specification

zy

July 6, 2019

Contents

theory *Design*
  imports *Main HOL−Word.Word HOL−Library.Log-Nat HOL−Lattice.Lattice*
  HOL.Lattices Lib.NonDetMonadVCG
begin
declare [[*smt-solver* = *z3*]]
declare [[*smt-timeout* = *300*]]
declare [[*z3-options* = *−memory*:*1000*]]

instantiation *nat* :: *leq*
begin

definition
  leq-nat-def[*simp*]: $p \sqsubseteq q \equiv (p{::}nat) \leq q$

instance ..

end

instance *nat* :: *quasi-order*
proof
  fix *x*::*nat* **and** *y*::*nat* **and** *z*::*nat*
  show $x \sqsubseteq x$ **using** *leq-nat-def* **by** *auto*
  show $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
    using *leq-nat-def* **by** *auto*
qed

instance *nat* :: *partial-order*
proof
  fix *x*::*nat* **and** *y*::*nat*
  show $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$ **using** *leq-nat-def* **by** *auto*
qed

instantiation *nat* :: *lattice*
begin

instance

proof
fix *x*::*nat* **and** *y*::*nat*
show ∃ *inf*. *is-inf x y inf* **unfolding** *is-inf-def leq-nat-def*
proof−
have (*Lattices.inf x y*) ≤ *x* **by** *auto*
moreover have (*Lattices.inf x y*)≤ *y* **by** *auto*
moreover have (∀ *z*. *z* ≤ *x* ∧ *z* ≤ *y* ⟶ *z* ≤ (*Lattices.inf x y*)) **by** *auto*
ultimately show ∃ *inf*≤*x*. *inf* ≤ *y* ∧ (∀ *z*. *z* ≤ *x* ∧ *z* ≤ *y* ⟶ *z* ≤ *inf*) **by** *fastforce*
qed
show ∃ *sup*. *is-sup x y sup* **unfolding** *is-sup-def leq-nat-def*
proof−
have (*Lattices.sup x y*) ≥ *x* **by** *auto*
moreover have *y*≤ (*Lattices.sup x y*) **by** *auto*
moreover have (∀ *z*. *x* ≤ *z* ∧ *y* ≤ *z* ⟶ (*Lattices.sup x y*) ≤ *z*)
by *auto*
ultimately show ∃ *sup*≥*x*. *y* ≤ *sup* ∧ (∀ *z*. *x* ≤ *z* ∧ *y* ≤ *z* ⟶ *sup* ≤ *z*)
by *fastforce*
qed
qed

end

datatype *bhdr-t* = *Bhdr* (*s-addr*:*nat*) (*e-addr*:*nat*)

primrec *b-size*::*bhdr-t* ⇒ *nat*
where *b-size* (*Bhdr s e*) = (*1* + *e* − *s*)

type-synonym *word32* = *32 word*
type-synonym *bitmap-t* = *word32*

— we declare fl to get finite the set of all the free blocks in a segregation matrix l defines the logarithm in base 2 for the second level segregation list sm is the logarithm in base 2 for the small block size, which will give the base size for first level segregations list for indexes bigger than 0. Index 0 will contain the free blocks of size smaller than sm. min block gives the minimum size block. This is included to discard those allocations of size smaller than min block, otherwise the implementation would include behaviours not contained in the specification

record *Sys-Config* =

l :: *nat*
sm :: *nat*
min-block::*nat*
overhead::*nat*
mem-size::*nat*

consts *conf*:: *Sys-Config*

specification(*conf*)
mbiggerl: *sm conf* ≥ *l conf*
min-block-gt-overhead:*min-block conf* > *overhead conf*
oh-gt-0: *overhead conf* > *0*

total-mem-gt-0: *mem-size conf* > *0*

apply (*rule exI*[*of* - (| *l* = *0*, *sm* = *0*, *min-block* = *2*, *overhead* = *1*,*mem-size* = *1* |)])
by *auto*

definition *sl* ::*Sys-Config* ⇒ *nat*
where *sl cfg* ≡ *2* ^ (*l cfg*)

declare *sl-def*[*simp*]

type-synonym *bhdr-matrix-t* = *nat* ⇒ *nat* ⇒ *bhdr-t set*

record *state-t* =
bhdr-matrix-f :: *bhdr-matrix-t*
alloced-bhdr-s :: *bhdr-t set*

definition *block-alloced*::*nat* ⇒ *state-t* ⇒ *bool*
where *block-alloced addr σ* ≡ ∃ *e-addr*. (*Bhdr addr e-addr*) ∈ (*alloced-bhdr-s σ*)

definition *get-alloced-block*::*nat* ⇒ *state-t* ⇒ *bhdr-t*
where *get-alloced-block addr σ* ≡ *THE b*. ∃ *e-addr*. *b* = (*Bhdr addr e-addr*) ∧ *b* ∈ (*alloced-bhdr-s σ*)

definition *set-bhdr-matrix*:: *bhdr-matrix-t* ⇒ *nat* ⇒ *nat* ⇒ *bhdr-t set* ⇒ *bhdr-matrix-t*
where *set-bhdr-matrix m i j v* ≡ *m*(*i*:= (*m i*)(*j*:= *v*))

definition *insert-block-bhdr-matrix*:: *bhdr-matrix-t* ⇒ *nat* ⇒ *nat* ⇒ *bhdr-t* ⇒ *bhdr-matrix-t*
where *insert-block-bhdr-matrix m i j b* ≡ *set-bhdr-matrix m i j* (*insert b* (*m i j*))

definition *free-blocks* ::*Sys-Config* ⇒ *state-t* ⇒ *bhdr-t set*
where *free-blocks cfg σ* ≡ ⋃{*x*. (∃ *f s*. *s* < *sl cfg* ∧ *x* = (*bhdr-matrix-f σ*) *f s*)}

definition *free-blocks-mat* ::*Sys-Config* ⇒ *bhdr-matrix-t* ⇒ *bhdr-t set*
where *free-blocks-mat cfg m* ≡ ⋃{*x*. (∃ *f s*. *s* < *sl cfg* ∧ *x* = *m f s*)}

lemma *free-blk-mat-s-eq*:
free-blocks conf s = free-blocks-mat conf (bhdr-matrix-f s)
unfolding *free-blocks-def free-blocks-mat-def* **..**

definition *all-blocks :: Sys-Config ⇒ state-t ⇒ bhdr-t set*
where
all-blocks cfg σ ≡ free-blocks cfg σ ∪ alloced-bhdr-s σ

definition *prev-hdr-s :: Sys-Config ⇒ bhdr-t ⇒ state-t ⇒ bhdr-t option*
where
prev-hdr-s cfg b σ ≡
(let s = {b′. (∀ s-addr e-addr. b = Bhdr s-addr e-addr ⟶
(∃ s-addr′ e-addr′. b′ = Bhdr s-addr′ e-addr′ ∧
b′ ∈ all-blocks cfg σ ∧ e-addr′ + 1 + overhead conf = s-addr))
} in
if s = {} then None else Some (THE x. x ∈ s))

definition *prev-free-hdr-s :: Sys-Config ⇒ bhdr-t ⇒ state-t ⇒ bhdr-t option*
where
prev-free-hdr-s cfg b σ ≡
(let s = {b′. (∀ s-addr e-addr. b = Bhdr s-addr e-addr ⟶
(∃ s-addr′ e-addr′. b′ = Bhdr s-addr′ e-addr′ ∧
b′ ∈ free-blocks cfg σ ∧ e-addr′ + 1 + overhead conf = s-addr))
} in
if s = {} then None else Some (THE x. x ∈ s))

definition *suc-hdr-s :: Sys-Config ⇒ bhdr-t ⇒ state-t ⇒ bhdr-t option*
where
suc-hdr-s cfg b σ ≡
(let s = {b′. (∀ s-addr e-addr. b = Bhdr s-addr e-addr ⟶
(∃ s-addr′ e-addr′. b′ = Bhdr s-addr′ e-addr′ ∧
b′ ∈ all-blocks cfg σ ∧ e-addr + 1 + overhead conf = s-addr′))
} in
if s = {} then None else Some (THE x. x ∈ s))

definition *suc-hdr-free-s :: Sys-Config ⇒ bhdr-t ⇒ state-t ⇒ bhdr-t option*
where
suc-hdr-free-s cfg b σ ≡
(let s = {b′. (∀ s-addr e-addr. b = Bhdr s-addr e-addr ⟶
(∃ s-addr′ e-addr′. b′ = Bhdr s-addr′ e-addr′ ∧
b′ ∈ free-blocks cfg σ ∧ e-addr + 1 + overhead conf = s-addr′))
} in
if s = {} then None else Some (THE x. x ∈ s))

definition *is-alloc :: bhdr-t ⇒ state-t ⇒ bool*
where *is-alloc b σ ≡ b ∈ alloced-bhdr-s σ*

definition *is-free* :: *Sys-Config* ⇒ *bhdr-t* ⇒ *state-t* ⇒ *bool*
where *is-free cfg b σ* ≡ *b* ∈ *free-blocks cfg σ*

definition *size-l1*::*nat* ⇒ *nat*
where *size-l1 i* ≡ *2* ^ *i*

definition *size-small-l1* ::*Sys-Config* ⇒ *nat* ⇒ *nat*
where *size-small-l1 cfg i* ≡
let size-i = *if* (*i*=*0*) *then 2 else size-l1 i in*
size-i ∗ *2*^(*sm cfg* −*1*)

lemma *small-level-i-l-i'*:*i*<*i'* ⟹ *size-small-l1 cfg i* ≤ *size-small-l1 cfg i'*
proof (*induct i'*)
case *0*
then show *?case* **unfolding** *size-small-l1-def size-l1-def*
by *auto*
next
case (*Suc i'*)
{assume *i*=*i'* **then have** *?case* **using** *Suc* **unfolding** *size-small-l1-def size-l1-def*
by *auto*
}
moreover {assume *i*<*i'*
then have *size-small-l1 cfg i* ≤ *size-small-l1 cfg i'* **using** *Suc* **by** *auto*
moreover have *size-small-l1 cfg i'* ≤ *size-small-l1 cfg* (*Suc i'*)
unfolding *size-small-l1-def size-l1-def* **by** *auto*
ultimately have *?case* **unfolding** *size-small-l1-def size-l1-def*
by *linarith*
}
ultimately show *?case* **using** *Suc* **by** *fastforce*
qed

lemma *small-level-Suci-lt-Suci'*:*i*<*i'* ⟹ *size-small-l1 cfg* (*Suc i*) < *size-small-l1 cfg* (*Suc i'*)
proof (*induct i'*)
case *0*
then show *?case* **unfolding** *size-small-l1-def size-l1-def*
by *auto*
next
case (*Suc i'*)
{assume *i*=*i'* **then have** *?case* **using** *Suc* **unfolding** *size-small-l1-def size-l1-def*
by *auto*
}
moreover {assume *i*<*i'*
then have *size-small-l1 cfg* (*Suc i*) < *size-small-l1 cfg* (*Suc i'*) **using** *Suc* **by** *auto*
moreover have *size-small-l1 cfg* (*Suc i'*) < *size-small-l1 cfg* (*Suc* (*Suc i'*))
unfolding *size-small-l1-def size-l1-def* **by** *auto*
ultimately have *?case* **unfolding** *size-small-l1-def size-l1-def*

```
    by linarith
  }
  ultimately show ?case using Suc by fastforce
qed

primrec range-l1 :: Sys-Config ⇒ nat ⇒ (nat×nat)
  where range-l1 cfg 0 = (0, (size-small-l1 cfg 0) −1)
  | range-l1 cfg (Suc n) = (size-small-l1 cfg (Suc n) ,2∗size-small-l1 cfg (Suc n) −
1)


lemma size-small-level-gt-1: Suc 0 < size-small-l1 cfg n
  apply (cases sm cfg, induct n)
  by (auto simp add:  size-small-l1-def size-l1-def Suc-lessI)

lemma mult-g-1:x > Suc 0 ⟹ n>1 ⟹ x < n∗x − 1
  by (simp add:  add.commute mult-eq-if)


lemma range-l1-fst-lt-snd:
  fst (range-l1 cfg i) < snd (range-l1 cfg i)
using mult-g-1 by(cases i; simp add: size-small-level-gt-1)

lemma snd-rang-i-eq-fst-rang-suc-i:
snd (range-l1 cfg n) + 1 = fst (range-l1 cfg (Suc  n))
  by (cases n, auto simp add: size-small-l1-def size-l1-def)


lemma range-level1-disj:i ≠ i' ⟹ range-l1 cfg i ≠ range-l1 cfg i'
   by (cases i; cases i', auto simp add: range-l1-def size-small-l1-def size-l1-def)


lemma range-l1-snd-i-lt-fst-i':i < i' ⟹ snd (range-l1 cfg i) < fst (range-l1 cfg
i')
proof (induct i' − i −1 arbitrary: i)
  case 0
  then have i' = Suc i by auto
  then show ?case using snd-rang-i-eq-fst-rang-suc-i
    by (metis less-add-one)
next
  case (Suc x)
  then have Suc i < i' by auto
  moreover have x = i' − (Suc i) − 1
    using Suc(2) by auto
  ultimately have snd (range-l1 cfg (Suc i)) < fst (range-l1 cfg i')
    using Suc(1) by fast
  moreover have  snd (range-l1 cfg i) < fst (range-l1 cfg (Suc i))
  using snd-rang-i-eq-fst-rang-suc-i
  by (metis less-add-one)
```

ultimately show *?case* **using** *range-l1-fst-lt-snd* **by** *auto*
qed

lemma *range-l1-i-not-0*:
assumes *a0*:*i*>*0*
shows *fst* (*range-l1 cfg i*) = *2*^(*i* + (*sm cfg* −*1*))
proof−
obtain *i'* **where** *i*= *Suc i'* **using** *a0*
using *Suc-pred'* **by** *blast*
then have *fst* (*range-l1 cfg i*) = (*size-small-l1 cfg i*) **by** *auto*
thus *?thesis* **unfolding** *size-small-l1-def size-l1-def Let-def* **using** *a0*
apply *auto*
by (*simp add*: *power-add*)
qed

lemma *range-l1-i-0*:
shows*fst* (*range-l1 cfg 0*) = *0*
by *auto*

definition *size-l2*::*Sys-Config* ⇒ *nat* ⇒ *nat*
where *size-l2 cfg i* ≡(*size-small-l1 cfg i*) *div* (*sl cfg*)

definition *l2-i-j* ::*Sys-Config* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where *l2-i-j cfg i j* ≡ *fst* (*range-l1 cfg i*)+(*size-l2 cfg i*)∗*j*

lemma *fst-level1-not-0*:*fst* (*range-l1 conf* (*Suc i*)) ≠ *0*
by (*cases i*,*auto simp add*: *size-small-l1-def size-l1-def*)

lemma *power2gt0*:*a*≤*b* ⟹ *0*<*a* ⟹((*2*::*nat*) ^*b div 2*^*a*) > *0*
by (*metis Euclidean-Division.div-eq-0-iff div-positive gr0I*
nat-power-less-imp-less pos2 power-not-zero)

lemma *power2gt*:*a*≤*b* ⟹ ((*2*::*nat*) ^*b div 2*^*a*) > *0*
using *power2gt0* **by** *fastforce*

lemma *power2gt1*:*a*≤*b* ⟹ *c*>*0* ⟹ ((*c*∗((*2*::*nat*) ^*b*)) *div 2*^*a*) > *0*
using *power2gt*
by (*metis div-less nat-mult-less-cancel1 neq0-conv pos2 td-gal-lt zero-less-power*)

lemma *size-l2-not-0*:*sm cfg* ≥ *l cfg* ⟹ *size-l2 cfg i*≠*0*
unfolding *size-l2-def size-small-l1-def size-l1-def*
apply (*cases i*) **using** *nat-power-eq-Suc-0-iff* **apply** *auto*
apply (*metis Suc-pred gr0I le-zero-eq power2gt power-Suc zero-le*)
apply (*cases sm cfg*)
using *power2gt1 Euclidean-Division.div-eq-0-iff le-Suc-eq* **by** *fastforce*+

lemma *power-greater-mod-zero*:
assumes *a0*:(*i*::*nat*)≥*j*
shows (*2*::*nat*) ^*i mod 2*^*j* = *0*
proof−
obtain *k* **where** *i*= *k*+*j* **using** *a0*
by (*metis le-add-diff-inverse2*)
also have (*2*::*nat*) ^ (*k* + *j*) = *2*^ (*k*)∗ *2*^*j* **using** *power-add* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *power-div-neutro*:
assumes *a0*:*sm cfg* ≥ *l cfg*
shows (*2*::*nat*) ∗ *2* ^ *k* ∗ *2* ^ (*sm cfg* − *Suc 0*) *div 2* ^ *l cfg* ∗ *2* ^ *l cfg* = *2* ∗ *2* ^ *k* ∗ *2* ^ (*sm cfg* − *Suc 0*)
proof(*cases l cfg*)
assume *l cfg* = *0*
then show *?thesis* **by** *auto*
next
fix *k*
assume *l cfg* = *Suc k*
then show *?thesis* **using** *power-greater-mod-zero*[*OF a0*]
by (*smt Suc-neq-Zero Suc-pred add.right-neutral assms div-mult-mod-eq gr0I le-zero-eq mod-mult-self2-is-0*
power-Suc semiring-normalization-rules(*16*))
qed

lemma *size-l2-m-zero0*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*sm cfg* = *0* **and** *a2*:*i*=*0*
shows (*size-l2 cfg i*) = *2*
using *a0 a1 a2* **unfolding** *size-l2-def size-small-l1-def Let-def size-l1-def*
by *auto*

lemma *size-l2-m-not-zero0*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*sm cfg* > *0* **and** *a2*:*i*=*0*
shows (*size-l2 cfg i*) = (*2*^((*sm cfg*) − (*l cfg*)))
using *a0 a1 a2* **unfolding** *size-l2-def size-small-l1-def Let-def size-l1-def*
apply *auto*
by (*simp add*: *power-diff*)

lemma *size-l2-i-not0*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*i*>*0*
shows (*size-l2 cfg i*) = (*2*^(*i* + (*sm cfg* − *1*) − (*l cfg*)))
using *a0 a1* **unfolding** *size-l2-def size-small-l1-def Let-def size-l1-def*

apply *auto*
by (*smt Suc-neq-Zero Suc-pred a1 add.commute add-Suc-right*
diff-Suc-Suc leD le-add1 less-le-trans less-or-eq-imp-le
linorder-neqE-nat numeral-2-eq-2 power-add power-diff)

lemma *size-l2-m-not0-i-not0*:
assumes *a0*:*sm cfg* $\geq$ *l cfg* **and** *a1*:*i*>*0* **and**
a2:(*sm cfg*)= (*Suc n*)
shows (*size-l2 cfg i*) = (*2*^(*i* + *n* − (*l cfg*)))
using *size-l2-i-not0*[*OF a0 a1*] *a2* **by** *auto*

lemma *l2-i-j-next*:
l2-i-j cfg i j + (*size-l2 cfg i*) = *l2-i-j cfg i* (*Suc j*)
unfolding *l2-i-j-def* **by** *auto*

lemma *l2-i-j-pow2*:*sm cfg* $\geq$ *l cfg* $\Longrightarrow$ *j* < (*sl cfg*) $\Longrightarrow$
l2-i-j cfg 0 j = ((*2*^(*sm cfg* − (*l cfg*)))∗*j*)
unfolding *l2-i-j-def size-l2-def size-small-l1-def Let-def*
apply *auto*
by (*metis One-nat-def Suc-less-SucD le-zero-eq less-numeral-extra*(*3*)
less-trans-Suc power-diff power-eq-if zero-neq-numeral)

lemma *l2-0-j-sm-0*:
assumes *a0*:*sm cfg* $\geq$ *l cfg* **and**
a1:*j*<(*sl cfg*) **and** *a2*:*sm cfg* = *0*
shows *l2-i-j cfg 0 j* = *0*
using *a0 a1 a2 l2-i-j-def* **by** *auto*

lemma *l2-0-suc-j-sm-0*:
assumes *a0*:*sm cfg* $\geq$ *l cfg* **and**
a1:*j*<(*sl cfg*) **and** *a2*:*sm cfg* = *0*
shows *l2-i-j cfg 0* (*j*+*1*) = *2*
using *a0 a1 a2* **unfolding** *l2-i-j-def size-l2-def*
by (*simp add*: *size-small-l1-def*)

lemma *l2-0-j-sm-not-0*:
assumes *a0*:*sm cfg* $\geq$ *l cfg* **and**
a2:*sm cfg* > *0*
shows *l2-i-j cfg 0 j* = ((*2*^(*sm cfg* − (*l cfg*)))∗*j*)
by (*simp add*: *a0 a2 l2-i-j-def size-l2-m-not-zero0*)

lemma *l2-Suc-i-j*:**assumes** *a0*:*sm cfg* $\geq$ *l cfg*
shows *l2-i-j cfg* (*Suc i*) *j* = *2*^(*Suc i* + (*sm cfg* −*1*)) + ((*2*^(*Suc i* +(*sm cfg*
−*1*) − (*l cfg*)))∗*j*)
using *a0 l2-i-j-def range-l1-i-not-0 size-l2-i-not0 zero-less-Suc* **by** *presburger*

lemma *l2-ij-lt-ij'*:
sm cfg $\geq$ *l cfg* $\Longrightarrow$ *j*<*j'* $\Longrightarrow$ *l2-i-j cfg i j* < *l2-i-j cfg i j'*

using *l2-i-j-next size-l2-not-0*
unfolding *l2-i-j-def* **by** *auto*

lemma *last-l2-i-first-l1-Suc-i*:
assumes *a0*:*sm cfg* ≥ *l cfg*
shows *l2-i-j cfg i* ((*sl cfg*)) = *size-small-l1 cfg* (*Suc i*)
proof(*cases i*)
assume *a00*:*i*=*0*
then show *?thesis* **using** *a0*
unfolding *l2-i-j-def size-small-l1-def size-l1-def size-l2-def Let-def*
apply *auto*
by (*smt One-nat-def Suc-pred div-by-Suc-0 gr0I le-add-diff-inverse2*
le-zero-eq mult.right-neutral power-0 power-Suc power-add power-diff zero-neq-numeral)
next
fix *k*
assume *a00*:*i* = *Suc k*
then show *?thesis*
unfolding *l2-i-j-def*
apply *auto*
unfolding *size-small-l1-def size-l1-def size-l2-def Let-def*
apply *auto* **using** *power-div-neutro*[*OF a0*] **by** *auto*
qed

lemma *last-l2-i-eq-first-l2-Suc-i*:
assumes *a0*:*sm cfg* ≥ *l cfg*
shows *l2-i-j cfg i* (*sl cfg*) = *l2-i-j cfg* (*Suc i*) *0*
using *last-l2-i-first-l1-Suc-i*[*OF a0*]
unfolding *l2-i-j-def* **by** *auto*

lemma *l2-ij-lt-Suci0*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*j* < (*sl cfg*)
shows *l2-i-j cfg i j* < *l2-i-j cfg* (*Suc i*) *0*
using *l2-ij-lt-ij'*[*OF a0 a1*] *last-l2-i-eq-first-l2-Suc-i*[*OF a0*]
by *auto*

lemma *l2-ij-lt-Sucij*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*j* < (*sl cfg*)
shows *l2-i-j cfg i j* < *l2-i-j cfg* (*Suc i*) *j'*
using *l2-ij-lt-ij'*[*OF a0*, *of 0 j Suc i*] *l2-ij-lt-Suci0*[*OF a0 a1*]
by (*metis Nat.add-0-right dual-order.strict-trans1*
l2-i-j-def le-add1 mult-0-right)

lemma *l2-ij-lt-i'j'*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*j* < (*sl cfg*) **and** *a1'*:*j'* < (*sl cfg*) **and** *a2*:*i*<*i'*
shows *l2-i-j cfg i j* < *l2-i-j cfg i' j'*

using *a2* **proof** (*cases i′*, *auto simp add*:*l2-ij-lt-Sucij*[*OF a0 a1*])
fix *n*
assume *a00*:*i′* = *Suc n*
then have *i*< *Suc n* **using** *a2* **by** *auto*
then show *l2-i-j cfg i j* < *l2-i-j cfg* (*Suc n*) *j′*
proof(*induct n*)
case *0*
then show *?case* **using** *a00 l2-ij-lt-Sucij*[*OF a0 a1*] **by** *auto*
next
case (*Suc n*)
then show *?case* **using** *l2-ij-lt-Sucij*[*OF a0*]
by (*metis a1 a1′ less-antisym less-trans*)
qed
qed

definition *range-l2*::*Sys-Config* ⇒ *nat* ⇒ *nat* ⇒ (*nat*×*nat*)
where *range-l2 cfg i j* ≡
(*l2-i-j cfg i j*,
l2-i-j cfg i j + (*size-l2 cfg i*) − *1*)
lemma *range-l2-disj*: *sm cfg* ≥ *l cfg* ⟹
fst (*range-l2 cfg i j*) ≤ *snd* (*range-l2 cfg i j*)
unfolding *range-l2-def* **using** *size-l2-not-0*
by (*metis Nat.add-diff-assoc2 One-nat-def Suc-leI add.commute fst-conv gr0I le-add2 snd-conv*)

lemma *snd-range-l2-i-j-less-fst-j′*:
sm cfg ≥ *l cfg* ⟹ *j* < *j′* ⟹
snd(*range-l2 cfg n j*) < *fst* (*range-l2 cfg n j′*)
unfolding *range-l2-def*
by (*metis One-nat-def Suc-lessI Suc-pred add-gr-0 fst-conv lessI less-add-same-cancel1*

less-imp-diff-less snd-conv l2-ij-lt-ij′ l2-i-j-next)

lemma *snd-range-l2-i-j-less-fst-i′-j′*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and** *a1*:*j*< *sl cfg* **and** *a1′*:*j′*< *sl cfg* **and** *a2*:*i* < *i′*
shows *snd*(*range-l2 cfg i j*) < *fst* (*range-l2 cfg i′ j′*)
proof(*auto simp add*: *range-l2-def*)
have *l2-i-j cfg i j* + *size-l2 cfg i* = *l2-i-j cfg i* (*Suc j*)
by (*simp add*: *l2-i-j-next*)
then have *l2-i-j cfg i j* + *size-l2 cfg i* − *Suc 0* < *l2-i-j cfg i* (*Suc j*)
by (*metis a0 diff-Suc-less gr-zeroI size-l2-not-0 zero-eq-add-iff-both-eq-0*)
also have *l2-i-j cfg i* (*Suc j*)≤ *l2-i-j cfg* (*Suc i*) *0*
by (*metis Suc-lessI a0 a1 less-or-eq-imp-le l2-ij-lt-ij′ last-l2-i-eq-first-l2-Suc-i*)
also have *l2-i-j cfg* (*Suc i*) *0* ≤ *l2-i-j cfg i′ j′*
apply (*cases Suc i* = *i′*)
apply (*simp add*: *a0 l2-ij-lt-ij′ less-mono-imp-le-mono*)
by (*metis Suc-lessI a0 a1′ a2 l2-ij-lt-i′j′ le-add1 le-add-same-cancel1 less-imp-le-nat less-trans order.strict-iff-order*)

```
  finally show  l2-i-j cfg i j + size-l2 cfg i − Suc 0 < l2-i-j cfg i' j'
    by auto
qed


lemma snd-range-l2-i-j-less-snd-i'-j':
  assumes a0:sm cfg ≥ l cfg and a1:i < i' and a2:j< sl cfg and a3:j'< sl cfg
  shows snd( range-l2 cfg i j) < snd (range-l2 cfg i' j')
  using snd-range-l2-i-j-less-fst-i'-j'[OF a0 a2 a3 a1] range-l2-disj[OF a0]
  using less-le-trans by blast

lemma fst-range-l2-i-j-less-snd-i'-j':
  assumes a0:sm cfg ≥ l cfg and a1:i < i' and a2:j< sl cfg and a3:j'< sl cfg
  shows fst( range-l2 cfg i j) < snd (range-l2 cfg i' j')
  using snd-range-l2-i-j-less-fst-i'-j'[OF a0 a2 a3 a1 ] range-l2-disj[OF a0]
  using le-less-trans less-le-trans by metis


lemma fst-range-l2-i-j-less-fst-i'-j':
  assumes a0:sm cfg ≥ l cfg and a1:i < i' and a2:j< sl cfg  and a3:j'< sl cfg
  shows fst( range-l2 cfg i j) < fst (range-l2 cfg i' j')
  using snd-range-l2-i-j-less-fst-i'-j'[OF a0 a2 a3 a1] range-l2-disj[OF a0]
  using dual-order.strict-trans2 by blast

lemma range-l2-in-l1:
  assumes a0:sm cfg ≥ l cfg and
          a1:j < sl cfg
        shows fst (range-l2 cfg i j) ≥ fst (range-l1 cfg i) ∧
              snd (range-l2 cfg i j) ≤ snd (range-l1 cfg i)
proof −
  have snd (range-l2 cfg i j) ≤ snd (range-l1 cfg i)
  proof(simp add: range-l2-def)
    have l2-i-j cfg i j + size-l2 cfg i = l2-i-j cfg i (Suc j)
      by (simp add: l2-i-j-next)
    moreover have l2-i-j cfg i (Suc j)≤ l2-i-j cfg i (sl cfg)
      using a0 a1 l2-ij-lt-ij' less-mono-imp-le-mono by auto
    moreover have l2-i-j cfg i (sl cfg) = fst (range-l1 cfg (Suc i))
      using last-l2-i-first-l1-Suc-i[OF a0]
      by auto
    ultimately show l2-i-j cfg i j + size-l2 cfg i − Suc 0
    ≤ snd (range-l1 cfg i)
      using One-nat-def le-diff-conv snd-rang-i-eq-fst-rang-suc-i
      by presburger
  qed
  moreover have fst (range-l2 cfg i j) ≥ fst (range-l1 cfg i)
    by (auto simp add: l2-i-j-def range-l2-def)
  ultimately show ?thesis by auto
qed
```

definition *l1-set*::*Sys-Config* ⇒ *nat* ⇒ *nat set*
where *l1-set cfg i* ≡
let r = *range-l1 cfg i in*
{*m*. *m*≥ *fst r* ∧ *m*≤ *snd r*}

lemma *l1-set-disj*: *i*≠*i'* ⟶ (*l1-set cfg i* ∩ *l1-set cfg i'*) = {}
unfolding *l1-set-def Let-def* **apply** *auto*
by (*meson dual-order.trans leD less-linear range-l1-snd-i-lt-fst-i'*)

definition *l2-set*::*Sys-Config* ⇒ *nat* ⇒ *nat* ⇒ *nat set*
where *l2-set cfg i j* ≡
let r = *range-l2 cfg i j in*
{*m*. *m*≥ *fst r* ∧ *m*≤ *snd r*}

abbreviation *r-gt-sm-0-i*::*nat* ⇒ *nat*
where *r-gt-sm-0-i r* ≡ (*floorlog 2 r* − *1*)

abbreviation *r-lt-sm-gt-0-j*::*Sys-Config* ⇒ *nat* ⇒ *nat*
where *r-lt-sm-gt-0-j cfg r* ≡ (*r div 2* ^ (*sm cfg* − *l cfg*))

abbreviation *r-gt-sm-gt-0-i*::*Sys-Config* ⇒ *nat* ⇒ *nat*
where *r-gt-sm-gt-0-i cfg r* ≡ (*Suc* ((*floorlog 2 r*) − *1* − (*sm cfg*)))

abbreviation *r-gt-sm-gt-0-j*::*Sys-Config* ⇒ *nat* ⇒ *nat*
where *r-gt-sm-gt-0-j cfg r* ≡ ((*r* − (*fst* (*range-l1 cfg* (*Suc* ((*floorlog 2 r*) − *1* − (*sm cfg*)))))) *div* (*size-l2 cfg* (*Suc* ((*floorlog 2 r*) − *1* − (*sm cfg*)))))

lemma *set-l2-in-l1*:
assumes *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*j* < *sl cfg*
shows *l2-set cfg i j* ⊆ *l1-set cfg i*
unfolding *l2-set-def l1-set-def Let-def*
by (*auto*;*meson a0 a1 dual-order.strict-trans1 leD le-less-linear range-l2-in-l1*)

lemma *l2-set-disj*:*sm cfg* ≥ *l cfg* ∧ *j*<(*sl cfg*) ∧ *j1*<(*sl cfg*) ∧ ¬(*i*=*i1* ∧ *j*=*j1*)
⟶
(*l2-set cfg i j* ∩ *l2-set cfg i1 j1*) = {}
unfolding *l2-set-def Let-def* **apply** (*cases j*=*j1*; *cases i*=*i1*; *auto*)
apply (*cases i*>*i1*, *auto*)
apply (*metis leD less-le-trans sl-def snd-range-l2-i-j-less-fst-i'-j'*)
apply (*metis dual-order.trans linorder-neqE-nat not-le sl-def snd-range-l2-i-j-less-fst-i'-j'*)
apply (*cases j*>*j1*, *auto*)
apply (*meson leD less-le-trans snd-range-l2-i-j-less-fst-j'*)
apply (*meson dual-order.trans leD linorder-neqE-nat snd-range-l2-i-j-less-fst-j'*)
by (*cases i*>*i1*;*cases j*>*j1*, *auto*; *metis leD le-less less-le-trans linorder-neqE-nat sl-def snd-range-l2-i-j-less-fst-i'-j'*)

lemma *j-i-sm-0*:$r \leq 2 \hat{} sm\ cfg - Suc\ 0 \Longrightarrow$
$i = 0 \Longrightarrow$
$l\ cfg \leq sm\ cfg \Longrightarrow$
$0 < sm\ cfg \Longrightarrow$
$j = r\ div\ 2 \hat{} (sm\ cfg - l\ cfg) \Longrightarrow$
$j<2 \hat{} l\ cfg \wedge$
$2 \hat{} (sm\ cfg - l\ cfg) * j \leq r \wedge$
$r \leq 2 \hat{} (sm\ cfg - l\ cfg) * j + 2 * 2 \hat{} (sm\ cfg - Suc\ 0)\ div\ 2 \hat{} l\ cfg - Suc\ 0$
apply *auto*
apply (*metis Suc-pred le-add-diff-inverse le-imp-less-Suc less-mult-imp-div-less power-add zero-less-numeral zero-less-power*)
by (*metis One-nat-def Suc-neq-Zero Suc-pred add.commute add-gr-0 dividend-less-times-div less-Suc-eq-le*
mult.commute numeral-2-eq-2 power-diff power-minus-mult zero-less-Suc zero-less-power)

lemma *div-e*:$(D::nat)\neq 0 \Longrightarrow E\neq 0 \Longrightarrow A\leq B \Longrightarrow (B - C)\ div\ E < D \Longrightarrow (A - C)\ div\ E < D$
using *diff-le-mono div-le-mono le-less-trans* **by** *blast*

lemma *div-e'*:$(D::nat)\neq 0 \Longrightarrow E\neq 0 \Longrightarrow A<B \Longrightarrow (B - C)\ div\ E < D \Longrightarrow (A - C)\ div\ E < D$
using *div-e*
by (*meson less-imp-le-nat*)

lemma *r-small*: $r \leq 2 * size\text{-}l1\ i * 2 \hat{} (sm\ cfg - Suc\ 0) - Suc\ 0 \Longrightarrow r < 2 * size\text{-}l1\ i * 2 \hat{} (sm\ cfg - Suc\ 0)$
unfolding *size-l1-def*
apply (*cases i*, *auto*)
apply (*metis Suc-pred le-imp-less-Suc mult-pos-pos zero-less-numeral zero-less-power*)
by (*metis Suc-le-lessD Suc-le-mono Suc-pred mult-pos-pos zero-less-numeral zero-less-power*)

lemma *B-dvd-A-imp-A-less-1-div-B-less-A-div-B*:
$A\neq 0 \Longrightarrow (B::nat)\ dvd\ A \Longrightarrow (A-1)\ div\ B < A\ div\ B$
by (*simp add*: *less-mult-imp-div-less*)

lemma *j-less-2-power-l*:
assumes
a0:$0 < sm\ cfg$ **and**
a1:$l\ cfg \leq sm\ cfg$ **and**
a2:$(2::nat) * 2 \hat{} i * 2 \hat{} (sm\ cfg - Suc\ 0) \leq r$ **and**
a3:$r \leq 4 * 2 \hat{} i * 2 \hat{} (sm\ cfg - Suc\ 0) - Suc\ 0$
shows $(r - 2 * 2 \hat{} i * 2 \hat{} (sm\ cfg - Suc\ 0))\ div$
$(2 * 2 \hat{} i * 2 \hat{} (sm\ cfg - Suc\ 0)\ div\ 2 \hat{} l\ cfg)$
$< 2 \hat{} l\ cfg$
proof−
have $((2::nat) * 2 \hat{} i * 2 \hat{} (sm\ cfg - Suc\ 0)\ div\ 2 \hat{} l\ cfg)\neq 0$

```
    using a0 a1 apply (cases i, cases sm cfg, auto)
    apply (metis (no-types) power2gt power-Suc)
   by (smt One-nat-def Suc-pred div-2-gt-zero div-mult-mult1 gr0I le-neq-implies-less
less-Suc-eq-le mult.commute mult-eq-1-iff mult-pos-pos nat-one-le-power numeral-eq-iff
numerals(1) one-le-mult-iff one-le-numeral power2gt1 power-minus-mult power-not-zero
semiring-norm(85) zero-less-numeral)
  moreover have (2::nat) ^ l cfg ≠0 by auto
  moreover have (((4::nat) * 2 ^ i * 2 ^ (sm cfg - Suc 0) - Suc 0) - 2 * 2 ^
i * 2 ^ (sm cfg - Suc 0)) div
    (2 * 2 ^ i * 2 ^ (sm cfg - Suc 0) div 2 ^ l cfg)
    < 2 ^ l cfg using a0 a1
  proof (cases i; cases sm cfg;  auto)
    fix x
    assume i = 0 and sm cfg = Suc x and l cfg ≤ Suc x
    then show (2 * 2 ^ x - Suc 0) div (2 * 2 ^ x div 2 ^ l cfg) < 2 ^ l cfg
     by (metis (full-types) Suc-neq-Zero a0 j-i-sm-0 less-or-eq-imp-le numeral-2-eq-2
power-Suc power-diff)
  next
    fix i' sm'
    assume a1:i = Suc i' and a2:sm cfg = Suc sm' and a3:l cfg ≤ Suc sm'
    moreover have ((4::nat) * 2 ^ i' * 2 ^ sm' div 2 ^ l cfg) dvd 4 * (2 ^ i' * 2
^ sm')
      using calculation apply (cases l cfg, auto)
      by (simp add: div-dvd-iff-mult le-imp-power-dvd)
    moreover have ((4::nat) * (2 ^ i' * 2 ^ sm')) div (4 * 2 ^ i' * 2 ^ sm' div 2
^ l cfg) = 2 ^ l cfg
      using calculation apply (subst div-div-eq-right, auto)
       by (metis (no-types, lifting) a0 add-diff-cancel-left' dvd-mult dvd-triv-right
even-numeral
        le-Suc-eq le-imp-power-dvd mult.assoc mult-dvd-mono plus-1-eq-Suc
        power-commutes power-minus-mult)
    moreover have (4::nat) * (2 ^ i' * 2 ^ sm') ≠0
      by auto
    ultimately show (4 * (2 ^ i' * 2 ^ sm') - Suc 0) div (4 * 2 ^ i' * 2 ^ sm'
div 2 ^ l cfg) < 2 ^ l cfg
      using B-dvd-A-imp-A-less-1-div-B-less-A-div-B
      by (metis One-nat-def)
  qed
  ultimately show ?thesis using div-e
    using a3 by blast
qed

lemma l1:4 * 2 ^ ia * 2 ^ sma ≤ r ⟹
    r ≤ 4 * 2 ^ (ia + sma) + 4 * 2 ^ (ia + sma) * ((r - 4 * 2 ^ ia * 2 ^ sma)
div (4 * 2 ^ ia * 2 ^ sma)) +
        4 * 2 ^ ia * 2 ^ sma -
        Suc 0
proof -
  fix nat :: nat and nata :: nat
```

```
  assume a1: 4 * 2 ^ nat * 2 ^ nata ≤ r
  have (0::nat) < 4 * 2 ^ nat * 2 ^ nata
    by simp
  then have f2: r < 4 * 2 ^ nat * 2 ^ nata + 4 * 2 ^ nat * 2 ^ nata * (r div (4
* 2 ^ nat * 2 ^ nata))
    by (meson dividend-less-times-div)
  have f3: 4 * (2::nat) ^ (nat + nata) = 4 * 2 ^ nat * 2 ^ nata
    by (simp add: power-add)
  have f4: 4 * 2 ^ nat * 2 ^ nata * Suc 0 = 4 * 2 ^ nat * 2 ^ nata
    by linarith
  have 4 * 2 ^ nat * 2 ^ nata * (r div (4 * 2 ^ nat * 2 ^ nata)) = 4 * 2 ^ nat *
2 ^ nata * Suc 0 + 4 * 2 ^ (nat + nata) * ((r - 4 * 2 ^ nat * 2 ^ nata) div (4
* 2 ^ nat * 2 ^ nata))
    using f3 a1 by (simp add: div-if)
  then have r < 4 * 2 ^ (nat + nata) + (4 * 2 ^ (nat + nata) * ((r - 4 * 2 ^
nat * 2 ^ nata) div (4 * 2 ^ nat * 2 ^ nata)) + 4 * 2 ^ nat * 2 ^ nata)
    using f4 f3 f2 by presburger
  then show r ≤ 4 * 2 ^ (nat + nata) + 4 * 2 ^ (nat + nata) * ((r - 4 * 2 ^
nat * 2 ^ nata) div (4 * 2 ^ nat * 2 ^ nata)) + 4 * 2 ^ nat * 2 ^ nata - Suc 0
    by linarith
qed

lemma l2:assumes
  a0:la ≤ sma
showsr ≤ 2 * 2 ^ sma +
        (2 ^ sma div 2 ^ la +
         2 ^ sma div 2 ^ la * ((r - 2 * 2 ^ sma) div (2 ^ sma div 2 ^ la))) -
        Suc 0
proof -
  have (r - 2 * 2 ^ sma) <  2 ^ sma div 2 ^ la +
         2 ^ sma div 2 ^ la * ((r - 2 * 2 ^ sma) div (2 ^ sma div 2 ^ la))
    by (meson  a0 power2gt dividend-less-times-div)
  then show r ≤ 2 * 2 ^ sma +
        (2 ^ sma div 2 ^ la +
         2 ^ sma div 2 ^ la * ((r - 2 * 2 ^ sma) div (2 ^ sma div 2 ^ la))) -
        Suc 0
    by linarith
qed

lemma l3:
  assumes a0: la ≤ sma and
          a1:4 * 2 ^ ia * 2 ^ sma ≤ r and
    a2:r ≤ 8 * 2 ^ ia * 2 ^ sma - Suc 0
  shows r ≤ 4 * (2 ^ ia * 2 ^ sma) +
        (2 * (2 ^ ia * 2 ^ sma) div 2 ^ la +
         2 * (2 ^ ia * 2 ^ sma) div 2 ^ la *
         ((r - 4 * 2 ^ ia * 2 ^ sma) div (2 * (2 ^ ia * 2 ^ sma) div 2 ^ la))) -
        Suc 0
proof-
```

```
  have 0<(2::nat) * (2 ^ ia * 2 ^ sma) div 2 ^ la
    using a0 by (auto intro: power2gt1 simp: power-add[THEN sym])
  then have (r - 4 * 2 ^ ia * 2 ^ sma) <
             (2 * (2 ^ ia * 2 ^ sma) div 2 ^ la + 2 * (2 ^ ia * 2 ^ sma) div 2 ^ la *
              ((r - 4 * 2 ^ ia * 2 ^ sma) div (2 * (2 ^ ia * 2 ^ sma) div 2 ^ la)))
    by (auto simp add: dividend-less-times-div)
  thus ?thesis by linarith
qed


lemma j-sm-gt-0: 0 < sm cfg ⟹
    range-l1 cfg (Suc i) =
    (size-l1 (Suc i) * (2::nat) ^ (sm cfg - Suc 0), 2 * size-l1 (Suc i) * 2 ^ (sm
cfg - Suc 0) - Suc 0) ⟹
    l cfg ≤ sm cfg ⟹
    size-l1 (Suc i) * 2 ^ (sm cfg - Suc 0) ≤ r ⟹
    r ≤ 2 * size-l1 (Suc i) * 2 ^ (sm cfg - Suc 0) - Suc 0 ⟹
    j = (r - (fst (range-l1 cfg (Suc i)))) div (size-l2 cfg (Suc i)) ⟹
    j<2 ^ l cfg ∧ l2-i-j cfg (Suc i) j ≤ r ∧ r ≤ l2-i-j cfg (Suc i) j + size-l2 cfg
(Suc i) - Suc 0
  apply (subst l2-Suc-i-j) apply auto[1]
  apply (auto simp add: size-l2-def size-l1-def j-less-2-power-l)
   apply (cases i; cases sm cfg; cases l cfg, auto)
  apply (auto simp add: add-leE le-add2 plus-1-eq-Suc  power-add power-diff  )
   apply (metis One-nat-def le-add-diff-inverse nat-add-left-cancel-le plus-1-eq-Suc

              power-Suc0-right power-add  times-div-less-eq-dividend )
   apply (metis One-nat-def le-add-diff-inverse nat-add-left-cancel-le plus-1-eq-Suc

              power-Suc0-right power-add  times-div-less-eq-dividend )
  apply (metis le-add-diff-inverse mult.assoc nat-add-left-cancel-le  times-div-less-eq-dividend)
  apply (smt One-nat-def add-leE le-add2 le-add-diff-inverse mult.assoc nat-add-left-cancel-le
plus-1-eq-Suc power-Suc0-right power-add power-diff times-div-less-eq-dividend zero-neq-numeral)
  apply (subst l2-Suc-i-j) apply auto[1]
  apply (cases i; cases sm cfg;cases l cfg; auto)
    apply (auto simp add: add-leE le-add2 plus-1-eq-Suc    power-add power-diff
Groups.add-ac(3)  add.commute  dividend-less-times-div leD  not-less-eq-eq power2gt)
  subgoal for sma la by (auto intro: l2)
  subgoal for ia sma la by (auto intro: l3)
  done

lemma set-l1-in-l2-sm-0-i-0:
  assumes a0:sm cfg ≥ l cfg and
          a1:r∈l1-set cfg 0 and a2:sm cfg = 0
        shows 0< sl cfg ∧ r ∈ l2-set cfg 0 0
  unfolding l2-set-def Let-def range-l2-def
proof(auto)
  have range-l1:range-l1 cfg 0 = (0, 2^(Suc ((sm cfg -1))) -1)
     using a0 a1
```

```
    by (auto simp add: size-small-l1-def)
  moreover have r ≥ 0 ∧ r ≤ 2^(Suc ((sm cfg −1))) −1
    using a1 calculation unfolding l1-set-def Let-def
    by (auto)
  ultimately have
     l2-i-j cfg 0 0 ≤ r ∧
              r ≤ l2-i-j cfg 0 0 + size-l2 cfg 0 − Suc 0
    using a0 a2 range-l1 unfolding size-l2-def size-small-l1-def
    by ( simp add: l2-i-j-def)
  then
  show l2-i-j cfg 0 0 ≤ r and
      r ≤ l2-i-j cfg 0 0 + size-l2 cfg 0 − Suc 0 by auto
qed

lemma set-l1-in-l2-sm-gt-0-i-0:
  assumes a0:sm cfg ≥ l cfg and
       a1:r∈l1-set cfg 0 and a2:sm cfg > 0
      shows r div 2 ^ (sm cfg − l cfg) < sl cfg ∧ r ∈ l2-set cfg 0 (r div 2 ^ (sm
cfg − l cfg))
  unfolding l2-set-def Let-def range-l2-def
proof(auto)
  have range-l1:range-l1 cfg 0 = (0, 2^(Suc ((sm cfg −1))) −1)
    using a0 a1
    by (auto simp add: size-small-l1-def)
  moreover have r ≥ 0 ∧ r ≤ 2^(Suc ((sm cfg −1))) −1
    using a1 calculation unfolding l1-set-def Let-def
    by (auto)
  ultimately show
      r div 2 ^ (sm cfg − l cfg)<2 ^ l cfg and
     l2-i-j cfg 0 ( r div 2 ^ (sm cfg − l cfg)) ≤ r and
     r ≤ l2-i-j cfg 0 ( r div 2 ^ (sm cfg − l cfg)) + size-l2 cfg 0 − Suc 0
    using a0 a2 range-l1 unfolding size-l2-def size-small-l1-def
    apply (auto simp add: l2-0-j-sm-not-0)
    using j-i-sm-0
    by blast+
qed

lemma set-l1-in-l2-sm-0-i-gt-0:
  assumes a0:sm cfg ≥ l cfg and
       a1:r∈l1-set cfg (Suc i) and a2:sm cfg = 0
      shows 0 < sl cfg ∧ r ∈ l2-set cfg (Suc i) 0
  unfolding l2-set-def Let-def range-l2-def
proof(auto)
   have range-l1:range-l1 cfg (Suc i) = (size-l1 (Suc i) * 2^((sm cfg −1)),
(2*size-l1 (Suc i) * 2^(sm cfg −1)) − 1)
    using a0 a1 by (cases i, auto simp add: size-small-l1-def)
  moreover have r:r ≥ size-l1 (Suc i) * 2^((sm cfg −1)) ∧
              r ≤ (2*size-l1 (Suc i) * 2^(sm cfg −1)) − 1
    using a1 a2 calculation unfolding l1-set-def Let-def
```

```
    by auto
  then show
      l2-i-j cfg (Suc i) 0 ≤ r and
      r ≤ l2-i-j cfg (Suc i) 0 + size-l2 cfg (Suc i) − Suc 0
    using  range-l1 a0
     apply (auto simp add: l2-i-j-def size-l1-def) using a2
     by (simp add: size-l2-i-not0)
qed

lemma set-l1-in-l2-sm-gt-0-i-gt-0:
  assumes a0:sm cfg ≥ l cfg and
       a1:r∈l1-set cfg (Suc i) and a2:sm cfg > 0
      shows (r − (fst (range-l1 cfg (Suc i)))) div (size-l2 cfg (Suc i)) < sl cfg ∧
            r ∈ l2-set cfg (Suc i) ((r − (fst (range-l1 cfg (Suc i)))) div (size-l2
cfg (Suc i)))
  unfolding l2-set-def Let-def range-l2-def
proof(clarsimp)
    have range-l1:range-l1 cfg (Suc i) = (size-l1 (Suc i) * 2^((sm cfg −1)),
(2*size-l1 (Suc i) * 2^(sm cfg −1)) − 1)
     using a0 a1 by (cases i, auto simp add: size-small-l1-def)
   moreover have r:r ≥ size-l1 (Suc i) * 2^((sm cfg −1)) ∧
              r ≤ (2*size-l1 (Suc i) * 2^(sm cfg −1)) − 1
     using a1 a2 calculation unfolding l1-set-def Let-def
     by auto
   then show
      (r − size-small-l1 cfg (Suc i)) div size-l2 cfg (Suc i) < 2 ^ l cfg ∧
      l2-i-j cfg (Suc i) ((r − size-small-l1 cfg (Suc i)) div size-l2 cfg (Suc i)) ≤
r ∧
       r ≤ l2-i-j cfg (Suc i) ((r − size-small-l1 cfg (Suc i)) div size-l2 cfg (Suc
i)) +
      size-l2 cfg (Suc i) − Suc 0
     using range-l1 a0 a2
     apply clarsimp apply (frule j-sm-gt-0) by auto
  qed

lemma set-l1-in-l2:
  assumes a0:sm cfg ≥ l cfg and
       a1:r∈l1-set cfg i
      shows ∃j< sl cfg. r ∈ l2-set cfg i j
proof(cases i; cases sm cfg)
  assume a00:i=0 and sm:sm cfg = 0
  then show ?thesis
    using a1 set-l1-in-l2-sm-0-i-0[OF a0 ] by force
next
  fix sma
   assume a00:i=0 and sm:sm cfg = Suc sma
   then show ?thesis
   using a1 set-l1-in-l2-sm-gt-0-i-0[OF a0 ] by force
next
```

fix *ia*
assume *a00*:*i*=*Suc ia* **and** *sm*:*sm cfg* = *0*
then show *?thesis*
using *a1 set-l1-in-l2-sm-0-i-gt-0*[*OF a0*] **by** *force*
next
fix *ia sma*
assume *a00*:*i*=*Suc ia* **and** *sm*:*sm cfg* = *Suc sma*
then show *?thesis*
using *a1 set-l1-in-l2-sm-gt-0-i-gt-0*[*OF a0*] **by** *force*
qed

definition *tlsf-matrix*::*Sys-Config* ⇒ *bhdr-matrix-t* ⇒ *bool*
where *tlsf-matrix cfg t* ≡
∀ *i j*. *j*<(*sl cfg*) ⟶
(∀ *b*. *b* ∈ *t i j* ⟶ (*b-size b*) ∈ *l2-set cfg i j*)

lemma *block-in-range0*:**assumes** *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*tlsf-matrix cfg t* **and**
a2:*b* ∈ *t 0 j* **and** *a3*:*j*<(*sl cfg*) **and** *a4*:*sm cfg* = *0*
shows *b-size b* ≥ *0* ∧ (*b-size b* ≤ *2* − *1*)
proof−
have (*b-size b*) ≥ *l2-i-j cfg 0 j* **and** (*b-size b*) ≤ (*l2-i-j cfg 0* (*Suc j*) −*1*)
using *a0 a2 a1 a3* **unfolding** *tlsf-matrix-def l2-set-def range-l2-def*
apply *auto*
by (*metis l2-i-j-next less-imp-le-nat*)
then show *?thesis*
using *a0 a3 a4 l2-0-suc-j-sm-0* **by** *auto*
qed

lemma *block-in-range1*:**assumes** *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*tlsf-matrix cfg t* **and**
a2:*b* ∈ *t 0 j* **and** *a3*:*j*<(*sl cfg*) **and** *a4*:*sm cfg* > *0*
shows *b-size b* ≥ ((*2*^(*sm cfg* − (*l cfg*)))∗*j*) ∧ (*b-size b* ≤ ((*2*^(*sm cfg* − (*l cfg*)))∗(*Suc j*)) − *1*)
proof−
have (*b-size b*) ≥ *l2-i-j cfg 0 j* **and** (*b-size b*) ≤ (*l2-i-j cfg 0* (*Suc j*) −*1*)
using *a0 a2 a1 a3* **unfolding** *tlsf-matrix-def l2-set-def range-l2-def*
apply *auto*
by (*metis l2-i-j-next less-imp-le-nat*)
then show *?thesis*
by (*simp add*: *a0 a4 l2-0-j-sm-not-0*)
qed

lemma *block-in-range0'*:**assumes** *a0*:*sm cfg* ≥ *l cfg* **and**
a1:*tlsf-matrix cfg t* **and**
a2:*b* ∈ *t* (*Suc i*) *j* **and** *a3*:*j*<(*sl cfg*)
shows *b-size b* ≥ *2*^(*Suc i* + (*sm cfg* −*1*)) + ((*2*^(*Suc i*+(*sm cfg* −*1*) − (*l*

```
cfg)))*j) ∧
        ( b-size b ≤ 2^(Suc i + (sm cfg −1)) + ((2^(Suc i+(sm cfg − 1) − (l
cfg)))*(j+1)) − 1)
proof−
  have (b-size b) ≥ l2-i-j cfg (Suc i) j and (b-size b) ≤ (l2-i-j cfg (Suc i) (Suc j)
−1)
    using a0 a2 a1 a3 unfolding tlsf-matrix-def l2-set-def range-l2-def
    apply auto
    by (metis  l2-i-j-next  less-imp-le-nat)
  then show ?thesis
    by (simp add: a0 l2-Suc-i-j)
qed

lemma r-sm-gt-0-r-lt-2-sm:
    assumes a0:sm cfg ≥ l cfg and
          a1:sm cfg > 0
    shows r ∈ l2-set cfg 0 (r div (2^((sm cfg ) − (l cfg))))
  unfolding l2-set-def Let-def range-l2-def
proof(auto)
  show l2-i-j cfg 0 (r div 2 ^ (sm cfg − l cfg)) ≤ r
  proof−
    have l2-i-j cfg 0 (r div 2 ^ (sm cfg − l cfg)) =
       ((2^(sm cfg − (l cfg)))*(r div 2 ^ (sm cfg − l cfg)))
      using l2-0-j-sm-not-0[OF a0 a1] by auto
    thus ?thesis by auto
  qed
next
  show r ≤ l2-i-j cfg 0 (r div 2 ^ (sm cfg − l cfg)) +
        size-l2 cfg 0 −
        Suc 0
    using l2-0-j-sm-not-0[OF a0 a1]
    by (metis Suc-pred a0 a1 add.commute
            add-gr-0 dividend-less-times-div less-Suc-eq-le size-l2-m-not-zero0
            zero-less-numeral zero-less-power)
qed

lemma r-l11:
  assumes
     a0:sm cfg ≥ l cfg and
     a1:sm cfg > 0 and
     a2:x=(floorlog 2 r) − 1 and
     a3:x≥sm cfg and a4:r≥ 2^(sm cfg)
   shows ∃x'. x = sm cfg + x' ∧
         r ≥ size-small-l1 cfg (Suc x') ∧
         r < size-small-l1 cfg (Suc (Suc x'))
proof−
  have F:¬ r< 2^(sm cfg) using a4 by auto
  have rgt2:r≥2
  proof(cases sm cfg)
```

```
    case 0
    then show ?thesis using a1 by auto
  next
    case (Suc nat)
    then show ?thesis using a1
      by (metis F le-less-linear less-2-cases nat-less-le
          nat-one-le-power nat-power-eq-Suc-0-iff zero-less-power)
  qed
  have 2^(floorlog 2 r - 1) ≤ r and rtop:r < 2^(floorlog 2 r)
    using floorlog-bounds rgt2 by force+
  moreover obtain x' where x':x=(sm cfg) + x' using a3
    using le-Suc-ex by auto
  moreover have size-small-l1 cfg (Suc x') ≤ r using a2 a4 calculation
    unfolding size-small-l1-def Let-def
    apply auto
     by (metis One-nat-def a1 add-Suc-right add-eq-if mult.commute neq0-conv
power-add size-l1-def)
  moreover have r < size-small-l1 cfg (Suc (Suc x')) using rtop a2 a4 a1 x'
    unfolding size-small-l1-def Let-def
    apply auto
   by (metis Suc-pred add.commute add-Suc-right add-gr-0 le0 less-nat-zero-code

        nat-less-le power-add size-l1-def zero-less-diff)
  ultimately show ?thesis by auto
qed

lemma r-l1: assumes a0:sm cfg ≥ l cfg and
          a1:sm cfg > 0 and
          a2:r≥ 2^(sm cfg)
    shows ∃ x x'. x = sm cfg + x' ∧ r ∈ l1-set cfg (Suc x')
proof−
  have F:¬ r< 2^(sm cfg) using a2 by auto
  have rgt2:r≥2
  proof(cases sm cfg)
    case 0
    then show ?thesis using a1 by auto
  next
    case (Suc nat)
    then show ?thesis using a1
      by (metis F le-less-linear less-2-cases nat-less-le
          nat-one-le-power nat-power-eq-Suc-0-iff zero-less-power)
  qed
  then obtain x where x:x=(floorlog 2 r) - 1
    by simp
  then have xgt0:x>0 using rgt2 unfolding floorlog-def by auto
  have x-gt-sm:x≥sm cfg using a2 x rgt2 unfolding floorlog-def
    by (simp add: le-log2-of-power le-nat-floor)
  obtain x' where suc:x = Suc x'
    using gr0-implies-Suc xgt0 by auto
```

```
  moreover have 2^(floorlog 2 r − 1) ≤ r and rtop:r < 2^(floorlog 2 r)
    using  floorlog-bounds rgt2 by force+
  ultimately show ?thesis using r-l11[OF a0 a1 x x-gt-sm a2]
    unfolding l1-set-def  Let-def apply auto
    by (smt One-nat-def Suc-eq-plus1 Suc-leI Suc-neq-Zero add-le-imp-le-diff
          mult.commute mult.left-commute numeral-2-eq-2 power-Suc2
         size-l1-def size-small-l1-def)
qed



lemma i-index-r-sm-gt-0: assumes a0:sm cfg ≥ l cfg and
             a1:sm cfg > 0 and
             a2:r≥ 2^(sm cfg)
      shows r ∈ l1-set cfg (Suc ((floorlog 2 r) − 1 − (sm cfg)))
proof−
  have F:¬ r< 2^(sm cfg) using a2 by auto
  have rgt2:r≥2
  proof(cases sm cfg)
    case 0
    then show ?thesis using a1 by auto
  next
    case (Suc nat)
    then show ?thesis using a1
      by (metis F le-less-linear less-2-cases nat-less-le
           nat-one-le-power nat-power-eq-Suc-0-iff zero-less-power)
  qed
  then obtain x where x:x=(floorlog 2 r) − 1
    by simp
  then have xgt0:x>0 using rgt2 unfolding floorlog-def by auto
  have x-gt-sm:x≥sm cfg using a2 x rgt2 unfolding floorlog-def
    by (simp add: le-log2-of-power le-nat-floor)
  obtain x' where suc:x = Suc x'
    using gr0-implies-Suc xgt0 by auto
  moreover have 2^(floorlog 2 r − 1) ≤ r and rtop:r < 2^(floorlog 2 r)
    using  floorlog-bounds rgt2 by force+
  ultimately show ?thesis using r-l11[OF a0 a1 x x-gt-sm a2] x a0
    unfolding l1-set-def  Let-def  apply auto
    by (simp add: size-l1-def size-small-l1-def)
qed

lemma i-index-r-sm-gt-0-r-lt-2: assumes a0:sm cfg ≥ l cfg and
             a1:sm cfg > 0 and
             a2:r< 2^(sm cfg)
           shows r ∈ l1-set cfg 0
  using a0 a1 a2 unfolding l1-set-def Let-def
proof −
  have f3: 0 < Suc (Suc 0) ^ sm cfg
    by (metis zero-less-Suc zero-less-power)
```

```
  have f1:sm cfg ≠ 0
    using a2 a1 by (metis gr-implies-not-zero)
  then show r ∈ {m. fst (range-l1 cfg 0) ≤ m ∧ m ≤ snd (range-l1 cfg 0)}
    apply (auto simp add: size-small-l1-def)
   by (metis One-nat-def Suc-pred f1 a2 f3 less-Suc-eq-le numeral-2-eq-2 power-eq-if)
qed

lemma i-index-r-sm-eq-0: assumes a0:sm cfg ≥ l cfg and
              a1:sm cfg = 0 and
              a2:r≥ 2^(sm cfg)+1
      shows r ∈ l1-set cfg ( ((floorlog 2 r) − 1 ))
proof−
  have F:¬ r< 2^(sm cfg) using a2 by auto
  have rgt2:r≥2 using a1 a2  by simp
  then obtain x where x:x=(floorlog 2 r) − 1
    by simp
  then have xgt0:x>0 using rgt2 unfolding floorlog-def by auto
  have x-gt-sm:x≥sm cfg using a2 x rgt2 unfolding floorlog-def
    by (simp add: le-log2-of-power le-nat-floor)
  obtain x' where suc:x = Suc x'
    using gr0-implies-Suc xgt0 by auto
  moreover have 2^(floorlog 2 r − 1) ≤ r and rtop:r < 2^(floorlog 2 r)
    using  floorlog-bounds rgt2 by force+
  then have 2 ∗ 2 ^ x' ≤ r using suc x a1 a2 by auto
    moreover have r ≤ 2 ∗ 2 ^ x' + size-l2 cfg (Suc x') − Suc 0
      using rtop  suc x a1 a2
      by (metis (no-types, lifting)  Suc-leI xgt0 suc
            add.commute add-cancel-right-right add-le-imp-le-diff a0 a1
           diff-is-0-eq diff-zero gr-implies-not0 le-add1 le-add2 mult-2 plus-1-eq-Suc

            power-Suc power-eq-if size-l2-i-not0 zero-less-diff)
   ultimately show ?thesis using x a0 a1
    unfolding l1-set-def  Let-def  size-small-l1-def size-l1-def  apply auto
    unfolding l1-set-def  Let-def  size-small-l1-def size-l1-def
     apply auto
    by (simp add: size-l2-i-not0)
  qed

lemma i-index-r-sm-eq-0-r-lt-2: assumes a0:sm cfg ≥ l cfg and
              a1:sm cfg = 0 and
              a2:r< 2^(sm cfg)+1
           shows r ∈ l1-set cfg 0
  using a0 a1 a2 unfolding l1-set-def Let-def
  by (auto simp add: size-small-l1-def)

lemma all-r-in-l2-sm-0-r-lt-sm:
  assumes a0:sm cfg ≥ l cfg and
        a1:sm cfg = 0 and
        a2:r< 2^(sm cfg)+1
```

```
      shows 0 < sl cfg ∧
            r ∈ l2-set cfg 0 0
  using i-index-r-sm-eq-0-r-lt-2[OF a0 a1 a2]
      set-l1-in-l2-sm-0-i-0[OF a0 - a1] by auto

lemma all-r-in-l2-sm-0-r-geqt-sm:
  assumes a0:sm cfg ≥ l cfg and
        a1:sm cfg = 0 and
        a2:r≥ 2^(sm cfg)+1
  shows 0 < sl cfg ∧ r ∈ l2-set cfg (r-gt-sm-0-i r) 0
  using i-index-r-sm-eq-0[OF a0 a1 a2]
      set-l1-in-l2-sm-0-i-gt-0[OF a0 - a1] a2
  by (metis One-nat-def a0 a1 le-0-eq less-Suc0 power-0 set-l1-in-l2 sl-def)

lemma all-r-in-l2-sm-gt-0-r-lt-sm:
  assumes a0:sm cfg ≥ l cfg and
        a1:sm cfg > 0 and
        a2:r< 2^(sm cfg)
      shows (r-lt-sm-gt-0-j cfg r) < sl cfg ∧
            r ∈ l2-set cfg 0 (r-lt-sm-gt-0-j cfg r)
  using i-index-r-sm-gt-0-r-lt-2[OF a0 a1 a2]
      set-l1-in-l2-sm-gt-0-i-0[OF a0 - a1] a2
  by auto

lemma all-r-in-l2-sm-gt-0-r-gt-sm:
  assumes a0:sm cfg ≥ l cfg and
        a1:sm cfg > 0 and
        a2:r≥ 2^(sm cfg)
      shows (r-gt-sm-gt-0-j cfg r)  < sl cfg ∧
            r ∈ l2-set cfg (r-gt-sm-gt-0-i cfg r) (r-gt-sm-gt-0-j cfg r)
  using i-index-r-sm-gt-0[OF a0 a1 a2]
      set-l1-in-l2-sm-gt-0-i-gt-0[OF a0 - a1] a2
  by auto

lemma all-r-in-l2:
  assumes a0:sm cfg ≥ l cfg
  shows ∃ i j. j < sl cfg ∧ r ∈ l2-set cfg i j
proof(cases sm cfg = 0)
  case True
  {assume a00:r< 2^(sm cfg)+1
    then have ?thesis
      using True all-r-in-l2-sm-0-r-lt-sm[OF a0 True a00]
      by fastforce
  }
  moreover {assume a00:r≥2^(sm cfg)+1
    then have ?thesis
      using True all-r-in-l2-sm-0-r-geqt-sm[OF a0]
      by fastforce
  } ultimately show ?thesis by fastforce
```

next
case *False*
then have *a00:sm cfg >0* **by** *auto*
then show *?thesis*
proof (*cases r< 2^(sm cfg)*)
case *True*
then show *?thesis*
using *True all-r-in-l2-sm-gt-0-r-lt-sm[OF a0 a00]*
by *fastforce*
next
case *False*
then have *2^sm cfg ≤ r* **by** *auto*
then show *?thesis*
using *all-r-in-l2-sm-gt-0-r-gt-sm[OF a0 a00]*
by *fastforce*
qed
qed

definition *mapping-insert-spec*::*Sys-Config ⇒ nat ⇒ (nat × nat) set*
where *mapping-insert-spec cfg r ≡ {(i,j). j < sl cfg ∧ r ∈ l2-set cfg i j}*

lemma *l2-set-not-empty:sm cfg ≥ l cfg ⟹ l2-set cfg i j≠{}*
unfolding *l2-set-def range-l2-def Let-def*
proof *auto*
assume *a0:l cfg ≤ sm cfg*
have *size-l2 cfg i > 0* **using** *size-l2-not-0[OF a0]* **by** *auto*
then show *∃ x≥l2-i-j cfg i j. x ≤ l2-i-j cfg i j + size-l2 cfg i − Suc 0*
by *auto*
qed

lemma *not-singleton-elements*:
¬ is-singleton A ⟹ A={} ∨ (∃ i j i′ j′. (i≠i′ ∨ j≠j′) ∧
(i,j)∈ A ∧ (i′,j′) ∈ A)
unfolding *is-singleton-def*
apply *auto*
by (*metis* (*no-types*, *hide-lams*) *insertI1 insert-absorb old.prod.exhaust singleton-insert-inj-eq′ subsetI*)

lemma *singleton-mapping-insert-spec*:
assumes *a0:sm cfg≥ l cfg*
shows *is-singleton* (*mapping-insert-spec cfg r*)
proof−
have *mapping-insert-spec cfg r ≠ {}*
unfolding *mapping-insert-spec-def*
using *all-r-in-l2[OF a0]* **by** *auto*
moreover have *∃ i j. (i,j)∈ mapping-insert-spec cfg r ∧*

$(\forall\, i'\ j'.\ (i',j') \in \mathit{mapping\text{-}insert\text{-}spec}\ \mathit{cfg}\ r \longrightarrow$
$i{=}i' \wedge j{=}j')$
using *calculation l2-set-disj a0* **unfolding** *mapping-insert-spec-def*
apply *auto*
using *linorder-neqE-nat* **by** *blast*
ultimately show *?thesis*
by (*metis not-singleton-elements*)
qed

definition *next-block*::*Sys-Config* ⇒ (*nat*×*nat*) ⇒ (*nat*×*nat*)
where *next-block cfg x* ≡ *if Suc* (*snd x*) < (*sl cfg*) *then* (*fst x*, (*snd x* + *1*))
else ((*fst x*)+*1*, *0*)

definition *block-lt*::('*a*::*wellorder*×'*a*) ⇒ ('*a*×'*a*) ⇒ *bool* (**infix** <$_b$ *50*)
where *block-lt x y* ≡
(*fst x* < *fst y*) ∨ (*fst x* = *fst y* ∧ *snd x* < *snd y*)

definition *block-let*::('*a*::*wellorder*×'*a*) ⇒ ('*a*×'*a*) ⇒ *bool* (**infix** ≤$_b$ *50*)
where *block-let x y* ≡ *x*=*y* ∨ *block-lt x y*

definition *block-gt*::('*a*::*wellorder*×'*a*) ⇒ ('*a*×'*a*) ⇒ *bool* (**infix** >$_b$ *50*)
where *block-gt x y* ≡ ¬ (*block-let x y*)

definition *block-get*::('*a*::*wellorder*×'*a*) ⇒ ('*a*×'*a*) ⇒ *bool* (**infix** ≥$_b$ *50*)
where *block-get x y* ≡ ¬ (*block-lt x y*)

thm *wellorder-Least-lemma*

lemma *b-lt-tran*:*a* <$_b$ *b* ⟹ *b* <$_b$ *c* ⟹ *a* <$_b$ *c*
unfolding *block-lt-def* **by** *auto*

lemma *b-let-tran*:*a* ≤$_b$ *b* ⟹ *b* ≤$_b$ *c* ⟹ *a* ≤$_b$ *c*
unfolding *block-let-def*
by (*auto intro*: *b-lt-tran*)

lemma *b-gt-tran*:*a* >$_b$ *b* ⟹ *b* >$_b$ *c* ⟹ *a* >$_b$ *c*
unfolding *block-gt-def block-let-def block-lt-def*
apply *auto*
by (*simp add*: *prod.expand*)

lemma *b-get-tran*:*a* ≥$_b$ *b* ⟹ *b* ≥$_b$ *c* ⟹ *a* ≥$_b$ *c*
unfolding *block-get-def block-gt-def block-let-def block-lt-def*
by *auto*

lemma *b-let-refl*:*a*≤$_b$*a*
unfolding *block-let-def*

by *auto*

lemma *b-get-refl*:$a \geq_b a$
unfolding *block-get-def block-gt-def block-let-def block-lt-def*
by *auto*

lemma *n-b-lt*:$\neg\ a <_b a$
unfolding *block-lt-def*
by *auto*

lemma *n-b-gt*:$\neg\ a >_b a$
unfolding *block-gt-def block-let-def*
by *auto*

lemma *antysimb*:$x \leq_b y \Longrightarrow y \leq_b x \Longrightarrow x = y$
unfolding *block-let-def block-lt-def*
by *auto*

lemma *next-block-bigger*:*sm cfg* $\geq$ *l cfg* $\Longrightarrow$
(ni,nj) = next-block cfg (i,j) $\Longrightarrow$
j < sl cfg $\Longrightarrow$
r$\in$*l2-set cfg i j* $\Longrightarrow$
rn$\in$ *l2-set cfg ni nj* $\Longrightarrow$
rn$>$*r*
unfolding *next-block-def l2-set-def Let-def*
apply (*cases* (*Suc j*) $<$ *sl cfg*)
by (*auto dest*: *snd-range-l2-i-j-less-fst-j'*[**where** *n*=*i* **and** *j*=*j* **and** *j'* = *Suc j*]
snd-range-l2-i-j-less-fst-i'-j'[**where** *j*=*j* **and** *j'*=*0* **and** *i*=*i* **and** *i'*=*Suc i*])

definition *mapping-insert*::*Sys-Config* $\Rightarrow$ *nat* $\Rightarrow$ (*nat*$\times$*nat*)
where *mapping-insert cfg r* $\equiv$
if (*sm cfg*) = *0 then*
(*if r* $<$((2^(*sm cfg*)) + *1*) *then* (*0*,*0*)
else (*r-gt-sm-0-i r*,*0*))
else (*if r* $<$(2^(*sm cfg*)) *then* (*0*,*r-lt-sm-gt-0-j cfg r*)
else (*r-gt-sm-gt-0-i cfg r*,*r-gt-sm-gt-0-j cfg r*))

lemma *mapping-insert-r-in-l2-set*:*sm cfg* $\geq$ *l cfg* $\Longrightarrow$
(*i*,*j*) = *mapping-insert cfg r* $\Longrightarrow$
r$\in$*l2-set cfg i j* $\wedge$ *j* $<$ *sl cfg*
unfolding *mapping-insert-def*
apply (*cases sm cfg* = *0*)
apply (*cases r*$<$ (*2* ^ *sm cfg*) + *1*)
apply (*simp add*: *all-r-in-l2-sm-0-r-lt-sm*)
apply *simp*
apply (*metis One-nat-def all-r-in-l2-sm-0-r-geqt-sm le-less-linear le-numeral-extra*(*3*)
numeral-2-eq-2 plus-1-eq-Suc power-eq-if)
apply (*cases r*$<$ (*2* ^ *sm cfg*))
apply (*auto dest*: *all-r-in-l2-sm-gt-0-r-lt-sm*)[*1*]

```
  using all-r-in-l2-sm-gt-0-r-gt-sm by auto

definition map-search:: Sys-Config ⇒ nat ⇒ (nat×nat)
  where map-search cfg r ≡ next-block cfg (mapping-insert cfg r)


lemma l2-set-search-gt-r:assumes a0:l cfg ≤ sm cfg and
       a1:(i,j) = (map-search cfg r)
     shows∀ r1∈ l2-set cfg i j. r1 > r
proof−
  {fix r1
    assume a00:r1 ∈l2-set cfg i j
    have r ∈ l2-set cfg (fst (mapping-insert cfg r))
                        (snd (mapping-insert cfg r)) ∧
          (snd (mapping-insert cfg r)) < sl cfg
      using mapping-insert-r-in-l2-set[OF a0, of - - r]
      using prod.collapse by blast
    then have r<r1
      using next-block-bigger[OF ] a0 a00
            a1[simplified map-search-def]
      by force

  } then show ?thesis by auto
qed


definition split-block::nat ⇒  bhdr-t ⇒ (bhdr-t×bhdr-t)
  where split-block r b = (Bhdr (s-addr b) ((s-addr b) + r − 1),
                           Bhdr ((s-addr b) + r + overhead conf) (e-addr b))

lemma split-size-sum:
  r > 0 ⟹ b-size b − overhead conf ≥ r ⟹
   (b1, b2) = split-block r b ⟹ b-size b1 + b-size b2 = b-size b − overhead conf
  unfolding split-block-def
  apply (cases b)
  by auto

definition join-block::bhdr-t ⇒ bhdr-t ⇒ bhdr-t
  where join-block b1 b2 ≡ Bhdr (s-addr b1) (e-addr b2)

type-synonym bitmap = nat ⇒ bool

definition fl-bitmap-f::Sys-Config ⇒ bhdr-matrix-t ⇒ bitmap
  where fl-bitmap-f cfg m ≡  (λi. (∃j< sl cfg. m i j≠{}))

definition sl-bitmap-f::bhdr-matrix-t ⇒ (nat ⇒ bitmap)
  where sl-bitmap-f m ≡  λi j. m i j≠{}

definition suitable-blocks::Sys-Config ⇒ (nat×nat) ⇒ state-t ⇒ (nat×nat) set
```

where *suitable-blocks cfg p σ* ≡
{(*i*,*j*). (*bhdr-matrix-f σ*) *i j*≠{} ∧ *p* ≤$_b$ (*i*,*j*) ∧ *j*< (*sl cfg*) }

definition *suitable-blocks-bitmap*::*Sys-Config* ⇒ (*nat*×*nat*) ⇒ *state-t* ⇒ (*nat*×*nat*) *set*
where *suitable-blocks-bitmap cfg p σ* ≡
let i = *fst p*; *j* = *snd p in*
{(*i*′,*j*′). (*fl-bitmap-f cfg* (*bhdr-matrix-f σ*)) *i*′ = *True* ∧ *j*′< (*sl cfg*) ∧
(*sl-bitmap-f* (*bhdr-matrix-f σ*)) *i*′ *j*′ = *True* ∧ ((*i*,*j*)≤$_b$(*i*′,*j*′))}

lemma *suitable-blocks-eq*:*suitable-blocks cfg p σ* = *suitable-blocks-bitmap cfg p σ*
unfolding *suitable-blocks-def suitable-blocks-bitmap-def fl-bitmap-f-def sl-bitmap-f-def Let-def* **by** *auto*

definition
Leastb :: (('*a*::*wellorder*×'*a*) ⇒ *bool*) ⇒ ('*a*::*wellorder*×'*a*) (**binder** *LEAST*$_b$ *10*)
where
Leastb P = (*THE x*. *P x* ∧ (∀ *y*. *P y* ⟶ *x* ≤$_b$ *y*))

instantiation *prod*:: (*ord*, *ord*) *ord*
begin
definition
less-prod-def:*p* < *q* ≡ (*fst p* < *fst q*) ∨ (*fst p* = *fst q* ∧ *snd p* < *snd q*)

definition
less-eq-prod-def:*p* ≤ *q* ≡ *p*=*q* ∨ ((*fst p* < *fst q*) ∨ (*fst p* = *fst q* ∧ *snd p* < *snd q*))
instance ..
end

instantiation *prod*:: (*linorder*, *linorder*) *linorder*
begin
instance
proof
fix *x y z* ::'*a*::*linorder* × '*b*::*linorder*
show (*x* < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)
unfolding *less-prod-def less-eq-prod-def* **by** *auto*
show *x* ≤ *x* **unfolding** *less-prod-def less-eq-prod-def* **by** *auto*
show *x* ≤ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z*
unfolding *less-prod-def less-eq-prod-def* **by** *auto*
show *x* ≤ *y* ⟹ *y* ≤ *x* ⟹ *x* = *y*
unfolding *less-prod-def less-eq-prod-def* **by** *auto*
show *x* ≤ *y* ∨ *y* ≤ *x*
unfolding *less-prod-def less-eq-prod-def* **apply** *auto*
using *less-linear prod-eqI* **by** *blast*
qed
end

definition *min-elem-set*::(*nat*×*nat*) *set* ⇒ (*nat*×*nat*)
where *min-elem-set s* ≡
(*LEAST x*. *x* ∈ *s*)

definition *mapping-search*:: *Sys-Config* ⇒ *nat* ⇒ (*nat*×(*nat*×*nat*))
where *mapping-search cfg r* ≡ *let r* = *if r*< (*min-block cfg*) *then min-block cfg* *else r*;
(*i*,*j*) = *mapping-insert cfg r*;
(*i*′,*j*′) = *next-block cfg* (*i*,*j*);
initial-size = *fst* (*range-l2 cfg i j*);
r′= *fst* (*range-l2 cfg i*′ *j*′) *in*
if initial-size = *r then* (*r*,(*i*,*j*))
else (*r*′, (*i*′,*j*′))

lemma *l2-set-mapping-search-geq-r*:
assumes *a0*:*l cfg* ≤ *sm cfg* **and**
a1:(*r*′, (*i*,*j*)) = (*mapping-search cfg r*)
shows*r*′≥ *r* ∧ (∀ *r1*∈ *l2-set cfg i j*. *r1* ≥ *r*′)
proof−
{assume *a00*:*r*<(*min-block cfg*)
then have *?thesis*
using *a1* **unfolding** *mapping-search-def Let-def*
apply *simp* **apply** (*split prod.splits*)+
apply (*case-tac fst* (*range-l2 cfg x1 x2*) = *min-block cfg*)
apply *auto*
using *l2-set-search-gt-r*[*OF a0*, *of i j* (*min-block cfg*)] **unfolding** *map-search-def*

by (*auto simp add*: *a0 l2-set-def Let-def range-l2-disj*)
}
moreover {assume *r*≥(*min-block cfg*)
then have *?thesis*
using *a1* **unfolding** *mapping-search-def Let-def*
apply *simp* **apply** (*split prod.splits*)+
apply (*case-tac fst* (*range-l2 cfg x1 x2*) = *r*)
apply *auto*
using *l2-set-search-gt-r*[*OF a0*, *of i j r*] **unfolding** *map-search-def*
by (*auto simp add*: *a0 l2-set-def Let-def range-l2-disj*)
}
ultimately show *?thesis* **by** *fastforce*
qed

definition *find-suitable-blocks-opt*::(*nat*×*nat*) ⇒ *state-t* ⇒ ((*nat*×*nat*) *set*) *option*
where *find-suitable-blocks-opt p s* ≡

```
  let x = suitable-blocks conf p s in
      if (x={}) then None else Some x


definition ij-level-empty::bhdr-matrix-t ⇒ nat ⇒ nat ⇒bool
  where ij-level-empty m i j ≡ m i j = {}

definition i-level-empty::Sys-Config ⇒ bhdr-matrix-t ⇒ nat  ⇒bool
  where i-level-empty cfg m i ≡ ∀ j< (sl cfg). ij-level-empty m i j

definition remove-elem-from-matrix :: bhdr-t ⇒ nat ⇒ nat ⇒ state-t ⇒state-t
  where remove-elem-from-matrix b i j σ ≡
    let matrix-not-elem = (Set.remove b ((bhdr-matrix-f σ) i j));
        new-matrix = set-bhdr-matrix (bhdr-matrix-f σ) i j matrix-not-elem  in
    σ(|bhdr-matrix-f:= new-matrix|)

definition remove-block::(nat×nat) ⇒ (state-t, bhdr-t) nondet-monad
  where remove-block p  ≡
 let i = fst p; j = snd p in
     do
       matrix ← gets (λσ. (bhdr-matrix-f σ) i j);
       b ← select matrix;
       modify (remove-elem-from-matrix b i j);
       return b
     od

definition add-block::bhdr-t ⇒ state-t ⇒ state-t
  where add-block b σ ≡
 let (i,j) = mapping-insert conf (b-size b)  in
   σ(|bhdr-matrix-f := insert-block-bhdr-matrix  (bhdr-matrix-f σ) i j b|)

definition malloc::nat ⇒ (state-t, nat) nondet-monad
  where malloc r ≡  let (r,(i,j)) = mapping-search conf r in
                    do set-ps ← gets (find-suitable-blocks-opt (i,j));
                       condition (λs. set-ps = None) (return 0)
                       (do p ← select (the set-ps);
                           b ← remove-block p;
                           (condition (λs. b-size b − r ≥ (min-block conf))
                               (do (b1,b2) ← gets (λs. (split-block r b));
                                modify (λs. s(|alloced-bhdr-s:= insert b1 (alloced-bhdr-s
s)|));
                                modify (add-block b2);
                                return (s-addr b1) od)
                               (do
                                modify (λs. s(|alloced-bhdr-s:= insert b (alloced-bhdr-s
s)|));
                                return (s-addr b)
                                od))
                        od)
```

```
                od


definition join-prev::bhdr-t ⇒ (state-t, bhdr-t) nondet-monad
  where
join-prev b ≡  do b' ← gets (prev-free-hdr-s conf b);
                  condition (λs. b' = None)
                    (return b)
                    (let (i,j) = mapping-insert conf (b-size(the b'));
                         b-join = join-block (the b') b
                     in
                     do
                       modify (remove-elem-from-matrix (the b') i j);
                       return b-join
                     od)
               od

definition join-suc::bhdr-t ⇒ (state-t, bhdr-t) nondet-monad
  where
join-suc b ≡ do b' ← gets (suc-hdr-free-s conf b);
                  condition (λs. b' = None)
                    (return b)
                    (let (i,j) = mapping-insert conf (b-size(the b'));
                         b-join = join-block b (the b')
                     in
                     do
                       modify (remove-elem-from-matrix (the b') i j);
                       return b-join
                     od)
               od


definition free::nat ⇒ (state-t, nat) nondet-monad
  where free addr ≡ condition (block-alloced addr)
                      (do b ← gets (get-alloced-block addr);
                        modify (λs. s(| alloced-bhdr-s := Set.remove b (alloced-bhdr-s
s)|));
                          b ← join-suc b;
                          b ← join-prev b;
                          modify (add-block b);
                          return 1
                      od)
                      (do
                        modify (λs. undefined);
                        return undefined
                       od)

inductive run::state-t list ⇒ bool where
```

single-s:*run* [*x*]
| *malloc*: ⟦*run* (*x*#*xs*); *v* = (*SOME v*. *v*∈(*fst* (*malloc r x*)))⟧ ⟹ *run* ((*snd v*)#*x*#*xs*)
| *free*: ⟦*run* (*x*#*xs*); *v* = (*SOME v*. *v*∈(*fst* (*free r x*)))⟧ ⟹ *run* ((*snd v*)#*x*#*xs*)

— properties
abbreviation *block-t-size* ::*bhdr-t* ⇒ *nat*
where *block-t-size b* ≡ *b-size b* + *overhead conf*

definition *wf-block*::*bhdr-t* ⇒ *bool*
where *wf-block b* ≡ *s-addr b* ≤ *e-addr b* ∧ *s-addr b* ≥ *overhead conf* ∧ *block-t-size b* ≥ *min-block conf* ∧
block-t-size b ≤ *mem-size conf*

definition *wf*:: *state-t* ⇒ *bool*
where *wf σ* ≡ ∀ *x* ∈ *all-blocks conf σ*. *wf-block x*

definition *disjoint-free-non-free*::*state-t* ⇒ *bool*
where *disjoint-free-non-free σ* ≡ *alloced-bhdr-s σ* ∩ *free-blocks conf σ* = {}

definition *disjoint-memory*::*bhdr-t* ⇒ *bhdr-t* ⇒ *bool*
where *disjoint-memory b1 b2* ≡
(*e-addr b1* + (*overhead conf*) < *s-addr b2* ∨
e-addr b2 + (*overhead conf*) < *s-addr b1*)

definition *disjoint-memory-set*:: *state-t* ⇒ *bool*
where *disjoint-memory-set σ* ≡
∀ *x1 x2*. *x1* ∈ *all-blocks conf σ* ∧ *x2* ∈ *all-blocks conf σ* ∧ *x1*≠*x2* ⟶ *disjoint-memory x1 x2*

definition *no-split-memory*::*state-t* ⇒ *bool*
where*no-split-memory σ* ≡ *let f* = *free-blocks conf σ in*
¬ (∃ *b1 b2*. *b1* ∈ *f* ∧ *b2* ∈ *f* ∧
(*s-addr b1* = *e-addr b2* + *1* + *overhead conf*))

definition *wf-adjacency-list*::*state-t* ⇒ *bool*
where *wf-adjacency-list σ* ≡ *tlsf-matrix conf* (*bhdr-matrix-f σ*)

definition *wf-bitmap1*::*state-t* ⇒ *bool*
where *wf-bitmap1 σ* ≡
∀ *i* . *fl-bitmap-f conf* (*bhdr-matrix-f σ*) *i* =
(∃ *j*< *sl conf*. *sl-bitmap-f* (*bhdr-matrix-f σ*) *i j*)

definition *wf-bitmap2*::*state-t* ⇒ *bool*
where *wf-bitmap2 σ* ≡
∀ *i* . ∀ *j* < *sl conf*.
(*bhdr-matrix-f σ*) *i j*≠{} =

sl-bitmap-f (*bhdr-matrix-f* σ) *i j*

definition *wf-bitmap*::*state-t* $\Rightarrow$ *bool*
where *wf-bitmap* $\sigma \equiv$ *wf-bitmap1* $\sigma \wedge$ *wf-bitmap2* σ

definition *sum-block*::*bhdr-t set* $\Rightarrow$ *nat*
where *sum-block* $\sigma \equiv$ *Finite-Set.fold* ($\lambda b\ s.$ *block-t-size* $b + s$) *0* σ

definition *all-block-mem-size*::*state-t* $\Rightarrow$ *bool*
where *all-block-mem-size* $\sigma \equiv$ *sum-block* (*all-blocks conf* σ) = *mem-size conf*

definition *inv*::*state-t* $\Rightarrow$ *bool*
where *inv* $\sigma \equiv$ *no-split-memory* $\sigma \wedge$ *disjoint-free-non-free* $\sigma \wedge$*disjoint-memory-set* $\sigma \wedge$ *wf* $\sigma \wedge$
wf-adjacency-list $\sigma \wedge$ *all-block-mem-size* σ

lemma *unique-get-alloced-block1*:*block-alloced addr* $\sigma \Longrightarrow \exists\, b.\ (\exists\, e\text{-}addr.\ b =$ *Bhdr addr e-addr*$) \wedge b \in$ *alloced-bhdr-s* σ
unfolding *block-alloced-def* **by** *auto*

lemma *diff-block-diff-s-addr*:**assumes** *a0*:*inv* σ **and**
a1:*b1*$\in$ *all-blocks conf* σ **and**
a2:*b2*$\in$ *all-blocks conf* σ **and**
a3:*b1*$\neq$*b2*
shows *s-addr b1* $\neq$ *s-addr b2*
proof(*auto*)
assume *a4*:*s-addr b1* = *s-addr b2*
moreover have *wf-block b1* $\wedge$ *wf-block b2* **using** *a0 a1 a2* **unfolding** *inv-def wf-def* **by** *auto*
moreover have *disjoint-memory b1 b2* **using** *a0 a1 a2 a3* **unfolding** *inv-def disjoint-memory-set-def*
by *auto*
ultimately show *False* **unfolding** *disjoint-memory-def*
unfolding *wf-block-def*
by *linarith*
qed

lemma *diff-block-diff-e-addr*:**assumes** *a0*:*inv* σ **and**
a1:*b1*$\in$ *all-blocks conf* σ **and**
a2:*b2*$\in$ *all-blocks conf* σ **and**
a3:*b1*$\neq$*b2*
shows *e-addr b1* $\neq$ *e-addr b2*
proof(*auto*)
assume *a4*:*e-addr b1* = *e-addr b2*
moreover have *wf-block b1* $\wedge$ *wf-block b2* **using** *a0 a1 a2* **unfolding** *inv-def wf-def* **by** *auto*
moreover have *disjoint-memory b1 b2* **using** *a0 a1 a2 a3* **unfolding** *inv-def disjoint-memory-set-def*
by *auto*

ultimately show *False* **unfolding** *disjoint-memory-def*
unfolding *wf-block-def*
by *linarith*
qed

lemma *same-addr-same-block*:
assumes *a0*:*inv* σ **and**
a1:($\exists$ *e-addr*. *b1* = *Bhdr addr e-addr*) $\wedge$ *b1* $\in$ *alloced-bhdr-s* σ **and**
a2:($\exists$ *e-addr*. *b2* = *Bhdr addr e-addr*) $\wedge$ *b2* $\in$ *alloced-bhdr-s* σ
shows *b1* = *b2*
using *a1 a2*
using *diff-block-diff-s-addr*[*OF a0 - -*] **unfolding** *all-blocks-def*
apply *auto*
by (*metis* (*no-types*) *bhdr-t.inject bhdr-t.sel*(*1*))

lemma $\exists!b$. *get-alloced-block addr* σ = *b*
by *auto*

context begin
private lemma *alloc-insert-no-split*: *no-split-memory s* $\Longrightarrow$ *no-split-memory* (*s*(|*alloced-bhdr-s* := *insert b* (*alloced-bhdr-s s*) |))
unfolding *no-split-memory-def free-blocks-def*
by *auto*
private lemma *alloc-insert-no-split′*: *no-split-memory* (*s*(|*bhdr-matrix-f* := *m* |)) $\Longrightarrow$ *no-split-memory* (*s*(| *alloced-bhdr-s* := *insert b* (*alloced-bhdr-s s*),*bhdr-matrix-f* := *m* |))
unfolding *no-split-memory-def free-blocks-def*
by *auto*
private lemma *alloc-insert-no-split″*: *no-split-memory* (*s*(|*bhdr-matrix-f* := *m* |)) $\Longrightarrow$ *no-split-memory* (*s*(| *bhdr-matrix-f* := *m*, *alloced-bhdr-s* := *insert b* (*alloced-bhdr-s s*) |))
unfolding *no-split-memory-def free-blocks-def*
by *auto*

lemma *subset-remove-no-split*: *no-split-memory s* $\Longrightarrow$ *free-blocks conf s* $\geq$ *free-blocks conf s′* $\Longrightarrow$ *no-split-memory s′*
unfolding *no-split-memory-def*
apply *auto*
by (*meson subset-iff*)

lemma *matrix-remove-no-split*: *no-split-memory s* $\Longrightarrow$
no-split-memory (*s*(|*bhdr-matrix-f* := *set-bhdr-matrix* (*bhdr-matrix-f s*) *i j* (*Set.remove b* (*bhdr-matrix-f s i j*)) |))
apply (*rule subset-remove-no-split*)
apply *assumption*
apply (*thin-tac no-split-memory* -)
unfolding *free-blocks-def set-bhdr-matrix-def*

apply *auto*
by *blast*

lemma *split-alloc-no-split*:
no-split-memory s $\Longrightarrow$
wf s $\Longrightarrow$
disjoint-memory-set s $\Longrightarrow$
b $\in$ *free-blocks conf s* $\Longrightarrow$
r > *0* $\Longrightarrow$ — split a zero is meaningless
r < *b-size b* − *overhead conf* $\Longrightarrow$
free-blocks conf s' = (*free-blocks conf s*) − {*b*} ∪ {*snd* (*split-block r b*)} $\Longrightarrow$
no-split-memory s'
unfolding *no-split-memory-def*
apply *auto*
subgoal
unfolding *wf-def wf-block-def split-block-def*
apply *auto*
by (*metis b-size.simps bhdr-t.exhaust-sel diff-add-inverse diff-le-self not-le plus-1-eq-Suc*)
subgoal for *b2*
unfolding *disjoint-memory-set-def disjoint-memory-def wf-def wf-block-def split-block-def*
apply *auto*
apply (*drule spec*[*of - b2*])
apply (*drule spec*[*of - b*])
apply (*auto simp*: *all-blocks-def*)
apply (*drule bspec*[*of - - b2*])
apply *blast*
apply (*cases b*)
by *auto*
subgoal for *b1*
unfolding *disjoint-memory-set-def disjoint-memory-def wf-def wf-block-def split-block-def*
apply *auto*
apply (*drule spec*[*of - b1*])
apply (*drule spec*[*of - b*])
apply (*auto simp*: *all-blocks-def*)
by *meson*
subgoal
by *metis*
done

thm *bspec*
thm *split-beta*

lemma *find-opt-is-free*:
find-suitable-blocks-opt (*i*,*j*) *s* = *Some ps* $\Longrightarrow$
(*i'*, *j'*) $\in$ *ps* $\Longrightarrow$
b $\in$ *bhdr-matrix-f s i' j'* $\Longrightarrow$
b $\in$ *free-blocks conf s*
unfolding *free-blocks-def*

apply (*subst Union-iff*)
apply (*rule bexI*)
apply *assumption*
apply *auto*
unfolding *find-suitable-blocks-opt-def suitable-blocks-def*
apply *auto*
by (*metis* (*no-types*, *lifting*) *case-prodD mem-Collect-eq option.discI option.inject*)

lemma *fst-range-in-set*: *l cfg* ≤ *sm cfg* ⟹ *fst* (*range-l2 cfg i j*) ∈ *l2-set cfg i j*
unfolding *l2-set-def Let-def*
by (*auto simp*: *range-l2-disj*)

lemma *map-search-r-ge-minblock*:
l cfg ≤ *sm cfg* ⟹ (*r'*,(*i*,*j*)) = *mapping-search cfg r* ⟹ *r'* ≥ *min-block cfg*
apply (*cases r* < *min-block cfg*)
unfolding *mapping-search-def*
apply (*auto simp*: *Let-def split*: *prod.splits if-splits*)
using *mapping-insert-r-in-l2-set next-block-bigger fst-range-in-set*
apply (*metis less-imp-le-nat*)
using *mapping-insert-r-in-l2-set next-block-bigger fst-range-in-set*
by (*smt le-cases less-le-trans*)

lemma *map-search-r-gt-0*:*l cfg* ≤ *sm cfg* ⟹ *min-block cfg* > *0* ⟹ (*r'*,(*i*,*j*)) = *mapping-search cfg r* ⟹ *r'* > *0*
using *map-search-r-ge-minblock* **by** *force*

lemma *free-blocks-insert-is-union*:
j < *sl cfg* ⟹ *free-blocks-mat cfg mat* = *f* ⟹
free-blocks cfg (*s*(| *bhdr-matrix-f* := *insert-block-bhdr-matrix mat i j b* |)) = *f* ∪ {*b*}
unfolding *free-blocks-mat-def free-blocks-def*
apply *rule*
subgoal
apply *rule*
apply *auto*
subgoal for *x ii jj*
apply (*drule spec*[*of - mat ii jj*])
apply *auto*
unfolding *insert-block-bhdr-matrix-def set-bhdr-matrix-def*
by (*auto split*:*if-splits*)
done
subgoal
apply *rule*
apply *auto*
unfolding *insert-block-bhdr-matrix-def set-bhdr-matrix-def*
apply *auto*
by (*metis insert-iff*)
done

lemma *insert-is-union-conf*:
mapping-insert conf (*b-size b*) = (*i*, *j*)⟹ *free-blocks-mat conf mat* = *f* ⟹
free-blocks conf (*s*(‖ *bhdr-matrix-f* := *insert-block-bhdr-matrix mat i j b* ‖)) = *f* ∪ {*b*}
apply (*rule free-blocks-insert-is-union*)
apply (*metis mapping-insert-r-in-l2-set mbiggerl*) **.**

lemma *neq-split*: *a* ≠ *b* ⟹ *a* < *b* ∨ *a* > (*b*::*nat*)
by *auto*

lemma *free-block-no-dup*:
wf-adjacency-list s ⟹ *b* ∈ *bhdr-matrix-f s i j* ⟹ *b* ∈*bhdr-matrix-f s i′ j′* ⟹
j < *sl conf* ⟹ *j′* < *sl conf* ⟹
i=*i′* ∧ *j*=*j′*
unfolding *wf-adjacency-list-def tlsf-matrix-def*
unfolding *l2-set-def*
apply (*frule spec*[*of* - *i*])
apply (*drule spec*[*of* - *i′*])
apply (*drule spec*[*of* - *j*])
apply (*drule spec*[*of* - *j′*])
apply (*subgoal-tac i* = *i′*)
apply (*auto simp*: *Let-def*)
defer
subgoal
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *b*])
apply *auto*
apply (*rule ccontr*)
apply (*drule neq-split*)
apply (*erule disjE*)
subgoal
using *snd-range-l2-i-j-less-fst-i′-j′*[*of conf j j′ i i′*, *OF mbiggerl*]
unfolding *sl-def* **by** *linarith*
subgoal
using *snd-range-l2-i-j-less-fst-i′-j′*[*of conf j′ j i′ i*, *OF mbiggerl*]
unfolding *sl-def* **by** *linarith*
done
subgoal
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *b*])
apply *auto*
apply (*rule ccontr*)
apply (*drule neq-split*)
apply (*erule disjE*)
subgoal
using *snd-range-l2-i-j-less-fst-j′*[*of conf j j′*, *OF mbiggerl*]
using *leD le-less-trans* **by** *blast*
subgoal

```
    using snd-range-l2-i-j-less-fst-j'[of conf j' j, OF mbiggerl]
    using leD le-less-trans by blast
   done
  done

lemma free-blocks-remove-is-minus:
  wf-adjacency-list s ⟹ b ∈ bhdr-matrix-f s i j ⟹ j < sl conf ⟹
  free-blocks-mat conf (set-bhdr-matrix (bhdr-matrix-f s) i j (Set.remove b (bhdr-matrix-f
s i j))) = free-blocks conf s − {b}
  unfolding free-blocks-def free-blocks-mat-def
  apply rule
  subgoal
   apply rule
   unfolding set-bhdr-matrix-def
   apply auto
   subgoal
    by (auto split:if-splits)
   subgoal for ai aj
    apply (auto split:if-splits)
    using free-block-no-dup
    unfolding sl-def
    by blast+
   done
  subgoal
   apply rule
   unfolding set-bhdr-matrix-def
   apply auto
   subgoal for x xi xj
   apply (rule exI[of - Set.remove b (bhdr-matrix-f s xi xj)])
   apply auto
    apply (rule exI[of - xi])
    apply auto
    apply (rule exI[of - xj])
    apply auto
    using free-block-no-dup
    apply simp
    apply (rule exI[of - xj])
    apply auto
    using free-block-no-dup
    by simp
   done
  done

lemma remove-is-minus-conf:
  wf-adjacency-list s ⟹ b ∈ bhdr-matrix-f s i j ⟹ mapping-insert conf (b-size
b) = (i, j) ⟹
  free-blocks-mat conf (set-bhdr-matrix (bhdr-matrix-f s) i j (Set.remove b (bhdr-matrix-f
s i j))) = free-blocks conf s − {b}
  apply (rule free-blocks-remove-is-minus)
```

apply *auto*
by (*metis mapping-insert-r-in-l2-set mbiggerl sl-def*)

lemma *suitable-blocks-j-lt-sl*:
find-suitable-blocks-opt (*i*, *j*) *s* = *Some ps* ⟹ (*i'*, *j'*) ∈ *ps* ⟹ *j'* < *sl conf*
unfolding *find-suitable-blocks-opt-def suitable-blocks-def*
by (*auto split*: *if-splits*)

lemma *suitable-blocks-ij-increase*:
find-suitable-blocks-opt (*i*, *j*) *s* = *Some ps* ⟹ (*i'*, *j'*) ∈ *ps* ⟹ (*i*, *j*) ≤$_b$ (*i'*, *j'*)
unfolding *find-suitable-blocks-opt-def suitable-blocks-def*
by (*auto split*:*if-splits*)

lemma *block-mat-size*:
wf-adjacency-list s ⟹ *b* ∈ *bhdr-matrix-f s i j* ⟹ *j* < *sl conf* ⟹ *b-size b* ∈ *l2-set conf i j*
unfolding *wf-adjacency-list-def tlsf-matrix-def*
by *blast*

lemma *size-l2-set-i-mono*:
l cfg ≤ *sm cfg* ⟹ *ja* < *sl cfg* ⟹ *jb* < *sl cfg* ⟹
b-size a ∈ *l2-set cfg ia ja* ⟹
b-size b ∈ *l2-set cfg ib jb* ⟹
b-size a ≤ *b-size b* ⟹ *ia* ≤ *ib*
unfolding *l2-set-def*
apply (*auto simp*: *Let-def*)
apply (*rule ccontr*)
by (*metis less-le-trans not-le sl-def snd-range-l2-i-j-less-fst-i'-j'*)

lemma *split-decrease-size*: *min-block conf* ≤ *b-size b* − *r* ⟹ *b-size* (*snd* (*split-block r b*)) ≤ *b-size b*
unfolding *split-block-def*
apply *auto*
apply (*cases b*)
by *auto*

lemma *inv-malloc-no-split-memory*:⦃λσ. *inv* σ⦄ (*malloc r*) ⦃λn σ. *no-split-memory* σ⦄
unfolding *malloc-def Let-def*
apply (*cases mapping-search conf r*)
apply (*rename-tac r' i j*)
apply *auto*
apply *wp*
apply (*case-tac split-block r' b*)
apply *auto*[*1*]
apply *wp*
apply (*subgoal-tac aa*= *fst* (*split-block r' ba*))
apply (*subgoal-tac baa*= *snd* (*split-block r' ba*))
apply *hypsubst-thin*

apply *wp*
apply (*simp add*: *prod-injects*(*2*))
apply *simp*
apply *wp*
unfolding *remove-block-def Let-def*
apply *wp*
apply (*subgoal-tac* $r' = fst$(*mapping-search conf r*))
apply *hypsubst-thin*
apply (*rule select-wp*)
apply *simp*
apply *wp*
apply (*rule select-wp*)
apply *wp*
apply *auto*
subgoal
  unfolding *inv-def*
  by *simp*
subgoal for r' *i j s* i' j' *b ps*
  unfolding *remove-elem-from-matrix-def Let-def*
  apply *auto*
  unfolding *add-block-def*
  apply *auto*
  apply (*cases mapping-insert conf* (*b-size* (*snd* (*split-block* r' *b*))))
  subgoal for *ia ja*
    apply *auto*
    apply (*rule alloc-insert-no-split'*)
    apply (*rule split-alloc-no-split*)
    apply (*simp add*:*inv-def*)
    apply (*erule conjE*)+
    apply *assumption*
    apply (*simp add*:*inv-def*)
    apply (*simp add*:*inv-def*)
    apply (*rule find-opt-is-free*)
    apply *assumption*+
    apply (*rule map-search-r-gt-0*[**where** *cfg*=*conf*])
    apply (*simp add*: *mbiggerl*)
    using *min-block-gt-overhead* **apply** *linarith*
    apply (*rule sym*)
    apply *assumption*
    using *min-block-gt-overhead* **apply** *linarith*
    apply (*rule free-blocks-insert-is-union*)
    prefer *2*
    apply (*rule free-blocks-remove-is-minus*)
    apply (*simp add*: *inv-def*)
    apply *assumption*
    using *suitable-blocks-j-lt-sl* **apply** *blast*
    using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **by** *metis*
  done
subgoal

unfolding *remove-elem-from-matrix-def Let-def*
apply *auto*
apply (*rule alloc-insert-no-split''*)
apply (*rule matrix-remove-no-split*)
unfolding *inv-def*
by *auto*
done
end
declare *select-wp*[*wp*]
context begin — disjoint free non free

lemma *alloc-free-non-free-disjoint*:
disjoint-free-non-free s ⟹ *wf-adjacency-list s* ⟹ *j* < *sl conf* ⟹ *b* ∈ *bhdr-matrix-f s i j* ⟹
disjoint-free-non-free
(*s*(|*bhdr-matrix-f* := *set-bhdr-matrix* (*bhdr-matrix-f s*) *i j* (*Set.remove b* (*bhdr-matrix-f s i j*)),
alloced-bhdr-s := *insert b* (*alloced-bhdr-s s*)|))
unfolding *disjoint-free-non-free-def*
apply (*subst free-blk-mat-s-eq*)
apply (*clarsimp simp del*: *sl-def*)
apply (*subst free-blocks-remove-is-minus*)
apply *assumption*+
apply (*subst free-blocks-remove-is-minus*)
apply *assumption*+
by *blast*

lemma *split-free-not-exist-fst*:
disjoint-memory-set s ⟹
wf s ⟹
r ≠ *b-size b* ⟹ — to avoid b and b1 being identical
split-block r b = (*b1*, *b2*) ⟹
b ∈ *free-blocks conf s* ⟹
b1 ∉ *all-blocks conf s*
apply *rule*
unfolding *disjoint-memory-set-def*
apply (*drule spec*[*of - b*])
apply (*drule spec*[*of - b1*])
apply (*auto simp*: *all-blocks-def split-block-def wf-def*)
subgoal
apply (*cases b*, *auto simp*: *wf-block-def*)
by (*metis Suc-pred Un-iff bhdr-t.sel*(*1*) *diff-add-inverse neq0-conv not-le oh-gt-0 zero-eq-add-iff-both-eq-0*)
subgoal
apply (*cases b*, *auto simp*: *disjoint-memory-def wf-block-def*)
apply (*metis Un-iff add-lessD1 bhdr-t.sel not-le*)
using *oh-gt-0* **by** *linarith*
subgoal

apply (*cases b*, *auto simp*: *wf-block-def*)
apply (*case-tac x1*, *auto*)
apply (*cases r*, *auto*)
by (*metis Un-iff bhdr-t.sel*(*1*) *leD oh-gt-0*)
subgoal
apply (*cases b*, *auto simp* : *disjoint-memory-def wf-block-def*)
apply (*metis Un-iff add-lessD1 bhdr-t.sel not-le*)
using *oh-gt-0* **by** *linarith*
done

lemma *split-free-not-free-fst*:
disjoint-memory-set s ⟹
wf s ⟹
r ≠ *b-size b* ⟹
split-block r b = (*b1*, *b2*) ⟹
b ∈ *free-blocks conf s* ⟹
b1 ∉ *free-blocks conf s*
using *split-free-not-exist-fst all-blocks-def*
by *blast*

lemma *split-free-not-exist-snd*:
disjoint-memory-set s ⟹
wf s ⟹
split-block r b = (*b1*, *b2*) ⟹
b ∈ *free-blocks conf s* ⟹
b2 ∉ *all-blocks conf s*
apply (*rule*)
unfolding *disjoint-memory-set-def*
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *b2*])
apply (*auto simp*: *all-blocks-def split-block-def wf-def*)
subgoal
apply (*cases b*, *auto*)
using *oh-gt-0* **by** *presburger*
subgoal
by (*cases b*) (*force simp*: *wf-block-def disjoint-memory-def*)
subgoal
apply (*cases b*, *auto*)
using *oh-gt-0* **by** *presburger*
subgoal
by (*cases b*) (*force simp*: *wf-block-def disjoint-memory-def*)
done

lemma *split-free-not-alloced-snd*:
disjoint-memory-set s ⟹
wf s ⟹
split-block r b = (*b1*, *b2*) ⟹
b ∈ *free-blocks conf s* ⟹
b2 ∉ *alloced-bhdr-s s*

using *split-free-not-exist-snd all-blocks-def*
by *blast*

lemma *split-fst-snd-neq*:
wf-block b $\Longrightarrow$ *b1* = *fst*(*split-block r b*) $\Longrightarrow$ *b2* = *snd*(*split-block r b*) $\Longrightarrow$ *b1* $\neq$ *b2*
unfolding *split-block-def wf-block-def*
apply (*cases b*)
apply *auto*
using *oh-gt-0*
by *linarith*

lemma *inv-malloc-disjoint-free-non-free*:$\{\!|\lambda\sigma.\ inv\ \sigma|\!\}$ (*malloc r*) $\{\!|\lambda n\ \sigma.\ disjoint\text{-}free\text{-}non\text{-}free\ \sigma|\!\}$
unfolding *malloc-def remove-block-def Let-def*
apply *wpsimp*
subgoal for *s r' i j*
apply *auto*
apply (*simp add*:*inv-def*)
subgoal for *ps i' j' b b1 b2*
unfolding *add-block-def remove-elem-from-matrix-def Let-def*
apply *auto*
apply (*cases mapping-insert conf* (*b-size* (*snd* (*split-block r' b*))))
apply (*rename-tac ia ja*)
apply *auto*
unfolding *disjoint-free-non-free-def*
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *blast*
apply (*rule refl*)
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *blast*
apply (*rule refl*)
subgoal for *ia ja*
proof
assume (*i'*, *j'*) $\in$ *ps b* $\in$ *bhdr-matrix-f s i' j' find-suitable-blocks-opt* (*i*, *j*) *s* = *Some ps*
hence *j'* < *sl conf*
using *suitable-blocks-j-lt-sl less-imp-le-nat* **by** *blast*
hence *b* $\in$ *free-blocks conf s*
unfolding *free-blocks-def*

```
      using ‹b ∈ -› by blast
    assume inv s
    hence wf-block b
      unfolding inv-def wf-def all-blocks-def
      using ‹b ∈ free-blocks conf s› by blast
    assume mapping-search conf r = (r', i, j)
    hence r' > 0
      using map-search-r-gt-0[OF mbiggerl] min-block-gt-overhead
      by (metis diff-self-eq-0 less-imp-diff-less)
    assume min-block conf ≤ b-size b − r'
    assume split-block r' b = (b1, b2)
    hence split-b:b1 = fst(split-block r' b) b2 = snd(split-block r' b)
      by simp+
    have b1 ∉ free-blocks conf s
      apply (rule split-free-not-free-fst[of s r' b])
      using ‹inv -› apply (force simp: inv-def)+
      using ‹- ≤ - − -› min-block-gt-overhead apply simp
      by fact+
    show b1 ∉ free-blocks conf s − {b} ∪ {b2}
      apply auto
      using split-fst-snd-neq[OF ‹wf-block b›, OF split-b] apply blast
      using ‹b1 ∉ -› by simp
  next
    assume inv s
    assume (i', j') ∈ ps b ∈ bhdr-matrix-f s i' j' find-suitable-blocks-opt (i, j)
s = Some ps
    hence j' < sl conf
      using  suitable-blocks-j-lt-sl less-imp-le-nat by blast
    hence b ∈ free-blocks conf s
      unfolding free-blocks-def
      using ‹b ∈ -› by blast
    assume mapping-search conf r = (r', i, j)
    hence r' > 0
      using map-search-r-gt-0[OF mbiggerl] min-block-gt-overhead
      by (metis diff-self-eq-0 less-imp-diff-less)
    assume split-block r' b = (b1, b2)
    hence split-b:b1 = fst(split-block r' b) b2 = snd(split-block r' b)
      by simp+
    have b2 ∉ alloced-bhdr-s s
      apply (rule split-free-not-alloced-snd)
      using ‹inv -› apply (force simp: inv-def)+
      by fact+
    show alloced-bhdr-s s ∩ (free-blocks conf s − {b} ∪ {b2}) = {}
      apply auto
      using ‹- ∉ -› ‹b2 = -› apply simp
      using ‹inv -›
      unfolding inv-def disjoint-free-non-free-def by blast
  qed
done
```

```
    subgoal
      unfolding remove-elem-from-matrix-def Let-def
      apply auto
      apply (rule alloc-free-non-free-disjoint)
      apply (auto simp: inv-def)
      using suitable-blocks-j-lt-sl by auto
    done
  done

thm prod-injects(2)
end

context begin — disjoint memory set

lemma disjoint-mem-sym: disjoint-memory a b ⟹ disjoint-memory b a
  unfolding disjoint-memory-def by blast

lemma alloc-disjoint-memory-set:
  disjoint-memory-set s ⟹ wf-adjacency-list s ⟹ j < sl conf ⟹ b ∈ bhdr-matrix-f
s i j ⟹
    disjoint-memory-set
          (s(|bhdr-matrix-f := set-bhdr-matrix (bhdr-matrix-f s) i j (Set.remove b
(bhdr-matrix-f s i j)),
             alloced-bhdr-s := insert b (alloced-bhdr-s s)|))
  unfolding disjoint-memory-set-def all-blocks-def
  apply (subst free-blk-mat-s-eq)+
  apply (simp del: sl-def)
  apply (subst free-blocks-remove-is-minus)
  apply assumption+
  apply (subst free-blocks-remove-is-minus)
  apply assumption+
  apply auto
  subgoal for x
    apply (drule spec[of - b])
    apply (drule spec[of - x])
    apply auto
    unfolding free-blocks-def sl-def
    using Union-iff by blast
  subgoal for x
    apply (drule spec[of - b])
    apply (drule spec[of - x])
    apply auto
    unfolding free-blocks-def sl-def
    using Union-iff by blast
  subgoal for x
    apply (drule spec[of - b])
    apply (drule spec[of - x])
    apply (auto simp: disjoint-mem-sym)
    unfolding free-blocks-def sl-def
```

using *Union-iff* **by** *blast*
subgoal for *x*
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *x*])
apply (*auto simp*: *disjoint-mem-sym*)
unfolding *free-blocks-def sl-def*
using *Union-iff* **by** *blast*
done

lemma *free-matrix-in-free-block*:
b ∈ *bhdr-matrix-f s i j* ⟹ *j* < *sl conf* ⟹ *b* ∈ *free-blocks conf s*
unfolding *free-blocks-def*
by *blast*

lemma *split-disjoint*:
r > *0* ∨ *s-addr b* > *0* ⟹
b1 = *fst* (*split-block r b*) ⟹
b2 = *snd* (*split-block r b*) ⟹
disjoint-memory b1 b2
unfolding *split-block-def disjoint-memory-def*
by *auto*

lemma *split-disjoint-fst*:
r < *b-size b* ⟹
disjoint-memory f b ⟹
(*b1*, *b2*) = *split-block r b* ⟹
disjoint-memory f b1
unfolding *disjoint-memory-def split-block-def*
apply (*erule disjE*)
subgoal
by *simp*
subgoal
by (*cases b*) *auto*
done

lemma *split-disjoint-snd*:
disjoint-memory f b ⟹
(*b1*, *b2*) = *split-block r b* ⟹
disjoint-memory f b2
unfolding *disjoint-memory-def split-block-def*
apply (*erule disjE*)
subgoal
by *simp*
subgoal
by (*cases b*) *auto*
done

declare *select-wp*[*wp*]
lemma *inv-malloc-disjoint-memory-set*:⦃λσ. *inv* σ⦄ (*malloc r*) ⦃λn σ. *disjoint-memory-set*

σ ⟫
unfolding *malloc-def Let-def remove-block-def*
apply *wpsimp*
subgoal for *s r′ i j*
apply *auto*
apply (*simp add*: *inv-def*)
subgoal for *ps i′ j′ b b1 b2*
unfolding *disjoint-memory-set-def all-blocks-def remove-elem-from-matrix-def Let-def add-block-def*
apply (*split prod.splits*)
apply *clarsimp*
apply (*subst* (*asm*) *free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *simp*
apply (*rule refl*)
apply (*subst* (*asm*) *free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *simp*
apply (*rule refl*)
subgoal for *ia ja xb yb*
apply (*drule free-matrix-in-free-block*)
using *suitable-blocks-j-lt-sl* **apply** *blast*
apply *auto*
unfolding *inv-def disjoint-memory-set-def all-blocks-def disjoint-free-non-free-def*
subgoal
apply (*rule split-disjoint*[*of r′ b*])
using *map-search-r-gt-0*[*OF mbiggerl*,*of r′ i j r*]
using *min-block-gt-overhead* **by** *auto*
subgoal
apply (*rule disjoint-mem-sym*)
apply (*rule split-disjoint-fst*[*of r′ b*])
using *min-block-gt-overhead*
by *force*+
subgoal
apply (*rule disjoint-mem-sym*)
apply (*rule split-disjoint-fst*[*of r′ b*])
using *min-block-gt-overhead*
by *force*+
subgoal
apply (*rule disjoint-mem-sym*)
apply (*rule split-disjoint*[*of r′ b*])
using *map-search-r-gt-0*[*OF mbiggerl*,*of r′ i j r*]
using *min-block-gt-overhead* **by** *auto*

```
      subgoal
        apply (rule split-disjoint-fst[of r' b])
        using min-block-gt-overhead
        by force+
      subgoal
        apply (rule split-disjoint-fst[of r' b])
        using min-block-gt-overhead
        by force+
      subgoal
        apply (rule disjoint-mem-sym)
        apply (rule split-disjoint-snd)
        apply blast
        by (rule sym)
      subgoal
        apply (rule disjoint-mem-sym)
        apply (rule split-disjoint-snd)
        apply blast
        by (rule sym)
      subgoal
        apply (rule split-disjoint-snd)
        apply blast
        by (rule sym)
      subgoal
        apply (rule split-disjoint-snd)
        apply blast
        by (rule sym)
      by blast+
    done
  subgoal for ps i' j' b
    unfolding remove-elem-from-matrix-def Let-def
    apply auto
    apply (rule alloc-disjoint-memory-set)
    apply (auto simp: inv-def)
    using suitable-blocks-j-lt-sl by auto
  done
 done
end

context begin — inv wf

lemma alloc-no-split-all-blocks:
  s' = s(|bhdr-matrix-f := set-bhdr-matrix (bhdr-matrix-f s) i j (Set.remove b
(bhdr-matrix-f s i j)),
        alloced-bhdr-s := insert b (alloced-bhdr-s s)|) ⟹
  wf-adjacency-list s ⟹
  j < sl conf ⟹
  b ∈ bhdr-matrix-f s i j ⟹
  all-blocks conf s' = all-blocks conf s
 unfolding all-blocks-def
```

apply *hypsubst-thin*
apply (*subst free-blk-mat-s-eq*)
apply *clarsimp*
apply (*subst free-blocks-remove-is-minus*)
by (*auto simp*: *free-matrix-in-free-block*)

lemma *split-wf-fst*:
wf-block b $\Longrightarrow$
split-block r b = (*b1*, *b2*) $\Longrightarrow$
r $\geq$ *min-block conf* $\Longrightarrow$
r $\leq$ *b-size b* $\Longrightarrow$
wf-block b1
unfolding *wf-block-def split-block-def*
apply (*cases b*)
apply *auto*
using *min-block-gt-overhead* **by** *linarith*

lemma *split-wf-snd*:
wf-block b $\Longrightarrow$
split-block r b = (*b1*, *b2*) $\Longrightarrow$
r $\geq$ *min-block conf* $\Longrightarrow$
b-size b $-$ *r* $\geq$ *min-block conf* $\Longrightarrow$
wf-block b2
unfolding *wf-block-def split-block-def*
apply (*cases b*)
apply *auto*
using *min-block-gt-overhead* **by** *linarith*

lemma *inv-malloc-wf*:$\{\!|\lambda\sigma.\ inv\ \sigma|\!\}$ (*malloc r*) $\{\!|\lambda n\ \sigma.\ wf\ \sigma\ |\!\}$
unfolding *malloc-def Let-def remove-block-def*
apply *wpsimp*
subgoal for *s r' i j*
apply *auto*
apply (*simp add*: *inv-def*)
subgoal for *ps i' j' b b1 b2*
unfolding *add-block-def Let-def remove-elem-from-matrix-def*
apply (*split prod.splits*)
apply *auto*
apply (*rename-tac ia ja*)
unfolding *wf-def all-blocks-def*
apply *clarsimp*
apply (*rule conjI*)
subgoal — well formed b1
apply (*rule split-wf-fst*)
defer
apply *assumption*
using *map-search-r-ge-minblock*[*OF mbiggerl*] **apply** *metis*
using *min-block-gt-overhead* **apply** *simp*
using *find-opt-is-free inv-def wf-def all-blocks-def* **by** *blast*

```
    apply (subst free-blocks-insert-is-union)
    using mapping-insert-r-in-l2-set[OF mbiggerl] apply metis
    apply (subst free-blocks-remove-is-minus)
    apply (simp add: inv-def)
    apply assumption
    using suitable-blocks-j-lt-sl apply blast
    apply (rule refl)
    apply auto
    subgoal — well formed b2
      apply (rule split-wf-snd)
      defer
      apply assumption
      using map-search-r-ge-minblock[OF mbiggerl] apply metis
      using min-block-gt-overhead apply simp
      using find-opt-is-free inv-def wf-def all-blocks-def by blast
    unfolding inv-def wf-def all-blocks-def by auto
  subgoal for ps i' j' b
    unfolding remove-elem-from-matrix-def Let-def
    apply auto
    unfolding wf-def
    apply (subst alloc-no-split-all-blocks)
    apply (rule refl)
    apply (simp add: inv-def)
    using suitable-blocks-j-lt-sl apply blast
    by (auto simp: inv-def wf-def)
  done
done

end

context begin — inv wf adjacency list

lemma add-block-wf-adjacency:
  wf-adjacency-list s ⟹
   wf-adjacency-list (add-block b s)
  by (auto split:prod.splits
          simp: wf-adjacency-list-def add-block-def Let-def
                insert-block-bhdr-matrix-def set-bhdr-matrix-def
                tlsf-matrix-def mapping-insert-r-in-l2-set mbiggerl)

lemma inv-malloc-wf-adjacency-list:⦃λσ. inv σ⦄ (malloc r ) ⦃λn σ. wf-adjacency-list
σ ⦄
  unfolding malloc-def Let-def remove-block-def
  apply wpsimp
  subgoal
    apply auto
    apply (simp add: inv-def)
    subgoal
      apply (rule add-block-wf-adjacency)
```

by (*auto simp*: *inv-def wf-adjacency-list-def set-bhdr-matrix-def*
tlsf-matrix-def remove-elem-from-matrix-def Let-def)
subgoal
by (*auto simp*: *inv-def wf-adjacency-list-def set-bhdr-matrix-def*
tlsf-matrix-def remove-elem-from-matrix-def Let-def)
done
done

end

context begin

lemma *wf-bitmap s* — this is not actually an invariant. Given the definition of bitmap, the property always holds
by (*auto simp*: *wf-bitmap-def wf-bitmap1-def wf-bitmap2-def*
fl-bitmap-f-def sl-bitmap-f-def)

end

context begin

lemma *split-block-size-t*:
sz = *block-t-size b* ⟹
split-block r b = (*b1*, *b2*) ⟹
min-block conf ≤ *r* ⟹
min-block conf ≤ *b-size b* − *r* ⟹
sz1 = *block-t-size b1* ⟹
sz2 = *block-t-size b2* ⟹
sz = *sz1* + *sz2*
unfolding *split-block-def*
apply (*cases b*)
apply *auto*
apply (*cases r*)
using *dual-order.strict-trans min-block-gt-overhead not-less oh-gt-0* **apply** *blast*
apply *auto*
apply (*subgoal-tac x2* ≥ (*x1* + *nat* + *overhead conf*))
apply *linarith*
using *min-block-gt-overhead* **by** *linarith*

lemma *split-neq-fst*:
split-block r b = (*b1*, *b2*) ⟹ *r* ≠ *b-size b* ⟹ *r* > *0* ⟹ *b* ≠ *b1*
by (*cases b*) (*auto simp*: *split-block-def*)

lemma *split-neq-snd*:
split-block r b = (*b1*, *b2*) ⟹ *r* > *0* ⟹ *b* ≠ *b2*
by (*cases b*) (*auto simp*: *split-block-def*)

lemma *sum-block-f-commute*:
comp-fun-commute (λ*b*. (+) (*block-t-size b*))

unfolding *comp-fun-commute-def comp-def*
by *auto*

lemma *minus-transposition*:
$b \geq c \Longrightarrow a + c = (b::nat) \Longrightarrow a = b - c$
by *simp*

lemma *Un-is-insert*: $A \cup \{b\} = insert\ b\ A$
by *simp*

lemma *all-block-is-finite*: *all-block-mem-size s* $\Longrightarrow$ *finite* (*all-blocks conf s*)
unfolding *all-block-mem-size-def sum-block-def*
using *fold-infinite total-mem-gt-0* **by** *force*

lemma *inv-malloc-all-block-mem-size*:$\{\!|\lambda\sigma.\ inv\ \sigma|\!\}$ (*malloc r*) $\{\!|\lambda n\ \sigma.\ all\text{-}block\text{-}mem\text{-}size\ \sigma|\!\}$
unfolding *malloc-def Let-def remove-block-def*
apply *wpsimp*
subgoal for *s r' i j*
apply *auto*
apply (*simp add*: *inv-def*)
subgoal for *ps i' j' b b1 b2*
unfolding *add-block-def Let-def remove-elem-from-matrix-def*
apply (*auto split*: *prod.splits*)
unfolding *all-block-mem-size-def all-blocks-def*
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
apply (*metis mapping-insert-r-in-l2-set mbiggerl*)
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *blast*
apply (*rule refl*)
subgoal for *ia ja*
proof −
assume *inv s*
hence *all-block-mem-size s disjoint-free-non-free s wf s disjoint-memory-set s*
by (*simp add*: *inv-def*)+
from ‹*all-block-mem-size s*›
have *mem-size conf* = *sum-block* (*free-blocks conf s* ∪ *alloced-bhdr-s s*)
unfolding *all-block-mem-size-def all-blocks-def*
by *simp*
assume *find-suitable-blocks-opt* (*i*, *j*) *s* = *Some ps*
mapping-search conf r = (*r'*, *i*, *j*)
(*i'*, *j'*) ∈ *ps b* ∈ *bhdr-matrix-f s i' j'*
have *r'* > *0*
using *map-search-r-gt-0*[*OF mbiggerl* - ‹*mapping-search conf r* = -›[*symmetric*]]
using *min-block-gt-overhead* **by** *linarith*

have *min-block conf* $\leq r'$
using *map-search-r-ge-minblock*[*OF mbiggerl* ‹*mapping-search conf r* = -›[*symmetric*]] **.**
assume *min-block conf* $\leq$ *b-size b* $- r'$
hence $r' \neq$ *b-size b*
using *min-block-gt-overhead* **by** *simp*
assume *split-block* r' *b* = (*b1*, *b2*)

have *finite* (*all-blocks conf s*)
apply (*rule all-block-is-finite*)
using ‹*inv* -› **by** (*simp add*: *inv-def*)

have *b* $\in$ *free-blocks conf s*
apply (*rule free-matrix-in-free-block*)
apply *fact*
using ‹(*i'*, *j'*) $\in$ *ps*› ‹*find-suitable-blocks-opt* (*i*, *j*) *s* = *Some ps*› *suitable-blocks-j-lt-sl* **by** *presburger*
moreover have *b* $\notin$ *alloced-bhdr-s s*
using *calculation* ‹*disjoint-free-non-free s*›
unfolding *disjoint-free-non-free-def* **by** *blast*
moreover have *b1* $\neq$ *b b2* $\neq$ *b*
using *split-neq-fst*[*OF* ‹*split-block* - - = -› ‹$r' \neq$ -› ‹$r' > 0$›]
using *split-neq-snd*[*OF* ‹*split-block* - - = -› ‹$r' > 0$›]
by *auto*
ultimately have *sum-block* (*insert b1* (*free-blocks conf s* $-$ {*b*} $\cup$ {*b2*} $\cup$ *alloced-bhdr-s s*)) =
sum-block (*all-blocks conf s* $\cup$ {*b1*} $\cup$ {*b2*} $-$ {*b*})
unfolding *all-blocks-def*
by (*blast intro*: *arg-cong*[**where** *f* = *sum-block*])
also have ... = *sum-block* (*all-blocks conf s*) + *block-t-size b1* + *block-t-size b2* $-$ *block-t-size b*
apply (*rule add-implies-diff*)
apply (*subst add.commute*)
unfolding *sum-block-def Un-is-insert*
apply (*subst Finite-Set.comp-fun-commute.fold-rec*[*OF sum-block-f-commute* ,*of* - *b 0*, *THEN sym*])
using ‹*finite* -› **apply** *blast*
using ‹*b* $\in$ *free-blocks conf s*›
apply (*simp add*: *all-blocks-def*)
apply (*subst comp-fun-commute.fold-insert*)
prefer *4*
apply (*subst comp-fun-commute.fold-insert*)
using *sum-block-f-commute* **apply** *simp*
apply *fact*
defer
apply *simp*
using ‹*finite* -› **apply** *blast*
subgoal
apply *auto*

apply (*metis* ‹*b1* ≠ *b*› ‹*split-block r′ b* = (*b1*, *b2*)› *bhdr-t.collapse bhdr-t.inject fst-conv snd-conv split-block-def*)
using *split-free-not-exist-snd*[*OF* ‹*disjoint-memory-set s*› ‹*wf s*› ‹*split-block r′ b* = (*b1*, *b2*)› ‹*b* ∈ *free-blocks conf s*›]
by *simp*
subgoal
by (*rule split-free-not-exist-fst*[*OF* ‹*disjoint-memory-set s*› ‹*wf s*› ‹*r′* ≠ -› ‹*split-block r′ b* = (*b1*, *b2*)› ‹*b* ∈ *free-blocks conf s*›])
done
also have ... = *mem-size conf* + *block-t-size b1* + *block-t-size b2* − *block-t-size b*
by (*metis* (*full-types*) ‹*all-block-mem-size s*› *all-block-mem-size-def*)
finally have *sum-block* (*insert b1* (*free-blocks conf s* − {*b*} ∪ {*b2*} ∪ *alloced-bhdr-s s*)) = *mem-size conf* + *block-t-size b1* + *block-t-size b2* − *block-t-size b* .
moreover have *block-t-size b1* + *block-t-size b2* = *block-t-size b*
apply (*rule split-block-size-t*[*of* - *b r′ b1 b2*,*symmetric*])
apply *auto*
by *fact*+
ultimately show *sum-block* (*insert b1* (*free-blocks conf s* − {*b*} ∪ {*b2*} ∪ *alloced-bhdr-s s*)) = *mem-size conf*
by *simp*
qed
done
subgoal for *ps i′ j′ b*
unfolding *remove-elem-from-matrix-def Let-def*
apply *auto*
unfolding *all-block-mem-size-def all-blocks-def*
apply (*subst free-blk-mat-s-eq*)
apply *clarsimp*
apply (*subst free-blocks-remove-is-minus*)
apply (*simp add*: *inv-def*)
apply *assumption*
using *suitable-blocks-j-lt-sl* **apply** *blast*
using *inv-def all-block-mem-size-def all-blocks-def*
by (*metis Un-insert-left find-opt-is-free insert-Diff*)
done
done

end
lemma *hoare-conjI1*:
⟦ {|*P*|} *f* {|*R*|}; {|*P*|} *f* {|*Q*|} ⟧ ⟹ {|*P*|} *f* {|λ*r s*. *Q r s* ∧ *R r s*|}
unfolding *valid-def* **by** *blast*

theorem *inv-malloc*: {|*inv*|} (*malloc r*) {|λ*n*. *inv* |}
unfolding *inv-def*
apply (*rule hoare-conjI1*)+
using *inv-malloc-no-split-memory inv-malloc-disjoint-free-non-free*

inv-malloc-disjoint-memory-set inv-malloc-wf inv-malloc-wf-adjacency-list
inv-malloc-all-block-mem-size
by (*auto simp add*:*inv-def*)

context
begin

lemma *suc-freeD*: *suc-hdr-free-s conf b s* = *Some b'* ⟹ *wf s* ⟹ *disjoint-memory-set s* ⟹ *b'* ∈ *free-blocks conf s* ∧ *e-addr b* + *1* + *overhead conf* = *s-addr b'*
unfolding *suc-hdr-free-s-def*
apply (*cases b*)
apply (*clarsimp split*: *if-splits simp*: *Let-def*)
subgoal for *bs be e'*
proof −
let *?P* = λ*x*. (∃ *e-addr'*. *x* = *Bhdr* (*Suc* (*be* + *overhead conf*)) *e-addr'*) ∧ *x* ∈ *free-blocks conf s*
assume *wf s disjoint-memory-set s*
assume *b* = *Bhdr bs be b'* = (*THE x*. *?P x*)
assume *Bhdr* (*Suc* (*be* + *overhead conf*)) *e'* ∈ *free-blocks conf s*
have ∃! *x*. *?P x*
apply *rule*
apply *rule*
apply *rule*
apply *rule*
apply *fact*
subgoal for *b*
apply *auto*
apply (*rule ccontr*)
apply (*insert* ‹*wf s*› ‹*disjoint-memory-set s*› ‹- ∈ -›)
unfolding *wf-def*
apply (*frule bspec*[*of* - - *b*])
using *all-blocks-def* **apply** *blast*
apply (*drule bspec*[*of* - - *Bhdr* (*Suc* (*be* + *overhead conf*)) *e'*])
using *all-blocks-def* **apply** *blast*
unfolding *disjoint-memory-set-def*
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *Bhdr* (*Suc* (*be* + *overhead conf*)) *e'*])
apply (*clarsimp simp*: *all-blocks-def*)
using *wf-block-def disjoint-memory-def*
by (*metis add-lessD1 bhdr-t.sel*(*1*) *not-le*)
done
thus *?thesis*
using *theI'*[*of ?P*]
by (*metis* (*no-types*, *lifting*) *bhdr-t.sel*(*1*))
qed
done

lemma *prev-freeD*: *prev-free-hdr-s conf b s* = *Some b'* ⟹ *wf s* ⟹ *disjoint-memory-set s* ⟹ *b'* ∈ *free-blocks conf s* ∧ *e-addr b'* + *1* + *overhead conf* = *s-addr b*

unfolding *prev-free-hdr-s-def*
apply (*cases b*)
apply (*clarsimp simp*: *Let-def split*: *if-splits*)
subgoal for *be b′s b′e*
proof −
let *?P* = λ*x*. (∃ *s-addr′*. *x* = *Bhdr s-addr′ b′e*) ∧ *x* ∈ *free-blocks conf s*
assume *wf s disjoint-memory-set s*
assume *b* = *Bhdr* (*Suc* (*b′e* + *overhead conf*)) *be b′* = (*THE x*. *?P x*)
assume *Bhdr b′s b′e* ∈ *free-blocks conf s*
have ∃! *x*. *?P x*
apply *rule*
apply *rule*
apply *rule*
apply *rule*
apply *fact*
subgoal for *b*
apply *auto*
apply (*rule ccontr*)
apply (*insert* ‹*wf s*› ‹*disjoint-memory-set s*› ‹- ∈ -›)
unfolding *wf-def*
apply (*frule bspec*[*of* - - *b*])
using *all-blocks-def* **apply** *blast*
apply (*drule bspec*[*of* - - *Bhdr b′s b′e*])
using *all-blocks-def* **apply** *blast*
unfolding *disjoint-memory-set-def*
apply (*drule spec*[*of* - *b*])
apply (*drule spec*[*of* - *Bhdr b′s b′e*])
apply (*clarsimp simp*: *all-blocks-def*)
using *wf-block-def disjoint-memory-def*
by *simp*
done
thus *?thesis*
using *theI′*[*of ?P*]
by (*metis* (*no-types*, *lifting*) *bhdr-t.sel*(*2*))
qed
done

lemma *free-blocks-in-matrix*:
wf-adjacency-list s ⟹ *b* ∈ *free-blocks conf s* ⟹ (*i*, *j*) = *mapping-insert conf* (*b-size b*) ⟹ *b* ∈ *bhdr-matrix-f s i j*
unfolding *wf-adjacency-list-def free-blocks-def tlsf-matrix-def*
apply (*auto simp del*: *sl-def*)
subgoal for *i′ j′*
apply (*drule spec*[*of* - *i′*])
apply (*drule spec*[*of* - *j′*])
apply (*clarsimp simp del*: *sl-def*)
apply (*drule spec*[*of* - *b*])
apply (*clarsimp simp del*: *sl-def*)
apply (*drule mapping-insert-r-in-l2-set*[*OF mbiggerl*])

```
    using l2-set-disj[rule-format] mbiggerl
    by (metis disjoint-iff-not-equal)
  done

lemma get-alloced-is-alloced: wf s ⟹ disjoint-memory-set s ⟹ block-alloced addr
s ⟹ b = get-alloced-block addr s ⟹ b ∈ alloced-bhdr-s s
proof −
  assume disjoint-memory-set s wf s
  assume block-alloced addr s
  then obtain e where (Bhdr addr e) ∈ (alloced-bhdr-s s)
    unfolding block-alloced-def by blast
  assume b = get-alloced-block addr s
  have ∃!e-addr. (Bhdr addr e-addr) ∈ (alloced-bhdr-s s)
    apply rule
    apply fact
proof −
fix e-addr :: nat
  assume a1: Bhdr addr e-addr ∈ alloced-bhdr-s s
  have f2: addr ≤ e
   by (metis (no-types) Un-iff ‹Bhdr addr e ∈ alloced-bhdr-s s› ‹wf s› all-blocks-def
bhdr-t.sel(1) bhdr-t.sel(2) wf-def wf-block-def)
have f3: addr ≤ e-addr
 using a1 by (metis (no-types) Un-iff ‹wf s› all-blocks-def bhdr-t.sel(1) bhdr-t.sel(2)
wf-def wf-block-def)
  have Bhdr addr e-addr = Bhdr addr e ∨ disjoint-memory (Bhdr addr e) (Bhdr
addr e-addr)
    using a1 ‹Bhdr addr e ∈ alloced-bhdr-s s› ‹disjoint-memory-set s› all-blocks-def
disjoint-memory-set-def by auto
  then show e-addr = e
    using f3 f2 by (simp add: disjoint-memory-def)
qed
  thus b ∈ alloced-bhdr-s s
    using ‹b = -›
    unfolding get-alloced-block-def
    by (smt theI)
qed

lemma remove-not-member-id: x ∉ S ⟹ S − {x} = S
  by simp

lemma suc-free-none-remove:
 suc-hdr-free-s conf b s = None ⟹ suc-hdr-free-s conf b (remove-elem-from-matrix
b′ i j s) = None
  unfolding remove-elem-from-matrix-def Let-def
  apply (cases b′ ∈ bhdr-matrix-f s i j)
  subgoal
    unfolding set-bhdr-matrix-def remove-def
    unfolding suc-hdr-free-s-def
    apply (cases b)
```

apply (*auto simp*: *Let-def split*: *if-splits*)
subgoal for *bs be be'*
apply (*drule spec*[*of* - *Bhdr* (*Suc* (*be* + *overhead conf*)) *be'*])
apply (*erule disjE*)
apply *blast*
unfolding *free-blocks-def*
by (*auto split*:*if-splits*)
done
subgoal
using *remove-not-member-id set-bhdr-matrix-def*
by (*simp add*: *remove-def*)
done

lemma *suc-free-none-equiv1*:
suc-hdr-free-s conf b (*add-block b' s*) = *None* ⟹ *suc-hdr-free-s conf b s* = *None*
unfolding *add-block-def suc-hdr-free-s-def insert-block-bhdr-matrix-def set-bhdr-matrix-def free-blocks-def*
apply (*cases b*)
apply (*auto simp*: *Let-def split*: *if-splits prod.splits*)
by (*metis insert-iff*)

lemma *suc-free-none-equiv2*:
suc-hdr-free-s conf b s = *None* ⟹ *e-addr b* = *e-addr b'* ⟹ *suc-hdr-free-s conf b' s* = *None*
unfolding *suc-hdr-free-s-def*
apply (*cases b*; *cases b'*)
by (*auto simp*: *Let-def split*: *if-splits*)

lemma *suc-free-none-equiv3*:
suc-hdr-free-s conf b (*remove-elem-from-matrix b' i j s*) = *None* ⟹
e-addr b + *1* + *overhead conf* ≠ *s-addr b'* ⟹ *suc-hdr-free-s conf b s* = *None*
unfolding *suc-hdr-free-s-def*
apply (*cases b*; *cases b'*)
subgoal for *s1 e1 s2 e2*
apply (*auto simp*: *Let-def split*: *if-splits*)
apply (*drule-tac x* = *Bhdr* (*Suc* (*e1* + *overhead conf*)) *e-addr'* **in** *spec*)
apply (*auto simp*: *free-blocks-def remove-elem-from-matrix-def set-bhdr-matrix-def split*: *if-splits*)
by (*metis bhdr-t.sel*(*1*) *member-remove*)
done

lemma *prev-free-none-equiv1*:
prev-free-hdr-s conf b (*add-block b' s*) = *None* ⟹ *prev-free-hdr-s conf b s* = *None*
unfolding *add-block-def prev-free-hdr-s-def insert-block-bhdr-matrix-def set-bhdr-matrix-def free-blocks-def*
apply (*cases b*)
apply (*auto simp*: *Let-def split*: *if-splits prod.splits*)
using *insert-iff* **by** *metis*

lemma *prev-free-none-equiv2*:
prev-free-hdr-s conf b s = None $\Longrightarrow$ *s-addr b = s-addr b'* $\Longrightarrow$ *prev-free-hdr-s conf b' s = None*
unfolding *prev-free-hdr-s-def*
apply (*cases b*; *cases b'*)
by (*auto simp*: *Let-def split*: *if-splits*)

lemma *prev-free-none-equiv3*:
prev-free-hdr-s conf b (*remove-elem-from-matrix b' i j s*) = *None* $\Longrightarrow$
e-addr b' + 1 + overhead conf $\neq$ *s-addr b* $\Longrightarrow$ *prev-free-hdr-s conf b s = None*
unfolding *prev-free-hdr-s-def*
apply (*cases b*; *cases b'*)
subgoal
apply (*auto simp*: *Let-def split*: *if-splits*)
apply (*drule-tac x = Bhdr s-addr' e-addr'* **in** *spec*)
apply (*auto simp*: *free-blocks-def remove-elem-from-matrix-def set-bhdr-matrix-def split*: *if-splits*)
by (*metis bhdr-t.sel*(*2*) *member-remove*)
done

lemma *wf-add-block-preserve*:
wf s $\Longrightarrow$ *wf-block b* $\Longrightarrow$ *wf* (*add-block b s*)
unfolding *add-block-def*
apply (*auto split*: *prod.splits*)
unfolding *wf-def all-blocks-def*
apply *clarsimp*
apply (*erule disjE*)
apply (*subst* (*asm*) *free-blocks-insert-is-union*)
apply (*metis mapping-insert-r-in-l2-set mbiggerl*)
using *free-blk-mat-s-eq* **by** *auto*

lemma *wf-remove-preserve*:
wf s $\Longrightarrow$ *wf* (*remove-elem-from-matrix b i j s*)
unfolding *remove-elem-from-matrix-def wf-def set-bhdr-matrix-def all-blocks-def free-blocks-def*
apply *auto* **by** *blast*

lemma *wf-preserve-3*:
wf s $\Longrightarrow$ *wf-block b* $\Longrightarrow$ *wf* (*add-block b* (*remove-elem-from-matrix b' i j s*))
apply (*rule wf-add-block-preserve*)
by (*rule wf-remove-preserve*)

lemma *sum-of-two-elems*:$x \neq y \Longrightarrow$ *sum f* $\{x,y\}$ = *f x + f y*
by *simp*

lemma *sum-of-three-elems*:$x \neq y \Longrightarrow x \neq z \Longrightarrow y \neq z \Longrightarrow$ *sum f* $\{x,y,z\}$ = *f x + f y + f z*
proof −

assume *a1*: $x \neq y$
assume *a2*: $x \neq z$
assume *a3*: $y \neq z$
have *f4*: $\forall A\ a.\ \mathit{infinite}\ A \vee \mathit{finite}\ (\mathit{insert}\ (a::'a)\ A)$
by (*meson finite.insertI*)
have *f5*: *finite* $\{y\}$
by *blast*
have $f\ z + \mathit{sum}\ f\ \{x, y\} = f\ x + f\ y + f\ z$
using *a1* **by** (*simp add*: *linordered-field-class.sign-simps*(*2*))
then show *?thesis*
using *f5 f4 a3 a2* **by** (*metis insertE insert-commute singletonD sum.insert*)
qed

lemma *all-blocks-size-gt-two-blocks*:
$\mathit{sum\text{-}block}\ S = a \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow x \neq y \Longrightarrow \mathit{finite}\ S \Longrightarrow$
$\mathit{block\text{-}t\text{-}size}\ x + \mathit{block\text{-}t\text{-}size}\ y \leq a$
unfolding *sum-block-def*
apply (*subst* (*asm*) *sum.eq-fold*[*unfolded comp-def*, *THEN sym*])
apply (*subst sum-of-two-elems*[**where** f = *block-t-size*, *symmetric*])
apply *assumption*
apply *hypsubst*
apply (*rule sum-mono2*)
by *auto*

lemma *all-blocks-size-gt-three-blocks*:
$\mathit{sum\text{-}block}\ S = a \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow z \in S \Longrightarrow$
$x \neq y \Longrightarrow x \neq z \Longrightarrow y \neq z \Longrightarrow \mathit{finite}\ S \Longrightarrow$
$\mathit{block\text{-}t\text{-}size}\ x + \mathit{block\text{-}t\text{-}size}\ y + \mathit{block\text{-}t\text{-}size}\ z \leq a$
unfolding *sum-block-def*
apply (*subst* (*asm*) *sum.eq-fold*[*unfolded comp-def*, *THEN sym*])
apply (*subst sum-of-three-elems*[**where** f = *block-t-size*, *symmetric*])
apply *assumption+*
apply *hypsubst*
apply (*rule sum-mono2*)
by *auto*

lemma *wf-join-block*:
wf-block b1 $\Longrightarrow$ *wf-block b2* $\Longrightarrow$
e-addr b1 +1 +overhead conf = *s-addr b2* $\Longrightarrow$
block-t-size b1 + block-t-size b2 $\leq$ *mem-size conf* $\Longrightarrow$
wf-block (*join-block b1 b2*)
unfolding *wf-block-def join-block-def*
apply (*cases b1*, *cases b2*)
by *auto*

lemma *join-block-assoc*:
join-block b1 (*join-block b2 b3*) = *join-block* (*join-block b1 b2*) *b3*
unfolding *join-block-def* **by** *simp*

lemma *wf-join-block-2*:
wf-block b1 $\Longrightarrow$ *wf-block b2* $\Longrightarrow$ *wf-block b3* $\Longrightarrow$
e-addr b1 +1 +overhead conf = s-addr b2 $\Longrightarrow$
e-addr b2 +1 +overhead conf = s-addr b3 $\Longrightarrow$
block-t-size b1 + block-t-size b2 + block-t-size b3 $\leq$ *mem-size conf* $\Longrightarrow$
wf-block (*join-block b1* (*join-block b2 b3*))
unfolding *wf-block-def join-block-def*
apply (*cases b1*, *cases b2*, *cases b3*)
by *auto*

lemma *free-blocks-simp*[*simp*]:
free-blocks cfg (*s*(| *alloced-bhdr-s*:= *t* |)) = *free-blocks cfg s*
unfolding *free-blocks-def* **by** *simp*

lemma *free-blocks-simp'*[*simp*]:
free-blocks cfg (*s*(| *alloced-bhdr-s*:= *t*, *bhdr-matrix-f* := *m* |)) = *free-blocks cfg* (*s*(| *bhdr-matrix-f* := *m* |))
unfolding *free-blocks-def* **by** *simp*

lemma *suc-free-simp*[*simp*]:
suc-hdr-free-s cfg b (*s*(| *alloced-bhdr-s*:= *t* |)) = *suc-hdr-free-s cfg b s*
unfolding *suc-hdr-free-s-def* **by** *auto*

lemma *prev-free-simp*[*simp*]:
prev-free-hdr-s cfg b (*s*(| *alloced-bhdr-s*:= *t* |)) = *prev-free-hdr-s cfg b s*
unfolding *prev-free-hdr-s-def* **by** *auto*

lemma *disjoint-add-block*:
$\forall$ *b'* $\in$ *all-blocks conf s*. *disjoint-memory b b'* $\Longrightarrow$
disjoint-memory-set s $\Longrightarrow$ *disjoint-memory-set* (*add-block b s*)
unfolding *add-block-def*
apply (*auto split*: *prod.splits*)
unfolding *disjoint-memory-set-def all-blocks-def*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply *rule*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply *rule*
apply (*subst free-blk-mat-s-eq*[*symmetric*])+
by (*auto simp*: *disjoint-mem-sym*)

lemma *disjoint-remove-block*:
disjoint-memory-set s $\Longrightarrow$
free-blocks conf s' $\leq$ *free-blocks conf s* $\Longrightarrow$
alloced-bhdr-s s' $\leq$ *alloced-bhdr-s s* $\Longrightarrow$
disjoint-memory-set s'
unfolding *disjoint-memory-set-def all-blocks-def*

by *blast*

lemma *remove-free-block-size-decrease*:
free-blocks conf (*s*(| *bhdr-matrix-f* := *set-bhdr-matrix* (*bhdr-matrix-f s*) *i j* ((*bhdr-matrix-f s i j*) − {*b*})|)) ≤ *free-blocks conf s*
apply *rule*
unfolding *set-bhdr-matrix-def free-blocks-def*
by (*auto split*: *if-splits*)

declare *sl-def*[*simp del*]

lemma *join-block-disjoint*:
e-addr b1 + *overhead conf* + *1* = *s-addr b2* ⟹ *wf-block b* ⟹
disjoint-memory b b1 ⟹ *disjoint-memory b b2* ⟹ *disjoint-memory b* (*join-block b1 b2*)
unfolding *join-block-def disjoint-memory-def wf-block-def*
by *auto*

lemma *disjoint-memory-preserve-3*:
disjoint-memory-set s ⟹ *wf-adjacency-list s* ⟹ *wf s* ⟹ *disjoint-free-non-free s* ⟹
e-addr b1 + *1* + *overhead conf* = *s-addr b2* ⟹ *j* < *sl conf* ⟹ *b1* ∈ *bhdr-matrix-f s i j* ⟹ *b2* ∈ *alloced-bhdr-s s* ⟹
disjoint-memory-set (*add-block* (*join-block b1 b2*) (*remove-elem-from-matrix b1 i j* (*s*(|*alloced-bhdr-s* := *alloced-bhdr-s s* − {*b2*}|))))
apply (*rule disjoint-add-block*)
subgoal
apply (*auto simp*: *all-blocks-def remove-elem-from-matrix-def*)
subgoal for *b′*
apply (*subst* (*asm*) *free-blk-mat-s-eq*)
apply *clarsimp*
apply (*subst* (*asm*) *free-blocks-remove-is-minus*)
apply *assumption*+
apply *auto*
unfolding *disjoint-memory-set-def*
apply (*rule disjoint-mem-sym*)
apply (*rule join-block-disjoint*)
apply *simp*
using *wf-def all-blocks-def* **apply** *blast*
apply (*simp add*: *all-blocks-def free-matrix-in-free-block*)
using *all-blocks-def disjoint-free-non-free-def* **by** *auto*
subgoal for *b′*
apply (*rule ccontr*)
apply (*subgoal-tac disjoint-memory b′* (*join-block b1 b2*))
using *disjoint-mem-sym* **apply** *simp*
apply (*rule join-block-disjoint*)
apply *simp*
apply (*simp add*: *all-blocks-def wf-def*)
apply (*metis Un-iff all-blocks-def disjoint-free-non-free-def disjoint-memory-set-def*

free-matrix-in-free-block in-empty-interE)
by (*simp add*: *all-blocks-def disjoint-memory-set-def*)
done
apply (*rule disjoint-remove-block*)
apply *assumption*
subgoal
unfolding *remove-elem-from-matrix-def Let-def*
apply *clarsimp*
using *remove-free-block-size-decrease remove-def*
by (*metis insert-absorb insert-subset*)
subgoal
unfolding *remove-elem-from-matrix-def* **by** *auto*
done

lemma *disjoint-memory-preserve-2*:
disjoint-memory-set s $\Longrightarrow$ *wf-adjacency-list s* $\Longrightarrow$ *wf s* $\Longrightarrow$ *disjoint-free-non-free s* $\Longrightarrow$
e-addr b1 + *1* + *overhead conf* = *s-addr b2* $\Longrightarrow$ *j* < *sl conf* $\Longrightarrow$ *b2* $\in$ *bhdr-matrix-f s i j* $\Longrightarrow$ *b1* $\in$ *alloced-bhdr-s s* $\Longrightarrow$
disjoint-memory-set (*add-block* (*join-block b1 b2*) (*remove-elem-from-matrix b2 i j* (*s*(|*alloced-bhdr-s* := *alloced-bhdr-s s* − {*b1*}|))))
apply (*rule disjoint-add-block*)
subgoal
apply (*auto simp*: *all-blocks-def remove-elem-from-matrix-def*)
subgoal for *b′*
apply (*subst* (*asm*) *free-blk-mat-s-eq*)
apply *clarsimp*
apply (*subst* (*asm*) *free-blocks-remove-is-minus*)
apply *assumption*+
apply *auto*
unfolding *disjoint-memory-set-def*
apply (*rule disjoint-mem-sym*)
apply (*rule join-block-disjoint*)
apply *simp*
using *wf-def all-blocks-def* **apply** *blast*
using *all-blocks-def disjoint-free-non-free-def free-matrix-in-free-block*
by *auto*
subgoal for *b′*
apply (*rule ccontr*)
apply (*subgoal-tac disjoint-memory b′* (*join-block b1 b2*))
using *disjoint-mem-sym* **apply** *simp*
apply (*rule join-block-disjoint*)
apply *simp*
apply (*simp add*: *all-blocks-def wf-def*)
apply (*metis Un-iff all-blocks-def disjoint-memory-set-def*)
by (*metis Un-iff all-blocks-def disjoint-free-non-free-def disjoint-memory-set-def free-matrix-in-free-block in-empty-interE*)
done
apply (*rule disjoint-remove-block*)

apply *assumption*
subgoal
unfolding *remove-elem-from-matrix-def Let-def*
apply *clarsimp*
using *remove-free-block-size-decrease remove-def*
by (*metis insert-absorb insert-subset*)
subgoal
unfolding *remove-elem-from-matrix-def* **by** *auto*
done

lemma *suc-free-noneD*:
suc-hdr-free-s conf b s = None $\Longrightarrow \forall\, b' \in$ *free-blocks conf s. e-addr b + 1 + overhead conf* $\neq$ *s-addr b'*
unfolding *suc-hdr-free-s-def*
apply (*cases b*)
apply (*auto split*: *if-splits*)
by (*metis bhdr-t.collapse*)

lemma *prev-free-noneD*:
prev-free-hdr-s conf b s = None $\Longrightarrow \forall\, b' \in$ *free-blocks conf s. e-addr b' + 1 + overhead conf* $\neq$ *s-addr b*
unfolding *prev-free-hdr-s-def*
apply (*cases b*)
apply (*auto split*: *if-splits*)
by (*metis bhdr-t.collapse*)

lemma *prev-free-some-equiv2*:
prev-free-hdr-s conf b s = Some p $\Longrightarrow$ *s-addr b = s-addr b'* $\Longrightarrow$ *prev-free-hdr-s conf b' s = Some p*
unfolding *prev-free-hdr-s-def*
apply (*cases b*; *cases b'*)
by (*auto split*: *if-splits*)

lemma *prev-free-some-equiv3*:
prev-free-hdr-s conf b (*remove-elem-from-matrix b' i j s*) = *Some p* $\Longrightarrow$
wf s $\Longrightarrow$ *disjoint-memory-set s* $\Longrightarrow$
e-addr b' + 1 + overhead conf $\neq$ *s-addr b* $\Longrightarrow$ *prev-free-hdr-s conf b s = Some p*
apply (*drule prev-freeD*)
apply (*rule wf-remove-preserve*, *simp*)
apply (*rule disjoint-remove-block*, *simp*)
apply (*metis remove-elem-from-matrix-def*
remove-free-block-size-decrease remove-def)
using *remove-elem-from-matrix-def* **apply** *simp*
unfolding *prev-free-hdr-s-def*
apply *auto*
subgoal — not empty
apply (*rule exI*[*of* - *p*])
apply (*auto simp*: *remove-elem-from-matrix-def set-bhdr-matrix-def free-blocks-def*

split: *if-splits*)
apply (*drule spec*[*of* - *bhdr-matrix-f s i j*], *blast*)
by (*drule spec*[*of* - *s-addr p*], *simp*)+
subgoal for *x* — equality
apply *rule*
subgoal — existence
apply *auto*
apply (*drule spec*[*of* - *s-addr p*], *simp*)
apply (*auto simp*: *remove-elem-from-matrix-def set-bhdr-matrix-def free-blocks-def*
split: *if-splits*)
by (*drule spec*[*of* - *bhdr-matrix-f s i j*], *blast*)
subgoal for *y* — uniqueness
apply (*cases b*)
apply *auto*
apply (*auto simp*: *remove-elem-from-matrix-def set-bhdr-matrix-def free-blocks-def*
split: *if-splits*)
using *free-matrix-in-free-block disjoint-memory-set-def wf-def*
disjoint-memory-def wf-block-def
by (*metis Un-iff add-lessD1 all-blocks-def bhdr-t.sel*(*2*) *leD*)+
done
done

lemma *prev-free-eq*:
s-addr b = *s-addr b'* ⟹ *prev-free-hdr-s conf b s* = *prev-free-hdr-s conf b' s*
using *prev-free-some-equiv2 prev-free-none-equiv2*
by (*metis not-None-eq*)

type-synonym *'a set-matrix* = *nat* ⇒ *nat* ⇒ *'a set*
definition *no-overlap-matrix* :: *'a set-matrix* ⇒ (*nat* ⇒ *nat* ⇒ *bool*) ⇒ *bool*
where
no-overlap-matrix s P ≡ (∀ *x i j i' j'*. *P i j* ∧ *P i' j'* ∧ *x* ∈ *s i j* ∧ *x* ∈ *s i' j'* ⟶
i = *i'* ∧ *j* = *j'*)
abbreviation *no-overlap-bhdr-matJ* :: *'a set-matrix* ⇒ *bool*
where
no-overlap-bhdr-matJ s ≡ *no-overlap-matrix s* (λ- *j*. *j* < *sl conf*)

lemma *no-overlap-bhdr-mat*: *wf-adjacency-list s* ⟹ *no-overlap-bhdr-matJ* (*bhdr-matrix-f s*)
unfolding *no-overlap-matrix-def*
using *free-block-no-dup* **by** *blast*

lemma *free-blocks-remove-is-minus'*:
no-overlap-bhdr-matJ mat ⟹ *b* ∈ *mat i j* ⟹ *j* < *sl conf* ⟹
free-blocks-mat conf mat = *f* ⟹
free-blocks-mat conf (*set-bhdr-matrix mat i j* (*Set.remove b* (*mat i j*))) = *f* −
{*b*}
unfolding *free-blocks-mat-def*

```
  apply rule
  subgoal
    apply rule
    unfolding set-bhdr-matrix-def
    apply auto
    subgoal
      by (auto split:if-splits)
    subgoal for ai aj
      apply (auto split:if-splits)
      unfolding no-overlap-matrix-def
      by blast+
    done
  subgoal
    apply rule
    unfolding set-bhdr-matrix-def
    apply auto
    subgoal for x xi xj
    apply (rule exI[of - Set.remove b (mat xi xj)])
    apply auto
      apply (rule exI[of - xi])
      apply auto
      apply (rule exI[of - xj])
      apply auto
      unfolding no-overlap-matrix-def
      apply simp
      apply (rule exI[of - xj])
      by auto
    done
  done

lemma no-overlap-mat-remove:
   no-overlap-bhdr-matJ mat ⟹ no-overlap-bhdr-matJ (set-bhdr-matrix mat i j
(Set.remove b (mat i j)))
  unfolding no-overlap-matrix-def set-bhdr-matrix-def by auto

lemma free4:
  assumes mapping-insert conf (Suc (e-addr bs) − s-addr bp) = (i, j)
        snd (mapping-insert conf (b-size bp)) = jp
        snd (mapping-insert conf (b-size bs)) = js
        wf-adjacency-list s
        bp ∈ bhdr-matrix-f s ip jp bs ∈ bhdr-matrix-f s is js
        wf-block bp wf-block b
        e-addr bp + 1 + overhead conf = s-addr b
        e-addr b + 1 + overhead conf = s-addr bs
 shows free-blocks conf
    (remove-elem-from-matrix bp ip jp
      (remove-elem-from-matrix bs is js
        (s(|alloced-bhdr-s := Set.remove b (alloced-bhdr-s s)|)))
     (|bhdr-matrix-f :=
```

insert-block-bhdr-matrix
(*bhdr-matrix-f*
(*remove-elem-from-matrix bp ip jp*
(*remove-elem-from-matrix bs is js*
(*s*(|*alloced-bhdr-s* := *Set.remove b* (*alloced-bhdr-s s*)|)))))
i j (*Bhdr* (*s-addr bp*) (*e-addr bs*))|)) = *free-blocks conf s* − {*bs*} − {*bp*} ∪ {*Bhdr* (*s-addr bp*) (*e-addr bs*)}
apply (*subst free-blocks-insert-is-union*)
apply (*metis* (*no-types*) *mapping-insert-r-in-l2-set mbiggerl assms*)
unfolding *remove-elem-from-matrix-def*
apply *clarsimp*
apply (*subst free-blocks-remove-is-minus'*)
apply (*rule no-overlap-mat-remove*)
apply (*rule no-overlap-bhdr-mat*)
apply *fact*
using *set-bhdr-matrix-def assms wf-block-def* **apply** *auto*[*1*]
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse assms* **apply** *metis*
apply (*subst free-blocks-remove-is-minus*)
apply *fact*+
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse assms* **apply** *metis*
by *rule*+

lemma *inv-free-no-split-memory* :{|λσ. *inv* σ ∧ *block-alloced addr* σ|} (*free addr*) {|λ*n* σ. *no-split-memory* σ |}
unfolding *free-def*
apply *wp*
unfolding *join-prev-def*
apply *wp*
unfolding *Let-def*
apply (*split prod.splits*)
unfolding *join-block-def*
apply (*intro allI impI*)
apply *wp*
apply (*drule prod-injects*(*2*))
apply (*erule conjE*)
apply *hypsubst*
apply *wp*
apply *wp*
unfolding *join-suc-def*
apply *wp*
unfolding *Let-def*
apply (*split prod.splits*)
apply (*intro allI impI*)
apply *wp*
apply (*drule prod-injects*(*2*))
apply (*erule conjE*)
apply *hypsubst*
apply *wp*

apply *wp*
apply *wp*
apply *wp*
apply *wp*
apply (*erule conjE*)
apply (*split if-splits*)
apply (*intro conjI impI*)
defer
apply *blast*
apply (*split if-splits*)
apply (*intro conjI impI*)
subgoal for *s* — the case when next block is not free
apply (*split if-splits*)
apply *auto*
subgoal — the case when prev block is not free
unfolding *add-block-def no-split-memory-def*
apply *clarsimp*
apply (*split prod.splits*)
apply (*intro conjI impI allI*)
apply (*subst free-blocks-insert-is-union*)
apply (*metis mapping-insert-r-in-l2-set mbiggerl*)
apply (*rule refl*)
apply (*subst free-blk-mat-s-eq*[*symmetric*])
apply *auto*
subgoal
apply (*subgoal-tac wf-block* (*get-alloced-block addr s*))
apply (*simp add*: *wf-block-def*)
by (*metis Un-iff all-blocks-def inv-def wf-def get-alloced-is-alloced*)
apply (*force dest*: *prev-free-noneD*)
apply (*force dest*: *suc-free-noneD*)
by (*metis Suc-eq-plus1 add.assoc no-split-memory-def plus-1-eq-Suc inv-def*)
subgoal for *bp* — the case when prev block is free
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s′* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?s′′* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr ?b*)) (*remove-elem-from-matrix bp ?i ?j ?s′*)
assume *prev-free-hdr-s conf ?b s* = *Some bp*
then obtain *b* **where** *b′*: *the* (*prev-free-hdr-s conf b s*) = *bp*
and *b*: *get-alloced-block addr s* = *b*
by *force*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s no-split-memory s*
by (*auto simp*: *inv-def*)
with *b b′* **have** *bp* ∈ *free-blocks conf s*
using *prev-freeD* ‹- = *Some bp*› **by** *blast*+
hence *bp* ∈ *bhdr-matrix-f s ?i ?j*

by (*simp add*: *invs*(*1*) *free-blocks-in-matrix*)
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*bp* ∈ *free-blocks conf s*› ‹*wf s*› **have** *wfbs*:*wf-block b wf-block bp*
using *all-blocks-def wf-def* **by** *blast*+
assume *suc-hdr-free-s conf ?b s* = *None*
show *no-split-memory ?s''*
unfolding *b*
unfolding *no-split-memory-def add-block-def remove-elem-from-matrix-def*
apply *clarsimp*
apply (*cases mapping-insert conf* (*Suc* (*e-addr b*) − *s-addr bp*))
apply (*rename-tac i' j'*)
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*thin-tac* -)
apply (*subst free-blocks-remove-is-minus*)
apply *fact*+
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse* **apply** *blast*
apply (*rule refl*)
apply (*thin-tac* -)
apply *auto*
subgoal
using ‹- = *Some bp*› *b invs prev-freeD wf-block-def wfbs* **by** *fastforce*
subgoal
by (*metis Suc-eq-plus1* ‹*bp* ∈ *free-blocks conf s*› *invs*(*4*) *add-Suc no-split-memory-def*)
subgoal
using ‹*suc-hdr-free-s* - *?b s* = *None*› *suc-free-noneD*
by (*metis Suc-eq-plus1 add-Suc b*)
subgoal
by (*metis Suc-eq-plus1 add-Suc no-split-memory-def invs*(*4*))
done
qed
done
subgoal for *s* — the case when next block is free
apply (*split if-splits*)
apply (*auto simp*: *join-block-def*)
subgoal for *bs* — the case when next block is not free
proof −
let *?b* = *get-alloced-block addr s*
let *?s'* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s''* = *add-block* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*)
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *b'*:*the* (*suc-hdr-free-s conf b s*) = *bs* **and**
b: *get-alloced-block addr s* = *b*

by *fastforce*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s*
by (*simp add*: *inv-def*)+
hence *invs'*: *wf-adjacency-list ?s'*
unfolding *wf-adjacency-list-def* **by** *simp*
from *b' b* **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
using *suc-freeD invs* ‹- = *Some bs*› **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i ?j*
using *free-blocks-in-matrix* ‹*wf-adjacency-list s*›
by *auto*
hence *bs* ∈ *bhdr-matrix-f ?s' ?i ?j*
by *simp*
assume *block-alloced addr s*
hence *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced*[*OF* ‹*wf s*› ‹*disjoint-memory-set s*› - *b*[*symmetric*]]
by *simp*
with ‹*bs* ∈ *free-blocks conf s*› **have** *wf-block b wf-block bs*
using ‹*wf s*› *wf-def all-blocks-def*
by *blast*+
assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*) = *None*
hence *prev-free-hdr-s conf b s* = *None*
unfolding *b*
apply −
apply (*drule prev-free-none-equiv2*, *simp*)
using ‹- = *s-addr bs*› ‹*wf-block bs*› ‹*wf-block b*› *wf-block-def*
by (*auto dest*: *prev-free-none-equiv3*)
show *no-split-memory ?s''*
unfolding *b no-split-memory-def add-block-def remove-elem-from-matrix-def*
apply *clarsimp*
apply (*cases mapping-insert conf* (*Suc* (*e-addr bs*) − *s-addr b*))
apply (*rename-tac i' j'*)
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*thin-tac* - = -)
apply (*subst free-blocks-remove-is-minus*)
apply *fact*+
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse* **apply** *blast*
apply (*rule refl*)
apply (*thin-tac* - = -)
apply (*auto*)
subgoal
using *b' suc-freeD*[*OF* - ‹*wf* -› ‹*disjoint-memory-set s*›]
using ‹*wf-block b*› ‹*wf-block bs*› *wf-block-def*
by (*smt* ‹*suc-hdr-free-s conf* (*get-alloced-block addr s*) *s* = *Some bs*› *add-leD1 b leD le-trans less-add-Suc1*)

subgoal
using ‹*prev-free-hdr-s conf b s = None*› *prev-free-noneD*
by (*metis Suc-eq-plus1 add.assoc plus-1-eq-Suc*)
subgoal
by (*metis Suc-eq-plus1* ‹*bs ∈ free-blocks conf s*› ‹*inv s*› *add-Suc no-split-memory-def inv-def*)
subgoal
by (*metis Suc-eq-plus1* ‹*inv s*› *add-Suc no-split-memory-def inv-def*)
done
qed
subgoal for *bs bp* — the case when next block is free
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i′* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j′* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s′* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*remove-elem-from-matrix bp ?i ?j* (*remove-elem-from-matrix bs ?i′ ?j′ ?s*))
assume *suc-hdr-free-s conf ?b s = Some bs*
then obtain *b* **where** *bs*: *the* (*suc-hdr-free-s conf b s*) = *bs*
and *b* : *get-alloced-block addr s* = *b*
by *force*

assume *inv s*
hence *invs*: *wf s disjoint-memory-set s no-split-memory s wf-adjacency-list s*
by (*auto simp*: *inv-def*)
with ‹- = *Some bs*› **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
unfolding *b*
using *suc-freeD* **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i′ ?j′*
using *free-blocks-in-matrix invs*(*4*) *prod.collapse* **by** *blast*
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*wf s*› ‹*bs* ∈*free-blocks conf s*› **have** *wf-block b wf-block bs*
unfolding *wf-def all-blocks-def* **by** *simp*+

assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i′ ?j′ ?s*) = *Some bp*
hence *prev-free-hdr-s conf b s = Some bp*
unfolding *b*
apply −
apply (*drule prev-free-some-equiv2*, *simp*)
apply (*drule prev-free-some-equiv3*)
using *wf-def all-blocks-def* ‹*wf s*› **apply** *force*
using *disjoint-memory-set-def all-blocks-def* ‹*disjoint-memory-set s*› **apply**

```
  force
      using ‹- = s-addr bs› ‹wf-block b› ‹wf-block bs› wf-block-def by simp+
    hence bp: the (prev-free-hdr-s conf b s) = bp
        bp ∈ free-blocks conf s
        e-addr bp + 1 + overhead conf = s-addr b
      using prev-freeD invs by force+
    hence bp ∈ bhdr-matrix-f s ?i ?j wf-block bp
      using wf-def all-blocks-def invs free-blocks-in-matrix by force+

    show no-split-memory ?s'
      unfolding b
      unfolding no-split-memory-def add-block-def
      apply (cases mapping-insert conf (b-size (Bhdr (s-addr bp) (e-addr bs))))
      apply clarsimp
      apply (subst free4)
      apply (assumption | thin-tac -; fact | rule)+
      apply auto
      subgoal
        using ‹wf-block bp› ‹wf-block bs› ‹wf-block b›
        using ‹- = s-addr b› ‹- = s-addr bs› wf-block-def
        by auto
      subgoal for b'
        by (metis Suc-eq-plus1 add-Suc bp(2) invs(3) no-split-memory-def)
      subgoal
     by (metis Suc-eq-plus1 ‹bs ∈ free-blocks conf s› add-Suc invs(3) no-split-memory-def)
      subgoal
        by (metis Suc-eq-plus1 add-Suc invs(3) no-split-memory-def)
      done
  qed
  done
done

end

lemma inv-free-disjoint-free-non-free:
  {|λσ. inv σ∧ block-alloced addr σ|} (free addr) {|λn σ. disjoint-free-non-free σ |}
  unfolding free-def join-prev-def join-suc-def join-block-def Let-def
  apply (wp | split prod.splits, intro allI impI, drule prod-injects(2), erule conjE,
clarsimp)+
  apply (erule conjE)
  apply (split if-splits)
  apply (intro conjI impI)
   defer
   apply blast
  apply (split if-splits)
  apply (intro conjI impI)
    apply (split if-splits)
    apply (intro conjI impI)
     apply auto
```

subgoal for *s* — the case when both prev and next are not free
unfolding *inv-def disjoint-free-non-free-def add-block-def*
apply (*cases mapping-insert conf* (*b-size* (*get-alloced-block addr s*)))
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply *rule*
apply (*subst free-blk-mat-s-eq*[*symmetric*]) **by** *auto*
subgoal for *s bp* — the case when prev is not free but next is free
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s′* = (*s*(|*alloced-bhdr-s* := *Set.remove* (*get-alloced-block addr s*) (*alloced-bhdr-s s*)|))
let *?s″* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr ?b*)) (*remove-elem-from-matrix bp ?i ?j ?s′*)
assume *prev-free-hdr-s conf ?b s* = *Some bp*
then obtain *b* **where** *b′*: *the* (*prev-free-hdr-s conf b s*) = *bp*
and *b*: *get-alloced-block addr s* = *b*
by *force*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free s*
by (*auto simp*: *inv-def*)
with *b b′* **have** *bp* ∈ *free-blocks conf s*
using *prev-freeD* ‹- = *Some bp*› **by** *blast*+
hence *bp* ∈ *bhdr-matrix-f s ?i ?j*
by (*simp add*: *invs*(*1*) *free-blocks-in-matrix*)
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*bp* ∈ *free-blocks conf s*› ‹*wf s*› **have** *wfbs*:*wf-block b wf-block bp*
using *all-blocks-def wf-def* **by** *blast*+
assume *suc-hdr-free-s conf ?b s* = *None*

show *disjoint-free-non-free ?s″*
unfolding *disjoint-free-non-free-def add-block-def Let-def*
remove-elem-from-matrix-def
apply (*auto split*: *prod.splits*)
apply (*subst* (*asm*) *free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*thin-tac* -)+
apply (*subst free-blocks-remove-is-minus*)
apply *fact*+
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse* **apply** *metis*
apply *rule*
unfolding *b*
apply *auto*

apply (*metis Un-iff* ‹*b* ∈ *alloced-bhdr-s s*› ‹*inv s*› *all-blocks-def bhdr-t.sel*(*2*) *diff-block-diff-e-addr*)
using *disjoint-free-non-free-def invs*(*4*) **by** *auto*
qed
subgoal for *s bs* — the case when prev is free but next is not free
proof −
let *?b* = *get-alloced-block addr s*
let *?s'* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s''* = *add-block* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*)
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *b'*:*the* (*suc-hdr-free-s conf b s*) = *bs* **and**
b: *get-alloced-block addr s* = *b*
by *fastforce*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free s*
by (*simp add*: *inv-def*)+
hence *invs'*: *wf-adjacency-list ?s'*
unfolding *wf-adjacency-list-def* **by** *simp*
from *b' b* **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
using *suc-freeD invs* ‹- = *Some bs*› **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i ?j*
using *free-blocks-in-matrix* ‹*wf-adjacency-list s*›
by *auto*
hence *bs* ∈ *bhdr-matrix-f ?s' ?i ?j*
by *simp*
assume *block-alloced addr s*
hence *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced*[*OF* ‹*wf s*› ‹*disjoint-memory-set s*› - *b*[*symmetric*]]
by *simp*
with ‹*bs* ∈ *free-blocks conf s*› **have** *wf-block b wf-block bs*
using ‹*wf s*› *wf-def all-blocks-def*
by *blast*+
assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*) = *None*
hence *prev-free-hdr-s conf b s* = *None*
unfolding *b*
apply −
apply (*drule prev-free-none-equiv2*, *simp*)
using ‹- = *s-addr bs*› ‹*wf-block bs*› ‹*wf-block b*› *wf-block-def*
by (*auto dest*: *prev-free-none-equiv3*)
show *disjoint-free-non-free ?s''*
unfolding *disjoint-free-non-free-def add-block-def Let-def*
remove-elem-from-matrix-def
apply (*auto split*: *prod.splits*)

apply (*subst* (*asm*) *free-blocks-insert-is-union*)
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **apply** *metis*
apply (*thin-tac* -)+
apply (*subst free-blocks-remove-is-minus*)
apply *fact*+
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] *prod.collapse* **apply** *metis*
apply *rule*
unfolding *b*
apply *auto*
apply (*metis* ‹*b* ∈ *alloced-bhdr-s s*› ‹*inv s*› *bhdr-t.exhaust-sel same-addr-same-block*)
using *disjoint-free-non-free-def* ‹*disjoint-free-non-free s*› **by** *auto*
qed
subgoal for *s bs bp* — the case when neither prev nor next is free
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i'* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j'* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s'* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*remove-elem-from-matrix bp ?i ?j* (*remove-elem-from-matrix bs ?i' ?j' ?s*))
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *bs*: *the* (*suc-hdr-free-s conf b s*) = *bs*
and *b* : *get-alloced-block addr s* = *b*
by *force*

assume *inv s*
hence *invs*: *wf s disjoint-memory-set s no-split-memory s wf-adjacency-list s disjoint-free-non-free s*
by (*auto simp*: *inv-def*)
with ‹- = *Some bs*› **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
unfolding *b*
using *suc-freeD* **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i' ?j'*
using *free-blocks-in-matrix invs*(*4*) *prod.collapse* **by** *blast*
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*wf s*› ‹*bs* ∈*free-blocks conf s*› **have** *wf-block b wf-block bs*
unfolding *wf-def all-blocks-def* **by** *simp*+

assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i' ?j' ?s*) = *Some bp*
hence *prev-free-hdr-s conf b s* = *Some bp*
unfolding *b*
apply −
apply (*drule prev-free-some-equiv2*, *simp*)

apply (*drule prev-free-some-equiv3*)
using *wf-def all-blocks-def* ‹*wf s*› **apply** *force*
using *disjoint-memory-set-def all-blocks-def* ‹*disjoint-memory-set s*› **apply** *force*
using ‹- = *s-addr bs*› ‹*wf-block b*› ‹*wf-block bs*› *wf-block-def* **by** *simp+*
hence *bp*: *the* (*prev-free-hdr-s conf b s*) = *bp*
bp ∈ *free-blocks conf s*
e-addr bp + *1* + *overhead conf* = *s-addr b*
using *prev-freeD invs* **by** *force+*
hence *bp* ∈ *bhdr-matrix-f s ?i ?j wf-block bp*
using *wf-def all-blocks-def invs free-blocks-in-matrix* **by** *force+*

show *disjoint-free-non-free ?s'*
unfolding *b*
unfolding *disjoint-free-non-free-def add-block-def Let-def*
apply (*split prod.splits*, *intro allI impI*, *clarsimp*)
apply (*subst free4*)
apply (*assumption* | *rule*)+
apply (*thin-tac* -; *fact*)+
unfolding *remove-elem-from-matrix-def*
apply *clarsimp*
apply *auto*
apply (*metis IntI Un-upper2* ‹*bs* ∈ *free-blocks conf s*› ‹*inv s*› *all-blocks-def bhdr-t.sel*(*2*) *contra-subsetD*
diff-block-diff-e-addr disjoint-free-non-free-def empty-iff invs(*5*) *sup-commute*)
using *disjoint-free-non-free-def invs*(*5*) **by** *auto*
qed
done

lemma *inv-free-disjoint-memory-set* :⦃λσ. *inv* σ∧ *block-alloced addr* σ⦄ (*free addr*) ⦃λn σ. *disjoint-memory-set* σ ⦄
unfolding *free-def join-prev-def join-suc-def join-block-def Let-def*
apply (*wp* | *split prod.splits*, *intro allI impI*, *drule prod-injects*(*2*), *erule conjE*, *clarsimp*)+
apply (*split if-splits*)
apply (*intro conjI impI*)
defer
apply *blast*
apply (*split if-splits*)
apply (*intro conjI impI*)
apply (*split if-splits*)
apply (*intro conjI impI*)
apply *auto*
subgoal for *s*
unfolding *inv-def disjoint-memory-set-def add-block-def all-blocks-def*
apply (*cases mapping-insert conf* (*b-size* (*get-alloced-block addr s*)))
apply *clarsimp*
apply (*subst* (*asm*) *insert-is-union-conf*, *assumption*)
apply *rule*

apply (*subst* (*asm*) *free-blk-mat-s-eq*[*symmetric*])
apply (*subst* (*asm*) *insert-is-union-conf*, *assumption*)
apply *rule*
apply (*subst* (*asm*) *free-blk-mat-s-eq*[*symmetric*])
by (*metis UnE Un-is-insert all-blocks-def disjoint-memory-set-def get-alloced-is-alloced insertE*)
subgoal for *s bp*
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s′* = (*s*(|*alloced-bhdr-s* := *Set.remove* (*get-alloced-block addr s*) (*alloced-bhdr-s s*)|))
let *?s″* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr ?b*)) (*remove-elem-from-matrix bp ?i ?j ?s′*)
assume *prev-free-hdr-s conf ?b s* = *Some bp*
then obtain *b* **where** *b′*: *the* (*prev-free-hdr-s conf b s*) = *bp*
and *b*: *get-alloced-block addr s* = *b*
by *force*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free s*
by (*auto simp*: *inv-def*)
with *b b′* **have** *bp* ∈ *free-blocks conf s*
e-addr bp + *1* + *overhead conf* = *s-addr b*
using *prev-freeD* ‹- = *Some bp*› **by** *blast*+
hence *bp* ∈ *bhdr-matrix-f s ?i ?j*
by (*simp add*: *invs*(*1*) *free-blocks-in-matrix*)
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*bp* ∈ *free-blocks conf s*› ‹*wf s*› **have** *wfbs*:*wf-block b wf-block bp*
using *all-blocks-def wf-def* **by** *blast*+
assume *suc-hdr-free-s conf ?b s* = *None*

show *disjoint-memory-set ?s″*
unfolding *remove-def*
apply (*rule disjoint-memory-preserve-3*[*unfolded join-block-def*])
unfolding *b*
apply *fact*+
using *mapping-insert-r-in-l2-set mbiggerl prod.collapse* **apply** *blast*
by *fact*+
qed
subgoal for *s bs*
proof −
let *?b* = *get-alloced-block addr s*
let *?s′* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bs*))

let *?s''* = *add-block* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*)
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *b'*:*the* (*suc-hdr-free-s conf b s*) = *bs* **and**
b: *get-alloced-block addr s* = *b*
by *fastforce*
assume *inv s*
hence *invs*: *wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free s*
by (*simp add*: *inv-def*)+
hence *invs'*: *wf-adjacency-list ?s'*
unfolding *wf-adjacency-list-def* **by** *simp*
from *b' b* **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
using *suc-freeD invs* ‹- = *Some bs*› **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i ?j*
using *free-blocks-in-matrix* ‹*wf-adjacency-list s*›
by *auto*
hence *bs* ∈ *bhdr-matrix-f ?s' ?i ?j*
by *simp*
assume *block-alloced addr s*
hence *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced*[*OF* ‹*wf s*› ‹*disjoint-memory-set s*› - *b*[*symmetric*]]
by *simp*
with ‹*bs* ∈ *free-blocks conf s*› **have** *wf-block b wf-block bs*
using ‹*wf s*› *wf-def all-blocks-def*
by *blast*+
assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i ?j ?s'*) = *None*
hence *prev-free-hdr-s conf b s* = *None*
unfolding *b*
apply −
apply (*drule prev-free-none-equiv2*, *simp*)
using ‹- = *s-addr bs*› ‹*wf-block bs*› ‹*wf-block b*› *wf-block-def*
by (*auto dest*: *prev-free-none-equiv3*)
show *disjoint-memory-set ?s''*
unfolding *b remove-def*
apply (*rule disjoint-memory-preserve-2*[*unfolded join-block-def*])
apply *fact*+
apply (*meson mapping-insert-r-in-l2-set mbiggerl prod.collapse*)
by *fact*+
qed
subgoal for *s bs bp*
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i'* = *fst* (*mapping-insert conf* (*b-size bs*))

let *?j'* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s'* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*remove-elem-from-matrix bp ?i ?j* (*remove-elem-from-matrix bs ?i' ?j' ?s*))
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *bs*: *the* (*suc-hdr-free-s conf b s*) = *bs*
and *b* : *get-alloced-block addr s* = *b*
by *force*

assume *inv s*
hence *invs*: *wf s disjoint-memory-set s no-split-memory s wf-adjacency-list s disjoint-free-non-free s*
by (*auto simp*: *inv-def*)
with ‹- = *Some bs*› **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
unfolding *b*
using *suc-freeD* **by** *blast+*
hence *bs* ∈ *bhdr-matrix-f s ?i' ?j'*
using *free-blocks-in-matrix invs*(*4*) *prod.collapse* **by** *blast*
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*wf s*› ‹*bs* ∈*free-blocks conf s*› **have** *wf-block b wf-block bs*
unfolding *wf-def all-blocks-def* **by** *simp+*

assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i' ?j' ?s*) = *Some bp*
hence *prev-free-hdr-s conf b s* = *Some bp*
unfolding *b*
apply −
apply (*drule prev-free-some-equiv2*, *simp*)
apply (*drule prev-free-some-equiv3*)
using *wf-def all-blocks-def* ‹*wf s*› **apply** *force*
using *disjoint-memory-set-def all-blocks-def* ‹*disjoint-memory-set s*› **apply** *force*
using ‹- = *s-addr bs*› ‹*wf-block b*› ‹*wf-block bs*› *wf-block-def* **by** *simp+*
hence *bp*: *the* (*prev-free-hdr-s conf b s*) = *bp*
bp ∈ *free-blocks conf s*
e-addr bp + *1* + *overhead conf* = *s-addr b*
using *prev-freeD invs* **by** *force+*
hence *bp* ∈ *bhdr-matrix-f s ?i ?j wf-block bp*
using *wf-def all-blocks-def invs free-blocks-in-matrix* **by** *force+*

show *disjoint-memory-set ?s'*
apply (*rule disjoint-add-block*)
subgoal
apply (*auto simp*: *remove-elem-from-matrix-def all-blocks-def*)
subgoal for *b'*
apply (*subst* (*asm*) *free-blk-mat-s-eq*)
apply *clarsimp*

apply (*subst* (*asm*) *free-blocks-remove-is-minus'*)
apply (*rule no-overlap-mat-remove*)
apply (*rule no-overlap-bhdr-mat*)
apply *fact*
subgoal
unfolding *set-bhdr-matrix-def*
apply *auto*
using ‹*bp* ∈ *bhdr-matrix-f s ?i ?j*› **apply** *auto*
using ‹*wf-block bp*› ‹*wf-block b*› ‹*wf-block bs*›
using ‹- = *s-addr b*› ‹- = *s-addr bs*› *wf-block-def* **by** *simp*
using *mapping-insert-r-in-l2-set mbiggerl prod.collapse* **apply** *blast*
apply (*subst remove-is-minus-conf*)
apply *fact+*
apply *simp*
apply *rule*
proof *auto*
assume *b'* ∈ *free-blocks conf s b'* ≠ *bs b'* ≠ *bp*
moreover have *disjoint-memory b' bp*
using ‹*b'* ∈ *free-blocks conf s*› ‹*b'* ≠ *bp*› *all-blocks-def bp*(*2*) *disjoint-memory-set-def invs*(*2*) **by** *auto*
moreover have *disjoint-memory b' bs*
using ‹*b'* ∈ *free-blocks conf s*› ‹*b'* ≠ *bs*› ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def disjoint-memory-set-def invs*(*2*) **by** *force*
moreover have *disjoint-memory b' b*
by (*metis Un-iff* ‹*b* ∈ *alloced-bhdr-s s*› ‹*b'* ∈ *free-blocks conf s*› *all-blocks-def disjoint-free-non-free-def disjoint-memory-set-def in-empty-interE invs*(*2*) *invs*(*5*))
moreover have *wf-block b'*
using ‹*b'* ∈ *free-blocks conf s*› *all-blocks-def invs*(*1*) *wf-def* **by** *force*
ultimately show *disjoint-memory* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) *b'*
using ‹*wf-block b*› ‹*wf-block bs*› ‹*wf-block bp*› *wf-def disjoint-memory-def*
using ‹- = *s-addr bs*› ‹- = *s-addr b*›
by (*smt Suc-eq-plus1 add-Suc add-lessD1 bhdr-t.sel leD le-less-trans not-less-eq-eq wf-block-def*)
qed
subgoal for *b'*
unfolding *b*
proof −
assume *b'* ∈ *alloced-bhdr-s s b'* ≠ *b*
moreover have *disjoint-memory b' bp*
by (*metis Un-iff all-blocks-def bp*(*2*) *calculation*(*1*) *disjoint-free-non-free-def disjoint-iff-not-equal disjoint-memory-set-def invs*(*2*) *invs*(*5*))
moreover have *disjoint-memory b' bs*
by (*metis Un-iff* ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def calculation*(*1*) *disjoint-free-non-free-def disjoint-memory-set-def in-empty-interE invs*(*2*) *invs*(*5*))
moreover have *disjoint-memory b' b*
using ‹*b* ∈ *alloced-bhdr-s s*› *all-blocks-def calculation*(*1*) *calculation*(*2*) *disjoint-memory-set-def invs*(*2*) **by** *auto*
moreover have *wf-block b'*
using *all-blocks-def calculation*(*1*) *invs*(*1*) *wf-def* **by** *auto*

ultimately show *disjoint-memory* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) *b′*
using ‹*wf-block b*› ‹*wf-block bs*› ‹*wf-block bp*› *wf-def disjoint-memory-def*
using ‹- = *s-addr bs*› ‹- = *s-addr b*›
by (*smt add.assoc add.commute bhdr-t.sel*(*1*) *bhdr-t.sel*(*2*) *join-block-def join-block-disjoint*)
qed
done
subgoal
apply (*rule disjoint-remove-block*)
apply *fact*
unfolding *remove-elem-from-matrix-def Let-def*
apply *clarsimp*
unfolding *set-bhdr-matrix-def free-blocks-def*
apply (*auto split*:*if-splits*)
by *blast*+
done
qed
done

lemma *inv-free-wf* :⦃λσ. *inv* σ∧ *block-alloced addr* σ⦄ (*free addr*) ⦃λn σ. *wf* σ ⦄
unfolding *free-def join-prev-def join-suc-def join-block-def Let-def*
apply (*wp* | *split prod.splits*, *intro allI impI*, *drule prod-injects*(*2*), *erule conjE*, *clarsimp*)+
apply (*split if-splits*)
apply (*intro conjI impI*)
defer
apply *blast*
apply (*split if-splits*)
apply (*intro conjI impI*)
apply (*split if-splits*)
apply (*intro conjI impI*)
apply *auto*
subgoal for *s*
unfolding *inv-def wf-def add-block-def all-blocks-def*
apply (*cases mapping-insert conf* (*b-size* (*get-alloced-block addr s*)))
apply *clarsimp*
apply (*subst* (*asm*) *insert-is-union-conf*, *assumption*)
apply (*subst free-blk-mat-s-eq*[*symmetric*])
apply *rule*
using *all-blocks-def get-alloced-is-alloced wf-def* **by** *fastforce*
subgoal for *s bp*
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s′* = (*s*(|*alloced-bhdr-s* := *Set.remove* (*get-alloced-block addr s*) (*alloced-bhdr-s s*)|))
let *?s″* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr ?b*)) (*remove-elem-from-matrix*

```
bp ?i ?j ?s')
    assume prev-free-hdr-s conf ?b s = Some bp
    then obtain b where b': the (prev-free-hdr-s conf b s) = bp
                    and b: get-alloced-block addr s = b
      by force
    assume inv s
    hence invs: wf-adjacency-list s wf s disjoint-memory-set s all-block-mem-size s
      by (auto simp: inv-def)
    with b b' have bp ∈ free-blocks conf s
                   e-addr bp + 1 +overhead conf = s-addr b
      using prev-freeD ‹- = Some bp› by blast+
    hence bp ∈ bhdr-matrix-f s ?i ?j
      by (simp add: invs(1) free-blocks-in-matrix)
    assume block-alloced addr s
    with invs have b ∈ alloced-bhdr-s s
      using get-alloced-is-alloced b by auto
    with ‹bp ∈ free-blocks conf s› ‹wf s› have wfbs:wf-block b wf-block bp
      using all-blocks-def wf-def by blast+
    assume suc-hdr-free-s conf ?b s = None
    have wf-block (join-block bp b)
      apply (rule wf-join-block)
      apply fact+
      apply (rule all-blocks-size-gt-two-blocks)
      apply (rule ‹all-block-mem-size s›[unfolded all-block-mem-size-def])
      using all-blocks-def ‹bp ∈ free-blocks conf s› apply simp
      using all-blocks-def ‹b ∈ -› apply simp
      using ‹wf-block b› ‹wf-block bp› ‹- = s-addr b› wf-block-def apply force
      apply (rule all-block-is-finite) by fact
    show wf ?s''
      apply (rule wf-preserve-3)
      using ‹wf s›
      unfolding wf-def all-blocks-def apply force
      unfolding b join-block-def[symmetric] by fact
  qed
  subgoal for s bs
  proof −
    let ?b = get-alloced-block addr s
    let ?s' = s(|alloced-bhdr-s := Set.remove ?b (alloced-bhdr-s s)|)
    let ?i = fst (mapping-insert conf (b-size bs))
    let ?j = snd (mapping-insert conf (b-size bs))
    let ?s'' = add-block (Bhdr (s-addr ?b) (e-addr bs)) (remove-elem-from-matrix
bs ?i ?j ?s')
    assume suc-hdr-free-s conf ?b s = Some bs
    then obtain b where b':the (suc-hdr-free-s conf b s) = bs and
                        b: get-alloced-block addr s = b
      by fastforce
    assume inv s
    hence invs: wf-adjacency-list s wf s disjoint-memory-set s all-block-mem-size s
      by (simp add: inv-def)+
```

```
    hence invs': wf-adjacency-list ?s'
      unfolding wf-adjacency-list-def by simp
    from b' b have bs ∈ free-blocks conf s
                   e-addr b + 1 + overhead conf = s-addr bs
      using suc-freeD invs ‹- = Some bs› by blast+
    hence bs ∈ bhdr-matrix-f s ?i ?j
      using free-blocks-in-matrix ‹wf-adjacency-list s›
      by auto
    hence bs ∈ bhdr-matrix-f ?s' ?i ?j
      by simp
    assume block-alloced addr s
    hence b ∈ alloced-bhdr-s s
      using get-alloced-is-alloced[OF ‹wf s› ‹disjoint-memory-set s› - b[symmetric]]
      by simp
    with ‹bs ∈ free-blocks conf s› have wf-block b wf-block bs
      using ‹wf s› wf-def all-blocks-def
      by blast+
  assume prev-free-hdr-s conf (Bhdr (s-addr ?b) (e-addr bs)) (remove-elem-from-matrix
bs ?i ?j ?s') = None
    hence prev-free-hdr-s conf b s = None
      unfolding b
      apply −
      apply (drule prev-free-none-equiv2, simp)
      using ‹- = s-addr bs› ‹wf-block bs› ‹wf-block b› wf-block-def
      by (auto dest: prev-free-none-equiv3)
    have wf-block (join-block b bs)
      apply (rule wf-join-block)
      apply fact+
      apply (rule all-blocks-size-gt-two-blocks)
      apply (rule ‹all-block-mem-size s›[unfolded all-block-mem-size-def])
      using all-blocks-def ‹b ∈ -› apply simp
      using all-blocks-def ‹bs ∈ free-blocks conf s› apply simp
      using ‹wf-block b› ‹wf-block bs› ‹- = s-addr bs› wf-block-def apply force
      apply (rule all-block-is-finite) by fact
    show wf ?s''
      unfolding b
      apply (rule wf-preserve-3)
      using ‹wf s› unfolding wf-def all-blocks-def apply force
      unfolding join-block-def[symmetric] by fact
  qed
  subgoal for s bs bp
  proof −
    let ?b = get-alloced-block addr s
    let ?i = fst (mapping-insert conf (b-size bp))
    let ?j = snd (mapping-insert conf (b-size bp))
    let ?s = s(|alloced-bhdr-s := Set.remove ?b (alloced-bhdr-s s)|)
    let ?i' = fst (mapping-insert conf (b-size bs))
    let ?j' = snd (mapping-insert conf (b-size bs))
    let ?s' = add-block (Bhdr (s-addr bp) (e-addr bs)) (remove-elem-from-matrix
```

bp ?i ?j (*remove-elem-from-matrix bs ?i' ?j' ?s*))
assume *suc-hdr-free-s conf ?b s = Some bs*
then obtain *b* **where** *bs*: *the* (*suc-hdr-free-s conf b s*) = *bs*
and *b* : *get-alloced-block addr s* = *b*
by *force*

assume *inv s*
hence *invs*: *wf s disjoint-memory-set s no-split-memory s wf-adjacency-list s all-block-mem-size s*
by (*auto simp*: *inv-def*)
with ‹- = *Some bs*› **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
unfolding *b*
using *suc-freeD* **by** *blast+*
hence *bs* ∈ *bhdr-matrix-f s ?i' ?j'*
using *free-blocks-in-matrix invs*(*4*) *prod.collapse* **by** *blast*
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*wf s*› ‹*bs* ∈*free-blocks conf s*› **have** *wf-block b wf-block bs*
unfolding *wf-def all-blocks-def* **by** *simp+*

assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix bs ?i' ?j' ?s*) = *Some bp*
hence *prev-free-hdr-s conf b s* = *Some bp*
unfolding *b*
apply −
apply (*drule prev-free-some-equiv2*, *simp*)
apply (*drule prev-free-some-equiv3*)
using *wf-def all-blocks-def* ‹*wf s*› **apply** *force*
using *disjoint-memory-set-def all-blocks-def* ‹*disjoint-memory-set s*› **apply** *force*
using ‹- = *s-addr bs*› ‹*wf-block b*› ‹*wf-block bs*› *wf-block-def* **by** *simp+*
hence *bp*: *the* (*prev-free-hdr-s conf b s*) = *bp*
bp ∈ *free-blocks conf s*
e-addr bp + *1* + *overhead conf* = *s-addr b*
using *prev-freeD invs* **by** *force+*
hence *bp* ∈ *bhdr-matrix-f s ?i ?j wf-block bp*
using *wf-def all-blocks-def invs free-blocks-in-matrix* **by** *force+*

have *wf-block* (*join-block bp* (*join-block b bs*))
apply (*rule wf-join-block-2*)
apply *fact+*
apply (*rule all-blocks-size-gt-three-blocks*)
using ‹*all-block-mem-size s*› *all-block-mem-size-def* **apply** *simp*
using *all-blocks-def* ‹*b* ∈ -› ‹*bp* ∈ *free-blocks* - -› ‹*bs* ∈ *free-blocks* - -›
apply *blast+*
using ‹*wf-block b*› ‹*wf-block bp*› ‹*wf-block bs*› *wf-block-def*
using ‹- = *s-addr b*› ‹- = *s-addr bs*› **apply** *force+*

```
    apply (rule all-block-is-finite) by fact
  hence wf-block (Bhdr (s-addr bp) (e-addr bs))
    unfolding join-block-def by simp
  show wf ?s'
    apply (rule wf-preserve-3)
    apply (rule wf-remove-preserve)
    using ‹wf s› wf-def all-blocks-def apply force
    by fact
 qed
 done

lemma inv-free-wf-adjacency-list :
 {|λσ. inv σ∧ block-alloced addr σ|} (free addr) {|λn σ. wf-adjacency-list σ |}
 unfolding free-def join-prev-def join-suc-def join-block-def Let-def
 apply (wp | split prod.splits, intro allI impI, drule prod-injects(2), erule conjE,
clarsimp)+
 apply auto
 by (force simp: inv-def wf-adjacency-list-def set-bhdr-matrix-def
                 tlsf-matrix-def remove-elem-from-matrix-def Let-def
       intro!: add-block-wf-adjacency)+

context
begin
lemma size-join-block:
 e-addr b1 + 1 + overhead conf = s-addr b2 ⟹
  wf-block b1 ⟹ wf-block b2 ⟹
  block-t-size (join-block b1 b2) = block-t-size b1 + block-t-size b2
 unfolding join-block-def
 apply (cases b1, cases b2)
 by (auto simp: wf-block-def)

private lemma h1:
 a ≠ bb ⟹ b ≠ bb ⟹ b ∉ s ⟹ a ∉ t ⟹
  insert (bb) (s − {a} ∪ (t − {b})) = s ∪ t ∪ {bb} − {a} − {b}
 by blast
private lemma h2:
 a ≠ bb ⟹ b ≠ bb ⟹ c ≠ bb ⟹ b ∉ s ⟹ a ∉ t ⟹ c ∉ t ⟹
  insert (bb) (s − {a} − {c} ∪ (t − {b})) = s ∪ t ∪ {bb} − {a} − {c} − {b}
 by blast

lemma inv-free-all-block-mem-size :{|λσ. inv σ∧ block-alloced addr σ|} (free addr)
{|λn σ. all-block-mem-size σ |}
 unfolding free-def join-prev-def join-suc-def join-block-def Let-def
 apply (wp | split prod.splits, intro allI impI, drule prod-injects(2), erule conjE,
clarsimp)+
 apply (split if-splits)
 apply (intro conjI impI)
  defer
  apply blast
```

```
  apply (split if-splits)
  apply (intro conjI impI)
    apply (split if-splits)
    apply (intro conjI impI)
  apply auto
  subgoal for s
    unfolding inv-def all-block-mem-size-def all-blocks-def add-block-def
    apply (cases mapping-insert conf (b-size (get-alloced-block addr s)))
    apply clarsimp
    apply (subst insert-is-union-conf)
    apply assumption
    apply (subst free-blk-mat-s-eq[symmetric])
    apply rule
    by (metis get-alloced-is-alloced insert-Diff insert-is-Un remove-def sup-assoc)
  subgoal for s bp
  proof −
    let ?b = get-alloced-block addr s
    let ?i = fst (mapping-insert conf (b-size bp))
    let ?j = snd (mapping-insert conf (b-size bp))
   let ?s' = (s(|alloced-bhdr-s := Set.remove (get-alloced-block addr s) (alloced-bhdr-s
s)|))
    let ?s'' = add-block (Bhdr (s-addr bp) (e-addr ?b)) (remove-elem-from-matrix
bp ?i ?j ?s')
    assume prev-free-hdr-s conf ?b s = Some bp
    then obtain b where b': the (prev-free-hdr-s conf b s) = bp
                     and b: get-alloced-block addr s = b
      by force
    assume inv s
   hence invs: wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free
s
                all-block-mem-size s
      by (auto simp: inv-def)
    with b b' have bp ∈ free-blocks conf s
                    e-addr bp + 1 +overhead conf = s-addr b
      using prev-freeD ‹- = Some bp› by blast+
    hence bp ∈ bhdr-matrix-f s ?i ?j
      by (simp add: invs(1) free-blocks-in-matrix)
    assume block-alloced addr s
    with invs have b ∈ alloced-bhdr-s s
      using get-alloced-is-alloced b by auto
    with ‹bp ∈ free-blocks conf s› ‹wf s› have wfbs:wf-block b wf-block bp
      using all-blocks-def wf-def by blast+
    have block-t-size (join-block bp b) = block-t-size bp + block-t-size b
      apply (rule size-join-block)
      by fact+
    have finite (all-blocks conf s)
      by (auto simp: all-block-is-finite ‹all-block-mem-size s›)
    assume suc-hdr-free-s conf ?b s = None
```

show *all-block-mem-size ?s''*
unfolding *b remove-def*
unfolding *add-block-def all-block-mem-size-def all-blocks-def remove-elem-from-matrix-def*
apply (*cases mapping-insert conf* (*b-size* (*Bhdr* (*s-addr bp*) (*e-addr b*))))
apply *clarsimp*
apply (*subst insert-is-union-conf*)
apply *simp*
apply (*thin-tac* -)
apply (*subst remove-is-minus-conf*)
apply *fact+*
apply *simp*
apply *rule*
apply *auto*
proof −
have *sum-block* (*insert* (*Bhdr* (*s-addr bp*) (*e-addr b*)) (*free-blocks conf s* − {*bp*} ∪ (*alloced-bhdr-s s* − {*b*}))) =
sum-block (*all-blocks conf s* ∪ {(*Bhdr* (*s-addr bp*) (*e-addr b*))} − {*bp*} − {*b*})
apply (*rule arg-cong*[**where** *f*= *sum-block*])
unfolding *all-blocks-def*
apply (*rule h1*)
using ‹*block-t-size* (*join-block bp b*) = *block-t-size bp* + *block-t-size b*› *join-block-def oh-gt-0* **apply** *force+*
using ‹*b* ∈ *alloced-bhdr-s s*› *disjoint-free-non-free-def invs*(*4*) **apply** *blast*
using ‹*bp* ∈ *free-blocks conf s*› *disjoint-free-non-free-def invs*(*4*) **by** *blast*
also have ... = *sum-block* (*all-blocks conf s*) + *block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr b*)) − *block-t-size bp* − *block-t-size b*
apply (*rule add-implies-diff*)
apply (*subst add.commute*)
unfolding *sum-block-def Un-is-insert*
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *b 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*b* ∈ *alloced-bhdr-s s*› ‹*bp* ∈ *free-blocks conf s*› *all-blocks-def disjoint-free-non-free-def invs*(*4*) **apply** *auto*[*1*]
apply (*rule add-implies-diff*)
apply (*subst* (*2*) *add.commute*)
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *bp 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*b* ∈ *alloced-bhdr-s s*› ‹*bp* ∈ *free-blocks conf s*› *all-blocks-def disjoint-free-non-free-def invs*(*4*) **apply** *auto*[*1*]
apply (*subst comp-fun-commute.fold-insert*)
apply (*rule sum-block-f-commute*)
apply *fact*
apply (*metis Un-iff* ‹*block-t-size* (*join-block bp b*) = *block-t-size bp* + *block-t-size b*› ‹*bp* ∈ *free-blocks conf s*› ‹*inv s*› *add-gr-0*
all-blocks-def bhdr-t.sel(*1*) *diff-block-diff-s-addr join-block-def less-add-same-cancel1 nat-less-le oh-gt-0*)

```
      by simp
    also have ... = sum-block (all-blocks conf s)
      using ‹block-t-size (join-block bp b) = block-t-size bp + block-t-size b›
      unfolding join-block-def by simp
   finally show sum-block (insert (Bhdr (s-addr bp) (e-addr b)) (free-blocks conf
s − {bp} ∪ (alloced-bhdr-s s − {b}))) = mem-size conf
      using ‹all-block-mem-size s› all-block-mem-size-def by simp
  qed
 qed
 subgoal for s bs
 proof −
  let ?b = get-alloced-block addr s
  let ?s' = s(| alloced-bhdr-s := Set.remove ?b (alloced-bhdr-s s) |)
  let ?i = fst (mapping-insert conf (b-size bs))
  let ?j = snd (mapping-insert conf (b-size bs))
  let ?s'' = add-block (Bhdr (s-addr ?b) (e-addr bs)) (remove-elem-from-matrix
bs ?i ?j ?s')
  assume suc-hdr-free-s conf ?b s = Some bs
  then obtain b where b':the (suc-hdr-free-s conf b s) = bs and
                      b: get-alloced-block addr s = b
    by fastforce
  assume inv s
  hence invs: wf-adjacency-list s wf s disjoint-memory-set s disjoint-free-non-free
s
              all-block-mem-size s
    by (simp add: inv-def)+
  hence invs': wf-adjacency-list ?s'
    unfolding wf-adjacency-list-def by simp
  from b' b have bs ∈ free-blocks conf s
                 e-addr b + 1 + overhead conf = s-addr bs
    using suc-freeD invs ‹- = Some bs› by blast+
  hence bs ∈ bhdr-matrix-f s ?i ?j
    using free-blocks-in-matrix ‹wf-adjacency-list s›
    by auto
  hence bs ∈ bhdr-matrix-f ?s' ?i ?j
    by simp
  assume block-alloced addr s
  hence b ∈ alloced-bhdr-s s
    using get-alloced-is-alloced[OF ‹wf s› ‹disjoint-memory-set s› - b[symmetric]]
    by simp
  with ‹bs ∈ free-blocks conf s› have wf-block b wf-block bs
    using ‹wf s› wf-def all-blocks-def
    by blast+
 assume prev-free-hdr-s conf (Bhdr (s-addr ?b) (e-addr bs)) (remove-elem-from-matrix
bs ?i ?j ?s') = None
  hence prev-free-hdr-s conf b s = None
    unfolding b
    apply −
    apply (drule prev-free-none-equiv2, simp)
```

using ‹- = *s-addr bs*› ‹*wf-block bs*› ‹*wf-block b*› *wf-block-def*
by (*auto dest*: *prev-free-none-equiv3*)
have *block-t-size* (*join-block b bs*) = *block-t-size b* + *block-t-size bs*
apply (*rule size-join-block*)
by *fact*+
have *finite* (*all-blocks conf s*)
by (*auto simp*: *all-block-is-finite* ‹*all-block-mem-size s*›)
show *all-block-mem-size ?s''*
unfolding *b remove-def*
unfolding *add-block-def all-block-mem-size-def all-blocks-def remove-elem-from-matrix-def*
apply (*cases mapping-insert conf* (*b-size* (*Bhdr* (*s-addr b*) (*e-addr bs*))))
apply *clarsimp*
apply (*subst insert-is-union-conf*)
apply *simp*
apply (*thin-tac* -)
apply (*subst remove-is-minus-conf*)
apply *fact*+
apply *simp*
apply *rule*
apply *auto*
proof −
have *sum-block* (*insert* (*Bhdr* (*s-addr b*) (*e-addr bs*)) (*free-blocks conf s* − {*bs*} ∪ (*alloced-bhdr-s s* − {*b*}))) =
sum-block (*all-blocks conf s* ∪ {(*Bhdr* (*s-addr b*) (*e-addr bs*))} − {*bs*} − {*b*})
apply (*rule arg-cong*[**where** *f*= *sum-block*])
unfolding *all-blocks-def*
apply (*rule h1*)
using ‹*block-t-size* (*join-block b bs*) = *block-t-size b* + *block-t-size bs*› *join-block-def oh-gt-0* **apply** *force*+
using ‹*b* ∈ *alloced-bhdr-s s*› *disjoint-free-non-free-def invs*(*4*) **apply** *blast*
using ‹*bs* ∈ *free-blocks conf s*› *disjoint-free-non-free-def invs*(*4*) **by** *blast*
also have ... = *sum-block* (*all-blocks conf s*) + *block-t-size* (*Bhdr* (*s-addr b*) (*e-addr bs*)) − *block-t-size bs* − *block-t-size b*
apply (*rule add-implies-diff*)
apply (*subst add.commute*)
unfolding *sum-block-def Un-is-insert*
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *b 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*b* ∈ *alloced-bhdr-s s*› ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def disjoint-free-non-free-def invs*(*4*) **apply** *auto*[*1*]
apply (*rule add-implies-diff*)
apply (*subst* (*2*) *add.commute*)
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *bs 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*b* ∈ *alloced-bhdr-s s*› ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def disjoint-free-non-free-def invs*(*4*) **apply** *auto*[*1*]

apply (*subst comp-fun-commute.fold-insert*)
apply (*rule sum-block-f-commute*)
apply *fact*
apply (*metis Un-iff* ‹*block-t-size* (*join-block b bs*) = *block-t-size b* + *block-t-size bs*› ‹*bs* ∈ *free-blocks conf s*› ‹*inv s*› *add-diff-cancel-right'* *all-blocks-def bhdr-t.sel*(2) *diff-block-diff-e-addr diff-is-0-eq join-block-def leD le-add2 oh-gt-0*)
by *simp*
also have ... = *sum-block* (*all-blocks conf s*)
using ‹*block-t-size* - = -›
unfolding *join-block-def* **by** *simp*
finally show *sum-block* (*insert* (*Bhdr* (*s-addr b*) (*e-addr bs*)) (*free-blocks conf s* − {*bs*} ∪ (*alloced-bhdr-s s* − {*b*}))) = *mem-size conf*
using ‹*all-block-mem-size s*› *all-block-mem-size-def* **by** *simp*
qed
qed
subgoal for *s bs bp*
proof −
let *?b* = *get-alloced-block addr s*
let *?i* = *fst* (*mapping-insert conf* (*b-size bp*))
let *?j* = *snd* (*mapping-insert conf* (*b-size bp*))
let *?s* = *s*(|*alloced-bhdr-s* := *Set.remove ?b* (*alloced-bhdr-s s*)|)
let *?i'* = *fst* (*mapping-insert conf* (*b-size bs*))
let *?j'* = *snd* (*mapping-insert conf* (*b-size bs*))
let *?s'* = *add-block* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*remove-elem-from-matrix bp ?i ?j* (*remove-elem-from-matrix bs ?i' ?j' ?s*))
assume *suc-hdr-free-s conf ?b s* = *Some bs*
then obtain *b* **where** *bs*: *the* (*suc-hdr-free-s conf b s*) = *bs*
and *b* : *get-alloced-block addr s* = *b*
by *force*

assume *inv s*
hence *invs*: *wf s disjoint-memory-set s no-split-memory s wf-adjacency-list s disjoint-free-non-free s*
all-block-mem-size s
by (*auto simp*: *inv-def*)
with ‹- = *Some bs*› **have** *bs* ∈ *free-blocks conf s*
e-addr b + *1* + *overhead conf* = *s-addr bs*
unfolding *b*
using *suc-freeD* **by** *blast*+
hence *bs* ∈ *bhdr-matrix-f s ?i' ?j'*
using *free-blocks-in-matrix invs*(4) *prod.collapse* **by** *blast*
assume *block-alloced addr s*
with *invs* **have** *b* ∈ *alloced-bhdr-s s*
using *get-alloced-is-alloced b* **by** *auto*
with ‹*wf s*› ‹*bs* ∈*free-blocks conf s*› **have** *wf-block b wf-block bs*
unfolding *wf-def all-blocks-def* **by** *simp*+

assume *prev-free-hdr-s conf* (*Bhdr* (*s-addr ?b*) (*e-addr bs*)) (*remove-elem-from-matrix*

bs ?i' ?j' ?s) = *Some bp*
hence *prev-free-hdr-s conf b s* = *Some bp*
unfolding *b*
apply −
apply (*drule prev-free-some-equiv2*, *simp*)
apply (*drule prev-free-some-equiv3*)
using *wf-def all-blocks-def* ‹*wf s*› **apply** *force*
using *disjoint-memory-set-def all-blocks-def* ‹*disjoint-memory-set s*› **apply** *force*
using ‹- = *s-addr bs*› ‹*wf-block b*› ‹*wf-block bs*› *wf-block-def* **by** *simp*+
hence *bp*: *the* (*prev-free-hdr-s conf b s*) = *bp*
bp ∈ *free-blocks conf s*
e-addr bp + *1* + *overhead conf* = *s-addr b*
using *prev-freeD invs* **by** *force*+
hence *bp* ∈ *bhdr-matrix-f s ?i ?j wf-block bp*
using *wf-def all-blocks-def invs free-blocks-in-matrix* **by** *force*+
have *block-t-size* (*join-block b bs*) = *block-t-size b* + *block-t-size bs*
apply (*rule size-join-block*)
by *fact*+
moreover have *block-t-size* (*join-block bp* (*join-block b bs*)) = *block-t-size bp* + *block-t-size* (*join-block b bs*)
apply (*rule size-join-block*)
using ‹- = *s-addr b*› *join-block-def* **apply** *simp*
apply *fact*
apply (*rule wf-join-block*)
apply *fact*+
apply (*rule all-blocks-size-gt-two-blocks*)
apply (*rule* ‹*all-block-mem-size s*›[*unfolded all-block-mem-size-def*])
using *all-blocks-def* ‹*b* ∈ -› **apply** *simp*
using *all-blocks-def* ‹*bs* ∈ *free-blocks conf s*› **apply** *simp*
using ‹*wf-block b*› ‹*wf-block bs*› ‹- = *s-addr bs*› *wf-block-def* **apply** *force*
apply (*rule all-block-is-finite*) **by** *fact*
ultimately have *block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) = *block-t-size bp* + *block-t-size b* + *block-t-size bs*
unfolding *join-block-def* **by** *auto*
have *finite* (*all-blocks conf s*)
by (*auto simp*: *all-block-is-finite* ‹*all-block-mem-size s*›)
show *all-block-mem-size ?s'*
unfolding *b*
unfolding *all-block-mem-size-def all-blocks-def add-block-def*
apply (*cases mapping-insert conf* (*b-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*))))
apply *clarsimp*
apply (*subst free4*)
apply *assumption*
apply *rule*+
apply (*thin-tac* -; *fact*)+
apply (*thin-tac* -)
unfolding *remove-elem-from-matrix-def*
apply *clarsimp*

unfolding *remove-def*
proof −
have *sum-block* (*insert* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*free-blocks conf s* − {*bs*} − {*bp*} ∪ (*alloced-bhdr-s s* − {*b*}))) =
sum-block (*all-blocks conf s* ∪ {*Bhdr* (*s-addr bp*) (*e-addr bs*)} − {*bs*} − {*bp*} − {*b*})
apply (*rule arg-cong*[**where** *f*= *sum-block*])
unfolding *all-blocks-def*
apply (*rule h2*)
using ‹*block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) = *block-t-size bp* + *block-t-size b* + *block-t-size bs*› *oh-gt-0* **apply** *force*+
apply (*metis bp*(*2*) *bp*(*3*) *invs*(*3*) *no-split-memory-def*)
using ‹*bs* ∈ *free-blocks conf s*› *disjoint-free-non-free-def invs*(*5*) **apply** *blast*
using *bp*(*2*) *disjoint-free-non-free-def invs*(*5*) **by** *blast*
also have ... = *sum-block* (*all-blocks conf s*) + *block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) − *block-t-size bs* − *block-t-size bp* − *block-t-size b*
apply (*rule add-implies-diff*)
apply (*subst add.commute*)
unfolding *sum-block-def Un-is-insert*
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *b 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*b* ∈ *alloced-bhdr-s s*› ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def bp*(*2*) *disjoint-free-non-free-def invs*(*5*) **apply** *auto*[*1*]
apply (*rule add-implies-diff*)
apply (*subst* (*2*) *add.commute*)
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *bp 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
using ‹*block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) = *block-t-size bp* + *block-t-size b* + *block-t-size bs*› *all-blocks-def bp*(*2*) *oh-gt-0* **apply** *fastforce*
apply (*rule add-implies-diff*)
apply (*subst* (*2*) *add.commute*)
apply (*subst comp-fun-commute.fold-rec*[*OF sum-block-f-commute*, *of* - *bs 0*, *symmetric*])
using ‹*finite* -› **apply** *blast*
apply (*simp add*: ‹*bs* ∈ *free-blocks conf s*› *all-blocks-def*)
apply (*subst comp-fun-commute.fold-insert*)
apply (*rule sum-block-f-commute*)
apply *fact*
apply (*metis Un-iff* ‹*block-t-size* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) = *block-t-size bp* + *block-t-size b* + *block-t-size bs*› ‹*bs* ∈ *free-blocks conf s*› ‹*inv s*›
add-cancel-left-left add-eq-0-iff-both-eq-0 all-blocks-def bhdr-t.sel(*2*) *diff-block-diff-e-addr neq0-conv oh-gt-0*)
by *simp*
also have ... = *sum-block* (*all-blocks conf s*)
using ‹*block-t-size* (*Bhdr* - -) = -› **by** *auto*

finally show *sum-block* (*insert* (*Bhdr* (*s-addr bp*) (*e-addr bs*)) (*free-blocks*

conf s − {bs} − {bp} ∪ (alloced-bhdr-s s − {b}))) = mem-size conf
using ‹*all-block-mem-size s*› *all-block-mem-size-def* **by** *simp*
qed
qed
done

end

theorem *inv-free*: ⦃λσ. *inv σ* ∧ *block-alloced addr σ*⦄ (*free addr*) ⦃λn σ. *inv σ* ⦄
unfolding *inv-def*
apply (*rule hoare-conjI1*)+
using *inv-free-no-split-memory inv-free-disjoint-free-non-free*
inv-free-disjoint-memory-set inv-free-wf inv-free-wf-adjacency-list
inv-free-all-block-mem-size
using *inv-def* **by** *auto*

thm *valid-def*

— properties

— malloc returns 0 if the size to be allocated is larger than the biggest available block

lemma *r-gt-max-block-fail*: (⦃(λσ. (*inv s*) ∧ (*suitable-blocks conf* (*snd* (*mapping-search conf r*)) σ = {})) ⦄
(*malloc r*) ⦃λ*ret s*. *ret* = *0*⦄)
unfolding *malloc-def remove-block-def Let-def find-suitable-blocks-opt-def*
by *wpsimp*

lemma *next-block-j-lt-sl*:
j < *sl conf* ⟹ *next-block conf* (*i*, *j*) = (*i′*, *j′*) ⟹ *j′* < *sl conf*
unfolding *next-block-def*
by (*auto split*: *if-splits*)

lemma *map-search-j-lt-sl*: *mapping-search conf r* = (*r′*, *i* ,*j*) ⟹ *j* < *sl conf*
unfolding *mapping-search-def*
apply (*auto split*: *prod.splits if-splits simp*: *Let-def*)
defer
apply (*rule next-block-j-lt-sl*)
defer
apply *assumption*
defer
apply (*rule next-block-j-lt-sl*)
defer
apply *assumption*
using *mapping-insert-r-in-l2-set*[*OF mbiggerl*] **by** *metis*+

lemma *r-gt-min-block-alloc*:
⦃(λσ′. σ=σ′ ∧ *inv* σ ∧ r′=*fst* (*mapping-search conf r*) ∧ *suitable-blocks conf* (*snd* (*mapping-search conf r*)) σ ≠{})⦄
malloc r
⦃(λ*addr* σ′. *addr* > 0 ∧
(∃ b ∈ *free-blocks conf* σ.
s-addr b = *addr* ∧ *b-size b* ≥ r′ ∧
((*b-size b* − r′) ≥ *min-block conf* ⟶
(∃ b′ b″. (*alloced-bhdr-s* σ′ = *alloced-bhdr-s* σ ∪ {b′}) ∧
(*free-blocks conf* σ) − {b} ∪ {b″} = *free-blocks conf* σ′ ∧
(b′,b″) = *split-block* r′ b)) ∧
((*b-size b* − r′) < *min-block conf* ⟶
(*alloced-bhdr-s* σ′ = *alloced-bhdr-s* σ ∪ {b}) ∧
(*free-blocks conf* σ) = *free-blocks conf* σ′ ∪ {b}))))⦄
unfolding *malloc-def remove-block-def Let-def*
apply *wpsimp*
apply (*intro conjI impI*)
subgoal — No suitable blocks – Trivially False
unfolding *find-suitable-blocks-opt-def Let-def*
by *force*
apply (*intro ballI conjI*)
subgoal for r′ *i j p b* — Split block
apply (*frule free-matrix-in-free-block*)
apply (*metis option.sel prod.collapse suitable-blocks-j-lt-sl*)
apply (*intro impI allI conjI*)
subgoal for *b1 b2*
using *inv-def wf-def wf-block-def split-block-def oh-gt-0*
by (*metis Un-iff all-blocks-def bhdr-t.sel*(*1*) *fst-conv neq0-conv not-le*)
apply (*rule bexI*[*of - b*])
defer
apply *assumption*
subgoal for *b1 b2*
apply (*intro conjI impI*)
using *split-block-def* **apply** *auto*[*1*]
using *min-block-gt-overhead* **apply** *linarith*
apply *simp-all*
apply *rule*
subgoal
unfolding *add-block-def Let-def remove-elem-from-matrix-def*
by (*auto split*: *prod.splits*)
subgoal
unfolding *add-block-def Let-def remove-elem-from-matrix-def*
apply (*split prod.splits*)
apply *clarsimp*
apply (*subst free-blocks-insert-is-union*)
apply (*metis mapping-insert-r-in-l2-set mbiggerl*)
apply (*subst free-blocks-remove-is-minus*)
apply (*force simp*: *inv-def*)

apply *assumption*
apply (*metis prod.collapse suitable-blocks-j-lt-sl*)
apply (*rule refl*)
by *blast*
done
done
subgoal for r' i j p b — No Split block
apply (*frule free-matrix-in-free-block*)
apply (*metis option.sel prod.collapse suitable-blocks-j-lt-sl*)
apply (*intro impI conjI*)
using *inv-def wf-def wf-block-def all-blocks-def oh-gt-0*
apply (*metis* (*full-types*) *Un-iff neq0-conv not-le*)
apply (*rule bexI*[*of* - *b*])
defer
apply *assumption*
apply (*intro conjI impI*)
apply (*rule refl*)
subgoal
proof −
assume $\exists\, y.$ *find-suitable-blocks-opt* (i, j) σ = *Some y*
then obtain *ps* **where** *find-suitable-blocks-opt* (i, j) σ = *Some ps*
by *blast*
assume *inv* σ
hence *wf-adjacency-list* σ
by (*force simp*: *inv-def*)
assume *mapping-search conf r* = (r', i, j)
hence *l1*:$r' \in$ *l2-set conf i j*
unfolding *mapping-search-def Let-def*
apply (*auto split*: *prod.splits if-splits*)
using *mbiggerl range-l2-disj fst-range-in-set l2-set-def* **by** *blast*+
have $j <$ *sl conf*
apply (*rule map-search-j-lt-sl*)
by *fact*
assume $p \in$ *the* (*find-suitable-blocks-opt* (i, j) σ)
hence *l2*:$(i,j) \leq_b p$
using *suitable-blocks-ij-increase*
by (*metis* ‹*find-suitable-blocks-opt* (i, j) σ = *Some ps*› *option.sel prod.collapse*)
have *snd p* $<$ *sl conf*
apply (*rule suitable-blocks-j-lt-sl*[*of* - - - - *fst p*])
apply *fact*
using ‹*find-suitable-blocks-opt* (i, j) σ = *Some ps*› ‹$p \in$ *the* (*find-suitable-blocks-opt* (i, j) σ)›
by *force*
assume $b \in$ *bhdr-matrix-f* σ (*fst p*) (*snd p*)
have *l3*:*b-size b* $\in$ *l2-set conf* (*fst p*) (*snd p*)
apply (*rule block-mat-size*)
by *fact*+
from *l1 l2 l3* **show** $r' \leq$ *b-size b*
unfolding *block-let-def block-lt-def*

```
      apply (auto simp: Let-def)
      subgoal
        using l2-set-mapping-search-geq-r[OF mbiggerl ‹mapping-search conf - =
-›[symmetric]]
        by blast
      subgoal
       apply (drule snd-range-l2-i-j-less-fst-i'-j'[OF mbiggerl ‹j< -› ‹(snd p)<-›])
        unfolding l2-set-def Let-def
        by auto
      subgoal
        apply (drule snd-range-l2-i-j-less-fst-j'[OF mbiggerl, of - - fst p])
        unfolding l2-set-def Let-def
        by auto
      done
  qed
  apply blast
  subgoal
    unfolding remove-elem-from-matrix-def Let-def
    by clarsimp
  unfolding remove-elem-from-matrix-def Let-def
  apply (subst (2) free-blk-mat-s-eq)
  apply clarsimp
  apply (subst free-blocks-remove-is-minus)
  apply (force simp: inv-def)
  apply assumption
  apply (metis prod.collapse suitable-blocks-j-lt-sl)
  by blast
done


lemma (b1, b2) = split-block r b ⟹ (b2, b1) = split-block r b ⟹ False
  unfolding split-block-def
  apply (cases b, cases b1, cases b2)
  using oh-gt-0 by auto

lemma exist-split-prev-equiv:
  wf s ⟹ disjoint-memory-set s ⟹
    (∃ b' bc r. b' ∈ free-blocks conf s ∧ (b', b) = split-block r bc) ⟷ (∃ bp.
prev-free-hdr-s conf b s = Some bp)
  apply rule
  apply auto
  subgoal for b' bc r
        — uncomment the following proof script to get a different problem
        — whether or not should we prove bp is THE b. P b ?
        — this means whether or not should we prove the uniqueness of bp

    unfolding prev-free-hdr-s-def
    apply auto
    apply (rule exI)
```

apply *auto*
apply (*cases b'*)
apply (*auto simp*: *split-block-def*)
by (*metis Suc-pred Un-iff add.commute all-blocks-def bhdr-t.sel(1) diff-0-eq-0 diff-add-inverse neq0-conv not-le oh-gt-0 wf-def wf-block-def*)
subgoal for *bp*
apply (*drule prev-freeD*, *simp-all*)
apply *rule*
apply *auto*
unfolding *split-block-def*
apply (*cases b*, *cases bp*)
apply *auto*
by (*metis Suc-le-eq Un-iff add-diff-inverse-nat all-blocks-def bhdr-t.sel(1) bhdr-t.sel(2) diff-Suc-Suc diff-zero less-imp-le-nat not-le wf-def wf-block-def*)
done

lemma *non-exist-split-prev-equiv*:
assumes *wf s disjoint-memory-set s*
shows (∀ *b' bc r*. *b'* ∈ *free-blocks conf s* ⟶ (*b'*, *b*) ≠ *split-block r bc*) ⟷ *prev-free-hdr-s conf b s* = *None*
using *exist-split-prev-equiv*[*OF assms*]
by (*metis option.distinct(1) option.exhaust*)

lemma *exist-split-suc-equiv*:
wf s ⟹ *disjoint-memory-set s* ⟹ *b* ∈ *alloced-bhdr-s s* ⟹
(∃ *b' bc r*. *b'* ∈ *free-blocks conf s* ∧ (*b*, *b'*) = *split-block r bc*) ⟷ (∃ *bs*. *suc-hdr-free-s conf b s* = *Some bs*)
apply *rule*
apply *auto*
subgoal for *b' bc r*
unfolding *suc-hdr-free-s-def*
apply *auto*
apply (*rule exI*[*of* - *b'*])
apply *auto*
apply (*cases b'*)
apply (*auto simp*: *split-block-def*)
by (*metis Suc-pred Un-iff add-diff-cancel-right' all-blocks-def bhdr-t.sel(1) diff-0-eq-0 diff-is-0-eq diff-zero neq0-conv oh-gt-0 wf-def wf-block-def*)
subgoal for *bp*
apply (*drule suc-freeD*, *simp-all*)
apply *rule*
apply *auto*
unfolding *split-block-def*
apply (*cases b*, *cases bp*)
apply *auto*
by (*metis Suc-le-eq Un-iff add-diff-inverse-nat all-blocks-def bhdr-t.sel(1) bhdr-t.sel(2) diff-Suc-Suc diff-zero less-imp-le-nat not-le wf-def wf-block-def*)
done

lemma *non-exist-split-suc-equiv*:
assumes *wf s disjoint-memory-set s b* ∈ *alloced-bhdr-s s*
shows (∀ *b′ bc r*. *b′* ∈ *free-blocks conf s* ⟶ (*b*, *b′*) ≠ *split-block r bc*) ⟷ *suc-hdr-free-s conf b s* = *None*
using *exist-split-suc-equiv*[*OF assms*]
by *auto*

lemma *exist-split--prev-suc-equiv*:
wf s ⟹ *disjoint-memory-set s* ⟹ *b* ∈ *alloced-bhdr-s s* ⟹
(∃ *b1 b2 bc bc′ r r′*. *b1* ∈ *free-blocks conf s* ∧ *b2* ∈ *free-blocks conf s*
∧ (*b1*, *bc′*) = *split-block r bc* ∧ (*b*, *b2*) = *split-block r′ bc′*)
⟷ (∃ *bs*. *suc-hdr-free-s conf b s* = *Some bs*) ∧ (∃ *bp*. *prev-free-hdr-s conf b s* = *Some bp*)
apply *rule*
apply *auto*
using *exist-split-suc-equiv* **apply** *blast*
apply (*subst exist-split-prev-equiv*[*symmetric*], *simp-all*)
subgoal for *b1 b2 bc bc′ r r′*
apply (*rule exI*[*of - b1*])
apply *auto*
apply (*rule exI*[*of - Bhdr* (*s-addr b1*) (*e-addr b*)])
apply (*rule exI*[*of - b-size b1*])
unfolding *split-block-def*
apply *clarsimp*
by (*metis Suc-pred Un-upper2 add.commute all-blocks-def bhdr-t.sel*(*1*) *contra-subsetD diff-0-eq-0*
diff-add-inverse neq0-conv not-le oh-gt-0 wf-def sup-commute wf-block-def)
subgoal for *bs bp*
apply (*drule suc-freeD*, *simp-all*)
apply (*drule prev-freeD*, *simp-all*)
apply (*rule exI*[*of - bp*], *simp*)
apply (*rule exI*[*of - bs*], *simp*)
apply (*rule exI*[*of - Bhdr* (*s-addr bp*) (*e-addr bs*)])
apply (*rule exI*[*of - Bhdr* (*s-addr b*) (*e-addr bs*)])
unfolding *split-block-def*
apply *clarsimp*
apply *rule*
apply (*rule exI*[*of - b-size bp*])
apply *rule*
apply (*metis Un-iff all-blocks-def b-size.simps bhdr-t.exhaust-sel diff-Suc-Suc diff-zero le-Suc-eq le-add-diff-inverse plus-1-eq-Suc wf-def wf-block-def*)
apply (*metis One-nat-def Un-iff add.left-neutral add-Suc all-blocks-def b-size.simps bhdr-t.exhaust-sel le-Suc-eq le-add-diff-inverse wf-def wf-block-def*)
apply (*rule exI*[*of - b-size b*])
apply *rule*
apply (*metis Un-iff all-blocks-def b-size.simps bhdr-t.exhaust-sel diff-Suc-Suc diff-zero le-Suc-eq le-add-diff-inverse plus-1-eq-Suc wf-def wf-block-def*)
apply (*metis One-nat-def Un-iff add.left-neutral add-Suc all-blocks-def b-size.simps*

bhdr-t.exhaust-sel le-Suc-eq le-add-diff-inverse wf-def wf-block-def)
done
done

lemma *free-addr-alloced*:
⦃$\lambda\sigma'$. σ=σ' ∧ *inv* σ' ∧ *block-alloced addr* σ ⦄ *free addr*
⦃$\lambda x\ \sigma'$.
let b = get-alloced-block addr σ in
alloced-bhdr-s σ = alloced-bhdr-s σ' ∪ {b} ∧
(*prev-free-hdr-s conf b σ = None ∧ suc-hdr-free-s conf b σ = None ⟶ free-blocks conf σ' = free-blocks conf σ ∪ {b}*) ∧
(*prev-free-hdr-s conf b σ ≠ None ∧ suc-hdr-free-s conf b σ = None ⟶*
free-blocks conf σ' ∪ {the (prev-free-hdr-s conf b σ)} = free-blocks conf σ ∪ {join-block (the (prev-free-hdr-s conf b σ)) b}) ∧
(*prev-free-hdr-s conf b σ = None ∧ suc-hdr-free-s conf b σ ≠ None ⟶*
free-blocks conf σ' ∪ {the (suc-hdr-free-s conf b σ)} = free-blocks conf σ ∪ {join-block b (the (suc-hdr-free-s conf b σ))}) ∧
(*prev-free-hdr-s conf b σ ≠ None ∧ suc-hdr-free-s conf b σ ≠ None ⟶*
free-blocks conf σ' ∪ {the (prev-free-hdr-s conf b σ)} ∪ {the (suc-hdr-free-s conf b σ)} =
free-blocks conf σ ∪ {join-block (the (prev-free-hdr-s conf b σ)) (join-block b (the (suc-hdr-free-s conf b σ)))})⦄
unfolding *free-def join-prev-def join-suc-def join-block-def Let-def*
apply (*wp* | *split prod.splits*, *intro allI impI*, *drule prod-injects*(*2*), *erule conjE*, *clarsimp*)+
apply (*split if-splits*)
apply (*rule conjI*)
defer
apply *blast*
apply (*rule impI*)
apply (*split if-splits*)
apply (*rule conjI*)
apply (*rule impI*)
subgoal for *s*
apply (*split if-splits*)
apply (*rule conjI*)
apply (*rule impI*)
subgoal — case 1
apply *clarsimp*
apply (*auto simp*: *inv-def add-block-def*
split: *prod.splits intro*: *get-alloced-is-alloced*)
using *insert-is-union-conf free-blk-mat-s-eq* **by** *force*+
apply (*rule impI*)
subgoal — case 2
apply *clarsimp*
apply (*intro conjI impI*)
subgoal for *bp*
by (*auto simp*: *add-block-def remove-elem-from-matrix-def inv-def*

split: *prod.splits intro*: *get-alloced-is-alloced*)
subgoal for *bp*
unfolding *add-block-def Let-def remove-elem-from-matrix-def inv-def*
apply (*split prod.splits*)
apply *clarsimp*
apply (*subst insert-is-union-conf*, *simp-all*)
apply (*subst remove-is-minus-conf*, *simp-all*)
apply (*drule prev-freeD*, *simp-all*)
apply (*rule free-blocks-in-matrix*, *simp-all*)
by (*auto dest*: *prev-freeD*)
done
done
apply (*rule impI*)
subgoal for *s*
apply (*rule conjI*)
apply (*rule impI*)
subgoal — case 3
apply (*subgoal-tac prev-free-hdr-s conf* (*get-alloced-block addr s*) *s* = *None*)
apply *clarsimp*
apply (*intro conjI impI*)
subgoal for *bp*
by (*auto simp*: *add-block-def remove-elem-from-matrix-def inv-def*
split: *prod.splits intro*: *get-alloced-is-alloced*)
subgoal for *bp*
unfolding *add-block-def Let-def remove-elem-from-matrix-def inv-def*
apply (*split prod.splits*)
apply *clarsimp*
apply (*subst insert-is-union-conf*, *simp-all*)
apply (*subst remove-is-minus-conf*, *simp-all*)
apply (*drule suc-freeD*, *simp-all*)
apply (*rule free-blocks-in-matrix*, *simp-all*)
by (*auto dest*: *suc-freeD*)
subgoal
apply *clarsimp*
apply (*drule prev-free-none-equiv2*, *simp*)
apply (*drule prev-free-none-equiv3*)
apply (*drule suc-freeD*, *simp-all add*:*inv-def*)
using *get-alloced-is-alloced*
by (*metis Un-iff add-Suc-right add-leD1 all-blocks-def not-less-eq-eq wf-def wf-block-def*)
done
apply (*rule impI*)
subgoal — case 4
apply (*subgoal-tac* ∃ *y*. *prev-free-hdr-s conf* (*get-alloced-block addr s*) *s* = *Some y*)
apply *clarsimp*
apply (*intro conjI impI*)
subgoal
by (*auto simp*: *add-block-def remove-elem-from-matrix-def inv-def*

```
            split: prod.splits intro: get-alloced-is-alloced)
      subgoal for bs bp' bp
        apply (subgoal-tac bp' = bp)
        apply hypsubst-thin
        subgoal
          apply (thin-tac - = Some bp)
          unfolding add-block-def Let-def
          apply (split prod.splits)
          apply clarsimp
          apply (subst free4)
          apply (auto simp: prev-freeD wf-def all-blocks-def inv-def
                      dest: prev-freeD suc-freeD intro!: free-blocks-in-matrix)
          using get-alloced-is-alloced
          by (metis UnI2 all-blocks-def wf-def)
        subgoal
          apply (drule prev-free-some-equiv2, simp)
          apply (drule prev-free-some-equiv3)
          apply (auto simp: inv-def wf-def all-blocks-def disjoint-memory-set-def
                      dest!: suc-freeD)
       by (metis Suc-n-not-le-n Un-iff add.commute all-blocks-def disjoint-memory-set-def

            get-alloced-is-alloced le-cases le-trans wf-def trans-le-add2 wf-block-def)
        done
      subgoal
        apply auto
        apply (drule prev-free-some-equiv2, simp)
        apply (drule prev-free-some-equiv3)
        apply (auto simp: inv-def wf-def all-blocks-def disjoint-memory-set-def
                    dest!: suc-freeD)
      by (metis Suc-n-not-le-n Un-iff add.commute all-blocks-def disjoint-memory-set-def

          get-alloced-is-alloced le-cases le-trans wf-def trans-le-add2 wf-block-def)
    done
  done
  done

lemma undefined = undefined
  by simp



end
```