P2L2 Threads and Concurrency

Goal

1. What are threads?
2. How is threads different from processes?
3. What data structures are used to implement and manage threads

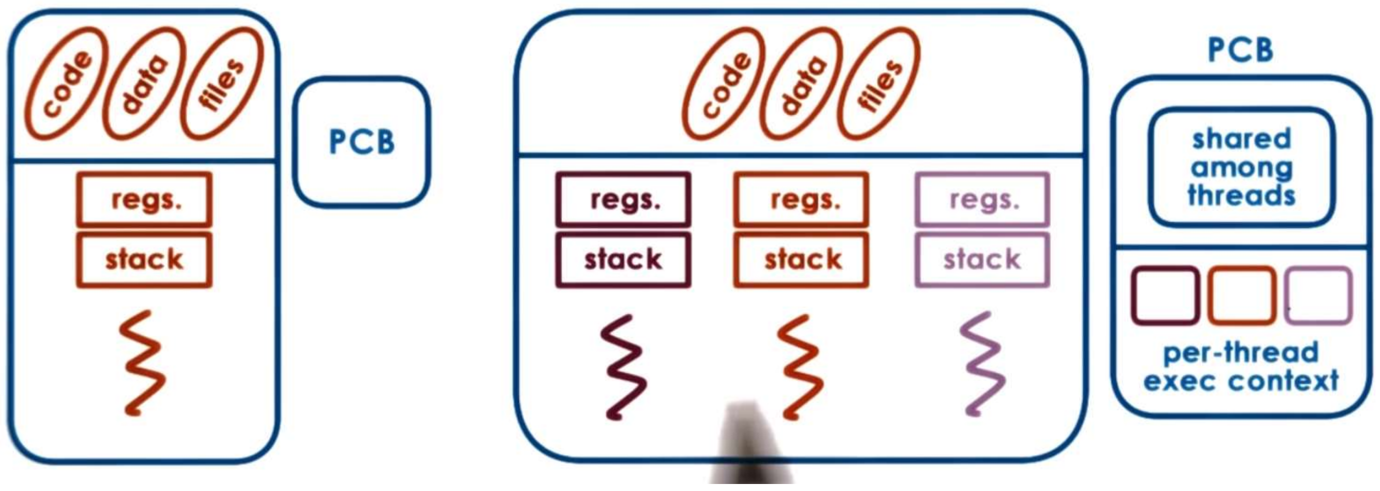
Properties of Thread that resembles toy shop workers

- active entity: execute unit of a process
- work simultaneously with others
- require coordination over common resources

1 Basics about Threading

1.1 Process vs. Thread

- A process is represented with its virtual address space, all its information is stored ina PCB.
- Thread represent multiple independent execution contexts.
- The contexts share the same virtual address space.
- But they will potentially access different portions of the address space and execute different instructions.
- Each thread have its own CP, stack pointer, stack and thread-specific register values.
- Separate data structures to represent the per-thread information.
- Process' PCB should include all the thread information too.



1.2 Benefits of Multi-threading

> One-liner: More efficient and incur lower overhead in context switching than multiprocessing.

- Parallelization --speed up execution
  - Each thread can run on different processor
  - Each thread can process different part of the input data

- Specialization – e.g. different thread handles different requests on web server
    - Threads can execute different portions of the program
    - Partition the task for different threads to execute.
    - Give higher priority to threads that handle more important tasks.
    - Hotter cache — each thread running on a differnet processor has access to its own processor cache.
- Lower Overhead
    - All threads using one shared virtual memory space–> less memory memory requirement.
    - No user/kernel mode switching IPC cost
    - Easier inter-thread communication via shared variables.

1.3 Are threads useful on single CPU?

What if # of threads > # of CPUs?

> One-liner: Hide latency.

- Sometimes a thread may execute a command that takes some time to process or need to wait for an event to happen, such as disk I'O or waiting for a web server somewhere else to send back the data you requested. Without multi-threading, they just wait there and waste the precious CPU time.
- Instead of waiting, if the wait/idle time is larger than 2* (thread context switching time) then it makes sense to switch to another thread and let the CPU do some useful work while the thread with latency is waiting.
- Context switching between threads take less time than that of processes:
    - Threads lives in the same virtual address space.
    - No need to create new virtual to physical address mapping.

1.4 Benefits of multi-threading to apps and the OS

- Allow OS to support multiple executiong contexts
- Utilize multi-CPUs
- Threads can be allocated to work on behalf of apps
- Or work on behalf of OS level services

1.5 Basic Thread Mechanisms

1.5.1 What do we need to support threads?

- thread data structure
    - identify threads, keep track of resource usage…
- mechanisms to create and manage thread
- mechanisms to safely coordinate among threads running concurrently in the same address space
    - Prevent threads from overwriting each other's input/results
    - For one thread to wait for results produced by another thread

1.5.2 Issues with threads and concurrency

- For multiprocessing isolation is not a problem ----they all live in differnt universe (virtual memeory space)
- Different threads t1 and t1 running concurrently can both legally access the same physical memory and make changes to it.

- Inconsistency/ racing issues:
  - t1 could read a variable while t2 is modifying it
  - or they both try to update the data at the same time

1.5.3 Synchronization Mechanisms

- Mutual Exclution (**mutex**)
  - only one thread at a time is allowed to perform an operation
  - Other threads have to wait till the one thread is done
- Condition Variable --inter-thread coordination
  - Wait on other thread to finish something
  - Specify what a thread is waiting for
  - e.g. A shipping thread should wait for all orders are completed before shipping.
  - It doesn't make sense to busy-wait/ repeatedly check if other threads are done
  - Wait til explicitly notified that all the threads it's waiting on are done.

2. Using Threads

2.1 Thread Creation

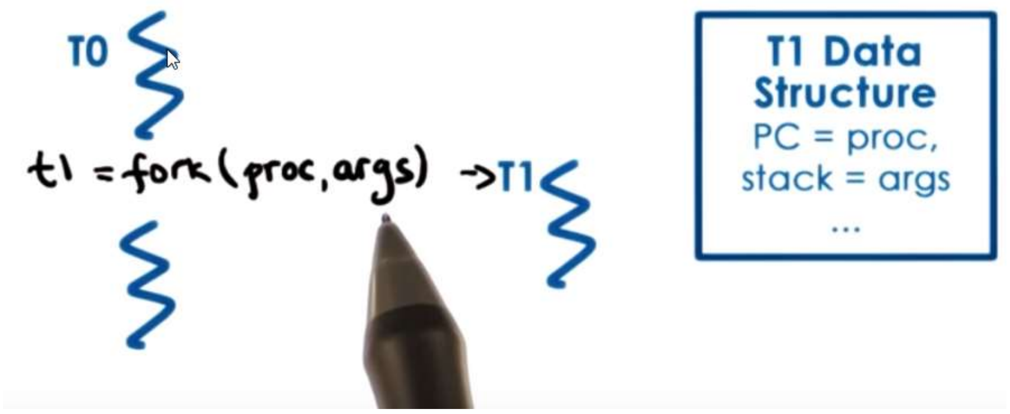
Thread data type:

- contains all info that can describe a thread
- Specific to one thread
- Includes (e.g.):
  - thread ID
  - Program Counter
  - Stack Pointer
  - register values
  - stack
  - attribtues

Thread Creation

Fork(proc, args)–proposed by Birrell

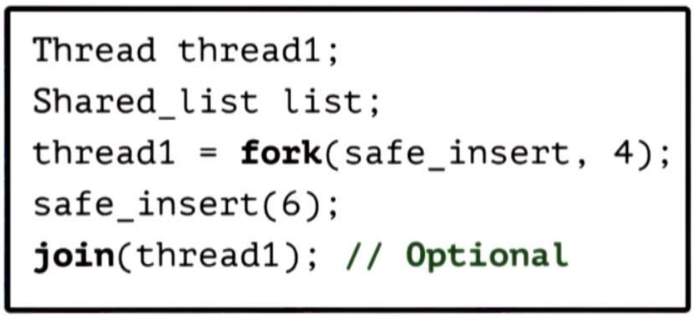
- create a thread
- not UNIX fork() for process
- execute proc with args as arguments
- Result in two threads executing concurrently



Join(thread)

- parent thread can join its child thread
- block until the thread completes
- return the result of child's computation to parent
- terminate child thread
- free memories allocated for child thread execution

Example:



```
Thread thread1;
Shared_list list;
thread1 = fork(safe_insert, 4);
safe_insert(6);
join(thread1); // Optional
```

2.2 Mutexes

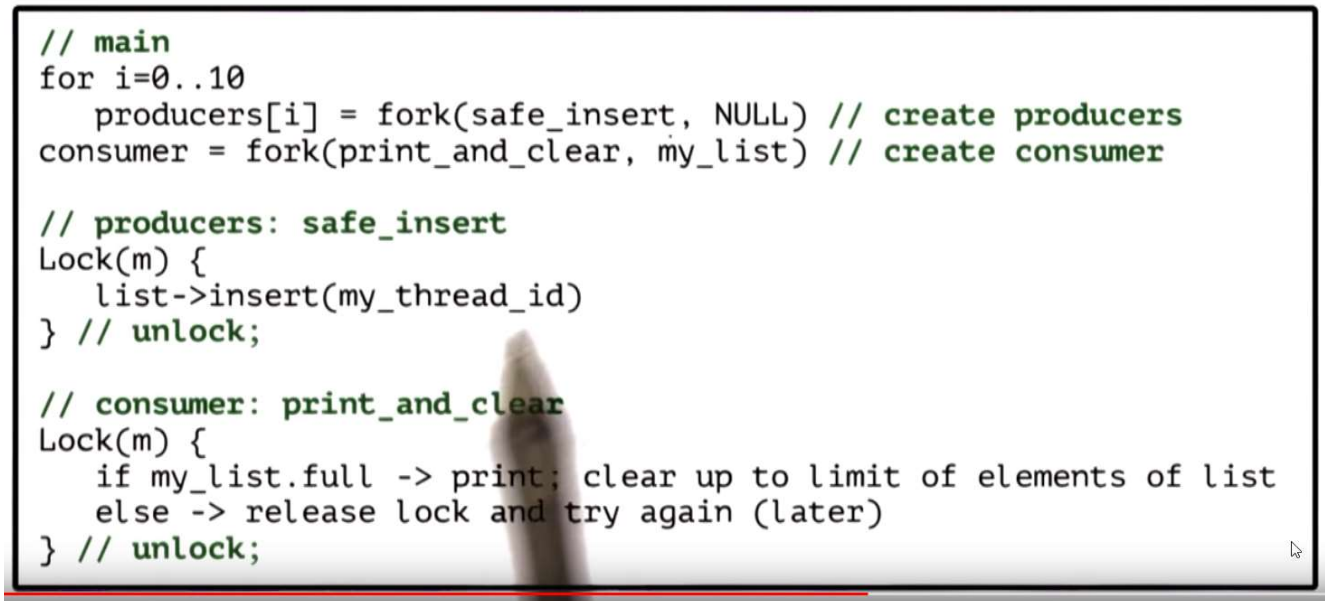
- Mutex is like a lock that should be used whenever accessing data or states that's shared among threads.
- When trying to access/modify shared data, the threads will try to acquire the mutex, if failed, the thread will be blocked at the lock acquiring operation, hence suspended until the mutex owner (the thread that successfully acquired the lock) releases the lock.
- Mutex data structure components:
  - locked?
  - current owner of mutex
  - blocked threads that are waiting on it
- Critical Section:
  - portion of the code protected by the mutex
  - any kind of operation that requires that only one thread at at time performs it
- Upon releasing of a mutex lock, any thread trying to acquire it can start executing the lock operation
- Birrell's lock construct:

```
lock(m) {
// critical section
} // unlock
```

Most APIs have lock(m) and unlock(m) operations.

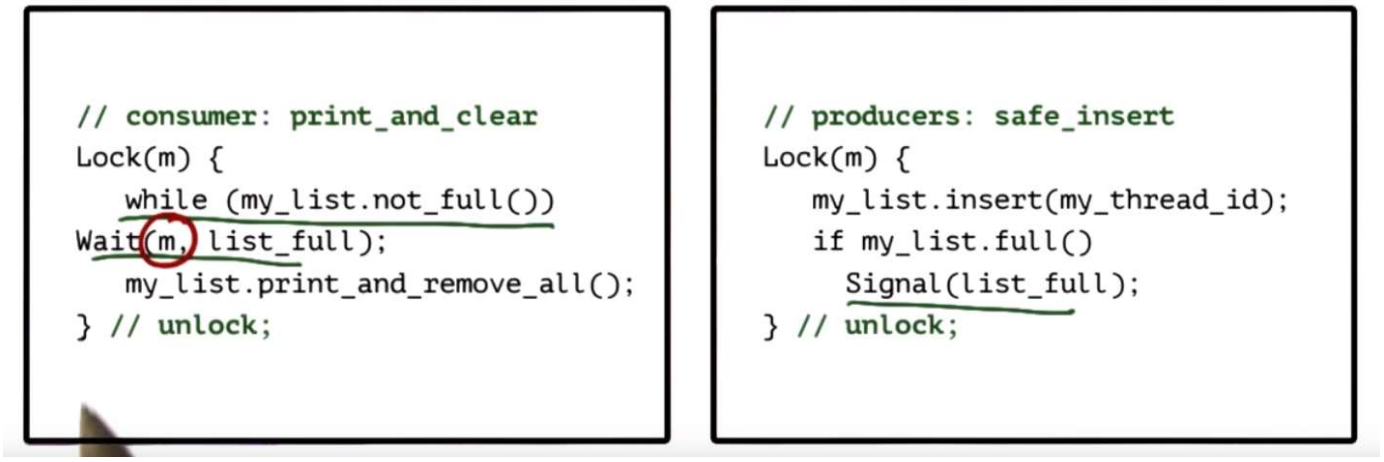
2.2.1 Producer and Consumer Example

- Producer thread adds data into a shared buffer.
- Consumer uses the data in shared buffer



```
// main
for i=0..10
   producers[i] = fork(safe_insert, NULL) // create producers
consumer = fork(print_and_clear, my_list) // create consumer

// producers: safe_insert
Lock(m) {
   list->insert(my_thread_id)
} // unlock;

// consumer: print_and_clear
Lock(m) {
   if my_list.full -> print; clear up to limit of elements of list
   else -> release lock and try again (later)
} // unlock;
```

2.3 Condition Variable



```
// consumer: print_and_clear
Lock(m) {
   while (my_list.not_full())
Wait(m, list_full);
   my_list.print_and_remove_all();
} // unlock;
```

```
// producers: safe_insert
Lock(m) {
   my_list.insert(my_thread_id);
   if my_list.full()
     Signal(list_full);
} // unlock;
```

Condition variables are used in association with some mutex.

The mutex in the wait() command of consumer thread is released when the consumer thread is waiting otherwise my_list will never become full.

2.3.1 Condition Variable API

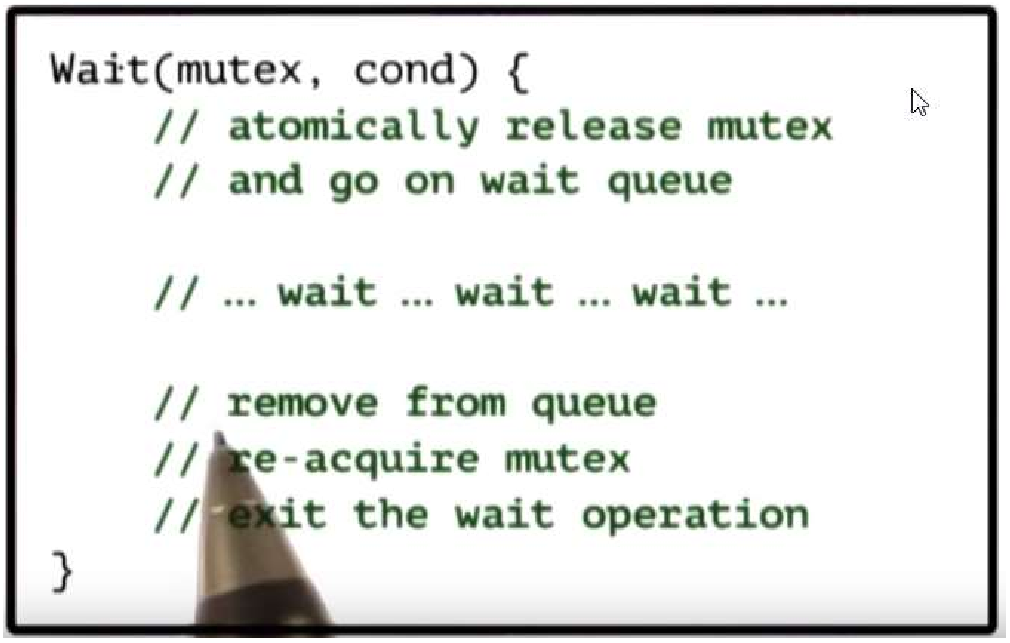
Components

- Condition Type/ Data Structure
- wait(mutex, cond)
  - release mutex when waiting for cond
  - re-acquire mutex after condition is satisfied
- signal(cond)
  - notify only one thread waiting on the condition that the condition is satisfied

- broadcast(cond)
    - notify all waiting threads that the condition is satisfied

Condition Variable Type should keep track of the mutex it's associated with and a list of waiting threads.

**Wait(mutex, cond) implementation **



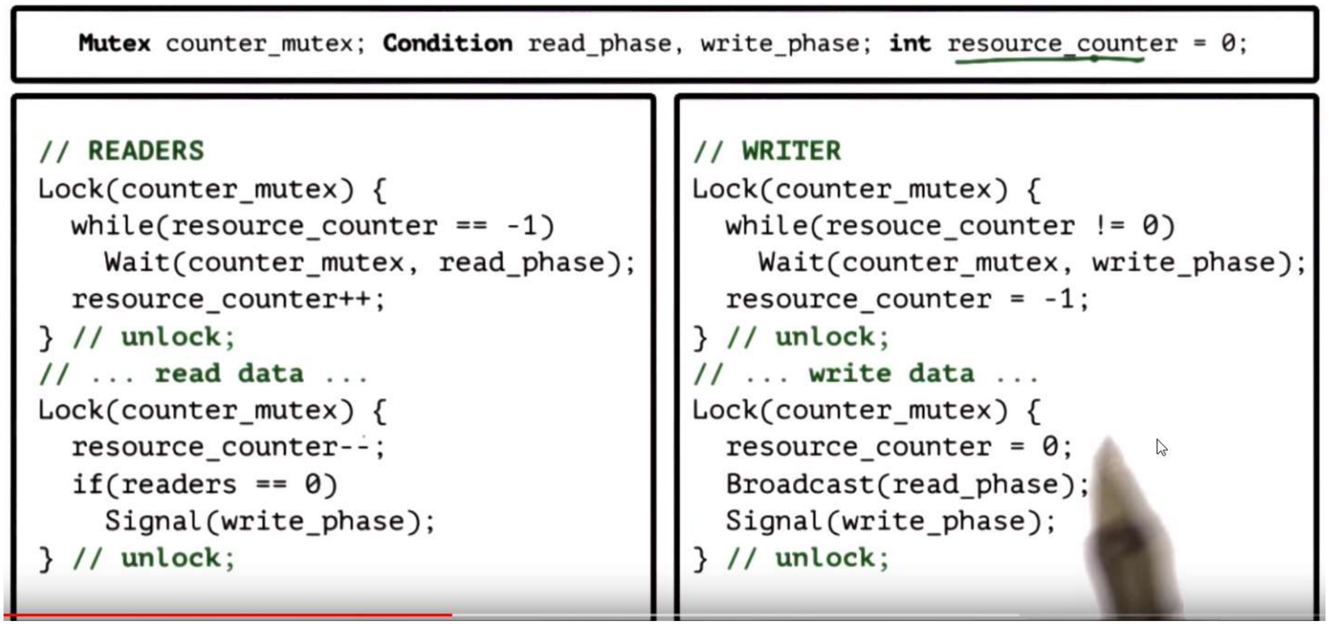
```
Wait(mutex, cond) {
    // atomically release mutex
    // and go on wait queue

    // ... wait ... wait ... wait ...

    // remove from queue
    // re-acquire mutex
    // exit the wait operation
}
```

3. Reader Writer Problem

One thread reads from and another thread writes to a shared data.

- At any given time, 0 or more reader threads can access the data and only writer 0 or 1 thread can write to the data.
- Reader thread cannot read while writer thread is modifying, writer thread cannot modify while reader thread is reading.
- mutex doesn't work here because it's too strict, it doesn't allow multiple readers to access the shared data at the same time
- Introduce resource counter, carrying the information about how many reader thread are accessing the data or if there is any writer thread modifying the data.
- But there needs to be a mutex on the resource counter.

- There can



```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

```
// READERS
Lock(counter_mutex) {
  while(resource_counter == -1)
    Wait(counter_mutex, read_phase);
  resource_counter++;
} // unlock;
// ... read data ...
Lock(counter_mutex) {
  resource_counter--;
  if(readers == 0)
    Signal(write_phase);
} // unlock;
```

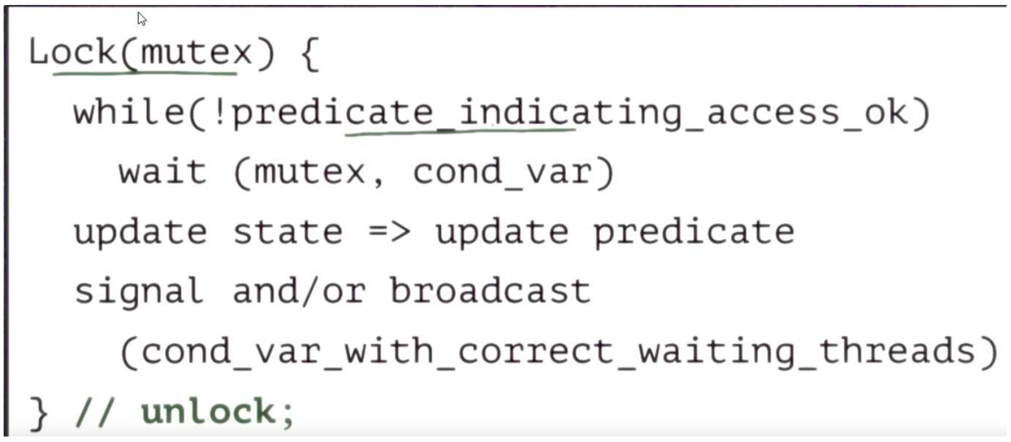
```
// WRITER
Lock(counter_mutex) {
  while(resouce_counter != 0)
    Wait(counter_mutex, write_phase);
  resource_counter = -1;
} // unlock;
// ... write data ...
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

In each reader thread, it checks if the data is currently being modified first -> read --> reduce resource counter and notify writer thread if it is the last reader.

In writer thread, it checks if there are reader thread currently reading the data -> if not then set resource counter to -1, indicating it's no longer available for reading --> write to data --> update resource counter and tell all reader thread that it's done writing + signal other writer thread that it is done writing.

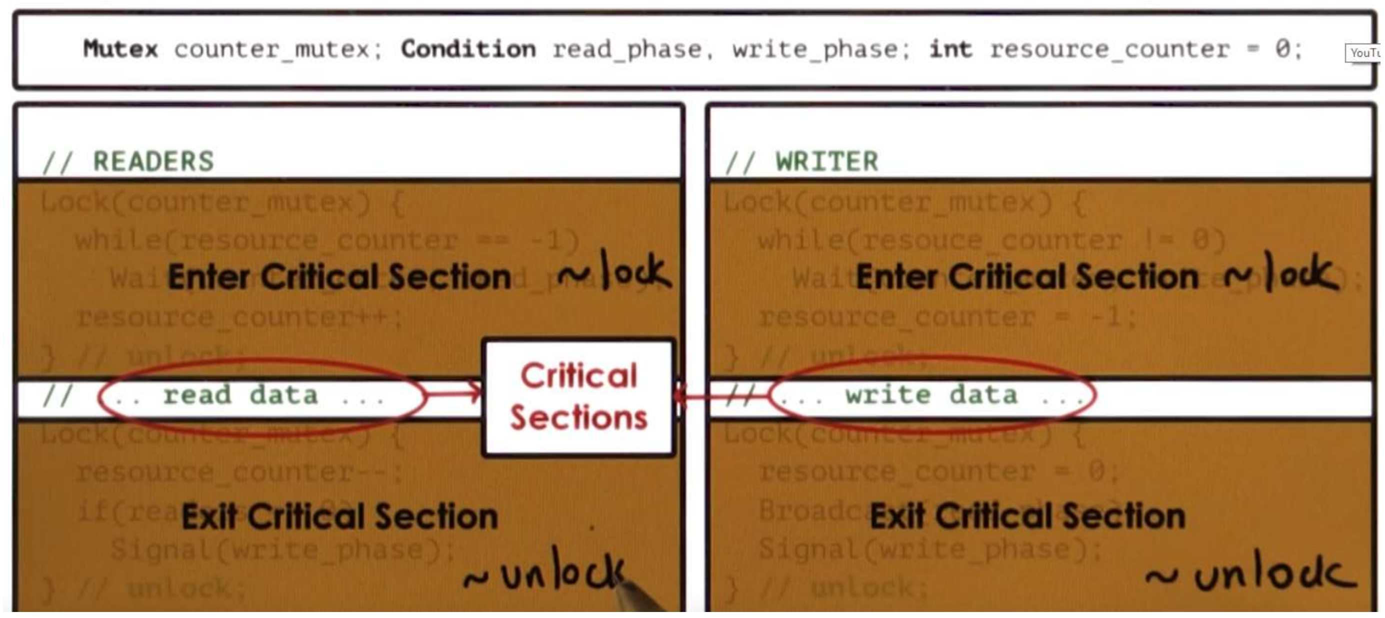
> This part is really about understanding the mechanism, notes won't help if you don't understand. If you understands, no note will be needed.

Structure of typical critical section



```
Lock(mutex) {
  while(!predicate_indicating_access_ok)
    wait (mutex, cond_var)
  update state => update predicate
  signal and/or broadcast
    (cond_var_with_correct_waiting_threads)
} // unlock;
```

Critical section sandwhiched by enter and exit critical section blocks.



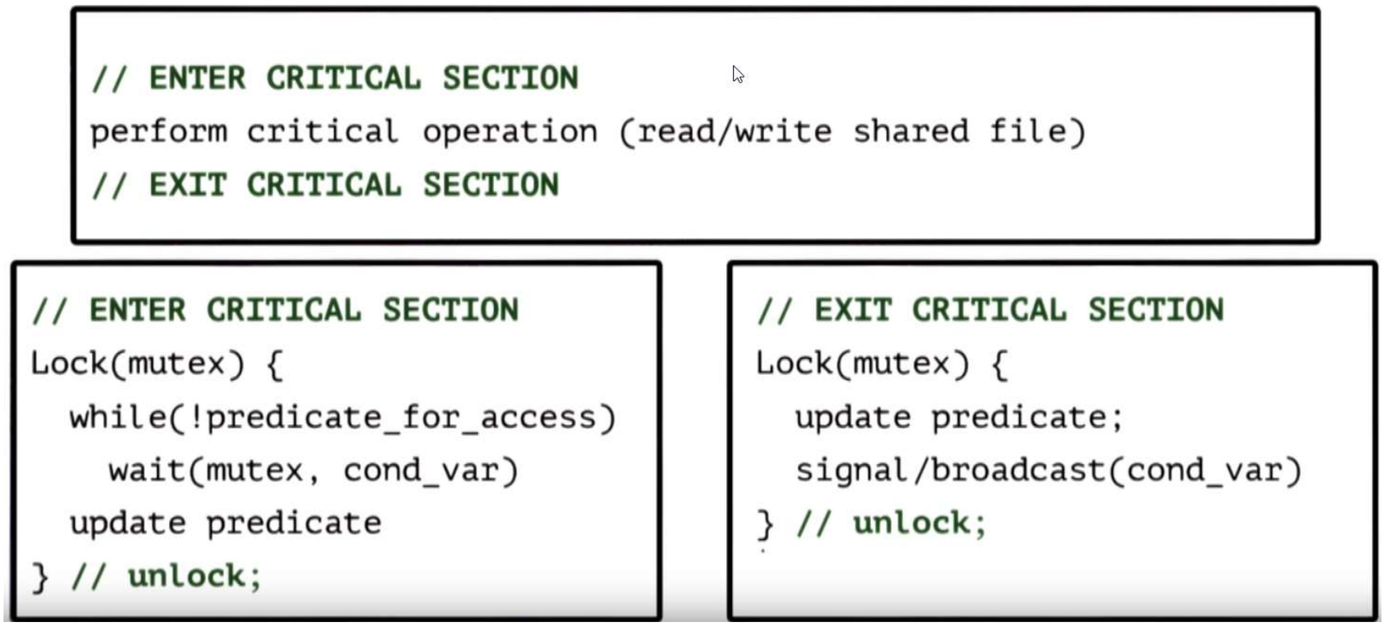
```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

```
// READERS
Lock(counter_mutex) {
  while(resource_counter == -1)
    Wai                    d_phase);
  resource_counter++;
} // unlock;
// .. read data ...
Lock(counter_mutex) {
  resource_counter--;
  if(rea
    Signal(write_phase);
} // unlock;
```

```
// WRITER
Lock(counter_mutex) {
  while(resouce_counter != 0)
    Wai                    e_phase);
  resource_counter = -1;
} // unlock;
// ... write data ...
Lock(counter_mutex) {
  resource_counter = 0;
  Broadc
  Signal(write_phase);
} // unlock;
```

- The four colored blocks only use one counter_mutex so only one thread can execute the colored blocks at one time ---- Only one thread can manipulate the resource_counter variable at a time.
- Multiple thread at a time can read data
- The predicate here is the resource_counter

Critical Section Control Summary

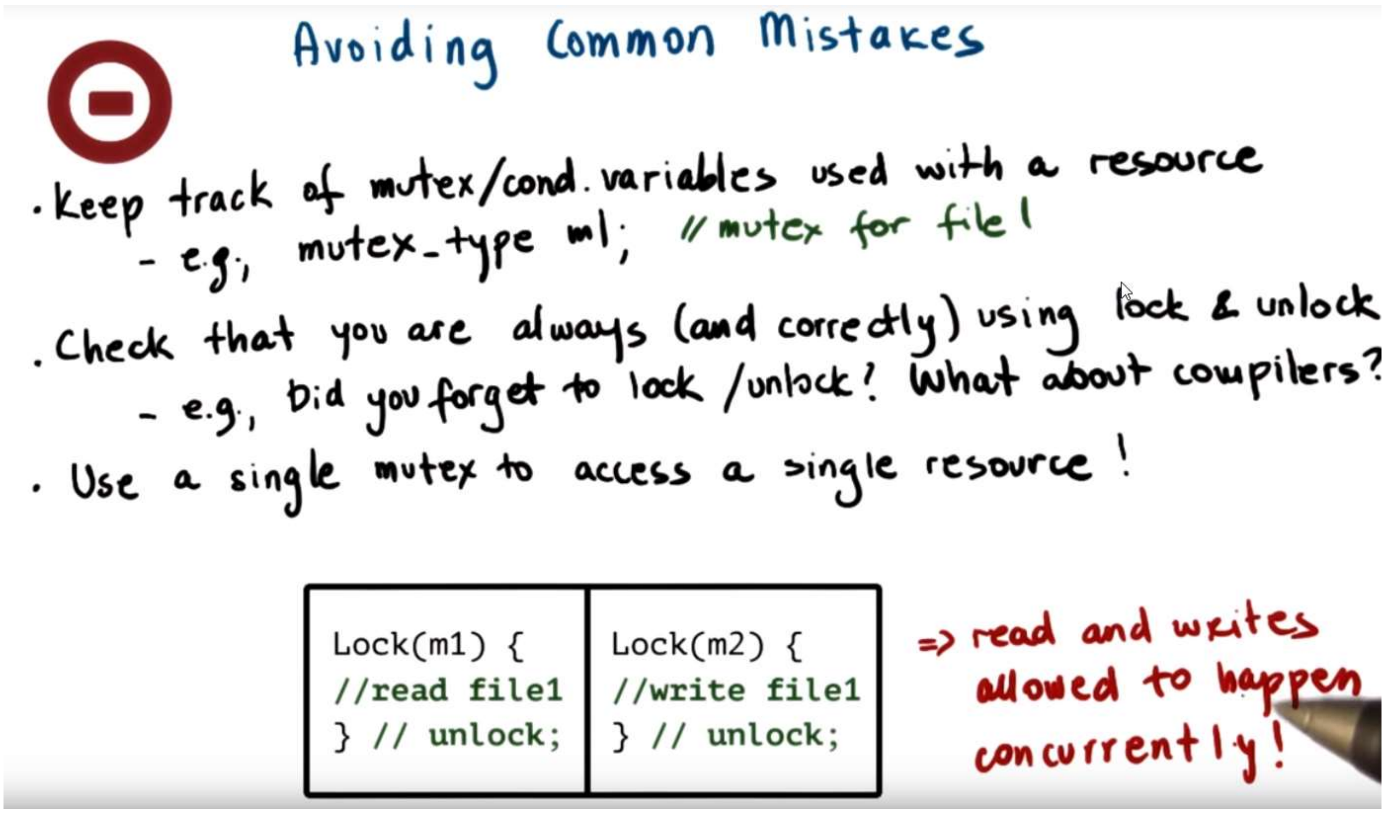


```
// ENTER CRITICAL SECTION
perform critical operation (read/write shared file)
// EXIT CRITICAL SECTION
```

```
// ENTER CRITICAL SECTION
Lock(mutex) {
  while(!predicate_for_access)
    wait(mutex, cond_var)
  update predicate
} // unlock;
```

```
// EXIT CRITICAL SECTION
Lock(mutex) {
  update predicate;
  signal/broadcast(cond_var)
} // unlock;
```

Common Pitfalls to avoid



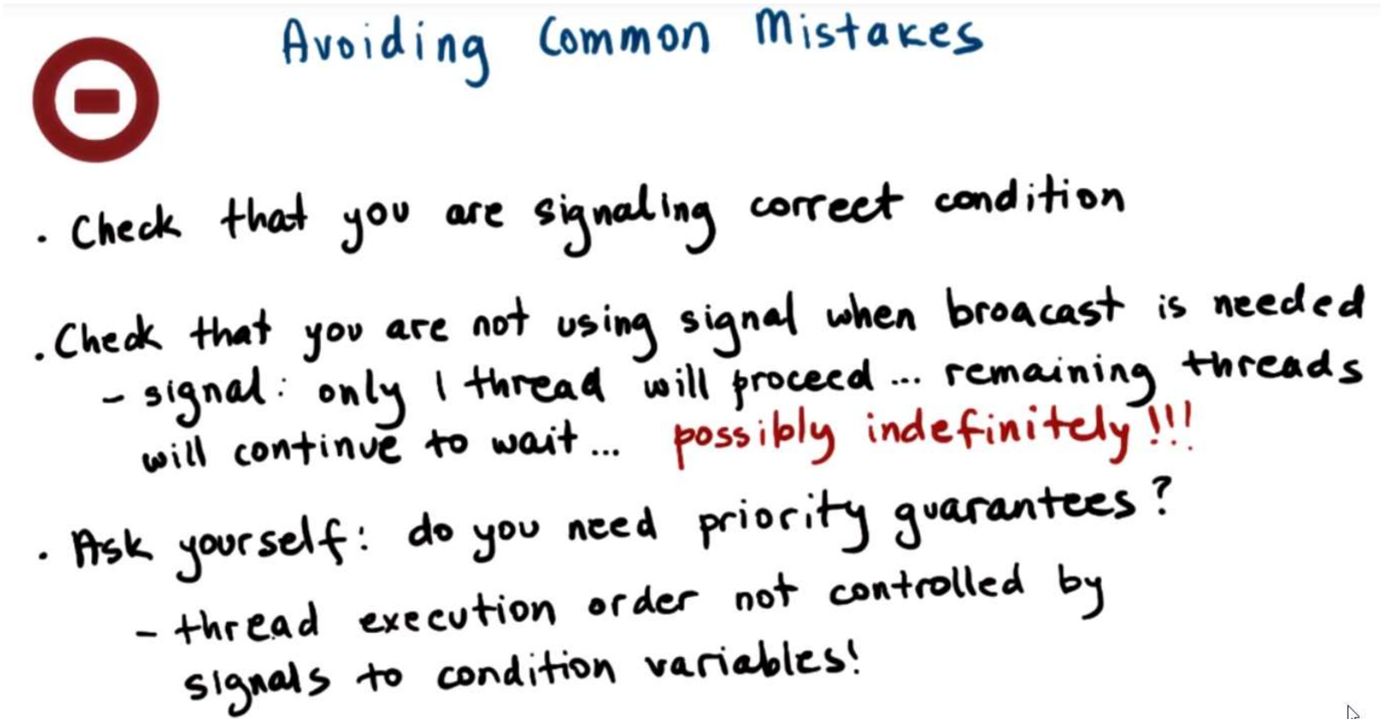
Avoiding Common Mistakes

- Keep track of mutex/cond. variables used with a resource
  - e.g., mutex_type m1; // mutex for file 1
- Check that you are always (and correctly) using lock & unlock
  - e.g., Did you forget to lock/unlock? What about compilers?
- Use a single mutex to access a single resource!

```
Lock(m1) {
//read file1
} // unlock;
```

```
Lock(m2) {
//write file1
} // unlock;
```

=> read and writes allowed to happen concurrently!



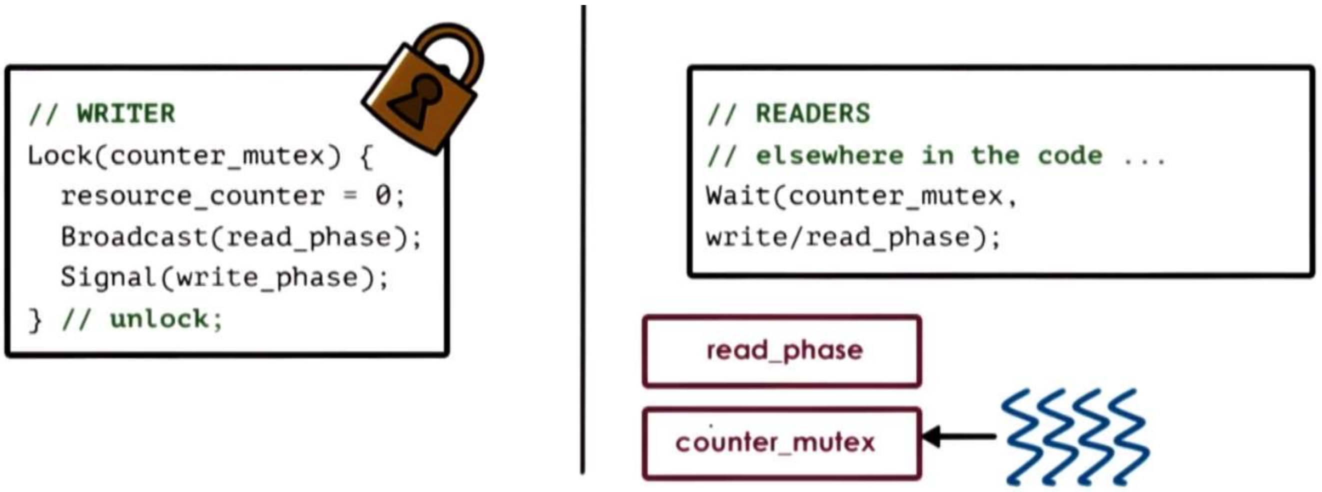
Avoiding Common Mistakes

- Check that you are signaling correct condition
- Check that you are not using signal when broacast is needed
  - signal: only 1 thread will proceed ... remaining threads will continue to wait ... possibly indefinitely!!!
- Ask yourself: do you need priority guarantees?
  - thread execution order not controlled by signals to condition variables!

4. Spurious Wake-ups and Deadlocks

4.1 Spurious/unnecessary wakeups

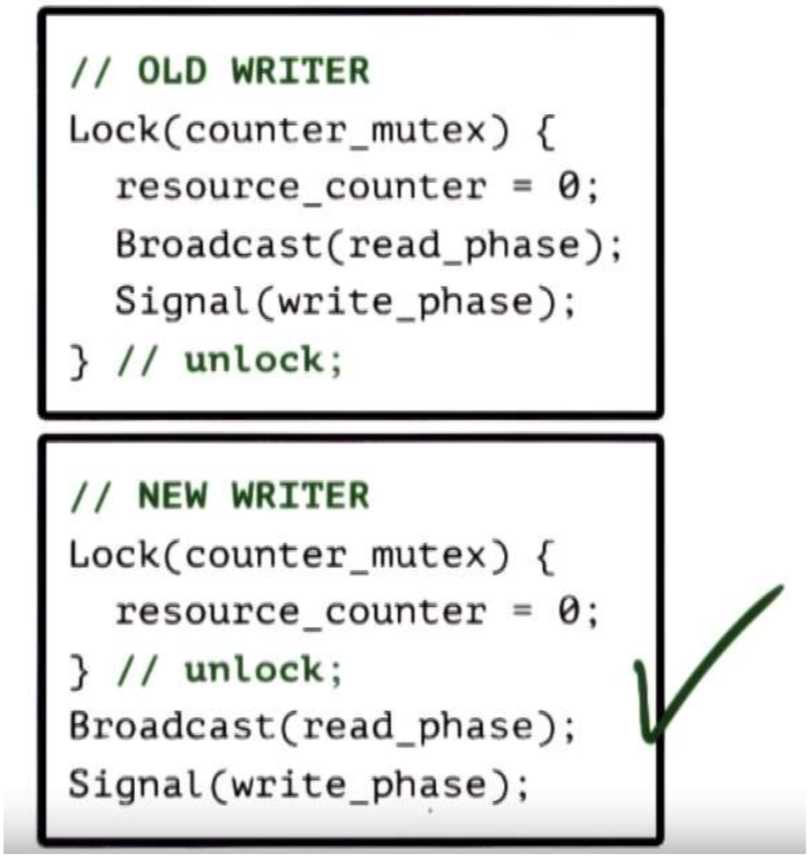
- May affects performance but not necessarily correctness
- When we wake threads up knowing they may not be able to proceed.
- We waste CPU cycles doing context switching for nothing, they wake up and then go to wait status again.
- Problem in the following screenshot:
  - unlock after broadcasting/signal, no other thread can get the one and only counter_mutex when they wake up from waiting



```
// WRITER
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

```
// READERS
// elsewhere in the code ...
Wait(counter_mutex,
write/read_phase);
```

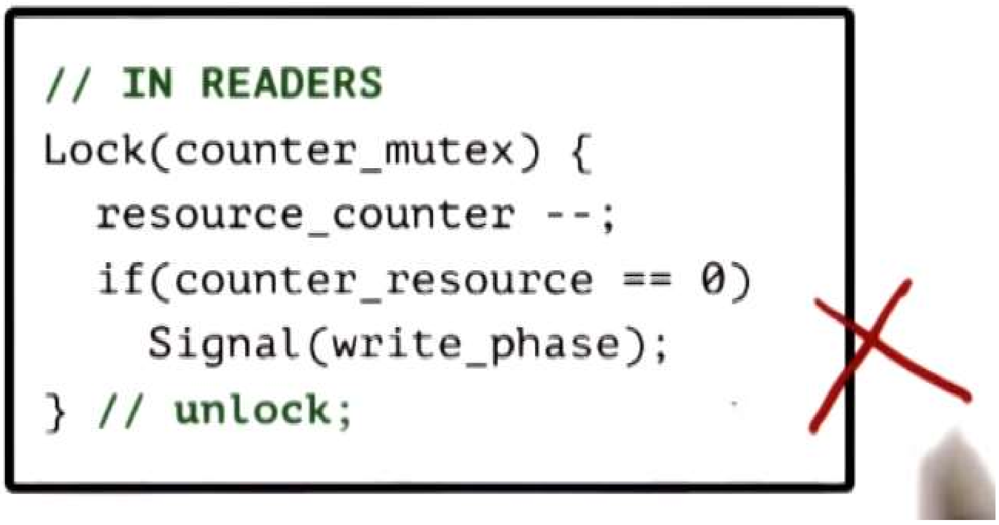
- Fix:
  - Note that we cannot always fix it this way.



```
// OLD WRITER
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

```
// NEW WRITER
Lock(counter_mutex) {
  resource_counter = 0;
} // unlock;
Broadcast(read_phase);
Signal(write_phase);
```

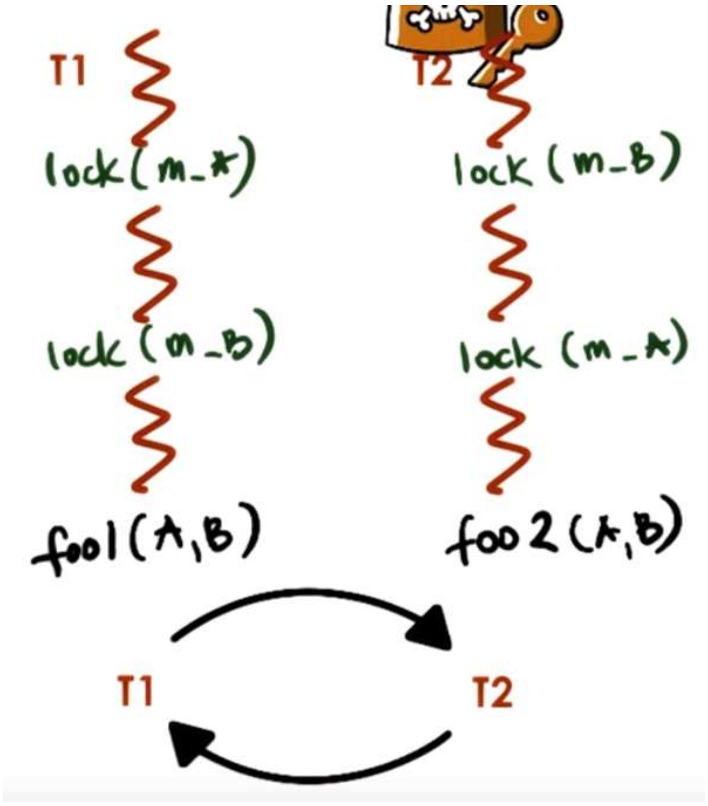
  - In this situation we cannot put signal after mutex unlock
    - Because the signal operation depends on the value of conter_resource, the shared resource the one and only counter_mutex is protecting.
    - If we unlock counter_mutex before signal, the counter_resource value might have already been changed when we want to findout if we should send the signal to write_phase.



```
// IN READERS
Lock(counter_mutex) {
  resource_counter --;
  if(counter_resource == 0)
    Signal(write_phase);
} // unlock;
```

4.2 Deadlocks

Definition: Two or more competing threads are waiting on each other to complete but none of them ever do... BECAUSE THEY ARE ALL WAITING ON SOMEONE ELSE AND DOING NOTHING.



How to avoid deadlock?

1. Fine-grained locking: unlock A before locking B
    - problem: the thread needs both A &B
2. Use one mega lock
    - It may be ok for some applications
    - problem: limits potential parallelism
3. Maintain a lock order
    - everyone first get lock for A then for B
    - prevents cycles in wait graph
    - It takes effort to ensure the order.

Cycle in wait graph <=> deadlock

What to do about a deadlock

1. preventions----expensive
2. detection & recovery ---- rollback, still expensive
3. Do nothing...hoping it won't happen, if it happens, roboot.

5. Multithreading Models

5.1 Kernel vs. User-level threads

Kernel Threads

- visible to kernel
- managed by kernel level components like kernel level scheduler
- kernel is multi-threading

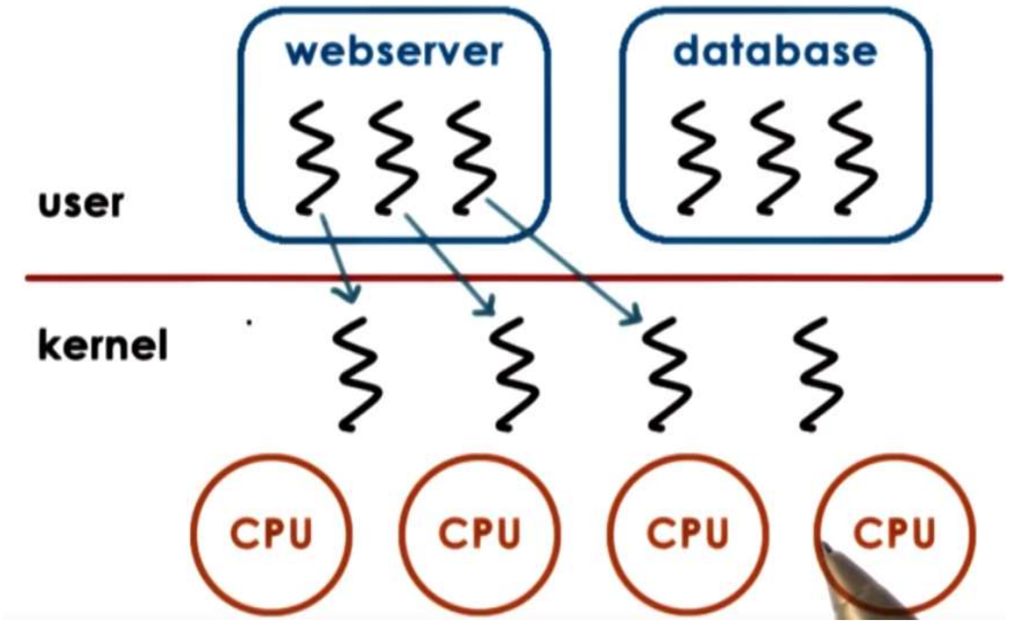
User level Threads

- Each process/application instance is multi-threaded
- For a user-level thread to be executed it must be associated with a kernel level thread
- Then the kernel level scheduler schedule the kernel level thread onto a CPU

5.2 Multithreading Models — Relationship between kernel and user level threads

5.2.1 One-to-one Model

- 1 user level thread ~ 1 kernel level thread
- When a user level thread is created, a kernel level thread is created or it is associated with an idle kernel level thread.



The good

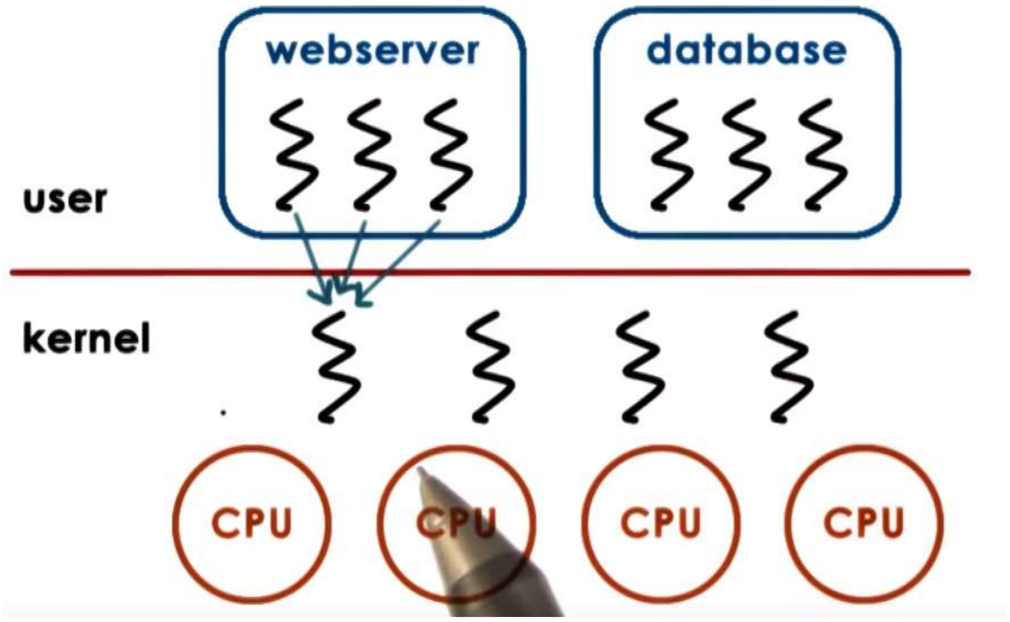
- The user level processes can directly benefit from the threading support that's available in the kernel.

The bad

- Must go to OS for operations --could be expensive
- OS may have limits on its policies, such as maximum number of threads
- Portability issue, the application can only run on kernels that provide exactly the support it needs.

5.2.2 Many-to-one Model

- All user level threads are mapped on to a single kernel thread.
- A thread management lib decides which one of the user-level thread will be mapped on to kernel level thread at a given point of time
-



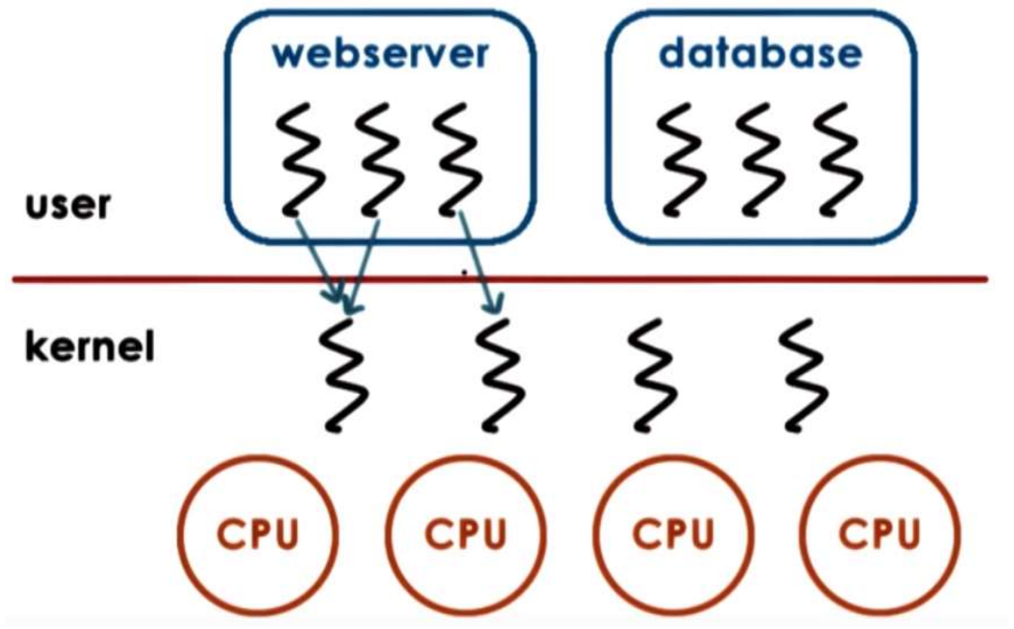
The good

- Totally portable
  - everything is done in the user-level thread lib
  - don't rely on any specific kernel-level support
  - Not limited by specific limits or policies of the kernel
- Don't have to rely on user kernel transitions, most things are done in user-level thread lib, saving the user-kernel mode switch cost.

The bad

- OS has no insights into application needs, it doesn't even know the application is multithreaded. All the OS can see is the single kernel level thread.
- When one of the user level application thread makes an I/O request, kernel level scheduler seeing the kernel level thread block will block the entire process —> The whole process is forced to wait on the thread making the I/O request.

5.2.3 Many-to-Many Model

- Alllow some user level threads of a process to be associated with one kernel level thread and others to be associated with another kernel level thread.
- 

The good

- Combine the best of both worlds.
- The kernel knows that the process is multithreaded since it was assigned to multiple kernel threads.
- If one user level thread blocks an I/O, the threads in the process that are assigned to other kernel level thread can still execute.
- The user-level threads may be scheduled onto any of the underlying kernel-level threads.
  - bound mapping: certain user-level thread mapped one-to-one permanently onto a kernel-level thread ----Able to treat certain user-level threads differently.

The bad

- Require coordination between user-and kernel level thread management.

5.2.2 Scope of Multithreading

System Scope

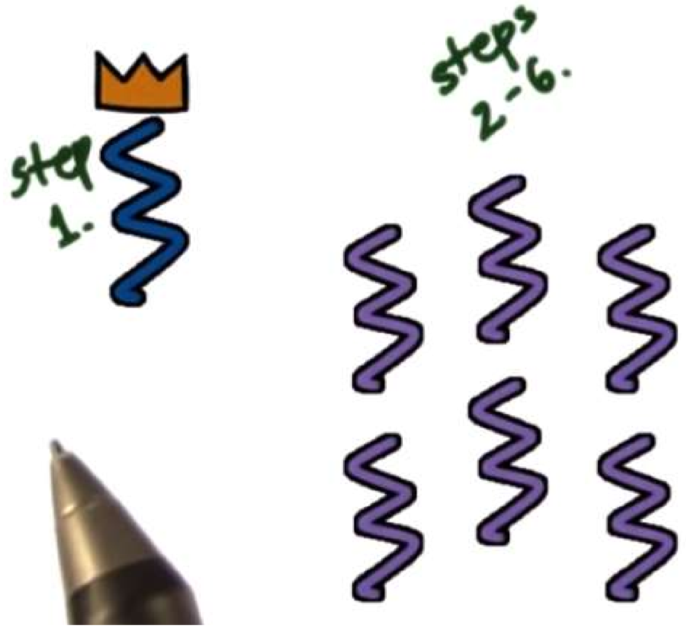
- System-wide thread management
- Supported by OS-level thread managers
- e.g. CPU Scheduler

Process Scope

- User-level Library
- Manages threads within a single process
- Different processes will be managed by different instances of the same library
- Different processes can link entirely different user-level libraries

5.3 Multithreading Patterns

5.3.1 Boss worker patterm



- Boss: assign work to workers
- Worker: performs entire task
- Throughput of the system limited by boss thread----MUST KEEP BOSS EFFICIENT
- throughput = 1/ boss_time_per_order

Boss-Worker Coordination

- directly signalling specific worker:
  - wait for that worker to accept the order from the boss----handshake
  - **Good:** the workers don't need to synchronize with each other
  - **Bad:** boss is too busy, throughput goes down
- Eatablish a queue between boss and workers:
  - similar to producer consumer queue
  - boss is producer of work
  - workers are consumers of work
  - **Good:** Boss doesn't have to wait for a worker to explicitly do handshake, hence improve throughput.
  - **Bad:** Workers have to synchronize their accesses to the shared queue.

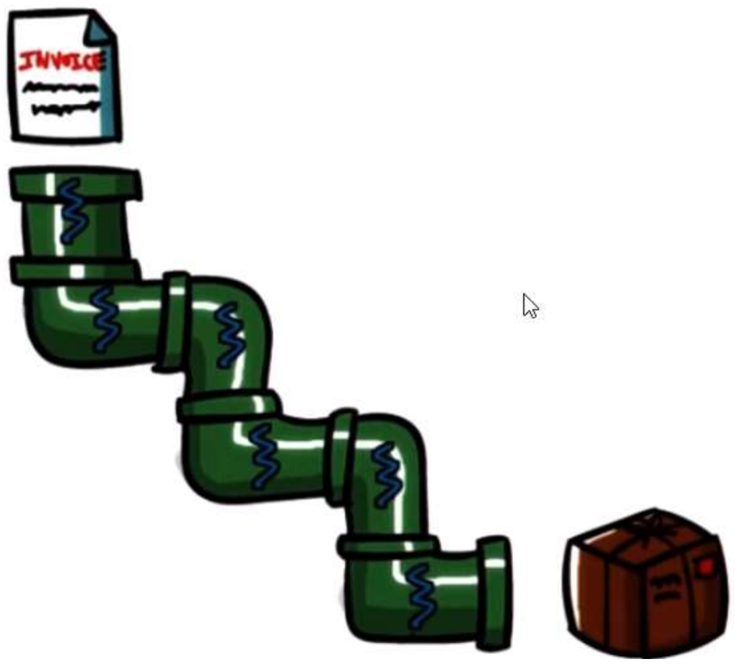
How many workers?

- more workers, faster consumption of work ququе, better throughput
- worker threads switching comes with overhead
- Options:
  - add worker dynamically on demand.
    - Not efficient if it takes a long time for a worker thread to arrive
  - pool of workers
    - workers already exists
    - create several threads when we decide that the pool size needs to be adjusted
    - **Good:** simplicity
    - **Bad:**
      - thread pool management and synchronization overhead
      - ignores locality: boss doesn't know what each worker was doing previously—> potential cold cache—bad for optimization

Boss - Worker Variants

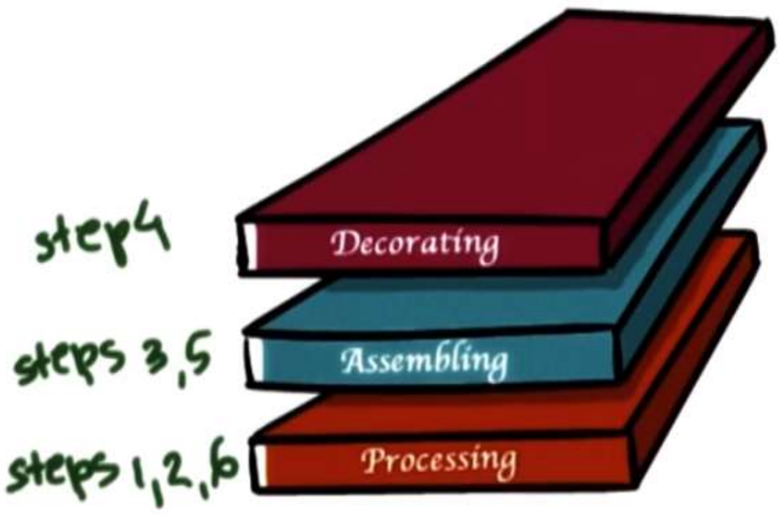
- # All worker created equal:
- workers specialized for certain tasks:
    - Boss has to do a little bit more work for each order to assign to the right type of worker
    - each thread may have better performance because they are performing similar task
    - **Good:** exploits locality
    - **Bad:** load balancing: how many threads should we assign to different tasks

5.3.2 Pipeline Pattern



- A task is divided into subtasks
- Each subtask performed by a different thread (they can work on the same subtasks of different big tasks)
- task == pipeline of threads
- multiple tasks concurrently in the system, in different pipeline stages
- every task are in different stage in the pipeline
- throughput == weakest link (longest stage in the pipeline)
    - Each pipeline stage is a thread pool
    - More threads in the thread pool for longer stages
    - Goal: have each stage process the same number of subtasks over the same period of time
- Ways to pass work among these different pipeline stages:
    - shared-buffer based mechanism [better way]
        - help correct for any small imbalances in the pipeline
    - explict communication between stages:
        - A thread in earlier stage have to wait till a thread in the next stage is free
- **Good:** Specialized threads and locality
- **Bad:** hard to maintain pipeline balancing and synchronization over time

5.3.3 Layered Pattern



- Each layer is a group of related subtasks
- the threads assigned to a layer can perform any one of the subtasks that correspond to it
- end-to-end task must pass up and down through all layers
- **Good:**
    - specialization and locality
    - less fine-grained than the pipeline
    - easier to decide how many threads should allocate per layer
- **Bad:**
    - may not be suitable for all applications (e.g. may not make sense to group step 1 and 6 together)
    - complex synchronization