P2 - Preemptive Scheduling

Drew Zagieboylo

2/23/18

P1 Postmortem

P1 - Nonpreemptive

- yield()
  - allow another thread to run
  - w/o yield() -> single threaded behavior

```
int thread2(int* arg) {
  minithread_fork(thread3, NULL);
  printf("Thread 2.\n");
  minithread_yield();
  return 0;
}
```

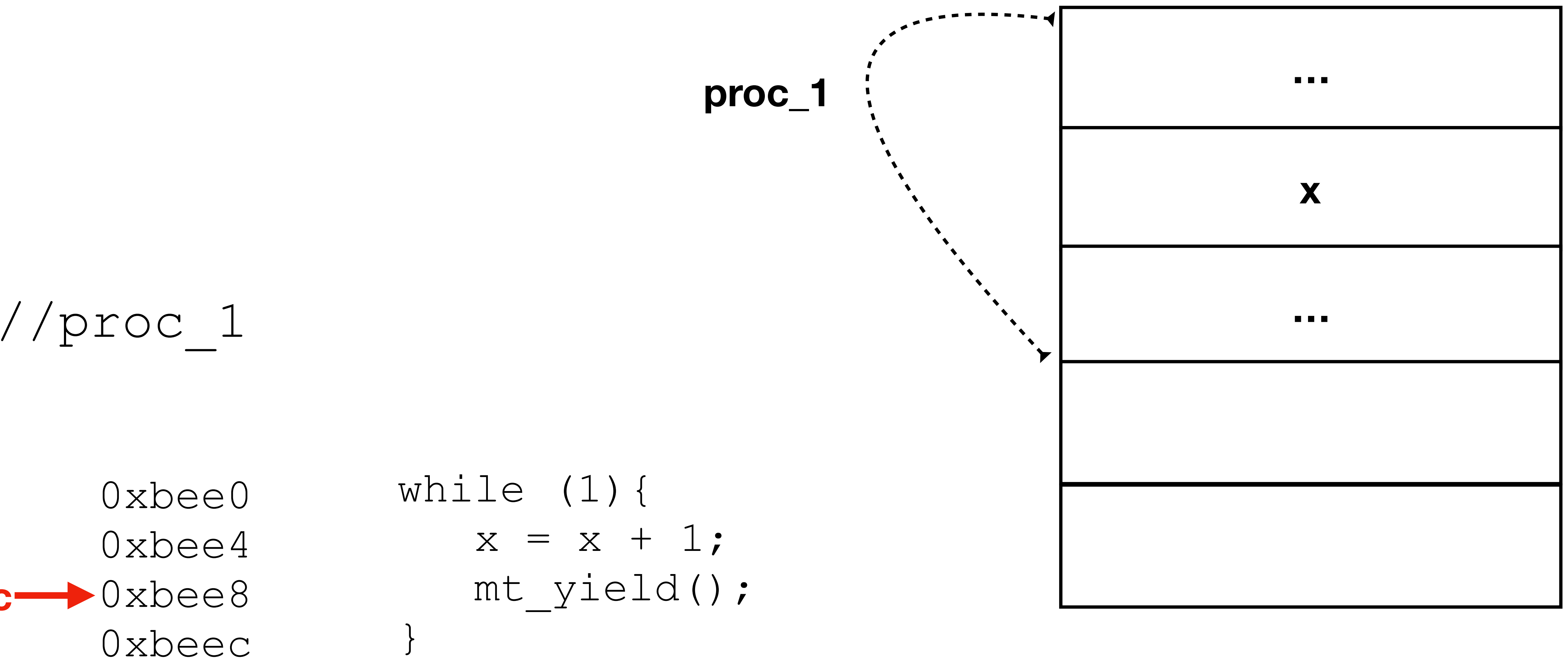
P2 - Thread Pre-emption

- How?
  - Interrupts! -> A type of Asynchronous execution
- When?
  - A timer -> uses HW clock
- What?
  - An ISR (interrupt service routine)

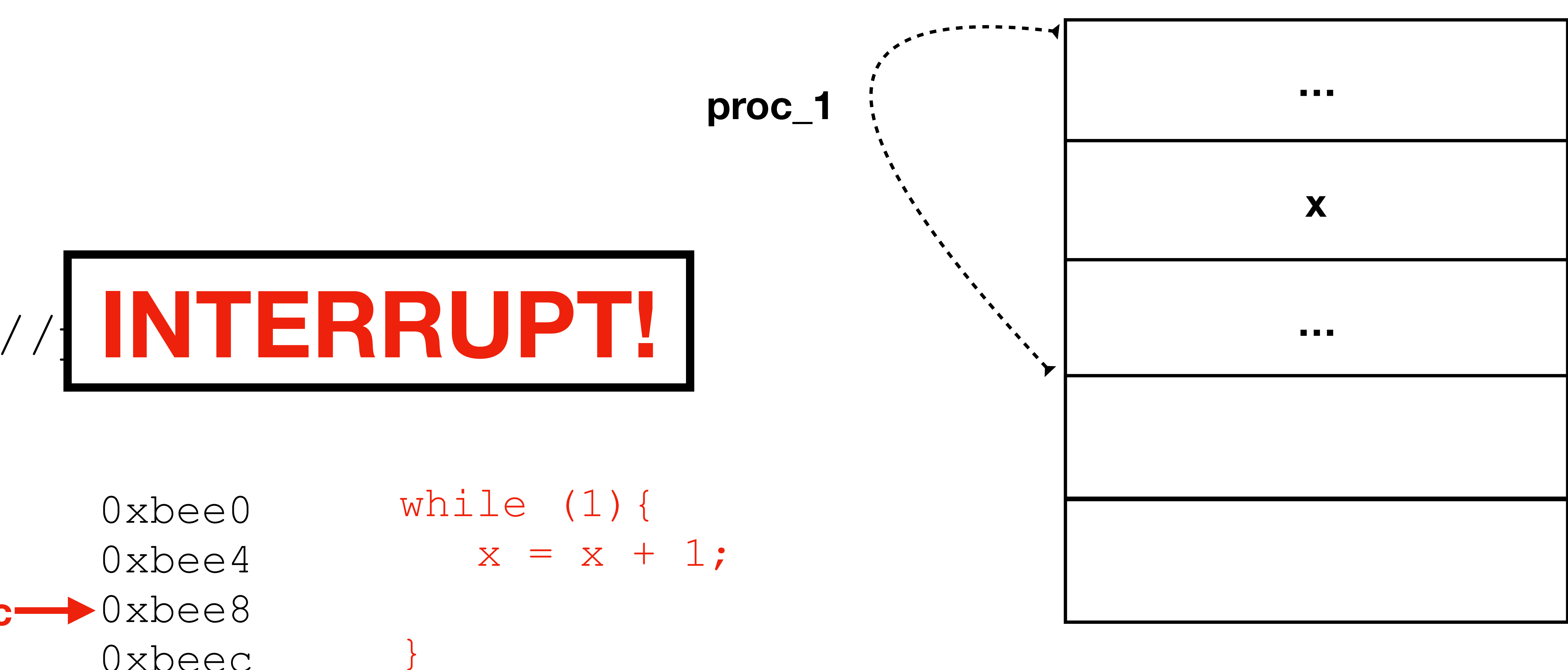
Interrupt Handling

- Description:
  - Register ISR for specific interrupt type
  - Enable/Disable Interrupts
  - Read Clock Value
- API:
  - minithread_clock_init(*isr*)
  - set_interrupt_level(*level*)
  - Global Variable: ‘*ticks*’
    - Number of clock ticks since OS start

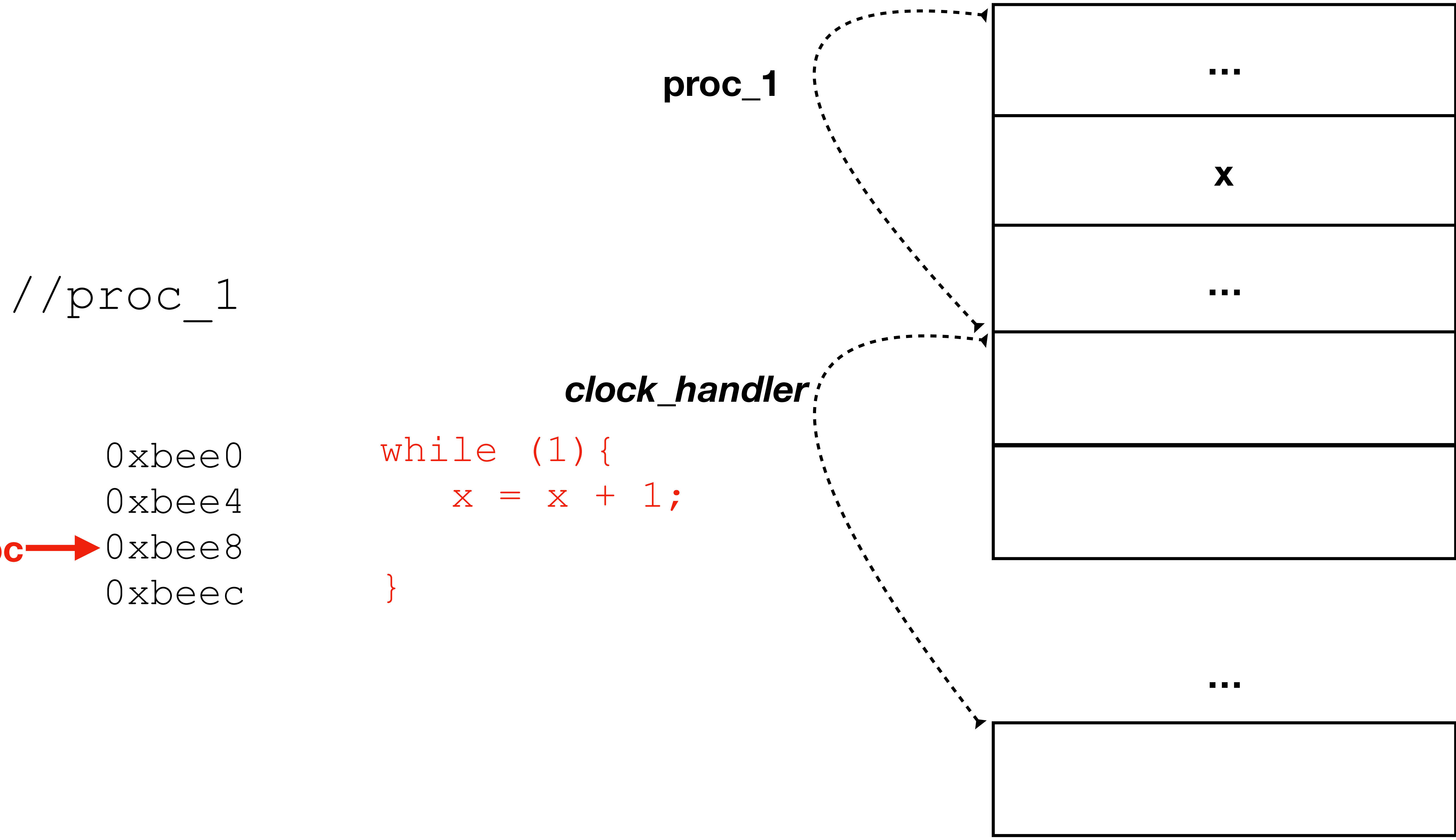
Interrupt Handling



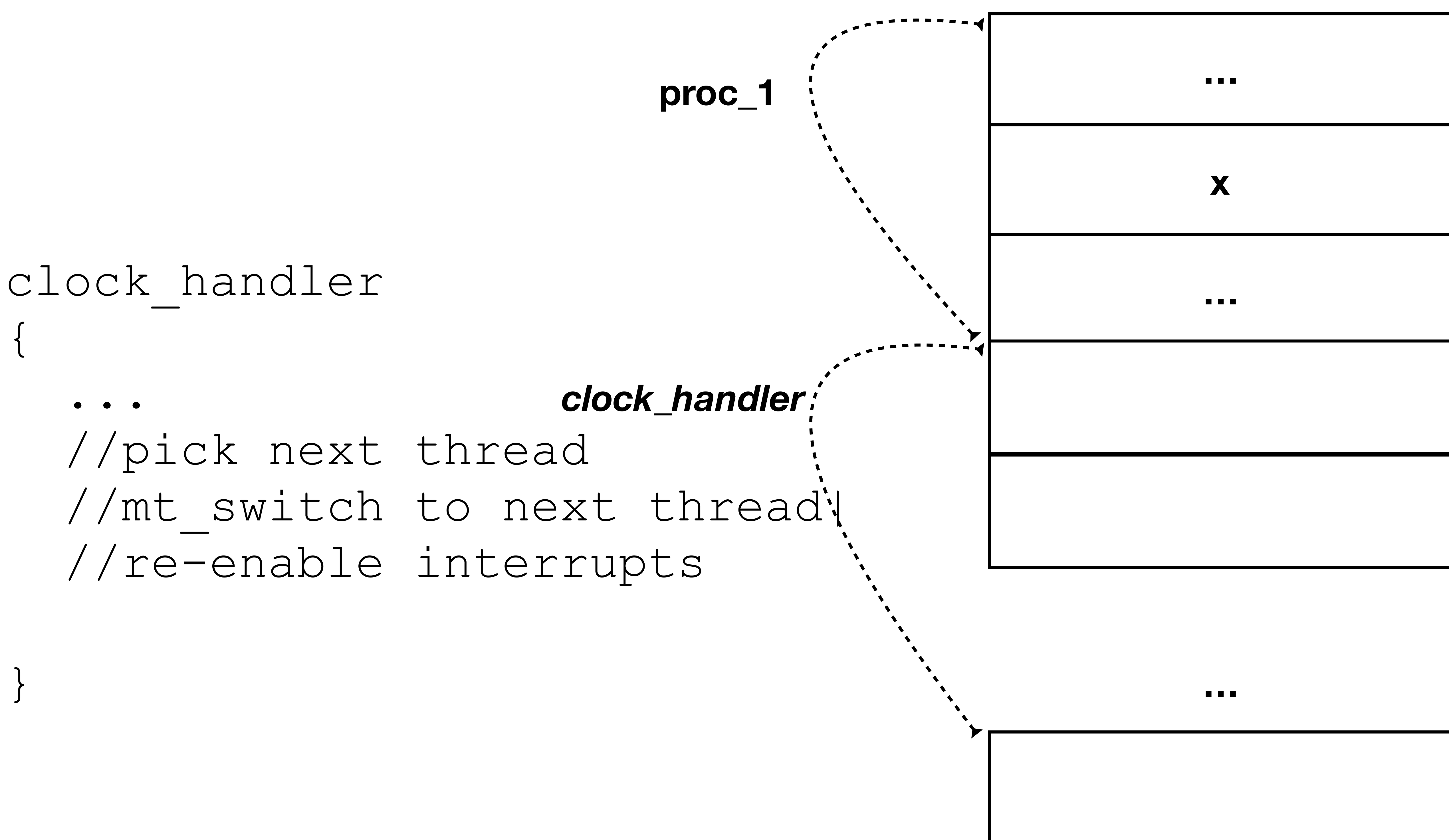
Interrupt Handling



Interrupt Handling



Interrupt Handling



Interrupt Safety

- Critical Section -> some need to be interrupt safe
- Don't forget to re-enable interrupts when done!
- When ISR starts:
  - Interrupts must be disabled
- DON'T block (sema_P) while handling interrupts
- Semaphore updates must be *interrupt-safe*

Semaphore

```
semaphore_P(sema) {
  sema->count--;
  if (count < 0) {
    queue_append(sema->q, minithread_self());
    minithread_stop();
  }
}
```

These lines must happen atomically ->
in Port OS this requires *interrupt safety*

Alarms!

- Description:
- Asynchronous execution
- Execute some function at a future time
- Can ‘cancel’ them
- *Interrupt Safety*

- API:
- `alarm_register(`*`delay`*`,` *`func`*`)`
- *`alarm_deregister(alarm)`*

Alarms!

- Every clock tick
  - Check alarms -> execute any that are due to execute
  - Must run in O(n), *n = number of ready alarms*
  - **NOT** O(*r*), *r = number of registered alarms*
  - (*You may need to modify your queue API)*

Alarms

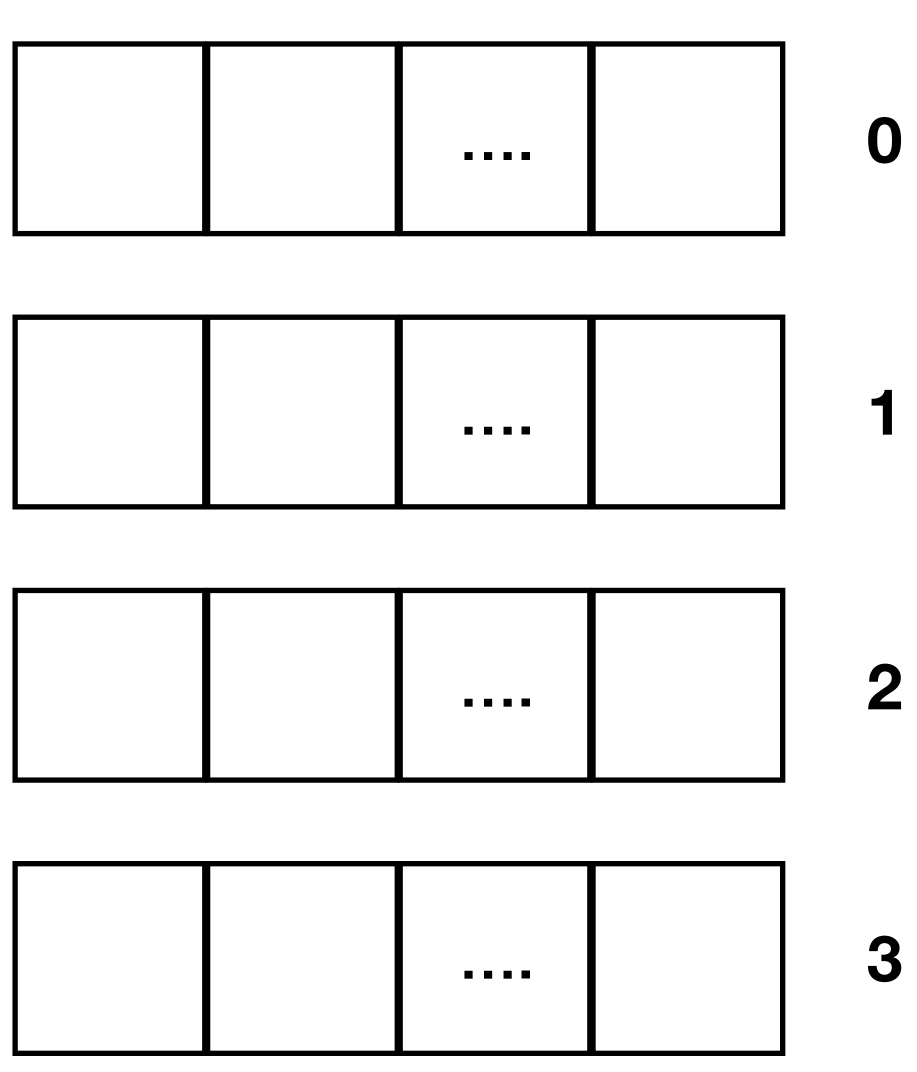
- You’ll implement ‘minithread_sleep_with_timeout’ as an exercise
- Deschedules thread for a fixed amount of time
- Should be a very short bit of code :)

Scheduling Algorithm

- Need a way to pick the next thread to run
- (Do this after everything else works)
- As of P1 - FIFO

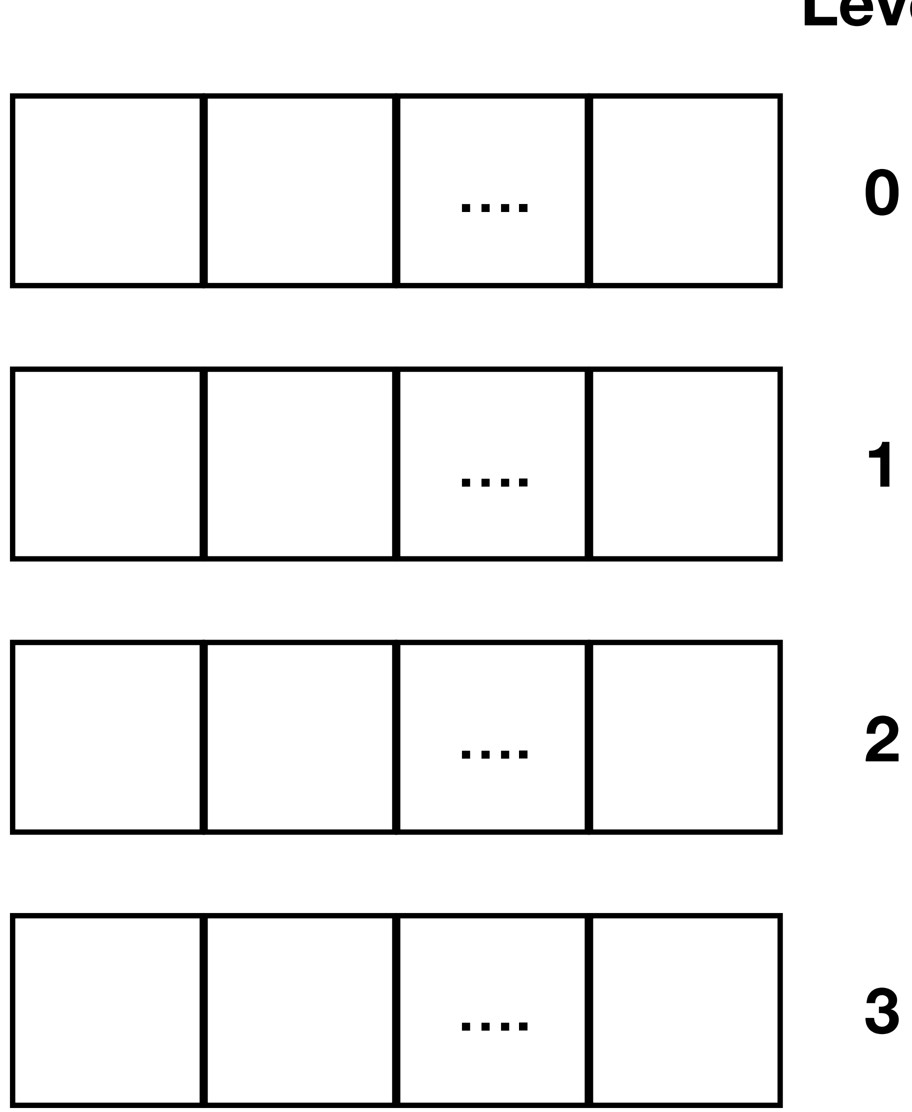
Multilevel Feedback Queue

- High Priority (Low Level Num)
  Quick Tasks -> need low latency
  - Usually I/O heavy
- Low Priority (High Level Num)
  Need more CPU time -> needs more throughput
  - computationally heavy



Multilevel Feedback Queue

- High Priority (Low Level Num)
  - Give more CPU time overall
  - Less CPU time per task
- Low Priority (High Level Num)
  - Less CPU time overall
  - More CPU time per task



Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	| | | .... | |	0
2^1	2.5t	| | | .... | |	1
2^2	1.5t	| | | .... | |	2
2^3	t	| | | .... | |	3

Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	....	0
2^1	2.5t	....	1
2^2	1.5t	....	2
2^3	t	....	3

Start a new Thread → (Level 0 queue)

Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	| | | .... | |	0
	After 1 tick, thread still executing		
2^1	2.5t	| | | .... | |	1
2^2	1.5t	| | | .... | |	2
2^3	t	| | | .... | |	3

Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	| | | .... | |	0
2^1	2.5t	| | | .... | |	1
2^2	1.5t	| | | .... | |	2
2^3	t	| | | .... | |	3

Demote thread to LVL 1

Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	[ ][ ][....][ ]	0
2^1		[ ][ ][....][ ]	1
2^2	1.5t	[ ][ ][....][ ]	2
2^3	t	[ ][ ][....][ ]	3

Pick another thread from LVL 0 to run

Multilevel Feedback Queue

Time Allocated Per Thread	Time Allocated Per Queue		Level
2^0	5t	| | | .... | |	0
2^1		| | | .... | |	1
2^2	1.5t	| | | .... | |	2
2^3	t	| | | .... | |	3

Eventually…
Pick a thread from LVL 1 Instead