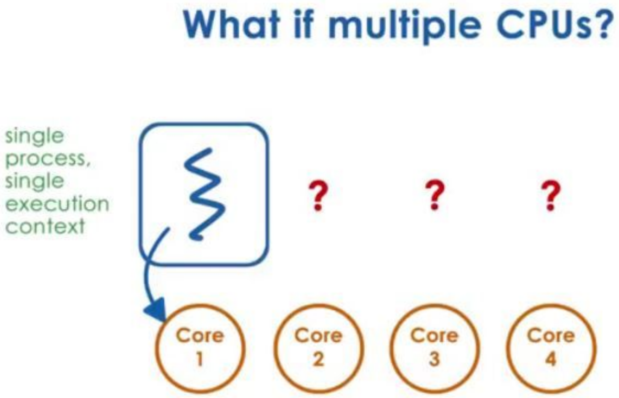
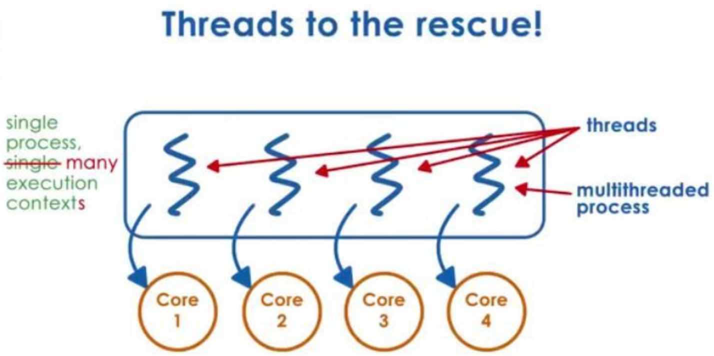
P2L2 - Threads and Concurrency

01 - Lesson Preview

During the last lecture we talked about processes and process management. We said that a process is represented with its address space and its execution context, and this has registers and stack and stack pointer.



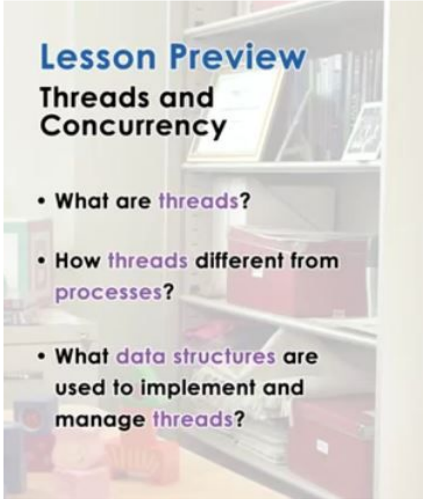
This type of process, the process that is represented in this way can only be executed at one CPU at a given point of time.



If we want the process to be able to execute on multiple CPUs to take advantage of multi-CPU systems or multi-core systems of today, that process has to have multiple execution contexts ***(amit - takeaway is that one process can't be on multiple CPUS with just one execution context) (amit - question - but why not??? When everything is anyway saved in PCB and PCB is in memory?) .*** We call such execution contexts within a single process, ***threads***.

In this lecture, we will explain what threads are and how they differ from processes. We will talk about some of the basic data structures and mechanisms that are needed to implement threads and manage and coordinate their execution.

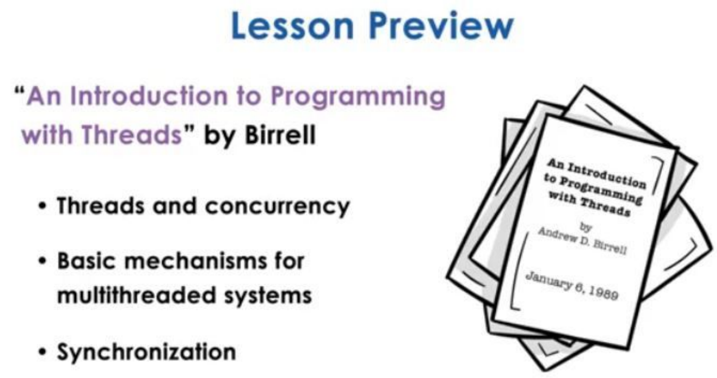
We will use Birrell's paper "An Introduction to Programming with Threads" to explain some of these concepts and this is an excellent paper that describes the fundamentals of multithreading, synchronization, and concurrency.



Lesson Preview

Threads and Concurrency

- What are threads?
- How threads different from processes?
- What data structures are used to implement and manage threads?



Lesson Preview

"An Introduction to Programming with Threads" by Birrell

- Threads and concurrency
- Basic mechanisms for multithreaded systems
- Synchronization

In a later lesson we will talk about a concrete multithreading system called ***pthreads***.

02 - Visual Metaphor

In this lesson we're talking about threads and concurrency. So how am I going to visualize these concepts? Well, one way you may think of threads is that each thread is like a worker in a toy shop. But what qualities make a worker in a toy shop similar to a thread? The first quality is that a worker in a toy shop is an active entity. Secondly, a worker in a toy shop works simultaneously with others. And finally, a worker in a toy shop requires coordination. Especially when multiple workers are working at the same time. And perhaps even contributing to the same toy order.

Let's elaborate on these points now. A worker is an active entity in the sense that it's executing a unit of work that is necessary for a given toy order. Many such workers can contribute to the entire effort required for an actual toy to be built. Next the worker can simultaneously work with others, this is pretty straightforward. You can imagine a shop floor with many workers, all simultaneously are concurrently hammering, sewing, building toys at the same time. They are working on the same order or others. And finally, while the workers can work simultaneously, this comes with some restrictions. Workers must coordinate their efforts in order to operate efficiently. For instance, workers may have to share tools, they may have to share some working areas, their workstations, or even parts while they're in the process of making toys and executing the toy orders.

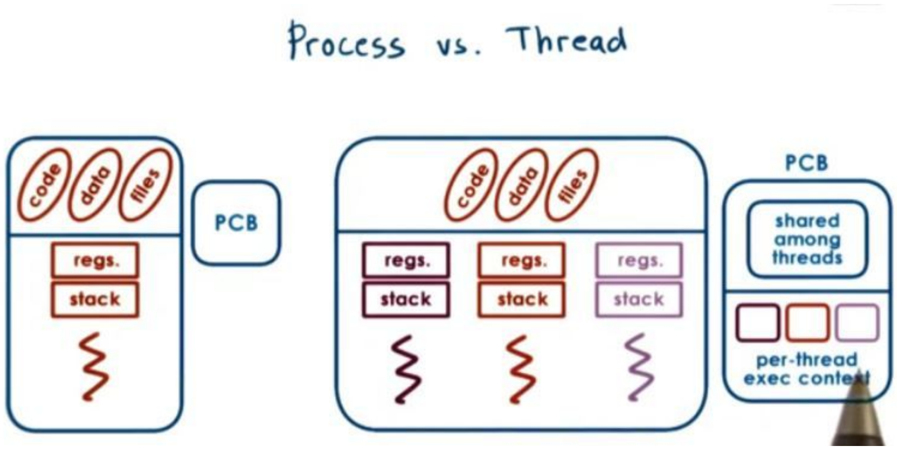
Now that we know about workers in a toy shop, what about threads? How do they fit into this analogy? First, threads are also active entities. Except in this case, ***threads execute a unit of work on behalf of a process.*** Next, threads can also work simultaneously with others, and this is where the term concurrency really applies. For instance, in modern systems that have multiple processors, multiple cores, you can have multiple threads really at the exact same time executing concurrently. But this obviously will require some level of coordination. And specifically when we talk about coordination, we're really mainly talking about coordinating access to the underlying platform resources. Sharing of I/O devices, the CPU course, memory, all of these and other systems resources must be carefully controlled and scheduled by the operating system.

This begs the question, how do we determine which thread gets access to these resources? And as you will see in this lesson, the answer to the question is very important, designed decision for both operating systems, as well as software developers in general.

03 - Process vs Thread

Let's try to understand better the differences between a process and a thread. To recap from the previous lessons, a single thread of **process is represented by its address space** (i.e. code/data/files). The address space will contain all of the virtual to physical address mappings for the process, for its code, its data, its heap section files, for everything. The **process is also represented by its execution context** (i.e. regs/stack) that contains information about the values of the registers, the stack pointer, program counter, etc. The operating system represents all this information in a process control block (PCB = code/data/files + regs/stack).

Threads, we said, represent multiple, independent execution contexts. They're part of the ***same virtual address space, which means that they will share all of the virtual to physical address mappings. They will share all the code, data, files (amit - so in terms of PCB, they will have same data section, same text section, same address mappings, same assembly code etc)***.



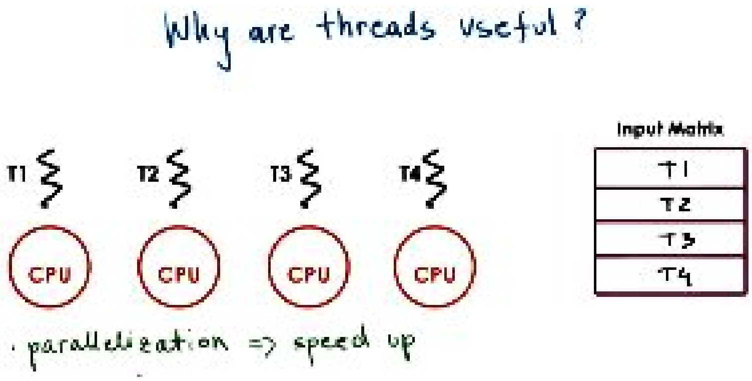
However, they will potentially execute different instructions, access different portions of that address space, operate on different portions of the input, and differ in other ways. This means that ***each thread will need to have a different program counter, stack pointer, stack, thread-specific registers (amit - so the question can be, what is common between threads and what should be per-thread? Things related to direct***

execution are per thread). So we will have, for each and every thread, we will have to have separate data structures to represent this per-thread information.

The operating system representation of such a multithreaded process will be a more complex process control block structure than what we saw before. **This will contain all of the information that's shared among all the threads. So, the virtual address mappings, description about the code and data, etc. And it will also have separate information about every single one of the execution contexts that are part of that process** ***(amit - so now we are saying that PCB will contain info about threads also)***.

04 - Why are threads useful?

Let's now discuss why are threads useful. We will do that by looking at an example on a multiprocessor or multicore system.



At any given point of time when running a single process there may be multiple threads belonging to the process each running concurrently on a different processor. ***One possibility is that each thread executes the same code, but for a different subset of the input***. For instance, for a different portion of an input array or an input matrix. For instance, T1 processes this portion of the input matrix, T2 processes the next one, and so forth. Now all of the threads execute the exact same code but they're not necessarily executing the exact same instruction at a single point in time. So every single one of them will still need to have its own private copy of the stack, registers, program counter, etc. ***By parallelizing the program in this manner, we achieve speed up***. ***We can process the input much faster than if only a single thread on a single CPU had to process the entire matrix.***

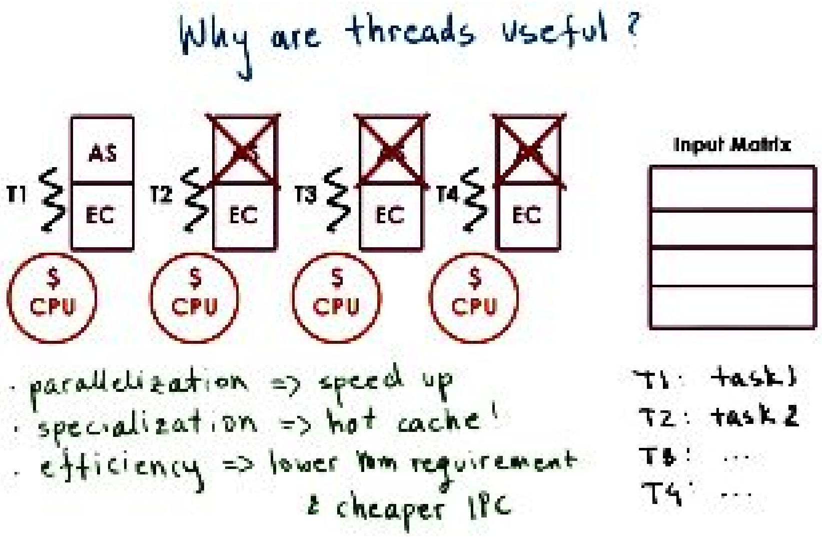
Next, threads may execute completely different portions of the program. For instance, you may designate certain threads for certain I/O tasks like input processing, or display rendering, or another option is to have different threads operate on different portions of the code that correspond to specific functions. ***For instance, in a large web service***

application, different threads can handle different types of customer requests. By specializing the different threads to run different tasks, or different portions of the program, we can differentiate how we manage those threads. So for instance, we can give higher priority to those threads that handle more important tasks or more important customers (amit - how can we prioritize threads).

Another important benefits of partitioning what exactly are the operations executed by each thread and on each CPU comes from the fact that performance is dependant on how much state can be present in the processor cache (amit - takeaway - always think about the cache also).

In this picture, each thread running on a different processor has access to its own processor cache. If the thread repeatedly executes a smaller portion of the code, so just one task, more of that state and of that program will be actually present in the cache. ***So one benefit from specialization is that we end up executing with a hotter cache. And that translates to gains in performance.***

You may ask why not just write a multiprocess application ***where every single processor runs a separate process. If we do that, since the processes do not share an address space (AS) we have to allocate for every single one of these contexts AS and execution context (EC). So the memory requirements, if this were a multiprocessor implementation, would be that we have to have four AS allocations and four EC allocations. A multithreaded implementation results in threads sharing an AS so we don't need to allocate memory for all of the AS information*** for these remaining EC ***(amit - takeaway - we have two things AS and EC, think in terms of these 2 factors, EC is what it needs to run on***



cpu so it is PC and stack pointer and registers and that needs to be per thread).

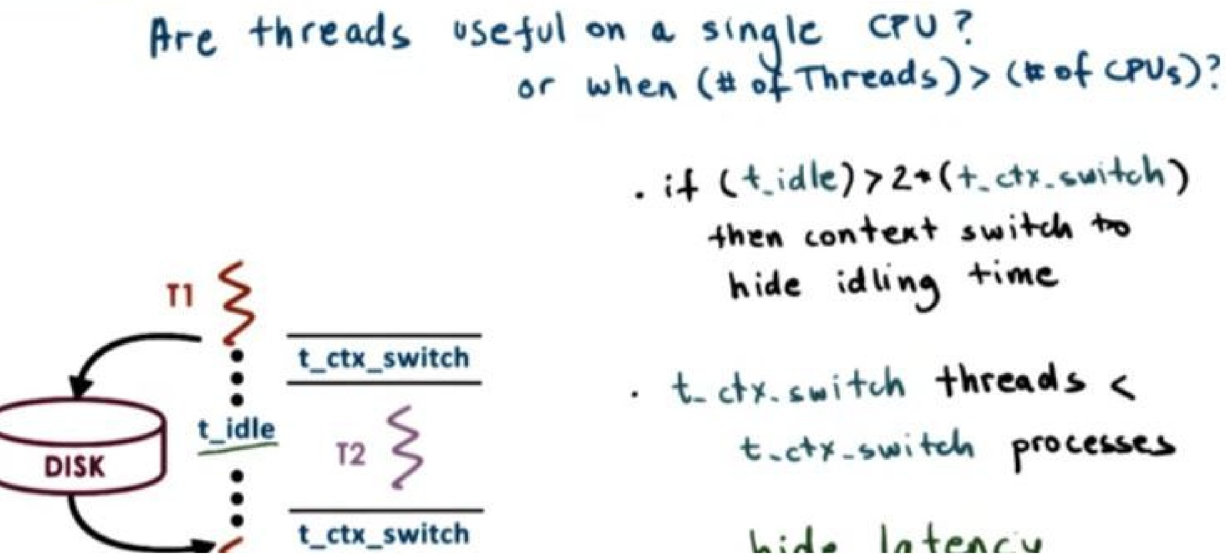
This implies that a multithreaded application is more memory efficient, it has lower memory requirements, than its multiprocess alternative (amit - comparison between the two).

As a result of that the application is more likely to fit in memory and not require as many swaps from disk compared to a multiprocess alternative (amit - more caching, less swapping).

Another issue is that communicating data, passing data, among processes, or synchronization among processes, requires IPC mechanisms that are more costly. As we'll see later in this lecture **communication and synchronization among threads in a single process is performed via shared variables** in that same process' AS so it does not require that same costly IPC. In summary, ***multithreaded programs are more efficient in their resource requirements than multiprocess programs and incur a lower overhead for their inter-thread communication than the corresponding IPC***.

05 - Benefits of Multithreading Single CPU

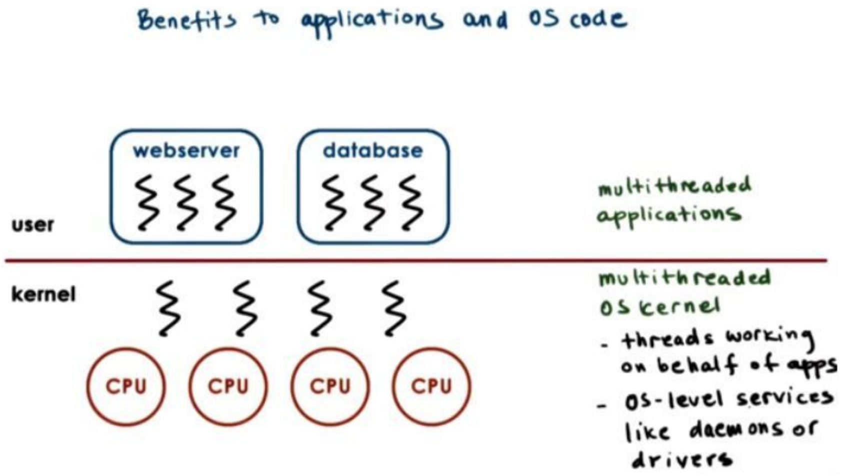
One question that you can ask is also: Are threads useful on a single CPU, or even more generally are threads useful when the number of threads is greater than the number of CPUs? To answer this question, let's consider a situation where a single thread T1 make a disk request. As soon as the request comes in the disk needs some amount of time to move the disk spindle to get to the appropriate data and



respond to the request. Let's call this time T_{idle}. During this time, T1 has nothing to do but wait, so the CPU is idle and does nothing. ***If this idle time is sufficiently longer than the time it takes to make a context switch then it starts making sense to perhaps context switch from T1 to some other thread T2 and have that thread do something useful*** ***(amit - so think in terms of context switching time required as well)***. Or specifically rather, we need the EC that's waiting on some event to be waiting for an amount of time that's longer than really two context switches so that it would make sense to switch to another thread and have that thread perform some operation and then switch back.

So basically ***as long as the time to context switch T_{ctx_switch} is such that T_{idle} is greater than twice the time to context switch it makes sense to context switch to another thread and hide the idling time***. Now this is true for both processes and threads, ***however, recall from the last lesson that we said that one of the most costly steps during the context switch is the time that's required to create the new virtual to physical address mappings for the new process that will be scheduled (amit - i dint realize this point but good to know)***. **Given that threads share an AS, when we're context switching among threads it is not necessary to re-create new virtual to physical address mappings.** So in the threads case, this costly step is avoided. **For this reason, the time to context switch among threads is less than the time to context switch among processes. The shorter the context switching time is, there will be more of these T_{idle} situations when a thread is idling where it will make sense to context switch to another thread and hide the wasted idling time.** Therefore, a multithreading is especially useful because it will allow us to hide more of the latency that's **associated with I/O operations.** And this is useful even on a single CPU.

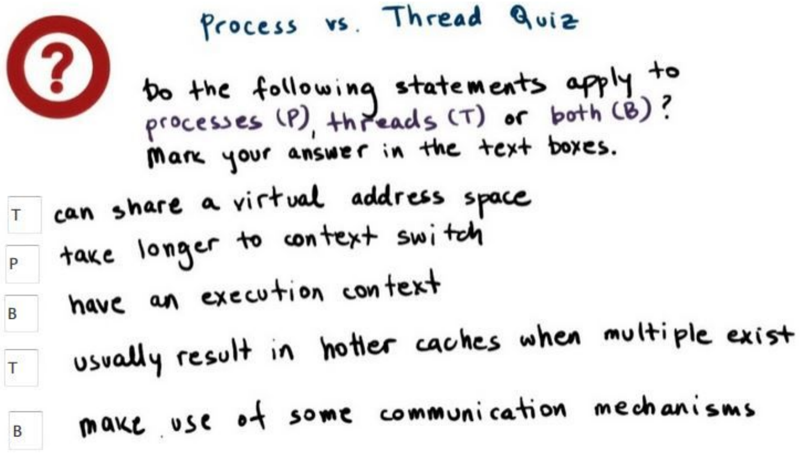
06 - Benefits of Multithreading Apps and OS


Benefits to applications and OS code
webserver
database
user
kernel
CPU
CPU
CPU
CPU
multithreaded applications
multithreaded OS kernel
- threads working on behalf of apps
- OS-level services like daemons or drivers

There are benefits of multithreading both to applications, when we have multithreaded applications, but also to the OS itself. ***By multithreading the OS's kernel we allow the OS to support multiple EC, and this is particularly useful when there are in fact multiple CPUs, so that the OS context can execute concurrently on different CPUs in a multiprocessor/multicore platform* *(amit - takeaway - multithreading is good for everyone)***. The OS's threads may run on behalf of certain applications or they may also run some OS level services, like certain daemons or device drivers.

07 - Process vs Threads Quiz

To make sure we understand some of the basic differences between a process and a thread, let's take a look at a quiz. You'll need to answer if the following statements apply to processes, threads, or both. And please mark your answers in the text boxes. The first statement is, can share a virtual address space. Next, take longer to context switch. The third one, have an execution context. The fourth one, usually result in hotter caches when multiple such entities exist. And then the last statement, make use of some communication mechanisms.



Process vs. Thread Quiz

Do the following statements apply to processes (P), threads (T) or both (B)? Mark your answer in the text boxes.

T can share a virtual address space

P take longer to context switch

B have an execution context

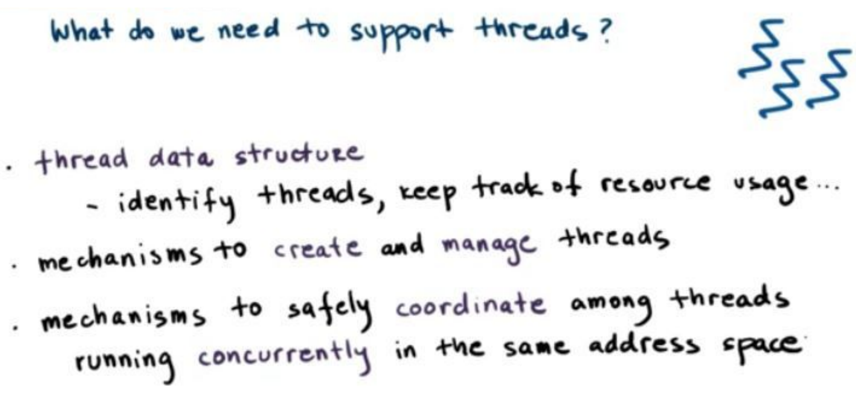
T usually result in hotter caches when multiple exist

B make use of some communication mechanisms

08 - Process vs Threads Quiz

The first statement applies to threads. Each thread belonging to a process shares the virtual address space with other threads in that process. And because threads share an address space, the context switch among them happens faster than processes. So, processes take longer to context switch. Both threads and processes have their execution context described with stack and registers. Because threads share the virtual address space, it is more likely that when multiple threads execute concurrently, the data that's needed by one thread is already in the cache, brought in by another thread. So, they typically result in hotter caches. Among processes, such sharing is really not possible. And then the last answer is B. We already saw that for processes, it makes sense for the operating system to support certain interprocess communication mechanisms. And we'll see that there are mechanisms for threads to communicate and coordinate and synchronize amongst each other.

09 - Basic Thread Mechanisms

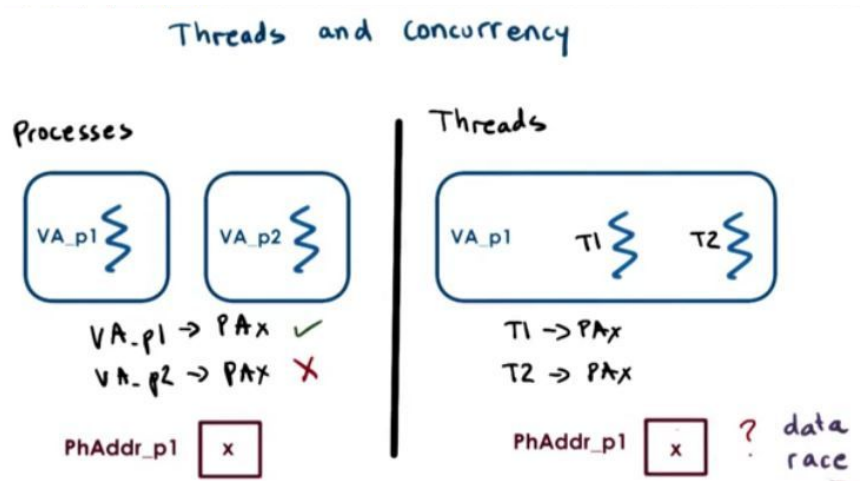


What do we need to support threads?

- thread data structure
  - identify threads, keep track of resource usage...
- mechanisms to create and manage threads
- mechanisms to safely coordinate among threads running concurrently in the same address space

Now that we know what threads are, what is it that we need in order to support them? ***First, we must have some data structure that will allow us to distinguish a thread from a process. This data structure should allow us to identify a specific thread and to keep track of their resource usage***. Then we must have some mechanisms to create and to manage threads. In addition, we also need mechanisms to allow threads to coordinate amongst each other. Especially when there are certain dependencies between their execution when these threads are executing concurrently. For instance, we need to make sure that threads that execute concurrently don't overwrite each other's inputs or each other's results. Or we need mechanisms that allow one thread to wait on results that should be produced by some other thread.1

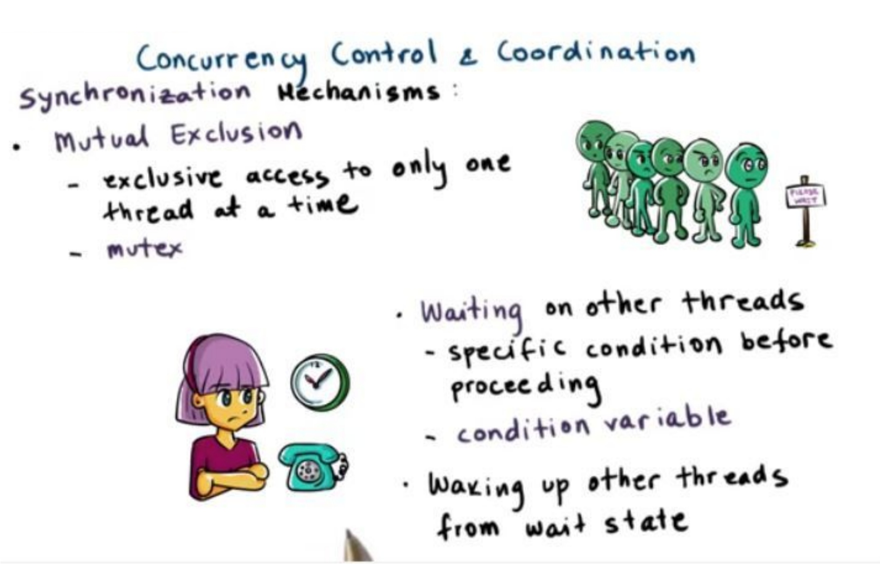
Well, when thinking about the type of coordination that's needed between threads, we must first think about the issues associated with concurrent execution. Let's first start by looking at processes. When processes run concurrently, they each operate within their own address space. The **operating system** together with the underlying hardware **will make sure that no access from one address space is allowed to be performed on memory that belongs to another.** Memory is, or state that belongs to the other address space. For instance, consider a physical address x that belongs to the process one address space. In that case, the mapping between the virtual address for p1 in process one and the physical address of x will be valid. Since the operating system is the one that establishes these mappings, **the operating system will not have a valid mapping for any**



address from the address space of p2 to x. So from the p2 address space, we simply will not be able to perform a valid access to this physical location.

Threads, on the other hand, share the same virtual-to-physical address mappings**.** So both T1 and T2, concurrently running as part of an address space, can both legally perform access to the same physical memory. And using the same virtual address on top of that. **But this introduces some problems.** If both T1 and T2 are allowed to access the data at the same time and modify it at the same time, then this could end up with some inconsistencies. One **thread may try to read the data**, **while the other one is modifying** it, so we just read some garbage. Or both threads are trying to update the data at the same time and their updates sort of overlap. ***This type of data race problem where multiple threads are accessing the same data at the same time is common in multithreaded environments, where threads execute concurrently***.

To **deal with these concurrency issues, we need mechanisms for threads to execute in an exclusive manner. We call this mutual exclusion.** Mutual exclusion is a mechanism where only one thread at a time is allowed to perform an operation. The remaining threads, if they want to perform the same operation, must wait their turn. The actual operation that must be performed in mutual exclusion may include some update to state or, in general, access to some data structure that's shared among all these threads. For this, Birrell and other threading systems, use what, what's called ***mutexes***. In addition, it is also **useful for threads to have a mechanism to wait on one another. And to exactly specify what are they waiting for.**


Concurrency Control & Coordination
Synchronization Mechanisms :
• Mutual Exclusion
- exclusive access to only one thread at a time
- mutex
• Waiting on other threads
- specific condition before proceeding
- condition variable
• Waking up other threads from wait state

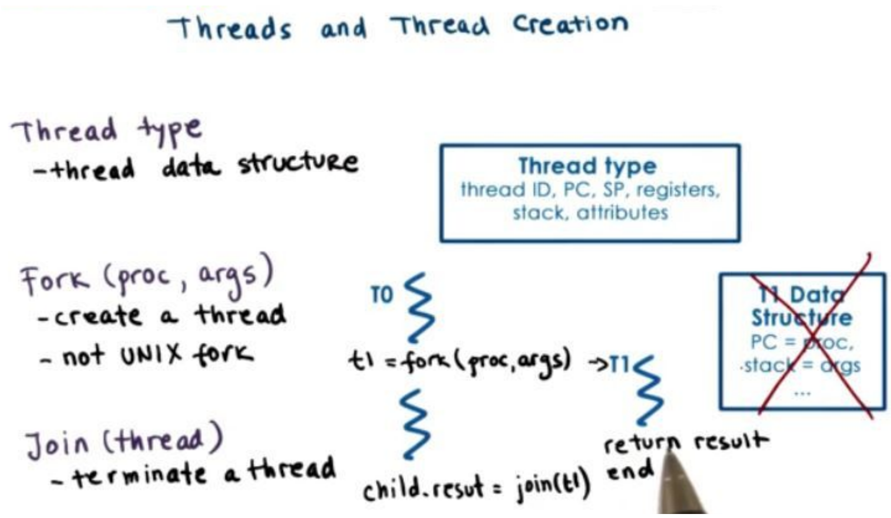
For instance a thread that's dealing with shipment processing must wait on all the items in a certain order to be processed before that order can be shipped. So it doesn't make sense to repeatedly check whether the remaining threads are done filling out the order. The thread just might as well wait until it's explicitly notified that the order is finalized so that it can at that point get up, pick up the order, and ship the package. Birrell talks about using so-called ***condition variables*** to handle this type of inter-thread coordination. We refer to both of these mechanisms as synchronization mechanisms. For completeness, Birrell also talks about mechanisms for waking up other threads from a wait state, but in this lesson, we will focus mostly on thread creation and these two synchronization mechanisms, ***mutexes and condition variables***. We will discuss this issue a little bit more in following lessons.

10 - Thread Creation

Let’s first look at how threads should be represented by an OS or a system library that provides multithreading support and also what is necessary for thread creation. Remember during this lesson we will base our discussion on the primitives that are described and used in Birrell’s paper. These don’t necessarily correspond to some interfaces that are available in real threading systems or programming languages and in our next lesson we will talk about pthreads which is an example of a threading interface supported by most modern OSes. So that will make the discussion a little bit more concrete. You can think of this lesson and the content of Birrell’s paper as explaining this content at a more fundamental level.

First, we need some data structure to represent a thread, actually. The thread type proposed by Birrell is a data structure that contains all information that’s ***specific to a thread and can describe a thread. This includes the thread identifier that the threading system will use to identify a specific thread, register values, in particular the stack of a thread, the program counter and the stack pointer, and any other thread specific data or attributes (amit - since they have ind EC, their DS should have things required for EC)***. These additional attributes, for instance, could be used by the thread management systems so that it can better decide how to schedule threads or how to debug errors with threads or other aspects of thread management.

For thread creation, Birrell proposes a ***fork*** call with two parameters, a proc argument, and that is the procedure that the created thread will start executing, and then args, which are the arguments for this procedure. ***This fork should not be confused with the UNIX system call fork that we previously discussed***. The UNIX system call fork creates a new system process that is an exact copy of the calling process and here fork creates a new thread that will execute this procedure with these arguments.



When a thread T0 calls a fork, a new thread T1 is created. That means that a new thread data structure of this type is created and its fields are

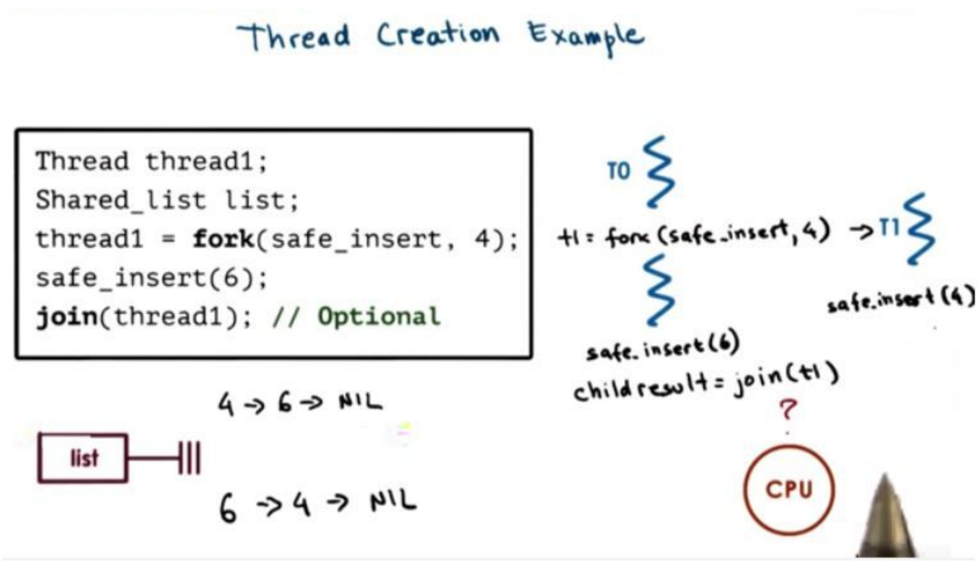
initialized such that its program counter will point to the first instruction of the procedure proc and these arguments will be available on the stack of the thread (amit - so question can be what really happens when we create a thread and this is our answer - two things, initialization of tis DS and PC). After the fork operation completes the process as a whole has two threads, T0 the parent thread, and T1, and these can both execute concurrently. T0 will execute the next operation after the fork call and T1 will start executing with the first instruction in proc and with the specified arguments.

So what happens when T1 finishes? Let's say it computed some result as a result of running proc and now somehow it needs to return that result or it may be just some status of the computation like success or error.

One programming practice would be to store the results of the computation in some well defined location in the address space that's accessible to all the threads and have some mechanism to notify either the parent or some other thread that the result is now available. More generally, however, we need some mechanism to determine that a thread is done and if necessary to retrieve its result or at least to determine the status of the computation, the success or error of its processing.

For instance, we can have an application where the parent thread does nothing but create a bunch of child threads that process different portions of an input array. The parent will still have to wait until all of its children finish the processing before it can exit so as not to force their early termination, for instance. To deal with this issue, Birrell proposes a mechanism he calls ***join***. It has the following semantic. When the parent thread calls join with the thread ID of the child thread, it will be blocked until the child completes. Join will return to the parent the result of the child's computation. At that point, the child for real exits the system any allocated data structure state for the child, all of the resources that were allocated for its execution will be freed and the child, at that point, is terminated. You should note that other than this mechanism where the parent is the one that's joining the child, in all other aspects the parent and the child thread are the completely equivalent; they can all access all resources that are available to the process as a whole and share them. And this is true both with respect to the hardware resources, CPU, memory, or actual state within the process.

11 - Thread Creation Example



Thread Creation Example

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

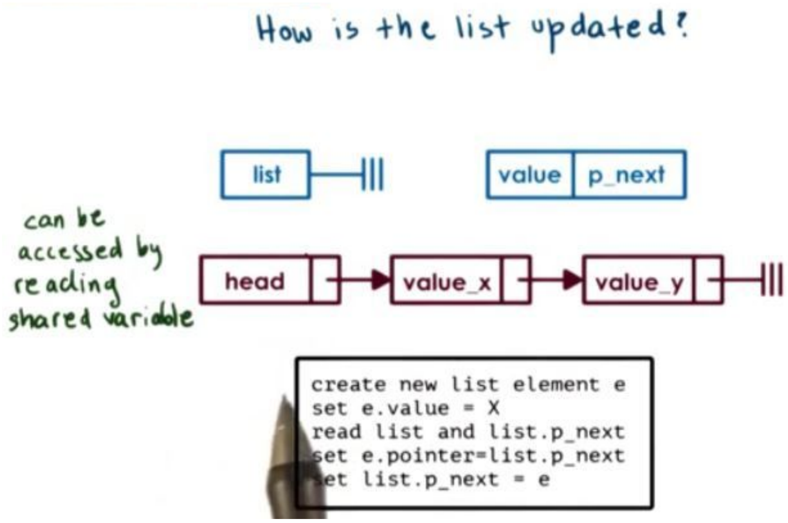
Here is a code snippet illustrating thread creation.

Two threads are involved in this

system, the parent thread that is executing this code and the child that gets created via this fork. Both threads perform this operation safe_insert, which manipulates some shared list that's initially empty, let's say. Let's assume that initially the process begins with one parent thread T0 in this case. At some point T0 calls fork and creates a child thread T1. Now T1 will need to execute safe_insert with an argument 4. As soon as thread T1 is created the parent thread continues its execution and at some point it will reach the point where it calls safe_insert with argument 6 in this case. So it's trying to insert the element 6 into the list. Because these threads are running concurrently and are constantly being switched when executing on the CPU, the order in which these safe_insert operations on the parent and the child threads is not clear. It is not guaranteed that when this fork operation completes the execution will actually switch to T1 and will allow T1 to perform its safe_insert before T0 does, or if after the fork, although the thread is created, T0 will continue and the safe_insert with argument 6 that T0 performs will be the first one to be performed. So as a result, both the list may have
a state where the child completes its safe_insert before the parent or the other way around, the parent completes before the child does. Both of these are possible executions.

Finally, the last operation in this code snippet is this join. So we're calling join with T1, if this join is called when it (T1) has actually completed, it will return immediately. If this join occurs while T1 is still executing, the parent thread will be blocked here until T1 finishes the end of the safe_insert operation. In this particular example the results of the child processing are available through this shared list. So really the join isn't a necessary part of the code. We will be able to access the result of the child thread regardless.

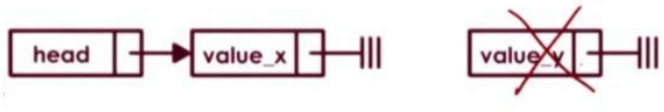
11 - Mutexes



```
create new list element e
set e.value = X
read list and list.p_next
set e.pointer=list.p_next
set list.p_next = e
```

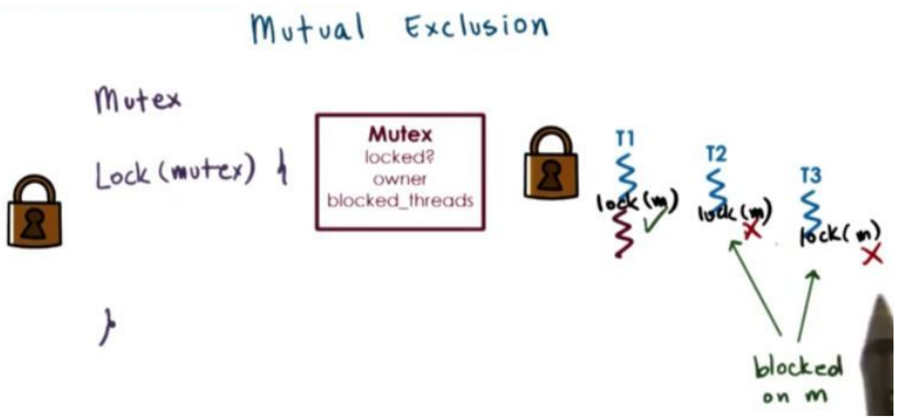
From the previous example where the threads were inserting elements in the list, how does this list actually get updated? A naive implementation may look as follows. Each list element has two fields, a value field and then a pointer that points to the next element in the list. We call this one p_next. The first element we call of the list head; it can be accessed by reading the value of the shared variable list. So in this code sample this is where this is happening, read list, as well as its list pointer element. Then each thread that needs to insert a new g element in the list will first create an element and set its value, and will then have to read the value of the head of the list, so this element list, and then it will have to set its pointer field to point to whatever is still in the list, and set the head of the list to point to the newly created element. What that means is for instance, when we are creating this element value_x, we will first create this data structure, and then we will first read the pointer of the list, originally this pointed to the list value_y, set the new elements pointer to point to value_y, and then set the head pointer to actually point to the newly created element. And in this way, new elements are basically inserted at the head of the list, they end up pointing to the rest of the list, and the head of the list points to the newly created element.

Clearly there is a problem if two threads that are running concurrently on ***two different CPUs at the same time try to update the pointer field of the first element in the list. We don't know what will be the outcome of this operation if two threads are executing it at the same time, and trying to set different values in the p_next field. There is also a problem if two threads are running on the CPU at the same time because there are operations are randomly interleaved (amit - question - in this case does OS keeps shared variable in the cache of any of the processor or it is always in main memory???). For instance, they may both read the initial value of the list, so this is where they are both reading the value of the list, and its pointer to the next element of the list, in this case null. They will both set the pointers p_next in their elements to be null, and then they will take turns setting the actual list pointer to point to them. Only one element will ultimately be successfully linked to the list and the other one will simply be lost.***

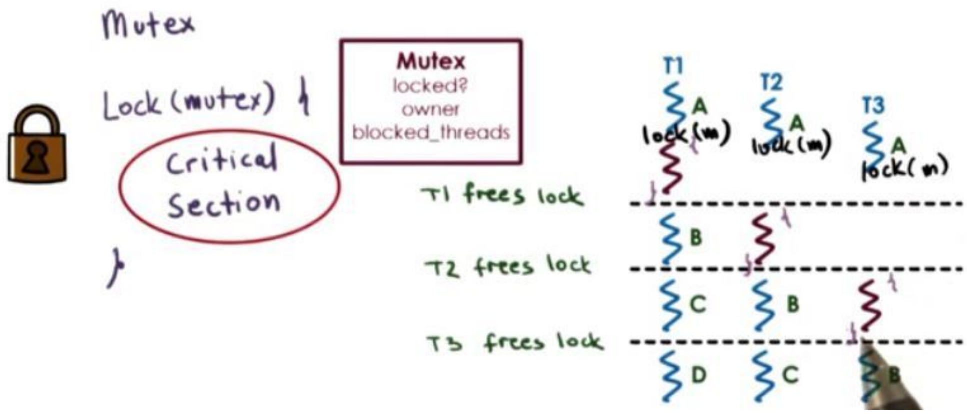


12 - Mutual Exclusion

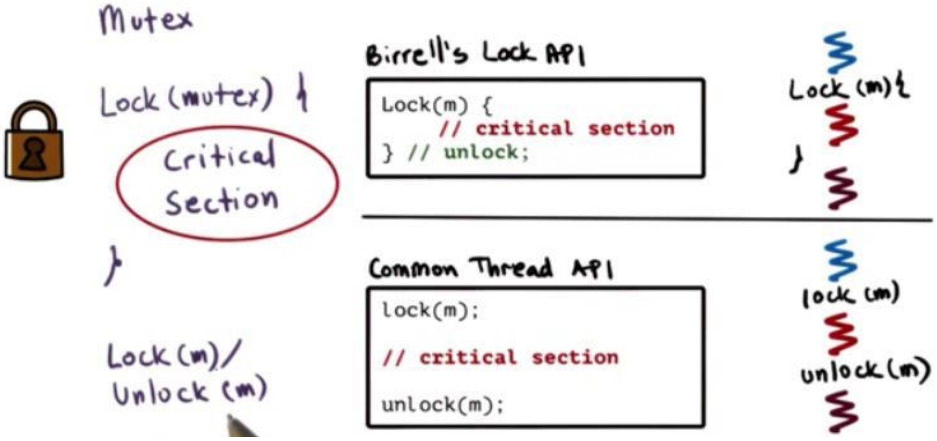
There is a danger in the previous example that the parent and the child thread will try to update the shared list at the same time potentially overwriting the list elements. This illustrates a key challenge with multithreading: that there is a need for a mechanism to enable ***mutual exclusion*** among the execution of concurrent threads. To do this OSes and threading libraries in general support a construct called ***mutex***. A mutex is like a lock that should be used whenever accessing data or state that is shared among threads. When a thread locks a mutex, it has exclusive access to the shared resource. Other threads attempting to lock the same mutex are not going to be successful. We also use the term "acquire the lock" or "acquire the mutex" to refer to this operation. So these unsuccessful threads, they will be what we call



blocked on the lock operation, meaning they'll be suspended here, they will not be able to proceed until the mutex owner, the lock holder releases it. ***This means that as a data structure the mutex should at least contain information about its status, is it locked or free, and it will have to have some type of list***, that doesn't necessarily have to be guaranteed to be ordered, but it has to be some sort of list of all the threads that are blocked on the mutex and are waiting for it to be freed. ***Another common element that's part of this mutex data structure is maintain some information about the owner of the mutex, who currently has the lock (amit - lets say we are creating DS for a mutex, what all will we need to add here??)***. A thread that has successfully locked the mutex has exclusive rights to it and can proceed with its execution. {mutex: status/list/info of current thread}



In this example, T1 gets access to the mutex and thus continues executing. Two other threads T2 and T3 that are also trying to lock the mutex will not be successful with that operation, they will be blocked on the mutex and they will have to wait until T1 releases the mutex. The portion of the code protected by the mutex is called ***critical section***. In Birrell's paper, this is any code within the curly brackets of the lock operation that he proposes should be used with mutexes. The critical section code should correspond to any kind of operation that requires that only one thread at a time performs this operation. For instance, it can be an update to a shared variable like the list, or incrementing/decrementing a counter, or performing any type of operation that requires mutual exclusion between the threads. Other than the critical section code, the rest of the code in the program, threads may execute concurrently. So using the same example, imagine that the three threads need to execute a sequence of code of block A, followed by the critical section and then some other portions of code blocks B, C, and D. All the code blocks with a letter can be executed concurrently, the critical sections, however, can be executed only by one thread at a time. So thread T2 cannot start the critical section until after thread T1 finishes it, thread T3 cannot start its critical section until any prior thread exits the critical section or releases the lock. All of the remaining portions of the code can be executed concurrently. So in summary, the threads are mutually exclusive with one another with respect to their execution of the critical section code. In the lock construct proposed by Birrell, again the



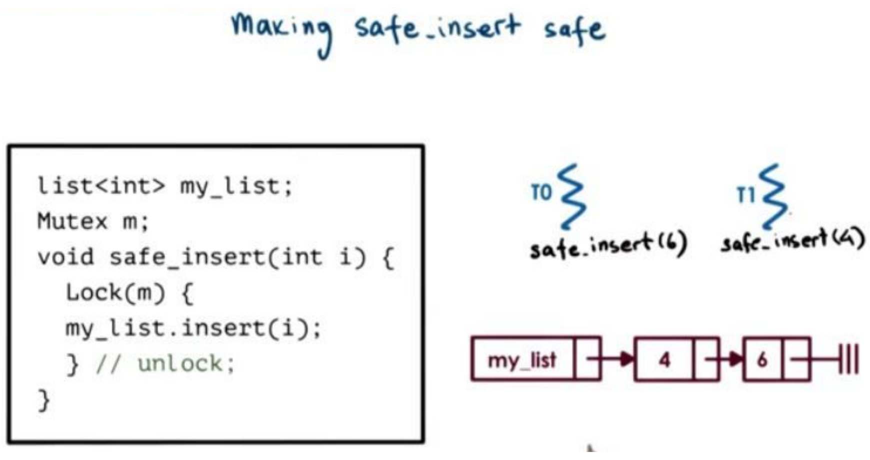
```
Lock(m) {
    // critical section
} // unlock;
```

```
lock(m);

// critical section

unlock(m);
```

critical section is the code between the curly brackets, and the semantics are such that upon acquiring a mutex a thread enters the lock block and when exiting the block with the closing of the curly bracket, the owner of the thread releases this mutex, frees the lock. So the critical section code in this example, that's the code between the curly brackets and the closing of the curly bracket implicitly frees the lock. When a lock is freed, at this point, any one of the threads that are waiting on the lock, or even a brand new thread just reaching the lock construct, can start executing the lock operation. For instance, if T3's lock statement coincides with release of the lock of T1, T3 may be the one to be the first one to execute, to acquire the lock and execute the critical section, although T2 was already waiting on it.

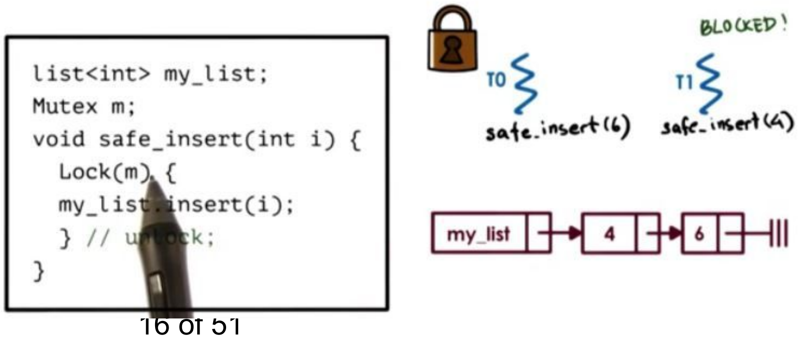
We will use Birrell's lock construct throughout this lecture, however, you should know that most common APIs have two separate call, lock and unlock. So in contrast to Birrell's lock API in this other interface, the ***lock/unlock interface***, both the lock and unlock operations must be used both explicitly and carefully when requesting a mutex and when accessing a critical section. What that means is that you must explicitly lock a mutex before entering a critical section and then we also must explicitly unlock the mutex when we finish the critical section. Otherwise, nobody else will be able to proceed.

13 - Mutex Example



```
list<int> my_list;
Mutex m;
void safe_insert(int i) {
  Lock(m) {
  my_list.insert(i);
  } // unlock;
}
```

Returning to the previous safe_insert example, let's demonstrate how to use of a mutex making the operation safe_insert actually safe. Just like in the threads creation code we have threads T0 and T1 and they both want to perform the safe_insert operation. The parent thread T0 wants to perform safe_insert with an element 6, and the child thread wants to perform safe_insert with a value of 4. Let's assume that once the parent created the child T1, it (T0) continued executing and was the first one to reach safe_insert with value 6. It will acquire the lock and start inserting the element 6 on the list. What that means, T0 has the lock and when T1, the child, reaches the safe_insert operation it will try to acquire the lock as well and it will not be successful, it will be blocked. At some later point, T0 will release the lock, T1 will acquire the lock, and then T1 will be able to insert its element onto the front of the list. So in this case, the final ordering of the list will be 4 followed by 6. Since we're always inserting in the front of the list. That's how we describe this operation. So, the parent inserted its element first, and then the child thread was the second one to insert the value 4 into the list.



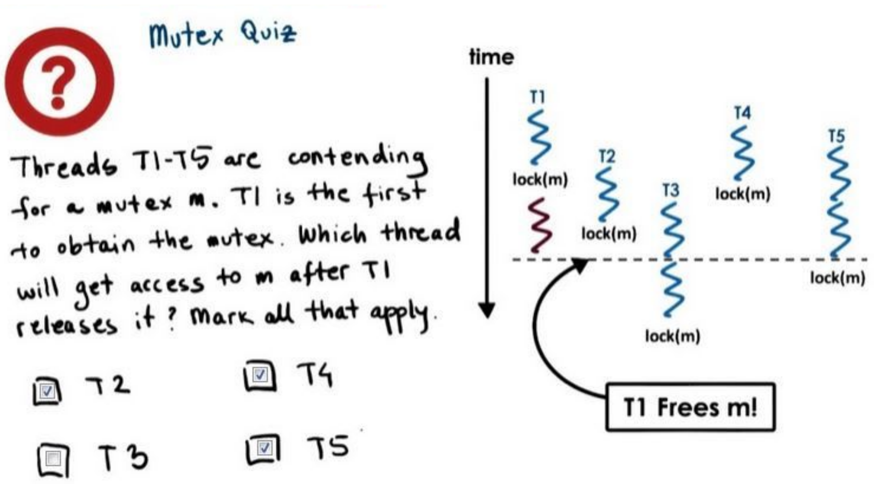
```
list<int> my_list;
Mutex m;
void safe_insert(int i) {
  Lock(m) {
  my_list.insert(i);
  } // unlock;
}
```

14 - Mutex Quiz

Let’s do a quiz now. Let’s take a look at what happens when threads contend for a mutex. For this quiz we will use the following diagram. We have five threads, T1 through T5, who want access to a shared resource, and the mutex m is used to insure mutual exclusion to that resource. T1 is the first to get access to the mutex, and the dotted line corresponds to the time when T1 finishes the execution of the critical section and frees m. The time when the remaining threads issue their mutex requests correspond to the lock(m) positions along this time axis. For the remaining threads which thread will be the one to get access to the mutex after T1 releases control? Your choices are T2, T3, T4, or T5, and you should mark all that apply.

14.b - Mutex Quiz

Looking at the diagram, both T2 and T4 have attempted to get the lock before it was released, so their requests will definitely be in the queue that’s associated with the mutex, the queue of pending requests. ***Any one of these two requests could be one of the requests that can get to execute first (amit - basically there is no defined order, anyone can run)***. The specification of the mutex doesn’t make any guarantees regarding the ordering of the lock operations, so it doesn’t really matter that T4 issued the lock request, the lock operation before T2, we don’t have a guarantee that these requests will be granted in order. But regardless, both T2’s and T4’s requests are pending on the mutex, then either one of these two threads would be viable candidates of who gets to execute next after T1 releases control. Thread T3 is definitely not a likely candidate since it doesn’t get to issue the lock operation until after T1 released it and there are already pending requests, so it is not going to be one of the next threads to execute. For T5, its a little tricky. From the diagram we see that the lock is released just as ***T5 is starting to perform the lock operation, so what can happen is T1 releases the lock and then we see that both T2 and T4 are actually pending on it, but just before we give the lock to one of these two threads on another CPU say T5 arrives makes a lock request, the lock is still free at that particular time and so T5 is the one that actually gets the lock (amit -basically it is not a FIFO queue***. So it is the one that gets to execute



Mutex Quiz

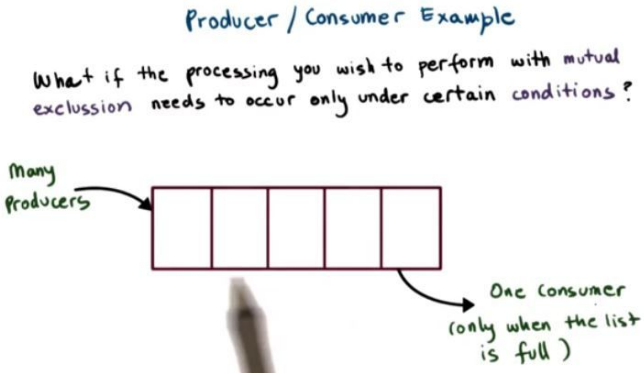
Threads T1-T5 are contending for a mutex m. T1 is the first to obtain the mutex. Which thread will get access to m after T1 releases it? Mark all that apply.

☑ T2 ☑ T4

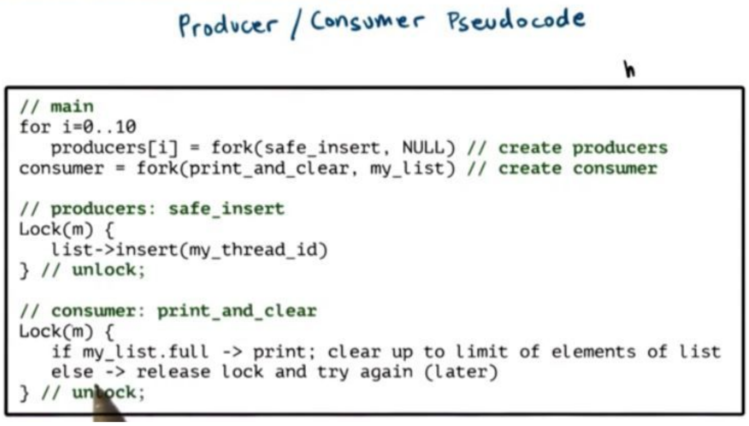
□ T3 ☑ T5

next. So either one of these, T2, T4, or T5 is a viable candidate of which one of the threads is going to get to execute after T1 releases the lock.

15 - Producer / Consumer Example


Producer / Consumer Example
What if the processing you wish to perform with mutual exclussion needs to occur only under certain conditions?
Many Producers
One Consumer
(only when the list is full)

For threads the first construct that Birrell advocates is mutual exclusion, and that's a binary operation. A resource is either free or you can access it, or its locked and busy and you have to wait. Once the resource becomes free you get a chance to try access the resource. However, what if the processing that you wish to perform needs to occur only under certain circumstances, under certain conditions? For instance, what if we have a number of threads that are inserting data to a list, these are producers of list items, and then we have one special consumer thread that has to print out and then clear the contents of this list once it reaches a limit, once, for instance, it is full. We'd like to make sure that this consumer thread only gets to execute its operation under these certain conditions when the list is actually full.

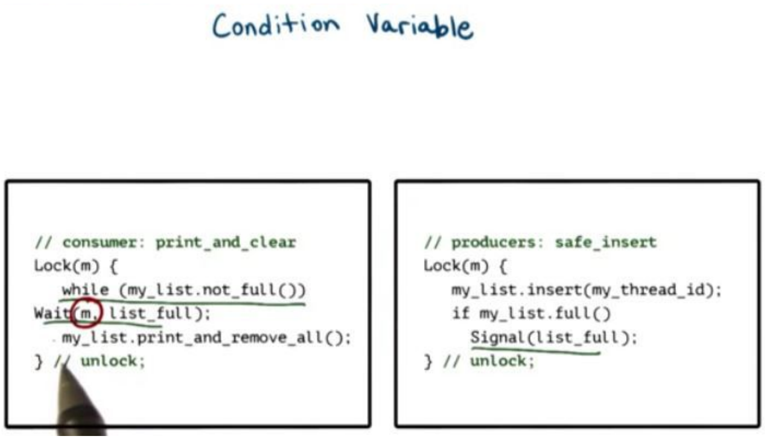


```
// main
for i=0..10
   producers[i] = fork(safe_insert, NULL) // create producers
consumer = fork(print_and_clear, my_list) // create consumer

// 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;
```

Let's look at this producer/consumer pseudocode example. In the main code, we create several producer threads and then one consumer thread. All of the producer threads will perform the safe_insert operation and the consumer thread will perform an operation print_and_clear the list. And this operation, as we said, needs to happen when once list is full only. For producers, the safe_insert operation is slightly modified from what we saw before. Here we don't specify the argument to safe_insert. When the producer thread is created, instead every single one of the threads needs to insert an element that has a value of the thread identifier. For the consumer thread here, it continuously waits on the lock and when the mutex is free it goes and checks if the list is full, if so it prints and clears up the elements of the list, otherwise it immediately releases the lock and tries to reacquire it again. Operating this way is clearly wasteful and it would be much more efficient if we could just tell the consumer when the list is actually full so that it can at that point go ahead and process the list.

16 - Condition Variable

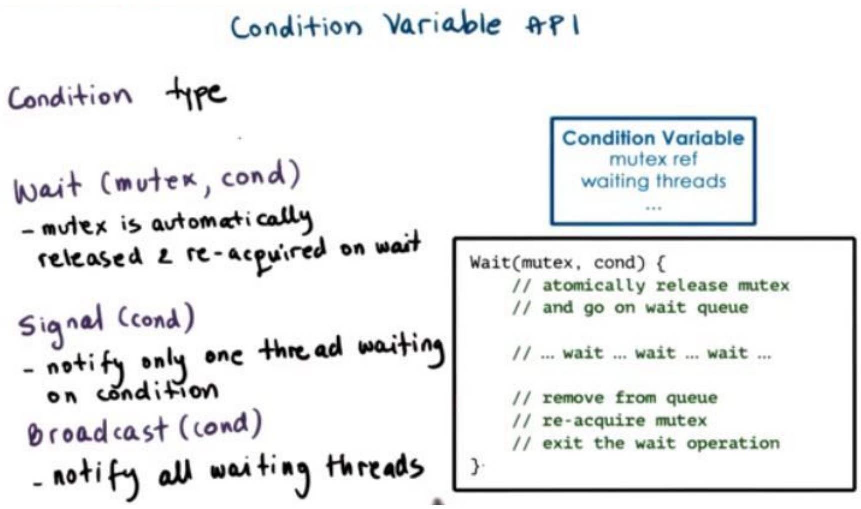
Birrell recognizes that this common situation in multithreaded environments and argues for a new construct, a ***condition variable***. He says that such condition variables could be used in conjunction with mutexes to control the behavior of concurrent threads. In this modified version of the producer/consumer code, the consumer locks the mutex and checks and if the list is not full it then suspends itself and then waits for the list to become full. The producers, on the other hand, once they insert an element into the list, they check to see whether that insertion resulted in the list becoming full, and only if that is the case, if the list is actually full will they signal that the list is full. This signal operation is clearly intended for whomever is waiting on this particular list_full notification, in this case the consumer. Note that while the consumer is waiting the only way that this predicate that it is waiting on can change, so that the only way that the list can become full, is if a producer thread actually obtains the mutex and inserts an element onto the list ***(amit - check in producer that when the list is full, it signals. This might not be true...we need to wait. TODO - check video again)***. What that means is that the semantics of the wait operation must be such that this mutex, that was acquired when we got to this point, has to be automatically released when we go into wait statement, and then automatically re-acquired once we come out of the wait statement, because we see that after we come out of the wait statement we actually need to modify the list and this mutex m was protecting our list. So when a consumer see that it must wait, it specifies the condition variable that it must wait on, and the wait operation takes as parameters both the condition variables as well as this mutex. Internally the implementation of the wait must insure the proper semantics, so it must insure that the mutex is released and then when we are actually removed from the wait operation, when this notification is received, we have to re-acquire the mutex. At that point the consumer, so it has the mutex, it’s allowed to modify the list, so to print its contents and remove its contents, and then the consumer will reach this curly bracket, this unlock operation, and at that point the mutex will indeed be released.



```
// 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;
```

17 - Condition Variable API



Condition Variable API

Condition type

Wait (mutex, cond)
- mutex is automatically released 2 re-acquired on wait

Signal (cond)
- notify only one thread waiting on condition

Broadcast (cond)
- notify all waiting threads

Condition Variable
mutex ref
waiting threads
...

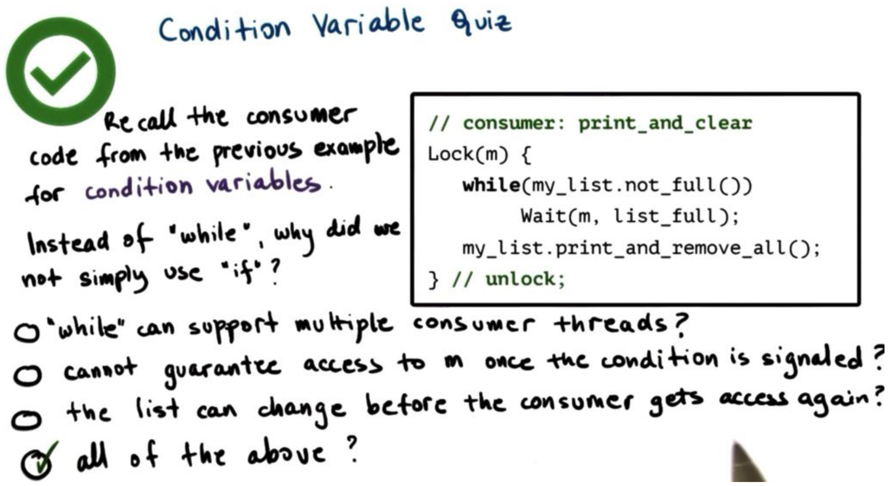
```
Wait(mutex, cond) {
    // atomically release mutex
    // and go on wait queue

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

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

To summarize, a common condition variable API will look as follows. First we must be able to create these data structures that correspond to the condition variables, so there must be some type that corresponds to condition variables. Then there is the wait construct that takes as arguments the mutex and a condition variable where a mutex is automatically released and reacquired if we have to wait for this condition to occur. When a notification for a particular condition needs to be issued it would be useful to be able to wake up the threads one at a time and for this Birrell proposes a signal API or it could also be useful to wake up all the threads that are waiting on a particular condition and for this Birrell proposes a broadcast API. ***The condition variable as a data structure, it must have basically a list of the waiting threads that should be notified in the event the condition is met, it also must have a reference to the mutex that is associated with the condition so that this wait operation can be implemented correctly so that this mutex can both be released and reacquired as necessary.*** As a reference, the way the wait operation would be implemented in a OS or in a threading library that support threads and support this condition variable, it would mean that in the implementation the very first thing that happened is that the mutex that's associated with the condition variable is released and the thread is placed on the wait queue of waiting threads, and then at some point when a notification is received what would have to happen is the thread is removed from the queue and this mutex is reacquired and then only then do we exit the wait, so only then will a thread return from this wait operation. One thing to note here is that when one a signal or broadcast we remove a thread from the queue and then the first thing that this thread needs to do is to re-acquire the mutex. So what that really also means is that on broadcast, although we are able to wake up all of the threads at the same time the mutex, it requires mutual exclusion so they will be able to acquire the mutex only one thread at a time, so only one thread at a time will be re-acquiring the mutex and exiting this wait operation. So it’s a little bit unclear that it's not always useful to broadcast if you’re really not going to be able to do much work once you wake up with more than one thread at a time.

18 - Condition Variable Quiz

For this quiz, let's recall the consumer code from the previous example for condition variables. Instead of while, why didn't we simply use if? Is it because, while can support multiple consumer threads? We cannot guarantee access to m once the condition is signaled? The list can change before the consumer gets access again? Or, all of the above?

18.b - Condition Variable Quiz



Condition Variable Quiz

Recall the consumer code from the previous example for condition variables.

Instead of "while", why did we not simply use "if"?

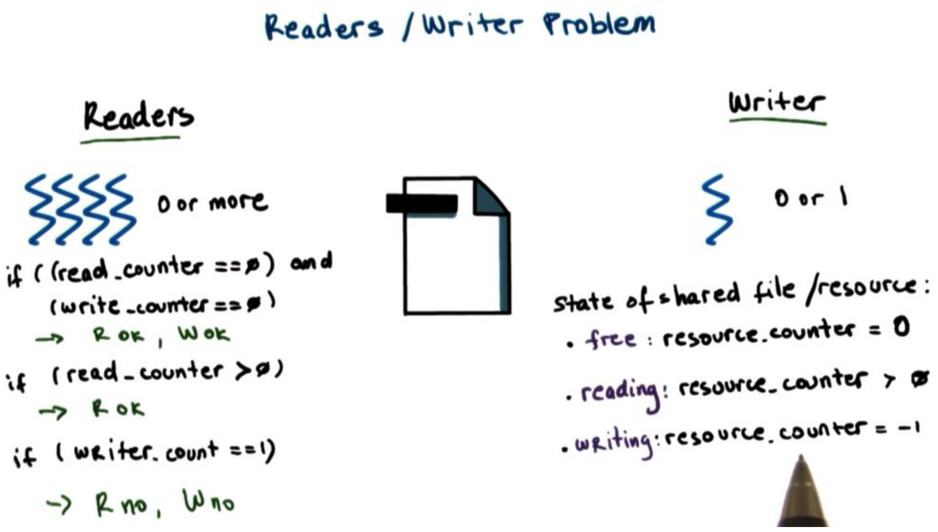
```
// consumer: print_and_clear
Lock(m) {
   while(my_list.not_full())
        Wait(m, list_full);
   my_list.print_and_remove_all();
} // unlock;
```

○ "while" can support multiple consumer threads?
○ cannot guarantee access to m once the condition is signaled?
○ the list can change before the consumer gets access again?
✓ all of the above ?

Hints:
- The list could change in an instant, so shouldn't we check the condition again as soon as we receive a signal?
- Remember, a scheduler can quickly swap (consumer) threads!

The correct answer is, all of the above. For instance, when we have multiple consumer threads, one consumer thread is waiting. It wakes up from the Wait statement because the list is full. However, before it gets to actually process that list, we acquired the mutex and processed that list. Newly arriving other consumers, they will acquire the mutex, see that the list is actually full, and print and remove its contents. So, the one consumer that was waiting on the list when the signal or broadcast notification arrived, when it comes out of the wait, the list, its state has already changed. So, we need the while in order to deal with situations where there are multiple of these consumer threads, multiple threads that are waiting on the same condition. This is related, also, to the second statement in that when we signal a condition, we have no way to guarantee the order in which the access to the mutex will be granted. So, there may be other threads that will require this mutex before you get to respond on the fact that a condition has been signaled. And so in that regard, if you cannot control the access to the mutex and guarantee it, you have no way to guarantee that the value of this list variable will not change before the consumer gets to have access to it again. So, all of these three factors contribute to the fact that we have to use while in order to make sure that when we wake up from a while statement, indeed this condition that we were waiting on, it is met.

19 - Reader / Writer Problem


Readers / Writer Problem
Readers
0 or more
if ((read_counter == ∅) and
(write_counter == ∅)
→ R ok, W ok
if (read_counter > ∅)
→ R ok
if (writer.count == 1)
→ R no, W no
Writer
0 or 1
State of shared file /resource:
• free : resource.counter = 0
• reading: resource_counter > ∅
• writing: resource.counter = -1

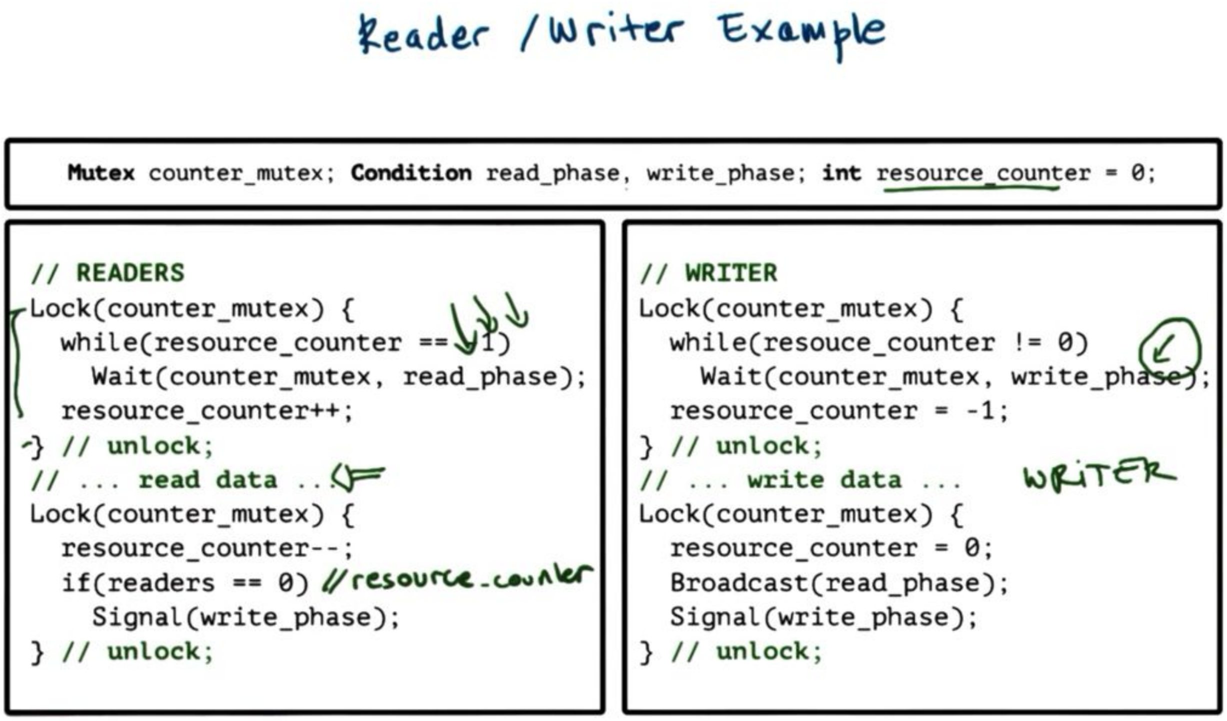
Mutex allow access to only one at a time.

Amit - few points:

1) **It is bad to lock the whole file as multiple readers can't read at the same time**
2) **So rather than file, we are protecting a variable that allows access to the files. We have three scenarios, so 3 possible values for the variable, 0, -1 and > 0.**
3) **Only one writer (why can't we have more than one writer?)**

There is a saying in computer science that all problems can be solved with one level of indirection. ***Here we basically produce a proxy resource (a helper variable, or a helper expression) - in this case it is the resource.counter variable - which reflects the state in which the current resource is already in. But what we will do is instead of controlling updates to the files/resources (controlling with the mutex who gets to access the file/resource), we will control who gets to update this proxy variable***. So as long as every access to the file/resource is updated with the proxy variable, we can use a mutex to control how the resource.counter is accessed and in this way we can monitor, control and coordinate accesses to the shared file/resource.

23 - Readers and Writers Example Part 1



Reader / Writer Example

```
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) //resource_counter
    Signal(write_phase);
} // unlock;
```

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

Let’s explain all of this with an actual reader/writer example. The actual shared file is accessed via read data and writes data operation. **As we see these read data and write data operations are outside of any lock construct both in the case of the readers as well as the writers. So what that means is that the actual access to the file is not really controlled, instead what we do is we introduce this helper variable resource_counter**

(amit - the point here is that, once the reader or the writer updates the value of the resource_counter inside the lock, the access to the file will be available or blocked accordingly, sot hat does not need to be a part of the lock)

(amit - also note that in case of writers, we block on when will the resource variable will be 0, meaning when will it be available)

and then we make sure that inner code on both the readers and the writer side before we perform the read operation, we first perform a controlled operation in which their resource counter is updated. Similarly on the writer side, before we write, we have to make sure that first resource_counters set to -1. Now that the changes to this proxy variable that will be what will be controlled and protected within the lock operation once we are done with the actual shared resource access with reading the file and writing the file, we again going to these lock blocks. This is where we update the resource counter value to reflect that the resource is now free that this one reader finished accessing it or that no writer is currently accessing the resource. So what we will need basically in our program we will need a mutex that is the counter mutex. This is the mutex that will always have to be acquired whenever the resource counter variable is accessed. And we will also need two variables to be able to read_phase and write_phase which will help us whether the reader or writer need to go next.

So let's see what happen when the very first reader tries to access the file. The reader will come in and lock the mutex. You will check what the resource counter value is and will be zero that's what the resource counter was initialized to. So it is not -1, we continue to the next operation, increment resource counter. The resource counter value now is 1 and then we unlock the mutex and we proceed accessing in the file. Subsequent reader comes in and let's says while executing this operation before it came to the unlock statement, the next reader comes in. The next reader when it comes, it will see that it cannot proceed with the lock operations; it will be blocked on the lock operation. So this way basically we are protecting the resource counter gets update, so only one thread at a time both on the readers and writers side will be able to update/access this resource_counter. However, let's see when that second reader came to the unlock operation; it will be able to join the first reader in this read data portion of the code. So we will have two threads at the same time reading the file.

Now let' see a writer thread comes in, so writer locks the mutex (the mutex is now free) and it checks the resource_counter value, the resource_counter value will have some positive number (we already allowed some number of readers to be accessing it, so we will say the value is two). So clearly the writer has to wait. It will wait in the wait operation. The wait operation specifies the counter mutex and it says it's going to wait for the write_phase. So what will happen at this point, remember we are performing this wait operation within the lock constructor, so the writer has the counter mutex, however when it enters the wait operation, the mutex automatically released, so the writer is somewhere suspend on a queue that is associated with the write_phase condition variable. The mutex at this point is free.

24 - Readers and Writers Example Part 2

So let's say the reader finished the accessing of the file, as they finish the accessing, they will first lock the counter mutex. This is why it is important that the write release the mutex, otherwise none of the reader can access the real critical session to update to the proxy variable that they are reading the file. So the reader exist the read phase, it will lock the mutex and decrement the resource_counter, and it will check the resource counter. **The resource counter check whether it is the last reader (the reader counter ==0). The very last reader will send signal to the write_phase (amit - this is imp, signal to the writer should go ONLY WHEN there are no more**

readers, so this condition is VERY IMP). It make sense to generate a signal to notify a potential writer that currently there is no reader performing reading operation. And only one writer at a time, it make no sense to use broadcast. The writer phase waiting on the writer section waiting on the write_phase condition variable, will be wake up. So what will happen internally is that the writer will be removed from the wait queue that is associated with the write_phase condition variable and the resource_counter will be reacquired before we come out of the wait operation. The very first thing we have to do is check the statement resource_counter one more time. The reason is that internally in the implementation of this operation; remove the thread associated with the write_phase operation variable and acquiring this resource_counter variable mutex are two different operation, and between them it is possible that another writer or reader has beat the writer that is waiting the mutex. So when we come out, although we have the mutex, someone else has acquired and changed the resource_counter value to reflect the phase that a reader or writer current accessing the file. And then release the mutex but actually there is another thread currently in code block that we want to protect them in the first place.

25 - Readers and Writers Example Part 3

So while the writer is executing in this write phase, there is another writer comes in. The writer is current pending on the write_phase variable. We currently have one writer and one writer waiting on the write_phase. There is another reader comes in and pending on the read_phase variable. We have two threads are waiting on two different variable and a writer is working on the critical session. So it will reset the resource counter to zero. In this code, we will do two things, one is broadcast to the read_phase and signal to write_phase. We signal to the writer_phase because we only allow one thread at a time with our write operation. We broadcast to those threads waiting on our read_phase, so potentially multiple threads will be waked up because we allow multiple threads in read phase. When multiple readers are waiting on the read_phase when we issue this broadcast, the phase in upper left corner requires a mutex. So when these threads are waiting on the read_phase is waking up from the wait, they will one at a time acquiring the mutex, check the resource_counter. If the resource_counter is not -1, since we just reset the value to 0. The resource_counter will increment the value. The first thread of the waiting ones will release the mutex and start reading the data. The remaining threads that are waiting on the read_phase,

(amit - note that reader has the condition

```
Lock
        while(resource_counter == -1)
                Wait
Unlock
```

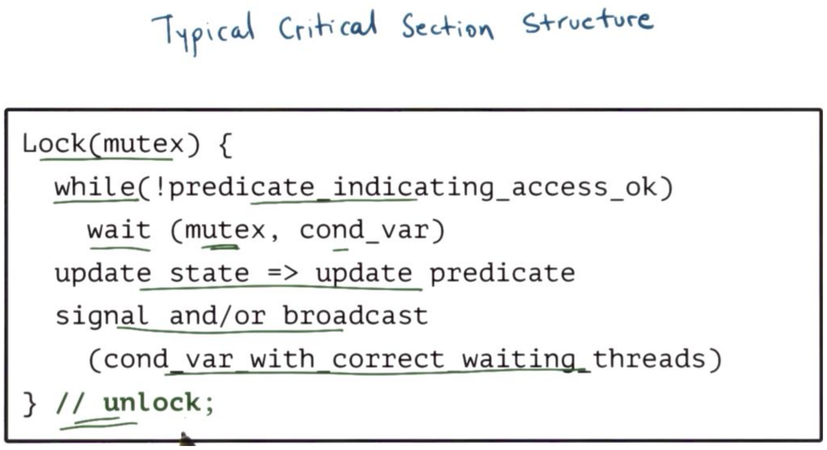
I was wondering that like writer thread that checks for while(resource_counter != 0), why we can't use the same condition for readers, and then we realized

that it is multiple readers, so when 2 readers are there, and third comes RC will be 2 != 0 will be true and it will start waiting, rather it should not and that will be done by -1 condition only. SO ALWAYS THINK ABOUT ALL POSSIBLE PERMUTATIONS)

(AMIT - also note that for writers, we say one writer that does not mean only one writer thread, that just means only one in the critical section, so this also means that when writer is over, it should signal to readers as well as to writer, so this means we need two different condition variables)

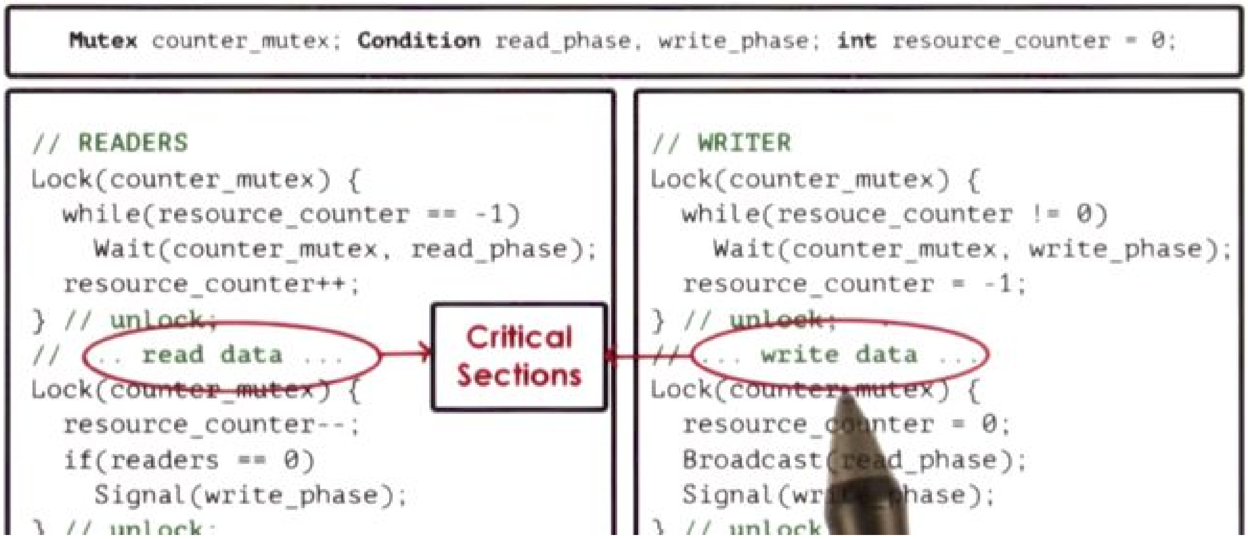
broadcast(readers)
signal(writer)

26 - Critical Section Structure



Typical Critical Section Structure

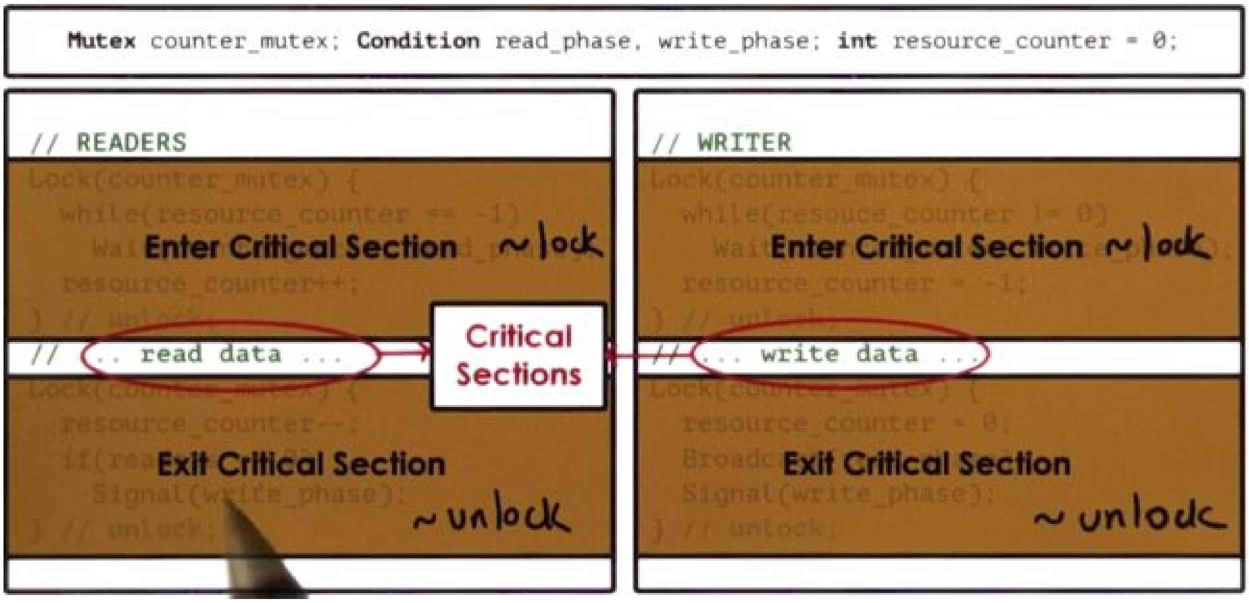
```
Lock(mutex) {
  while(!predicate_indicating_access_ok)
    wait (mutex, cond_var)
  update state => update predicate
  signal and/or broadcast
    (cond_var with correct waiting threads)
} // unlock;
```



```
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;
```

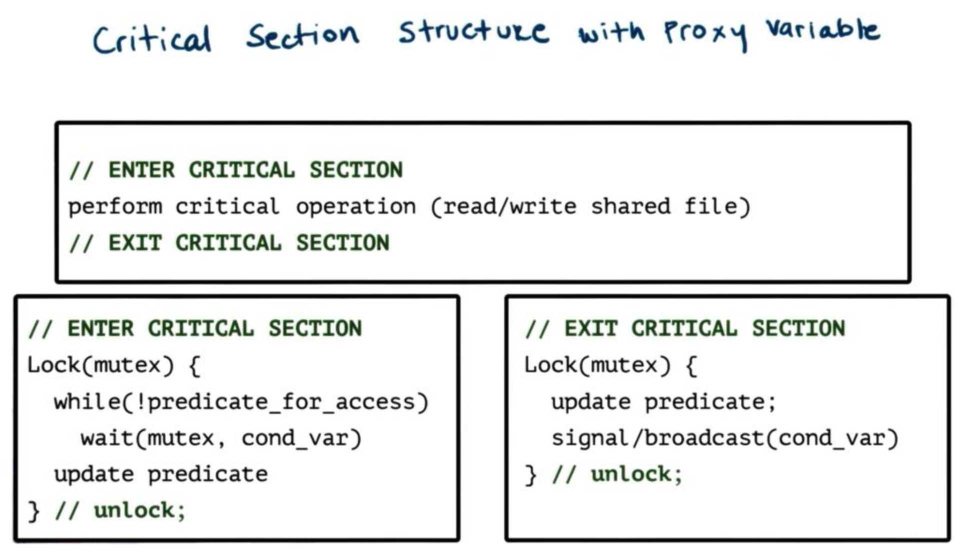
```
// WRITER
Lock(counter_mutex) {
  while(resouce_counter != 0)
    Wait(counter_mutex, write_phase);
  resource_counter = -1;
} // unlock;
// ... write data ...
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```



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

(amit - chek that in reader-writer real CS is the file read and write and not just the variable update. Variable update is just to ease out the operations)

27 - Critical Section Structure with Proxy



Critical Section Structure with Proxy Variable

```
// 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;
```

The actual read/write operations have to be protected by the enter/exit critical sections. The critical sections have the following structure:

- lock the mutex
- check for predicate
- if the predicate has not been made, they will wait in the while loop
- if the predicate is ok, it gets updated

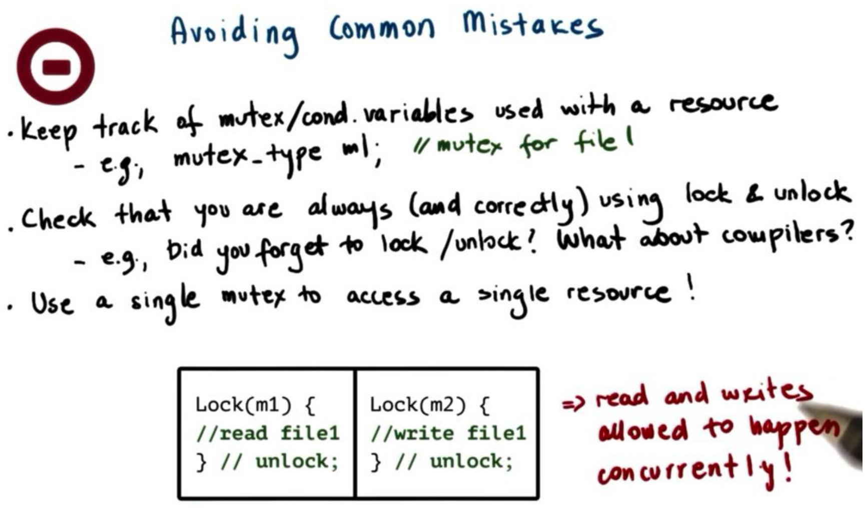
- update the predicate
- signal/broadcast the condition variable

The above structures allow us to control the proxy variable (in the mutex), however, we can perform more complex tasks - not only the boolean types that mutexes usually allow. The mutexes usually allow for a single thread access scenarios, however, with the above examples more complex situations can be accessed. In the examples, the policy that multiple reader threads can access the file or a single writer thread was implemented. The default behavior of mutexes alone does not allow us to directly enforce this kind of policy

28 - Common Pitfalls

Now let's look at some frequent problems that come up when writing multi-threaded applications.

First, make sure to keep track of the mutex and condition variables that are specifically used with



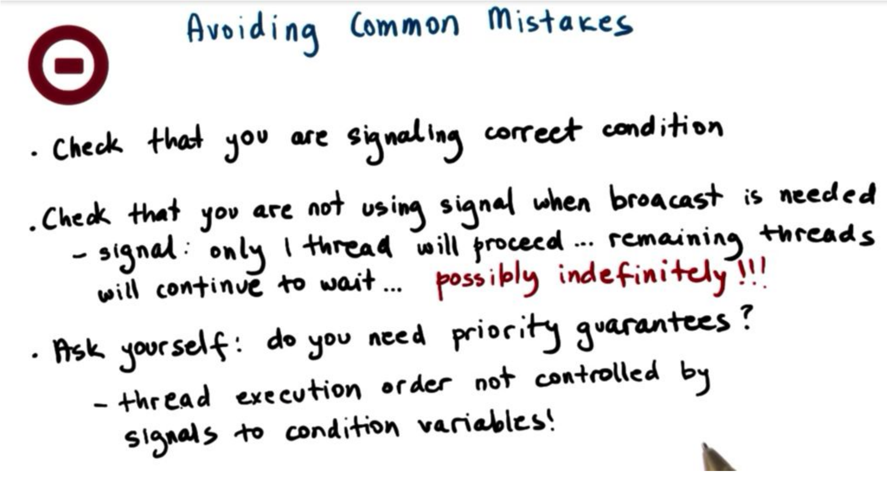
Avoiding Common Mistakes

- Keep track of mutex/cond. variables used with a resource
  - e.g., mutex_type m1; // mutex for file1
- 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

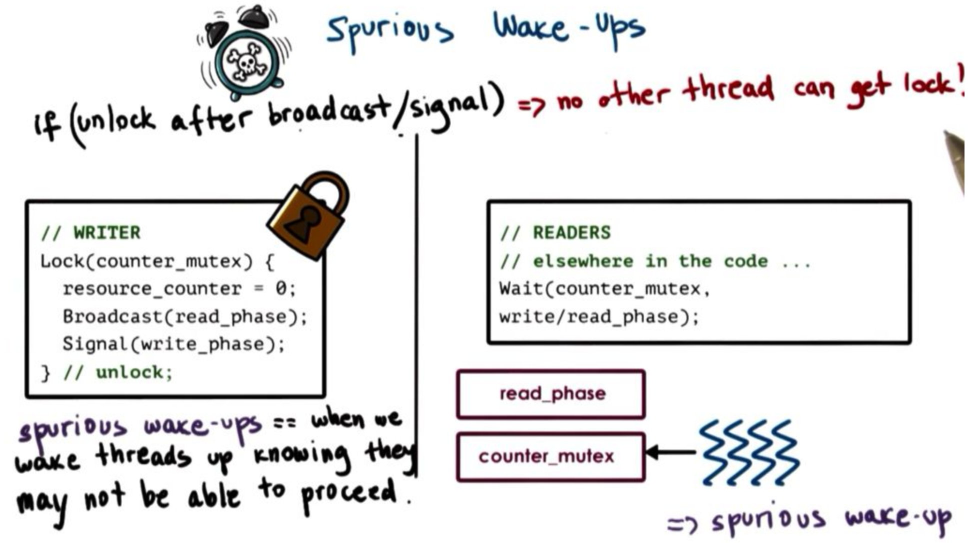
- Check that you are signaling correct condition
- Check that you are not using signal when broadcast is needed
  - signal: only 1 thread will proceed ... remaining threads will continue to wait ... possibly indefinitely!!!
- Ask yourself: do you need priority guarantees?
  - thread execution order not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

a given shared resource. What that means, for instance, is that when defining these variables make sure to write immediately a comment, which shared resource, which operation, which other piece of shared state, do you want this synchronization variable to be used with. For instance, you're creating a mutex m1 and you want to use it to protect the state of a file, file 1. Next, make sure that if a variable or a piece of code is protected with a mutex in one portion of your code, that you're always consistently protecting that same variable, or that same type of operation with the same mutex everywhere else in your count. **Basically, a common mistake is that sometimes we simply forget to use the lock/unlock construct. And therefore, sometimes access the variable in a safe way. And if we don't use the lock and unlock, then it won't be accessed in a safe way period**. Some compilers will sometimes generate warnings or, or even errors, to tell us that there is a potentially dangerous situation, where shared variable is and isn't used with a mutex in different places in the code. Or, maybe they will generate a warning to tell us that there is a lock construct that's not followed by the appropriate unlock construct. So certainly you can rely on compilers and tools to help avoid mistakes but it's just easier not to make them in the first place. **Another common mistakes that's just as bad as not locking a resource, is to use different mutexes for a single resource. So, some threads read the same file by locking mutex m1, and other threads write to the same file by locking mutex m2**.

At the same time, different threads can hold different mutexes and they can perform concurrently operations on this file, which is not what we want to be happening. So this scenario can lead to these undesirable situation, actually dangerous situations where different types of accesses happen concurrently.

Also it's important to make sure that when you're using a signal or a broadcast you're actually signaling the correct condition. That's the only way that you can make sure that the correct set of threads are potentially going to be notified. Again, using comments when you are declaring these conditions can be helpful. Also make sure that you're not using signal when you actually need to use broadcast. Note that the opposite is actually safe. If you need to use signal but use broadcast, that's fine. You will still end up waking up one thread or more. And you will not affect the correctness of the program. You may just end up affecting its performance. But that's not as dangerous. Remember that with a signal only one thread will be woken up to proceed. And if, when the condition occurred we had more than one thread waiting on the condition. The remaining threads will continue to wait. And in fact they may continue to wait possibly indefinitely. Using a signal instead of a broadcast can also possibly cause deadlocks. And we'll talk about that shortly. You also have to remember that the use of signal or broadcast or rather, the order of signal or broadcast. Doesn't do anything about making any kind of priority guarantees as far as which one of the threads will execute next. As we explained in the previous example, the execution of the threads is not directly controlled by the order in which we issue signals to a condition variables. Two other common pitfalls spurious wake ups and dead locks, deserve special attention and we will discuss these two in more detail next.

29 - Spurious Wake Ups


Spurious Wake-Ups
if (unlock after broadcast/signal) => no other thread can get lock!
spurious wake-ups == when we wake threads up knowing they may not be able to proceed.
read_phase
counter_mutex
=> spurious wake-up

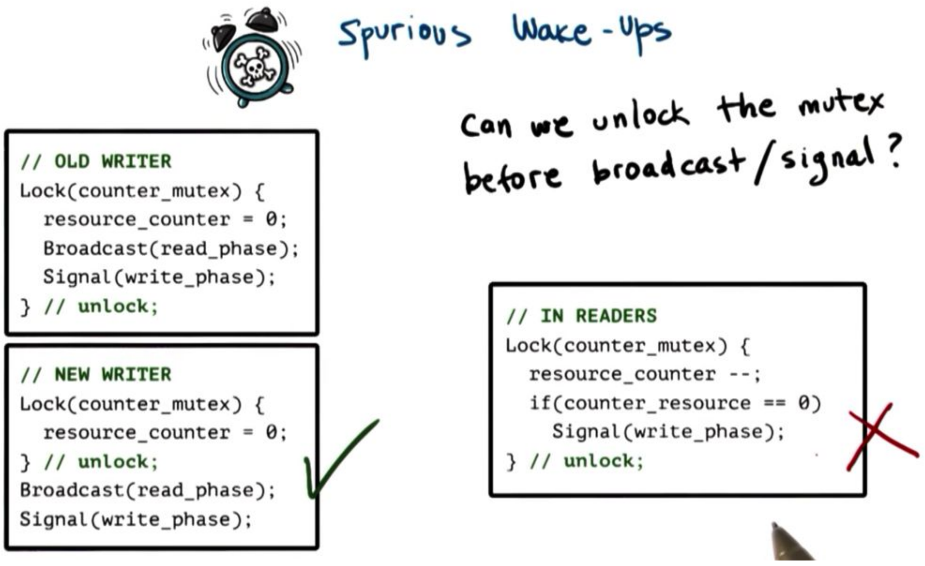
```
// WRITER
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

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

One pitfall that doesn't necessarily affect correctness, but may impact performance, is what we call spurious or unnecessary wake-ups. Let's look at this code for a writer and readers. Let's say currently there is a writer that's performing a write operation, so it is the one that has the lock counter mutex, so this is the shared lock. And then elsewhere in the program, readers for instance, are waiting on a condition variable, read_phase. So there are a number of readers that are associated with the wait queue that's part of that condition variable. So what can happen when this writer issues the broadcast operation, this broadcast can start removing threads from the wait queue that's associated with the read phase condition variable, and that can start happening, perhaps in another core, before the writer has completed the rest of the operations in the lock construct. **Now, if that's the case, we have the writer on one core. It holds still the lock, and it's executing basically this portion of the code. And, at another core, on another CPU, the threads that are waking up from this queue that's associated with the condition variable that's part of the wait statement, they have to, the very first thing they do is, they have to reacquire the mutex**. We explained this before. So that means the very first thing that these threads will do will try to re acquire the mutex. The mutex is still held by the writer thread. The writer thread still has the mutex. So none of these threads will be able to proceed. They'll be woken up from one queue that's associated with the condition variable, and they'll have to be placed on the queue that's associated with the mutex. So we will end up with this type of situation as a result of this. ***This is what we call spurious wake-up. We signaled we woke up the threads. And that wake-up was unnecessary. They have to now wait again. The program will still execute correctly. However, we will waste cycles by basically context switching these threads to run on the CPU and then back again to wait on the wait queue.*** The problem is that when we unlock only after we've issued the broadcast or the signal operation, no other thread will be able to get the lock. So spurious wake-ups is this situation when we're waking threads up, we're issuing the broadcast or the signal, and we know that it is possible that some of the threads may not be able to proceed. It will really depend on the ordering of the different operations. So, would this always work, though? Can we always unlock the mutex before we actually broadcast our signal?

For instance by using this trick, we can transform the old writer code into this code where, we first unlock, and then we perform the broadcast and signal operations. This clearly will work just fine. The resulting code will avoid the problem of spurious wake-ups, and the program remains correct. In other cases, however, this would not be possible. We cannot restructure the program in this way. So if we look at what's happening at the readers, t**he signal operation is embedded in this if clause. And the if statement relies on the value of resource_counter. Now, resource_counter was the shared resource that this mutex was protecting in the first place (amit - takeaway - watch out for operations..any case where we are using the protected variable again after unlock to just signal or something like this - then that is a bug).**

So we cannot unlock and then continue accessing the shared resource. That will affect the correctness of the program.



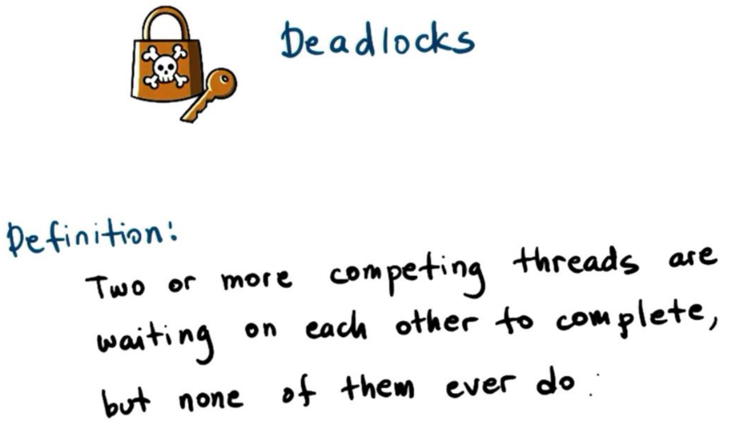
```
// OLD WRITER
Lock(counter_mutex) {
  resource_counter = 0;
  Broadcast(read_phase);
  Signal(write_phase);
} // unlock;
```

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

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

Therefore, this technique of unlocking before we perform the broadcast or signal doesn't work in this particular case or in similar cases. **When the signal or broadcast is dependant on some condition that is checked in the mutex, the signal or broadcast must be within the lock!**

30 - Deadlocks Introduction



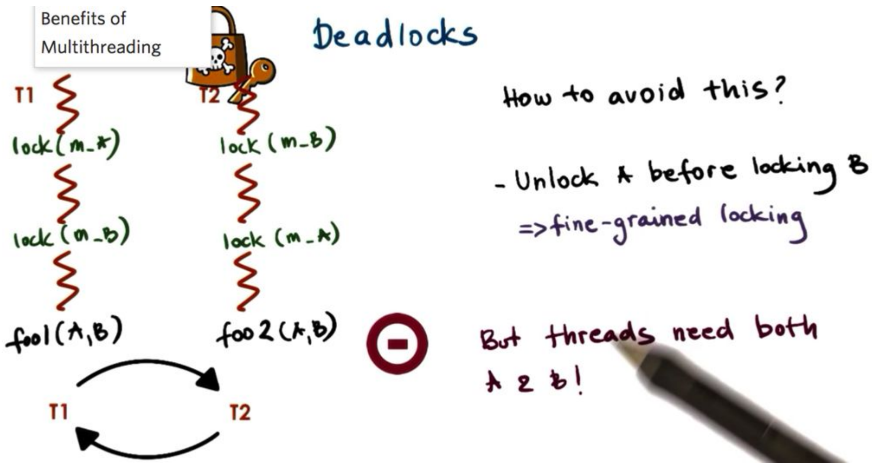
Deadlocks

Definition:

Two or more competing threads are waiting on each other to complete, but none of them ever do.

Deadlocks

31 - Deadlocks

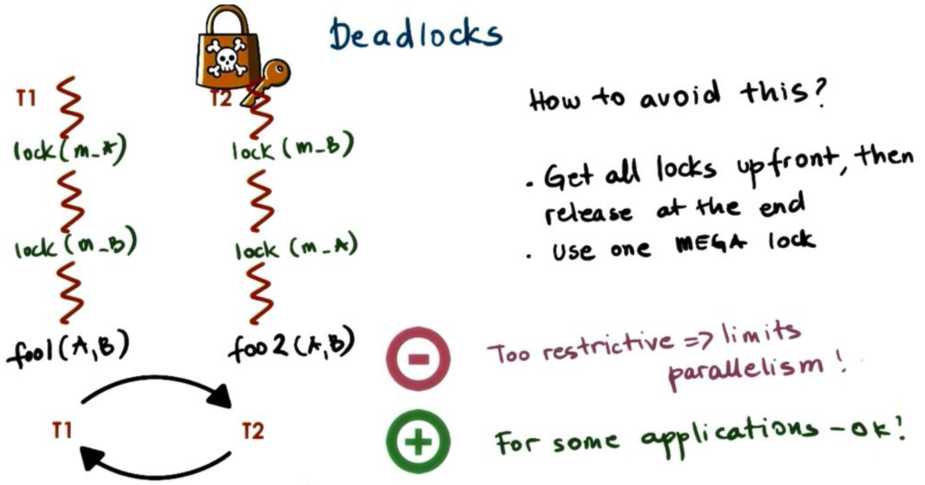


Deadlocks

How to avoid this?

- Unlock A before locking B
  => fine-grained locking

⊖ But threads need both A & B!



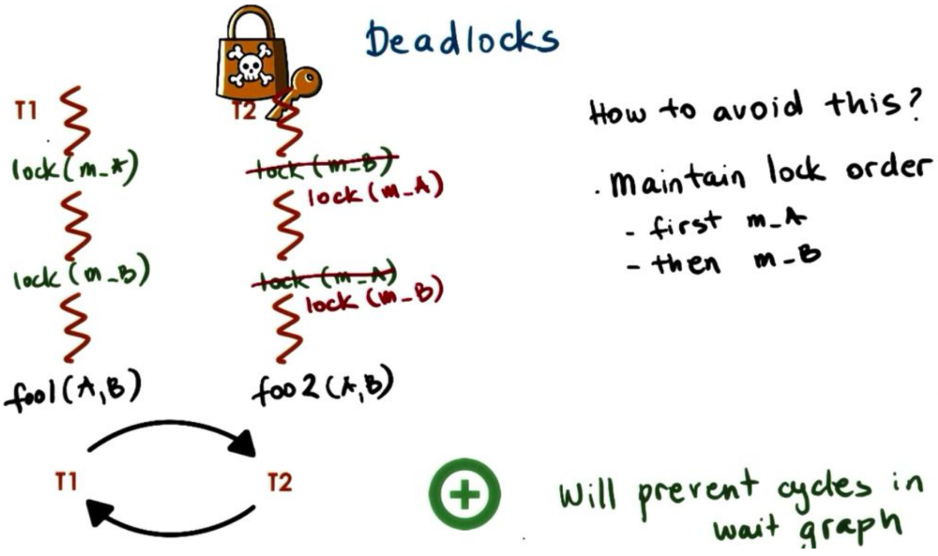
Deadlocks

How to avoid this?

- Get all locks upfront, then release at the end
- Use one MEGA lock

⊖ Too restrictive => limits parallelism!

⊕ For some applications - ok!



Deadlocks

How to avoid this?

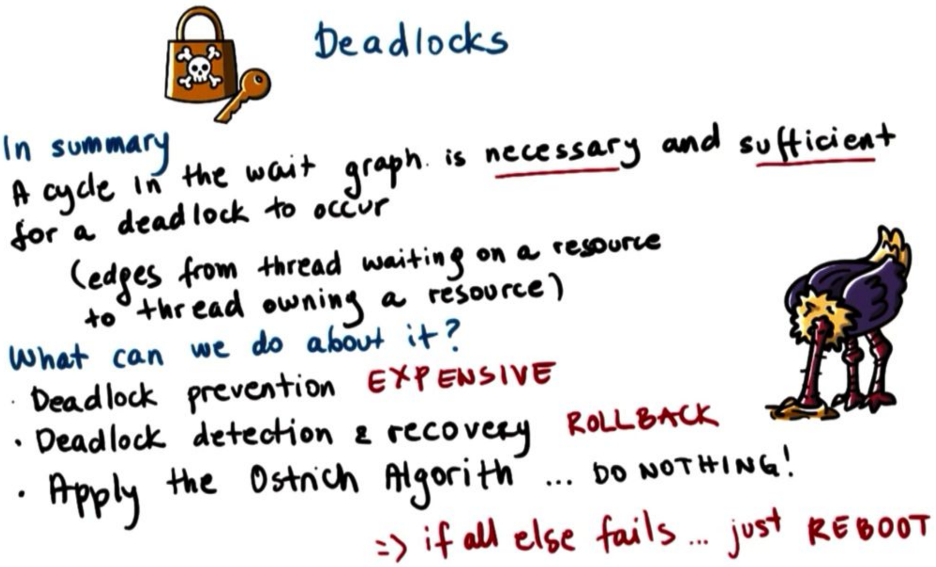
- Maintain lock order
  - first m_A
  - then m_B

⊕ Will prevent cycles in wait graph

When using nested mutexes, be sure to use the mutexes in the same order when the mutexes are used by other threads.

(amit - fine grained locking does not work when both variables are required)

32 - Deadlocks Summary



Deadlocks

In summary

A cycle in the wait graph is necessary and sufficient for a deadlock to occur

(edges from thread waiting on a resource to thread owning a resource)

What can we do about it?

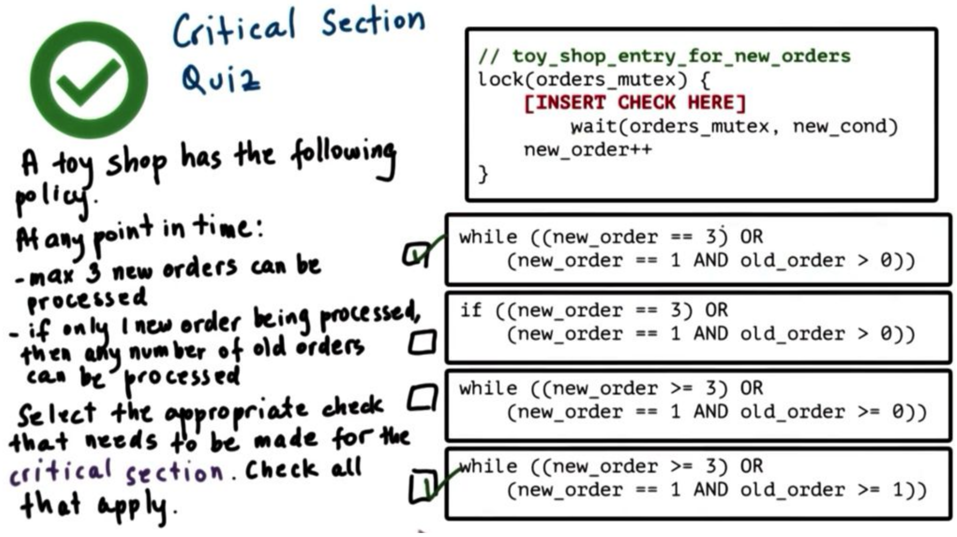
- Deadlock prevention EXPENSIVE
- Deadlock detection & recovery ROLLBACK
- Apply the Ostrich Algorith ... DO NOTHING!

=> if all else fails ... just REBOOT

33 - Critical Section Quiz

Let's take a quiz in which we'll take a look at an example of a critical section. The critical section that we will look at corresponds to a critical section in a toy shop similar to the toy shop examples that we looked at. In the toy shop, there will be new orders that will be coming in. As well as there will be orders for repairs of toys that have already been process, so, like, old orders that need to be revisited. Only a certain number of threads, a certain number of workers, will be able to operate in the toy shop at any given point of time. So there will be a mutex, orders_mutex, that controls which workers have access to the toy shop. Basically, which orders can be processed. The toy shop has the following policy. At any given point of time, there can be up to three new orders processed in the toy shop. In addition, if there is up to only one new order being processed, then any number of requests to service old orders can be handled in the toy shop. The code shown in this box describes the critical section entry code that's executed by the workers performing new orders. As expected, we first lock the mutex and then check a condition, and this condition must correspond to this policy. Depending on this condition, on this predicate, we determine whether the thread, whether the new order, can go ahead and be processed in the toy shop, or if we must wait. With the wait statement, we use a condition variable new_cond as well as we include the mutex. Because as we mentioned, a mutex must be associated with a wait statement so that it can be atomically released. The predicate statement that we must check on before determining whether a thread will wait or can proceed, is missing. For this quiz, you need to select the appropriate check that needs to be made in order to enter the critical section. There are four choices given here. You should select all that apply.

34 - Critical Section Quiz



Critical Section Quiz

A toy shop has the following policy.
At any point in time:
- max 3 new orders can be processed
- if only 1 new order being processed, then any number of old orders can be processed

Select the appropriate check that needs to be made for the critical section. Check all that apply.

```
// toy_shop_entry_for_new_orders
lock(orders_mutex) {
    [INSERT CHECK HERE]
        wait(orders_mutex, new_cond)
    new_order++
}
```

☑
```
while ((new_order == 3) OR
    (new_order == 1 AND old_order > 0))
```

□
```
if ((new_order == 3) OR
    (new_order == 1 AND old_order > 0))
```

□
```
while ((new_order >= 3) OR
    (new_order == 1 AND old_order >= 0))
```

☑
```
while ((new_order >= 3) OR
    (new_order == 1 AND old_order >= 1))
```

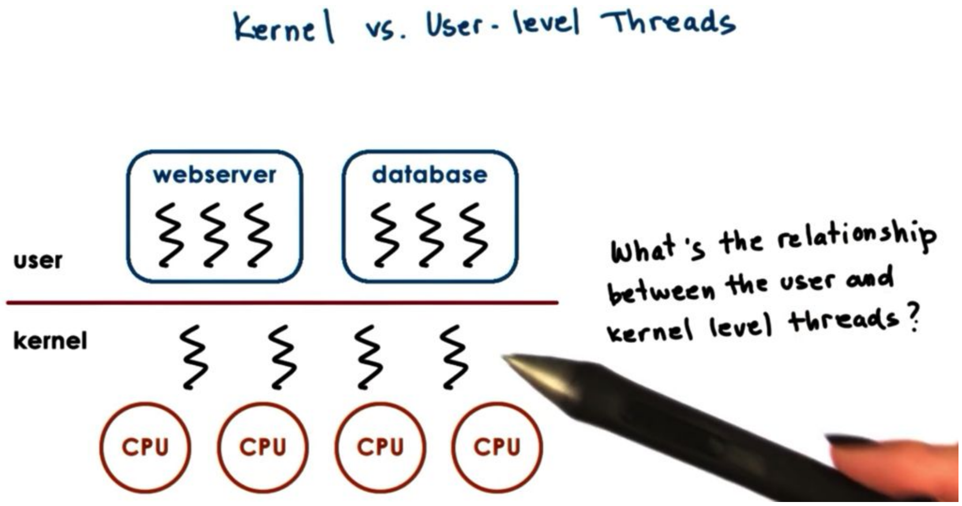
Feedback:
- Which conditional follows the policy most clnosely?
- Does checking for more than 3 new orders affect the correct execution of this code?
- If we do not have any old order, then should we have to wait?
- One part of the policy is misleading as stated. It should state “if any old orders are being processed than only one new order can be processed”

Hints:

• Pay attention to the comparison operators... which values satisfy the conditions?

The first code snippet is correct because it perfectly aligns with the policy. If new_order is equal to 3, clearly an incoming thread will not be able to proceed. So this will guarantee that there cannot be more than three new orders processed in the toy shop. Note, by the way, that new_ order cannot be larger than 3, given that the only way that it will get updated is once we come out of this wait statement. So, the maximum value that new_order can receive in this code is 3. The second part of this statement also perfectly aligns with the second part of the policy. If we have a situation in which there are some number of old_orders in the system, and one request for a new_order has already entered the toy shop, then any incoming new_order will have to be blocked at the wait statement. So, this piece of code, this answer, is correct. The second code snippet is almost identical to the first one, however, it uses if as opposed to while. We explained that if creates a problem, in that, when we come out of the statement, when we thought that this condition was satisfied. It is possible that, in the meantime, another thread has come in and executed this particular lock operation or even the critical section entry for the old_order, and has therefore changed the value of this predicate. If we don't go back to reevaluate the predicate, it is possible that we will miss such a case and therefore enter the actual critical section. So, enter the toy shop in a way that violates this policy. So this answer is not correct because it uses this if as opposed to a while. The third statement is incorrect because it checks whether old_order is greater than or equal to 0. So, what this means means if that a new incoming order will be blocked if there is already one new_order in the system and old_order is equal to 0. And basically no other orders for toy repairs, no other old_orders are in the system. That is clearly not the desired behavior. We want to allow up to three new orders to be processed in the system. So this statement is incorrect. The fourth statement is basically identical to the first one except for the fact that it uses greater than or equal to 3. As we already pointed out, new_order will really not even receive a value greater than 3 the way it's updated here. So this statement will result in the identical behavior as the first statement. So both of these are correct.

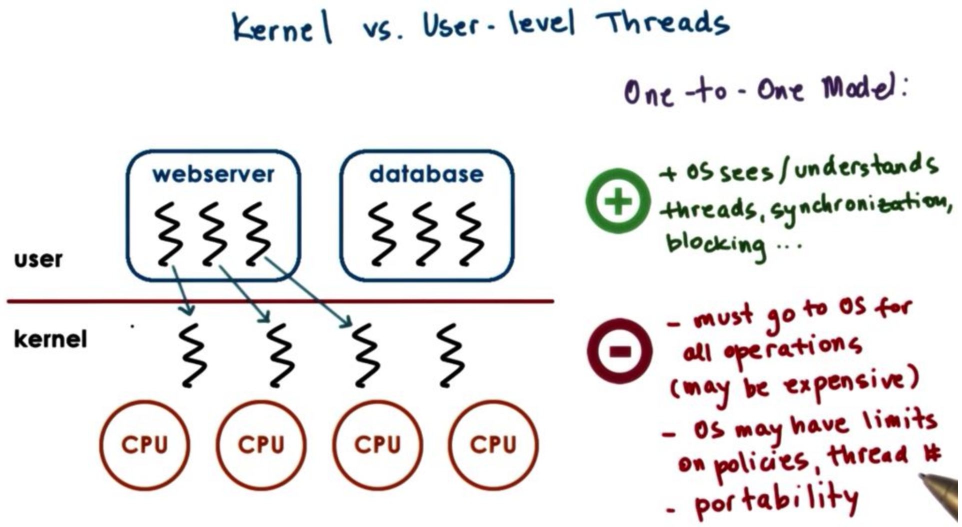
35 - Kernel vs User-Level Threads


Kernel vs. User-level Threads
webserver
database
user
kernel
CPU
CPU
CPU
CPU
What's the relationship between the user and kernel level threads?

We said earlier that threads can exist at both the kernel and the user level. Let's take a look more at what we mean by this distinction. Kernel-level threads imply that the operating system itself is multi-threaded. Kernel-level threads are visible to the kernel and are managed by kernel-level components like the kernel-level scheduler. So it is the operating system scheduler that decides how these threads will be mapped onto the underlying physical CPUs and which of them will be run at any given time.

Some of these kernel-level threads will be there to support processes (to run user-level threads). Some other threads may be there to run operating system services or daemons (amit - takeaway - every KT is NOT available for ULT). At the user-level the processes themselves are multi-threaded. For a user-level thread to execute it must be associated with a kernel-level thread and the OS must schedule that kernel-level thread onto a CPU.

Let’s investigate the relationship between user-level threads and kernel-level threads. We will now look at three such models.

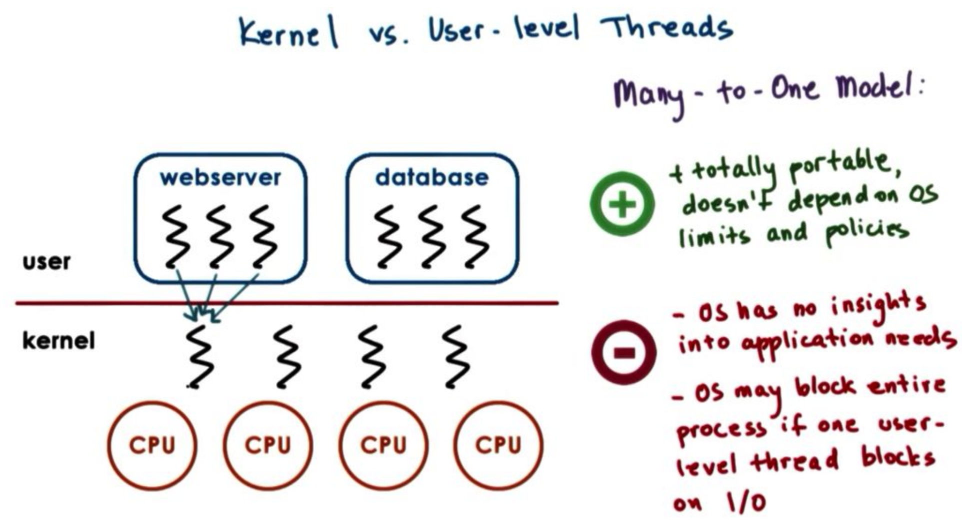

Kernel vs. User-level Threads
One-to-One Model:
webserver
database
user
kernel
CPU
CPU
CPU
CPU
+ OS sees/understands threads, synchronization, blocking ...
- must go to OS for all operations (may be expensive)
- OS may have limits on policies, thread #
- portability

The first model is a one-to-one model. Here each user-level thread has a kernel-level thread associated. When a process creates a user-level thread either a kernel-level thread is created or an available kernel-level thread is associated to the user-level thread. **This means the operating system can see all the user-level threads: it understands that the process is multi-threaded. It also understands what the process needs in terms of synchronization, blocking, etc. (amit - in this model OS can have maximum information about ULT)**

...

The downside of this approach is that for every thread we must go to the kernel. As we discussed earlier, this can be expensive. This model also means that since we’re relying on the kernel to do the thread-level management to do the scheduling, synchronization, etc. we’re limited by the policies at the kernel level.

...



Kernel vs. User-level Threads

Many-to-One Model:

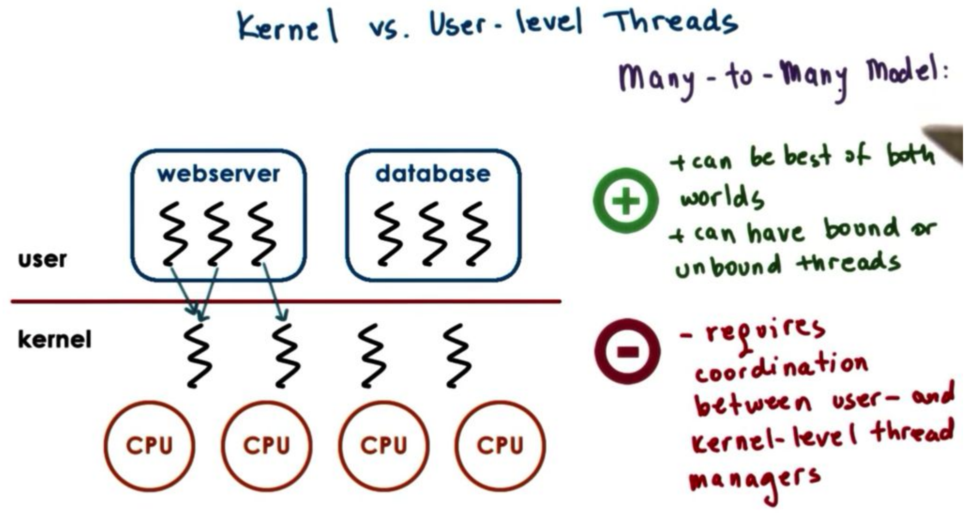
+ totally portable, doesn't depend on OS limits and policies

- OS has no insights into application needs
- OS may block entire process if one user-level thread blocks on I/O

The second model is the many-to-one model.

Amit -

1) **We don't cross the boundary from user to kernel often**
2) **User level library manages scheduling of ul threads etc and sync.**
3) **Check that anytime kernel does not know abou UL threads there is always a risk where the whole process can be blocked - think in this terms.**

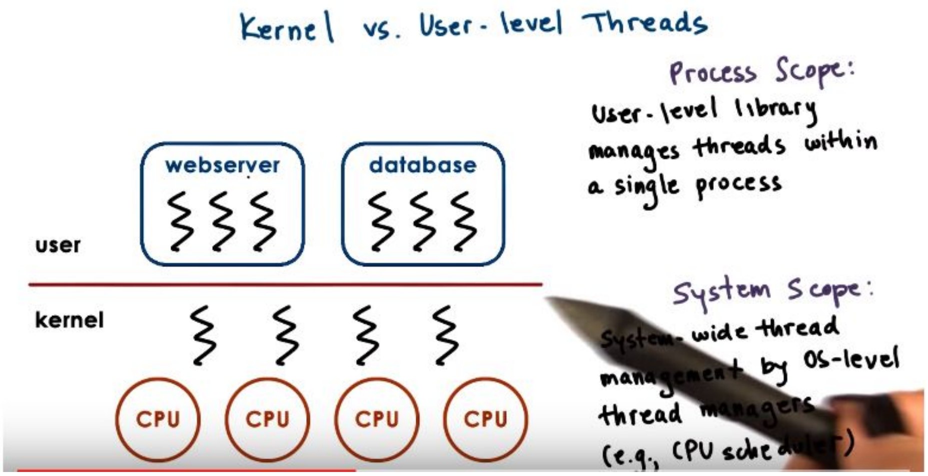


Kernel vs. User-level Threads

Many-to-Many Model:

+ can be best of both worlds
+ can have bound or unbound threads

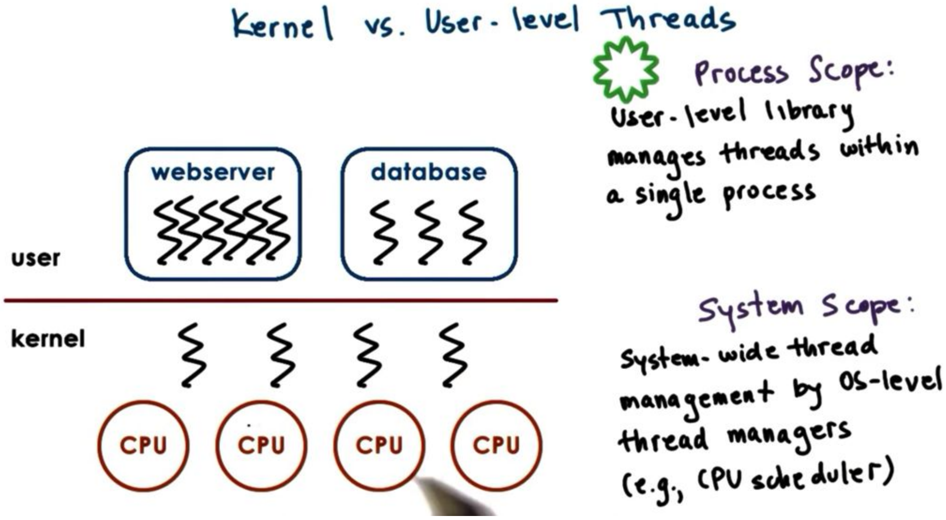
- requires coordination between user- and kernel-level thread managers



Kernel vs. User-level Threads

Process Scope:
User-level library manages threads within a single process

System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)

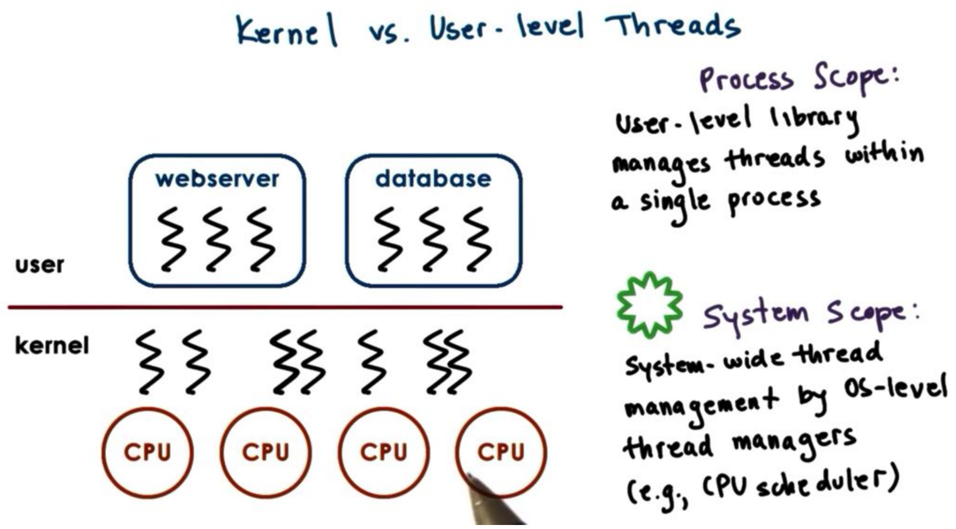


Kernel vs. User-level Threads

Process Scope:
User-level library manages threads within a single process

System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)

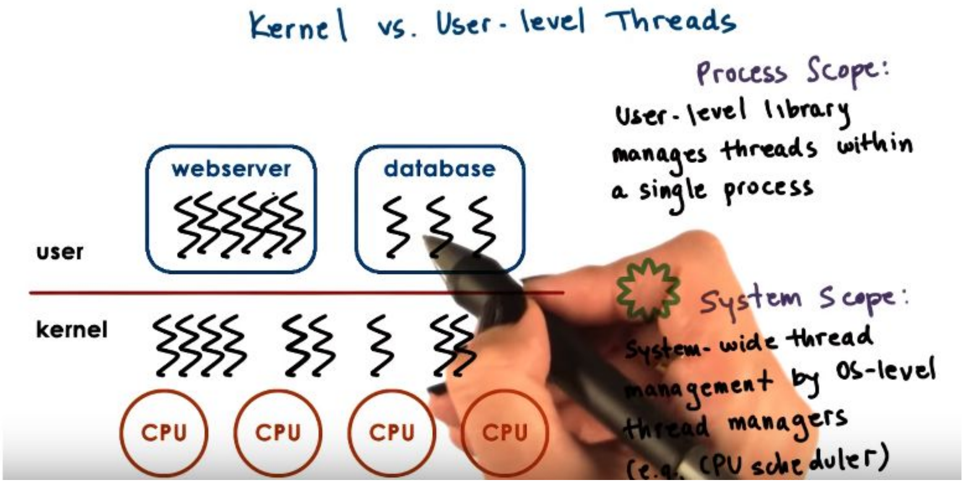
If one process has twice the number of threads as another, the kernel level thread scheduler will not know this if the user level threads have process scope. In the example above, half of the kernel level threads would be given to each process instead of ⅓ being given to the database process and ⅔ being give to the webserver process.



Kernel vs. User-level Threads

Process Scope:
User-level library manages threads within a single process

System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)



Kernel vs. User-level Threads

Process Scope:
User-level library manages threads within a single process

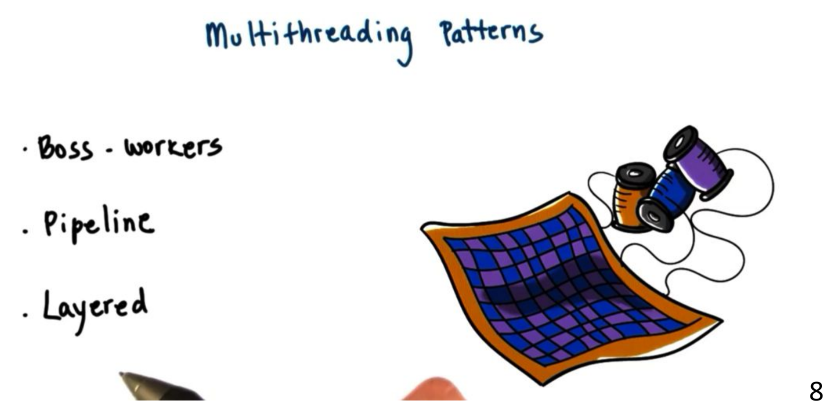
System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)

Kernel thread scheduler will move kernel threads around to satisfy the needs of the user level threads. Since webserver has more threads than database, the cpu attached to the webserver process will have more kernel level threads allocated for it (amit - so system scope is better).t

36 - Multithreading Models

37 - Scope of Multithreading

38 - Multithreading Patterns

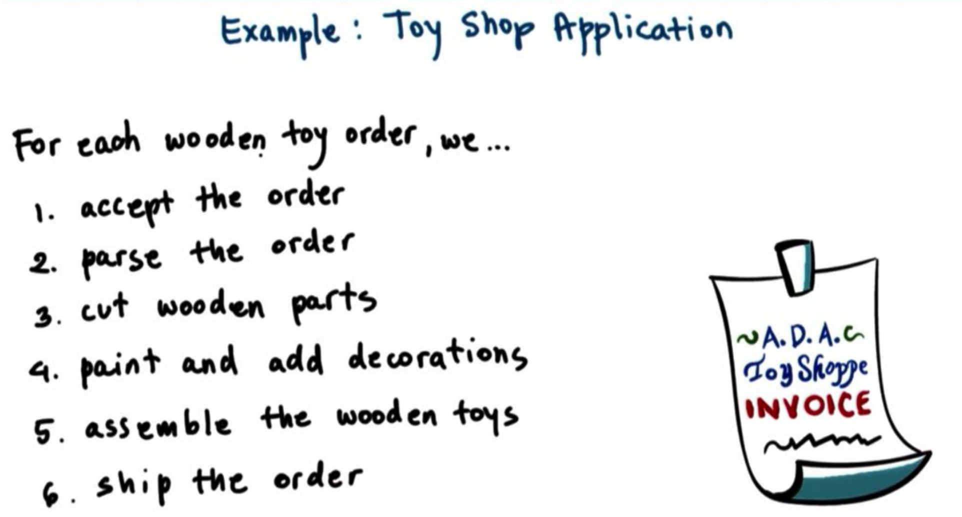


Multithreading Patterns

- Boss - workers
- Pipeline
- Layered

8

39 - Boss/Workers Pattern

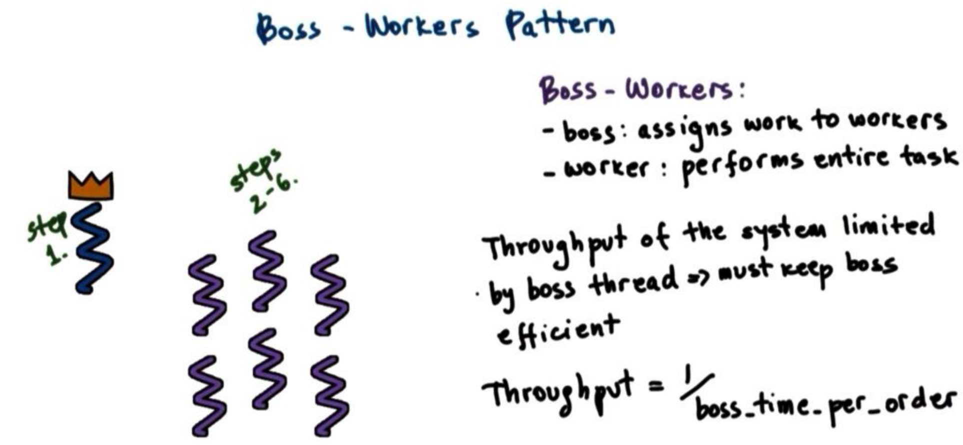


Example : Toy Shop Application

For each wooden toy order, we...

1. accept the order
2. parse the order
3. cut wooden parts
4. paint and add decorations
5. assemble the wooden toys
6. ship the order

Boss worker pattern overview



Boss - Workers Pattern

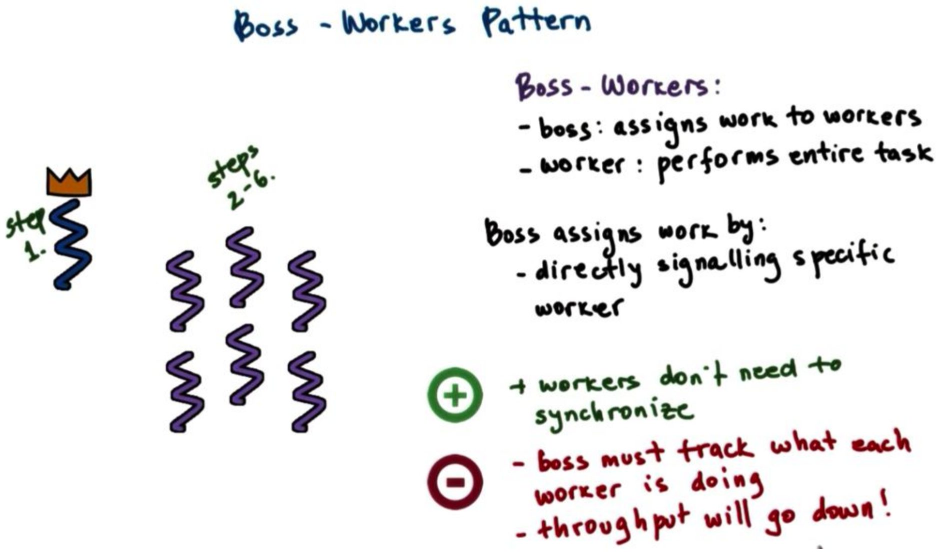
Boss - Workers:
- boss: assigns work to workers
- worker : performs entire task

· Throughput of the system limited by boss thread => must keep boss efficient

Throughput = 1/boss_time_per_order

(amit - imp that in boss-worker pattern, the throughput of the system is dependent on boss only because that is the front end that takes requests, of that is busy it will affect the system. Also check the inverse relationship here)

Boss worker example 1: Boss signals task to individual thread



Boss - Workers:
- boss: assigns work to workers
- worker: performs entire task

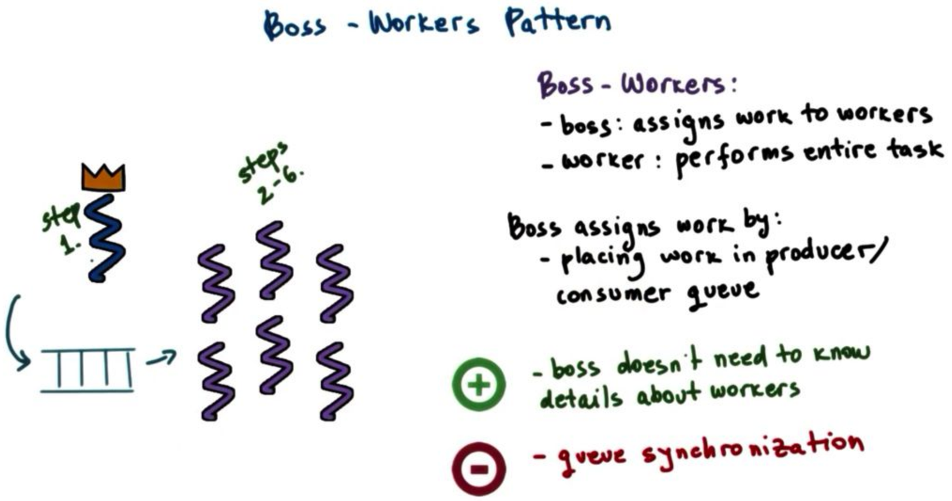
Boss assigns work by:
- directly signalling specific worker

+ workers don't need to synchronize

- boss must track what each worker is doing

- throughput will go down!

Boss worker example 2: Boss adds task to a queue. Thread retrieves task from the queue. ***Results in lower time per task that the boss needs***. Overall better throughput as a result than Boss worker example 1 above.



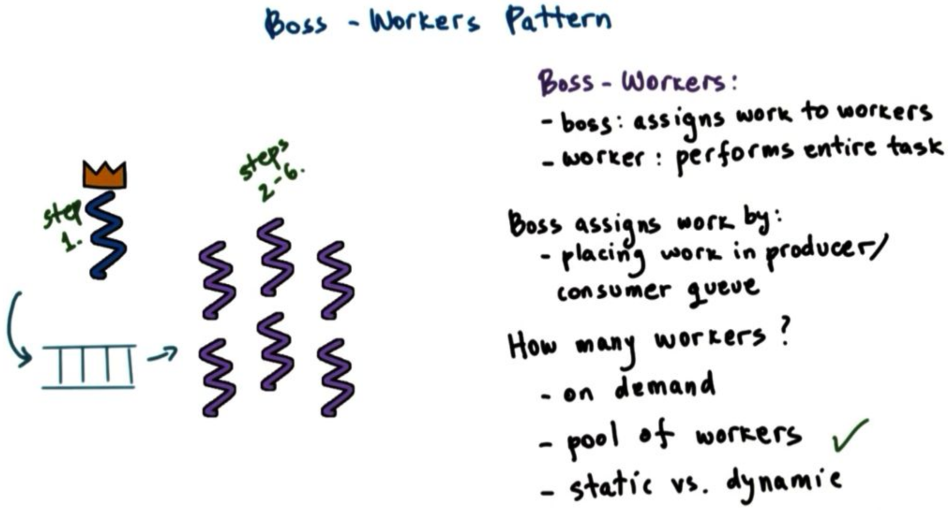
Boss - Workers:
- boss: assigns work to workers
- worker: performs entire task

Boss assigns work by:
- placing work in producer/consumer queue

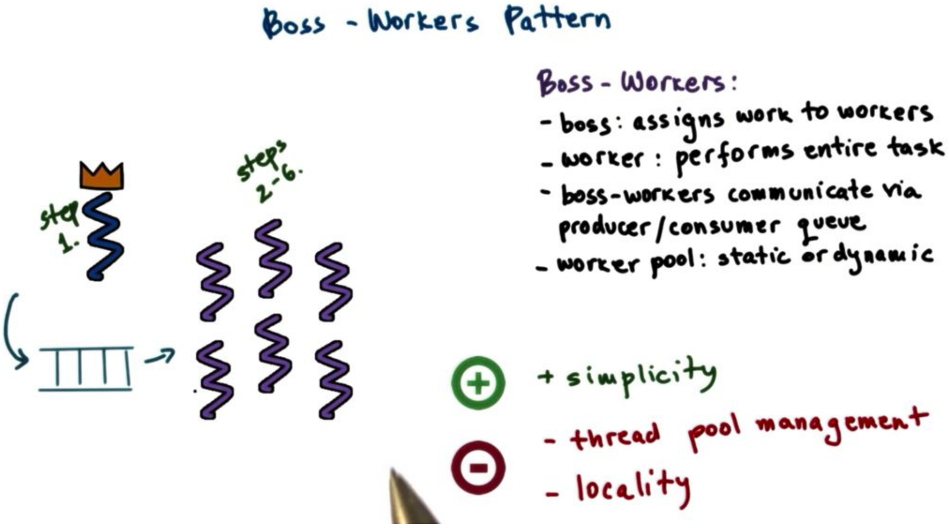
- boss doesn't need to know details about workers

- queue synchronization

40 - How Many Workers



*dynamic refers to the fact that one adds to the pool of workers already created, but multiple threads are added instead of only one thread.


Boss - Workers Pattern
step 1.
steps 2-6.
Boss - Workers:
- boss: assigns work to workers
- worker: performs entire task
- boss-workers communicate via producer/consumer queue
- worker pool: static or dynamic
+ simplicity
- thread pool management
- locality

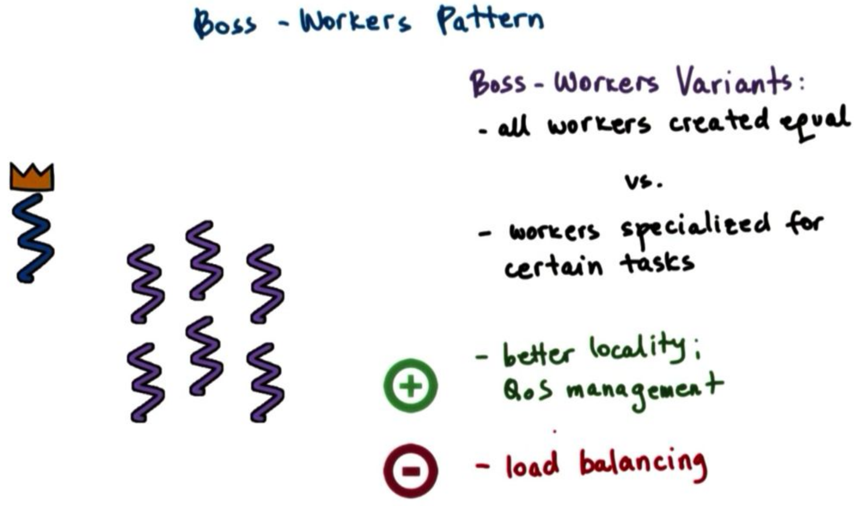
Summary:
The boss assigns work to the workers. Each worker performs the entire task. The boss and the workers communicate via shared producer-consumer queue. We use a worker pool-based approach to manage a number of threads in the system where we potentially adjust the size of the pool dynamically. Benefit is in simplicity. The drawback is in overhead related to the management of the thread pool, including synchronization for the shared buffer. **Another negative of this approach is that it ignores locality (amit - this is an imp point that we normally ignore).** The boss doesn't keep track of what each worker is doing. If we have a situation where a worker just completed a similar type of task or identical type of task, it is more likely that the same worker will be more efficient in performing the exact same task in the future. Or it may be that

it already has a tool that is required to build that particular type of toy nearby on its desk. But if the boss does not know that the workers are doing, it has no way to make these kind of optimizations.

41 - Boss/Workers Variants

Boss has to do more work per assignment if trying to specialize the workers. However, improvements in each worker's abilities due to specialization should compensate for the increase in boss time resources.

Locality -> i.e. hot cache
QoS -> Quality of Service Management
Load Balancing -> How many threads need to exist for a particular task?



Boss - Workers Pattern

Boss - Workers Variants:

- all workers created equal

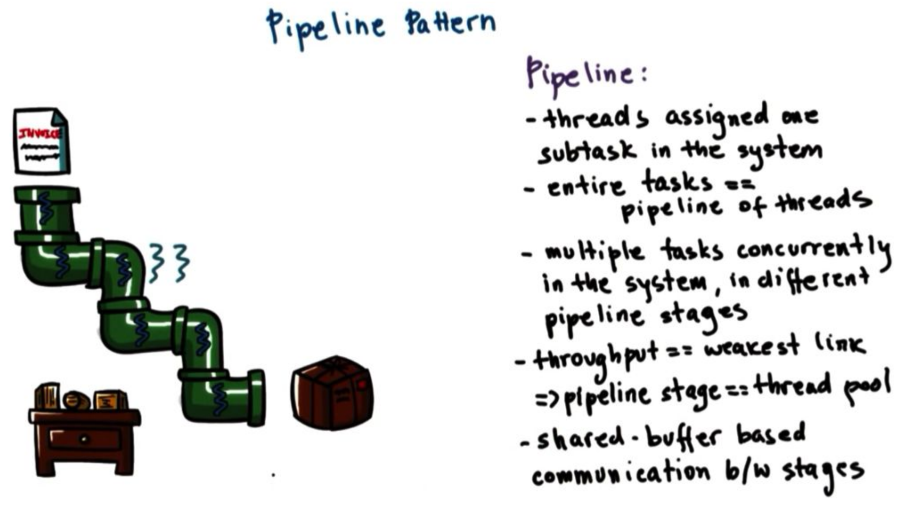
vs.

- workers specialized for certain tasks

⊕ - better locality; QoS management

⊖ - load balancing

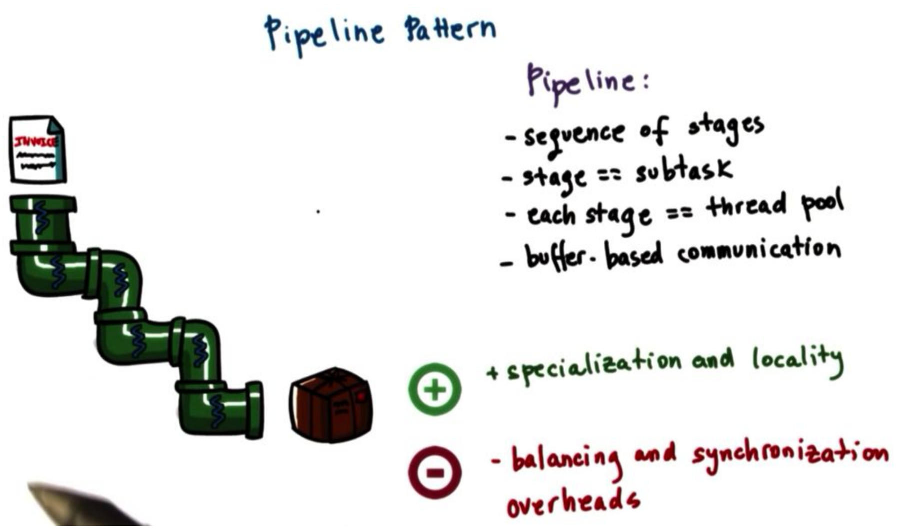
42 - Pipeline Pattern



Pipeline Pattern

Pipeline:

- threads assigned one subtask in the system
- entire tasks == pipeline of threads
- multiple tasks concurrently in the system, in different pipeline stages
- throughput == weakest link
  => pipeline stage == thread pool
- shared-buffer based communication b/w stages



Pipeline Pattern

Pipeline:

- sequence of stages
- stage == subtask
- each stage == thread pool
- buffer-based communication

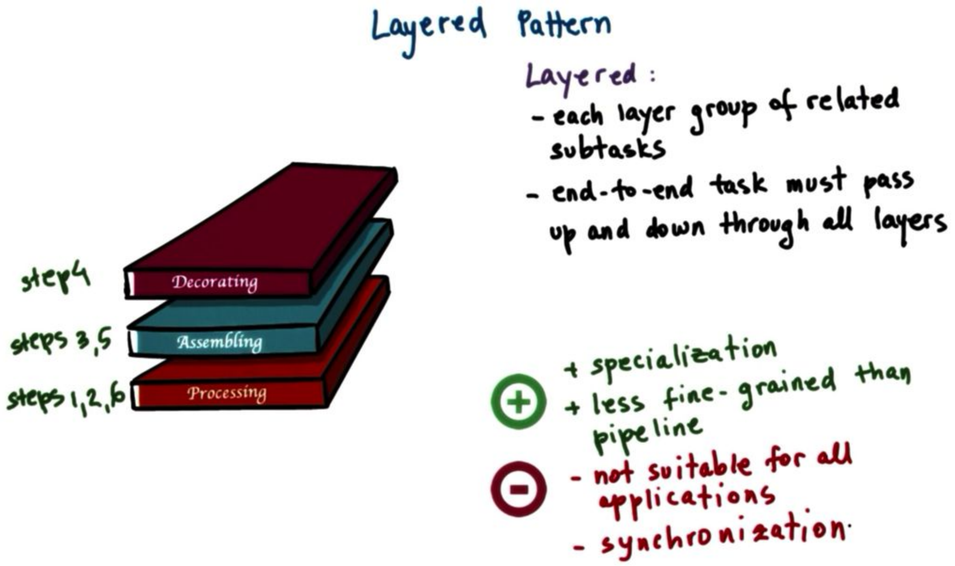
⊕ + specialization and locality

⊖ - balancing and synchronization overheads

Amit - check that throughput depends on the weakest link. Imp point. This slow stage can be enhanced by adding more worker threads for that stage.

Check that we have better locality here because the same thread keeps on doing the same kind of work. So locality is better than boss-worker.

43 - Layered Pattern



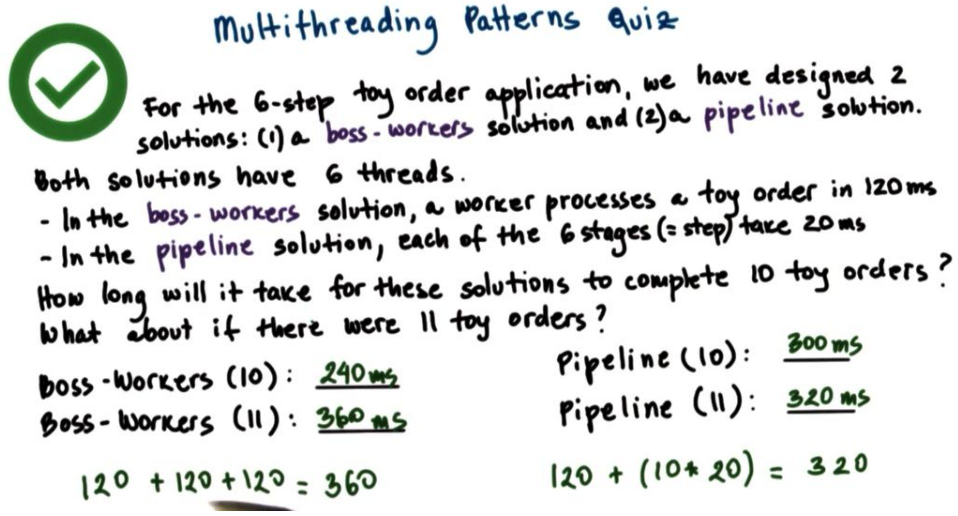
Layered Pattern

Layered:
- each layer group of related subtasks
- end-to-end task must pass up and down through all layers

+ specialization
+ less fine-grained than pipeline
- not suitable for all applications
- synchronization.

44 - Multithreading Patterns Quiz

45 - Multithreading Patterns Quiz



Multithreading Patterns Quiz

For the 6-step toy order application, we have designed 2 solutions: (1) a boss-workers solution and (2) a pipeline solution.

Both solutions have 6 threads.
- In the boss-workers solution, a worker processes a toy order in 120ms
- In the pipeline solution, each of the 6 stages (= step) take 20ms

How long will it take for these solutions to complete 10 toy orders? What about if there were 11 toy orders?

Boss-Workers (10): 240ms
Boss-Workers (11): 360 ms

Pipeline (10): 300ms
Pipeline (11): 320 ms

120 + 120 + 120 = 360

120 + (10 * 20) = 320

Quiz Help
Boss-Worker Formula:

- time_to_finish_1_order * ceiling (num_orders / num_concurrent_threads)

Pipeline Formula:

- time_to_finish_first_order + (remaining_orders * time_to_finish_one_stage)

You do not have to include units (ms) in your answers.

Feedback:

- Boss-Worker (10) takes more time (or is invalid)
- Boss-Worker (11) takes more time (or is invalid)
- Pipeline (11) takes less time

(amit - REMEMBER THAT BOSS WORKER WILL NOT TAKE PART AS IT WILL BE JUST ALLOCATING JOBS, SO EFFECTIVELY HOW MUCH DATA IS PRODUCED DEPENDS ON HOW MANY WORKER THREADS ARE THERE....WHICH IS 5 IN OUR CASE)

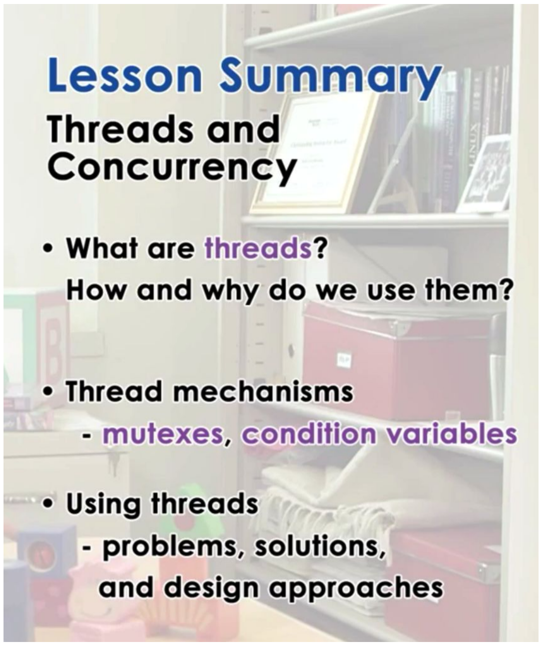
Hints:

- You do not have to include units (ms) in your answer
- In the Boss-Worker solutions, 1 thread is the boss, and the others are workers
- In the Boss-Worker solutions, assume it takes no delay for the boss thread to assign orders
- In the Pipeline solutions, the pipeline must be filled before a task completes each 20 ms

Okay. Let's take a look at what's happening in the system. In the first case, we have ten toy orders. Both solutions have six threads each. For the boss-workers case, that means that one of the threads will be the boss, and then the remaining five threads will be the workers.For the pipeline model, each of the six threads will perform one stage, one step in the toy order application. So, for the boss-workers case, because we have five worker threads, at any given point of time, these workers will be able to process up to five toy orders. So, if we have ten toy orders, for the boss-worker model, the workers will process the first five orders, given that we have five workers, at the same time. And every single one of them will take 120 milliseconds, so the first five toy orders will be processed in 120 milliseconds. The next five orders will take additional 120 milliseconds for a total of 240 milliseconds. For the pipeline case, the very first toy order will take 120 milliseconds to go through the six stages of the pipeline. So 6 times 20 milliseconds. Then, once the first toy order exits the pipeline, that means that the second toy order is already in the last stage of the pipeline. So we'll take another 20 seconds to finish.

And then the third toy order will be immediately afterwards. It will take additional 20 milliseconds to finish. So given that we have nine remaining orders after the first one, the total processing time for the pipeline case when we have ten toy orders is as follows. 120 for the first one, and then 9 times 20 to complete the last stage of every single one of the remaining nine toys. That's 300 milliseconds. Now, if we have 11 toy orders, we will process the first ten in the exact same manner as before. We have five worker threads. They can only process five toy orders at the same time. So the first ten out of these 11 will be processed in 240 milliseconds. Then, the 11th order will take another 120 milliseconds. Only one of the workers will be busy. Only one of the workers will be processing that toy order. However, it will take an additional 120 milliseconds to complete all of the 11 toy orders for a total of 360 milliseconds. For the pipeline case, applying the exact same reasoning as before, when we have 11 toys, it will take 120 to process the first one. And then for the remaining ten, it will take another 20 milliseconds for every single one of them to finish the last stage of the pipeline. So we'll take a total of 320 millisecond for the pipeline approach to process 11 toys. If we look at these results, we see that the boss-worker model is better in one case, when there are only ten toy orders in the system. And then the pipeline approach is better in the other case, when there are 11 toy orders in the system. **This illustrates the fact that there isn't a single way to say that one pattern or the other is better. As this example illustrates, the answer to that can depend very much on the input that that application receives. For one input, an input of ten toy orders, one implementation is better, whereas for another input of 11 toy orders, the other implementation is better. And finally, you should note that we really simplified the calculation of the execution times of the different models because we ignored overheads due to synchronization, overheads due to passing data among the threads through the shared memory queues**. In reality, you'd actually have to perform a little bit more complex experimental analysis to come up with these answers and draw conclusions as to which pattern is better suited for a particular application.

46 - Lesson Summary



Lesson Summary

Threads and Concurrency

- What are threads?
  How and why do we use them?
- Thread mechanisms
  - mutexes, condition variables
- Using threads
  - problems, solutions, and design approaches