Concurrency: An Introduction, Threads

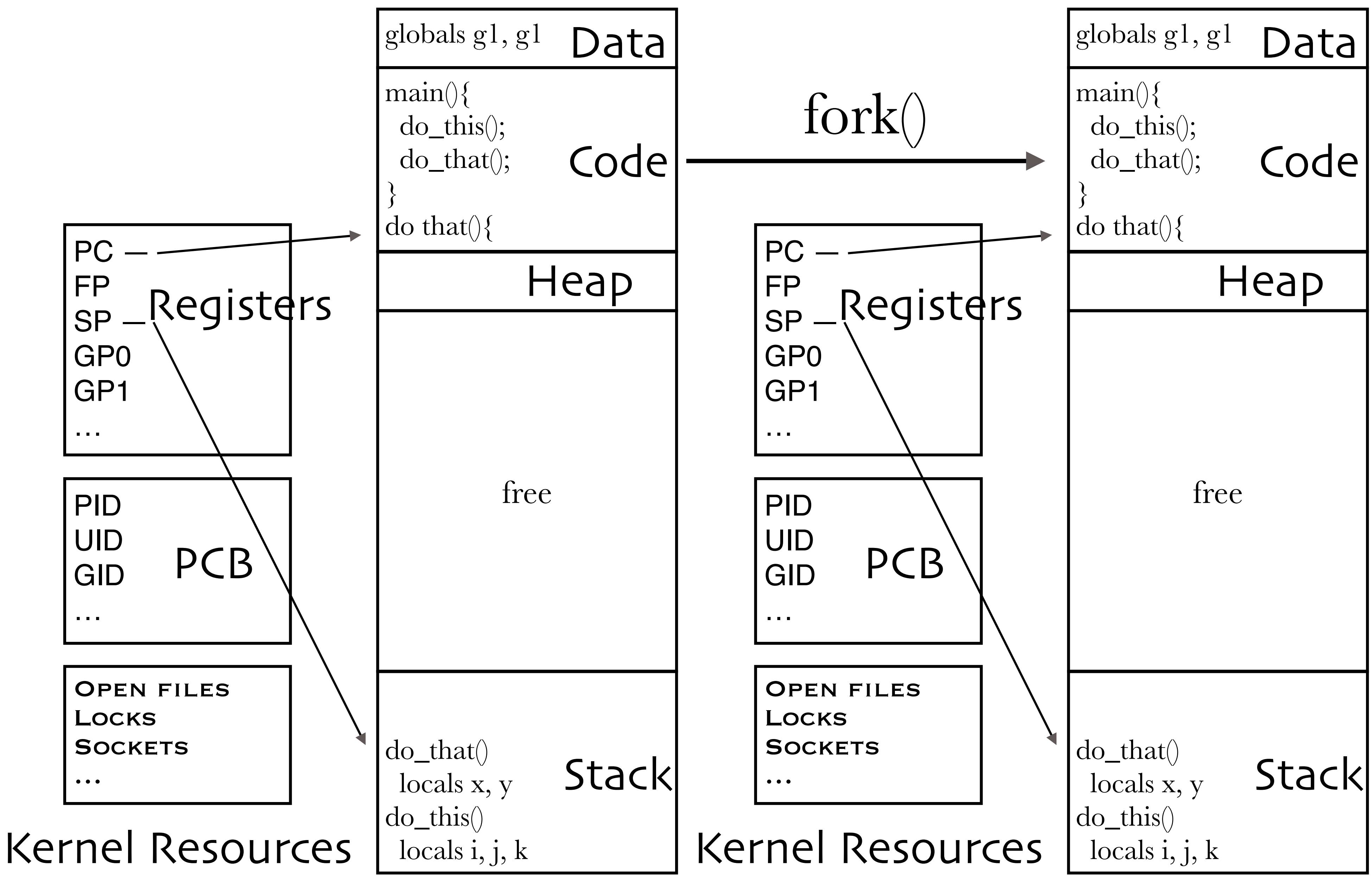
Chapter 26

Threads - The plan

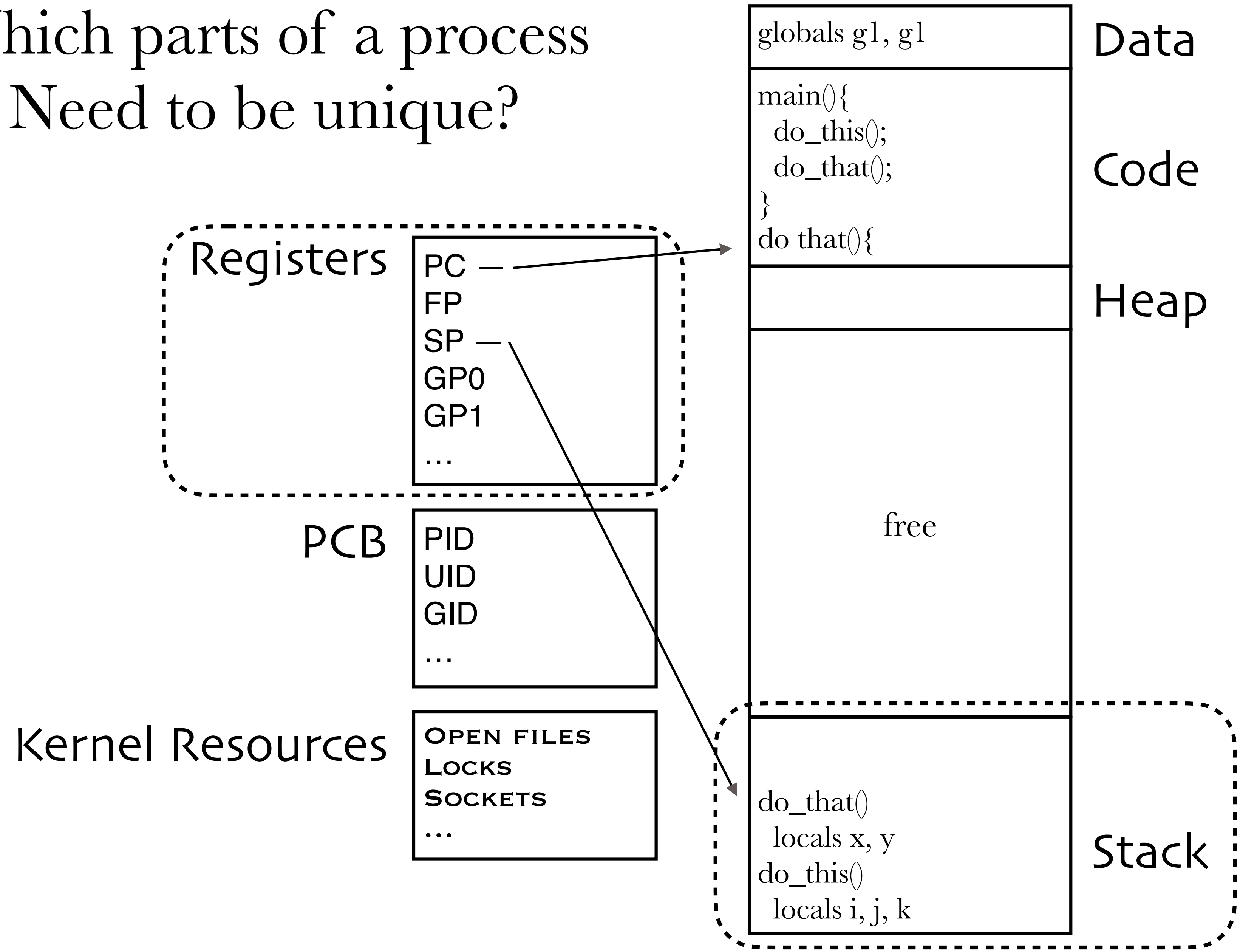
1. What exactly is a thread?
2. The history of threads
3. Why use threads aka. benefits of threads
4. Threading issues
5. Thread creation overview
6. Threading from the kernel's point of view

1. What exactly is a thread

- That abstract something that makes a unique thread of execution within a process possible
- What if we had a fork but the process space was not cloned?
  - i.e. what if two processes shared the same process space, how would that work?


globals g1, g1
Data
main(){
do_this();
do_that();
}
do that(){
Code
Heap
free
do_that()
locals x, y
do_this()
locals i, j, k
Stack
PC
FP
SP
GP0
GP1
...
Registers
PID
UID
GID
...
PCB
Open files
Locks
Sockets
...
Kernel Resources
fork()
globals g1, g1
Data
main(){
do_this();
do_that();
}
do that(){
Code
Heap
free
do_that()
locals x, y
do_this()
locals i, j, k
Stack
PC
FP
SP
GP0
GP1
...
Registers
PID
UID
GID
...
PCB
Open files
Locks
Sockets
...
Kernel Resources

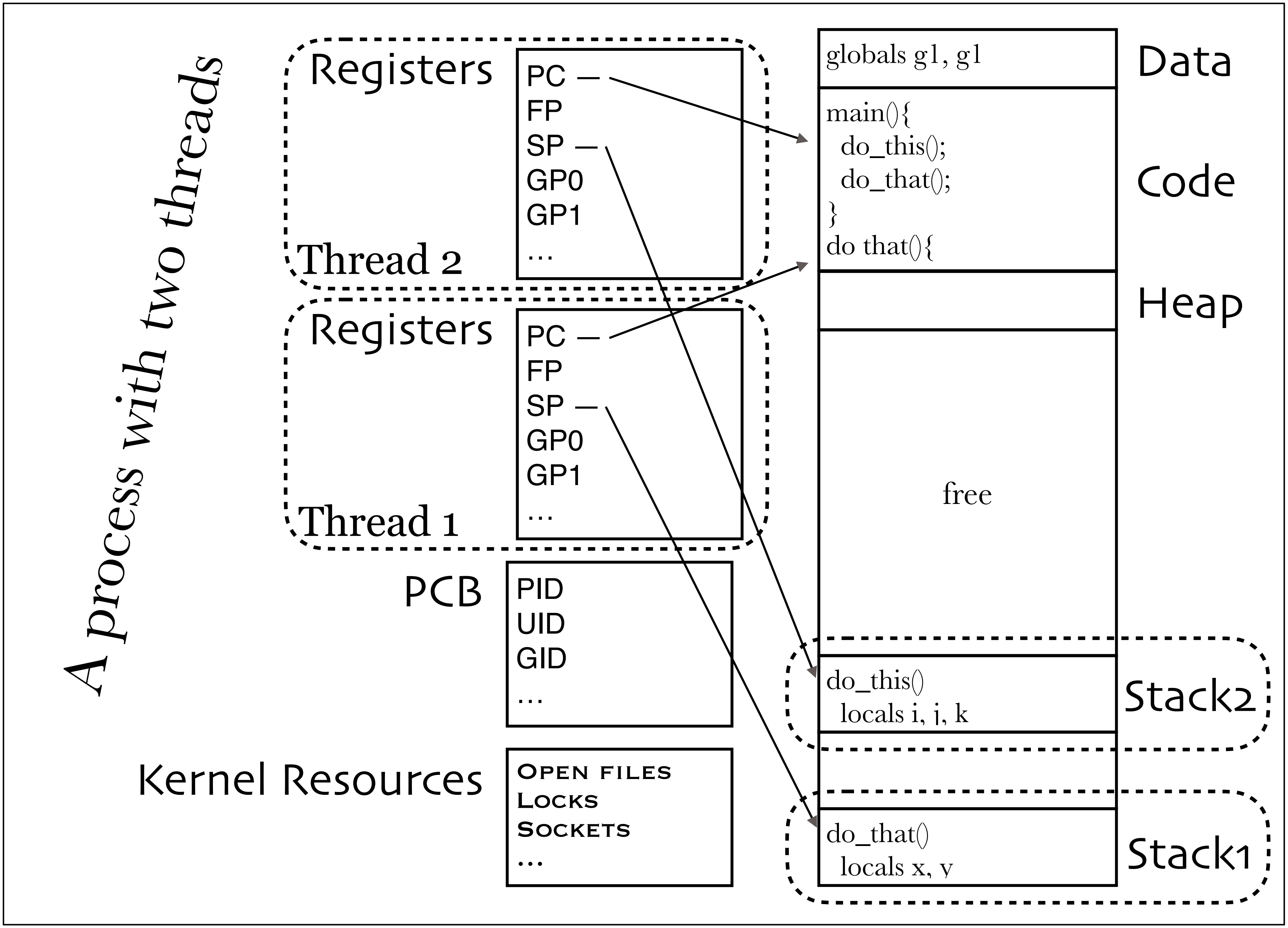
Which parts of a process Need to be unique?



A simple process - with a single thread of execution

1. What exactly is a thread

- That abstract something that makes a unique thread of execution within a process possible
  - The program counter, stack, frame, and the general purpose registers and the stack itself cannot be shared
- The rest of the address space, data, code & heap. The PCB and kernel resources can* be shared
- A multi threaded program thus has more than one thread of execution within the same process



A multithreaded process

What exactly is a thread

- That abstract something that makes a unique thread of execution within a process possible
    - The program counter, stack, frame, and the general purpose registers and the stack itself cannot be shared
- The rest of the address space, data, code & heap. The PCB and kernel resources can* be shared
- A multi threaded program thus has more than one thread of execution within the same process
- You can think of threads as function level parallelism - a process splitting itself into two or more parallel function executions

What exactly is a thread

- Sometimes called a lightweight process - hold that thought
  - You don’t need to clone the process space
  - What abut the stack?
- A thread exists within the concept of a process.
  - A process is said to contain one or more threads of execution
  - A thread cannot exists without a process

PCB vs PCB+TCB

- Process Control Block.
  - Registers
  - PID
  - Process State
  - Address Space
  - Open Files
  - Child Processes
  - Pending Alarms
  - Signals and handlers
  - Account Info

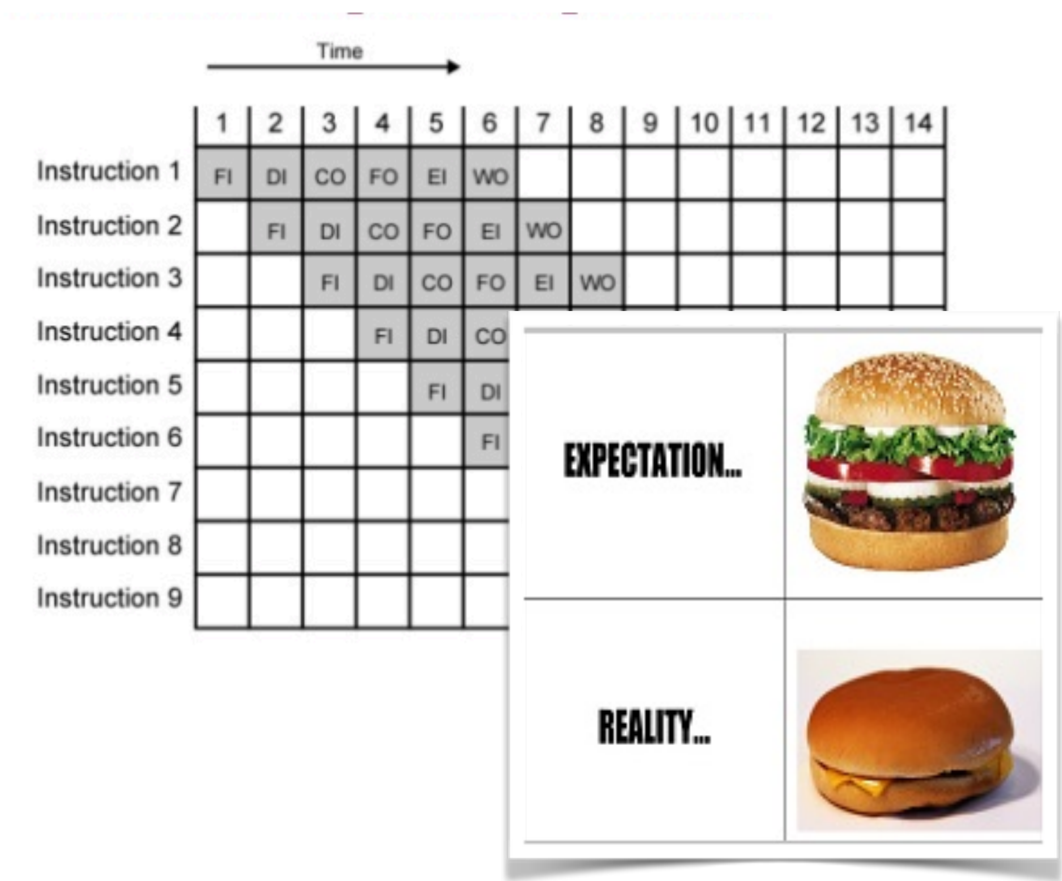
- Process Control Block
  - Address Space
  - Open Files
  - Child Processes
  - Pending Alarms
  - Signals and handlers
  - Account Info
- Thread Control Block
  - Registers
  - Thread ID
  - Thread State

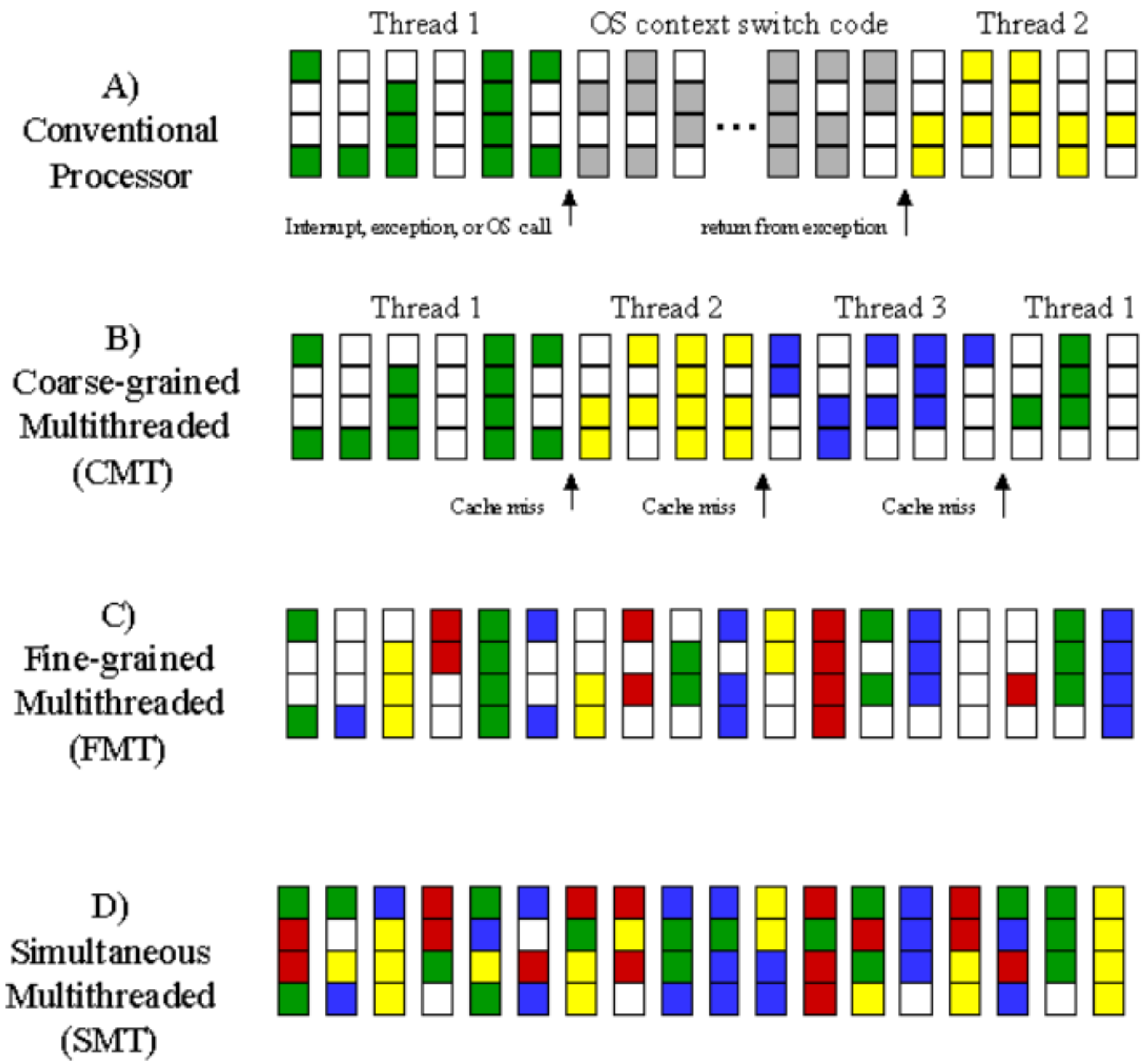

SITUATION:
There is a problem.
Let's use multithreading.
YEAH!
SOON:
SITUATION:
rheTe
are
97
prms.oble

2. Thread History - Pipelines - Detour

- To speed up CPUs, instructions were pipelined. Meaning there were broken up, into stages.

    1. Fetch instructions.
    2. Decode instruction.
    3. Compute operands
    4. Fetch operands
    5. Execute.
    6. Write-back.





https://www.realworldtech.com/alpha-ev8-smt/

MUTITHREADING

- To make pipelines more efficient CPUs were designed to support multithreading
- In multithreading the CPU had multiple sets of registers and was able to switch between 'threads' in varying degrees of effectiveness (previous slide - b,c,d)**
- Multithreading does not not mean context switches were no longer necessary.
  - When a process blocked for I/O for or time quantum expired, the OS trap handler runs on that threads and schedules another process
  - Threads allowed the OS to schedule multiple processes from the ready queue onto the CPU
- To be clear multithreading is a CPU hardware concept to switch rapidly between instructions of various processes **Threads** in this chapters is dealing with functional level parallelism (software) ***

NON BLOCKING PROCESSES

- Simultaneously GUI based computing was increasingly more popular
- GUI based programs often have a unique need for non-blocking processes
- A non blocking process is one that seems to do two or more things simultaneously - or something in the background while blocked for user input.
    - A word processor that seems to spell check in the background
    - A program that seems to autosave in the background
    - A spreadsheet that updates graphs when new data in entered
    - A file browser that knows when a new files appears in the directory without being refreshed.
- If a multithreading CPU can switch between two ‘threads’ of execution, why not split a process into two or more threads of execution. Allow one to run while another blocks. **

3. Why thread! Why!

- With that, we put aside multithreading CPUs and we focus on process threads only.
- Improved responsiveness - If one thread is blocked for I/O another thread can continue to run.
- At about this time, multi CPU and then multicore systems started to arrive on the scene.
  - Threads lent themselves to easily utilize multiple CPUs and cores when executed on a multiprocessor or multicore system.
- Creating a new thread is much less resource intensive then creating a process - remember a thread is a light weight process**
- Since threads exist inside a process, sharing data between threads is SO much easier than sharing data between processes. ***

4. Thread have problems

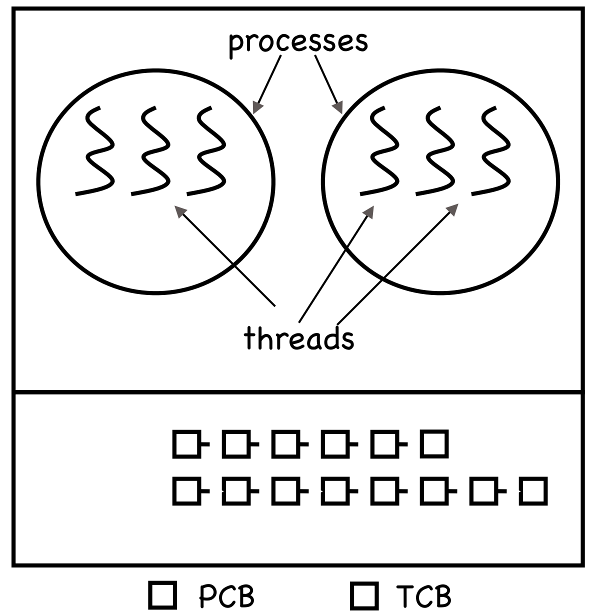
- A thread exists inside a process
  - You can't exec inside a thread - well you can…
  - If one thread generates a trap, or corrupts the heap or another threads stack the process goes…
  - You cannot change the security context of a thread, as this is defined at the process level…
  - When you parallelize function, you no longer control the order they execute in**
  - The biggest issue with threads is shared data between threads - Race Condition.***

5. Thread Creation

- This is clearly explained in chapters 27 and introduced even in chapter 26.
- This is self reading.
- This is also for you to take the time to try this out on your own

Kernel & threads

- While the posix linux thread library has been widely adopted, there is still great differences between systems on how threads are implemented and managed
- On systems that support thread, the kernel schedules threads
- Threads from the ready queue are scheduled onto the CPU



- Linux implements threads as processes - NOT TRUE. It uses a PCB to keep information about a thread.

User and Kernel Threads

- I wish I could make user threads go away
  - It's an old and irrelevant concept.
  - If you wanted to develop and test threads on a system that did not support threads.
  - You could emulate threads in user space.
  - Obviously if any thread blocked, the home process blocked.

LOCKS

CHAPTER 28

BEFORE WE LOOK AT LOCKS

- Why do we even need locks?
  - When multiple threads alter a shared data, i.e. counter, a linked list, a stack, etc. the shared data could be invalid, or incorrect.
  - Linked list - What if two or more threads add an item to the head or tail of the list at the same instance
  - Stack & Queues- what if two or more threads push or pop an time onto/from the stack at the same instance
  - Shared counters - A operation to increment a counter is not atomic, multiple smaller operations

APPENDING TO THE HEAD OF A LINKED LIST

- Simple pseudo code to add a new node to the head of the linked list.

```
prepend( list *head, int item){
  node = list * malloc(sizeof (node));
  node->item = item;
  node->next = head;
  return node;
}
head = prepend(head, i);
```

- What if two thread invoked the below statement on two separate cores at the same instance

(T_1) head = prepend(head, i)

(T_2) head = prepend(head, j)

- The linked list would likely be corrupted!!
- Knowing what is global and local is important.
  - identical local variables between threads are distinct.
- A stack of course could just be a linked list.

INCREMENTING A SHARED COUNTER

- Another even seemingly innocent code

```
increment_global (int times){
  for (i =0; i < times; i++){
    global_counter = global_counter + 1
  }
}

(T1)  increment_global(5000);
(T2)  increment_global(5000);
```

- *Demo

- The issue is caused by something defined as a **Race Condition**.
  - Outcome depends on the order of execution**
  - The two threads could read the counter at the same time…
- As you can see it does not matter that the for loop runs in parallel.
- It only matters that the shared variable is updated in parallel.
- This bit of code that cannot be updated in parallel is called a **critical section.**

IDENTIFYING THE CRITICAL SECTION.

- Consider the following code again

```
prepend( list *head, int *item){
  node = list * malloc(sizeof (node));
  node->item = item;
  node->next = head;
  return node;
}
```

(T_1) head = prepend(head, i);

(T_2) head = prepend(head, j);

- Where is the critical section?
  - In this case the copy of the head is being made when the fiction is being invoked.
  - And the head is being updated when the function returns.
  - The entire function call is a critical section.

NON ATOMIC INSTRUCTIONS

- As can be seen, the issue is that functions or even statements like global_counter++ may appear but are not atomic.
- global_counter++ for example is a collection of assembly instructions:

    <GPRegister1> = global_counter;

    <GPRegister2> = 1;

    <GPRegister1> = <GPRegister1> + <GPRegister2>;

    global_counter = <GPRegister1>;

- The result above is not deterministic
- Depends on the order the parallel threads execute the above set of instructions.

LOCKS TO THE RESCUE (KINDA…)

- Locks allow us to ensure that critical sections are mutually exclusive.
- They allow instructions to execute as if there were atomic (non devisable)

```
lock(lock1);
  global_counter = global_counter + 1;
unlock(lock1);
```

```
lock(lock2);
  head = prepend(head, i);
unlock(lock2);
```

- Prone to bugs when either not used correctly, an incorrect lock is used, or when a lock is not correctly unlocked
- When multiple threads try to acquire locks, there is a significant performance hit*-Demo
- As of present - a necessary evil of parallel coding.

EVALUATING LOCKS

- A lock must provide
  - Mutual exclusion
    - else you don't really have a lock
- A good lock solution must also strive for
  - Fairness/Progress: Threads wanting to access the critical section must be allowed in in some fair way (Worse case, is starvation possible?)
  - Performance: Minimize time overhead added by using the lock

A FAILED ATTEMPT (A BROKEN LOCK)

- Preface - Almost all locks we'll look at, have to be initialized before being used.
- Initialization happens before threads are launched.

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {   // 0->lock is available, 1->held
     mutex->flag = 0;
}

void lock(lock_t *mutex) {
   while (mutex->flag == 1) // TEST the flag
         1; // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it!
}

void unlock(lock_t *mutex) {
   mutex->flag = 0;
}
```

- If two threads read the lock while flag = 0, mutual exclusion will be violated.
- You will also notice the fairness and performance issues (These are hard to fix)

ASIDE: DEKKER & PETERSON'S LOCK

```
int flag[2];  // <- Global Array
int turn;     // <- Global variable.

void init() {
   // indicate you intend to hold the lock w/ 'flag'
   flag[0] = flag[1] = 0;
   // whose turn is it? (thread 0 or 1)
   turn = 0;
}

void lock() {
   // 'self' is the thread ID of caller, 0 or 1.  self will be different for both threads
   flag[self] = 1;
   // make it other thread's turn.  Note: 1-self will the the other thread's ID
   turn = 1 - self;   // GENIUS. Use a race condition to make one thread the winner
   while ((flag[1-self] == 1) && (turn == 1 - self)) ; // spin-wait while it's not your turn
}

void unlock() {
  flag[self] = 0;
}
```

- You used to be able to write a lock strictly using C.
- We exploit a race condition to make one thread a winner when there is contention for the critical section.
- Architectures without strong cache coherency and systems with a **relaxed memory consistency** will breaks this algorithm. *Demo

WHAT IS RELAXED MEMORY CONSISTENCY?

```
int l = 1;
Int x = 0;

void *loop(void *arg){    //Thread 1
   while (l) { };
   if ( x == 0) {
     printf("This system has a relaxed memory consistency model\n");
   }
   return NULL;
}

void *unlock(void *arg){.  // Thread 2
   x = 1;
   l = 0;
}
```

- The Spare and Power PC architecture could have updates to memory that could arrive in an out of sequence order.

ATOMIC INSTRUCTIONS

- Locks can be built using various CPU instructions which are guaranteed to be atomic and consistent across cores.

Test-And-Set

- Atomic exchange**
- Pseudo code is as follows

```
int TestAndSet(int *old_ptr, int new) {
   int old = *old_ptr; // fetch old value at
   old_ptr *old_ptr = new; // store 'new' into  old_ptr
   return old;
}
```

- Yes, it is as simple as it seems

Building a lock from test-and-set.

```
typedef struct __lock_t {
   int flag;
} lock_t;

void init(lock_t *lock) { lock->flag = 0; }

void lock(lock_t *lock) {
   while (TestAndSet(&lock->flag, 1) == 1) {
      //spin
   }
}

void unlock(lock_t *lock) { lock->flag = 0; }
```

ATOMIC INSTRUCTIONS

- Locks can be built using various CPU instructions which are guaranteed to be atomic and consistent across cores.

Compare-And-Swap

- Atomic compare and Swap
- Pseudo code is as follows

```
int CompareAndSwap(int *ptr, int expected, int new) {
   int original = *ptr;
   if (original == expected)
      *ptr = new;
  return original;
}
```

- Similar to Test-and-set

Building a lock from compare-and-swap.

```
typedef struct __lock_t {
   int flag;
} lock_t;

void init(lock_t *lock) { lock->flag = 0; }

void lock(lock_t *lock) {
   while (CompareAndSwap(&lock->flag, 0,1) == 1{
     //spin
    }
}

void unlock(lock_t *lock) { lock->flag = 0; }
```

ENSURING PROGRESS

- The two locks we have seen so far don't progress and use spin locks which have poor performance.

Fetch-And-Add

- Atomic Increment**
- Pseudo code is as follows

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- Similar to Test-and-set, except we increment ptr instead of setting a value

Building a lock from compare-and-swap.

```
typedef struct __lock_t {
   int ticket;
   int turn;
} lock_t;

void lock_init(lock_t *lock) {
   lock->ticket = lock->turn  = 0
}

void lock(lock_t *lock) {
   int myturn = FetchAndAdd(&lock->ticket);
   while (lock->turn != myturn)
     // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

BASIC LOCKS - IN SUMMARY

- We have seen that various atomic operations can used to build locks.
- The earlier basic locks suffer from
  - Lack of progress (possible starvation)
  - Poor performance - Spin locks
- Fetch-And-Add does bring a ‘turn’ which ensures progress
  - It does not address the poor performance
- There are solutions which utilize Queues to park waiting threads - 28.14

Condition Variables (Process Synchronization)

Chapter 30

First a bit of recap…

Race Conditions

Sequence of instructions in T1 & T2 are determined. However sequence of instructions between T1 & T2 are not!

int counter = 10

T_1: Counter = Counter + 1

T_2: Counter = Counter -1

```
cout << counter << endl
> 10
```

1. T_1: GP Register $_{T1}$ = counter
2. T_1: GP Register $_{T1}$ = GP Register $_{T1}$ + 1
3. T_1: counter = GP Register $_{T1}$
4. T_2: GP Register $_{T2}$ = counter
5. T_2: GP Register $_{T2}$ = GP Register $_{T2}$ - 1
6. T_2: counter = GP Register $_{T2}$

One more thing….

Mutual Exclusion solution requirements…

int counter = 10

T_1: Counter = Counter + 1

T_2: Counter = Counter -1

1. T_1: GP Register $_{T1}$ = counter
2. T_2: GP Register $_{T2}$ = counter
3. T_1: GP Register $_{T1}$ = GP Register $_{T1}$ + 1
4. T_1: counter = GP Register $_{T1}$
5. T_2: GP Register $_{T2}$ = GP Register $_{T2}$ - 1
6. T_2: counter = GP Register $_{T2}$

cout << counter << endl

> 9

First a bit of recap…

Race Conditions

int counter = 10

T_1: Counter = Counter + 1

T_2: Counter = Counter -1

1. T_1: GP Register $_{T1}$ = counter
2. T_1: GP Register $_{T1}$ = GP Register $_{T1}$ + 1
3. T_2: GP Register $_{T2}$ = counter
4. T_2: GP Register $_{T2}$ = GP Register $_{T2}$ - 1
5. T_2: counter = GP Register $_{T2}$
6. T_1: counter = GP Register $_{T1}$

```
cout << counter << endl
> 11
```

First a bit of recap…

Race Conditions

int counter = 10

T_1: Counter = Counter + 1

T_2: Counter = Counter -1

1. T_1: GP Register $_{T1}$ = counter
2. T_1: GP Register $_{T1}$ = GP Register $_{T1}$ + 1
3. T_2: GP Register $_{T2}$ = counter
4. T_2: counter = GP Register $_{T2}$
5. T_2: GP Register $_{T2}$ = GP Register $_{T2}$ - 1
6. T_1: counter = GP Register $_{T1}$

** This is not a possible sequence of interactions **

One more thing…

A critical section solution must satisfy.

In our textbook

1. Mutual Exclusion
2. Fairness (no starvation)
3. Performance

In pretty much any other textbook

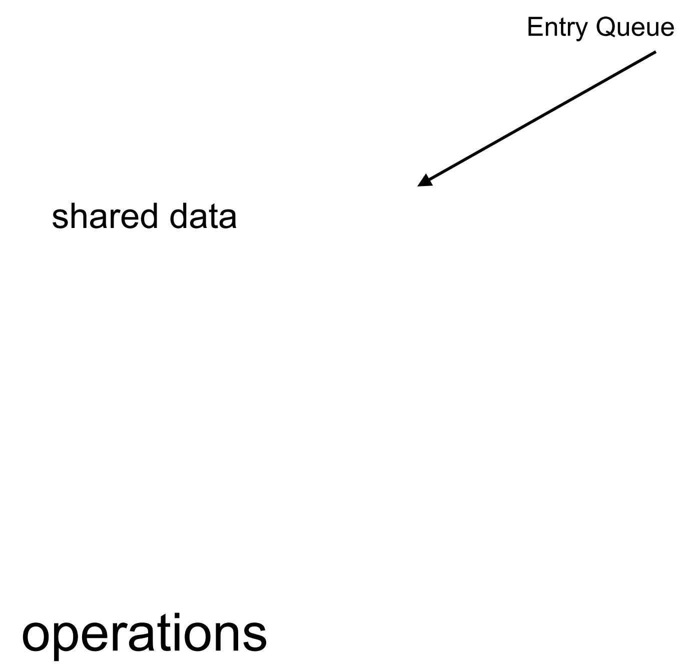
1. Mutual Exclusion
2. Progress -

   - only threads that are trying to enter the critical section should participate in who gets to enter the critical section *

3. Bound Waiting (no starvation)

Monitors (the missing topic)

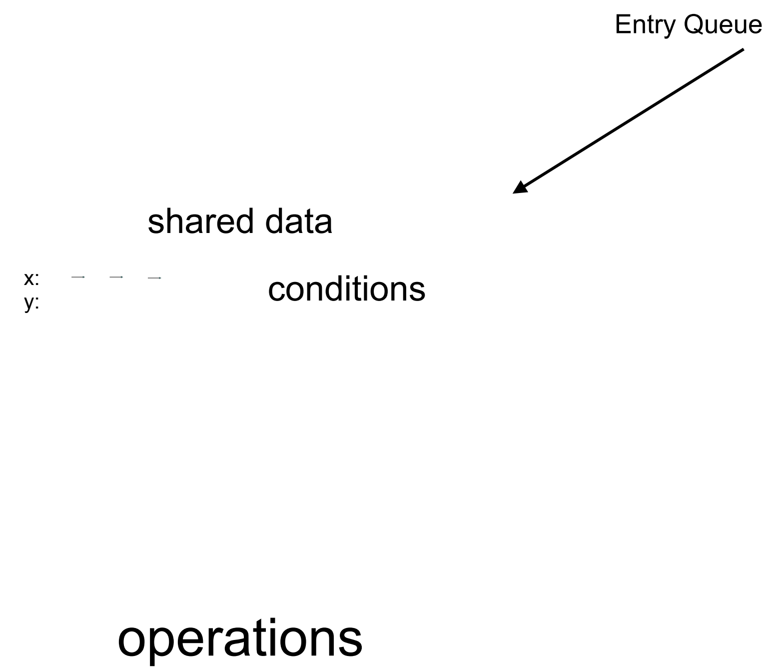
- Monitors are a synchronization construct
- Monitors help illustrate the concept of a condition variable
  - Each monitor protects one or more shared data
  - Only one thread is allowed to execute inside at a time, i.e code inside the monitor is executed by one thread at a time.
  - lock(monitor) and unlock(monitor) to enter/exit.
  - Each monitor has an entry queue (i.e. No spin lock or starvation).
    - i.e. it ensures mutual exclusion, progress*, fairness and performance
  - So far they look just like a fancy lock (with a Queue)



Schematic view of a monitor

Monitors & Synchronization

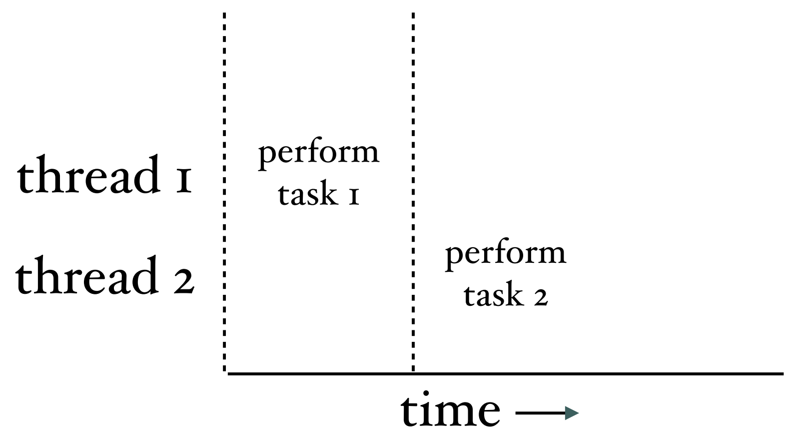
- What Monitors bring to the party is the ability for threads to synchronize:
  - To release the monitor and wait() for a condition to occur
  - To signal() other threads that a condition has occurred.
  - A monitor can have any number of condition variables.
    - Condition Variable X, Y
  - The only operations you can do on a condition as wait and signal
    - wait(x) — pause the current thread and place on queue x, release the monitor
    - signal(x) — wake exactly one thread (when we leave monitor). If the queue is empty, the signal is ignored.



Schematic view of a monitor

The simplest type of synchronization

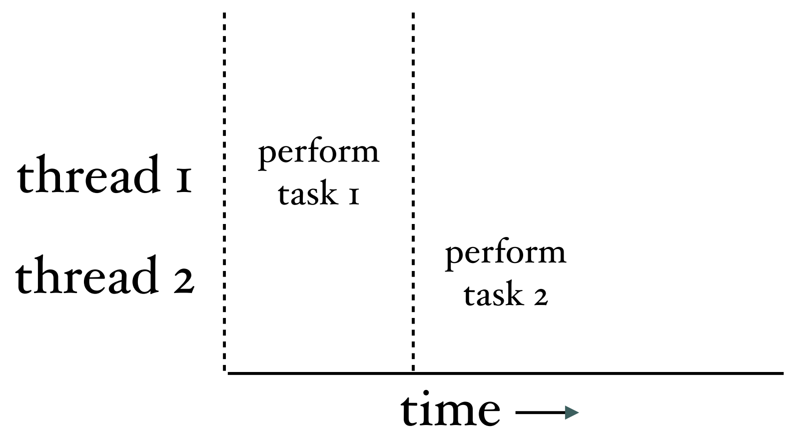
```
void task1(){
  do_something_first;
   signal(x);
}

void task2(){
   wait(x);
    do_something_else;
}
```



- Works if thread2 runs before thread1
- Does not work if thread1 runs before thread2. The signal will be lost. The wait will wait forever.

The simplest type of synchronization

```
void task1(){
  do_something_first;
  t1_done = 1;
  signal(x);
}

void task2(){
   if (!t1_done) wait(x);
   do_something_else;
}
```

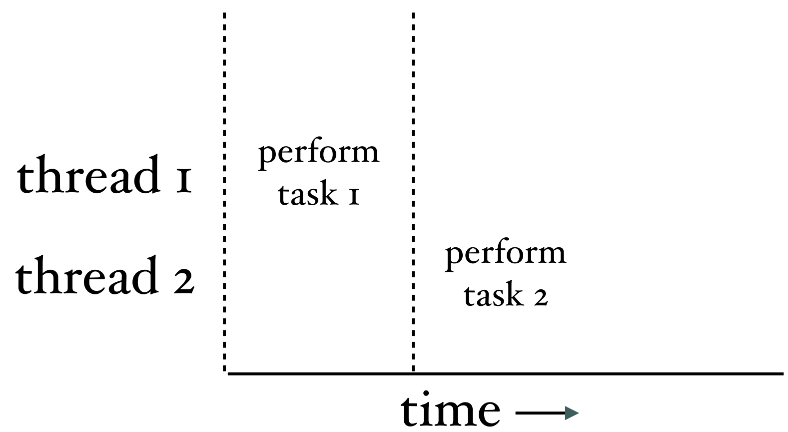


- We are almost there.
- What if the signal() occurs after the if() but before the wait()

The simplest type of synchronization

```
void task1(){
  do_something_first;
  lock(m);
  t1_done = 1;
  signal(x);
  unlock(m)
}
```

```
void task2(){
   lock(m)
   if (! t1_done)
      wait(x);
   unlock(m)
   do_something_else;
}
```



- Putting the signal outside the monitor can lead to race conditions
- The monitor is locked again after thread 2 is resumed, so the the lock must be released.
- The lock prevents the spurious signal() that we saw in the last slide.

Pthread and monitors

* Posix Threads provide the primitives for you to build a monitor.
* Pthread locks in particular can be used both as a basic locks or as a monitor.
* But the same primitives can be used in different and sometimes undefined ways.

```
pthread_mutex_t m = mutex_init;     // lock.   /* NOTE: pseudo code*/
pthread_cond_t c = cond_init;      // condition variable
// ————————————————————
pthread_mutex_lock(m); // in effect this is a monitor.
pthread_cond_wait(c, m); // wait on condition variable c and release lock m.
pthread_cond_signal(c); // Signal condition variable
pthread_unlock(m); // release lock/leave monitor
```

Pthread and monitors

- But the same primitives can be used in different and sometimes undefined ways.

```
pthread_mutex_t m1 = mutex_init; // lock
pthread_cond_t c = cond_init;      // condition var
// ——————————————————
thread_1(){
   pthread_mutex_lock(m1);
   pthread_cond_wait(c, m1);
   pthread_unlock(m1);
}
```

```
thread_2(){
    pthread_cond_signal(c);
}
```

Thread1 seems to implement a monitor

Thread 2 does not.. T1 might wait forever

Pthread and monitors

* But the same primitives can be used in different and sometimes undefined ways.

```
pthread_mutex_t m1 = mutex_init; // lock
pthread_cond_t c = cond_init;     // condition var
// ——————————————————
thread_1(){
   pthread_mutex_lock(m1);
   pthread_cond_wait(c, m1);
   pthread_unlock(m1);
}
```

```
thread_2(){
   pthread_mutex_lock(m2);
   pthread_cond_wait(c, m2);
   pthread_unlock(m2);
}
```

Valid code, but this is not a monitor

You, the programmer need to limit a condition variable to a lock

Pthread and monitors

* But the same primitives can be used in different and sometimes undefined ways.

```
pthread_mutex_t m1 = mutex_init; // lock
pthread_mutex_t m2 = mutex_init; // lock
pthread_cond_t c = cond_init;      // condition var
// ——————————————————–
thread_1(){
   pthread_mutex_lock(m1);
   pthread_cond_wait(c, m1);
   pthread_unlock(m1);
}
```

```
thread_2(){
    pthread_mutex_lock(m2);
    pthread_cond_wait(c, m1);
    pthread_unlock(m2);
}
```

No way you know what you're doing!!

Pthread and monitors

* But the same primitives can be used in different and sometimes undefined ways.

```
pthread_mutex_t m1 = mutex_init; // lock
pthread_cond_t c = cond_init;     // condition var
pthread_cond_t d = cond_init;     // condition var
// ————————————————
thread_1(){
   pthread_mutex_lock(m1);
   pthread_cond_wait(c, m1);
   pthread_signal(d);
   pthread_unlock(m1);
}
```

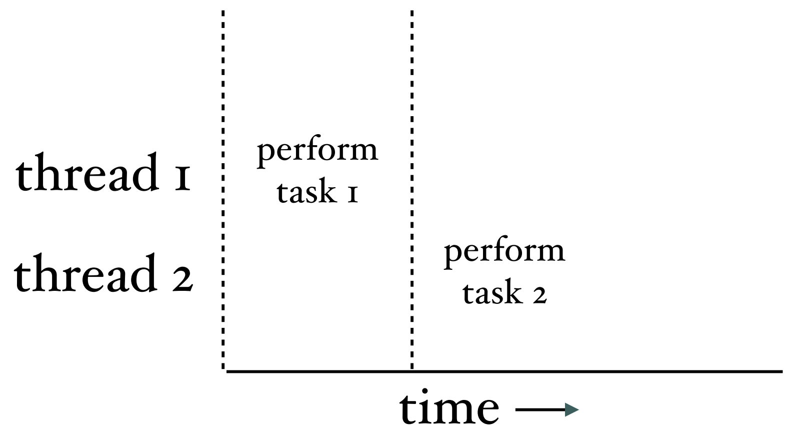
```
thread_2(){
   pthread_mutex_lock(m1);
   pthread_cond_wait(d, m1);
   phread_cond_signal(c);
   pthread_unlock(m1);
}
```

This is a valid monitor,
but this code will deadlock

Pthread and monitors

```
void task1(){
  do_something_first;
  lock(m);
  t1_done = 1;
  signal(x);
  unlock(m)
}
```

```
void task2(){
  lock(m);
  while (! t1_done) **
    wait(x, m);
  unlock(m)
  do_something_else;
}
```

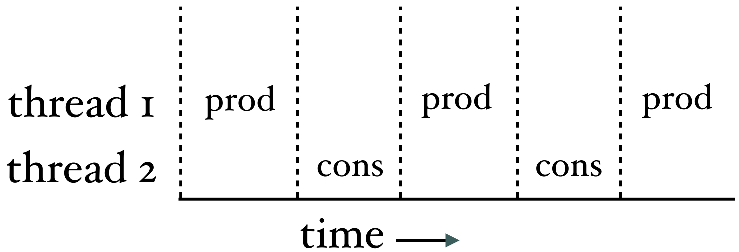


Time 10:30 video part 3 - I say - we skip wait if t1_done has not occurred.
Wrong - we skip wait if t1_done has occurred

Producer Consumer Problem

- A very common type of shared variable and synchronization problem in a multi threaded (parallel) programming.
- Core case - a single value that is being written to by one thread and being consumed by another thread.
- Variations include
  - Multiple producers and consumers
  - What if we have a buffer (usually a circular buffer of limited size)
  - Permutations of the above

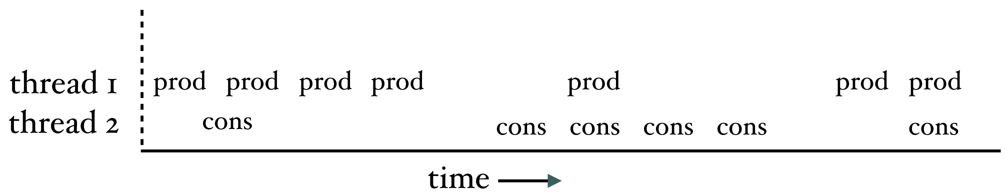
Producer Consumer Single buffer




```
void *consumer(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
      Pthread_mutex_lock(&mutex);
      if (count == 0)**
         Pthread_cond_wait(&cond, &mutex);
      int tmp = buffer;
      count—;
      Pthread_cond_signal(&cond);
      Pthread_mutex_unlock(&mutex);
      printf("%d\n", tmp);
   }
}
```

```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
int count = 0;
int buffer;

void *producer(void *arg) {
   int i;
   for ( i = 0; i < loops; i++){
      Pthread_mutex_lock(&mutex);
      if (count == 1)
         Pthread_cond_wait(&cond, &mutex);
      buffer = i or something;
      count++;
      Pthread_cond_signal(&cond);
      Pthread_mutex_unlock(&mutex);
   }
}
```

Producer Consumer Bound buffer




```
int loops; // must initialize somewhere...
cond_t not_empty, not_full;
mutex_t mutex;
int count = 0;
int buffer[SIZE];
```

I miss-speak in the videos, this solution cannot be used for multiple producers/ consumers. This is a single producer/ consumer solution since each thread will run loops times.

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        if (count == 0)
            Pthread_cond_wait(&not_empty, &mutex);
        int tmp = get();
        count = count - 1 ;
        Pthread_cond_signal(&not_full);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

```
void *producer(void *arg) {
    int i;
    for ( i = 0; i < loops; i++){
        Pthread_mutex_lock(&mutex);
        if (count == SIZE)
            Pthread_cond_wait(&not_full, &mutex);
        put(something);
        count++;
        Pthread_cond_signal(&not_empty);
        Pthread_mutex_unlock(&mutex);
    }
}
```

Semaphores
(Process Synchronization - another way)

Chapter 31

What are semaphores

- Semaphores are a more general purpose synchronization tool
  - i.e. They are be used in threads, parent/child processes or between unrelated processes**
  - As we'll see they are be used both as locks and to synchronization processes
- Word of caution if you've kinda be tuned out
  - It is super super easy to confuse semaphores and condition variables
  - They both use signal() and wait(). POSIX Semaphores use post() & wait()
  - Under the right conditions it might even seem like they have similar functionality.
  - They are very different in implementation and in functionality.

Definition of semaphores in pseudo code

```
sem_t  S;
sem_init (&S, 0, 1);

sem_wait(sem_t *S){
   // decrement value of semaphore
   // if value of semaphore is negative
        // Queue
  //else
       // continue
}
```

```
sem_post(sem_t *S){
   // increment value of semaphore
   // if value of semaphore is >= zero
      // wake first waiting process
}
```

- Operations on semaphores are performed indivisibly i.e. they are atomic instructions. i.e. When one thread or process is modifying the semaphore, no other process can be modifying the semaphore

Semaphores vs Condition Variables

* Condition variable has a queue, if queue is empty, the signal is lost
    * Associated with a lock which has to be acquired before the wait()
* Semaphores, also has a queue, but it's associated with an integer, not a condition
    * It is not associated with any lock
    * It is _not_ associated with this abstract things you call a condition variables and you decide what that condition means.
    * It is associated with a real integer.
        * If said integer is less than zero - stop! Get in line.
        * If said integer is greater than zero continue.

Writing a basic lock with a semaphore

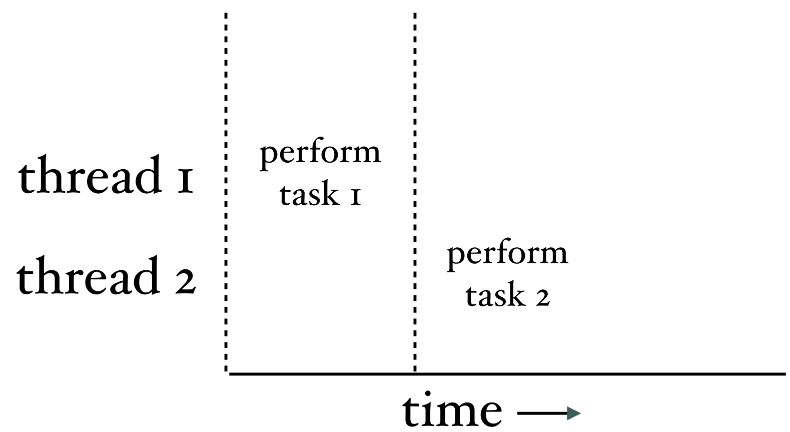
- Basic Locking semaphores are also called binary semaphores.

```
sem_t  M;
sem_init (&M, 0, ??);

sem_wait(&M);
// critical section
sem_post(&M);
```

Semaphores for Ordering

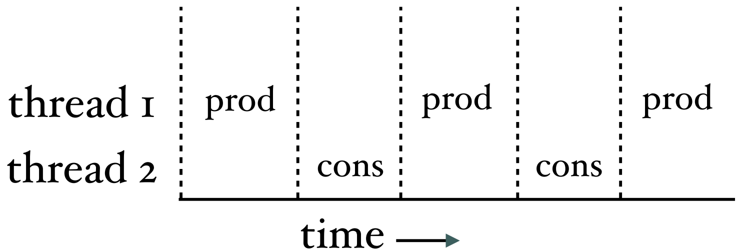
- As stated previously, semaphores can also be used to synchronize thread ordering

```
sem_t  S;
sem_init (&S, 0, ??);

void *task_1(){
    do_something;
    sem_post(&s);
}

void *task_2(){
    sem_wait(&s);
    do_after_thing;
}
```



If you go back and review slides from condition variables, you will see that you do not have to worry about task_1 sending the signal before task_2 does a wait( ).

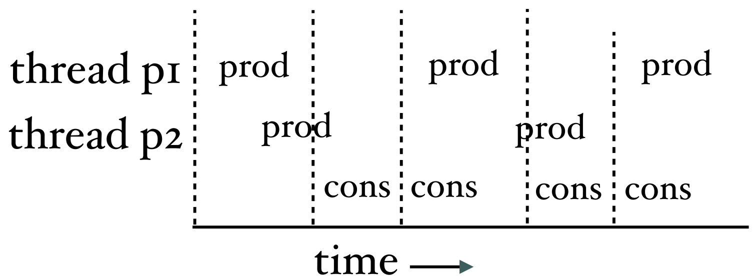
Producer Consumer Single buffer




```
void *consumer(void *arg) {
    for (i = 0; i < loops; i++) {
      sem_wait(&full);
      int tmp = buffer;
      sem_post(&empty);
     printf("%d\n", tmp);
  }
}
```

```
int loops; // must initialize somewhere...
sem_t empty, full;
int buffer;

main(){
  sem_init(&empty, 0, ??) **;
  sem_init(&full, 0, ??);
  //launch producer & consumer threads
}

void *producer(void *arg) {
  for ( i = 0; i < loops; i++){
      sem_wait(empty);
       buffer = i or something;
      sem_post(&full);
  }
}
```

(2)Producer, Consumer, bound buffer

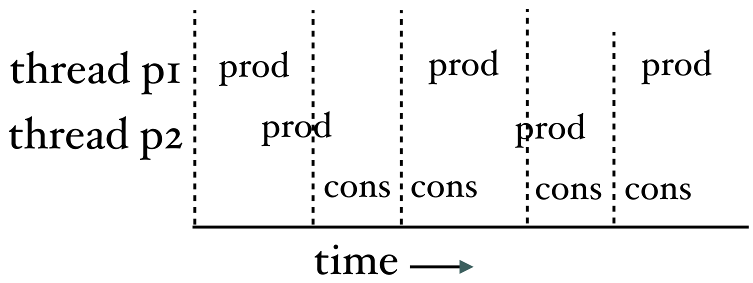



```
void put(int value){
    buffer[fill] = value
    fill = (fill + 1) % MAX;
}
```

- As you can see, semaphores can be initialized to number greater than 0 or 1 and this has interesting uses.

```
int loops; // must initialize somewhere...
sem_t empty, full;
int buffer[MAX];

main(){
  sem_init(&empty, 0,  ??);
  sem_init(&full, 0, ??);
  //launch 2 x producer & consumer
threads
}

void *producer(void *arg) {
  for ( i = 0; i < loops/2; i++){
      sem_wait(empty);
      put(i or something);
      sem_post(&full);
  }
}
```

(2)Producer, Consumer, bound buffer




```
void put(int value){
    buffer[fill] = value
    fill = (fill + 1) % MAX;
}
```

- Back to square one if we have simultaneous producers running though - a race condition in updating fill.

```
int loops; // must initialize somewhere...
sem_t empty, full;
int buffer[MAX];

main(){
  sem_init(&empty, 0,  ??);
  sem_init(&full, 0, ??);
  //launch 2 x producer & consumer
threads
}

void *producer(void *arg) {
  for ( i = 0; i < loops/2; i++){
      sem_wait(empty);
      put(i or something);
      sem_post(&full);
  }
}
```

Semaphores and deadlocks

```
producer()
    sem_wait(&full);*
    buffer = something
    sem_post(&empty);
```

Deadlocked

```
consumer()
     sem_wait(&full);
    buffer = something
    sem_post(&empty);
```

```
producer()
    sem_wait(&empty);*
    buffer = something
    sem_post(&full);
```

Deadlocked

```
consumer()
    sem_post(&full);
    buffer = something
    sem_wait(&empty);
```

```
producer()
    sem_wait(&empty);*
    buffer = something
    sem_post(&full);

consumer()
    sem_wait(&full);
    buffer = something
    sem_post(&empty);
```

Consumer reads empty buffer

```
main()
    sem_init(&empty, 0, 0)
    sem_init(&full, 0, 1)
```

Reader-Writer Lock

- Imagine a _typical_ word/excel or a system binary file
- You can have any number of readers
- But only one writer and no readers when writing

```
acquire_writelock(){
    sem_wait(&filelock);
}
```

```
acquire_readlock(){
    sem_wait(&lock);
    readers++;
    If (readers == 1)
        sem_wait(&filelock)
    sem_post(&lock)
}
```

In Summary

- There are other problems as discussed in the book - dining philosophers
- Semaphores are the the Swiss Army knife of process synchronization
  - Both a lock and a synchronization primitive
  - Works with threads, parent/child processes or completely unrelated process
- Not without its problems
  - To general purpose? Easier to get yourself in trouble? Deprecated on MacOS...
  - It is certainly slower that pthread locks and pthread condition variables.