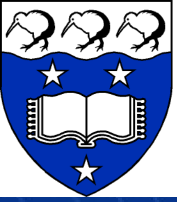
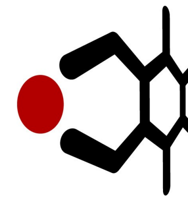


THE UNIVERSITY OF AUCKLAND
NEW ZEALAND



DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

Executing Esterel on Multicore Processors

Simon Yuan, Li Hsien Yoong, and Partha S Roop

2008-04-05

Background

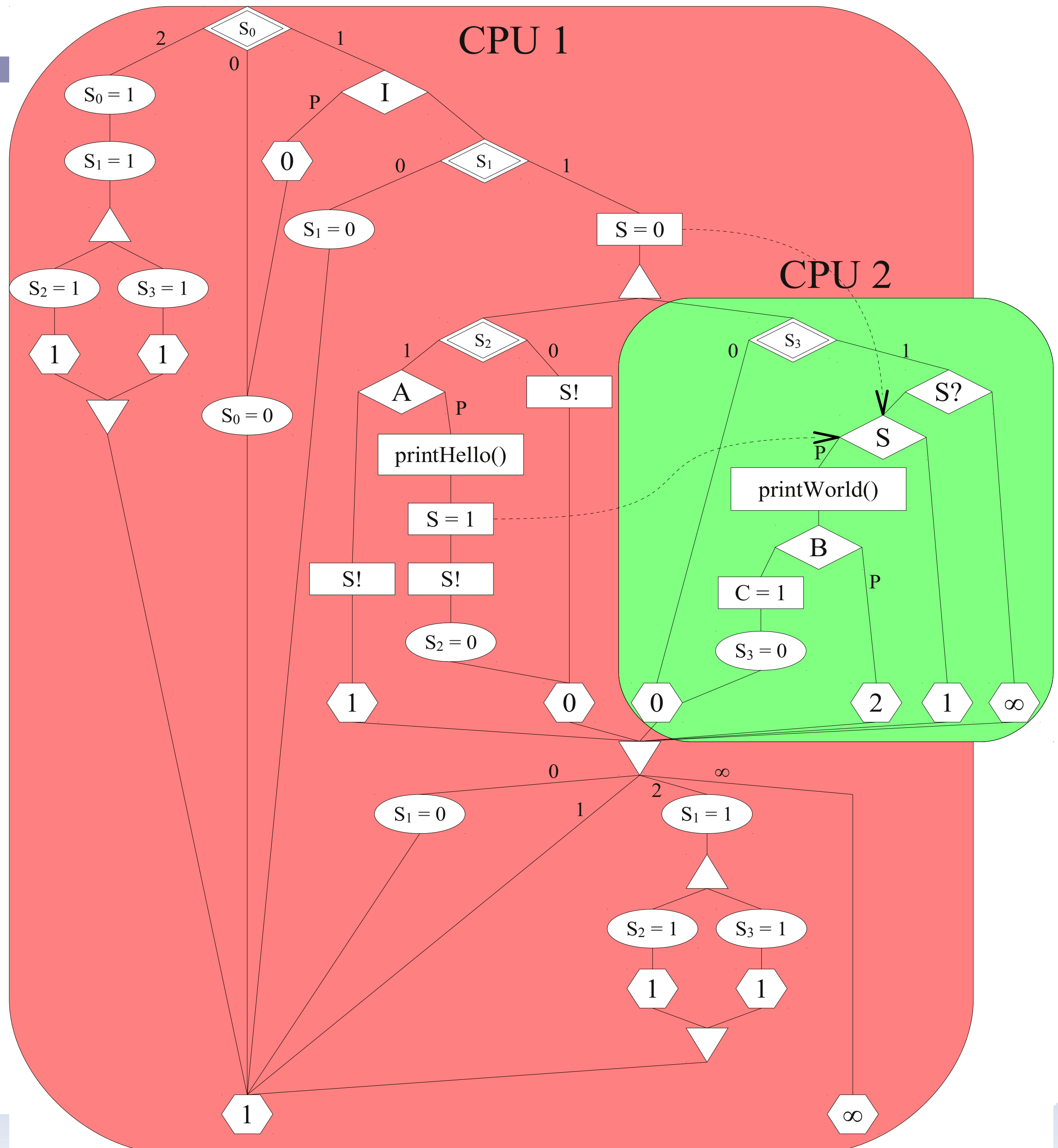
- Architecture-specific compilation of Esterel
  - Reactive processors, custom Esterel processors, multicores
- Distribution on multicore processors
  - Limitations and problems encountered
- Possible solutions

Current approach

- Prototype implementation
  - POSIX threads on standard multicore desktops
  - Multicore Microblaze
- Shared memory architecture
- Distribution on cores done according to layout of Esterel threads
  - Signal dependencies are resolved at run-time

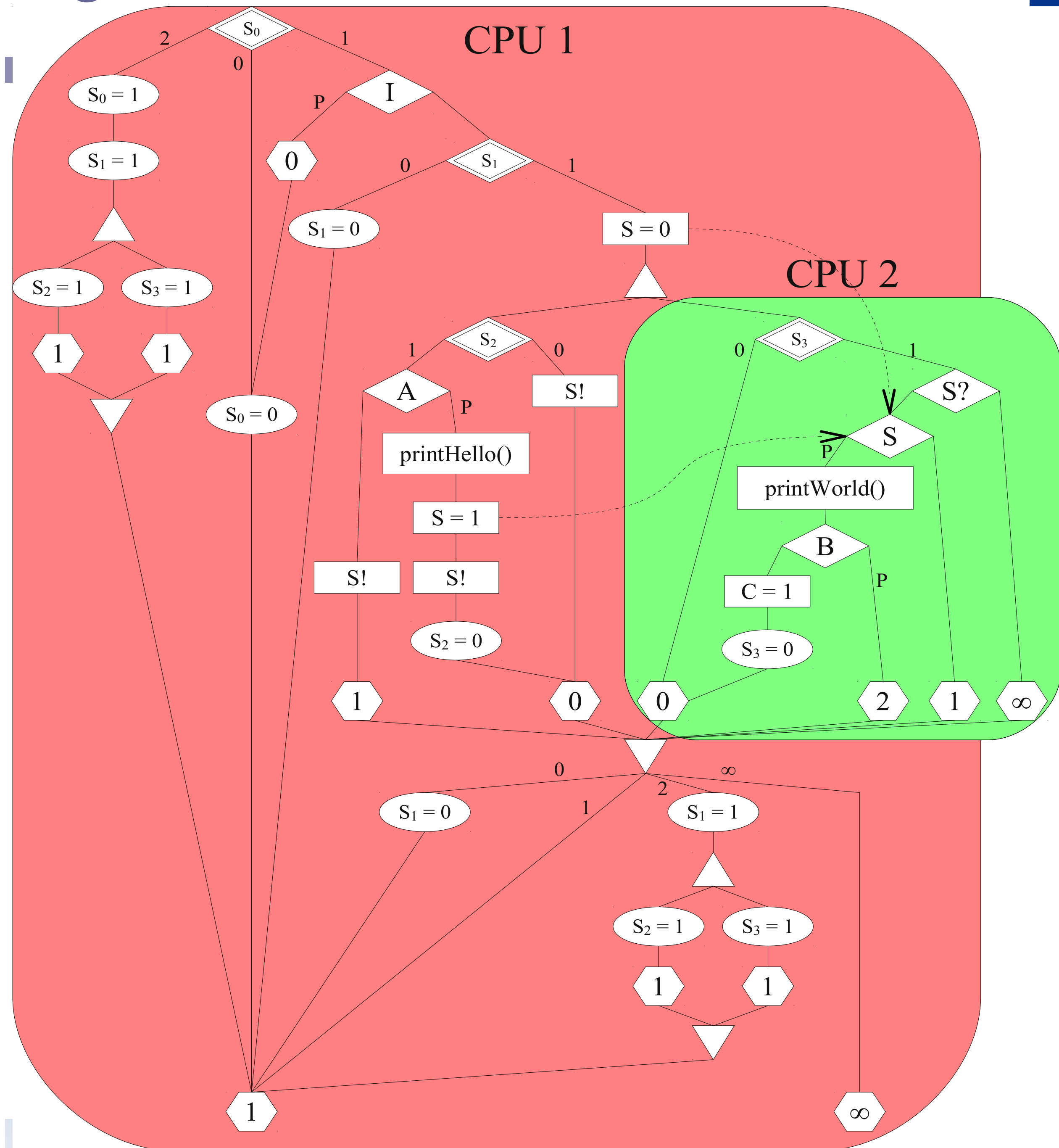
The intermediate format – GRC

```
module HelloWorld:
input A, B, I;
output C;
procedure printHello()();
procedure printWorld()();
abort
  loop
    trap Restart in
      signal S in
        await A;
        call printHello()();
        emit S;
        ||
        await S;
        call printWorld()();
        present B then
           exit Restart
        else
           emit C
        end present
      end signal;
      halt
    end trap
  end loop
when I
end module
```



Dynamic thread scheduling

```c
int term_0 = 0;
int term_1 = 0;
int term_2 = 0;
int lock_t2 = 0;

int esterel_module() {
    lock_t2 = 1;
    while (1) {
        term_1 = thread_1();
        term_2 = thread_2();
        term_0 = term_1 | term_2;
        if (term_0 != INFINITY)
            return term_0;
    }
}

int thread_1() {
    // ...
    if (lock_t2)
        return INIFINITY;
    else {
        // Inside guarded body
    }
    // ...
}

int thread_2() {
    // ..
    lock_t2 = 0;
    // ...
}
```



Code partitioning – implementation

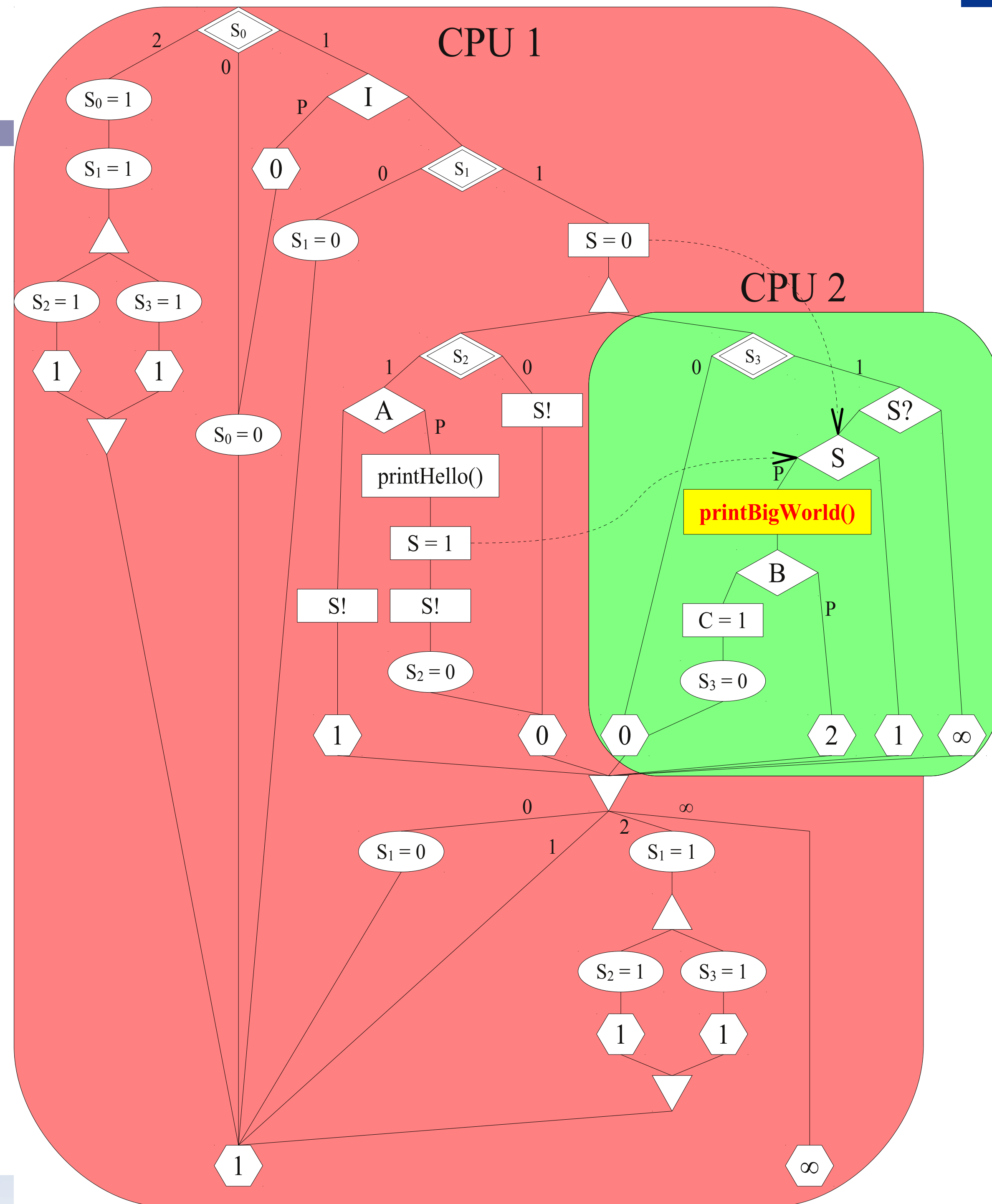
```
int main() {
    int cpu_id = get_proc_id();

    if (cpu_id == CPU0) {
        initiallize_cpu0();
    } else {
        initiallize_cpu1();
        esterel_module(1);
    }

    while (1) {
        sample_inputs();
        esterel_module(0);
        process_outputs();
    }
}
```

- Implementation using POSIX Threads and dual-core Microblaze are both done based on the same principle.

```
int esterel_module(int isSlaveCPU) {
    if (isSlaveCPU) {
    WAIT_FORK:
        wait_fork(pc); // Blocking
        goto *pc;
    }
    while (1) {
        // ...
        fork(&&CPU1);
        goto CPU0;
    CPU1:
        term_p1 = 0;
        do {
            term_1 = thread_1();
            term_2 = thread_2();
            term_p1 = term_1 | term_2;
        } while (term_p1 == INIFINITY);
        join(cpu1);
        goto WAIT_FORK;
    CPU0:
        term_p0 = 0;
        do {
            term_3 = thread_3();
            term_4 = thread_4();
            term_p0 = term_3 | term_4;
        } while (term_p0 == INIFINITY);
        join_all(); // Blocking
        // ...
        term_0 = term_p0 | term_p1;
        if (term_0 != INFINITY)
            return term_0;
    }
}
```

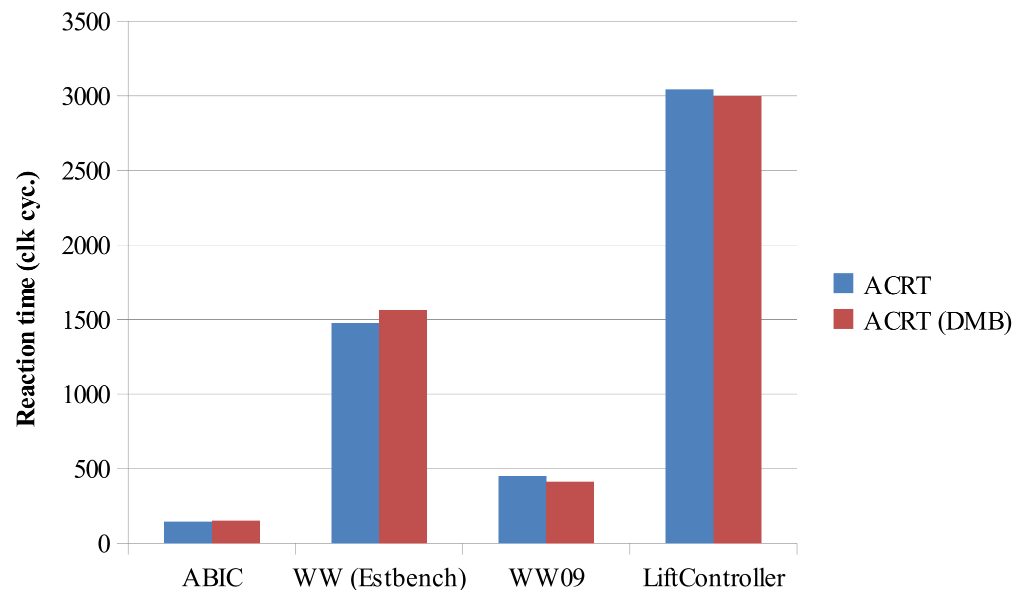
Limitations and problems

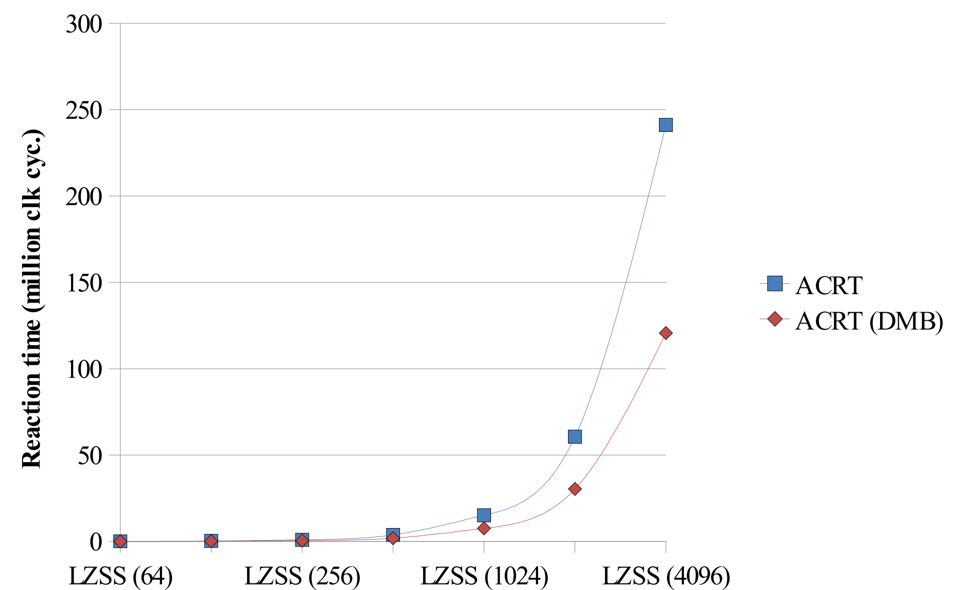
- Parallelisation limited to root level forks.
- Difficult to balance the load because of the above limitation.
- Signal locks and unlocks are not atomic.
    - Mutex required



Some results

- Except the LZSS example, executing on dual core Microblaze gave little speed up if not worse.
- What went wrong?





Some results

- The table shows the percentage of time a core was idling following a fork.
- Clearly the load balancing algorithm isn't working too well...

Examples	Max (%)	Min (%)	Avg (%)
ABIC	52	48	50
WW (Berry)	72	1	40
WW09	49	9	34
LiftController	96	94	94
LZSS	0	0	0

Solution One

- A naïve answer to the problem:
  - Parallelize only suitable programs!
- Use Esterel as a coordiation language to express control and parallel tasks for data-dominated programs.

Solution Two

- Implement signal locks using hardware mutex
- Perform dynamic load balancing
  - Simple hardware design to speed this up.

Implementing a hardware mutex

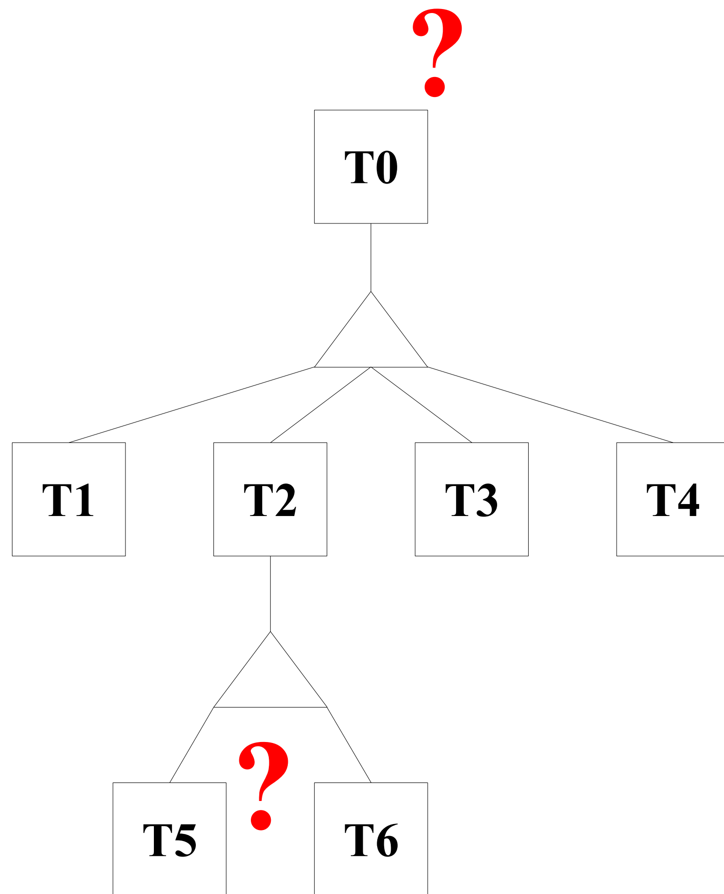
- Implement it as a peripheral on the Local Memory Bus (LMB) for single cycle memory mapped access.
- A look-up table will be used to implement counting semaphors.
- The table index is simply a unique ID to a lock variable.
- The hardware interface consists of 3 memory-mapped registers:one for writing initial value, one for unlocking, and one for reading

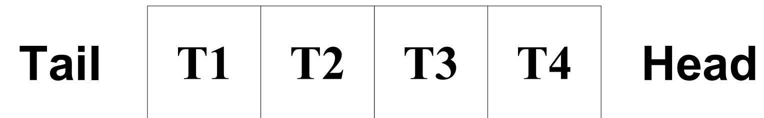
```
if (lock) {
  term &= INFINITY;
} else {
  if (guardedSig)
    /* do something */ ;
}
```

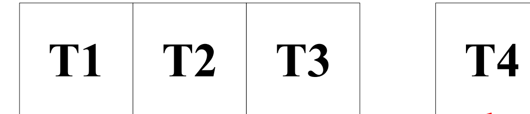
ID	Count
0x000009f8	2
0x00003c10	1
0x0000112c	0
0x00000934	1

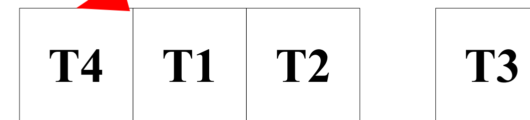
```
if (readLock(&guardedSig)) {
  term &= INFINITY;
} else {
  if (guardedSig)
    /* do something */ ;
}
```

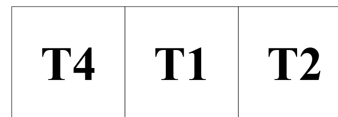
What does it take to do dynamic load balancing?



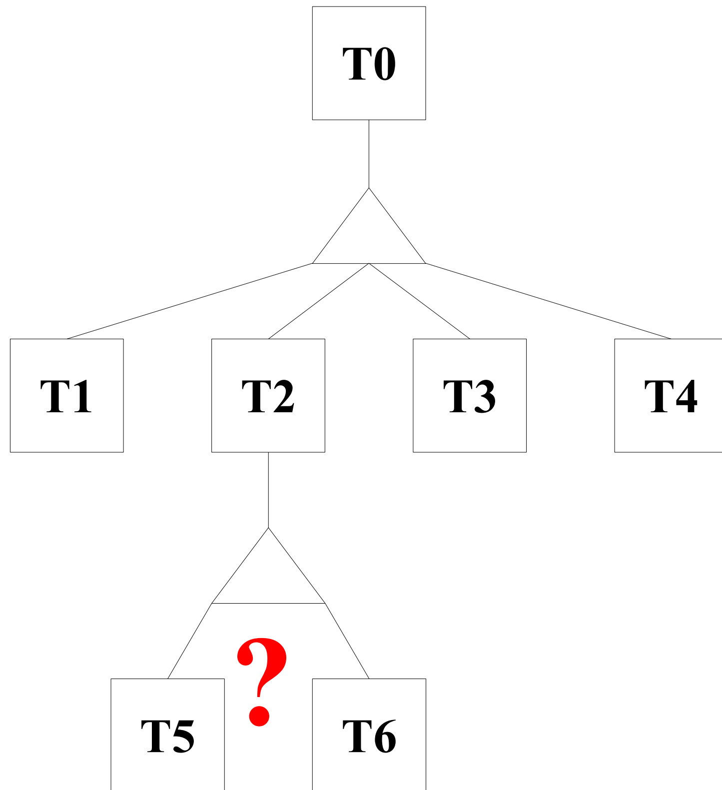


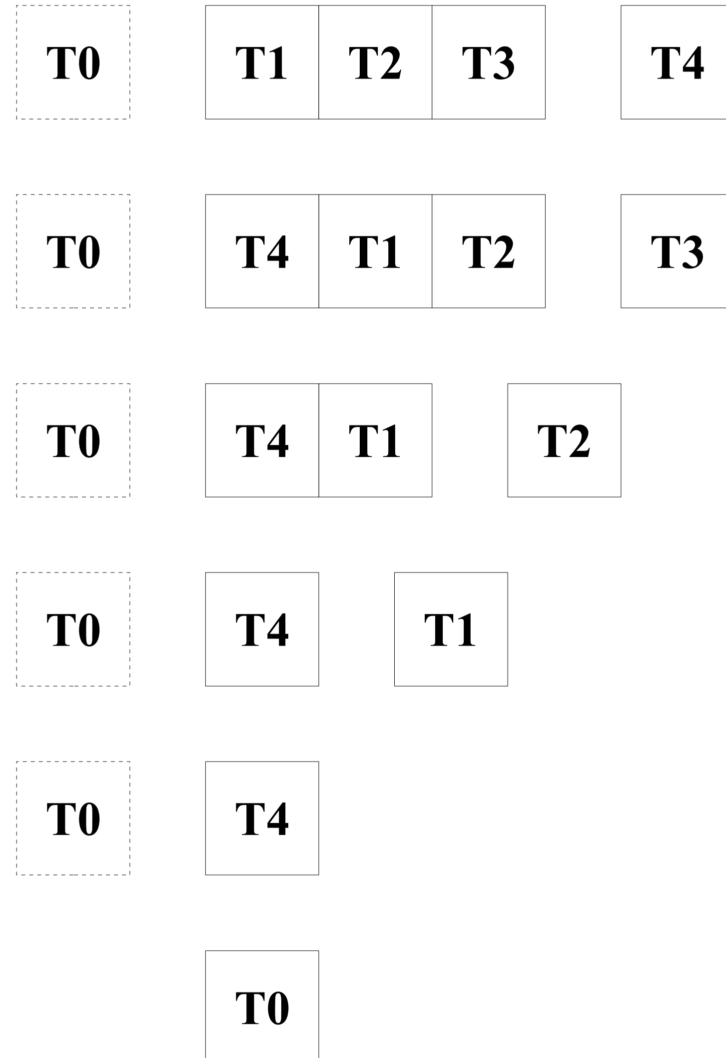




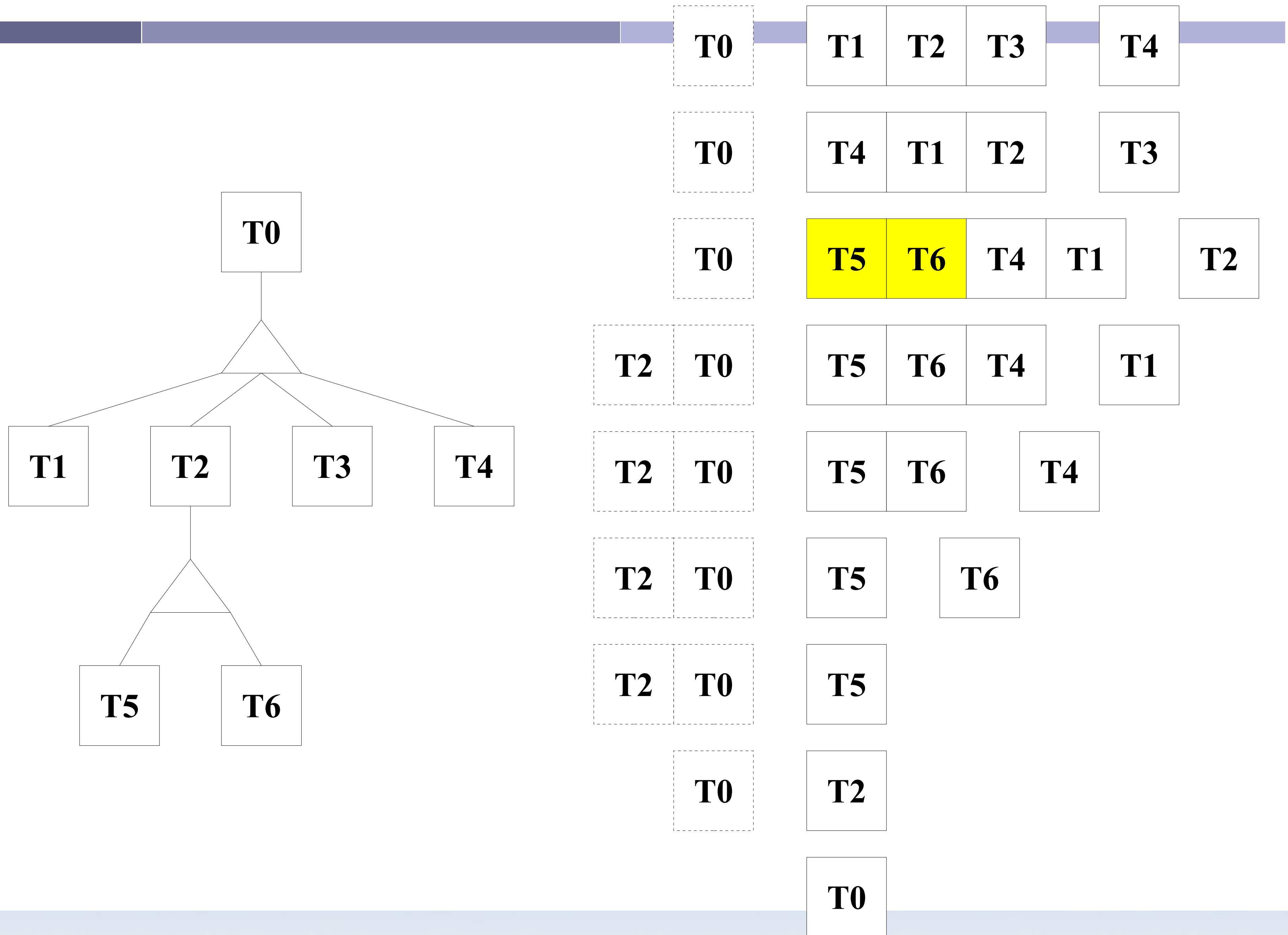


What does it take to do dynamic load balancing?

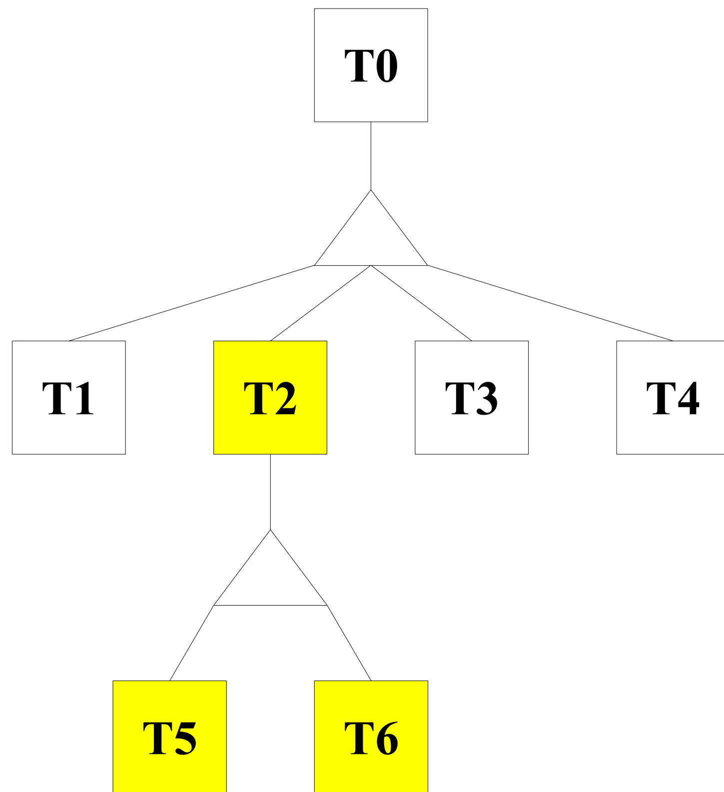


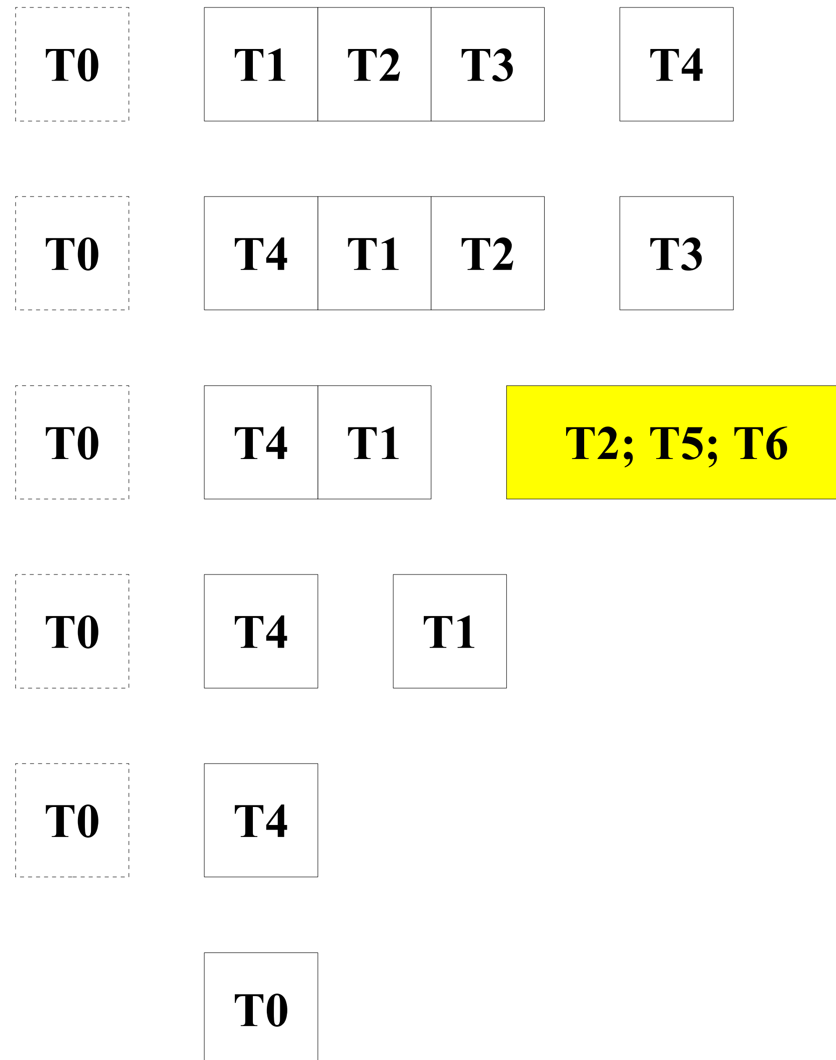


What does it take to do dynamic load balancing?

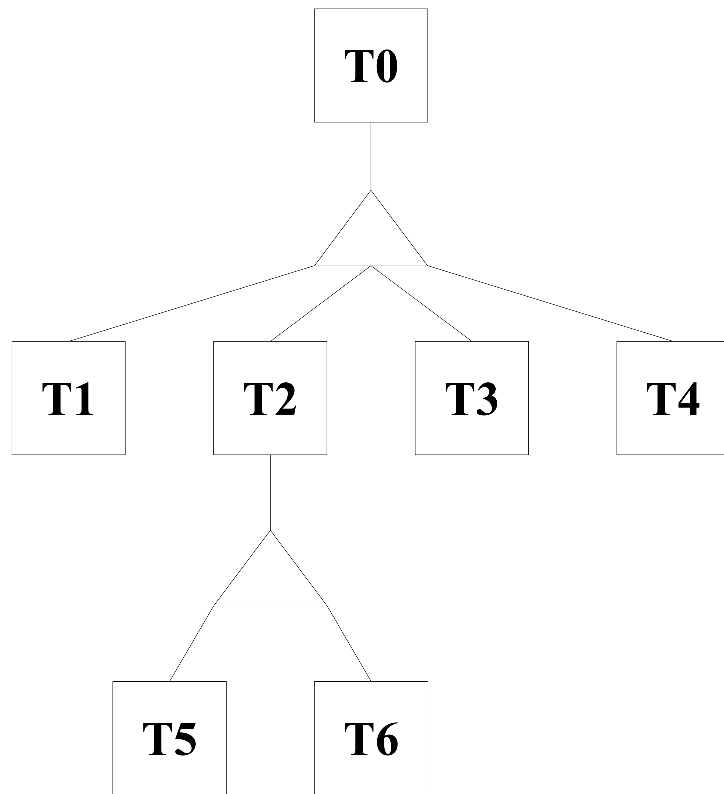


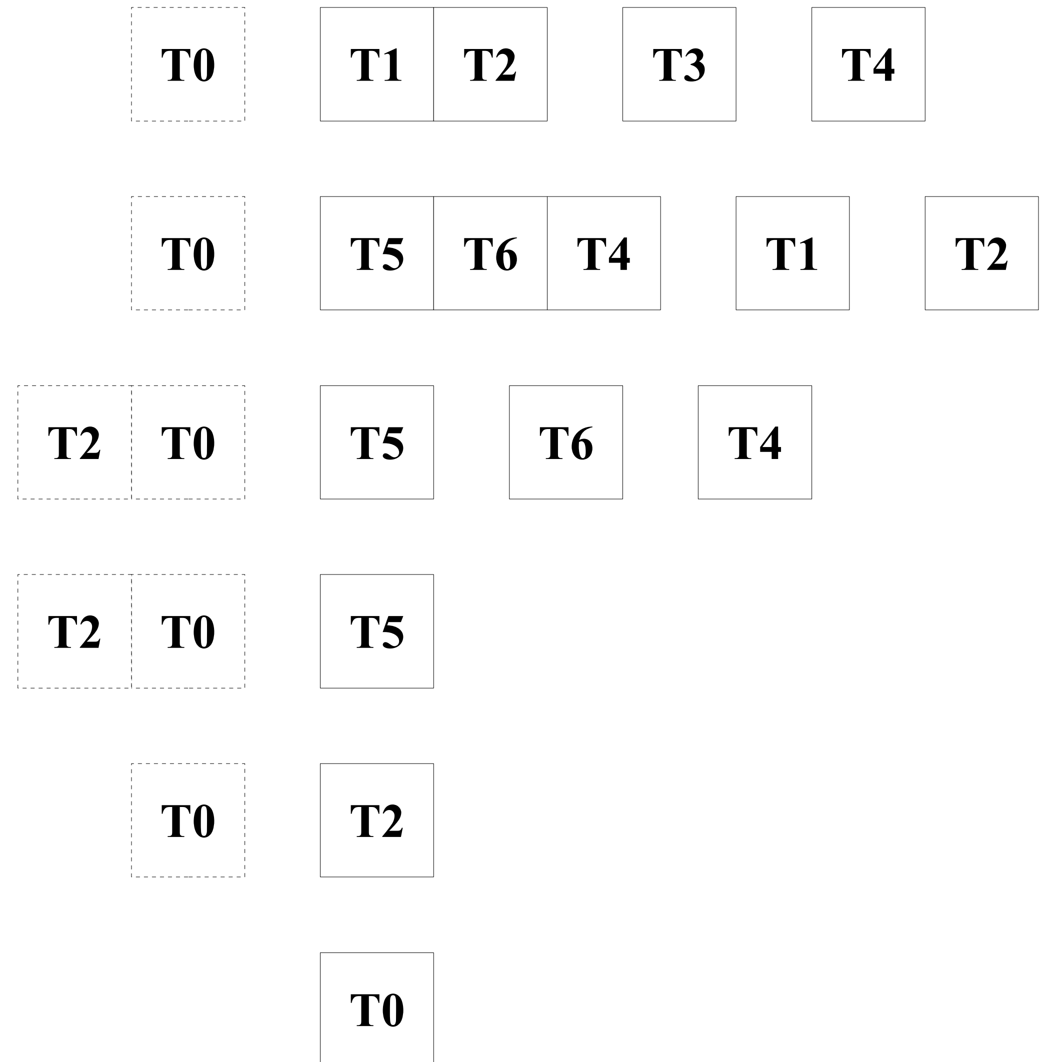
Flexible granularity





Distributing load for multicore





Concluding remarks

- Just a queue implemented in hardware for efficiency
- With finer granularity, we can achieve better load balance, but at the cost of more overhead from context switching