Week 3

Synchronization Primitives

Oana Balmau

January 17, 2023

Schedule Highlights

Week 3 **Process Management**	jan 16 C Review: C Basics	jan 17 **Synchronization Primitives (1/2)** Optional reading: OSTEP Chapters 25 – 32 **add/drop deadline**	jan 18 OS Shell Assignment Released	jan 19 **Synchronization Primitives (2/2)** OS Shell Assignment Overview – with Jiaxuan	jan 20
Week 4 **Process Management**	jan 23 C Tools: GDB basics	jan 24 **Multi-process Structuring (1/2)** **Team registration deadline**	jan 25	jan 26 **Multi-process Structuring (2/2)**	jan 27
Week 5 **Process Management**	jan 30 C Review: Pointers & Memory Allocation I	jan 31 **Multithreading (1/2)** Practice Exercises Sheet: Process Management	feb 1	feb 2 **Multithreading (2/2)**	feb 3

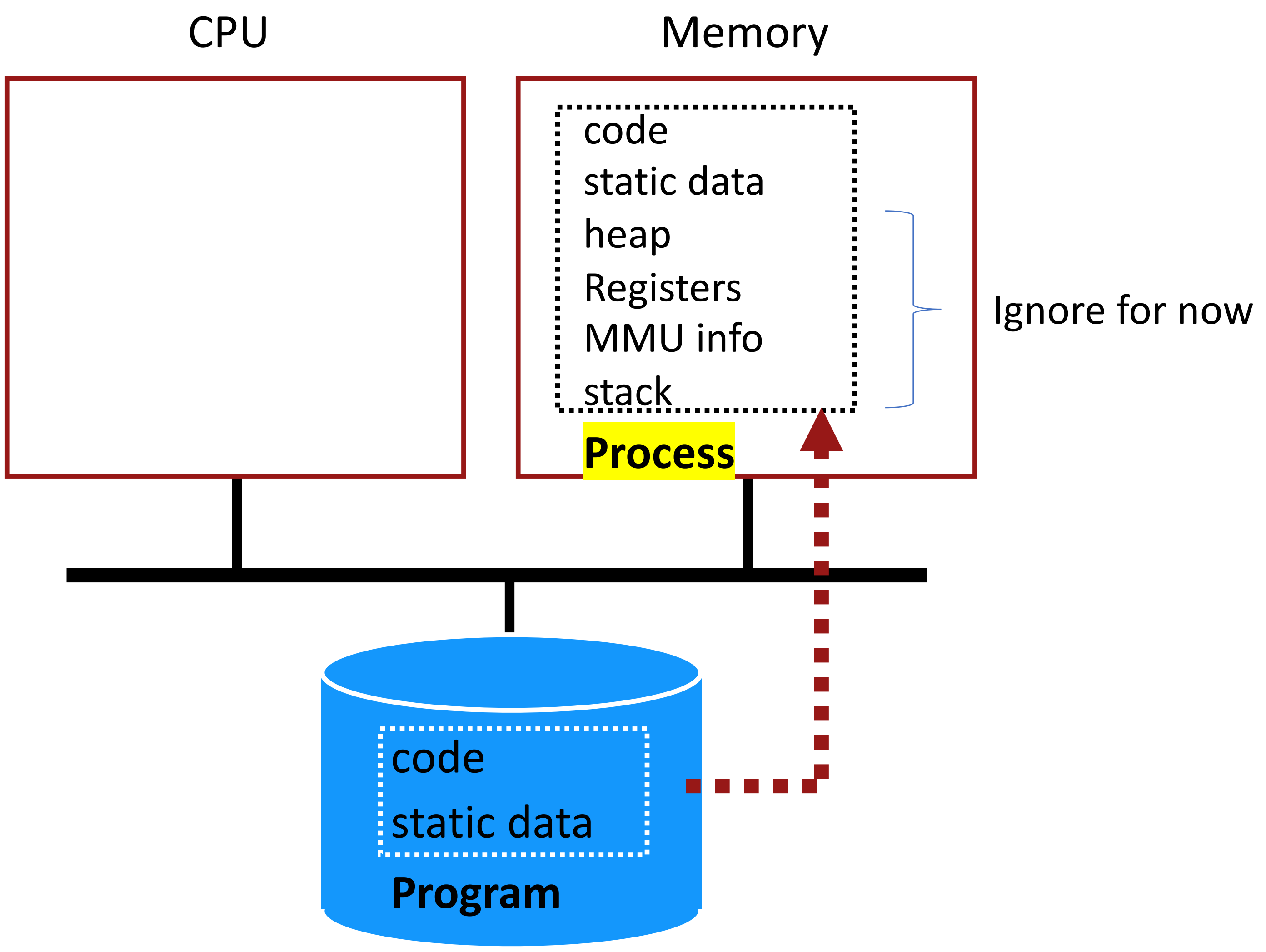
One more week to create teams.

Jiaxuan will explain the assignment during part of this class.

Recap from Week 2

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

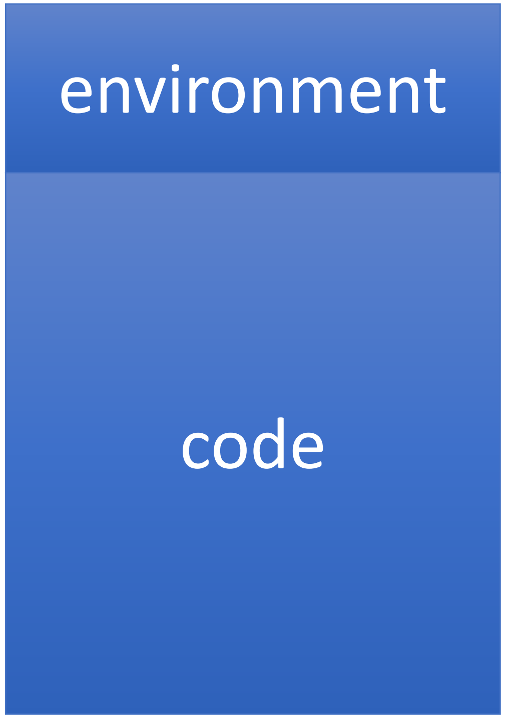
Recap from Week 2
Process = Program in execution



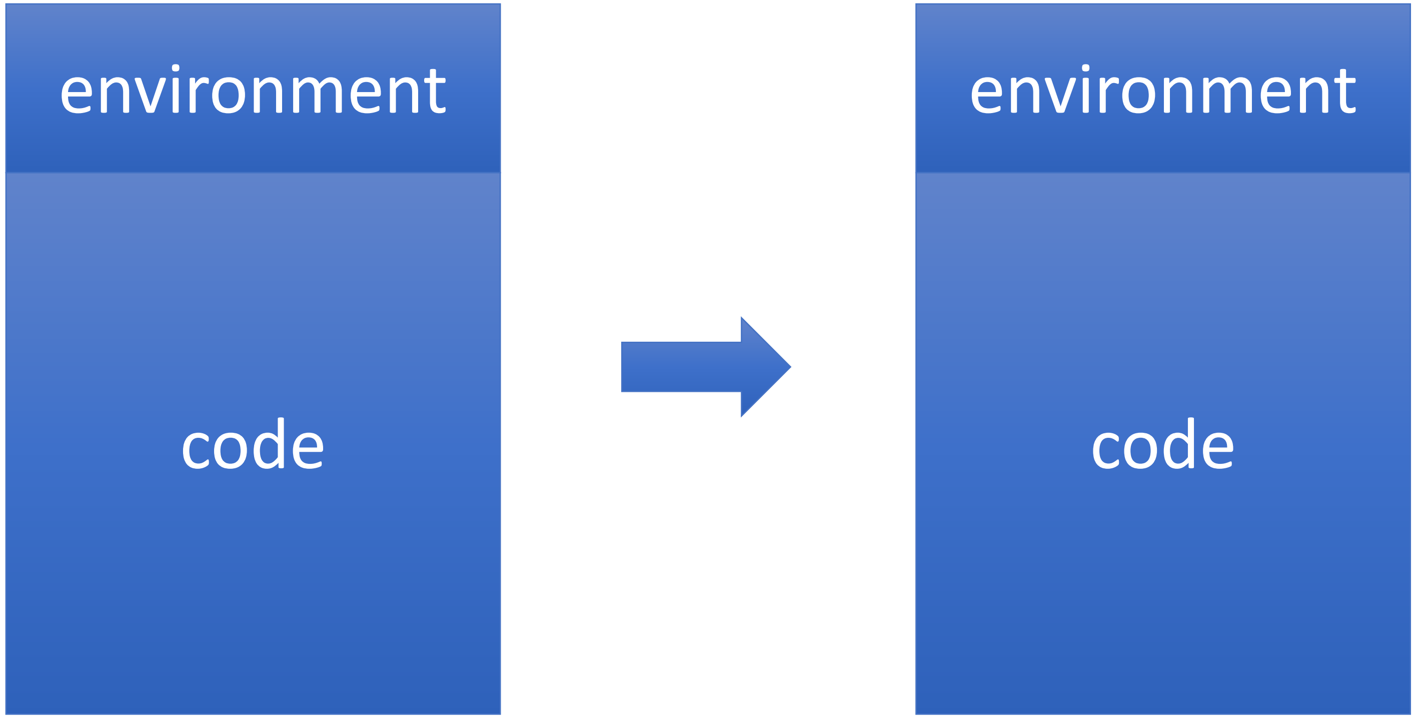
Recap from Week 2
Linux Process Primitives

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

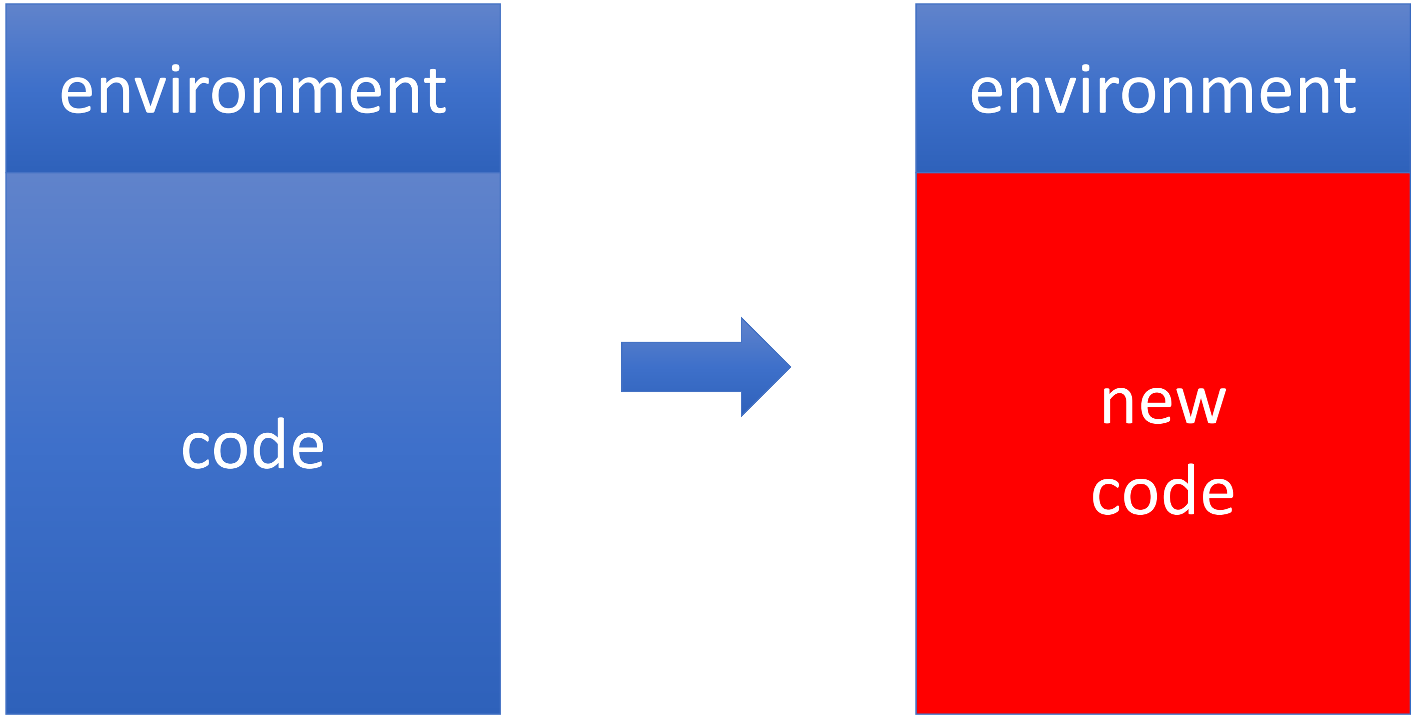
Recap from Week 2
Process = Environment + Code



Recap from Week 2
After a fork()

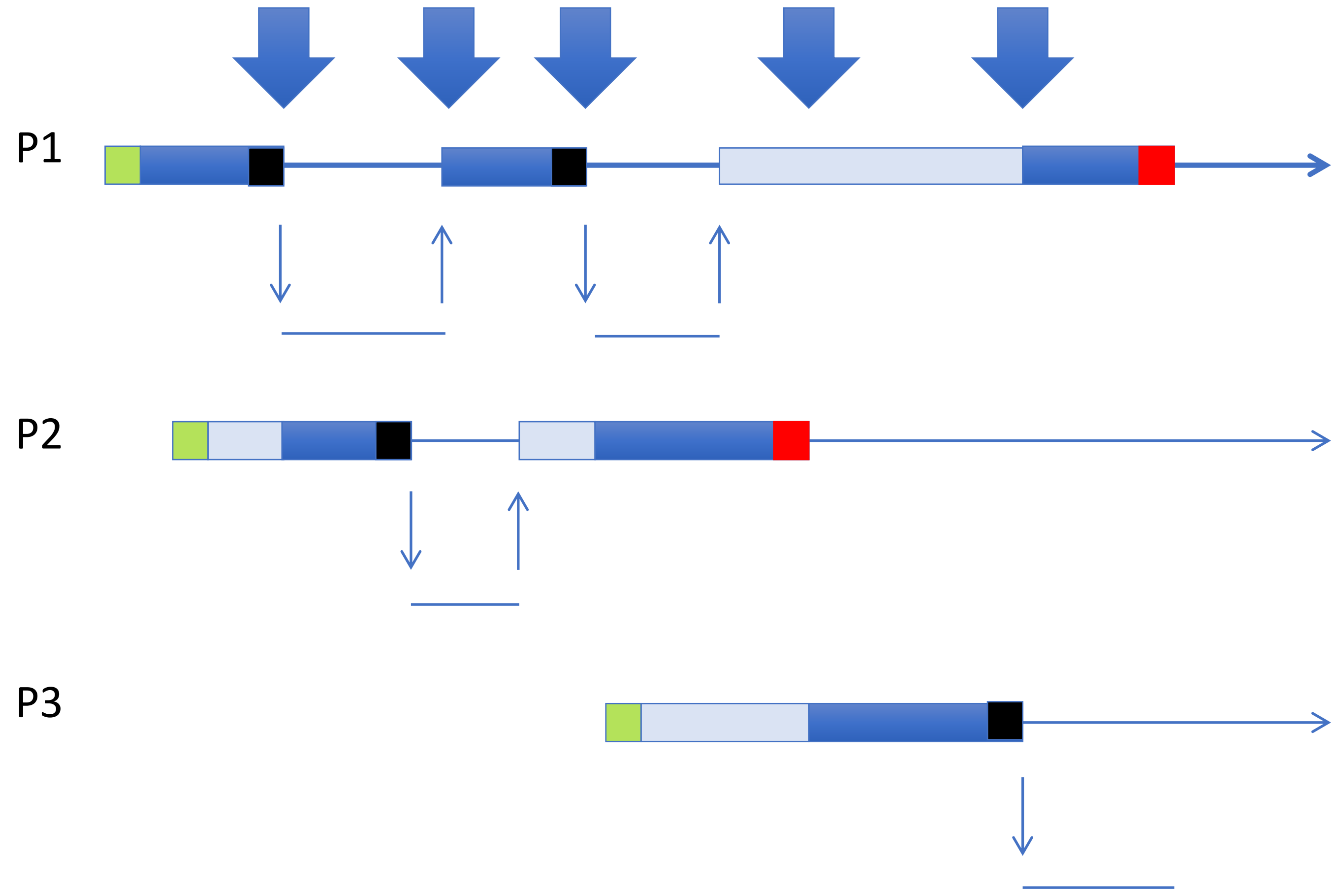


Recap from Week 2
After an exec() in the Child

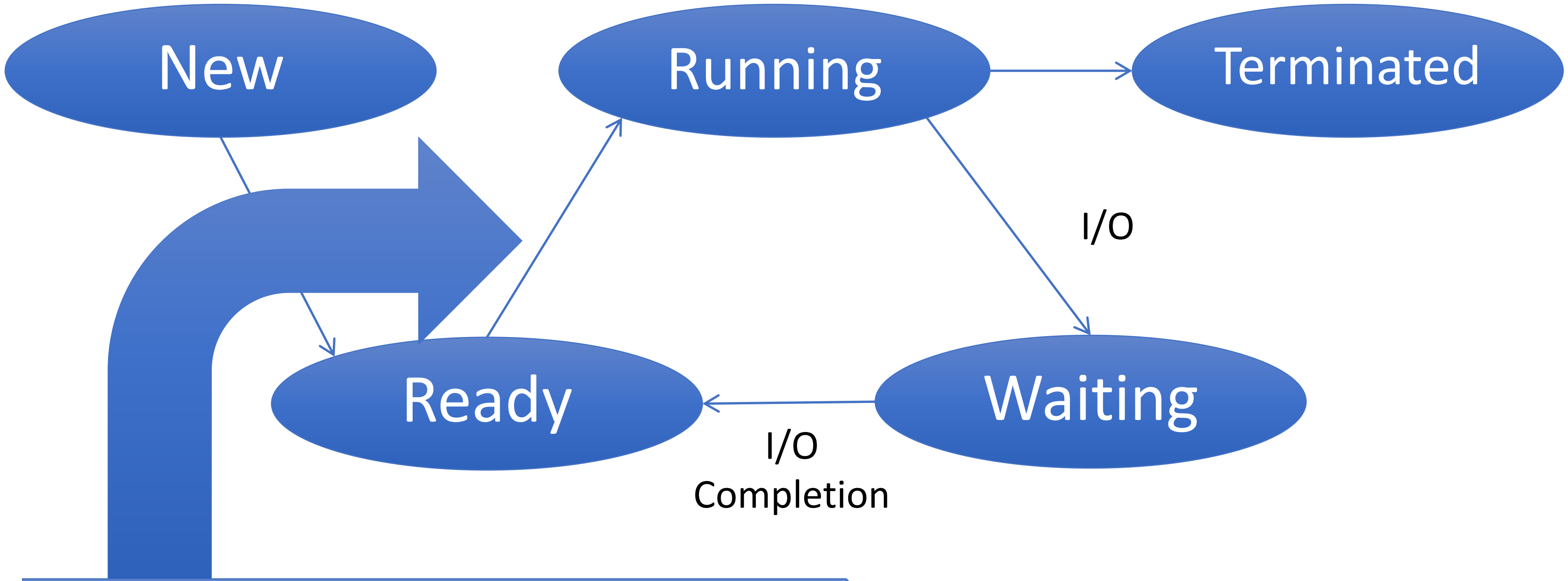


Recap from Week 2
Process Switch

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



Recap from Week 2
Process Scheduling



Many processes may be ready.
Process scheduler picks one.

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

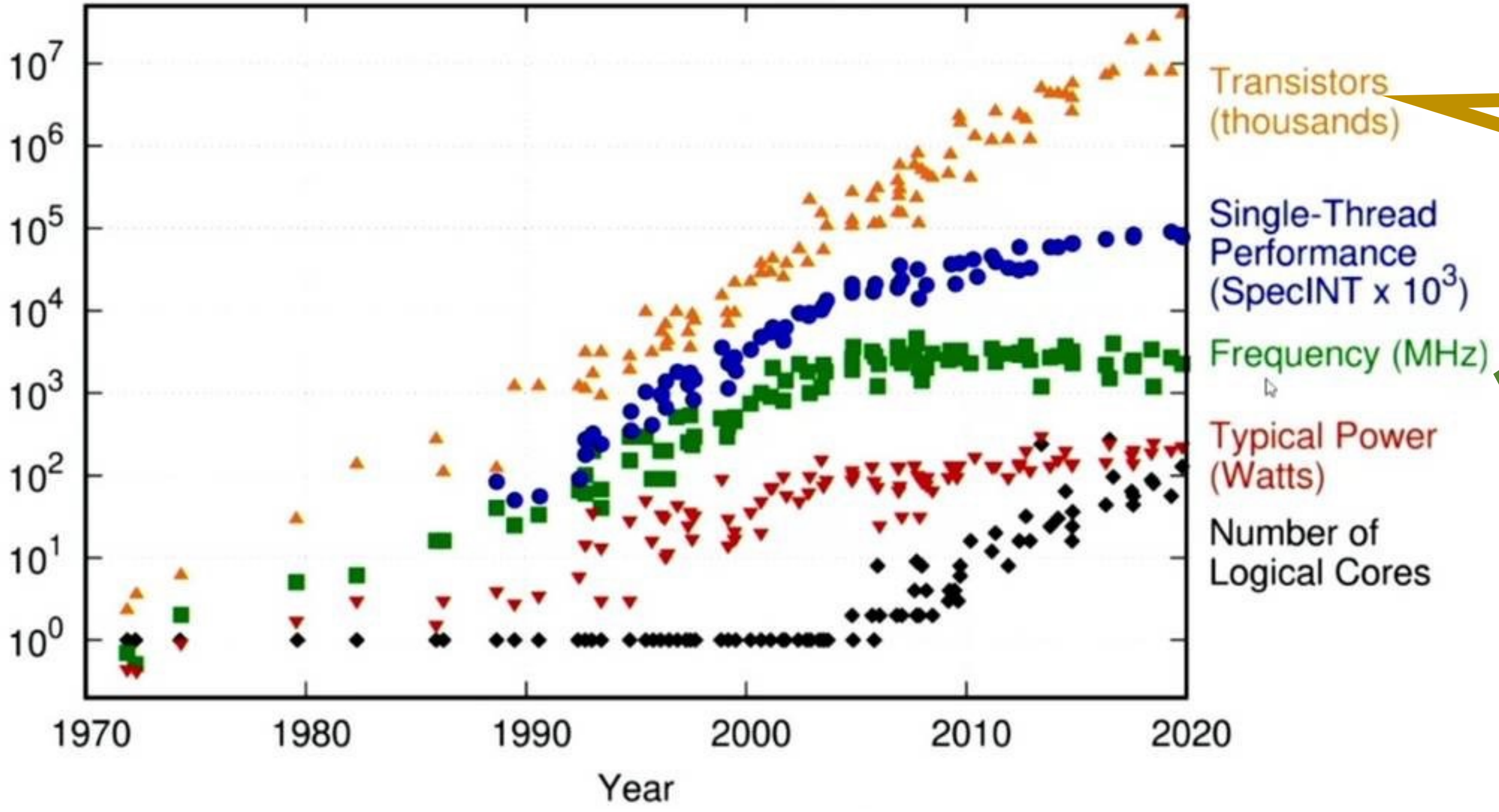
Real world concurrency

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

Key Concepts for Today

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

Motivation for Concurrency – Moore’s Law

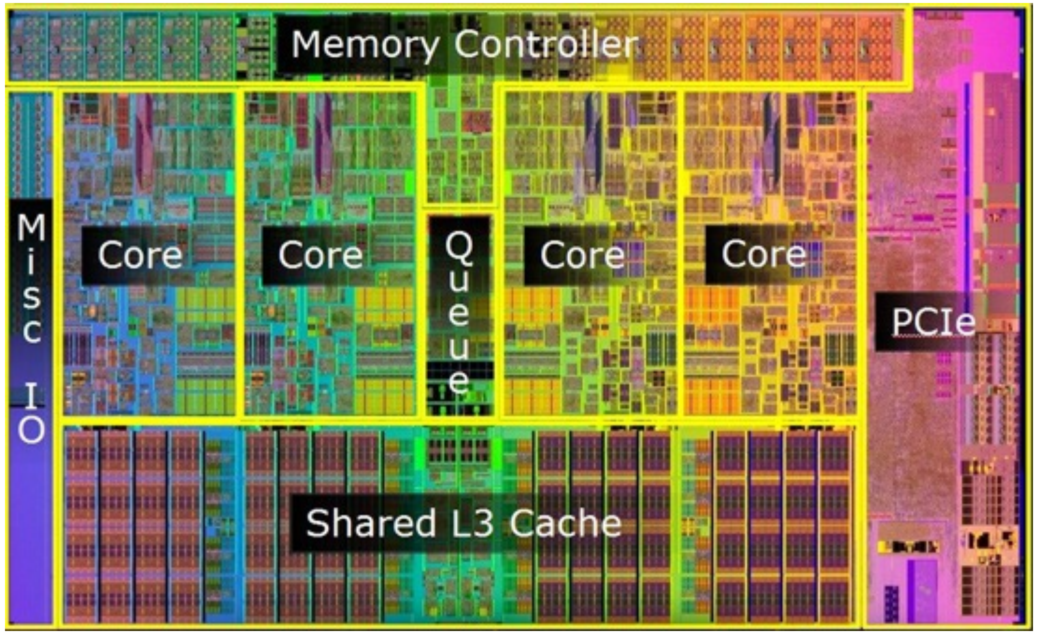


Transistor count still rising

Clock speed flattening sharply

Motivation for Concurrency

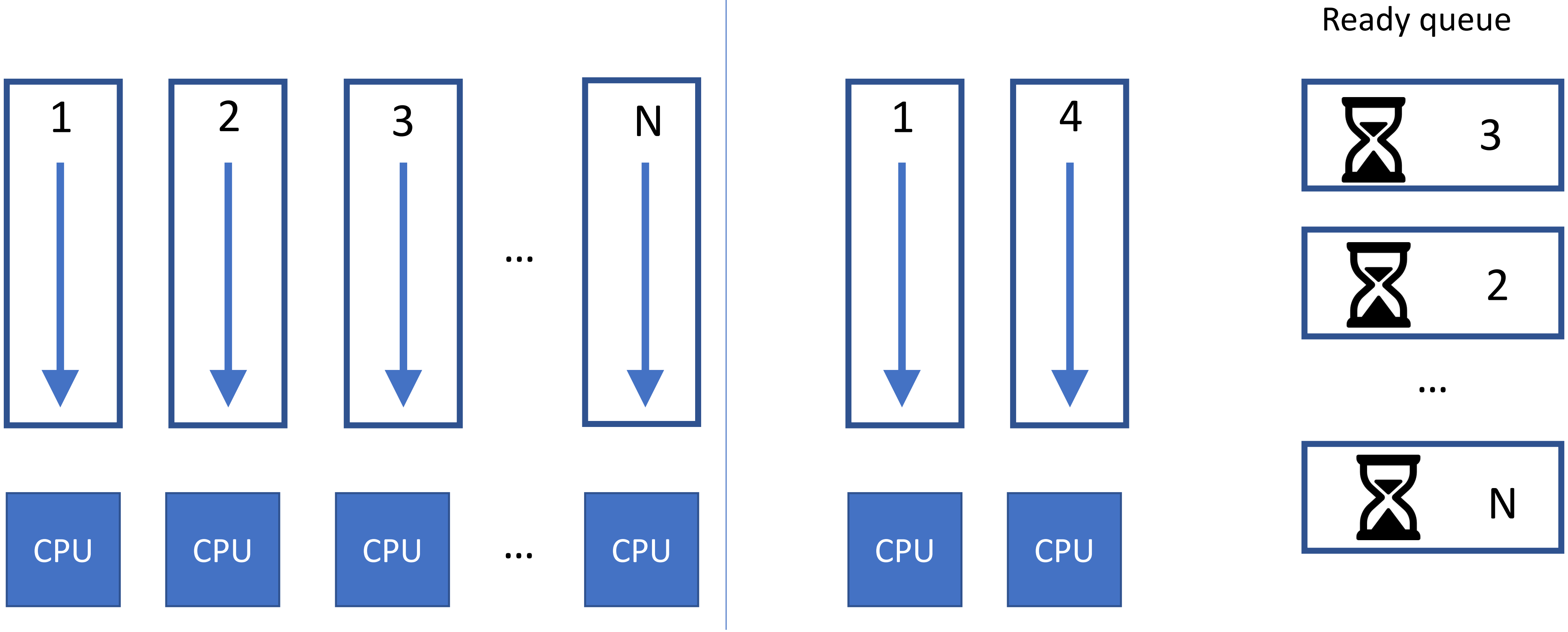
- CPU trend: Same speed, but multiple cores
- Goal: write applications that fully utilize many cores



Concurrency Abstraction vs Reality

Programmer Abstraction

Physical Reality



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

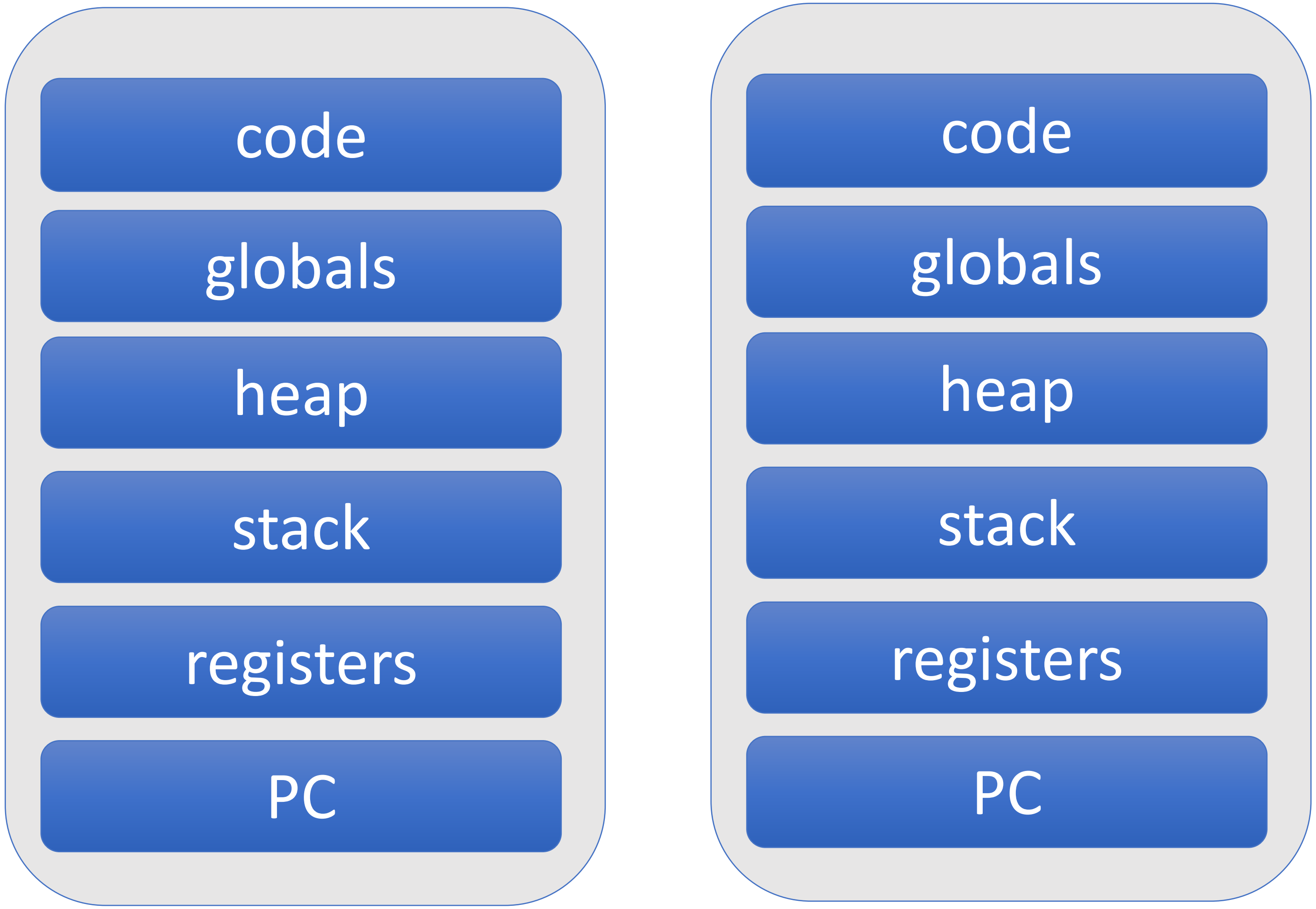
Concurrency – Option 2

Will see synchronization principles **today**

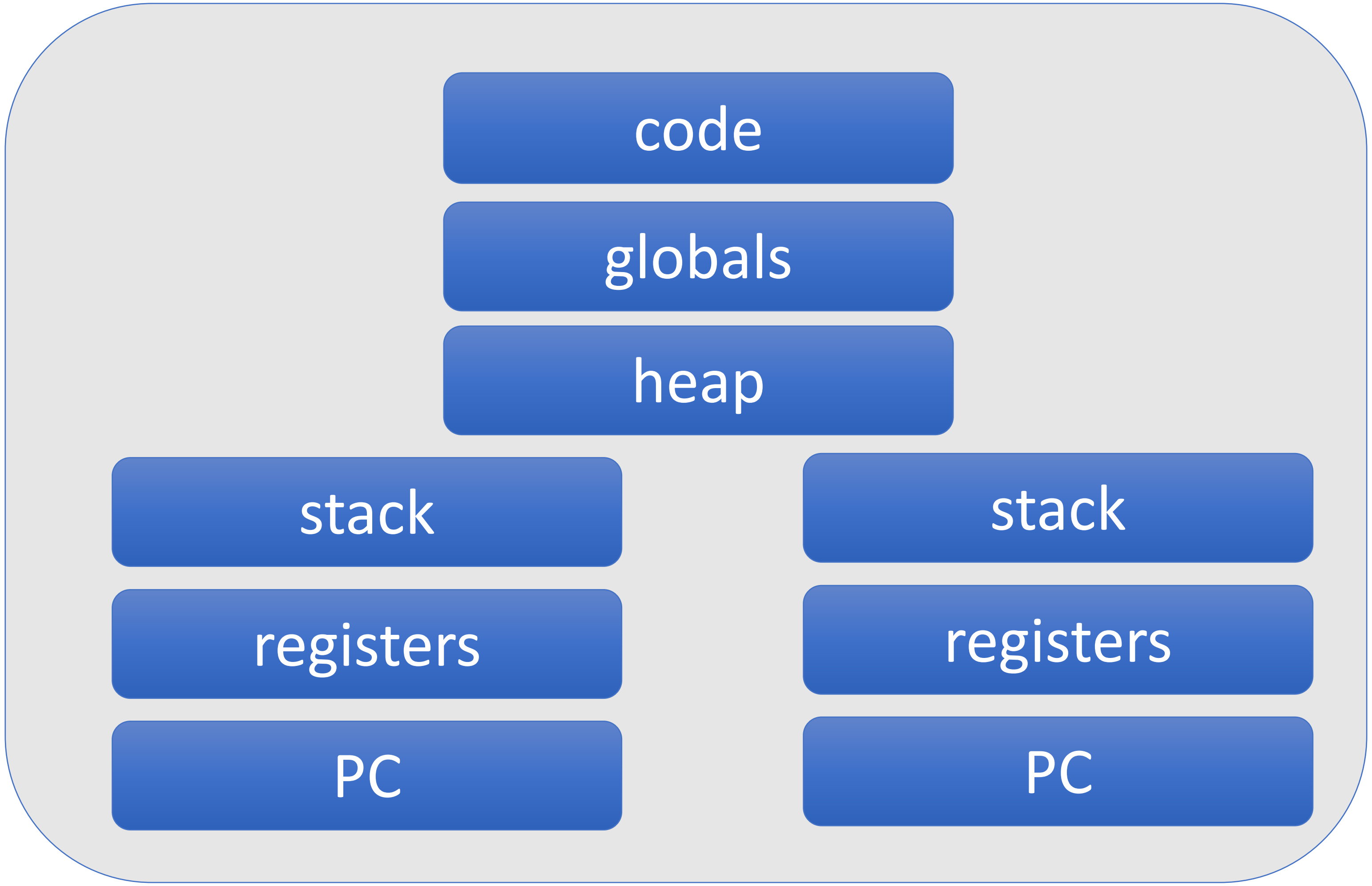
Will see practical examples in two weeks

- 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

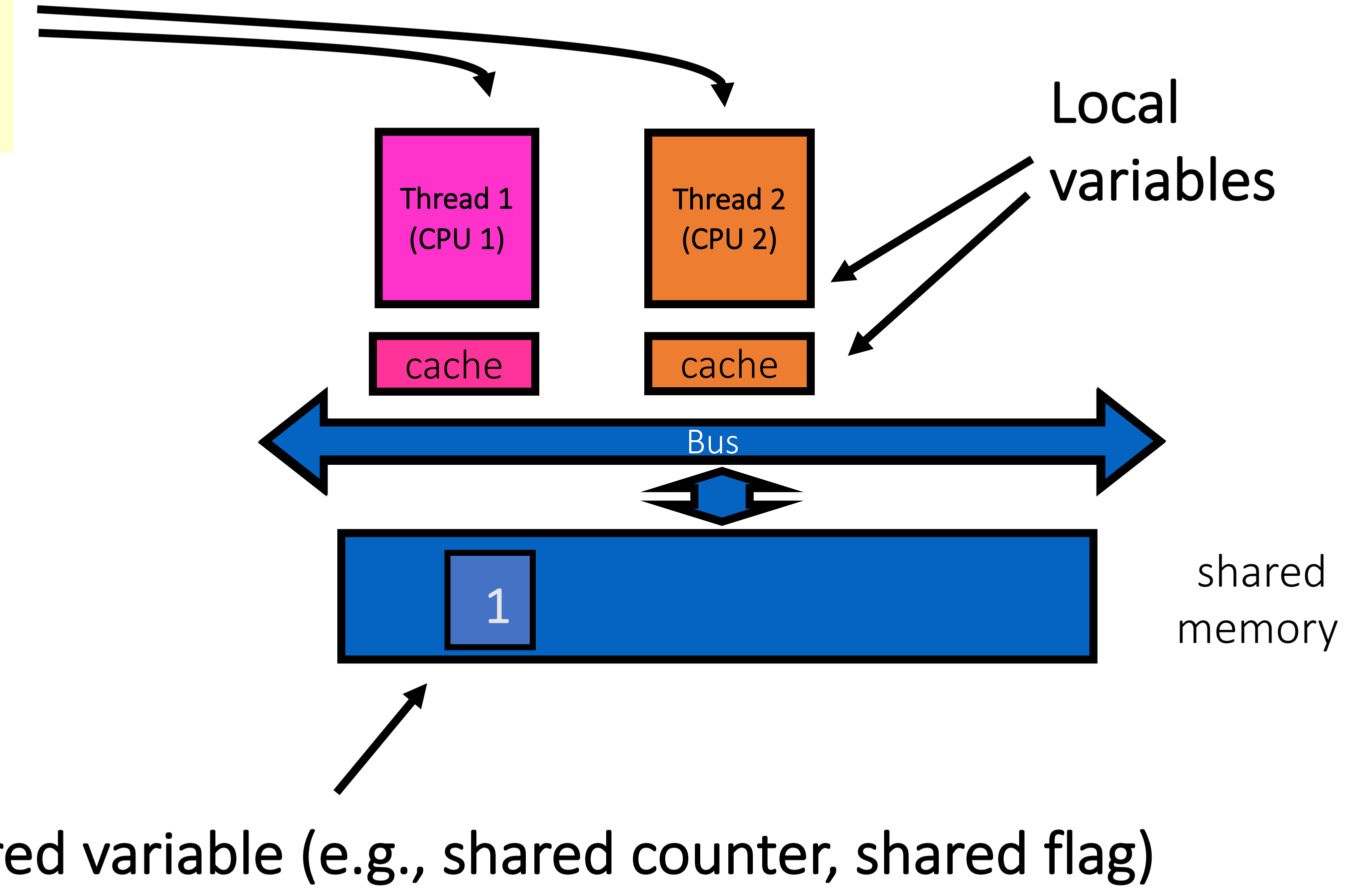
- Processes provide separation
  - In particular, memory separation (no shared data)
  - Suitable for coarse-grain interaction

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

Where Things Reside

```
void primePrint {
  int i =
ThreadID.get(); // IDs
in {0..9}
  for (j = i*10⁹+1,
j<(i+1)*10⁹; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

code



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**

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: ?	%eax: ?
%rip: 0x195	%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	%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: 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

state of T1 saved in T1 pcb upon context switch

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

state of T2 loaded in CPU

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

process control blocks:

Thread 1	Thread 2
%eax: 101	%eax: ?
%rip: 0x1a2	%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	%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: 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
- 0x19d mov %eax, 0x9cd4

T2 →

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	%eax: ? %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: 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

T1 →

- `0x195   mov 0x9cd4, %eax`
- `0x19a   add $0x1, %eax`
- `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	%eax: ?
%rip: 0x19d	%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: 100
%eax: 100
%rip = 0x19a

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: 100`
`%eax: 101`
`%rip = 0x19d`

process control blocks:

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

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

	Thread 1	Thread 2
process control blocks:	%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?

- Imagine 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
- That is, 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 do unrelated work

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

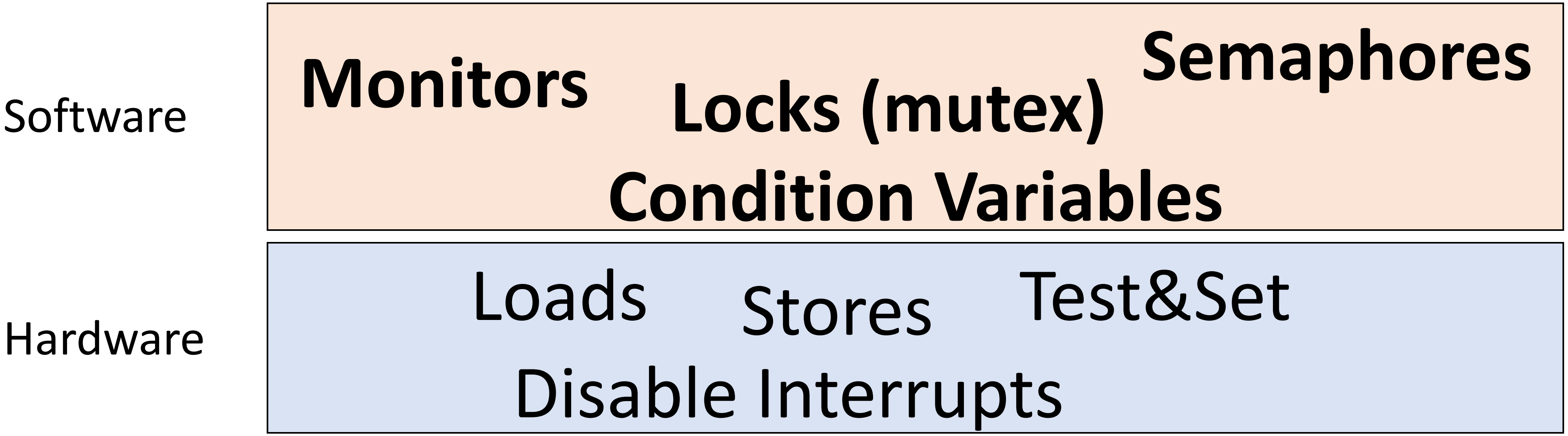
- Critical section:
  - No other thread can change data
- So you are (mostly) ok

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



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*);

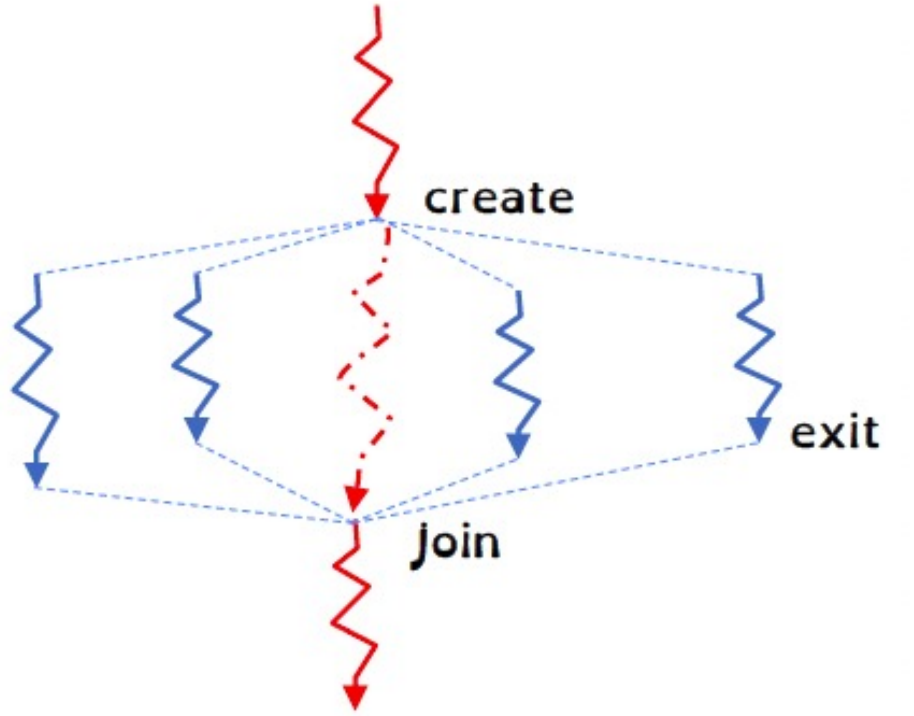
- Terminate calling thread.
- Returns a value via *retval.*

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 exit status 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

```c
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

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

Thread B

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

Deadlocks

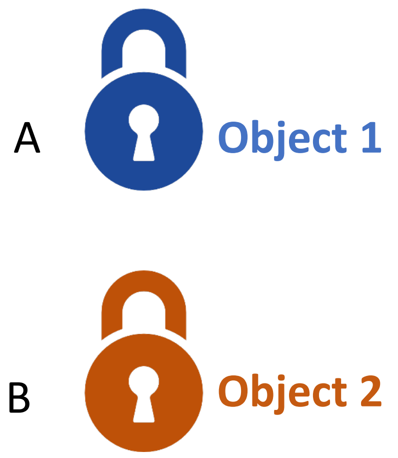
- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help.
- Classic example:

Thread A

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

Thread B

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



Deadlocks

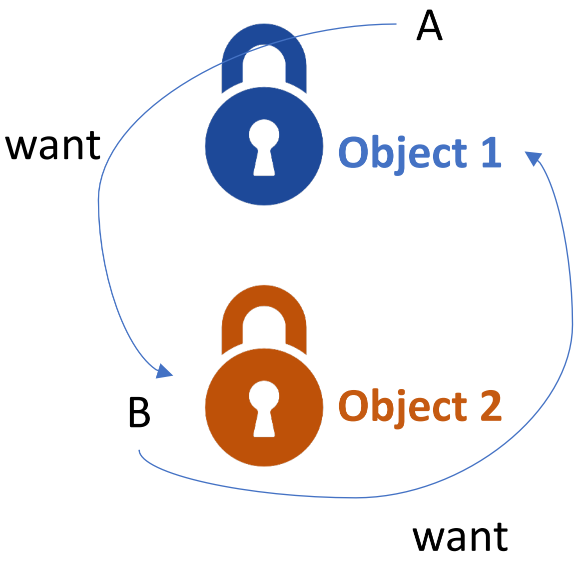
- Threads are stuck waiting for blocked resources and no amount of retry (backoff) will help.
- Classic example:

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

```
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;
}
```

```
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;
}
```

```
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");
}
```

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

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);
```

A waits forever…

Thread B

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

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 , &mutex);
pthread_mutex_unlock(&mutex);
```

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.

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 , &mutex);
pthread_mutex_unlock(&mutex);
```

Example:

If process P running A and B receives and 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, &mutex);
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 unblock mutex.

- `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;
}
```

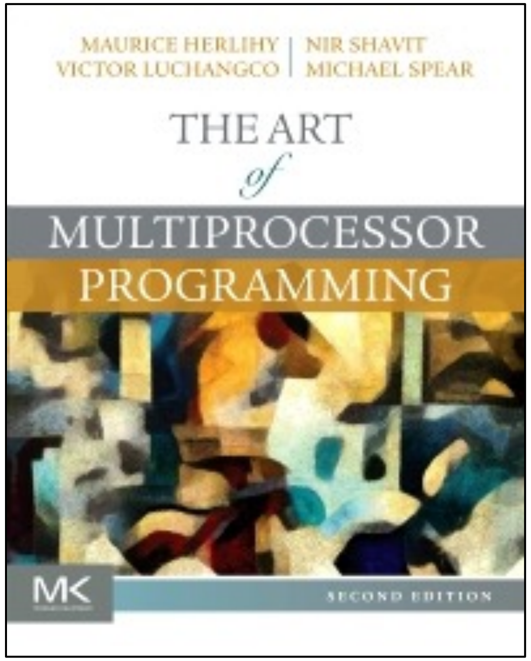
```c
int main (void){

  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;
}
```

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).