Exam Questions
Proof Methods for Concurrent Programs

11 February 2010

Instructions: only printed documents are authorised. You can admit the result of one question and move on. Leave optional questions until the end.

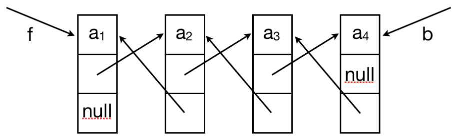
Your goal is to implement and prove correct some support functions used by the *A-Steal* multiprocessor scheduling algorithm.

Exercise 1. (Data structures)

A *double ended queue* (often abbreviated to *deque*) is a data structure that implements a queue for which elements can be added to or removed from the front (head) or back (tail). We follow the C++ interface (`std::deque`) to name the different operations:

`push_back`	insert element at back
`push_front`	insert element at front
`pop_back`	remove element from back
`pop_front`	remove element from front

We use *doubly-linked lists* to implement deques. A doubly-linked list can be graphically represented as



where `f` and `b` are the pointers to the front and back elements. When the doubly-linked list is empty, `f` = `b` = `null`. An implementation of the `push_front` procedure is given below:

```
push_front(a,f,b) {
  t := new (a,f,null);
  if (f = null) then b := t else [f+2] := t;
  f := t;
}
```

Remark that our notation for procedures assumes that the arguments (e.g. `a`, `f`, `b`) are passed by reference (in other terms, they can be considered global variables), while the other variables (e.g. `t`) are local.

1. Give a sequential implementation to the `pop_back(f,b,r)` and `pop_front(f,b,r)` procedures:
   - `f` and `b` are as above, while `r` is used to return either the popped value or `null` if the deque is empty;
   - your code should not leak memory.

   Answer.

```
pop_front (f,b,r) {
   if (f = null)
   then r := null
   else {
     t := f;
     f := [f+1];
     if (f = null)
```

```
          then b := null
          else [f+2] := null;
        r := [t];
        dispose (t,t+1,t+2);
      }
   }

   pop_back (f,b,r) {
      if (b = null)
      then r := null
      else {
        t := b;
        b := [b+2];
        if (b = null)
          then f := null
          else [b+1] := null;
        r := [t];
        dispose (t,t+1,t+2);
      }
   }
```

Let α range over lists of values (ϵ denotes the empty list and $\cdot$ is the concatenation operator). Recall the recursive specification of doubly-linked list segments:

$$\begin{aligned}\text{dlseg}\ \epsilon\ (\mathtt{f}, \mathtt{f'}, \mathtt{b'}, \mathtt{b}) &= \textit{empty} \ \wedge\ \mathtt{f} = \mathtt{b'} \ \wedge\ \mathtt{f'} = \mathtt{b} \\ \text{dlseg}\ (\mathtt{a} \cdot \alpha)\ (\mathtt{f}, \mathtt{f'}, \mathtt{b'}, \mathtt{b}) &= \exists \mathtt{j}.\, \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{f'} \ *\ \text{dlseg}\ \alpha\ (\mathtt{j}, \mathtt{f}, \mathtt{b'}, \mathtt{b})\end{aligned}$$

and consider the definition of doubly-linked lists below:

$$\text{dls}\ \alpha\ (\mathtt{f}, \mathtt{b}) = \text{dlseg}\ \alpha\ (\mathtt{f}, \mathtt{null}, \mathtt{null}, \mathtt{b})$$

Intuitively dls α $(\mathtt{f}, \mathtt{b})$ should be read "f, b are the ends of a doubly-linked list representing the list of values α".

2. Prove the following properties:

$$\begin{array}{ll} (a.) & \text{dls}\ (\mathtt{a} \cdot \epsilon)\ (\mathtt{f}, \mathtt{b}) \Leftrightarrow \mathtt{f} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{f} \\ (b.) & \text{dls}\ \alpha\ (\mathtt{f}, \mathtt{b}) \Rightarrow (\mathtt{f} = \mathtt{null} \Rightarrow (\alpha = \epsilon \wedge \mathtt{b} = \mathtt{null})) \\ (c.) & \text{dls}\ \alpha\ (\mathtt{f}, \mathtt{b}) \Rightarrow (\mathtt{f} \neq \mathtt{null} \Rightarrow (\exists \mathtt{a}, \alpha'.\ \alpha = \mathtt{a} \cdot \alpha')) \end{array}$$

Answer.
(a.)

$$\begin{array}{l} \text{dls}\ \mathtt{a}\ (\mathtt{f}, \mathtt{b}) \\ \Leftrightarrow\ \text{dlseg}\ \mathtt{a}\ (\mathtt{f}, \textit{null}, \textit{null}, \mathtt{b}) \\ \Leftrightarrow\ \exists \mathtt{j}.\ \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}, \textit{null}\ *\ \text{dlseg}\ \epsilon\ (\mathtt{j}, \mathtt{f}, \textit{null}, \mathtt{b}) \\ \Leftrightarrow\ \exists \mathtt{j}.\ \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}, \textit{null}\ *\ (\textit{empty} \wedge \mathtt{j} = \textit{null} \wedge \mathtt{f} = \mathtt{b}) \\ \Leftrightarrow\ \mathtt{f} \mapsto \mathtt{a}, \textit{null}, \textit{null}\ \wedge\ \mathtt{f} = \mathtt{b} \end{array}$$

(b.) Suppose that $(s, h) \vdash$ dls α $(\mathtt{f}, \mathtt{b})$ and $\mathtt{f} = \textit{null}$. Since $\mathtt{f} = \textit{null}$, it cannot exists a $\mathtt{j}$ and a $h' \subseteq h$ such that $(s, h') \vdash \mathtt{f} \mapsto \_, \mathtt{j}, \_$. It must then hold that $(s, h) \vdash \textit{empty} \wedge f = \textit{null} \wedge \mathtt{b} = \textit{null}$. We conclude that (dls α $(\mathtt{f}, \mathtt{b}) \wedge \mathtt{f} = \textit{null}) \Rightarrow (\textit{empty} \wedge \mathtt{b} = \textit{null})$ and the result follow by uncurrying.

(c.) Suppose that $(s, h) \vdash$ dls α $(\mathtt{f}, \mathtt{b})$ and $\mathtt{f} \neq \textit{null}$. Since dls α $(\mathtt{f}, \mathtt{b})$ = dlseg α $(\mathtt{f}, \textit{null}, \textit{null}, \mathtt{b})$ and $\mathtt{f} \neq \textit{null}$, it must exist $\mathtt{j}, \mathtt{a}, \alpha'$ such that $(s, h) \vdash \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{f'} *$ dlseg α' $(\mathtt{j}, \mathtt{f}, \textit{null}, \mathtt{b})$. By the definition of dlseg, we deduce $(s, h) \vdash$ dlseg $a \cdot \alpha'$ $(\mathtt{f}, \textit{null}, \textit{null}, \mathtt{b})$, and in turn $(s, h) \vdash$ dls $a \cdot \alpha'$ $(\mathtt{f}, \mathtt{b})$. Thus, $\alpha = \mathtt{a} \cdot \alpha'$ for some $\mathtt{a}, \alpha'$.

3. Prove that the operations on deques satisfy the specifications below:

(*a.*)	$\{\text{dls } \alpha\ (\texttt{f},\texttt{b})\}$	`push_front(a,f,b)`	$\{\text{dls } (\texttt{a}\cdot\alpha)\ (\texttt{f},\texttt{b})\}$
(*b.*)	$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b})\}$	`pop_front(f,b,r)`	$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \ \wedge\ \texttt{r} = \texttt{null}\}$
(*c.*)	$\{\text{dls } (\texttt{a}\cdot\alpha)\ (\texttt{f},\texttt{b})\}$	`pop_front(f,b,r)`	$\{\text{dls } \alpha\ (\texttt{f},\texttt{b}) \ \wedge\ \texttt{r} = \texttt{a}\}$
(*d. optional*)	$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b})\}$	`pop_back(f,b,r)`	$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \ \wedge\ \texttt{r} = \texttt{null}\}$
(*e. optional*)	$\{\text{dls } (\alpha\cdot\texttt{a})\ (\texttt{f},\texttt{b})\}$	`pop_back(f,b,r)`	$\{\text{dls } \alpha\ (\texttt{f},\texttt{b}) \ \wedge\ \texttt{r} = \texttt{a}\}$

Answer.

(a.) $\{\text{dls } \alpha\ (\texttt{f},\texttt{b})\}$ `push_front(a,f,b)` $\{\text{dls } \texttt{a}\cdot\alpha\ (\texttt{f},\texttt{b})\}$.

$\{\text{dls } \alpha\ (\texttt{f},\texttt{b})\}$
  `t := new (a,f,null);`
$\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \text{dls } \alpha\ (\texttt{f},\texttt{b})\}$
  `if (f = null) then`
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \text{dls } \alpha\ (\texttt{f},\texttt{b}) \wedge \texttt{f} = null\}$
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null \wedge \texttt{b} = null \wedge \texttt{f} = null \wedge \alpha = \epsilon\}$
 $\{\texttt{t} \mapsto \texttt{a}, null, null \wedge \texttt{b} = null \wedge \texttt{f} = null \wedge \alpha = \epsilon\}$
   `b := t`
 $\{\texttt{t} \mapsto \texttt{a}, null, null \wedge \texttt{b} = \texttt{t} \wedge \texttt{f} = null \wedge \alpha = \epsilon\}$
$\{\text{dls } \texttt{a}\cdot\alpha\ (\texttt{t},\texttt{b})\}$
  `else`
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \text{dls } \alpha\ (\texttt{f},\texttt{b}) \wedge \texttt{f} \neq null\}$
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \text{dls } \alpha\ (\texttt{f},\texttt{b}) \wedge \texttt{f} \neq null \wedge \alpha = \texttt{a}'\cdot\alpha'\}$
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \exists \texttt{j}.\ \texttt{f} \mapsto \texttt{a'},\texttt{j}, null\ * \text{dlseg } \alpha'\ (\texttt{j},\texttt{f}, null, \texttt{b})\}$
   `[f+2] := t;`
 $\{\texttt{t} \mapsto \texttt{a},\texttt{f}, null\ *\ \exists \texttt{j}.\ \texttt{f} \mapsto \texttt{a'},\texttt{j},\texttt{t}\ * \text{dlseg } \alpha'\ (\texttt{j},\texttt{f}, null, \texttt{b})\}$
$\{\text{dls } \texttt{a}\cdot\alpha\ (\texttt{t},\texttt{b})\}$
  `f := t;`
$\{\text{dls } \texttt{a}\cdot\alpha\ (\texttt{f},\texttt{b})\}$

(b.)

$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b})\}$
  `if (f = null) then`
$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \wedge \texttt{f} = \texttt{null}\}$
  `r := null`
$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \wedge \texttt{f} = \texttt{null} \wedge \texttt{r} = \texttt{null}\}$
  `else {`
$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \wedge \texttt{f} \neq \texttt{null}\}$
$\{\exists \text{a}, \alpha.\ \epsilon = \texttt{a}\cdot\alpha\}$
$\{false\}$

```
    t := f;
    f := [f+1];
    if (f = null)
    then b := null
    else [f+2] := null;
    r := [t];
    dispose (t,t+1,t+2);
  }
```

$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \wedge \texttt{f} = \texttt{null} \wedge \texttt{r} = \texttt{null}\}$
$\{\text{dls } \epsilon\ (\texttt{f},\texttt{b}) \wedge \texttt{r} = \texttt{null}\}$

(c.) We prove separately the case where $\alpha = \epsilon$ and the case where $\alpha = \texttt{a}'\cdot\alpha'$ for some $\texttt{a}', \alpha'$. If $\alpha = \epsilon$ we can derive:

$\{\mathrm{dls}\ \mathtt{a} \cdot \epsilon\ (\mathtt{f}, \mathtt{b})\}$

`if (f = null) then`

$\{\mathit{false}\}$

`r := null`

`else {`

$\{\mathrm{dls}\ \mathtt{a} \cdot \epsilon\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{f} \neq \mathtt{null}\}$

$\{\mathtt{f} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{f}\}$

`t := f;`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{f} = \mathtt{t}\}$

`f := [f+1];`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{t} \wedge \mathtt{f} = \mathtt{null}\}$

`if (f = null) then`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{t} \wedge \mathtt{f} = \mathtt{null}\}$

`b := null`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{null} \wedge \mathtt{f} = \mathtt{null}\}$

`else`

$\{\mathit{false}\}$

`[f+2] := null;`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{null} \wedge \mathtt{f} = \mathtt{null}\}$

`r := [t];`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{null}, \mathtt{null} \wedge \mathtt{b} = \mathtt{null} \wedge \mathtt{f} = \mathtt{null} \wedge \mathtt{r} = \mathtt{a}\}$

`dispose (t,t+1,t+2);`

$\{\mathit{empty} \wedge \mathtt{b} = \mathtt{null} \wedge \mathtt{f} = \mathtt{null} \wedge \mathtt{r} = \mathtt{a}\}$

$\{\mathrm{dls}\ \epsilon\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{r} = \mathtt{a}\}$

`}`

$\{\mathrm{dls}\ \epsilon\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{r} = \mathtt{a}\}$

If $\alpha = \mathtt{a}' \cdot \alpha'$ for some $\mathtt{a}', \alpha'$, we can derive:

$\{\mathrm{dls}\ \mathtt{a} \cdot \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b})\}$

`if (f = null) then`

$\{\mathit{false}\}$

`r := null`

`else {`

$\{\mathrm{dls}\ \mathtt{a} \cdot \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{f} \neq \mathtt{null}\}$

$\{\exists \mathtt{j}.\ \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{j}, \mathtt{f}, \mathtt{null}, \mathtt{b})\}$

`t := f;`

$\{\exists \mathtt{j}.\ \mathtt{t} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{j}, \mathtt{t}, \mathtt{null}, \mathtt{b}) \wedge \mathtt{f} = \mathtt{t}\}$

`f := [f+1];`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{j}, \mathtt{t}, \mathtt{null}, \mathtt{b}) \wedge \mathtt{f} = \mathtt{j}\}$

`if (f = null) then`

$\{\mathit{false}\}$

`b := null`

`else`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{j}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{j}, \mathtt{t}, \mathtt{null}, \mathtt{b}) \wedge \mathtt{f} = \mathtt{j}\}$

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{f}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{t}, \mathtt{null}, \mathtt{b})\}$

`[f+2] := null;`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{f}, \mathtt{null} * \mathrm{dlseg}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{null}, \mathtt{null}, \mathtt{b})\}$

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{f}, \mathtt{null} * \mathrm{dls}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b})\}$

`r := [t];`

$\{\mathtt{t} \mapsto \mathtt{a}, \mathtt{f}, \mathtt{null} * \mathrm{dls}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{r} = \mathtt{a}\}$

`dispose (t,t+1,t+2);`

$\{\mathrm{dls}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{r} = \mathtt{a}\}$

`}`

$\{\mathrm{dls}\ \mathtt{a}' \cdot \alpha'\ (\mathtt{f}, \mathtt{b}) \wedge \mathtt{r} = \mathtt{a}\}$

4. Recall the definition of *precise predicate*. Is dlseg α (f,f',b',b) a precise predicate?

Answer. If f = b then the doubly-linked-list segment might be empty (thus denoting the empty heap), or circular (thus denoting a non-empty heap). This implies that dlseg α (f,f',b',b) is not precise. It can be shown that if f' = b' = *null* then the segment is non-touching: dls α (f,b) is precise (however the proof is far from easy - see Sec. 4.3 of Reynold's lectures).

In what follows you can assume that dls α (f,b) is precise.

Exercise 2. (The *A-Steal* scheduler)

The *A-Steal* algorithm implements task scheduling for several processors. A separate deque with the thread-identifiers (tids) of the threads to be executed is maintained for each processor. To execute the next thread, the processor gets the first element from the deque (using the pop_front deque operation). If the current thread forks, it is put back to the front of the deque (push_front) and a new thread is executed. When one of the processors finishes execution of its own threads (i.e. its deque is empty), it can *steal* a thread from another processor: it gets the last element from the deque of another processor (pop_back) and executes it.

We introduce a handy C-like notation for arrays: if a points to a series of n contiguous memory locations, we write a[i] for [a+i].

Consider a system with n processors, indexed from 0 to $n-1$. To implement the A-Steal algorithm we use two arrays, named a and l, and n deques. The array a has $2n$ entries, and a[2*i] and a[2*i+1] are the front and back addresses of the deque associated to the processor i. The elements of the deques are tids (you can assume that a tid is just an integer). The array l has n entries, and each l[i] is a resource that protects the locations a[2*i] and a[2*i+1] and the associated deque.

1. Implement the fork and schedule procedures of the A-Steal algorithm, according to their informal specification below:

   fork(proc,tid) stores tid at the front of the deque of processor proc;

   schedule(proc):tid pops (and returns) the tid at the front of the deque of processor proc; if the deque of processor proc is empty, it pops (and returns) the tid at the back of the deque of another processor which has a non-empty deque.

   The procedures fork and schedule should not fail (schedule might loop), and, needless to say, should be robust to concurrent invocations.

Answer.

```
fork (proc,tid) {
  with l[proc] do {
    push_front (tid, a[2*proc], a[2*proc+1]);
  }
}

schedule (proc) {
  with l[proc] do {
    tid := pop_front (a[2*proc], a[2*proc+1]);
  };
  while (tid = null) do {
    proc = (proc+1) mod n;
    with l[proc] do {
      tid := pop_back (a[2*proc], a[2*proc+1]);
    }
  }
}
```

2. Define the resource invariant associated to each resource `l[i]`.

Answer. The resource invariant of the resource `l[i]` is

$$\exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{i}], \mathtt{a}[2*\mathtt{i}+1])$$

3. Prove that

$$(a.) \qquad \{0 \leq \mathtt{proc} \leq n-1\}\ \mathtt{fork(proc,tid)}\ \{\mathit{true}\}$$

$$(b.) \qquad \{0 \leq \mathtt{proc} \leq n-1\}\ \mathtt{schedule(proc)}\ \{\mathtt{tid} \neq \mathtt{null}\}$$

Answer.
(a.)

$\{0 \leq \mathtt{proc} \leq n-1\}$
```
  with l[proc] do {
```
$\{0 \leq \mathtt{proc} \leq n-1 \land \exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
```
    push_front (tid, a[2*proc], a[2*proc+1]);
```
$\{\text{dls}\ \mathtt{tid}\cdot\alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
$\{\exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
```
  }
```
$\{\mathit{true}\}$

(b.)

$\{0 \leq \mathtt{proc} \leq n-1\}$
```
  with l[proc] do {
```
$\{0 \leq \mathtt{proc} \leq n-1 \land \exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
```
    tid := pop_front (a[2*proc], a[2*proc+1]);
```
$\{0 \leq \mathtt{proc} \leq n-1 \land \exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1]) \land (\mathtt{tid} = \mathtt{a} \lor \mathtt{tid} = \mathtt{null}\}$
```
  };
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null}\}$
```
  while (tid = null) do {
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null}\}$
```
    proc = (proc+1) mod n;
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null}\}$
```
    with l[proc] do {
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null} \land \exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
```
      tid := pop_back (a[2*proc], a[2*proc+1]);
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null} \land \exists\alpha.\ \text{dls}\ \alpha\ (\mathtt{a}[2*\mathtt{proc}], \mathtt{a}[2*\mathtt{proc}+1])\}$
```
    }
```
$\{0 \leq \mathtt{proc} \leq n-1 \land (\exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t} \lor \mathtt{tid} = \mathtt{null}\}$
```
  }
```
$\{0 \leq \mathtt{proc} \leq n-1 \land \exists \mathtt{t}.\ \mathtt{tid} = \mathtt{t}\}$