Progress of Concurrent Objects with Partial Methods

Hongjin Liang and Xinyu Feng
University of Science and Technology of China

To appear at POPL 2018

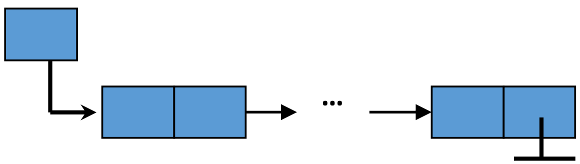
Previous work

- Linearizability ←→ Contextual Refinement **[PLDI'13]**
- Linearizability + Lock-freedom / Wait-freedom ←→ Contextual Refinement **[LICS'14]**
- Linearizability + Deadlock-freedom / Starvation-freedom ←→ Contextual Refinement **[POPL'16]**

Object with partial objects:

- Linearizability + ?? ←→ ??
  - This talk

Concurrent object **O**



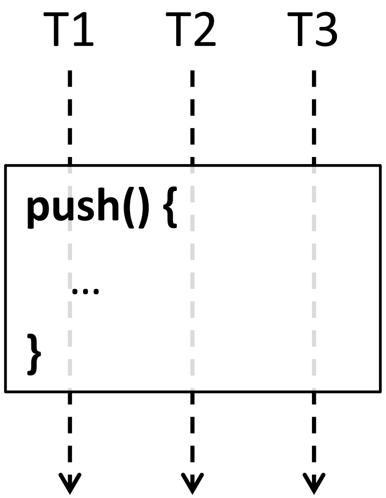
```
void push(int v) {
    ...
}

int pop() {
    ...
}
```

[*java.util.concurrent*]

Client code **C**

...	...
push(7);	push(6);
x = pop();	...
...	



Example: lock-based counter

```
inc() {
   acq();
   cnt := cnt + 1;
   rel();
}


// internal functions
acq() {
  …
}
rel() {
  …
}
```

It's an object with total methods

because inc() always terminates if executed sequentially

Example of an object with partial methods: test-and-set (TAS) lock

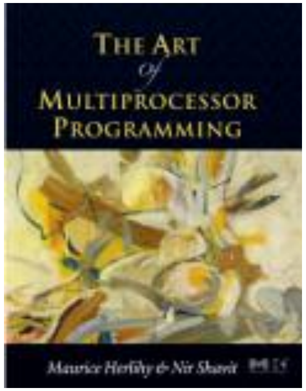
```
acq() {
    local succ;
    succ := false;
    while( ! succ ) {
        succ := cas(L, 0, 1);
    }
}

rel() {
    L := 0;
}
```

acq() is supposed not to terminate if the lock has been acquired.

Our work: specify and verify correctness of objects with partial methods

Standard correctness of **O**



[Herlihy & Shavit]

- Linearizability
  - Correctness about functionality/atomicity
  - Require **O** to have the same effect as an atomic spec **S**

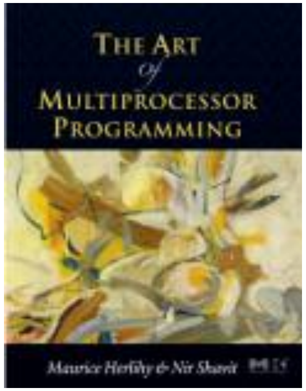
    Atomic spec for counters:

    INC() { cnt := cnt+1; }

    Atomic spec for locks:

    ACQ(){ L := 1; }   REL() { L := 0; }
  - Not talk about termination/liveness properties

Standard correctness of **O**



[Herlihy & Shavit]

- Progress properties
  - Lock-freedom (LF)
  - Wait-freedom (WF)
  - Starvation-freedom (SF)
  - Deadlock-freedom (DF)

Methods must always terminate in sequential executions

All of them are limited to objects with total methods (e.g., the counter satisfies DF).

None applies to objects with partial methods (e.g. locks).

Contextual refinement (CR) as correctness of **O**

Client C

```
…                ||   …
x := 7;          ||   y := pop();
push( x );       ||   print(y);
…                ||   …
```

Concrete object **O**

```
void push(int v) {
    …
}
int pop() {
    …
}
```

Is behavior of **C**[**O**] the same as **C**[$\mathcal{A}$]?

Abstract object $\mathcal{A}$

push

pop

Contextual refinement (CR) as correctness of **O**

- **O** $\sqsubseteq_{\mathbf{ctxt}}$ $\mathcal{A}$ iff $\forall$**C**. ObsBeh(**C**[**O**]) $\sqsubseteq$ ObsBeh(**C**[$\mathcal{A}$])
- linearzability + progress (LF/WF/DF/SF) $\Leftrightarrow$ CR

  [Gotsman & Yang'11, Liang et al'13]
  - ObsBeh: termination-sensitive
  - LF and WF objects: no assumption on scheduling
  - DF and SF objects: assume fair scheduling
  - Atomic spec S as $\mathcal{A}$ for WF/SF objects, non-atomic $\mathcal{A}$ for LF/DF objects

No abstraction $\mathcal{A}$ for objects with partial methods!

Problems for objects with partial methods

- No progress properties
- No abstractions $\mathcal{A}$ for contextual refinements

- Consequences
  - Cannot treat locks as objects.
    Treat acq() and rel() as internal functions instead when verifying the counter.
  - Redo the verification of acq() and rel() in different contexts.

```
inc() {
  acq();
  cnt := cnt + 1;
  rel();
}

// internal functions
acq() {
  ...
}
rel() {
  ...
}
```

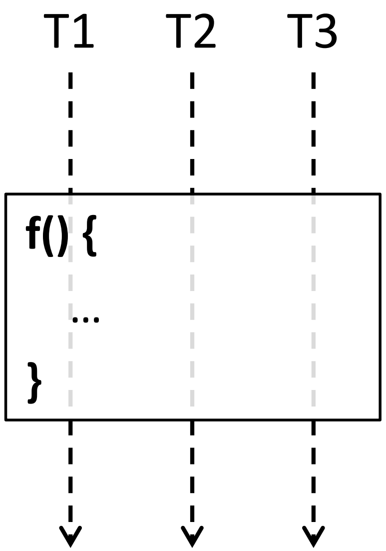
Our work

- Progress properties for objects with partial methods
  - Partial starvation-freedom (PSF)
  - Partial deadlock-freedom (PDF)
  - SF and DF are specializations of PSF and PDF
- 4 general patterns for abstractions to establish CR
  - For PSF/PDF objects under strongly/weakly fair scheduling
- Equivalence result (Abstraction Theorem)
  - Linearizability + PSF/PDF $\Leftrightarrow$ CR with proper abstraction
- Program logic
  - Extending the existing logic LiLi for SF & DF [Liang & Feng'16]

SF and DF as Progress Properties

[Herlihy and Shavit 2011]

- SF: under fair scheduling, every thread can finish its method call
- DF: under fair scheduling, there always exists some thread that can finish its method call



Fair scheduling: every T gets eventually executed

Need new progress properties

- SF: under fair scheduling, every thread can finish its method call
- DF: under fair scheduling, there always exists some thread that can finish its method call

```
l.acq();   ||   l.acq();
…          ||   …
l.rel();
```

May never terminate

SF and DF always expect termination of methods.

Need new progress properties

- SF: under fair scheduling, every thread can finish its method call
- DF: under fair scheduling, there always exists some thread that can finish its method call

l.acq();	‖	l.acq();
…	‖	…
l.rel();	‖	

Lock doesn't satisfy SF or DF.

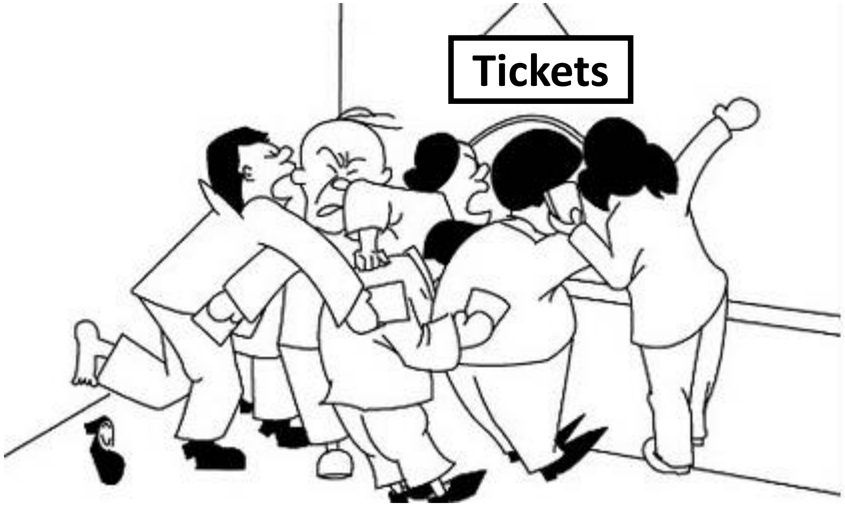
Should blame client instead of obj. for non-termination

What are "good" locks?

Example: TAS lock vs. ticket lock

TAS lock

```
acq() {
    local succ;
    succ := false;
    while( ! succ ) {
        succ := cas(L, 0, 1);
    }
}
rel() {
    L := 0;
}
```



Example: TAS lock vs. ticket lock

TAS lock

```
acq() {
   local succ;
   succ := false;
   while( ! succ ) {
      succ := cas(L, 0, 1);
   }
}
rel() {
   L := 0;
}
```

client:

```
acq();           ||   while(true){
rel();           ||      acq();
print(1);        ||      rel();
                 ||   }
```

It may not print 1.

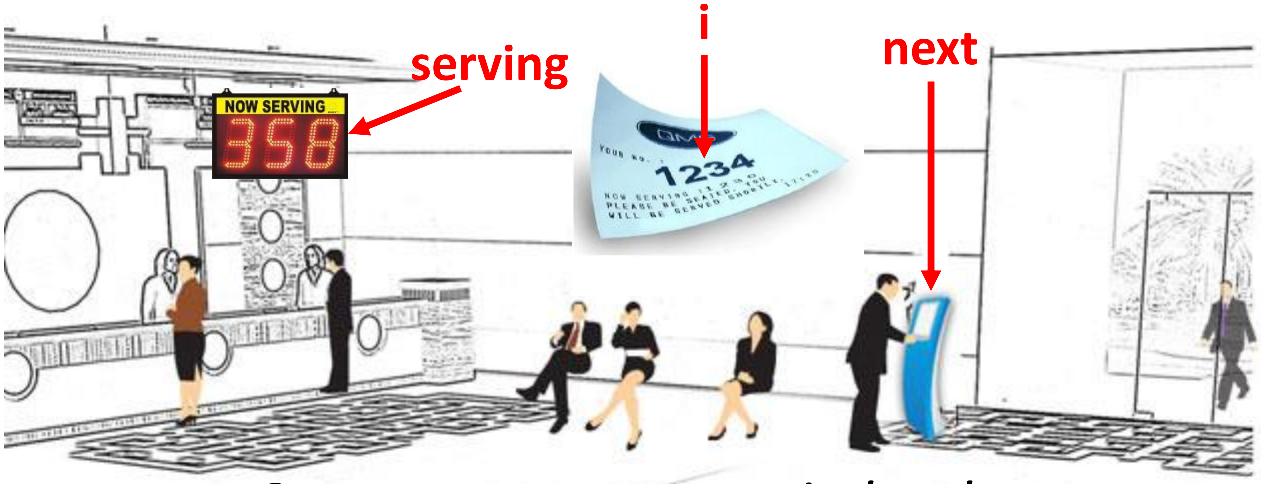
Example: TAS lock vs. ticket lock

```
acq() {
   local i;
   i := getAndInc( next );
   while( i != serving ) {} ;
}
rel() {  serving := serving + 1;  }
```

the next available ticket

currently being served

Ticket lock



Queue management in banks

Example: TAS lock vs. ticket lock

```
acq() {
   local i;
   i := getAndInc( next );
   while( i != serving ) {} ;
}
rel() {  serving := serving + 1;  }
```

Ticket lock

client:

```
acq();          ||   while(true){
rel();          ||       acq();
print(1);       ||       rel();
                ||   }
```

It must print 1
under fair scheduling

Different impl exhibit different progress properties

New progress properties

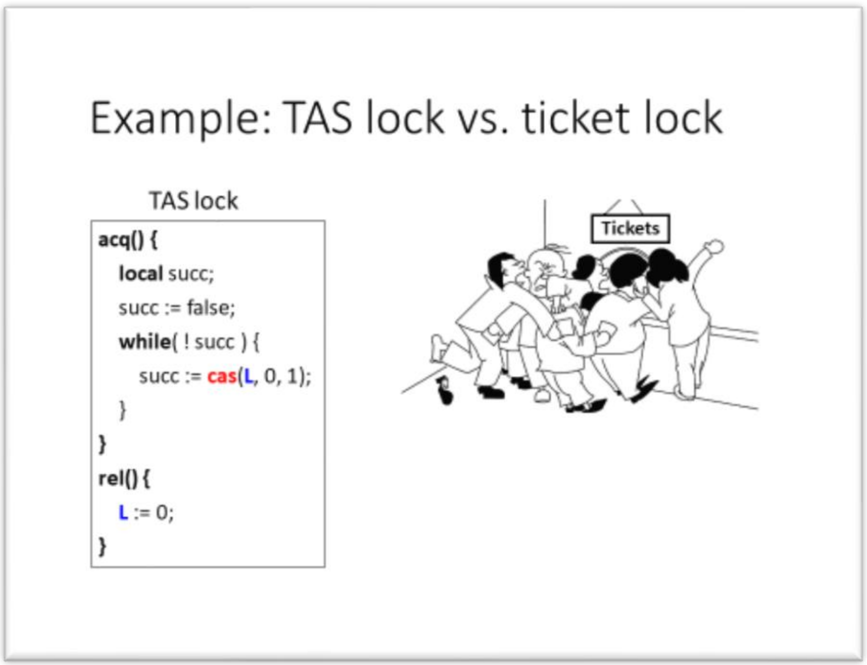
- Partial starvation-freedom (PSF-χ)
  - Under scheduling with χ-fairness, every thread can finish its method call, unless pending method invocations are always blocked
- Partial deadlock-freedom (PDF-χ)
  - Under scheduling with χ-fairness, there always exists some thread that can finish its method call, unless pending method invocations are always blocked
- χ-fairness: strong fairness or weak fairness
  - Need to distinguish them for blocking primitives
  - Will explain later

New progress properties

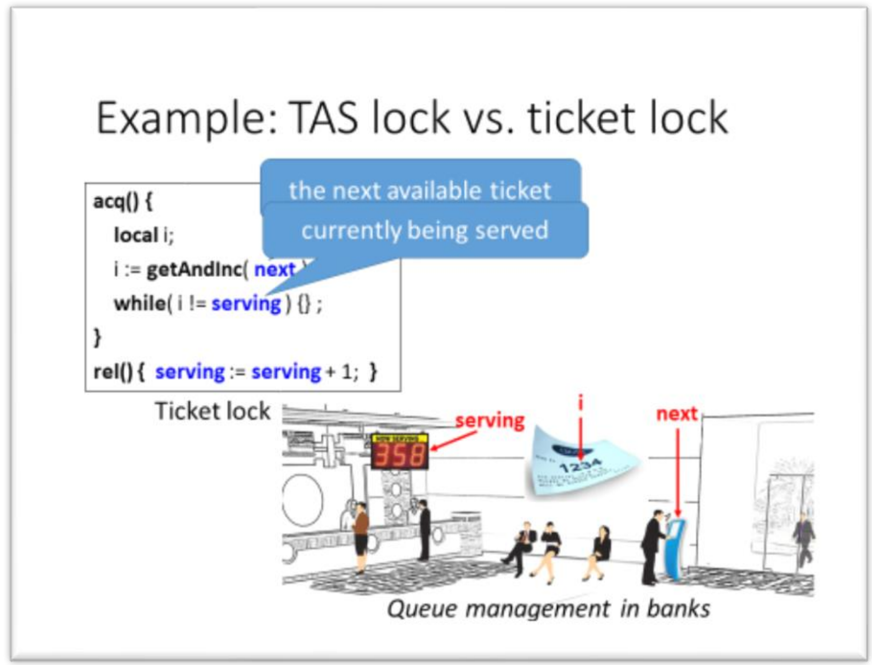
- Partial starvation-freedom (PSF-χ)
  - Under scheduling with χ-fairness, every thread can finish its method call, unless pending method invocations are always blocked
- Partial deadlock-freedom (PDF-χ)
  - Under scheduling with χ-fairness, there always exists some thread that can finish its method call, unless pending method invocations are always blocked
- SF and DF are specializations of PSF and PDF

Example: TAS lock vs. ticket lock

Different impl exhibit different progress properties



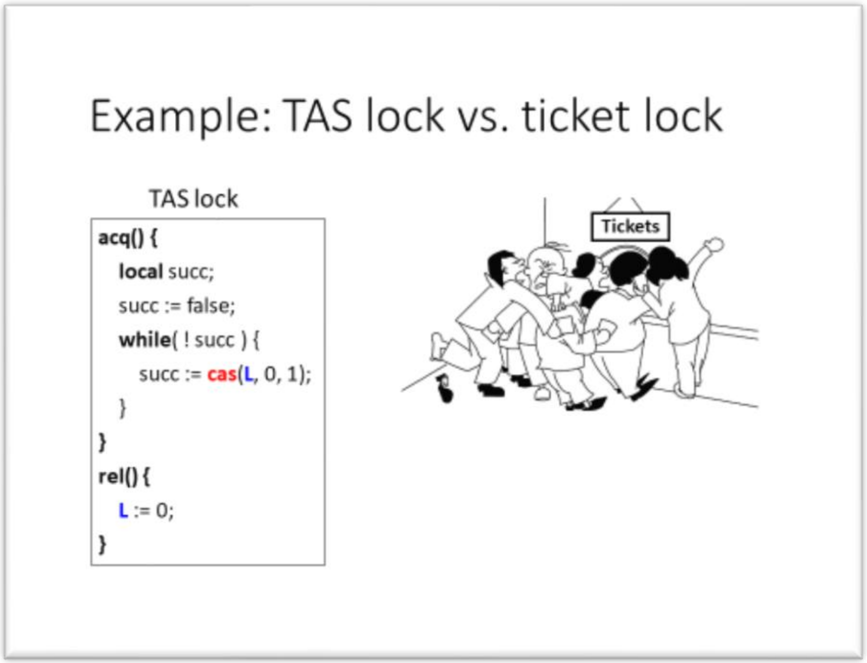
PDF

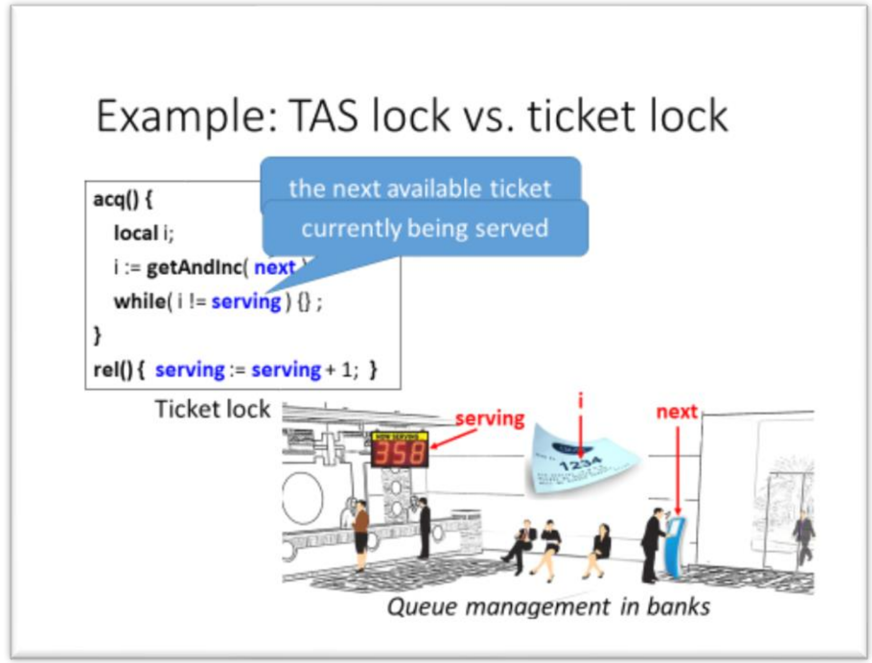


PSF

Example: TAS lock vs. ticket lock

Different impl exhibit different progress properties





What are the abstractions for locks?

Our work

- Progress properties for objects with partial methods
  - Partial starvation-freedom (PSF)
  - Partial deadlock-freedom (PDF)
  - SF and DF are specializations of PSF and PDF
- 4 general patterns for abstractions to establish CR
  - For PSF/PDF objects under strongly/weakly fair scheduling
- Equivalence result (Abstraction Theorem)
  - Linearizability + PSF/PDF $\Leftrightarrow$ CR with proper abstraction
- Program logic
  - Extending the existing logic LiLi for SF & DF [Liang & Feng'16]

Abstractions for partial methods

- Recall that linearizability requires O to have the same effect as an atomic spec S
    - Atomic spec for locks: ACQ(){ L := 1; } REL() { L := 0; }
- Problem: ACQ() does not specify that lock acquire should not return when lock is unavailable
- Solution: atomic partial spec in form of await(B){C}
    - If B doesn't hold, block; otherwise, execute C atomically
    - The code is called "enabled" when B holds
    - ACQ(){ await(L=0){ L := 1 }; } REL() { L := 0; }

Atomic partial specs are insufficient for abstractions

Consider the client behaviors with the three locks:

client:

$[\ ]_{ACQ}$;
 $[\ ]_{REL}$;
 print(1);
	$\|\|$	while(true){
		   $[\ ]_{ACQ}$;
		   $[\ ]_{REL}$;
		 }

- Atomic partial spec:

  ACQ(){ await(L=0){ L := 1 }; }
  REL() { L := 0; }

- TAS locks

- Ticket locks

Atomic partial specs are insufficient for abstractions

Consider the client behaviors with the three locks:

client:

$[\ ]_{ACQ}$;
$[\ ]_{REL}$;
print(1);
	‖	while(true){
		    $[\ ]_{ACQ}$;
		    $[\ ]_{REL}$;
		}

- Atomic partial spec:

```
ACQ(){ await(L=0){ L := 1 }; }
REL() { L := 0; }
```

every thread which is infinitely often enabled will be executed

eventually always enabled will be executed

It must print 1 under strong fairn

It may not print 1 under weak fairness

Atomic partial specs are insufficient for abstractions

Consider the client behavio

client:

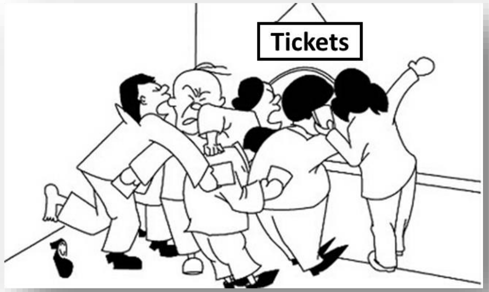
$[\ ]_{ACQ};$
$[\ ]_{REL};$
print(1);

||

while(true){
  $[\ ]_{ACQ};$
  $[\ ]_{REL};$
}

TAS lock

```
acq() {
  local succ;
  succ := false;
  while( ! succ ) {
    succ := cas(L, 0, 1);
  }
}
rel() {
  L := 0;
}
```



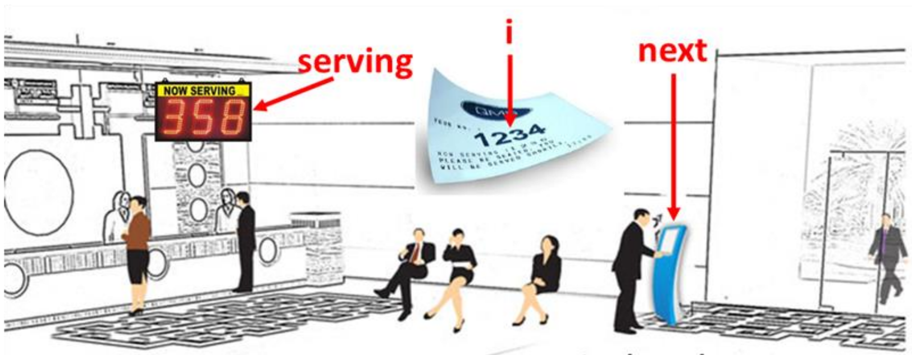
- TAS locks

Behaviors are the same under strong & weak fairness

It may not print 1.

```
acq() {
   local i;
   i := getAndInc( next );
   while( i != serving ) {} ;
}
rel() {  serving := serving + 1;  }
```

Ticket lock



Queue management in banks

client:

$[\ ]_{ACQ}$;
$[\ ]_{REL}$;
print(1);
	while(true){
	$[\ ]_{ACQ}$;
	$[\ ]_{REL}$;
	}

It **must** print 1
under **strong/weak** fairness

- Atomic partial spec:

  ACQ(){ await(L=0){ L := 1 }; }
  REL() { L := 0; }

- TAS locks
- Ticket locks

Example: locks

$[\ ]_{ACQ};$
$[\ ]_{REL};$
print(1);

||

```
while(true){
    [ ]ACQ;
    [ ]REL;
}
```

	Atomic partial spec	Ticket lock	TAS lock
Strong fairness	Must print 1	Must print 1	May not print 1
Weak fairness	May not print 1	Must print 1	May not print 1

Problem #1: await blocks cannot be abstraction for the same impl. under **different fairness**

Example: locks

```
[ ]ACQ;                 ||  while(true){
[ ]REL;                 ||      [ ]ACQ;
print(1);               ||      [ ]REL;
                        ||  }
```

	Atomic partial spec	Ticket lock	TAS lock
Strong fairness	Must print 1	Must print 1	May not print 1
Weak fairness	May not print 1	Must print 1	May not print 1

Problem #1: await blocks cannot be abstraction for the same impl. under **different fairness**

Problem #2: await blocks cannot serve as abstraction for **different implementations**, which exhibit different progress

Example: locks

$[\ ]_{ACQ}$;
$[\ ]_{REL}$;
print(1);

||

```
while(true){
    [ ]ACQ;
    [ ]REL;
}
```

	Atomic partial spec	Ticket lock	TAS lock
Strong fairness	Must print 1	Must print 1	May not print 1
Weak fairness	May not print 1	Must print 1	May not print 1

We need more than one abstraction!

2 progress (PSF vs. PDF) **x** 2 fairness (Strong vs. Weak)

Can we systematically generate all of them?

Our solution

- Code wrappers: syntactic transformations that turn await(B){C} to proper (possibly non-atomic) specs

Obj. Impl. O

Atomic partial spec:
await(B){C}

Contextual
refinement

P-χ-Wrapper(await(B){C})
P $\in$ {PSF, PDF}
χ $\in$ {Strong / Weak Fairness}

Our solution

- Code wrappers: syntactic transformations that turn await(B){C} to proper (possibly non-atomic) specs

P-χ-Wrapper(await(B){C})

P $\in$ {PSF, PDF}

χ $\in$ {Strong / Weak Fairness}

	PSF	PDF
Strong fairness	?	?
Weak fairness	?	?

Our solution

- Code wrappers: syntactic transformations that turn await(B){C} to proper (possibly non-atomic) specs

execute unless eventually always disabled

PSF-sfair-wrapper(await(B){C}) = await(B){C}

must return unless eventually always disabled

ACQ(){ await(L=0){ L := 1 }; } REL() { L := 0; }

could be abstraction for **ticket locks** under strong fairness execute if eventually always enabled

PSF-wfair-wrapper(await(B){C}) = **?**

must return unless eventually always disabled execute if eventually always enabled

guarantee to execute C if B is **infinitely often** true

PSF-wfair-wrapper(await(B){C}) = **?**

must return unless eventually always disabled

	Atomic partial spec	**Ticket lock**	**TAS lock**
Strong fairness	Must print 1	Must print 1	May not print 1
Weak fairness	May not print 1	Must print 1	May not print 1

$[\ ]_{ACQ}$;
$[\ ]_{REL}$;
print(1);

||

while(true){
$[\ ]_{ACQ}$;
$[\ ]_{REL}$;
}

guarantee to execute C if B is infinitely often true

execute if eventually always enabled

PSF-wfair-wrapper(await(B){C}) =

a blocking queue of (t, ‘B’) pairs

listid := listid ++ [(cid, ‘B’)];

await(**B** $\wedge$ cid = **enhd**(listid)){ **C**; listid := listid\cid; }

return the first thread on listid whose enabling condition is true

If B is infinitely often true,

B $\wedge$ cid = **enhd**(listid) will be eventually always true

so C will be eventually executed, ensuring PSF

PDF-wfair-wrapper(await(B){C}) = await(B){C} ?

at least one method call returns
unless all are eventually always disabled

ACQ(){ await(L=0){ L := 1 }; } REL() { L := 0; }

could be abstraction for **TAS locks** under weak fairness

$[\ ]_{ACQ}$;
 $[\ ]_{REL}$;
 print(1);
	∥	while(true){
		 $[\ ]_{ACQ}$;
		 $[\ ]_{REL}$;
		 }

However, in general await(B){C} cannot be PDF abstraction

PDF-wfair-wrapper(await(B){C}) = await(B){C} ?

However, in general await(B){C} cannot be PDF abstraction

P-POP() { await(!emp(S)) { pops S }

PUSH(x) { ret x::S }

$[\ ]_{P\_POP()}$	$\|\|$	while(true){
print(1);		$\quad [\ ]_{PUSH(0)}$
		}

Must print 1

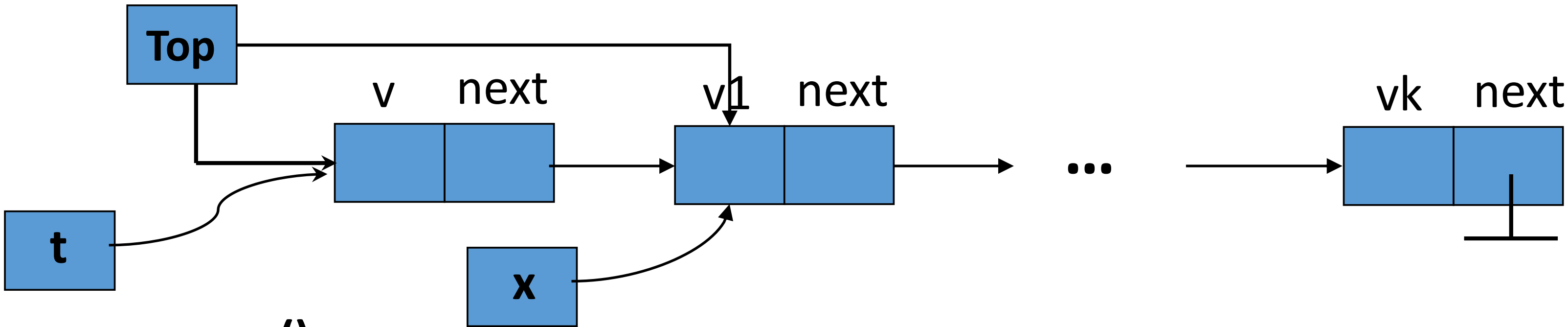
PDF-wfair-wrapper(await(B){C}) =   await(B){C} ?

However, in general await(B){C} cannot be PDF abstraction

Now consider a CAS impl:

Example: Treiber stack with partial pop

[Treiber'86]

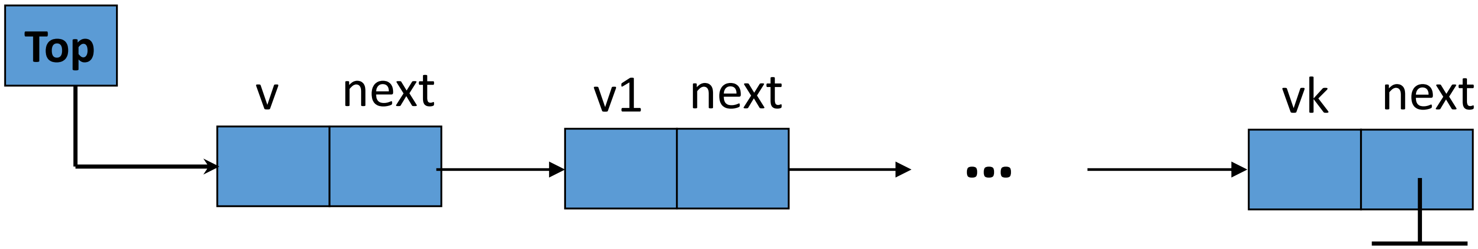



```
p-pop():
local b:=false, x, t, v;
while(!b){
    t := Top;
    if (t != null) {
        v := t.data;  x := t.next;
        b := cas(&Top, t, x);    }
}   return v;
```

blocked if stack is empty

Example: Treiber stack with partial pop

[Treiber'86]



p-pop():

```
local b:=false, x, t, v;
while(!b){
    t := Top;
    ) {
    ta;  x := t.next;
        b := cas(&Top, t, x);    }
}  return v;
```

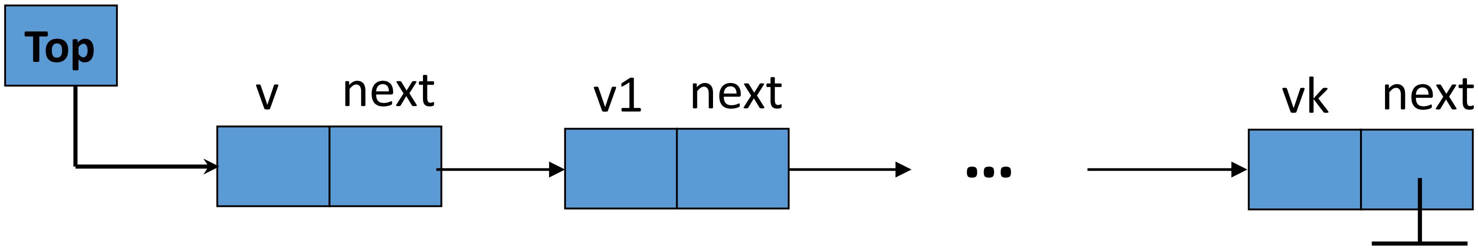
may not terminate if cas always fails

$[\ ]_{P\_POP()}$ print(1); || while(true){ $[\ ]_{PUSH(0)}$ }

May not print 1

Example: Treiber stack with partial pop

[Treiber'86]



p-pop():

```
local b:=false, x, t, v;
while(!b){
    t := Top;
    if (t != null) {
        v := t.data;  x := t.next;
        b := cas(&Top, t, x);    }
} return v;
```

$[\ ]_{P\_POP()}$ print(1); $\parallel$ while(true){ $[\ ]_{PUSH(0)}$ }

May not print 1

But impl satisfies PDF!

Provides too much progress than PDF impl.
Fail to consider delay by env.

await(B){C} ?

However, in general await(B){C} cannot be PDF abstraction

```
p-pop():
local b:=false, x, t, v;
while(!b){
   t := Top;
   if (t != null) {
      v := t.data;  x := t.next;
      b := cas(&Top, t, x);    }
}   return v;
```

```
P-POP() { await( !emp(S) ) { pops S }
PUSH(x) { ret x::S }
```

```
[ ]P_POP()          ||   while(true){
print(1);           ||       [ ]PUSH(0)
                    ||   }
```

May not print 1

Must print 1

Our solution (first attempt)

PDF-wfair-wrapper(await(B){C}) =

initialize to false

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;

Our solution (first attempt)

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;

My success delays others

Our solution (first attempt)

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;

My success delays others

Set it back to false
(delay is temporary)

Our solution (first attempt)

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;

may not terminate if done is infinitely often true (even if B is always true)

$[\ ]_{P\_POP()}$ $\quad\|\quad$ while(true){
print(1); $\quad\|\quad$ $[\ ]_{PUSH(0)}$
}

Our solution (first attempt)

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;

However, either executes C and terminates, or gets blocked without executing C

Cannot abstract cases that are blocked after executing C!

Example: blocked after popping items

push'(v):

1 push(v);

2 DLY_LOOP;

pop'():

3 local v := pop();

4 DLY_LOOP;

5 return v;

DLY_LOOP =

await(¬done) { done := true };

done := false;

client:

```
r0 := pop'();        ||   push'(1);
print(r0);           ||   push'(2);
                     ||   r1 := pop'();
                     ||   print(r1);
                     ||   while (true) {
                     ||       push'(0);
                     ||   }
```

It's possible to only print 1, under weak fairness.

Example: blocked after popping items

```
PUSH(v):
  await(¬done) {
    S := v :: S;
    done := true; }
  done := false;

POP():
  local v;
  await(S != nil ∧ ¬done){
    v := head(S);  S := tail(S);
    done := true;   }
  done := false;
  return v;
```

client:

	PUSH(1);
	PUSH(2);
r0 := POP();	r1 := POP();
print(r0);	print(r1);
	while (true) {
	PUSH(0);
	}

It's impossible to only print 1, under weak fairness.

Not abstraction for push' and pop'

Our solution

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;
await($\neg$done){ };

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;
await($\neg$done){ };

allow the methods to not terminate if done is infinitely often true

PDF-sfair-wrapper(await(

while(done){ };
await(B $\wedge$ $\neg$done){ C; done := true; };
done := false;
while(done){ };

Code wrappers in summary

PSF-sfair-wrapper(await(B){C}) = await(B){C}

PSF-wfair-wrapper(await(B){C}) =

listid := listid ++ [(cid, ‘B’)];
await(B $\wedge$ cid = **enhd**(listid)){ C; listid := listid\cid; }

PDF-sfair-wrapper(await(B){C}) =

while(done){ };
await(B $\wedge$ $\neg$done){ C; done := true; }; done := false;
while(done){ };

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; }; done := false;
await($\neg$done){ };

Code wrappers in summary

PSF-sfair-wrapper(await(B){C}) = await(B){C}

PSF-wfair-wrapper(await(B){C}) =

listid := listid ++ [(cid, ‘B’)];
await(B $\wedge$ cid = **enhd**(listid)){ C; listid := listid\cid; }

PDF-sfair-wrapper(await(B){C}) =

while(done){ };
await(B $\wedge$ $\neg$done){ C; done := true; }; done := false;
while(done){ };

PDF-wfair-wrapper(await(B){C}) =

await(B $\wedge$ $\neg$done){ C; done := true; }; done := false;
await($\neg$done){ };

	PSF	PDF
Strong fairness	!	!
Weak fairness	!	!

Abstraction Theorem

- Linearzability + PSF/PDF $\Leftrightarrow$ Contextual Refinements
  - Abstractions are generated by corresponding wrappers
  - Justify the wrappers: they are refined by PSF/PDF impl
  - Justify PSF/PDF: they imply progress-aware CR

- Allow modular verification of clients
  - Instead of reasoning about C[O], we reason about C[$\mathcal{A}$], if O is linearizable and PSF/PDF w.r.t S

Program logic for PSF & PDF objects

Extend the logic LiLi for SF & DF objects [Liang & Feng'16]

If D, R, G $\vdash$ {p} O : S , then we have:

a) O is *lineariza*... t... S

b) O is PDF

c) if R & G sa...

O is PSF

Extend LiLi's inference rules to support await & strong/weak fairness

Conclusion

- Study progress of objects with partial methods
  - 2 new progress properties: PSF & PDF
  - 4 wrappers to generate abstractions for PSF/PDF objects under strongly/weakly fair scheduling
  - A new program logic for PSF & PDF

Thank you!