2 ∈ *free-stm8-precond3 V t b* ∩ ⦃´*i t* = *4*⦄ ⟹
x = *free-stm8-atombody-rest-cond3* (*V2*⦇*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*), *bn* := (*bn V2*)(*t* := *bn V2 t div 4*)⦈) *t b* ⟹
x⦇*freeing-node* := (*freeing-node x*) (*t* := *Some* ⦇*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*) (*bn x t*)⦈)⦈
∈ ⦃´(*Pair V*) ∈ *Mem-pool-free-guar t*⦄
apply(*unfold Mem-pool-free-guar-def*)
apply(*rule pairv-rId*)
apply(*rule pairv-IntI*) **apply**(*rule pairv-IntI*)

apply(*unfold gvars-conf-stable-def gvars-conf-def*)[*1*]
apply *clarify* **apply**(*simp add*:*Let-def set-bit-def*)
apply *clarify* **using** *mempool-free-stm8-atombody-rest-one-finalstm-h1-h1*[*of V t b V2*] **apply** *blast*

using *mempool-free-stm8-atombody-rest-one-finalstm-h1-2*[*of V t b α V2 x*
x⦇*freeing-node* := *freeing-node x*(*t* ↦
⦇*pool* = *pool b*, *level* = *lvl x t*, *block* = *bn x t*,
data = *block-ptr* (*mem-pool-info x* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info x* (*pool b*))) *div 4* ˆ *lvl x t*)
(*bn x t*)⦈)⦈] **apply** *fast*

apply(*simp add*:*Let-def*)
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-I1*:
x∈⦃´*invariant.inv*⦄ ⟹
x∈⦃´*allocating-node t* = *None*⦄ ⟹
x∈⦃´*invariant.inv* ∧ ´*allocating-node t* = *None*⦄
by *auto*

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h3-h1*:
inv V ∧
pool b ∈ *mem-pools V2* ∧
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ∧ *lvl V2 t* ≤ *level b* ∧ *NULL* < *lvl V2 t* ⟹
mem-pools V = *mem-pools V2* ∧

lvl V t = *lvl V2 t* ⟹
n-max (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ (*lvl V2 t* − *Suc NULL*) =
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*) **apply** *auto*[*1*]
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h3*:
invariant.inv V ∧
pool b ∈ *mem-pools V2* ∧
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ∧
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ∧
level b < *length* (*lsizes V2 t*) ∧
bn V t < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V2 t*)) ∧
lvl V2 t ≤ *level b* ∧
NULL < *lvl V2 t* ⟹
mem-pools V = *mem-pools V2* ∧
(∀ *p*.
(∀ *i*. *length* (*bits* (*levels* (*mem-pool-info V2 p*) ! *i*)) =
length (*bits* (*levels* (*if p* = *pool b*
then mem-pool-info V (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) !
lvl V t)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool*
b)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]|)]|)
else mem-pool-info V p) !
i)))) ∧
(∀ *p*. *p* ≠ *pool b* ⟶ *mem-pool-info V2 p* = *mem-pool-info V p*) ∧
(∀ *j*. *j* ≠ *lvl V t* ⟶ *levels* (*mem-pool-info V* (*pool b*)) ! *j* = *levels* (*mem-pool-info*
V2 (*pool b*)) ! *j*) ∧
bits (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) =
list-updates-n ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b*
div 4 ˆ (*level b* − *lvl V t*) := *FREE*])
(*block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4*) (*i V2 t*) *NOEXIST* ∧

block b div 4 ˆ (*level b* − *lvl V t*) = *bn V2 t* ∧
lvl V t = *lvl V2 t* ∧ *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V*
t = *lsz V2 t* ∧ *lsizes V t* = *lsizes V2 t* ∧ *i V2 t* ≤ *4* ∧ *i V2 t* = *4* ⟹
x = *V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *Suc NULL*), *bn* := (*bn V2*)(*t* := *bn V2*
t div 4),
mem-pool-info := (*mem-pool-info V2*)
(*pool b* := *mem-pool-info V2* (*pool b*)
(|*levels* := (*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl*
V2 t − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* −
Suc NULL)))[*bn V2 t div 4* := *FREEING*]|)]|))|) ⟹
bn V2 t div 4
< *length* (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))

[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
(*lvl V2 t* − *Suc NULL*)))
apply(*rule subst*[**where** *s*= *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))) **and** *t* =
length (*bits* ((*levels* (*mem-pool-info V2* (*pool b*)))
[*lvl V2 t* − *Suc NULL* := (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! (*lvl V2 t* − *Suc NULL*)))[*bn V2 t div 4* := *FREEING*]|)] !
(*lvl V2 t* − *Suc NULL*)))])
apply(*subgoal-tac* ∀ *j*. *j* ≠ *lvl V t* ⟶ *levels* (*mem-pool-info V* (*pool b*)) ! *j* = *levels* (*mem-pool-info V2* (*pool b*)) ! *j*)
prefer *2* **apply** *fast*
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) =
list-updates-n ((*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*])
(*block b div 4* ˆ (*level b* − *lvl V t*) *div 4* ∗ *4*) (*i V2 t*) *NOEXIST*)
prefer *2* **apply** *fast*
using *mempool-free-stm8-atombody-rest-one-finalstm-h1-h1*[*of V t b V2* (*lvl V2 t* − *Suc NULL*)] **apply** *auto*[*1*]

apply(*rule subst*[**where** *s*=(*n-max* (*mem-pool-info V* (*pool b*))) ∗ *4* ˆ (*lvl V2 t* − *Suc 0*)
and *t*=*length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! (*lvl V2 t* − *Suc 0*)))])
using *mempool-free-stm8-atombody-rest-one-finalstm-h3-h1* **apply** *fast*

apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*))
= (*n-max* (*mem-pool-info V* (*pool b*))) ∗ *4* ˆ *level b*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def Let-def*)

apply(*rule lm11*[*of V2 t b V*]) **apply** *simp* **apply** *simp* **apply** *metis*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm-h4*:
(¬ *free-block-r V2 t* ⟶ *freeing-node V t* = *None*) ∧
α = (*if* ∃ *y*. *freeing-node V t* = *Some y then lvl V t* + *1 else NULL*) ∧
V ∈ (*if NULL* < α *then UNIV else* {}) ⟹ *free-block-r V2 t*
apply *auto*
done

lemma *mempool-free-stm8-atombody-rest-one-finalstm*:
V∈*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ⟹
{*free-stm8-precond2 V t b*} ∩ {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ≠ {} ⟹
V2 ∈ *free-stm8-precond3 V t b* ∩ {|´*i t* = *4*|} ⟹

$\{$*free-stm8-atombody-rest-cond3* $(V2(\!|lvl := (lvl\ V2)(t := lvl\ V2\ t - 1),\ bn := (bn\ V2)(t := bn\ V2\ t\ div\ 4)|\!))\ t\ b\}$
$\subseteq \{\!|\acute{}(freeing\text{-}node\text{-}update$
$(\lambda\text{-}.\ \acute{}freeing\text{-}node(t \mapsto$
$(\!|pool = pool\ b,\ level = \acute{}lvl\ t,\ block = \acute{}bn\ t,$
$data = block\text{-}ptr\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b)))\ div\ 4\ \hat{}\ \acute{}lvl\ t)\ (\acute{}bn\ t)|\!))))$
$\in \{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha - 1)|\!\}$

apply(*rule subsetI*)
apply(*subgoal-tac* $x = free\text{-}stm8\text{-}atombody\text{-}rest\text{-}cond3\ (V2(\!|lvl := (lvl\ V2)(t := lvl\ V2\ t - 1),\ bn := (bn\ V2)(t := bn\ V2\ t\ div\ 4)|\!))\ t\ b$)
prefer *2* **apply** *fast*
apply(*subgoal-tac* $x(\!|freeing\text{-}node := (freeing\text{-}node\ x)\ (t := Some\ (\!|pool = pool\ b,\ level = lvl\ x\ t,\ block = bn\ x\ t,$
$data = block\text{-}ptr\ (mem\text{-}pool\text{-}info\ x\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ x\ (pool\ b)))\ div\ 4\ \hat{}\ lvl\ x\ t)\ (bn\ x\ t)|\!))|\!)$
$\in \{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha{-}1)$)
apply *blast*

apply(*rule IntI*)

using *mempool-free-stm8-atombody-rest-one-finalstm-h1*[*of V t b* α *V2*] **apply** *meson*

apply(*rule IntI*)
apply(*rule IntI*)
apply(*rule IntI*)

apply(*rule mempool-free-stm8-atombody-rest-one-finalstm-I1*)
using *mempool-free-stm8-atombody-rest-one-finalstm-inv*[*of V t b* α *V2*] **apply** *meson*
using *mempool-free-stm8-atombody-rest-one-finalstm-h2*[*of V t b* α *V2*] **apply** *meson*

apply(*simp add:Let-def gvars-conf-stable-def gvars-conf-def block-ptr-def set-bit-def*)
apply(*subgoal-tac* $length\ (bits\ (levels\ (mem\text{-}pool\text{-}info\ V\ (pool\ b))\ !\ level\ b)) =$
$length\ (bits\ ((levels\ (mem\text{-}pool\text{-}info\ V2\ (pool\ b)))$
$[lvl\ V2\ t - Suc\ NULL := (levels\ (mem\text{-}pool\text{-}info\ V2\ (pool\ b))\ !\ (lvl\ V2\ t - Suc\ NULL))$
$(\!|bits := (bits\ (levels\ (mem\text{-}pool\text{-}info\ V2\ (pool\ b))\ !\ (lvl\ V2\ t - Suc\ NULL)))[bn\ V2\ t\ div\ 4 := FREEING]|\!)]\ !$
$level\ b))$)
prefer *2* **apply**(*subgoal-tac* $\forall j.\ j \neq lvl\ V\ t \longrightarrow levels\ (mem\text{-}pool\text{-}info\ V\ (pool\ b))\ !\ j$
$= levels\ (mem\text{-}pool\text{-}info\ V2\ (pool\ b))\ !\ j$) **prefer** *2* **apply** *fast*
apply(*subgoal-tac* $bits\ (levels\ (mem\text{-}pool\text{-}info\ V2\ (pool\ b))\ !\ lvl\ V\ t) =$

list-updates-n ((*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V* *t*))[*block* *b* *div* *4* ˆ (*level* *b* − *lvl* *V* *t*) := *FREE*])
(*block* *b* *div* *4* ˆ (*level* *b* − *lvl* *V* *t*) *div* *4* ∗ *4*) (*i* *V2* *t*) *NOEXIST*)
prefer *2* **apply** *fast*
using *mempool-free-stm8-atombody-rest-one-finalstm-h1-h1*[*of* *V* *t* *b* *V2* *level* *b*]
apply *argo*
apply *auto*[*1*]

apply(*simp add*:*Let-def gvars-conf-stable-def gvars-conf-def block-ptr-def set-bit-def*)
apply(*rule conjI*)
using *mempool-free-stm8-atombody-rest-one-finalstm-h3* **apply** *blast*

apply(*rule conjI*)
apply(*rule subst*[**where** *s*=*lvl* *V* *t* **and** *t*=*lvl* *V2* *t*]) **apply** *fast*
apply (*metis Nat.add-diff-assoc div-mult2-eq plus-1-eq-Suc power-add power-commutes power-one-right*)

apply(*rule conjI*)
apply (*metis Suc-pred le-imp-less-Suc nat-le-linear not-less*)

apply(*rule conjI*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info* *V* (*pool* *b*)) **and** *t*=*ALIGN4* (*max-sz* (*mem-pool-info* *V* (*pool* *b*)))])
using *inv-maxsz-align4* **apply** *auto*[*1*]
apply *clarify* **apply**(*rule conjI*)
apply *fast*

apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *level* *b*)) = (*n-max* (*mem-pool-info* *V* (*pool* *b*))) ∗ *4* ˆ *level* *b*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac level* *b* > *0*) **prefer** *2* **apply** *auto*[*1*]
apply(*subgoal-tac block* *b* < *n-max* (*mem-pool-info* *V* (*pool* *b*)) ∗ *4* ˆ *level* *b*) **prefer** *2* **apply** *argo*
apply(*subgoal-tac block* *b* *div* *4* ˆ (*level* *b* − *lvl* *V2* *t*) = *bn* *V2* *t*) **prefer** *2* **apply** *metis*
using *lm11*[*of* *V2* *t* *b* *V*] **apply** *meson*

using *mempool-free-stm8-atombody-rest-one-finalstm-h4*[*of* *V2* *t* *V*] **apply** *fast*

apply(*simp add*:*Let-def*)
apply *clarsimp*
apply *auto*
done

term *free-stm8-precond2* *V* *t* *b*

lemma *mempool-free-stm8-atombody-rest-one*:
V∈*mp-free-precond8-3* *t* *b* α ∩ ⦃´*cur* = *Some* *t*⦄ ⟹

$\{$*free-stm8-precond2 V t b*$\}$ $\cap$ $\{\!|$*NULL* $<$ ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)$|\!\}$ $\neq$ $\{\}$ $\Longrightarrow$
V2 $\in$ *free-stm8-precond3 V t b* $\cap$ $\{\!|$´*i t* = *4*$|\!\}$ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*lvl* := ´*lvl*(*t* := ´*lvl t* − *1*);;
´*bn* := ´*bn*(*t* := ´*bn t div 4*);;
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node*(*t* $\mapsto$(|*pool* = *pool b*, *level* = ´*lvl t*, *block* = ´*bn t*,
data = *block-ptr* (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ^ ´*lvl t*) (´*bn t*)|)))
sat$_p$ [$\{$*V2*$\}$, $\{$(*s*, *t*). *s* = *t*$\}$, *UNIV*,
$\{\!|$´(*Pair V*) $\in$ *Mem-pool-free-guar t*$|\!\}$ $\cap$ *mp-free-precond8-inv t b* (α−*1*)]
apply(*rule Seq*[**where** *mid*=$\{$*free-stm8-atombody-rest-cond3* (*free-stm8-atombody-rest-cond2* (*free-stm8-atombody-rest-cond1 V2 t b*) *t b*) *t b*$\}$])
apply(*rule Seq*[**where** *mid*=$\{$*free-stm8-atombody-rest-cond2* (*free-stm8-atombody-rest-cond1 V2 t b*) *t b*$\}$])
apply(*rule Seq*[**where** *mid*=$\{$*free-stm8-atombody-rest-cond1 V2 t b*$\}$])

apply(*rule Basic*)
apply *fast* **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply *fast* **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply(*simp add*:*set-bit-def Let-def*) **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
apply(*rule subst*[**where** *s*=*bn V2* **and** *t*=*bn* (*V2*(|*lvl* := (*lvl V2*)(*t* := *lvl V2 t* − *1*)|))])
apply *auto*[*1*]
using *mempool-free-stm8-atombody-rest-one-finalstm*[*of V t b* α *V2*] **apply** *meson*
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*
done

lemma *mempool-free-stm8-atombody-rest*:
V$\in$*mp-free-precond8-3 t b* α $\cap$ $\{\!|$´*cur* = *Some t*$|\!\}$ $\Longrightarrow$
$\{$*free-stm8-precond2 V t b*$\}$ $\cap$ $\{\!|$*NULL* $<$ ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)$|\!\}$ $\neq$ $\{\}$ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*lvl* := ´*lvl*(*t* := ´*lvl t* − *1*);;
´*bn* := ´*bn*(*t* := ´*bn t div 4*);;
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node*(*t* $\mapsto$(|*pool* = *pool b*, *level* = ´*lvl t*, *block* = ´*bn t*,

data = *block-ptr* (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ^ ´*lvl t*) (´*bn t*)⦈))
sat_p [*free-stm8-precond3 V t b* ∩ ⦃´*i t* = *4*⦄, {(*s*, *t*). *s* = *t*}, *UNIV*,
⦃´(*Pair V*) ∈ *Mem-pool-free-guar t*⦄ ∩ *mp-free-precond8-inv t b* (α − *1*)]
using *mempool-free-stm8-atombody-rest-one*[*of V t b α*]
Allprecond[**where** *U*=*free-stm8-precond3 V t b* ∩ ⦃´*i t* = *4*⦄ **and**
P=*Some* (´*lvl* := ´*lvl*(*t* := ´*lvl t* − *1*);;
´*bn* := ´*bn*(*t* := ´*bn t div 4*);;
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node*(*t* ↦⦇*pool* = *pool b*, *level* = ´*lvl t*, *block* = ´*bn t*,
data = *block-ptr* (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ^ ´*lvl t*) (´*bn t*)⦈)) **and**
rely={(*x*, *y*). *x* = *y*} **and**
guar= *UNIV* **and** *post*= ⦃´(*Pair V*) ∈ *Mem-pool-free-guar t*⦄ ∩ *mp-free-precond8-inv t b* (α − *1*)]
apply *meson*
done

abbreviation *free-stm8-bd2-cond1 V t b* ≡ *V*⦇*j* := (*j V*)(*t* := *lvl V t*)⦈
abbreviation *free-stm8-bd2-cond2 V t b* ≡ *V*⦇*lbn* := (*lbn V*)(*t* := *bn V t*)⦈
abbreviation *free-stm8-bd2-cond3 V t b* ≡ *V*⦇*lvl* := (*lvl V*)(*t* := *j V t* − *1*)⦈
abbreviation *free-stm8-bd2-cond4 V t b* ≡ *V*⦇*bn* := (*bn V*)(*t* := *lbn V t div 4*)⦈
abbreviation *free-stm8-bd2-cond5 V t b* ≡
let minf = *mem-pool-info V* (*pool b*) *in*
V⦇*mem-pool-info*:= (*mem-pool-info V*) (*pool b* := *minf* ⦇*levels* := (*levels minf*)
[*lvl V t* := ((*levels minf*) ! (*lvl V t*)) ⦇*bits* := (*bits* ((*levels minf*) ! (*lvl V t*)))
[*bn V t* := *FREEING*]⦈] ⦈)⦈

lemma *mempool-free-stm8-atombody-else-blockfit*:
V ∈ *mp-free-precond8-3 t b α* ∩ ⦃´*cur* = *Some t*⦄ ⟹
free-stm8-precond2 V t b ∈ ⦃*block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*) (´*lsz t*)⦄
apply(*simp add*:*block-fits-def block-ptr-def buf-size-def set-bit-def*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info V* (*pool b*)) **and** *t*=*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))])
apply(*simp add*: *inv-def*) **using** *inv-mempool-info-maxsz-align4*[*rule-format*,*of V pool b*] **apply** *metis*

apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*))) ! *level b*)) = (*n-max* (*mem-pool-info V* (*pool b*))) ∗ *4* ^ (*level b*))
prefer *2* **apply**(*simp add*: *inv-def inv-mempool-info-def Let-def*)

apply(*subgoal-tac max-sz* (*mem-pool-info V* (*pool b*)) *mod 4* ^ *lvl V t* = *0*)
prefer *2* **apply**(*subgoal-tac* ∃ *n*. *max-sz* (*mem-pool-info V* (*pool b*)) = (*4* ∗ *n*) ∗ (*4* ^ *n-levels* (*mem-pool-info V* (*pool b*))))
prefer *2* **apply**(*simp add*:*inv-def*) **using** *inv-mempool-info-def*[*rule-format*, *of V*] **apply** *meson*

apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*))) = *n-levels* (*mem-pool-info V* (*pool b*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def*) **apply** *metis*
apply(*simp add*: *inv-def inv-mempool-info-def*)
using *ge-pow-mod-0*[*of lvl V t n-levels* (*mem-pool-info V* (*pool b*))]
apply (*metis add-diff-inverse-nat add-lessD1 ge-pow-mod-0 le-antisym nat-less-le*)

apply(*subgoal-tac block b div 4* ˆ (*level b* − *lvl V t*) < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *lvl V t*)
prefer *2* **apply** (*metis* (*no-types*, *lifting*) *add-lessD1 inv-mempool-info-def invariant.inv-def le-Suc-ex*)

apply(*rule block-fits0-h1*[*of max-sz* (*mem-pool-info V* (*pool b*)) *4* ˆ *lvl V t*
block b div 4 ˆ (*level b* − *lvl V t*) *n-max* (*mem-pool-info V* (*pool b*))])
apply *blast* **apply** *blast*
done

lemma *mempool-free-stm8-atombody-else-inv-mempool-info*:
inv-mempool-info V ⟹
inv-mempool-info
(*V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)|))
apply(*simp add*:*inv-mempool-info-def append-free-list-def set-bit-def*) **apply** *clarify*
apply(*rule conjI*) **apply** *meson*
apply(*rule conjI*) **apply** *meson*
apply(*rule conjI*) **apply** *meson*
apply(*rule conjI*) **apply** *meson*
apply(*rule conjI*) **apply** *meson*
apply *clarify*
apply(*subgoal-tac* (∀ *i*<*length* (*levels* (*mem-pool-info V* (*pool b*))).
length (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *i*)) = *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *i*))
prefer *2* **apply**(*simp add*:*Let-def*)
apply(*case-tac i* = *lvl V t*)
by *auto*

lemma *mempool-free-stm8-atombody-else-inv-bitmap-freelist*:
inv-mempool-info V ∧ *inv-bitmap-freelist V* ∧ *inv-aux-vars V* ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ⟹
block b div 4 ˆ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) ⟹

lvl V t ≤ *level b* ⟹
freeing-node V t = *Some blka* ⟹
pool blka = *pool b* ⟹
level blka = *lvl V t* ⟹
block blka = *block b div 4* ^ (*level b* − *lvl V t*) ⟹
inv-bitmap-freelist
(*V*⦇*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)⦈)
apply(*simp add:inv-bitmap-freelist-def append-free-list-def set-bit-def block-ptr-def*)
apply *clarify*
apply(*simp add:Let-def*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info V* (*pool b*)) **and** *t*=*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))])
apply (*metis inv-mempool-info-maxsz-align4*)

apply(*rule conjI*) **apply** *clarify* **apply**(*rename-tac ii jj*)
apply(*case-tac ii* ≠ *lvl V t*) **apply** *force*
apply(*case-tac jj* = *block b div 4* ^ (*level b* − *lvl V t*))
apply *clarsimp*

apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) ! *jj* = *bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] ! *lvl V t*)
⦇*free-list* := *free-list* ((*levels* (*mem-pool-info V* (*pool b*))) [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) + *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t* ∗ (*block b div 4* ^ (*level b* − *lvl V t*))]⦈] !
ii) ! *jj*)
prefer *2* **apply** *fastforce*
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info V* (*pool b*))!*ii*)) = *length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := ((*levels* (*mem-pool-info V* (*pool b*))) [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]⦈] ! *lvl V t*)
⦇*free-list* := *free-list* ((*levels* (*mem-pool-info V* (*pool b*))) [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)

(|*bits* := (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) !
lvl *V* *t*))[*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) := *FREE*]|)] ! *lvl* *V* *t*) @
[*buf* (*mem-pool-info* *V* (*pool* *b*)) + *max-sz*
(*mem-pool-info* *V* (*pool* *b*)) *div* *4* ^ *lvl* *V* *t* ∗ (*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*))]|)]
! *ii*)))
prefer *2* **apply** *fastforce*
apply(*subgoal-tac* *free-list* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ii*) @
[*buf* (*mem-pool-info* *V* (*pool* *b*)) +
max-sz (*mem-pool-info* *V* (*pool* *b*)) *div* *4* ^ *lvl* *V* *t*
∗ (*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*))] = *free-list* ((*levels* (*mem-pool-info* *V* (*pool*
b)))
[*lvl* *V* *t* := ((*levels* (*mem-pool-info* *V* (*pool* *b*)))
[*lvl* *V* *t* := (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V* *t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) !
lvl *V* *t*))[*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) := *FREE*]|)] !
lvl *V* *t*)
(|*free-list* := *free-list* ((*levels* (*mem-pool-info* *V* (*pool*
b))) [*lvl* *V* *t* := (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *lvl* *V* *t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info* *V* (*pool*
b)) ! *lvl* *V* *t*))[*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*) := *FREE*]|)] ! *lvl* *V* *t*) @
[*buf* (*mem-pool-info* *V* (*pool* *b*)) +
max-sz (*mem-pool-info* *V* (*pool* *b*)) *div* *4* ^ *lvl* *V* *t*
∗ (*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*))]|)] ! *ii*))
prefer *2* **apply** *clarsimp*
apply(*case-tac* *bits* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ii*) ! *jj* = *FREE*)
apply(*subgoal-tac* (*buf* (*mem-pool-info* *V* (*pool* *b*)) + *jj* ∗ (*max-sz* (*mem-pool-info*
V (*pool* *b*)) *div* *4* ^ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ii*))))
prefer *2* **apply** *simp*
apply *clarsimp*

apply(*subgoal-tac* (*buf* (*mem-pool-info* *V* (*pool* *b*)) + *jj* ∗ (*max-sz* (*mem-pool-info*
V (*pool* *b*)) *div* *4* ^ *ii*)
∉ *set* (*free-list* (*levels* (*mem-pool-info* *V* (*pool* *b*)) ! *ii*))))
prefer *2* **apply** *simp*
apply(*subgoal-tac* *buf* (*mem-pool-info* *V* (*pool* *b*)) + *max-sz* (*mem-pool-info*
V (*pool* *b*)) *div* *4* ^ *lvl* *V* *t* ∗ (*block* *b* *div* *4* ^ (*level* *b* − *lvl* *V* *t*))
≠ *buf* (*mem-pool-info* *V* (*pool* *b*)) + *jj* ∗ (*max-sz* (*mem-pool-info*
V (*pool* *b*)) *div* *4* ^ *ii*))
prefer *2* **apply**(*subgoal-tac* *max-sz* (*mem-pool-info* *V* (*pool* *b*)) *div* *4* ^ *lvl*
V *t* > *0*)
prefer *2* **apply**(*simp* *add*:*inv-mempool-info-def* *Let-def*)
apply(*subgoal-tac* ∃ *n*>*NULL*. *max-sz* (*mem-pool-info* *V* (*pool*
b)) = *4* ∗ *n* ∗ *4* ^ *n-levels* (*mem-pool-info* *V* (*pool* *b*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac* *lvl* *V* *t* < *n-levels* (*mem-pool-info* *V* (*pool* *b*)))
prefer *2* **apply** *auto*[*1*]
apply(*metis* *divisors-zero* *ge-pow-mod-0* *gr0I* *mod0-div-self*
mult-0-right *power-not-zero* *zero-neq-numeral*)

apply *auto*[*1*]

apply(*subgoal-tac buf* (*mem-pool-info V* (*pool b*)) + *jj* ∗ (*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ˆ *ii*)
∉ *set* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t*
∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]))
prefer *2* **apply** *auto*[*1*]

apply *auto*[*1*]

apply(*rule conjI*)
apply *clarify* **apply**(*rename-tac ii jj*)
apply(*case-tac ii* ≠ *lvl V t*) **apply** *force*
apply(*subgoal-tac free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t*
∗ (*block b div 4* ˆ (*level b* − *lvl V t*))] = *free-list* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]⦈] !
lvl V t)
⦇*free-list* := *free-list* ((*levels* (*mem-pool-info V* (*pool b*))) [*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*]⦈] ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t*
∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]⦈] ! *ii*))
prefer *2* **apply** *clarsimp*
apply(*case-tac jj* < *length* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*)))
apply (*simp add*: *nth-append*)
apply(*case-tac jj* = *length* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*)))
apply(*subgoal-tac* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]) ! *jj*
= *buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*)))
prefer *2* **apply** *clarsimp*
apply(*subgoal-tac block b div 4* ˆ (*level b* − *lvl V t*) < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ˆ *ii*)
prefer *2* **apply** (*metis inv-mempool-info-def*)
apply *auto*[*1*]

apply *auto*[*1*]

apply(*subgoal-tac distinct* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *i*)))
prefer *2* **apply** *auto*[*1*] **apply**(*rename-tac ii*)
apply(*case-tac ii* ≠ *lvl V t*) **apply** *force*
apply(*subgoal-tac free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ^ *lvl V t*
∗ (*block b div 4* ^ (*level b* − *lvl V t*))] = *free-list* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] !
lvl V t)
(|*free-list* := *free-list* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*]|)] ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ^ *lvl V t* ∗
(*block b div 4* ^ (*level b* − *lvl V t*))]|)] ! *ii*))
prefer *2* **apply** *clarsimp*

apply(*subgoal-tac buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ^ *lvl V t* ∗ (*block b div 4* ^ (*level b* − *lvl V t*))
∉ *set* (*free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*)))
prefer *2* **apply**(*subgoal-tac get-bit* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) = *FREEING*)
prefer *2* **apply**(*simp add*:*inv-aux-vars-def*) **apply** *metis*
apply (*metis BlockState.distinct*(*15*) *semiring-normalization-rules*(*7*))
apply *auto*
done

lemma
pool b ∈ *mem-pools V* ⟹
lvl V t ≤ *level b* ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b div 4 ^ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) ⟹
V2 = *V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),

free-block-r := (*free-block-r* *V*)(*t* := *False*)⦈ ⟹
∃ *lv bl*. *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv*) = (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv*)) [*bl* := *FREE*]
∧ (∀ *lv'*. *lv* ≠ *lv'* ⟶ *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv'*) = *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv'*))
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) = (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) [*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*])
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac* ∀ *lv'*. *lvl V t* ≠ *lv'* ⟶ *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv'*) = *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv'*))
prefer *2* **apply** *clarify* **apply** *auto*[*1*]
apply(*rule exI*[**where** *x*=*lvl V t*],*auto*)
done

lemma *mempool-free-stm8-atombody-else-inv-bitmap*:
inv-bitmap V ∧ *inv-aux-vars V* ⟹
pool b ∈ *mem-pools V* ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ⟹
block b div 4 ^ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) ⟹
lvl V t ≤ *level b* ⟹
freeing-node V t = *Some blka* ⟹
pool blka = *pool b* ⟹
level blka = *lvl V t* ⟹
block blka = *block b div 4* ^ (*level b* − *lvl V t*) ⟹
V2 = *V*⦇*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)⦈ ⟹
inv-bitmap V2
apply(*unfold inv-bitmap-def*) **apply** *clarify*

apply(*case-tac p* = *pool b*)
apply(*subgoal-tac* ∃ *lv bl*. *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv*) ! *bl* = *FREEING*
∧ *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv*) = (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv*)) [*bl* := *FREE*]
∧ (∀ *lv'*. *lv* ≠ *lv'* ⟶ *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv'*) = *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv'*)))
prefer *2* **apply**(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) ! (*block b div 4* ^ (*level b* − *lvl V t*)) = *FREEING*)

prefer *2* **apply**(*simp add*:*inv-aux-vars-def*) **apply** *metis*
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lvl V t*) = (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) [*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*])
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac* ∀ *lv'*. *lvl V t* ≠ *lv'* ⟶ *bits* (*levels* (*mem-pool-info V2* (*pool b*)) ! *lv'*) = *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lv'*))
prefer *2* **apply** *clarify* **apply** *auto*[*1*]
apply(*rule exI*[**where** *x*=*lvl V t*]) **apply** *auto*[*1*]

apply(*subgoal-tac length* (*levels* (*mem-pool-info V* (*pool b*))) = *length* (*levels* (*mem-pool-info V2* (*pool b*))))
prefer *2* **apply**(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)

apply(*subgoal-tac inv-bitmap-mp V* (*pool b*)) **prefer** *2* **apply**(*simp add*:*inv-bitmap-def*)

apply(*rule subst*[**where** *s*=*V2* **and** *t*=*V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)|)]) **apply** *fast*

using *inv-bitmap-freeing2free*[*of V pool b V2*] **apply** *fast*

apply(*subgoal-tac mem-pool-info V p* = *mem-pool-info*
(*V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)|)) *p*)
prefer *2* **apply**(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*subgoal-tac mem-pools V* = *mem-pools* (*V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))),

free-block-r := (*free-block-r* *V*)(*t* := *False*)⦈))
prefer *2* **apply**(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
by (*smt BlockState.distinct*(*13*))

lemma *mempool-free-stm8-atombody-else-inv-aux-vars*:
inv-mempool-info V ∧ *inv-aux-vars V* ⟹
allocating-node V t = *None* ⟹
pool b ∈ *mem-pools V* ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ⟹
block b div 4 ˆ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) ⟹
bn V t = *block b div 4* ˆ (*level b* − *lvl V t*) ⟹
lvl V t ≤ *level b* ⟹
freeing-node V t = *Some blka* ⟹
pool blka = *pool b* ⟹
level blka = *lvl V t* ⟹
block blka = *block b div 4* ˆ (*level b* − *lvl V t*) ⟹
inv-aux-vars
(*V*⦇*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t*) (*block b div 4* ˆ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)⦈)
apply(*simp add*:*inv-aux-vars-def append-free-list-def set-bit-def block-ptr-def*) **apply** *clarify*
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info V* (*pool b*)) **and** *t*=*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))])
apply (*metis inv-mempool-info-maxsz-align4*)

apply(*rule conjI*) **apply** *clarify*
apply(*subgoal-tac* ¬(*pool n* = *pool blka* ∧ *level n* = *level blka* ∧ *block n* = *block blka*))
prefer *2* **apply** *blast*
apply(*case-tac pool n* ≠ *pool blka*) **apply** *auto*[*1*]
apply(*case-tac level n* ≠ *level blka*) **apply** (*metis* (*no-types*, *lifting*) *nth-list-update-neq*)
apply(*case-tac block n* ≠ *block blka*)
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*],
free-list :=
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +

max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]⦈] !
level n) !
block n = *bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level n*) ! *block n*)
prefer *2* **apply**(*subgoal-tac bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*],
free-list :=
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
max-sz (*mem-pool-info V* (*pool b*)) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]⦈] !
level n) = (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*])
prefer *2* **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *metis*
apply *fast*

apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level n*) ! *block n* = *FREEING*
∧ (*lvl V t* ≠ *level n* ∨ *block b div 4* ˆ (*level b* − *lvl V t*) ≠ *block n*))
prefer *2* **apply**(*case-tac lvl V t* = *level n*) **apply**(*case-tac block b div 4* ˆ (*level b* − *lvl V t*) = *block n*)
apply *clarsimp* **apply** *clarsimp* **apply** *clarsimp*
apply(*subgoal-tac mem-block-addr-valid V n*)
prefer *2* **apply**(*simp add*:*mem-block-addr-valid-def*)
apply(*subgoal-tac blka* ≠ *n*)
prefer *2* **apply** *metis*
apply (*metis option.inject*)
apply *clarify*
apply(*subgoal-tac mem-block-addr-valid V n*)
prefer *2* **apply**(*simp add*:*mem-block-addr-valid-def*)
apply (*metis option.inject*)

apply(*rule conjI*) **apply** *clarify*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
prefer *2* **apply** *blast*
apply(*case-tac lvl V t* = *level n*) **apply**(*case-tac block b div 4* ˆ (*level b* − *lvl V t*) = *block n*)
apply *metis* **apply** *clarsimp* **apply** *clarsimp*

apply *clarify*
apply(*rule conjI*) **apply** *clarify*

apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level n*) ! *block n* = *ALLOCATING*)
prefer *2* **apply**(*case-tac lvl V t* = *level n*) **apply**(*case-tac block b div 4* ^ (*level b* − *lvl V t*) = *block n*)
apply *clarsimp* **apply** *clarsimp* **apply** *clarsimp*
apply(*subgoal-tac mem-block-addr-valid V n*)
prefer *2* **apply**(*simp add*:*mem-block-addr-valid-def*)
apply *metis*

apply *clarify*
apply(*subgoal-tac mem-block-addr-valid V n*)
prefer *2* **apply**(*simp add*:*mem-block-addr-valid-def*)
apply *metis*
done

lemma *mempool-free-stm8-atombody-else-inv-bitmap0*:
inv-mempool-info V ∧ *inv-bitmap0 V* ⟹
allocating-node V t = *None* ⟹
pool b ∈ *mem-pools V* ⟹
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) ⟹
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) ⟹
block b div 4 ^ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) ⟹
bn V t = *block b div 4* ^ (*level b* − *lvl V t*) ⟹
lvl V t ≤ *level b* ⟹
freeing-node V t = *Some blka* ⟹
pool blka = *pool b* ⟹
level blka = *lvl V t* ⟹
block blka = *block b div 4* ^ (*level b* − *lvl V t*) ⟹
inv-bitmap0
(*V*⦇*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)⦈)
apply(*simp add*:*inv-bitmap0-def inv-mempool-info-def append-free-list-def set-bit-def block-ptr-def ALIGN4-def Let-def*) **apply** *clarsimp*
apply(*subgoal-tac get-bit-s V* (*pool b*) *0 i* ≠ *NOEXIST*) **prefer** *2*
apply(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
⦇*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*],
free-list :=
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
(*max-sz* (*mem-pool-info V* (*pool b*)) + *3*) *div 4* ∗ *4*

div 4 ^ lvl V t ∗ (block b div 4 ^ (level b − lvl V t))]⦈] !
0)) = length (bits (levels (mem-pool-info V (pool b)) ! 0)))
prefer *2*
apply(*case-tac lvl V t = 0*) **apply** *clarsimp*
apply(*rule subst*[**where** *s*=(*bits (levels (mem-pool-info V (pool b)) ! 0*))[*block b div 4 ^ (level b − lvl V t) := FREE*]
and *t*=*bits ((levels (mem-pool-info V (pool b)))*
[0 := (levels (mem-pool-info V (pool b)) ! 0)
⦇bits := (bits (levels (mem-pool-info V (pool b)) ! NULL))[block b div 4 ^ level b := FREE],
free-list :=
free-list (levels (mem-pool-info V (pool b)) ! NULL) @
[buf (mem-pool-info V (pool b)) + (max-sz (mem-pool-info V (pool b)) + 3) div 4 ∗ 4 ∗ (block b div 4 ^ level b)]⦈] !
0)]) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]
apply *auto*[*1*]

apply(*case-tac lvl V t = 0*)
apply(*case-tac i = block b div 4 ^ (level b − lvl V t)*)
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
done

lemma *mempool-free-stm8-atombody-else-inv-bitmapn*:
inv-mempool-info V ∧ inv-bitmapn V ⟹
allocating-node V t = None ⟹
pool b ∈ mem-pools V ⟹
level b < length (levels (mem-pool-info V (pool b))) ⟹
block b < length (bits (levels (mem-pool-info V (pool b)) ! level b)) ⟹
block b div 4 ^ (level b − lvl V t) < length (bits (levels (mem-pool-info V (pool b)) ! lvl V t)) ⟹
bn V t = block b div 4 ^ (level b − lvl V t) ⟹
lvl V t ≤ level b ⟹
freeing-node V t = Some blka ⟹
pool blka = pool b ⟹
level blka = lvl V t ⟹
block blka = block b div 4 ^ (level b − lvl V t) ⟹
inv-bitmapn
(V⦇freeing-node := (freeing-node V)(t := None),
mem-pool-info := (set-bit-free (mem-pool-info V) (pool b) (lvl V t) (block b div 4 ^ (level b − lvl V t)))
(pool b := append-free-list (set-bit-free (mem-pool-info V) (pool b) (lvl V t) (block b div 4 ^ (level b − lvl V t)) (pool b)) (lvl V t)
(block-ptr (mem-pool-info V (pool b)) (ALIGN4 (max-sz (mem-pool-info V (pool b))) div 4 ^ lvl V t) (block b div 4 ^ (level b − lvl V t)))),
free-block-r := (free-block-r V)(t := False)⦈)
apply(*simp add:inv-bitmapn-def inv-mempool-info-def append-free-list-def set-bit-def block-ptr-def ALIGN4-def Let-def*) **apply** *clarsimp*

apply(*subgoal-tac get-bit-s V (pool b) (length (levels (mem-pool-info V (pool b))) − Suc 0) i ≠ DIVIDED*) **prefer** *2*
apply(*subgoal-tac length (bits ((levels (mem-pool-info V (pool b)))
[lvl V t := (levels (mem-pool-info V (pool b)) ! lvl V t)
(|bits := (bits (levels (mem-pool-info V (pool b)) ! lvl V t))[block b div 4 ˆ (level b − lvl V t) := FREE],
free-list :=
free-list (levels (mem-pool-info V (pool b)) ! lvl V t) @
[buf (mem-pool-info V (pool b)) +
(max-sz (mem-pool-info V (pool b)) + 3) div 4 ∗ 4 div 4 ˆ lvl V t ∗ (block b div 4 ˆ (level b − lvl V t))]|)] !
(length (levels (mem-pool-info V (pool b))) − Suc 0)))
= length (bits (levels (mem-pool-info V (pool b)) ! (length (levels (mem-pool-info V (pool b))) − Suc 0)))*) **prefer** *2*
apply(*case-tac lvl V t = (length (levels (mem-pool-info V (pool b))) − Suc 0)*)
apply *clarsimp*
apply(*rule subst*[**where** *s=(bits (levels (mem-pool-info V (pool b)) ! (length (levels (mem-pool-info V (pool b))) − Suc 0)))
[block b div 4 ˆ (level b − lvl V t) := FREE]*
and *t=bits ((levels (mem-pool-info V (pool b)))
[(length (levels (mem-pool-info V (pool b))) − Suc 0) :=
(levels (mem-pool-info V (pool b)) ! (length (levels (mem-pool-info V (pool b))) − Suc 0))
(|bits := (bits (levels (mem-pool-info V (pool b)) ! NULL))[block b div 4 ˆ level b := FREE],
free-list :=
free-list (levels (mem-pool-info V (pool b)) ! NULL) @
[buf (mem-pool-info V (pool b)) + (max-sz (mem-pool-info V (pool b)) + 3) div 4 ∗ 4 ∗ (block b div 4 ˆ level b)]|)] !
(length (levels (mem-pool-info V (pool b))) − Suc 0)*]) **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*case-tac lvl V t = (length (levels (mem-pool-info V (pool b))) − Suc 0)*)
apply(*case-tac i = block b div 4 ˆ (level b − lvl V t)*)
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
done

lemma *mempool-free-stm8-atombody-else-inv-bitmap-not4free*:
*lvl V t = NULL ∨
¬ partner-bits (set-bit-free (mem-pool-info V) (pool b) (lvl V t) (block b div 4 ˆ (level b − lvl V t)) (pool b)) (lvl V t)
(block b div 4 ˆ (level b − lvl V t)) ⟹
inv-mempool-info V ∧ inv-bitmap-not4free V ⟹
allocating-node V t = None ⟹
pool b ∈ mem-pools V ⟹
level b < length (levels (mem-pool-info V (pool b))) ⟹
block b < length (bits (levels (mem-pool-info V (pool b)) ! level b)) ⟹
block b div 4 ˆ (level b − lvl V t) < length (bits (levels (mem-pool-info V (pool*

b)) ! *lvl V t*)) $\Longrightarrow$
bn V t = *block b div 4* ^ (*level b* − *lvl V t*) $\Longrightarrow$
lvl V t $\leq$ *level b* $\Longrightarrow$
freeing-node V t = *Some blka* $\Longrightarrow$
pool blka = *pool b* $\Longrightarrow$
level blka = *lvl V t* $\Longrightarrow$
block blka = *block b div 4* ^ (*level b* − *lvl V t*) $\Longrightarrow$
inv-bitmap-not4free
(*V*(|*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)|))
apply(*simp add*:*inv-bitmap-not4free-def inv-mempool-info-def append-free-list-def set-bit-def block-ptr-def ALIGN4-def Let-def*)
apply *clarsimp*

apply(*case-tac lvl V t* = *0*)
apply *clarsimp*
apply(*simp add*:*partner-bits-def Let-def*) **apply** *auto*[*1*]

apply *clarsimp*
apply(*case-tac i* = *lvl V t*)
apply(*simp add*:*partner-bits-def Let-def*)
apply *clarsimp*
apply(*case-tac j div 4* = *block b div 4* ^ (*level b* − *lvl V t*) *div 4*)
apply *auto*[*1*]
apply *auto*[*1*]

apply(*simp add*:*partner-bits-def Let-def*)
apply *clarsimp*
done

lemma *mempool-free-stm8-atombody-else-inv*:
lvl V t = *NULL* $\vee$
$\neg$ *partner-bits* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block b div 4* ^ (*level b* − *lvl V t*)) $\Longrightarrow$
inv V $\Longrightarrow$
allocating-node V t = *None* $\Longrightarrow$
pool b $\in$ *mem-pools V* $\Longrightarrow$
level b < *length* (*levels* (*mem-pool-info V* (*pool b*))) $\Longrightarrow$
block b < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*)) $\Longrightarrow$
data b = *block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info*

V (*pool b*))) *div 4* ^ *level b*) (*block b*) $\Longrightarrow$
level b < *length* (*lsizes V t*) $\Longrightarrow$
∀ *ii*<*length* (*lsizes V t*). *lsizes V t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *ii* $\Longrightarrow$
block b div 4 ^ (*level b* − *lvl V t*) < *length* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)) $\Longrightarrow$
bn V t = *block b div 4* ^ (*level b* − *lvl V t*) $\Longrightarrow$
lvl V t ≤ *level b* $\Longrightarrow$
free-block-r V t $\Longrightarrow$
lsz V t = *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t* $\Longrightarrow$
blk V t = *block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) $\Longrightarrow$
cur V = *Some t* $\Longrightarrow$
data blka = *buf* (*mem-pool-info V* (*pool b*)) + *block b div 4* ^ (*level b* − *lvl V t*) ∗ (*max-sz* (*mem-pool-info V* (*pool b*)) *div 4* ^ *lvl V t*) $\Longrightarrow$
block b div 4 ^ (*level b* − *lvl V t*) < *n-max* (*mem-pool-info V* (*pool b*)) ∗ *4* ^ *lvl V t* $\Longrightarrow$
freeing-node V t = *Some y* $\Longrightarrow$
pool y = *pool b* $\Longrightarrow$
level y = *lvl V t* $\Longrightarrow$
block y = *block b div 4* ^ (*level b* − *lvl V t*) $\Longrightarrow$
inv
(*V*⦇*freeing-node* := (*freeing-node V*)(*t* := *None*),
mem-pool-info := (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))
(*pool b* := *append-free-list* (*set-bit-free* (*mem-pool-info V*) (*pool b*) (*lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)) (*pool b*)) (*lvl V t*)
(*block-ptr* (*mem-pool-info V* (*pool b*)) (*ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ^ *lvl V t*) (*block b div 4* ^ (*level b* − *lvl V t*)))),
free-block-r := (*free-block-r V*)(*t* := *False*)⦈)
apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def Mem-pool-free-guar-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def append-free-list-def set-bit-def*)
apply *smt*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-mempool-info* **apply** *blast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-bitmap-freelist* **apply** *blast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-bitmap* **apply** *blast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-aux-vars* **apply** *blast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-bitmap0* **apply** *blast*
apply(*rule conjI*) **using** *mempool-free-stm8-atombody-else-inv-bitmapn* **apply** *blast*
using *mempool-free-stm8-atombody-else-inv-bitmap-not4free* **apply** *blast*
done

lemma *mp-free-stm8-intI*:
$\{V\} \subseteq \{\!|\,\acute{}(\textit{free-block-r-update}\ (\lambda\text{-}.\ \acute{}\textit{free-block-r}(t := \textit{False}))) \in A|\!\} \Longrightarrow$
$\{V\} \subseteq \{\!|\,\acute{}(\textit{free-block-r-update}\ (\lambda\text{-}.\ \acute{}\textit{free-block-r}(t := \textit{False}))) \in B|\!\} \Longrightarrow$
$\{V\} \subseteq \{\!|\,\acute{}(\textit{free-block-r-update}\ (\lambda\text{-}.\ \acute{}\textit{free-block-r}(t := \textit{False}))) \in A \cap B|\!\}$
by *auto*

lemma *mempool-free-stm8-atombody-else-h1*:
$V \in \{\!|\,\acute{}\textit{free-block-r}\ t|\!\} \cap \textit{mp-free-precond8-3}\ t\ b\ \alpha \cap \{\!|\,\acute{}\textit{cur} = \textit{Some}\ t|\!\} \Longrightarrow$
$\{\textit{free-stm8-precond2}\ V\ t\ b\} \cap - \{\!|\textit{NULL} < \acute{}\textit{lvl}\ t \wedge \textit{partner-bits}\ (\acute{}\textit{mem-pool-info}\ (\textit{pool}\ b))\ (\acute{}\textit{lvl}\ t)\ (\acute{}\textit{bn}\ t)|\!\} \neq \{\} \Longrightarrow$
$\{\textit{let}\ V2 = \textit{free-stm8-precond2}\ V\ t\ b\ \textit{in}\ V2(\!|\textit{mem-pool-info} := (\textit{mem-pool-info}\ V2)$
$(\textit{pool}\ b := \textit{append-free-list}\ (\textit{mem-pool-info}\ V2\ (\textit{pool}\ b))\ (\textit{lvl}\ V2\ t)\ (\textit{blk}\ V2\ t))|\!)\}$
$\subseteq \{\!|\,\acute{}(\textit{free-block-r-update}\ (\lambda\text{-}.\ \acute{}\textit{free-block-r}(t := \textit{False})))$
$\in \{\!|\,\acute{}(\textit{Pair}\ V) \in \textit{Mem-pool-free-guar}\ t|\!\} \cap \textit{mp-free-precond8-inv}\ t\ b\ 0|\!\}$
apply(*rule mp-free-stm8-intI*)

apply(*simp add:Mem-pool-free-guar-def*)
apply(*rule disjI1*) **apply**(*rule conjI*)
apply(*simp add:gvars-conf-stable-def gvars-conf-def append-free-list-def set-bit-def block-ptr-def*) **apply** *clarify*
apply(*rename-tac ii blk*)
apply(*case-tac lvl V t = ii*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*] =
bits ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*],
free-list := *free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) + *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))
div 4 ^ *lvl V t* ∗ (*block b div 4* ^ (*level b* − *lvl V t*))]|)] !
ii))
prefer *2* **apply** *auto*[*1*]
apply (*metis length-list-update*)
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *ii*) =
bits ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ^ (*level b* − *lvl V t*) := *FREE*],
free-list := *free-list* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) + *ALIGN4* (*max-sz* (*mem-pool-info V* (*pool b*)))
div 4 ^ *lvl V t* ∗ (*block b div 4* ^ (*level b* − *lvl V t*))]|)] !
ii))
prefer *2* **apply** *auto*[*1*]

apply *auto*[*1*]
apply(*rule conjI*) **apply** *clarsimp*
using *mempool-free-stm8-atombody-else-inv*[*of V t b*] **apply** *metis*

apply *clarify* **apply**(*simp add*:*lvars-nochange-def*)

apply(*rule mp-free-stm8-intI*)
apply *clarsimp*
apply(*rule conjI*)
using *mempool-free-stm8-atombody-else-inv*[*of V t b*] **apply** *metis*
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*case-tac lvl V t* = *level b*)
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*] =
bits ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*],
free-list :=
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
ALIGN4 (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]|)] !
level b))
prefer *2* **apply** *auto*[*1*]
apply (*metis length-list-update*)
apply(*subgoal-tac bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *level b*) =
bits ((*levels* (*mem-pool-info V* (*pool b*)))
[*lvl V t* := (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*)
(|*bits* := (*bits* (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*))[*block b div 4* ˆ (*level b* − *lvl V t*) := *FREE*],
free-list :=
free-list (*levels* (*mem-pool-info V* (*pool b*)) ! *lvl V t*) @
[*buf* (*mem-pool-info V* (*pool b*)) +
ALIGN4 (*max-sz* (*mem-pool-info V* (*pool b*))) *div 4* ˆ *lvl V t* ∗ (*block b div 4* ˆ (*level b* − *lvl V t*))]|)] !
level b))
prefer *2* **apply** *auto*[*1*]
apply *metis*
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)
apply(*simp add*:*append-free-list-def set-bit-def block-ptr-def*)

apply *clarsimp*

done

lemma *mempool-free-stm8-atombody-else'*:
$V \in \{\!|\acute{}free\text{-}block\text{-}r\ t|\!\} \cap mp\text{-}free\text{-}precond8\text{-}3\ t\ b\ \alpha \cap \{\!|\acute{}cur = Some\ t|\!\} \Longrightarrow$
$\Gamma \vdash_I$ *Some* (*IF block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *append-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*blk t*))
FI;;
´*free-block-r* := ´*free-block-r* (*t* := *False*))
sat_p [{*free-stm8-precond2 V t b*} ∩ − {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|},
{(*s*, *t*). *s* = *t*}, *UNIV*, {|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ *mp-free-precond8-inv t b 0*]
apply(*case-tac* {*free-stm8-precond2 V t b*} ∩ − {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} = {})
using *Emptyprecond*[*of Some* (*IF block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *append-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*blk t*))
FI;;
´*free-block-r* := ´*free-block-r* (*t* := *False*)) {(*s*, *t*). *s* = *t*}
UNIV {|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ *mp-free-precond8-inv t b 0*] **apply** *metis*

apply(*rule Seq*[**where** *mid*={*let V2* = *free-stm8-precond2 V t b in V2*(|*mem-pool-info* := (*mem-pool-info V2*)
(*pool b* := *append-free-list* (*mem-pool-info V2* (*pool b*)) (*lvl V2 t*) (*blk V2 t*)))|)}])

apply(*rule Cond*)
using *stable-id2* **apply** *fast*

apply(*rule subst*[**where** *s*={} **and** *t*={*free-stm8-precond2 V t b*} ∩ − {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ∩
− {|*block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*)
(´*lsz t*)|}])
using *mempool-free-stm8-atombody-else-blockfit*[*of V t b α*] **apply** *fast*

apply(*rule Basic*) **apply**(*simp add*:*Let-def*)
apply *auto*[*1*] **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule subst*[**where** *s*={} **and** *t*={*free-stm8-precond2 V t b*} ∩ − {|*NULL* < ´*lvl t* ∧ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} ∩
− {|*block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*)
(´*lsz t*)|}])
using *mempool-free-stm8-atombody-else-blockfit*[*of V t b α*] **apply** *fast*
using *Emptyprecond* **apply** *blast*

apply *fast*

apply(*rule Basic*)
using *mempool-free-stm8-atombody-else-h1*[*of V t b* α] **apply** *fast*

apply *fast*
using *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*
done

lemma *mempool-free-stm8-atombody-else*:
$V \in$ *mp-free-precond8-3 t b* $\alpha \cap$ {|´*cur* = *Some t*|} $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (*IF block-fits* (´*mem-pool-info* (*pool b*)) (´*blk t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := *append-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*blk t*))
FI;;
´*free-block-r* := ´*free-block-r* (*t* := *False*))
sat_p [{*free-stm8-precond2 V t b*} $\cap -$ {|*NULL* < ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|},
{(*s*, *t*). *s* = *t*}, *UNIV*, {|´(*Pair V*) $\in$ *Mem-pool-free-guar t*|} $\cap$ *mp-free-precond8-inv t b 0*]
apply(*subgoal-tac V*$\in${|´*free-block-r t*|} $\cap$ *mp-free-precond8-3 t b* $\alpha \cap$ {|´*cur* = *Some t*|})
prefer *2* **apply**(*subgoal-tac mp-free-precond8-1 t b* α = {|´*free-block-r t*|} $\cap$ *mp-free-precond8-1 t b* α)
prefer *2* **using** *mp-free-precond8-1-imp-free-block-r*[*of t b* α] *Int-absorb1*[*of mp-free-precond8-1 t b* α {|´*free-block-r t*|}] **apply** *metis*
apply *auto*[*1*]
using *mempool-free-stm8-atombody-else*′[*of V t b* α] **apply** *metis*
done

lemma *mempool-free-stm8-atombody-rest-extpost*:
$V \in$*mp-free-precond8-3 t b* $\alpha \cap$ {|´*cur* = *Some t*|} $\Longrightarrow$
{*free-stm8-precond2 V t b*} $\cap$ {|*NULL* < ´*lvl t* $\wedge$ *partner-bits* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*bn t*)|} $\neq$ {} $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*lvl* := ´*lvl*(*t* := ´*lvl t* − *1*);;
´*bn* := ´*bn*(*t* := ´*bn t div 4*);;
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node*(*t* $\mapsto$(|*pool* = *pool b*, *level* = ´*lvl t*, *block* = ´*bn t*,
data = *block-ptr* (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ^ ´*lvl t*) (´*bn t*)|)))
sat_p [*free-stm8-precond3 V t b* $\cap$ {|´*i t* = *4*|}, {(*s*, *t*). *s* = *t*}, *UNIV*,
{|´(*Pair V*) $\in$ *Mem-pool-free-guar t*|} $\cap$ (*mp-free-precond8-inv t b* (α − *1*) $\cup$ *mp-free-precond8-inv t b 0*)]
apply(*rule Conseq*[*of free-stm8-precond3 V t b* $\cap$ {|´*i t* = *4*|} *free-stm8-precond3 V t b* $\cap$ {|´*i t* = *4*|}
{(*s*, *t*). *s* = *t*} {(*s*, *t*). *s* = *t*} *UNIV UNIV*

$\{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha - 1)$
$\{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap (mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha - 1) \cup mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ 0)$
Some $(\acute{}lvl := \acute{}lvl(t := \acute{}lvl\ t - 1);;$
$\acute{}bn := \acute{}bn(t := \acute{}bn\ t\ div\ 4);;$
$\acute{}mem\text{-}pool\text{-}info := set\text{-}bit\text{-}freeing\ \acute{}mem\text{-}pool\text{-}info\ (pool\ b)\ (\acute{}lvl\ t)\ (\acute{}bn\ t);;$
$\acute{}freeing\text{-}node := \acute{}freeing\text{-}node(t \mapsto (\!|pool = pool\ b,\ level = \acute{}lvl\ t,\ block = \acute{}bn\ t,$
$data = block\text{-}ptr\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b)))\ div\ 4\ \hat{}\ \acute{}lvl\ t)\ (\acute{}bn\ t)|\!)))])$
apply *fast* **apply** *fast* **apply** *fast* **apply** *auto*[*1*]
using *mempool-free-stm8-atombody-rest* **apply** *blast*
done

lemma *mempool-free-stm8-atombody-else-extpost*:
$V\in mp\text{-}free\text{-}precond8\text{-}3\ t\ b\ \alpha \cap \{\!|\acute{}cur = Some\ t|\!\} \Longrightarrow$
$\Gamma \vdash_I Some\ (IF\ block\text{-}fits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}blk\ t)\ (\acute{}lsz\ t)\ THEN$
$\acute{}mem\text{-}pool\text{-}info := \acute{}mem\text{-}pool\text{-}info(pool\ b := append\text{-}free\text{-}list\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}blk\ t))$
$FI;;$
$\acute{}free\text{-}block\text{-}r := \acute{}free\text{-}block\text{-}r\ (t := False)\ )$
$sat_p\ [\{free\text{-}stm8\text{-}precond2\ V\ t\ b\} \cap - \{\!|NULL < \acute{}lvl\ t \wedge partner\text{-}bits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}bn\ t)|\!\},$
$\{(s,\ t).\ s = t\},\ UNIV,\ \{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\}$
$\cap (mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha - 1) \cup mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ 0)]$
apply(*rule Conseq*[*of* $\{free\text{-}stm8\text{-}precond2\ V\ t\ b\} \cap - \{\!|NULL < \acute{}lvl\ t \wedge partner\text{-}bits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}bn\ t)|\!\}$
$\{free\text{-}stm8\text{-}precond2\ V\ t\ b\} \cap - \{\!|NULL < \acute{}lvl\ t \wedge partner\text{-}bits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}bn\ t)|\!\}$
$\{(s,\ t).\ s = t\}\ \{(s,\ t).\ s = t\}\ UNIV\ UNIV$
$\{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ 0$
$\{\!|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}free\text{-}guar\ t|\!\} \cap (mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ (\alpha - 1) \cup mp\text{-}free\text{-}precond8\text{-}inv\ t\ b\ 0)$
$Some\ (IF\ block\text{-}fits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}blk\ t)\ (\acute{}lsz\ t)\ THEN$
$\acute{}mem\text{-}pool\text{-}info := \acute{}mem\text{-}pool\text{-}info(pool\ b := append\text{-}free\text{-}list\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}blk\ t))$
$FI;;$
$\acute{}free\text{-}block\text{-}r := \acute{}free\text{-}block\text{-}r\ (t := False)\ )])$
apply *fast* **apply** *fast* **apply** *fast* **apply** *auto*[*1*]
using *mempool-free-stm8-atombody-else* **apply** *blast*
done

lemma *mempool-free-stm8-atombody*:
$\Gamma \vdash_I Some\ (\acute{}mem\text{-}pool\text{-}info := set\text{-}bit\text{-}free\ \acute{}mem\text{-}pool\text{-}info\ (pool\ b)\ (\acute{}lvl\ t)\ (\acute{}bn\ t);;$
$\acute{}freeing\text{-}node := \acute{}freeing\text{-}node(t := None);;$
$IF\ NULL < \acute{}lvl\ t \wedge partner\text{-}bits\ (\acute{}mem\text{-}pool\text{-}info\ (pool\ b))\ (\acute{}lvl\ t)\ (\acute{}bn\ t)$

THEN ´*i* := ´*i*(*t* := *0*);;
WHILE ´*i t* < *4*
DO ´*bb* := ´*bb*(*t* := ´*bn t div 4* ∗ *4* + ´*i t*);;
´*mem-pool-info* := *set-bit-noexist* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bb t*);;
´*block-pt* := ´*block-pt*(*t* := *block-ptr* (´*mem-pool-info* (*pool b*)) (´*lsz t*) (´*bb t*));;
IF ´*bn t* ≠ ´*bb t* ∧
block-fits (´*mem-pool-info* (*pool b*)) (´*block-pt t*)
(´*lsz t*) *THEN* ´*mem-pool-info* := ´*mem-pool-info*
(*pool b* := *remove-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*block-pt t*)) *FI*;;
´*i* := ´*i*(*t* := *Suc* (´*i t*))
OD;;
(´*lvl* := ´*lvl*(*t* := ´*lvl t* − *1*);; ´*bn* := ´*bn*(*t* := ´*bn t div 4*);;
´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node*(*t* ↦
(|*pool* = *pool b*, *level* = ´*lvl t*, *block* = ´*bn t*,
data = *block-ptr* (´*mem-pool-info* (*pool b*)) (*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*))) *div 4* ˆ ´*lvl t*) (´*bn t*)|)))
ELSE IF block-fits (´*mem-pool-info* (*pool b*)) (´*blk t*)
(´*lsz t*) *THEN* ´*mem-pool-info* := ´*mem-pool-info*
(*pool b* := *append-free-list* (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*blk t*)) *FI*;;
´*free-block-r* := ´*free-block-r*(*t* := *False*)
FI) sat_p [*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ∩ {*V*} ∩ *UNIV* ∩ {*Va*}, {(*s*, *t*). *s* = *t*}, *UNIV*,
{|´(*Pair Va*) ∈ *UNIV*|} ∩ ({|´(*Pair V*) ∈ *Mem-pool-free-guar t*|}
∩ (*mp-free-precond8-inv t b* (α − *1*) ∪ *mp-free-precond8-inv t b 0*))]

apply(*rule subst*[**where** *s*=*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ∩ {*V*} ∩ {*Va*}
and *t*=*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ∩ {*V*} ∩ *UNIV* ∩ {*Va*}])
apply *blast*
apply(*rule subst*[**where** *s*={|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ (*mp-free-precond8-inv t b* (α − *1*))
and *t*={|´(*Pair Va*) ∈ *UNIV*|} ∩ ({|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ (*mp-free-precond8-inv t b* (α − *1*)))])
apply *blast*

apply(*case-tac V* ≠ *Va*)
apply(*rule subst*[**where** *s*={} **and** *t*=*mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ∩ {*V*} ∩ {*Va*}])
apply *fast* **using** *Emptyprecond*[*of* - {(*s*, *t*). *s* = *t*} *UNIV*
{|´(*Pair Va*) ∈ *UNIV*|} ∩ ({|´(*Pair V*) ∈ *Mem-pool-free-guar t*|} ∩ (*mp-free-precond8-inv t b* (α − *1*) ∪ *mp-free-precond8-inv t b 0*))] **apply** *auto*[*1*]

apply(*case-tac mp-free-precond8-3 t b* α ∩ {|´*cur* = *Some t*|} ∩ {*V*} ∩ {*Va*} = {})

using *Emptyprecond* **apply** *metis*

apply(*rule subst*[**where** $s=\{V\}$ **and** $t=$*mp-free-precond8-3 t b* $\alpha \cap \{\!|\acute{}cur = Some\ t|\!\} \cap \{V\} \cap \{Va\}$])
using *two-int-one*[*of mp-free-precond8-3 t b* $\alpha \cap \{\!|\acute{}cur = Some\ t|\!\}$ *V Va*] **apply** *fast*
apply(*subgoal-tac* $V \in$*mp-free-precond8-3 t b* $\alpha \cap \{\!|\acute{}cur = Some\ t|\!\}$)
prefer *2* **apply** *fast*

apply(*rule Seq*[**where** *mid*=$\{$*free-stm8-precond2 V t b*$\}$])
apply(*rule Seq*[**where** *mid*=$\{$*free-stm8-precond1 V t b*$\}$])

apply(*rule Basic*)
apply *force*
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Basic*)
using *mempool-free-stm8-atombody-h1* **apply** *fast*
using *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule Cond*)
using *stable-id2* **apply** *fast*
apply(*case-tac* $\{$*free-stm8-precond2 V t b*$\} \cap \{\!|NULL < \acute{}lvl\ t \wedge$ *partner-bits* ($\acute{}$*mem-pool-info* (*pool b*)) ($\acute{}$*lvl t*) ($\acute{}$*bn t*)$|\!\} = \{\}$)
using *Emptyprecond* **apply** *metis*

apply(*rule Seq*[**where** *mid*=*free-stm8-precond4 Va t b*])
apply(*rule Seq*[**where** *mid*=*free-stm8-precond3 Va t b*])

apply(*rule Basic*)
apply *simp* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*rule While*)
using *stable-id2* **apply** *fast* **apply**(*simp add*:*Let-def*) **apply** *auto*[*1*] **using** *stable-id2* **apply** *fast*
using *mempool-free-stm8-set4partbits-while*[*of V Va t b* α] **apply** *fast*
apply *fast*

apply(*rule subst*[**where** $s=\{\!|\acute{}(Pair\ V) \in$ *Mem-pool-free-guar t*$|\!\}$
$\cap$ (*mp-free-precond8-inv t b* $(\alpha - 1) \cup$ *mp-free-precond8-inv t b 0*)
and $t=\{\!|\acute{}(Pair\ Va) \in UNIV|\!\} \cap (\{\!|\acute{}(Pair\ V) \in$ *Mem-pool-free-guar*

t⦄
∩ (mp-free-precond8-inv t b (α − 1) ∪ mp-free-precond8-inv
t b 0))])
apply auto[1]
using mempool-free-stm8-atombody-rest-extpost[of V t b α] **apply** fast
apply(rule subst[**where** s=⦃´(Pair V) ∈ Mem-pool-free-guar t⦄
∩ (mp-free-precond8-inv t b (α − 1) ∪ mp-free-precond8-inv
t b 0)
and t=⦃´(Pair Va) ∈ UNIV⦄ ∩ (⦃´(Pair V) ∈ Mem-pool-free-guar
t⦄
∩ (mp-free-precond8-inv t b (α − 1) ∪ mp-free-precond8-inv
t b 0))])
apply auto[1]
using mempool-free-stm8-atombody-else-extpost[of V t b α] **apply** fast

apply fast
done

lemma {(s,t). s = t} = Id **by** auto

abbreviation st8-while-body t b ≡
(t ► ´lsz := ´lsz (t := ´lsizes t ! (´lvl t)));;
(t ► ´blk := ´blk (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bn t)));;

(t ► ATOMIC

´mem-pool-info := set-bit-free ´mem-pool-info (pool b) (´lvl t) (´bn t);;
´freeing-node := ´freeing-node (t := None);;

IF ´lvl t > 0 ∧ partner-bits (´mem-pool-info (pool b)) (´lvl t) (´bn t) THEN
FOR ´i := ´i(t := 0);
´i t < 4;
´i := ´i(t := ´i t + 1) DO
´bb := ´bb (t := (´bn t div 4) ∗ 4 + ´i t);;
´mem-pool-info := set-bit-noexist ´mem-pool-info (pool b) (´lvl t) (´bb t);;
´block-pt := ´block-pt (t := block-ptr (´mem-pool-info (pool b)) (´lsz t) (´bb
t));;
IF ´bn t ≠ ´bb t ∧ block-fits (´mem-pool-info (pool b))
(´block-pt t)
(´lsz t) THEN

´mem-pool-info := ´mem-pool-info ((pool b) :=
remove-free-list (´mem-pool-info (pool b)) (´lvl t) (´block-pt t))
FI
ROF;;

(
´lvl := ´lvl (t := ´lvl t − 1);;
´bn := ´bn (t := ´bn t div 4);;

´*mem-pool-info* := *set-bit-freeing* ´*mem-pool-info* (*pool b*) (´*lvl t*) (´*bn t*);;
´*freeing-node* := ´*freeing-node* (*t* := *Some* ⦇*pool* = (*pool b*), *level* = (´*lvl t*),
block = (´*bn t*),
data = *block-ptr* (´*mem-pool-info* (*pool b*))
(((*ALIGN4* (*max-sz* (´*mem-pool-info* (*pool b*)))) *div* (*4* ^ (´*lvl*
t))))
(´*bn t*) ⦈)
)

ELSE
IF block-fits (´*mem-pool-info* (*pool b*)) (´*blk t*) (´*lsz t*) *THEN*

´*mem-pool-info* := ´*mem-pool-info* ((*pool b*) :=
append-free-list (´*mem-pool-info* (*pool b*)) (´*lvl t*) (´*blk t*))
FI;;

´*free-block-r* := ´*free-block-r* (*t* := *False*)
FI

END)

lemma *mp-free-precond8-inv-0-stb* :
stable (*mp-free-precond8-inv t b* $(\alpha - 1)$ ∪ *mp-free-precond8-inv t b 0*) (*Mem-pool-free-rely*
t)
apply(*rule stable-un2*)
using *mp-free-precond8-inv-stb*[*of t b* $\alpha - 1$] **apply** *fast*
using *mp-free-precond8-inv-stb*[*of t b 0*] **apply** *fast*
done

lemma *mempool-free-stm8-body-terminate*:
$\Gamma \vdash_I$ *Some* (*st8-while-body t b*)
sat_p [*mp-free-precond8-inv t b* α ∩ ⦃α > *0*⦄, *Mem-pool-free-rely t*, *Mem-pool-free-guar*
t,
mp-free-precond8-inv t b $(\alpha - 1)$ ∪ *mp-free-precond8-inv t b 0*]
apply(*rule Seq*[**where** *mid*=*mp-free-precond8-3 t b* α])
apply(*rule Seq*[**where** *mid*=*mp-free-precond8-2 t b* α])

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-free-precond8-1-stb*[*of t b* α] **apply** *blast*
using *mp-free-precond8-2-stb*[*of t b* α] **apply** *blast*

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-free-precond8-1 t b* α ∩ ⦃´*cur* = *Some t*⦄ ∩ {*V*} = {})
apply *auto*[*1*] **apply** *clarsimp* **apply**(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)

apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V, V(\|lsz := (lsz\ V)(t := lsizes\ V\ t\ !\ (lvl\ V\ t))\|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* $(V, V(\|lsz := (lsz\ V)(t := lsizes\ V\ t\ !\ (lvl\ V\ t))\|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-free-precond8-2-stb* **apply** *blast*
using *mp-free-precond8-3-stb* **apply** *blast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-free-precond8-2 t b* $\alpha \cap \{\!\!\{ {}^{\acute{}}cur = Some\ t \}\!\!\} \cap \{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarsimp* **apply**(*rule conjI*)
apply(*simp add*: *gvars-conf-stable-def gvars-conf-def Mem-pool-free-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V, V(\|blk := (blk\ V)$
$(t := block\text{-}ptr\ (mem\text{-}pool\text{-}info\ V\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ (pool\ b)))\ div\ 4\ \hat{}\ lvl\ V\ t)$
$(block\ b\ div\ 4\ \hat{}\ (level\ b - lvl\ V\ t)))\|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* $(V, V(\|blk := (blk\ V)$
$(t := block\text{-}ptr\ (mem\text{-}pool\text{-}info\ V\ (pool\ b))\ (ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ (pool\ b)))\ div\ 4\ \hat{}\ lvl\ V\ t)$
$(block\ b\ div\ 4\ \hat{}\ (level\ b - lvl\ V\ t)))\|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-free-precond8-3-stb* **apply** *blast*
using *mp-free-precond8-inv-0-stb*[*of t b* α] **apply** *fast*
apply(*rule allI*)
apply(*rule Await*)
using *stable-id2* **apply** *blast* **using** *stable-id2* **apply** *blast*
apply *clarify* **using** *mempool-free-stm8-atombody*[*of b t* α] **apply** *auto*[*1*]

done

lemma *loopbody-sat-invterm-imp-inv-post'*:
$\Gamma \vdash_I$ *Some P sat*$_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$, *rely*, *guar*, *mp-free-precond8-inv t b* $(\alpha - 1) \cup$ *mp-free-precond8-inv t b 0*]
$\Longrightarrow \Gamma \vdash_I$ *Some P sat*$_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$, *rely*, *guar*,*mp-free-precond8 t b*]
using *Conseq* [*of mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$ *mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$
rely rely guar guar mp-free-precond8-inv t b $(\alpha - 1) \cup$ *mp-free-precond8-inv t b 0*
mp-free-precond8 t b Some P] **by** *blast*

lemma *stm8-inv-imp-prepost2'*:
$(\forall \alpha.\ \Gamma \vdash_I$ *Some P sat*$_p$ [*mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$, *rely*, *guar*,
mp-free-precond8-inv t b $(\alpha - 1) \cup$ *mp-free-precond8-inv t b 0*])
$\Longrightarrow \Gamma \vdash_I$ *Some P sat*$_p$ [*mp-free-precond8 t b* $\cap \{\!|$´*free-block-r t*$|\!\}$, *rely*, *guar*,*mp-free-precond8 t b*]

apply(*rule subst*[**where** *s*=$\forall v.\ v \in$*mp-free-precond8 t b* $\cap \{\!|$´*free-block-r t*$|\!\} \longrightarrow$
$\Gamma \vdash_I$ *Some P sat*$_p$ [$\{v\}$, *rely*, *guar*,*mp-free-precond8 t b*] **and**
t=$\Gamma \vdash_I$ *Some P sat*$_p$ [*mp-free-precond8 t b* $\cap \{\!|$´*free-block-r t*$|\!\}$, *rely*, *guar*,*mp-free-precond8 t b*]])
using *allpre-eq-pre*[*of mp-free-precond8 t b* $\cap \{\!|$´*free-block-r t*$|\!\}$
Some P rely guar mp-free-precond8 t b] **apply** *blast*

apply(*rule allI*) **apply**(*rule impI*)
apply(*subgoal-tac* $\exists \alpha.\ v \in$ *mp-free-precond8-inv t b* $\alpha \cap \{\!|\alpha > 0|\!\}$)
prefer *2* **using** *looppre-imp-exist-*α*gt0* **apply** *blast*

apply(*erule exE*)

using *sat-pre-imp-allinpre*[*of Some P - rely guar mp-free-precond8 t b*]
loopbody-sat-invterm-imp-inv-post' **apply** *blast*
done

lemma *mempool-free-stm8-body*:
$\Gamma \vdash_I$ *Some* (*st8-while-body t b*)
sat$_p$ [*mp-free-precond8 t b* $\cap \{\!|$´*free-block-r t*$|\!\}$, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond8 t b*]

using *stm8-inv-imp-prepost2'*[*of* (*st8-while-body t b*) *t b Mem-pool-free-rely t Mem-pool-free-guar t*]
mempool-free-stm8-body-terminate[*of t b*] **apply** *fast*
done

lemma *mempool-free-stm8*:
$\Gamma \vdash_I$ *Some* (*WHILE* ´*free-block-r t DO*

st8-while-body t b
OD)
sat_p [*mp-free-precond8 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *mp-free-precond9 t b*]
apply(*rule While*)
using *mp-free-precond8-stb*[*of t b*] **apply** *blast*
apply *simp-inv* **apply** *auto*[*1*]
using *mp-free-precond9-stb*[*of t b*] **apply** *auto*[*1*]

using *mempool-free-stm8-body*[*of t b*] **apply** *fast*

apply(*simp add*: *Mem-pool-free-guar-def*)

done

20.4 statement 9

abbreviation *stm9-precond-while Va t b*
$\equiv$ {*V*. *inv V* $\wedge$ *cur V* = *cur Va* $\wedge$ *tick V* = *tick Va* $\wedge$ (*V*,*Va*)$\in$*gvars-conf-stable* $\wedge$ *freeing-node V t* = *freeing-node Va t* $\wedge$ *allocating-node V t* = *allocating-node Va t*
$\wedge$ ($\forall$ *p*. *levels* (*mem-pool-info V p*) = *levels* (*mem-pool-info Va p*))
$\wedge$ ($\forall$ *p*. *p* $\neq$ *pool b* $\longrightarrow$ *mem-pool-info V p* = *mem-pool-info Va p*)
$\wedge$ ($\forall$ *t'*. *t'* $\neq$ *t* $\longrightarrow$ *lvars-nochange t' V Va*)}

lemma *va-precond-while*: *inv Va* $\Longrightarrow$ *Va* $\in$ *stm9-precond-while Va t b*
by (*simp add*:*gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)

lemma *mempool-free-stm9-resch-inv-help*:
cur V = *Some t* $\Longrightarrow$ *thd-state V t* = *RUNNING* $\Longrightarrow$
(*SOME ta*. *ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY*) = *t* $\Longrightarrow$
V = *V*(|*cur* := *Some* (*SOME ta*. *ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY*),
thd-state := (*thd-state V*)(*t* := *READY*, *SOME ta*. *ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY* := *RUNNING*)|)
apply *auto*
proof −
assume *a1*: *thd-state V t* = *RUNNING*
assume *a2*: *cur V* = *Some t*
have (*thd-state V*)(*t* := *RUNNING*) = *thd-state V*
using *a1* **by** (*metis fun-upd-triv*)
then show *V* = *V*(|*cur* := *Some t*, *thd-state* := (*thd-state V*)(*t* := *RUNNING*)|)
using *a2* **by** *simp*
qed

lemma *mempool-free-stm9-resch-inv*:
cur V = *Some t* $\Longrightarrow$ *inv V* $\Longrightarrow$ *inv* (*V*(|*cur* := *Some* (*SOME ta*. *ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY*),
thd-state := (*thd-state V*)(*t* := *READY*, *SOME ta*. *ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY* := *RUNNING*)|))

apply(*subgoal-tac thd-state V t = RUNNING*)
apply(*case-tac* (*SOME ta. ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta = READY*) = *t*)
apply(*subgoal-tac V = V*⦇*cur* := *Some* (*SOME ta. ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY*),
thd-state := (*thd-state V*)(*t* := *READY*, *SOME ta. ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY* := *RUNNING*)⦈)
apply *simp* **using** *mempool-free-stm9-resch-inv-help*[*of V t*] **apply** *auto*[*1*]
apply(*subgoal-tac thd-state V* (*SOME ta. ta* $\neq$ *t* $\longrightarrow$ *thd-state V ta* = *READY*) = *READY*)
apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*)
apply(*rule conjI*) **apply**(*simp add*: *inv-aux-vars-def mem-block-addr-valid-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def*)

apply (*metis* (*mono-tags*, *lifting*) *someI-ex*)

apply(*simp add*:*inv-def inv-cur-def*) **apply** *auto*[*1*]
done

lemma *mempool-free-stm9-ifpart-one*:
Va $\in$ *mp-free-precond9 t b* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
V $\in$ *stm9-precond-while Va t b* $\cap$ ⦃*wait-q* (´*mem-pool-info* (*pool b*)) = []⦄ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (*IF* ´*need-resched t THEN reschedule FI*)
sat$_p$ [{*V*}, {(*x*, *y*). *x* = *y*}, *UNIV*, ⦃´(*Pair Va*) $\in$ *Mem-pool-free-guar t*⦄ $\cap$ *Mem-pool-free-post t*]
apply(*rule Cond*)
apply(*simp add*:*stable-def*)

apply(*simp add*:*reschedule-def*)
apply(*rule Seq*[**where** *mid*={*V*⦇*thd-state* := (*thd-state V*)(*the* (*cur V*) := *READY*)⦈
⦇*cur* := *Some* (*SOME t*. (*thd-state* (*V*⦇*thd-state* := (*thd-state V*)(*the* (*cur V*) := *READY*)⦈)) *t* = *READY*)⦈}])
apply(*rule Seq*[**where** *mid*={*V*⦇*thd-state* := (*thd-state V*)(*the* (*cur V*) := *READY*)⦈}])
apply(*rule Basic*)
apply *auto*[*1*] **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply *auto*[*1*] **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*)
apply *auto*[*1*] **apply**(*simp add*:*Mem-pool-free-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)

apply(*rule conjI*) **using** *mempool-free-stm9-resch-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*) **apply**(*simp add*:*Mem-pool-free-post-def*)
using *mempool-free-stm9-resch-inv* **apply** *auto*[*1*] **apply** *auto*[*1*] **apply**(*simp add*:*stable-def*)+

apply(*simp add*:*Skip-def*)
apply(*rule Basic*) **apply** *auto*[*1*] **apply**(*simp add*:*Mem-pool-free-guar-def*)
apply(*rule disjI1*) **apply**(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*simp add*:*Mem-pool-free-post-def*)
apply(*simp add*:*stable-def*)+
done

lemma *mempool-free-stm9-ifpart*:
Va ∈ *mp-free-precond9 t b* ∩ {|´*cur* = *Some t*|} ⟹
Γ $\vdash_I$ *Some* (*IF* ´*need-resched t THEN reschedule FI*)
sat_p [*stm9-precond-while Va t b* ∩ {|*wait-q* (´*mem-pool-info* (*pool b*)) = []|},
{(*x*, *y*). *x* = *y*}, *UNIV*, {|´(*Pair Va*) ∈ *Mem-pool-free-guar t*|} ∩ *Mem-pool-free-post t*]
using *mempool-free-stm9-ifpart-one*[*of Va t b*]
Allprecond[**where** *U*=*stm9-precond-while Va t b* ∩ {|*wait-q* (´*mem-pool-info* (*pool b*)) = []|} **and**
P= *Some* (*IF* ´*need-resched t THEN reschedule FI*) **and** *rely*={(*x*, *y*). *x* = *y*} **and**
guar= *UNIV* **and** *post*= {|´(*Pair Va*) ∈ *Mem-pool-free-guar t*|} ∩ *Mem-pool-free-post t*]
by *blast*

lemma *mempool-free-stm9-loopbody-one*:
Va ∈ *mp-free-precond9 t b* ∩ {|´*cur* = *Some t*|} ⟹
Vb ∈ *stm9-precond-while Va t b* ∩ {|*wait-q* (´*mem-pool-info* (*pool b*)) ≠ []|} ⟹
Γ $\vdash_I$ *Some* (´*th* := ´*th*(*t* := *hd* (*wait-q* (´*mem-pool-info* (*pool b*))));;
´*mem-pool-info* := ´*mem-pool-info*
(*pool b* := ´*mem-pool-info* (*pool b*)(|*wait-q* := *tl* (*wait-q* (´*mem-pool-info* (*pool b*)))|));;
´*thd-state* := ´*thd-state*(´*th t* := *READY*);;
´*need-resched* := ´*need-resched*(*t* := *True*))
sat_p [{*Vb*},{(*x*, *y*). *x* = *y*}, *UNIV*, *stm9-precond-while Va t b*]
apply(*rule Seq*[**where** *mid*={*Vb*(|*th* := (*th Vb*) (*t* := *hd* (*wait-q* ((*mem-pool-info Vb*) (*pool b*))))|)
(|*mem-pool-info* := (*mem-pool-info Vb*)
(*pool b* := (*mem-pool-info Vb*) (*pool b*)(|*wait-q* := *tl* (*wait-q* ((*mem-pool-info Vb*) (*pool b*)))|))|)
(|*thd-state* := (*thd-state Vb*)(*hd* (*wait-q* ((*mem-pool-info Vb*) (*pool b*))) := *READY*) |)}])

apply(*rule Seq*[**where** *mid*={*Vb*(|*th* := (*th Vb*) (*t* := *hd* (*wait-q* ((*mem-pool-info Vb*) (*pool b*))))|)

(|mem-pool-info := (mem-pool-info Vb)
(pool b := (mem-pool-info Vb) (pool b)(|wait-q := tl (wait-q
((mem-pool-info Vb) (pool b)))|))|)}])

apply(*rule Seq*[**where** *mid*={*Vb*(|*th* := (*th Vb*) (*t* := *hd* (*wait-q* ((*mem-pool-info*
Vb) (*pool b*))))|)}])

apply(*rule Basic*) **apply** *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*) **apply** *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*)+
apply(*rule Basic*) **apply** *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*)+

apply(*rule Basic*) **apply** *clarify* **apply**(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def*
gvars-conf-def)
apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def inv-thd-waitq-def*)
apply(*rule conjI*) **apply**(*simp add*: *inv-thd-waitq-def*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify* **apply** (*rule conjI*) **apply** *clarify* **apply**(*rule*
conjI) **apply** *clarify*
apply (*smt List.nth-tl Nitpick.size-list-simp*(*2*) *Suc-mono gr-implies-not0*
hd-conv-nth in-set-conv-nth length-pos-if-in-set lessI list.set-sel(*1*))
apply *clarify* **apply** (*meson list.set-sel*(*2*)) **apply** *clarify* **apply** (*metis list.set-sel*(*1*))
apply(*rule conjI*) **apply** *clarify* **apply** (*metis hd-Cons-tl set-ConsD*)
apply(*rule conjI*) **apply** *clarify* **apply** (*metis* (*no-types*, *lifting*) *List.nth-tl*
Nitpick.size-list-simp(*2*)
One-nat-def Suc-mono length-tl nat.inject)
apply *clarify* **apply**(*rule conjI*) **apply** *clarify* **apply** (*metis list.set-sel*(*2*))
apply *clarify* **apply**(*rule conjI*) **apply** *clarify* **apply** (*metis list.set-sel*(*2*))
apply *clarify* **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*: *inv-mempool-info-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*: *inv-bitmap-freelist-def*)
apply(*rule conjI*) **apply**(*simp add*: *inv-bitmap-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-aux-vars-def mem-block-addr-valid-def*)
apply(*rule conjI*) **apply** *metis* **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def partner-bits-def*)

apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *force*
apply(*rule conjI*)
apply *clarify* **apply**(*simp add*:*lvars-nochange-def*)
apply(*simp add*:*lvars-nochange-def*)
by (*simp add*:*stable-def*)+

lemma *mempool-free-stm9-loopbody*:

$Va \in$ *mp-free-precond9 t b* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*th* := ´*th*(*t* := *hd* (*wait-q* (´*mem-pool-info* (*pool b*))));;
´*mem-pool-info* := ´*mem-pool-info*
(*pool b* := ´*mem-pool-info* (*pool b*)⦇*wait-q* := *tl* (*wait-q* (´*mem-pool-info* (*pool b*)))⦈);;
´*thd-state* := ´*thd-state*(´*th t* := *READY*);;
´*need-resched* := ´*need-resched*(*t* := *True*))
sat_p [*stm9-precond-while Va t b* $\cap$ ⦃*wait-q* (´*mem-pool-info* (*pool b*)) $\neq$ []⦄,
{(*x*, *y*). *x* = *y*}, *UNIV*, *stm9-precond-while Va t b*]
using *mempool-free-stm9-loopbody-one*
Allprecond[**where** *U*=*stm9-precond-while Va t b* $\cap$ ⦃*wait-q* (´*mem-pool-info* (*pool b*)) $\neq$ []⦄ **and**
P= *Some* (´*th* := ´*th*(*t* := *hd* (*wait-q* (´*mem-pool-info* (*pool b*))));;
´*mem-pool-info* := ´*mem-pool-info*(*pool b* := ´*mem-pool-info* (*pool b*)⦇*wait-q* := *tl* (*wait-q* (´*mem-pool-info* (*pool b*)))⦈);;
´*thd-state* := ´*thd-state*(´*th t* := *READY*);;
´*need-resched* := ´*need-resched* (*t* := *True*))
and *rely*={(*x*, *y*). *x* = *y*} **and** *guar*= *UNIV* **and** *post*= *stm9-precond-while Va t b*]
by *blast*

lemma *mempool-free-stm9-body-loopinv*:
$Va \in$ *mp-free-precond9 t b* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (*WHILE wait-q* (´*mem-pool-info* (*pool b*)) $\neq$ []
DO ´*th* := ´*th*(*t* := *hd* (*wait-q* (´*mem-pool-info* (*pool b*))));;
´*mem-pool-info* := ´*mem-pool-info*
(*pool b* := ´*mem-pool-info* (*pool b*)⦇*wait-q* := *tl* (*wait-q* (´*mem-pool-info* (*pool b*)))⦈);;
´*thd-state* := ´*thd-state*(´*th t* := *READY*);;
´*need-resched* := ´*need-resched*(*t* := *True*)
OD;;
IF ´*need-resched t THEN reschedule FI*)
sat_p [*stm9-precond-while Va t b*, {(*x*, *y*). *x* = *y*}, *UNIV*, {*s*. (*Va*,*s*)∈*Mem-pool-free-guar t*} $\cap$ *Mem-pool-free-post t*]
apply(*rule Seq*[**where** *mid*=*stm9-precond-while Va t b* $\cap$ ⦃ *wait-q* (´*mem-pool-info* (*pool b*)) = []⦄])
apply(*rule While*)
apply(*simp add*:*stable-def*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*)
using *mempool-free-stm9-loopbody*[*of Va t b*] **apply** *simp*
apply *simp*

using *mempool-free-stm9-ifpart* **by** *blast*

lemma *mempool-free-stm9-body*:
mp-free-precond9 t b $\cap$ ⦃´*inv*⦄ $\cap$ ⦃´*cur* = *Some t*⦄ $\cap$ {*Va*} $\neq$ {} $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (*WHILE wait-q* (´*mem-pool-info* (*pool b*)) $\neq$ []

DO ´th := ´th(t := hd (wait-q (´mem-pool-info (pool b))));;
´mem-pool-info := ´mem-pool-info(pool b := ´mem-pool-info (pool b)⦇wait-q := tl (wait-q (´mem-pool-info (pool b)))⦈);;
´thd-state := ´thd-state(´th t := READY);;
´need-resched := ´need-resched(t := True)
OD;;
IF ´need-resched t THEN reschedule FI)
sat_p [*mp-free-precond9 t b* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*Va*},
{(*s*, *t*). *s* = *t*}, *UNIV*, ⦃´(*Pair Va*) ∈ *Mem-pool-free-guar t*⦄ ∩ *Mem-pool-free-post t*]
apply(*subgoal-tac inv Va*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac Va* ∈ *mp-free-precond9 t b* ∩ ⦃´*inv*⦄ ∩ ⦃´*cur* = *Some t*⦄) **prefer** *2* **apply** *simp*
using *mempool-free-stm9-body-loopinv*[*of Va t b*] *va-precond-while*[*of Va t b*]
Conseq[**where** *pre*={*Va*} **and** *pre′*=*stm9-precond-while Va t b* **and** *rely*={(*x*, *y*). *x* = *y*} **and** *rely′*={(*x*, *y*). *x* = *y*}
and *guar*=*UNIV* **and** *guar′*=*UNIV* **and** *post′*=⦃´(*Pair Va*) ∈ *Mem-pool-free-guar t*⦄ ∩ *Mem-pool-free-post t*
and *post*=⦃´(*Pair Va*) ∈ *Mem-pool-free-guar t*⦄ ∩ *Mem-pool-free-post t*
and *P*=*Some* (*WHILE wait-q* (´*mem-pool-info* (*pool b*)) ≠ []
DO ´th := ´th(t := hd (wait-q (´mem-pool-info (pool b))));;
´mem-pool-info := ´mem-pool-info
(pool b := ´mem-pool-info (pool b)⦇wait-q := tl (wait-q (´mem-pool-info (pool b)))⦈);;
´thd-state := ´thd-state(´th t := READY);;
´need-resched := ´need-resched(t := True)
OD;;
IF ´need-resched t THEN reschedule FI)]
apply *force*
done

lemma *mempool-free-stm9*:
Γ $\vdash_I$ *Some* (*t* ▶ *ATOMIC*
WHILE wait-q (´*mem-pool-info* (*pool b*)) ≠ [] *DO*
´th := ´th (t := hd (wait-q (´mem-pool-info (pool b))));;
´mem-pool-info := ´mem-pool-info (pool b := ´mem-pool-info (pool b)
⦇wait-q := tl (wait-q (´mem-pool-info (pool b)))⦈);;
´thd-state := ´thd-state (´th t := READY);;
´need-resched := ´need-resched(t := True)
OD;;

IF ´need-resched t THEN
reschedule
FI
END)
sat_p [*mp-free-precond9 t b*, *Mem-pool-free-rely t*, *Mem-pool-free-guar t*, *Mem-pool-free-post t*]
apply(*simp add*:*stm-def*)
apply(*rule Await*)

using *mp-free-precond9-stb* **apply** *auto*[*1*]
apply (*simp add*: *mem-pool-free-post-stb*)

apply(*rule allI*)
apply(*rule Await*)
apply(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)
apply(*rule allI*)
apply(*case-tac* $V = Va$) **apply** *simp*
apply(*case-tac mp-free-precond9 t b* $\cap$ {|´*inv*|} $\cap$ {|´*cur* = *Some t*|} $\cap$ {*Va*} = {})
apply *simp* **apply** (*simp add*: *Emptyprecond stable-id2*)
apply *clarify* **using** *mempool-free-stm9-body* **apply** *force*
apply *simp* **apply** (*simp add*: *Emptyprecond stable-id2*)
done

20.5 final proof

lemma *Mempool-free-satRG*: *Evt-sat-RG* Γ (*Mem-pool-free-RGCond t b*)
apply(*simp add*:*Evt-sat-RG-def*)
apply (*simp add*: *Mem-pool-free-def Mem-pool-free-RGCond-def*)
apply(*rule Evt-Basic*)
apply(*simp add*:*body-def guard-def*)
apply(*rule Seq*[**where** *mid*=*mp-free-precond9 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond8 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond7 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond6 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond5 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond4 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond3 t b*])
apply(*rule Seq*[**where** *mid*=*mp-free-precond2 t b*])

using *mempool-free-stm1*[*of t b*] **apply** *fast*
using *mempool-free-stm2*[*of t b*] **apply** *fast*
using *mempool-free-stm3*[*of t b*] **apply** *fast*
using *mempool-free-stm4*[*of t b*] **apply** *force*
using *mempool-free-stm5*[*of t b*] **apply** *fast*
using *mempool-free-stm6*[*of t b*] **apply** *fast*
using *mempool-free-stm7*[*of t b*] **apply** *fast*
using *mempool-free-stm8*[*of t b*] **apply** *force*
using *mempool-free-stm9*[*of t b*] **apply** *force*

apply(*simp add*: *stable-equiv mem-pool-free-pre-stb*)

apply(*simp add*: *Mem-pool-free-guar-def*)
done

end

theory *func-cor-mempoolalloc*
imports *func-cor-lemma ../../adapter-SIMP/picore-SIMP-lemma*
begin

21 Functional correctness of *k_mem_pool_alloc*

21.1 intermediate conditions and their stable to rely cond

abbreviation *mp-alloc-precond1 t p tm* ≡
Mem-pool-alloc-pre t ∩ ⦃*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*⦄

lemma *mp-alloc-precond1-ext-stb*: *stable* (⦃*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*simp add*:*gvars-conf-stable-def*)
unfolding *gvars-conf-def* **apply** *metis*
done

lemma *mp-alloc-precond1-stb*: *stable* (*mp-alloc-precond1 t p tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
apply(*simp add*:*mem-pool-alloc-pre-stb*)
apply(*simp add*:*mp-alloc-precond1-ext-stb*)
done

abbreviation *mp-alloc-precond2 t p tm* ≡
mp-alloc-precond1 t p tm ∩ ⦃´*tmout t* = *tm*⦄

lemma *mp-alloc-precond2-ext-stb*: *stable* (⦃´*tmout t* = *tm*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond2-stb*: *stable* (*mp-alloc-precond2 t p tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
apply(*simp add*:*mp-alloc-precond1-stb*)
apply(*simp add*:*mp-alloc-precond2-ext-stb*)
done

abbreviation *mp-alloc-precond3 t p tm* ≡
mp-alloc-precond2 t p tm ∩ ⦃´*endt t* = *0*⦄

lemma *mp-alloc-precond3-ext-stb*: *stable* ($\{\!|$´*endt t* = *0*$|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond3-stb*: *stable* (*mp-alloc-precond3 t p tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  apply(*simp add*:*mp-alloc-precond2-stb*)
  apply(*simp add*:*mp-alloc-precond3-ext-stb*)
done

abbreviation *mp-alloc-precond4 t p tm* ≡
  mp-alloc-precond2 t p tm ∩ $\{\!|$´*endt t* ≥ *0*$|\!\}$

lemma *mp-alloc-precond4-ext-stb*: *stable* ($\{\!|$´*endt t* ≥ *0*$|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*)
done

lemma *mp-alloc-precond4-stb*: *stable* (*mp-alloc-precond4 t p tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  apply(*simp add*:*mp-alloc-precond2-stb*)
  using *mp-alloc-precond4-ext-stb* **apply** *auto*
done

abbreviation *mp-alloc-precond5 t p tm* ≡
  mp-alloc-precond4 t p tm ∩ $\{\!|$´*mempoolalloc-ret t* = *None*$|\!\}$

lemma *mp-alloc-precond5-ext-stb*: *stable* ($\{\!|$´*mempoolalloc-ret t* = *None*$|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond5-stb*: *stable* (*mp-alloc-precond5 t p tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond4-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond5-ext-stb*)
done

abbreviation *mp-alloc-precond6 t p tm* ≡
  mp-alloc-precond5 t p tm ∩ $\{\!|$´*ret t* = *ESIZEERR*$|\!\}$

lemma *mp-alloc-precond6-ext-stb*: *stable* ($\{\!|$´*ret t* = *ESIZEERR*$|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond6-stb*: *stable* (*mp-alloc-precond6 t p tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond5-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond6-ext-stb*)
done

abbreviation *mp-alloc-precond7-ext t p sz timeout* $\equiv$
 $\{$*s*. (*rf s t* $\longrightarrow$ (*timeout* = *FOREVER* $\longrightarrow$ (*ret s t* = *ESIZEERR* $\wedge$ *mempoolalloc-ret s t* = *None*
 $\vee$ *ret s t* = *OK* $\wedge$ ($\exists$ *mblk*. *mempoolalloc-ret s t* = *Some mblk* $\wedge$ *alloc-memblk-valid s p sz mblk*)))
 $\wedge$ (*timeout* = *NOWAIT* $\longrightarrow$ ((*ret s t* = *ENOMEM* $\vee$ *ret s t* = *ESIZEERR*) $\wedge$ *mempoolalloc-ret s t* = *None*)
 $\vee$ (*ret s t* = *OK* $\wedge$ ($\exists$ *mblk*. *mempoolalloc-ret s t* = *Some mblk* $\wedge$ *alloc-memblk-valid s p sz mblk*)))
 $\wedge$ (*timeout* > *0* $\longrightarrow$ ((*ret s t* = *ETIMEOUT* $\vee$ *ret s t* = *ESIZEERR*) $\wedge$ *mempoolalloc-ret s t* = *None*)
 $\vee$ (*ret s t* = *OK* $\wedge$ ($\exists$ *mblk*. *mempoolalloc-ret s t* = *Some mblk* $\wedge$ *alloc-memblk-valid s p sz mblk*))))
 $\wedge$ ($\neg$*rf s t* $\longrightarrow$ *mempoolalloc-ret s t* = *None*)
 $\wedge$ (*timeout* = *FOREVER* $\longrightarrow$ *tmout s t* = *FOREVER*)$\}$

abbreviation *mp-alloc-precond7 t p sz timeout* $\equiv$
  mp-alloc-precond1 t p timeout $\cap$ *mp-alloc-precond7-ext t p sz timeout*

abbreviation *mp-alloc-precond7-inv t p sz timeout* α $\equiv$
  mp-alloc-precond7 t p sz timeout
 $\cap$ $\{\!|$ α = (*if* ´*rf t* $\vee$ ´*mempoolalloc-ret t* $\neq$ *None then 0* ~~(* if timeout ≠ 0 (NOWAIT), rf is true *)~~
 else if timeout > *0 then* ´*endt t* − ´*tick*
 ~~(* in rely cond, tick' ≥ tick, thus convergent β ≤ α, not < α, thus not absolutely convergent *)~~
 else 1)
 ~~(* cannot find convergent α for FOREVER, so just set 1 *)~~ $|\!\}$

lemma *mp-alloc-precond7-ext-stb*: *stable* (*mp-alloc-precond7-ext t p sz timeout*) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)

apply(*rule impI*)
  using *mp-alloc-post-stb*
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*)
  apply(*case-tac x* = *y*)
    apply *simp* **apply** *clarify*
    apply(*simp add*:*alloc-memblk-valid-def gvars-conf-def gvars-conf-stable-def*)
done

lemma *mp-alloc-precond7-stb*: *stable* (*mp-alloc-precond7 t p sz timeout*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond7-ext-stb*)
done

abbreviation *mp-alloc-precond1-0 t p sz tm* ≡
  mp-alloc-precond7 t p sz tm ∩ {|¬ ´*rf t*|}

lemma *mp-alloc-precond1-0-ext-stb*: *stable* {|¬ ´*rf t*|} (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)
apply(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond1-0-stb*: *stable* (*mp-alloc-precond1-0 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond7-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond1-0-ext-stb*)
done

abbreviation *mp-alloc-precond1-1 t p sz tm* ≡
  mp-alloc-precond1-0 t p sz tm

lemma *mp-alloc-precond1-1-stb*: *stable* (*mp-alloc-precond1-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
  using *mp-alloc-precond1-0-stb* **by** *simp*

abbreviation *mp-alloc-precond1-2 t p sz tm* ≡
  mp-alloc-precond1-1 t p sz tm ∩ {|´*alloc-lsize-r t* = *False*|}

lemma *mp-alloc-precond1-2-stb*: *stable* (*mp-alloc-precond1-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-1-stb* **apply** *auto*[*1*]
  apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*)

apply(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

abbreviation *mp-alloc-precond1-3 t p sz tm* ≡
  mp-alloc-precond1-2 t p sz tm ∩ {|´*alloc-l t* = −*1*|}

lemma *mp-alloc-precond1-3-ext-stb*: *stable* {|´*alloc-l t* = −*1*|} (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond1-3-stb*: *stable* (*mp-alloc-precond1-3 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-2-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond1-3-ext-stb*)
done

abbreviation *mp-alloc-precond1-4 t p sz tm* ≡
  mp-alloc-precond1-3 t p sz tm ∩ {|´*free-l t* = −*1*|}

lemma *mp-alloc-precond1-4-ext-stb*: *stable* {|´*free-l t* = −*1*|} (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def*) **apply** *smt*
done

lemma *mp-alloc-precond1-4-stb*: *stable* (*mp-alloc-precond1-4 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-3-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond1-4-ext-stb*)
done

abbreviation *mp-alloc-precond1-5 t p sz tm* ≡
  mp-alloc-precond1-4 t p sz tm ∩ {|´*lsizes t* = [*ALIGN4* (*max-sz* (´*mem-pool-info p*))]|}

lemma *mp-alloc-precond1-5-ext-stb*: *stable* {|´*lsizes t* = [*ALIGN4* (*max-sz* (´*mem-pool-info p*))]|} (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)

apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac* $x=y$) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-5-stb*: *stable* (*mp-alloc-precond1-5 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-4-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-5-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-ext t p sz tm* ≡
⦃(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info p*))) *div* (*4* ^ *ii*))
∧ *length* (´*lsizes t*) ≤ *n-levels* (´*mem-pool-info p*)
∧ (´*i t* = *0* ⟶ ´*alloc-l t* = −*1* ∧ ´*free-l t* = −*1* ∧ *length* (´*lsizes t*) = *1*)
∧ ´*i t* ≤ *n-levels* (´*mem-pool-info p*)
∧ −*1* ≤ ´*free-l t* ∧ ´*free-l t* ≤ *int* (´*i t*) − *1* ∧ ´*free-l t* ≤ ´*alloc-l t*
∧ ´*alloc-l t* = *int* (´*i t*) − *1*
∧ (´*alloc-l t* ≥ *0* ⟶ (∀ *ii*. *ii* ≤ *nat* (´*alloc-l t*) ⟶ ´*lsizes t* ! *ii* ≥ *sz*))
∧ (¬ ´*alloc-lsize-r t* ⟶ (´*i t* = *0* ⟶ *length* (´*lsizes t*) = *1*) ∧ (´*i t* > *0* ⟶ *length* (´*lsizes t*) = ´*i t*))
∧ (´*alloc-lsize-r t* ⟶ *length* (´*lsizes t*) = ´*i t* + *1* ∧ ´*i t* < *n-levels* (´*mem-pool-info p*) ∧ ´*lsizes t* ! (´*i t*) < *sz*)⦄

abbreviation *mp-alloc-precond1-6 t p sz tm* ≡
mp-alloc-precond1-1 t p sz tm ∩ *mp-alloc-precond1-6-ext t p sz tm*

abbreviation *mp-alloc-lsizeloop-α-cond t p α* ≡
⦃α = (*if* ´*alloc-lsize-r t*
then 0 else n-levels (´*mem-pool-info p*) − ´*i t*) ⦄

abbreviation *mp-alloc-lsizestm-loopinv t p sz tm α* ≡
mp-alloc-precond1-6 t p sz tm ∩ *mp-alloc-lsizeloop-α-cond t p α*

abbreviation *mp-alloc-lsizestm-loopcond t p* ≡ ⦃ ´*i t* < *n-levels* (´*mem-pool-info p*) ∧ ¬ ´*alloc-lsize-r t* ⦄

lemma *lsizestm-loopinv-imp-precond*:
mp-alloc-lsizestm-loopinv t p sz tm α ⊆ *mp-alloc-precond1-6 t p sz tm*
by *auto*

lemma *lsizestm-loopinv-αgt0-imp-loopcond*:
mp-alloc-lsizestm-loopinv t p sz tm α ∩ ⦃α > *0*⦄ ⊆ *mp-alloc-lsizestm-loopcond t p*
by *clarsimp*

lemma *lsizestm-loopinv-αeq0-imp-notloopcond*:
mp-alloc-lsizestm-loopinv t p sz tm α ∩ {|α = *0*|} ⊆ − *mp-alloc-lsizestm-loopcond t p*
by *clarsimp*

lemma *lsizestm-loopinv-αeq0-imp-notloopcond2*:
mp-alloc-lsizestm-loopinv t p sz tm 0 ⊆ − *mp-alloc-lsizestm-loopcond t p*
by *clarsimp*

lemma *lsizestm-pre-loopcond-imp-loopinv-αgt0*:
x∈*mp-alloc-precond1-6 t p sz tm* ∩ *mp-alloc-lsizestm-loopcond t p* ⟹
∃ α. *x*∈*mp-alloc-lsizestm-loopinv t p sz tm* α ∩ {|α > *0*|}
by *clarsimp*

lemma *lsizestm-pre-notloopcond-imp-loopinv-αeq0*:
x∈*mp-alloc-precond1-6 t p sz tm* ∩ − *mp-alloc-lsizestm-loopcond t p* ⟹
x∈*mp-alloc-lsizestm-loopinv t p sz tm 0*
apply *clarsimp*
apply(*rule conjI*)
apply *clarify* **apply** *simp*
apply *clarify* **apply** *simp*
done

lemma *lsizestm-pre-notloopcond-imp-loopinv-αeq0'*:
mp-alloc-precond1-6 t p sz tm ∩ − *mp-alloc-lsizestm-loopcond t p*
⊆ *mp-alloc-lsizestm-loopinv t p sz tm 0*
apply *clarsimp*
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify*
apply *clarify* **apply** *simp*
done

lemma *lsizestm-pre-notloopcond-eq-loopinv-αeq0*:
mp-alloc-precond1-6 t p sz tm ∩ − *mp-alloc-lsizestm-loopcond t p*
= *mp-alloc-lsizestm-loopinv t p sz tm 0*
apply(*rule subset-antisym*)
using *lsizestm-pre-notloopcond-imp-loopinv-αeq0'*[*of t p tm sz*] **apply** *blast*
apply(*rule Int-greatest*)
using *lsizestm-loopinv-imp-precond*[*of t p tm sz 0*] **apply** *blast*
using *lsizestm-loopinv-αeq0-imp-notloopcond2*[*of t p tm sz*] **apply** *blast*
done

lemma *lsizeloop-inv-cond-eq-αgt0*:
mp-alloc-lsizestm-loopinv t p sz tm α ∩ *mp-alloc-lsizestm-loopcond t p*
= *mp-alloc-lsizestm-loopinv t p sz tm* α ∩ {|α > *0*|}

apply(*rule subset-antisym*)
apply(*rule Int-greatest*)
  apply *fast*
  apply *clarify* **apply** *auto*[*1*]
apply(*rule Int-greatest*)
  apply *fast*
  apply *clarsimp*
done

lemma *mp-alloc-precond1-6-ext-stb*: *stable* (*mp-alloc-precond1-6-ext t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
   apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
 apply(*rule conjI*) **apply** *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
 apply *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-stb*: *stable* (*mp-alloc-precond1-6 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-1-stb* **apply** *auto*[*1*]
  using *mp-alloc-precond1-6-ext-stb* **apply** *auto*
done

lemma *mp-alloc-lsizeloop-α-cond-stb*: *stable* (*mp-alloc-lsizeloop-α-cond t p α*) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*) **apply** *auto*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)+
done

lemma *mp-alloc-lsizestm-loopinv-stb*: *stable* (*mp-alloc-lsizestm-loopinv t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-6-stb* **apply** *fast*
using *mp-alloc-lsizeloop-α-cond-stb* **apply** *fast*
done

lemma *mp-alloc-lsizestm-loopinv-presv-rely*:
$s \in$ *mp-alloc-lsizestm-loopinv t p sz tm* $\alpha \Longrightarrow (s,r) \in$ *Mem-pool-alloc-rely t* $\Longrightarrow \exists \beta \leq \alpha.$ $r \in$ *mp-alloc-lsizestm-loopinv t p sz tm* β
apply(*rule exI*[**where** *x*=*α*])
apply(*rule conjI*) **apply** *fast*
using *mp-alloc-lsizestm-loopinv-stb*[*of t p tm sz α*] **apply**(*unfold stable-def*) **apply**

meson
done

abbreviation *mp-alloc-precond1-6-1 t p sz tm* $\alpha \equiv$
mp-alloc-lsizestm-loopinv t p sz tm $\alpha \cap$ *mp-alloc-lsizestm-loopcond t p*

lemma *mp-alloc-precond1-6-1-ext-stb*: *stable* (*mp-alloc-lsizestm-loopcond t p*) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-1-stb*: *stable* (*mp-alloc-precond1-6-1 t p sz tm* α) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-lsizestm-loopinv-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-6-1-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-10 t p sz tm* $\alpha \equiv$
mp-alloc-precond1-6-1 t p sz tm $\alpha \cap \{\!|\,{}'i\ t > 0|\!\}$

lemma *mp-alloc-precond1-6-10-ext-stb*: *stable* ($\{\!|\,{}'i\ t > 0|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-10-stb*: *stable* (*mp-alloc-precond1-6-10 t p sz tm* α) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-6-1-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-6-10-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-11 t p sz tm* $\alpha \equiv$
mp-alloc-precond1-6-1 t p sz tm $\alpha \cap - \{\!|\,{}'i\ t > 0|\!\}$

lemma *mp-alloc-precond1-6-11-ext-stb*: *stable* ($- \{\!|\,{}'i\ t > 0|\!\}$) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule allI*) **apply**(*rule impI*)

apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-11-stb*: *stable* (*mp-alloc-precond1-6-11 t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-6-1-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-6-11-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-2-ext t p sz tm α* ≡
⦃(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info p*))) *div* (*4* ˆ *ii*))
∧ *length* (´*lsizes t*) ≤ *n-levels* (´*mem-pool-info p*)
∧ (´*i t* = *0* ⟶ ´*alloc-l t* = −*1* ∧ ´*free-l t* = −*1* ∧ *length* (´*lsizes t*) = *1*)
∧ ´*i t* ≤ *n-levels* (´*mem-pool-info p*)
∧ −*1* ≤ ´*free-l t* ∧ ´*free-l t* ≤ *int* (´*i t*) − *1* ∧ ´*free-l t* ≤ ´*alloc-l t*
∧ ´*alloc-l t* = *int* (´*i t*) − *1*
∧ (´*alloc-l t* ≥ *0* ⟶ (∀ *ii*. *ii* ≤ *nat* (´*alloc-l t*) ⟶ ´*lsizes t* ! *ii* ≥ *sz*))
∧ (¬ ´*alloc-lsize-r t* ⟶ (´*i t* = *0* ⟶ *length* (´*lsizes t*) = *1*) ∧ (´*i t* > *0* ⟶ *length* (´*lsizes t*) = ´*i t* + *1*))
~~(* here ´i t + 1 is different from mp-alloc-precond1-6-ext, *)~~
∧ (´*alloc-lsize-r t* ⟶ *length* (´*lsizes t*) = ´*i t* + *1* ∧ ´*i t* < *n-levels* (´*mem-pool-info p*) ∧ ´*lsizes t* ! (´*i t*) < *sz*)⦄
∩ *mp-alloc-lsizeloop-α-cond t p α*

abbreviation *mp-alloc-precond1-6-2 t p sz tm α* ≡
mp-alloc-precond1-2 t p sz tm ∩ *mp-alloc-precond1-6-2-ext t p sz tm α*

lemma *mp-alloc-precond1-6-2-ext-stb*: *stable* (*mp-alloc-precond1-6-2-ext t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply** *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
using *mp-alloc-lsizeloop-α-cond-stb* **apply** *fast*
done

lemma *mp-alloc-precond1-6-2-stb*: *stable* (*mp-alloc-precond1-6-2 t p sz tm α*) (*Mem-pool-alloc-rely t*)

apply(*rule stable-int2*)
using *mp-alloc-precond1-2-stb* **apply** *auto*[*1*]
using *mp-alloc-precond1-6-2-ext-stb* **apply** *auto*
done

abbreviation *mp-alloc-precond1-6-20 t p sz tm* $\alpha \equiv$
mp-alloc-precond1-6-2 t p sz tm $\alpha \cap$ ⦃´*lsizes t* ! ´*i t* < *sz*⦄

lemma *mp-alloc-precond1-6-20-ext-stb*: *stable* (⦃´*lsizes t* ! ´*i t* < *sz*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-20-stb*: *stable* (*mp-alloc-precond1-6-20 t p sz tm* α) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-6-2-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-6-20-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-21 t p sz tm* $\alpha \equiv$
mp-alloc-precond1-6-2 t p sz tm $\alpha \cap -$ ⦃´*lsizes t* ! ´*i t* < *sz*⦄

lemma *mp-alloc-precond1-6-21-ext-stb*: *stable* (− ⦃´*lsizes t* ! ´*i t* < *sz*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-6-21-stb*: *stable* (*mp-alloc-precond1-6-21 t p sz tm* α) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-6-2-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-6-21-ext-stb*)
done

abbreviation *mp-alloc-precond1-6-21-1-ext t p sz tm* $\alpha \equiv$
⦃(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info p*))) *div* (*4* ^ *ii*))
∧ *length* (´*lsizes t*) ≤ *n-levels* (´*mem-pool-info p*)
∧ (´*i t* = *0* ⟶ ´*alloc-l t* = *0* ∧ ´*free-l t* = −*1* ∧ *length* (´*lsizes t*) = *1*)
∧ ´*i t* ≤ *n-levels* (´*mem-pool-info p*)

$\wedge$ $-1 \leq$ ´*free-l t* $\wedge$ ´*free-l t* $\leq$ *int* (´*i t*) $-$ *1* $\wedge$ ´*free-l t* $\leq$ ´*alloc-l t*
$\wedge$ ´*alloc-l t* $=$ *int* (´*i t*)
$\wedge$ (´*alloc-l t* $\geq$ *0* $\longrightarrow$ ($\forall$ *ii*. *ii* $<$ *nat* (´*alloc-l t*) $\longrightarrow$ ´*lsizes t* ! *ii* $\geq$ *sz*))
$\wedge$ ($\neg$ ´*alloc-lsize-r t* $\longrightarrow$ (´*i t* $=$ *0* $\longrightarrow$ *length* (´*lsizes t*) $=$ *1*) $\wedge$ (´*i t* $>$ *0* $\longrightarrow$ *length* (´*lsizes t*) $=$ ´*i t* $+$ *1*))
$\wedge$ (´*alloc-lsize-r t* $\longrightarrow$ *length* (´*lsizes t*) $=$ ´*i t* $+$ *1* $\wedge$ ´*i t* $<$ *n-levels* (´*mem-pool-info p*) $\wedge$ ´*lsizes t* ! (´*i t*) $<$ *sz*)
$\wedge$ $\neg$ ´*lsizes t* ! ´*i t* $<$ *sz*$\}\!\!\}$ $\cap$ *mp-alloc-lsizeloop-α-cond t p α*

abbreviation *mp-alloc-precond1-6-21-1 t p sz tm α* $\equiv$
mp-alloc-precond1-2 t p sz tm $\cap$ *mp-alloc-precond1-6-21-1-ext t p sz tm α*

lemma *mp-alloc-precond1-6-21-1-ext-stb*: *stable* (*mp-alloc-precond1-6-21-1-ext t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
using *mp-alloc-lsizeloop-α-cond-stb* **apply** *fast*
done

lemma *mp-alloc-precond1-6-21-1-stb*: *stable* (*mp-alloc-precond1-6-21-1 t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-2-stb* **apply** *auto*[*1*]
using *mp-alloc-precond1-6-21-1-ext-stb* **apply** *auto*
done

abbreviation *mp-alloc-precond1-6-21-2-ext t p sz tm α* $\equiv$
$\{\!\!\{$($\forall$ *ii*$<$*length* (´*lsizes t*). ´*lsizes t* ! *ii* $=$ (*ALIGN4* (*max-sz* (´*mem-pool-info p*))) *div* (*4* ^ *ii*))
$\wedge$ *length* (´*lsizes t*) $\leq$ *n-levels* (´*mem-pool-info p*)
$\wedge$ (´*i t* $=$ *0* $\longrightarrow$ ´*alloc-l t* $=$ *0* $\wedge$ *length* (´*lsizes t*) $=$ *1*)
$\wedge$ ´*i t* $\leq$ *n-levels* (´*mem-pool-info p*)
$\wedge$ $-1 \leq$ ´*free-l t* $\wedge$ ´*free-l t* $\leq$ *int* (´*i t*) $\wedge$ ´*free-l t* $\leq$ ´*alloc-l t*
$\wedge$ ´*alloc-l t* $=$ *int* (´*i t*)
$\wedge$ (´*alloc-l t* $\geq$ *0* $\longrightarrow$ ($\forall$ *ii*. *ii* $<$ *nat* (´*alloc-l t*) $\longrightarrow$ ´*lsizes t* ! *ii* $\geq$ *sz*))
$\wedge$ ($\neg$ ´*alloc-lsize-r t* $\longrightarrow$ (´*i t* $=$ *0* $\longrightarrow$ *length* (´*lsizes t*) $=$ *1*) $\wedge$ (´*i t* $>$ *0* $\longrightarrow$ *length* (´*lsizes t*) $=$ ´*i t* $+$ *1*))
$\wedge$ (´*alloc-lsize-r t* $\longrightarrow$ *length* (´*lsizes t*) $=$ ´*i t* $+$ *1* $\wedge$ ´*i t* $<$ *n-levels* (´*mem-pool-info p*) $\wedge$ ´*lsizes t* ! (´*i t*) $<$ *sz*)
$\wedge$ $\neg$ ´*lsizes t* ! ´*i t* $<$ *sz*$\}\!\!\}$ $\cap$ *mp-alloc-lsizeloop-α-cond t p α*

abbreviation *mp-alloc-precond1-6-21-2 t p sz tm α* $\equiv$
mp-alloc-precond1-2 t p sz tm $\cap$ *mp-alloc-precond1-6-21-2-ext t p sz tm α*

lemma *mp-alloc-precond1-6-21-2-ext-stb*: *stable* (*mp-alloc-precond1-6-21-2-ext t p sz tm α*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
   apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
 apply *clarify* **apply**(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
using *mp-alloc-lsizeloop-α-cond-stb* **apply** *fast*
done

lemma *mp-alloc-precond1-6-21-2-stb*: *stable* (*mp-alloc-precond1-6-21-2 t p sz tm α*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-2-stb* **apply** *auto*[*1*]
  using *mp-alloc-precond1-6-21-2-ext-stb* **apply** *auto*
done

abbreviation *mp-alloc-precond1-7 t p sz tm* ≡
 mp-alloc-precond1-6 t p sz tm ∩ ⦃´*i t* ≥ *n-levels* (´*mem-pool-info p*) ∨ ´*alloc-lsize-r t* ⦄

lemma *mp-alloc-precond1-7-ext-stb*: *stable* (⦃´*i t* ≥ *n-levels* (´*mem-pool-info p*) ∨ ´*alloc-lsize-r t* ⦄) (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*rule conjI*)
    apply *clarify*
    apply(*simp add*:*Mem-pool-alloc-rely-def*)
    apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
    apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)

    apply *clarify*
    apply(*simp add*:*Mem-pool-alloc-rely-def*)
    apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
    apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-7-stb*: *stable* (*mp-alloc-precond1-7 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-6-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond1-7-ext-stb*)
done

abbreviation *mp-alloc-precond1-70-ext t p sz tm* ≡
  ⦃(∀ *ii*<*length* (´*lsizes t*). ´*lsizes t* ! *ii* = (*ALIGN4* (*max-sz* (´*mem-pool-info p*))) *div* (*4* ^ *ii*))
    ∧ *length* (´*lsizes t*) ≤ *n-levels* (´*mem-pool-info p*)
    ∧ ´*alloc-l t* < *int* (*n-levels* (´*mem-pool-info p*))
    ∧ −*1* ≤ ´*free-l t* ∧ ´*free-l t* ≤ ´*alloc-l t*
    ∧ (´*alloc-l t* = −*1* ∧ ´*free-l t* = −*1* ∧ *length* (´*lsizes t*) = *1*
      ∨ (´*alloc-l t* ≥ *0* ∧ (∀ *ii*. *ii* ≤ *nat* (´*alloc-l t*) ⟶ ´*lsizes t* ! *ii* ≥ *sz*)
        ∧ ((´*alloc-l t* = *int* (*length* (´*lsizes t*)) − *1*) ∧ *length* (´*lsizes t*) = *n-levels* (´*mem-pool-info p*)
          ∨ ´*alloc-l t* = *int* (*length* (´*lsizes t*)) − *2* ∧ ´*lsizes t* ! *nat* (´*alloc-l t* + *1*) < *sz*)))⦄

abbreviation *mp-alloc-precond1-70 t p sz tm* ≡
  mp-alloc-precond1-1 t p sz tm ∩ *mp-alloc-precond1-70-ext t p sz tm*

lemma *mp-alloc-precond1-70-ext-stb*: *stable* (*mp-alloc-precond1-70-ext t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-70-stb*: *stable* (*mp-alloc-precond1-70 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-1-stb* **apply** *auto*[*1*]
  using *mp-alloc-precond1-70-ext-stb* **apply** *auto*
done

lemma *precnd17-bl-170*: *mp-alloc-precond1-7 t p sz tm* ⊆ *mp-alloc-precond1-70 t p sz tm*
  apply *clarify* **apply**(*case-tac i x t* = *0*)
    apply *clarify* **apply** *auto*[*1*]
    apply *clarify*
    apply(*rule IntI*) **apply** *auto*[*1*] **apply** *clarify*
    apply(*rule conjI*) **apply** *simp*
    apply(*rule conjI*) **apply** *simp*
    apply(*rule conjI*) **apply** *simp*
    apply *simp*
    apply(*case-tac alloc-lsize-r x t*) **apply** *auto*
done

abbreviation *mp-alloc-precond1-70-1 t p sz tm* ≡
  mp-alloc-precond1-70 t p sz tm ∩ ⦃´*alloc-l t* < *0*⦄

lemma *mp-alloc-precond1-70-1-ext-stb*: *stable* (⦃´*alloc-l t* < *0*⦄) (*Mem-pool-alloc-rely*

t)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-70-1-stb*: *stable* (*mp-alloc-precond1-70-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-70-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-70-1-ext-stb*)
done

abbreviation *mp-alloc-precond1-70-2 t p sz tm* ≡
mp-alloc-precond1-70 t p sz tm ∩ − ⦃´*alloc-l t* < *0*⦄

lemma *mp-alloc-precond1-70-2-ext-stb*: *stable* (− ⦃´*alloc-l t* < *0*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-70-2-stb*: *stable* (*mp-alloc-precond1-70-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-70-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-70-2-ext-stb*)
done

abbreviation *mp-alloc-precond1-70-2-1 t p sz tm* ≡
mp-alloc-precond1-70-2 t p sz tm ∩ ⦃´*free-l t* < *0*⦄

lemma *mp-alloc-precond1-70-2-1-ext-stb*: *stable* (⦃´*free-l t* < *0*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-70-2-1-stb*: *stable* (*mp-alloc-precond1-70-2-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)

using *mp-alloc-precond1-70-2-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-70-2-1-ext-stb*)
done

abbreviation *mp-alloc-precond1-70-2-2 t p sz tm* ≡
mp-alloc-precond1-70-2 t p sz tm ∩ − ⦃´*free-l t* < *0*⦄

lemma *mp-alloc-precond1-70-2-2-ext-stb*: *stable* (− ⦃´*free-l t* < *0*⦄) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond1-70-2-2-stb*: *stable* (*mp-alloc-precond1-70-2-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-70-2-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond1-70-2-2-ext-stb*)
done

lemma *alloc-memblk-data-valid-stb*:
blk x t = *buf* (*mem-pool-info x p*) +
block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*free-l x t*)) ∗
(*max-sz* (*mem-pool-info x p*) *div 4* ˆ *nat* (*free-l x t*)) ⟹
block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*free-l x t*)) < *n-max* (*mem-pool-info x p*) ∗ *4* ˆ *nat* (*free-l x t*) ⟹
allocating-node x t =
Some ⦇*pool* = *p*, *level* = *nat* (*free-l x t*), *block* = *block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*free-l x t*)),
data = *blk x t*⦈ ⟹
(*x*, *y*) ∈ *lvars-nochange-rel t* ⟹
(*x*, *y*) ∈ *gvars-conf-stable* ⟹
alloc-memblk-data-valid y p (*the* (*allocating-node y t*))
apply(*subgoal-tac blk x t* = *blk y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac lsizes x t* = *lsizes y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac free-l x t* = *free-l y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac allocating-node x t* = *allocating-node y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply (*simp add*: *gvars-conf-def gvars-conf-stable-def*)

done

abbreviation *mp-alloc-precond2-1-ext t p sz tm* ≡
{|(´*blk t* = *NULL* ∧ ´*allocating-node t* = *None*)
∨ (´*blk t* > *NULL* ∧ ´*alloc-memblk-data-valid p* (*the* (´*allocating-node t*))
∧ ´*allocating-node t* = *Some* (|*pool* = *p*, *level* = *nat* (´*free-l t*),
block = (*block-num* (´*mem-pool-info p*) (´*blk t*)
((´*lsizes t*)!(*nat* (´*free-l t*)))),
data = ´*blk t* |)
∧ (∃ *n*. *n* < *n-max* (´*mem-pool-info p*) ∗ (*4* ˆ (*nat* (´*free-l t*)))
∧ ´*blk t* = *buf* (´*mem-pool-info p*) + *n* ∗ (*max-sz* (´*mem-pool-info p*)
div (*4* ˆ (*nat* (´*free-l t*))))))|}

abbreviation *mp-alloc-precond2-1 t p sz tm* ≡
{*s*. *inv s*} ∩ {|´*freeing-node t* = *None*|} ∩ {|*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*|} ∩
mp-alloc-precond7-ext t p sz tm ∩ {|¬ ´*rf t*|}
∩ *mp-alloc-precond1-70-ext t p sz tm* ∩ − {|´*alloc-l t* < *0*|}
∩ − {|´*free-l t* < *0*|} ∩ *mp-alloc-precond2-1-ext t p sz tm*

term *mp-alloc-precond2-1 t p sz tm*

lemma *mp-alloc-freenode-stb*:
stable {|´*freeing-node t* = *None*|} (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

lemma *mp-alloc-precond2-1-ext-stb*: *stable* (*mp-alloc-precond2-1-ext t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*rule conjI*) **apply** *clarify* **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*rule impI*)+ **apply**(*rule allI*) **apply**(*rule impI*) **apply**(*rule disjI2*)
apply(*subgoal-tac buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply(*subgoal-tac free-l x t* = *free-l y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *smt*
apply(*subgoal-tac blk x t* = *blk y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac allocating-node x t* = *allocating-node y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac lsizes x t* = *lsizes y t*)

prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac n-max* (*mem-pool-info x p*) = *n-max* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *smt*
apply(*subgoal-tac block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*free-l x t*))
= *block-num* (*mem-pool-info y p*) (*blk y t*) (*lsizes y t* ! *nat* (*free-l y t*)))
prefer *2* **apply**(*simp add*: *block-num-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
done

lemma *mp-alloc-precond2-1-stb*: *stable* (*mp-alloc-precond2-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply (*simp add*: *stable-inv-alloc-rely1*)
apply (*simp add*: *mp-alloc-freenode-stb*)
apply (*simp add*: *mp-alloc-precond1-ext-stb*)
apply (*simp add*: *mp-alloc-precond7-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-0-ext-stb*)
using *mp-alloc-precond1-70-ext-stb* **apply** *blast*
apply (*simp add*: *mp-alloc-precond1-70-2-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-70-2-2-ext-stb*)
using *mp-alloc-precond2-1-ext-stb* **by** *blast*

abbreviation *mp-alloc-precond2-1-0 t p sz tm* ≡
mp-alloc-precond2-1 t p sz tm ∩ {|´*blk t* = *NULL*|}

lemma *mp-alloc-precond2-1-0-ext-stb*: *stable* ({|´*blk t* = *NULL*|}) (*Mem-pool-alloc-rely t*)
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond2-1-0-stb*: *stable* (*mp-alloc-precond2-1-0 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond2-1-stb* **apply** *auto*[*1*]
apply(*simp add*:*mp-alloc-precond2-1-0-ext-stb*)
done

abbreviation *mp-alloc-precond2-1-1 t p sz tm* ≡
  mp-alloc-precond2-1 t p sz tm ∩ −⦃´*blk t* = *NULL*⦄

term *mp-alloc-precond2-1-1 t p sz tm*

lemma *mp-alloc-precond2-1-1-ext-stb*: *stable* (−⦃´*blk t* = *NULL*⦄) (*Mem-pool-alloc-rely t*)
  apply(*simp add*:*stable-def*) **apply** *clarify*
  apply(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
  apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond2-1-1-stb*: *stable* (*mp-alloc-precond2-1-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond2-1-stb* **apply** *auto*[*1*]
  apply(*simp add*:*mp-alloc-precond2-1-1-ext-stb*)
done

abbreviation *mp-alloc-precond2-1-1-loopinv-ext t p sz tm* ≡
  −⦃´*blk t* = *NULL*⦄ ∩ ⦃´*from-l t* ≤ ´*alloc-l t* ∧ ´*from-l t* ≥ ´*free-l t* ∧ ´*allocating-node t* = *Some* (|*pool* = *p*, *level* = *nat* (´*from-l t*),
    block = *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*from-l t*))),
    data = ´*blk t* |)
  ∧ ´*alloc-memblk-data-valid p* (*the* (´*allocating-node t*))
  ∧ (∃ *n*. *n* < *n-max* (´*mem-pool-info p*) ∗ (*4* ˆ (*nat* (´*from-l t*)))
    ∧ ´*blk t* = *buf* (´*mem-pool-info p*) + *n* ∗ (*max-sz* (´*mem-pool-info p*) *div* (*4* ˆ (*nat* (´*from-l t*))))) ⦄

abbreviation *mp-alloc-precond2-1-1-loopinv t p sz tm* ≡
  {*s*. *inv s*} ∩ ⦃´*freeing-node t* = *None*⦄ ∩ ⦃*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*⦄ ∩ *mp-alloc-precond7-ext t p sz tm* ∩ ⦃¬ ´*rf t*⦄
  ∩ *mp-alloc-precond1-70-ext t p sz tm* ∩ − ⦃´*alloc-l t* < *0*⦄
  ∩ − ⦃´*free-l t* < *0*⦄ ∩ *mp-alloc-precond2-1-1-loopinv-ext t p sz tm*

lemma *alloc-memblk-data-valid-stb2*:
  blk x t = *buf* (*mem-pool-info x p*) +
    block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*from-l x t*)) ∗
    (*max-sz* (*mem-pool-info x p*) *div 4* ˆ *nat* (*from-l x t*)) =⇒
  block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*from-l x t*)) < *n-max* (*mem-pool-info x p*) ∗ *4* ˆ *nat* (*from-l x t*) =⇒
  allocating-node x t =
  Some (|*pool* = *p*, *level* = *nat* (*from-l x t*), *block* = *block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*from-l x t*)),
    data = *blk x t*|) =⇒

$(x, y) \in$ *lvars-nochange-rel t* $\Longrightarrow$
$(x, y) \in$ *gvars-conf-stable* $\Longrightarrow$
alloc-memblk-data-valid y p (the (allocating-node y t))
apply(*subgoal-tac blk x t = blk y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac buf (mem-pool-info x p) = buf (mem-pool-info y p)*)
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac lsizes x t = lsizes y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac from-l x t = from-l y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac max-sz (mem-pool-info x p) = max-sz (mem-pool-info y p)*)
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac allocating-node x t = allocating-node y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply (*simp add*: *gvars-conf-def gvars-conf-stable-def*)
done

lemma *mp-alloc-precond2-1-1-loopinv-ext-stb*: *stable (mp-alloc-precond2-1-1-loopinv-ext t p sz tm) (Mem-pool-alloc-rely t)*
apply(*rule stable-int2*)
apply (*simp add*: *mp-alloc-precond2-1-1-ext-stb*)

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*subgoal-tac buf (mem-pool-info x p) = buf (mem-pool-info y p)*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply(*subgoal-tac from-l x t = from-l y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac max-sz (mem-pool-info x p) = max-sz (mem-pool-info y p)*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *smt*
apply(*subgoal-tac blk x t = blk y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac allocating-node x t = allocating-node y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac lsizes x t = lsizes y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac n-max (mem-pool-info x p) = n-max (mem-pool-info y p)*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *smt*
apply(*subgoal-tac block-num (mem-pool-info x p) (blk x t) (lsizes x t ! nat (from-l x t))*
= block-num (mem-pool-info y p) (blk y t) (lsizes y t ! nat (from-l y t)))

prefer *2* **apply**(*simp add*: *block-num-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*

done

lemma *mp-alloc-precond2-1-1-loopinv-stb*: *stable* (*mp-alloc-precond2-1-1-loopinv t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)

apply (*simp add*: *stable-inv-alloc-rely1*)
apply (*simp add*: *mp-alloc-freenode-stb*)
apply (*simp add*: *mp-alloc-precond1-ext-stb*)
apply (*simp add*: *mp-alloc-precond7-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-0-ext-stb*)
using *mp-alloc-precond1-70-ext-stb* **apply** *blast*
apply (*simp add*: *mp-alloc-precond1-70-2-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-70-2-2-ext-stb*)
using *mp-alloc-precond2-1-1-loopinv-ext-stb* **apply** *auto*[*1*]
done

abbreviation *mp-alloc-precond2-1-2 t p sz tm* ≡
{*s*. *inv s*} ∩ {|´*freeing-node t* = *None*|} ∩ {|*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*|} ∩ *mp-alloc-precond7-ext t p sz tm* ∩ {|¬ ´*rf t*|}
∩ *mp-alloc-precond1-70-ext t p sz tm* ∩ − {|´*alloc-l t* < *0*|} ∩ − {|´*free-l t* < *0*|} ∩ −{|´*blk t* = *NULL*|}
∩ {|´*allocating-node t* = *Some* (|*pool* = *p*, *level* = *nat* (´*alloc-l t*),
block = *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*alloc-l t*))),
data = ´*blk t* |) ∧ ´*alloc-memblk-data-valid p* (*the* (´*allocating-node t*)) |}

term *mp-alloc-precond2-1-1-loopinv t p sz tm*
term *mp-alloc-precond2-1-2 t p sz tm*

lemma *alloc-memblk-data-valid-stb3*:
blk x t = *buf* (*mem-pool-info x p*) +
block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*alloc-l x t*)) ∗
(*max-sz* (*mem-pool-info x p*) *div 4* ˆ *nat* (*alloc-l x t*)) =⇒
block-num (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*alloc-l x t*)) < *n-max* (*mem-pool-info x p*) ∗ *4* ˆ *nat* (*alloc-l x t*) =⇒

allocating-node x t =
Some (|*pool* = *p*, *level* = *nat* (*alloc-l x t*), *block* = *block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*alloc-l x t*)),
data = *blk x t*|) $\Longrightarrow$
(*x*, *y*) ∈ *lvars-nochange-rel t* $\Longrightarrow$
(*x*, *y*) ∈ *gvars-conf-stable* $\Longrightarrow$
alloc-memblk-data-valid y p (*the* (*allocating-node y t*))
apply(*subgoal-tac blk x t* = *blk y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac lsizes x t* = *lsizes y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac alloc-l x t* = *alloc-l y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac allocating-node x t* = *allocating-node y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply (*simp add*: *gvars-conf-def gvars-conf-stable-def*)
done

lemma *mp-alloc-precond2-1-2-stb*: *stable* (*mp-alloc-precond2-1-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply (*simp add*: *stable-inv-alloc-rely1*)
apply (*simp add*: *mp-alloc-freenode-stb*)
apply (*simp add*: *mp-alloc-precond1-ext-stb*)
apply (*simp add*: *mp-alloc-precond7-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-0-ext-stb*)
using *mp-alloc-precond1-70-ext-stb* **apply** *blast*
apply (*simp add*: *mp-alloc-precond1-70-2-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-70-2-2-ext-stb*)
apply (*simp add*: *mp-alloc-precond2-1-1-ext-stb*)

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*: *block-num-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac blk x t* = *blk y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac lsizes x t* = *lsizes y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)

apply(*subgoal-tac alloc-l x t = alloc-l y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac allocating-node x t = allocating-node y t*)
prefer *2* **apply**(*simp add*: *lvars-nochange-rel-def lvars-nochange-def*)
apply(*subgoal-tac n-max* (*mem-pool-info x p*) = *n-max* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*alloc-l x t*))
= *block-num* (*mem-pool-info y p*) (*blk y t*) (*lsizes y t* ! *nat* (*alloc-l y t*)))
prefer *2* **apply**(*simp add*: *block-num-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
by (*metis Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*) *Mem-block.select-convs*(*4*) *option.sel*)

abbreviation *mp-alloc-precond2-1-3 t p sz tm* ≡
mp-alloc-precond1-70-2-2 t p sz tm ∩ −⦃´*blk t* = *NULL*⦄
∩ ⦃´*alloc-blk-valid p* (*nat* (´*alloc-l t*)) (*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*alloc-l t*))))
(´*blk t*) ∧ ´*allocating-node t* = *None*⦄

lemma *mp-alloc-precond2-1-3-stb*: *stable* (*mp-alloc-precond2-1-3 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-70-2-2-stb* **apply** *auto*[*1*]

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def block-num-def*)
done

abbreviation *mp-alloc-precond2-1-4 t p sz tm* ≡
mp-alloc-precond1 t p tm ∩ ⦃¬´*rf t*⦄ ∩ ⦃ (*tm* = *FOREVER* ⟶ ´*tmout t* = *FOREVER*) ⦄
∩ {*s*. (∃ *mblk*. *mempoolalloc-ret s t* = *Some mblk* ∧ *alloc-memblk-valid s p sz*

mblk)}

lemma *mp-alloc-precond2-1-4-stb*: *stable* (*mp-alloc-precond2-1-4 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-stb* **apply** *auto*[*1*]
apply(*simp add*:*stable-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*simp add*:*stable-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *auto*[*1*]
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*alloc-memblk-valid-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
done

abbreviation *mp-alloc-precond1-8 t p sz tm* ≡
mp-alloc-precond1 t p tm ∩ ⦃¬´*rf t*⦄ ∩ ⦃ (*tm* = *FOREVER* ⟶ ´*tmout t* = *FOREVER*) ⦄
∩ {*s*. (*ret s t* = *OK* ∧ (∃ *mblk*. *mempoolalloc-ret s t* = *Some mblk* ∧ *alloc-memblk-valid s p sz mblk*))
∨ ((*ret s t* = *ESIZEERR* ∨ *ret s t* = *EAGAIN* ∨ *ret s t* = *ENOMEM*) ∧ *mempoolalloc-ret s t* = *None*) }

lemma *mp-alloc-precond1-8-stb*: *stable* (*mp-alloc-precond1-8 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-stb* **apply** *auto*[*1*]
apply(*simp add*:*stable-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply (*smt mem-Collect-eq mp-alloc-precond2-ext-stb stable-def*)

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*rule conjI*)
apply *clarify*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x*=*y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*alloc-memblk-valid-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply *clarify*

apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

abbreviation *mp-alloc-precond1-8-0 t p sz tm* ≡
mp-alloc-precond1 t p tm ∩ ⦃ (*tm* = *FOREVER* ⟶ ´*tmout t* = *FOREVER*) ⦄
∩ {*s*. (*ret s t* = *OK* ∧ (∃ *mblk*. *mempoolalloc-ret s t* = *Some mblk* ∧ *alloc-memblk-valid s p sz mblk*))
∨ ((*ret s t* = *ESIZEERR* ∨ *ret s t* = *EAGAIN* ∨ *ret s t* = *ENOMEM*) ∧ *mempoolalloc-ret s t* = *None*) }

lemma *mp-alloc-precond1-8-0-stb*: *stable* (*mp-alloc-precond1-8-0 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-stb* **apply** *auto*[*1*]
apply (*smt mem-Collect-eq mp-alloc-precond2-ext-stb stable-def*)

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*rule conjI*)
apply *clarify*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*alloc-memblk-valid-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def*)
apply(*case-tac x=y*) **apply** *simp* **apply** *clarify*
apply(*simp add*:*lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

abbreviation *mp-alloc-precond1-8-1 t p sz tm* ≡
mp-alloc-precond1-8 t p sz tm
∩ ⦃´*ret t* = *OK* ∨ *tm* = *NOWAIT* ∨ ´*ret t* = *ESIZEERR*⦄

lemma *mp-alloc-precond1-8-1-stb*: *stable* (*mp-alloc-precond1-8-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)

apply *auto*
done

abbreviation *mp-alloc-precond1-8-1-1 t p sz tm* ≡
mp-alloc-precond1-8-0 t p sz tm ∩ ⦃´*ret t* = *OK* ∨ *tm* = *NOWAIT* ∨ ´*ret t* = *ESIZEERR*⦄ ∩ ⦃´*rf t* = *True*⦄

lemma *mp-alloc-precond1-8-1-1-stb*: *stable* (*mp-alloc-precond1-8-1-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-8-0-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-1-2 t p sz tm* ≡
mp-alloc-precond1-8-1-1 t p sz tm ∩ ⦃´*ret t* = *EAGAIN*⦄

lemma *mp-alloc-precond1-8-1-2-stb*: *stable* (*mp-alloc-precond1-8-1-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-1-1-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-1-3 t p sz tm* ≡
mp-alloc-precond1-8-1-1 t p sz tm ∩ −⦃´*ret t* = *EAGAIN*⦄

lemma *mp-alloc-precond1-8-1-3-stb*: *stable* (*mp-alloc-precond1-8-1-3 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-1-1-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-2 t p sz tm* ≡
mp-alloc-precond1-8 t p sz tm
∩ −⦃´*ret t* = *OK* ∨ *tm* = *NOWAIT* ∨ ´*ret t* = *ESIZEERR*⦄

lemma *mp-alloc-precond1-8-2-stb*: *stable* (*mp-alloc-precond1-8-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)

done

abbreviation *mp-alloc-precond1-8-2-1 t p sz tm* ≡
mp-alloc-precond1-8-2 t p sz tm ∩ {´*ret t* = *EAGAIN*}

lemma *mp-alloc-precond1-8-2-1-stb*: *stable* (*mp-alloc-precond1-8-2-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-2-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-2-2 t p sz tm* ≡
mp-alloc-precond1-8-2 t p sz tm ∩ −{´*ret t* = *EAGAIN*}

lemma *mp-alloc-precond1-8-2-2-stb*: *stable* (*mp-alloc-precond1-8-2-2 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-2-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-2-3 t p sz tm* ≡
mp-alloc-precond1-8-2-2 t p sz tm ∩ {´*tmout t* ≠ *FOREVER*} ∩ {*tm* > *0*}

lemma *mp-pred1823-eq*: *mp-alloc-precond1-8-2-3 t p sz tm* = *mp-alloc-precond1-8-2-2 t p sz tm* ∩ {´*tmout t* ≠ *FOREVER*}
by *auto*

lemma *mp-alloc-precond1-8-2-3-stb*: *stable* (*mp-alloc-precond1-8-2-3 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*)
using *mp-alloc-precond1-8-2-2-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *simp*
done

abbreviation *mp-alloc-precond1-8-2-20 t p sz tm* ≡
mp-alloc-precond1-8-2-2 t p sz tm ∩ −{´*tmout t* ≠ *FOREVER*}

lemma *mp-alloc-precond1-8-2-20-stb*: *stable* (*mp-alloc-precond1-8-2-20 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond1-8-2-2-stb* **apply** *auto*[*1*]
apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)

done

abbreviation *mp-alloc-precond1-8-2-4 t p sz tm* ≡ *mp-alloc-precond1-8-2-2 t p sz tm* ∩ ⦃*tm* > *0*⦄
lemma *mp-alloc-precond1-8-2-4-stb*: *stable* (*mp-alloc-precond1-8-2-4 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-8-2-2-stb* **apply** *auto*[*1*]
  apply(*unfold stable-def*) **apply** *simp*
done

abbreviation *mp-alloc-precond1-8-2-40 t p sz tm* ≡
  mp-alloc-precond1-8-2-4 t p sz tm ∩ ⦃´*tmout t* < *0*⦄

lemma *mp-alloc-precond1-8-2-40-stb*: *stable* (*mp-alloc-precond1-8-2-40 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-8-2-4-stb* **apply** *blast*
  apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

term *mp-alloc-precond1-8-2-40 t p sz tm*

abbreviation *mp-alloc-precond1-8-2-41 t p sz tm* ≡
  mp-alloc-precond1-8-2-4 t p sz tm ∩ −⦃´*tmout t* < *0*⦄

lemma *mp-alloc-precond1-8-2-41-stb*: *stable* (*mp-alloc-precond1-8-2-41 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*)
  using *mp-alloc-precond1-8-2-4-stb* **apply** *blast*
  apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

abbreviation *mp-alloc-precond1-8-2-5 t p sz tm* ≡
  mp-alloc-precond1-8-0 t p sz tm ∩ ⦃*tm* > *0*⦄ ∩ −⦃´*ret t* = *OK* ∨ *tm* = *NOWAIT* ∨ ´*ret t* = *ESIZEERR*⦄ ∩ −⦃´*ret t* = *EAGAIN*⦄
    ∩ ⦃´*tmout t* < *0*⦄ ∩ ⦃´*rf t*⦄
term *mp-alloc-precond1-8-2-5 t p sz tm*

lemma *mp-alloc-precond1-8-2-5-stb*: *stable* (*mp-alloc-precond1-8-2-5 t p sz tm*) (*Mem-pool-alloc-rely t*)
  apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
  using *mp-alloc-precond1-8-0-stb* **apply** *blast*
  apply(*unfold stable-def*) **apply**(*simp add*:*Mem-pool-alloc-rely-def*)
  apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
  apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)

apply(*simp add:Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*simp add:Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
done

21.2 proof of each statement

21.3 stm1

lemma *mp-alloc-stm1-lm0*:
cur V = *Some t* $\Longrightarrow$ *inv V* $\Longrightarrow$
V⦇*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])⦈
∈ ⦃´(*Pair V*) ∈ *Mem-pool-alloc-guar t*⦄
apply *auto* **apply**(*simp add:Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*⦇*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add:lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm1-lm1*: *mp-alloc-precond1-6-10 t p sz timeout* α ∩ ⦃´*cur* = *Some t*⦄ ∩ {*V*}
⊆ ⦃´(*lsizes-update* (λ-. ´*lsizes*(*t* := ´*lsizes t* @ [*ALIGN4* (´*lsizes t* ! (´*i t* − *Suc NULL*) *div 4*)])))
∈ ⦃´(*Pair V*) ∈ *Mem-pool-alloc-guar t*⦄ ∩
mp-alloc-precond1-6-2 t p sz timeout α⦄
apply *clarify*
apply(*rule IntI*) **using** *mp-alloc-stm1-lm0* **apply** *blast*
apply(*rule IntI*) **prefer** *2* **apply** *clarsimp*
apply(*subgoal-tac lsizes* (*V*⦇*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])⦈) *t*
= *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])
prefer *2* **apply** *auto*[*1*]
apply(*simp add*: *subst*[**where** *s*=*lsizes* (*V*⦇*lsizes* := (*lsizes V*)(*t* := *lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)])⦈) *t*
and *t*=*lsizes V t* @ [*ALIGN4* (*lsizes V t* ! (*i V t* − *Suc NULL*) *div 4*)]])

apply(*rule conjI*) **apply** *clarify*
apply(*case-tac ii* < *length* (*lsizes V t*)) **apply** (*metis nth-append*)
apply(*case-tac ii* = *length* (*lsizes V t*))
apply(*subgoal-tac* (*lsizes V t* @ [*ALIGN4* (*ALIGN4* (*max-sz* (*mem-pool-info V p*)) *div 4* ˆ (*i V t* − *Suc NULL*) *div 4*)]) ! *ii*
= *ALIGN4* (*ALIGN4* (*max-sz* (*mem-pool-info V p*)) *div 4* ˆ (*i V t* − *Suc NULL*) *div 4*))
prefer *2* **apply** (*meson nth-append-length*)
apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info V p*)) *div 4* ˆ (*i V t* −

Suc NULL) div 4
= ALIGN4 (max-sz (mem-pool-info V p)) div 4 ^ ii)
prefer 2 apply (metis Divides.div-mult2-eq One-nat-def power-minus-mult zero-le-numeral)
apply(subgoal-tac (ALIGN4 (max-sz (mem-pool-info V p)) div 4 ^ ii) mod 4 = 0)
prefer 2 apply(subgoal-tac ∃ n>0. max-sz (mem-pool-info V p) = (4 * n) * (4 ^ n-levels (mem-pool-info V p)))
prefer 2 apply(simp add:inv-def inv-mempool-info-def) apply metis
apply (metis (no-types, lifting) inv-maxsz-align4 less-imp-le-nat
m-mod-div mod-mult-self1-is-0 mult.assoc pow-mod-0)
apply (metis align40)
apply linarith
apply (simp add: le-nat-iff nth-append)

apply(rule IntI) prefer 2 apply clarify apply auto[1]
apply(rule IntI) prefer 2 apply clarify apply auto[1]
apply(rule IntI) prefer 2 apply clarify apply auto[1]
apply(rule IntI) apply simp apply simp-inv apply metis
apply simp
done

lemma mp-alloc-stm1-lm:
Γ $\vdash_I$ Some (IF ´i t > 0 THEN
(t ► ´lsizes := ´lsizes(t := ´lsizes t @ [ALIGN4 (´lsizes t ! (´i t − 1) div 4)]))
FI) sat_p [mp-alloc-precond1-6-1 t p sz timeout α, Mem-pool-alloc-rely t,
Mem-pool-alloc-guar t, mp-alloc-precond1-6-2 t p sz timeout α]
apply(rule Cond)
using mp-alloc-precond1-6-1-stb apply simp

apply(unfold stm-def)
apply(rule Await)
using mp-alloc-precond1-6-10-stb[of t p timeout sz α] apply fast
using mp-alloc-precond1-6-2-stb apply simp
apply(rule allI)
apply(rule Basic)
apply(case-tac mp-alloc-precond1-6-10 t p sz timeout α ∩ {|´cur = Some t|} ∩ {V} = {})
apply auto[1]
using mp-alloc-stm1-lm1[of t p timeout sz α] apply auto[1]
apply simp using stable-id2 apply auto[1]
using stable-id2 apply auto[1]

apply(unfold Skip-def)
apply(rule Basic) apply clarify
apply simp
apply(simp add:Mem-pool-alloc-guar-def) apply auto[1]
using mp-alloc-precond1-6-11-stb[of t p timeout sz α] apply fast

using *mp-alloc-precond1-6-2-stb* **apply** *fast*
apply(*simp add*:*Mem-pool-alloc-guar-def*)
done

21.4 stm2

lemma *mp-alloc-stm2-lm2-1*:
$cur\ V = Some\ t \Longrightarrow inv\ V \Longrightarrow\ V(|alloc\text{-}lsize\text{-}r := (alloc\text{-}lsize\text{-}r\ V)(t := True)|)$
$\in \{|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}alloc\text{-}guar\ t|\}$
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* $(V, V(|alloc\text{-}lsize\text{-}r := (alloc\text{-}lsize\text{-}r\ V)(t := True)|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm2-lm2*:
$mp\text{-}alloc\text{-}precond1\text{-}6\text{-}20\ t\ p\ sz\ timeout\ \alpha \cap \{|\acute{}cur = Some\ t|\} \cap \{V\}$
$\subseteq \{|\acute{}(alloc\text{-}lsize\text{-}r\text{-}update\ (\lambda\text{-}.\ \acute{}alloc\text{-}lsize\text{-}r(t := True)))$
$\in \{|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}alloc\text{-}guar\ t|\} \cap mp\text{-}alloc\text{-}lsizestm\text{-}loopinv\ t\ p\ sz\ timeout\ 0\ |\}$
apply *clarify*
apply(*rule IntI*)
using *mp-alloc-stm2-lm2-1* **apply** *simp*
apply(*rule IntI*) **prefer** *2*
apply(*case-tac* $i\ V\ t = 0$) **apply**(*simp add*:*inv-def inv-mempool-info-def*) **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply** *auto*[*1*]
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply**(*simp add*:*alloc-memblk-valid-def*)
apply *simp* **apply**(*subgoal-tac* $(V, V(|alloc\text{-}lsize\text{-}r := (alloc\text{-}lsize\text{-}r\ V)(t := True)|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm2-lm4-1*:
$cur\ V = Some\ t \Longrightarrow inv\ V \Longrightarrow V(|alloc\text{-}l := (alloc\text{-}l\ V)(t := int\ (i\ V\ t))|) \in \{|\acute{}(Pair\ V) \in Mem\text{-}pool\text{-}alloc\text{-}guar\ t|\}$
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* $(V, V(|alloc\text{-}l := (alloc\text{-}l\ V)(t := int\ (i\ V\ t))|)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm2-lm4*:
$mp\text{-}alloc\text{-}precond1\text{-}6\text{-}21\ t\ p\ sz\ timeout\ \alpha \cap \{|\acute{}cur = Some\ t|\} \cap \{V\}$

$\subseteq$ {|´(*alloc-l-update* (λ-. ´*alloc-l*(t := *int* (´*i t*))))
$\in$ {|´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*|} $\cap$
mp-alloc-precond1-6-21-1 t p sz timeout α|}
apply *clarify*
apply(*rule IntI*)
using *mp-alloc-stm2-lm4-1* **apply** *simp*
apply(*rule IntI*) **prefer** *2*
apply(*case-tac i V t = 0*) **apply** *simp* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply**(*simp add*:*alloc-memblk-valid-def*)
apply *simp*
apply(*subgoal-tac* (V,V(|*alloc-l* := (*alloc-l V*)(t := *int* (*i V t*))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

done

lemma *mp-alloc-stm2-lm5-1-1*:
cur V = *Some t* $\Longrightarrow$ *inv V* $\Longrightarrow$ V(|*free-l* := (*free-l V*)(t := *int* (*i V t*))|) $\in$ {|´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*|}
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (V,V(|*free-l* := (*free-l V*)(t := *int* (*i V t*))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm2-lm5-1*:
mp-alloc-precond1-6-21-1 t p sz timeout α $\cap$ {|´*cur* = *Some t*|} $\cap$ {V}
$\subseteq$ {|´(*free-l-update* (λ-. ´*free-l*(t := *int* (´*i t*))))
$\in$ {|´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*|} $\cap$
mp-alloc-precond1-6-21-2 t p sz timeout α|}
apply *clarify*
apply(*rule IntI*)
using *mp-alloc-stm2-lm5-1-1* **apply** *simp*
apply(*rule IntI*) **prefer** *2*
apply(*case-tac i V t = 0*) **apply** *simp* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*rule IntI*) **prefer** *2* **apply**(*simp add*:*alloc-memblk-valid-def*)
apply *simp*
apply(*subgoal-tac* (V,V(|*free-l* := (*free-l V*)(t := *int* (*i V t*))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

done

lemma *mp-alloc-stm2-lm5*:
$\Gamma \vdash_I$ *Some* $(t \blacktriangleright$ ´*free-l* := ´*free-l* $(t := int$ (´*i t*)))
sat$_p$ [*mp-alloc-precond1-6-21-1 t p sz timeout* α, *Mem-pool-alloc-rely t*,
Mem-pool-alloc-guar t, *mp-alloc-precond1-6-21-2 t p sz timeout* α]
apply(*unfold stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-6-21-1-stb* **apply** *simp*
using *mp-alloc-precond1-6-21-2-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-6-21-1 t p sz timeout* $\alpha \cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*] **using** *mp-alloc-stm2-lm5-1*[*of t p timeout sz* α] **apply** *auto*[*1*]
apply *simp* **using** *stable-id2* **apply** *auto*[*1*]
using *stable-id2* **apply** *auto*[*1*]
done

lemma *mp-alloc-stm2-lm6*:
$\Gamma \vdash_I$ *Some SKIP sat*$_p$ [*mp-alloc-precond1-6-21-1 t p sz timeout* α, *Mem-pool-alloc-rely t*,
Mem-pool-alloc-guar t, *mp-alloc-precond1-6-21-2 t p sz timeout* α]
apply(*unfold Skip-def*)
apply(*rule Basic*)
apply *clarify* **apply**(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*)
apply *simp+* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*) **apply** *auto*[*1*]
using *mp-alloc-precond1-6-21-1-stb* **apply** *simp*
using *mp-alloc-precond1-6-21-2-stb* **apply** *simp*
done

lemma *mp-alloc-stm2-lm7-1*:
cur V = *Some t* $\Longrightarrow$ *inv V* $\Longrightarrow$ *V*(|*i* := (*i V*)(*t* := (*i V t*) + *1*)|) $\in$ {|´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*|}
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*(|*i* := (*i V*)(*t* := (*i V t*) + *1*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm2-lm7*:
mp-alloc-precond1-6-21-2 t p sz timeout $\alpha \cap$ {|´*cur* = *Some t*|} $\cap$ {*V*}
$\subseteq$ {|´(*i-update* (λ-. ´*i*(*t* := *Suc* (´*i t*))))
$\in$ {|´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*|} $\cap$
mp-alloc-lsizestm-loopinv t p sz timeout (α − *1*)|}
apply *clarify*
apply(*rule IntI*)

using *mp-alloc-stm2-lm7-1* **apply** *simp*
apply(*rule IntI*) **prefer** *2*
apply(*case-tac* $i\ V\ t = 0$) **apply** *simp*
apply(*simp add*:*inv-def inv-mempool-info-def*)
apply(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*)
apply *clarsimp* **apply**(*subgoal-tac* $(V, V(\!|i := (i\ V)(t := (i\ V\ t) + 1)|\!)) \in$ *lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *clarify* **apply**(*case-tac* $ii < i\ V\ t$) **apply** *auto*[*1*]
apply(*case-tac* $ii = i\ V\ t$) **apply** *simp* **apply** *simp*
apply *clarsimp*
done

lemma *subset-un-I1*[*intro*]: $A \subseteq B \Longrightarrow A \subseteq B \cup C$ **by** *auto*
lemma *subset-un-I2*[*intro*]: $A \subseteq C \Longrightarrow A \subseteq B \cup C$ **by** *auto*

lemma *mp-alloc-stm2-lm8*:
$P \subseteq \{\!|\,\acute{}(\textit{alloc-lsize-r-update}\ (\lambda\text{-}.\ \acute{}\textit{alloc-lsize-r}(t := \textit{True})))$
$\in \{\!|\,\acute{}(\textit{Pair}\ V) \in \textit{Mem-pool-alloc-guar}\ t|\!\} \cap B\ |\!\} \Longrightarrow$
$P \subseteq \{\!|\,\acute{}(\textit{alloc-lsize-r-update}\ (\lambda\text{-}.\ \acute{}\textit{alloc-lsize-r}(t := \textit{True})))$
$\in \{\!|\,\acute{}(\textit{Pair}\ V) \in \textit{Mem-pool-alloc-guar}\ t|\!\} \cap (A \cup B)\ |\!\}$
apply *auto*
done

lemma *mp-alloc-stm2-lm9*:
$P \subseteq \{\!|\,\acute{}(\textit{i-update}\ (\lambda\text{-}.\ \acute{}i(t := \textit{Suc}\ (\acute{}i\ t))))$
$\in \{\!|\,\acute{}(\textit{Pair}\ V) \in \textit{Mem-pool-alloc-guar}\ t|\!\} \cap A|\!\} \Longrightarrow$
$P \subseteq \{\!|\,\acute{}(\textit{i-update}\ (\lambda\text{-}.\ \acute{}i(t := \textit{Suc}\ (\acute{}i\ t))))$
$\in \{\!|\,\acute{}(\textit{Pair}\ V) \in \textit{Mem-pool-alloc-guar}\ t|\!\} \cap (A \cup B)|\!\}$
by *auto*

lemma *mp-alloc-stm2-lm*:
$\Gamma \vdash_I$ *Some* (*IF* ´*lsizes t* ! ´*i t* < *sz THEN*
t ► ´*alloc-lsize-r* := ´*alloc-lsize-r*(*t* := *True*)
ELSE (*t* ► ´*alloc-l* := ´*alloc-l*(*t* := *int* (´*i t*)));;
IF ¬ *level-empty* (´*mem-pool-info p*) (´*i t*) *THEN*
t ► ´*free-l* := ´*free-l*(*t* := *int* (´*i t*))
FI;;
(*t* ► ´*i* := ´*i*(*t* := *Suc* (´*i t*)))
FI) sat_p [*mp-alloc-precond1-6-2 t p sz timeout* α, *Mem-pool-alloc-rely t*,
Mem-pool-alloc-guar t, *mp-alloc-lsizestm-loopinv t p sz timeout* (α − *1*)
∪ *mp-alloc-lsizestm-loopinv t p sz timeout 0*]
apply(*rule Cond*)
using *mp-alloc-precond1-6-2-stb* **apply** *simp*

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-6-20-stb* **apply** *simp*
apply(*rule stable-un2*)
using *mp-alloc-lsizestm-loopinv-stb* **apply** *fast*
using *mp-alloc-lsizestm-loopinv-stb* **apply** *fast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-6-20 t p sz timeout* α $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*]
apply(*rule mp-alloc-stm2-lm8*[*of - t - mp-alloc-lsizestm-loopinv t p sz timeout 0 mp-alloc-lsizestm-loopinv t p sz timeout* (α − *1*)])
using *mp-alloc-stm2-lm2*[*of t p timeout sz* α] **apply** *meson*
apply *simp* **using** *stable-id2* **apply** *auto*[*1*]
using *stable-id2* **apply** *auto*[*1*]

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-6-21-2 t p sz timeout* α])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-6-21-1 t p sz timeout* α])

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-6-21-stb* **apply** *simp*
using *mp-alloc-precond1-6-21-1-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-6-21 t p sz timeout* α $\cap$ {|´*cur* = *Some t*|} $\cap$ {*V*} = {})
apply *auto*[*1*] **using** *mp-alloc-stm2-lm4*[*of t p timeout sz*] **apply** *presburger*
apply *simp* **using** *stable-id2* **apply** *fast*
using *stable-id2* **apply** *fast*

apply(*rule Cond*)
using *mp-alloc-precond1-6-21-1-stb* **apply** *simp*

using *Conseq*[**where** *pre*=*mp-alloc-precond1-6-21-1 t p sz timeout* α $\cap$ {|¬ *level-empty* (´*mem-pool-info p*) (´*i t*)|}
and *pre'*=*mp-alloc-precond1-6-21-1 t p sz timeout* α **and** *rely*=*Mem-pool-alloc-rely t*
and *rely'*=*Mem-pool-alloc-rely t* **and** *guar*=*Mem-pool-alloc-guar t* **and** *guar'*=*Mem-pool-alloc-guar t*
and *post*=*mp-alloc-precond1-6-21-2 t p sz timeout* α **and** *post'*=*mp-alloc-precond1-6-21-2 t p sz timeout* α
and *P*=*Some* (*t* ▶ ´*free-l* := ´*free-l* (*t* := *int* (´*i t*)))]
mp-alloc-stm2-lm5[*of t p timeout sz*] **apply** *fast*

using *Conseq*[**where** *pre=mp-alloc-precond1-6-21-1 t p sz timeout* $\alpha \cap -$ ⦃¬ *level-empty* (´*mem-pool-info p*) (´*i t*)⦄
and *pre′=mp-alloc-precond1-6-21-1 t p sz timeout* α **and** *rely=Mem-pool-alloc-rely t*
and *rely′=Mem-pool-alloc-rely t* **and** *guar=Mem-pool-alloc-guar t* **and** *guar′=Mem-pool-alloc-guar t*
and *post=mp-alloc-precond1-6-21-2 t p sz timeout* α **and** *post′=mp-alloc-precond1-6-21-2 t p sz timeout* α
and *P=Some SKIP*]
mp-alloc-stm2-lm6[*of t p timeout sz*] **apply** *fast*
apply(*simp add:Mem-pool-alloc-guar-def*)

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-6-21-2-stb* **apply** *simp*
apply(*rule stable-un2*)
using *mp-alloc-lsizestm-loopinv-stb* **apply** *fast*
using *mp-alloc-lsizestm-loopinv-stb* **apply** *fast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-6-21-2 t p sz timeout* $\alpha \cap$ ⦃´*cur = Some t*⦄ $\cap$ {*V*} = {})
apply *auto*[*1*]
apply(*rule mp-alloc-stm2-lm9*[*of - t - mp-alloc-lsizestm-loopinv t p sz timeout* ($\alpha - 1$) *mp-alloc-lsizestm-loopinv t p sz timeout 0*])
using *mp-alloc-stm2-lm7*[*of t p timeout sz* α] **apply** *meson*
apply *simp* **using** *stable-id2* **apply** *fast*
using *stable-id2* **apply** *fast*

apply(*simp add:Mem-pool-alloc-guar-def*)
done

term ⦃´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*⦄ ∩ *mp-alloc-precond2-1 t p sz timeout*

21.5 lsize while loop

abbreviation *alloc-lsize-loopbody t p sz* ≡
IF ´*i t* > *0 THEN*
(*t* ▶ ´*lsizes* := ´*lsizes*(*t* := ´*lsizes t* @ [*ALIGN4* (´*lsizes t* ! (´*i t* − *1*) *div 4*)]))
FI;;
IF ´*lsizes t* ! ´*i t* < *sz THEN*
(*t* ▶ ´*alloc-lsize-r* := ´*alloc-lsize-r*(*t* := *True*))
ELSE
(*t* ▶ ´*alloc-l* := ´*alloc-l*(*t* := *int* (´*i t*)));;
IF ¬ *level-empty* (´*mem-pool-info p*) (´*i t*) *THEN*
(*t* ▶ ´*free-l* := ´*free-l*(*t* := *int* (´*i t*)))
FI;;
(*t* ▶ ´*i* := ´*i*(*t* := ´*i t* + *1*))

FI

lemma *lsize-loop-body-terminate*:
$\Gamma \vdash_I$ *Some* (*alloc-lsize-loopbody t p sz*)
sat$_p$ [*mp-alloc-lsizestm-loopinv t p sz tm* α $\cap$ $\{\!\!\{\alpha > 0\}\!\!\}$, *Mem-pool-alloc-rely t*, *Mem-pool-alloc-guar t*,
mp-alloc-lsizestm-loopinv t p sz tm (α − *1*) $\cup$ *mp-alloc-lsizestm-loopinv t p sz tm 0*]

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-6-2 t p sz tm* α])

apply(*rule subst*[**where** *s*=*mp-alloc-precond1-6-1 t p sz tm* α **and**
t=*mp-alloc-lsizestm-loopinv t p sz tm* α $\cap$ $\{\!\!\{\alpha > 0\}\!\!\}$])
using *lsizeloop-inv-cond-eq-*α*gt0*[*of t p tm sz* α] **apply** *fast*
using *mp-alloc-stm1-lm*[*of t p tm sz* α] **apply** *blast*

using *mp-alloc-stm2-lm* **apply** *simp*

done

lemma *lsizeloopbody-sat-invterm-imp-inv-post*:
$\Gamma \vdash_I$ *Some P sat*$_p$ [*pre*, *rely*, *guar*,
mp-alloc-lsizestm-loopinv t p sz tm (α − *1*) $\cup$ *mp-alloc-lsizestm-loopinv t p sz tm 0*]
$\Longrightarrow$ $\Gamma \vdash_I$ *Some P sat*$_p$ [*pre*, *rely*, *guar*,*mp-alloc-precond1-6 t p sz tm*]
apply(*rule Conseq* [*of pre pre*
rely rely guar guar mp-alloc-lsizestm-loopinv t p sz tm (α − *1*) $\cup$ *mp-alloc-lsizestm-loopinv t p sz tm 0*
mp-alloc-precond1-6 t p sz tm Some P])
apply *fast+*
done

lemma *lsizeloopbody-term-imp-prepost*:
($\forall$ α. $\Gamma \vdash_I$ *Some P sat*$_p$ [*mp-alloc-lsizestm-loopinv t p sz tm* α $\cap$ $\{\!\!\{\alpha > 0\}\!\!\}$, *rely*, *guar*,
mp-alloc-lsizestm-loopinv t p sz tm (α − *1*) $\cup$ *mp-alloc-lsizestm-loopinv t p sz tm 0*])
$\Longrightarrow$ $\Gamma \vdash_I$ *Some P sat*$_p$ [*mp-alloc-precond1-6 t p sz tm* $\cap$ *mp-alloc-lsizestm-loopcond t p*,
rely, *guar*, *mp-alloc-precond1-6 t p sz tm*]

apply(*rule subst*[**where** *s*=$\forall$ *v*. *v*$\in$*mp-alloc-precond1-6 t p sz tm* $\cap$ *mp-alloc-lsizestm-loopcond t p* $\longrightarrow$
$\Gamma \vdash_I$ *Some P sat*$_p$ [$\{v\}$, *rely*, *guar*, *mp-alloc-precond1-6 t p sz tm*] **and**
t=$\Gamma \vdash_I$ *Some P sat*$_p$ [*mp-alloc-precond1-6 t p sz tm* $\cap$ *mp-alloc-lsizestm-loopcond*

t p,
rely, *guar*, *mp-alloc-precond1-6 t p sz tm*]])
using *allpre-eq-pre*[*of mp-alloc-precond1-6 t p sz tm* ∩ *mp-alloc-lsizestm-loopcond t p*
Some P rely guar mp-alloc-precond1-6 t p sz tm]
apply *meson*

apply(*rule allI*) **apply**(*rule impI*)
apply(*subgoal-tac* $\exists\alpha.$ *v* ∈ *mp-alloc-lsizestm-loopinv t p sz tm α* ∩ {|α > *0*|})
prefer *2* **using** *lsizestm-pre-loopcond-imp-loopinv-αgt0*[*of - t p tm sz*] **apply** *meson*

apply(*erule exE*)
using *sat-pre-imp-allinpre*[*of Some P - rely guar mp-alloc-precond1-6 t p sz tm*]
lsizeloopbody-sat-invterm-imp-inv-post[*of P - rely guar t p tm sz*] **apply** *meson*
done

lemma *lsize-loop-body-satprepost*:
Γ $\vdash_I$ *Some* (*alloc-lsize-loopbody t p sz*)
sat_p [*mp-alloc-precond1-6 t p sz timeout* ∩ *mp-alloc-lsizestm-loopcond t p*,
Mem-pool-alloc-rely t, *Mem-pool-alloc-guar t*, *mp-alloc-precond1-6 t p sz timeout*]
using *lsizeloopbody-term-imp-prepost*[*of alloc-lsize-loopbody t p sz t p timeout sz Mem-pool-alloc-rely t Mem-pool-alloc-guar t*]
lsize-loop-body-terminate[*of t sz p timeout*] **apply** *fast*
done

lemma *lsize-loop-stm*:
Γ $\vdash_I$ *Some* (*WHILE ´i t* < *n-levels* (*´mem-pool-info p*) ∧ ¬ *´alloc-lsize-r t DO*
alloc-lsize-loopbody t p sz
OD) sat_p [*mp-alloc-precond1-6 t p sz timeout*, *Mem-pool-alloc-rely t*,
Mem-pool-alloc-guar t, *mp-alloc-precond1-7 t p sz timeout*]
apply(*rule While*)
using *mp-alloc-precond1-6-stb* **apply** *simp*
apply(*rule Int-greatest*) **apply**(*rule Int-greatest*) **apply**(*rule Int-greatest*) **apply**(*rule Int-greatest*)
apply(*rule Int-greatest*)
apply *force+*
using *mp-alloc-precond1-7-stb* **apply** *simp*

using *lsize-loop-body-satprepost*[*of t sz p timeout*] **apply** *fast*

apply(*simp add*:*Mem-pool-alloc-guar-def*)
done

21.6 stm3

lemma *mp-alloc-stm3-lm3-1*: *ii* < *n-levels mp* ⟹

length (*levels mp*) = *n-levels mp* $\Longrightarrow$
free-list (*levels mp* ! *ii*) = [] $\Longrightarrow$
rmhead-free-list mp ii = *mp*
apply(*simp add*:*rmhead-free-list-def*)
by (*metis Mem-pool.surjective Mem-pool.update-convs*(*5*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*2*) *list-update-id*)

lemma *mp-alloc-stm3-lm3-2*:
head-free-list mp ii = *NULL* $\Longrightarrow$
ii < *n-levels mp* $\Longrightarrow$
NULL < *buf mp* $\Longrightarrow$
$\forall$ *i*<*n-levels mp*.
$\forall$ *j*<*length* (*free-list* (*levels mp* ! *i*)).
buf mp $\leq$ *free-list* (*levels mp* ! *i*) ! *j* $\Longrightarrow$
length (*levels mp*) = *n-levels mp* $\Longrightarrow$
free-list (*levels mp* ! *ii*) $\neq$ [] $\Longrightarrow$
False
apply(*simp add*:*head-free-list-def*)
apply(*subgoal-tac hd* (*free-list* (*levels mp* ! *ii*)) $\neq$ *NULL*)
apply *simp*
using *hd-conv-nth* **by** *force*

lemma *mp-alloc-stm3-lm3*:
Va $\in$ *mp-alloc-precond1-70-2-2 t p sz timeout* $\cap$ $\{\!|$´*cur* = *Some t*$|\!\}$ $\Longrightarrow$
(*if level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) *then*
{*V*. *V* = *Va*(|*blk*:=(*blk Va*)(*t*:=*NULL*)|) $\wedge$ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}
else
{*V*. *V* = *Va*(|*blk*:=(*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)(*p*:=*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))|)
$\wedge$ $\neg$ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}) $\cap$
$-$ $\{\!|$´*blk t* $\neq$ *NULL*$|\!\}$
$\subseteq$ $\{\!|$´*id* $\in$ $\{\!|$´(*Pair Va*) $\in$ *Mem-pool-alloc-guar t*$|\!\}$ $\cap$ *mp-alloc-precond2-1 t p sz timeout*$|\!\}$
apply *clarsimp* **apply** *meson*

apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *simp*
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*Va*,*Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)

apply(*subgoal-tac* (*Va*,*Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def*

lvars-nochange1-def)

apply *clarsimp*
apply(*subgoal-tac nat* (*free-l Va t*) $\geq$ *0* $\wedge$ *nat* (*free-l Va t*) < *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *linarith*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) > *0*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply *meson*
apply(*subgoal-tac* $\forall$ *i*<*length* (*levels* (*mem-pool-info Va p*)).
$\forall$ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)).
buf (*mem-pool-info Va p*) $\leq$ (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def Let-def*)
apply *clarify* **apply** (*metis Suc-leI lessI not-le trans-le-add1*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply *metis*
apply(*subgoal-tac* $\forall$ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))).
buf (*mem-pool-info Va p*) $\leq$ *free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply(*rule conjI*)
apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def rmhead-free-list-def*) **apply** *clarsimp*
apply(*case-tac nat* (*free-l Va t*) $\neq$ *i*) **apply** *simp* **apply** *simp*
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*⦇*blk* := (*blk Va*)(*t* := *NULL*)⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *meson*
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)

apply *simp* **apply**(*subgoal-tac* $(Va, Va(|blk := (blk\ Va)(t := NULL)|)) \in$ *lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*) **apply**(*simp add*:*rmhead-free-list-def*)

apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *simp*
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $(Va, Va(|blk := (blk\ Va)(t := NULL)|)) \in$ *lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)

apply(*subgoal-tac* $(Va, Va(|blk := (blk\ Va)(t := NULL)|)) \in$ *lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarsimp*
apply(*subgoal-tac nat* (*free-l Va t*) $\geq 0 \wedge$ *nat* (*free-l Va t*) $<$ *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *linarith*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) > 0)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*) **apply** *meson*
apply(*subgoal-tac* $\forall i<$*length* (*levels* (*mem-pool-info Va p*)).
$\forall j<$*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)).
buf (*mem-pool-info Va p*) $\leq$ (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def Let-def*)
apply *clarify* **apply** (*metis Suc-leI lessI not-less trans-le-add1*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*) **apply** *metis*
apply(*subgoal-tac* $\forall j<$*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))).
buf (*mem-pool-info Va p*) $\leq$ *free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)

apply(*rule conjI*)
apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def rmhead-free-list-def*) **apply** *clarsimp*
apply(*case-tac nat* (*free-l Va t*) $\neq$ *i*) **apply** *simp* **apply** *simp*
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *meson*
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*) **apply**(*simp add*:*rmhead-free-list-def*)

apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *simp*
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)

apply(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarsimp*
apply(*subgoal-tac nat* (*free-l Va t*) $\geq$ *0* $\wedge$ *nat* (*free-l Va t*) < *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *linarith*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) > *0*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply *meson*
apply(*subgoal-tac* $\forall$ *i*<*length* (*levels* (*mem-pool-info Va p*)).
$\forall$ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)).
buf (*mem-pool-info Va p*) $\leq$ (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def Let-def*)
apply *clarify* **apply** (*metis Suc-leI lessI not-less trans-le-add1*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply *metis*
apply(*subgoal-tac* $\forall$ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))).
buf (*mem-pool-info Va p*) $\leq$ *free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)
apply(*rule conjI*)
apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def rmhead-free-list-def*) **apply** *clarsimp*
apply(*case-tac nat* (*free-l Va t*) $\neq$ *i*) **apply** *simp* **apply** *simp*
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(\|*blk* := (*blk Va*)(*t* := *NULL*)\|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *meson*
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(\|*blk* := (*blk Va*)(*t* := *NULL*)\|))$\in$*lvars-nochange1-4all*)

using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*simp add*:*rmhead-free-list-def*)

apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *simp*
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)

apply(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)

apply *clarsimp*
apply(*subgoal-tac nat* (*free-l Va t*) $\geq$ *0* $\wedge$ *nat* (*free-l Va t*) < *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *linarith*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) > *0*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*) **apply** *meson*
apply(*subgoal-tac* ∀ *i*<*length* (*levels* (*mem-pool-info Va p*)).
∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)).
buf (*mem-pool-info Va p*) ≤ (*free-list* (*levels* (*mem-pool-info Va p*) ! *i*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def Let-def*)
apply *clarify* **apply** (*metis Suc-leI lessI not-less trans-le-add1*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*) **apply** *metis*
apply(*subgoal-tac* ∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))).
buf (*mem-pool-info Va p*) ≤ *free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *j*)
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def inv-bitmap-freelist-def*)

apply(*rule conjI*)
apply(*unfold Mem-pool-alloc-guar-def*)[*1*] **apply**(*rule UnI1*) **apply** *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def rmhead-free-list-def*) **apply** *clarsimp*
apply(*case-tac nat* (*free-l Va t*) $\neq$ *i*) **apply** *simp* **apply** *simp*
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *meson*
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*)
apply(*case-tac free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*))) = [])
apply(*subgoal-tac rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) = *mem-pool-info Va p*)
apply *simp* **apply**(*subgoal-tac* (*Va*, *Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
using *mp-alloc-stm3-lm3-1*[*of nat* (*free-l Va t*) *mem-pool-info Va p*] **apply** *meson*
using *mp-alloc-stm3-lm3-2*[*of mem-pool-info Va p nat* (*free-l Va t*)] **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*)
apply(*rule conjI*) **apply**(*simp add*:*rmhead-free-list-def*) **apply**(*simp add*:*rmhead-free-list-def*)

done

lemma *mp-alloc-stm3-lm2-1*:
length (*bits* (*levels mp* ! *ii*)) =
length (*bits* ((*levels mp*) [*ii* := ((*levels mp*) [*ii* := (*levels mp* ! *ii*) (|*free-list* := *fl*|)] ! *ii*)
(|*bits* := (*bits* ((*levels mp*) [*ii* := (*levels mp* ! *ii*) (|*free-list* := *fl*|)] ! *ii*))
[*jj* := *ALLOCATING*]|)] ! *ii*))
apply(*case-tac ii* < *length* (*levels mp*))
apply *simp*
apply *auto*
done

lemma *mp-alloc-stm3-lm2-2*:
length (*bits* (*levels mp* ! *ii*)) =
length (*bits* ((*levels mp*) [*ii* := (*levels mp* ! *ii*) (|*free-list* := *fl*, *bits* := (*bits* (*levels mp* ! *ii*)) [*jj* := *ALLOCATING*]|)] ! *ii*))
apply(*case-tac ii* < *length* (*levels mp*))
apply *simp* **apply** *auto*
done

lemma *mp-alloc-stm3-body-meminfo*:
pa ≠ *p* ⟹
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
pa = (*mem-pool-info Va*) *pa*
by(*simp add*: *set-bit-def*)

lemma *mp-alloc-stm3-body-minf-buf*:
buf (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) = *buf* (*mem-pool-info Va p*)
by (*simp add*: *set-bit-def rmhead-free-list-def*)

lemma *mp-alloc-stm3-body-minf-maxsz*:
max-sz (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) = *max-sz* (*mem-pool-info Va p*)
by (*simp add*: *set-bit-def rmhead-free-list-def*)

lemma *mp-alloc-stm3-body-minf-nmax*:
n-max (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))

p) = *n-max* (*mem-pool-info Va p*)
by (*simp add*: *set-bit-def rmhead-free-list-def*)

lemma *mp-alloc-stm3-body-minf-nlvls*:
n-levels (*set-bit-allocating* ((*mem-pool-info Va*)(p := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) p
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) = *n-levels* (*mem-pool-info Va p*)
by (*simp add*: *set-bit-def rmhead-free-list-def*)

lemma *mp-alloc-stm3-body-len-lvls*:
length (*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(p := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) p
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p)) = *length* (*levels* (*mem-pool-info Va p*))
by(*simp add*: *set-bit-def rmhead-free-list-def*)

lemma *mp-alloc-stm3-body-len-bits*:
length (*bits* (*levels* (*set-bit-allocating*
((*mem-pool-info Va*)(p := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) p
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) !
ii)) = *length* (*bits* ((*levels* (*mem-pool-info Va p*))!*ii*))
apply(*simp add*: *set-bit-def rmhead-free-list-def block-num-def head-free-list-def*)
by (*smt Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*1*) *Mem-pool-lvl.update-convs*(*2*) *list-update-beyond list-updt-samelen not-less nth-list-update-eq nth-list-update-neq*)

lemma *mp-alloc-stm3-body-frlst-otherlvl*:
ii $\neq$ *nat* (*free-l Va t*) $\Longrightarrow$
free-list (*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(p := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) p
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p) ! *ii*) = *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)

by(*simp add*: *set-bit-def rmhead-free-list-def block-num-def head-free-list-def*)

lemma *mp-alloc-stm3-body-frlst-samelvl*:
$ii < length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)) \Longrightarrow ii = nat\ (free\text{-}l\ Va\ t) \Longrightarrow$
$free\text{-}list\ (levels\ (set\text{-}bit\text{-}allocating\ ((mem\text{-}pool\text{-}info\ Va)(p := rmhead\text{-}free\text{-}list\ (mem\text{-}pool\text{-}info\ Va\ p)\ (nat\ (free\text{-}l\ Va\ t))))\ p$
$(nat\ (free\text{-}l\ Va\ t))$
$(block\text{-}num\ (rmhead\text{-}free\text{-}list\ (mem\text{-}pool\text{-}info\ Va\ p)\ (nat\ (free\text{-}l\ Va\ t)))$
$(head\text{-}free\text{-}list\ (mem\text{-}pool\text{-}info\ Va\ p)\ (nat\ (free\text{-}l\ Va\ t)))$
$(ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p))\ div\ 4\ \hat{}\ nat\ (free\text{-}l\ Va\ t)))$
$p)\ !\ ii) = tl\ (free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ ii))$
by(*simp add*: *set-bit-def rmhead-free-list-def block-num-def head-free-list-def*)

lemma *mp-alloc-stm3-lm2-inv-1-1*: $(jj::nat) \neq (a - b)\ div\ c \Longrightarrow$
$(a - b)\ mod\ c = 0 \Longrightarrow$
$c \neq 0 \Longrightarrow$
$b + jj * c \neq a$ **by** *auto*

lemma *mp-alloc-stm3-lm2-inv-1-2*:
$\exists n>0.\ (a::nat) = 4 * n * 4\ \hat{}\ b \Longrightarrow$
$ii < b \Longrightarrow 0 < a\ div\ 4\ \hat{}\ ii$
by (*smt div-eq-0-iff divisors-zero less-imp-le-nat m-mod-div mod-if not-gr0 pow-mod-0 power-not-zero zero-neq-numeral*)

lemma *mp-alloc-stm3-lm2-inv-1*:
$\neg\ level\text{-}empty\ (mem\text{-}pool\text{-}info\ Va\ p)\ ii \Longrightarrow p \in mem\text{-}pools\ Va \Longrightarrow$
$inv\text{-}mempool\text{-}info\ Va \Longrightarrow$
$\forall ii<length\ (lsizes\ Va\ t).\ lsizes\ Va\ t\ !\ ii = ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p))\ div\ 4\ \hat{}\ ii \Longrightarrow$
$\forall p\in mem\text{-}pools\ Va.$
$\forall i<length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)).$
$(\forall j<length\ (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ i)).$
$(get\text{-}bit\text{-}s\ Va\ p\ i\ j = FREE) =$
$(buf\ (mem\text{-}pool\text{-}info\ Va\ p) + j * (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4\ \hat{}\ i)$
$\in set\ (free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ i)))) \wedge$
$(\forall j<length\ (free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ i)).$
$\exists n<n\text{-}max\ (mem\text{-}pool\text{-}info\ Va\ p) * 4\ \hat{}\ i.$
$free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ i)\ !\ j =$
$buf\ (mem\text{-}pool\text{-}info\ Va\ p) + n * (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4\ \hat{}\ i)) \wedge$
$distinct\ (free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ i)) \Longrightarrow$
$length\ (lsizes\ Va\ t) \leq length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)) \Longrightarrow$
$ii < length\ (lsizes\ Va\ t) \Longrightarrow$
$length\ (bits\ (levels\ (set\text{-}bit\text{-}allocating\ ((mem\text{-}pool\text{-}info\ Va)(p := rmhead\text{-}free\text{-}list\ (mem\text{-}pool\text{-}info\ Va\ p)\ ii))\ p\ ii$

((*head-free-list* (*mem-pool-info Va p*) *ii* − *buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*)) *div*
lsizes Va t ! *ii*)
p) !
ii)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)) ⟹
jj < *length* (*bits* (*levels* (*set-bit-allocating*
(λ*a*. *if a* = *p then rmhead-free-list* (*mem-pool-info Va p*) *ii else mem-pool-info Va a*) *p ii*
((*head-free-list* (*mem-pool-info Va p*) *ii* − *buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*)) *div*
lsizes Va t ! *ii*)
p) !
ii)) ⟹
nat (*free-l Va t*) = *ii* ⟹
jj ≠ (*head-free-list* (*mem-pool-info Va p*) *ii* − *buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*)) *div lsizes Va t* ! *ii* ⟹
buf (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*) ≠ *head-free-list* (*mem-pool-info Va p*) *ii*
apply(*subgoal-tac buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*) = *buf* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*rmhead-free-list-def*)
apply(*subgoal-tac lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii*)
prefer *2* **apply** *metis*
apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info Va p*)) = *max-sz* (*mem-pool-info Va p*))
prefer *2* **using** *inv-mempool-info-maxsz-align4* **apply** *blast*
apply(*subgoal-tac* (*head-free-list* (*mem-pool-info Va p*) *ii* − *buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*)) *mod lsizes Va t* ! *ii* = *0*)
prefer *2* **apply**(*simp add*:*inv-mempool-info-def head-free-list-def Let-def*)
apply(*subgoal-tac* ∃ *n*. *hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)) =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*))
prefer *2* **apply**(*simp add*:*level-empty-def*)
apply(*subgoal-tac* ∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)).
(∃ *n*<*n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *ii*. *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *j* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)))
prefer *2* **apply** (*simp add*: *linorder-not-less*)
apply (*metis* (*full-types*, *hide-lams*) *hd-conv-nth length-greater-0-conv*)
apply (*metis* (*no-types*, *hide-lams*) *add-diff-cancel-left′ mod-mult-self2-is-0*)

apply(*subgoal-tac lsizes Va t* ! *ii* ≠ *0*)
prefer *2* **apply**(*simp add*:*inv-mempool-info-def Let-def*)
apply(*subgoal-tac* ∃ *n*>*0*. *max-sz* (*mem-pool-info Va p*) = *4* ∗ *n* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *blast*

apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *simp*
using *mp-alloc-stm3-lm2-inv-1-2*[*of max-sz* (*mem-pool-info Va p*) *n-levels* (*mem-pool-info Va p*) *ii*]
apply (*metis* (*no-types*, *lifting*) *add-lessD1 le-eq-less-or-eq less-imp-add-positive*)

using *mp-alloc-stm3-lm2-inv-1-1*[*of jj head-free-list* (*mem-pool-info Va p*) *ii buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*) *lsizes Va t* ! *ii*]
apply *auto*[*1*]
done

lemma *mp-alloc-stm3-lm2-inv-2*:
$(a::nat) + jj * b \neq c \Longrightarrow \exists n.\ a + n * b = c \Longrightarrow$
$(c - a)\ div\ b \neq jj$ **by** *auto*

lemma *mp-alloc-stm3-lm2-inv-thd-waitq*:
inv-thd-waitq Va ⟹
inv-thd-waitq
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
apply(*simp add*:*inv-thd-waitq-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def*
head-free-list-def set-bit-def block-num-def)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def*
head-free-list-def set-bit-def block-num-def) **apply** *metis*
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def*
head-free-list-def set-bit-def block-num-def)
apply(*simp add*: *rmhead-free-list-def*
head-free-list-def set-bit-def block-num-def) **apply** *metis*
done

lemma *mp-alloc-stm3-lm2-inv-aux-vars-1*:
¬ (*pool n* = *p* ∧ *level n* = *nat* (*free-l Va t*) ∧ *block n* =
block-num (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))) ⟹
get-bit-s (*Va*⦇*mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈)
(*pool n*) (*level n*) (*block n*) = *get-bit-s Va* (*pool n*) (*level n*) (*block n*)

apply(*rule subst*[**where** *t*= *get-bit-s*
(*Va*⦇*mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈)
(*pool n*) (*level n*) (*block n*) **and** *s*=*get-bit-s*
(*Va*⦇*mem-pool-info* :=
set-bit-allocating (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈)
(*pool n*) (*level n*) (*block n*)])
apply(*simp add*:*rmhead-free-list-def set-bit-def*)
apply (*smt Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*2*) *linorder-not-less list-update-beyond nth-list-update-eq nth-list-update-neq*)
apply(*simp add*:*rmhead-free-list-def set-bit-def*)
apply (*smt Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*2*) *linorder-not-less list-update-beyond nth-list-update-eq nth-list-update-neq*)
done

lemma *mp-alloc-stm3-lm2-inv-aux-vars-2*:
inv-mempool-info Va ⟹
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
p ∈ *mem-pools Va* ⟹
pool n = *p* ∧
level n = *nat* (*free-l Va t*) ∧
block n =
block-num (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list*

(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) $\Longrightarrow$
get-bit-s (*Va*(|*mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*)))|))
(*pool n*) (*level n*) (*block n*) =
get-bit
(*set-bit-allocating* (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*)))
apply(*rule subst*[**where** *t*=*p* **and** *s*=*pool n*]) **apply** *simp*
apply(*rule subst*[**where** *t*=*nat* (*free-l Va t*) **and** *s*=*level n*]) **apply** *simp*
apply(*rule subst*[**where** *t*=(*block-num* (*mem-pool-info Va* (*pool n*)) (*free-list* (*levels* (*mem-pool-info Va* (*pool n*)) ! *level n*) ! *NULL*)
(*max-sz* (*mem-pool-info Va* (*pool n*)) *div 4* ^ *level n*)) **and** *s*=*block n*])
apply(*simp add*:*level-empty-def block-num-def rmhead-free-list-def head-free-list-def*)
apply (*metis hd-conv-nth inv-mempool-info-maxsz-align4*)
apply(*rule subst*[**where** *t*=*get-bit-s*
(*Va*(|*mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*pool n* := *rmhead-free-list* (*mem-pool-info Va* (*pool n*)) (*level n*))) (*pool n*) (*level n*)
(*block n*)|))
(*pool n*) (*level n*) (*block n*) **and**
s=*get-bit* (*set-bit-allocating* ((*mem-pool-info Va*)(*pool n* := *rmhead-free-list* (*mem-pool-info Va* (*pool n*)) (*level n*))) (*pool n*) (*level n*)
(*block n*)) (*pool n*) (*level n*) (*block n*)]) **apply** *auto*[*1*]
apply(*simp add*:*rmhead-free-list-def set-bit-def*)
apply(*case-tac level n* $\geq$ *length* (*levels* (*mem-pool-info Va* (*pool n*))))
apply *auto*
done

lemma *mp-alloc-stm3-lm2-inv-aux-vars*:
$\neg$ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) $\Longrightarrow$
allocating-node Va t = *None* $\Longrightarrow$
freeing-node Va t = *None* $\Longrightarrow$
inv-cur Va $\wedge$ *inv-thd-waitq Va* $\wedge$ *inv-mempool-info Va* $\wedge$ *inv-bitmap-freelist Va* $\wedge$ *inv-bitmap Va* $\wedge$ *inv-aux-vars Va* $\Longrightarrow$
p $\in$ *mem-pools Va* $\Longrightarrow$
ETIMEOUT $\leq$ *timeout* $\Longrightarrow$
timeout = *ETIMEOUT* $\longrightarrow$ *tmout Va t* = *ETIMEOUT* $\Longrightarrow$

$\neg$ *rf Va t* $\Longrightarrow$
$\forall$ *ii<length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* $\Longrightarrow$
length (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) $\Longrightarrow$
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) $\Longrightarrow$
free-l Va t $\leq$ *alloc-l Va t* $\Longrightarrow$
$\neg$ *free-l Va t* < *OK* $\Longrightarrow$
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* $\wedge$ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) $\vee$
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* $\wedge$ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* $\Longrightarrow$
length (*lsizes Va t*) $\leq$ *length* (*levels* (*mem-pool-info Va p*)) $\Longrightarrow$
nat (*free-l Va t*) < *length* (*lsizes Va t*) $\Longrightarrow$
inv-aux-vars
(*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* $\mapsto$
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|))|))
apply(*unfold inv-aux-vars-def*)
apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac freeing-node*
(*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))

(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|)))|))
= *freeing-node Va*)
prefer *2* **apply** *simp*
apply(*subgoal-tac get-bit-s Va* (*pool n*) (*level n*) (*block n*) = *FREEING*)
prefer *2* **apply** *auto*[*1*]
apply(*case-tac* (*pool n*) = *p* ∧ (*level n*) = *nat* (*free-l Va t*)
∧ (*block n*) = (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))))
apply(*subgoal-tac get-bit-s Va p* (*nat* (*free-l Va t*)) (*block-num* (*mem-pool-info Va p*)
((*free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*)))) ! *0*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ (*nat* (*free-l Va t*))))
= *FREE*)
prefer *2* **apply**(*simp add*:*level-empty-def*) **using** *inv-bitmap-freelist-fl-FREE*
apply *auto*[*1*]
apply(*subgoal-tac block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))
= (*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))))
prefer *2* **apply**(*simp add*:*rmhead-free-list-def head-free-list-def block-num-def level-empty-def*)
apply (*metis hd-conv-nth inv-mempool-info-maxsz-align4*)

apply *simp*

apply(*subgoal-tac get-bit-s*
  (*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
    mem-pool-info :=
      set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
      (*nat* (*free-l Va t*))
      (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
        (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
        (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ˆ *nat* (*free-l Va t*)))),
    allocating-node := *allocating-node Va*(*t* ↦
      (|*pool* = *p*, *level* = *nat* (*free-l Va t*),
        block = *block-num*
          (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
            p (*nat* (*free-l Va t*))
            (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
            (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
              (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ˆ *nat* (*free-l Va t*)))
            p)
          (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
            (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ˆ *nat* (*free-l Va t*)),
        data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|))|))
  (*pool n*) (*level n*) (*block n*) = *get-bit-s Va* (*pool n*) (*level n*) (*block n*))
apply *simp*
apply(*subgoal-tac get-bit-s*
  (*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
    mem-pool-info :=
      set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
      (*nat* (*free-l Va t*))
      (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
        (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
        (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ˆ *nat* (*free-l Va t*)))),
    allocating-node := *allocating-node Va*(*t* ↦
      (|*pool* = *p*, *level* = *nat* (*free-l Va t*),
        block = *block-num*
          (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*

(*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
(*pool n*) (*level n*) (*block n*)
= *get-bit-s* (*Va*⦇ *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈) (*pool n*) (*level n*) (*block n*))
prefer *2* **apply** *force*
apply(*frule mp-alloc-stm3-lm2-inv-aux-vars-1*) **apply** *simp*

apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac get-bit-s*
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))

(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
(*pool n*) (*level n*) (*block n*)
= *get-bit-s* (*Va*⦇ *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))⦈) (*pool n*) (*level n*) (*block n*))
prefer *2* **apply** *force*
apply(*case-tac* (*pool n*) = *p* ∧ (*level n*) = *nat* (*free-l Va t*)
∧ (*block n*) = (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))))
apply(*subgoal-tac get-bit-s* (*Va*⦇ *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))⦈) (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
apply *simp*
apply(*subgoal-tac get-bit* (*set-bit-allocating* (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))) = *ALLOCATING*)
prefer *2*
apply(*rule set-bit-get-bit-eq*[*of nat* (*free-l Va t*) *mem-pool-info Va p*
block-num (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)

(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*)) *set-bit-allocating* (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*)))])
apply *simp* **apply**(*simp add*:*level-empty-def*) **using** *inv-bitmap-freelist-fl-bnum-in*[*of Va p nat* (*free-l Va t*) *0*]
apply (*meson le-trans length-greater-0-conv linorder-not-less*) **apply** *simp*

apply(*subgoal-tac get-bit-s* (*Va*(| *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))|)) (*pool n*) (*level n*) (*block n*)
= *get-bit* (*set-bit-allocating* (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))))

apply *simp*
apply(*rule subst*[**where** *t*=*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))
and *s*=*block-num* (*mem-pool-info Va p*) (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)) ! *NULL*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))])
apply(*simp add*:*level-empty-def block-num-def rmhead-free-list-def head-free-list-def*)
apply (*metis hd-conv-nth inv-mempool-info-maxsz-align4*)
using *mp-alloc-stm3-lm2-inv-aux-vars-2*[*of Va p t*] **apply** *blast*

apply(*subgoal-tac get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *FREEING*)
prefer *2*
apply(*subgoal-tac get-bit-s*
(*Va*(|*mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*

(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈)
(*pool n*) (*level n*) (*block n*) = *get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*))
prefer *2* **using** *mp-alloc-stm3-lm2-inv-aux-vars-1*[*of - p Va t*] **apply** *blast*
apply *simp*
apply(*subgoal-tac mem-block-addr-valid Va n*)
prefer *2* **apply**(*simp add*:*mem-block-addr-valid-def*)
apply (*metis mp-alloc-stm3-body-meminfo mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-maxsz*)
apply(*subgoal-tac* ∃ *t*. *freeing-node Va t* = *Some n*) **prefer** *2* **apply** *metis*
apply(*subgoal-tac* ∀ *ta*. *freeing-node Va ta* = *freeing-node*
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈) *ta*)
prefer *2* **apply** *force*
apply *metis*

apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac get-bit-s*
(*Va*(| *blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|))|))
(*pool n*) (*level n*) (*block n*)
= *get-bit-s* (*Va*(| *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))|)) (*pool n*) (*level n*) (*block n*))
prefer *2* **apply** *force*
apply(*subgoal-tac get-bit-s* (*Va*(| *mem-pool-info* :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))|)) (*pool n*) (*level n*) (*block n*)
= *get-bit-s* (*Va*(| *mem-pool-info* := *set-bit-allocating* (*mem-pool-info*

Va) *p*
(*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈) (*pool n*) (*level n*) (*block n*))
prefer *2* **apply**(*simp add*:*rmhead-free-list-def set-bit-def block-num-def*)
apply (*smt Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective linorder-not-less list-update-beyond nth-list-update-eq nth-list-update-neq*)
apply(*subgoal-tac get-bit-s* (*Va*⦇ *mem-pool-info* := *set-bit-allocating* (*mem-pool-info Va*) *p*
(*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))⦈) (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
apply *simp*
apply(*case-tac t* = *ta*)

apply(*subgoal-tac* (*pool n*) = *p* ∧ (*level n*) = *nat* (*free-l Va t*)
∧ (*block n*) = *block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply**(*rule conjI*) **apply** *auto*[*1*] **apply**(*rule conjI*) **apply** *auto*[*1*]
apply(*subgoal-tac* (*block n*) = *block-num* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac block-num* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))

p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))
= *block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*:*level-empty-def block-num-def set-bit-def rmhead-free-list-def*)
apply *simp*
apply(*subgoal-tac nat* (*free-l Va t*) < *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply** *simp*
apply(*subgoal-tac block-num* (*mem-pool-info Va p*) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))
< *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))))
prefer *2* **apply**(*rule subst*[**where** *t*=*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) **and** *s*=*max-sz* (*mem-pool-info Va p*)])
apply (*metis inv-mempool-info-maxsz-align4*)
apply(*frule inv-bitmap-freelist-fl-bnum-in*[*of Va p nat* (*free-l Va t*) *0*])
apply *simp* **apply** *simp* **apply** *simp* **apply**(*simp add*:*level-empty-def*)
apply(*simp add*:*level-empty-def head-free-list-def*) **apply** (*metis hd-conv-nth*)

using *set-bit-get-bit-eq2*[*of nat* (*free-l Va t*) *Va p block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) *ALLOCATING*] **apply** *metis*

apply(*subgoal-tac allocating-node*
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*

(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
p (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
ta = *allocating-node Va ta*) **prefer** *2* **apply** *force*
apply(*subgoal-tac get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
prefer *2* **apply** *metis*
apply(*subgoal-tac block-num* (*mem-pool-info Va p*)
((*free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*)))) ! *0*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ˆ (*nat* (*free-l Va t*)))
= *block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*:*block-num-def rmhead-free-list-def head-free-list-def*)
apply (*simp add*: *hd-conv-nth inv-mempool-info-maxsz-align4 level-empty-def*)
apply(*case-tac* (*pool n*) = *p* ∧ (*level n*) = *nat* (*free-l Va t*)
∧ (*block n*) = (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))))
apply(*subgoal-tac get-bit-s Va p* (*nat* (*free-l Va t*)) (*block-num* (*mem-pool-info Va p*)
((*free-list* ((*levels* (*mem-pool-info Va p*)) ! (*nat* (*free-l Va t*)))) ! *0*)
(*max-sz* (*mem-pool-info Va p*) *div 4* ˆ (*nat* (*free-l Va t*)))) = *FREE*)
prefer *2* **apply**(*simp add*:*level-empty-def*) **using** *inv-bitmap-freelist-fl-FREE*[*of Va p nat* (*free-l Va t*) *0*]
apply *auto*[*1*]

apply *simp*

apply(*subgoal-tac get-bit-s* (*Va*(|*mem-pool-info* :=
set-bit-allocating (*mem-pool-info Va*) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))|))
(*pool n*) (*level n*) (*block n*) = *get-bit-s Va* (*pool n*) (*level n*) (*block n*))
prefer *2* **apply** (*metis set-bit-get-bit-neq2*)
apply(*rule subst*[**where** *t*=*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) **and** *s*=*max-sz* (*mem-pool-info Va p*)])
apply (*metis inv-mempool-info-maxsz-align4*)
apply (*simp add*: *hd-conv-nth level-empty-def*)
apply (*smt nat-less-iff nth-equalityI set-bit-get-bit-eq set-bit-get-bit-neq set-bit-prev-len zle-int*)

apply(*rule conjI*)

apply *clarify*
apply(*case-tac* (*pool n*) = *p* ∧ (*level n*) = *nat* (*free-l Va t*)
∧ (*block n*) = (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))))
apply(*subgoal-tac allocating-node*
(*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)

(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈) *t* =
Some n)
prefer *2* **apply**(*rule subst*[**where** *t*=*allocating-node*
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈) *t* **and** *s*=*Some* ⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)

(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat*
(*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va*
t))⦈])
apply *force*
apply(*rule subst*[**where** *t*=*block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info*
Va p) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*)
(*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va*
t)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^
nat (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat*
(*free-l Va t*))
and *s*=*block-num* (*rmhead-free-list* (*mem-pool-info Va p*)
(*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l*
Va t))])
apply(*simp add*: *set-bit-def rmhead-free-list-def block-num-def*)
apply(*simp add*:*mem-block-addr-valid-def*)
apply(*subgoal-tac buf* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*
(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p) +
block-num (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) ∗
(*max-sz* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*
(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va*
t)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va*
t)))
p) *div*
4 ^ *nat* (*free-l Va t*)) = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va*
t)))

apply *auto*[*1*]
apply(*rule subst*[**where** *t*=*buf* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p) **and** *s*=*buf* (*mem-pool-info Va p*)])
apply(*simp add*:*set-bit-def block-num-def rmhead-free-list-def*)
apply(*rule subst*[**where** *t*=*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
and *s*=*block-num* (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))])
apply(*simp add*:*set-bit-def block-num-def rmhead-free-list-def*)
apply(*rule subst*[**where** *t*=*max-sz* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p) **and** *s*=*max-sz* (*mem-pool-info Va p*)])
apply(*simp add*:*set-bit-def block-num-def rmhead-free-list-def*)
apply(*rule subst*[**where** *t*=*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) **and** *s*=*max-sz* (*mem-pool-info Va p*)])
apply (*metis inv-mempool-info-maxsz-align4*)

apply(*rule ref-byblkn-self*[*of Va p head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*free-l Va t*))])
apply(*simp add*:*level-empty-def head-free-list-def*)
using *inv-buf-le-fl*[*of Va p nat* (*free-l Va t*) *0*]
apply (*smt hd-conv-nth length-greater-0-conv nat-less-iff zle-int*)
apply(*simp add*:*level-empty-def head-free-list-def*)
using *inv-fl-mod-sz0*[*of Va p nat* (*free-l Va t*) *0*]
apply (*smt hd-conv-nth le-eq-less-or-eq le-trans length-greater-0-conv nat-eq-iff nat-less-iff*)
apply *auto*[*1*]

apply(*subgoal-tac get-bit* (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
prefer *2*
apply(*subgoal-tac get-bit-s*

(Va(|mem-pool-info :=
set-bit-allocating ((mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p
(nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))|))
(pool n) (level n) (block n) = get-bit (mem-pool-info Va) (pool n) (level n) (block n))
prefer *2* **using** *mp-alloc-stm3-lm2-inv-aux-vars-1[of - p Va t]* **apply** *blast*
apply *force*

apply(*subgoal-tac* $\exists$ *ta. ta* $\neq$ *t* $\wedge$ *allocating-node Va ta = Some n*)
prefer *2* **apply**(*subgoal-tac mem-block-addr-valid Va n*) **apply** *metis*
apply(*simp add:mem-block-addr-valid-def*)
apply (*metis mp-alloc-stm3-body-meminfo mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-maxsz*)
apply *auto[1]*

apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac* $\forall$ *t. freeing-node*
(Va(|blk := (blk Va)(t := head-free-list (mem-pool-info Va p) (nat (free-l Va t))),
mem-pool-info :=
set-bit-allocating ((mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p
(nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))),
allocating-node := allocating-node Va(t $\mapsto$
(|pool = p, level = nat (free-l Va t),
block = block-num
(set-bit-allocating
((mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p
(nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))
p)
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)),
data = head-free-list (mem-pool-info Va p) (nat (free-l Va t))|))|))

t = *freeing-node Va t*)
prefer *2* **apply** *force*
apply *auto*[*1*]

apply(*rule conjI*)

apply *clarify*
apply(*case-tac t* = *t1*)
apply(*subgoal-tac get-bit-s Va* (*pool n1*) (*level n1*) (*block n1*) = *FREE*)
prefer *2*
apply(*subgoal-tac pool n1* = *p* ∧ *level n1* = *nat* (*free-l Va t*) ∧ *block n1* = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)) = *block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def block-num-def*)
apply(*subgoal-tac block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l

Va t)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4*
ˆ *nat* (*free-l Va t*))) =
block-num (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat*
(*free-l Va t*))))
prefer *2* **apply**(*simp add:set-bit-def block-num-def rmhead-free-list-def*)
apply(*simp add:level-empty-def head-free-list-def*)
using *inv-bitmap-freelist-fl-FREE*[*of Va p nat* (*free-l Va t*) *0*]
apply (*smt hd-conv-nth inv-mempool-info-maxsz-align4 le-trans length-greater-0-conv linorder-not-less*)
apply(*subgoal-tac get-bit-s Va* (*pool n2*) (*level n2*) (*block n2*) = *ALLOCATING*)
prefer *2* **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*case-tac t* = *t2*)
apply(*subgoal-tac get-bit-s Va* (*pool n2*) (*level n2*) (*block n2*) = *FREE*)
prefer *2*
apply(*subgoal-tac pool n2* = *p* ∧ *level n2* = *nat* (*free-l Va t*) ∧ *block n2* = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat*

(*free-l Va t*)) = *block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def block-num-def*)
apply(*subgoal-tac block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))) =
block-num (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))))
prefer *2* **apply**(*simp add*:*set-bit-def block-num-def rmhead-free-list-def*)
apply(*unfold level-empty-def head-free-list-def*)[*1*]
using *inv-bitmap-freelist-fl-FREE*[*of Va p nat* (*free-l Va t*) *0*]
apply (*smt hd-conv-nth inv-mempool-info-maxsz-align4 le-trans length-greater-0-conv linorder-not-less*)
apply(*subgoal-tac get-bit-s Va* (*pool n1*) (*level n1*) (*block n1*) = *ALLOCATING*)
prefer *2* **apply**(*subgoal-tac allocating-node Va t1* = *Some n1*) **prefer** *2* **apply** *auto*[*1*]
apply *blast*
apply *auto*[*1*]

apply(*subgoal-tac allocating-node Va t1* = *Some n1*)
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac allocating-node Va t2* = *Some n2*)
prefer *2* **apply** *auto*[*1*]
apply *auto*[*1*]

apply *clarify*
apply(*case-tac t* = *t1*)
apply(*subgoal-tac get-bit-s Va* (*pool n1*) (*level n1*) (*block n1*) = *FREE*)
prefer *2*
apply(*subgoal-tac pool n1* = *p* ∧ *level n1* = *nat* (*free-l Va t*) ∧ *block n1* = *block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat*

(*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac block-num*
(*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) = *block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def block-num-def*)
apply(*subgoal-tac block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))) =
block-num (*mem-pool-info Va p*)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
((*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))))
prefer *2* **apply**(*simp add*:*set-bit-def block-num-def rmhead-free-list-def*)
apply(*simp add*:*level-empty-def head-free-list-def*)
using *inv-bitmap-freelist-fl-FREE*[*of Va p nat* (*free-l Va t*) *0*]
apply (*smt hd-conv-nth inv-mempool-info-maxsz-align4 le-trans length-greater-0-conv linorder-not-less*)
apply(*subgoal-tac get-bit-s Va* (*pool n2*) (*level n2*) (*block n2*) = *FREEING*)
prefer *2* **apply** *auto*[*1*]
apply *auto*[*1*]

apply(*subgoal-tac allocating-node Va t1* = *Some n1*)
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac allocating-node Va t2* = *Some n2*)
prefer *2* **apply** *auto*[*1*]

apply *auto*[*1*]
done

lemma *mp-alloc-stm3-lm2-inv-bitmap0*:
inv-mempool-info Va ∧ *inv-bitmap0 Va* ⟹
p ∈ *mem-pools Va* ⟹
inv-bitmap0
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
apply(*simp add*:*set-bit-def*)
apply(*rule subst*[**where** *s*=*inv-bitmap0*
(*Va*⦇*mem-pool-info* := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))
⦇*levels* := (*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
[*nat* (*free-l Va t*) :=
(*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) ! *nat* (*free-l Va t*))
⦇*bits* := (*bits* (*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) !
nat (*free-l Va t*)))
[*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) :=
ALLOCATING]⦈]⦈)⦈)])
apply(*simp add*:*inv-bitmap0-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) > *0*) **prefer** *2*
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*) **apply** *fastforce*

apply(*subgoal-tac* $\forall$ *i*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *0*)).
(*bits* (*levels* (*mem-pool-info Va p*) ! *0*)) ! *i* $\neq$ *NOEXIST*)
prefer *2* **apply**(*simp add*:*inv-def inv-bitmap0-def*) **apply** *metis*

apply(*case-tac nat* (*free-l Va t*) = *0*)
apply(*simp add*:*inv-bitmap0-def Let-def rmhead-free-list-def block-num-def*)
apply *clarsimp*
apply(*case-tac i* = (*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)))
apply(*subgoal-tac bits* ((*levels* (*mem-pool-info Va p*))
[*NULL* := ((*levels* (*mem-pool-info Va p*))
[*NULL* := (*levels* (*mem-pool-info Va p*) ! *NULL*)
⦇*free-list* := *tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *NULL*))⦈] !
NULL)
⦇*bits* := (*bits* ((*levels* (*mem-pool-info Va p*))
[*NULL* := (*levels* (*mem-pool-info Va p*) ! *NULL*)
⦇*free-list* := *tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *NULL*))⦈] !
NULL))
[(*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) :=
ALLOCATING]⦈] ! *NULL*) ! *i* = *ALLOCATING*) **prefer** *2*
apply(*rule subst*[**where** *s*=(*bits* (*levels* (*mem-pool-info Va p*) ! *0*))
[(*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) := *ALLOCATING*]])
apply *fastforce*
apply *simp*
apply *force*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *NULL*))
[(*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) := *ALLOCATING*] ! *i* $\neq$ *NOEXIST*) **prefer** *2*
apply *force*
apply *simp*

apply(*simp add*:*inv-bitmap0-def Let-def rmhead-free-list-def block-num-def*)
done

lemma *mp-alloc-stm3-lm2-inv-bitmapn*:
inv-mempool-info Va $\wedge$ *inv-bitmapn Va* $\Longrightarrow$

$p \in$ *mem-pools Va* $\Longrightarrow$
inv-bitmapn
(*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* $\mapsto$
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|)|))
apply(*simp add*:*set-bit-def*)
apply(*rule subst*[**where** *s*=*inv-bitmapn*
(*Va*(|*mem-pool-info* := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))
(|*levels* := (*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))))
[*nat* (*free-l Va t*) :=
(*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) ! *nat* (*free-l Va t*))
(|*bits* := (*bits* (*levels* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) !
nat (*free-l Va t*)))
[*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*)) :=
ALLOCATING]|)]|))|))])
apply(*simp add*:*inv-bitmapn-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Va p*)) > *0*) **prefer** *2*
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*) **apply** *fastforce*

apply(*subgoal-tac* $\forall$ *i*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! (*length* (*levels* (*mem-pool-info Va p*)) − *Suc 0*))).
(*bits* (*levels* (*mem-pool-info Va p*) ! (*length* (*levels* (*mem-pool-info Va p*)) − *Suc 0*))) ! *i* $\neq$ *DIVIDED*)
prefer *2* **apply**(*simp add*:*inv-def inv-bitmapn-def*) **apply** *metis*

apply(*case-tac nat* (*free-l Va t*) = *length* (*levels* (*mem-pool-info Va p*)) − *Suc 0*)
apply(*simp add*:*inv-bitmapn-def Let-def rmhead-free-list-def block-num-def*)
apply *clarsimp*
apply(*case-tac i* = (*head-free-list* (*mem-pool-info Va p*) (*length* (*levels* (*mem-pool-info Va p*)) − *Suc NULL*) −
buf (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ (*length* (*levels* (*mem-pool-info Va p*)) − *Suc NULL*)))
apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! (*length* (*levels* (*mem-pool-info Va p*)) − *Suc NULL*)))
[(*head-free-list* (*mem-pool-info Va p*) (*length* (*levels* (*mem-pool-info Va p*)) − *Suc NULL*) −
buf (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ (*length* (*levels* (*mem-pool-info Va p*)) − *Suc NULL*)) :=
ALLOCATING] ! *i* ≠ *DIVIDED*) **prefer** *2*
apply(*rule subst*[**where** *s*=(*bits* (*levels* (*mem-pool-info Va p*) ! *0*))
[(*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) := *ALLOCATING*]])
apply *fastforce*
apply *simp*
apply *force*

apply(*subgoal-tac* (*bits* (*levels* (*mem-pool-info Va p*) ! *NULL*))
[(*head-free-list* (*mem-pool-info Va p*) *NULL* − *buf* (*mem-pool-info Va p*)) *div*
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) := *ALLOCATING*] ! *i* ≠ *DIVIDED*) **prefer** *2*
apply *force*
apply *simp*

apply(*simp add*:*inv-bitmapn-def Let-def rmhead-free-list-def block-num-def*)
done

lemma *mp-alloc-stm3-lm2-inv-bitmap-not4free*:
inv-mempool-info Va ∧ *inv-bitmap-not4free Va* ⟹
p ∈ *mem-pools Va* ⟹
inv-bitmap-not4free
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))

(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))),
allocating-node := allocating-node Va(t ↦
⦇pool = p, level = nat (free-l Va t),
block = block-num
(set-bit-allocating ((mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p (nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t))) (head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))
p)
(head-free-list (mem-pool-info Va p) (nat (free-l Va t))) (ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)),
data = head-free-list (mem-pool-info Va p) (nat (free-l Va t))⦈)⦈)
apply(*rule subst*[**where** *s=inv-bitmap-not4free (Va⦇mem-pool-info :=*
set-bit-allocating ((mem-pool-info Va)(p :=
rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p (nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t))) (head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))⦈)])
apply(*simp add: inv-bitmap-not4free-def Let-def partner-bits-def set-bit-def rmhead-free-list-def block-num-def*)

apply(*simp add: inv-bitmap-not4free-def Let-def partner-bits-def set-bit-def rmhead-free-list-def block-num-def*)
apply *clarsimp*
apply(*case-tac nat (free-l Va t) = i*) **prefer** *2* **apply** *auto*[*1*]

apply(*subgoal-tac bits ((levels (mem-pool-info Va p))*
[nat (free-l Va t) :=
((levels (mem-pool-info Va p))
[nat (free-l Va t) := (levels (mem-pool-info Va p) ! nat (free-l Va t))
⦇free-list := tl (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t)))⦈] !
nat (free-l Va t))
⦇bits := (bits ((levels (mem-pool-info Va p))
[nat (free-l Va t) := (levels (mem-pool-info Va p) ! nat (free-l Va t))
⦇free-list := tl (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t)))⦈] !
nat (free-l Va t)))
[(head-free-list (mem-pool-info Va p) (nat (free-l Va t)) − buf (mem-pool-info Va p)) div
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)) :=

ALLOCATING]⦈] !
i) = (*bits* (*levels* (*mem-pool-info Va p*) ! *i*)) [(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) − *buf* (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) :=
ALLOCATING]) **prefer** *2* **apply** *simp*
apply *simp*
apply(*case-tac* (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) − *buf* (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) = *j div 4* ∗ *4*)
apply *auto*[*1*]
apply(*case-tac* (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) − *buf* (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) = *Suc* (*j div 4* ∗ *4*))
apply(*subgoal-tac Suc* (*j div 4* ∗ *4*) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))) **prefer** *2*
apply (*metis list-update-beyond not-less*)
apply *auto*[*1*]
apply(*case-tac* (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) − *buf* (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) = *j div 4* ∗ *4* + *2*)
apply(*subgoal-tac j div 4* ∗ *4* + *2* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))) **prefer** *2*
apply (*metis list-update-beyond not-less*)
apply *auto*[*1*]
apply(*case-tac* (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) − *buf* (*mem-pool-info Va p*)) *div*
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *nat* (*free-l Va t*)) = *j div 4* ∗ *4* + *3*)
apply(*subgoal-tac j div 4* ∗ *4* + *3* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))) **prefer** *2*
apply (*metis list-update-beyond not-less*)
apply *auto*[*1*]

apply *simp*
done

lemma *mp-alloc-stm3-lm2-inv-mempool-info*:
inv-mempool-info Va ∧
p ∈ *mem-pools Va* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
¬ *free-l Va t* < *OK* ⟹
nat (*free-l Va t*) < *length* (*lsizes Va t*) ⟹
inv-mempool-info

(*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|)))|))
apply(*simp add*:*inv-mempool-info-def*)
apply(*simp add*: *rmhead-free-list-def*
head-free-list-def set-bit-def block-num-def)
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply *clarsimp* **apply**(*simp add*:*Let-def*)
apply(*case-tac nat* (*free-l Va t*) = *i*)
apply(*subgoal-tac length* (*bits* (*levels* (*mem-pool-info Va p*) ! (*nat* (*free-l Va t*))))
= *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ (*nat* (*free-l Va t*)))
prefer *2* **apply** *metis*
using *mp-alloc-stm3-lm2-2*[**where** *ii*=*nat* (*free-l Va t*) **and** *mp*=*mem-pool-info Va p* **and**
fl=*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))) **and**
jj=(*hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))) − *buf* (*mem-pool-info Va p*)) *div*
lsizes Va t ! *nat* (*free-l Va t*)] **apply** *metis*
apply *simp*
done

lemma *mp-alloc-stm3-lm2-inv-bitmap-freelist*:
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
inv-bitmap-freelist Va ∧ *inv-mempool-info Va* ⟹
p ∈ *mem-pools Va* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹

alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
free-l Va t ≤ *alloc-l Va t* ⟹
¬ *free-l Va t* < *OK* ⟹
length (*lsizes Va t*) ≤ *length* (*levels* (*mem-pool-info Va p*)) ⟹
nat (*free-l Va t*) < *length* (*lsizes Va t*) ⟹
inv-bitmap-freelist
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
apply(*simp add*:*inv-bitmap-freelist-def*)
apply *clarify*
apply(*case-tac pa* ≠ *p*) **apply**(*simp add*:*Let-def*)
using *mp-alloc-stm3-body-meminfo* **apply** *smt*
apply(*simp add*:*Let-def*)
apply(*rule subst*[**where** *t*=*length* (*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p)) **and** *s*=*length* (*levels* (*mem-pool-info Va p*))])
using *mp-alloc-stm3-body-len-lvls* **apply** *metis*
apply(*rule subst*[**where** *t*=*buf* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) **and** *s*=*buf* (*mem-pool-info Va p*)])
using *mp-alloc-stm3-body-minf-buf* **apply** *metis*

apply(*rule subst*[**where** *t=n-max* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) **and** *s=n-max* (*mem-pool-info Va p*)])
using *mp-alloc-stm3-body-minf-nmax* **apply** *metis*
apply(*rule subst*[**where** *t=max-sz* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) **and** *s=max-sz* (*mem-pool-info Va p*)])
using *mp-alloc-stm3-body-minf-maxsz* **apply** *metis*

apply *clarify* **apply**(*rename-tac pa ii*)
apply(*subgoal-tac length* (*bits* (*levels* (*set-bit-allocating*
((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p) ! *ii*))=*length* (*bits* ((*levels* (*mem-pool-info Va p*))!*ii*)))
prefer *2* **using** *mp-alloc-stm3-body-len-bits* **apply** *metis*

apply(*rule conjI*)
apply *clarify* **apply**(*rule iffI*) **apply**(*rename-tac pa ii jj*)

apply(*case-tac nat* (*free-l Va t*) = *ii*)

apply(*case-tac jj* = (*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*))))

apply(*subgoal-tac get-bit* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*lsizes Va t* ! *nat* (*free-l Va t*))))
p ii jj = *ALLOCATING*)
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def*)
apply (*metis BlockState.distinct*(*17*))

apply(*subgoal-tac get-bit* (*mem-pool-info Va*) *p ii jj* = *FREE*)

prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def*)
apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** (*metis mp-alloc-stm3-body-len-lvls*)
apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
≠ *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
prefer *2* **apply**(*simp add*:*block-num-def*) **using** *mp-alloc-stm3-lm2-inv-1*
apply *simp*
apply(*simp add*: *set-bit-def rmhead-free-list-def head-free-list-def*)
using *list-nhd-in-tl-set* **apply** *metis*

apply(*subgoal-tac get-bit* (*mem-pool-info Va*) *p ii jj* = *FREE*)
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def*)
apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** (*metis mp-alloc-stm3-body-len-lvls*)
apply(*simp add*: *set-bit-def rmhead-free-list-def head-free-list-def*)

apply(*rename-tac pa ii jj*)
apply(*subgoal-tac length* (*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p)) = *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **using** *mp-alloc-stm3-body-len-lvls* **apply** *metis*
apply(*case-tac nat* (*free-l Va t*) = *ii*)

apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))))
prefer *2* **using** *mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-maxsz mp-alloc-stm3-body-frlst-samelvl* **apply** *metis*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply**(*metis list.set-sel*(*2*) *tl-Nil*)
apply(*subgoal-tac get-bit* (*mem-pool-info Va*) *p ii jj* = *FREE*)
prefer *2* **apply** *metis*
apply(*subgoal-tac block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))

(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)) ≠ *jj*)
prefer *2*
apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
≠ *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
prefer *2* **apply**(*subgoal-tac distinct* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** *metis*
apply(*simp add*:*head-free-list-def*)
using *dist-hd-nin-tl* **apply** (*metis* (*mono-tags*, *hide-lams*) *le-eq-less-or-eq le-trans linorder-not-less*)
apply(*simp add*:*block-num-def*)
apply(*subgoal-tac buf* (*rmhead-free-list* (*mem-pool-info Va p*) *ii*) = *buf* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*:*rmhead-free-list-def*)
apply(*subgoal-tac* ∃ *n*. *head-free-list* (*mem-pool-info Va p*) *ii* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*))
prefer *2* **apply**(*simp add*:*head-free-list-def level-empty-def*)
apply (*smt add-lessD1 hd-conv-nth le-eq-less-or-eq length-greater-0-conv less-imp-add-positive*)
using *mp-alloc-stm3-lm2-inv-2* **apply** (*metis inv-mempool-info-maxsz-align4*)
apply(*simp add*: *set-bit-def rmhead-free-list-def head-free-list-def*)

apply(*subgoal-tac buf* (*mem-pool-info Va p*) + *jj* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** (*metis mp-alloc-stm3-body-frlst-otherlvl mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-maxsz*)
apply(*subgoal-tac get-bit* (*mem-pool-info Va*) *p ii jj* = *FREE*)
prefer *2* **apply**(*simp add*: *set-bit-def rmhead-free-list-def*)
apply(*simp add*: *set-bit-def rmhead-free-list-def head-free-list-def*)

apply(*rule conjI*)

apply *clarify*
apply(*rename-tac pa ii jj*)
apply(*subgoal-tac length* (*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* ! *nat* (*free-l Va t*)))
p)) = *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **using** *mp-alloc-stm3-body-len-lvls* **apply** *metis*
apply(*case-tac nat* (*free-l Va t*) = *ii*)

apply(*subgoal-tac* (*free-list*
(*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p) ! *ii*)) = (*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))))
prefer *2* **apply**(*simp add*:*level-empty-def set-bit-def rmhead-free-list-def head-free-list-def*)

apply(*subgoal-tac tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)) ! *jj* = (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)) ! *Suc jj*)
prefer *2* **apply**(*rule List.nth-tl*)
apply(*subgoal-tac length* (*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))) = *length* (*free-list*
(*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*)))
p) ! *ii*)))
prefer *2* **apply** *simp*
apply *metis*
apply(*subgoal-tac* (∃ *n*. *n* < *n-max* (*mem-pool-info Va p*) ∗ (*4* ˆ *ii*) ∧ (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)) ! *Suc jj* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)))

using *mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-nmax mp-alloc-stm3-body-minf-maxsz*
apply *metis*
apply(*subgoal-tac Suc jj* < *length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply**(*subgoal-tac jj* < *length* (*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))))
prefer *2* **apply** *metis*
apply(*simp add*:*level-empty-def*)
apply *metis*

using *mp-alloc-stm3-body-minf-buf mp-alloc-stm3-body-minf-maxsz mp-alloc-stm3-body-minf-nmax mp-alloc-stm3-body-frlst-otherlvl* **apply** *metis*

apply(*case-tac nat* (*free-l Va t*) = *ii*)
apply(*subgoal-tac* (*free-list*
(*levels* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*
(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l*
Va t)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l*
Va t)))
p) ! *ii*)) = (*tl* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))))
prefer *2* **apply**(*simp add:level-empty-def set-bit-def rmhead-free-list-def head-free-list-def*)
apply(*subgoal-tac distinct* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** *simp*
using *distinct-tl* **apply** *metis*

apply(*subgoal-tac distinct* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
prefer *2* **apply** (*metis mp-alloc-stm3-body-len-lvls*)
using *mp-alloc-stm3-body-frlst-otherlvl* **apply** *metis*
done

lemma *mp-alloc-stm3-lm2-inv-bitmap*:
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
inv-mempool-info Va ∧ *inv-bitmap-freelist Va* ∧ *inv-bitmap Va* ⟹
p ∈ *mem-pools Va* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
free-l Va t ≤ *alloc-l Va t* ⟹
¬ *free-l Va t* < *OK* ⟹
length (*lsizes Va t*) ≤ *length* (*levels* (*mem-pool-info Va p*)) ⟹
nat (*free-l Va t*) < *length* (*lsizes Va t*) ⟹
inv-bitmap
(*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info*
Va p) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*
(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l*
Va t))) (*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*free-l*
Va t)))

p)
(head-free-list (mem-pool-info Va p) (nat (free-l Va t))) (ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)),
data = head-free-list (mem-pool-info Va p) (nat (free-l Va t))⦈)⦈)
apply(*subgoal-tac inv-bitmap (set-bit-s Va p (nat (free-l Va t))*
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t))) (head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))
ALLOCATING))
prefer *2*
apply(*subgoal-tac get-bit-s Va p (nat (free-l Va t))*
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))) = FREE)
prefer *2* **apply**(*simp add:level-empty-def*)
apply(*subgoal-tac (block-num (mem-pool-info Va p) (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t)) ! NULL)*
(max-sz (mem-pool-info Va p) div 4 ^ nat (free-l Va t)))
= (block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))))
using *inv-bitmap-freelist-fl-FREE[of Va p nat (free-l Va t) 0]* **apply** *simp*
apply(*simp add:block-num-def rmhead-free-list-def head-free-list-def*)
apply (*simp add: hd-conv-nth inv-mempool-info-maxsz-align4*)
using *inv-bitmap-presv-setbit[of Va p (nat (free-l Va t))*
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))) ALLOCATING set-bit-s Va p (nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t))) (head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))
ALLOCATING] **apply** *simp*

apply(*rule inv-bitmap-presv-mpls-mpi2[of (set-bit-s Va p (nat (free-l Va t))*
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t))) (head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t)))
ALLOCATING) (Va⦇*blk := (blk Va)(t := head-free-list (mem-pool-info Va p) (nat (free-l Va t))),*
mem-pool-info :=
set-bit-allocating ((mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))) p (nat (free-l Va t))
(block-num (rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(head-free-list (mem-pool-info Va p) (nat (free-l Va t)))
(ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ nat (free-l Va t))),
allocating-node := allocating-node Va(t ↦

(|*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))|)|)|)])
apply(*simp add*: *set-bit-s-def set-bit-def block-num-def rmhead-free-list-def head-free-list-def*)
apply(*simp add*: *set-bit-s-def set-bit-def block-num-def rmhead-free-list-def head-free-list-def*)
apply *clarsimp* **apply**(*simp add*: *set-bit-s-def set-bit-def block-num-def rmhead-free-list-def head-free-list-def*)
apply (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*2*)
linorder-not-less list-update-beyond nth-list-update-eq nth-list-update-neq)
by *simp*

lemma *mp-alloc-stm3-lm2-inv*:
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
inv Va ⟹
allocating-node Va t = *None* ⟹
freeing-node Va t = *None* ⟹
p ∈ *mem-pools Va* ⟹
ETIMEOUT ≤ *timeout* ⟹
timeout = *ETIMEOUT* ⟶ *tmout Va t* = *ETIMEOUT* ⟹
¬ *rf Va t* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
free-l Va t ≤ *alloc-l Va t* ⟹
¬ *free-l Va t* < *0* ⟹
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* ⟹
inv (*Va*(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info :=
set-bit-allocating ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*

(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes Va t* !
nat (*free-l Va t*))),
allocating-node := *allocating-node Va*(*t* ↦
⦇*pool* = *p*, *level* = *nat* (*free-l Va t*),
block = *block-num*
(*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list*
(*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
(*block-num* (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l*
Va t)))
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
(*lsizes Va t* ! *nat* (*free-l Va t*)))
p)
(*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))) (*lsizes*
Va t ! *nat* (*free-l Va t*)),
data = *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦈)⦈)
apply(*subgoal-tac nat* (*free-l Va t*) < *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac length* (*lsizes Va t*) ≤ *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac nat* (*free-l Va t*) < *length* (*lsizes Va t*))
prefer *2* **apply** *linarith*
apply(*simp add*:*inv-def*)
apply(*rule conjI*)
apply(*simp add*:*inv-cur-def*)
apply(*rule conjI*)
using *mp-alloc-stm3-lm2-inv-thd-waitq* **apply** *fast*
apply(*rule conjI*)
using *mp-alloc-stm3-lm2-inv-mempool-info* **apply** *fast*
apply(*rule conjI*)
using *mp-alloc-stm3-lm2-inv-bitmap-freelist* **apply** *fast*
apply(*rule conjI*) **using** *mp-alloc-stm3-lm2-inv-bitmap* **apply** *simp*
apply(*rule conjI*) **using** *mp-alloc-stm3-lm2-inv-aux-vars* **apply** *simp*
apply(*rule conjI*) **using** *mp-alloc-stm3-lm2-inv-bitmap0* **apply** *simp*
apply(*rule conjI*) **using** *mp-alloc-stm3-lm2-inv-bitmapn* **apply** *simp*
using *mp-alloc-stm3-lm2-inv-bitmap-not4free* **apply** *simp*
done

lemma *mp-alloc-stm3-lm2-3-1*:
(*a*::*nat*) *mod b* = *0* ⟹ *c* ∗ *b* ∗ (*a div b*) = *c* ∗ *a* **by** *auto*

lemma *mp-alloc-stm3-lm2-3*:
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
inv Va ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹

free-l Va t ≤ *alloc-l Va t* ⟹
p ∈ *mem-pools Va* ⟹
¬ *free-l Va t* < *0* ⟹
max-sz (*mem-pool-info Va p*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) ⟹
let fl = *hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))); *mp* = *mem-pool-info Va p*
in ∃ *n*<*n-max mp* ∗ *4* ˆ *nat* (*free-l Va t*). *fl* = *buf mp* + *n* ∗ (*max-sz mp div 4* ˆ *nat* (*free-l Va t*)) ⟹
hd (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)))
< *buf* (*mem-pool-info Va p*) + *n-max* (*mem-pool-info Va p*) ∗ *max-sz* (*mem-pool-info Va p*)
apply(*subgoal-tac nat* (*free-l Va t*) < *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add:inv-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))) ≥ *buf* (*mem-pool-info Va p*))
prefer *2* **apply**(*simp add*: *inv-def*) **using** *inv-buf-le-fl*[*of Va p nat* (*free-l Va t*) *0*]
apply (*simp add*: *hd-conv-nth level-empty-def*)
apply (*simp add*: *hd-conv-nth level-empty-def Let-def*)
apply *clarify*

apply(*subgoal-tac max-sz* (*mem-pool-info Va p*) *mod* (*4* ˆ *nat* (*free-l Va t*)) = *0*)
prefer *2* **apply** (*metis ge-pow-mod-0 inv-mempool-info-def inv-def*)
apply(*subgoal-tac n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*free-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *max-sz* (*mem-pool-info Va p*))
prefer *2* **apply**(*subgoal-tac n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*free-l Va t*) ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*free-l Va t*))
= *n-max* (*mem-pool-info Va p*) ∗ *max-sz* (*mem-pool-info Va p*))
prefer *2*
using *mp-alloc-stm3-lm2-3-1*[*of max-sz* (*mem-pool-info Va p*) *4* ˆ *nat* (*free-l Va t*) *n-max* (*mem-pool-info Va p*)] **apply** *auto*[*1*]
apply(*subgoal-tac n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*free-l Va t*))
< (*n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*free-l Va t*)) ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*free-l Va t*)))
prefer *2* **apply** (*metis inv-mempool-info-def inv-def mp-alloc-stm3-lm2-inv-1-2 mult-less-mono1*)
apply *linarith*

apply *simp*
done

lemma *mp-alloc-stm3-lm2-5*:
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
inv Va ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
free-l Va t ≤ *alloc-l Va t* ⟹
p ∈ *mem-pools Va* ⟹
¬ *free-l Va t* < *0* ⟹

(hd (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t))) − buf (mem-pool-info Va p)) div
(max-sz (mem-pool-info Va p) div 4 ^ nat (free-l Va t))
< n-max (mem-pool-info Va p) ∗ 4 ^ nat (free-l Va t)
apply(subgoal-tac nat (free-l Va t) < length (levels (mem-pool-info Va p)))
prefer 2 **apply**(simp add:inv-def inv-mempool-info-def Let-def)
apply(subgoal-tac hd (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t))) ≥ buf (mem-pool-info Va p))
prefer 2 **apply**(simp add: inv-def) **using** inv-buf-le-fl[of Va p nat (free-l Va t) 0]
apply (simp add: hd-conv-nth level-empty-def)
apply (simp add: hd-conv-nth level-empty-def)
by (metis block-num-def inv-bitmap-freelist-fl-bnum-in inv-mempool-info-def length-greater-0-conv inv-def)

lemma mp-alloc-stm3-lm2-4:
inv Va ∧
p ∈ mem-pools Va ∧
free-list (levels (mem-pool-info Va p) ! nat (free-l Va t)) ≠ [] ⟹
nat (free-l Va t) < length (levels (mem-pool-info Va p)) ⟹ NULL < hd (free-list (levels (mem-pool-info Va p) ! nat (free-l Va t)))
using inv-imp-fl-lt0 **apply**(simp add:Let-def)
by (simp add: hd-conv-nth)

lemma mp-alloc-stm3-lm2:
Va ∈ mp-alloc-precond1-70-2-2 t p sz timeout ∩ {|´cur = Some t|} ⟹
¬ level-empty (mem-pool-info Va p) (nat (free-l Va t)) ⟹
{|let vb = Va(|blk := (blk Va)(t := head-free-list (mem-pool-info Va p) (nat (free-l Va t))),
mem-pool-info := (mem-pool-info Va)(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))|)
in ´(=) (vb(|mem-pool-info :=
set-bit-allocating (mem-pool-info vb) p (nat (free-l vb t))
(block-num (mem-pool-info vb p) (blk vb t) (lsizes vb t ! nat (free-l vb t)))|)) ∧
¬ level-empty (mem-pool-info Va p) (nat (free-l Va t))|}
⊆ {|´(allocating-node-update
(λ-. ´allocating-node(t ↦
(|pool = p, level = nat (´free-l t), block = block-num (´mem-pool-info p) (´blk t) (´lsizes t ! nat (´free-l t)),
data = ´blk t|))))
∈ {|´(Pair Va) ∈ Mem-pool-alloc-guar t|} ∩ mp-alloc-precond2-1 t p sz timeout|}
apply(subgoal-tac head-free-list (mem-pool-info Va p) (nat (free-l Va t)) ≠ NULL)
prefer 2
apply(subgoal-tac (nat (free-l Va t)) < length (levels (mem-pool-info Va p)))
prefer 2 **apply**(simp add:inv-def inv-mempool-info-def Let-def) **apply** force
apply(simp add:head-free-list-def level-empty-def) **using** mp-alloc-stm3-lm2-4

apply *simp*

apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply *clarify*
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply *clarify* **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def block-num-def*)
apply(*case-tac nat* (*free-l Va t*) = *i*)
using *mp-alloc-stm3-lm2-1*[*of mem-pool-info Va p nat* (*free-l Va t*)
tl (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)))
(*hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))) − *buf* (*mem-pool-info Va p*)) *div*
lsizes Va t ! *nat* (*free-l Va t*)] **apply** *meson*
apply *simp*
apply(*rule conjI*)
using *mp-alloc-stm3-lm2-inv* **apply** *simp*

apply *clarsimp* **apply**(*simp add*:*lvars-nochange-def*)

apply(*rule conjI*)
using *mp-alloc-stm3-lm2-inv* **apply** *simp*

apply(*rule conjI*) **apply** *clarsimp* **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*: *rmhead-free-list-def head-free-list-def set-bit-def*)

apply(*simp add*:*block-num-def*)
apply(*rule subst*[**where** *t*=*buf* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p* (*nat* (*free-l Va t*))
((*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) −
buf (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *div*
lsizes Va t ! *nat* (*free-l Va t*))

p) **and** *s=buf* (*mem-pool-info Va p*)])
apply(*simp add:head-free-list-def rmhead-free-list-def set-bit-def*)
apply(*rule subst*[**where** *t=max-sz* (*set-bit-allocating* ((*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *p*
(*nat* (*free-l Va t*))
((*head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) −
buf (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *div*
lsizes Va t ! *nat* (*free-l Va t*))
p) **and** *s=max-sz* (*mem-pool-info Va p*)])
apply(*simp add:head-free-list-def rmhead-free-list-def set-bit-def*)
apply(*simp add:head-free-list-def*)
apply(*subgoal-tac lsizes Va t* ! *nat* (*free-l Va t*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ (*nat* (*free-l Va t*)))
prefer *2* **apply** *auto*[*1*]
apply(*subgoal-tac max-sz* (*mem-pool-info Va p*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add:inv-def inv-mempool-info-def Let-def*)
apply (*metis align40 mod-mult-self1-is-0 semiring-normalization-rules*(*17*))
apply(*subgoal-tac let fl* = *hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*)));
mp = (*mem-pool-info Va p*) *in*
($\exists$ *n*. *n* < *n-max mp* ∗ (*4* ^ (*nat* (*free-l Va t*)))
$\wedge$ *fl* = *buf mp* + *n* ∗ (*max-sz mp div* (*4* ^ (*nat* (*free-l Va t*))))))
prefer *2* **apply**(*simp add:inv-def inv-bitmap-freelist-def level-empty-def Let-def*)
apply(*subgoal-tac* (*nat* (*free-l Va t*)) < *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply** (*simp add*: *inv-mempool-info-def Let-def*) **apply** (*smt in-set-conv-nth list*.*set-sel*(*1*))
apply(*rule conjI*)
apply (*metis add-diff-cancel-left′ div-mult-self-is-m mult-is-0 neq0-conv*)

apply(*rule subst*[**where** *t=n-max* (*set-bit-allocating* (λ*a*. *if a* = *p then rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) *else mem-pool-info Va a*)
p (*nat* (*free-l Va t*))
((*hd* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*free-l Va t*))) −
buf (*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))) *div*
lsizes Va t ! *nat* (*free-l Va t*))
p) **and** *s=n-max* (*mem-pool-info Va p*)])
apply(*simp add:rmhead-free-list-def set-bit-def*)

apply(*rule conjI*)
using *mp-alloc-stm3-lm2-5* **apply** *metis*
using *mp-alloc-stm3-lm2-3* **apply**(*simp add:Let-def*)
done

lemma *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) $\neq$ *NULL* $\Longrightarrow$
{*Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|), *Va* (|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),

mem-pool-info := (*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈} ∩
⦃*NULL* < ´*blk t*⦄ =
{*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈}
by *simp*

lemma *mp-alloc-stm3-lm1-1*:
inv Va ⟹ *p* ∈ *mem-pools Va* ⟹ *nat* (*free-l Va t*) < *length* (*levels* (*mem-pool-info Va p*)) ⟹
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) ⟹
{*V*. *V* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈
∧ *blk V t* ≠ *NULL*}
={*V*. *V* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈}
apply(*rule equalityI*) **apply**(*rule subsetI*)
apply *clarsimp*
apply(*rule subsetI*)
apply *clarsimp*
apply(*simp add*: *head-free-list-def*)
by (*simp add*: *level-empty-def mp-alloc-stm3-lm2-4*)

lemma *mp-alloc-stm3-lm1*:
mp-alloc-precond1-70-2-2 t p sz timeout ∩ ⦃´*cur* = *Some t*⦄ ∩ {*Va*} = {*Va*}
⟹ $\Gamma \vdash_I$ *Some* (*IF level-empty* (´*mem-pool-info p*) (*nat* (´*free-l t*)) *THEN*
´*blk* := ´*blk*(*t* := *NULL*)
ELSE
´*blk* := ´*blk*(*t* := *head-free-list* (´*mem-pool-info p*) (*nat* (´*free-l t*)));;

´*mem-pool-info* := ´*mem-pool-info* (*p* := *rmhead-free-list* (´*mem-pool-info p*) (*nat* (´*free-l t*)))

FI;;

IF ´*blk t* ≠ *NULL THEN*
´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*free-l t*))
(*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*))));;
´*allocating-node* := ´*allocating-node* (*t* := *Some* ⦇*pool* = *p*, *level* = *nat* (´*free-l t*),
block = (*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*)))), *data* = ´*blk t* ⦈)

FI) sat_p [*mp-alloc-precond1-70-2-2 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*Va*},
{(*s*, *t*). *s* = *t*}, *UNIV*, {|´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*|} ∩ *mp-alloc-precond2-1 t p sz timeout*]

apply(*subgoal-tac Va*∈*mp-alloc-precond1-70-2-2 t p sz timeout* ∩ {|´*cur* = *Some t*|})
prefer *2* **apply** *auto*[*1*]
apply(*rule Seq*[**where** *mid*=
if level-empty (*mem-pool-info Va p*) (*nat* (*free-l Va t*)) *then*
{*V*. *V* = *Va*(|*blk*:=(*blk Va*)(*t*:=*NULL*)|) ∧ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}
else
{*V*. *V* = *Va*(|*blk*:=(*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)(*p*:=*rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))|)
∧ ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}])

apply(*rule Cond*)
apply(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Seq*[**where** *mid*={*V*. *V* = *Va*(|*blk*:=(*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))|)
∧ ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}])
apply(*rule Basic*)
apply *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)
apply(*rule Basic*)
apply *clarify*

apply *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply *simp*

apply(*rule Cond*)
apply(*simp add*:*stable-def*)
apply(*case-tac* ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))
prefer *2*
apply(*subgoal-tac* {*Va*(|*blk* := (*blk Va*)(*t* := *NULL*)|), *Va*
(|*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),

mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈} ∩
⦃¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦄ = {})
prefer *2* **apply** *auto*[*1*]
using *Emptyprecond*[**where** *P*=*Some* (´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*free-l t*))
(*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*))));;
´*allocating-node* := ´*allocating-node* (*t* := *Some* ⦇*pool* = *p*, *level* = *nat* (´*free-l t*),
block = (*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*)))), *data* = ´*blk t* ⦈))
and *rely*={(*x*,*y*). *x*=*y*} **and** *guar*=*UNIV*
and *post*=⦃´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*⦄ ∩ *mp-alloc-precond2-1 t p sz timeout*]
apply *meson* **apply** *auto*[*1*]

apply(*rule subst*[**where** *t*= (*if level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))
then ⦃´(=) (*Va*⦇*blk* := (*blk Va*)(*t* := *NULL*)⦈) ∧ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦄
else ⦃´(=) (*Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈) ∧
¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))⦄) ∩
⦃´*blk t* ≠
NULL⦄ **and** *s* = {*V*. *V* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈
∧ ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))
∧ *blk V t* ≠ *NULL*}])
apply *auto*[*1*]
apply(*rule subst*[**where** *t*= {*V*. *V* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈
∧ ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))
∧ *blk V t* ≠ *NULL*} **and** *s* = {*V*. *V* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),

mem-pool-info := (mem-pool-info Va)
(p := rmhead-free-list (mem-pool-info Va p) (nat (free-l Va t)))⦈
∧ ¬ level-empty (mem-pool-info Va p) (nat (free-l Va t))}])
apply(*subgoal-tac* (*nat* (*free-l Va t*)) < *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def Let-def*) **apply** *force*
apply *simp*
using *mp-alloc-stm3-lm1-1* **apply** *force*

apply(*rule Seq*[**where** *mid*=
{*V*. *let vb* = *Va*⦇*blk* := (*blk Va*)(*t* := *head-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))),
mem-pool-info := (*mem-pool-info Va*) (*p* := *rmhead-free-list* (*mem-pool-info Va p*) (*nat* (*free-l Va t*)))⦈
in V = *vb*⦇ *mem-pool-info* := *set-bit-allocating* (*mem-pool-info vb*) *p* (*nat* (*free-l vb t*))
(*block-num* (*mem-pool-info vb p*) (*blk vb t*) ((*lsizes vb t*)!(*nat* (*free-l vb t*))))⦈
∧ ¬ *level-empty* (*mem-pool-info Va p*) (*nat* (*free-l Va t*))}])
apply(*rule Basic*) **apply** *clarsimp*
apply *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)
apply(*rule Basic*)
using *mp-alloc-stm3-lm2* **apply** *meson*
apply *simp* **apply**(*simp add*:*stable-def*)
using *stable-id2*[*of* ⦃´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*⦄ ∩ *mp-alloc-precond2-1 t p sz timeout*]
apply *meson*

apply(*unfold Skip-def*)
apply(*rule Basic*)
using *mp-alloc-stm3-lm3* **apply** *meson*
apply *simp* **apply**(*simp add*:*stable-def*)
using *stable-id2*[*of* ⦃´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*⦄ ∩ *mp-alloc-precond2-1 t p sz timeout*]
apply *meson*
apply *simp*

done

lemma *mp-alloc-stm3-lm*:
Γ $\vdash_I$ *Some* (*t* ► *ATOMIC*
IF level-empty (´*mem-pool-info p*) (*nat* (´*free-l t*)) *THEN*
´*blk* := ´*blk*(*t* := *NULL*)
ELSE
´*blk* := ´*blk*(*t* := *head-free-list* (´*mem-pool-info p*) (*nat* (´*free-l t*)));;

´*mem-pool-info* := ´*mem-pool-info* (*p* := *rmhead-free-list* (´*mem-pool-info* *p*) (*nat* (´*free-l t*)))

FI;;

IF ´*blk t* ≠ *NULL THEN*
´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*free-l t*))
(*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*))));;
´*allocating-node* := ´*allocating-node* (*t* := *Some* (|*pool* = *p*, *level* = *nat* (´*free-l t*),
block = (*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*free-l t*)))), *data* = ´*blk t* |))
FI
END)
sat_p [*mp-alloc-precond1-70-2-2 t p sz timeout*, *Mem-pool-alloc-rely t*, *Mem-pool-alloc-guar t*,
mp-alloc-precond2-1 t p sz timeout]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-70-2-2-stb* **apply** *simp*
using *mp-alloc-precond2-1-stb* **apply** *simp*
apply(*clarify*)
apply(*rule Await*)
using *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*

apply *clarify*
apply(*case-tac V = Va*) **prefer** *2* **apply** *simp* **using** *Emptyprecond* **apply** *auto*[*1*]
apply *simp*
apply(*case-tac mp-alloc-precond1-70-2-2 t p sz timeout*
∩ {|´*cur* = *Some t*|} ∩ {*Va*} = {})
using *Emptyprecond* **apply** *auto*[*1*]
apply(*subgoal-tac mp-alloc-precond1-70-2-2 t p sz timeout*
∩ {|´*cur* = *Some t*|} ∩ {*Va*} = {*Va*})
prefer *2* **using** *int1-eq*[**where** *P*=*mp-alloc-precond1-70-2-2 t p sz timeout* ∩ {|´*cur* = *Some t*|}] **apply** *meson*
using *mp-alloc-stm3-lm1*[*of t p timeout sz*] **apply** *auto*[*1*]
done

term *mp-alloc-precond1-70-2-2 t p sz timeout*
term *mp-alloc-precond2-1 t p sz timeout*

21.7 stm4

abbreviation *mp-alloc-precond2-1-1-loopinv-0 t p sz tm* ≡
mp-alloc-precond2-1-1-loopinv t p sz tm ∩ {|´*from-l t* < ´*alloc-l t*|}

lemma *mp-alloc-precond2-1-1-loopinv-0-stb*: *stable* (*mp-alloc-precond2-1-1-loopinv-0* *t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond2-1-1-loopinv-stb* **apply** *auto*[*1*]
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*] **apply** *clarify*
apply(*simp add*: *block-num-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

abbreviation *mp-alloc-precond2-1-1-loopinv-1′ t p sz tm* ≡
mp-alloc-precond2-1-1 t p sz tm ∩ {|´*from-l t* ≤ ´*alloc-l t* ∧ ´*from-l t* ≥ ´*free-l t*
∧ ´*allocating-node t* = *Some* (|*pool* = *p*, *level* = *nat* (´*from-l t* + *1*),
block = *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*from-l t* + *1*))),
data = ´*blk t* |) |}

abbreviation *mp-alloc-precond2-1-1-loopinv-1 t p sz tm* ≡
{*s*. *inv s*} ∩ {|´*freeing-node t* = *None*|} ∩ {|*p* ∈ ´*mem-pools* ∧ *tm* ≥ −*1*|} ∩ *mp-alloc-precond7-ext t p sz tm* ∩ {|¬ ´*rf t*|}
∩ *mp-alloc-precond1-70-ext t p sz tm* ∩ − {|´*alloc-l t* < *0*|}
∩ − {|´*free-l t* < *0*|} ∩ −{|´*blk t* = *NULL*|} ∩ {|´*from-l t* < ´*alloc-l t*|}
∩ {|´*from-l t* ≤ ´*alloc-l t* ∧ ´*from-l t* ≥ ´*free-l t*
∧ ´*allocating-node t* = *Some* (|*pool* = *p*, *level* = *nat* (´*from-l t* + *1*),
block = *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*from-l t* + *1*))),
data = ´*blk t* |)
∧ ´*alloc-memblk-data-valid p* (*the* (´*allocating-node t*))
∧ (∃ *n*. *n* < *n-max* (´*mem-pool-info p*) ∗ (*4* ˆ (*nat* (´*from-l t* + *1*)))
∧ ´*blk t* = *buf* (´*mem-pool-info p*) + *n* ∗ (*max-sz* (´*mem-pool-info p*) *div* (*4* ˆ (*nat* (´*from-l t* + *1*))))) |}

term *mp-alloc-precond2-1-1-loopinv-0 t p sz tm*
term *mp-alloc-precond2-1-1-loopinv-1 t p sz tm*
term *mp-alloc-precond2-1-1-loopinv-1′ t p sz tm*

lemma *mp-alloc-precond2-1-1-loopinv-1′-stb*: *stable* (*mp-alloc-precond2-1-1-loopinv-1′* *t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*)
using *mp-alloc-precond2-1-1-stb* **apply** *auto*[*1*]
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*] **apply** *clarify*

apply(*simp add*: *block-num-def lvars-nochange-rel-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-precond2-1-1-loopinv-1-stb*: *stable* (*mp-alloc-precond2-1-1-loopinv-1 t p sz tm*) (*Mem-pool-alloc-rely t*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*) **apply**(*rule stable-int2*) **apply**(*rule stable-int2*)
apply(*rule stable-int2*)
apply (*simp add*: *stable-inv-alloc-rely1*)
apply (*simp add*: *mp-alloc-freenode-stb*)
apply (*simp add*: *mp-alloc-precond1-ext-stb*)
apply (*simp add*: *mp-alloc-precond7-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-0-ext-stb*)
using *mp-alloc-precond1-70-ext-stb* **apply** *blast*
apply (*simp add*: *mp-alloc-precond1-70-2-ext-stb*)
apply (*simp add*: *mp-alloc-precond1-70-2-2-ext-stb*)
using *mp-alloc-precond2-1-1-ext-stb* **apply** *blast*
apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*simp add*:*Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*

apply(*simp add*:*stable-def*) **apply** *clarify*
apply(*subgoal-tac buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply(*subgoal-tac from-l x t* + *1* = *from-l y t* + *1*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply(*subgoal-tac blk x t* = *blk y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac allocating-node x t* = *allocating-node y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac lsizes x t* = *lsizes y t*)
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*
apply(*subgoal-tac n-max* (*mem-pool-info x p*) = *n-max* (*mem-pool-info y p*))
prefer *2* **apply**(*simp add*: *Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def*)
apply *metis*
apply(*subgoal-tac block-num* (*mem-pool-info x p*) (*blk x t*) (*lsizes x t* ! *nat* (*from-l x t* + *1*))
= *block-num* (*mem-pool-info y p*) (*blk y t*) (*lsizes y t* ! *nat* (*from-l y t* + *1*)))

prefer *2* **apply**(*simp add*: *block-num-def Mem-pool-alloc-rely-def lvars-nochange-rel-def lvars-nochange-def*)
apply(*case-tac x=y*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-rely-def gvars-conf-stable-def gvars-conf-def lvars-nochange-rel-def lvars-nochange-def*)
apply *smt*

done

abbreviation *mp-alloc-stm4-pre-precond1 Va t p* ≡
Va⦇*bn* := (*bn Va*) (*t* := *block-num* (*mem-pool-info Va p*) (*blk Va t*) ((*lsizes Va t*)!(*nat* (*from-l Va t*))))⦈
abbreviation *mp-alloc-stm4-pre-precond2 Va t p* ≡
Va⦇*mem-pool-info* := *set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*)) (*bn Va t*)⦈
abbreviation *mp-alloc-stm4-pre-precond3 Va t p* ≡
Va⦇*mem-pool-info* := *set-bit-allocating* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t* + *1*)) (*4* ∗ *bn Va t*)⦈
abbreviation *mp-alloc-stm4-pre-precond4 Va t p* ≡
Va⦇*allocating-node* := (*allocating-node Va*)(*t* := *Some* ⦇*pool* = *p*, *level* = *nat* (*from-l Va t* + *1*),
block = *4* ∗ *bn Va t*, *data* = *blk Va t* ⦈)⦈
abbreviation *mp-alloc-stm4-pre-precond5 Va t p* ≡ *Va*⦇*i* := (*i Va*) (*t* := *1*)⦈

definition *mp-alloc-stm4-pre-precond-f Va t p*
≡ *mp-alloc-stm4-pre-precond5*
(*mp-alloc-stm4-pre-precond4*
(*mp-alloc-stm4-pre-precond3*
(*mp-alloc-stm4-pre-precond2*
(*mp-alloc-stm4-pre-precond1 Va t p*) *t p*) *t p*) *t p*) *t p*

abbreviation *mp-alloc-stm4-loopinv Va t mp*
≡ {*V*. *cur V* = *cur Va* ∧ *tick V* = *tick Va* ∧ *thd-state V* = *thd-state Va* ∧ (*V*,*Va*)∈*gvars-conf-stable*
∧ (∀ *p*. *p* ≠ *mp* ⟶ *mem-pool-info V p* = *mem-pool-info Va p*)
∧ *wait-q* (*mem-pool-info V mp*) = *wait-q* (*mem-pool-info Va mp*)
∧ (∀ *t'*. *t'* ≠ *t* ⟶ *lvars-nochange t' V Va*)
∧ (∀ *jj*. *jj* ≠ *nat* (*from-l Va t* + *1*) ⟶ *levels* (*mem-pool-info V mp*) ! *jj* = *levels* (*mem-pool-info Va mp*) ! *jj*)
∧ (*bits* (*levels* (*mem-pool-info V mp*) ! *nat* (*from-l Va t* + *1*))
= *list-updates-n* (*bits* (*levels* (*mem-pool-info Va mp*) ! *nat* (*from-l Va t* + *1*))) (*bn Va t* ∗ *4* + *1*) (*i V t* − *1*) *FREE*)
∧ (*free-list* (*levels* (*mem-pool-info V mp*) ! *nat* (*from-l Va t* + *1*))
= *inserts* (*map* (λ*ii*. (*lsizes Va t*) ! (*nat* (*from-l Va t* + *1*)) ∗ *ii* + *blk V t*) [*1*..<*i V t*])
(*free-list* (*levels* (*mem-pool-info Va mp*) ! *nat* (*from-l Va t* + *1*))))
∧ *j V* = *j Va* ∧ *ret V* = *ret Va* ∧ *endt V* = *endt Va* ∧ *rf V* = *rf Va* ∧ *tmout V* = *tmout Va*
∧ *lsizes V* = *lsizes Va* ∧ *alloc-l V* = *alloc-l Va* ∧ *free-l V* = *free-l Va*

$\wedge$ *from-l V = from-l Va* $\wedge$ *blk V = blk Va* $\wedge$ *nodev V = nodev Va*
$\wedge$ *bn V = bn Va* $\wedge$ *alloc-lsize-r V = alloc-lsize-r Va* $\wedge$ *lvl V = lvl Va* $\wedge$ *bb V = bb Va*
$\wedge$ *block-pt V = block-pt Va* $\wedge$ *th V = th Va* $\wedge$ *need-resched V = need-resched Va*
$\wedge$ *mempoolalloc-ret V = mempoolalloc-ret Va* $\wedge$ *freeing-node V = freeing-node Va*
$\wedge$ *allocating-node V = allocating-node Va*
$\wedge$ *i V t > 0* $\wedge$ *i V t* $\leq$ *4*}

lemma *in-mp-alloc-stm4-loopinv*: *i V t = 1* $\Longrightarrow$ *V* $\in$ *mp-alloc-stm4-loopinv V t mp*
apply *simp*
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply**(*simp add*:*lvars-nochange-def*)
apply(*simp add*:*inserts-def*)
done

abbreviation *mp-alloc-stm4-while-precond1 V t p* $\equiv$
V(|*lbn* := (*lbn V*) (*t* := *4* ∗ *bn V t* + *i V t*)|)
abbreviation *mp-alloc-stm4-while-precond2 V t p* $\equiv$
V(|*lsz* := (*lsz V*) (*t* := *lsizes V t* ! *nat* (*from-l V t* + *1*))|)
abbreviation *mp-alloc-stm4-while-precond3 V t p* $\equiv$
V(|*block2* := (*block2 V*) (*t* := *lsz V t* ∗ *i V t* + *blk V t*)|)
abbreviation *mp-alloc-stm4-while-precond4 V t p* $\equiv$
V(|*mem-pool-info* := *set-bit-free* (*mem-pool-info V*) *p* (*nat* (*from-l V t* + *1*)) (*lbn V t*)|)
abbreviation *mp-alloc-stm4-while-precond5 V t p* $\equiv$
V(|*mem-pool-info* := (*mem-pool-info V*) (*p* := *append-free-list* (*mem-pool-info V p*) (*nat* (*from-l V t* + *1*)) (*block2 V t*))|)

lemma *mp-alloc-stm4-pre-in*:
{*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*} $\cap$
{|*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)|}
$\subseteq$ {|´*mp-alloc-stm4-while-precond5 t p*
$\in$ {*mp-alloc-stm4-while-precond5*
(*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*) *t p*}|}
by *auto*

lemma *mp-alloc-stm4-lsizes*: *lsizes Va t = lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-froml*: *from-l Va t* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-buf*: *buf* (*mem-pool-info Va q*) = *buf* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *q*)
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def set-bit-def*)

lemma *mp-alloc-stm4-nmax*: *n-max* (*mem-pool-info Va q*) = *n-max* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *q*)
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def set-bit-def*)

lemma *mp-alloc-stm4-pre-maxsz*: *max-sz* (*mem-pool-info Va q*) = *max-sz* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *q*)
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def set-bit-def*)

lemma *mp-alloc-stm4-blk*: *blk Va* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*)
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-blockfit-help-1*:
p ∈ *mem-pools Va* ⟹ *inv Va* ⟹
(∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii*) ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) = *max-sz* (*mem-pool-info Va p*) ⟹
Suc (*nat* (*from-l Va t*)) < *length* (*lsizes Va t*) ⟹
max-sz (*mem-pool-info Va p*) = *4* ˆ *nat* (*from-l Va t*) ∗ *lsizes Va t* ! *nat* (*from-l Va t*)
apply(*simp add*:*inv-def inv-mempool-info-def*)
apply(*subgoal-tac* ∃ *n*>*0*. *max-sz* (*mem-pool-info Va p*) = *4* ∗ *n* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** *meson*
apply(*subgoal-tac max-sz* (*mem-pool-info Va p*) *mod* (*4* ˆ *nat* (*from-l Va t*)) = *0*)
prefer *2* **apply** *clarsimp* **using** *ge-pow-mod-0*[*of nat* (*from-l Va t*) *n-levels* (*mem-pool-info Va p*)] **apply** *auto*[*1*]
using *mod0-div-self*[*of max-sz* (*mem-pool-info Va p*) *4* ˆ *nat* (*from-l Va t*)] **apply** *simp*
done

lemma *mp-alloc-stm4-blockfit-help-2*:
ii ≥ *0* ⟹ (*a*::*nat*) *mod 4* ˆ *nat* (*ii*+*1*) = *0* ⟹ *a div 4* ˆ *nat* (*ii*+*1*) ∗ *4* = *a div 4* ˆ (*nat ii*)
apply(*subgoal-tac nat* (*ii*+*1*) = *nat ii* + *1*) **prefer** *2* **apply** *auto*[*1*]
by *auto*

lemma *mp-alloc-stm4-blockfit-help3*:
inv Va ⟹

p ∈ *mem-pools Va* ⟹
nmax > *0* ⟹
maxsz mod frml = *0* ⟹
n < *nmax* ∗ *frml* ⟹
blk Va t = *buf* (*mem-pool-info Va p*) + *n* ∗ (*maxsz div frml*) ⟹
maxsz div frml + *blk Va t*
≤ *nmax* ∗ *maxsz* + *buf* (*mem-pool-info Va p*)
apply(*case-tac n* = *0*)
apply (*metis* (*no-types*, *lifting*) *Nat.add-0-right add-le-mono1 div-le-dividend div-mult-self1-is-m le-trans mult-is-0*)

apply(*subgoal-tac blk Va t* ≤ *buf* (*mem-pool-info Va p*) + (*nmax* ∗ *frml* − *1*) ∗ (*maxsz div frml*))
prefer *2* **apply** *auto*[*1*]
by (*smt add.left-commute add-diff-cancel-left′ le-diff-conv mp-alloc-stm3-lm2-3-1 mult-eq-if nat-add-left-cancel-le not-less-zero*)

lemma *mp-alloc-stm4-blockfit-help4*:
inv Va ∧
p ∈ *mem-pools Va* ∧
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ∧
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ∧
from-l Va t < *alloc-l Va t* ∧
¬ *free-l Va t* < *OK* ∧
free-l Va t ≤ *from-l Va t* ⟹
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*) ∗ *4* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*)
apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *mod 4* ˆ *nat* (*from-l Va t* + *1*) = *0*)
prefer *2*
apply(*rule subst*[**where** *t*=*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) **and** *s*=*max-sz* (*mem-pool-info Va p*)])
apply (*metis inv-maxsz-align4*)
apply(*subgoal-tac nat* (*from-l Va t* + *1*) < *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply** (*smt nat-less-iff*)
apply(*simp add*:*inv-def inv-mempool-info-def Let-def*)
apply(*subgoal-tac n-levels* (*mem-pool-info Va p*) = *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply** *simp*
apply (*metis ge-pow-mod-0*)
apply(*rule mp-alloc-stm4-blockfit-help-2*)
apply (*metis int-nat-eq linorder-not-less nat-int neq0-conv zless-nat-conj*)
apply *simp*
done

lemma *mp-alloc-stm4-blockfit-help*:
Va ∈ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄ ∧ *ii* < *4*

$\Longrightarrow$ *block-fits* (*mem-pool-info Va p*) (*lsizes Va t* ! *nat* (*from-l Va t* + *1*) ∗ *ii* + *blk Va t*)
(*lsizes Va t* ! *nat* (*from-l Va t* + *1*))
apply(*simp add*:*block-fits-def block-num-def buf-size-def*)

apply(*case-tac alloc-l Va t* = *ETIMEOUT* ∧ *free-l Va t* = *ETIMEOUT* ∧ *length* (*lsizes Va t*) = *Suc NULL*)
apply *simp* **apply** *simp*

apply(*subgoal-tac lsizes Va t* ! *nat* (*from-l Va t* + *1*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*))
prefer *2* **apply**(*subgoal-tac nat* (*from-l Va t* + *1*) < *length* (*lsizes Va t*))
prefer *2* **apply** (*smt nat-less-iff*)
apply *simp*
apply(*subgoal-tac lsizes Va t* ! *nat* (*from-l Va t*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*))
prefer *2* **apply**(*subgoal-tac nat* (*from-l Va t*) < *length* (*lsizes Va t*))
prefer *2* **apply** (*smt nat-less-iff*)
apply *simp*

apply(*rule subst*[**where** *t*=*lsizes Va t* ! *nat* (*from-l Va t* + *1*) ∗ *ii* + *blk Va t* + *lsizes Va t* ! *nat* (*from-l Va t* + *1*)
and *s*=*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*) ∗ *ii* + *blk Va t* + *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*)])
apply *simp*
apply(*subgoal-tac* (*blk Va t* − *buf* (*mem-pool-info Va p*)) *div lsizes Va t* ! *nat* (*from-l Va t*)
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*))
prefer *2* **apply** *force*
apply(*subgoal-tac blk Va t* ≥ *buf* (*mem-pool-info Va p*))
prefer *2* **apply** *force*
apply(*subgoal-tac* (*blk Va t* − *buf* (*mem-pool-info Va p*)) *mod* (*ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*)) = *0*)
prefer *2* **apply** (*metis diff-add-inverse inv-maxsz-align4 mod-mult-self2-is-0*)

apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*) ∗ *ii* + *blk Va t* +
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*)
≤ *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*) ∗ *4* + *blk Va t*)
prefer *2* **apply** *simp*

apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t* + *1*) ∗ *4*
= *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*))
prefer *2* **using** *mp-alloc-stm4-blockfit-help4* **apply** *simp*

apply(*subgoal-tac ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*) + *blk Va t*
$\leq$ *n-max* (*mem-pool-info Va p*) $*$ *max-sz* (*mem-pool-info Va p*) + *buf* (*mem-pool-info Va p*))
apply *simp*
apply *clarify*
apply(*subgoal-tac n-max* (*mem-pool-info Va p*) > *0*)
prefer *2* **apply** (*metis gr0I mult-is-0 not-less-zero*)
apply(*subgoal-tac max-sz* (*mem-pool-info Va p*) *mod 4* ˆ *nat* (*from-l Va t*) = *0*)
prefer *2* **apply**(*subgoal-tac* $\exists$ *n*>*0*. *max-sz* (*mem-pool-info Va p*) = (*4* $*$ *n*) $*$ (*4* ˆ *n-levels* (*mem-pool-info Va p*)))
prefer *2* **apply** (*metis inv-mempool-info-def inv-def*)
apply (*smt ge-pow-mod-0 of-nat-less-imp-less zless-nat-conj zless-nat-eq-int-zless*)

using *mp-alloc-stm4-blockfit-help3*[*of Va p n-max* (*mem-pool-info Va p*) *max-sz* (*mem-pool-info Va p*) *4* ˆ *nat* (*from-l Va t*) - *t*]
by (*metis inv-maxsz-align4*)

lemma $a \geq b \Longrightarrow c * d \geq e \Longrightarrow (a{::}nat) - b \leq c * d - e \Longrightarrow a + e - b \leq c * d$
by (*simp add*: *Nat.le-diff-conv2*)

lemma $a + b > c \Longrightarrow a + b - c \leq d \Longrightarrow e + f < b \Longrightarrow e + a + f - Suc\ c < d$
by *simp*

lemma $(x{::}nat) > y \Longrightarrow \exists n.\ (4{::}nat)\ \hat{}\ x = 4\ \hat{}\ y * n$
apply(*subgoal-tac* $4\ \hat{}\ x = 4\ \hat{}\ y * 4\ \hat{}\ (x - y)$) **prefer** *2* **apply** *auto*
by (*metis add-diff-inverse-nat less-imp-le-nat not-less power-add*)

lemma *int-empt1*: ($\forall v.\ v \in P \longrightarrow v \notin Q$) $\Longrightarrow P \cap Q = \{\}$ **by** *auto*

lemma *mp-alloc-stm4-blockfit1-1*:
allocating-node Va t =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*), *block* = *block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *blk Va t*|) $\wedge$ *data* (*the* (*allocating-node Va t*)) = *buf* (*mem-pool-info Va p*) +
block (*the* (*allocating-node Va t*)) $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *level* (*the* (*allocating-node Va t*)))
$\wedge$ *block* (*the* (*allocating-node Va t*)) < *n-max* (*mem-pool-info Va p*) $*$ *4* ˆ *level* (*the* (*allocating-node Va t*)) $\Longrightarrow$
alloc-blk-valid Va p (*nat* (*from-l Va t*)) (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) (*blk Va t*)

apply(*simp add*:*block-num-def*) **apply** *auto*
done

lemma *mp-alloc-stm4-blockfit1*:
$Va \in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
$V \in$ *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* $\cap$ ⦃´*i t* <
4⦄ $\Longrightarrow$
$\forall v.\ v \in \{$*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*$\}$ $\longrightarrow$
$v \notin -$ ⦃*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)⦄
apply *simp*
apply(*simp add*:*block-fits-def buf-size-def set-bit-def*)
apply(*subgoal-tac lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *lsizes Va t*)
prefer *2* **using** *mp-alloc-stm4-lsizes* **apply** *metis*
apply(*subgoal-tac from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *from-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-froml* **apply** *metis*
apply(*subgoal-tac buf* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) =
buf (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-pre-buf* **apply** *metis*
apply(*subgoal-tac n-max* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*)
= *n-max* (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-nmax* **apply** *metis*
apply(*subgoal-tac max-sz* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*)
p) = *max-sz* (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-pre-maxsz* **apply** *metis*
apply(*subgoal-tac blk* (*mp-alloc-stm4-pre-precond-f Va t p*) = *blk Va*)
prefer *2* **using** *mp-alloc-stm4-blk* **apply** *metis*

apply(*subgoal-tac buf* (*mem-pool-info V p*) = *buf* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*))
prefer *2* **apply** (*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac n-max* (*mem-pool-info V p*) = *n-max* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*))
prefer *2* **apply** (*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac max-sz* (*mem-pool-info V p*) = *max-sz* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*))
prefer *2* **apply** (*simp add*:*gvars-conf-stable-def gvars-conf-def*)

apply(*rule subst*[**where** *t*=*buf* (*mem-pool-info V p*) **and** *s*=*buf* (*mem-pool-info*
Va p)]) **apply** *simp*
apply(*rule subst*[**where** *t*=*n-max* (*mem-pool-info V p*) **and** *s*=*n-max* (*mem-pool-info*
Va p)]) **apply** *simp*
apply(*rule subst*[**where** *t*=*max-sz* (*mem-pool-info V p*) **and** *s*=*max-sz* (*mem-pool-info*
Va p)]) **apply** *simp*
apply(*rule subst*[**where** *t*=*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* **and**
s=*lsizes Va t*]) **apply** *simp*
apply(*rule subst*[**where** *t*=*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* **and**

s=from-l Va t]) **apply** *simp*
apply(*rule subst*[**where** *t=blk* (*mp-alloc-stm4-pre-precond-f Va t p*) **and** *s=blk Va*]) **apply** *simp*

using *mp-alloc-stm4-blockfit-help* [*of Va t p timeout sz i V t*] **apply**(*unfold block-fits-def buf-size-def*) **apply** *simp*
apply(*case-tac alloc-l Va t = ETIMEOUT* ∧ *free-l Va t = ETIMEOUT* ∧ *length* (*lsizes Va t*) = *Suc NULL*)
apply *simp*
apply *simp*
apply(*subgoal-tac alloc-blk-valid Va p* (*nat* (*from-l Va t*)) (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) (*blk Va t*))
prefer *2* **using** *mp-alloc-stm4-blockfit1-1*[*of Va t p*] **apply** *argo* **apply** *metis*
done

lemma *mp-alloc-stm4-blockfit*:
Va ∈ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|} ⟹
V ∈ *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* ∩ {|´*i t* < *4*|} ⟹
{*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*} ∩
− {|*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)|} = {}
using *mp-alloc-stm4-blockfit1*[*of Va t p timeout sz V*]
int-empt1[*of* {*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*} − {|*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)|}]
apply *meson*

done

term *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|}

term *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|}
term *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* ∩ {|´*i t* < *4*|}
term {*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*} ∩
− {|*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)|}

lemma *mp-alloc-stm4-inv-mif-buf*: *buf* (*mem-pool-info Va pa*) = *buf* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-inv-mif-mxsz*: *max-sz* (*mem-pool-info Va pa*) = *max-sz* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-inv-mif-nmax*: *n-max* (*mem-pool-info Va pa*) = *n-max* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-inv-mif-nlvls*: *n-levels* (*mem-pool-info Va pa*) = *n-levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-inv-mif-len*: *length* (*levels* (*mem-pool-info Va pa*)) = *length* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*))
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-inv-bits-len*: *length* (*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*) ! *ii*))
= *length* (*bits* (*levels* (*mem-pool-info Va pa*) ! *ii*))

apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*case-tac* $p \neq pa$) **apply**(*simp add*: *set-bit-def*)
apply *simp*
apply(*case-tac nat* (*from-l Va t*) = *ii*) **apply** *simp*
apply (*metis set-bit-prev-len set-bit-prev-len2*)
by (*metis set-bit-prev-len set-bit-prev-len2*)

lemma *inserts-comm*:
inserts ilst lst @ [*v*] = *inserts* (*ilst* @ [*v*]) *lst*
by (*simp add*: *inserts-def*)

lemma *mp-alloc-stm4-while-isucc''*:
nat (*from-l Vb t* + *1*) < *length* (*levels* (*mem-pool-info Vb p*)) $\Longrightarrow$
$V \in$ *mp-alloc-stm4-loopinv Vb t p* $\cap$ $\{\!|$ *´i t* < *4* $|\!\}$ $\Longrightarrow$
i V t > *0* $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (*´i* := *´i*(*t* := *Suc* (*´i t*))) sat_p

[{*mp-alloc-stm4-while-precond5*
(*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*) *t p*},
{(*s*, *t*). *s* = *t*}, *UNIV*, *mp-alloc-stm4-loopinv Vb t p*]
apply(*rule Basic*)

apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*) **apply** *clarsimp*
apply(*rule conjI*) **apply** *clarsimp*
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply *clarsimp* **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*case-tac ia* $\neq$ *nat* (*from-l Vb t* + *1*)) **apply** *auto*[*1*]
apply(*case-tac ia* < *length* (*levels* (*mem-pool-info Vb p*))) **apply** *fastforce*
apply *fastforce*
apply *clarsimp*
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*) **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply *clarsimp* **apply**(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def*)
apply(*rule conjI*)
apply *clarsimp*
apply(*simp add*:*append-free-list-def set-bit-def lvars-nochange-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def*)
apply(*subgoal-tac length* (*levels* (*mem-pool-info Vb p*)) = *length* (*levels* (*mem-pool-info V p*)))
prefer *2* **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac nat* (*from-l Vb t* + *1*) < *length* (*levels* (*mem-pool-info Vb p*)))
prefer *2* **apply**(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*)
apply(*simp add*:*append-free-list-def set-bit-def*)
using *lst-updts-eq-updts-updt*[*of i V t bits* (*levels* (*mem-pool-info Vb p*) ! *nat* (*from-l Vb t* + *1*))
Suc (*bn Vb t* $*$ *4*) *FREE*]
apply (*simp add*: *semiring-normalization-rules*(*7*))
apply(*simp add*:*append-free-list-def set-bit-def*)

using *inserts-comm* **apply** *fast*

apply *fast* **using** *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*
done

lemma *mp-alloc-stm4-while-isucc*:
$Va \in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
$V \in$ *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* $\cap$ ⦃´*i t* < *4*⦄ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*i* := ´*i*(*t* := *Suc* (´*i t*))) sat_p
[{*mp-alloc-stm4-while-precond5*
(*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*) *t p*},
{(*s*, *t*). *s* = *t*}, *UNIV*, *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p*]
apply(*rule mp-alloc-stm4-while-isucc''*)
apply *clarsimp*
apply(*rule subst*[**where** *s*=*from-l Va t* **and** *t*=*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*])
using *mp-alloc-stm4-pre-froml* **apply** *blast*
apply(*rule subst*[**where** *s*=*length* (*levels* (*mem-pool-info Va p*)) **and**
t=*length* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*)))])
using *mp-alloc-stm4-inv-mif-len* **apply** *blast*
apply(*subgoal-tac n-levels* (*mem-pool-info Va p*) = *length* (*levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add*:*inv-def inv-mempool-info-def*) **apply** *metis*
apply *linarith*

apply *assumption*

apply *clarsimp*
done

lemma *mp-alloc-stm4-while-help*:
$Va \in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\Longrightarrow$
$V \in$ *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* $\cap$ ⦃´*i t* < *4*⦄ $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*lbn* := ´*lbn*(*t* := *4* ∗ ´*bn t* + ´*i t*);;
´*lsz* := ´*lsz*(*t* := ´*lsizes t* ! *nat* (´*from-l t* + *1*));;
´*block2* := ´*block2*(*t* := ´*lsz t* ∗ ´*i t* + ´*blk t*);;
´*mem-pool-info* := *set-bit-free* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (´*lbn t*);;
IF block-fits (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info*(*p* := *append-free-list* (´*mem-pool-info p*) (*nat* (´*from-l t* + *1*)) (´*block2 t*))
FI;;

´*i* := ´*i*(*t* := *Suc* (´*i t*)))
sat_p [{*V*}, {(*s*, *t*). *s* = *t*}, *UNIV*, *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p*]
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-while-precond5*
(*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-while-precond1 V t p*}])

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Cond*)
apply(*simp add*:*stable-def*)
apply(*rule Basic*)
using *mp-alloc-stm4-pre-in* **apply** *blast* **apply** *simp* **apply**(*simp add*:*stable-def*)
apply(*simp add*:*stable-def*)

apply(*unfold Skip-def*)
apply(*rule subst*[**where** *t*={*mp-alloc-stm4-while-precond4*
(*mp-alloc-stm4-while-precond3*
(*mp-alloc-stm4-while-precond2*
(*mp-alloc-stm4-while-precond1 V t p*) *t p*) *t p*) *t p*} ∩
− {|*block-fits* (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*)|} **and** *s*={}])
using *mp-alloc-stm4-blockfit*[*of Va t p timeout sz V*] **apply** *metis*
using *Emptyprecond* **apply** *metis*
apply *simp*

using *mp-alloc-stm4-while-isucc*[*of Va t p timeout sz V*] **apply** *fast*

done

lemma *mp-alloc-stm4-while-1*: $\{\!|4 \le \acute{}i\ t|\!\} = - \{\!|\acute{}i\ t < 4|\!\}$ **by** *auto*

term *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ $\{\!|\acute{}cur = Some\ t|\!\}$

lemma *mp-alloc-stm4-while*:
Va $\in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ $\{\!|\acute{}cur = Some\ t|\!\} \Longrightarrow$
$\Gamma \vdash_I$ *Some* (*WHILE ´i t < 4 DO*
*´lbn := ´lbn (t := 4 * ´bn t + ´i t);;*
´lsz := ´lsz (t := (´lsizes t) ! (nat (´from-l t + 1)));;
*´block2 := ´block2(t := ´lsz t * ´i t + ´blk t);;*

´mem-pool-info := set-bit-free ´mem-pool-info p (nat (´from-l t + 1)) (´lbn t);;

IF block-fits (´mem-pool-info p) (´block2 t) (´lsz t) THEN

´mem-pool-info := ´mem-pool-info (p :=
append-free-list (´mem-pool-info p) (nat (´from-l t + 1)) (´block2 t))
FI;;
´i := ´i(t := Suc (´i t))
OD) sat_p [*mp-alloc-stm4-loopinv (mp-alloc-stm4-pre-precond-f Va t p) t p*, $\{(s, t).\ s = t\}$, *UNIV*,
mp-alloc-stm4-loopinv (mp-alloc-stm4-pre-precond-f Va t p) t p $\cap$ $\{\!|\acute{}i\ t \ge 4|\!\}$]
apply(*rule While*)
apply(*simp add*:*stable-def*)
apply(*rule subst*[**where** *t*=$-\ \{\!|\acute{}i\ t < 4|\!\}$ **and** *s*=$\{\!|4 \le \acute{}i\ t|\!\}$]) **using** *mp-alloc-stm4-while-1*[*of t*] **apply** *simp*
apply *simp*
apply(*simp add*:*stable-def*)
using *mp-alloc-stm4-while-help*[*of Va t p timeout sz*]
Allprecond[*of mp-alloc-stm4-loopinv (mp-alloc-stm4-pre-precond-f Va t p) t p* $\cap\{\!|\acute{}i\ t < 4|\!\}$
*Some (´lbn := ´lbn(t := 4 * ´bn t + ´i t);;*
´lsz := ´lsz(t := ´lsizes t ! nat (´from-l t + 1));;
*´block2 := ´block2(t := ´lsz t * ´i t + ´blk t);;*
´mem-pool-info := set-bit-free ´mem-pool-info p (nat (´from-l t + 1)) (´lbn t);;
IF block-fits (´mem-pool-info p) (´block2 t)
(´lsz t) THEN ´mem-pool-info := ´mem-pool-info
(p := append-free-list (´mem-pool-info p) (nat (´from-l t + 1)) (´block2 t)) FI;;
´i := ´i(t := Suc (´i t))) $\{(s, t).\ s = t\}$ *UNIV*
mp-alloc-stm4-loopinv (mp-alloc-stm4-pre-precond-f Va t p) t p] **apply** *clarsimp*

apply *force*
done

lemma *mp-alloc-stm4-pre-precond-f-in-mp-alloc-stm4-loopinv*:
mp-alloc-stm4-pre-precond-f Va t p $\in$ *mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p*
apply(*subgoal-tac i* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *1*)
using *in-mp-alloc-stm4-loopinv* **apply** *meson*
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
done

lemma *mp-alloc-stm4-mempools*: (*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* $\Longrightarrow$ *mem-pools x* = *mem-pools Va*
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)

lemma *mp-alloc-stm4-mempools2*: *mem-pools* (*mp-alloc-stm4-pre-precond-f Va t p*) = *mem-pools x* $\Longrightarrow$ *mem-pools x* = *mem-pools Va*
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-inv-cur*:
cur Va = *Some t* $\Longrightarrow$ $\forall$ *ta*. (*t* = *ta*) = (*thd-state Va ta* = *RUNNING*) $\Longrightarrow$
(*cur* (*mp-alloc-stm4-pre-precond-f Va t p*) = *Some ta*) = (*thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) *ta* = *RUNNING*)
by (*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-inv-thd-state*: (*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* $\Longrightarrow$
thd-state x = *thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) $\Longrightarrow$
$\forall$ *pa*. *pa* $\neq$ *p* $\longrightarrow$ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* $\Longrightarrow$
wait-q (*mem-pool-info x p*) = *wait-q* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) $\Longrightarrow$
inv-thd-waitq Va $\Longrightarrow$ *inv-thd-waitq x*
apply(*subgoal-tac* $\forall$ *q*$\in$*mem-pools x*. *wait-q* (*mem-pool-info x q*)
= *wait-q* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *q*))
prefer *2* **apply** *clarify* **apply**(*case-tac q* = *p*) **apply** *simp* **apply** *simp*
apply(*subgoal-tac* $\forall$ *q*$\in$*mem-pools Va*. *wait-q* (*mem-pool-info Va q*)
= *wait-q* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *q*))
prefer *2* **apply** *clarify* **apply**(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)
apply(*subgoal-tac thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) = *thd-state Va*)
prefer *2* **apply** (*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*subgoal-tac mem-pools x* = *mem-pools Va*)
prefer *2* **apply** (*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*:*inv-thd-waitq-def*)
apply *clarify*

by *blast*

lemma *inv-mpinfo-inv-mpinfo-stm4*:
  inv-mempool-info Va $\Longrightarrow$ *inv-mempool-info* (*mp-alloc-stm4-pre-precond-f Va t p*)
  apply(*simp add*:*inv-mempool-info-def mp-alloc-stm4-pre-precond-f-def*)
  apply(*simp add*:*Let-def*) **apply** *clarify*
  apply(*rule conjI*)
    apply(*simp add*: *set-bit-def*) **apply** *auto*[*1*]
  apply(*rule conjI*)
    apply(*simp add*: *set-bit-def*)
    apply(*rule conjI*) **apply** *clarify* **apply** *auto*[*1*] **apply** *clarify* **apply** *auto*[*1*]
  apply(*rule conjI*) **apply**(*simp add*: *set-bit-def*) **apply** *auto*[*1*]
  apply(*rule conjI*) **apply**(*simp add*: *set-bit-def*) **apply** *auto*[*1*]
  apply(*rule conjI*) **apply**(*simp add*: *set-bit-def*) **apply** *auto*[*1*]
  apply *clarify* **apply**(*rename-tac pa ii*)
  apply(*subgoal-tac length* (*bits* (*levels* ((*set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*)))) *pa*) !
ii)) = *length* (*bits* (*levels* (*mem-pool-info Va pa*) ! *ii*)))
    prefer *2* **apply** (*metis* (*no-types*, *lifting*) *fun-upd-apply set-bit-def*
      set-bit-prev-len set-bit-prev-len2)
  apply(*subgoal-tac length* (*bits* (*levels* (*set-bit-allocating*
(*set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*))))
p (*nat* (*from-l Va t* + *1*))
(*4* $*$ *block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*))) *pa*) !
ii)) = *length* (*bits* (*levels* ((*set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*)))) *pa*) !
ii)))
    prefer *2* **apply**(*case-tac ii* = *nat* (*from-l Va t* + *1*))
      using *set-bit-prev-len set-bit-def* **apply** *auto*[*1*]
      using *set-bit-def* **apply** *auto*[*1*]
  apply(*subgoal-tac n-max* (*set-bit-allocating*
(*set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))
p (*nat* (*from-l Va t* + *1*)) (*4* $*$ *block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *pa*) $*$
4 ^ *ii* = *n-max* (*mem-pool-info Va pa*) $*$ *4* ^ *ii*)
    prefer *2* **apply**(*simp add*: *set-bit-def*)
  apply(*subgoal-tac length* (*levels* (*set-bit-allocating*
(*set-bit-divide* (*mem-pool-info Va*) *p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*))))

p (*nat* (*from-l Va t* + *1*))
(*4* ∗ *block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *pa*))
= *length* (*levels* (*mem-pool-info Va pa*)))
prefer *2* **apply**(*simp add*: *set-bit-def*)
apply *metis*
done

lemma *mp-alloc-stm4-inv-mempool-info*:
(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
inv-mempool-info Va ⟹ *inv-mempool-info x*
apply(*subgoal-tac inv-mempool-info* (*mp-alloc-stm4-pre-precond-f Va t p*))
prefer *2* **using** *inv-mpinfo-inv-mpinfo-stm4* **apply** *simp*
using *gvars-conf-stb-inv-mpinf* **apply** *simp*
done

lemma *mp-alloc-stm4-lvl-len*:
p ∈ *mem-pools Va* ⟹ (*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
length (*levels* (*mem-pool-info x pa*)) = *length* (*levels* (*mem-pool-info Va pa*))
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-maxsz*:
p ∈ *mem-pools Va* ⟹ (*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
max-sz (*mem-pool-info x pa*) = *max-sz* (*mem-pool-info Va pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-buf*:
p ∈ *mem-pools Va* ⟹ (*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
buf (*mem-pool-info x pa*) = *buf* (*mem-pool-info Va pa*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
apply(*simp add*: *set-bit-def*)
done

lemma *mp-alloc-stm4-pres-mpinfo*:
pa ≠ *p* ⟶ *mem-pool-info Va pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def set-bit-def*)
done

lemma *mp-alloc-stm4-froml*:
from-l x = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) ⟹

from-l x = from-l Va
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-stm4-pre-precond-f-lvars-nochange*:
$t' \neq t \Longrightarrow$ *lvars-nochange* t' *Va* (*mp-alloc-stm4-pre-precond-f Va t p*)
apply(*simp add*:*lvars-nochange-def mp-alloc-stm4-pre-precond-f-def*)
done

lemma *mp-alloc-stm4-pre-precond-f-tick*:
tick Va = *tick* (*mp-alloc-stm4-pre-precond-f Va t p*)
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-def-frnode*:
freeing-node Va = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*)
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-mpls*:
$p \in$ *mem-pools Va* $\Longrightarrow$ (x, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable*
$\Longrightarrow p \in$ *mem-pools x*
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def gvars-conf-stable-def gvars-conf-def*)
done

lemma *mp-alloc-stm4-pre-precond-f-rf*:
rf (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $\Longrightarrow$ *rf Va t*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-ret*:
mempoolalloc-ret Va = *mempoolalloc-ret* (*mp-alloc-stm4-pre-precond-f Va t p*)
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-tmout*:
tmout Va = *tmout* (*mp-alloc-stm4-pre-precond-f Va t p*)
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-lsz*:
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) = *lsizes Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-allocl*:
alloc-l (*mp-alloc-stm4-pre-precond-f Va t p*) = *alloc-l Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-froml*:
from-l (*mp-alloc-stm4-pre-precond-f Va t p*) = *from-l Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-freel*:
free-l (*mp-alloc-stm4-pre-precond-f Va t p*) = *free-l Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *mp-alloc-stm4-pre-precond-f-blk*:
blk (*mp-alloc-stm4-pre-precond-f Va t p*) = *blk Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *same-level-set-bit-l*:$i1 \neq i' \Longrightarrow$
levels ((*set-bit mp-info p i' j' b*) *p*)!*i1* = *levels* (*mp-info p*)!*i1*
unfolding *set-bit-def*
by *auto*

lemma *same-bit-set-bit-l*:$i1 \neq i' \Longrightarrow$
bits (*levels* ((*set-bit mp-info p i' j' b*) *p*)!*i1*) = *bits* (*levels* (*mp-info p*)!*i1*)
using *same-level-set-bit-l*
by *auto*

lemma *same-bit-set-bit-j*:
$j1 \neq j' \Longrightarrow$
bits (*levels* ((*set-bit mp-info p i' j' b*) *p*)!*i1*)!*j1* = *bits* (*levels* (*mp-info p*)!*i1*)!*j1*
apply(*simp add*: *set-bit-get-bit-neq set-bit-def*)
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.surjective*
Mem-pool-lvl.update-convs(*1*)
list-update-beyond not-less nth-list-update-eq nth-list-update-neq)

lemma *set-bit-set-bit*:
$l1 \neq l2 \lor b1 \neq b2 \Longrightarrow$
set-bit-s (*set-bit-s Va p l1 b1 st1*) *p l2 b2 st2* =
set-bit-s ((*set-bit-s Va p l2 b2 st2*)) *p l1 b1 st1*
unfolding *set-bit-s-def set-bit-def*
apply *auto*
apply (*cases b1=b2*) **apply** *auto*
apply (*simp add*: *list-update-swap*)
apply (*simp add*: *list-update-swap*)
apply (*cases l1=l2*) **apply** *auto*
apply (*cases l1*< *length* (*levels* (*mem-pool-info Va p*)))
by (*auto simp add*: *list-update-swap*)

lemma *get-bit-set-bit-set-bit*:
assumes *a0*:$l1 \neq l2 \lor b1 \neq b2$ **and**
a1:*l1* < *length* (*levels* ((*mem-pool-info Va*) *p*)) **and**
a2:*b1* < *length* (*bits* (*levels* ((*mem-pool-info Va*) *p*) ! *l1*))
shows *get-bit-s* (*set-bit-s* (*set-bit-s Va p l1 b1 st1*) *p l2 b2 st2*) *p l1 b1* = *st1*
proof−
have *a1'*:*l1* < *length* (*levels* ((*mem-pool-info* (*set-bit-s Va p l2 b2 st2*)) *p*))
using *a1* **unfolding** *set-bit-s-def set-bit-def* **by** *auto*
have *a2'*:*b1* < *length* (*bits* (*levels* ((*mem-pool-info* (*set-bit-s Va p l2 b2 st2*)) *p*)

! *l1*))
  using *a2* **unfolding** *set-bit-s-def set-bit-def* **apply** *auto*
  by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.select-convs*(*1*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*1*) *a1*
    length-list-update nth-list-update-eq nth-list-update-neq)
  show *?thesis* **using** *set-bit-get-bit-eq2*[*OF a1′ a2′*] *set-bit-set-bit*[*OF a0*] **unfolding** *set-bit-s-def* **by** *auto*
qed

lemma *mp-alloc-stm4-pre-precond-f-same-bits*:**assumes**
  a0:*i1*=(*nat* (*from-l Va t*)) **and**
  a1:*i2* = (*nat* (*from-l Va t* + *1*)) **and**
  a2:*Va′* = *mp-alloc-stm4-pre-precond-f Va t p*
  shows ∀ *i j*. ¬((*i*=*i1*) ∨(*i*=*i2*)) ⟶
    get-bit (*mem-pool-info Va′*) *p i j* = *get-bit* (*mem-pool-info Va*) *p i j*
  using *a0 a1 a2 set-bit-get-bit-neq* **unfolding** *mp-alloc-stm4-pre-precond-f-def*
  by *auto*

lemma *same-bit-mp-alloc-stm4-pre-precond-f*:
  i1=(*nat* (*from-l Va t*)) ⟹
  j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) ⟹
  i2 = (*nat* (*from-l Va t* + *1*)) ⟹
  j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) ⟹
  Va′ = *mp-alloc-stm4-pre-precond-f Va t p* ⟹
  ∀ *i j*. ¬((*i*=*i1* ∧ *j*=*j1*) ∨(*i*=*i2* ∧ *j*=*j2*)) ⟶
    get-bit (*mem-pool-info Va′*) *p i j* = *get-bit* (*mem-pool-info Va*) *p i j*
  using *set-bit-get-bit-neq*
  apply (*auto simp add*:*mp-alloc-stm4-pre-precond-f-def*)
  by *metis*+

lemma *same-bit-mp-alloc-stm4-pre-precond-f1*:
  assumes
  a1:¬((*l*=(*nat* (*from-l Va t*)) ∧ *b*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) ∨
    (*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=(*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))))
  shows (*get-bit-s Va p l b* = *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b*)
  using *a1 same-bit-mp-alloc-stm4-pre-precond-f* **by** *metis*

lemma *same-bit-mp-alloc-stm4-pre-precond-f11*:
  assumes *a0*:(*l*=(*nat* (*from-l Va t*)) ∧ *b*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
    a1:*l* ≥ *length* (*levels* (*mem-pool-info Va p*)) ∨
      b≥*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
  shows *get-bit-s Va p l b* = *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*

l b
using *a0 a1* **unfolding** *mp-alloc-stm4-pre-precond-f-def set-bit-def*
apply *auto*
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective list-update-beyond not-less nth-list-update-eq nth-list-update-neq*)

lemma *same-bit-mp-alloc-stm4-pre-precond-f12*:
assumes *a0*:(*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
a1:*l* ≥ *length* (*levels* (*mem-pool-info Va p*)) ∨
b≥*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
shows *get-bit-s Va p l b* = *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b*
using *a0 a1* **unfolding** *mp-alloc-stm4-pre-precond-f-def set-bit-def*
apply *auto*
apply (*metis list-update-beyond nth-list-update-neq*)
by (*smt Mem-pool-lvl.simps*(*1*) *Mem-pool-lvl.simps*(*4*) *Mem-pool-lvl.surjective length-list-update list-update-beyond*
not-less nth-list-update-eq nth-list-update-neq)

lemma *same-bit-mp-alloc-stm4-pre-precond-f2*:
assumes *a1*:*l* ≥ *length* (*levels* (*mem-pool-info Va p*)) ∨
b≥*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
shows *get-bit-s Va p l b* = *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b*
apply (*cases* ¬ ((*l*=(*nat* (*from-l Va t* + *1*)) ∧
b=*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) ∨
(*l*=(*nat* (*from-l Va t*)) ∧ *b*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))))
using *same-bit-mp-alloc-stm4-pre-precond-f1* **apply** *fast*
using *a1* **by** (*auto intro*: *same-bit-mp-alloc-stm4-pre-precond-f11 same-bit-mp-alloc-stm4-pre-precond-f12*)

lemma *same-bit-mp-alloc-stm4-pre-precond-divided*:
assumes *a0*:(*l*=(*nat* (*from-l Va t*)) ∧ *b*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
a1:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
a2:*b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*)) **and**
a3:(*from-l Va t*) ≥ *0*
shows *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b* = *DIVIDED*
proof−
have *l* ≠ *nat* (*from-l Va t* + *1*) ∨ *b* ≠ *4*∗*b* **using** *a0 a3* **by** *fastforce*
then show *?thesis* **using** *a0 a1 a2 a3 set-bit-get-bit-eq2*
unfolding *mp-alloc-stm4-pre-precond-f-def*
using *set-bit-get-bit-neq* **by** *auto*
qed

lemma *same-bit-mp-alloc-stm4-pre-precond-allocating*:
assumes *a0*:(*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=*4*∗(*block-num* (*mem-pool-info Va*

p) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
a1:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
a2:*b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*)) **and**
a3:(*from-l Va t*) $\geq$ *0*
shows *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b* = *ALLOCATING*
proof−
let *?Va* = *set-bit-s Va p* (*nat* (*from-l Va t*))
(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *DIVIDED*
have *a1'*:*l* < *length* (*levels* (*mem-pool-info ?Va p*))
using *a1* **unfolding** *set-bit-s-def set-bit-def* **by** *auto*
moreover have *a2'*:*b*<*length* (*bits* (*levels* (*mem-pool-info ?Va p*) ! *l*))
using *a2* **unfolding** *set-bit-s-def set-bit-def*
by (*simp add*: *a0 a3 eq-nat-nat-iff*)
ultimately show *?thesis*
using *a0 set-bit-get-bit-eq2 a3*
unfolding *mp-alloc-stm4-pre-precond-f-def set-bit-s-def*
using *set-bit-get-bit-eq* **by** *auto*
qed

lemma *eq-free-list-set-bit-l*:
free-list (*levels* ((*set-bit mp-info p i' j' b*) *p*)!*i1*) = *free-list* (*levels* (*mp-info p*)!*i1*)
unfolding *set-bit-def*
apply (*cases i'*< *length* (*levels* (*mp-info p*)); *auto*)
by (*metis* (*no-types*, *lifting*) *Mem-pool-lvl.select-convs*(*2*) *Mem-pool-lvl.surjective Mem-pool-lvl.update-convs*(*1*) *nth-list-update-eq nth-list-update-neq*)

lemma *eq-free-list-mp-alloc-stm4-pre-precond-f*:
free-list (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *l*) =
free-list (*levels* (*mem-pool-info Va p*) ! *l*)
unfolding *mp-alloc-stm4-pre-precond-f-def* **using** *eq-free-list-set-bit-l*
by *auto*

lemma *mp-alloc-stm4-pre-precond-f-i*:(*i* (*mp-alloc-stm4-pre-precond-f Va t p*)) *t* = *Suc 0* $\wedge$
($\forall$ *t'*. *t'*$\neq$*t* $\longrightarrow$ (*i* (*mp-alloc-stm4-pre-precond-f Va t p*)) *t'* = (*i Va*) *t'*)
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *force*

lemma *mp-alloc-stm4-pre-precond-f-bn*:
(*bn* (*mp-alloc-stm4-pre-precond-f Va t p*)) *t* =
block-num (*mem-pool-info Va p*) (*blk Va t*) ((*lsizes Va t*)!(*nat* (*from-l Va t*)))
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *force*

lemma *mp-alloc-stm4-pre-precond-f-allocating*:
(*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*)) *t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t* + *1*),
block = *4* ∗ *block-num* (*mem-pool-info Va p*) (*blk Va t*) ((*lsizes Va t*)!(*nat*

(*from-l Va t*))),
        data = *blk Va t* ⦈
  unfolding *mp-alloc-stm4-pre-precond-f-def*
  by *auto*

lemma *get-bit-x-l-b*:
    assumes *a0*:(*l*=(*nat* (*from-l* (*Va*::*State*) *t*)) ∧ *b*=(*block-num* (*mem-pool-info* *Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
      a1:(*from-l Va t*) ≥ *0* **and**
      a2:∀ *jj*. *jj* ≠ *nat* (*from-l Va t* + *1*) ⟶
         levels (*mem-pool-info x p*) ! *jj* =
         levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
       a4:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
       a5:*b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
     shows *get-bit-s x p l b* = *DIVIDED*
  using *a0 a2 a1 a4 a5 same-bit-mp-alloc-stm4-pre-precond-divided* **by** *auto*

lemma *get-bit-x-l1-b4*:
   assumes *a0*:(*l*=(*nat* (*from-l* (*Va*::*State*) *t* + *1*)) ∧ *b*=*4*∗(*block-num* (*mem-pool-info* *Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
      a1:(*from-l Va t*) ≥ *0* **and**
      a3:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f* *Va t p*) *t* + *1*)) =
        list-updates-n
          (*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
               nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
          (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
       a4:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
       a5:*b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
     shows *get-bit-s x p l b* = *ALLOCATING*
  using *a0 a1 a3 a4 a5 same-bit-mp-alloc-stm4-pre-precond-allocating*
       mp-alloc-stm4-pre-precond-f-bn
       mp-alloc-stm4-pre-froml
  by (*metis lessI list-updates-n-neq mult.commute*)

lemma *get-bit-x-l1-b41*:
  assumes *a0*:*l*=(*nat* (*from-l* (*Va*::*State*) *t* + *1*)) ∧
              (*b*=*Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) ∨
              b=*Suc*(*Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va* *t* ! *nat* (*from-l Va t*))))) ∨
             b=*Suc*(*Suc*(*Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes* *Va t* ! *nat* (*from-l Va t*))))))) **and**
      a1:(*from-l Va t*) ≥ *0* **and**
      a2:∀ *jj*. *jj* ≠ *nat* (*from-l Va t* + *1*) ⟶
         levels (*mem-pool-info x p*) ! *jj* =
         levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**

a3:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a4:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
a5: *b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
shows *get-bit-s x p l b* = *FREE*
using *a0 a1 a3 a4 a5*
apply *auto*
using *mp-alloc-stm4-pre-precond-f-bn*
mp-alloc-stm4-pre-froml mp-alloc-stm4-inv-bits-len Suc-numeral add-2-eq-Suc
add-Suc-right
by (*smt add.commute lessI less-add-same-cancel2*
list-updates-n-eq mult.commute nat-less-le neq0-conv not-le semiring-norm(*5*))+

lemma *get-bit-x-l1-b41′*:
assumes *a0*:*l*=(*nat* (*from-l* (*Va*::*State*) *t* + *1*)) ∧
(*b*=(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*
(*from-l Va t*)))) + *1* ∨
b=(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*
(*from-l Va t*))))+*2* ∨
b=(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*
(*from-l Va t*))))+*3*) **and**
a1:(*from-l Va t*) ≥ *0* **and**
a2:∀ *jj*. *jj* ≠ *nat* (*from-l Va t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* =
levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a3:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a4:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
a5: *b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*))
shows *get-bit-s x p l b* = *FREE*
using *a0 a1 a3 a4 a5*
apply *auto*
using *mp-alloc-stm4-pre-precond-f-bn*
mp-alloc-stm4-pre-froml mp-alloc-stm4-inv-bits-len
apply (*metis add.right-neutral less-not-refl list-updates-n-eq mult.commute nat-add-left-cancel-less*
not-less zero-less-numeral)
by (*smt add.commute add-Suc less-Suc-eq less-add-same-cancel2 list-updates-n-eq*
mp-alloc-stm4-inv-bits-len mp-alloc-stm4-pre-froml mp-alloc-stm4-pre-precond-f-bn
mult.commute nat-less-le numeral-3-eq-3) +

lemma *get-bit-x-stm4-pre-eq*:

assumes
a0:∀ *jj*. *jj* ≠ *nat* (*from-l* (*Va*::*State*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* =
levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a1:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a2:*l* = *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) **and**
a3: *b1* = (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) **and**
a4:*b2* = *Suc* (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) **and**
a5:*b3* = *Suc* (*Suc* (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)))
shows ∀ *i j*. ¬((*i*=*l* ∧ *j*=*b1*) ∨(*i*=*l* ∧ *j*=*b2*) ∨ (*i*=*l* ∧ *j*=*b3*)) ⟶
get-bit-s x p i j = *get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p i j*
using *a0 a1 a2 a3 a4 a5* **apply** *clarsimp*
apply (*auto simp add*: *mp-alloc-stm4-pre-precond-f-froml*)
by (*metis* (*no-types*) *add-2-eq-Suc′ add-Suc-right eval-nat-numeral*(*3*)
less-Suc-eq list-updates-n-neq not-less)

lemma *same-bit-mp-alloc-stm4-pre-precond-f-x*:
assumes *a0*:∀ *jj*. *jj* ≠ *nat* (*from-l* (*Va*::*State*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* =
levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a1:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a2:*i1*=(*nat* (*from-l Va t*)) **and**
a3:*j1*= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
a4:*i2* = *nat* (*from-l Va t* + *1*) **and**
a5:*j2* = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
a6:*j3* = *Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) **and**
a7:*j4* = *Suc*(*Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))) **and**
a8:*j5* = *Suc*(*Suc*(*Suc*(*4*∗(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))))
shows ∀ *i j*. ¬((*i*=*i1* ∧ *j*=*j1*) ∨(*i*=*i2* ∧ *j*=*j2*) ∨ (*i*=*i2* ∧ *j*=*j3*)∨ (*i*=*i2* ∧ *j*=*j4*)∨ (*i*=*i2* ∧ *j*=*j5*)) ⟶
get-bit-s x p i j = *get-bit-s Va p i j*
using *a0 a1 a2 a4 a4 a5 a6 a7 a8 get-bit-x-stm4-pre-eq*
same-bit-mp-alloc-stm4-pre-precond-f

proof−
 {**fix** *i j*
  assume *a00*:¬((*i*=*i1* ∧ *j*=*j1*) ∨(*i*=*i2* ∧ *j*=*j2*) ∨ (*i*=*i2* ∧ *j*=*j3*)∨ (*i*=*i2* ∧ *j*=*j4*)∨ (*i*=*i2* ∧ *j*=*j5*))
  then have *get-bit-s Va p i j* =
   get-bit-s (*mp-alloc-stm4-pre-precond-f Va t p*) *p i j*
  using *same-bit-mp-alloc-stm4-pre-precond-f1 a2 a3 a4 a5*
  by *auto*
  also have *get-bit-s x p i j* =
   get-bit-s (*mp-alloc-stm4-pre-precond-f Va t p*) *p i j*
  using *a00 a0 a1 a2 a4 a4 a5 a6 a7 a8 get-bit-x-stm4-pre-eq*[*OF a0 a1*]
   mp-alloc-stm4-pre-precond-f-bn
  by (*metis mult.commute*)
  finally have *get-bit-s x p i j* = *get-bit-s Va p i j* **.**
 } **thus** *?thesis* **by** *fastforce*
qed

lemma *same-bit-mp-alloc-x-va*:
 assumes
  a0:∀ *jj*. *jj* ≠ *nat* (*from-l Va t* + *1*) ⟶
   levels (*mem-pool-info x p*) ! *jj* =
   levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
  a1: *bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
   list-updates-n
    (*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
     nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
    (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
  a2:¬((*l*=(*nat* (*from-l Va t*)) ∧ *b*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) ∨
   (*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=(*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))) ∨
   (*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=*Suc*((*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))) ∨
   (*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=*Suc*(*Suc*((*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))))∨
   (*l*=(*nat* (*from-l Va t* + *1*)) ∧ *b*=*Suc*(*Suc*(*Suc*((*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))))))))
  shows (*get-bit-s x p l b* = *get-bit-s Va p l b*)
 using *same-bit-mp-alloc-stm4-pre-precond-f-x*[*OF a0 a1*] *a2*
 by *auto*

lemma *free-list-x*:
 assumes *a0*:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
 inserts

(*map* (*λii. lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
shows
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) = (*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))@
[*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
1 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
2 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
3 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*]
using *a0*
by (*simp add*: *numeral-3-eq-3 numeral-Bit0 inserts-def*)
lemma *listx1*:
jj = *length* (*free-list* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) ⟹
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
(*free-list* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))@
[*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *1* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *2* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *3* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*] ⟹
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))! *jj* =
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *1* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*

by *auto*

lemma *listx3*:
jj = *Suc*(*Suc* (*length* (*free-list* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))))) ⟹
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va*
t p) *t* + *1*)) =
(*free-list* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))@
[*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *1* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *2* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *3* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*] ⟹
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va*
t p) *t* + *1*))! *jj* =
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *3* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*
by (*simp add*: *nth-append*)

lemma *set-free-x-va*: **assumes** *a0*:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l*
(*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
shows *set* (*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l Va t* + *1*))) =
set (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*from-l Va t* + *1*))) ∪
{*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
1 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,*lsizes* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
2 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,

lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
3 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*}
proof−
have *free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l Va t* + *1*)) =
(*free-list*
(*levels* (*mem-pool-info Va p*) !
nat (*from-l Va t* + *1*)))@
[*lsizes Va t* !
nat (*from-l Va t* + *1*) ∗
1 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l Va t* + *1*) ∗
2 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes Va t !
nat (*from-l Va t* + *1*) ∗
3 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*]
using *free-list-x*[*OF a0*]
by (*metis eq-free-list-mp-alloc-stm4-pre-precond-f mp-alloc-stm4-pre-froml mp-alloc-stm4-pre-precond-f-lsz*)
then show *?thesis* **using** *mp-alloc-stm4-pre-froml mp-alloc-stm4-pre-precond-f-lsz*
by (*metis empty-set list.simps*(*15*) *set-append*)
qed

lemma *free-list-x-subset-va*:
assumes *a0*:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (*λii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
shows *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *nat* (*from-l Va t* + *1*))) ⊆
set (*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l Va t* + *1*)))
proof−
have *free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l Va t* + *1*)) =
(*free-list*
(*levels* (*mem-pool-info Va p*) !
nat (*from-l Va t* + *1*)))@
[*lsizes Va t* !
nat (*from-l Va t* + *1*) ∗
1 +

blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l Va t* + *1*) ∗
2 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes Va t !
nat (*from-l Va t* + *1*) ∗
3 +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*]
using *free-list-x*[*OF a0*]
by (*metis eq-free-list-mp-alloc-stm4-pre-precond-f mp-alloc-stm4-pre-froml mp-alloc-stm4-pre-precond-f-lsz*)
then show *?thesis* **by** *auto*
qed

lemma *free-level-x-va*:
assumes
a0:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* =
levels (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj*
shows ∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
free-list (*levels* (*mem-pool-info x p*) !*jj*) = *free-list* (*levels* (*mem-pool-info Va p*)! *jj*)
by (*simp add*: *assms eq-free-list-mp-alloc-stm4-pre-precond-f*)

lemma *mp-alloc-stm4-pre-precond-f-bitmap-not-free*:
assumes *a0*:(*get-bit-s Va p l b* ≠ *FREE*) **and**
a1:*l* < *length* (*levels* (*mem-pool-info Va p*)) **and**
a2:*b*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *l*)) **and**
a3:(*from-l Va t*) ≥ *0*
shows (*get-bit-s* (*mp-alloc-stm4-pre-precond-f Va t p*) *p l b* ≠ *FREE*)
using *a0 a1 a2 a3 same-bit-mp-alloc-stm4-pre-precond-divided same-bit-mp-alloc-stm4-pre-precond-allocating*
same-bit-mp-alloc-stm4-pre-precond-f1 BlockState.distinct(*11*) *BlockState.distinct*(*17*)
proof−
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))
have *i1orj1*:*?i1* ≠ *?i2* ∨ *?j1* ≠ *?j2* **using** *a3* **by** *auto*
{**assume** *a00*:¬((*l*=*?i1* ∧ *b*=*?j1*) ∨(*l*=*?i2* ∧ *b*=*?j2*))
then have *?thesis* **using** *same-bit-mp-alloc-stm4-pre-precond-f1 a0*
by *auto*
}
moreover {**assume** *a00*:(*l*=*?i1* ∧ *b*=*?j1*)
have *?thesis*
using *same-bit-mp-alloc-stm4-pre-precond-divided*[*OF a00 a1 a2 a3*]
by *auto*
}

moreover {**assume** *a00*:(*l*=*?i2* ∧ *b*=*?j2*)
have *?thesis*
using *same-bit-mp-alloc-stm4-pre-precond-allocating*[*OF a00 a1 a2 a3*]
by *auto*
}
ultimately show *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-pre-inv-nmax*: *n-max* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa*) ∗ *4* ˆ *ii* =
n-max (*mem-pool-info Va pa*) ∗ *4* ˆ *ii*
unfolding *mp-alloc-stm4-pre-precond-f-def set-bit-def*
by *auto*

lemma *allocating-next-notexists*:*inv-bitmap Va* ⟹
p∈*mem-pools Va* ⟹
ii < *length* (*levels* (*mem-pool-info Va p*)) ⟹
jj < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)) ⟹
get-bit-s Va p ii jj = *ALLOCATING* ⟹
ii < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii* + *1*) (*jj* ∗ *4*)
unfolding *inv-bitmap-def inv-mempool-info-def Let-def*
by *auto*

lemma *block-n*:
assumes
a0:*lsizes Va t* ! *nat* (*from-l Va t*) = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *nat* (*from-l Va t*) **and**
a1:*p* ∈ *mem-pools Va* **and**
a2:*inv-mempool-info Va* **and**
a3:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:*from-l Va t* < *alloc-l Va t* **and**
a6:*OK* ≤ *from-l Va t*
shows (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) = *n*
proof−
have (∃ *n*>*NULL*. *max-sz* (*mem-pool-info Va p*) = *4* ∗ *n* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*)) ∧
NULL < *n-max* (*mem-pool-info Va p*) ∧
NULL < *n-levels* (*mem-pool-info Va p*) ∧ *n-levels* (*mem-pool-info Va p*) = *length* (*levels* (*mem-pool-info Va p*))
using *a2 a1* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
then show *?thesis* **using** *assms mp-alloc-stm3-lm2-inv-1-2 inv-mempool-info-maxsz-align4*[*OF*

a2] *nat-less-iff*
  unfolding *block-num-def Let-def* **apply** *auto*
  by (*smt less-numeral-extra*(*3*))
qed

definition *addr*::*nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *addr m-size init l n* ≡ *init* + *n* ∗(*m-size div 4* ˆ *l*)

definition *next-addr* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where *next-addr m-size c-addr l n* ≡ (*m-size div 4* ˆ (*l* + *1*))∗*n* + *c-addr*

lemma *next-level-addr*:
  assumes
    a0:∃ *m*. *m-size* = *m*∗*4*ˆ *p* **and**
    a1:*p* > *l*+*1*
 shows *next-addr m-size* (*addr m-size init l n*) *l ch* = *addr m-size init* (*l*+*1*) (*n*∗*4* + *ch*)
 unfolding *next-addr-def addr-def*
  proof(*induct ch*)
    case *0*
    then show *?case*
      apply *auto*
       by (*smt One-nat-def a0 a1 dvd-mult-div-cancel dvd-triv-right less-imp-le-nat mult.commute mult.left-commute nonzero-mult-div-cancel-left power-Suc0-right power-add power-le-dvd power-not-zero zero-neq-numeral*)
  next
    case (*Suc ch*)
    obtain *m* **where** *m-size* = *m*∗(*4*::*nat*) ˆ *p* **using** *a0* **by** *auto*
    then show *?case* **using** *Suc a1* **by** *auto*
  qed

lemma *next-level-addr-eq1*:
  assumes
    a0:∃ *m*. *m-size* = *m*∗*4*ˆ *p* **and**
    a1:*p* > *l*+*1*
 shows *next-addr m-size* (*addr m-size init l n*) *l 0* = *addr m-size init l n*
  using *next-level-addr*[*OF a0 a1*] **unfolding** *next-addr-def addr-def*
  by *linarith*

lemma *next-level-addr-eq*:
  assumes
    a0:∃ *m*. *m-size* = *m*∗*4*ˆ *p* **and**
    a1:*p* > *l*+*1*
 shows *addr m-size init* (*l* + *1*) (*n* ∗ *4*) = *addr m-size init l n*
  using *next-level-addr*[*OF a0 a1*] *next-level-addr-eq1*[*OF a0 a1*]
  by *auto*

lemma *diff-n-m-addr*: **assumes** *a0*:$n\neq m$ **and** *a1*:*m-size* $\geq$ *4*ˆ*l*
shows *addr m-size init l n* $\neq$ *addr m-size init l m*
using *a0 a1* **unfolding** *addr-def*
by (*auto simp add*: *Euclidean-Division.div-eq-0-iff*)

lemma *lsizes-addr*:
assumes *a0*:*p* $\in$ *mem-pools Va* **and**
a1:$\forall$ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a2:*length* (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) **and**
a3:*inv-aux-vars Va* $\wedge$ *inv-bitmap Va* $\wedge$ *inv-mempool-info Va* $\wedge$ *inv-bitmap-freelist Va* **and**
a4:*l*+*1* <*length* (*lsizes Va t*)
shows $\forall$ *j*. (*lsizes Va t* ! (*l*+*1*)) ∗ *j* +
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *l*)) =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*l* + *1*)
((*block-num* (*mem-pool-info Va p*) (*buf* (*mem-pool-info Va p*) +
n ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *l*))
(*lsizes Va t* ! *l*))∗*4* + *j*)
proof−
{**fix** *j*
let *?blk* = (*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *l*))
obtain *m* **where** *max-sz*:*max-sz* (*mem-pool-info Va p*) = *4* ∗ *m* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*)
using *a3 a0* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
have *b1*:*?blk* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) *l*
(*block-num* (*mem-pool-info Va p*) *?blk* (*lsizes Va t* ! *l*))
using *a4 a0 a1 a3*
unfolding *addr-def block-num-def* **apply** *auto*
by (*metis add.commute add-lessD1 div-mult-self-is-m*
inv-mempool-info-maxsz-align4 plus-1-eq-Suc)
have *b2*:(*lsizes Va t* ! (*l*+*1*)) ∗ *j* + *?blk* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*l*+*1*)
((*block-num* (*mem-pool-info Va p*) *?blk* (*lsizes Va t* ! *l*))∗*4* + *j*)
using *assms a4 inv-mempool-info-maxsz-align4 max-sz b1 next-level-addr*
unfolding *next-addr-def*
by (*smt le-eq-less-or-eq le-less-trans*)
}**thus** *?thesis* **by** *auto*
qed

lemma *free-list-updates-inv1*:
assumes *a0*:*p* $\in$ *mem-pools Va* **and**
a1:$\neg$ *free-l Va t* < *OK* **and**

a2:*free-l Va t* $\leq$ *from-l Va t* **and**
a3:*alloc-l Va t* $<$ *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a4:*from-l Va t* $<$ *alloc-l Va t* **and**
a5:*alloc-l Va t* $=$ *int* (*length* (*lsizes Va t*)) $-$ *1* $\wedge$ *length* (*lsizes Va t*) $=$ *n-levels* (*mem-pool-info Va p*) $\vee$
alloc-l Va t $=$ *int* (*length* (*lsizes Va t*)) $-$ *2* $\wedge$ *lsizes Va t* ! *nat* (*alloc-l Va t* $+$ *1*) $<$ *sz* **and**
a6:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) $+$ *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
$<$ *n-max* (*mem-pool-info Va p*) $*$ *4* ^ *nat* (*from-l Va t*) **and**
a7:*blk Va t* $=$ *buf* (*mem-pool-info Va p*) $+$ *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) **and**
a8:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* **and**
a9:*from-l x* $=$ *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a10:*freeing-node x* $=$ *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a11:*allocating-node x* $=$ *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a12:$\forall$ *pa*. *pa* $\neq$ *p* $\longrightarrow$ *mem-pool-info x pa* $=$ *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a13:$\forall$ *jj*. *jj* $\neq$ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*) $\longrightarrow$
levels (*mem-pool-info x p*) ! *jj* $=$ *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a14:$\forall$ *ii*$<$*length* (*lsizes Va t*). *lsizes Va t* ! *ii* $=$ *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* **and**
a15:*i x t* $=$ *4* **and**
a16:*lsizes x* $=$ *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a17:*length* (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) **and**
a18:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)) $=$
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $*$ *4*)) *3 FREE* **and**
a19:
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)) $=$
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*) $*$
ii $+$
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..$<$*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*))) **and**
a20:*allocating-node Va t* $=$
Some (|*pool* $=$ *p*, *level* $=$ *nat* (*from-l Va t*),
block $=$ *block-num* (*mem-pool-info Va p*)

(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))⦈ **and**
a21:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* **and**
a22:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a23:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a24:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows ∀ *j*<*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)).
(*get-bit-s x p ii j* = *FREE*) =
(*buf* (*mem-pool-info x p*) + *j* ∗ (*max-sz* (*mem-pool-info x p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*)))
proof−
{
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))
fix *j*
let *?mp* = *mem-pool-info x p*
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
assume *a00*:*j*<*length ?bts*
then have *a00′*:*j* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *mp-alloc-stm4-inv-bits-len*
by (*metis a13 a18 length-list-update-n*)
have *inv-bitmap1*:(∀ *j*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)).
(*get-bit-s Va p ii j* = *FREE*) =
(*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))))
using *a21 a0 a22 mp-alloc-stm4-lvl-len*[*OF a0 a8*]
unfolding *Let-def inv-bitmap-freelist-def*
by *fastforce*+
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a1 a2* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
using *mp-alloc-stm4-lvl-len*[*OF a0 a8*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a0 a8*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a0 a8*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a9*] **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a8*] **by** *auto*

have *lsizes-x-va*:*lsizes x* = *lsizes Va*
by (*simp add*: *a16 mp-alloc-stm4-pre-precond-f-lsz*)
have *len-eq*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a22 a8 mp-alloc-stm4-inv-bits-len*
unfolding *gvars-conf-stable-def gvars-conf-def*
by *fastforce*
then have *get-bits-va*:(*get-bit-s Va p ii j* = *FREE*) =
(*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4*
ˆ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
using *inv-bitmap1 a00* **by** *auto*
have *inv-mempool-info-mp Va p*
using *a21 mem-pools a0* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *a19'*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a22 mp-alloc-stm4-inv-mif-len*
by (*simp add*: *len-levels*)
{ assume *a03*:*ii*≠*?i1* ∧ *ii*≠*?i2*
then have *eq-get-bit-i-j*:*get-bit-s x p ii j* = *get-bit-s Va p ii j*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a9*[*simplified mp-alloc-stm4-froml*[*OF a9*], *THEN sym*]] *a18*] **by** *fast*
moreover have *free-list* (*levels* (*mem-pool-info x p*) ! *ii*) =
free-list (*levels* (*mem-pool-info Va p*) ! *ii*)
using *free-level-x-va*[*OF a13*] *a03 a9 from-l* **by** *metis*
ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
using *get-bits-va eq-get-bit-i-j*
by (*simp add*: *buf maxsz*)
}
moreover { assume *a03*:*ii*=*?i1*
then have *free*:*free-list* (*levels* (*mem-pool-info x p*) ! *ii*) =
free-list (*levels* (*mem-pool-info Va p*) ! *ii*)
using *free-level-x-va*[*OF a13*] *a03 a9 from-l from-l-gt0* **by** *auto*
{ assume *a04*:*j*≠*?j1*
then have *eq-get-bit-i-j*:*get-bit-s x p ii j* = *get-bit-s Va p ii j*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a9*[*simplified from-l*, *THEN sym*]] *a18*]
a03 from-l-gt0
by (*simp add*: *eq-nat-nat-iff*)
then have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
using *free* **by** (*simp add*: *buf get-bits-va maxsz*)
}
moreover { assume *a04*:*j*=*?j1*
then have (*?bts* ! *j* = *DIVIDED*)
using *get-bit-x-l-b a03 a13 a00 a22 len-levels*
len-eq a13 from-l from-l-gt0
by (*simp add*: *a9*)

moreover have *buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*) ∉
set (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *get-bits-va a03 a20 a21 a04 a7* **unfolding** *inv-aux-vars-def*
by (*metis BlockState.distinct*(*17*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
by (*simp add*: *buf free maxsz*)
}
ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
by *auto*
}
moreover { assume *a03*:*ii*=*?i2*
then have *block-n*:(*block-num* (*mem-pool-info Va p*)
(*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) = *n*
proof−
have *lsizes Va t* ! *nat* (*from-l Va t*) =
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ
(*nat* (*from-l Va t*))
using *a14 lsizes-x-va a16 a1 a2 a4 a5 a9 from-l* **by** *auto*
thus *?thesis* **using** *block-n a21 a0 a0 a7 a3 a4 from-l-gt0*
by *blast*
qed
obtain *m* **where** *max-sz*:*max-sz* (*mem-pool-info Va p*) = *4* ∗ *m* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*)
using *a21 a0* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
have *ls*:*4* ˆ *ii dvd 4* ∗ *m* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*) **using** *a03 a22*
by (*metis dvd-triv-right inv-mempool len-levels less-imp-le-nat power-le-dvd*)
have *b0*:*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*) =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) *ii j*
unfolding *addr-def* **by** *auto*
have *suc-from-l-lt-lsize*:(*nat* (*from-l Va t*)) + *1*< *length* (*lsizes Va t*)
using *a4 a5 from-l-gt0* **by** *linarith*
have *b2*:∀ *j*. (*lsizes Va t* ! *nat* (*from-l Va t* + *1*)) ∗ *j* + *blk Va t* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*nat* (*from-l Va t* + *1*))
((*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))∗*4* + *j*)
using *lsizes-addr*[*OF a0 a14 a17 a21 suc-from-l-lt-lsize*] *a7 from-l-gt0 block-n*
by (*simp add*: *Suc-nat-eq-nat-zadd1 add.commute*)
then have *b2*:∀ *j*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *j* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*nat* (*from-l Va t* + *1*))
((*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*

(*from-l Va t*)))∗*4* + *j*)
  by (*metis mp-alloc-stm4-blk mp-alloc-stm4-pre-precond-f-froml mp-alloc-stm4-pre-precond-f-lsz*)
  { assume *a04*:*j*≠*?j2* ∧ *j*≠*Suc ?j2* ∧ *j*≠*Suc* (*Suc ?j2*) ∧ *j*≠*Suc* (*Suc* (*Suc ?j2*))
   then have *eq-get-bit-i-j*:*get-bit-s x p ii j* = *get-bit-s Va p ii j*
    using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a9*[*simplified from-l*, *THEN sym*]] *a18*]
       a03 from-l-gt0
    by (*simp add*: *eq-nat-nat-iff*)
   { assume *get-bit-s Va p ii j* = *FREE*
    then have (*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
       ∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
     using *get-bits-va* **by** *blast*
    then have (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
     using *free-list-x-subset-va*[*OF a19*] *a03 buf maxsz* **by** *fastforce*
   }
   moreover {
    assume *get-bit-s Va p ii j* ≠ *FREE*
     then have *not-in-free-Va*: (*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
       ∉ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
     using *get-bits-va* **by** *blast*
    then have (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∉ *set ?fl*)
    proof−
     have ∀ *k*. *k*<*4* ⟶ (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*)) ≠
        lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
          nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *k* +
          blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*
      using *diff-n-m-addr b2 b0 a03 a04 Euclidean-Division.div-eq-0-iff buf inv-mempool*
       ls max-sz maxsz
      by (*smt Groups.mult-ac*(*2*) *add.right-neutral add-2-eq-Suc′ add-Suc-right*
       dvd-div-mult-self less-2-cases less-Suc-eq div-greater-zero-iff
       mult-is-0 neq0-conv numeral-Bit0 power-not-zero)
     then show *?thesis* **using** *buf maxsz not-in-free-Va set-free-x-va*[*OF a19*, *simplified*] *a03*
      apply *auto*
      by *presburger*
    qed
    then have (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∉ *set ?fl*)
     using *a03 buf maxsz a04 set-free-x-va*[*OF a19*] **by** *auto*
   } ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
    using *eq-get-bit-i-j* **by** *auto*
  }
  moreover { assume *a04*:*j*=*?j2*
   then have *a03′*:*ii* = *nat* (*from-l x t* + *1*) ∧

$j = 4*block\text{-}num\ (mem\text{-}pool\text{-}info\ x\ p)\ (blk\ x\ t)\ (lsizes\ x\ t\ !\ nat\ (from\text{-}l\ x\ t))$
using *a22 a23 a24 from-l buf from-l a03*
unfolding *block-num-def*
by (*simp add*: *mp-alloc-stm4-pre-precond-f-blk mp-alloc-stm4-pre-precond-f-lsz*)
have (*?bts* ! *j* = *ALLOCATING*)
using *from-l-gt0 a22 a00 a03 len-levels a04 a18 from-l get-bit-x-l1-b4 len-eq*
by (*metis a04 a18 get-bit-x-l1-b4 len-eq*)
then have *bts-j-not-free*:(*?bts* ! *j* ≠ *FREE*)
by *auto*
moreover have *not-in-free-Va*:*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*) ∉
set (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
proof−
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a20 a21 a7* **unfolding** *inv-aux-vars-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*)
Mem-block.select-convs(*2*) *Mem-block.select-convs*(*3*))
have *noexist-bits* (*mem-pool-info Va p*) (*?i1* + *1*) (*?j1* ∗ *4*)
proof−
have *?i1* < *length* (*levels* (*mem-pool-info Va p*))−*1*
using *a19′ from-l-gt0 a3 a4 inv-mempool* **by** *auto*
moreover have *?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
using *calculation*
by (*simp add*: *a6 a7 inv-mempool*)
ultimately show *?thesis*
using *alloc-i1-j1 a21 a19′ a00′ a0 a03 a04*
unfolding *Let-def inv-bitmap-def*
by (*smt One-nat-def Suc-pred inv-mempool less-Suc-eq*)
qed
then have (*get-bit-s Va p ii j* = *NOEXIST*)
using *a03 a04 from-l-gt0*
by (*simp add*: *mult.commute nat-add-distrib*)
then show *?thesis* **using** *get-bits-va*
by *simp*
qed
have (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∉ *set ?fl*)
proof−
have ∀ *k*. *k*>*0* ∧ *k*<*4* ⟶ (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*)) ≠
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *k* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*
using *diff-n-m-addr b2 b0 a03 a04 buf inv-mempool*
maxsz
by (*metis a19′ add-cancel-right-right div-greater-zero-iff*
mp-alloc-stm3-lm2-inv-1-2 mult.commute neq0-conv)
then show *?thesis* **using** *buf maxsz not-in-free-Va set-free-x-va*[*OF a19*, *simplified*] *a03*
by *auto*

qed
then have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
using *bts-j-not-free* **by** *auto*
}
moreover {
assume *a04*:*j*=*Suc ?j2* ∨ *j* = *Suc* (*Suc ?j2*) ∨ *j* = *Suc* (*Suc* (*Suc ?j2*))
then have *a04′*:*j*=(*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *1*) ∨
j = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *2*) ∨
j = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *3*)
by *auto*
have (*?bts* ! *j* = *FREE*)
using *get-bit-x-l1-b41*[*OF conjI*[*OF a03 a04*] *from-l-gt0* - *a18 a19′ a00′*]
using *a13 a9 from-l* **by** *auto*
moreover have *buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*
using *a03 a04′*[*simplified*] *set-free-x-va*[*OF a19*, *simplified b2 buf*[*THEN sym*] *maxsz*[*THEN sym*]]
using *b0*[*simplified buf*[*THEN sym*] *maxsz*[*THEN sym*] *a03*] **by** *auto*
ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*) **by** *auto*
} ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*)
by *auto*
} ultimately have (*?bts* ! *j* = *FREE*) = (*buf ?mp* + *j* ∗ (*max-sz ?mp div 4* ˆ *ii*) ∈ *set ?fl*) **by** *auto*
} then show *?thesis* **by** *auto*
qed

lemma *free-list-updates-inv2*:
assumes *a0*:*p* ∈ *mem-pools Va* **and**
a1:¬ *free-l Va t* < *OK* **and**
a2:*free-l Va t* ≤ *from-l Va t* **and**
a3:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a4:*from-l Va t* < *alloc-l Va t* **and**
a5: *alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
a6:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a7:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a8:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**

a9:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a10:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a11:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a12:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a13:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a14:
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a15:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* **and**
a16:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a17:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a18:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows ∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*)).
∃ *n*<*n-max* (*mem-pool-info x p*) ∗ *4* ˆ *ii*.
free-list (*levels* (*mem-pool-info x p*) ! *ii*) ! *j* =
buf (*mem-pool-info x p*) + *n* ∗ (*max-sz* (*mem-pool-info x p*) *div 4* ˆ *ii*)

proof−
{ **let** *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))
let *?mp* = *mem-pool-info x p*
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
fix *j*
assume *a00*:*j*<*length ?fl*
have *inv-bitmap2*:(∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)).
∃ *n*<*n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *ii*.
free-list (*levels* (*mem-pool-info Va p*) ! *ii*) ! *j* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*))
using *a15 a0 a16 mp-alloc-stm4-lvl-len*[*OF a0 a8*]
unfolding *Let-def inv-bitmap-freelist-def*
by *fastforce*+

have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a1 a2* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
using *mp-alloc-stm4-lvl-len*[*OF a0 a8*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a0 a8*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a0 a8*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a9*] **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a8*] **by** *auto*
have *lsizes-x-va*:*lsizes x* = *lsizes Va*
by (*simp add*: *a12 mp-alloc-stm4-pre-precond-f-lsz*)
have *len-eq*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a16 a8 mp-alloc-stm4-inv-bits-len*
unfolding *gvars-conf-stable-def gvars-conf-def*
by *fastforce*
have *inv-mempool-info-mp Va p*
using *a15 mem-pools a0* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *a15'*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a16 mp-alloc-stm4-inv-mif-len*
by (*simp add*: *len-levels*)
have *nmax*: *n-max* (*mem-pool-info x p*) = *n-max* (*mem-pool-info Va p*)
using *a8* **unfolding** *gvars-conf-stable-def gvars-conf-def* **apply** *auto*
by (*metis mp-alloc-stm4-nmax*)
{ assume *a03*:*ii*≠ *?i2*
then have ∃ *n*<*n-max ?mp* ∗ *4* ˆ *ii*. *?fl* ! *j* = *buf ?mp* + *n* ∗ (*max-sz ?mp div 4* ˆ *ii*)
using *a0 a00 a10 buf eq-free-list-mp-alloc-stm4-pre-precond-f*
inv-bitmap2 maxsz nmax
by (*simp add*: *eq-free-list-mp-alloc-stm4-pre-precond-f mp-alloc-stm4-pre-precond-f-froml*
)
}
moreover {
assume *a03*:*ii*= *?i2*
then have *suc-from-l-lt-lsize*:(*nat* (*from-l Va t*)) + *1*< *length* (*lsizes Va t*)
using *a4 a5 from-l-gt0* **by** *linarith*
then have *lsize-i*:*lsizes Va t* ! *nat* (*from-l Va t*) =
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ
(*nat* (*from-l Va t*))
using *a11 add-lessD1 suc-from-l-lt-lsize* **by** *blast*
then have *block-n*:(*block-num* (*mem-pool-info Va p*)
(*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) = *n*
using *block-n a0 a3 a4 from-l-gt0 a15 a7* **by** *blast*
have *lsize-ii*:*lsizes Va t* ! *ii* =
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii*

using *a11 from-l-gt0 suc-from-l-lt-lsize*
by (*simp add*: *a03*)
{**assume** *a04*:*j*< *length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
then have *free-list* (*levels* (*mem-pool-info Va p*) ! *ii*) ! *j* = *?fl* ! *j*
using *a14*[*simplified mp-alloc-stm4-pre-precond-f-froml eq-free-list-mp-alloc-stm4-pre-precond-f*] *a03*
unfolding *inserts-def*
by (*simp add*: *nth-append*)
moreover have ∃ *n*<*n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *ii*.
free-list (*levels* (*mem-pool-info Va p*) ! *ii*) ! *j* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
using *a04 inv-bitmap2* **by** *fastforce*
ultimately have ∃ *n*<*n-max ?mp* ∗ *4* ˆ *ii*. *?fl* ! *j* = *buf ?mp* + *n* ∗ (*max-sz ?mp div 4* ˆ *ii*)
using *buf maxsz nmax* **by** *auto*
}
moreover { **assume** *a04*:*j* = *length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
then have *fl-lsizes*:*?fl* ! *j* = *lsizes Va t* ! *nat* (*from-l Va t* + *1*) ∗ *1* + *blk Va t*
using *free-list-x*[*OF a14*] *a03 a9 eq-free-list-mp-alloc-stm4-pre-precond-f nth-append-length*
by (*metis* (*no-types*, *lifting*) *a18 from-l lsizes-x-va mp-alloc-stm4-blk*)
let *?nb* = (*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *1*)
have *eq-suc-from-l*:*nat* (*from-l Va t* + *1*) = *nat* (*from-l Va t*) + *1* **using** *from-l-gt0* **by** *auto*
from *fl-lsizes*[*simplified a7 this*] **have** *?fl* ! *j* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*nat* (*from-l Va t*) + *1*)
(*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *1*)
using *spec*[*OF lsizes-addr*[*OF a0 a11 a13 a15 suc-from-l-lt-lsize*], *of 1*]
by *auto*
moreover have *length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
n-max (*mem-pool-info x p*) ∗ *4* ˆ *ii* **using** *a15 a0*
unfolding *inv-mempool-info-def Let-def*
by (*simp add*: *a15′ len-eq nmax*)
moreover have *?nb* < *n-max* (*mem-pool-info x p*) ∗ *4* ˆ *ii*
using *a6 a03 a0 nmax lsize-ii lsize-i eq-suc-from-l inv-mempool-info-maxsz-align4*[*OF conjunct1*[*OF conjunct2*[*OF conjunct2*[*OF a15*]]], *simplified a0*]
unfolding *block-num-def*
by *auto*
ultimately have

?nb < n-max ?mp ∗ 4 ^ ii ∧ ?fl ! j = buf ?mp + ?nb ∗ (max-sz ?mp div 4 ^ ii)
using block-n a7 buf nmax maxsz a6 a03 eq-suc-from-l **unfolding** addr-def
by auto
then have ∃ n<n-max ?mp ∗ 4 ^ ii. ?fl ! j = buf ?mp + n ∗ (max-sz ?mp div 4 ^ ii) **by** auto
}
moreover { assume a04:j = Suc (length (free-list (levels (mem-pool-info Va p) ! ii)))
then have fl-lsizes:?fl ! j = lsizes Va t ! nat (from-l Va t + 1) ∗ 2 + blk Va t
using free-list-x[OF a14] a03 a9 eq-free-list-mp-alloc-stm4-pre-precond-f nth-append-length a18 from-l lsizes-x-va mp-alloc-stm4-blk
by (metis add.right-neutral add-Suc-right nth-Cons-Suc nth-append-length-plus)
let ?nb = (block-num (mem-pool-info Va p)
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t)) ∗ 4 + 2)
have eq-suc-from-l:nat (from-l Va t + 1) = nat (from-l Va t) + 1 **using** from-l-gt0 **by** auto
from fl-lsizes[simplified a7 this] **have** ?fl ! j =
addr (max-sz (mem-pool-info Va p)) (buf (mem-pool-info Va p)) (nat (from-l Va t) + 1)
(block-num (mem-pool-info Va p)
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t)) ∗ 4 + 2)
using spec[OF lsizes-addr[OF a0 a11 a13 a15 suc-from-l-lt-lsize], of 2 n]
by fastforce
moreover have length (bits (levels (mem-pool-info x p) ! ii)) =
n-max (mem-pool-info x p) ∗ 4 ^ ii **using** a15 a0
unfolding inv-mempool-info-def Let-def
by (simp add: a15′ len-eq nmax)
moreover have ?nb < n-max (mem-pool-info x p) ∗ 4 ^ ii
using a6 a03 a0 nmax lsize-ii lsize-i eq-suc-from-l
inv-mempool-info-maxsz-align4[OF conjunct1[OF conjunct2[OF conjunct2[OF a15]]], simplified a0]
unfolding block-num-def
by auto
ultimately have
?nb < n-max ?mp ∗ 4 ^ ii ∧ ?fl ! j = buf ?mp + ?nb ∗ (max-sz ?mp div 4 ^ ii)
using block-n a7 buf nmax maxsz a6 a03 eq-suc-from-l **unfolding** addr-def
by auto
then have ∃ n<n-max ?mp ∗ 4 ^ ii. ?fl ! j = buf ?mp + n ∗ (max-sz ?mp div 4 ^ ii) **by** auto
}
moreover { assume a04:j = Suc (Suc (length (free-list (levels (mem-pool-info Va p) ! ii))))

then have *fl-lsizes*:*?fl* ! *j* = *lsizes Va t* ! *nat* (*from-l Va t* + *1*) ∗ *3* + *blk Va t*
using *free-list-x*[*OF a14*] *a03 a9 eq-free-list-mp-alloc-stm4-pre-precond-f nth-append-length a18 from-l lsizes-x-va mp-alloc-stm4-blk*
by (*metis add.right-neutral add-Suc-right nth-Cons-Suc nth-append-length-plus*)
let *?nb* = (*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *3*)
have *eq-suc-from-l*:*nat* (*from-l Va t* + *1*) = *nat* (*from-l Va t*) + *1* **using** *from-l-gt0* **by** *auto*
from *fl-lsizes*[*simplified a7 this*] **have** *?fl* ! *j* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*nat* (*from-l Va t*) + *1*)
(*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∗ *4* + *3*)
using *spec*[*OF lsizes-addr*[*OF a0 a11 a13 a15 suc-from-l-lt-lsize*], *of 3 n*]
by *fastforce*
moreover have *length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
n-max (*mem-pool-info x p*) ∗ *4* ^ *ii* **using** *a15 a0*
unfolding *inv-mempool-info-def Let-def*
by (*simp add*: *a15′ len-eq nmax*)
moreover have *?nb* < *n-max* (*mem-pool-info x p*) ∗ *4* ^ *ii*
using *a6 a03 a0 nmax lsize-ii lsize-i eq-suc-from-l inv-mempool-info-maxsz-align4*[*OF conjunct1*[*OF conjunct2*[*OF conjunct2*[*OF a15*]]], *simplified a0*]
unfolding *block-num-def*
by *auto*
ultimately have
?nb < *n-max ?mp* ∗ *4* ^ *ii* ∧ *?fl* ! *j* = *buf ?mp* + *?nb* ∗ (*max-sz ?mp div 4* ^ *ii*)
using *block-n a7 buf nmax maxsz a6 a03 eq-suc-from-l* **unfolding** *addr-def*
by *auto*
then have ∃ *n*<*n-max ?mp* ∗ *4* ^ *ii*. *?fl* ! *j* = *buf ?mp* + *n* ∗ (*max-sz ?mp div 4* ^ *ii*) **by** *auto*
} **ultimately have** ∃ *n*<*n-max ?mp* ∗ *4* ^ *ii*. *?fl* ! *j* = *buf ?mp* + *n* ∗ (*max-sz ?mp div 4* ^ *ii*)
using *a00 free-list-x*[*OF a14*, *simplified eq-free-list-mp-alloc-stm4-pre-precond-f mp-alloc-stm4-pre-precond-f-froml*] *a03*
by *fastforce*
} **ultimately have** ∃ *n*<*n-max ?mp* ∗ *4* ^ *ii*. *?fl* ! *j* = *buf ?mp* + *n* ∗ (*max-sz ?mp div 4* ^ *ii*)
by *auto*
} **then show** *?thesis* **by** *auto*
qed

lemma *next-block-less-length-bits*:
assumes
 a0:*n* < *length* (*bits* (*levels pi* ! *ii*)) **and**
 a1:(*ii*+*1*) < *length* (*levels pi*) **and**
 a2:(∀ *i*<*length* (*levels pi*).
 length (*bits* (*levels pi* ! *i*)) = *n-max pi* ∗ *4* ˆ *i*)
shows *4*∗*n* + *3* < *length* (*bits* (*levels pi* ! (*ii*+*1*)))
proof−
 have *n*< *n-max pi* ∗ *4* ˆ *ii* **using** *a0 a1 a2* **by** *auto*
 moreover have *length* (*bits* (*levels pi*!(*ii*+*1*))) = *n-max pi* ∗ *4* ˆ(*ii*+*1*) **using** *a1 a2* **by** *auto*
 ultimately show *?thesis* **by** *auto*
qed

lemma *distinct-lists*: **assumes**
a0:*distinct l1* **and**
a1:*distinct l2* **and**
a2:∀ *e*∈*set l2*. *e* ∉ *set l1*
shows *distinct* (*l1* @ *l2*)
 using *assms*
 by(*induct l1*, *auto*)

lemma *free-list-updates-inv3*:
 assumes *a0*:*p* ∈ *mem-pools Va* **and**
 a1:¬ *free-l Va t* < *OK* **and**
 a2:*free-l Va t* ≤ *from-l Va t* **and**
 a3:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
 a4:*from-l Va t* < *alloc-l Va t* **and**
a5:*alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
 alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
 a6:*block-num* (*mem-pool-info Va p*)
 (*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
 (*lsizes Va t* ! *nat* (*from-l Va t*))
 < *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
 a7:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)) **and**
 a8:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
 a9:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
 a10:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
 a11:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
 a12:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
 a13:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
 levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**

a14:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a15:*i x t* = *4* **and**
a16:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a17:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a18:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a19:
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a20:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*))|) **and**
a21:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va* **and**
a22:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a23:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a24:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows *distinct* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*))
proof−
{
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*)))
let *?mp* = *mem-pool-info x p*
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
have *inv-bitmap1*:(∀ *j*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)).
(*get-bit-s Va p ii j* = *FREE*) =

(*buf* (*mem-pool-info Va p*) + *j* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))) **and**
inv-bitmap2:(∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)).
∃ *n*<*n-max* (*mem-pool-info Va p*) ∗ *4* ^ *ii*.
free-list (*levels* (*mem-pool-info Va p*) ! *ii*) ! *j* =
buf (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *ii*)) **and**
inv-bitmap3:*distinct* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a21 a0 a22 mp-alloc-stm4-lvl-len*[*OF a0 a8*]
unfolding *Let-def inv-bitmap-freelist-def*
by *fastforce+*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a1 a2* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
using *mp-alloc-stm4-lvl-len*[*OF a0 a8*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a0 a8*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a0 a8*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a9*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a8*] **by** *auto*
have *lsizes-x-va*:*lsizes x* = *lsizes Va*
by (*simp add*: *a16 mp-alloc-stm4-pre-precond-f-lsz*)
have *len-eq*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a22 a8 mp-alloc-stm4-inv-bits-len*
unfolding *gvars-conf-stable-def gvars-conf-def*
by *fastforce*
have *inv-mempool-info-mp Va p*
using *a21 mem-pools a0* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *a22'*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a22 mp-alloc-stm4-inv-mif-len*
by (*simp add*: *len-levels*)
{ assume *a03*:*ii*≠*?i2*
have *free-list* (*levels* (*mem-pool-info x p*) ! *ii*) =
free-list (*levels* (*mem-pool-info Va p*) ! *ii*)
using *free-level-x-va*[*OF a13*] *a03 a9 from-l* **by** *metis*
then have *distinct* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*))
using *inv-bitmap3* **by** *auto*
}
moreover { assume *a03*:*ii*=*?i2*
then have *block-n*:(*block-num* (*mem-pool-info Va p*)
(*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) = *n*

proof−
have *lsizes Va t* ! *nat* (*from-l Va t*) =
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ
(*nat* (*from-l Va t*))
using *a14 lsizes-x-va a16 a1 a2 a4 a5 a9 from-l* **by** *auto*
thus *?thesis* **using** *block-n a21 a0 a0 a7 a3 a4 from-l-gt0*
by *blast*
qed
then have *get-bit-s Va p* (*nat* (*from-l Va t*)) *n* = *ALLOCATING*
using *a20 a13 a21 a7* **unfolding** *inv-aux-vars-def*
by (*metis Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
moreover have *n-len*: *n*< *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *nat* (*from-l Va t*)))
using *a03 a22′ a6 a7 inv-mempool local.block-n* **by** *auto*
ultimately have *noexist-bits* (*mem-pool-info Va p*) *ii* (*n* ∗ *4*)
using *allocating-next-notexists*[*OF conjunct1*[*OF conjunct2*[*OF a21*]] *a0* -] *a21 a0 a03 a21 a0 a22′*
from-l-gt0 from-l-suc inv-mempool **by** *auto*
then have *get-bit-s Va p ii* (*n*∗*4* + *1*) ≠ *FREE* ∧
get-bit-s Va p ii (*n*∗*4* + *2*) ≠ *FREE* ∧
get-bit-s Va p ii (*n*∗*4* + *3*) ≠ *FREE*
by (*simp add*: *mult.commute*)
moreover have *n*∗*4* + *3* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *nat* (*from-l Va t* + *1*)))
using *a03 a22′ n-len inv-mempool from-l-gt0 next-block-less-length-bits from-l-suc*
by *simp*
ultimately have *not-in-freelist*:(*buf* (*mem-pool-info Va p*) + (*n*∗*4* + *1*) ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∉ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))) ∧
(*buf* (*mem-pool-info Va p*) + (*n*∗*4* + *2*) ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∉ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))) ∧
(*buf* (*mem-pool-info Va p*) + (*n*∗*4* + *3*) ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *ii*)
∉ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*)))
using *inv-bitmap1 a03*
by (*metis* (*no-types*, *lifting*) *add-lessD1 numeral-3-eq-3*
one-add-one plus-1-eq-Suc semiring-normalization-rules(*21*))
obtain *m* **where** *max-sz*:*max-sz* (*mem-pool-info Va p*) = *4* ∗ *m* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*)
using *a21 a0* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
have *ls*:*4* ˆ *ii dvd 4* ∗ *m* ∗ *4* ˆ *n-levels* (*mem-pool-info Va p*) **using** *a03 a22*
by (*metis dvd-triv-right inv-mempool len-levels less-imp-le-nat power-le-dvd*)
have *suc-from-l-lt-lsize*:(*nat* (*from-l Va t*)) + *1*< *length* (*lsizes Va t*)
using *a4 a5 from-l-gt0* **by** *linarith*
have *b2*:∀ *j*. (*lsizes Va t* ! *nat* (*from-l Va t* + *1*)) ∗ *j* + *blk Va t* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*)) (*nat* (*from-l Va t* + *1*))
((*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*

(*from-l Va t*)))∗*4* + *j*)
using *lsizes-addr*[*OF a0 a14 a17 a21 suc-from-l-lt-lsize*] *a7 from-l-gt0 block-n*
by (*simp add*: *Suc-nat-eq-nat-zadd1 add.commute*)
then have *b2*:∀ *j*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *j* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t* =
addr (*max-sz* (*mem-pool-info Va p*)) (*buf* (*mem-pool-info Va p*))
(*nat* (*from-l Va t* + *1*))
((*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat*
(*from-l Va t*)))∗*4* + *j*)
by (*metis mp-alloc-stm4-blk mp-alloc-stm4-pre-precond-f-froml mp-alloc-stm4-pre-precond-f-lsz*)
then have *distinct* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*))
proof−
have *h1*:*distinct* [*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l*
(*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *1* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*) ∗ *2* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*) ∗ *3* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*] **using** *b2 a03 a22*′ *inv-mempool*
mp-alloc-stm3-lm2-inv-1-2 **unfolding** *addr-def*
by (*smt add-diff-cancel-left*′ *distinct-length-2-or-more*
distinct-singleton mult-cancel-right nat-less-le num.distinct(*3*)
num.distinct(*5*)
numeral-eq-iff numeral-eq-one-iff semiring-norm(*85*))
have *h2*:∀ *e*∈*set* [*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l*
(*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗ *1* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*) ∗ *2* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*,
lsizes (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*) ∗ *3* +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*].
e ∉ *set* (*free-list* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *b2 a03 not-in-freelist local.block-n* **unfolding** *addr-def* **apply** *auto*
by (*metis* (*no-types*) *not-in-freelist semiring-normalization-rules*(*12*))
show *?thesis*

using *distinct-lists*[*OF inv-bitmap3 h1 h2*] *free-list-x*[*OF a19*]
by (*metis a03 eq-free-list-mp-alloc-stm4-pre-precond-f mp-alloc-stm4-pre-froml*)
qed
} **ultimately have** *distinct* (*free-list* (*levels* (*mem-pool-info x p*) ! *ii*))
by *auto*

} **then show** *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-bitmap-freelist*:
assumes $a0$:$p \in$ *mem-pools Va* **and**
$a1$:$\neg$ *free-l Va t* $<$ *OK* **and**
$a2$:*free-l Va t* $\leq$ *from-l Va t* **and**
$a3$:*alloc-l Va t* $<$ *int* (*n-levels* (*mem-pool-info Va p*)) **and**
$a4$:*from-l Va t* $<$ *alloc-l Va t* **and**
$a4'$:*alloc-l Va t* $=$ *int* (*length* (*lsizes Va t*)) $-$ *1* $\wedge$ *length* (*lsizes Va t*) $=$ *n-levels* (*mem-pool-info Va p*) $\vee$
alloc-l Va t $=$ *int* (*length* (*lsizes Va t*)) $-$ *2* $\wedge$ *lsizes Va t* ! *nat* (*alloc-l Va t* $+$ *1*) $<$ *sz* **and**
$a5$:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) $+$ *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div* *4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
$<$ *n-max* (*mem-pool-info Va p*) $*$ *4* ^ *nat* (*from-l Va t*) **and**
$a6$:*blk Va t* $=$ *buf* (*mem-pool-info Va p*) $+$ *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div* *4* ^ *nat* (*from-l Va t*)) **and**
$a7$:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* **and**
$a8$:*from-l x* $=$ *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
$a9$:*freeing-node x* $=$ *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
$a10$:*allocating-node x* $=$ *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
$a11$:$\forall$ *pa*. *pa* $\neq$ *p* $\longrightarrow$ *mem-pool-info x pa* $=$ *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
$a12$:$\forall$ *jj*. *jj* $\neq$ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*) $\longrightarrow$
levels (*mem-pool-info x p*) ! *jj* $=$ *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
$a12'$:$\forall$ *ii*$<$*length* (*lsizes Va t*). *lsizes Va t* ! *ii* $=$ *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div* *4* ^ *ii* **and**
$a12''$:*i x t* $=$ *4* **and**
$a12'''$:*lsizes x* $=$ *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
$a12''''$:*length* (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) **and**
$a13$:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)) $=$
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $*$ *4*)) *3 FREE* **and**
$a14$:
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*)) $=$
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $+$ *1*) $*$
ii $+$
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)

[*Suc NULL..<4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a15: *inv-mempool-info Va* **and**
a16:*inv-bitmap-freelist Va* **and**
a17:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*
4 ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*
4 ˆ *nat* (*from-l Va t*))|) **and**
a18:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist*
Va **and**
a19:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows*inv-bitmap-freelist x*
proof−
{ **fix** *p′*
assume *a00*:*p′*∈*mem-pools x*
{**assume** *p*≠*p′*
moreover have *mem-pool-info x p′* = *mem-pool-info Va p′*
using *mp-alloc-stm4-pres-mpinfo*
by (*metis a11 calculation*)
ultimately have *inv-bitmap-freelist-mp x p′*
using *a18 a00 mp-alloc-stm4-lvl-len*[*OF a0 a7*] *mp-alloc-stm4-maxsz*[*OF*
a0 a7]
mp-alloc-stm4-buf[*OF a0 a7*] *mp-alloc-stm4-froml*[*OF a8*] *mp-alloc-stm4-mempools*[*OF*
a7]
by(*simp add*: *inv-bitmap-freelist-def Let-def*)
}
moreover { **assume** *eq-p*:*p*=*p′*
let *?mp* = *mem-pool-info x p′*
have *inv-mempool-info-mp Va p′*
using *a15 eq-p mp-alloc-stm4-mempools*[*OF a7*] *a00* **unfolding** *inv-mempool-info-def*
by *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
{**fix** *i*
assume *a01*:*i*<*length* (*levels ?mp*)
then have *inv-bitmap1*:(∀ *j*<*length* (*bits* (*levels* (*mem-pool-info Va p′*) ! *i*)).
(*get-bit-s Va p′ i j* = *FREE*) =
(*buf* (*mem-pool-info Va p′*) + *j* ∗ (*max-sz* (*mem-pool-info Va*
p′) *div 4* ˆ *i*)
∈ *set* (*free-list* (*levels* (*mem-pool-info Va p′*) ! *i*)))) **and**
inv-bitmap2:(∀ *j*<*length* (*free-list* (*levels* (*mem-pool-info Va p′*) !
i)).
∃ *n*<*n-max* (*mem-pool-info Va p′*) ∗ *4* ˆ *i*.
free-list (*levels* (*mem-pool-info Va p′*) ! *i*) ! *j* =

buf (mem-pool-info Va p′) + n ∗ (max-sz (mem-pool-info Va p′) div 4 ˆ i)) **and**
inv-bitmap3:distinct (free-list (levels (mem-pool-info Va p′) ! i))
using a16 eq-p mp-alloc-stm4-mempools[OF a7] a00 a01 mp-alloc-stm4-lvl-len[OF a0 a7]
unfolding Let-def inv-bitmap-freelist-def
by fastforce+
let ?bts = bits (levels ?mp ! i) **and** ?fl = free-list (levels ?mp ! i)
have f1:(∀ j<length ?bts. (?bts ! j = FREE) = (buf ?mp + j ∗ (max-sz ?mp div 4 ˆ i) ∈ set ?fl))
using assms free-list-updates-inv1 a00 a01 eq-p **by** blast
have f2: (∀ j<length ?fl. ∃ n<n-max ?mp ∗ 4 ˆ i. ?fl ! j = buf ?mp + n ∗ (max-sz ?mp div 4 ˆ i))
using assms free-list-updates-inv2 a00 a01 eq-p **by** blast
have f3:distinct ?fl **using** assms free-list-updates-inv3 a00 a01 eq-p **by** blast
note conjI[OF f1 conjI[OF f2 f3]]
} **then** **have** inv-bitmap-freelist-mp x p′ **by** auto
}
ultimately have inv-bitmap-freelist-mp x p′ **by** auto
}
thus ?thesis **unfolding** inv-bitmap-freelist-def **by** auto
qed

lemma noexists-eq-bits: **assumes**
a0:∀ j. j≥ jj ∧ j≤ Suc(Suc (Suc jj)) ⟶
get-bit-s x p ii j = get-bit-s Va p ii j **and**
a1:noexist-bits (mem-pool-info Va p) ii jj
shows noexist-bits (mem-pool-info x p) ii jj
using a0 a1
by simp

lemma mp-alloc-stm4-inv-bitmap1:
assumes
a0:inv Va **and**
a1:p ∈ mem-pools Va **and**
a2:∀ ii<length (lsizes Va t). lsizes Va t ! ii = ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ˆ ii **and**
a4:alloc-l Va t < int (n-levels (mem-pool-info Va p)) **and**
a5:¬ free-l Va t < OK **and**
a6:free-l Va t ≤ from-l Va t **and**
a7:allocating-node Va t =
Some (|pool = p, level = nat (from-l Va t),
block = block-num (mem-pool-info Va p)
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ˆ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t)),
data = buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ˆ nat (from-l Va t))|) **and**
a8:n = block-num (mem-pool-info Va p)

(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) = *NULL* **and**
a9:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a10:*from-l Va t* < *alloc-l Va t* **and**

a11:*n* < *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a12:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a13:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a14:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a15:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a16:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a17:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a18:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a01:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a02:*jj* < *length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*))
shows (*get-bit-s x p ii jj* = *FREE* ∨ *get-bit-s x p ii jj* = *FREEING* ∨ *get-bit-s x p ii jj* = *ALLOCATED* ∨ *get-bit-s x p ii jj* = *ALLOCATING* ⟶
(*NULL* < *ii* ⟶ *get-bit-s x p* (*ii* − *1*) (*jj div 4*) = *DIVIDED*) ∧
(*ii* < *length* (*levels* (*mem-pool-info x p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info x p*) (*ii* + *1*) (*jj* ∗ *4*)))
proof−
let *?mp* = *mem-pool-info x p*
have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*

have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a6 a5* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
using *mp-alloc-stm4-lvl-len*[*OF a1 a13*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a1 a13*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a1 a13*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a18*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a13*] **by** *auto*
have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a17* **by** *auto*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*) **and**
?i1′ = (*nat* (*from-l Va t*)) − *1* **and**
?j1′ = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *div 4* **and**
?i2′ = (*nat* (*from-l Va t*)) + *2*
let *?j20′* = *?j2* ∗ *4* **and** *?j21′* = (*?j2*+*1*) ∗ *4* **and** *?j22′* = (*?j2*+*2*)∗*4* **and** *?j23′* = (*?j2*+*3*)∗*4* **and** *?j24′* = (*?j2*+*4*)∗*4*
let *?mp* = *mem-pool-info x p*
have *inv-mempool-info-mp Va p*
using *a1 mem-pools inv* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis i1-len a9 a12 a1 inv inv-mempool-info-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
let *?bts* = *bits* (*levels ?mp* ! *ii*)
let *?btsva* = (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
have *a01′*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a01 len-levels* **by** *auto*
then have *inv-bitmap1*:

$\forall j < length\ (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ ii)).$
$(?btsva\ !\ j = FREE \lor ?btsva\ !\ j = FREEING \lor ?btsva\ !\ j = ALLOCATED$
$\lor\ ?btsva\ !\ j = ALLOCATING \longrightarrow$
$(ii > 0 \longrightarrow (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ (ii - 1)))\ !\ (j\ div$
$4) = DIVIDED)$
$\land\ (ii < length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)) - 1 \longrightarrow noexist\text{-}bits$
$(mem\text{-}pool\text{-}info\ Va\ p)\ (ii{+}1)\ (j{*}4)\ ))$
$\land\ (?btsva\ !\ j = DIVIDED \longrightarrow ii > 0 \longrightarrow (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va$
$p)\ !\ (ii - 1)))\ !\ (j\ div\ 4) = DIVIDED)$
$\land\ (?btsva\ !\ j = NOEXIST \longrightarrow ii < length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p))$
$-\ 1$
$\longrightarrow noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ (ii{+}1)\ (j{*}4))$
$\land\ (?btsva\ !\ j = NOEXIST \land ii > 0 \longrightarrow (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va$
$p)\ !\ (ii - 1)))\ !\ (j\ div\ 4) \neq DIVIDED)$
using *inv mem-pools a1*
unfolding *Let-def inv-bitmap-def*
by *blast*
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a7 a0 a12* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
then have *alloc-predi1-j1*:$?i1 > 0 \longrightarrow get\text{-}bit\text{-}s\ Va\ p\ (?i1 - 1)\ (?j1\ div\ 4) =$
$DIVIDED$
using *inv-bitmap1 i1-len j1-len inv a1* **unfolding** *Let-def inv-bitmap-def* **by**
blast
have *nexisti2*:*noexist-bits* (*mem-pool-info Va p*) *?i2 ?j2*
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*]
i1-len j1-len
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less*
plus-1-eq-Suc)
have *nexisti3*:$?i2 < length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)) - 1 \longrightarrow$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j20' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j21' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j22' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j23'$
proof−
{ **assume** $?i2 < length\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)) - 1$
then have *a00*:$\forall j{<}length\ (bits\ (levels\ (mem\text{-}pool\text{-}info\ Va\ p)\ !\ ?i2)).$
$get\text{-}bit\text{-}s\ Va\ p\ ?i2\ j = NOEXIST \longrightarrow noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va$
$p)\ ?i2'\ (j * 4)$
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*]
i2-len
from-l-suc **by** *auto*
then have $noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j20' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j21' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j22' \land$
$noexist\text{-}bits\ (mem\text{-}pool\text{-}info\ Va\ p)\ ?i2'\ ?j23'$
using *j2-len nexisti2 Suc-lessD*
by (*smt One-nat-def add.commute add-2-eq-Suc' add-Suc-right numeral-3-eq-3*

plus-1-eq-Suc)
}
thus *?thesis* **by** *fastforce*
qed
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
have *a02'*:*jj* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a02 a13* **unfolding** *gvars-conf-def gvars-conf-stable-def*
by (*simp add*: *mp-alloc-stm4-inv-bits-len*)
have *eq-len*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *mp-alloc-stm4-inv-bits-len a14 a15 length-list-update-n*
by *metis*
have *inv-va*:(*?btsva* ! *jj* = *FREE* ∨ *?btsva* ! *jj* = *FREEING* ∨ *?btsva* ! *jj* = *ALLOCATED* ∨ *?btsva* ! *jj* = *ALLOCATING* ⟶
(*ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*)))
∧ (*?btsva* ! *jj* = *DIVIDED* ⟶ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*?btsva* ! *jj* = *NOEXIST* ⟶ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*))
∧ (*?btsva* ! *jj* = *NOEXIST* ∧ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) ≠ *DIVIDED*)
using *inv-bitmap1 a02'* **by** *auto*
{ **assume** *a05*:¬((*ii*=*?i1* ∧ *jj*=*?j1*) ∨
(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*< *?j2*+*4*) ∨
(*ii*=*?i2'* ∧ *jj*≥ *?j20'* ∧ *jj*< *?j24'*) ∨
(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*))
then have *a050'*:¬(*ii*=*?i1* ∧ *jj*=*?j1*) **and**
a051': ¬(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*<*?j2* + *4*) **and**
a052':¬(*ii*=*?i2'* ∧ *jj*≥ *?j20'* ∧ *jj*< *?j24'*) **and**
a053': ¬(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*)
by *force*+
have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*, *of ii jj*]
using *a050' a051'* **by** *auto*
have *eq-get-bit-i1-j1*:*ii*>*0* ⟶ *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *get-bit-s Va p* (*ii*−*1*) (*jj div 4*)
proof−
{ **assume** *a06*:*ii*>*0*
then have ¬((*ii* − *1*) = *?i1* ∧ *jj div 4* = *?j1*)
using *a050' a051' from-l-suc* **by** *fastforce*
moreover have ∀*j*. *j*≥ *?j2* ∧ *j*≤ *?j2*+*3* ⟶ ¬((*ii* − *1*) = *?i2* ∧ *jj div 4* = *j*)
using *a051' a052' from-l-gt0* **by** *fastforce*
ultimately have *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *get-bit-s Va p* (*ii*−*1*) (*jj*

div 4)
  using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*,
    of ii −1 jj div 4] **by** *auto*
  } **thus** *?thesis* **by** *auto* **qed**
  have *eq-get-bit-i2-j2*:∀ *j*. *j*≥ (*jj* ∗ *4*) ∧ *j*≤ *Suc*(*Suc* (*Suc* (*jj* ∗ *4*))) ⟶
    get-bit-s x p (*ii*+*1*) *j* = *get-bit-s Va p* (*ii*+*1*) *j*
  proof−
   { **fix** *j*
    assume *a00*:*j*≥ (*jj* ∗ *4*) ∧ *j*≤ (*jj* ∗ *4*)+*3*
    then have *n1*:¬((*ii* + *1*) = *?i1* ∧ *j* = *?j1*)
     using *a053′ from-l-suc* **by** *auto*
    have *n2*:∀ *j*. *j*≥ *?j2* ∧ *j*≤ *?j2*+*3* ⟶ ¬((*ii* + *1*) = *?i2* ∧ *jj* ∗ *4* = *j*)
    using *a050′ from-l-gt0* **by** *fastforce*
    have *get-bit-s x p* (*ii*+*1*) *j* = *get-bit-s Va p* (*ii*+*1*) *j*
   using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*,
    of ii + *1 j*] *n1 n2 a00*
     apply (*cases j*=*jj*∗*4*) **by** *auto*
   } **thus** *?thesis* **by** *auto*
  qed
  { **assume** *a06*: *get-bit-s x p ii jj* = *FREE* ∨
      get-bit-s x p ii jj = *FREEING* ∨
      get-bit-s x p ii jj = *ALLOCATED* ∨
      get-bit-s x p ii jj = *ALLOCATING*
   { **assume** *a07*: *NULL* < *ii*
    then have *get-bit-s x p* (*ii* − *1*) (*jj div 4*) = *DIVIDED*
     using *a06 a07 eq-get-bit-i1-j1 eq-get-bit-i-j*
     using *inv-va* **by** *auto*
   }
   moreover {
    assume *a07*:*ii* < *length* (*levels* (*mem-pool-info x p*)) − *1*
    then have *ilen*:*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
     by (*simp add*: *len-levels*)
    have *get-bit-s Va p ii jj* = *FREE* ∨
      get-bit-s Va p ii jj = *FREEING* ∨
      get-bit-s Va p ii jj = *ALLOCATED* ∨
     get-bit-s Va p ii jj = *ALLOCATING* **using** *eq-get-bit-i-j a06* **by** *auto*
    then have *noexist-bits* (*mem-pool-info Va p*) (*ii* + *1*) (*jj* ∗ *4*)
     using *ilen inv-va*
     by *simp*
    then have *noexist-bits* (*mem-pool-info x p*) (*ii* + *1*) (*jj* ∗ *4*)
     using *eq-get-bit-i2-j2* **by** (*simp add*: *numeral-3-eq-3*)
   }
   ultimately have *?thesis* **by** *auto*
  } **then have** *?thesis* **by** *auto*
 }
 moreover {
  assume *a06*:(*ii*=*?i1* ∧ *jj*=*?j1*)

```
    then have  get-bit-s x p ii jj = DIVIDED
      using get-bit-x-l-b a14 a18 from-l from-l-gt0 i1-len j1-len by presburger
    then have ?thesis by auto
  }
  moreover {
    assume a06: (ii=?i2 ∧ jj≥ ?j2 ∧ jj<?j2+4)
    then have a06':jj=?j2 ∨ jj=?j2+1 ∨ jj=?j2+2 ∨ jj =?j2 + 3 by auto
    { assume a07:NULL < ii
      { assume a08:jj=?j2
        then have  get-bit:get-bit-s x p ii jj = ALLOCATING
        using a02 a06 a15 eq-len get-bit-x-l1-b4 i2-len from-l-gt0 i1-len j1-len
        by (metis  mult.commute)
        then have get-bit-s x p (ii−1) (jj div 4) = DIVIDED
          using a06 a08 get-bit-x-l-b a14 a18 from-l from-l-gt0 i1-len j1-len
          by (simp add: a18  i1-len j1-len from-l-suc)
      }
      moreover {
        assume a07:jj≠?j2
        have  a07':jj div 4 = ?j1 using a06 a07 by auto
        have  get-bit-s x p ii jj = FREE
       using  a06 a02 a15  a07  from-l mp-alloc-stm4-inv-bits-len a18 mp-alloc-stm4-pre-precond-f-bn
          by (auto simp add:   mp-alloc-stm4-pre-precond-f-bn)
        have get-bit-s x p (ii−1) (jj div 4) = DIVIDED
          using a06 a07' a14 a18 from-l from-l-gt0 i1-len j1-len
          by (simp add: a18 get-bit-x-l-b i1-len j1-len from-l-suc)
      }
      ultimately have get-bit-s x p (ii−1) (jj div 4) = DIVIDED by fastforce
    }
    moreover { assume a07:ii < length (levels (mem-pool-info x p)) − 1
      then have get-s:∀ j. j≥ (jj ∗ 4) ∧ j≤ Suc(Suc (Suc (jj ∗ 4))) ⟶
                 get-bit-s x p (ii+1) j = get-bit-s Va p (ii+1) j
      using same-bit-mp-alloc-x-va[OF a14[simplified a18[simplified mp-alloc-stm4-froml[OF
a18],
                                  THEN sym]] a15, of ii + 1 jj∗4]  a06
           by (metis Suc-1 Suc-eq-plus1 a14 a18 add.right-neutral add-Suc-right
add-left-cancel
            from-l from-l-suc same-bit-mp-alloc-stm4-pre-precond-f1 zero-neq-numeral)
      then have noexist-bits (mem-pool-info x p) (ii + 1) (jj∗4)
        using a07[simplified len-levels]  a06 inv-va nexisti2
              noexists-eq-bits[OF get-s] a06'
        by fastforce
    }
    ultimately have ?thesis by fastforce
  }
  moreover {
    assume a06: (ii=?i2' ∧ jj≥ ?j20' ∧ jj< ?j24')
    then have a06':jj=?j20' ∨ jj=?j20'+1 ∨ jj=?j20'+2 ∨ jj=?j20'+3 ∨
                 jj=?j21' ∨ jj=?j21'+1 ∨ jj=?j21'+2 ∨ jj=?j21'+3 ∨
                 jj=?j22' ∨ jj=?j22'+1 ∨ jj=?j22'+2 ∨ jj=?j22'+3 ∨
```

jj=?j23′ ∨ *jj=?j23′+1* ∨ *jj=?j23′+2* ∨ *jj=?j23′ + 3*
by *presburger*
then have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*],
THEN sym]] *a15*, *of ii jj*] **using** *a06*
by (*simp add*: *from-l-suc*)
have *i2-lt-length*:*?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1* **using** *a06 a01*
by (*simp add*: *len-levels*)
{ **assume** *a07*: *get-bit-s x p ii jj* = *FREE* ∨
get-bit-s x p ii jj = *FREEING* ∨
get-bit-s x p ii jj = *ALLOCATED* ∨
get-bit-s x p ii jj = *ALLOCATING*
have *get-bit-s Va p ii jj* = *NOEXIST*
using *a07 a06 inv-va nexisti3*[*simplified i2-lt-length*] *a06′*
by *auto*
then have *get-bit-s x p ii jj* = *NOEXIST* **using** *eq-get-bit-i-j* **by** *auto*
} **then have** *?thesis* **by** *auto*
}
moreover {
assume *a06*: (*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*)
then have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*],
THEN sym]] *a15*, *of ii jj*]
by *linarith*
then have *get-bit-divided*:*get-bit-s x p ii jj* = *DIVIDED* **using** *a06 alloc-predi1-j1* **by** *simp*
{ **assume** *a06*: *get-bit-s x p ii jj* = *FREE* ∨ *get-bit-s x p ii jj* = *FREEING* ∨
get-bit-s x p ii jj = *ALLOCATED* ∨ *get-bit-s x p ii jj* = *ALLOCATING*
then have *?thesis* **using** *get-bit-divided* **by** *auto*
} **then have** *?thesis* **by** *fastforce*
}
ultimately show *?thesis* **by** *fastforce*
qed

lemma *mp-alloc-stm4-inv-bitmap2*:
assumes
a0:*inv Va* **and**
a1:*p* ∈ *mem-pools Va* **and**
a2:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a3:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:¬ *free-l Va t* < *OK* **and**
a6:*free-l Va t* ≤ *from-l Va t* **and**
a7:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),

block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))⦈ **and**
a8:*n* = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*) = *NULL* **and**
a9:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a10:*from-l Va t* < *alloc-l Va t* **and**
a11:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)) **and**
a12:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a13:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a14:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3* *FREE* **and**
a15:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a16:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a17:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a01:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a02:*jj* < *length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) **and**
a03:*get-bit-s x p ii jj* = *DIVIDED* **and**
a04:*0* < *ii*
shows *get-bit-s x p* (*ii* − *1*) (*jj div 4*) = *DIVIDED*
proof−
let *?mp* = *mem-pool-info x p*

have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
  using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a6 a5* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
  using *mp-alloc-stm4-lvl-len*[*OF a1 a12*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
  using *mp-alloc-stm4-maxsz*[*OF a1 a12*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
  using *mp-alloc-stm4-buf*[*OF a1 a12*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
  using *mp-alloc-stm4-froml*[*OF a17*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
  using *from-l-gt0* **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a12*] **by** *auto*
have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a16*
  by *auto*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*) **and**
?i1′ = (*nat* (*from-l Va t*)) − *1* **and**
?j1′ = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *div 4* **and**
?i2′ = (*nat* (*from-l Va t*)) + *2*
let *?j20′* = *?j2* ∗ *4* **and** *?j21′* = (*?j2*+*1*) ∗ *4* **and** *?j22′* = (*?j2*+*2*)∗*4* **and**
  ?j23′ = (*?j2*+*3*)∗*4* **and** *?j24′* = (*?j2*+*4*)∗*4*
let *?mp* = *mem-pool-info x p*
have *inv-mempool-info-mp Va p*
  using *a1 mem-pools inv* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
  using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
  using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
  by (*metis i1-len a9 a11 a1 inv inv-mempool-info-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
  using *i1-len i2-len j1-len inv-mempool from-l-suc*
  by *simp*
let *?bts* = *bits* (*levels ?mp* ! *ii*)
let *?btsva* = (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))

have *a01′*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a01 len-levels* **by** *auto*
then have *inv-bitmap1*:
∀ *j* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)).
(*?btsva* ! *j* = *FREE* ∨ *?btsva* ! *j* = *FREEING* ∨ *?btsva* ! *j* = *ALLOCATED* ∨ *?btsva* ! *j* = *ALLOCATING* ⟶
(*ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*j*∗*4*)))
∧ (*?btsva* ! *j* = *DIVIDED* ⟶ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*?btsva* ! *j* = *NOEXIST* ⟶ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*j*∗*4*))
∧ (*?btsva* ! *j* = *NOEXIST* ∧ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) ≠ *DIVIDED*)
using *inv mem-pools a1*
unfolding *Let-def inv-bitmap-def*
by *blast*
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a7 a0 a11* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
then have *alloc-predi1-j1*:*?i1* > *0* ⟶ *get-bit-s Va p* (*?i1* − *1*) (*?j1 div 4*) = *DIVIDED*
using *inv-bitmap1 i1-len j1-len inv a1* **unfolding** *Let-def inv-bitmap-def* **by** *blast*
have *nexisti2*:*noexist-bits* (*mem-pool-info Va p*) *?i2 ?j2*
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*] *i1-len j1-len*
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less plus-1-eq-Suc*)
have *nexisti3*:*?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j20′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j21′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j22′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j23′*
proof−
{ assume *?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
then have *a00*:∀ *j*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*)).
get-bit-s Va p ?i2 j = *NOEXIST* ⟶ *noexist-bits* (*mem-pool-info Va p*) *?i2′* (*j* ∗ *4*)
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*] *i2-len*
from-l-suc **by** *auto*
then have *noexist-bits* (*mem-pool-info Va p*) *?i2′ ?j20′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j21′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j22′* ∧

noexist-bits (*mem-pool-info Va p*) *?i2′ ?j23′*
using *j2-len nexisti2 Suc-lessD*
by (*smt One-nat-def add.commute add-2-eq-Suc′ add-Suc-right numeral-3-eq-3 plus-1-eq-Suc*)
}
thus *?thesis* **by** *fastforce*
qed
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
have *a02′*:*jj* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a02 a12* **unfolding** *gvars-conf-def gvars-conf-stable-def*
by (*simp add*: *mp-alloc-stm4-inv-bits-len*)
have *eq-len*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *mp-alloc-stm4-inv-bits-len a13 a14 length-list-update-n*
by *metis*
have *inv-va*:(*?btsva* ! *jj* = *FREE* ∨ *?btsva* ! *jj* = *FREEING* ∨ *?btsva* ! *jj* = *ALLOCATED* ∨ *?btsva* ! *jj* = *ALLOCATING* ⟶
(*ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*)))
∧ (*?btsva* ! *jj* = *DIVIDED* ⟶ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*?btsva* ! *jj* = *NOEXIST* ⟶ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*))
∧ (*?btsva* ! *jj* = *NOEXIST* ∧ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) ≠ *DIVIDED*)
using *inv-bitmap1 a02′* **by** *auto*
{ assume *a05*:¬((*ii*=*?i1* ∧ *jj*=*?j1*) ∨
(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*< *?j2*+*4*) ∨
(*ii*=*?i2′* ∧ *jj*≥ *?j20′* ∧ *jj*< *?j24′*) ∨
(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*))
then have *a050′*:¬(*ii*=*?i1* ∧ *jj*=*?j1*) **and**
a051′: ¬(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*<*?j2* + *4*) **and**
a052′:¬(*ii*=*?i2′* ∧ *jj*≥ *?j20′* ∧ *jj*< *?j24′*) **and**
a053′: ¬(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*)
by *force+*
have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*], *THEN sym*]] *a14*, *of ii jj*]
using *a050′ a051′* **by** *auto*
have *eq-get-bit-i1-j1*:*ii*>*0* ⟶ *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *get-bit-s Va p* (*ii*−*1*) (*jj div 4*)
proof−
{ assume *a06*:*ii*>*0*
then have ¬((*ii* − *1*) = *?i1* ∧ *jj div 4* = *?j1*)
using *a050′ a051′ from-l-suc* **by** *fastforce*
moreover have ∀ *j*. *j*≥ *?j2* ∧ *j*≤ *?j2*+*3* ⟶ ¬((*ii* − *1*) = *?i2* ∧ *jj div 4* =

j)
using *a051′ a052′ from-l-gt0* by *fastforce*
ultimately have *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *get-bit-s Va p* (*ii*−*1*) (*jj div 4*)
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*], *THEN sym*]] *a14*,
of ii −*1 jj div 4*] by *auto*
} thus *?thesis* by *auto* qed
have *eq-get-bit-i2-j2*:∀ *j*. *j*≥ (*jj* ∗ *4*) ∧ *j*≤ *Suc*(*Suc* (*Suc* (*jj* ∗ *4*))) ⟶
get-bit-s x p (*ii*+*1*) *j* = *get-bit-s Va p* (*ii*+*1*) *j*
proof−
{ fix *j*
assume *a00*:*j*≥ (*jj* ∗ *4*) ∧ *j*≤ (*jj* ∗ *4*)+*3*
then have *n1*:¬((*ii* + *1*) = *?i1* ∧ *j* = *?j1*)
using *a053′ from-l-suc* by *auto*
have *n2*:∀ *j*. *j*≥ *?j2* ∧ *j*≤ *?j2*+*3* ⟶ ¬((*ii* + *1*) = *?i2* ∧ *jj* ∗ *4* = *j*)
using *a050′ from-l-gt0* by *fastforce*
have *get-bit-s x p* (*ii*+*1*) *j* = *get-bit-s Va p* (*ii*+*1*) *j*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*], *THEN sym*]] *a14*,
of ii + *1 j*] *n1 n2 a00*
apply (*cases j*=*jj*∗*4*) by *auto*
} thus *?thesis* by *auto*
qed
then have *?thesis*
using *a03 a04 eq-get-bit-i1-j1 eq-get-bit-i-j inv-va* by *auto*
}
moreover {
assume *a06*:(*ii*=*?i1* ∧ *jj*=*?j1*)
then have *?thesis* using *a03 a04*
by (*metis Suc-eq-plus1 Suc-pred a13 a17*
add.commute add-2-eq-Suc′ add-cancel-right-right
alloc-predi1-j1 from-l from-l-suc
same-bit-mp-alloc-stm4-pre-precond-f zero-neq-numeral)
}
moreover {
assume *a06*: (*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*<*?j2*+*4*)
then have *a06′*:*jj*=*?j2* ∨ *jj*=*?j2*+*1* ∨ *jj*=*?j2*+*2* ∨ *jj* =*?j2* + *3* by *auto*
have *l1*:(*ii* − *1*) = *?i1* ∧ (*jj div 4*) = *?j1*
using *a2 a03 a04 a01 a02 a02′ a06 a13 a14 a06′*
from-l-gt0 from-l-suc
by (*metis add.commute add-mult-distrib2*
diff-add-inverse2 div-nat-eqI mult.commute
nat-mult-1-right plus-1-eq-Suc)
then have *?thesis* using *get-bit-x-l-b*[*OF l1*] *a13*
from-l from-l-gt0 from-l-suc i1-len
by (*simp add*: *a17 j1-len l1*)
}
moreover {

assume *a06*: (*ii*=*?i2′* ∧ *jj*≥ *?j20′* ∧ *jj*< *?j24′*)
then have *a06′*:*jj*=*?j20′* ∨ *jj*=*?j20′*+*1* ∨ *jj*=*?j20′*+*2* ∨ *jj*=*?j20′*+*3* ∨
jj=*?j21′* ∨ *jj*=*?j21′*+*1* ∨ *jj*=*?j21′*+*2* ∨ *jj*=*?j21′*+*3* ∨
jj=*?j22′* ∨ *jj*=*?j22′*+*1* ∨ *jj*=*?j22′*+*2* ∨ *jj*=*?j22′*+*3* ∨
jj=*?j23′* ∨ *jj*=*?j23′*+*1* ∨ *jj*=*?j23′*+*2* ∨ *jj*=*?j23′* + *3*
by *presburger*
then have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*],
THEN sym]] *a14*, *of ii jj*] **using** *a06*
by (*simp add*: *from-l-suc*)
moreover have *i2-lt-length*:*?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
using
a06[*simplified len-levels*] *a01*[*simplified len-levels*]
by *simp*
then have *get-bit-s Va p ii jj* = *NOEXIST*
using *a06 a01 a06 inv-va nexisti3 a06′ a06*[*simplified len-levels*] *a01*[*simplified len-levels*]
by *auto*
ultimately have *?thesis*
using *a03* **by** *auto*
}
moreover {
assume *a06*: (*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*)
then have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*],
THEN sym]] *a14*, *of ii jj*]
by *linarith*
have *?thesis* **using** *a03 a04 a13 a17 from-l inv-va same-bit-mp-alloc-stm4-pre-precond-f1*
calculation(*1*) *calculation*(*2*) *calculation*(*3*) *calculation*(*4*)
by (*smt Suc-pred add-diff-cancel-left′ int-nat-eq inv-va of-nat-Suc plus-1-eq-Suc*)
}
ultimately show *?thesis* **by** *fastforce*
qed

lemma *mp-alloc-stm4-inv-bitmap3*:
assumes
a0:*inv Va* **and**
a1:*p* ∈ *mem-pools Va* **and**
a2:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:¬ *free-l Va t* < *OK* **and**
a6:*free-l Va t* ≤ *from-l Va t* **and**
a7:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*

4 ^ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t)),
data = buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t))⦈ **and**
a8:n = block-num (mem-pool-info Va p)
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t)) ∨
max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t) = NULL **and**
a9:block-num (mem-pool-info Va p)
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t))
< n-max (mem-pool-info Va p) ∗ 4 ^ nat (from-l Va t) **and**
a10:from-l Va t < alloc-l Va t **and**

a11:n < n-max (mem-pool-info Va p) ∗ 4 ^ nat (from-l Va t) **and**
a12:blk Va t = buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)) **and**
a13:(x, mp-alloc-stm4-pre-precond-f Va t p) ∈ gvars-conf-stable **and**
a14:∀ jj. jj ≠ nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1) ⟶
levels (mem-pool-info x p) ! jj = levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) ! jj **and**
a15:bits (levels (mem-pool-info x p) ! nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)) =
list-updates-n
(bits (levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) !
nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)))
(Suc (bn (mp-alloc-stm4-pre-precond-f Va t p) t ∗ 4)) 3 FREE **and**
a16:free-list (levels (mem-pool-info x p) ! nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)) =
inserts
(map (λii. lsizes (mp-alloc-stm4-pre-precond-f Va t p) t !
nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1) ∗
ii +
blk (mp-alloc-stm4-pre-precond-f Va t p) t)
[Suc NULL..<4])
(free-list
(levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) !
nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1))) **and**
a17:lsizes x = lsizes (mp-alloc-stm4-pre-precond-f Va t p) **and**
a18:from-l x = from-l (mp-alloc-stm4-pre-precond-f Va t p) **and**
a01:ii < length (levels (mem-pool-info x p)) **and**
a02:jj < length (bits (levels (mem-pool-info x p) ! ii)) **and**
a03:get-bit-s x p ii jj = NOEXIST **and**
a04:ii < length (levels (mem-pool-info x p)) − 1
shows *noexist-bits (mem-pool-info x p) (ii + 1) (jj ∗ 4)*
proof−
let *?mp = mem-pool-info x p*

have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
  using *a0* **unfolding** *inv-def* **by** *auto*
 have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a6 a5* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
  using *mp-alloc-stm4-lvl-len*[*OF a1 a13*] **by** *simp*
 have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
  using *mp-alloc-stm4-maxsz*[*OF a1 a13*] **by** *simp*
 have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
  using *mp-alloc-stm4-buf*[*OF a1 a13*] **by** *simp*
 have *from-l*:*from-l x* = *from-l Va*
  using *mp-alloc-stm4-froml*[*OF a18*] **by** *auto*
 have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
  using *from-l-gt0* **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a13*] **by** *auto*
 have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a17*
  by *auto*
 let *?i1*=(*nat* (*from-l Va t*)) **and**
 ?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
 ?i2 = (*nat* (*from-l Va t* + *1*)) **and**
 ?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*) **and**
 ?i1' = (*nat* (*from-l Va t*)) − *1* **and**
 ?j1' = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *div 4* **and**
 ?i2' = (*nat* (*from-l Va t*)) + *2*
 let *?j20'* = *?j2* ∗ *4* **and** *?j21'* = (*?j2*+*1*) ∗ *4* **and** *?j22'* = (*?j2*+*2*)∗*4* **and**
   ?j23' = (*?j2*+*3*)∗*4* **and** *?j24'* = (*?j2*+*4*)∗*4*
 let *?mp* = *mem-pool-info x p*
 have *inv-mempool-info-mp Va p*
  using *a1 mem-pools inv* **unfolding** *inv-mempool-info-def* **by** *auto*
 note *inv-mempool*=*this*[*simplified Let-def*]
 have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
  using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
 have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
  using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
 have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
  by (*metis i1-len a9 a12 a1 inv inv-mempool-info-def*)
 have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
  using *i1-len i2-len j1-len inv-mempool from-l-suc*
  by *simp*
 let *?bts* = *bits* (*levels ?mp* ! *ii*)
 let *?btsva* = (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))

have *a01′:ii < length (levels (mem-pool-info Va p))*
using *a01 len-levels* **by** *auto*
then have *inv-bitmap1*:
∀ *j < length (bits (levels (mem-pool-info Va p) ! ii))*.
(*?btsva ! j = FREE ∨ ?btsva ! j = FREEING ∨ ?btsva ! j = ALLOCATED*
∨ ?btsva ! j = ALLOCATING ⟶
(*ii > 0* ⟶ (*bits (levels (mem-pool-info Va p) ! (ii − 1))) ! (j div*
4) = DIVIDED)
∧ (*ii < length (levels (mem-pool-info Va p)) − 1* ⟶ *noexist-bits*
(mem-pool-info Va p) (ii+1) (j∗4)))
∧ (*?btsva ! j = DIVIDED* ⟶ *ii > 0* ⟶ (*bits (levels (mem-pool-info Va*
p) ! (ii − 1))) ! (j div 4) = DIVIDED)
∧ (*?btsva ! j = NOEXIST* ⟶ *ii < length (levels (mem-pool-info Va p))*
− 1
⟶ *noexist-bits (mem-pool-info Va p) (ii+1) (j∗4)*)
∧ (*?btsva ! j = NOEXIST ∧ ii > 0* ⟶ (*bits (levels (mem-pool-info Va*
p) ! (ii − 1))) ! (j div 4) ≠ DIVIDED)
using *inv mem-pools a1*
unfolding *Let-def inv-bitmap-def*
by *blast*
have *alloc-i1-j1:get-bit-s Va p ?i1 ?j1 = ALLOCATING*
using *a7 a0 a12* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis (no-types) Mem-block.select-convs(1) Mem-block.select-convs(2) Mem-block.select-convs(3)*)
then have *alloc-predi1-j1:?i1 > 0* ⟶ *get-bit-s Va p (?i1 − 1) (?j1 div 4) =*
DIVIDED
using *inv-bitmap1 i1-len j1-len inv a1* **unfolding** *Let-def inv-bitmap-def* **by**
blast
have *nexisti2:noexist-bits (mem-pool-info Va p) ?i2 ?j2*
using *a1 conjunct1[OF conjunct2[OF inv], simplified Let-def inv-bitmap-def]*
i1-len j1-len
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less*
plus-1-eq-Suc)
have *nexisti3:?i2 < length (levels (mem-pool-info Va p)) − 1* ⟶
noexist-bits (mem-pool-info Va p) ?i2′ ?j20′ ∧
noexist-bits (mem-pool-info Va p) ?i2′ ?j21′ ∧
noexist-bits (mem-pool-info Va p) ?i2′ ?j22′ ∧
noexist-bits (mem-pool-info Va p) ?i2′ ?j23′
proof−
{ assume *?i2 < length (levels (mem-pool-info Va p)) − 1*
then have *a00:∀ j<length (bits (levels (mem-pool-info Va p) ! ?i2))*.
get-bit-s Va p ?i2 j = NOEXIST ⟶ *noexist-bits (mem-pool-info Va*
p) ?i2′ (j ∗ 4)
using *a1 conjunct1[OF conjunct2[OF inv], simplified Let-def inv-bitmap-def]*
i2-len
from-l-suc **by** *auto*
then have *noexist-bits (mem-pool-info Va p) ?i2′ ?j20′* ∧
noexist-bits (mem-pool-info Va p) ?i2′ ?j21′ ∧
noexist-bits (mem-pool-info Va p) ?i2′ ?j22′ ∧

noexist-bits (*mem-pool-info Va p*) *?i2′ ?j23′*
using *j2-len nexisti2 Suc-lessD*
by (*smt One-nat-def add.commute add-2-eq-Suc′ add-Suc-right numeral-3-eq-3 plus-1-eq-Suc*)
}
thus *?thesis* **by** *fastforce*
qed
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
have *a02′*:*jj* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a02 a13* **unfolding** *gvars-conf-def gvars-conf-stable-def*
by (*simp add*: *mp-alloc-stm4-inv-bits-len*)
have *eq-len*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *mp-alloc-stm4-inv-bits-len a14 a15 length-list-update-n*
by *metis*
have *inv-va*:(*?btsva* ! *jj* = *FREE* ∨ *?btsva* ! *jj* = *FREEING* ∨ *?btsva* ! *jj* = *ALLOCATED* ∨ *?btsva* ! *jj* = *ALLOCATING* ⟶
(*ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*)))
∧ (*?btsva* ! *jj* = *DIVIDED* ⟶ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*?btsva* ! *jj* = *NOEXIST* ⟶ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*))
∧ (*?btsva* ! *jj* = *NOEXIST* ∧ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) ≠ *DIVIDED*)
using *inv-bitmap1 a02′* **by** *auto*
{ **assume** *a05*:¬((*ii*=*?i1* ∧ *jj*=*?j1*) ∨
(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*< *?j2*+*4*) ∨
(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*))
then have *a050′*:¬(*ii*=*?i1* ∧ *jj*=*?j1*) **and**
a051′: ¬(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*<*?j2* + *4*) **and**
a053′: ¬(*?i1* >*0* ∧ *ii* = (*?i1* − *1*) ∧ *jj* = *?j1 div 4*)
by *force*+
have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*, *of ii jj*]
using *a050′ a051′* **by** *auto*
have *eq-get-bit-i2-j2*:∀ *j*. *j*≥ (*jj* ∗ *4*) ∧ *j*≤ *Suc*(*Suc* (*Suc* (*jj* ∗ *4*))) ⟶
get-bit-s x p (*ii*+*1*) *j* = *get-bit-s Va p* (*ii*+*1*) *j*
proof−
{ **fix** *j*
assume *a00*:*j*≥ (*jj* ∗ *4*) ∧ *j*≤ (*jj* ∗ *4*)+*3*
then have *n1*:¬((*ii* + *1*) = *?i1* ∧ *j* = *?j1*)
using *a053′ from-l-suc* **by** *auto*
have *n2*:∀ *j*. *j*≥ *?j2* ∧ *j*≤ *?j2*+*3* ⟶ ¬((*ii* + *1*) = *?i2* ∧ *jj* ∗ *4* = *j*)
using *a050′ from-l-gt0* **by** *fastforce*

have *get-bit-s x p (ii+1) j = get-bit-s Va p (ii+1) j*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*,
of ii + 1 j] *n1 n2 a00*
apply (*cases j=jj*4*) **by** *auto*
} **thus** *?thesis* **by** *auto*
qed
then have *?thesis*
using *a04 len-levels eq-get-bit-i-j a03 inv-va* **by** (*simp add*: *numeral-3-eq-3*)
}
moreover {
assume *a06*:(*ii=?i1* ∧ *jj=?j1*)
then have *get-bit-s x p ii jj = DIVIDED*
using *get-bit-x-l-b a14 a18 from-l from-l-gt0 i1-len j1-len* **by** *presburger*
then have *?thesis* **using** *a03* **by** *auto*
}
moreover {
assume *a06*: (*ii=?i2* ∧ *jj≥ ?j2* ∧ *jj<?j2+4*)
then have *a06′*:*jj=?j2* ∨ *jj=?j2+1* ∨ *jj=?j2+2* ∨ *jj =?j2 + 3* **by** *auto*
then have *get-s*:∀ *j*. *j≥* (*jj * 4*) ∧ *j≤ Suc*(*Suc* (*Suc* (*jj * 4*))) ⟶
get-bit-s x p (ii+1) j = get-bit-s Va p (ii+1) j
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*],
THEN sym]] *a15*, *of ii + 1 jj*4*] *a06*
by (*metis Suc-1 Suc-eq-plus1 a14 a18 add.right-neutral add-Suc-right add-left-cancel from-l from-l-suc same-bit-mp-alloc-stm4-pre-precond-f1 zero-neq-numeral*)
then have *?thesis*
using *a04*[*simplified len-levels*] *a06 inv-va nexisti2*
noexists-eq-bits[*OF get-s*] *a06′*
by *fastforce*
}
moreover {
assume *a06*: (*?i1 >0* ∧ *ii* = (*?i1 − 1*) ∧ *jj* = *?j1 div 4*)
then have *eq-get-bit-i-j*:*get-bit-s x p ii jj = get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified a18*[*simplified mp-alloc-stm4-froml*[*OF a18*],
THEN sym]] *a15*, *of ii jj*]
by *linarith*
then have *get-bit-divided*:*get-bit-s x p ii jj = DIVIDED* **using** *a06 alloc-predi1-j1* **by** *simp*
then have *?thesis* **using** *get-bit-divided a03* **by** *auto*
}
ultimately show *?thesis* **by** *fastforce*
qed

lemma *mp-alloc-stm4-inv-bitmap4*:
assumes
a0:*inv Va* **and**
a1:*p* ∈ *mem-pools Va* **and**

a2:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* **and**
a3:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:¬ *free-l Va t* < *OK* **and**
a6:*free-l Va t* ≤ *from-l Va t* **and**
a7:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
        block = *block-num* (*mem-pool-info Va p*)
                (*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
                (*lsizes Va t* ! *nat* (*from-l Va t*)),
        data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*))|) **and**
a8:*n* = *block-num* (*mem-pool-info Va p*)
      (*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
      (*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*) = *NULL* **and**
a9:*block-num* (*mem-pool-info Va p*)
  (*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
  (*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ^ *nat* (*from-l Va t*) **and**
a10:*from-l Va t* < *alloc-l Va t* **and**
a11:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) **and**
a12:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a13:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
   levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a14:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
 (*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
        nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
 (*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a15:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
 (*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
            nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
            ii +
            blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
   [*Suc NULL*..<*4*])
 (*free-list*
   (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
    nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a16:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**

a17:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a01:*ii* < *length* (*levels* (*mem-pool-info x p*)) **and**
a02:*jj* < *length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) **and**
a03:*get-bit-s x p ii jj* = *NOEXIST* **and**
a04:*0* < *ii*
shows *get-bit-s x p* (*ii* − *1*) (*jj div 4*) ≠ *DIVIDED*
proof−
let *?mp* = *mem-pool-info x p*
have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a6 a5* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info x p*)) = *length* (*levels* (*mem-pool-info Va p*))
using *mp-alloc-stm4-lvl-len*[*OF a1 a12*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a1 a12*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a1 a12*] **by** *simp*
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a17*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a12*] **by** *auto*
have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a16*
by *auto*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))**4*) **and**
?i1' = (*nat* (*from-l Va t*)) − *1* **and**
?j1' = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *div 4* **and**
?i2' = (*nat* (*from-l Va t*)) + *2*
let *?j20'* = *?j2* ∗ *4* **and** *?j21'* = (*?j2*+*1*) ∗ *4* **and** *?j22'* = (*?j2*+*2*)∗*4* **and**
?j23' = (*?j2*+*3*)∗*4* **and** *?j24'* = (*?j2*+*4*)∗*4*
let *?mp* = *mem-pool-info x p*
have *inv-mempool-info-mp Va p*
using *a1 mem-pools inv* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a1 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*

have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis i1-len a9 a11 a1 inv inv-mempool-info-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
let *?bts* = *bits* (*levels ?mp* ! *ii*)
let *?btsva* = (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
have *a01'*:*ii* < *length* (*levels* (*mem-pool-info Va p*))
using *a01 len-levels* **by** *auto*
then have *inv-bitmap1*:
$\forall$ *j* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*)).
(*?btsva* ! *j* = *FREE* $\vee$ *?btsva* ! *j* = *FREEING* $\vee$ *?btsva* ! *j* = *ALLOCATED* $\vee$ *?btsva* ! *j* = *ALLOCATING* $\longrightarrow$
(*ii* > *0* $\longrightarrow$ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) = *DIVIDED*)
$\wedge$ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* $\longrightarrow$ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*j*∗*4*)))
$\wedge$ (*?btsva* ! *j* = *DIVIDED* $\longrightarrow$ *ii* > *0* $\longrightarrow$ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) = *DIVIDED*)
$\wedge$ (*?btsva* ! *j* = *NOEXIST* $\longrightarrow$ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
$\longrightarrow$ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*j*∗*4*))
$\wedge$ (*?btsva* ! *j* = *NOEXIST* $\wedge$ *ii* > *0* $\longrightarrow$ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*j div 4*) $\neq$ *DIVIDED*)
using *inv mem-pools a1*
unfolding *Let-def inv-bitmap-def*
by *blast*
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a7 a0 a11* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
then have *alloc-predi1-j1*:*?i1* > *0* $\longrightarrow$ *get-bit-s Va p* (*?i1* − *1*) (*?j1 div 4*) = *DIVIDED*
using *inv-bitmap1 i1-len j1-len inv a1* **unfolding** *Let-def inv-bitmap-def* **by** *blast*
have *nexisti2*:*noexist-bits* (*mem-pool-info Va p*) *?i2 ?j2*
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*] *i1-len j1-len*
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less plus-1-eq-Suc*)
have *nexisti3*:*?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1* $\longrightarrow$
noexist-bits (*mem-pool-info Va p*) *?i2' ?j20'* $\wedge$
noexist-bits (*mem-pool-info Va p*) *?i2' ?j21'* $\wedge$
noexist-bits (*mem-pool-info Va p*) *?i2' ?j22'* $\wedge$
noexist-bits (*mem-pool-info Va p*) *?i2' ?j23'*
proof−
{ **assume** *?i2* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
then have *a00*:$\forall$ *j*<*length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*)).

get-bit-s Va p ?i2 j = NOEXIST ⟶ *noexist-bits* (*mem-pool-info Va p*) *?i2′* (*j* ∗ *4*)
using *a1 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*] *i2-len*
from-l-suc **by** *auto*
then have *noexist-bits* (*mem-pool-info Va p*) *?i2′ ?j20′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j21′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j22′* ∧
noexist-bits (*mem-pool-info Va p*) *?i2′ ?j23′*
using *j2-len nexisti2 Suc-lessD*
by (*smt One-nat-def add.commute add-2-eq-Suc′ add-Suc-right numeral-3-eq-3 plus-1-eq-Suc*)
}
thus *?thesis* **by** *fastforce*
qed
let *?bts* = *bits* (*levels ?mp* ! *ii*) **and** *?fl* = *free-list* (*levels ?mp* ! *ii*)
have *a02′*:*jj* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *a02 a12* **unfolding** *gvars-conf-def gvars-conf-stable-def*
by (*simp add*: *mp-alloc-stm4-inv-bits-len*)
have *eq-len*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *ii*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *ii*))
using *mp-alloc-stm4-inv-bits-len a13 a14 length-list-update-n*
by *metis*
have *inv-va*:(*?btsva* ! *jj* = *FREE* ∨ *?btsva* ! *jj* = *FREEING* ∨ *?btsva* ! *jj* = *ALLOCATED* ∨ *?btsva* ! *jj* = *ALLOCATING* ⟶
(*ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*)))
∧ (*?btsva* ! *jj* = *DIVIDED* ⟶ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) = *DIVIDED*)
∧ (*?btsva* ! *jj* = *NOEXIST* ⟶ *ii* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*ii*+*1*) (*jj*∗*4*))
∧ (*?btsva* ! *jj* = *NOEXIST* ∧ *ii* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*ii* − *1*))) ! (*jj div 4*) ≠ *DIVIDED*)
using *inv-bitmap1 a02′* **by** *auto*
{ assume *a05*:¬((*ii*=*?i1* ∧ *jj*=*?j1*) ∨
(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*< *?j2*+*4*) ∨
(*ii*=*?i2′* ∧ *jj*≥ *?j20′* ∧ *jj*< *?j24′*))
then have *a050′*:¬(*ii*=*?i1* ∧ *jj*=*?j1*) **and**
a051′: ¬(*ii*=*?i2* ∧ *jj*≥ *?j2* ∧ *jj*<*?j2* + *4*) **and**
a052′:¬(*ii*=*?i2′* ∧ *jj*≥ *?j20′* ∧ *jj*< *?j24′*)
by *force+*
have *eq-get-bit-i-j*:*get-bit-s x p ii jj* = *get-bit-s Va p ii jj*
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*], *THEN sym*]] *a14*, *of ii jj*]
using *a050′ a051′* **by** *auto*
have *eq-get-bit-i1-j1*:*ii*>*0* ⟶ *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *get-bit-s Va p*

$(ii-1)$ $(jj\ div\ 4)$
proof−
{ assume *a06*:$ii>0$
then have $\neg((ii - 1) = ?i1 \wedge jj\ div\ 4 = ?j1)$
using *a050′ a051′ from-l-suc* **by** *fastforce*
moreover have $\forall j.\ j\geq\ ?j2 \wedge j\leq\ ?j2+3 \longrightarrow \neg((ii - 1) = ?i2 \wedge jj\ div\ 4 = j)$
using *a051′ a052′ from-l-gt0* **by** *fastforce*
ultimately have *get-bit-s x p* $(ii-1)$ $(jj\ div\ 4)$ = *get-bit-s Va p* $(ii-1)$ $(jj\ div\ 4)$
using *same-bit-mp-alloc-x-va*[*OF a13*[*simplified a17*[*simplified mp-alloc-stm4-froml*[*OF a17*], *THEN sym*]] *a14*,
of ii −1 jj div 4] **by** *auto*
} thus *?thesis* **by** *auto* **qed**
then have *?thesis*
using *a03 a04 eq-get-bit-i1-j1 eq-get-bit-i-j inv-va* **by** *auto*
}
moreover {
assume *a06*:$(ii=?i1 \wedge jj=?j1)$
then have *get-bit-s x p ii jj = DIVIDED*
using *get-bit-x-l-b a13 a17 from-l from-l-gt0 i1-len j1-len* **by** *presburger*
then have *?thesis* **using** *a03* **by** *auto*
}
moreover {
assume *a06*: $(ii=?i2 \wedge jj\geq\ ?j2 \wedge jj<?j2+4)$
then have *a06′*:$jj=?j2 \vee jj=?j2+1 \vee jj=?j2+2 \vee jj =?j2 + 3$ **by** *auto*
{ assume *a08*:$jj=?j2$
then have *get-bit*:*get-bit-s x p ii jj = ALLOCATING*
using *a02 a06 a14 eq-len get-bit-x-l1-b4 a04 i2-len from-l-gt0 i1-len j1-len*
by (*metis mult.commute*)
then have *?thesis* **using** *a03* **by** *auto*
}
moreover {
assume *a07*:$jj\neq ?j2$
have *a07′*:$jj\ div\ 4 = ?j1$ **using** *a06 a07* **by** *auto*
have *get-bit-s x p ii jj = FREE*
using *a06 a02 a14 a07 from-l mp-alloc-stm4-inv-bits-len a17 mp-alloc-stm4-pre-precond-f-bn*
by (*auto simp add*: *mp-alloc-stm4-pre-precond-f-bn*)
then have *?thesis* **using** *a03* **by** *auto*
}
ultimately have *?thesis* **using** *a06* **by** *fastforce*
}
moreover {
assume *a06*: $(ii=?i2' \wedge jj\geq\ ?j20' \wedge jj<\ ?j24')$
then have *a06′*:$jj=?j20' \vee jj=?j20'+1 \vee jj=?j20'+2 \vee jj=?j20'+3 \vee$
$jj=?j21' \vee jj=?j21'+1 \vee jj=?j21'+2 \vee jj=?j21'+3 \vee$
$jj=?j22' \vee jj=?j22'+1 \vee jj=?j22'+2 \vee jj=?j22'+3 \vee$
$jj=?j23' \vee jj=?j23'+1 \vee jj=?j23'+2 \vee jj=?j23' + 3$
by *presburger*

have *ij*:(*ii*−*1* = *?i2* ∧ (*jj div 4*)≥ *?j2* ∧ (*jj div 4*) ≤ *?j2* + *3*)
using *a04 a06 from-l-gt0* **by** *auto*
{ assume *a08*:(*jj div 4*)=*?j2*
then have *get-bit*:*get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *ALLOCATING*
using *ij a02 a14 eq-len get-bit-x-l1-b4 a04 i2-len from-l-gt0 i1-len j1-len*
by (*metis Suc-lessD j2-len mult.commute*)
then have *?thesis* **using** *a03* **by** *auto*
}
moreover {
assume *a07*:(*jj div 4*)≠*?j2*
then have *ii*−*1* = *?i2* ∧ (*jj div 4* = *Suc ?j2* ∨ *jj div 4* = *Suc* (*Suc ?j2*) ∨
jj div 4 = *Suc* (*Suc* (*Suc?j2*)))
using *ij* **by** *auto*
then have *get-bit-s x p* (*ii*−*1*) (*jj div 4*) = *FREE*
using *ij a01 a02 i2-len j2-len*
get-bit-x-l1-b41[*OF - from-l-gt0*[*simplified from-l a17*]
a13[*simplified a17*[*THEN sym*] *from-l*] *a14*,*of ii*−*1 jj*
div 4]
by (*metis Suc-lessD mult.commute*)
then have *?thesis* **using** *a03* **by** *auto*
}
ultimately have *?thesis* **using** *a06* **by** *fastforce*
}
ultimately show *?thesis* **by** *fastforce*
qed

lemma *mp-alloc-stm4-inv-bitmap*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:*ETIMEOUT* ≤ *timeout* **and**
a4:*timeout* = *ETIMEOUT* ⟶ *tmout Va t* = *ETIMEOUT* **and**
a5:¬ *rf Va t* **and**
a6:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info*
Va p)) *div 4* ˆ *ii* **and**
a7:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a8:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a9:¬ *free-l Va t* < *OK* **and**
a10:*NULL* < *buf* (*mem-pool-info Va p*) ∨ *NULL* < *n* ∧ *NULL* < *max-sz* (*mem-pool-info*
Va p) *div 4* ˆ *nat* (*from-l Va t*) **and**
a11:*free-l Va t* ≤ *from-l Va t* **and**
a12:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*
4 ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*

$4 \text{ ^ } nat\ (from\text{-}l\ Va\ t))\rrparenthesis$ **and**
$a13{:}n = block\text{-}num\ (mem\text{-}pool\text{-}info\ Va\ p)$
$(buf\ (mem\text{-}pool\text{-}info\ Va\ p) + n * (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t)))$
$(lsizes\ Va\ t\ !\ nat\ (from\text{-}l\ Va\ t)) \lor$
$max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t) = NULL$ **and**
$a14{:}block\text{-}num\ (mem\text{-}pool\text{-}info\ Va\ p)$
$(buf\ (mem\text{-}pool\text{-}info\ Va\ p) + n * (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t)))$
$(lsizes\ Va\ t\ !\ nat\ (from\text{-}l\ Va\ t))$
$< n\text{-}max\ (mem\text{-}pool\text{-}info\ Va\ p) * 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t)$ **and**
$a15{:}from\text{-}l\ Va\ t < alloc\text{-}l\ Va\ t$ **and**
$a16{:}cur\ Va = Some\ t$ **and**
$a17{:}n < n\text{-}max\ (mem\text{-}pool\text{-}info\ Va\ p) * 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t)$ **and**
$a18{:}blk\ Va\ t = buf\ (mem\text{-}pool\text{-}info\ Va\ p) + n * (max\text{-}sz\ (mem\text{-}pool\text{-}info\ Va\ p)\ div\ 4 \text{ ^ } nat\ (from\text{-}l\ Va\ t))$ **and**
$a19{:}mempoolalloc\text{-}ret\ Va\ t = None$ **and**
$a20{:}\forall\ ii \le nat\ (alloc\text{-}l\ Va\ t).\ sz \le lsizes\ Va\ t\ !\ ii$ **and**
$a21{:}alloc\text{-}l\ Va\ t = int\ (length\ (lsizes\ Va\ t)) - 1 \land length\ (lsizes\ Va\ t) = n\text{-}levels\ (mem\text{-}pool\text{-}info\ Va\ p) \lor$
$alloc\text{-}l\ Va\ t = int\ (length\ (lsizes\ Va\ t)) - 2 \land lsizes\ Va\ t\ !\ nat\ (alloc\text{-}l\ Va\ t + 1) < sz$ **and**
$a22{:}i\ x\ t = 4$ **and**
$a23{:}cur\ x = cur\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)$ **and**
$a24{:}tick\ x = tick\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)$ **and**
$a25{:}thd\text{-}state\ x = thd\text{-}state\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)$ **and**
$a26{:}(x,\ mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p) \in gvars\text{-}conf\text{-}stable$ **and**
$a27{:}\forall\ pa.\ pa \ne p \longrightarrow mem\text{-}pool\text{-}info\ x\ pa = mem\text{-}pool\text{-}info\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ pa$ **and**
$a28{:}wait\text{-}q\ (mem\text{-}pool\text{-}info\ x\ p) = wait\text{-}q\ (mem\text{-}pool\text{-}info\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ p)$ **and**
$a29{:}\forall\ t'.\ t' \ne t \longrightarrow lvars\text{-}nochange\ t'\ x\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)$ **and**
$a30{:}\forall\ jj.\ jj \ne nat\ (from\text{-}l\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t + 1) \longrightarrow$
$levels\ (mem\text{-}pool\text{-}info\ x\ p)\ !\ jj = levels\ (mem\text{-}pool\text{-}info\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ p)\ !\ jj$ **and**
$a31{:}bits\ (levels\ (mem\text{-}pool\text{-}info\ x\ p)\ !\ nat\ (from\text{-}l\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t + 1)) =$
$list\text{-}updates\text{-}n$
$(bits\ (levels\ (mem\text{-}pool\text{-}info\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ p)\ !$
$nat\ (from\text{-}l\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t + 1)))$
$(Suc\ (bn\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t * 4))\ 3\ FREE$ **and**
$a32{:}free\text{-}list\ (levels\ (mem\text{-}pool\text{-}info\ x\ p)\ !\ nat\ (from\text{-}l\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t + 1)) =$
$inserts$
$(map\ (\lambda ii.\ lsizes\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t\ !$
$nat\ (from\text{-}l\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t + 1) *$
$ii +$
$blk\ (mp\text{-}alloc\text{-}stm4\text{-}pre\text{-}precond\text{-}f\ Va\ t\ p)\ t)$
$[Suc\ NULL..{<}4])$

(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) *t* + *1*))) **and**
a33:*j* *x* = *j* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a34:*ret* *x* = *ret* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a35:*endt* *x* = *endt* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a36:*rf* *x* = *rf* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a37:*tmout* *x* = *tmout* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a38:*lsizes* *x* = *lsizes* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a39:*alloc-l* *x* = *alloc-l* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a40:*free-l* *x* = *free-l* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a41:*from-l* *x* = *from-l* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a42:*blk* *x* = *blk* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a43:*nodev* *x* = *nodev* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a44:*bn* *x* = *bn* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a45:*alloc-lsize-r* *x* = *alloc-lsize-r* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a46:*lvl* *x* = *lvl* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a47:*bb* *x* = *bb* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a48:*block-pt* *x* = *block-pt* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a49:*th* *x* = *th* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a50:*need-resched* *x* = *need-resched* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a51:*mempoolalloc-ret* *x* = *mempoolalloc-ret* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*)
and
a52:*freeing-node* *x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*) **and**
a53:*allocating-node* *x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f* *Va* *t* *p*)
shows *inv-bitmap* *x*
proof−
let *?mp* = *mem-pool-info* *x* *p*
have *inv*:*inv-aux-vars* *Va* ∧ *inv-bitmap* *Va* ∧ *inv-mempool-info* *Va* ∧ *inv-bitmap-freelist*
Va
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l* *Va* *t* **using** *a11* *a9* **by** *linarith*
have *len-levels*:*length* (*levels* (*mem-pool-info* *x* *p*)) = *length* (*levels* (*mem-pool-info*
Va *p*))
using *mp-alloc-stm4-lvl-len*[*OF* *a2* *a26*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info* *x* *p*) = *max-sz* (*mem-pool-info* *Va* *p*)
using *mp-alloc-stm4-maxsz*[*OF* *a2* *a26*] **by** *simp*
have *buf*:*buf* (*mem-pool-info* *x* *p*) = *buf* (*mem-pool-info* *Va* *p*)
using *mp-alloc-stm4-buf*[*OF* *a2* *a26*] **by** *simp*
have *from-l*:*from-l* *x* = *from-l* *Va*
using *mp-alloc-stm4-froml*[*OF* *a41*] **by** *auto*
have *mem-pools*:*mem-pools* *x* = *mem-pools* *Va* **using** *mp-alloc-stm4-mempools*[*OF*
a26] **by** *auto*
have *lsizes-x-va*:*lsizes* *x* = *lsizes* *Va* **using** *mp-alloc-stm4-pre-precond-f-lsz* *a38*
by *auto*
have *from-l-gt0*:*OK* ≤ *from-l* *Va* *t* **using** *a11* *a9* **by** *linarith*
{ **fix** *p'*
assume *a00*:*p'*∈*mem-pools* *x*
let *?i1*=(*nat* (*from-l* *Va* *t*)) **and**

?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*4*∗*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i1' = (*nat* (*from-l Va t*)) − *1* **and**
?j1' = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) *div 4* **and**
?i2' = (*nat* (*from-l Va t*)) + *2* **and**
?j2' = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) ∗ *16*

have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a12 a0 a18* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
{**assume** *p*≠*p'*
moreover have *mem-pool-info x p'* = *mem-pool-info Va p'*
using *mp-alloc-stm4-pres-mpinfo*
by (*metis a27 calculation*)
ultimately have *inv-bitmap-mp x p'*
using *a00 inv len-levels maxsz buf from-l mem-pools*
by(*simp add*: *inv-bitmap-def Let-def*)
}
moreover { **assume** *eq-p*:*p*=*p'*
let *?mp* = *mem-pool-info x p*
have *inv-mempool-info-mp Va p*
using *eq-p mem-pools a00 inv* **unfolding** *inv-mempool-info-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
{**fix** *i*
assume *a01*:*i*<*length* (*levels ?mp*)
let *?bts* = *bits* (*levels ?mp* ! *i*)
let *?btsva* = (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))
have *a01'*:*i* < *length* (*levels* (*mem-pool-info Va p*))
using *a01 len-levels* **by** *auto*
then have *inv-bitmap1*:
∀ *j* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *i*)).
(*?btsva* ! *j* = *FREE* ∨ *?btsva* ! *j* = *FREEING* ∨ *?btsva* ! *j* = *ALLOCATED* ∨ *?btsva* ! *j* = *ALLOCATING* ⟶
(*i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*i* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*i*+*1*) (*j*∗*4*)))
∧ (*?btsva* ! *j* = *DIVIDED* ⟶ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*?btsva* ! *j* = *NOEXIST* ⟶ *i* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*i*+*1*) (*j*∗*4*))
∧ (*?btsva* ! *j* = *NOEXIST* ∧ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info*

Va p) ! (*i* − *1*))) ! (*j div 4*) ≠ *DIVIDED*)
using *inv eq-p mem-pools a00*
unfolding *Let-def inv-bitmap-def*
by *blast*
let *?bts* = *bits* (*levels ?mp* ! *i*) **and** *?fl* = *free-list* (*levels ?mp* ! *i*)
have *f1*:∀ *j* < *length ?bts*.
(*?bts* ! *j* = *FREE* ∨ *?bts* ! *j* = *FREEING* ∨ *?bts* ! *j* = *ALLOCATED* ∨ *?bts* ! *j* = *ALLOCATING* ⟶
(*i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info x p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*i* < *length* (*levels* (*mem-pool-info x p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info x p*) (*i*+*1*) (*j*∗*4*)))
∧ (*?bts* ! *j* = *DIVIDED* ⟶ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info x p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*?bts* ! *j* = *NOEXIST* ⟶ *i* < *length* (*levels* (*mem-pool-info x p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info x p*) (*i*+*1*) (*j*∗*4*))
∧ (*?bts* ! *j* = *NOEXIST* ∧ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info x p*) ! (*i* − *1*))) ! (*j div 4*) ≠ *DIVIDED*)
proof−
{ fix *j*
assume *a02*:*j*<*length ?bts*
then have *a02′*:*j* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))
using *a26* **unfolding** *gvars-conf-def gvars-conf-stable-def*
by (*simp add*: *mp-alloc-stm4-inv-bits-len*)
have *eq-len*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *i*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *i*))
using *mp-alloc-stm4-inv-bits-len a30 a31 length-list-update-n*
by *metis*
have *inv-va*:(*?btsva* ! *j* = *FREE* ∨ *?btsva* ! *j* = *FREEING* ∨ *?btsva* ! *j* = *ALLOCATED* ∨ *?btsva* ! *j* = *ALLOCATING* ⟶
(*i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*i* < *length* (*levels* (*mem-pool-info Va p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info Va p*) (*i*+*1*) (*j*∗*4*)))
∧ (*?btsva* ! *j* = *DIVIDED* ⟶ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*?btsva* ! *j* = *NOEXIST* ⟶ *i* < *length* (*levels* (*mem-pool-info Va p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info Va p*) (*i*+*1*) (*j*∗*4*))
∧ (*?btsva* ! *j* = *NOEXIST* ∧ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info Va p*) ! (*i* − *1*))) ! (*j div 4*) ≠ *DIVIDED*)
using *inv-bitmap1 a02′ eq-p* **by** *auto*
let *?goal1* = (*?bts* ! *j* = *FREE* ∨ *?bts* ! *j* = *FREEING* ∨ *?bts* ! *j* = *ALLOCATED* ∨ *?bts* ! *j* = *ALLOCATING* ⟶
(*i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info x p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
∧ (*i* < *length* (*levels* (*mem-pool-info x p*)) − *1* ⟶ *noexist-bits* (*mem-pool-info x p*) (*i*+*1*) (*j*∗*4*)))

let *?goal2* =(*?bts* ! *j* = *DIVIDED* ⟶ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info* *x p*) ! (*i* − *1*))) ! (*j div 4*) = *DIVIDED*)
let *?goal3* =(*?bts* ! *j* = *NOEXIST* ⟶ *i* < *length* (*levels* (*mem-pool-info* *x p*)) − *1*
⟶ *noexist-bits* (*mem-pool-info x p*) (*i*+*1*) (*j*∗*4*))
let *?goal4* = (*?bts* ! *j* = *NOEXIST* ∧ *i* > *0* ⟶ (*bits* (*levels* (*mem-pool-info* *x p*) ! (*i* − *1*))) ! (*j div 4*) ≠ *DIVIDED*)
have *?goal1* **using** *eq-p*
mp-alloc-stm4-inv-bitmap1[*OF a0 a2 a6 a8 a9 a11 a12 a13 a14 a15 a17 a18 a26 a30 a31 a32 a38 a41 a01 a02*]
by *auto*
moreover have *?goal2*
using *mp-alloc-stm4-inv-bitmap2*[*OF a0 a2 a6 a7 a8 a9 a11 a12 a13 a14 a15 a18 a26 a30 a31 a32 a38 a41 a01 a02*]
by *auto*
moreover have *?goal3*
using *mp-alloc-stm4-inv-bitmap3*[*OF a0 a2 a6 a8 a9 a11 a12 a13 a14 a15 a17 a18 a26 a30 a31 a32 a38 a41 a01 a02*]
by *auto*
moreover have *?goal4* **using** *mp-alloc-stm4-inv-bitmap4*[*OF a0 a2 a6 a7 a8 a9 a11 a12 a13 a14 a15 a18 a26 a30 a31 a32 a38 a41 a01 a02*]
by *auto*
ultimately have *?goal1* ∧ *?goal2* ∧ *?goal3* ∧ *?goal4*
by *blast*
} thus *?thesis* **by** *auto*
qed
} then have *inv-bitmap-mp x p′* **using** *eq-p* **by** *auto*
} ultimately have *inv-bitmap-mp x p′* **by** *fastforce*
} then show *?thesis* **unfolding** *inv-bitmap-def* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars1*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info* *Va p*)) *div 4* ˆ *ii* **and**
a4:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a5:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a6:¬ *free-l Va t* < *OK* **and**
a7:*free-l Va t* ≤ *from-l Va t* **and**
a8:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),

data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))⦈ **and**
a9:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a10:*from-l Va t* < *alloc-l Va t* **and**
a11:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)) **and**
a12:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a13:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a14:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a15:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a16:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a17:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a18:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a20:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a21:*freeing-node x t'* = *Some m*
shows *get-bit-s x* (*pool m*) (*level m*) (*block m*) = *FREEING*
proof−
have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a7 a6* **by** *linarith*
have *inv-aux-va*:(∀ *t n*. *freeing-node Va t* = *Some n* ⟶
get-bit (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *FREEING*)
using *a0* **unfolding** *inv-def inv-aux-vars-def*
by *blast*
let *?i1*=(*nat* (*from-l Va t*)) **and**

?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*)
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a12*] **by** *auto*
have *inv-mempool-info-mp Va p*
using *a2 mem-pools inv* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a18*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a2 a5 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a10 a2 a5 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis a0 a2 a9 a11 i1-len inv-mempool-info-def invariant.inv-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a17*
by *auto*
{ assume *t′*= *t*
then have *?thesis*
using *a1 a19 a21*
by (*metis mp-alloc-stm4-pre-precond-f-def-frnode option.distinct*(*1*))
}
moreover {assume *a00*:*t′*≠*t*
then have *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t′* = *freeing-node Va t′*
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *auto*
then have *eq-alloc*:*freeing-node Va t′* = *freeing-node x t′*
using *a19* **by** *auto*
then have *t2-same-allocating-node-Va*:*freeing-node Va t′* = *Some m*
using *a0 a21 a19*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def inv-aux-vars-def*
by *auto*
then have *diff-t*:¬(*pool m* = *p* ∧ *level m* = *?i1* ∧ *block m* = *?j1*)
using *a00 a21 a8 inv* **unfolding** *inv-aux-vars-def*
by (*metis Mem-block.simps*(*1*) *Mem-block.simps*(*2*) *Mem-block.simps*(*3*) *a11*)
{
assume *pool m* ≠ *p*
then have *?thesis* **using** *a0 a13 a21 eq-alloc mp-alloc-stm4-pres-mpinfo*

```
        unfolding inv-aux-vars-def invariant.inv-def
        by metis
    } note not-pool-p-allocating = this
    moreover {
      assume a01:pool m = p
      have bit-m-va-alloc:get-bit (mem-pool-info Va) (pool m) (level m) (block m)
= FREEING
        using a21 eq-alloc inv-aux-va by presburger
      have maxsz:max-sz (mem-pool-info x p) = max-sz (mem-pool-info Va p)
        using mp-alloc-stm4-maxsz[OF a2 a12] by simp
      have buf:buf (mem-pool-info x p) = buf (mem-pool-info Va p)
        using mp-alloc-stm4-buf[OF a2 a12] by simp
      have alloc-i1-j1:get-bit-s Va p ?i1 ?j1 = ALLOCATING
        using a8 a0 a11 unfolding inv-aux-vars-def invariant.inv-def
        by (metis (no-types) Mem-block.select-convs(1) Mem-block.select-convs(2)
Mem-block.select-convs(3))
      have nexisti2:noexist-bits (mem-pool-info Va p) ?i2 ?j2
      using a2 conjunct1[OF conjunct2[OF inv], simplified Let-def inv-bitmap-def]
i1-len j1-len
        alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
      by (smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less
plus-1-eq-Suc)
      { assume a02:(level m = ?i1 ∧ block m = ?j1)
        then have ?thesis using diff-t a01 by auto
      }
      moreover {
        assume a02:¬(level m = ?i1 ∧ block m = ?j1)
        { assume a03:¬(level m = ?i2 ∧ (block m)≥ ?j2 ∧ (block m)<?j2 + 4)
          then have eq-get-bit-i-j:get-bit-s x p (level m) (block m) =
                                get-bit-s Va p (level m) (block m)
            using same-bit-mp-alloc-x-va[OF a14[simplified
              a18[simplified mp-alloc-stm4-froml[OF a18], THEN sym]] a15, of level
m block m]
            a01 a02 by auto
          then have ?thesis using a01 a20 inv-aux-va not-pool-p-allocating
              a21 eq-alloc inv-aux-va by force
        }
        moreover {
          assume a03:(level m = ?i2 ∧ (block m)≥ ?j2 ∧ (block m)<?j2 + 4)
          then have block m = ?j2 ∨ block m = ?j2 + 1 ∨ block m = ?j2 + 2 ∨
block m = ?j2 + 3
            by auto
          then have ?thesis using bit-m-va-alloc nexisti2 a01 a03 by auto
        } ultimately have ?thesis by fastforce
      } ultimately have ?thesis by fastforce
    } ultimately have ?thesis by fastforce
  } ultimately show ?thesis by auto
qed
```

lemma *mp-alloc-stm4-inv-aux-vars2*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t = None* **and**
a2:*p ∈ mem-pools Va* **and**
a3:*alloc-l Va t < int (n-levels (mem-pool-info Va p))* **and**
a4:*¬ free-l Va t < OK* **and**
a5:*free-l Va t ≤ from-l Va t* **and**
a6:*block-num (mem-pool-info Va p)*
(buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ˆ nat (from-l Va t)))
(lsizes Va t ! nat (from-l Va t))
< n-max (mem-pool-info Va p) ∗ 4 ˆ nat (from-l Va t) **and**
a7:*from-l Va t < alloc-l Va t* **and**
a8:*blk Va t = buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ˆ nat (from-l Va t))* **and**
a9:*(x, mp-alloc-stm4-pre-precond-f Va t p) ∈ gvars-conf-stable* **and**
a10:*∀ pa. pa ≠ p ⟶ mem-pool-info x pa = mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) pa* **and**
a11:*∀ jj. jj ≠ nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1) ⟶*
levels (mem-pool-info x p) ! jj = levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) ! jj **and**
a12:*bits (levels (mem-pool-info x p) ! nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)) =*
list-updates-n
(bits (levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) !
nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)))
(Suc (bn (mp-alloc-stm4-pre-precond-f Va t p) t ∗ 4)) 3 FREE **and**
a13:*from-l x = from-l (mp-alloc-stm4-pre-precond-f Va t p)* **and**
a14:*freeing-node x = freeing-node (mp-alloc-stm4-pre-precond-f Va t p)* **and**
a54:*get-bit-s x (pool m) (level m) (block m) = FREEING ∧ mem-block-addr-valid x m*
shows *(∃ t. freeing-node x t = Some m)*
proof−
have *inv*:*inv-aux-vars Va ∧ inv-bitmap Va ∧ inv-mempool-info Va ∧ inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0 ≤ from-l Va t* **using** *a5 a4* **by** *linarith*
have *block-valid-va*:*mem-block-addr-valid Va m*
using *a2 a9 a54 mp-alloc-stm4-buf mp-alloc-stm4-maxsz*
unfolding *mem-block-addr-valid-def* **by** *auto*
have *inv-aux-va*:*(∀ n. get-bit (mem-pool-info Va) (pool n) (level n) (block n) = FREEING ∧ mem-block-addr-valid Va n*
⟶ (∃ t. freeing-node Va t = Some n))
using *a0* **unfolding** *inv-def inv-aux-vars-def*
by *blast*
{**assume** *(pool m) ≠ p*
then have *get-bit-s Va (pool m) (level m) (block m) = get-bit-s x (pool m)*

(*level m*) (*block m*)
  using *a10*
  by (*metis mp-alloc-stm4-pres-mpinfo*)
 then have *?thesis* **using** *a54 inv-aux-va block-valid-va a14 mp-alloc-stm4-pre-precond-f-def-frnode*
  by *metis*
 }
 moreover{
 assume *a01*:*pool m* = *p*
 let *?i1*=(*nat* (*from-l Va t*)) **and**
 ?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
 ?i2 = (*nat* (*from-l Va t* + *1*)) **and**
 ?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*)
 have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a9*] **by** *auto*
 have *inv-mempool-info-mp Va p*
  using *a2 mem-pools inv* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
 note *inv-mempool*=*this*[*simplified Let-def*]
 have *from-l*:*from-l x* = *from-l Va*
  using *mp-alloc-stm4-froml*[*OF a13*] **by** *auto*
 have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
  using *from-l-gt0* **by** *auto*
 have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
  using *a7 a2 a3 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
 have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
  using *a7 a2 a3 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
  by *auto*
 have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
  by (*metis a0 a2 a6 a8 i1-len inv-mempool-info-def invariant.inv-def*)
 have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
  using *i1-len i2-len j1-len inv-mempool from-l-suc*
  by *simp*
 { assume *a02*:¬(((*level m*)=*?i1* ∧ (*block m*)=*?j1*) ∨
   ((*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)< *?j2*+*4*))
  then have *a020′*:¬((*level m*)=*?i1* ∧ (*block m*)=*?j1*) **and**
   a021′: ¬((*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)<*?j2* + *4*)
   by *force+*
  then have *eq-get-bit-i-j*:*get-bit-s x p* (*level m*) (*block m*) = *get-bit-s Va p* (*level m*) (*block m*)
   using *same-bit-mp-alloc-x-va*[*OF a11*[*simplified a13*[*simplified mp-alloc-stm4-froml*[*OF a13*], *THEN sym*]] *a12*, *of level m block m*]
   using *a020′ a021′* **by** *auto*
  then have *?thesis* **using** *a01 a54 inv-aux-va block-valid-va a14 mp-alloc-stm4-pre-precond-f-def-frnode*
   by *metis*

}
moreover{
assume *a02*:((*level m*)=*?i1* ∧ (*block m*)=*?j1*)
then have *get-bit-s x p ?i1 ?j1* = *DIVIDED*
by (*simp add*: *a11 from-l-gt0 from-l-suc i1-len j1-len mp-alloc-stm4-pre-precond-f-froml same-bit-mp-alloc-stm4-pre-precond-divided*)
then have *?thesis* **using** *a54 a02 a01* **by** *auto*
}
moreover{
assume *a02*:(*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)< *?j2+4*
then have *get-bit-s x p ?i2 ?j2* = *ALLOCATING*
using *i2-len j2-len a12 get-bit-x-l1-b4*[*OF - from-l-gt0 a12*, *of ?i2 ?j2*]
by (*metis* (*no-types*, *lifting*) *add-2-eq-Suc′ add-Suc-right add-lessD1 mult.commute*)
moreover {
assume *a07*:(*block m*)≠*?j2*
then have (*level m*) = *?i2* ∧ ((*block m*) = *Suc ?j2* ∨ (*block m*) = *Suc* (*Suc ?j2*) ∨ (*block m*) = *Suc* (*Suc* (*Suc?j2*)))
using *a02* **by** *auto*
then have *get-bit-s x p* (*level m*) (*block m*) = *FREE*
using *a02 i2-len j2-len get-bit-x-l1-b41*[*OF - from-l-gt0*[*simplified from-l a13*] *a11*[*simplified a13*[*THEN sym*] *from-l*] *a12*,*of level m block m*]
by (*metis Suc-lessD mult.commute*)
}
ultimately have *?thesis* **using** *a54 a02 a01* **by** *fastforce*
} **ultimately have** *?thesis* **by** *auto*
} **ultimately show** *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars3*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a4:¬ *free-l Va t* < *OK* **and**
a5:*free-l Va t* ≤ *from-l Va t* **and**
a6:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))|) **and**
a7:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))

(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) * *4* ^ *nat* (*from-l Va t*) **and**
a8:*from-l Va t* < *alloc-l Va t* **and**
a9:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* * (*max-sz* (*mem-pool-info Va p*) *div*
4 ^ *nat* (*from-l Va t*)) **and**
a10:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a11:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *pa* **and**
a12:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*) ! *jj* **and**
a13:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va*
t p) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* * *4*)) *3 FREE* **and**
a14:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f*
Va t p) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) *
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a15:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a16:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a17:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a18:*allocating-node x t'* = *Some m*
shows *get-bit-s x* (*pool m*) (*level m*) (*block m*) = *ALLOCATING*
proof−
have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist*
Va
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a5 a4* **by** *linarith*
have *inv-aux-va*:(∀ *t n*. *allocating-node Va t* = *Some n* ⟶
get-bit (*mem-pool-info Va*) (*pool n*) (*level n*) (*block n*) = *ALLOCATING*)
using *a0* **unfolding** *inv-def inv-aux-vars-def*
by *blast*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va*
t))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va*
t))*4)
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF*

a10] **by** *auto*
have *inv-mempool-info-mp Va p*
using *a2 mem-pools inv* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
note *inv-mempool=this*[*simplified Let-def*]
have *from-l*:*from-l x = from-l Va*
using *mp-alloc-stm4-froml*[*OF a16*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t + 1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *i1-len*:*?i1 < length* (*levels* (*mem-pool-info Va p*))
using *a8 a2 a3 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2 < length* (*levels* (*mem-pool-info Va p*))
using *a8 a2 a3 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1 < length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis a0 a2 a7 a9 i1-len inv-mempool-info-def invariant.inv-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
have *lsizes-x-va*:*lsizes x = lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a15*
by *auto*
{**assume** *a00*:$t' \neq t$
then have *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) t' = *allocating-node Va* t'
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *auto*
then have *eq-alloc*:*allocating-node Va* t' = *allocating-node x* t'
using *a17* **by** *auto*
then have *diff-t*:¬(*pool m = p* ∧ *level m = ?i1* ∧ *block m = ?j1*)
using *a00 a18 a6 inv* **unfolding** *inv-aux-vars-def*
by (*metis Mem-block.simps*(*1*) *Mem-block.simps*(*2*) *Mem-block.simps*(*3*) *a9*)
{
assume *pool m* ≠ *p*
then have *?thesis*
by (*metis a11 a18 eq-alloc inv-aux-va mp-alloc-stm4-pres-mpinfo*)
} **note** *not-pool-p-allocating = this*
moreover {
assume *a01*:*pool m = p*
have *bit-m-va-alloc*:*get-bit* (*mem-pool-info Va*) (*pool m*) (*level m*) (*block m*) = *ALLOCATING*
using *a18 eq-alloc inv-aux-va* **by** *presburger*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a2 a10*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a2 a10*] **by** *simp*
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1 = ALLOCATING*
using *a6 a0 a9* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))

have *nexisti2*:*noexist-bits* (*mem-pool-info Va p*) *?i2 ?j2*
using *a2 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*] *i1-len j1-len*
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less plus-1-eq-Suc*)
{ assume *a02*:(*level m* = *?i1* ∧ *block m* = *?j1*)
then have *?thesis* **using** *diff-t a01* **by** *auto*
}
moreover {
assume *a02*:¬(*level m* = *?i1* ∧ *block m* = *?j1*)
{ assume *a03*:¬(*level m* = *?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)<*?j2* + *4*)
then have *eq-get-bit-i-j*:*get-bit-s x p* (*level m*) (*block m*) =
get-bit-s Va p (*level m*) (*block m*)
using *same-bit-mp-alloc-x-va*[*OF a12*[*simplified*
a16[*simplified mp-alloc-stm4-froml*[*OF a16*], *THEN sym*]] *a13*, *of level m block m*]
a01 a02 **by** *auto*
then have *?thesis* **using** *a01 a17 inv-aux-va not-pool-p-allocating*
a18 eq-alloc inv-aux-va **by** *force*
}
moreover {
assume *a03*:(*level m* = *?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)<*?j2* + *4*)
then have *block m* = *?j2* ∨ *block m* = *?j2* + *1* ∨ *block m* = *?j2* + *2* ∨ *block m* = *?j2* + *3*
by *auto*
then have *?thesis* **using** *bit-m-va-alloc nexisti2 a01 a03* **by** *auto*
}
ultimately have *?thesis* **by** *fastforce*
} ultimately have *?thesis* **by** *fastforce*
} ultimately have *?thesis* **by** *auto*
}
moreover {
assume *t′*=*t*
then have (*pool m*) = *p* ∧ (*level m*) = *?i2* ∧ (*block m*) = *?j2*
by (*metis Mem-block.simps*(*1*) *Mem-block.simps*(*2*) *Mem-block.simps*(*3*) *a17 a18*
mp-alloc-stm4-pre-precond-f-allocating mult.commute option.sel)
then have *?thesis* **using** *get-bit-x-l1-b4*[*OF - from-l-gt0 a13 i2-len*] *j2-len*
by (*metis Suc-lessD mult.commute*)
}
ultimately show *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars4*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**

a3:$\forall$ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:$\neg$ *free-l Va t* < *OK* **and**
a6:*free-l Va t* $\leq$ *from-l Va t* **and**
a7:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*))|) **and**
a8:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) $*$ *4* ^ *nat* (*from-l Va t*) **and**
a9:*from-l Va t* < *alloc-l Va t* **and**
a10:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* $*$ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) **and**
a11:*alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* $\wedge$ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) $\vee$
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* $\wedge$ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
a12:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* **and**
a13:$\forall$ *pa*. *pa* $\neq$ *p* $\longrightarrow$ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a14:$\forall$ *jj*. *jj* $\neq$ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) $\longrightarrow$
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a15:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* $*$ *4*)) *3 FREE* **and**
a16:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) $*$
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a17:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**

$a18$:*from-l x = from-l (mp-alloc-stm4-pre-precond-f Va t p)* **and**
$a19$:*blk x = blk (mp-alloc-stm4-pre-precond-f Va t p)* **and**
$a20$:*allocating-node x = allocating-node (mp-alloc-stm4-pre-precond-f Va t p)* **and**
$a21$:*get-bit-s x (pool m) (level m) (block m) = ALLOCATING* $\wedge$ *mem-block-addr-valid x m*
shows ($\exists$ *t. allocating-node x t = Some m*)
proof−
have *inv*:*inv-aux-vars Va* $\wedge$ *inv-bitmap Va* $\wedge$ *inv-mempool-info Va* $\wedge$ *inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* $\leq$ *from-l Va t* **using** *a6 a5* **by** *linarith*
have *block-valid-va*:*mem-block-addr-valid Va m*
using *a2 a12 a21 mp-alloc-stm4-buf mp-alloc-stm4-maxsz*
unfolding *mem-block-addr-valid-def* **by** *auto*
have *data-m*:*data m =*
buf (mem-pool-info x (pool m)) + (block m) ∗ ((max-sz (mem-pool-info x (pool m))) div (4 ^ (level m)))
using *a21* **unfolding** *mem-block-addr-valid-def* **by** *auto*
have *inv-aux-va*:($\forall$ *n. get-bit (mem-pool-info Va) (pool n) (level n) (block n) = ALLOCATING* $\wedge$ *mem-block-addr-valid Va n*
$\longrightarrow$ ($\exists$ *t. allocating-node Va t = Some n*))
using *a0* **unfolding** *inv-def inv-aux-vars-def*
by *blast*
{ assume *a00*:(*pool m*) $\neq$ *p*
then obtain t' **where** *allocating-node Va* t' *= Some m* **using** *inv-aux-va*
by (*metis a13 a21 block-valid-va mp-alloc-stm4-pres-mpinfo*)
moreover have $t' \neq t$ **using** *a2* **unfolding** *inv-def inv-aux-vars-def*
using *a00 a7 calculation* **by** *auto*
then have *allocating-node (mp-alloc-stm4-pre-precond-f Va t p)* t' *= allocating-node Va* t'
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *auto*
then have *eq-alloc*:*allocating-node Va* t' *= allocating-node x* t'
using *a20* **by** *auto*
ultimately have *?thesis* **by** *auto*
}
moreover{
assume *a01*:*pool m = p*
let *?i1*=(*nat (from-l Va t)*) **and**
?j1= (*block-num (mem-pool-info Va p) (blk Va t) (lsizes Va t ! nat (from-l Va t)))* **and**
?i2 = (nat (from-l Va t + 1)) **and**
?j2 = (block-num (mem-pool-info Va p) (blk Va t) (lsizes Va t ! nat (from-l Va t))∗4)
have *mem-pools*:*mem-pools x = mem-pools Va* **using** *mp-alloc-stm4-mempools[OF a12]* **by** *auto*
have *inv-mempool-info-mp Va p*
using *a2 mem-pools inv* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
note *inv-mempool=this[simplified Let-def]*
have *from-l*:*from-l x = from-l Va*

using *mp-alloc-stm4-froml*[*OF a18*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
using *a9 a2 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a9 a2 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis a0 a2 a8 a10 i1-len inv-mempool-info-def invariant.inv-def*)
have *j2-len*:*Suc* (*Suc* (*Suc* *?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
have *lsizes-x-va*:*lsizes x* = *lsizes Va*
by (*simp add*: *a17 mp-alloc-stm4-pre-precond-f-lsz*)
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a2 a12*] **by** *simp*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a2 a12*] **by** *simp*
{ assume *a02*:¬(((*level m*)=*?i1* ∧ (*block m*)=*?j1*) ∨
((*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)< *?j2*+*4*))
then have *a020′*:¬((*level m*)=*?i1* ∧ (*block m*)=*?j1*) **and**
a021′: ¬((*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)<*?j2* + *4*)
by *force+*
then have *eq-get-bit-i-j*:*get-bit-s x p* (*level m*) (*block m*) = *get-bit-s Va p* (*level m*) (*block m*)
using *same-bit-mp-alloc-x-va*[*OF a14*[*simplified*
a18[*simplified mp-alloc-stm4-froml*[*OF a18*], *THEN sym*]] *a15*, *of level m block m*]
using *a020′ a021′* **by** *auto*
then have *get-bit-s Va* (*pool m*) (*level m*) (*block m*) = *ALLOCATING* ∧ *mem-block-addr-valid Va m*
using *a01 a21 block-valid-va* **by** *auto*
then obtain *t′* **where** *allocating-node Va t′* = *Some m* **using** *inv-aux-va* **by** *auto*
moreover have *t′*≠*t* **using** *a02 a7 a10 calculation* **by** *auto*
then have *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t′* = *allocating-node Va t′*
unfolding *mp-alloc-stm4-pre-precond-f-def* **by** *auto*
then have *eq-alloc*:*allocating-node Va t′* = *allocating-node x t′*
using *a20* **by** *auto*
ultimately have *?thesis* **by** *auto*
}
moreover{
assume *a02*:((*level m*)=*?i1* ∧ (*block m*)=*?j1*)
then have *get-bit-s x p ?i1 ?j1* = *DIVIDED*
by (*simp add*: *a14 from-l-gt0 from-l-suc i1-len j1-len mp-alloc-stm4-pre-precond-f-froml*

same-bit-mp-alloc-stm4-pre-precond-divided)
then have *?thesis* **using** *a21 a02 a01* **by** *auto*
}
moreover{
assume *a02*:(*level m*)=*?i2* ∧ (*block m*)≥ *?j2* ∧ (*block m*)< *?j2*+*4*
then have *block-n*:(*block-num* (*mem-pool-info Va p*)
(*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) = *n*
proof−
have *lsizes Va t* ! *nat* (*from-l Va t*) =
ALIGN4 (*max-sz* (*mem-pool-info Va p*)) *div 4* ^
(*nat* (*from-l Va t*))
using *a3 lsizes-x-va a5 a6 a9 a11 a5 from-l* **by** *auto*
thus *?thesis* **using** *block-n inv a2 a10 a4 a9 from-l-gt0*
by *blast*
qed
then have *get-bit-s x p ?i2 ?j2* = *ALLOCATING*
using *i2-len j2-len a15 get-bit-x-l1-b4*[*OF - from-l-gt0 a15*, *of ?i2 ?j2*]
by (*metis* (*no-types*, *lifting*) *add-2-eq-Suc′ add-Suc-right add-lessD1 mult.commute*)
{ assume *a03*:*block m* = *?j2*
then have *m* = (|*pool* = *p*, *level* = *?i2*, *block* = *?j2*,
data = *buf* (*mem-pool-info x p*) +
?j2 ∗ (*max-sz* (*mem-pool-info x p*) *div 4* ^ *?i2*)
|) **using** *data-m a03 a02 a01* **by** *auto*
moreover have *blk x t* = *buf* (*mem-pool-info x p*) +
?j1 ∗ ((*max-sz* (*mem-pool-info x p*) *div 4* ^ *?i1*))
using *a10*[*simplified buf*[*THEN sym*] *maxsz*[*THEN sym*]] *block-n a19*
mp-alloc-stm4-blk
by *metis*
then have *allocating-node x t* = *Some* (|*pool* = *p*, *level* = *?i2*, *block* = *?j2*,
data = *buf* (*mem-pool-info x p*) +
?j1 ∗ (*max-sz* (*mem-pool-info x p*) *div 4* ^
?i1)|)
using *a20 a19 mp-alloc-stm4-blk mp-alloc-stm4-pre-precond-f-allocating*
by (*metis mult.commute*)
ultimately have *?thesis* **using** *buf maxsz next-level-addr-eq* **unfolding**
addr-def
by (*metis from-l-suc i2-len inv-mempool*)
}
moreover {
assume *a07*:(*block m*)≠*?j2*
then have (*level m*) = *?i2* ∧ ((*block m*) = *Suc ?j2* ∨
(*block m*) = *Suc* (*Suc ?j2*) ∨ (*block m*) = *Suc* (*Suc* (*Suc?j2*)))
using *a02* **by** *auto*
then have *get-bit-s x p* (*level m*) (*block m*) = *FREE*
using *a02 i2-len j2-len*
get-bit-x-l1-b41[*OF - from-l-gt0*[*simplified from-l a18*]
a14[*simplified a18*[*THEN sym*] *from-l*] *a15*,*of level m*
block m]
by (*metis Suc-lessD mult.commute*)

then have *?thesis* **using** *a21 a01* **by** *auto*
}
ultimately have *?thesis* **by** *auto*
}
ultimately have *?thesis* **by** *auto*
} **ultimately show** *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars5*:
assumes
a0:*inv Va* **and**
a1:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a2: $t1 \neq t2 \wedge$ *freeing-node x t1* = *Some n1* $\wedge$ *freeing-node x t2* = *Some n2*
shows $\neg$ (*pool n1* = *pool n2* $\wedge$ *level n1* = *level n2* $\wedge$ *block n1* = *block n2*)
proof−
have $t1 \neq t2 \wedge$ *freeing-node Va t1* = *Some n1* $\wedge$ *freeing-node Va t2* = *Some n2*
using *a1 a2*
by (*metis mp-alloc-stm4-pre-precond-f-def-frnode*)
then have $\neg$ (*pool n1* = *pool n2* $\wedge$ *level n1* = *level n2* $\wedge$ *block n1* = *block n2*)
using *a0* **unfolding** *inv-def inv-aux-vars-def* **by** *auto*
then show *?thesis*
by *blast*
qed

lemma *mp-alloc-stm4-inv-aux-vars6*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:$p \in$ *mem-pools Va* **and**
a3:*length* (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) **and**
a4:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a5:$\neg$ *free-l Va t* < *OK* **and**
a6:*free-l Va t* $\leq$ *from-l Va t* **and**
a7:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))|) **and**
a8:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a9:*from-l Va t* < *alloc-l Va t* **and**
a10:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*)) **and**

a11:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a12:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a13:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a14:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a15:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a16:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a17:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a18:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*t1* ≠ *t2* ∧ *allocating-node x t1* = *Some n1* ∧ *allocating-node x t2* = *Some n2*
shows ¬ (*pool n1* = *pool n2* ∧ *level n1* = *level n2* ∧ *block n1* = *block n2*)
proof−
have *inv*:*inv-aux-vars Va* ∧ *inv-bitmap Va* ∧ *inv-mempool-info Va* ∧ *inv-bitmap-freelist Va*
using *a0* **unfolding** *inv-def* **by** *auto*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a6 a5* **by** *linarith*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*)
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a11*] **by** *auto*
have *inv-mempool-info-mp Va p*
using *a2 mem-pools inv* **unfolding** *inv-mempool-info-def Let-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a17*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))

using *a9 a2 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a9 a2 a4 from-l-gt0 inv* **unfolding** *inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis a0 a2 a8 a10 i1-len inv-mempool-info-def invariant.inv-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) !
?i2))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
have *lsizes-x-va*:*lsizes x* = *lsizes Va* **using** *mp-alloc-stm4-pre-precond-f-lsz a16*
by *auto*
have *maxsz*:*max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*)
using *mp-alloc-stm4-maxsz*[*OF a2 a11*] **by** *simp*
have *buf*:*buf* (*mem-pool-info x p*) = *buf* (*mem-pool-info Va p*)
using *mp-alloc-stm4-buf*[*OF a2 a11*] **by** *simp*
have *alloc-i1-j1*:*get-bit-s Va p ?i1 ?j1* = *ALLOCATING*
using *a7 a0 a10* **unfolding** *inv-aux-vars-def invariant.inv-def*
by (*metis* (*no-types*) *Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
have *nexisti2*:*noexist-bits* (*mem-pool-info Va p*) *?i2 ?j2*
using *a2 conjunct1*[*OF conjunct2*[*OF inv*], *simplified Let-def inv-bitmap-def*]
i1-len j1-len
alloc-i1-j1 from-l-suc i2-len i1-len j1-len a1
by (*smt One-nat-def Suc-pred add.commute inv-mempool nat-add-left-cancel-less*
plus-1-eq-Suc)
{ assume *t*≠*t1* **and** *t*≠*t2*
then have *?thesis*
using *a0 a19 a18 inv-aux-vars-def*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def* **by** *force*
}
moreover {
assume *a00*:*t*=*t1*
then have *t2*≠*t* **using** *a19* **by** *auto*
then have *t2-same-allocating-node-Va*:*allocating-node Va t2* = *Some n2*
using *a0 a19 a18*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def inv-aux-vars-def*
by *force*
then have *get-bit-n2*:*get-bit-s Va* (*pool n2*) (*level n2*) (*block n2*) = *ALLOCATING*
using *a0 a19 a18 inv-aux-vars-def*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def* **by** *force*
have ¬ (*pool n1* = *pool n2* ∧ *level n1* = *level n2* ∧ *block n1* = *block n2*) =
(*pool n1* = *pool n2* ⟶ ¬(*level n1* = *level n2* ∧ *block n1* = *block n2*))
by *auto*
moreover {
assume *a02*:*pool n1* = *pool n2*
have *n1* = (|*pool* = *p*, *level* = *?i2*, *block* = *?j2*,
data = *blk Va t*
|) **using** *a19*

by (*simp add*: *a00 a18 mp-alloc-stm4-pre-precond-f-allocating mult.commute*)
then have ¬(*level n1* = *level n2* ∧ *block n1* = *block n2*)
using *get-bit-n2 a02 nexisti2* **by** *auto*
}
then have *?thesis* **by** *auto*
}
moreover {
assume *a00*:*t*=*t2*
then have *t1*≠*t* **using** *a19* **by** *auto*
then have *t2-same-allocating-node-Va*:*allocating-node Va t1* = *Some n1*
using *a0 a19 a18 inv-aux-vars-def*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def* **by** *force*
then have *get-bit-n2*:*get-bit-s Va* (*pool n1*) (*level n1*) (*block n1*) = *ALLOCATING*
using *a0 a19 a18 inv-aux-vars-def*
unfolding *mp-alloc-stm4-pre-precond-f-def invariant.inv-def* **by** *force*
have ¬ (*pool n1* = *pool n2* ∧ *level n1* = *level n2* ∧ *block n1* = *block n2*) =
(*pool n1* = *pool n2* ⟶ ¬(*level n1* = *level n2* ∧ *block n1* = *block n2*))
by *auto*
moreover {
assume *a02*:*pool n1* = *pool n2*
have *n2* = (|*pool* = *p*, *level* = *?i2*, *block* = *?j2*,
data = *blk Va t*
|) **using** *a19*
by (*simp add*: *a00 a18 mp-alloc-stm4-pre-precond-f-allocating mult.commute*)
then have ¬(*level n1* = *level n2* ∧ *block n1* = *block n2*)
using *get-bit-n2 a02 nexisti2* **by** *auto*
}
then have *?thesis* **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars7*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a4:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a5:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a6:¬ *free-l Va t* < *OK* **and**
a7:*free-l Va t* ≤ *from-l Va t* **and**
a8:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),

data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div* *4* ˆ *nat* (*from-l Va t*))⟧ **and**
a9:*n* = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) = *NULL* **and**
a10:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a11:*from-l Va t* < *alloc-l Va t* **and**
a12:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a23:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a14:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a15:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a16:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a17:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a18:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a20:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a21:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a22:*allocating-node x t1* = *Some n1* ∧ *freeing-node x t2* = *Some n2*
shows ¬ (*pool n1* = *pool n2* ∧ *level n1* = *level n2* ∧ *block n1* = *block n2*)
proof−
{**assume** *pool n1* = *pool n2*
moreover have *get-bit-s x* (*pool n1*) (*level n1*) (*block n1*) = *ALLOCATING*
using *mp-alloc-stm4-inv-aux-vars3 assms* **by** *blast*
moreover have *get-bit-s x* (*pool n2*) (*level n2*) (*block n2*) = *FREEING*

using *mp-alloc-stm4-inv-aux-vars1 assms* **by** *blast*
ultimately have *?thesis* **by** *auto*
} **thus** *?thesis* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-aux-vars*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t = None* **and**
a2:$p \in$ *mem-pools Va* **and**
a3:$\forall$ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a4:*length* (*lsizes Va t*) $\leq$ *n-levels* (*mem-pool-info Va p*) **and**
a5:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a6:$\neg$ *free-l Va t* < *OK* **and**
a7:*free-l Va t* $\leq$ *from-l Va t* **and**
a8:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*))|) **and**
a9:*n* = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) $\vee$
max-sz (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) = *NULL* **and**
a10:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a11:*from-l Va t* < *alloc-l Va t* **and**
a12:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a13:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) $\in$ *gvars-conf-stable* **and**
a14:$\forall$ *pa*. *pa* $\neq$ *p* $\longrightarrow$ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a15:$\forall$ *jj*. *jj* $\neq$ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) $\longrightarrow$
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a16:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**

a17:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (*λii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a18:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a20:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a21:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a22:*alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
a23:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows *inv-aux-vars x* **unfolding** *inv-aux-vars-def*
using
mp-alloc-stm4-inv-aux-vars1[*OF assms*(*1*−*9*,*11*−*22*)]
mp-alloc-stm4-inv-aux-vars2[*OF assms*(*1*−*3*,*6*−*8*,*11*−*17*,*20*−*21*)]
mp-alloc-stm4-inv-aux-vars3[*OF assms*(*1*−*3*,*6*−*7*,*8*−*9*,*11*−*20*,*22*)]
mp-alloc-stm4-inv-aux-vars4[*OF assms*(*1*−*4*,*6*−*9*,*11*−*13*,*23*,*14*−*20*,*24*,*22*)]
mp-alloc-stm4-inv-aux-vars5[*OF assms*(*1*,*21*)]
mp-alloc-stm4-inv-aux-vars6[*OF assms*(*1*−*3*,*5*−*9*,*11*−*20*,*22*)] *mp-alloc-stm4-inv-aux-vars7*[*OF assms*(*1*−*22*)]
by *auto*

lemma *mp-alloc-stm4-inv-bitmap0*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a4:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a5:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a6:¬ *free-l Va t* < *OK* **and**
a7:*free-l Va t* ≤ *from-l Va t* **and**
a8:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*

4 ^ *nat* (*from-l Va t*))⦈ **and**
a9: n = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + n ∗ (*max-sz* (*mem-pool-info Va p*) *div* 4 ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div* 4 ^ *nat* (*from-l Va t*) = *NULL* **and**
a10:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + n ∗ (*max-sz* (*mem-pool-info Va p*) *div* 4 ^ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ 4 ^ *nat* (*from-l Va t*) **and**
a11:*from-l Va t* < *alloc-l Va t* **and**
a12:*blk Va t* = *buf* (*mem-pool-info Va p*) + n ∗ (*max-sz* (*mem-pool-info Va p*) *div* 4 ^ *nat* (*from-l Va t*)) **and**
a13:(x, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a14:∀ *pa*. *pa* ≠ p ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a15:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a16:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a17:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a18:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a20:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a21:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a22:*alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
a23:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows *inv-bitmap0 x*
proof(*simp add*: *inv-bitmap0-def Let-def*)
{ **fix** p' j

assume *a00*:*p′* ∈ *mem-pools x*
assume *a01*:*j*<*length* (*bits* (*levels* (*mem-pool-info x p′*) ! *NULL*))
{ assume *p′*≠ *p*
then have *get-bit-s x p′ NULL j* ≠ *NOEXIST*
by (*metis a0 a00 a01 a13 a14 inv-bitmap0-def invariant.inv-def mp-alloc-stm4-mempools mp-alloc-stm4-pres-mpinfo*)
}
moreover { assume *a02*:*p′* = *p*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*)
have *from-l-gt0*:(*nat* (*from-l Va t* + *1*)) > *0*
using *a6 a7* **by** *linarith*
have *zero-lt-len-levels*:*0* < *length* (*levels* (*mem-pool-info x p*))
by (*metis a0 a13 a2 inv-mempool-info-def invariant.inv-def mp-alloc-stm4-lvl-len*)
then have *len-eq*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *0*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *0*))
using *a13 mp-alloc-stm4-inv-bits-len*
unfolding *gvars-conf-stable-def gvars-conf-def*
by *fastforce*
have *from-l-gt0*:*0* ≤ *from-l Va t* **using** *a7 a6* **by** *linarith*
{ assume *a04*:*j* = *?j1*
then have *get-bit-s x p′ NULL j* ≠ *NOEXIST* **using** *a00 a01 a02 get-bit-x-l-b a13 mp-alloc-stm4-lvl-len*[*OF a2 a13*] *len-eq mp-alloc-stm4-froml*[*OF a19*] *from-l-gt0 a19 a0 a15 a2 same-bit-mp-alloc-stm4-pre-precond-divided*
unfolding *inv-bitmap0-def inv-def* **apply** *auto*
using *mp-alloc-stm4-pre-precond-f-same-bits zero-lt-len-levels*
by (*smt BlockState.distinct*(*19*) *nat-0-iff*)
}
moreover {
assume *a04*:*j*≠*?j1*
then have *eq-get-bit-i-j*:*get-bit-s x p 0 j* = *get-bit-s Va p 0 j*
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified a19*[*simplified mp-alloc-stm4-froml*[*OF a19*], *THEN sym*]] *a16*, *of 0 j*]
using *from-l-gt0* **by** *auto*
then have *get-bit-s x p′ NULL j* ≠ *NOEXIST*
using *a0* **unfolding** *inv-def inv-bitmap0-def a00 a01*
by (*metis a01 a02 a2 len-eq*)
} ultimately have *get-bit-s x p′ NULL j* ≠ *NOEXIST* **by** *auto*
} ultimately have *get-bit-s x p′ NULL j* ≠ *NOEXIST* **by** *auto*
} then show ∀ *p*∈*mem-pools x*.
∀ *i*<*length* (*bits* (*levels* (*mem-pool-info x p*) ! *NULL*)).
get-bit-s x p NULL i ≠ *NOEXIST* **by** *auto*
qed

lemma *mp-alloc-stm4-inv-bitmapn*:
assumes
a0:*inv Va* **and**
a1:*p* ∈ *mem-pools Va* **and**
a2:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a3:¬ *free-l Va t* < *OK* **and**
a4:*free-l Va t* ≤ *from-l Va t* **and**
a5:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a6:*from-l Va t* < *alloc-l Va t* **and**
a7:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a8:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a9:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a10:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a11:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a12:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a13:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows *inv-bitmapn x*
proof(*simp add*: *inv-bitmapn-def Let-def*)
{ **fix** *p′ j*
let *?k* = (*length* (*levels* (*mem-pool-info x p′*)) − *Suc 0*)
assume *a00*:*p′* ∈ *mem-pools x*
assume *a01*:*j*<*length* (*bits* (*levels* (*mem-pool-info x p′*) ! *?k*))
{ **assume** *p′*≠ *p*
then have *get-bit-s x p′ ?k j* ≠ *DIVIDED*
using *a00 a01 a0 a8 a9 mp-alloc-stm4-mempools mp-alloc-stm4-pres-mpinfo*
unfolding *inv-bitmapn-def inv-def*

by (*metis One-nat-def*)
}
moreover { **assume** *a02*:$p' = p$
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))**4*)
have *from-l-gt0*:(*nat* (*from-l Va t* + *1*)) > *0*
using *a3 a4* **by** *linarith*
have *zero-lt-len-levels*:*0* < *length* (*levels* (*mem-pool-info x p*))
by (*metis a0 a8 a1 inv-mempool-info-def invariant.inv-def mp-alloc-stm4-lvl-len*)
then have *len-eq*:*length* (*bits* (*levels* (*mem-pool-info x p*) ! *0*)) =
length (*bits* (*levels* (*mem-pool-info Va p*) ! *0*))
using *a8 mp-alloc-stm4-inv-bits-len*
unfolding *gvars-conf-stable-def gvars-conf-def*
by *fastforce*
have *mem-pools*:*mem-pools x* = *mem-pools Va* **using** *mp-alloc-stm4-mempools*[*OF a8*] **by** *auto*
have *inv-mempool-info-mp Va p*
using *a1 mem-pools a0* **unfolding** *inv-def inv-mempool-info-def Let-def* **by** *auto*
note *inv-mempool*=*this*[*simplified Let-def*]
have *from-l*:*from-l x* = *from-l Va*
using *mp-alloc-stm4-froml*[*OF a13*] **by** *auto*
have *from-l-suc*:*nat* (*from-l Va t* + *1*) = *nat*(*from-l Va t*) + *1*
using *from-l-gt0* **by** *auto*
have *i1-len*:*?i1* < *length* (*levels* (*mem-pool-info Va p*))
using *a6 a1 a2 from-l-gt0 a0* **unfolding** *inv-def inv-mempool-info-def Let-def*
by *auto*
have *i2-len*:*?i2* < *length* (*levels* (*mem-pool-info Va p*))
using *a0 a6 a1 a2 from-l-gt0* **unfolding** *inv-def inv-mempool-info-def Let-def*
by *auto*
have *j1-len*:*?j1* < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i1*))
by (*metis a0 a1 a5 a7 i1-len inv-mempool-info-def invariant.inv-def*)
have *j2-len*:*Suc* (*Suc* (*Suc ?j2*)) < *length* (*bits* (*levels* (*mem-pool-info Va p*) ! *?i2*))
using *i1-len i2-len j1-len inv-mempool from-l-suc*
by *simp*
have *from-l-gt0*:$0 \leq$ *from-l Va t* **using** *a4 a3* **by** *linarith*
{ **assume** *a03*:*?i2* = *?k*
{ **assume** *a04*:$j \geq$ *?j2* $\wedge$ $j <$ *?j2*+*4*
{ **assume** *a05*:*j*=*?j2*
then have *get-bit-s x* p' *?i2 j* = *ALLOCATING* **using** *a00 a01 a02 a03*
get-bit-x-l1-b4[*OF - from-l-gt0 a11 i2-len*,*of ?j2*] *a8 zero-lt-len-levels*
mp-alloc-stm4-lvl-len[*OF a1 a8*] *len-eq mp-alloc-stm4-froml*[*OF a13*]
from-l-gt0 a13 j2-len
by (*meson Suc-lessD mult.commute*)

then have *get-bit-s x p′ ?i2 j* ≠ *DIVIDED* **by** *auto*
}
moreover {
assume *a05*:*j*≥ *?j2* + *1* ∧ *j*<*?j2*+*4*
then have *get-bit-s x p′ ?i2 j* = *FREE* **using** *a00 a01 a02 a03*
get-bit-x-l1-b41[*OF - from-l-gt0 a10*[*simplified from-l a13*[*THEN sym*]]
a11 i2-len,*of ?j2*] *a8 zero-lt-len-levels*
mp-alloc-stm4-lvl-len[*OF a1 a8*] *len-eq mp-alloc-stm4-froml*[*OF a13*]
from-l-gt0 a13 j2-len a11 mp-alloc-stm4-pre-precond-f-bn
by (*smt One-nat-def add.commute add-Suc-shift length-list-update-n*
list-updates-n-eq
numeral-2-eq-2 numeral-3-eq-3 numeral-Bit0 plus-1-eq-Suc)
then have *get-bit-s x p′ ?i2 j* ≠ *DIVIDED* **by** *auto*
} ultimately have *get-bit-s x p′ ?i2 j* ≠ *DIVIDED* **using** *a04* **by** *fastforce*
}
moreover{
assume ¬(*j*≥ *?j2* ∧ *j*<*?j2*+*4*)
moreover have *eq-get-bit-i-j*:*get-bit-s x p ?i2 j* = *get-bit-s Va p ?i2 j*
using *a03 from-l-suc same-bit-mp-alloc-x-va*[*OF*
a10[*simplified a13*[*simplified mp-alloc-stm4-froml*[*OF a13*], *THEN sym*]]
a11, *of ?i2 j*]
from-l-gt0 calculation
by *force*
ultimately have *get-bit-s x p′ ?i2 j* ≠ *DIVIDED*
using *a0 a02 a03* **unfolding** *inv-def inv-bitmapn-def Let-def*
by (*metis One-nat-def a01 a8 a10 a11 a1 length-list-update-n*
mp-alloc-stm4-inv-bits-len mp-alloc-stm4-lvl-len)
} ultimately have *get-bit-s x p′ ?k j* ≠ *DIVIDED* **using** *a03* **by** *auto*
}
moreover {
assume *?i2* ≠ *?k*
moreover have *?i2*<*?k*
using *calculation a00 a02 a8 i2-len mem-pools mp-alloc-stm4-lvl-len* **by** *auto*
then have *?i1*≠*?k*
by *linarith*
ultimately have *eq-get-bit-i-j*:*get-bit-s x p ?k j* = *get-bit-s Va p ?k j*
using *from-l-suc same-bit-mp-alloc-x-va*[*OF*
a10[*simplified a13*[*simplified mp-alloc-stm4-froml*[*OF a13*], *THEN sym*]]
a11, *of ?i2 j*]
from-l-gt0
by (*metis a10 a13 from-l same-bit-mp-alloc-stm4-pre-precond-f1*)
then have *get-bit-s x p′ ?k j* ≠ *DIVIDED*
using *a0 a02* **unfolding** *inv-def inv-bitmapn-def Let-def*
by (*metis One-nat-def a01 a8 a10 a11 a1 length-list-update-n*
mp-alloc-stm4-inv-bits-len mp-alloc-stm4-lvl-len)
} ultimately have *get-bit-s x p′ ?k j* ≠ *DIVIDED* **by** *auto*
} ultimately have *get-bit-s x p′ ?k j* ≠ *DIVIDED* **by** *auto*
}
then show ∀ *p*∈*mem-pools x*.

∀ *i*<*length* (*bits* (*levels* (*mem-pool-info x p*) ! (*length* (*levels* (*mem-pool-info x p*)) − *Suc NULL*))).
get-bit-s x p (*length* (*levels* (*mem-pool-info x p*)) − *Suc NULL*) *i* ≠ *DIVIDED* **by** *auto*

qed

lemma *mp-alloc-stm4-inv-bitmap4free*:
assumes
a0:*inv Va* **and**
a1:*freeing-node Va t* = *None* **and**
a2:*p* ∈ *mem-pools Va* **and**
a3:∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* **and**
a4:*length* (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) **and**
a5:*alloc-l Va t* < *int* (*n-levels* (*mem-pool-info Va p*)) **and**
a6:¬ *free-l Va t* < *OK* **and**
a7:*free-l Va t* ≤ *from-l Va t* **and**
a8:*allocating-node Va t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*))|) **and**
a9:*n* = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) = *NULL* **and**
a10:*block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) **and**
a11:*from-l Va t* < *alloc-l Va t* **and**
a12:*blk Va t* = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*)) **and**
a13:(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* **and**
a14:∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *pa* **and**
a15:∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) ! *jj* **and**
a16:*bits* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !

nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* **and**
a17:*free-list* (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)) =
inserts
(*map* (λ*ii*. *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ∗
ii +
blk (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
[*Suc NULL*..<*4*])
(*free-list*
(*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*))) **and**
a18:*lsizes x* = *lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a19:*from-l x* = *from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a20:*freeing-node x* = *freeing-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a21:*allocating-node x* = *allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) **and**
a22:*alloc-l Va t* = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* **and**
a23:*blk x* = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*)
shows *inv-bitmap-not4free x*
proof−
{ **fix** *p′ i j*
assume *a00*:*p′* ∈ *mem-pools x* **and**
a01:*i*<*length* (*levels* (*mem-pool-info x p′*)) **and**
a02:*j*<*length* (*bits* (*levels* (*mem-pool-info x p′*) ! *i*))
{ **assume** *a03*:*0* < *i* **and**
a04:*get-bit-s x p′ i* (*Suc* (*Suc* (*j div 4* ∗ *4*))) = *FREE* **and**
a05:*get-bit-s x p′ i* (*Suc* (*j div 4* ∗ *4*)) = *FREE* **and**
a06:*get-bit-s x p′ i* (*j div 4* ∗ *4*) = *FREE*
{ **assume** *p′*≠ *p*
then have *get-bit-s x p′ i* (*j div 4* ∗ *4* + *3*) ≠ *FREE*
using *a00 a01 a0 a8* **using** *a00 a01 a0 a8 a9 mp-alloc-stm4-mempools mp-alloc-stm4-pres-mpinfo*
unfolding *inv-bitmap-not4free-def inv-def*
by (*metis a02 a03 a04 a05 a06 a13 a14 add.commute*
add-2-eq-Suc′ partner-bits-def plus-1-eq-Suc)
} **note** *not-p* = *this*
moreover{
assume *a07*:*p′* = *p*
let *?i1*=(*nat* (*from-l Va t*)) **and**
?j1= (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))) **and**
?i2 = (*nat* (*from-l Va t* + *1*)) **and**
?j2 = (*block-num* (*mem-pool-info Va p*) (*blk Va t*) (*lsizes Va t* ! *nat* (*from-l Va t*))∗*4*)
have *from-l-gt0*:(*nat* (*from-l Va t* + *1*)) > *0*

```
      using a6 a7 by linarith
    have zero-lt-len-levels:0 < length (levels (mem-pool-info x p))
     using a0 a2 mp-alloc-stm4-lvl-len unfolding inv-mempool-info-def inv-def
      using a01 a07 gr-implies-not0 by blast
    then have len-eq:length (bits (levels (mem-pool-info x p) ! 0)) =
      length (bits (levels (mem-pool-info Va p) ! 0))
      using a13 mp-alloc-stm4-inv-bits-len
       unfolding gvars-conf-stable-def gvars-conf-def
       by fastforce
  have mem-pools:mem-pools x = mem-pools Va using mp-alloc-stm4-mempools[OF
a13] by auto
     have inv-mempool-info-mp Va p
        using a2 mem-pools a0 unfolding inv-def inv-mempool-info-def Let-def
by auto
     note inv-mempool=this[simplified Let-def]
     have from-l:from-l x = from-l Va
       using mp-alloc-stm4-froml[OF a19] by auto
     have from-l-suc:nat (from-l Va t + 1) = nat(from-l Va t) + 1
       using from-l-gt0 by auto
     have i1-len:?i1 < length (levels (mem-pool-info Va p))
        using a2 a11 a5  from-l-gt0 a0 unfolding inv-def inv-mempool-info-def
Let-def
       by auto
     have  i2-len:?i2 < length (levels (mem-pool-info Va p))
        using a0 a5 a2 a11 from-l-gt0 unfolding inv-def inv-mempool-info-def
Let-def
       by auto
     have j1-len:?j1 < length (bits (levels (mem-pool-info Va p) ! ?i1))
       using assms(11) assms(13) i1-len inv-mempool by presburger
      have j2-len:Suc (Suc (Suc ?j2)) < length (bits (levels (mem-pool-info Va
p) ! ?i2))
       using i1-len i2-len j1-len  inv-mempool  from-l-suc
       by simp

     have from-l-gt0:0 ≤ from-l Va t
       using a6 a7 by linarith
     { assume a08:i≠?i1 ∧ i≠?i2
       then have eq-get-bit-i-j:get-bit-s x p i (j div 4 * 4 + 3)  = get-bit-s Va
p i (j div 4 * 4 + 3)
         using same-bit-mp-alloc-x-va
               [OF a15[simplified a19[simplified mp-alloc-stm4-froml[OF a19],
                    THEN sym]] a16, of i (j div 4 * 4 + 3)]
            from-l-gt0 by auto
       moreover have eq-get-bit-i-j:get-bit-s x p i (j div 4 * 4)  = get-bit-s Va
p i (j div 4 * 4)
         using same-bit-mp-alloc-x-va
               [OF a15[simplified a19[simplified mp-alloc-stm4-froml[OF a19],
                    THEN sym]] a16, of i (j div 4 * 4)] a08
          from-l-gt0 by auto
```

moreover have *eq-get-bit-i-j*:*get-bit-s x p i* (*Suc* (*j div 4* ∗ *4*)) = *get-bit-s Va p i* (*Suc* (*j div 4* ∗ *4*))
using *same-bit-mp-alloc-x-va*
[*OF a15*[*simplified a19*[*simplified mp-alloc-stm4-froml*[*OF a19*],
THEN sym]] *a16*, *of i* (*Suc* (*j div 4* ∗ *4*))] *a08*
from-l-gt0 **by** *auto*
moreover have *eq-get-bit-i-j*:*get-bit-s x p i* (*Suc* (*Suc* (*j div 4* ∗ *4*))) = *get-bit-s Va p i* (*Suc* (*Suc* (*j div 4* ∗ *4*)))
using *same-bit-mp-alloc-x-va*
[*OF a15*[*simplified a19*[*simplified mp-alloc-stm4-froml*[*OF a19*],
THEN sym]] *a16*, *of i* (*Suc* (*Suc* (*j div 4* ∗ *4*)))] *a08*
from-l-gt0 **by** *auto*
ultimately have *get-bit-s x p′ i* (*j div 4* ∗ *4* + *3*) ≠ *FREE*
using *a07 a03 a04 a05 a06 a01 a02 a0 a13 a15 a16 a2 mp-alloc-stm4-inv-bits-len*
mp-alloc-stm4-lvl-len
unfolding *inv-bitmap-not4free-def inv-def Let-def partner-bits-def*
by (*metis add.commute add-2-eq-Suc′ length-list-update-n plus-1-eq-Suc*)
}
moreover **{** **assume** *i*=*?i1*
then have *get-bit-s x p′ i* (*j div 4* ∗ *4* + *3*) ≠ *FREE*
using *not-p a0 a02 a03 a04 a05 a06 a15 a19 a2 from-l from-l-gt0 from-l-suc i1-len j1-len*
mp-alloc-stm4-inv-bits-len mp-alloc-stm4-pre-precond-f-bitmap-not-free
same-bit-mp-alloc-stm4-pre-precond-f1
unfolding *inv-bitmap-not4free-def invariant.inv-def partner-bits-def*
by (*smt add-2-eq-Suc′ add-eq-self-zero le-zero-eq nat-int-add not-one-le-zero*
plus-1-eq-Suc)
}note *i1*= *this*
moreover **{**
assume *a08*:*i*=*?i2*
{ **assume** *j* ≥ *?j2* ∧ *j* ≤ *?j2* + *3*
then have *j* = *?j2* ∨ *j*= *?j2* + *1* ∨ *j*= *?j2* + *2* ∨ *j* = *?j2* + *3*
by *auto*
then have *j div 4* ∗ *4* = *?j2* **by** *auto*
moreover have *get-bit-s x p′ i ?j2* = *ALLOCATING*
using *get-bit-x-l1-b4*[*OF - from-l-gt0 a16 i2-len*] *a08 a07 mult.commute j2-len*
by (*metis Suc-lessD*)
ultimately have *get-bit-s x p′ i* (*j div 4* ∗ *4* + *3*) ≠ *FREE* **using** *a06* **by** *auto*
}
moreover **{**
assume ¬(*j*≥*?j2* ∧ *j*≤ *?j2* + *3*)
then have *j* < *?j2* ∨ *j* > *?j2* + *3*
by *auto*
moreover **{** **assume** *j*<*?j2*
then have *j div 4* ∗ *4* + *3* < *?j2*
by *presburger*
moreover have *get-bit-s x p i* (*j div 4*∗*4*) = *get-bit-s Va p i* (*j div*

*4*4*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4*4+1*) = *get-bit-s Va p i* (*j div 4*4+1*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4+1*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4*4+2*) = *get-bit-s Va p i* (*j div 4*4+2*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4+2*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4*4+3*) = *get-bit-s Va p i* (*j div 4*4+3*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4+3*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
ultimately have *get-bit-s x p' i* (*j div 4 * 4 + 3*) $\neq$ *FREE*
using *same-bit-mp-alloc-x-va*[*OF - a16*] *a15 a01 a02 a03 a04 a05 a06 a07 a08 a00*
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len i1
unfolding *inv-def inv-bitmap-not4free-def partner-bits-def*
by (*smt add.commute add-2-eq-Suc' length-list-update-n plus-1-eq-Suc*)
}
moreover{
assume *j*>*?j2 + 3*
then have *j div 4 * 4 > ?j2 + 3*
by *presburger*
moreover have *get-bit-s x p i* (*j div 4*4*) = *get-bit-s Va p i* (*j div 4*4*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4*4+1*) = *get-bit-s Va p i* (*j div 4*4+1*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4*4+1*)]

a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4**4+2*) = *get-bit-s Va p i* (*j div 4**4+2*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4**4+2*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
moreover have *get-bit-s x p i* (*j div 4**4+3*) = *get-bit-s Va p i* (*j div 4**4+3*)
using *same-bit-mp-alloc-x-va*[*OF a15*[*simplified from-l a19*[*THEN sym*]] *a16*, *of i* (*j div 4**4+3*)]
a01 a02 a03 a04 a05 a06 a07 a08 a00 calculation
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len from-l-suc
by (*auto simp add*: *a16*)
ultimately have *get-bit-s x p′ i* (*j div 4* * *4* + *3*) ≠ *FREE*
using *same-bit-mp-alloc-x-va*[*OF - a16*] *a15 a01 a02 a03 a04 a05 a06 a07 a08 a00*
a0 a16 a19 a2 from-l i2-len mp-alloc-stm4-inv-bits-len i1
unfolding *inv-def inv-bitmap-not4free-def partner-bits-def*
by (*smt add.commute add-2-eq-Suc′ length-list-update-n plus-1-eq-Suc*)
}
ultimately have *get-bit-s x p′ i* (*j div 4* * *4* + *3*) ≠ *FREE* **by** *auto*
} ultimately have *get-bit-s x p′ i* (*j div 4* * *4* + *3*) ≠ *FREE* **by** *auto*
} ultimately have *get-bit-s x p′ i* (*j div 4* * *4* + *3*) ≠ *FREE* **by** *auto*
} ultimately have *get-bit-s x p′ i* (*j div 4* * *4* + *3*) ≠ *FREE* **by** *auto*
}
} then show *inv-bitmap-not4free x*
unfolding *inv-bitmap-not4free-def Let-def partner-bits-def*
by *auto*
qed

lemma *mp-alloc-stm4-whlpst-in-post-inv*:
inv Va ⟹
freeing-node Va t = *None* ⟹
p ∈ *mem-pools Va* ⟹
ETIMEOUT ≤ *timeout* ⟹
timeout = *ETIMEOUT* ⟶ *tmout Va t* = *ETIMEOUT* ⟹
¬ *rf Va t* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ˆ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
¬ *free-l Va t* < *OK* ⟹
NULL < *buf* (*mem-pool-info Va p*) ∨ *NULL* < *n* ∧ *NULL* < *max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) ⟹
free-l Va t ≤ *from-l Va t* ⟹

allocating-node Va t =
Some (|*pool* = *p*, *level* = *nat* (*from-l Va t*),
block = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*
4 ˆ *nat* (*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)),
data = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div*
4 ˆ *nat* (*from-l Va t*))|) ⟹
n = *block-num* (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat*
(*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*)) ∨
max-sz (*mem-pool-info Va p*) *div 4* ˆ *nat* (*from-l Va t*) = *NULL* ⟹
block-num (*mem-pool-info Va p*)
(*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ˆ *nat*
(*from-l Va t*)))
(*lsizes Va t* ! *nat* (*from-l Va t*))
< *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) ⟹
from-l Va t < *alloc-l Va t* ⟹
cur Va = *Some t* ⟹
n < *n-max* (*mem-pool-info Va p*) ∗ *4* ˆ *nat* (*from-l Va t*) ⟹
blk Va t = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4*
ˆ *nat* (*from-l Va t*)) ⟹
mempoolalloc-ret Va t = *None* ⟹
∀ *ii*≤*nat* (*alloc-l Va t*). *sz* ≤ *lsizes Va t* ! *ii* ⟹
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels*
(*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*)
< *sz* ⟹
i x t = *4* ⟹
cur x = *cur* (*mp-alloc-stm4-pre-precond-f Va t p*) ⟹
tick x = *tick* (*mp-alloc-stm4-pre-precond-f Va t p*) ⟹
thd-state x = *thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) ⟹
(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
∀ *pa*. *pa* ≠ *p* ⟶ *mem-pool-info x pa* = *mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *pa* ⟹
wait-q (*mem-pool-info x p*) = *wait-q* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*) ⟹
∀ *t'*. *t'* ≠ *t* ⟶ *lvars-nochange t' x* (*mp-alloc-stm4-pre-precond-f Va t p*) ⟹
∀ *jj*. *jj* ≠ *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*) ⟶
levels (*mem-pool-info x p*) ! *jj* = *levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f*
Va t p) *p*) ! *jj* ⟹
bits (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*)
t + *1*)) =
list-updates-n
(*bits* (*levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*) !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)))
(*Suc* (*bn* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* ∗ *4*)) *3 FREE* ⟹
free-list (*levels* (*mem-pool-info x p*) ! *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va*

t p) t + 1)) =
inserts
(map (λii. lsizes (mp-alloc-stm4-pre-precond-f Va t p) t !
*nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1) **
ii +
blk (mp-alloc-stm4-pre-precond-f Va t p) t)
[Suc NULL..<4])
(free-list
(levels (mem-pool-info (mp-alloc-stm4-pre-precond-f Va t p) p) !
nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1))) ⟹
j x = j (mp-alloc-stm4-pre-precond-f Va t p) ⟹
ret x = ret (mp-alloc-stm4-pre-precond-f Va t p) ⟹
endt x = endt (mp-alloc-stm4-pre-precond-f Va t p) ⟹
rf x = rf (mp-alloc-stm4-pre-precond-f Va t p) ⟹
tmout x = tmout (mp-alloc-stm4-pre-precond-f Va t p) ⟹
lsizes x = lsizes (mp-alloc-stm4-pre-precond-f Va t p) ⟹
alloc-l x = alloc-l (mp-alloc-stm4-pre-precond-f Va t p) ⟹
free-l x = free-l (mp-alloc-stm4-pre-precond-f Va t p) ⟹
from-l x = from-l (mp-alloc-stm4-pre-precond-f Va t p) ⟹
blk x = blk (mp-alloc-stm4-pre-precond-f Va t p) ⟹
nodev x = nodev (mp-alloc-stm4-pre-precond-f Va t p) ⟹
bn x = bn (mp-alloc-stm4-pre-precond-f Va t p) ⟹
alloc-lsize-r x = alloc-lsize-r (mp-alloc-stm4-pre-precond-f Va t p) ⟹
lvl x = lvl (mp-alloc-stm4-pre-precond-f Va t p) ⟹
bb x = bb (mp-alloc-stm4-pre-precond-f Va t p) ⟹
block-pt x = block-pt (mp-alloc-stm4-pre-precond-f Va t p) ⟹
th x = th (mp-alloc-stm4-pre-precond-f Va t p) ⟹
need-resched x = need-resched (mp-alloc-stm4-pre-precond-f Va t p) ⟹
mempoolalloc-ret x = mempoolalloc-ret (mp-alloc-stm4-pre-precond-f Va t p) ⟹
freeing-node x = freeing-node (mp-alloc-stm4-pre-precond-f Va t p) ⟹
allocating-node x = allocating-node (mp-alloc-stm4-pre-precond-f Va t p) ⟹ inv x
apply(*simp add:inv-def*)
apply(*rule conjI*) **apply**(*simp add:inv-cur-def*) **apply** *clarify* **using** *mp-alloc-stm4-inv-cur*
apply *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-thd-state* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mempool-info* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-bitmap-freelist* **apply** *blast*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-bitmap* **unfolding** *inv-def* **apply** *blast*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-aux-vars* **unfolding** *inv-def* **apply** *blast*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-bitmap0* **unfolding** *inv-def* **apply** *blast*
apply (*rule conjI*) **using** *mp-alloc-stm4-inv-bitmapn* **unfolding** *inv-def* **apply** *blast*
using *mp-alloc-stm4-inv-bitmap4free* **unfolding** *inv-def* **by** *blast*

lemma *mp-alloc-stm4-whlpst-in-post-h1*:
p ∈ *mem-pools Va* ⟹
inv Va ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
from-l Va t < *alloc-l Va t* ⟹
¬ *free-l Va t* < *0* ⟹
free-l Va t ≤ *from-l Va t* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* ⟹
blk Va t = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) ⟹
(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
allocating-node (*mp-alloc-stm4-pre-precond-f Va t p*) *t* =
Some (|*pool* = *p*, *level* = *nat* (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*),
block = *block-num* (*mem-pool-info x p*) (*blk* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)
(*lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* !
nat (*from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* + *1*)),
data = *blk* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*|)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def block-num-def*)
apply(*rule subst*[**where** *s*=*buf* (*mem-pool-info Va p*) **and** *t*=*buf* (*mem-pool-info x p*)])
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def set-bit-def*)

apply(*rule subst*[**where** *s*=*n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) **and** *t*=*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) −
buf (*mem-pool-info Va p*)])
apply *arith*

apply(*subgoal-tac* ∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii*)
prefer *2* **using** *inv-maxsz-align4*[*of Va*] **apply** *metis*

apply(*rule subst*[**where** *s*=*lsizes Va t* ! *nat* (*from-l Va t*) *div 4* **and** *t*=*lsizes Va t* ! *nat* (*from-l Va t* + *1*)])
apply (*smt div-mult-self1-is-m mp-alloc-stm4-blockfit-help4 nat-less-iff semiring-normalization-rules*(*7*) *zero-less-numeral*)
by (*smt div-eq-0-iff m-mod-div mod-mult-self2-is-0 mp-alloc-stm4-blockfit-help4 nat-less-iff nonzero-mult-div-cancel-right semiring-normalization-rules*(*7*))

lemma *mp-alloc-stm4-whlpst-in-post-h2*:

p ∈ *mem-pools Va* ⟹
inv Va ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
from-l Va t < *alloc-l Va t* ⟹
¬ *free-l Va t* < *0* ⟹
free-l Va t ≤ *from-l Va t* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* ⟹
blk Va t = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) ⟹
(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
data (*the* (*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)) =
buf (*mem-pool-info x p*) +
block (*the* (*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)) ∗
(*max-sz* (*mem-pool-info x p*) *div 4* ^ *level* (*the* (*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*)))
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def block-num-def*)

apply(*rule subst*[**where** *s*=*buf* (*mem-pool-info Va p*) **and** *t*=*buf* (*mem-pool-info x p*)])
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def set-bit-def*)

apply(*rule subst*[**where** *s*=*n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) **and** *t*=*buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) −
buf (*mem-pool-info Va p*)])
apply *arith*

apply(*subgoal-tac* ∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii*)
prefer *2* **using** *inv-maxsz-align4*[*of Va*] **apply** *metis*

apply(*rule subst*[**where** *s*=*lsizes Va t* ! *nat* (*from-l Va t*) *div 4* **and** *t*=*lsizes Va t* ! *nat* (*from-l Va t* + *1*)])
apply (*smt div-mult-self1-is-m mp-alloc-stm4-blockfit-help4 nat-less-iff*
semiring-normalization-rules(*7*) *zero-less-numeral*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info Va p*) **and** *t*=*max-sz* (*mem-pool-info x p*)])
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def set-bit-def*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*) *div 4*
and *t*=*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t* + *1*)])
apply (*metis inv-maxsz-align4 mp-alloc-stm4-blockfit-help4 nonzero-mult-div-cancel-right*

zero-neq-numeral)
apply(*rule subst*[**where** *s=max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)
and *t=lsizes Va t* ! *nat* (*from-l Va t*)])
apply (*smt nat-less-iff*)
apply(*subgoal-tac* ∃ *m*>*0*. *max-sz* (*mem-pool-info Va p*) = (*4* ∗ *m*) ∗ (*4* ^ *n-levels* (*mem-pool-info Va p*)))
prefer *2* **apply**(*simp add:inv-def inv-mempool-info-def Let-def*) **apply** *metis*
by (*smt add-left-cancel inv-maxsz-align4 mp-alloc-stm4-blockfit-help4 mult.assoc mult-is-0 nonzero-mult-div-cancel-left semiring-normalization-rules*(*7*))

lemma *mp-alloc-stm4-whlpst-in-post-h3-1*:
from-l Va t ≥*0* ⟹ *n* < *n-max* (*mem-pool-info Va p*) ∗ *4* ^ *nat* (*from-l Va t*) ⟹
4 ∗ *n* < *n-max* (*mem-pool-info Va p*) ∗ *4* ^ *nat* (*from-l Va t* + *1*)
by (*smt mult.assoc Divides.div-mult2-eq Suc-nat-eq-nat-zadd1 div-eq-0-iff div-mult-mult1-if gr-implies-not0 mult.commute mult-eq-0-iff power-Suc semiring-normalization-rules*(*7*) *zero-less-numeral*)

lemma *mp-alloc-stm4-whlpst-in-post-h3*:
p ∈ *mem-pools Va* ⟹
inv Va ⟹
alloc-l Va t < *int* (*n-levels* (*mem-pool-info Va p*)) ⟹
from-l Va t < *alloc-l Va t* ⟹
¬ *free-l Va t* < *0* ⟹
free-l Va t ≤ *from-l Va t* ⟹
∀ *ii*<*length* (*lsizes Va t*). *lsizes Va t* ! *ii* = *ALIGN4* (*max-sz* (*mem-pool-info Va p*)) *div 4* ^ *ii* ⟹
length (*lsizes Va t*) ≤ *n-levels* (*mem-pool-info Va p*) ⟹
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *1* ∧ *length* (*lsizes Va t*) = *n-levels* (*mem-pool-info Va p*) ∨
alloc-l Va t = *int* (*length* (*lsizes Va t*)) − *2* ∧ *lsizes Va t* ! *nat* (*alloc-l Va t* + *1*) < *sz* ⟹
n < *n-max* (*mem-pool-info Va p*) ∗ *4* ^ *nat* (*from-l Va t*) ⟹
(*x*, *mp-alloc-stm4-pre-precond-f Va t p*) ∈ *gvars-conf-stable* ⟹
blk Va t = *buf* (*mem-pool-info Va p*) + *n* ∗ (*max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)) ⟹
block (*the* (*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*))
< *n-max* (*mem-pool-info x p*) ∗ *4* ^ *level* (*the* (*allocating-node* (*mp-alloc-stm4-pre-precond-f Va t p*) *t*))
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def block-num-def*)
apply(*rule subst*[**where** *s=max-sz* (*mem-pool-info Va p*) *div 4* ^ *nat* (*from-l Va t*)
and *t=lsizes Va t* ! *nat* (*from-l Va t*)])
using *inv-maxsz-align4* **apply** *auto*[*1*]
apply(*rule subst*[**where** *s=n-max* (*mem-pool-info Va p*) **and** *t=n-max* (*mem-pool-info x p*)])
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def set-bit-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* ∃ *m*>*0*. *max-sz* (*mem-pool-info Va p*) = (*4* ∗ *m*) ∗ (*4* ^ *n-levels* (*mem-pool-info Va p*)))

prefer *2* **apply**(*simp add:inv-def inv-mempool-info-def Let-def*) **apply** *metis*
apply(*subgoal-tac nat (from-l Va t) < n-levels (mem-pool-info Va p)*) **prefer** *2*
apply *linarith*

apply(*rule subst*[**where** *s=n* **and** *t=(n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)) div*
(max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)))])
apply (*simp add: mp-alloc-stm3-lm2-inv-1-2*)
apply *clarsimp*

apply(*rule mp-alloc-stm4-whlpst-in-post-h3-1*)
apply *arith* **apply** *blast*

done

lemma *mp-alloc-stm4-whlpst-in-post-h4*:
p ∈ mem-pools Va ⟹
inv Va ⟹
alloc-l Va t < int (n-levels (mem-pool-info Va p)) ⟹
from-l Va t < alloc-l Va t ⟹
¬ free-l Va t < 0 ⟹
free-l Va t ≤ from-l Va t ⟹
∀ ii<length (lsizes Va t). lsizes Va t ! ii = ALIGN4 (max-sz (mem-pool-info Va p)) div 4 ^ ii ⟹
length (lsizes Va t) ≤ n-levels (mem-pool-info Va p) ⟹
alloc-l Va t = int (length (lsizes Va t)) − 1 ∧ length (lsizes Va t) = n-levels (mem-pool-info Va p) ∨
alloc-l Va t = int (length (lsizes Va t)) − 2 ∧ lsizes Va t ! nat (alloc-l Va t + 1) < sz ⟹
n < n-max (mem-pool-info Va p) ∗ 4 ^ nat (from-l Va t) ⟹
blk Va t = buf (mem-pool-info Va p) + n ∗ (max-sz (mem-pool-info Va p) div 4 ^ nat (from-l Va t)) ⟹
(x, mp-alloc-stm4-pre-precond-f Va t p) ∈ gvars-conf-stable ⟹
(∃ n<n-max (mem-pool-info x p) ∗ 4 ^ nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1).
blk (mp-alloc-stm4-pre-precond-f Va t p) t =
buf (mem-pool-info x p) +
n ∗ (max-sz (mem-pool-info x p) div 4 ^ nat (from-l (mp-alloc-stm4-pre-precond-f Va t p) t + 1)))
apply(*rule subst*[**where** *s=n-max (mem-pool-info Va p)* **and** *t=n-max (mem-pool-info x p)*])
apply(*simp add:gvars-conf-stable-def gvars-conf-def mp-alloc-stm4-pre-precond-f-def set-bit-def*)
apply(*rule subst*[**where** *s=buf (mem-pool-info Va p)* **and** *t=buf (mem-pool-info x p)*])
apply(*simp add:gvars-conf-stable-def gvars-conf-def mp-alloc-stm4-pre-precond-f-def set-bit-def*)
apply(*rule subst*[**where** *s=from-l Va* **and** *t=from-l (mp-alloc-stm4-pre-precond-f Va t p)*])

apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*rule subst*[**where** *s*=*blk Va* **and** *t*=*blk* (*mp-alloc-stm4-pre-precond-f Va t p*)])
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*rule subst*[**where** *s*=*max-sz* (*mem-pool-info Va p*) **and** *t*=*max-sz* (*mem-pool-info x p*)])
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def mp-alloc-stm4-pre-precond-f-def set-bit-def*)
apply(*rule exI*[**where** $x=4 * n$])
by (*smt inv-maxsz-align4 mp-alloc-stm4-blockfit-help4 mp-alloc-stm4-whlpst-in-post-h3-1 mult.assoc semiring-normalization-rules*(*7*))

lemma *mp-alloc-stm4-whlpst-in-post*:
$Va \in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ $\{\!|\,$´*cur* = *Some t*$|\!\}$ $\Longrightarrow$
mp-alloc-stm4-loopinv (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* $\cap$ $\{\!|\,$´*i t* $\geq$ *4*$|\!\}$
$\subseteq$ $\{\!|\,$´(*Pair Va*) $\in$ *Mem-pool-alloc-guar t*$|\!\}$ $\cap$ *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **using** *mp-alloc-stm4-mempools2* **apply** *metis*
apply *clarify*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mif-buf* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mif-mxsz* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mif-nmax* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mif-nlvls* **apply** *metis*
apply(*rule conjI*) **using** *mp-alloc-stm4-inv-mif-len* **apply** *metis*
apply *clarify* **using** *mp-alloc-stm4-inv-bits-len* **apply** *metis*
apply(*rule conjI*)
using *mp-alloc-stm4-whlpst-in-post-inv*[*of Va t p timeout - sz -*] **apply** *auto*[*1*]
apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac lvars-nochange t′ x* (*mp-alloc-stm4-pre-precond-f Va t p*))
prefer *2* **apply** *metis*
apply(*subgoal-tac lvars-nochange t′ Va* (*mp-alloc-stm4-pre-precond-f Va t p*))
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-lvars-nochange*[*of - t Va p*] **apply** *metis*
using *lvars-nochange-trans*[*of - Va mp-alloc-stm4-pre-precond-f Va t p -*] *lvars-nochange-sym* **apply** *metis*
using *mp-alloc-stm4-pre-precond-f-tick* **apply** *metis*

apply(*rule conjI*)
apply *clarsimp*
using *mp-alloc-stm4-whlpst-in-post-inv*[*of Va t p timeout - sz -*] **apply** *auto*[*1*]

apply(*rule conjI*)

apply *clarsimp*
using *mp-alloc-stm4-pre-precond-f-def-frnode* **apply** *metis*

apply(*rule conjI*)
apply *clarsimp*
using *mp-alloc-stm4-pre-precond-f-mpls* **apply** *metis*

apply(*rule conjI*)
apply *clarsimp*
apply(*rule conjI*) **apply** *clarsimp*
apply(*subgoal-tac rf Va t*) **prefer** *2* **using** *mp-alloc-stm4-pre-precond-f-rf* **apply** *metis*
apply *fast*
apply(*rule conjI*) **apply** *clarsimp*
using *mp-alloc-stm4-pre-precond-f-ret* **apply** *metis*
apply *clarsimp* **using** *mp-alloc-stm4-pre-precond-f-tmout* **apply** *metis*

apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac rf Va t*) **prefer** *2* **using** *mp-alloc-stm4-pre-precond-f-rf* **apply** *metis*
apply *fast*

apply(*rule conjI*)
apply *clarsimp*
apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac max-sz* (*mem-pool-info x p*) = *max-sz* (*mem-pool-info Va p*))
prefer *2* **apply**(*subgoal-tac max-sz* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*)
= *max-sz* (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-pre-maxsz* **apply** *metis*
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *lsizes Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-lsz* **apply** *metis*
apply *metis*
apply(*subgoal-tac n-levels* (*mem-pool-info x p*) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*subgoal-tac n-levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*)
= *n-levels* (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-inv-mif-nlvls* **apply** *metis*
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *lsizes Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-lsz* **apply** *metis*
apply *metis*
apply(*rule conjI*)
apply(*subgoal-tac alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *alloc-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*

apply *metis*
apply(*rule conjI*)
apply(*rule subst*[**where** *t= free-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* **and** *s= free-l Va t*])
using *mp-alloc-stm4-pre-precond-f-freel* **apply** *metis*
apply *linarith*
apply(*rule conjI*)
apply(*subgoal-tac alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *alloc-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*
apply(*rule subst*[**where** *t= free-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* **and** *s= free-l Va t*])
using *mp-alloc-stm4-pre-precond-f-freel* **apply** *metis*
apply *linarith*

apply(*rule disjI2*)
apply(*rule subst*[**where** *s=alloc-l Va* **and** *t=alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*)])
using *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*
apply(*rule subst*[**where** *s=lsizes Va* **and** *t=lsizes* (*mp-alloc-stm4-pre-precond-f Va t p*)])
using *mp-alloc-stm4-pre-precond-f-lsz* **apply** *metis*
apply(*rule conjI*) **apply** *linarith*
apply(*rule conjI*) **apply** *blast*
apply(*subgoal-tac n-levels* (*mem-pool-info x p*) = *n-levels* (*mem-pool-info Va p*))
prefer *2* **apply**(*subgoal-tac n-levels* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p*) *p*)
= *n-levels* (*mem-pool-info Va p*))
prefer *2* **using** *mp-alloc-stm4-inv-mif-nlvls* **apply** *metis*
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply *metis*

apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *alloc-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*
apply *arith*

apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac free-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *free-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-freel* **apply** *metis*
apply *arith*

apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac blk Va t* > *0*) **prefer** *2*
apply(*simp add*:*inv-def inv-mempool-info-def*)
apply(*subgoal-tac blk* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *blk Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-blk* **apply** *metis*

apply *arith*

apply(*rule conjI*)
apply *clarsimp*
apply(*subgoal-tac alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *alloc-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*
apply(*subgoal-tac from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *from-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-froml* **apply** *metis*
apply *arith*

apply *clarsimp*
apply(*rule conjI*)
apply(*subgoal-tac alloc-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *alloc-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-allocl* **apply** *metis*
apply(*subgoal-tac from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *from-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-froml* **apply** *metis*
apply *arith*

apply(*rule conjI*)
apply(*subgoal-tac from-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *from-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-froml* **apply** *metis*
apply(*subgoal-tac free-l* (*mp-alloc-stm4-pre-precond-f Va t p*) *t* = *free-l Va t*)
prefer *2* **using** *mp-alloc-stm4-pre-precond-f-freel* **apply** *metis*
apply *arith*

apply(*rule conjI*)
using *mp-alloc-stm4-whlpst-in-post-h1* **apply** *blast*

apply(*rule conjI*)
using *mp-alloc-stm4-whlpst-in-post-h2* **apply** *blast*

apply(*rule conjI*)
using *mp-alloc-stm4-whlpst-in-post-h3* **apply** *blast*

using *mp-alloc-stm4-whlpst-in-post-h4* **apply** *blast*
done

lemma *thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) = *thd-state Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma *thd-state* (*mp-alloc-stm4-pre-precond-f Va t p*) = *thd-state Va*
by(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)

lemma ∀ *p*∈*mem-pools Va*. *wait-q* (*mem-pool-info Va p*) = *wait-q* (*mem-pool-info* (*mp-alloc-stm4-pre-precond-f Va t p1*) *p*)
apply *clarify*
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*)
apply(*simp add*: *set-bit-def*)

done

term *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout*
term *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*

lemma *mp-alloc-stm4-lm1-1*:
$Va \in$ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* $\cap$ {|´*cur* = *Some t*|} $\Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*bn* := ´*bn* (*t* := *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*from-l t*))));;
´*mem-pool-info* := *set-bit-divide* ´*mem-pool-info p* (*nat* (´*from-l t*)) (´*bn t*);;
´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (*4* ∗ ´*bn t*);;
´*allocating-node* := ´*allocating-node* (*t* := *Some* (|*pool* = *p*, *level* = *nat* (´*from-l t* + *1*),
block = *4* ∗ ´*bn t*, *data* = ´*blk t* |));;
FOR ´*i* := ´*i* (*t* := *Suc 0*);
´*i t* < *4*;
´*i* := ´*i* (*t* := *Suc* (´*i t*)) *DO*
´*lbn* := ´*lbn* (*t* := *4* ∗ ´*bn t* + ´*i t*);;
´*lsz* := ´*lsz* (*t* := (´*lsizes t*) ! (*nat* (´*from-l t* + *1*)));;
´*block2* := ´*block2*(*t* := ´*lsz t* ∗ ´*i t* + ´*blk t*);;
´*mem-pool-info* := *set-bit-free* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (´*lbn t*);;

IF block-fits (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*) *THEN*
´*mem-pool-info* := ´*mem-pool-info* (*p* :=
append-free-list (´*mem-pool-info p*) (*nat* (´*from-l t* + *1*)) (´*block2 t*))
FI
ROF) sat_p [{*Va*}, {(*s*, *t*). *s* = *t*}, *UNIV*,
{|´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*|} ∩ *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*]

apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-pre-precond4*
(*mp-alloc-stm4-pre-precond3*
(*mp-alloc-stm4-pre-precond2*
(*mp-alloc-stm4-pre-precond1 Va t p*) *t p*) *t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-pre-precond3*
(*mp-alloc-stm4-pre-precond2*
(*mp-alloc-stm4-pre-precond1 Va t p*) *t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-pre-precond2*
(*mp-alloc-stm4-pre-precond1 Va t p*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-pre-precond1 Va t p*}])

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Seq*[**where** *mid*={*mp-alloc-stm4-pre-precond-f Va t p*}])
apply(*rule Basic*)
apply(*simp add*:*mp-alloc-stm4-pre-precond-f-def*) **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Conseq*[**where** *pre*={*mp-alloc-stm4-pre-precond-f Va t p*}
and *pre'*=*mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p*
and *rely*={(*s*, *t*).*s* = *t*} **and** *rely'*={(*s*, *t*).*s* = *t*} **and** *guar*=*UNIV* **and** *guar'*=*UNIV*

and *post'*=*mp-alloc-stm4-loopinv* (*mp-alloc-stm4-pre-precond-f Va t p*) *t p* ∩ {|´*i t* ≥ *4*|}])
using *mp-alloc-stm4-pre-precond-f-in-mp-alloc-stm4-loopinv* **apply** *auto*[*1*]
apply *simp* **apply** *simp* **using** *mp-alloc-stm4-whlpst-in-post*[*of Va t p timeout sz*] **apply** *argo*
using *mp-alloc-stm4-while*[*of Va t p timeout sz*] **apply** *fastforce*
done

term *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|}
term *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*
term {|´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*|} ∩ *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*

lemma *mp-alloc-stm4-lm1*:
mp-alloc-precond2-1-1-loopinv-0 t p sz timeout ∩ {|´*cur* = *Some t*|} ∩ {*Va*} = {*Va*} ⟹
$\Gamma \vdash_I$ *Some* (´*bn* := ´*bn*(*t* := *block-num* (´*mem-pool-info p*) (´*blk t*) (´*lsizes t* ! *nat* (´*from-l t*)));;
´*mem-pool-info* := *set-bit-divide* ´*mem-pool-info p* (*nat* (´*from-l t*)) (´*bn t*);;
´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (*4* ∗ ´*bn t*);;
´*allocating-node* := ´*allocating-node*(*t* ↦ (|*pool* = *p*, *level* = *nat* (´*from-l t* + *1*), *block* = *4* ∗ ´*bn t*, *data* = ´*blk t*|));;
(´*i* := ´*i*(*t* := *Suc NULL*);;
WHILE ´*i t* < *4*
DO ´*lbn* := ´*lbn*(*t* := *4* ∗ ´*bn t* + ´*i t*);; ´*lsz* := ´*lsz*(*t* := ´*lsizes t* ! *nat* (´*from-l t* + *1*));;
´*block2* := ´*block2*(*t* := ´*lsz t* ∗ ´*i t* + ´*blk t*);;
´*mem-pool-info* := *set-bit-free* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (´*lbn t*);;
IF block-fits (´*mem-pool-info p*) (´*block2 t*)

(´*lsz t*) *THEN* ´*mem-pool-info* := ´*mem-pool-info*
(*p* := *append-free-list* (´*mem-pool-info p*) (*nat* (´*from-l t* + *1*)) (´*block2 t*)) *FI*;;
´*i* := ´*i*(*t* := *Suc* (´*i t*))
OD)) sat_p [*mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*Va*},
{(*s*, *t*). *s* = *t*}, *UNIV*,
{|´(*Pair Va*) ∈ *Mem-pool-alloc-guar t*|} ∩ *mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*]
apply(*rule subst*[**where** *t*=*mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*Va*} **and** *s*={*Va*}])
apply *metis*
apply(*subgoal-tac Va* ∈ *mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ {|´*cur* = *Some t*|})
prefer *2* **apply** *auto*[*1*]
using *mp-alloc-stm4-lm1-1* **apply** *meson*
done

term *mp-alloc-precond2-1-1-loopinv t p sz timeout*
term *mp-alloc-precond2-1-2 t p sz timeout*

lemma *mp-alloc-stm4-lm*:
Γ $\vdash_I$ *Some* (*WHILE* ´*from-l t* < ´*alloc-l t DO*
(*t* ▶ *ATOMIC*
´*bn* := ´*bn* (*t* := *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*from-l t*))));;

´*mem-pool-info* := *set-bit-divide* ´*mem-pool-info p* (*nat* (´*from-l t*)) (´*bn t*);;

´*mem-pool-info* := *set-bit-allocating* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (*4* ∗ ´*bn t*);;

´*allocating-node* := ´*allocating-node* (*t* := *Some* (|*pool* = *p*, *level* = *nat* (´*from-l t* + *1*),
block = *4* ∗ ´*bn t*, *data* = ´*blk t* |));;

FOR ´*i* := ´*i* (*t* := *1*);
´*i t* < *4*;
´*i* := ´*i* (*t* := ´*i t* + *1*) *DO*
´*lbn* := ´*lbn* (*t* := *4* ∗ ´*bn t* + ´*i t*);;
´*lsz* := ´*lsz* (*t* := (´*lsizes t*) ! (*nat* (´*from-l t* + *1*)));;
´*block2* := ´*block2*(*t* := ´*lsz t* ∗ ´*i t* + ´*blk t*);;

´*mem-pool-info* := *set-bit-free* ´*mem-pool-info p* (*nat* (´*from-l t* + *1*)) (´*lbn t*);;

IF block-fits (´*mem-pool-info p*) (´*block2 t*) (´*lsz t*) *THEN*

´*mem-pool-info* := ´*mem-pool-info* (*p* :=

append-free-list (´*mem-pool-info p*) (*nat* (´*from-l t* + *1*)) (´*block2 t*))
FI
ROF

END);;
(t ▶ ´*from-l* := ´*from-l*(t := ´*from-l t* + *1*))
OD) sat_p [*mp-alloc-precond2-1-1-loopinv t p sz timeout*, *Mem-pool-alloc-rely t*, *Mem-pool-alloc-guar t*,
mp-alloc-precond2-1-2 t p sz timeout]
apply(*rule While*)
using *mp-alloc-precond2-1-1-loopinv-stb* **apply** *simp*
apply(*rule Int-greatest*) **apply**(*rule Int-greatest*) **apply**(*rule Int-greatest*)
apply(*rule Int-greatest*) **apply**(*rule Int-greatest*) **apply**(*rule Int-greatest*)
apply(*rule Int-greatest*) **apply**(*rule Int-greatest*)
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*] **apply** *clarify* **apply** *auto*[*1*] **apply** *auto*[*1*]
apply(*rule subst*[**where** *t*=⦃´*from-l t* ≤ ´*alloc-l t* ∧ ´*allocating-node t* =
Some (|*pool* = *p*, *level* = *nat* (´*from-l t*), *block* = *block-num* (´*mem-pool-info p*) (´*blk t*) (´*lsizes t* ! *nat* (´*from-l t*)),
data = ´*blk t*|)⦄ **and** *s*=⦃´*from-l t* ≤ ´*alloc-l t*⦄ ∩ ⦃´*allocating-node t* =
Some (|*pool* = *p*, *level* = *nat* (´*from-l t*), *block* = *block-num* (´*mem-pool-info p*) (´*blk t*) (´*lsizes t* ! *nat* (´*from-l t*)),
data = ´*blk t*|)⦄]) **apply** *auto*[*1*]

using *mp-alloc-precond2-1-2-stb* **apply** *simp*

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond2-1-1-loopinv-1 t p sz timeout*])

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond2-1-1-loopinv-0-stb* **apply** *auto*[*1*]
using *mp-alloc-precond2-1-1-loopinv-1-stb* **apply** *simp*
apply *clarify*
apply(*rule Await*)
using *stable-id2* **apply** *fast* **using** *stable-id2* **apply** *fast*
apply *clarify*
apply(*case-tac V* = *Va*) **prefer** *2* **apply** *simp* **using** *Emptyprecond* **apply** *auto*[*1*]
apply *simp*
apply(*case-tac mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*Va*} = {})
using *Emptyprecond*[*of* - {(*s*, *t*). *s* = *t*} *UNIV*] **apply** *auto*[*1*]
apply(*subgoal-tac mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*Va*} = {*Va*})
prefer *2* **using** *int1-eq*[**where** *P*=*mp-alloc-precond2-1-1-loopinv-0 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄] **apply** *meson*
using *mp-alloc-stm4-lm1*[*of t p timeout sz*] **apply** *auto*[*1*]

apply(*unfold stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-1-loopinv-1-stb* **apply** *simp*
using *mp-alloc-precond2-1-1-loopinv-stb* **apply** *auto*[*1*]
apply *clarify*
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond2-1-1-loopinv-1 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
apply *auto*[*1*]
apply(*subgoal-tac mp-alloc-precond2-1-1-loopinv-1 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {*V*})
prefer *2* **using** *int1-eq*[**where** *P*=*mp-alloc-precond2-1-1-loopinv-1 t p sz timeout* ∩ {|´*cur* = *Some t*|}] **apply** *meson*
apply *simp*
apply(*rule conjI*) **apply**(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*) **apply**(*subgoal-tac* (*V*,*V*(|*from-l* := (*from-l V*)(*t* := *from-l V t* + *1*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*) **apply**(*subgoal-tac* (*V*,*V*(|*from-l* := (*from-l V*)(*t* := *from-l V t* + *1*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** (*metis less-minus-one-simps*(*1*))
apply(*rule conjI*) **apply** *smt*
apply (*metis* (*no-types*, *hide-lams*) *Mem-block.simps*(*2*) *Mem-block.simps*(*3*) *Mem-block.simps*(*4*) *option.sel*)

apply *simp* **using** *stable-id2* **apply** *blast* **using** *stable-id2* **apply** *blast*

apply(*simp add*:*Mem-pool-alloc-guar-def*)
done

21.8 stm5

lemma *mp-alloc-stm5-lm-1-inv-mempool-info*:
free-l V t ≤ *alloc-l V t* ⟹
alloc-l V t < *int* (*n-levels* (*mem-pool-info V p*)) ⟹
p ∈ *mem-pools V* ⟹
inv-mempool-info V ⟹
¬ *free-l V t* < *OK* ⟹
NULL < *blk V t* ⟹
inv-mempool-info

(*V*⦇*mem-pool-info* := (*mem-pool-info V*)
(*p* := *mem-pool-info V p*
⦇*levels* := (*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]⦈]⦈),
allocating-node := (*allocating-node V*)(*t* := *None*)⦈)
apply(*simp add*:*inv-mempool-info-def*)
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply(*rule conjI*) **apply** *metis*
apply *clarify* **apply**(*rename-tac ii*) **apply**(*subgoal-tac length* (*bits* ((*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]⦈] !
ii))=*length* (*bits* (*levels* (*mem-pool-info V p*) ! *ii*)))
prefer *2* **apply**(*case-tac ii* = *nat* (*alloc-l V t*)) **apply** *force* **apply** *force*
apply *metis*
done

lemma *mp-alloc-stm5-lm-1-inv-bitmap-h1*:
allocating-node V t =
Some ⦇*pool* = *p*, *level* = *nat* (*alloc-l V t*), *block* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*), *data* = *blk V t*⦈ ⟹
∀ *t n*. *allocating-node V t* = *Some n* ⟶ *get-bit-s V* (*pool n*) (*level n*) (*block n*) = *ALLOCATING* ⟹
get-bit-s V p (*nat* (*alloc-l V t*)) ((*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)) = *ALLOCATING*
by *fastforce*

lemma *mp-alloc-stm5-lm-1-inv-bitmap-freelist*:
allocating-node V t =
Some ⦇*pool* = *p*, *level* = *nat* (*alloc-l V t*), *block* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*), *data* = *blk V t*⦈ ⟹
alloc-l V t < *int* (*n-levels* (*mem-pool-info V p*)) ⟹
p ∈ *mem-pools V* ⟹
inv-mempool-info V ∧ *inv-aux-vars V* ∧ *inv-bitmap-freelist V* ⟹
inv-bitmap-freelist
(*V*⦇*mem-pool-info* := (*mem-pool-info V*)
(*p* := *mem-pool-info V p*
⦇*levels* := (*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))

[(*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*) := *ALLOCATED*]⦈]⦈),
allocating-node := (*allocating-node* *V*)(*t* := *None*)⦈)
apply(*rule* *subst*[**where** *s*=*inv-bitmap-freelist*
(*V*⦇*mem-pool-info* := (*mem-pool-info* *V*)
(*p* := *mem-pool-info* *V* *p*
⦇*levels* := (*levels* (*mem-pool-info* *V* *p*))
[*nat* (*alloc-l* *V* *t*) := (*levels* (*mem-pool-info* *V* *p*) ! *nat* (*alloc-l* *V* *t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info* *V* *p*) ! *nat* (*alloc-l* *V* *t*)))
[(*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*) := *ALLOCATED*]⦈]⦈)⦈)])
apply(*simp* *add*:*inv-bitmap-freelist-def*)
apply(*rule* *subst*[**where** *s*=*inv-bitmap-freelist* (*set-bit-s* *V* *p* (*nat* (*alloc-l* *V* *t*))
((*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*)) *ALLOCATED*)])
apply(*unfold* *set-bit-s-def* *set-bit-def*)[*1*] **apply** *blast*
apply(*subgoal-tac* *get-bit-s* *V* *p* (*nat* (*alloc-l* *V* *t*))
((*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*)) = *ALLOCATING*) **prefer** *2*
apply(*subgoal-tac* $\forall$ *t* *n*. *allocating-node* *V* *t* = *Some* *n* $\longrightarrow$ *get-bit-s* *V* (*pool* *n*) (*level* *n*) (*block* *n*) = *ALLOCATING*) **prefer** *2*
apply(*simp* *add*:*inv-aux-vars-def* *Let-def*)
using *mp-alloc-stm5-lm-1-inv-bitmap-h1* **apply** *blast*

using *inv-bitmap-freelist-presv-setbit-notfree*[*of* *p* *V* *ALLOCATED* *nat* (*alloc-l* *V* *t*)
(*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*)]
apply *fastforce*
done

lemma *mp-alloc-stm5-lm-1-inv-bitmap*:
allocating-node *V* *t* =
Some ⦇*pool* = *p*, *level* = *nat* (*alloc-l* *V* *t*), *block* = (*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*), *data* = *blk* *V* *t*⦈ $\Longrightarrow$
p $\in$ *mem-pools* *V* $\Longrightarrow$
inv-bitmap *V* $\wedge$ *inv-aux-vars* *V* $\Longrightarrow$
inv-bitmap
(*V*⦇*mem-pool-info* := (*mem-pool-info* *V*)
(*p* := *mem-pool-info* *V* *p*
⦇*levels* := (*levels* (*mem-pool-info* *V* *p*))
[*nat* (*alloc-l* *V* *t*) := (*levels* (*mem-pool-info* *V* *p*) ! *nat* (*alloc-l* *V* *t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info* *V* *p*) ! *nat* (*alloc-l* *V* *t*)))
[(*blk* *V* *t* − *buf* (*mem-pool-info* *V* *p*)) *div* *lsizes* *V* *t* ! *nat* (*alloc-l* *V* *t*) := *ALLOCATED*]⦈]⦈),
allocating-node := (*allocating-node* *V*)(*t* := *None*)⦈)
apply(*rule* *subst*[**where** *s*=*inv-bitmap*
(*V*⦇*mem-pool-info* := (*mem-pool-info* *V*)
(*p* := *mem-pool-info* *V* *p*

(|levels := (levels (mem-pool-info V p))
[nat (alloc-l V t) := (levels (mem-pool-info V p) ! nat (alloc-l V t))
(|bits := (bits (levels (mem-pool-info V p) ! nat (alloc-l V t)))
[(blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t) := ALLOCATED]|)]|)|)|)])
apply(simp add:inv-bitmap-def)

apply(rule subst[**where** s=inv-bitmap (set-bit-s V p (nat (alloc-l V t))
((blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t)) ALLOCATED)])
apply(unfold set-bit-s-def set-bit-def)[1] **apply** blast

apply(subgoal-tac get-bit-s V p (nat (alloc-l V t))
((blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t)) = ALLOCATING) **prefer** 2
apply(subgoal-tac ∀ t n. allocating-node V t = Some n ⟶ get-bit-s V (pool n) (level n) (block n) = ALLOCATING) **prefer** 2
apply(simp add:inv-aux-vars-def Let-def)
using mp-alloc-stm5-lm-1-inv-bitmap-h1 **apply** blast

using inv-bitmap-presv-setbit[of V p nat (alloc-l V t) (blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t)
ALLOCATED set-bit-s V p (nat (alloc-l V t)) ((blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t)) ALLOCATED]
apply blast
done

lemma mp-alloc-stm5-lm-1-inv-aux-vars:
(blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t) < n-max (mem-pool-info V p) ∗ 4 ˆ nat (alloc-l V t) ⟹
blk V t =
buf (mem-pool-info V p) +
(blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t) ∗ (max-sz (mem-pool-info V p) div 4 ˆ nat (alloc-l V t)) ⟹
0 < blk V t ⟹
allocating-node V t =
Some (|pool = p, level = nat (alloc-l V t), block = (blk V t − buf (mem-pool-info V p)) div lsizes V t ! nat (alloc-l V t),
data = blk V t|) ⟹
alloc-l V t < int (n-levels (mem-pool-info V p)) ⟹
p ∈ mem-pools V ⟹
inv-mempool-info V ∧ inv-aux-vars V ⟹
∀ ii<length (lsizes V t). lsizes V t ! ii = ALIGN4 (max-sz (mem-pool-info V p)) div 4 ˆ ii ⟹
inv-aux-vars
(V(|mem-pool-info := (mem-pool-info V)
(p := mem-pool-info V p
(|levels := (levels (mem-pool-info V p))
[nat (alloc-l V t) := (levels (mem-pool-info V p) ! nat (alloc-l V t))

(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]|)]|)),
allocating-node := (*allocating-node V*)(*t* := *None*)|))
apply(*unfold inv-aux-vars-def*)
apply(*subgoal-tac get-bit-s V p* (*nat* (*alloc-l V t*))
((*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)) = *ALLOCATING*) **prefer** *2*
using *mp-alloc-stm5-lm-1-inv-bitmap-h1* **apply** *presburger*

apply(*subgoal-tac mem-block-addr-valid V* ((|*pool* = *p*, *level* = *nat* (*alloc-l V t*),
block = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*), *data* = *blk V t*|))) **prefer** *2*
apply(*simp add*:*mem-block-addr-valid-def*)

apply(*rule conjI*)
apply *clarify*
apply(*subgoal-tac freeing-node V ta* = *Some n*) **prefer** *2* **apply** *force*

apply(*subgoal-tac* ¬(*pool n* = *p* ∧ *level n* = *nat* (*alloc-l V t*)
∧ *block n* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)))
prefer *2* **apply** *metis*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *FREEING*) **prefer** *2*
apply *presburger*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *get-bit-s*
(*V*(|*mem-pool-info* := (*mem-pool-info V*)
(*p* := *mem-pool-info V p*
(|*levels* := (*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]|)]|)),
allocating-node := (*allocating-node V*)(*t* := *None*)|)) (*pool n*) (*level n*) (*block n*)) **prefer** *2*
apply(*case-tac pool n* ≠ *p*) **apply** *force*
apply(*case-tac level n* ≠ *nat* (*alloc-l V t*)) **apply** *force*
apply(*case-tac block n* ≠ (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*))
apply(*case-tac level n* ≥ *length* (*levels* (*mem-pool-info V* (*pool n*)))) **apply** *fastforce*
apply *force* **apply** *blast*
apply *argo*

apply(*rule conjI*)
apply *clarify*
apply(*subgoal-tac* ∃ *ta*. *freeing-node V ta* = *Some n*) **prefer** *2*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *FREEING*) **prefer**

2
apply(*case-tac pool n* $\neq$ *p*) **apply** *force*
apply(*case-tac level n* $\neq$ *nat* (*alloc-l V t*)) **apply** *force*
apply(*case-tac block n* $\neq$ (*blk V t* $-$ *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*))
apply(*case-tac level n* $\geq$ *length* (*levels* (*mem-pool-info V* (*pool n*))))
apply *fastforce* **apply** *force*
apply(*case-tac level n* $\geq$ *length* (*levels* (*mem-pool-info V* (*pool n*))))
apply *fastforce*
apply(*case-tac block n* $\geq$ *length* (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))))
apply *fastforce* **apply** *fastforce*
apply(*subgoal-tac mem-block-addr-valid V n*) **prefer** *2*
apply(*simp add*:*mem-block-addr-valid-def*)
apply *blast*
apply *force*

apply(*rule conjI*)
apply *clarify*
apply(*subgoal-tac t* $\neq$ *ta*) **prefer** *2* **apply** *fastforce*
apply(*subgoal-tac allocating-node V ta* = *Some n*) **prefer** *2* **apply** *force*

apply(*subgoal-tac* $\neg$(*pool n* = *p* $\wedge$ *level n* = *nat* (*alloc-l V t*)
$\wedge$ *block n* = (*blk V t* $-$ *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)))
prefer *2* **apply** (*metis Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*))
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *ALLOCATING*) **prefer** *2* **apply** *presburger*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *get-bit-s*
(*V*⦇*mem-pool-info* := (*mem-pool-info V*)
(*p* := *mem-pool-info V p*
⦇*levels* := (*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
⦇*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* $-$ *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]⦈]⦈),
allocating-node := (*allocating-node V*)(*t* := *None*)⦈) (*pool n*) (*level n*) (*block n*)) **prefer** *2*
apply(*case-tac pool n* $\neq$ *p*) **apply** *force*
apply(*case-tac level n* $\neq$ *nat* (*alloc-l V t*)) **apply** *force*
apply(*case-tac block n* $\neq$ (*blk V t* $-$ *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*))
apply(*case-tac level n* $\geq$ *length* (*levels* (*mem-pool-info V* (*pool n*)))) **apply** *fastforce*
apply *force* **apply** *blast*
apply *argo*

apply(*rule conjI*)

apply *clarify*
apply(*subgoal-tac nat* (*alloc-l V t*) < *length* (*levels* (*mem-pool-info V p*))) **prefer** *2*
apply(*simp add*:*inv-mempool-info-def Let-def*)
apply (*metis int-nat-eq of-nat-0-less-iff of-nat-less-imp-less*)
apply(*subgoal-tac* (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)
< *length* (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))))
prefer *2*
apply(*simp add*:*inv-mempool-info-def Let-def*)
apply(*subgoal-tac* ¬(*pool n* = *p* ∧ *level n* = *nat* (*alloc-l V t*)
∧ *block n* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)))
prefer *2*
apply(*case-tac pool n* ≠ *p*) **apply** *fastforce*
apply(*case-tac level n* ≠ *nat* (*alloc-l V t*)) **apply** *fastforce*
apply(*case-tac block n* ≠ (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)) **apply** *fastforce*
apply *simp*

apply(*subgoal-tac* ∃ *ta*. *ta* ≠ *t* ∧ *allocating-node V ta* = *Some n*) **prefer** *2*
apply(*subgoal-tac get-bit-s V* (*pool n*) (*level n*) (*block n*) = *ALLOCATING*) **prefer** *2*
apply(*case-tac pool n* ≠ *p*) **apply** *force*
apply(*case-tac level n* ≠ *nat* (*alloc-l V t*)) **apply** *force*
apply(*case-tac block n* ≠ (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*))
apply(*case-tac level n* ≥ *length* (*levels* (*mem-pool-info V* (*pool n*))))
apply *fastforce* **apply** *force*
apply(*case-tac level n* ≥ *length* (*levels* (*mem-pool-info V* (*pool n*))))
apply *fastforce*
apply(*case-tac block n* ≥ *length* (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))))
apply *fastforce* **apply** *fastforce*
apply(*subgoal-tac mem-block-addr-valid V n*) **prefer** *2*
apply(*simp add*:*mem-block-addr-valid-def*)
apply (*metis Mem-block.select-convs*(*1*) *Mem-block.select-convs*(*2*) *Mem-block.select-convs*(*3*) *option.sel*)

apply *auto*[*1*]

apply(*rule conjI*)
apply *clarify*
apply *auto*[*1*]

apply(*rule conjI*)
apply *clarify*
apply(*subgoal-tac allocating-node V t1* = *Some n1*) **prefer** *2*
apply(*case-tac t* = *t1*) **apply** *force* **apply** *force*

apply(*subgoal-tac allocating-node V t2 = Some n1*) **prefer** *2*
  apply(*case-tac t = t2*) **apply** *force* **apply** *force*
apply *metis*

apply *clarify*
apply(*subgoal-tac allocating-node V t1 = Some n1*) **prefer** *2*
  apply(*case-tac t = t1*) **apply** *force* **apply** *force*
apply(*subgoal-tac freeing-node V t2 = Some n1*) **prefer** *2* **apply** *force*
apply *metis*
done

lemma *mp-alloc-stm5-lm-1-inv-bitmap0*:
p ∈ *mem-pools V* ⟹
    inv-mempool-info V ∧ *inv-bitmap0 V* ⟹
    inv-bitmap0
     (*V*(|*mem-pool-info* := (*mem-pool-info V*)
         (*p* := *mem-pool-info V p*
           (|*levels* := (*levels* (*mem-pool-info V p*))
              [*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
                 (|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
                   [(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l*
V t) := *ALLOCATED*]|)]|)),
         allocating-node := (*allocating-node V*)(*t* := *None*)|))
apply(*simp add*:*inv-bitmap0-def Let-def*)
apply *clarsimp*
apply(*subgoal-tac length* (*levels* (*mem-pool-info V p*)) > *0*) **prefer** *2*
  apply(*simp add*:*inv-mempool-info-def Let-def*) **apply** *fastforce*

apply(*case-tac nat* (*alloc-l V t*) = *0*)
  apply(*case-tac i* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat*
(*alloc-l V t*))
    apply *fastforce* **apply** *force*
by *fastforce*

lemma *mp-alloc-stm5-lm-1-inv-bitmapn*:
p ∈ *mem-pools V* ⟹
  inv-mempool-info V ∧ *inv-bitmapn V* ⟹
  inv-bitmapn
   (*V*(|*mem-pool-info* := (*mem-pool-info V*)
       (*p* := *mem-pool-info V p*
         (|*levels* := (*levels* (*mem-pool-info V p*))
            [*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
               (|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
                 [(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l*
V t) := *ALLOCATED*]|)]|)),
       allocating-node := (*allocating-node V*)(*t* := *None*)|))
apply(*simp add*:*inv-bitmapn-def Let-def*)
apply *clarsimp*

apply(*subgoal-tac length* (*levels* (*mem-pool-info V p*)) > *0*) **prefer** *2*
apply(*simp add*:*inv-mempool-info-def Let-def*) **apply** *fastforce*

apply(*case-tac nat* (*alloc-l V t*) = *length* (*levels* (*mem-pool-info V p*)) − *Suc 0*)
apply(*case-tac i* = (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*))
apply *fastforce* **apply** *force*
by *fastforce*

lemma *mp-alloc-stm5-lm-1-inv-bitmap-not4free*:
(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) < *n-max* (*mem-pool-info V p*) ∗ *4* ˆ *nat* (*alloc-l V t*) =⇒
alloc-l V t < *int* (*n-levels* (*mem-pool-info V p*)) =⇒
p ∈ *mem-pools V* =⇒
inv-mempool-info V ∧ *inv-bitmap-not4free V* =⇒
inv-bitmap-not4free
(*V*(|*mem-pool-info* := (*mem-pool-info V*)
(*p* := *mem-pool-info V p*
(|*levels* := (*levels* (*mem-pool-info V p*))
[*nat* (*alloc-l V t*) := (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))
(|*bits* := (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*)))
[(*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) := *ALLOCATED*]|)]|)),
allocating-node := (*allocating-node V*)(*t* := *None*)|))
apply(*subgoal-tac length* (*levels* (*mem-pool-info V p*)) > *0*) **prefer** *2*
apply(*simp add*:*inv-mempool-info-def Let-def*) **apply** *fastforce*
apply(*subgoal-tac nat* (*alloc-l V t*) < *length* (*levels* (*mem-pool-info V p*))) **prefer** *2*
apply(*simp add*:*inv-mempool-info-def Let-def*)
apply (*metis int-nat-eq of-nat-0-less-iff of-nat-less-imp-less*)

apply(*simp add*:*inv-bitmap-not4free-def partner-bits-def Let-def*)
apply *clarsimp*

apply(*subgoal-tac* (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*)
< *length* (*bits* (*levels* (*mem-pool-info V p*) ! *nat* (*alloc-l V t*))))
prefer *2*
apply(*simp add*:*inv-mempool-info-def Let-def*)

apply(*case-tac nat* (*alloc-l V t*) < *length* (*levels* (*mem-pool-info V p*)))
apply(*case-tac i* = *nat* (*alloc-l V t*))
apply(*case-tac* (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) = *j div 4* ∗ *4*)
apply *fastforce*
apply(*case-tac* (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l V t*) = *Suc* (*j div 4* ∗ *4*))
apply *fastforce*
apply(*case-tac* (*blk V t* − *buf* (*mem-pool-info V p*)) *div lsizes V t* ! *nat* (*alloc-l*

$V\ t) = Suc\ (Suc\ (j\ div\ 4 * 4)))$
  apply *fastforce*
  apply(*case-tac* $(blk\ V\ t - buf\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ lsizes\ V\ t\ !\ nat\ (alloc\text{-}l\ V\ t) = j\ div\ 4 * 4 + 3$)
  apply *fastforce*
  apply *fastforce*
 apply *force*
by *blast*

lemma *mp-alloc-stm5-lm-1-inv*:
$(blk\ V\ t - buf\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ lsizes\ V\ t\ !\ nat\ (alloc\text{-}l\ V\ t) < n\text{-}max\ (mem\text{-}pool\text{-}info\ V\ p) * 4\ \hat{}\ nat\ (alloc\text{-}l\ V\ t) \Longrightarrow$
$blk\ V\ t = buf\ (mem\text{-}pool\text{-}info\ V\ p) + (blk\ V\ t - buf\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ lsizes\ V\ t\ !\ nat\ (alloc\text{-}l\ V\ t) *$
$(max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ p)\ div\ 4\ \hat{}\ nat\ (alloc\text{-}l\ V\ t)) \Longrightarrow$
$allocating\text{-}node\ V\ t =$
$Some\ (\!|pool = p,\ level = nat\ (alloc\text{-}l\ V\ t),\ block = (blk\ V\ t - buf\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ lsizes\ V\ t\ !\ nat\ (alloc\text{-}l\ V\ t),$
$data = blk\ V\ t|\!) \Longrightarrow$
$free\text{-}l\ V\ t \le alloc\text{-}l\ V\ t \Longrightarrow$
$alloc\text{-}l\ V\ t < int\ (n\text{-}levels\ (mem\text{-}pool\text{-}info\ V\ p)) \Longrightarrow$
$length\ (lsizes\ V\ t) \le n\text{-}levels\ (mem\text{-}pool\text{-}info\ V\ p) \Longrightarrow$
$p \in mem\text{-}pools\ V \Longrightarrow$
$inv\ V \Longrightarrow$
$\forall ii{<}length\ (lsizes\ V\ t).\ lsizes\ V\ t\ !\ ii = ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ 4\ \hat{}\ ii \Longrightarrow$
$\neg\ free\text{-}l\ V\ t < OK \Longrightarrow$
$NULL < blk\ V\ t \Longrightarrow$
$\forall ii{\le}nat\ (alloc\text{-}l\ V\ t).\ sz \le lsizes\ V\ t\ !\ ii \Longrightarrow$
$alloc\text{-}l\ V\ t = int\ (length\ (lsizes\ V\ t)) - 1 \wedge length\ (lsizes\ V\ t) = n\text{-}levels\ (mem\text{-}pool\text{-}info\ V\ p) \vee$
$alloc\text{-}l\ V\ t = int\ (length\ (lsizes\ V\ t)) - 2 \wedge lsizes\ V\ t\ !\ nat\ (int\ (length\ (lsizes\ V\ t)) - 1) < sz \Longrightarrow$
$inv\ (V(\!|mem\text{-}pool\text{-}info := (mem\text{-}pool\text{-}info\ V)$
$(p := mem\text{-}pool\text{-}info\ V\ p$
$(\!|levels := (levels\ (mem\text{-}pool\text{-}info\ V\ p))$
$[nat\ (alloc\text{-}l\ V\ t) := (levels\ (mem\text{-}pool\text{-}info\ V\ p)\ !\ nat\ (alloc\text{-}l\ V\ t))$
$(\!|bits := (bits\ (levels\ (mem\text{-}pool\text{-}info\ V\ p)\ !\ nat\ (alloc\text{-}l\ V\ t)))$
$[(blk\ V\ t - buf\ (mem\text{-}pool\text{-}info\ V\ p))\ div\ lsizes\ V\ t\ !\ nat\ (alloc\text{-}l\ V\ t) := ALLOCATED]|\!)]|\!)),$
$allocating\text{-}node := (allocating\text{-}node\ V)(t := None)|\!))$
 apply(*simp add*:*inv-def*)
 apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*)
 apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*)
  apply(*rule conjI*) **apply** *metis* **apply** *metis*
 apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-mempool-info* **apply** *blast*
 apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-bitmap-freelist* **apply** *blast*
 apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-bitmap* **apply** *blast*
 apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-aux-vars* **apply** *blast*

  apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-bitmap0* **apply** *blast*
  apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv-bitmapn* **apply** *blast*
          using *mp-alloc-stm5-lm-1-inv-bitmap-not4free* **apply** *blast*
done

term *mp-alloc-precond2-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|}

lemma *mp-alloc-stm5-lm-1*:
  mp-alloc-precond2-1-2 t p sz timeout ∩ {|´*cur* = *Some t*|} ∩ {*V*} ≠ {} ⟹
   Γ $\vdash_I$ *Some* (´*mem-pool-info* :=
      set-bit-alloc ´*mem-pool-info p* (*nat* (´*alloc-l t*)) (*block-num* (´*mem-pool-info p*) (´*blk t*) (´*lsizes t* ! *nat* (´*alloc-l t*)));;
      ´*allocating-node* := ´*allocating-node* (*t* := *None*))
   sat$_p$ [*mp-alloc-precond2-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*},
     {(*s*, *t*). *s* = *t*}, *UNIV*,{|´(*Pair V*) ∈ *Mem-pool-alloc-guar t*|} ∩ *mp-alloc-precond2-1-3 t p sz timeout*]
  apply(*subgoal-tac mp-alloc-precond2-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*}={*V*})
  prefer *2* **using** *int1-eq*[**where** *P*=*mp-alloc-precond2-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|}] **apply** *meson*

  apply *simp*

  apply(*rule Seq*[**where** *mid*={*V*(|*mem-pool-info* := *set-bit-alloc* (*mem-pool-info V*) *p* (*nat* (*alloc-l V t*))
      (*block-num* ((*mem-pool-info V*) *p*) (*blk V t*) (*lsizes V t* ! *nat* (*alloc-l V t*)))|)}])

  apply(*rule Basic*)
   apply *simp* **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

  apply(*rule Basic*)
   apply *clarsimp* **apply**(*simp add*: *set-bit-def block-num-def*)
   apply(*rule conjI*)
    apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
    apply(*rule conjI*)
     apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*) **apply** *clarsimp*
      apply(*case-tac i* = *nat* (*alloc-l V t*)) **apply**(*case-tac i* < *length* (*levels* (*mem-pool-info V p*)))
      apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
    apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv* **apply** *clarsimp*
     apply(*simp add*:*lvars-nochange-def*)
   apply(*rule conjI*) **using** *mp-alloc-stm5-lm-1-inv* **apply** *clarsimp*
    apply(*case-tac alloc-l V t* = *int* (*length* (*lsizes V t*)) − *1* ∧ *length* (*lsizes V t*) = *n-levels* (*mem-pool-info V p*))
     apply *simp* **apply** *simp*

   apply *simp* **apply**(*simp add*:*stable-def*) **using** *stable-id2* **apply** *metis*
done

lemma *mp-alloc-stm5-lm*:
$\Gamma \vdash_I$ *Some* (t ► ´*mem-pool-info* := *set-bit-alloc* ´*mem-pool-info p* (*nat* (´*alloc-l* t))
(*block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*alloc-l t*))));;
´*allocating-node* := ´*allocating-node* (t := *None*)
) sat_p [*mp-alloc-precond2-1-2 t p sz timeout*, *Mem-pool-alloc-rely t*, *Mem-pool-alloc-guar t*,
mp-alloc-precond2-1-3 t p sz timeout]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-2-stb* **apply** *auto*[*1*]
using *mp-alloc-precond2-1-3-stb* **apply** *auto*[*1*]

apply *clarify*
apply(*case-tac mp-alloc-precond2-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
apply *simp* **using** *Emptyprecond* **apply** *metis*
using *mp-alloc-stm5-lm-1*[*of t p timeout sz*] **apply** *clarsimp*
done

term *mp-alloc-precond2-1-2 t p sz timeout*
term *mp-alloc-precond2-1-3 t p sz timeout*

21.9 stm6

lemma *mp-alloc-stm6-lm*:
$\Gamma \vdash_I$ *Some* (t ► ´*mempoolalloc-ret* := ´*mempoolalloc-ret* (t :=
Some (|*pool* = p, *level* = *nat* (´*alloc-l t*),
block = *block-num* (´*mem-pool-info p*) (´*blk t*) ((´*lsizes t*)!(*nat* (´*alloc-l* t))),
data = ´*blk t* |)))
sat_p [*mp-alloc-precond2-1-3 t p sz timeout*, *Mem-pool-alloc-rely t*, *Mem-pool-alloc-guar t*,
mp-alloc-precond2-1-4 t p sz timeout]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-3-stb* **apply** *simp*
using *mp-alloc-precond2-1-4-stb* **apply** *simp*
apply *clarify*
apply(*rule Basic*)
apply *clarsimp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*,*V*(|*mempoolalloc-ret* := *mempoolalloc-ret V*(t ↦

(|*pool* = *p*, *level* = *nat* (*alloc-l V t*), *block* = *block-num* (*mem-pool-info V p*) (*blk V t*) (*lsizes V t* ! *nat* (*alloc-l V t*)),
data = *blk V t*|))|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*,*V*(|*mempoolalloc-ret* := *mempoolalloc-ret V*(*t* ↦
(|*pool* = *p*, *level* = *nat* (*alloc-l V t*), *block* = *block-num* (*mem-pool-info V p*) (*blk V t*) (*lsizes V t* ! *nat* (*alloc-l V t*)),
data = *blk V t*|))|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*rule conjI*)
apply (*smt int-nat-eq inv-maxsz-align4 less-imp-le-nat not-less of-nat-less-iff*)
apply *clarify*
apply(*subgoal-tac* ¬(*alloc-l V t* = *int* (*length* (*lsizes V t*)) − *1* ∧ *length* (*lsizes V t*) = *n-levels* (*mem-pool-info V p*)))
prefer *2* **apply** *auto*[*1*]
apply *simp* **apply** (*smt Suc-nat-eq-nat-zadd1 inv-maxsz-align4 lessI nat-int power-Suc*)

apply *simp* **using** *stable-id2* **apply** *metis* **using** *stable-id2* **apply** *metis*
done

21.10 stm7

abbreviation *mp-alloc-stm7-precond1 Va* ≡ *Va*(|*thd-state* := (*thd-state Va*)(*the* (*cur Va*) := *BLOCKED*)|)
abbreviation *mp-alloc-stm7-precond3 Va t p* ≡
Va(|*mem-pool-info* := (*mem-pool-info Va*)(*p* := (*mem-pool-info Va p*)(|*wait-q* := (*wait-q* (*mem-pool-info Va p*))@ [*the* (*cur Va*)]|))|)

lemma *mp-alloc-stm7-lm-2-1*: (λ*a*. *if a* = *p then mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|)
else mem-pool-info (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|)) *a*) *x*
= (λ*a*. *if a* = *p then mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|)
else mem-pool-info Va a) *x*
apply(*case-tac x* = *p*)
apply *auto*
done

lemma *mp-alloc-stm7-lm-2-2*:
cur Va = *Some t* ⟹
(λ*a*. *if a* = *p*

then mem-pool-info Va p(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*the* (*cur* (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|)))]|)
else mem-pool-info (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|) *a*)
=
(λ*a*. *if a* = *p then mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|) *else mem-pool-info Va a*)
using *mp-alloc-stm7-lm-2-1* **by** *auto*

lemma *mp-alloc-stm7-lm-2*:
cur Va = *Some t* $\Longrightarrow$
(λ*a*. *if a* = *p then*
mem-pool-info (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|) *p*
(|*wait-q* := *wait-q* (*mem-pool-info* (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|) *p*)
@ [*the* (*cur* (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|)))]|)
else mem-pool-info (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|) *a*)
=
(*mem-pool-info Va*)(*p* := *mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|))
apply(*rule subst*[**where** *t*=*mem-pool-info* (*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*)|) *p*
and *s*=*mem-pool-info Va p*]) **apply** *simp*
apply(*simp add*:*fun-upd-def*)
using *mp-alloc-stm7-lm-2-2* **apply** *auto*
done

lemma *mp-alloc-stm7-swap-ifbody-inv*:
p $\in$ *mem-pools Va* $\Longrightarrow$
inv Va $\Longrightarrow$
cur Va = *Some t* $\Longrightarrow$
(*if ta* = *t then BLOCKED else thd-state Va ta*) = *READY* $\Longrightarrow$
inv (*mp-alloc-stm7-precond3 Va t p*
(|*cur* := *Some* (*SOME ta*. *ta* $\neq$ *t* $\wedge$ (*ta* $\neq$ *t* $\longrightarrow$ *thd-state Va ta* = *READY*)),
thd-state :=
λ*x*. *if x* = (*SOME ta*. *ta* $\neq$ *t* $\wedge$ (*ta* $\neq$ *t* $\longrightarrow$ *thd-state Va ta* = *READY*)) *then RUNNING*
else thd-state
(*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|)),
cur := *Some* (*SOME ta*. (*ta* = *t* $\longrightarrow$ *BLOCKED* = *READY*) $\wedge$ (*ta* $\neq$ *t* $\longrightarrow$ *thd-state Va ta* = *READY*))|))
x|))
apply(*subgoal-tac thd-state Va t* = *RUNNING*)
prefer *2* **apply**(*simp add*:*inv-def inv-cur-def*) **apply** *auto*[*1*]
apply(*subgoal-tac ta* $\neq$ *t* $\wedge$ *thd-state Va ta* = *READY*)

prefer *2* **apply** *auto*[*1*] **using** *Thread-State-Type.distinct*(*3*) **apply** *presburger*
apply(*subgoal-tac* (*SOME ta. ta* ≠ *t* ∧ (*ta* ≠ *t* ⟶ *thd-state Va ta* = *READY*)) ≠ *t*)
prefer *2* **using** *exE-some*[**where** *P*=λ*tb. tb* ≠ *t* ∧ (*tb* ≠ *t* ⟶ *thd-state Va tb* = *READY*)
and *c*=*SOME tb. tb* ≠ *t* ∧ (*tb* ≠ *t* ⟶ *thd-state Va tb* = *READY*)]
apply *auto*[*1*]
apply(*subgoal-tac thd-state Va* (*SOME ta. ta* ≠ *t* ∧ (*ta* ≠ *t* ⟶ *thd-state Va ta* = *READY*)) = *READY*)
prefer *2* **using** *exE-some*[**where** *P*=λ*tb. tb* ≠ *t* ∧ (*tb* ≠ *t* ⟶ *thd-state Va tb* = *READY*)
and *c*=*SOME tb. tb* ≠ *t* ∧ (*tb* ≠ *t* ⟶ *thd-state Va tb* = *READY*)]
apply *auto*[*1*]

apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*)
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** (*metis* (*no-types*, *lifting*) *Thread-State-Type.distinct*(*5*) *diff-is-0-eq′*
less-Suc-eq less-Suc-eq-le nth-Cons-0 nth-append nth-mem)
apply *auto*[*1*]
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*) **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply**(*simp add*: *inv-aux-vars-def mem-block-addr-valid-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def partner-bits-def*) **apply** *meson*
done

lemma *mp-alloc-stm7-swap-elsebody-inv*:
p ∈ *mem-pools Va* ⟹
inv Va ⟹
cur Va = *Some t* ⟹
(*if ta* = *t then BLOCKED else thd-state Va ta*) ≠ *READY* ⟹
inv (*cur-update Map.empty*
(*Va*⦇*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*),
mem-pool-info := (*mem-pool-info Va*)(*p* := *mem-pool-info Va p*⦇*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]⦈)⦈))
apply(*subgoal-tac thd-state Va t* = *RUNNING*)
prefer *2* **apply**(*simp add*:*inv-def inv-cur-def*) **apply** *auto*[*1*]

apply(*simp add*:*inv-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-cur-def*) **apply** *auto*[*1*]

apply(*rule conjI*) **apply**(*simp add*:*inv-thd-waitq-def*)
apply(*rule conjI*) **apply** *auto*[*1*]
apply(*rule conjI*) **apply** (*metis Thread-State-Type*.*distinct*(*6*) *diff-is-0-eq′ less-Suc-eq*
less-Suc-eq-le nth-Cons-0 nth-append nth-mem)
apply (*metis* (*no-types*, *lifting*) *Thread-State-Type*.*distinct*(*5*))
apply(*rule conjI*) **apply**(*simp add*:*inv-mempool-info-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-freelist-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap-def*) **apply**(*simp add*:*Let-def*)
apply(*rule conjI*) **apply**(*simp add*: *inv-aux-vars-def mem-block-addr-valid-def*) **apply** *meson*
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmap0-def*)
apply(*rule conjI*) **apply**(*simp add*:*inv-bitmapn-def*)
apply(*simp add*:*inv-bitmap-not4free-def partner-bits-def*) **apply** *meson*

done

lemma *mp-alloc-stm7-lm-1*:
mp-alloc-precond1-8-2-2 t p sz timeout $\cap$ ⦃´*cur* = *Some t*⦄ $\cap$ $\{V\}$ $\cap$ *UNIV* $\cap$ $\{Va\} \neq \{\} \Longrightarrow$
$\Gamma \vdash_I$ *Some* (´*thd-state* := ´*thd-state*(*the* ´*cur* := *BLOCKED*);;
´*mem-pool-info* := ´*mem-pool-info*(*p* := ´*mem-pool-info p*⦇*wait-q* := *wait-q* (´*mem-pool-info p*) @ [*the* ´*cur*] ⦈);;
swap)
sat_p [*mp-alloc-precond1-8-2-2 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\cap$ $\{V\}$ $\cap$ *UNIV* $\cap$ $\{Va\}$,
$\{(s, t).\ s = t\}$, *UNIV*, ⦃´(*Pair Va*) $\in$ *UNIV*⦄ $\cap$ (⦃´(*Pair V*) $\in$ *Mem-pool-alloc-guar t*⦄ $\cap$
(*mp-alloc-precond1-8-2-2 t p sz timeout*))]
apply(*subgoal-tac V* = *Va*)
prefer *2* **apply** *simp*
apply(*subgoal-tac mp-alloc-precond1-8-2-2 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\cap$ $\{V\}$ $\cap$ *UNIV* $\cap$ $\{Va\} = \{Va\}$)
prefer *2* **apply** *auto*[*1*]
apply(*rule subst*[**where** *t*=*mp-alloc-precond1-8-2-2 t p sz timeout* $\cap$ ⦃´*cur* = *Some t*⦄ $\cap$ $\{V\}$ $\cap$ *UNIV* $\cap$ $\{Va\}$ **and** *s*=$\{V\}$])
apply *simp*
apply *clarsimp*

apply(*rule Seq*[**where** *mid*={*mp-alloc-stm7-precond3* (*mp-alloc-stm7-precond1 Va*) *t p*}])
apply(*rule Seq*[**where** *mid*={*mp-alloc-stm7-precond1 Va*}])

apply(*rule Basic*)
apply(*simp add*:*fun-upd-def*)
apply *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)

apply *simp* **using** *mp-alloc-stm7-lm-2*[*of Va t p*] **apply** *metis*
apply *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*simp add*:*swap-def*)
apply(*rule Cond*)
apply(*simp add*:*stable-def*)

apply(*case-tac* {*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|))|)} ∩
{|∃ *t*. ´*thd-state t* = *READY*|} = {})
apply *simp* **using** *Emptyprecond* **apply** *metis*
apply(*rule subst*[**where** *t*={*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|))|)} ∩
{|∃ *t*. ´*thd-state t* = *READY*|} **and** *s*={*Va*(|*thd-state* := (*thd-state Va*)(*t* := *BLOCKED*),
mem-pool-info := (*mem-pool-info Va*)
(*p* := *mem-pool-info Va p*(|*wait-q* := *wait-q* (*mem-pool-info Va p*) @ [*t*]|))|)}])
apply *simp*
apply(*rule Seq*[**where** *mid*={*let V* = *mp-alloc-stm7-precond3* (*mp-alloc-stm7-precond1 Va*) *t p in*
V(|*cur* := *Some* (*SOME t*. (*thd-state V*) *t* = *READY*)|)}])

apply(*rule Basic*)
apply *auto*[*1*] **apply** *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
using *mp-alloc-stm7-swap-ifbody-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*)
using *mp-alloc-stm7-swap-ifbody-inv* **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
using *mp-alloc-stm7-swap-ifbody-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*)
using *mp-alloc-stm7-swap-ifbody-inv* **apply** *auto*[*1*]

apply *simp* **apply**(*simp add*:*stable-def*) **using** *stable-id2* **apply** *metis*

apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
using *mp-alloc-stm7-swap-elsebody-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*)
using *mp-alloc-stm7-swap-elsebody-inv* **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*rule disjI1*)
apply(*rule conjI*)
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
using *mp-alloc-stm7-swap-elsebody-inv* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange-def*)
using *mp-alloc-stm7-swap-elsebody-inv* **apply** *auto*[*1*]

apply *simp* **apply**(*simp add*:*stable-def*) **using** *stable-id2* **apply** *metis*
apply *simp*

done

lemma *mp-alloc-stm7-lm*:
$\Gamma \vdash_I$ *Some* (*t* ► *ATOMIC*
´*thd-state* := ´*thd-state*(*the* ´*cur* := *BLOCKED*);;
´*mem-pool-info* := ´*mem-pool-info*(*p* := ´*mem-pool-info p*⦇*wait-q* := *wait-q*
(´*mem-pool-info p*) @ [*the* ´*cur*] ⦈);;
swap
END) sat_p [*mp-alloc-precond1-8-2-2 t p sz timeout*, *Mem-pool-alloc-rely t*,
Mem-pool-alloc-guar t,
mp-alloc-precond1-8-2-2 t p sz timeout]
apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-8-2-2-stb* **apply** *simp*
using *mp-alloc-precond1-8-2-2-stb* **apply** *simp*
apply *clarify*
apply(*rule Await*)
using *stable-id2* **apply** *metis*
using *stable-id2* **apply** *metis*
apply *clarify*
apply(*case-tac mp-alloc-precond1-8-2-2 t p sz timeout* ∩

$\{\!|\,\acute{}cur = Some\ t|\!\} \cap \{V\} \cap UNIV \cap \{Va\} = \{\})$
using *Emptyprecond* **apply** *metis*
using *mp-alloc-stm7-lm-1* **apply** *meson*
done

term *mp-alloc-precond1-8-2-2 t p sz timeout*

21.11 final proof

lemma *mp-alloc-stm8-guar*:
cur V = Some t $\Longrightarrow$ *inv V* $\Longrightarrow$ *V*(|*rf* := (*rf V*)(*t* := *True*)|) ∈ {|´(*Pair V*) ∈ *Mem-pool-alloc-guar t*|}
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*(|*rf* := (*rf V*)(*t* := *True*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *mp-alloc-stm9-guar*:
cur V = Some t $\Longrightarrow$ *inv V* $\Longrightarrow$ *V*(|*ret* := (*ret V*)(*t* := *ETIMEOUT*)|) ∈ {|´(*Pair V*) ∈ *Mem-pool-alloc-guar t*|}
apply *auto* **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*(|*ret* := (*ret V*)(*t* := *ETIMEOUT*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
done

lemma *Mempool-alloc-satRG*: *Evt-sat-RG* Γ (*Mem-pool-alloc-RGCond t p sz timeout*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def Evt-sat-RG-def*)
apply(*unfold Mem-pool-alloc-def*)
apply(*rule Evt-Basic*)
apply(*unfold body-def guard-def snd-conv fst-conv*)

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond7 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond6 t p timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond5 t p timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond4 t p timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond3 t p timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond2 t p timeout*])

apply(*simp add*:*stm-def*)
apply(*rule Await*)

```
    using mp-alloc-precond1-stb apply auto[1]
    using mp-alloc-precond2-stb apply simp
    apply(rule allI)
      apply(rule Basic)
      apply(case-tac mp-alloc-precond1 t p timeout ∩ {|´cur = Some t|} ∩ { V } =
{})
        apply auto[1] apply simp
        apply(rule conjI)
          apply(simp add:Mem-pool-alloc-guar-def) apply(rule disjI1)
          apply(rule conjI) apply(simp add:gvars-conf-stable-def gvars-conf-def)
          apply(rule conjI)
         apply(subgoal-tac (V,V(|tmout := (tmout V)(t := timeout)|))∈lvars-nochange1-4all)
         using glnochange-inv0 apply auto[1] apply(simp add:lvars-nochange1-4all-def
lvars-nochange1-def)
           apply(simp add:lvars-nochange-def)
       apply(subgoal-tac (V,V(|tmout := (tmout V)(t := timeout)|))∈lvars-nochange1-4all)
         using glnochange-inv0 apply auto[1] apply(simp add:lvars-nochange1-4all-def
lvars-nochange1-def)
        apply(simp add:stable-def)+


  apply(simp add:stm-def)
  apply(rule Await)
    using mp-alloc-precond2-stb apply simp
    using mp-alloc-precond3-stb apply simp
    apply(rule allI)
      apply(rule Basic)
      apply(case-tac mp-alloc-precond2 t p timeout ∩ {|´cur = Some t|} ∩ { V } =
{})
        apply auto[1] apply simp
        apply(rule conjI)
          apply(simp add:Mem-pool-alloc-guar-def) apply(rule disjI1)
          apply(rule conjI) apply(simp add:gvars-conf-stable-def gvars-conf-def)
          apply(rule conjI)
         apply(subgoal-tac (V,V(|endt := (endt V)(t := NULL)|))∈lvars-nochange1-4all)
         using glnochange-inv0 apply auto[1] apply(simp add:lvars-nochange1-4all-def
lvars-nochange1-def)
           apply(simp add:lvars-nochange-def)
       apply(subgoal-tac (V,V(|endt := (endt V)(t := NULL)|))∈lvars-nochange1-4all)
         using glnochange-inv0 apply auto[1] apply(simp add:lvars-nochange1-4all-def
lvars-nochange1-def)
        apply simp apply(simp add:stable-def) apply(simp add:stable-def)


  apply(unfold stm-def)[1]
  apply(rule Await)
  using mp-alloc-precond3-stb apply simp
  using mp-alloc-precond4-stb apply simp
  apply clarify
```

apply(*rule Cond*)
apply(*simp add*:*stable-def*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond3 t p timeout* $\cap$ {´*cur* = *Some t*} $\cap$ {*V*} =
{})
apply *auto*[*1*] **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *auto*[*1*]
apply(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*,*V*(|*endt* := (*endt V*)(*t* := *tick V* + *nat* (*tmout V*
t))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def*
lvars-nochange1-def)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* (*V*,*V*(|*endt* := (*endt V*)(*t* := *tick V* + *nat* (*tmout V*
t))|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def*
lvars-nochange1-def)
apply *simp* **apply**(*simp add*:*stable-def*) **apply**(*simp add*:*stable-def*)

apply(*unfold Skip-def*)[*1*]
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond3 t p timeout* $\cap$ {´*cur* = *Some t*} $\cap$ {*V*} $\cap$
− {*OK* < *timeout*} = {})
apply *auto*[*1*] **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*)+
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond4-stb* **apply** *simp*
using *mp-alloc-precond5-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond2 t p timeout* $\cap$ {´*cur* = *Some t*} $\cap$ {*V*} =
{})
apply *auto*[*1*] **apply** *simp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* (*V*,*V*(|*mempoolalloc-ret* := (*mempoolalloc-ret V*)(*t* :=
None)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def*
lvars-nochange1-def)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* (*V*,*V*(|*mempoolalloc-ret* := (*mempoolalloc-ret V*)(*t* :=
None)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def*

lvars-nochange1-def)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond5-stb* **apply** *simp*
using *mp-alloc-precond6-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond5 t p timeout* $\cap$ $\{\!|$´*cur* = *Some t*$|\!\}$ $\cap$ $\{V\}$ = $\{\}$)
apply *auto*[*1*] **apply** *simp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V,V(\!|$*ret* := (*ret V*)(*t* := *ESIZEERR*)$|\!))\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* $(V,V(\!|$*ret* := (*ret V*)(*t* := *ESIZEERR*)$|\!))\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond6-stb* **apply** *simp*
using *mp-alloc-precond7-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond6 t p timeout* $\cap$ $\{\!|$´*cur* = *Some t*$|\!\}$ $\cap$ $\{V\}$ = $\{\}$)
apply *auto*[*1*] **apply** *simp*
apply(*rule conjI*)
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*rule conjI*) **apply**(*simp add*:*gvars-conf-stable-def gvars-conf-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V,V(\!|$*rf* := (*rf V*)(*t* := *False*)$|\!))\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*lvars-nochange-def*)
apply(*subgoal-tac* $(V,V(\!|$*rf* := (*rf V*)(*t* := *False*)$|\!))\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*rule While*)
using *mp-alloc-precond7-stb* **apply** *simp*
apply(*simp add*:*Mem-pool-alloc-post-def*) **apply** *auto*[*1*]
using *mp-alloc-post-stb* **apply** *simp*

prefer *2* **apply**(*simp add*:*Mem-pool-alloc-guar-def*)

prefer *2* **apply** (*simp add*: *stable-equiv mem-pool-alloc-pre-stb*)
prefer *2* **apply**(*simp add*:*Mem-pool-alloc-guar-def*)

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-8 t p sz timeout*])
apply(*rule Seq2*[**where** *mida*=*mp-alloc-precond1-7 t p sz timeout* **and** *midb*=*mp-alloc-precond1-70 t p sz timeout*])

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-6 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-5 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-4 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-3 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-2 t p sz timeout*])
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-1 t p sz timeout*])

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-0-stb* **apply** *simp*
using *mp-alloc-precond1-1-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-0 t p sz timeout* $\cap$ $\{\!|$´*cur* = *Some t*$|\!\}$ $\cap$ $\{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (V,$V(\!|blk := (blk\ V)(t := NULL)|\!)$)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (V,$V(\!|blk := (blk\ V)(t := NULL)|\!)$)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*subgoal-tac* (V,$V(\!|blk := (blk\ V)(t := NULL)|\!)$)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-1-stb* **apply** *simp*
using *mp-alloc-precond1-2-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-1 t p sz timeout* $\cap$ $\{\!\!|$´*cur* = *Some t*$|\!\!\}$ $\cap$ $\{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*,*V*(|*alloc-lsize-r* := (*alloc-lsize-r V*)(*t* := *False*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*,*V*(|*alloc-lsize-r* := (*alloc-lsize-r V*)(*t* := *False*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*subgoal-tac* (*V*,*V*(|*alloc-lsize-r* := (*alloc-lsize-r V*)(*t* := *False*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-2-stb* **apply** *simp*
using *mp-alloc-precond1-3-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-2 t p sz timeout* $\cap$ $\{\!\!|$´*cur* = *Some t*$|\!\!\}$ $\cap$ $\{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*,*V*(|*alloc-l* := (*alloc-l V*)(*t* := *ETIMEOUT*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*,*V*(|*alloc-l* := (*alloc-l V*)(*t* := *ETIMEOUT*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)

apply(*simp add*:*alloc-memblk-valid-def*)
apply(*subgoal-tac* (*V*, *V*⦇*alloc-l* := (*alloc-l V*)(*t* := *ETIMEOUT*)⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-3-stb* **apply** *simp*
using *mp-alloc-precond1-4-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-3 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*V*} = {})
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*, *V*⦇*free-l* := (*free-l V*)(*t* := *ETIMEOUT*)⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*, *V*⦇*free-l* := (*free-l V*)(*t* := *ETIMEOUT*)⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*subgoal-tac* (*V*, *V*⦇*free-l* := (*free-l V*)(*t* := *ETIMEOUT*)⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-4-stb* **apply** *simp*
using *mp-alloc-precond1-5-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-4 t p sz timeout* ∩ ⦃´*cur* = *Some t*⦄ ∩ {*V*} = {})
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (*V*, *V*⦇*lsizes* := (*lsizes V*)(*t* := [*ALIGN4* (*max-sz* (*mem-pool-info V p*))])⦈)∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]

apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* $(V, V(\!|lsizes := (lsizes\ V)(t := [ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ p))])|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*subgoal-tac* $(V, V(\!|lsizes := (lsizes\ V)(t := [ALIGN4\ (max\text{-}sz\ (mem\text{-}pool\text{-}info\ V\ p))])|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*stable-def*)+

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-5-stb* **apply** *simp*
using *mp-alloc-precond1-6-stb* **apply** *simp*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac* $mp\text{-}alloc\text{-}precond1\text{-}5\ t\ p\ sz\ timeout \cap \{\!|{}'cur = Some\ t|\!\} \cap \{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* $(V, V(\!|i := (i\ V)(t := 0)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* $(V, V(\!|i := (i\ V)(t := 0)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*simp add*:*alloc-memblk-valid-def*)
apply(*rule conjI*)
apply(*subgoal-tac* $(V, V(\!|i := (i\ V)(t := 0)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*simp add*:*inv-def inv-mempool-info-def*) **apply** (*meson Suc-leI*)
apply *simp* **apply**(*simp add*:*stable-def*) **using** *stable-id2* **apply** *auto*[*1*]

using *lsize-loop-stm*[*of t p sz timeout*] **apply** *clarsimp*

using *precnd17-bl-170* **apply** *simp*

apply(*rule Cond*)
using *mp-alloc-precond1-70-stb* **apply** *simp*

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-70-1-stb* **apply** *simp*
using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-70-1 t p sz timeout* $\cap$ $\{\!\!|$´*cur* = *Some t*$|\!\!\}$ $\cap$ $\{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *simp*
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*, *V*$(\!|$*ret* := (*ret V*)(*t* := *ESIZEERR*)$|\!)$)$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule IntI*) **prefer** *2*
apply(*case-tac i V t* = *0*) **apply**(*simp add*:*inv-def inv-mempool-info-def*)
apply *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac* (*V*, *V*$(\!|$*ret* := (*ret V*)(*t* := *ESIZEERR*)$|\!)$)$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *simp* **using** *stable-id2* **apply** *auto*[*1*] **using** *stable-id2* **apply** *auto*[*1*]

apply(*rule Cond*)
using *mp-alloc-precond1-70-2-stb* **apply** *simp*

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-70-2-1-stb* **apply** *simp*
using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-70-2-1 t p sz timeout* $\cap$ $\{\!\!|$´*cur* = *Some t*$|\!\!\}$ $\cap$ $\{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *simp*
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)

apply(*rule disjI1*)
apply(*subgoal-tac* $(V, V(\!|ret := (ret\ V)(t := ENOMEM)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule IntI*) **prefer** *2*
apply(*case-tac* $i\ V\ t = 0$) **apply**(*simp add*:*inv-def inv-mempool-info-def*)
apply *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac* $(V, V(\!|ret := (ret\ V)(t := ENOMEM)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *simp* **using** *stable-id2* **apply** *auto*[*1*] **using** *stable-id2* **apply** *auto*[*1*]

apply(*rule Seq*[**where** *mid=mp-alloc-precond2-1 t p sz timeout*])

using *mp-alloc-stm3-lm* **apply** *simp*

apply(*rule Cond*)
using *mp-alloc-precond2-1-stb* **apply** *simp*

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-0-stb* **apply** *simp*
using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac* $mp\text{-}alloc\text{-}precond2\text{-}1\text{-}0\ t\ p\ sz\ timeout \cap \{\!|\,\acute{}cur = Some\ t|\!\} \cap \{V\} = \{\}$)
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *simp*
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* $(V, V(\!|ret := (ret\ V)(t := EAGAIN)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule IntI*) **prefer** *2*
apply(*case-tac* $i\ V\ t = 0$) **apply**(*simp add*:*inv-def inv-mempool-info-def*)
apply *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac* $(V, V(\!|ret := (ret\ V)(t := EAGAIN)|\!)) \in lvars\text{-}nochange1\text{-}4all$)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *simp* **using** *stable-id2* **apply** *auto*[*1*] **using** *stable-id2* **apply** *auto*[*1*]

apply(*rule Seq*[**where** *mid=mp-alloc-precond2-1-4 t p sz timeout*])
apply(*rule Seq*[**where** *mid=mp-alloc-precond2-1-3 t p sz timeout*])
apply(*rule Seq*[**where** *mid=mp-alloc-precond2-1-2 t p sz timeout*])
apply(*rule Seq*[**where** *mid=mp-alloc-precond2-1-1-loopinv t p sz timeout*])

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-1-stb* **apply** *simp*
using *mp-alloc-precond2-1-1-loopinv-stb* **apply** *simp*

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond2-1-1 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply** *simp*
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*, *V*(|*from-l* := (*from-l V*)(*t* := *free-l V t*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule IntI*) **prefer** *2*
apply(*case-tac i V t* = *0*) **apply**(*simp add*:*inv-def inv-mempool-info-def*)
apply *simp*
apply(*rule IntI*) **prefer** *2* **apply** *simp*
apply(*subgoal-tac* (*V*, *V*(|*from-l* := (*from-l V*)(*t* := *free-l V t*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *simp* **using** *stable-id2* **apply** *auto*[*1*] **using** *stable-id2* **apply** *auto*[*1*]

using *mp-alloc-stm4-lm* **apply** *simp*

using *mp-alloc-stm5-lm* **apply** *simp*

using *mp-alloc-stm6-lm* **apply** *simp*

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond2-1-4-stb* **apply** *simp*

using *mp-alloc-precond1-8-stb* **apply** *auto*[*1*]

apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond2-1-4 t p sz timeout* $\cap$ {|´*cur* = *Some t*|}
$\cap$ { *V* } = {})
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*) **apply**(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*(|*ret* := (*ret V*)(*t* := *OK*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*)
apply(*subgoal-tac* (*V*,*V*(|*ret* := (*ret V*)(*t* := *OK*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *auto*[*1*] **apply** *auto*[*1*] **apply** *auto*[*1*]
apply *auto*[*1*] **apply**(*simp add*:*alloc-memblk-valid-def*) **apply** *auto*[*1*]
apply(*simp add*:*alloc-memblk-valid-def*) **apply** *auto*[*1*]
apply *simp* **using** *stable-id2* **apply** *auto*[*1*] **using** *stable-id2* **apply** *auto*[*1*]

apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def*)
apply(*simp add*:*Mem-pool-alloc-guar-def*)

apply(*rule Cond*)
using *mp-alloc-precond1-8-stb* **apply** *simp*

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-8-1-1 t p sz timeout*])

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-8-1-stb* **apply** *auto*[*1*]
using *mp-alloc-precond1-8-1-1-stb* **apply** *auto*[*1*]
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-8-1 t p sz timeout* $\cap$ {|´*cur* = *Some t*|}
$\cap$ { *V* } = {})
apply *auto*[*1*] **apply** *clarify*
apply(*rule IntI*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*) **apply**(*rule disjI1*)
apply(*subgoal-tac* (*V*,*V*(|*rf* := (*rf V*)(*t* := *True*)|))$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*

lvars-nochange-def)
apply(*rule IntI*) **apply**(*rule IntI*) **apply**(*rule IntI*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def gvars-conf-stable-def gvars-conf-def*)
apply(*subgoal-tac* (V,V(|*rf* := (*rf* V)(t := *True*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply *simp*
apply(*simp add*:*alloc-memblk-valid-def*) **apply** *simp*
apply(*simp add*:*stable-def*)+

apply(*rule Cond*)
using *mp-alloc-precond1-8-1-1-stb* **apply** *auto*[*1*]

apply(*simp add*:*stm-def*)
apply(*rule Await*)
using *mp-alloc-precond1-8-1-2-stb* **apply** *auto*[*1*]
using *mp-alloc-precond7-stb* **apply** *auto*[*1*]
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-8-1-2 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {V} = {})
apply *auto*[*1*] **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def gvars-conf-stable-def gvars-conf-def lvars-nochange-def*)
apply(*rule disjI1*)
apply(*subgoal-tac* (V,V(|*ret* := (*ret* V)(t := *ENOMEM*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply(*subgoal-tac* (V,V(|*ret* := (*ret* V)(t := *ENOMEM*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*] **apply**(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def*)
apply *simp*
apply(*simp add*:*stable-def*) **apply** *auto*[*1*]
apply(*simp add*:*stable-def*)

apply(*simp add*:*Skip-def*)
apply(*rule Basic*)
apply *auto*[*1*] **apply**(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *auto*[*1*]

using *mp-alloc-precond1-8-1-3-stb* **apply** *auto*[*1*]
using *mp-alloc-precond7-stb* **apply** *auto*[*1*]

apply(*simp add*:*Mem-pool-alloc-guar-def*)

apply(*rule Cond*)
using *mp-alloc-precond1-8-2-stb* **apply** *simp*

apply(*simp add*:*Skip-def*)
apply(*rule Basic*) **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *auto*[*1*]
using *mp-alloc-precond1-8-2-1-stb* **apply** *simp*
using *mp-alloc-precond7-stb* **apply** *simp*

apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-8-2-2 t p sz timeout*])

using *mp-alloc-stm7-lm* **apply** *simp*

apply(*rule Cond*)
using *mp-alloc-precond1-8-2-2-stb* **apply** *auto*[*1*]
apply(*rule Seq*[**where** *mid*=*mp-alloc-precond1-8-2-4 t p sz timeout*])

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-8-2-3-stb mp-pred1823-eq* **apply** *auto*[*1*]
using *mp-alloc-precond1-8-2-4-stb* **apply** *blast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-8-2-3 t p sz timeout* $\cap$ $\{\!|$ ´*cur* = *Some t*$|\!\}$ $\cap$ $\{V\}$ = $\{\}$)
apply *auto*[*1*] **apply** *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply**(*rule disjI1*)
apply(*simp add*:*Mem-pool-alloc-guar-def lvars-nochange1-def lvars-nochange-def*
gvars-conf-stable-def gvars-conf-def)
apply(*subgoal-tac* $(V, V(\!|tmout := (tmout\ V)(t := int\ (endt\ V\ t) - int\ (tick\ V))|\!))$$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply(*subgoal-tac* $(V, V(\!|tmout := (tmout\ V)(t := int\ (endt\ V\ t) - int\ (tick\ V))|\!))$$\in$*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*) **apply** *auto*[*1*]
apply(*simp add*:*stable-def*) **apply** *auto*[*1*] **apply**(*simp add*:*stable-def*)
apply *auto*[*1*]

apply(*rule Cond*)
using *mp-alloc-precond1-8-2-4-stb* **apply** *blast*

apply(*rule Seq*[**where** *mid=mp-alloc-precond1-8-2-5 t p sz timeout*])

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-8-2-40-stb* **apply** *blast*
using *mp-alloc-precond1-8-2-5-stb* **apply** *blast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-8-2-40 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
apply *auto*[*1*] **apply** *auto*[*1*]
using *mp-alloc-stm8-guar* **apply** *simp*
apply(*subgoal-tac* (*V*,*V*(|*rf* := (*rf V*)(*t* := *True*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
using *mp-alloc-stm8-guar* **apply** *simp*
apply(*subgoal-tac* (*V*,*V*(|*rf* := (*rf V*)(*t* := *True*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
using *mp-alloc-stm8-guar* **apply** *simp*
apply *simp* **apply**(*simp add*:*stable-def*) **apply** *auto*[*1*] **apply**(*simp add*:*stable-def*) **apply** *auto*[*1*]

apply(*unfold stm-def*)[*1*]
apply(*rule Await*)
using *mp-alloc-precond1-8-2-5-stb* **apply** *blast*
using *mp-alloc-precond7-stb* **apply** *blast*
apply(*rule allI*)
apply(*rule Basic*)
apply(*case-tac mp-alloc-precond1-8-2-5 t p sz timeout* ∩ {|´*cur* = *Some t*|} ∩ {*V*} = {})
apply *auto*[*1*] **apply** *auto*[*1*]
using *mp-alloc-stm9-guar* **apply** *simp*
apply(*subgoal-tac* (*V*,*V*(|*ret* := (*ret V*)(*t* := *ETIMEOUT*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
using *mp-alloc-stm9-guar* **apply** *simp*
apply(*subgoal-tac* (*V*,*V*(|*ret* := (*ret V*)(*t* := *ETIMEOUT*)|))∈*lvars-nochange1-4all*)
using *glnochange-inv0* **apply** *auto*[*1*]
apply(*simp add*:*lvars-nochange1-4all-def lvars-nochange1-def lvars-nochange-def*)
apply *simp* **apply**(*simp add*:*stable-def*) **apply** *auto*[*1*] **apply**(*simp add*:*stable-def*)

apply(*unfold Skip-def*)[*1*]
apply(*rule Basic*)

apply *auto*[*1*]
prefer *2* **using** *mp-alloc-precond1-8-2-41-stb* **apply** *fast*
prefer *2* **using** *mp-alloc-precond7-stb* **apply** *blast*
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *auto*[*1*]

apply(*simp add*:*Mem-pool-alloc-guar-def*)

apply(*unfold Skip-def*)[*1*]
apply(*rule Basic*)
apply *auto*[*1*]
apply(*simp add*:*Mem-pool-alloc-guar-def*) **apply** *auto*[*1*]
using *mp-alloc-precond1-8-2-20-stb* **apply** *fast*
using *mp-alloc-precond7-stb* **apply** *blast*

apply(*simp add*:*Mem-pool-alloc-guar-def*)+
done

end

theory *memory-manage-sys*
imports *rg-cond func-cor-other func-cor-mempoolfree func-cor-mempoolalloc ../../picore/PiCore-ext*
begin

22 Formal specification of Zephyr memory management

context *event-hoare* **begin**

definition *mk-react-sys* :: (('*a*, '*b*, '*s*, '*prog*) *esys*, '*s*) *PiCore-Validity.rgformula set* ⇒ ('*a*, '*b*, '*s*, '*prog*) *esys* **where**
‹*mk-react-sys rgfs* ≡ *react-sys* (*map rgformula.Com* (*list-of-set rgfs*))›

end

lemma *event-hoareI*: *event-hoare ptranI None prog-validityI rghoare-pI*
apply(*rule event-hoare.intro*)
apply(*rule event-validity.intro*)
apply(*rule event-compI*)
apply(*rule event-validity-axioms.intro*)
apply(*erule prog-validity-defI*)
apply(*rule event-hoare-axioms.intro*)
using *rgsound-pI* **by** *blast*

consts
all-ref :: *mempool-ref set*
max-sz :: *nat*

max-timeout :: *int*
all-blocks :: *Mem-block set*
all-threads :: *Thread set*

axiomatization where
all-ref-finite: ‹*finite all-ref*› **and**
all-blocks-finite: ‹*finite all-blocks*› **and**
all-threads-finite: ‹*finite all-threads*› **and**
all-threads-nonempty: ‹*all-threads* $\neq$ {}› **and**
all-ref-nonempty: ‹*all-ref* $\neq$ {}› **and**
max-timeout-nonneg: ‹*max-timeout* $\geq$ *0*›

definition *Thread-RGF* :: *Thread* $\Rightarrow$ ((*EL* $\times$ *Parameter list* $\times$ *Core*, *'a*, *State*, *State com option*) *esys*, *State*) *PiCore-Validity.rgformula*
where *Thread-RGF t* $\equiv$
(| *rgformula.Com* = *mk-react-sys* (($\bigcup$ *p*$\in$*all-ref*. $\bigcup$ *sz*$\leq$*max-sz*. $\bigcup$ *timeout*$\in${($-$*1*)..*max-timeout*}. {*Mem-pool-alloc-RGCond t p sz timeout*}) $\cup$
($\bigcup$ *b*$\in$*all-blocks*. {*Mem-pool-free-RGCond t b*})),
Pre = (*Mem-pool-free-pre t* $\cap$ *Mem-pool-alloc-pre t*),
Rely = (*Mem-pool-free-rely t* $\cap$ *Mem-pool-alloc-rely t*),
Guar = (*Mem-pool-free-guar t* $\cup$ *Mem-pool-alloc-guar t*),
Post = (*Mem-pool-free-post t* $\cup$ ($\bigcup$ *p*$\in$*all-ref*. $\bigcup$ *sz*$\leq$*max-sz*. $\bigcup$ *timeout*$\in${($-$*1*)..*max-timeout*}. *Mem-pool-alloc-post t p sz timeout*)) |)

definition *Scheduler-RGF*
where *Scheduler-RGF* $\equiv$
(| *rgformula.Com* = *mk-react-sys* ($\bigcup$ *t*$\in$*all-threads*. {*Schedule-RGCond t*}),
Pre = {*s*. *inv s*}, *Rely* = *Schedule-rely*, *Guar* = *Schedule-guar*, *Post* = {*s*. *inv s*} |)

definition *Timer-RGF*
where *Timer-RGF* $\equiv$
(| *rgformula.Com* = *mk-react-sys* {*Tick-RGCond*},
Pre = {|*True*|}, *Rely* = *Tick-rely*, *Guar* = *Tick-guar*, *Post* = {|*True*|} |)

definition *Memory-manage-system-Spec*
where *Memory-manage-system-Spec k* $\equiv$
case k of ($\mathcal{T}$ *t*) $\Rightarrow$ *Thread-RGF t*
| $\mathcal{S}$ $\Rightarrow$ *Scheduler-RGF*
| *Timer* $\Rightarrow$ *Timer-RGF*

23 functional correctness of the whole specification

definition *sys-rely* $\equiv$ {}

definition *sys-guar* ≡ *Tick-guar* ∪ *Schedule-guar* ∪ ($\bigcup$ *t*. (*Mem-pool-free-guar t* ∪ *Mem-pool-alloc-guar t*))

lemma *scheduler-esys-sat*: *Evt-sat-RG* Γ (*Memory-manage-system-Spec* $\mathcal{S}$)
apply(*simp add*: *Evt-sat-RG-def Memory-manage-system-Spec-def mk-react-sys-def Scheduler-RGF-def*)
apply(*rule event-hoare.Evt-react-set*)
apply(*fact event-hoareI*)
apply *auto*[*1*]
using *Schedule-satRG* **apply**(*simp add*:*Schedule-RGCond-def Evt-sat-RG-def Schedule-def*)
apply *fast*
apply(*simp add*: *Schedule-RGCond-def*)
apply(*simp add*: *Schedule-RGCond-def*)
apply(*simp add*: *Schedule-RGCond-def*)
apply(*simp add*: *Schedule-RGCond-def*)
apply(*simp add*: *Schedule-RGCond-def*)
apply(*fact all-threads-nonempty*)
using *all-threads-finite* **apply** *simp*
using *stable-inv-sched-rely* **apply**(*simp add*: *PiCore-Hoare.stable-def*)
apply(*simp add*: *Schedule-guar-def*)
done

lemma *stable-int2-r*: ‹*stable p r* $\Longrightarrow$ *stable p* (*r* ∩ *s*)› **by** (*simp add*: *stable-def*)

thm *event-hoare.Evt-react-set*

lemma *thread-esys-sat*: *Evt-sat-RG* Γ (*Memory-manage-system-Spec* ($\mathcal{T}$ *t*))
apply(*simp add*: *Evt-sat-RG-def Memory-manage-system-Spec-def mk-react-sys-def Thread-RGF-def*)
apply(*rule event-hoare.Evt-react-set'*)
apply(*fact event-hoareI*)
apply *auto*[*1*]
using *Mempool-alloc-satRG* **apply**(*simp add*: *Evt-sat-RG-def Mem-pool-alloc-RGCond-def*)
apply *force*
apply(*simp add*: *Mem-pool-alloc-RGCond-def*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def Mem-pool-alloc-post-def*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def Mem-pool-alloc-post-def*)
apply(*simp add*: *Mem-pool-alloc-RGCond-def Mem-pool-alloc-post-def*)
using *Mempool-alloc-satRG* **apply**(*simp add*: *Evt-sat-RG-def Mem-pool-alloc-RGCond-def Mem-pool-free-RGCond-def*)
apply (*smt Collect-cong Evt-sat-RG-def Mem-pool-free-RGCond-def Mempool-free-satRG rgformula.select-convs*(*1*) *rgformula.select-convs*(*2*) *rgformula.select-convs*(*3*) *rgformula.select-convs*(*4*) *rgformula.select-convs*(*5*))
apply(*simp add*: *Mem-pool-free-RGCond-def*)
apply(*simp add*: *Mem-pool-free-RGCond-def*)
apply(*simp add*: *Mem-pool-free-RGCond-def*)
apply(*simp add*: *Mem-pool-free-RGCond-def Mem-pool-free-post-def*)

apply(*simp add*: *Mem-pool-free-RGCond-def Mem-pool-free-post-def*)
apply(*simp add*: *Mem-pool-free-RGCond-def Mem-pool-free-post-def*)
apply *auto*[*1*]
using *all-ref-nonempty* **apply** *blast*
using *max-timeout-nonneg* **apply** *fastforce*
using *all-ref-finite all-blocks-finite* **apply** *simp*
apply(*subst stable-equiv*)
using *mem-pool-free-pre-stb stable-int2-r* **apply** *blast*
apply(*simp add*: *Mem-pool-free-guar-def*)
using *Mem-pool-free-post-def* **by** *auto*

lemma *timer-esys-sat*: *Evt-sat-RG* Γ (*Memory-manage-system-Spec Timer*)
apply(*simp add*: *Memory-manage-system-Spec-def Evt-sat-RG-def Timer-RGF-def mk-react-sys-def*)
apply(*rule event-hoare.Evt-react-set*)
apply(*rule event-hoare.intro*)
apply(*rule event-validity.intro*)
apply(*rule event-compI*)
apply(*rule event-validity-axioms.intro*)
apply(*erule prog-validity-defI*)
apply(*rule event-hoare-axioms.intro*)
using *rgsound-pI* **apply** *blast*
apply *auto*[*1*]
using *Tick-satRG* **apply**(*simp add*:*Tick-RGCond-def Evt-sat-RG-def Tick-def*)
apply *fast*
apply(*simp add*:*Tick-RGCond-def*)+
apply(*simp add*: *PiCore-Hoare.stable-def*)
apply(*simp add*: *Tick-guar-def*)
done

lemma *esys-sat*: *Evt-sat-RG* Γ (*Memory-manage-system-Spec k*)
apply(*induct k*)
using *scheduler-esys-sat* **apply** *fast*
using *thread-esys-sat* **apply** *fast*
using *timer-esys-sat* **apply** *fast*
done

lemma *s0-esys-pre*: {*s0*} ⊆ *rgformula.Pre* (*Memory-manage-system-Spec k*)
apply(*induct k*)
apply(*simp add*:*Memory-manage-system-Spec-def Scheduler-RGF-def*)
using *s0-inv* **apply** *fast*
apply(*simp add*:*Memory-manage-system-Spec-def Thread-RGF-def*)
using *s0-inv s0a4 s0a10* **apply** *auto*[*1*]
apply(*simp add*:*Memory-manage-system-Spec-def Timer-RGF-def*)
done

lemma *alloc-free-eq-guar*: *Mem-pool-free-guar x* = *Mem-pool-alloc-guar x*
by(*simp add*:*Mem-pool-free-guar-def Mem-pool-alloc-guar-def*)

lemma *alloc-free-eq-rely*: *Mem-pool-free-rely x* = *Mem-pool-alloc-rely x*
by(*simp add*:*Mem-pool-free-rely-def Mem-pool-alloc-rely-def*)

lemma *esys-guar-in-other*:
jj ≠ *k* ⟶ *rgformula.Guar* (*Memory-manage-system-Spec jj*) ⊆ *rgformula.Rely* (*Memory-manage-system-Spec k*)
apply *auto*
apply(*induct jj*)
apply(*induct k*)
apply *simp*
apply(*simp add*: *Memory-manage-system-Spec-def Scheduler-RGF-def Thread-RGF-def*)
using *schedguar-in-allocrely* **apply**(*simp add*:*Mem-pool-free-rely-def Mem-pool-alloc-rely-def*)
apply *auto*[*1*]
apply(*simp add*: *Memory-manage-system-Spec-def Scheduler-RGF-def Timer-RGF-def*)
using *schedguar-in-tickrely* **apply** *auto*[*1*]
apply(*induct k*)
apply(*simp add*: *Memory-manage-system-Spec-def Scheduler-RGF-def Thread-RGF-def*)
apply *auto*[*1*]
using *allocguar-in-schedrely alloc-free-eq-guar* **apply** *fast*
using *allocguar-in-schedrely* **apply** *fast*
apply(*simp add*: *Memory-manage-system-Spec-def Thread-RGF-def*)
apply *auto*[*1*]
using *allocguar-in-allocrely alloc-free-eq-guar alloc-free-eq-rely* **apply** *fast+*
apply(*simp add*: *Memory-manage-system-Spec-def Timer-RGF-def Thread-RGF-def*)
apply *auto*[*1*]
using *allocguar-in-tickrely alloc-free-eq-guar alloc-free-eq-rely* **apply** *fast+*
apply(*induct k*)
apply(*simp add*: *Memory-manage-system-Spec-def Scheduler-RGF-def Timer-RGF-def*)
using *tickguar-in-schedrely* **apply** *fast*
apply(*simp add*: *Memory-manage-system-Spec-def Thread-RGF-def Timer-RGF-def*)
apply *auto*[*1*]
using *tickguar-in-allocrely alloc-free-eq-guar alloc-free-eq-rely* **apply** *fast+*
done

lemma *esys-guar-in-sys*: *rgformula.Guar* (*Memory-manage-system-Spec k*) ⊆ *sys-guar*
apply(*induct k*)
apply(*simp add*: *Memory-manage-system-Spec-def Scheduler-RGF-def sys-guar-def*)
apply *auto*[*1*]
apply(*simp add*: *Memory-manage-system-Spec-def Thread-RGF-def sys-guar-def*)
apply *auto*[*1*]
apply(*simp add*: *Memory-manage-system-Spec-def Timer-RGF-def sys-guar-def*)
apply *auto*[*1*]
done

lemma *mem-sys-sat*: *rghoare-pes* Γ *Memory-manage-system-Spec* {*s0*} *sys-rely* *sys-guar UNIV*
apply(*rule Par*[*of* Γ *Memory-manage-system-Spec* {*s0*} *sys-rely sys-guar UNIV*])
apply *clarify* **using** *esys-sat*
using *Evt-sat-RG-def* **apply** *fastforce*

using *s0-esys-pre* **apply** *fast*
apply(*simp add*:*sys-rely-def*)
using *esys-guar-in-other* **apply** *fast*
using *esys-guar-in-sys* **apply** *fast*
apply *simp*
done

end

theory *memory-management-inv*
imports *memory-manage-sys*
begin

24 invariant verification

theorem *invariant-presv-pares* Γ *inv* (λ*k*. *rgformula.Com* (*Memory-manage-system-Spec* *k*)) {*s0*} *sys-rely*
apply(*rule invariant-theorem*[**where** *G*=*sys-guar* **and** *pst* = *UNIV*])
using *mem-sys-sat* **apply** *fast*
apply(*simp add*:*sys-rely-def PiCore-Hoare.stable-def*)
apply(*simp add*:*sys-guar-def*)
apply(*subst stable-equiv*)
apply(*rule stable-un-R*) **apply**(*rule stable-un-R*)
using *tick-guar-stb-inv* **apply**(*simp add*:*stable-def*)
using *sched-guar-stb-inv* **apply**(*simp add*:*stable-def*)
apply(*rule stable-un-S*) **apply** *clarify* **apply**(*rule stable-un-R*)
using *alloc-guar-stb-inv alloc-free-eq-guar* **apply**(*simp add*:*stable-def*)
using *alloc-guar-stb-inv* **apply**(*simp add*:*stable-def*)
using *s0-inv* **apply** *simp*
done

end