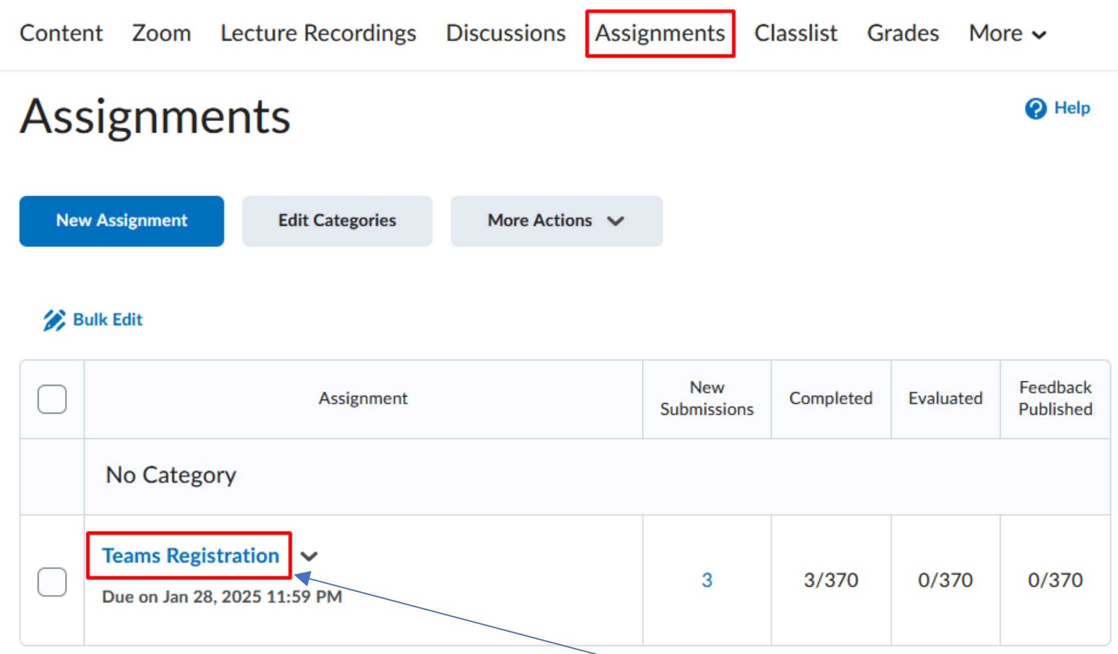
Week 3

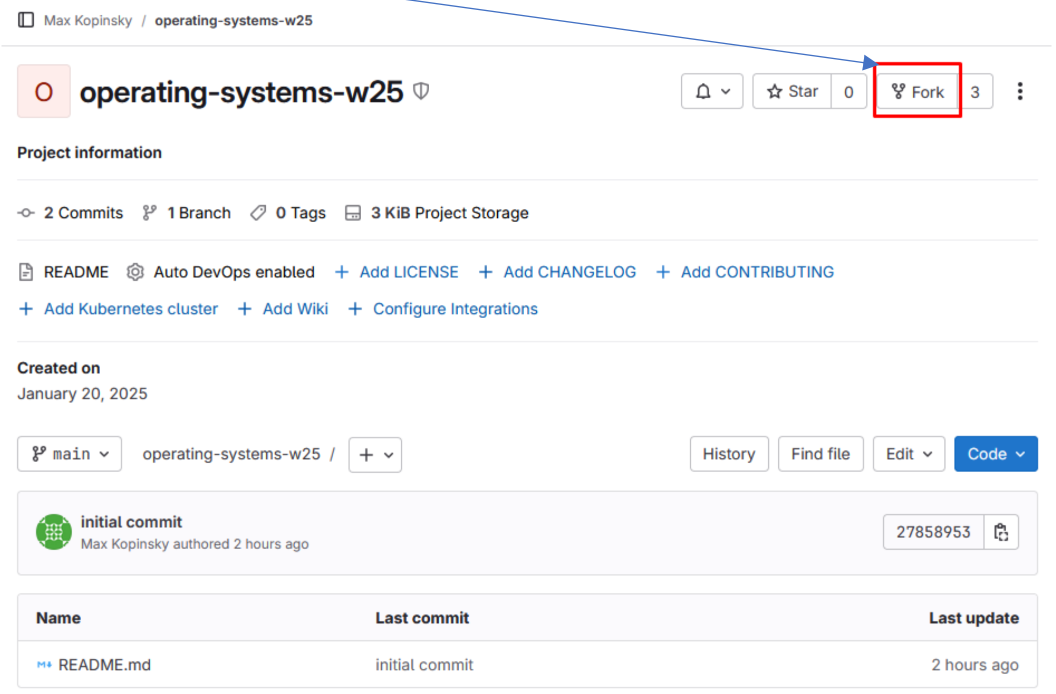
Synchronization Primitives

Max Kopinsky

21 January, 2025

Teams Registration is Open





Teams Registration is Open

- You **must** fork the repository. You won't be able to submit your work to my repository. You can change the name of your fork if you want.

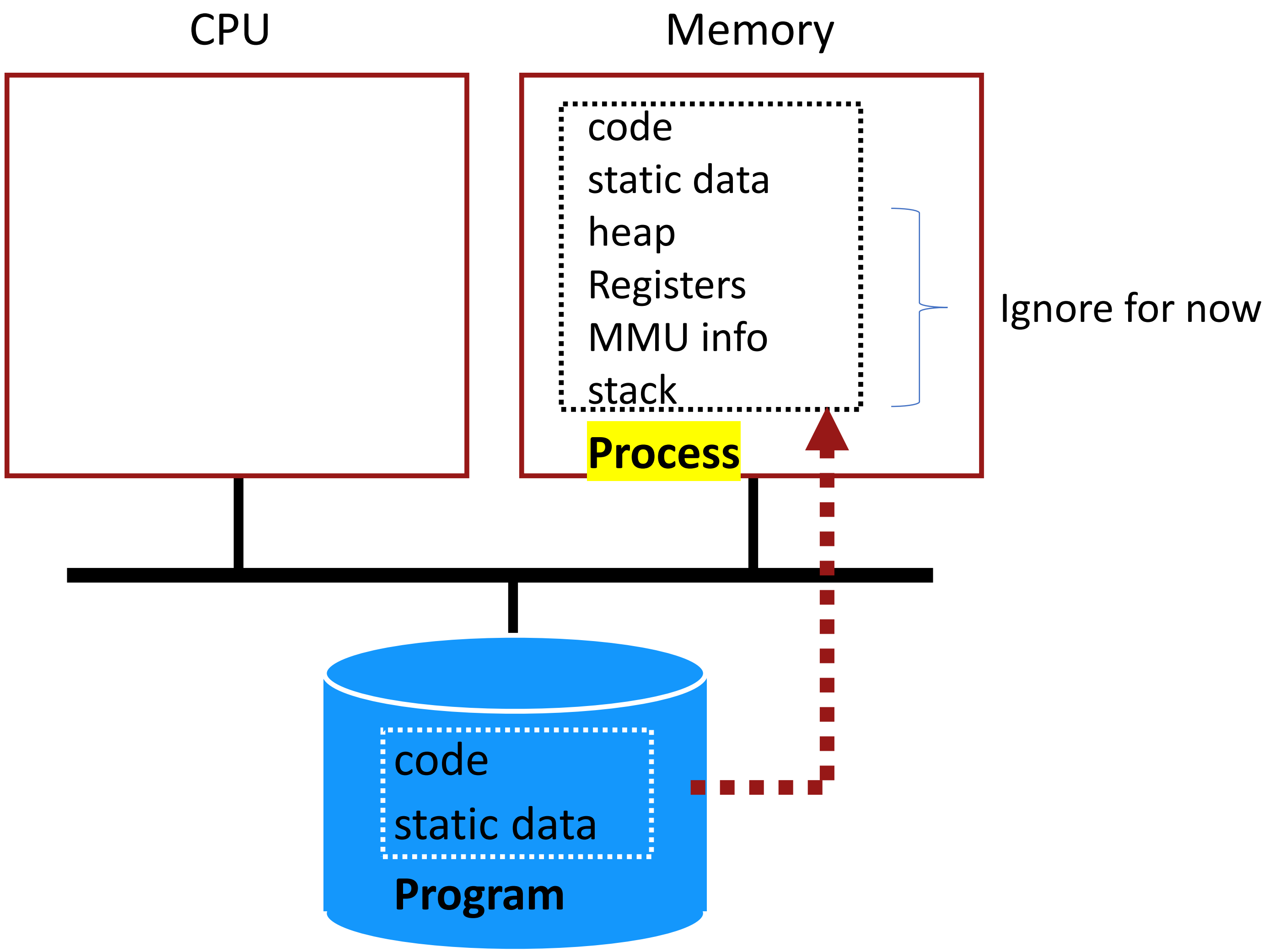
- If you want to work alone, you must submit a team registration with no teammate. Otherwise our autograder doesn't know about you.

- Your registration must be a **plain text** file with some information about your team. **Read the directions carefully.** The submissions are processed by a script. You will consume **a lot** of TA time if we have to track you down to correct errors.

Recap from Week 2

- Process
- Linux process tree
- Process switch
- Process scheduler

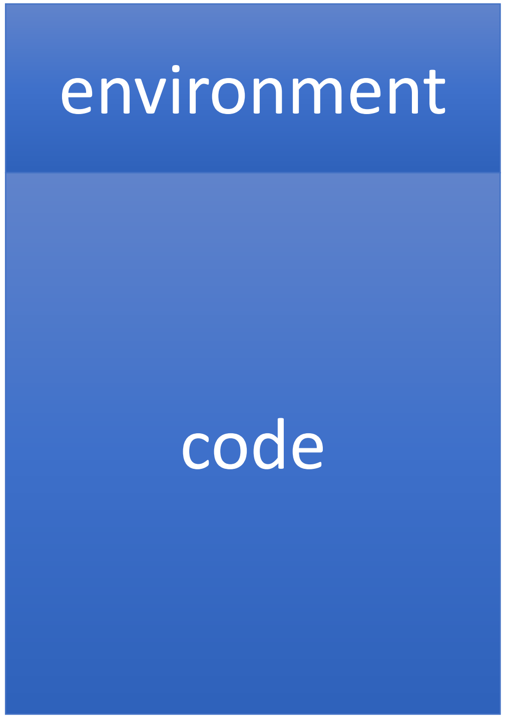
Recap from Week 2
Process = Program in execution



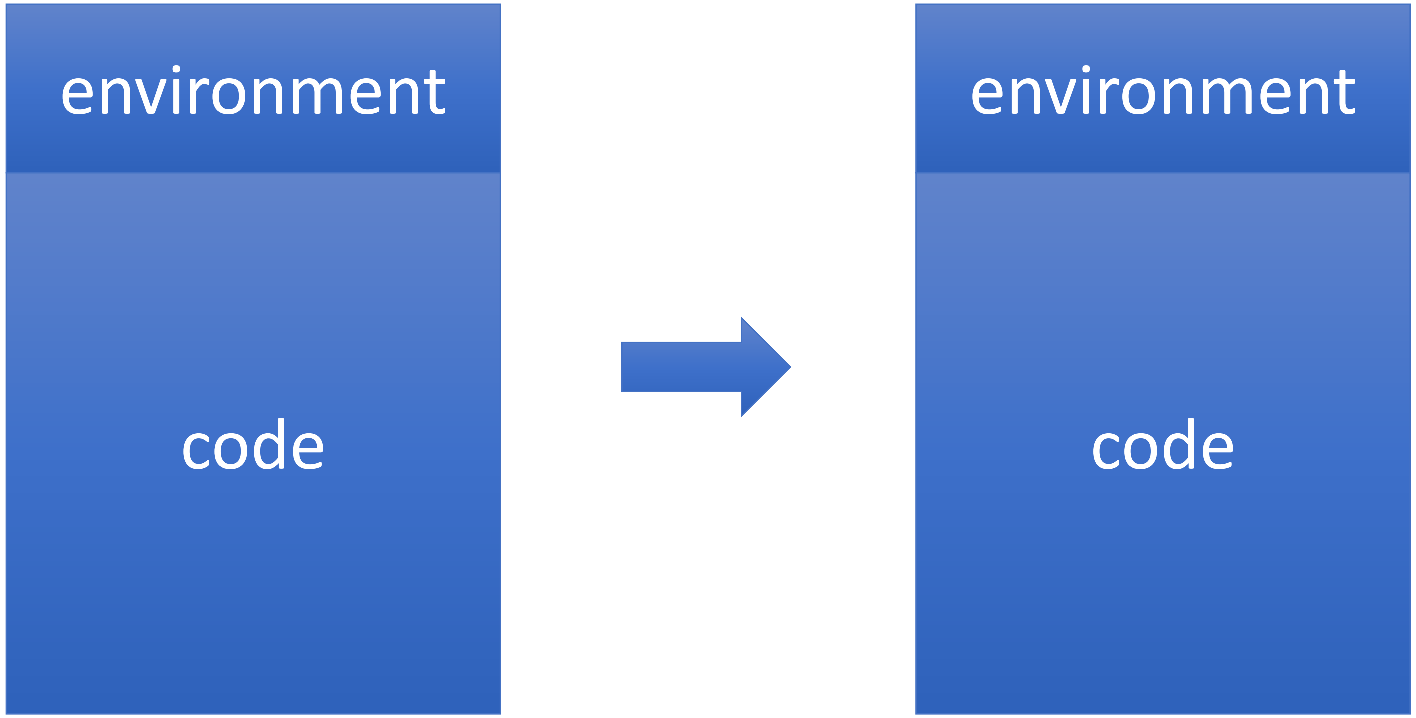
Recap from Week 2
Linux Process Primitives

- pid = fork()
- exec(filename)
- exit()
- wait()

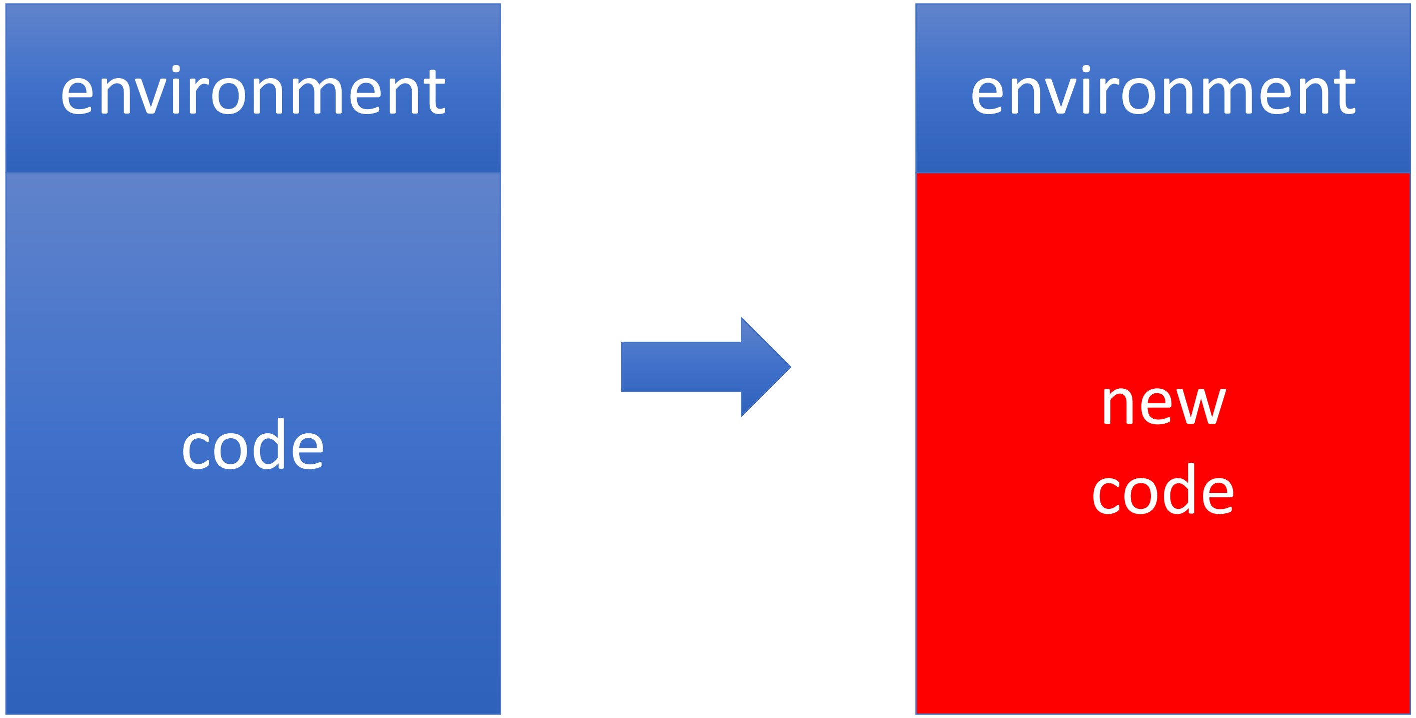
Recap from Week 2
Process = Environment + Code



Recap from Week 2
After a fork()

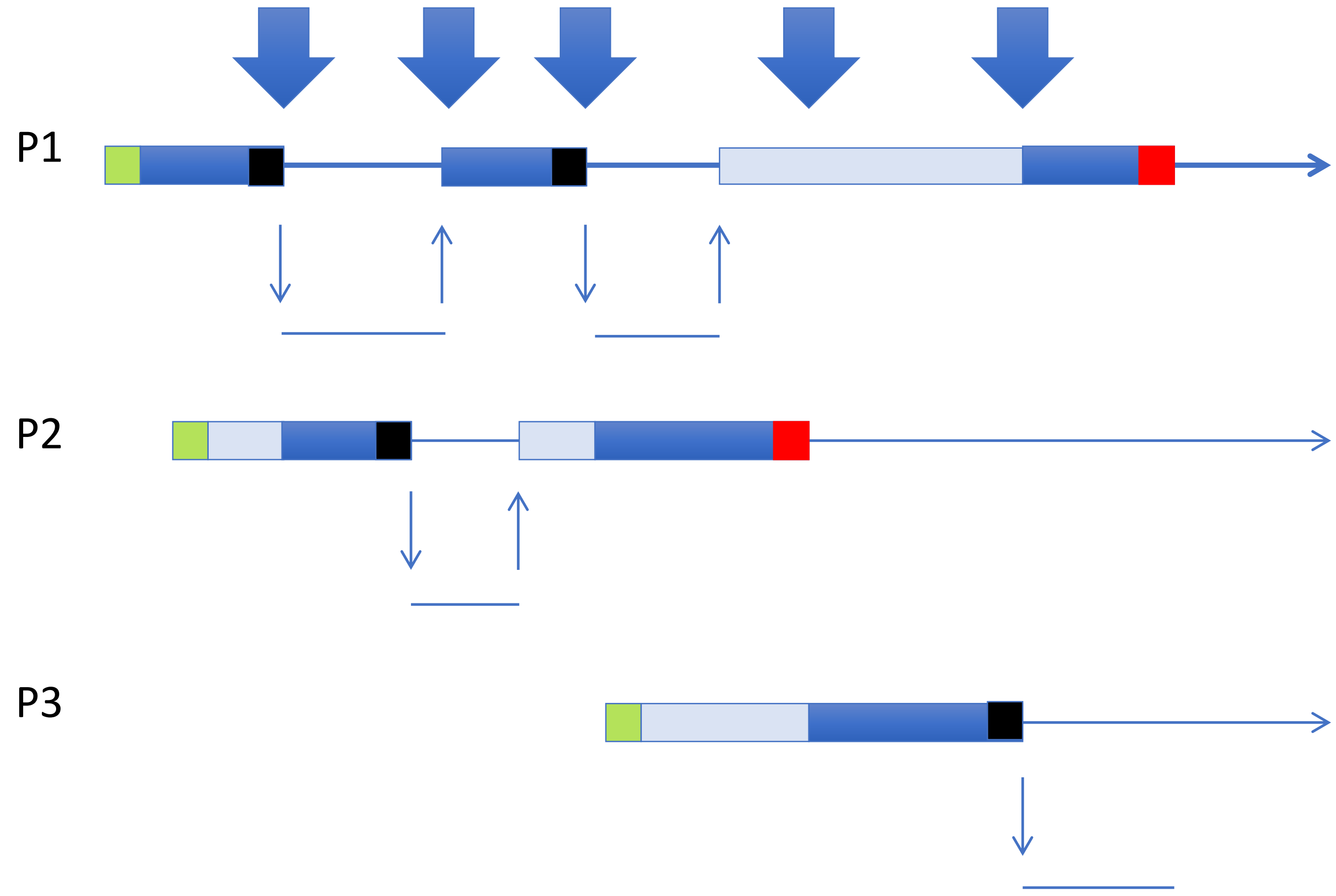


Recap from Week 2
After an exec() in the Child

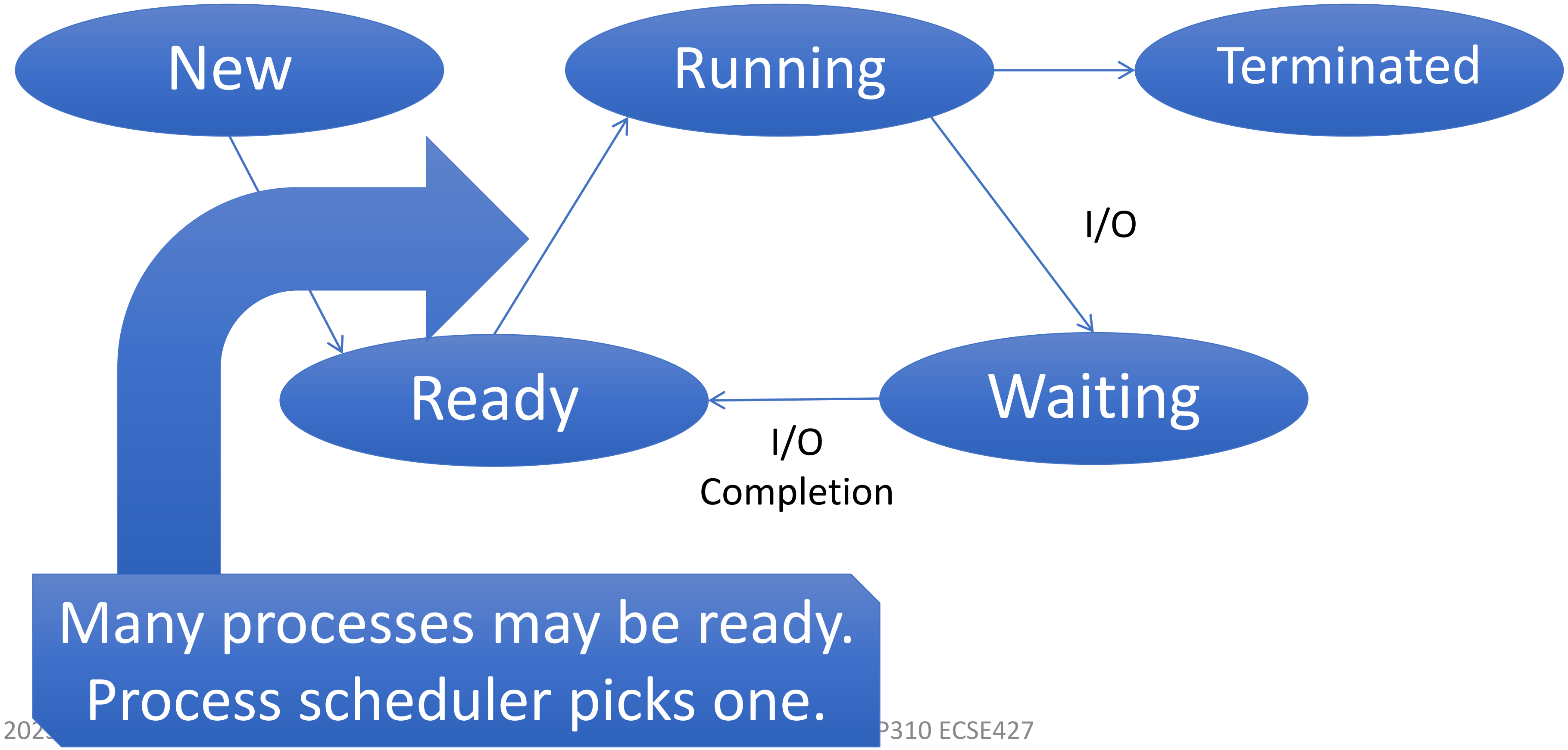


Recap from Week 2
Process Switch

- Change of process using the CPU
- Save and restore registers and other info



Recap from Week 2
Process Scheduling



Questions from last week?

Before we begin with today's topic

- Concurrency is a large sub-field of computer science
- In this course, we get a small taste of it
- If you enjoy this lecture, consider COMP-409 Concurrent programming
    - Highly recommend for a strong systems background

Real world concurrency

- Millions of drivers on highway at once.
- Student does homework while watching Netflix.
- Faculty has lunch while grading papers

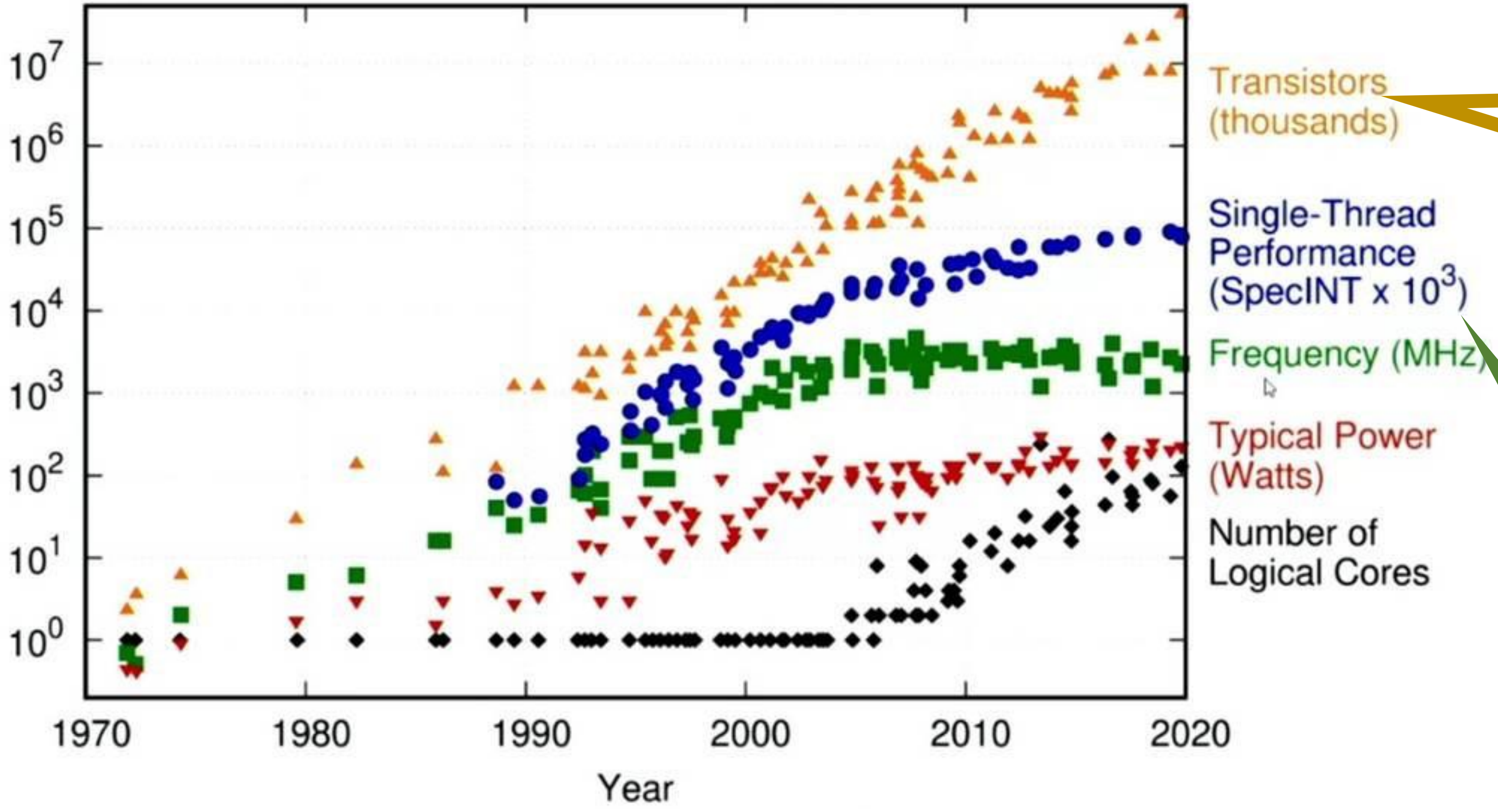
Real world concurrency

- Millions of drivers on highway at once.
- Student does homework while watching Netflix.
- Faculty has lunch while grading papers… and watching Netflix.

Key Concepts for Today

- Process vs Thread
- Mutual Exclusion
- Locking
- Deadlocks
- Conditional variables

Motivation for Concurrency – Moore's Law

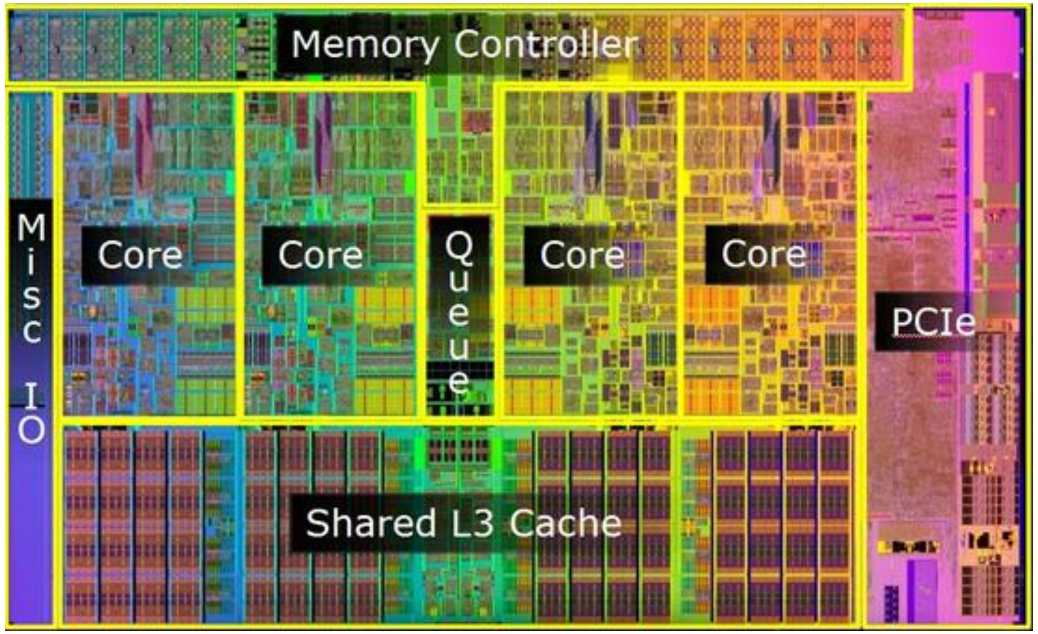


Transistor count still rising

Clock speed flattening sharply

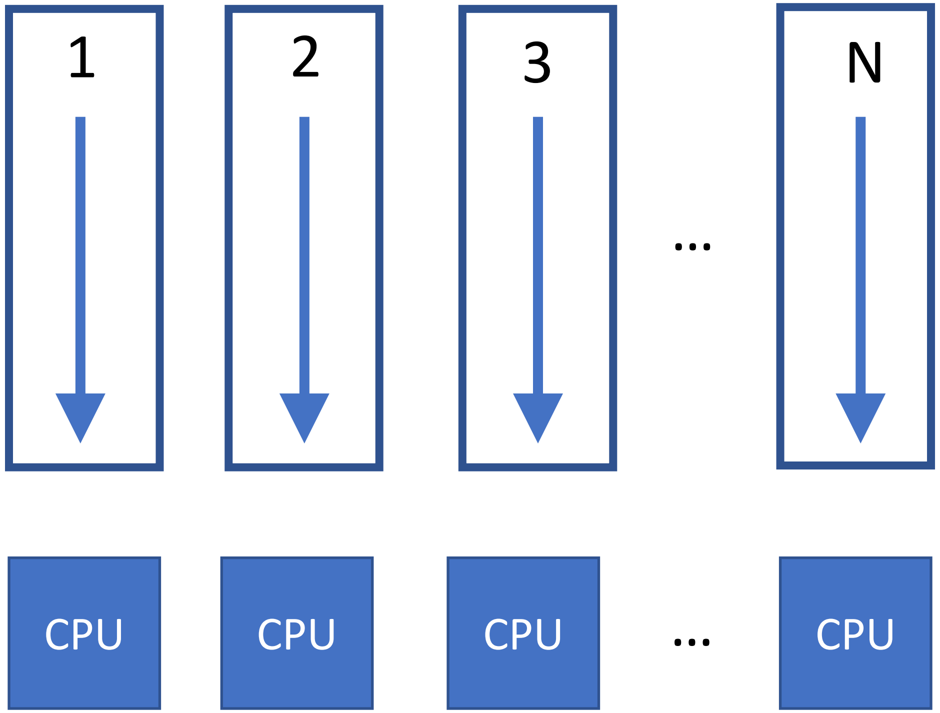
Motivation for Concurrency

- CPU trend: Same speed, but multiple cores
- Goal: write applications that fully utilize many cores



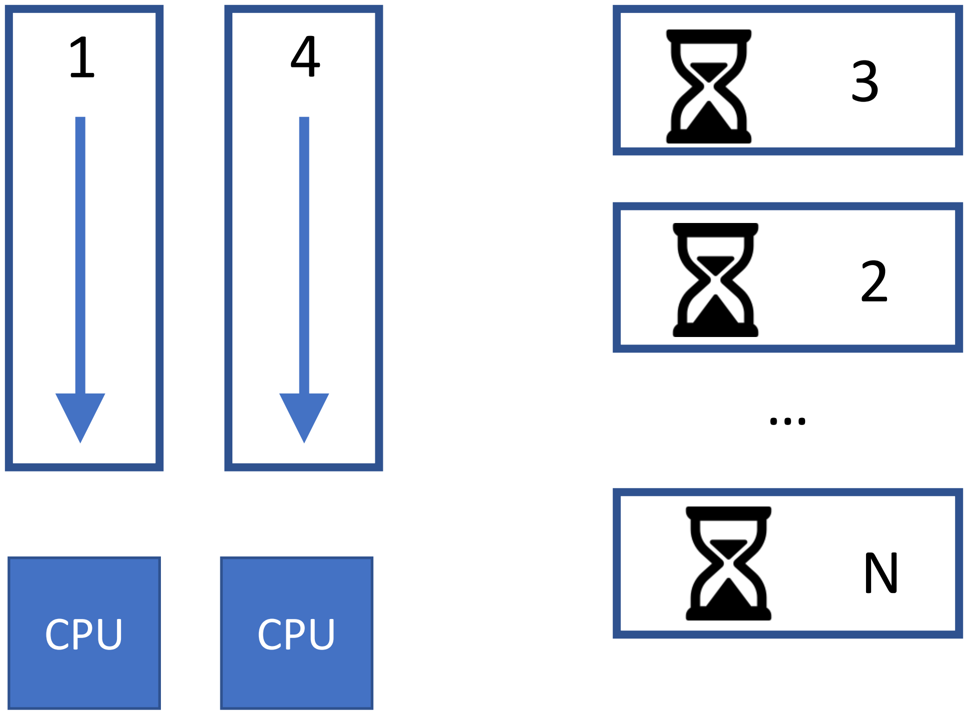
Concurrency Abstraction vs Reality

Programmer Abstraction
(limitless virtualized CPUs)



Physical Reality
(small # of physical CPUs)

Ready queue



Concurrency – Option 1

- Build apps from many communicating **processes**
- Communicate through **message passing**
  - No shared memory
- Pros
  - If one process crashes, other processes unaffected
- Cons
  - High communication overheads
  - Expensive context switching

Concurrency – Option 1

- Build apps from many communicating **processes**
- Communicate through **message passing**
  - No shared memory
- Pros
  - If one process crashes, other processes unaffected
- Cons
  - High communication overheads
  - Expensive context switching

Will see next week

Concurrency – Option 2

- New abstraction: **thread**
- Multiple threads in a process
- Threads are like processes except
  - Multiple threads in the same process share an address space
  - **Communicate through shared address space**
  - If one thread crashes,
    - the entire process, including other threads, crashes

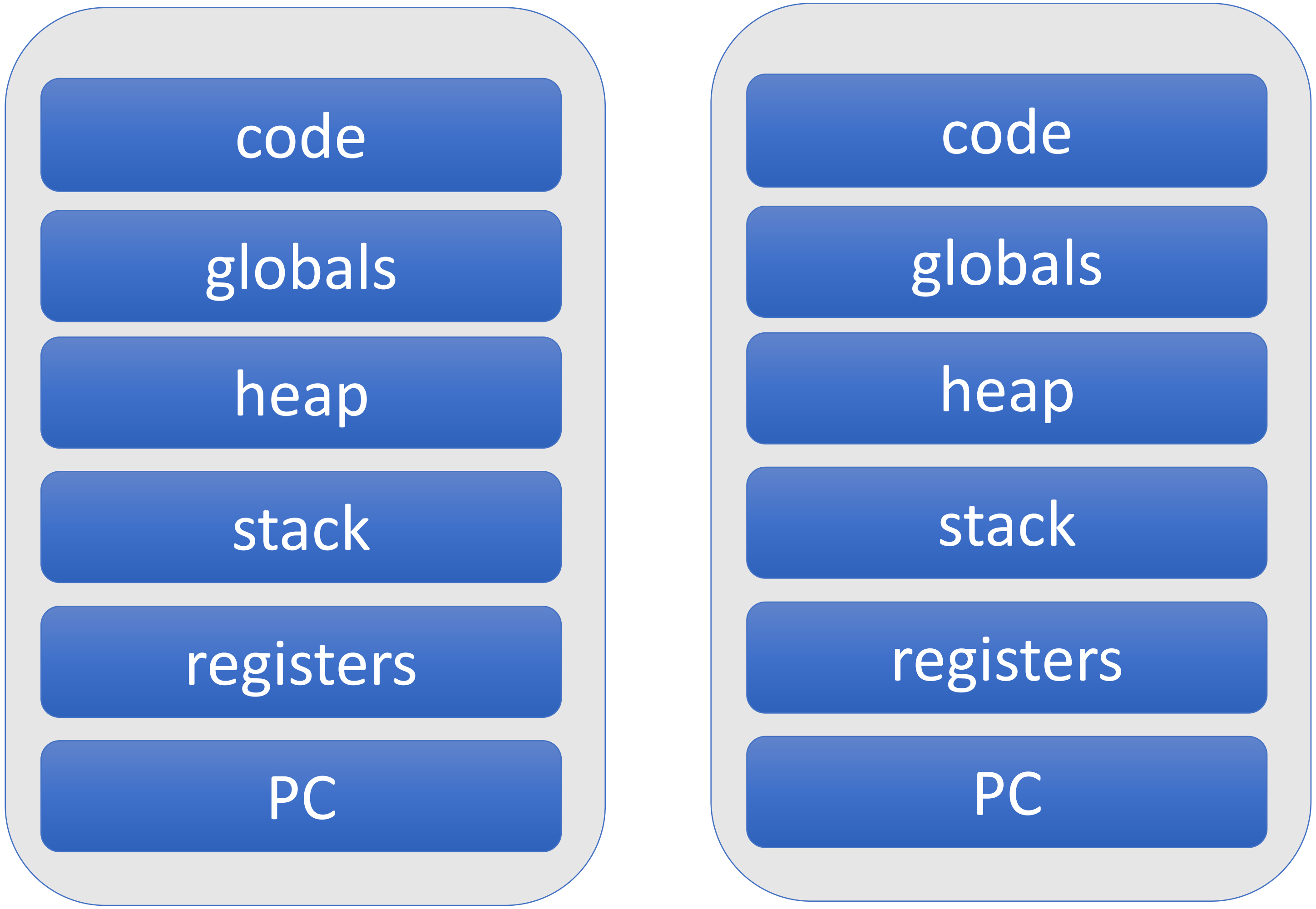
Concurrency – Option 2

Will see synchronization principles **today**

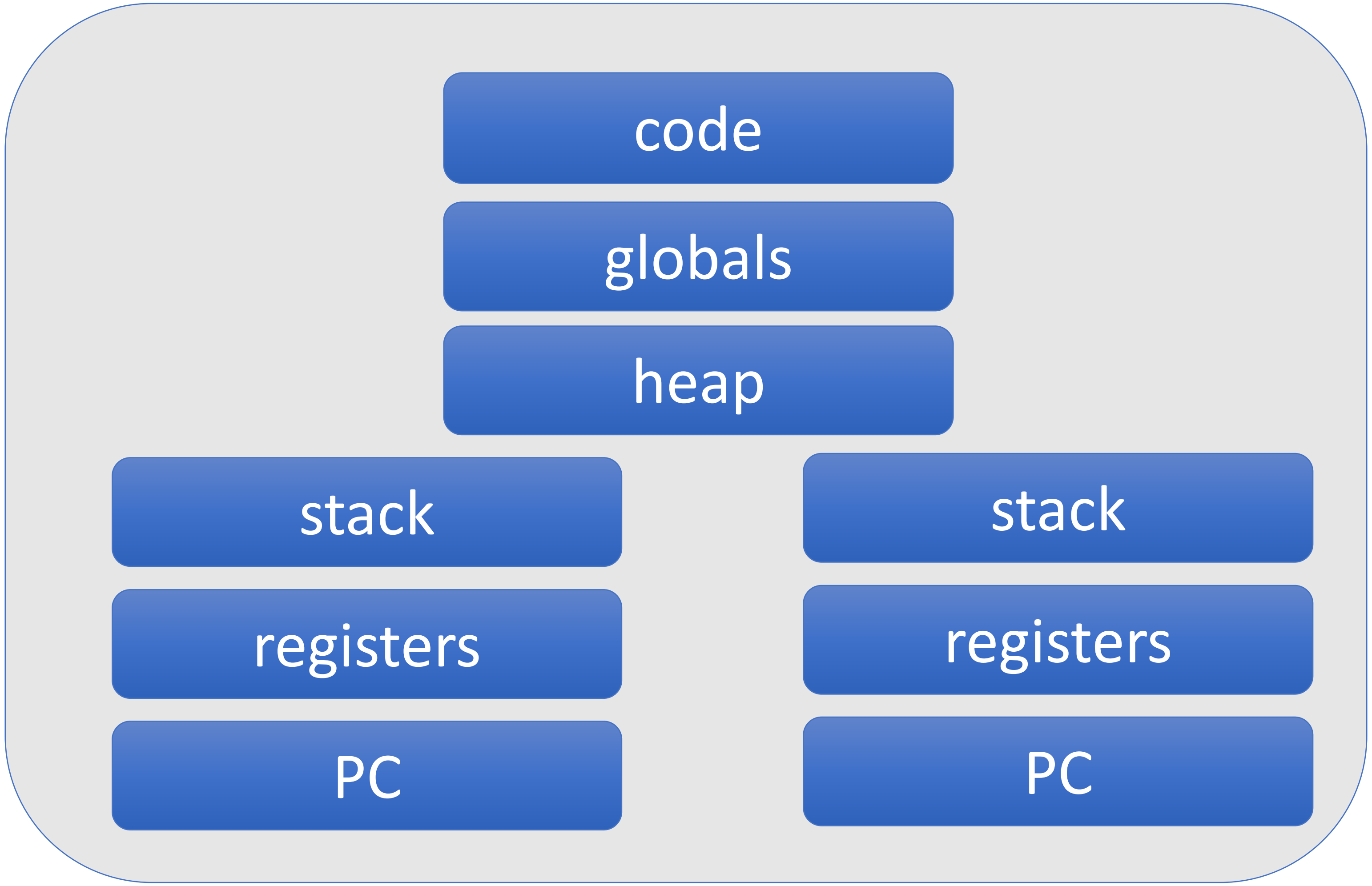
Will see practical examples later

- New abstraction: **thread**
- Multiple threads in a process
- Threads are like processes except
  - Multiple threads in the same process share an address space
  - **Communicate through shared address space**
  - If one thread crashes,
    - the entire process, including other threads, crashes

Two Processes



Two Threads in a Process

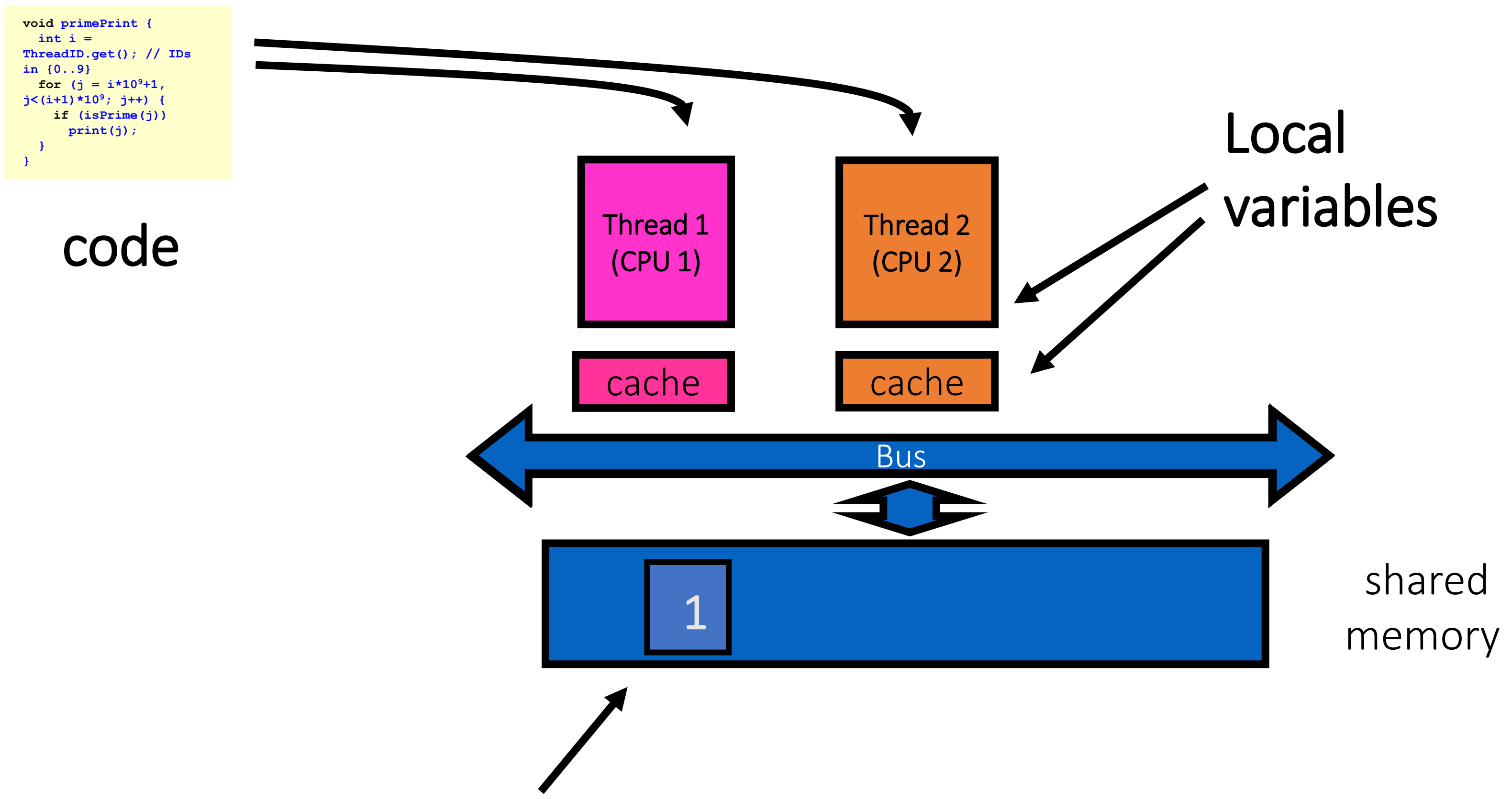


In General

- Processes provide separation
  - In particular, memory separation (no shared data)
  - Suitable for coarse-grain interaction

- Threads do not provide separation
  - In particular, threads share memory (shared data)
  - Suitable for tighter integration

Where Things Reside



shared variable (e.g., shared counter, shared flag)

Shared Data

- Advantage:
  - Many threads can read/write it

- Disadvantage:
  - Many threads can read/write it

Shared Data

- Advantage:
  - Many threads can read/write it

- Disadvantage:
  - Many threads can read/write it
  - Can lead to *data races*

Data Race

- Unexpected/unwanted access to shared data
- Result of *interleaving* of thread executions
- **Program must be correct for all interleavings**
  - (another COMP 525 topic)

A Common Mistake/Misunderstanding: A Single Line of Code is not Atomic

- a = a + 1
- Is in reality
  - Load a from memory into register
  - Increment register
  - Store register value in memory
- Instruction sequence may be interleaved
- (Some machines have atomic increments)

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

	Thread 1	Thread 2
%eax	?	?
%rip	0x195	0x195

T1 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- T1 → 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T1 →

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T1

Thread context switch!

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

state of T1 saved in T1 pcb upon context switch

state of T2 loaded in CPU

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

note that code region is common to T1 & T2

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x19a

process control blocks:

	Thread 1	Thread 2
%eax	101	?
%rip	0x1a2	0x195

T2 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 102
%rip = 0x19d

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- T2 → 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; // balance in shared memory at 0x9cd4

Desired result ☺

State:
0x9cd4: 102
%eax: 102
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T2 →

Another schedule

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

	Thread 1	Thread 2
%eax	?	?
%rip	0x195	0x195

T1 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

T1 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:

Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- T1 → 0x19d mov %eax, 0x9cd4

Thread context switch!

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

	Thread 1	Thread 2
%eax	101	?
%rip	0x19d	0x195

T2 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

process control blocks:

	Thread 1	Thread 2
%eax	101	?
%rip	0x19d	0x195

T2 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

T2 →

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T2 →

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T2

Thread context switch!

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x19d

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- T1 → 0x19d mov %eax, 0x9cd4

Thread Schedule #2

balance = balance + 1; // balance in shared memory at 0x9cd4

WRONG Result! ☹
Final value of balance is 101

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

- 0x195 mov 0x9cd4, %eax
- 0x19a add $0x1, %eax
- 0x19d mov %eax, 0x9cd4

T1 →

Non-Determinism

Concurrency leads to non-deterministic results

- Different results even with same inputs
- Race conditions

Whether bug manifests or not depends on CPU schedule!

How to program?

- **Assume scheduler is malicious**
- Assume scheduler will pick bad ordering at some point…

Basic Approach to Multithreading

1. Divide "work" among multiple threads &
2. Share data
    - Which data is shared?
        - **Global variables and heap**
        - Not local variables, not read-only variables
    - Where is shared data accessed?
        - Put shared data access in **critical section**

Critical Section

- Want 3 instructions to execute as an uninterruptable group
- We say we want them to be **“atomic"**

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

Need **mutual exclusion** for critical sections

- If thread A is in critical section C, thread B can’t enter C
- Ok if other processes/threads do unrelated work

Aside: Non-Atomic Single Instructions

- We just implemented `x = x + 1` with:

```
mov 0x123, %eax
add $0x1, %eax
mov %eax, 0x123
```

- But x86 has this instruction:

```
add $0x1, 0x123
```

Aside: Non-Atomic Single Instructions

- We just implemented `x = x + 1` with:

```
mov 0x123, %eax
add $0x1, %eax
mov %eax, 0x123
```

- But x86 has this instruction:

```
add $0x1, 0x123
```

- No difference! This single instruction is not atomic.
  - (The similar `lock add` instruction is atomic.)

Mutual Exclusion

- Prevents simultaneous access to a shared resource.
  - In this case, shared resource = shared memory region

- How can we achieve mutual exclusion?
  - Today we will first see library support (pthreads)
  - Then, we will see implementation of synchronization primitives.

Why this (mostly) works

- Critical section:
  - No other thread can change data
- So you are (mostly) ok

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software	**Monitors** **Locks (mutex)** **Semaphores** **Condition Variables**
Hardware	Loads Stores Test&Set Disable Interrupts

POSIX Thread Libraries (pthreads)

- Thread API for C/C++
- User-level library:
  - **#include <pthread.h>**
  - Compile and link with *-pthread*.
- Support for thread creation, termination, synchronization.
- See more details here: https://man7.org/linux/man-pages/man3/

Pthreads: Thread Creation and Destruction

- `pthread_create()`
- `pthread_exit()`
- `pthread_join()`

pthread_create()

```
int pthread_create(
  pthread_t * thread, pthread_attr_t * attr,
  void *(*start_routine)(void *), void * arg
);
```

- Create thread, in *thread.*
- Run *start_routine* with arguments *arg.*
- *attr* points to a *pthread_attr_t* structure. If *attr* is NULL, then the thread is created with default attributes (ok in most cases).
- On success, return 0; on error, return an error number.

`pthread_exit()`

void pthread_exit(void **retval*);

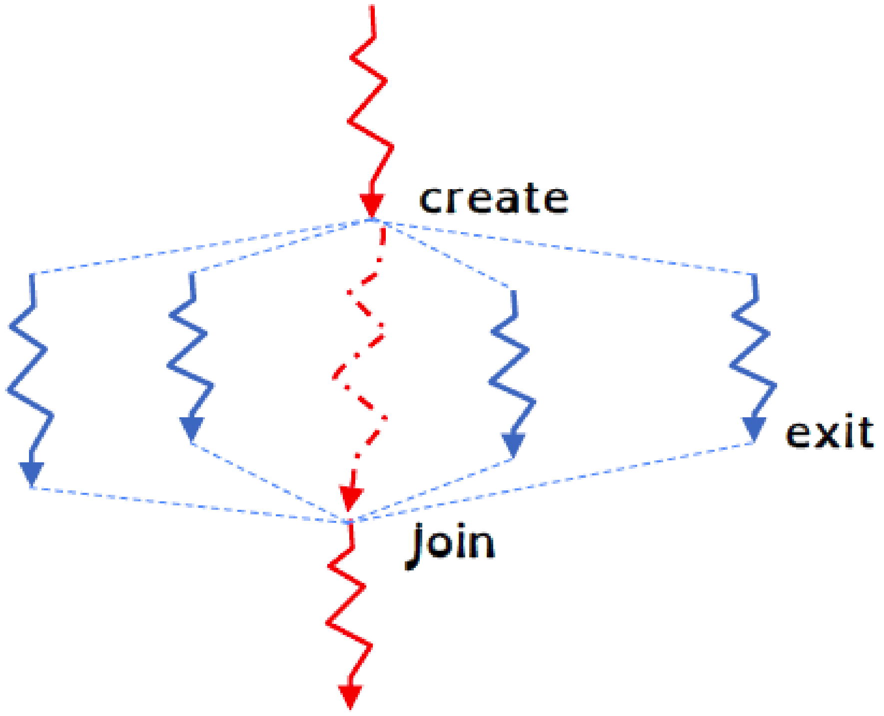
- Terminate calling thread.
- Returns a value via *retval.*

pthread_join()

int pthread_join(pthread_t *thread*, void ***retval*);

- Join with a terminated thread.
- Waits for the thread specified by *thread* to exit.
- If *retval* is not NULL, then **pthread_join**() copies the return value of the target thread into the location pointed to by *retval*.
- On success, return 0; on error, return an error number.

Fork-Join Pattern for threads



Main thread creates (forks) collection of sub-threads passing them args to work on…
… and then joins with them, collecting the results

Note: In this example, start_routine is the same for all threads; only args are different.

Fork-join example

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  printf("main: begin\n");
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");
  // join waits for the threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: end\n");
}
```

Counting example – What is the final answer?

```
int count;
void *mythread(void *arg) {
  int j;
  for (j = 0; j < 1000000; j++){
    count +=1;
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  count = 0;
  pthread_create(&p1, NULL, mythread, NULL);
  pthread_create(&p2, NULL, mythread, NULL);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("%d \n", count);
}
```

Pthreads: Locks

- Pthread_mutex_lock(mutex)
- Pthread_mutex_unlock(mutex)

Pthread_mutex_lock(mutex)

- If lock is held by another thread, block
- If lock is not held by another thread
  - Acquire lock
  - Proceed

Pthread_mutex_unlock(mutex)

- Release lock

Counting example revisited – What is the final answer?

```
pthread_mutex_t count_mutex;
int count;

void *mythread(void *arg) {
  int j;
  for (j = 0; j < 1000000; j++){
     pthread_mutex_lock(&count_mutex);
     count +=1;
     pthread_mutex_unlock(&count_mutex);
  }
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  pthread_mutex_init(&count_mutex, NULL);

  count = 0;
  pthread_create(&p1, NULL, mythread, NULL);
  pthread_create(&p2, NULL, mythread, NULL);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("%d \n", count);
}
```

Deadlocks

- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help.
- Classic example:

Thread A
```
1 lock(object1)
2 lock(object2)
3 //do stuff
4 unlock(object2)
5 unlock(object1)
  …
```

Thread B
```
1 lock(object2)
2 lock(object1)
3 //do stuff
4 unlock(object1)
5 unlock(object2)
…
```

Object 1

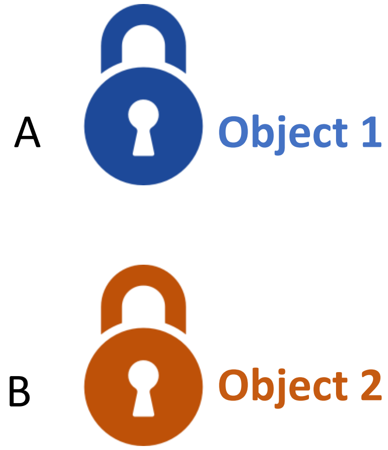
Object 2

Deadlocks

- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help.
- Classic example:

```
Thread A
1 lock(object1)
2 lock(object2)
3 //do stuff
4 unlock(object2)
5 unlock(object1)
  …
```

```
Thread B
1 lock(object2)
2 lock(object1)
3 //do stuff
4 unlock(object1)
5 unlock(object2)
…
```

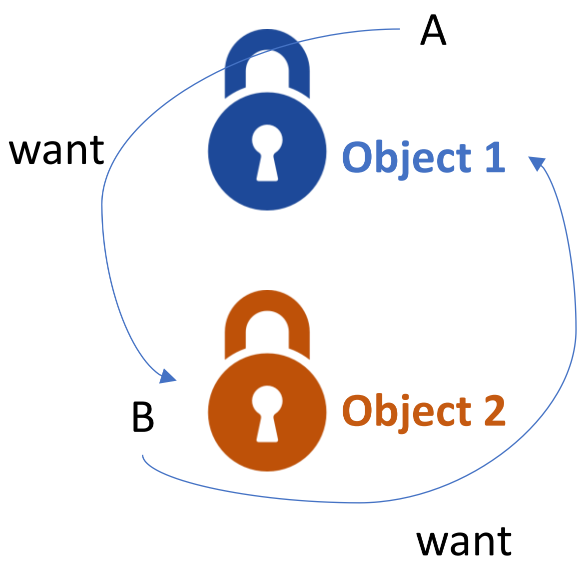


Deadlocks

- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help.
- Classic example:

Thread A
```
lock(object1)
lock(object2)
//do stuff
unlock(object2)
unlock(object1)
 …
```

Thread B
```
lock(object2)
lock(object1)
//do stuff
unlock(object1)
unlock(object2)
…
```



Deadlock example

```c
pthread_mutex_t lock1;
pthread_mutex_t lock2;


void *a_func(void *arg) {
 long j;
 for (j = 0; j < 100000000; j++) {
   pthread_mutex_lock(&lock1);
   pthread_mutex_lock(&lock2);
   printf("A");
   pthread_mutex_unlock(&lock2);
   pthread_mutex_unlock(&lock1);
 }
 return NULL;
}
```

```c
void *b_func(void *arg) {
 long j;
 for (j = 0; j < 100000000; j++) {
   pthread_mutex_lock(&lock2);
   pthread_mutex_lock(&lock1);
   printf("B");
   pthread_mutex_unlock(&lock1);
   pthread_mutex_unlock(&lock2);
 }
 return NULL;
}
```

```c
int main(int argc, char *argv[]) {
 pthread_t a, b;
 pthread_mutex_init(&lock1, NULL);
 pthread_mutex_init(&lock2, NULL);
 pthread_create(&a, NULL, a_func, NULL);
 pthread_create(&b, NULL, b_func, NULL);
 pthread_join(a, NULL);
 pthread_join(b, NULL);
 printf("End!\n");
}
```

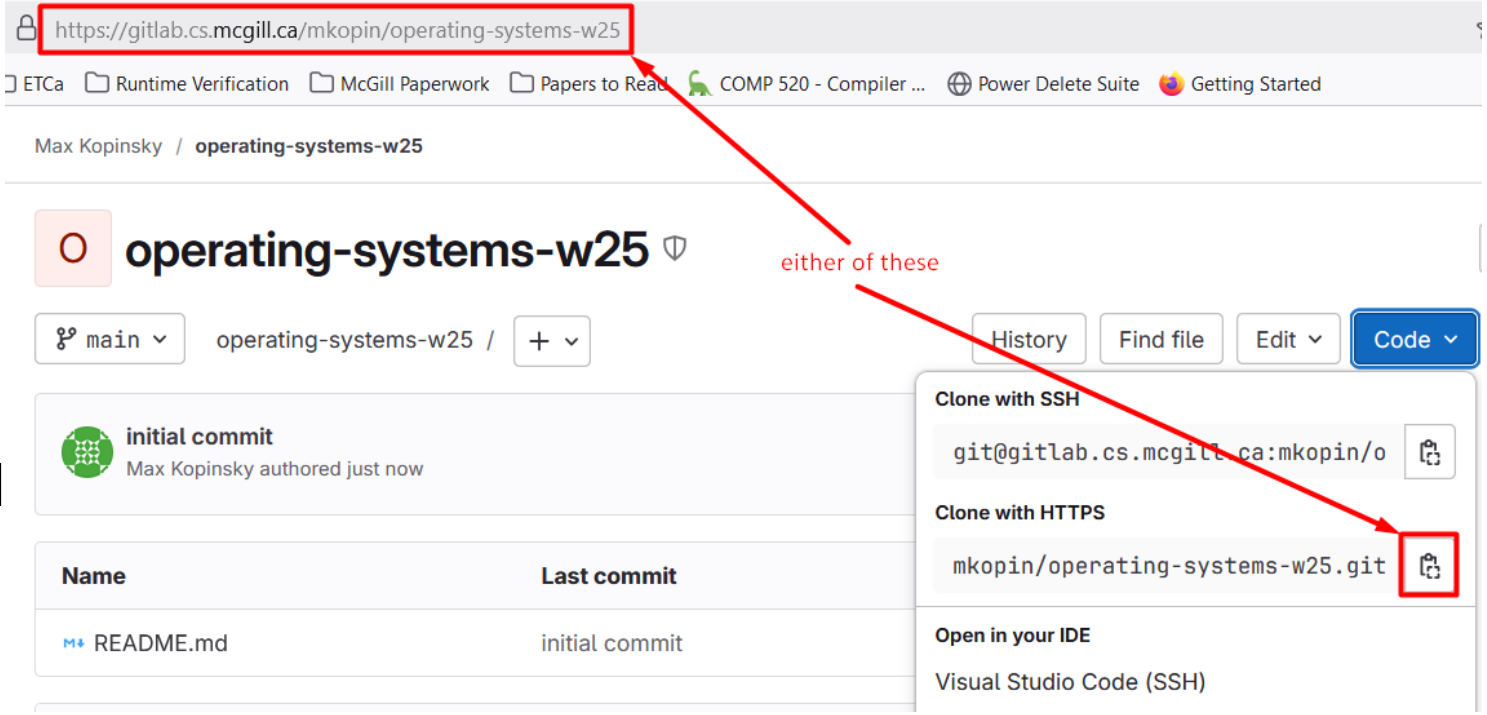
Week 3

Synchronization Primitives

Max Kopinsky
23 January, 2025

Assignment 1 Release – Clone

- Clone your fork
  - **on mimi**
  - (+ your machine, if you want)
  - Use your SOCS user+passwd



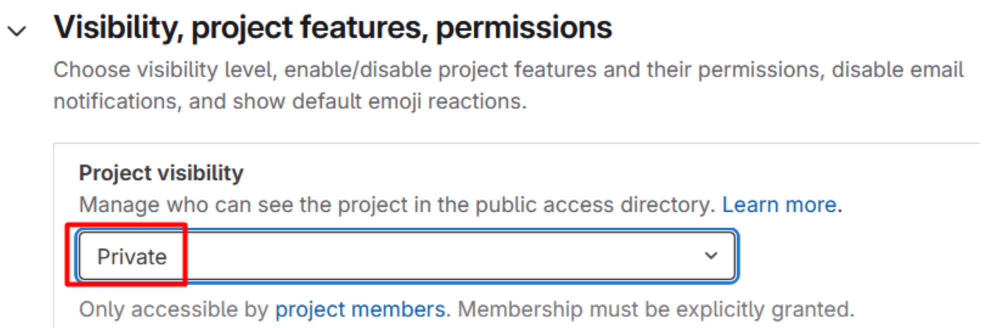
This command (with your own repo link)

```
mkopin@teach-node-04:~
$ git clone https://gitlab.cs.mcgill.ca/mkopin/operating-systems-w25
Cloning into 'operating-systems-w25'...
Username for 'https://gitlab.cs.mcgill.ca': mkopin
Password for 'https://mkopin@gitlab.cs.mcgill.ca':
warning: redirecting to https://gitlab.cs.mcgill.ca/mkopin/operating-systems-w25.git/
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
```

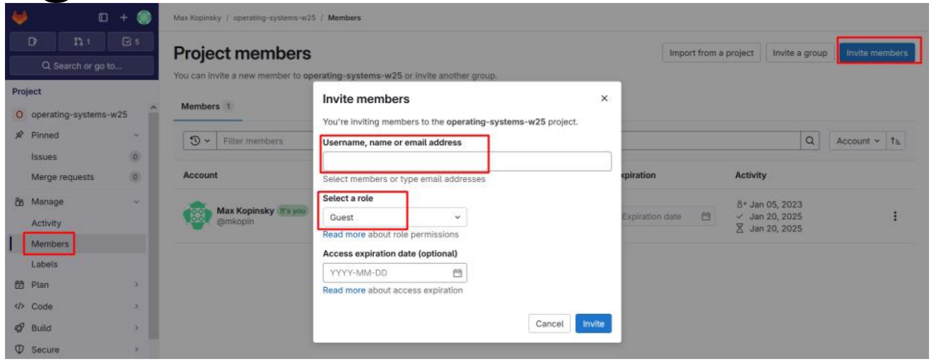
You can ignore this warning if you get it

Assignment 1 Release – Visibility Settings

- From the GitLab page for your repository
- Left panel > settings > General > Visibility
  - Change “Project Visibility” to **private**
  - If you do not do this, other students can copy your work. You would **both** be in trouble.



- Add your partner, myself, and the autograding TA to members:
  - Partner role: developer
  - Also add: @mkopin (me!) with role Reporter
  - And add @mohamad.danesh as Reporter too



Assignment 1 Release – Git Merge

- Finally whenever we release an assignment, you must update your fork with the changes from our "upstream" fork to get the materials (instructions, test cases, code, …) for that assignment.
- These commands will also be posted on Discord.

```
$ git remote add staff https://gitlab.cs.mcgill.ca/mkopin/operating-systems-w25
$ git fetch staff
$ git checkout main
$ git merge staff/main
```

This one must be to my repository

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software	**Monitors** ✅ **Locks (mutex)** **Semaphores** **Condition Variables**
Hardware	Loads Stores Test&Set Disable Interrupts

Condition Variables

- Used when thread A needs to wait for an event done by thread B
- A waits until a certain condition is true
  - First test condition,
  - If condition not true, call `pthread_cond_wait()`
    - A blocks until another thread "signals" the condition variable

- At some point B makes the condition true
  - Then B calls `pthread_cond_signal()`, which unblocks A.

Advantage of Condition Variable of Locks

- A waits until a certain condition is true
  - A goes to sleep here.
  - Does not keep CPU busy.

Condition variables use (incorrect)

Thread A

```
x = f (a , b) ;
if (x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Find the data race.

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Interrupt

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal(&cv);
```

:(Broadcast missed by A

Condition variables use (incorrect)

Thread A

```
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv);
```

Thread B

```
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
```

A waits forever...

Condition variables use (still incorrect)

Thread A

```
pthread_mutex_lock(&mutex ) ;
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv, &mutex);
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex_lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
pthread_mutex_unlock(&mutex);
```

Remember: Condition variable is a shared resource between A and B
→ Every time you use a condition variable you must also use a mutex to prevent the race condition.

One more issue…

Sometimes, the wait function might return even though the condition variable has not actually been signaled. **This is part of the API** as an unfortunate necessity.

Thread A

```
pthread_mutex_lock(&mutex ) ;
x = f ( a , b ) ;
if ( x < 0 || x > 9)
  pthread_cond_wait (&cv, &mutex);
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex_lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
pthread_mutex_unlock(&mutex);
```

Example:

If process P running A and B receives an OS signal

- Any thread in P can be chosen to process the signal.
- → A might be chosen to process the signal handling function
- → wait returns with an error code → A runs even if condition is not true…

How can we fix this?

Condition variables use (correct)

- Retest the condition after pthread_cond_wait() returns.
  - This is most easily done using a loop.

Thread A

```
pthread_mutex_lock(&mutex ) ;
while (1) {
  x = f ( a , b ) ;
  if ( x < 0 || x > 9) {
    pthread_cond_wait (&cv, &mutex);
  } else {
    break;
  }
}
pthread_mutex_unlock(&mutex) ;
```

Thread B

```
pthread_mutex_lock(&mutex );
//change a and b;
x = f (a , b) ;
if ( x >= 0 && x <= 9)
  pthread_cond_signal (&cv);
pthread_mutex_unlock(&mutex);
```

Conditional Variables Interface

- pthread_cond_init(pthread_cond_t *cv, pthread_condattr_t *cattr)
  - Initialize the conditional variable, cattr can be NULL

- pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex)
  - Block thread until condition is true, and atomically unlock mutex.
  - When the thread is unblocked, reacquire the mutex as if by pthread_mutex_lock
    - Might cause thread to block again!

- pthread_cond_signal(pthread_cond_t *cv)
  - Unblock one thread at random that is blocked by the condition variable

- pthread_cond_broadcast(pthread_cond_t *cv)
  - Unblock all threads that are blocked on the condition variable pointed to by cv.

Condition Variable Example

```c
pthread_cond_t is_zero;
pthread_mutex_t mutex;
int shared_data = 100;


void *thread_func(void *arg){
 while(shared_data > 0) {
  pthread_mutex_lock(&mutex);
  shared_data--;
  printf("%d ", shared_data);
  pthread_mutex_unlock(&mutex);
 }

 printf("Signaling main\n");
 pthread_cond_signal(&is_zero);
 return NULL;
}
```

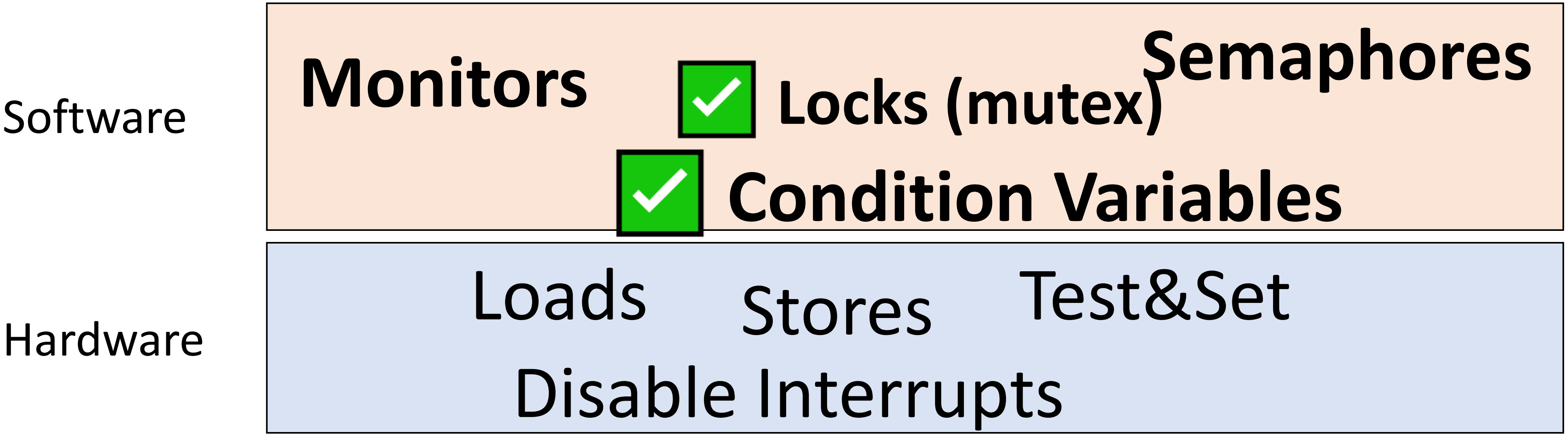
```c
int main (void){

 pthread_t tid;
 void * exit_status;
 int i;

 pthread_cond_init(&is_zero, NULL);
 pthread_mutex_init(&mutex, NULL);

 pthread_create(&tid, NULL, thread_func, NULL);

 pthread_mutex_lock(&mutex);
 printf("Start waiting in main\n");
 while(shared_data!=0)
  pthread_cond_wait(&is_zero, &mutex);
 pthread_mutex_unlock(&mutex);

 printf("Done waiting in main!\n");

 pthread_join(tid, &exit_status);
 return 0;
}
```

Synchronization

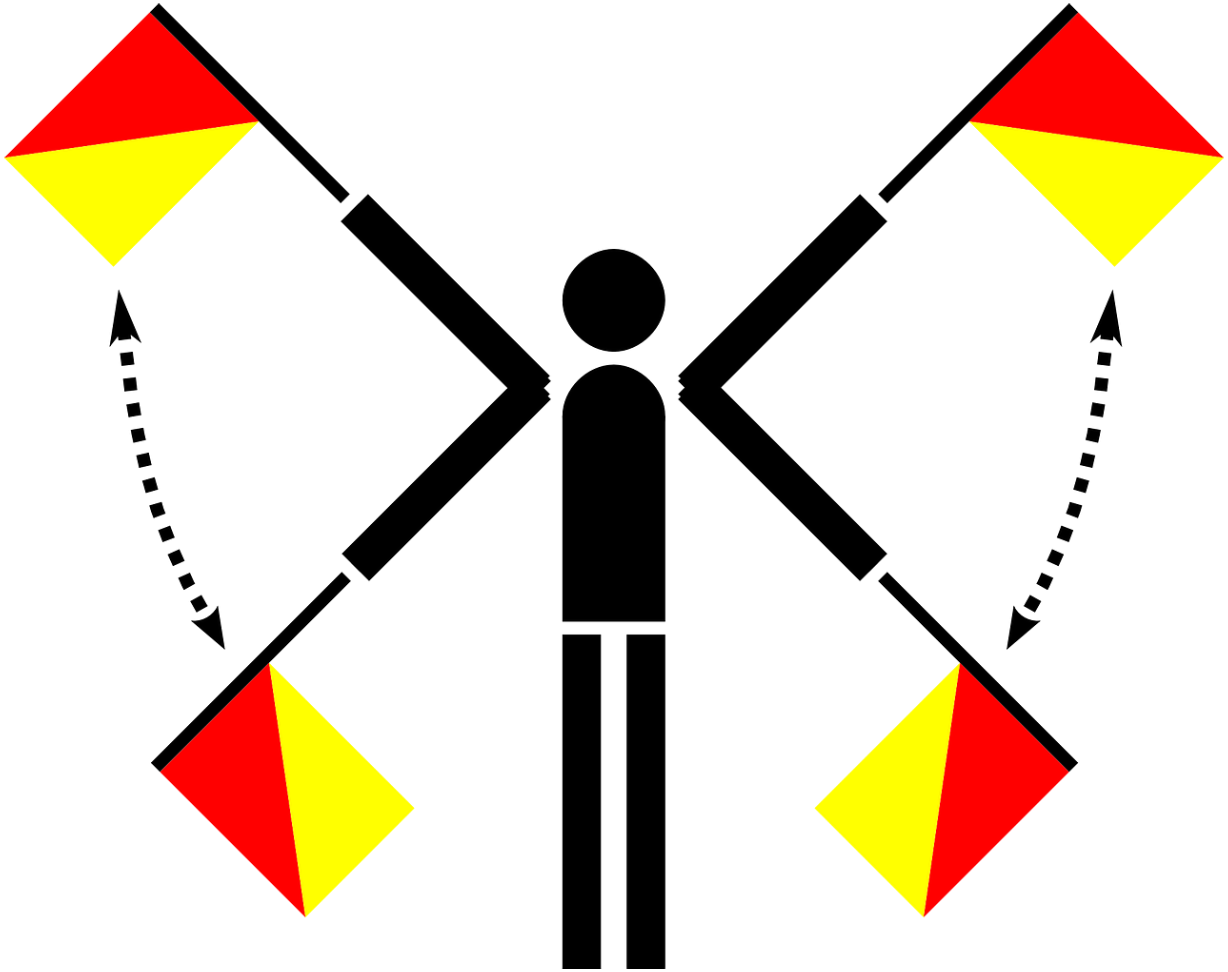
Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



Semaphores



What are Semaphores?

- A shared, non-negative counter.

- Two primary operations:
  - Wait → attempts to decrement the counter; blocks when counter is 0.
  - Post (or Signal) → attempts to increment the counter.

- #include<semaphore.h>

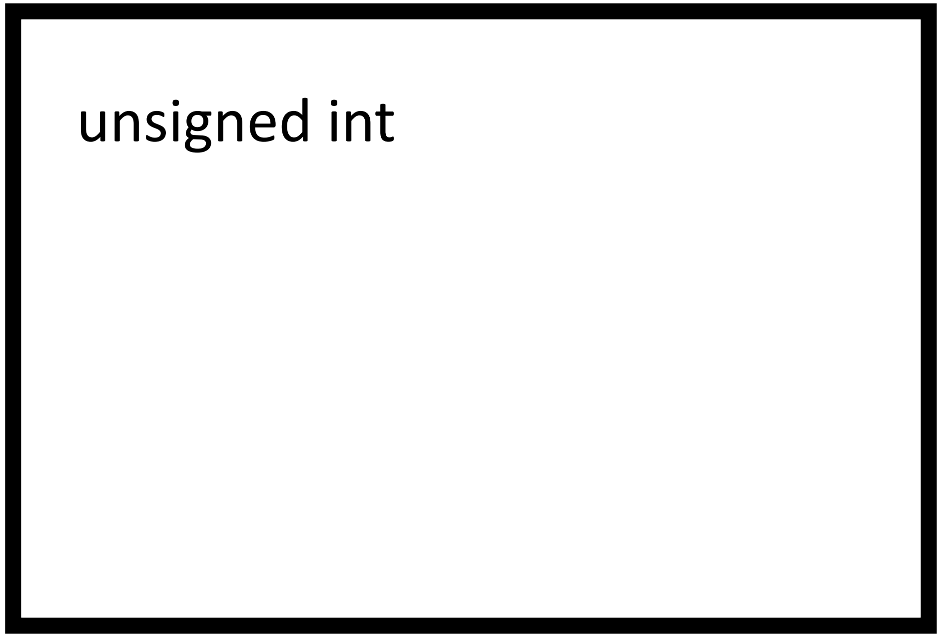
Changes are atomic

unsigned int

Semaphore

Changes are atomic

2 operations



Semaphore

wait()

post()

unsigned int

Semaphore

Changes are atomic

2 operations

wait()

try to decrement semaphore value

block if value = 0

post()

increment semaphore value

unsigned int v

3

Semaphore

```
wait() {

 while(1){
  if (v>0){
   v--;
   return;
  }
 }

}
```

Atomic execution

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

unsigned int v

2

Semaphore

```
wait() {

 while(1){
  if (v>0){
   v--;
   return;
  }
 }
}
```

Atomic execution

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

unsigned int v

0

Semaphore

```
wait() {

 while(1){
  if (v>0){
   v--;
   return;
  }
 }
}
```

Atomic execution

Wait until semaphore value becomes positive again

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

unsigned int v

1

Semaphore

```
wait() {

 while(1){
  if (v>0){
   v--;
   return;
  }
 }
}
```

Atomic execution

Once the value is positive, the thread that
Was waiting is able to decrement the value.

Note: wait() is not actually implemented like this. This is how the behavior looks like to the programmer

unsigned int v

0

Semaphore

```
post() {
v++;
return;
}
```

Atomic execution

Note: post() is not actually implemented like this. This is how the behavior looks like to the programmer

```
post() {
    v++;
    return;
}
```

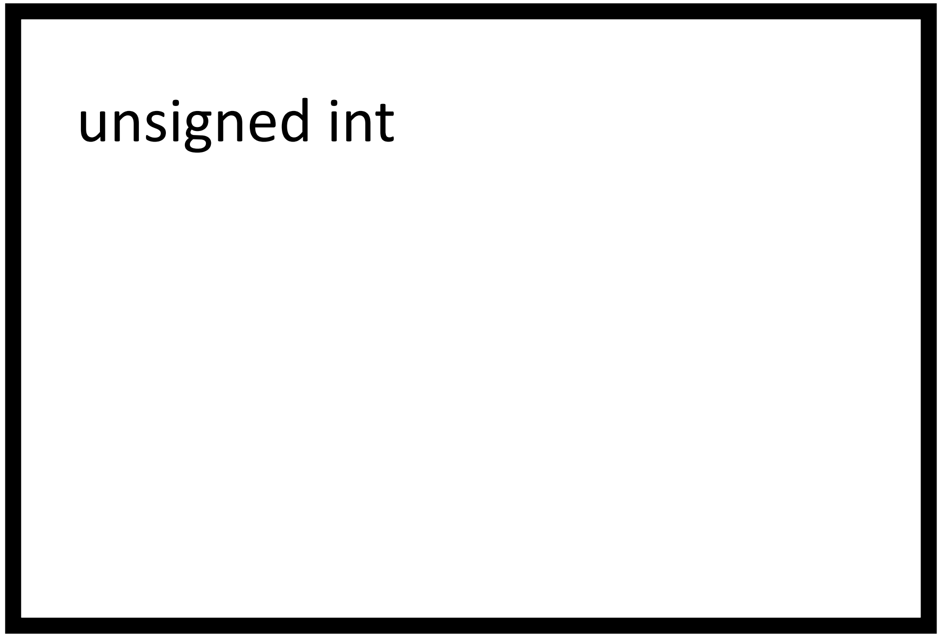
Atomic execution

unsigned int v

1

Semaphore

Note: post() is not actually implemented like this. This is how the behavior looks like to the programmer



Semaphore

One last thing:

- Semaphore value can be initialized upon creation to a positive value
- or 0

Semaphore Uses

Mutual exclusion

- A semaphore with its counter initialized to 1 can be used as a lock.

Bound the concurrency

- Only allow X threads out of N to proceed.

Producer-consumer problem

- More complex use of semaphores. Will see in 2 weeks.

Semaphores interface

- int sem_init(sem_t **sem*, int *pshared*, unsigned *value*);
- int sem_post(sem_t **sem*);
- int sem_wait(sem_t **sem*);

For more details: https://man7.org/linux/man-pages/man0/semaphore.h.0p.html

int sem_init(sem_t **sem*, int *pshared*, unsigned *value*);

- Initializes the semaphore * *sem*. The initial value of the semaphore is *value*.
- If *pshared* is 0, the semaphore is shared among all threads of a process.
- If *pshared* is not zero, the semaphore is shared *between processes* but
  - Must be stored somewhere that multiple processes can see, e.g. file-mapped memory.
  - Don’t worry about this use case. We won’t discuss process-shared memory in this course.
- Return 0 on success, -1 on failure.

int sem_wait(sem_t **sem*);

- If the *sem* has a value > 0, decrement the value by 1.
- If *sem* has value 0, the caller will be blocked until *sem* has a value larger than 0.
- Return 0 on success, -1 on failure.

int sem_post(sem_t **sem*);

- Increment the value of *sem* by 1.
- If threads are blocked waiting for the semaphore, one of them (at random) will return successfully from its call to *sem_wait*(); the semaphore value is immediately decremented.
  - (this doesn’t happen until one of those threads is next scheduled – but focus on concepts!)
- Return 0 on success, -1 on failure.

Semaphores example

Note: If 2 is changed to 1, we have a lock behavior

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


pthread_t threads[5];
int tid[5];
sem_t sem;

void * thread_func(void *arg){
 int tid_ = tid[* (int *) arg];
 printf("Thread %d created\n", tid_);
 int j;

 sem_wait(&sem);
 for (j=0; j<3; j++){
  printf("T%d run %d\n", tid_, j);
  sleep(2);
 }
 sem_post(&sem);
}
```

```
int main(){

 sem_init(&sem, 0, 2);
 //sem initialized for all threads in the process; allow only 2 threads in the critical section at a time;

 int i;
 for (i=0; i<5; i++){
  tid[i]=i;
  pthread_create(&threads[i], NULL, thread_func, &tid[i]);
 }

 for (i=0; i<5; i++){
  pthread_join(threads[i], NULL);
 }
 sem_destroy(&sem);
 return 0;
}
```

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Software

Monitors ✅ **Locks (mutex)** ✅ **Semaphores**

✅ **Condition Variables**

Hardware

Loads Stores Test&Set

Disable Interrupts

Monitors

- Collection of variables and functions
- Threads can only access monitor functions
  - Variables are private to the monitor
- Only one process at a time can execute code inside the monitor.

Monitors

- Pthreads does not offer a monitor primitive ☹
- … but possible to implement monitor semantics using mutex and condition variables ☺
- We won’t get into this in this course. Investigate if you’re curious! ☺

Synchronization

How are locks, semaphores, cond var, etc **implemented**?

Software

Monitors Locks (mutex) Semaphores

Condition Variables

Hardware

Loads Stores Test&Set

Disable Interrupts

Reminder of Assumptions:

- Multitasking OS
  - “Tasks” can be threads or processes
- Single-processor system
  - Only one task runs at any time
- **Not** assuming OS helps with synchronization
  - Of course, all the modern OSes do
  - But if you need to know how to implement… its because you don’t have one.

Lock implementation motivating example “Too much milk”

- Alice and Bob are roommates
- They want to coordinate grocery shopping
    - Need to be careful to not buy too much of perishable items, like milk.

Schedule that leads to too much milk

Time	Alice	Bob
3:00	Look in Fridge. Out of milk.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in Fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home, put milk away.

Problem specifications

Safety

- Never more than one person buys

Liveness

- Someone buys if needed

Lock implementation: first attempt

Restrict ourselves to use only atomic load and store operations as building blocks.

Idea: Use **a note** to avoid buying too much milk:

- Leave a note on fridge before buying (kind of “lock”)
- Remove note after buying (kind of “unlock”)
- Don’t buy if note (wait)

Synchronization

How are locks, semaphores, cond var, etc **implemented**?

Software	Monitors, Locks (mutex), Semaphores, Condition Variables
Hardware	**Loads**, **Stores**, **Test&Set**, **Disable Interrupts**

Lock implementation: first attempt (**incorrect**)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

Lock implementation: first attempt (incorrect)

```
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note; }
}
```

load (atomic)

store (atomic)

Lock implementation: first attempt (incorrect)

Thread A

```
if (noMilk) {
```

Thread B

```
if (noMilk) {
  if (noNote) {
```

Aside: when I want to anthropomorphize threads/processes like this, thread A will be “Alice” and thread B will be “Bob.”

Lock implementation: first attempt (incorrect)

Thread A

```
if (noMilk) {


  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

Thread B

```

if (noMilk) {
  if (noNote) {
```

Lock implementation: first attempt (incorrect)

Thread A

```
if (noMilk) {

  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

Buy milk twice ☹
Remember: scheduler can create any interleaving
We must assume a malicious scheduler

Thread B

```
if (noMilk) {
  if (noNote) {

    leave Note;
    buy milk;
    remove note; }
}
```

First attempt result

- Still too much milk but only occasionally!
  - This is worse than a consistent error, because it is harder to catch.
- Thread can get context switched **after checking milk** (and note) **but before buying** milk!

Lock implementation: second attempt (incorrect)

- How about labeled notes?

<u>Thread A</u>

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

<u>Thread B</u>

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Problem solved?

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

Proc switch

leave note A;

```
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

<u>Thread A</u>

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Proc switch

<u>Thread B</u>

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Proc switch

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Proc switch

Not quite...

Lock implementation: second attempt (incorrect)

- How about labeled notes?

Thread A

```
leave note A;
if (noNote B) {
  if (noMilk) {
    buy milk;
  }
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Not quite…

Nobody buys milk ☹

Lock implementation: third attempt (correct)

- How about labeled notes? (similar to Peterson, 1981)

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

This works!

B has priority to buy

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;



if (noMilk)
  buy milk;
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Happened before

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
```

Happened before

Wait for note B
to be removed

```
if (noMilk)
  buy milk;
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 1

Thread A

```
leave note A;
while (note B)
  do nothing;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Wait for note B
to be removed

```
if (noMilk)
  buy milk;
remove note A;
```

Correctness argument. Case 2

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 2

Thread A

```
leave note A;
while (note B)
  do nothing;
if (noMilk)
  buy milk;
remove note A;
```

Happened before

Thread B

```
leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;
```

Correctness argument. Case 2

Thread A

leave note A;
while (note B)
  do nothing;

Wait for note B
to be removed

if (noMilk)
  buy milk;
remove note A;

Thread B

leave note B;
if (noNote A) {
  if (noMilk)
    buy milk;
}
remove note B;

Happened before

Prove Correctness?

- By hand, surprisingly hard and often requires insight
- Mechanical tool exists: “Linear Temporal Logic”
- I can show an LTL specification + proof of solution 3 in OH if interested

- Even with LTL, still surprisingly tricky!
- If this kind of automated proof is interesting, try take 525
    - (Hopefully, it’s taught again soon ☹)

Attempt 3 discussion

Solution 3 works, but it’s really unsatisfactory.

- Complex, even for this simple example.
    - Hard to convince yourself that this really works.
- What we have guarantees that A *or* B can enter critical section
    - But we usually want to know that they *both* can, eventually
    - (In this case, we just know that they are trying to do the same thing)
- A’s code is different from B’s –what if lots of threads?
    - Code would have to be slightly different for each thread.
- While A is waiting, it is consuming CPU time.
    - This is called “busy-waiting”.

There must be a better way!

- Version of Solution 3 where both threads have same code exists
  - See Peterson’s Algorithm (1981)
- Have higher-level hardware primitives than atomic load & store
  - Compare-and-swap, test&set, LL/SC
- Build higher-level abstractions on this hardware support
- **Or: implement abstractions at OS level**

Fun Fact: It Gets Worse

- Modern multicore processors execute instructions *out of order*
- Normally not possible to notice
  - Hardware is carefully designed so that single thread can't tell
- But multiple threads can observe re-ordered memory accesses
- Re-ordering note bookkeeping breaks Solution 3
- With only atomic loads/stores (and nothing "stronger"), mutual exclusion on modern processors would be impossible.

Synchronization

How are locks, semaphores, cond var, etc **implemented**?

Software

Monitors Locks (mutex) Semaphores

Condition Variables

Hardware

Loads **Stores** **Test&Set**

Disable Interrupts

Disabling interrupts

- Idea: lock implementation code executed in kernel mode.
  - No need for “spinlock” when we have a scheduler!
- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable
- Thread that can’t take a lock goes to sleep
- Danger! What if preemption timer fires during lock implementation?
- (Again: this idea is for single-processor systems.)

Disabling interrupts

```
int value = FREE;

Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Unlock() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    place on ready queue;
  }
  else {
    value = FREE;
  }
  enable interrupts;
}
```

Disabling interrupts discussion

Why do we need to disable interrupts?

- Avoid preemption between checking and setting lock value
- Otherwise, two threads could think that they both have lock

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

Critical section is short in kernel mode

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

Enable interrupts here?

☹ Release can check the queue and not wake up thread

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

Enable interrupts here?

- Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
- Misses wakeup, but still holds lock (deadlock!) ☹☹

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

Want to enable interrupts after sleep()

- But how?

What about enabling interrupts when going to sleep?

```
Lock() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  }
  else {
    value = BUSY;
  }
  enable interrupts;
}
```

Want to enable interrupts after sleep()

- But how? **In scheduler.**

In scheduler, since interrupts are disabled when you call sleep():

- Responsibility of the next thread to re-enable interrupts
- When the sleeping thread wakes up, returns to lock() and re-enables interrupts

Thread A		Thread B
.		
.		
disable ints		
sleep	context switch →	
		sleep return
		enable ints
		.
		.
		.
		disable int
	← context switch	sleep
sleep return		
enable ints		
.		
.		

Simulation of locks with disabled interrupts

Value: 0 | waiters | owner | READY

Running

Thread A

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

INIT

```
   int value = 0;

Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Ready

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1	waiters	owner

READY

Running

Thread A

Ready

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
INIT
   int value = 0;

Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Ready / Running

Thread A

Ready / Running

Thread B

INIT

```
int value = 0;
```

Thread A:

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

Thread B:

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Ready Running

Thread A

Running Waiting

Thread B

INIT
```
int value = 0;
```

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Running

Thread A

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

INIT

```
int value = 0;
```

```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Ready Waiting

Thread B

```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Simulation of locks with disabled interrupts

Value: 1 | waiters | owner | READY

Running / Ready — *Thread A*

Running / Ready — *Thread B*

INIT
```
int value = 0;
```

Thread A:
```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

Thread B:
```
lock.Lock();
 …
 critical section;
 …
lock.Unlock();
```

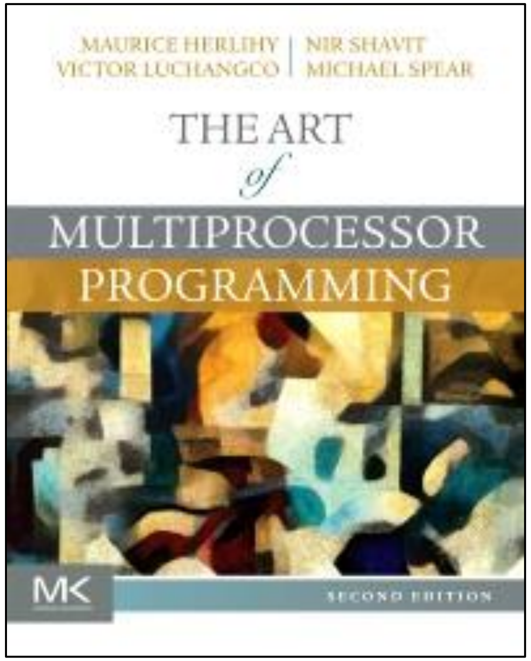
```
Lock() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Unlock() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Further Optional Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 25 – 31 (inclusive)
https://pages.cs.wisc.edu/~remzi/OSTEP/

Reading on concurrency: Herlihy & Shavit: The Art of Multiprocessor Programming, 2nd edition.



Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Maurice Herlihy (Brown University), Prof. Natacha Crooks (UC Berkeley).