EECS 388: Embedded Systems

7. Threads and Multitasking

Heechul Yun

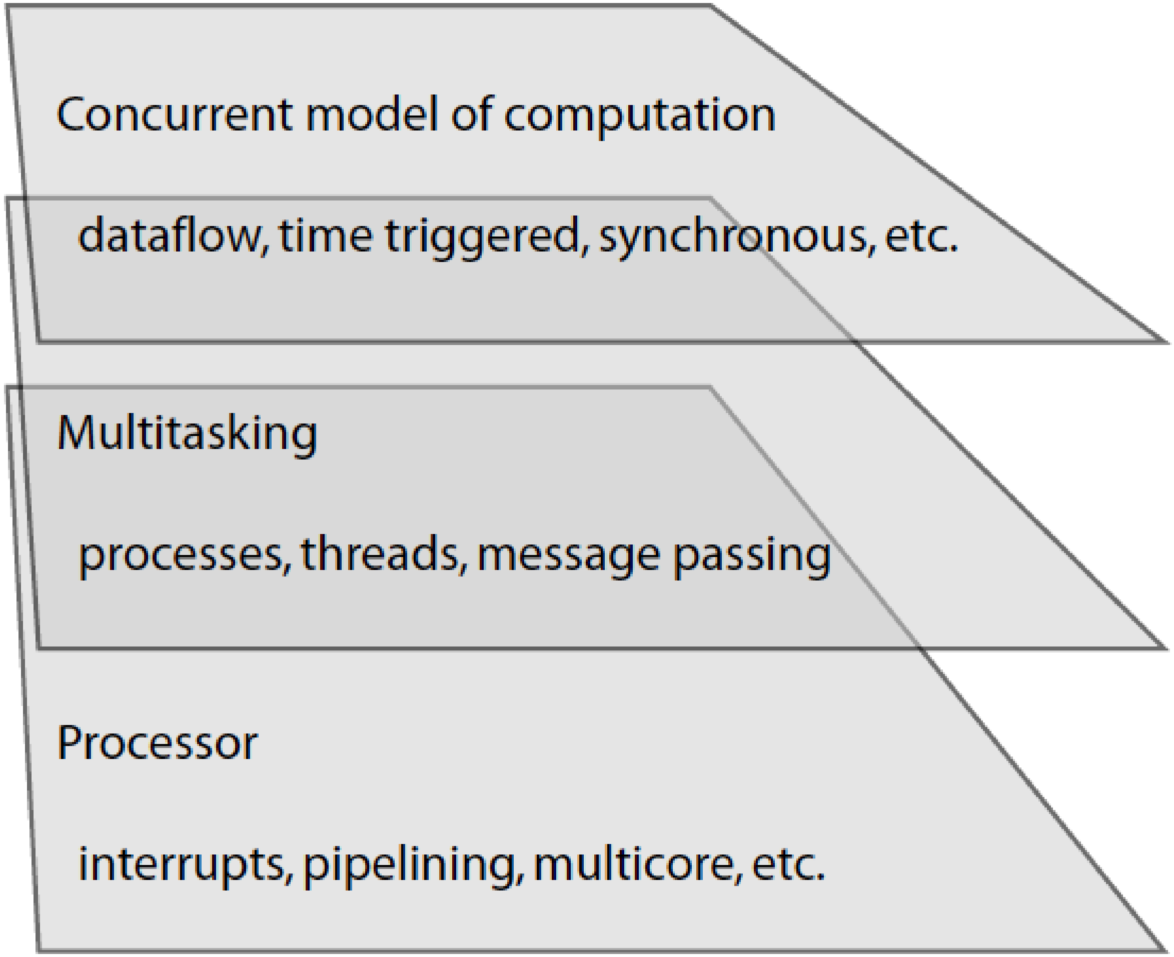
Agenda

- Threads
- Scheduling
- Mutual exclusion
- Problems with threads

Concurrency in Software

- Objects (tanks, planes, …) are moving concurrently and independently
- How to model concurrency in software?

Abstractions for Concurrency



Concurrent model of computation

dataflow, time triggered, synchronous, etc.

Multitasking

processes, threads, message passing

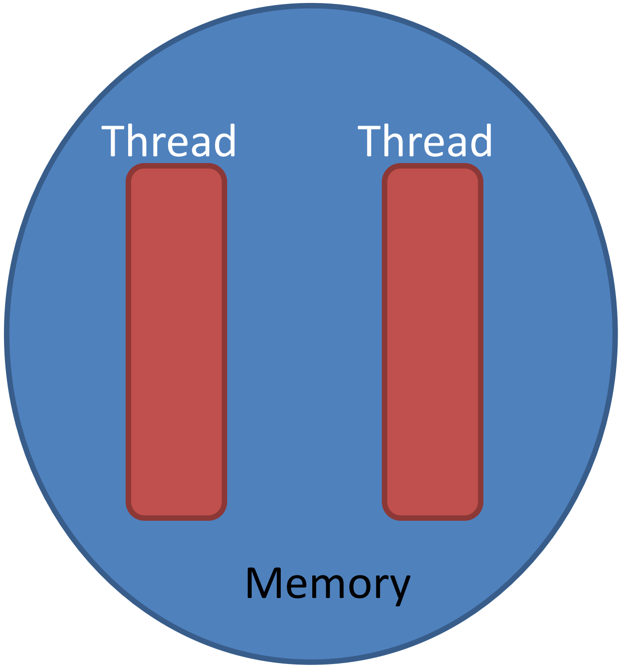
Processor

interrupts, pipelining, multicore, etc.

Thread

Threads in Computing

- Each thread is a sequential code
- Own independent flow of control (execution)
- Each thread has its own stack
- Memory is shared



Pthread

- IEEE POSIX standard threading API

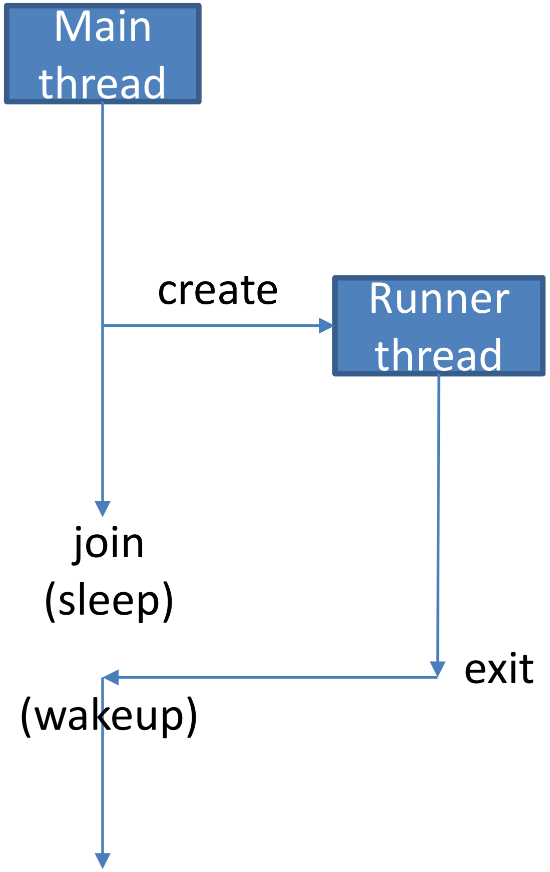
- Pthread API
  - Thread management
    - create, destroy, detach, join, set/query thread attributes
  - Synchronization
    - Mutexes –lock, unlock
    - Condition variables – signal/wait

Pthread API

- pthread_attr_init – initialize the thread attributes object
  - int pthread_attr_init(pthread_attr_t *attr);
  - defines the attributes of the thread created
- pthread_create – create a new thread
  - int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
  - upon success, a new thread id is returned in thread
- pthread_join – wait for thread to exit
  - int pthread_join(pthread_t thread, void **value_ptr);
  - calling process blocks until thread exits
- pthread_exit – terminate the calling thread
  - void pthread_exit(void *value_ptr);
  - make return value available to the joining thread

Pthread Example 1

```
#include <pthread.h>
#include <stdio.h>

int sum; /* data shared by all threads */
void *runner (void  *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for(i=1 ; i<=upper ; i++)
        sum += i;
    pthread_exit(0);
}

int main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    fprintf(stdout, “sum = %d\n”, sum);
}
```



Main thread

create

Runner thread

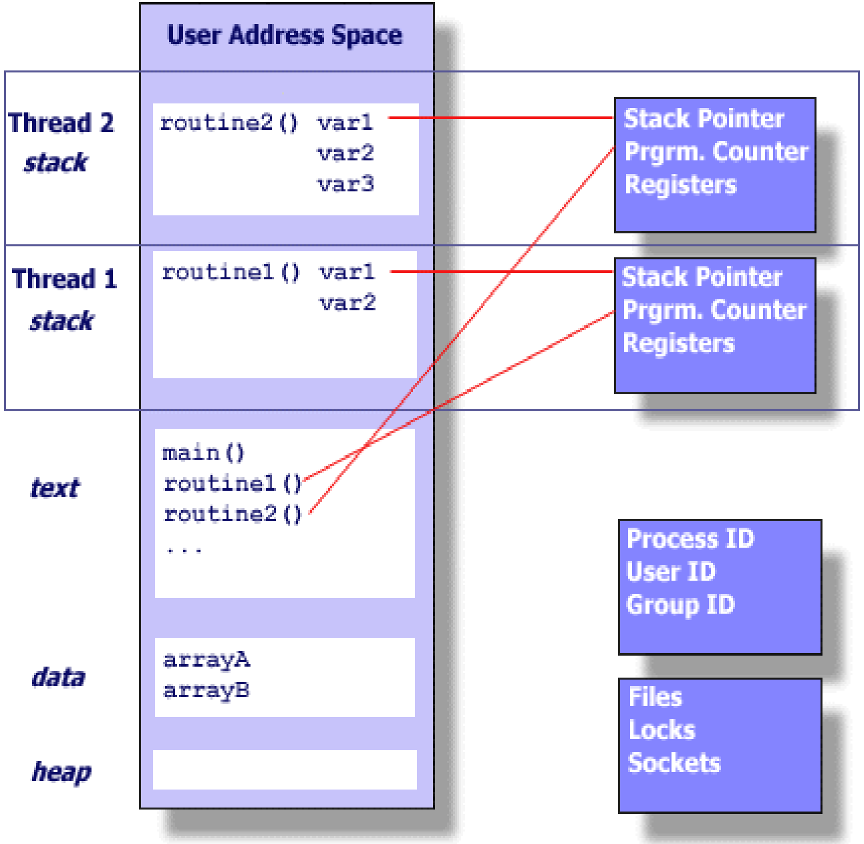
join (sleep)

exit

(wakeup)

Pthread Example 2

```
#include <pthread.h>
#include <stdio.h>

int arrayA[10], arrayB[10];

void *routine1(void  *param)
{
    int var1, var2
    …
}
void *routine2(void  *param)
{
    int var1, var2, var3
    …
}


int main (int argc, char *argv[])
{
    /* create the thread */
    pthread_create(&tid[0], &attr, routine1, NULL);
    pthread_create(&tid[1], &attr, routine2, NULL);
    pthread_join(tid[0]); pthread_join(tid[1]);
}
```



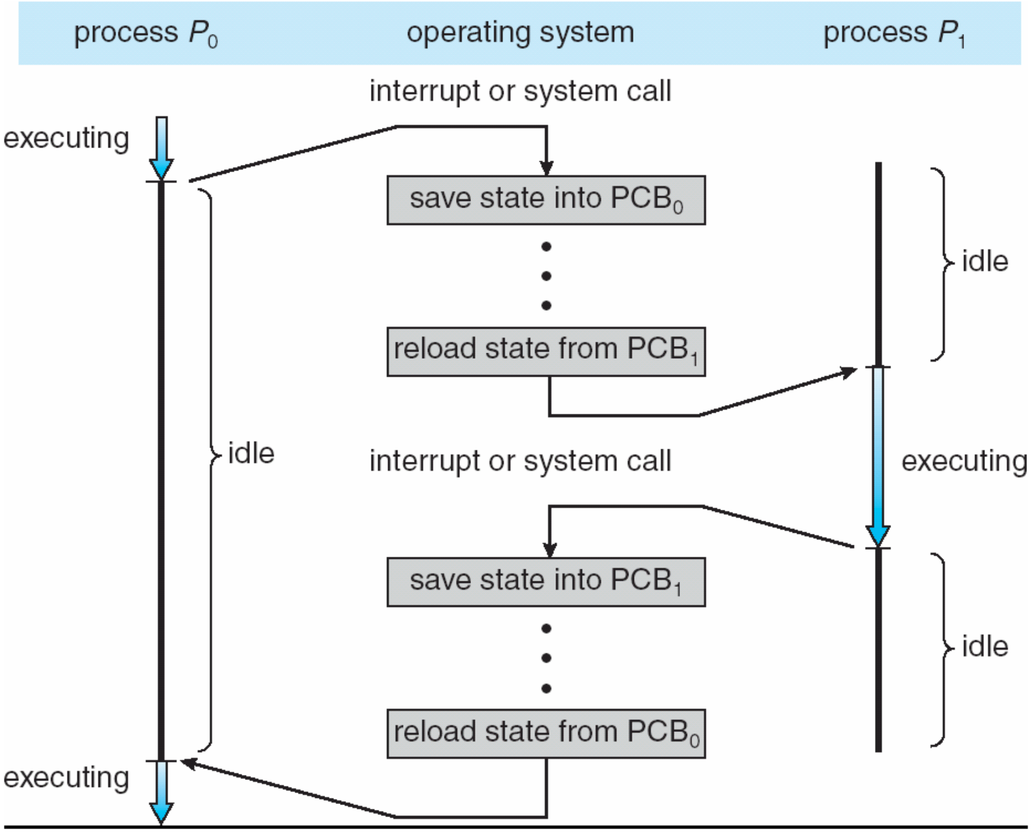
CPU Scheduling

- CPU scheduling is a **policy** to decide
  - **Which** thread to run next?
  - **When** to schedule the next thread?
  - **How long**?

- Context switching is a **mechanism**
  - To change the running thread

Context Switching

- Suspend the current thread and resume a next one from its last suspended state



Context Switching

- Overhead
  - Save and restore CPU states
  - Warm up instruction and data cache
    - Cache data of previous process is not useful for new process

- In Linux 3.6.0 on an Intel Xeon 2.8Ghz
  - About 1.8 us
  - ~ 5040 CPU cycles
  - ~ thousands of instructions

Non-Preemptive Scheduler

- Once a thread is scheduled, it can continue to use the CPU until it finishes or voluntarily relinquishes itself (yield)
- Pros and Cons

  ++ minimal overhead

  --- possible starvation

  --- fairness, response time, …

Preemptive Scheduler

- Each thread is given a certain time slice, after which it is preempted by the scheduler to schedule a next thread.
- A preemptive scheduler is periodically activated at a fixed time interval (“tick”), which is typically implemented as a timer interrupt
- Pros and Cons

++ responsive, fair

--- overhead (context switching is not free)

Race Condition

Initial condition: *counter = 5*

Thread 1

```
R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);
```

Thread 2

```
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
```

- What are the possible outcome?

Race Condition

Initial condition: *counter = 5*

```
R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
```

counter = 5

```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R1);
counter = store (R2);
```

counter = 4

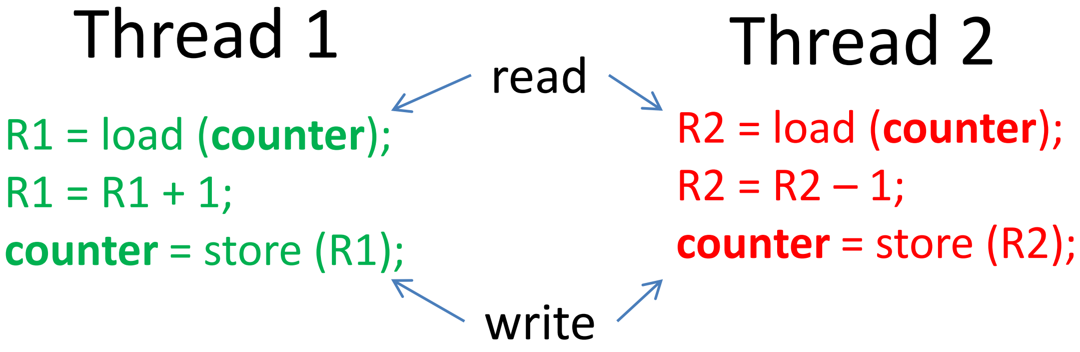
```
R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);
counter = store (R1);
```

counter = 6

- Why this happens?

Race Condition

- A situation when two or more threads **read and write** shared data at the same time
- Correctness depends on the execution order



Thread 1

R1 = load (**counter**);
R1 = R1 + 1;
counter = store (R1);

Thread 2

R2 = load (**counter**);
R2 = R2 – 1;
counter = store (R2);

- How to prevent race conditions?

Critical Section

- Code sections of potential race conditions

Thread 1	Thread 2
Do something	Do something
..	..
R1 = load (**counter**);	R2 = load (**counter**);
R1 = R1 + 1;	R2 = R2 – 1;
counter = store (R1);	**counter** = store (R2);
...	..
Do something	Do something

Critical sections

Critical Section

- Code sections of potential race conditions

Thread 1	Thread 2
Do something	Do something
..	..
R1 = load (**counter**);	R2 = load (**counter**);
R1 = R1 + 1;	R2 = R2 – 1;
counter = store (R1);	**counter** = store (R2);
...	..
Do something	Do something

Critical
sections

Mutual Exclusion

- A property that requires only one thread can enter its critical section at a time among multiple concurrent threads
- Lock (mutex) is a mechanism to provide mutual exclusion

Lock

- General solution
  - Protect critical section via a lock
  - Acquire on enter, release on exit

```
do {
    acquire lock;

        critical section

    release lock;

        remainder section

} while(TRUE);
```

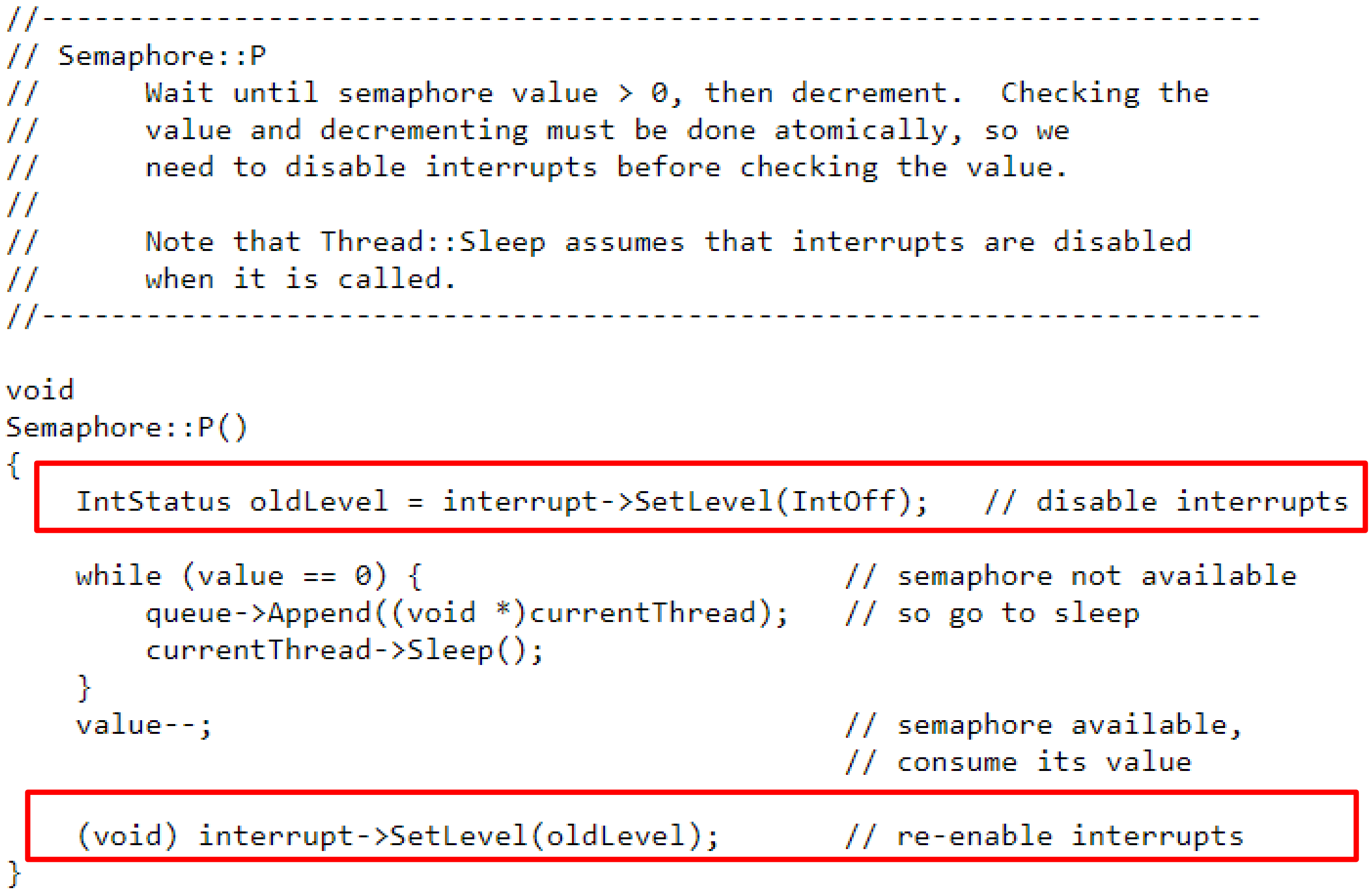
How to Implement a Lock?

- Unicore processor
  - No true concurrency

    one thread at a time
  - Threads are *interrupted* by the OS
    - scheduling events: timer interrupt, device interrupts
- Disabling interrupt
  - Threads can't be interrupted

```
do {

    disable interrupts;
        critical section
    enable interrupts;

        remainder section
} while(TRUE);
```

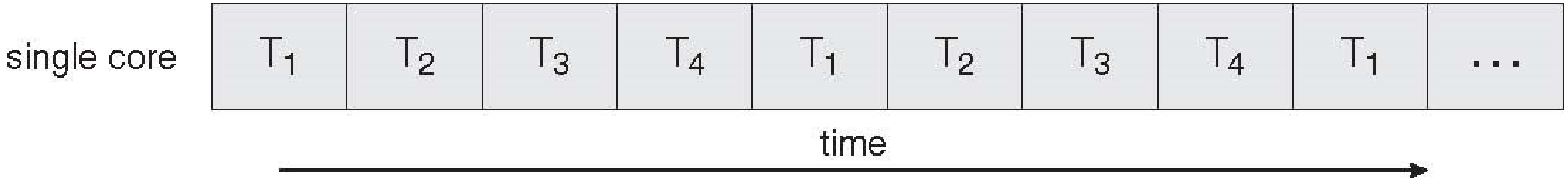
How to implement a Lock?



```
//----------------------------------------------------------------------
// Semaphore::P
//      Wait until semaphore value > 0, then decrement.  Checking the
//      value and decrementing must be done atomically, so we
//      need to disable interrupts before checking the value.
//
//      Note that Thread::Sleep assumes that interrupts are disabled
//      when it is called.
//----------------------------------------------------------------------

void
Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);   // disable interrupts

    while (value == 0) {                        // semaphore not available
        queue->Append((void *)currentThread);   // so go to sleep
        currentThread->Sleep();
    }
    value--;                                    // semaphore available,
                                                // consume its value

    (void) interrupt->SetLevel(oldLevel);       // re-enable interrupts
}
```

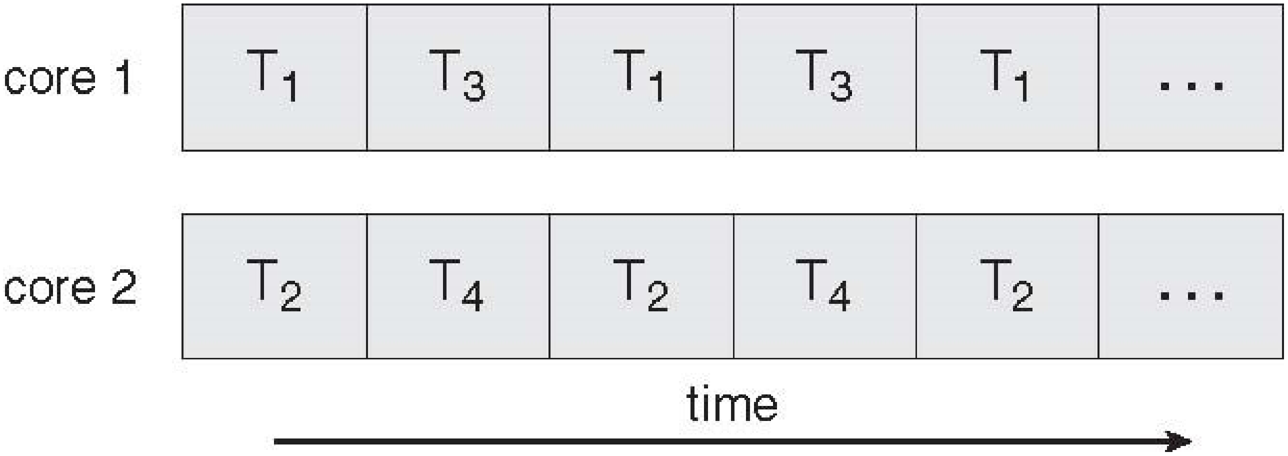
https://people.cs.uchicago.edu/~odonnell/OData/Courses/CS230/NACHOS/code/threads/synch.cc

Single-core vs. Multicore CPU



single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	. . .

Single core execution



core 1	T_1	T_3	T_1	T_3	T_1	. . .
core 2	T_2	T_4	T_2	T_4	T_2	. . .

Multiple core execution

How to Implement a Lock?

- Multicore processor
  - True concurrency
    - More than one active threads sharing memory
  - Disabling interrupts doesn’t solve the problem
    - More than one threads are executing at a time
- Hardware support
  - Synchronization instructions: atomic read and write

- More on EECS678

The Problems of Threads

- Hard to write correct multithread software
- Hard to understand
- Hard to verify

Why Difficult?

- Thread interleaving is non-deterministic
- There are so many possible interleaving
- Hard to test/reproduce/debug

Summary

- Threads
  - An abstraction for sequential program
  - Can model concurrency
  - Share memory
  - Require careful synchronization
- Context switching
  - Suspend/resume execution among multiple threads
- Mutual exclusion
  - To avoid race condition