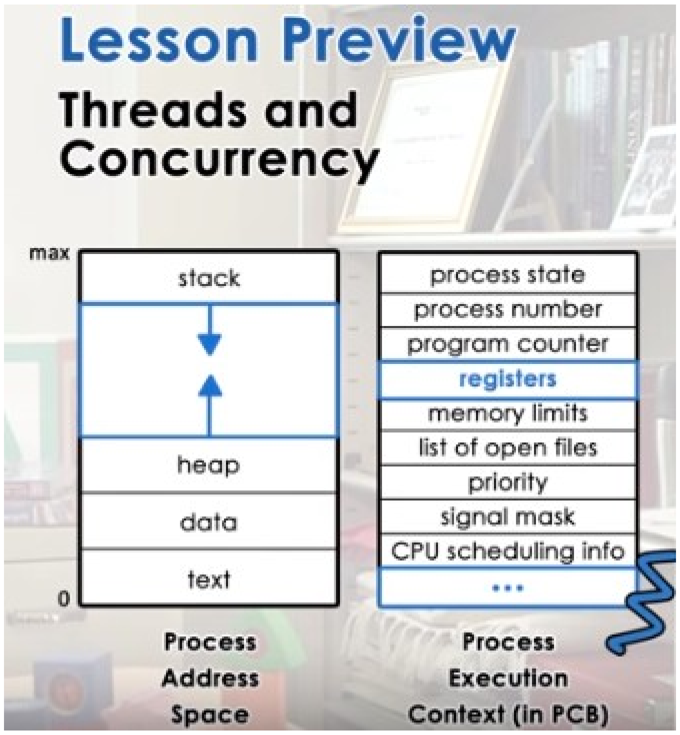
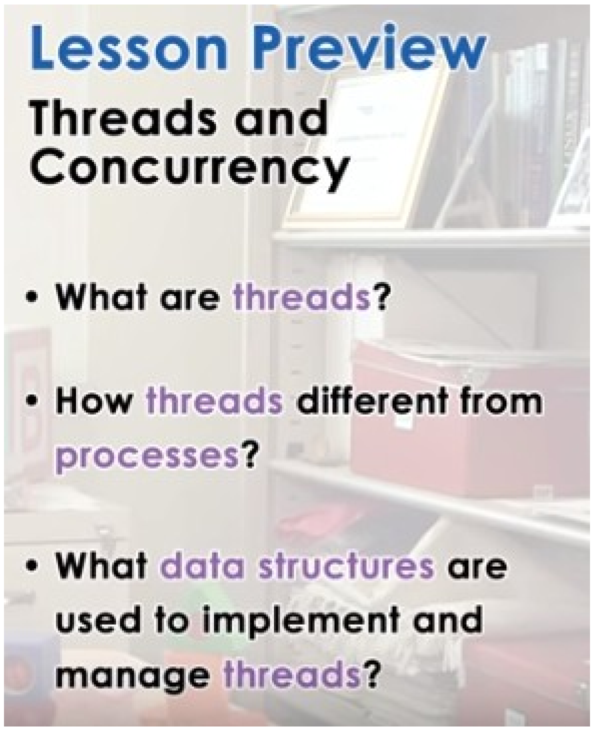
thread /θr'ɛd/

若一個 thread 沒有 acquire lock, 則它會被堵在 lock 語句之前: 第 31 段
concurrency 就是指 '不同 thread 同時改一個變量值': MO
synchronization 就是指 '用 mutex 和 condition variable 來防止 不同 thread 同時改一個變量值 ' 的這麼個過程: 第 9 段



Lesson Preview

Threads and Concurrency

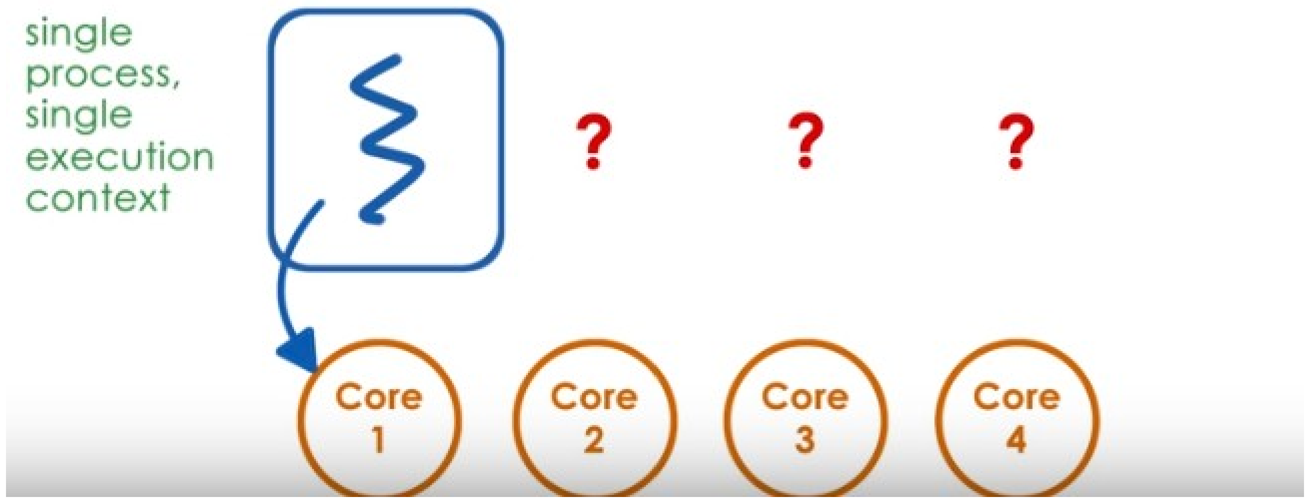


Lesson Preview

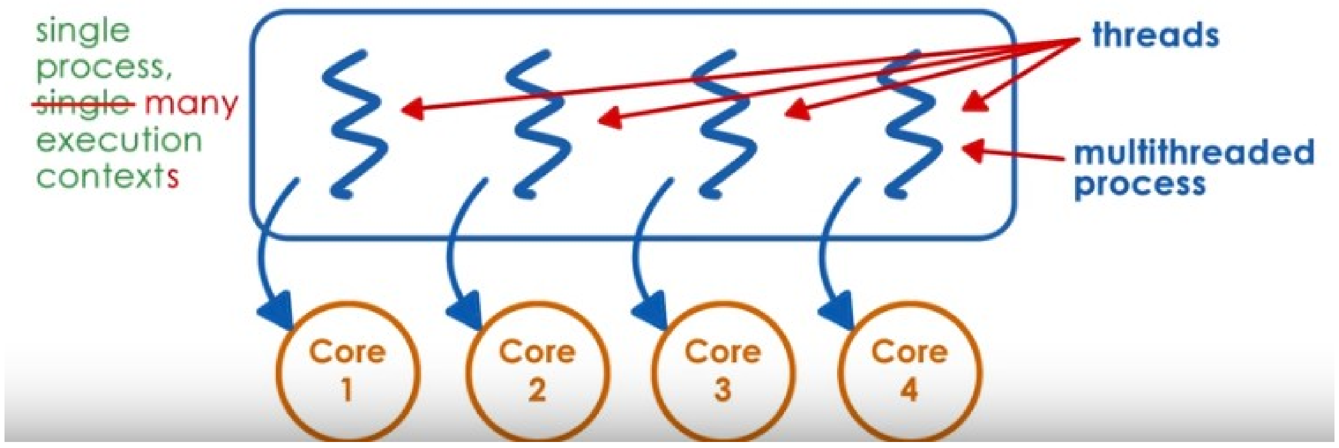
Threads and Concurrency

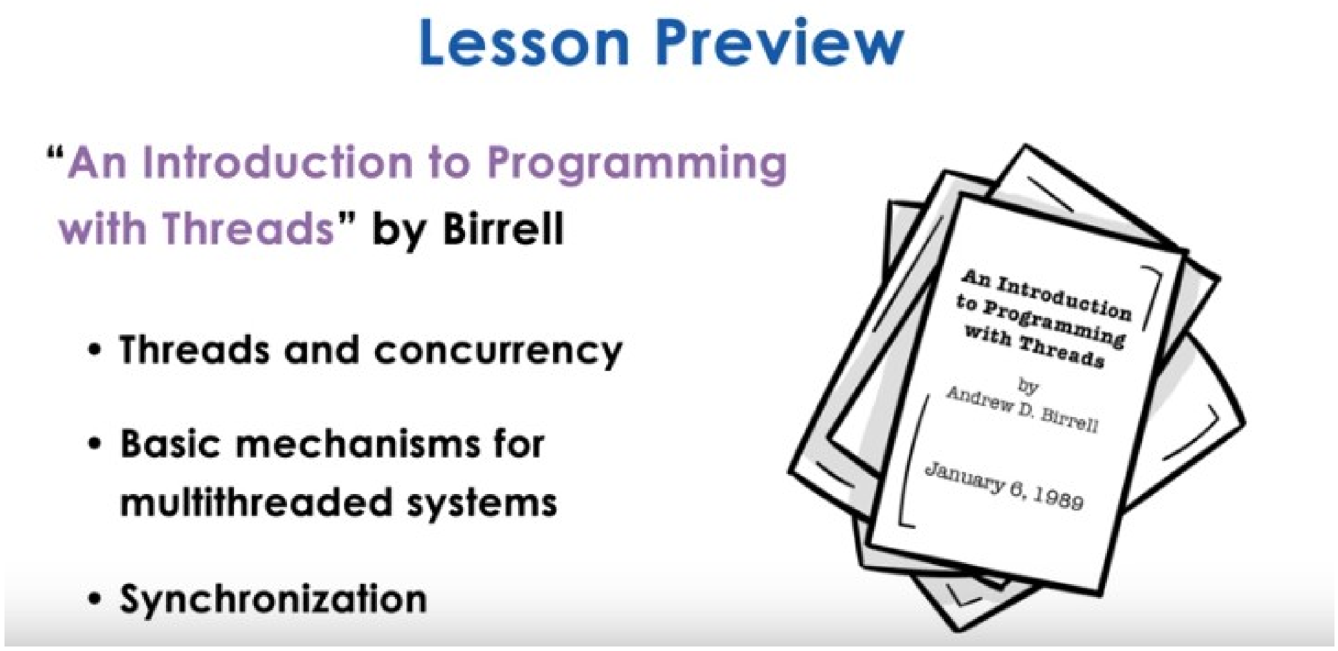
- What are threads?
- How threads different from processes?
- What data structures are used to implement and manage threads?

What if multiple CPUs?



Threads to the rescue!





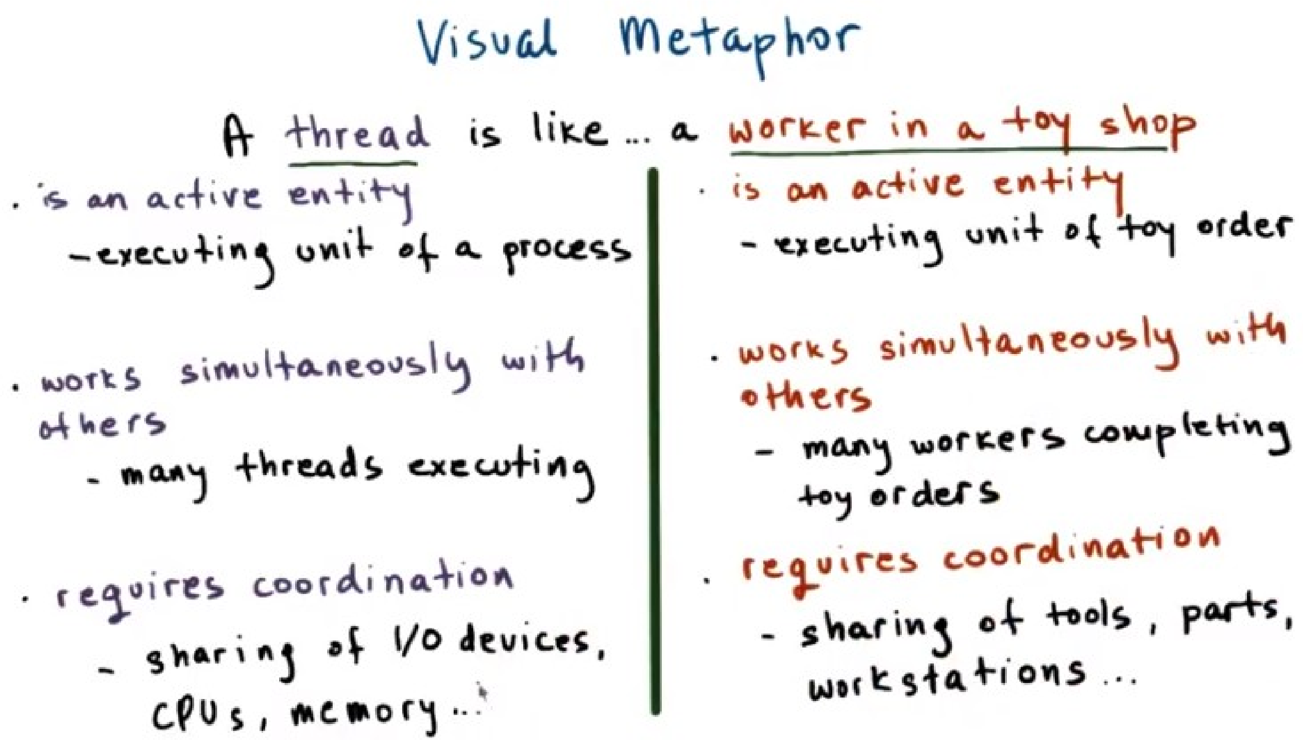
Lesson Preview

"An Introduction to Programming with Threads" by Birrell

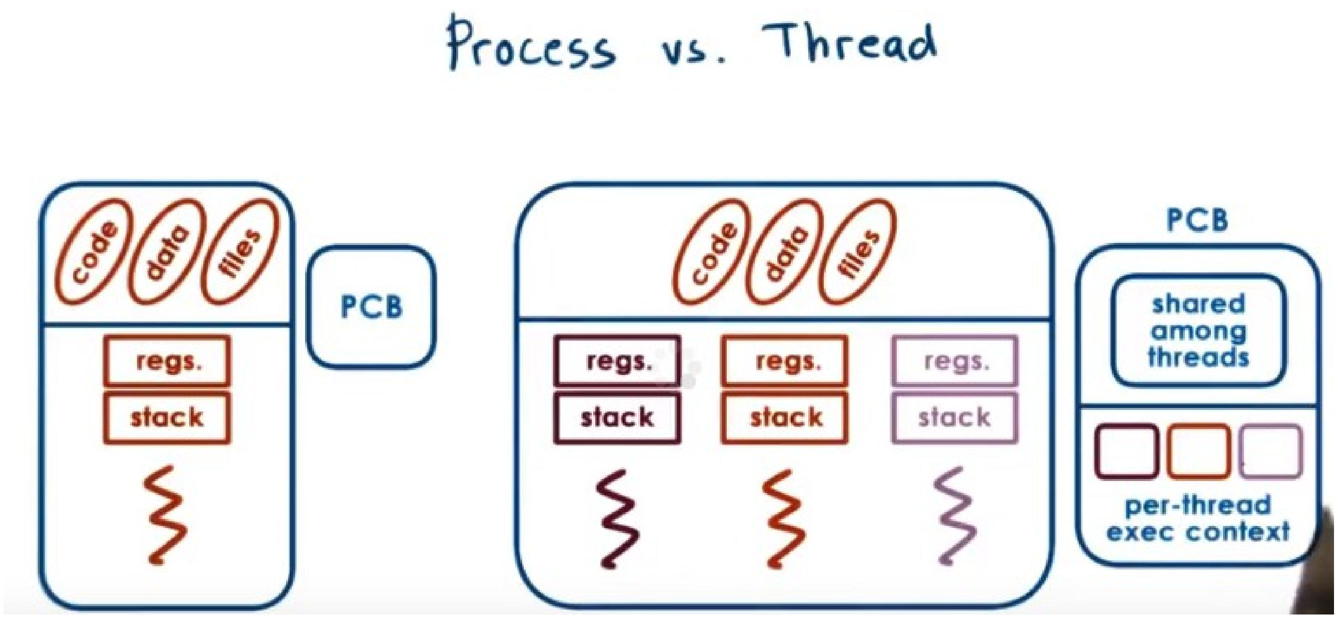
- Threads and concurrency
- Basic mechanisms for multithreaded systems
- Synchronization

An Introduction to Programming with Threads
by
Andrew D. Birrell
January 6, 1989

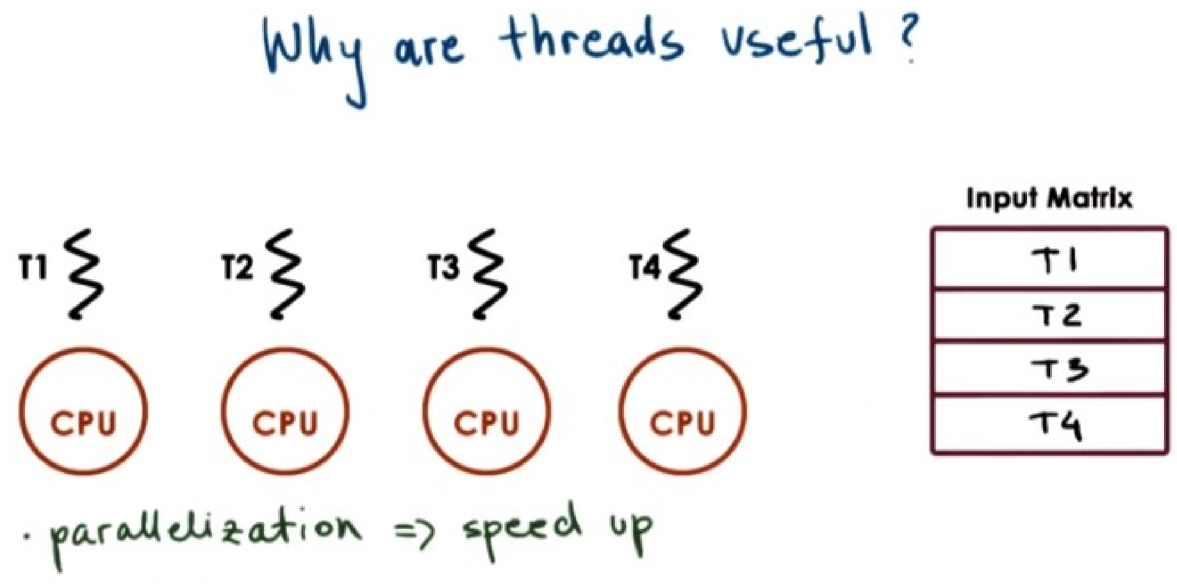
1. During the last lecture, we talked about processes and process management. We said that a process is represented with its address space and its execution context. And this has registers and stack and stack pointer. This type of process, the process that's represented in this way, can only execute at one CPU at a given point of time. If we want a process to be able to execute on multiple CPUs, to take advantage of multi-CPU systems, of multi-core systems today, that process has to have multiple execution contexts. We call such execution context 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 to 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 fundamentals of multithreading, synchronization, and concurrency. In a later lesson, we will talk about a concrete multithreading system called Pthreads.

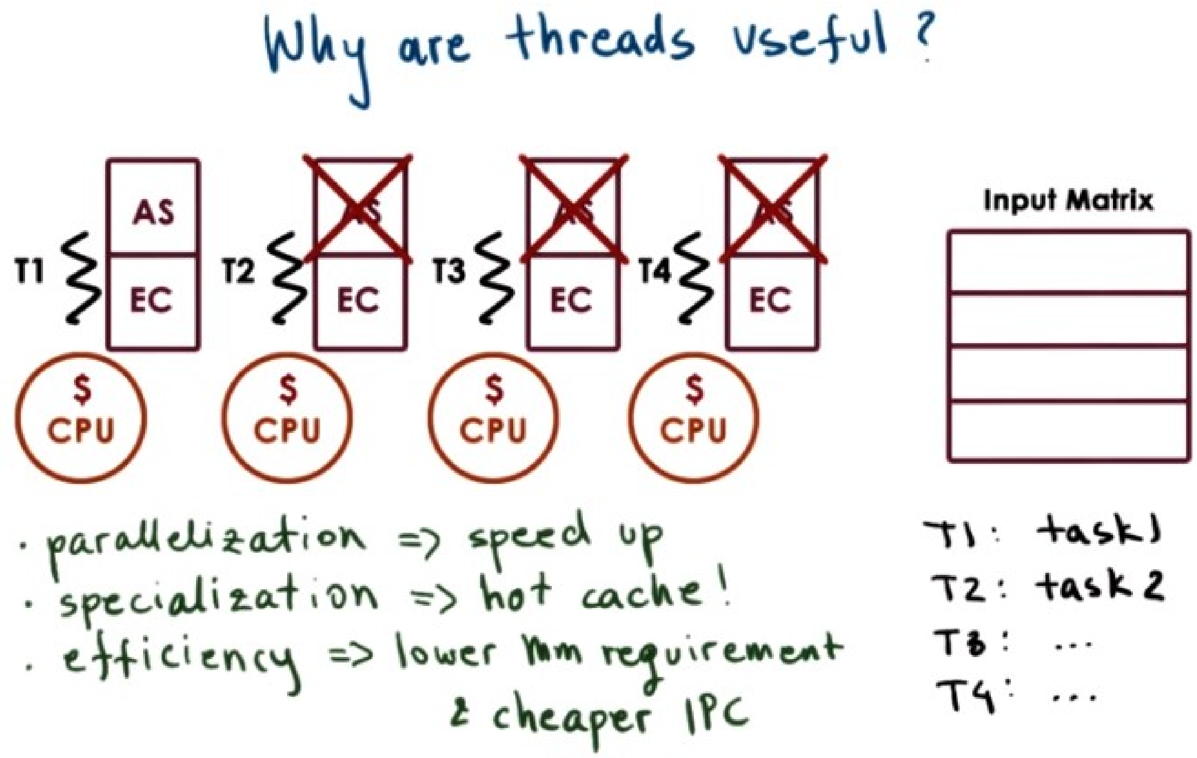

Visual Metaphor
A thread is like ... a worker in a toy shop
· is an active entity
- executing unit of a process
· works simultaneously with others
- many threads executing
· requires coordination
- sharing of I/O devices, CPUs, memory ...
· is an active entity
- executing unit of toy order
· works simultaneously with others
- many workers completing toy orders
· requires coordination
- sharing of tools, parts, workstations ...

2. 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.



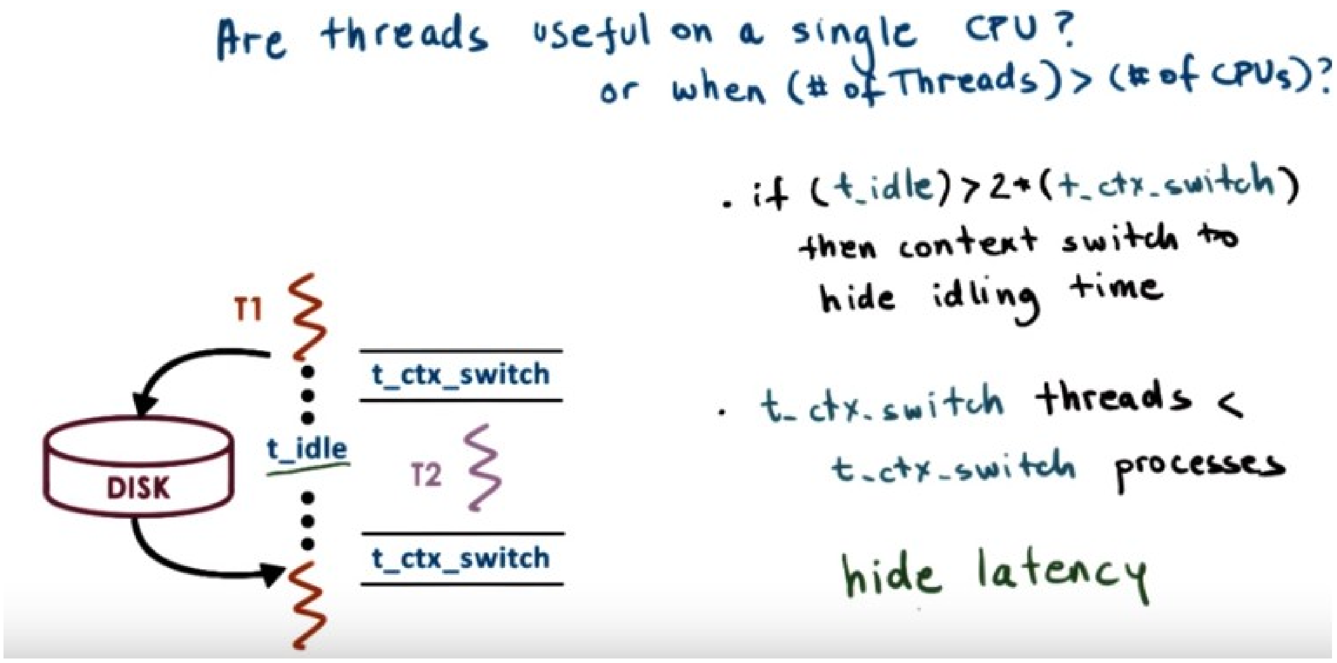
3. Let's try to understand better the differences between a process and a thread. To recap from the previous lessons, a single thread of process is represented by its address space. The address space will contain all of the virtual to physical address mappings for the process, for its code, its data. Keep section files for everything. The process is also represented by its execution context 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. 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. 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. 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



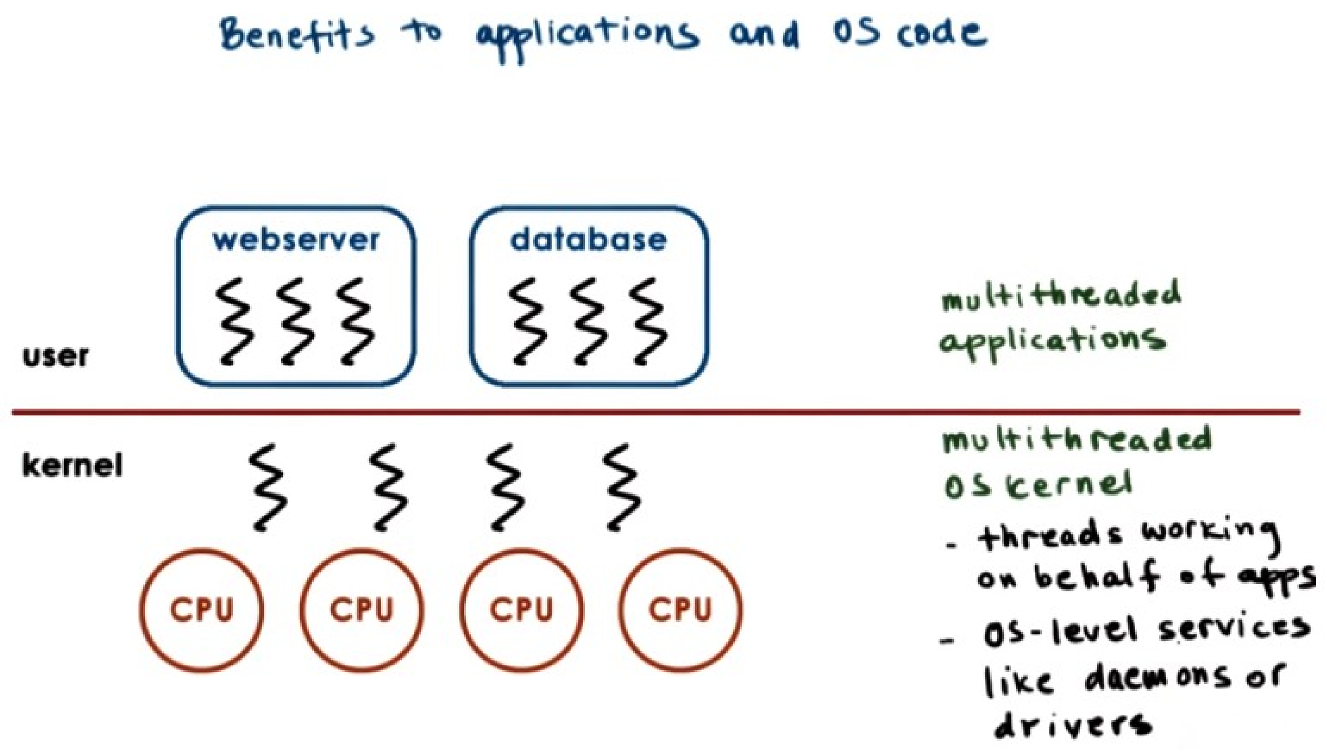


4. Lets now discuss why are threads useful. We will do that by looking at an example on a multiprocessor or a multicore system. At any given point of time when running a single process, there may be multiple threads belonging to the process, each running concurrently on a different processor. One possibility is that each thread executes the same code, but for a different subset of the input. For instance for a different portion of 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, although the threads execute the exact same code, 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 counters, et cetera. 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 input/output 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 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. Another important benefit of partitioning what exactly are the operations executed by each thread, and on each CPU, comes from the fact that performance is dependent on how much state can be present in the processor cache. 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 than 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 multi-process application where every single processor runs a separate process? If we do that, since the processes do not share an address space we have to allocate for every single one of these contacts address space and execution context. So, the memory requirements, if this were a multi-process implementation, would be that we have to have four address space allocations and four execution context allocations. A multi-threaded implementation results in threads sharing an address space so we don't need to allocate memory for all of the address space information for these remaining execution contexts. This implies that a multi-threaded application is more memory efficient. It has lower memory requirements than its multiprocessor alternative. As a result of that, the application is more likely to fit in memory and not required as many swaps from disk compared to a multi-processed alternative. Another issue is that communicating data, passing data among processes or among processes requires interprocess communication 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 address spaced. So it does not require that same kind of costly interprocess communication (IPC). In summary, multithreaded programs are more efficient in their resource requirements than multiprocess programs and incur lower overheads for their inter thread communication then the corresponding interprocess alternatives.

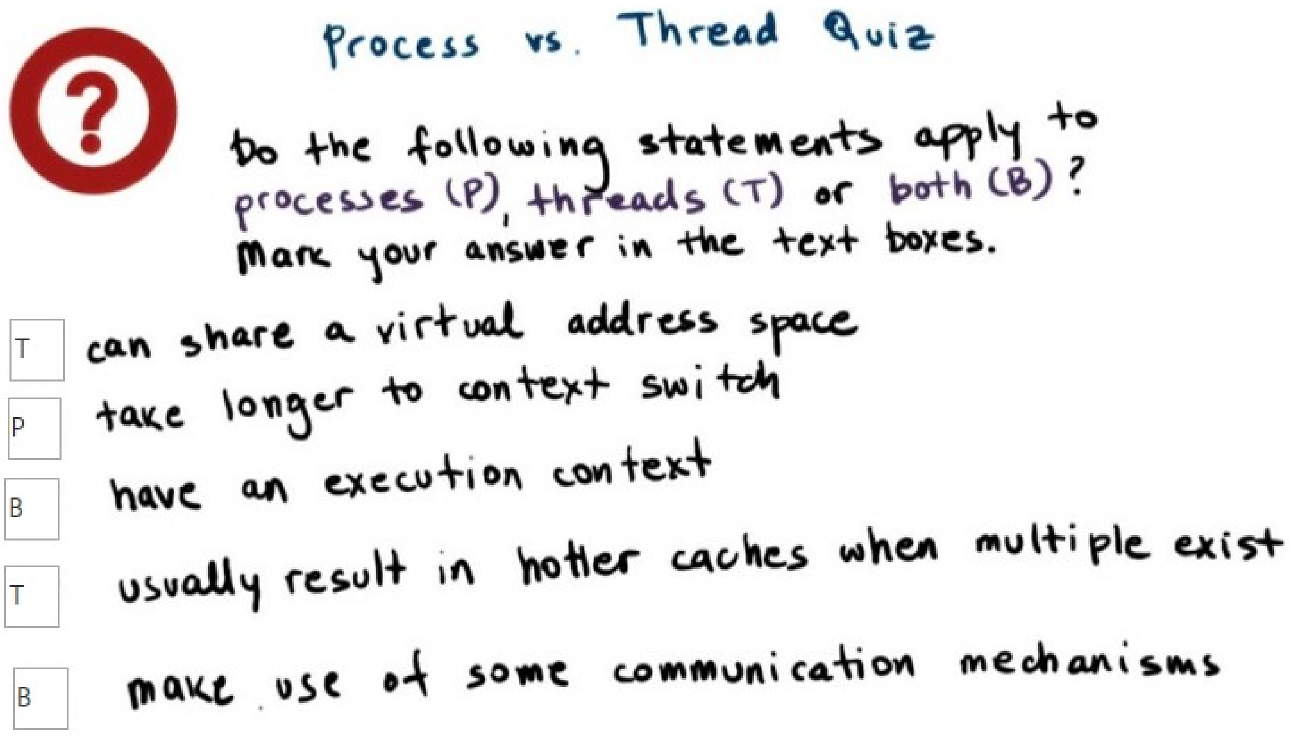


5. One question that you can ask is also, are threads useful on a single CPU? Or even more generally, are threads useful when the number of threads is greater than the number of CPUs? To answer this question, let's consider a situation where a single thread, T1, makes a disk request. As soon as the request 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, thread 1 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 thread one to some other thread, T2, and have that thread do something useful. Or specifically rather, we need the execution context that's waiting on some event to be waiting for an amount of time that's longer than, really, two context switches (ctx_switch). So that it would make sense to switch to another thread. Have that thread perform some operation. And then switch back. So basically as long the time to context switch, t context 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, 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, that will be scheduled. Given that threads share an address space. When we're context switching among threads, it is not necessary to recreate 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 to context switch among processes. The shorter the contact switching time is there will be more of these t_idle situations when a thread is idling where it will make sense to contact switch to another thread and hide the wasted, the idling time. Therefore, multithreading is especially useful because it allow us to hide more of the latency(等待時間) that's associated with IO operations and this is useful even in a single CPU.


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

6. There are benefits of multithreading both to applications when we have multithreaded applications, but also to the operating system, itself. By multithreading the operating system's kernel, we allow the operating system to support multiple execution contexts. And this is particularly useful when there are, in fact, multiple CPUs, so that the operating system context can execute concurrently on different CPUs in a multiprocessor or multicore platform. The operating system's threads may run on behalf of certain applications. Or, they may also run some operating system level services, like certain daemons or device drivers.

7. 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.

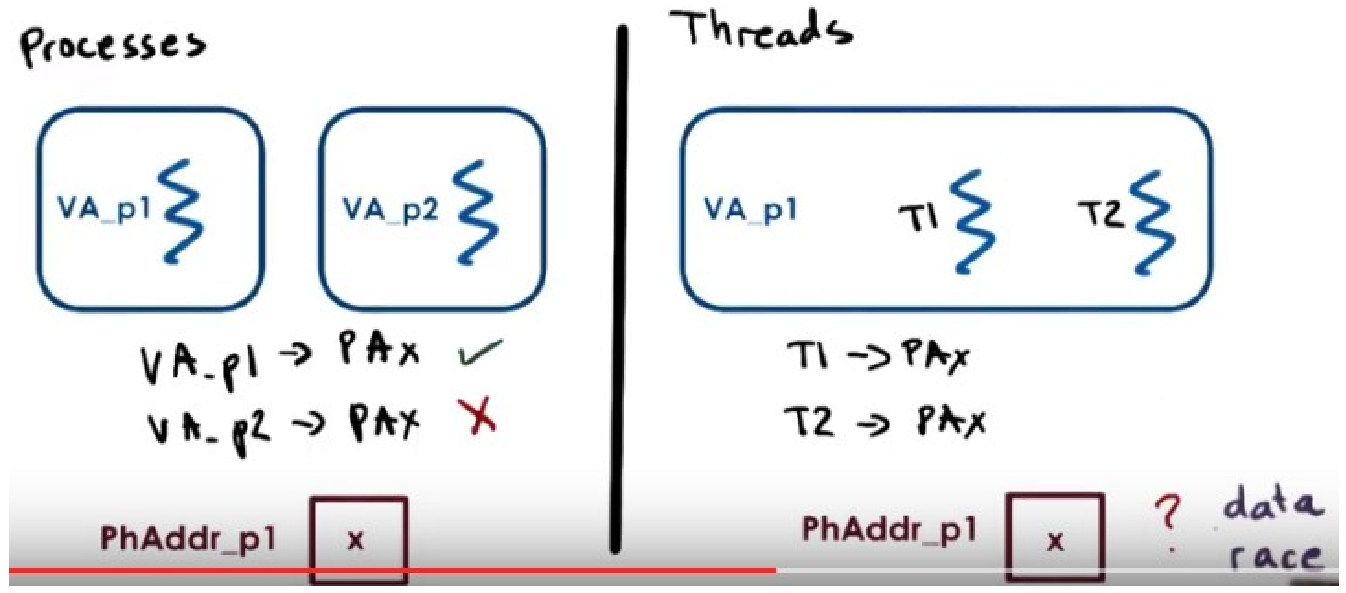
T	can share a virtual address space
P	take longer to context switch
B	have an execution context
T	usually result in hotter caches when multiple exist
B	make use of some communication mechanisms

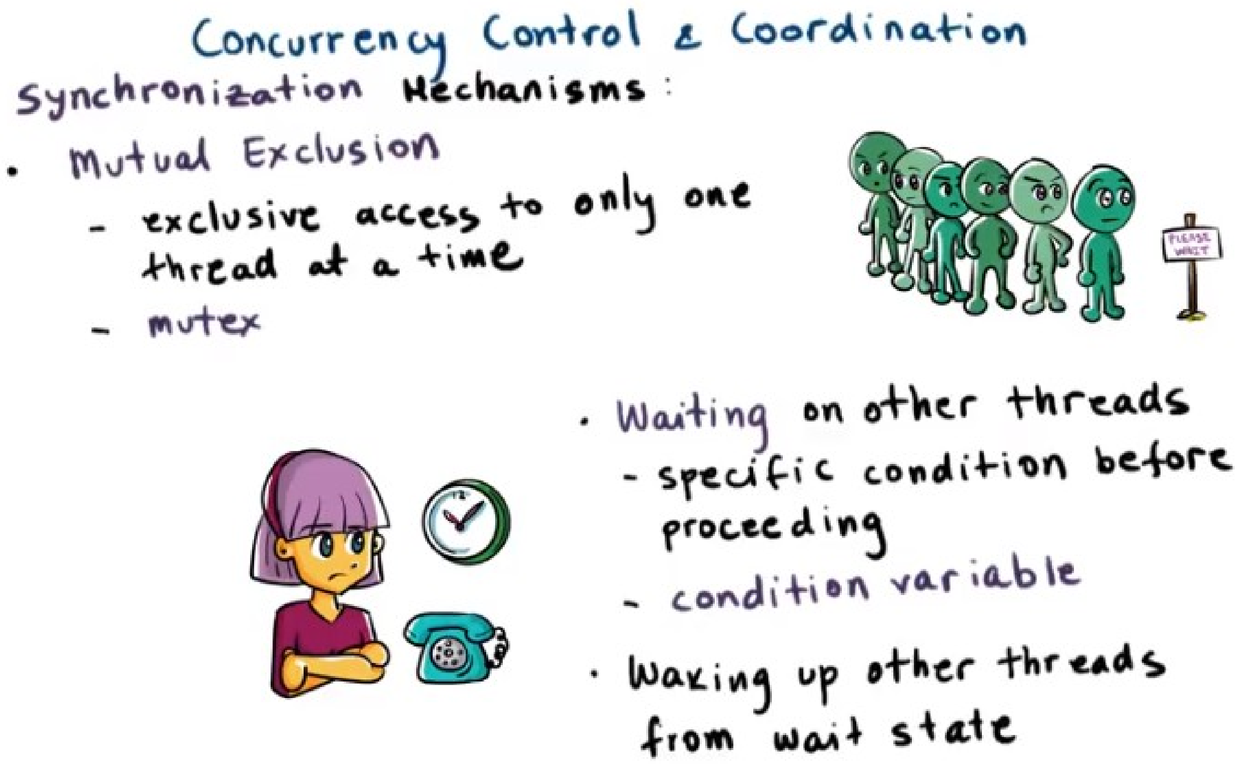
8. The first statement applies to threads. Each thread belonging to a process shares the virtual address space with other threads in that process. And because threads share an address space, the context switch among them happens faster than processes. So, processes take longer to context switch. Both 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.

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

Threads and Concurrency




PLEASE WAIT

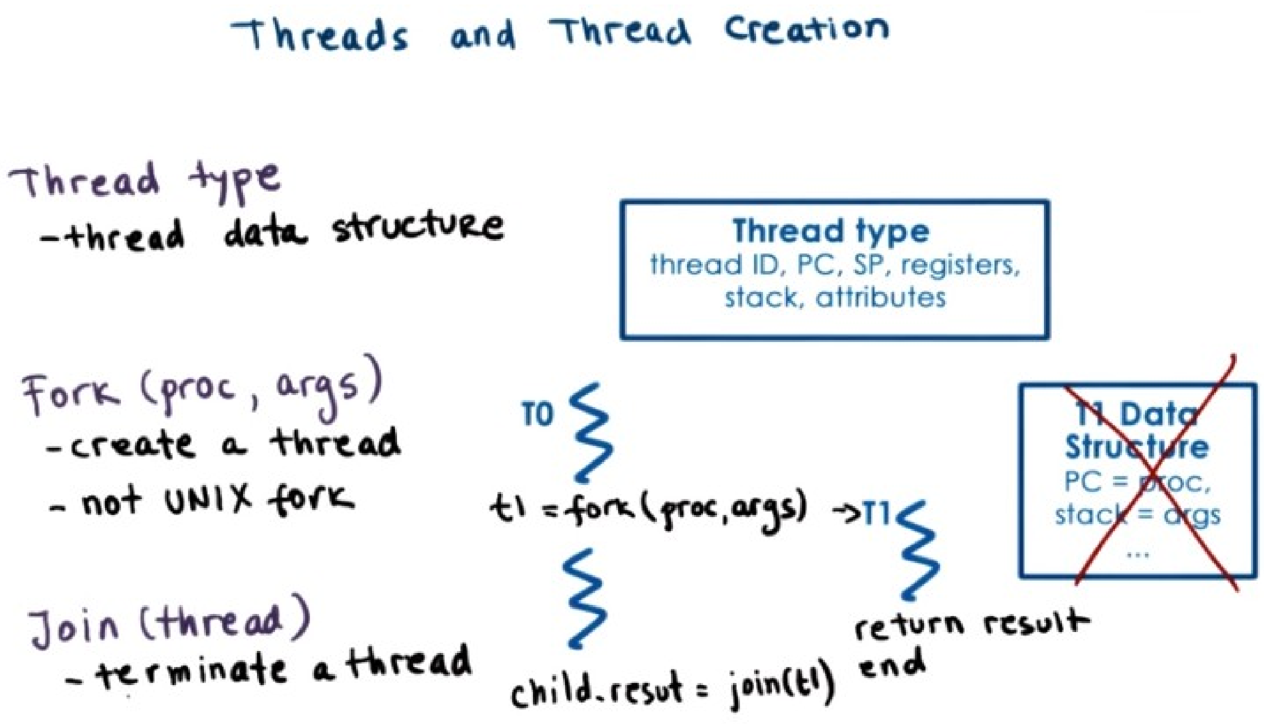
Concurrency Control & Coordination

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

9. 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. 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

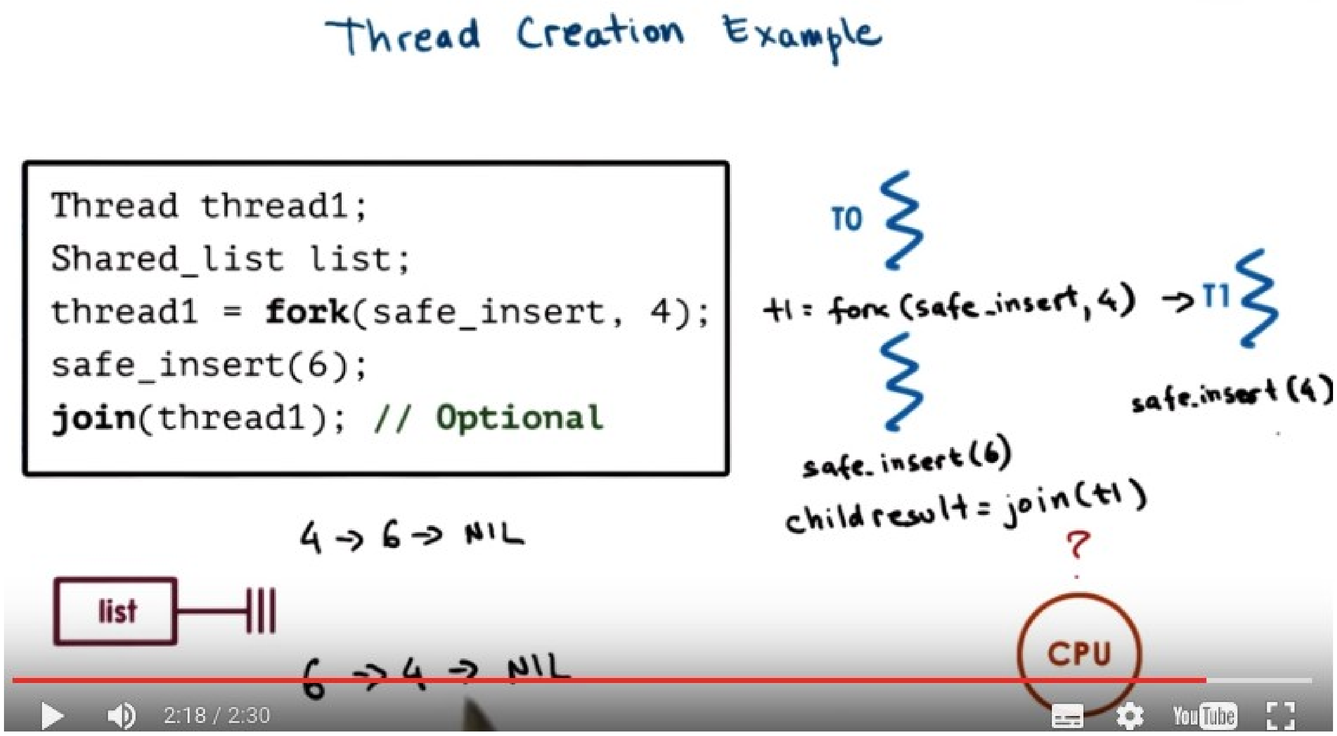
the same time and their updates sort of overlap. This type of data race problem where multiple threads are accessing the same data at the same time is common in multithreaded environments, where threads execute concurrently. To deal with these concurrency issues, we need mechanisms for threads to execute 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 (我: mutex 應該就是 mutual exclusive 的合併). 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. 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 (由此可知, synchronization 就是指 '用 mutex 和 condition variable 來防止不同 thread 同時操作' 的這麼個過程). We will discuss this issue a little bit more in following lessons.



上圖中的 PC 即 program counter, SP 即 stack pointer.

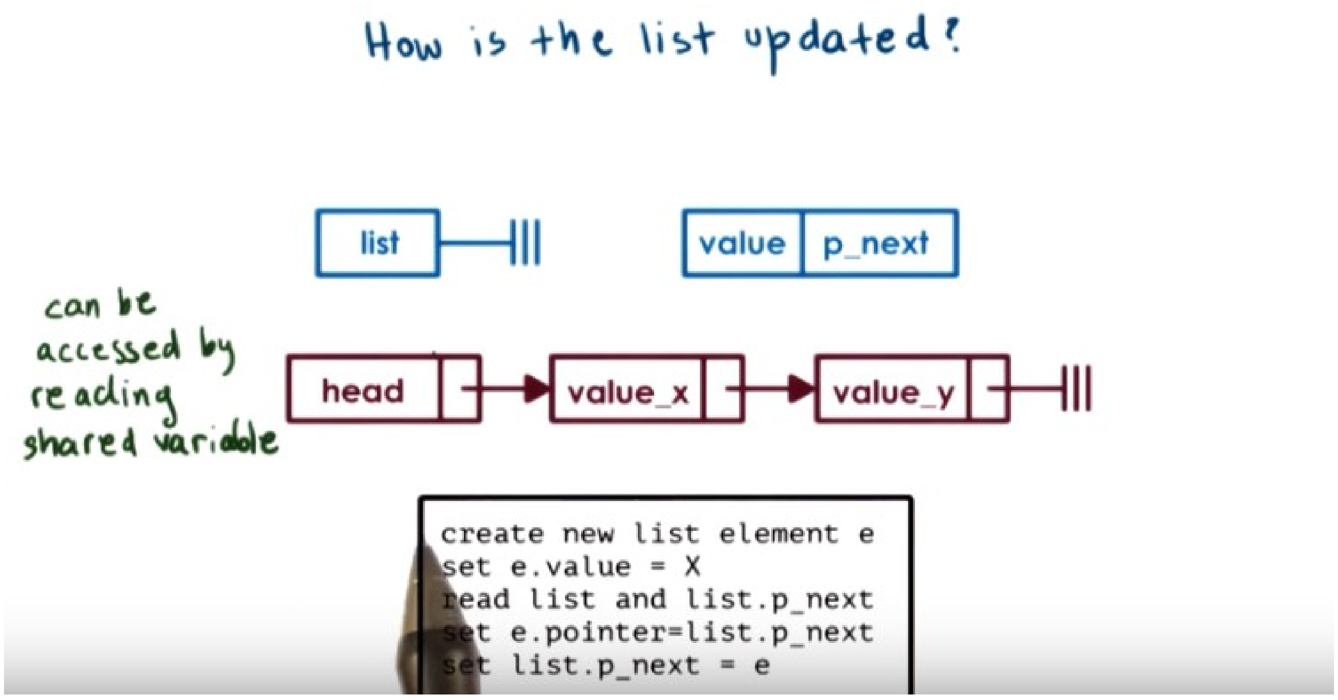
10. Let's first look at how threads should be represented by an operating system or a system library that

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 Burrell's paper. These don't necessarily correspond to some interfaces that are available in the real threading systems or, 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 operating systems. So that will make the discussion a little bit more concrete. You can think of this lesson and the content of Burrell's paper as explaining this content at a more fundamental level. But first we need some data structure to represent a thread actually. The thread type proposed by Burrell is the data structure that contains all information that's specific to a thread and that 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 program counter and the stack pointer, the stack of the thread, and any other thread specific data or attributes. 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, Burrell 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 called fork that we previously discussed. The Unix system call creates a new process and 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 new thread data structure of this type is created and its fields are initialized such that its program counter will point to first instruction of the procedure proc, and these arguments will be available on this stack of the thread. 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 proc, and now somehow it needs to return that result. Or, it may be just some, 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 they 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 the 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 portion of an input array. And the parent thread 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, Burrell proposes a mechanism he calls Join. It has the following semantic. (對 join 的理解 1/2) 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 structures saved for the child, all of the resources that were allocated for its execution will be freed and the child is at that point 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 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.



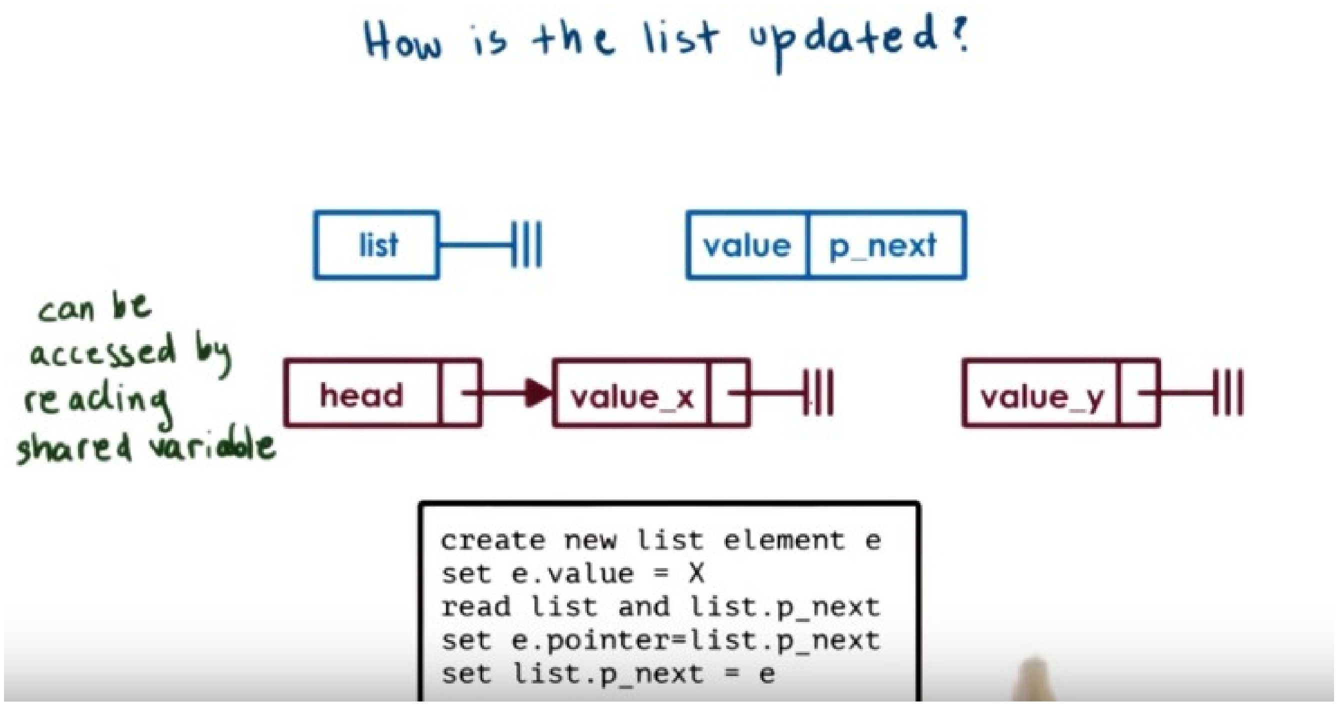
```
Thread thread1;
Shared_list list;
thread1 = fork(safe_insert, 4);
safe_insert(6);
join(thread1); // Optional
```

11. Here's a code snippet illustrating thread creation. Two threads are involved in this system. The parent thread that's executing this code and the child that gets created via this fork. Both threads perform this operations safe_insert which manipulates some shared list that's initially empty, let's say. Let's assume initially the process begins with one parent thread, T0 in this case (方框中的那整段代碼就是 T0, T0 這個 thread 的作用就是: create T1 這個 thread, 並且执行 safe_insert(6). 另一個例子見 P2L3 第 4 段). At some point thread 0 calls fork and it creates a child T1. Now, T1 will need to execute safe_insert with an argument 4. As soon as T1 is created, the parent thread continues its execution and at some point it will reach a point where it calls safe_insert of 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 thread 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 for 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. (對 join 的理解 2/2) Finally, the last operation in this code snippet is this join. So we're calling join with T1. If this join is called when 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 results of the child thread regardless.


How is the list updated?

```
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
```

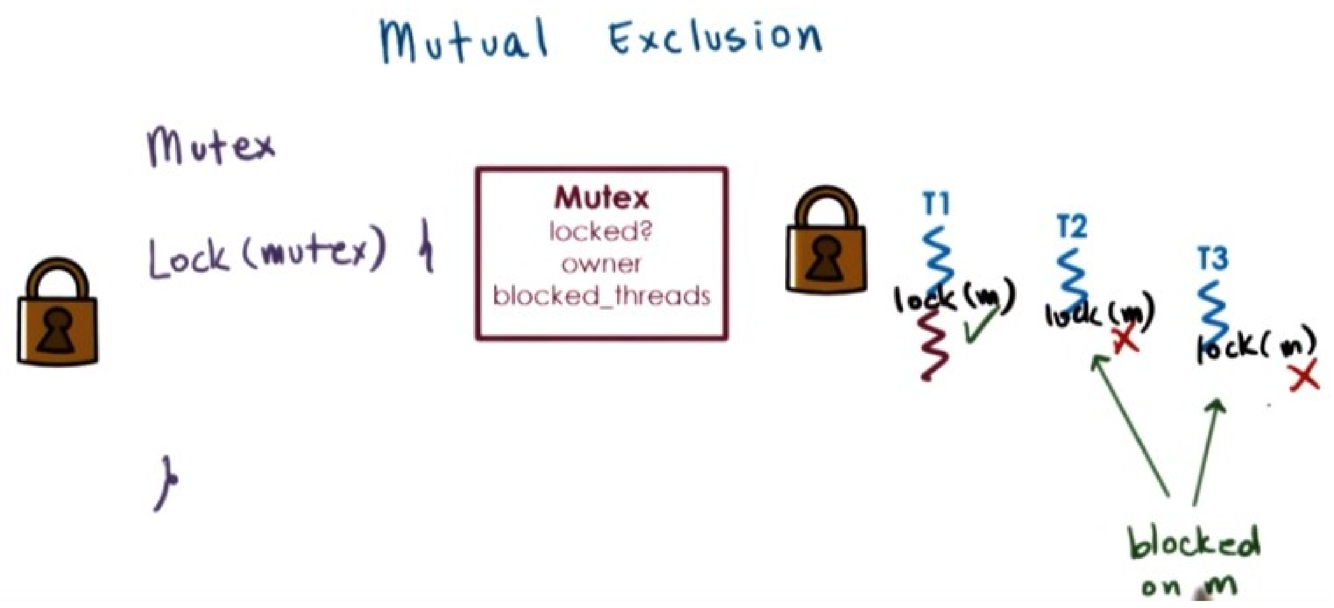
以上代碼中的 list 是指 head 這個節點, e 即 value_x 這個節點, 最後兩句:
set e.pointer = list.p_next, 本句即使 value_x 指向 value_y, 本句中的 list.p_next 即 value_y 那個節點,
set list.p_next = e, 本句即使 head 指向 value_x, 本句中的 list.p_next 即 head 中的 next pointer.

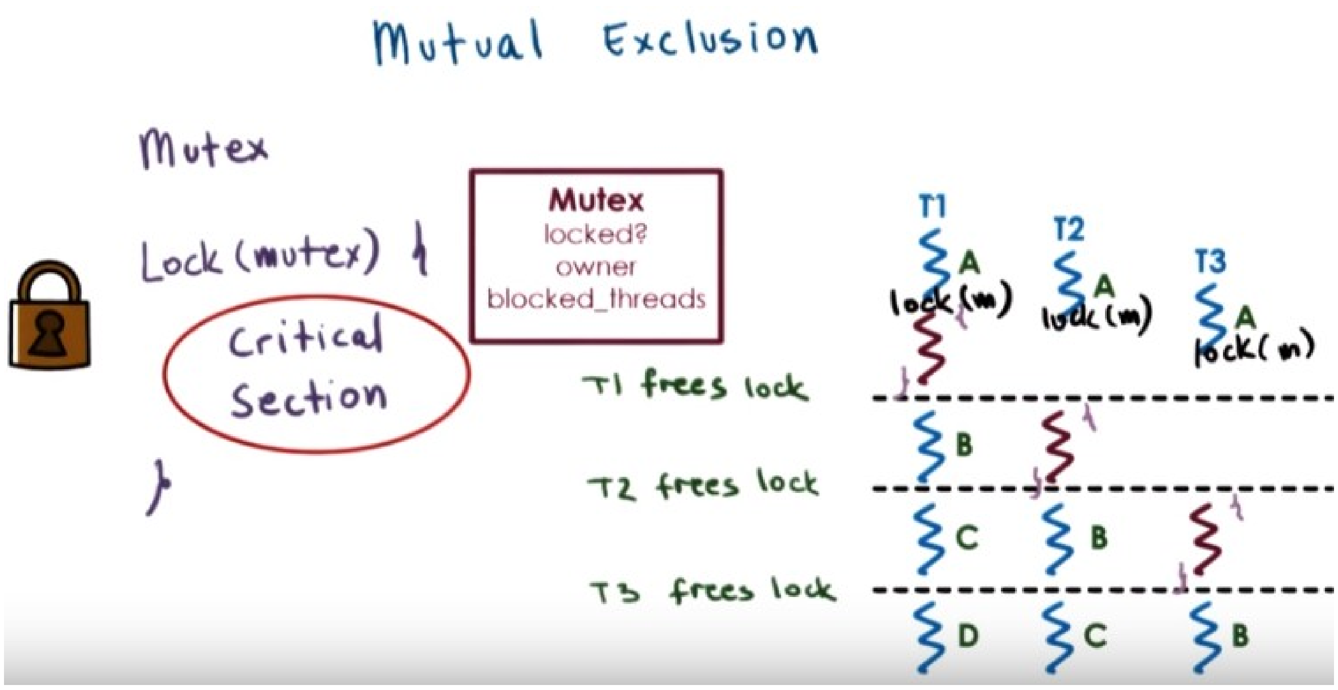

How is the list updated?

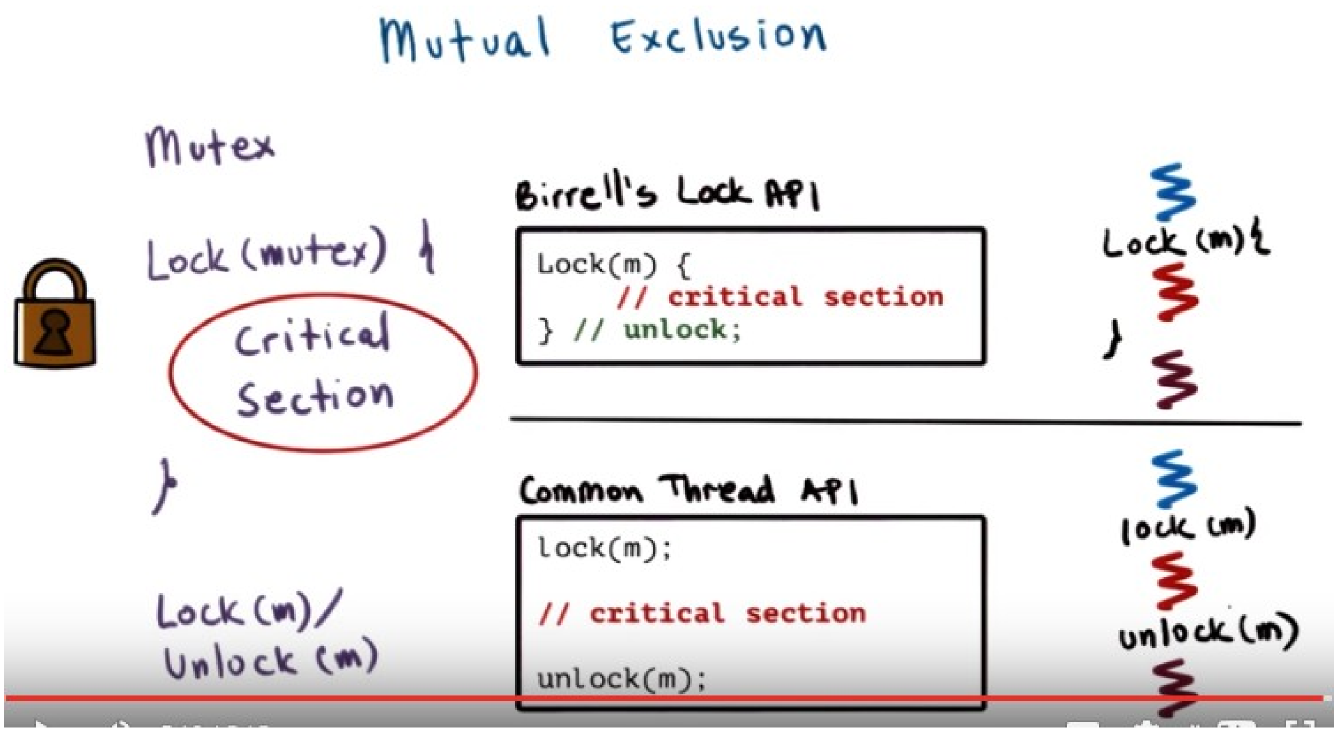
```
create new list element e
set e.value = X
read list and list.p_next
set e.pointer=list.p_next
set list.p_next = e
```

12. >From the previous example where the threads were inserting elements in the list, how does this list actually get updated? An implementation make 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 it 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 it's list pointer element. Then, each thread that needs to insert a new element in the list, will first create an element and set it's value. And we'll 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 it will have to 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 value Y. Set the new elements pointer to point to value Y. And then set the head pointer to actual point to the newly created element. And in this way, new elements are basically inserted at the head of the list. The end of 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 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 their operations are randomly interleaved. (剩下的這些字沒看懂) For instance, they may both read the initial value of the list. So this is where they're both reading the value of the list. And it's pointer to the next element of the list. In this case null. They were 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.







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

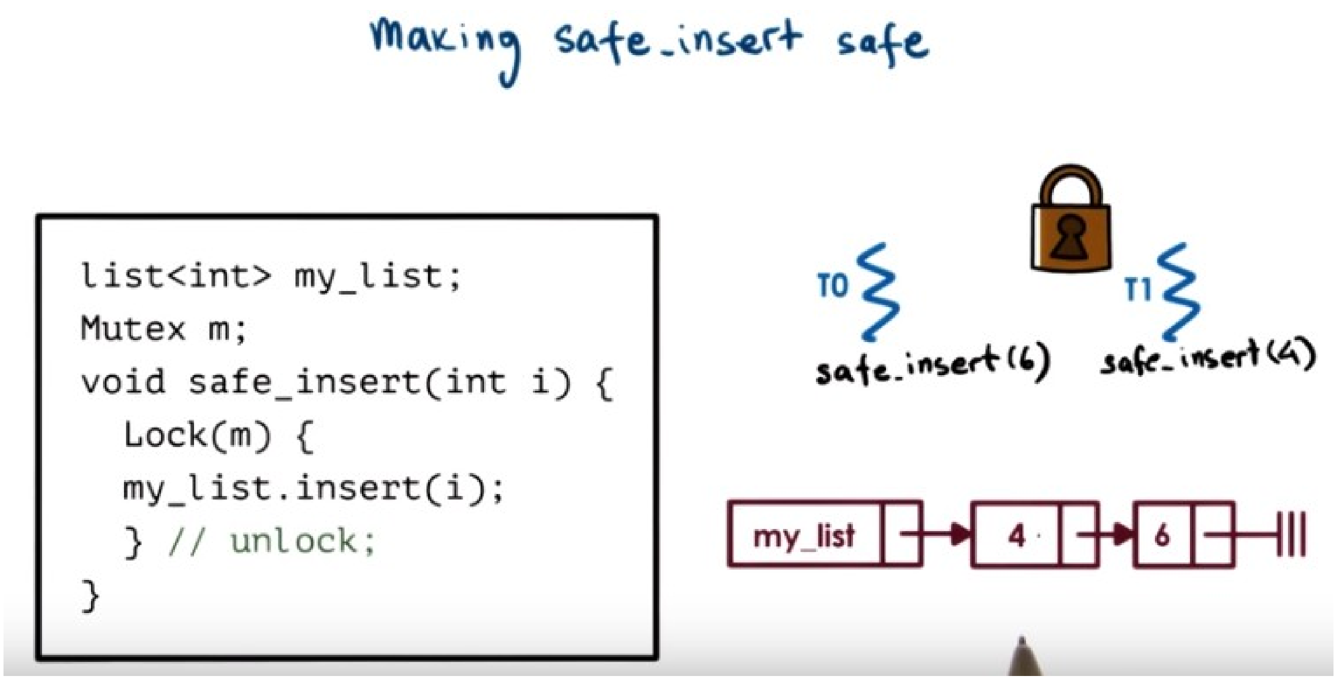
```
lock(m);

// critical section

unlock(m);
```

13. 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 overriding the list elements. This illustrates a key challenge with multi-threading that there is a need for a mechanism to enable mutual exclusion among the execution of concurrent threads. To do this, operating systems and threading libraries in general support a construct called mutex. A mutex is like a lock that should be used whenever accessing data or stayed that's shared among threads. When a thread locks a mutex, it has exclusive access to the

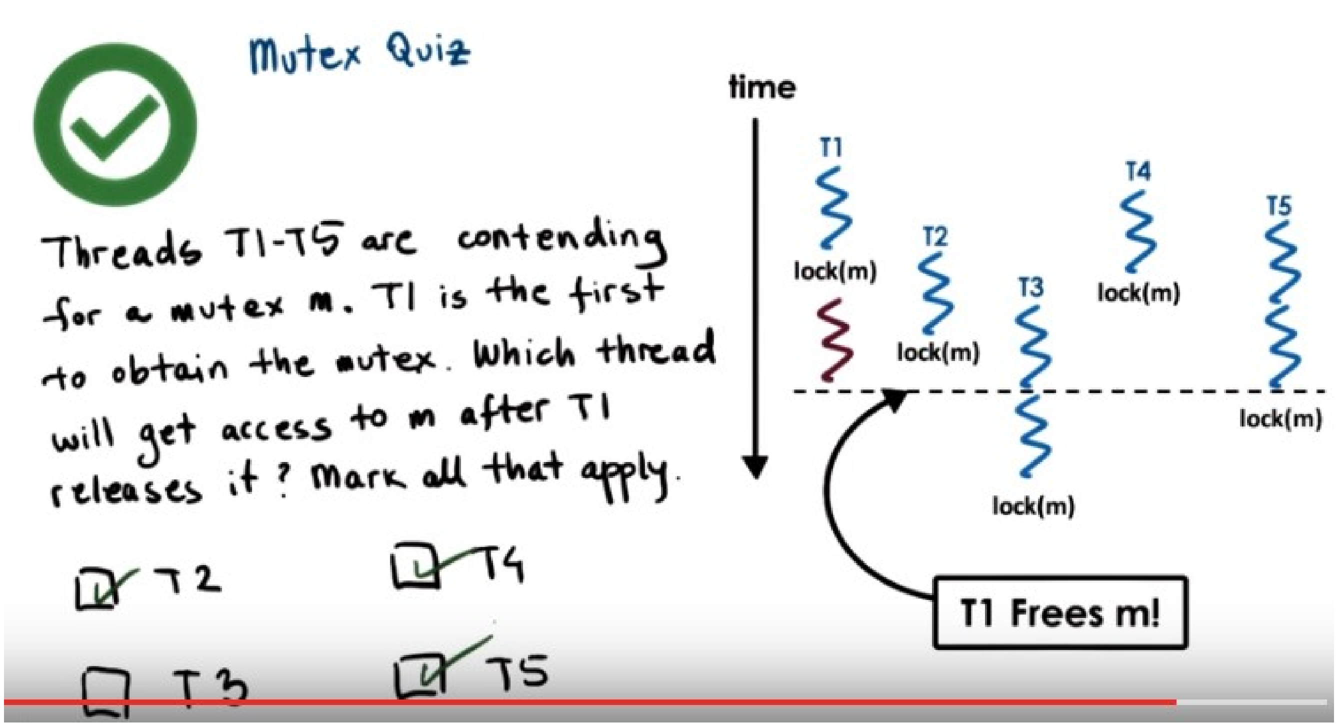
shared resource. Other threads attempting to lock the same mutex are not going to be successful. We also use the term, acquire the lock or acquire the mutex, to refer to this operation. So these unsuccessful threads, they will be what we call blocked on the lock operation, meaning they'll be suspended here, they will not be able to proceed until the mutex owner(一個 thread), 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 a some sort of list of all the threads that are blocked on the mutex and that are waiting for it to be free. Another common element that's part of this mutex data structure is to maintain some information about the owner of the mutex, who currently has the lock. A thread that has successfully locked the mutex, has exclusive rights to it, and can proceed with its execution. In this example, T1 gets access to the mutex and thus continues executing. Two other threads, T2 and T3, that are also trying to log the mutex will not be successful with that operation. They will be blocked on the mutex, and they will have to wait until thread one releases the mutex. The portion of the code protected by the mutex, is called critical section. In paper, this is any code within the curly brackets of the lock operating that he proposes to be used with mutexes. The critical section code should correspond with any kind of operation that requires that only one thread at a time performs this operation. For instance, it can be updated to shared variable like the list or increment a counter. Or performing any type of operation that requires mutual execution between the threads. Other than the critical section code, the rest of the code in the program, the threads execution concurrently. So using the same example, imagine that the three threads need to execute a sequence of code among block A fold 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 two cannot start the critical section until after thread one finishes them. Thread three cannot start its critical section until any prior thread exits the critical section, or releases the lock(因為 thread two 也 lock 了 critical section 的). 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 Burrell, again, the critical section is the code between the curly brackets. And the semantics are such that upon acquiring a mutex, a thread enters the locked block. And then 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 log, or even a brand new thread that's just reaching the (它自己的)lock construct, can start executing the lock operation. For instance, if T3's lock statement coincides with the release of the lock, of thread one. 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 in it. Who will use Burrell's lock construct throughout this lecture. However, you should know that most common APIs have two separate calls, lock and unlock. So, in contrast to Burrell's lock API, in this other interface, the lock/unlock interface, both the lock and the unlock operations must be used both explicitly and carefully when requesting a mutex, and when accessing a critical section. What that means is that we must explicitly lock a mutex before entering a critical section. And then we also must explicitly unlock the mutex when we finish the critical session. Otherwise nobody else will be able to proceed.



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

14. Returning to the previous safe_insert example, let's demonstrate how to use new techniques 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 continued executing and was the first one to reach safe_insert with value 6. It will acquire the log and start inserting the element 6 on the list. What that means, T0 has the log 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.

15. 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 ensure mutual exclusion to that resource. T1 is the first one 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, and T5, and you should mark all that apply.



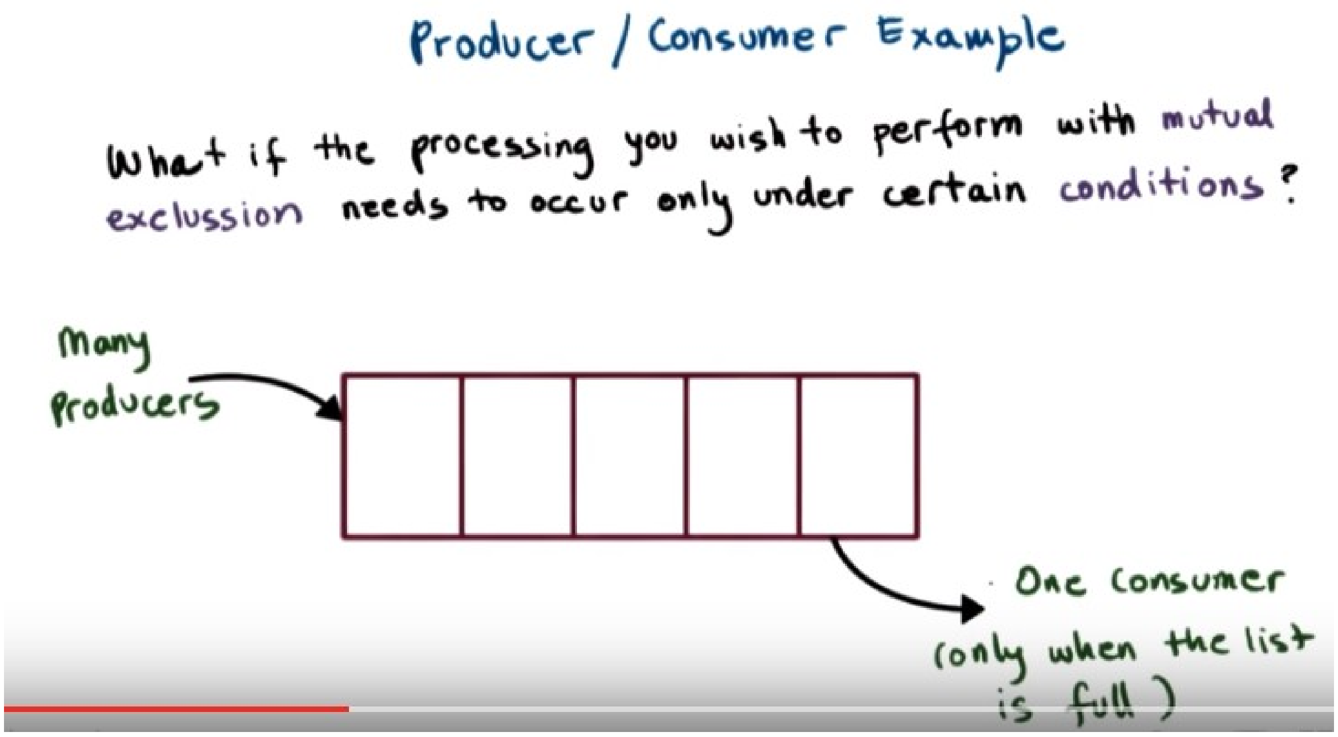
Mutex Quiz

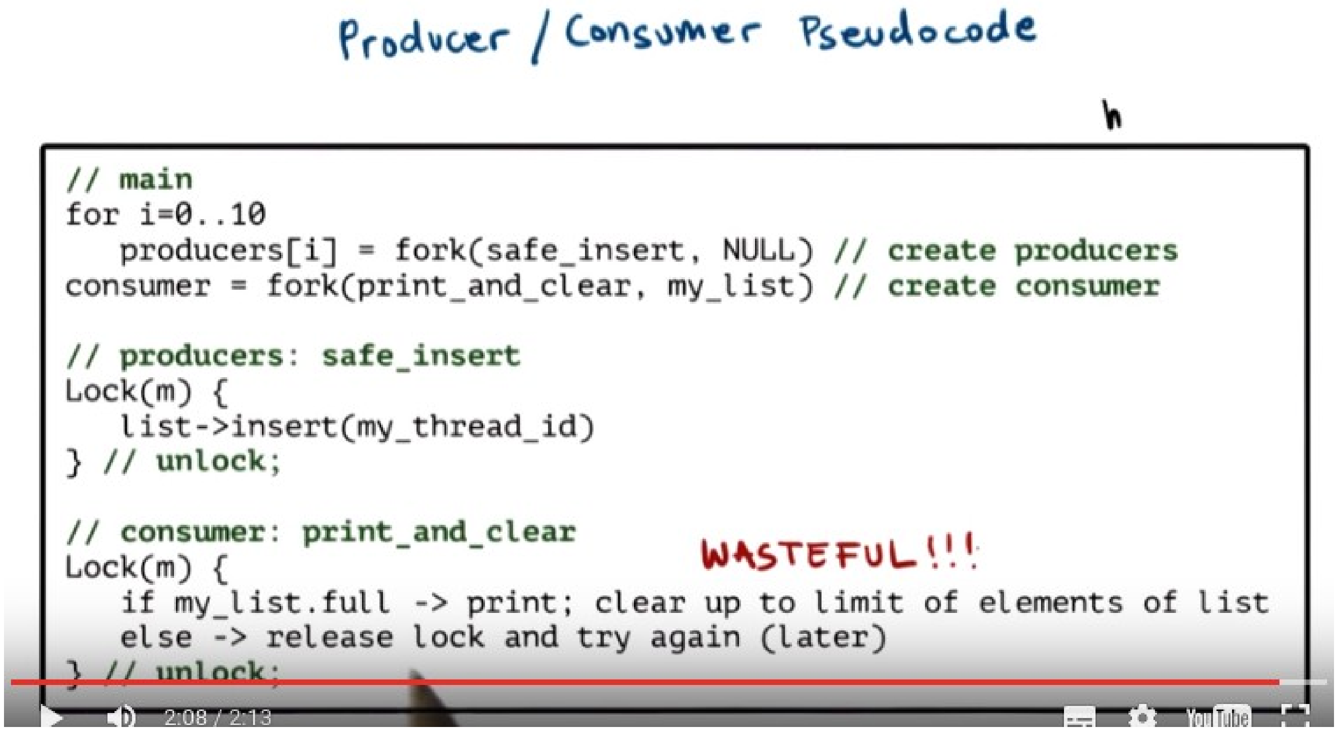
Threads T1-T5 are contending for a mutex m. T1 is the first to obtain the mutex. Which thread will get access to m after T1 releases it? Mark all that apply.

[✓ T2] [✓ T4]

T3 [✓ T5]

16. 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. The specification of the mutex doesn't make any guarantees regarding the ordering of the lock operations. So, it doesn't really matter that the thread 4 issued the lock operation, the lock request before T2. We don't have a guarantee that these requests will be granted in order. But, regardless, since both T2s and T4s requests are pending on the mutex, then either one of these two threads will 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 their already pending requests. So it's not going to be one of the next threads to execute. For T5 it's 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. So, it is the one that gets to execute 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.


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)



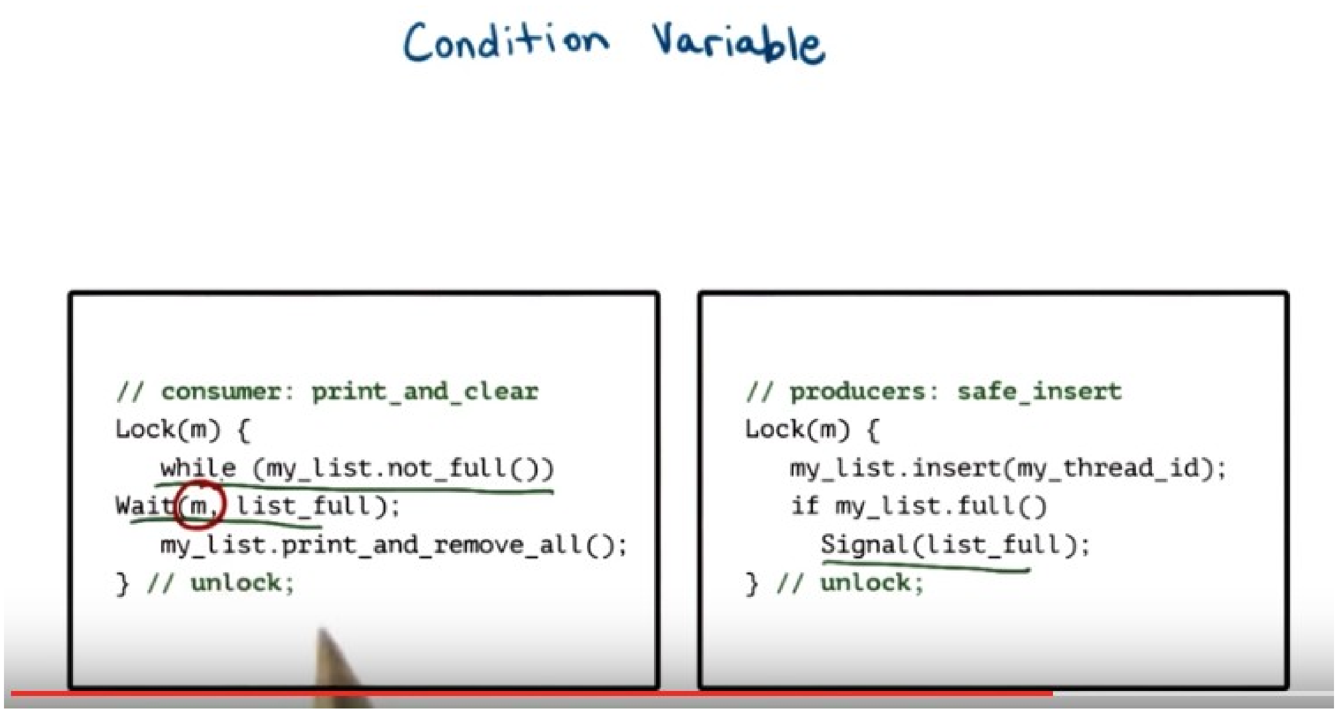
```
// main
for i=0..10
   producers[i] = fork(safe_insert, NULL) // create producers
consumer = fork(print_and_clear, my_list) // create consumer

// 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;
```

由上圖可知, Lock(m)語句是代碼中的, 由於所有 thread 都會执行相同的代碼, 所以所有 thread 都會 Lock(m), 前面幾個圖都是這樣的.

17. 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 it's locked and busy and you have to wait. Once the resource becomes free, you get a chance to try to 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 really gets to execute its operation under these certain conditions when the list is actually full. 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 print, will perform an operation print and clear the list. And this operation, as we said, needs to happen once the 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 element of the list. Otherwise, it immediately releases the lock and then 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.

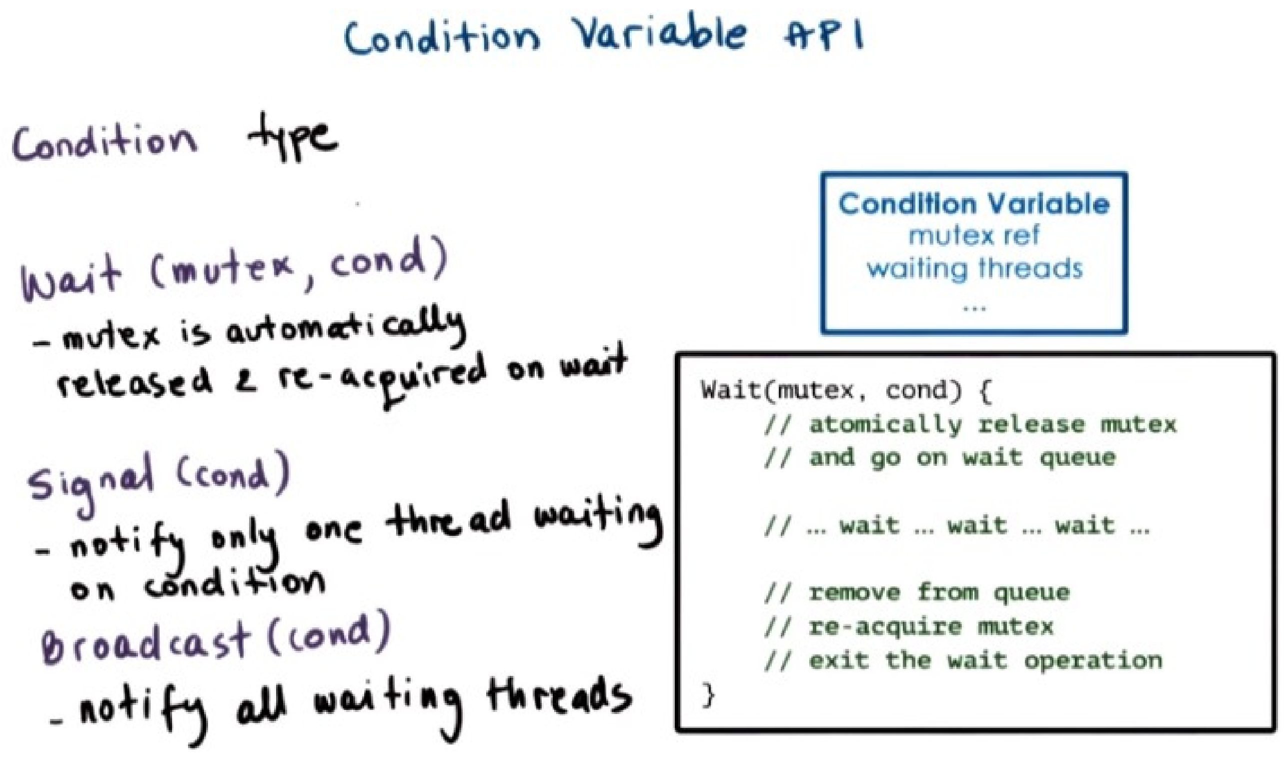


Condition Variable

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

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

18. Burrell recognizes that this can be a common situation in multithreaded environments. And argues for a new construct, a condition variable. He says that such condition variables should 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 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 actually is 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 consumers. Not that while the consumer is waiting, the only way that this predicate that its waiting on can change, so 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. 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 the wait statement. And then automatically reacquired once we come out of the wait statement. Because we see, 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 sees that it must wait it specifies the condition variable(即 list_full, condition variable 之作用就是: 當右邊的 producer 用 Signal(list_full)來 signal 後, 左邊的 consumer 才能跳出 wait. 而 Mutex m 的作用就是 lock: 比如圖中, 當別人(producer) acquire 了這個 mutex 後, 圖中左邊的 consumer 就不能执行 Lock(m){...}中的那段. 更詳細的例子見 P2L3 的第 16 段) that it must wait on. And the wait operation takes its parameters, both the condition variables as well as this mutex. Internally the implementation of the wait must ensure the proper semantics. It must ensure that the mutex is released. And then, when we're actually removed from the wait operation, when this notification is received, we have to reacquire 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



Condition Variable API

Condition type

Wait (mutex, cond)
- mutex is automatically released & 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
...

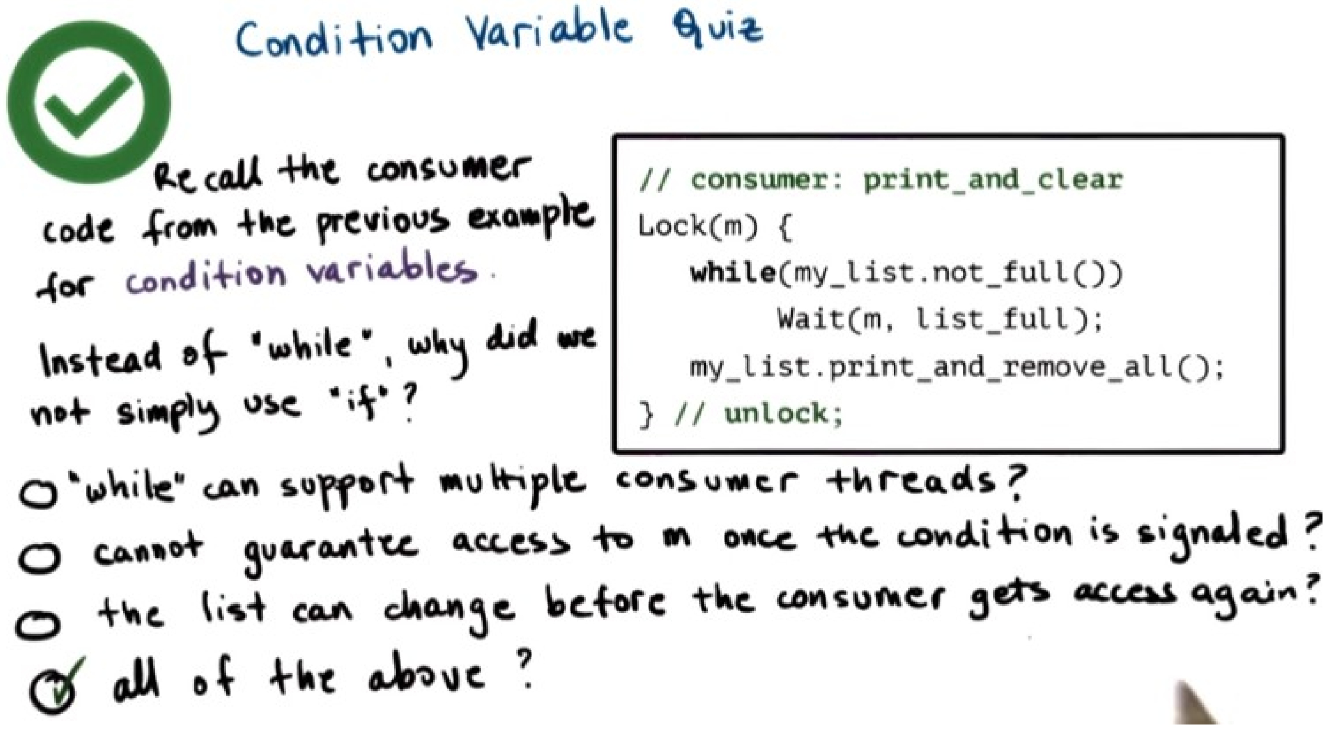
```
Wait(mutex, cond) {
    // atomically release mutex
    // and go on wait queue

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

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

19. To summarize a common condition variable API, we'll look as follows. First, we must be able to create these data structures that correspond to the condition variable, so there must be some type that corresponds to condition variables. Then there is the wait construct that takes as arguments the mutex and the condition variable. Where a mutex is automatically released and re-acquired 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 threads one at a time. And for this, Birrell proposes a signal API. Or it could also be useful to wake up all of 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's associated with the condition so that this wait operation can be implemented correctly so that this mutex can be both released and re-acquired as necessary. As a reference, the way the wait operation would be implemented in a operation system or in a threading library that supports threads and that 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 this thread is placed on this wait queue of waiting threads. And then at some point when a notification is received, what would have to happen is that 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, you know, when on 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 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 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.

20. 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?



Condition Variable Quiz

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

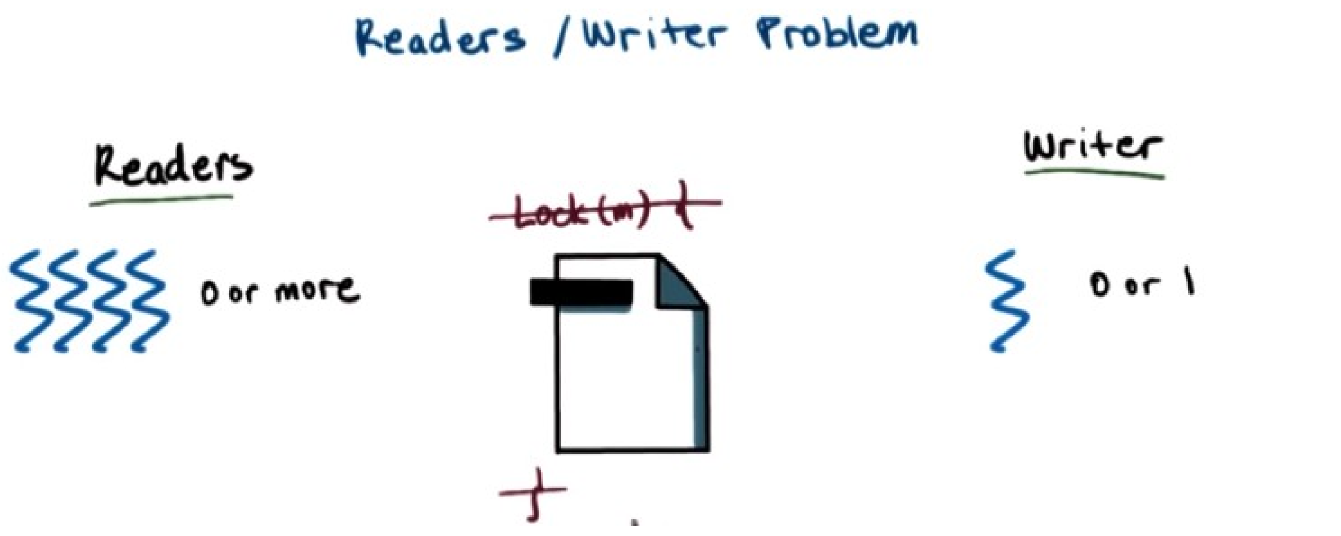
Instead of "while", why did we not simply use "if"?

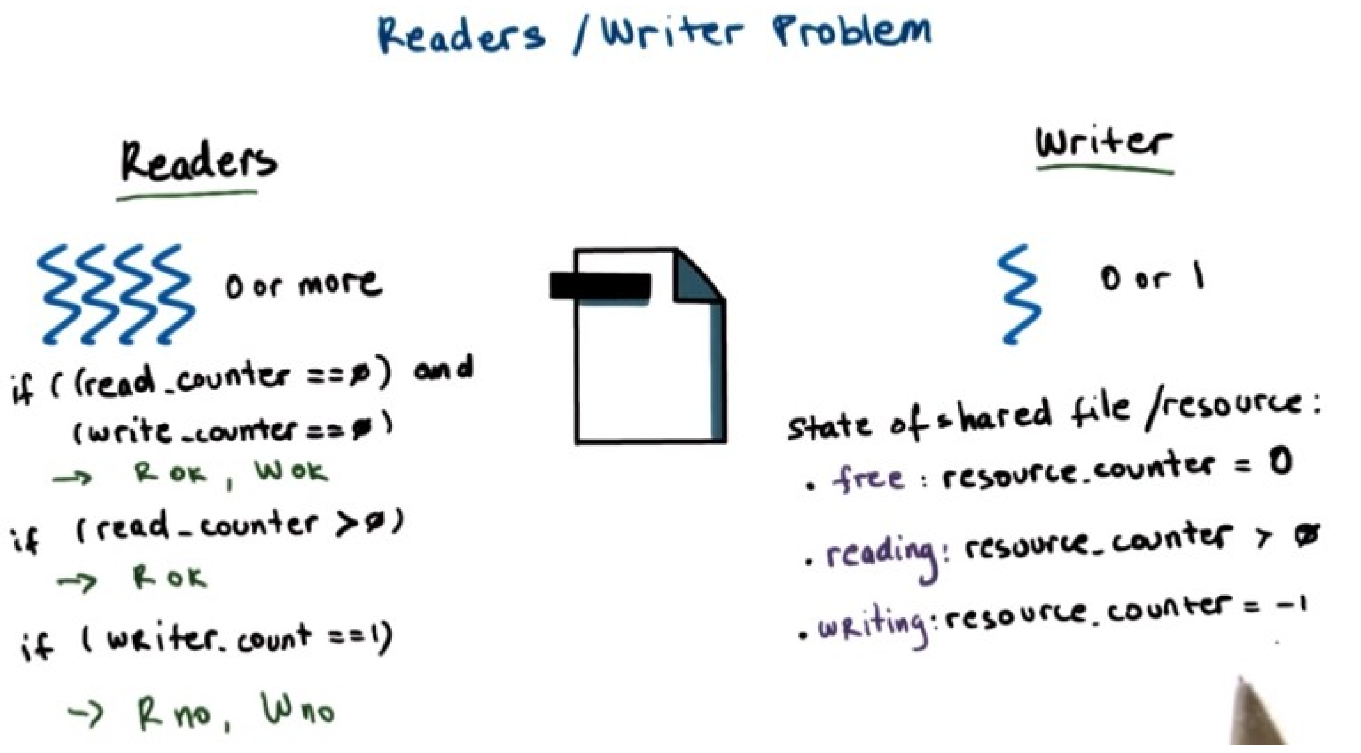
```
// consumer: print_and_clear
Lock(m) {
   while(my_list.not_full())
        Wait(m, list_full);
   my_list.print_and_remove_all();
} // unlock;
```

- ○ "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?]

(Piazza 上有人問此題)

21. 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 reacquire 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.

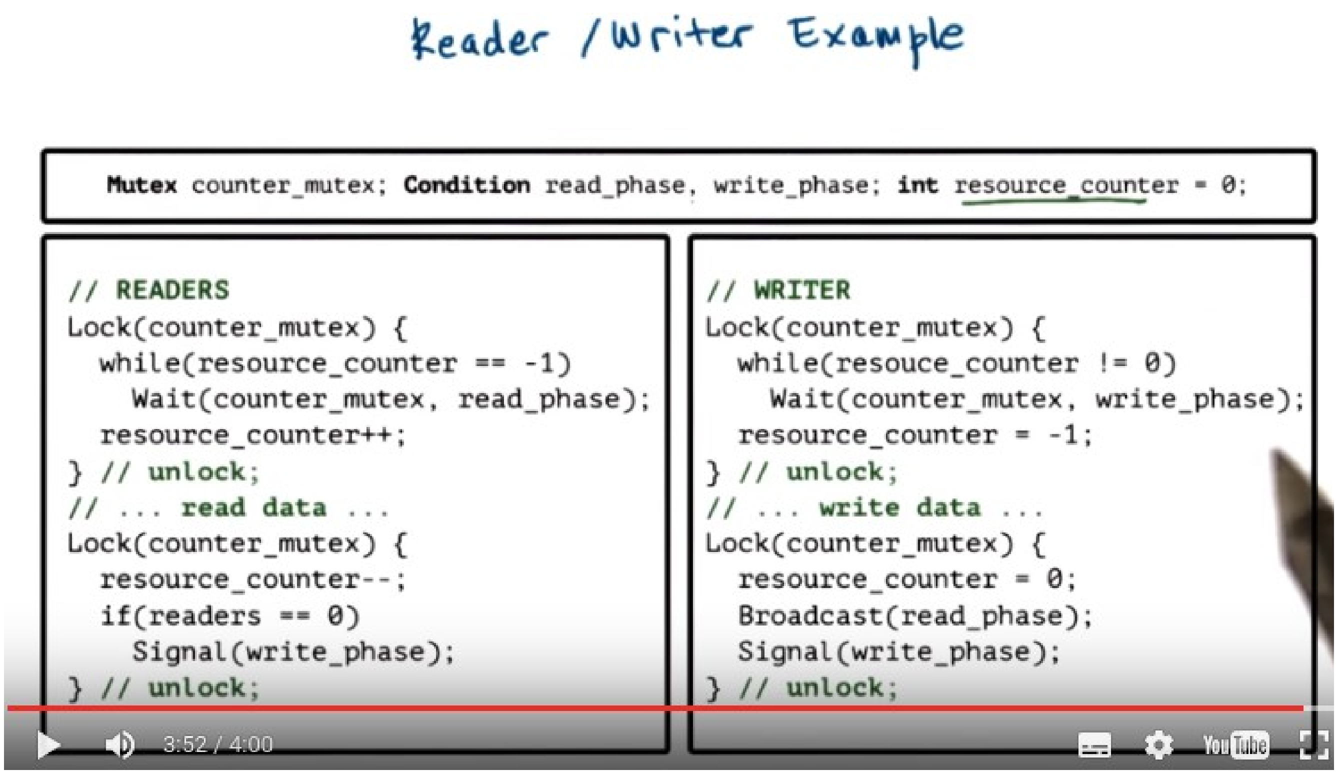



Readers / Writer Problem
Readers
0 or more
if ((read_counter == 0) and
(write_counter == 0)
→ R ok, W ok
if (read_counter > 0)
→ R ok
if (writer.count == 1)
→ R no, W no
Writer
0 or 1
State of shared file /resource:
• free : resource_counter = 0
• reading: resource_counter > 0
• writing: resource_counter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

日他大爺, 老子一晚上的 note(23-37 段)全被 LibreOffice corruption 給弄沒了, 害得老子重新弄.

22. Now let's see how mutexes and condition variables can be combined for use in a scenario that's common in multithreading systems, multithreading applications, as well as at the operating system level. And this is a scenario where there're multiple threads. But of two different kinds. Some subset of threads that want to perform read operation to access the shared state for reading. And other set of threads that want to perform write operations or that want to access that same shared variable, shared state, and modify it. We call these types of problems the readers/writer problem. The readers/writer problem is such that at any given point of time, zero or more of the reader's threads can access this

resource, but only zero or one writer threads can access the resource concurrently at the same time. And clearly we cannot have a situation in which a reader and a writer thread are accessing the shared resource at the same time. One naive approach to solving this problem would be to simply protect the entire resource, let's say this is a file, with a mutex. And put a lock and unlock operations around it. So whenever anyone of these types of threads is accessing the resource, they will lock the mutex, perform their respective access, and then release the mutex. This is, however, too restrictive for the readers/writer problem. Mutexes allow only one thread at a time to access the critical section. So they have a binary state, either zero and the resource is free and accessible, or one and the resource is locked and you have to wait. This is perfectly fine for the writers, since that's exactly the kind of semantic that we want to achieve. But for readers, with mutexes, we cannot express that multiple readers can be performing access to the shared resource at the same time. So how can we solve this problem? Well, we can start by trying to come up with the situations in which it is okay versus it is not okay to perform certain type of access to the resource. So we'll start enumerating these things and we'll try to express this based on the number of readers and writers that are also performing an operation. In the simple case when there are no readers and no writers accessing this resource, both a read operation can be granted, so an upcoming read operation, as well as a upcoming or pending write operation can be granted. As we describe the problem, if the read counter is greater than zero, so if there are more readers already accessing this file, it's perfectly fine to allow another reader to access the file concurrently. Since readers are not modifying it, so it's perfectly okay to grant that request. And then finally, if there is a writer that's performing an access to the resource, then we cannot allow neither the reads nor the writes. So, given these statements, we can express the state in which the shared file or the shared resource is as follows. If the resource is free, then some resource_counter variable, we can say well that will be zero in that case. If the resource is accessed for reading, then this resource_counter variable will be greater than zero. And so it will be actually equal to the number of readers that are currently reading this file. And let's say we can encode the fact that currently there is a writer, somebody's writing to this resource, by choosing that, in that case, the resource_counter variable should take the value negative 1. And of course, this will indicate that there is exactly one writer currently accessing the resource. So there is a saying in computer science that, all problems can be solved with one level of indirection. And here, basically, we produce a proxy resource, a helper variable or a helper expression. In this case, it is this resource_counter variable. This resource_counter variable reflects the state in which the current resource already is. But what we will do, as opposed to controlling the updates to the file, so controlling, with a mutex, who gets to access the file, we will control who gets to update this proxy variable, this proxy expression. So as long as any access to the file is first reflected via an update to this proxy expression, we can use a mutex to control how this counter is accessed. And in that way, basically monitor, control and coordinate among the different accesses that we want to allow to the actual shared file.


Reader / Writer Example

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

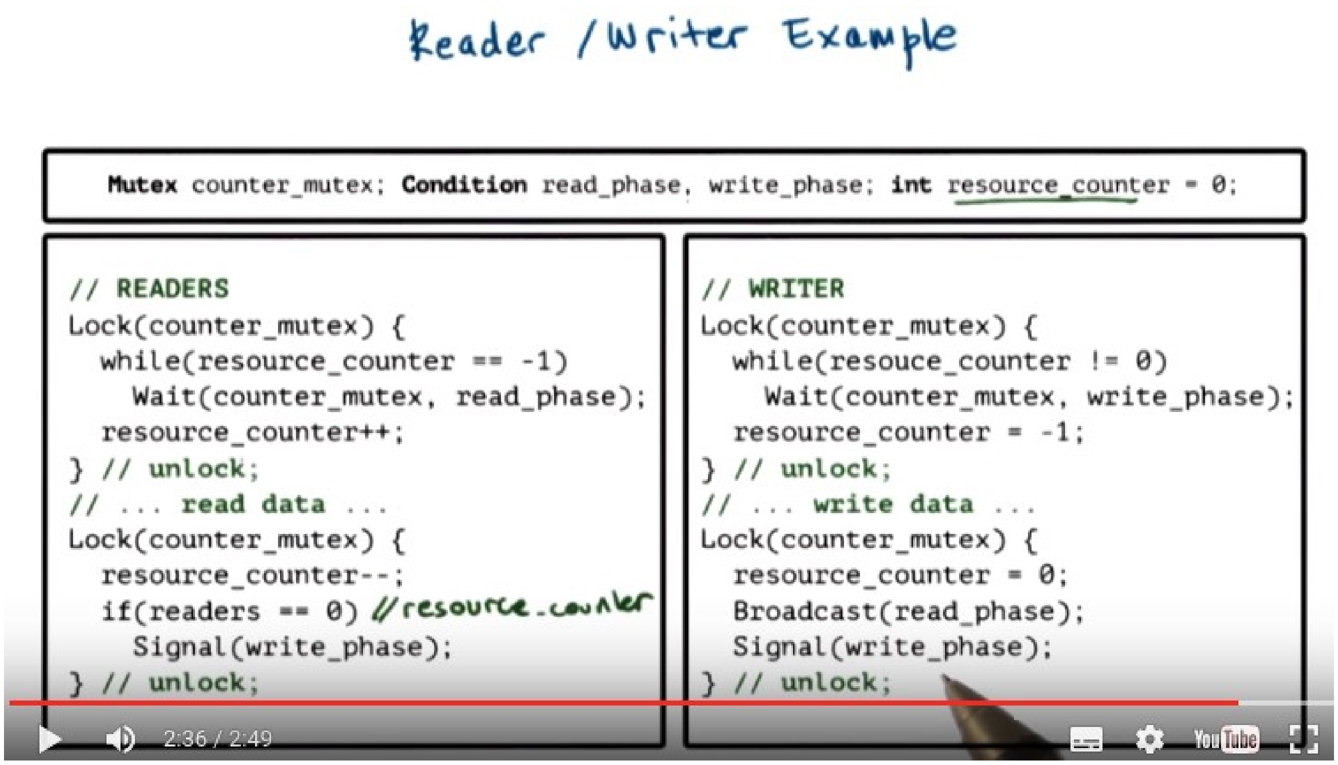
```
// READERS
Lock(counter_mutex) {
  while(resource_counter == -1)
    Wait(counter_mutex, read_phase);
  resource_counter++;
} // unlock;
// ... read data ...
Lock(counter_mutex) {
  resource_counter--;
  if(readers == 0)
    Signal(write_phase);
} // unlock;
```

```
// 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;
```

上圖左邊的 if(readers == 0)中的 readers 應該就是 resource_counter.

23. So let's explain all of this with an actual reader writer example. The actual shared file is accessed via these read data and write data operations. 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 of 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. And then we make sure that in our code, on both the readers and the writer side, before we perform the read operation, we first perform a controlled operation in which the resource counter is updated. Similarly on the writer side. Before we write, we have to make sure that first the resource counter is set to negative 1. Now that, the changes to this proxy variable, that will be what will be controlled, that will be what will be protected within the lock operations. Once we are done with the actual shared resource access, with reading the file or writing the file, we again go into 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 that resource. So it will need, basically in our, in our program, we will need a mutex. That, that's the counter mutex. This is the mutex that will always have to be acquired whenever the resource counter variable is accessed. And then we will also need two variables, read_phase and write_phase, which will tell us whether the readers or the writers need to go next. So lets explain this. Let's see what happens when the very first reader tries to access the file. The reader will come in, it will log the mutex, it will check what the resource counter value is, and it will be zeroed. That's what the resource counter was initialized at. So it is not negative 1, perfect. We continue to the next operation, we increment resource_counter. Resource_counter will now have a value of 1. And then we unlock the mutex and we proceed accessing the file. (如果此時) A subsequent reader comes in, and let's say, while it (指第一個 reader) executing this operation, before it (指第一個 reader) came to the unlock statement, the next reader comes in. The next reader, when it comes in, it will see that it can not proceed with that lock operation. So it will be blocked on the lock operation (當第一個 reader 在 read file 時, lock 就被

release 了, 第二個 reader 就可以进入 lock 語句, 修改 resource_counter 的值, 然後出 lock 語句, 然後 read file). So this way, basically, we are protecting how resource_counter gets updated. So only one thread at a time both on the readers and the writer side will be able to update to access this resource counter. However, let's say 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 reading the file. Now let's say a writer thread now comes in. So the writer locks the mutex, let's say 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 it will be, let's say, two. So clearly the writer has to wait. Now, it will wait, in the wait operation it specifies the counter mutex, and it says it's going to wait for the write phase. What will happen at this point, remember, we are performing this wait operation within the lock construct, so the writer has the counter mutex, it's the owner of the counter mutex. However when it enters the wait operation, the mutex is automatically released. So the writer is somewhere suspended on a queue that's associated with the write_phase condition variable, and the mutex is at this point free.



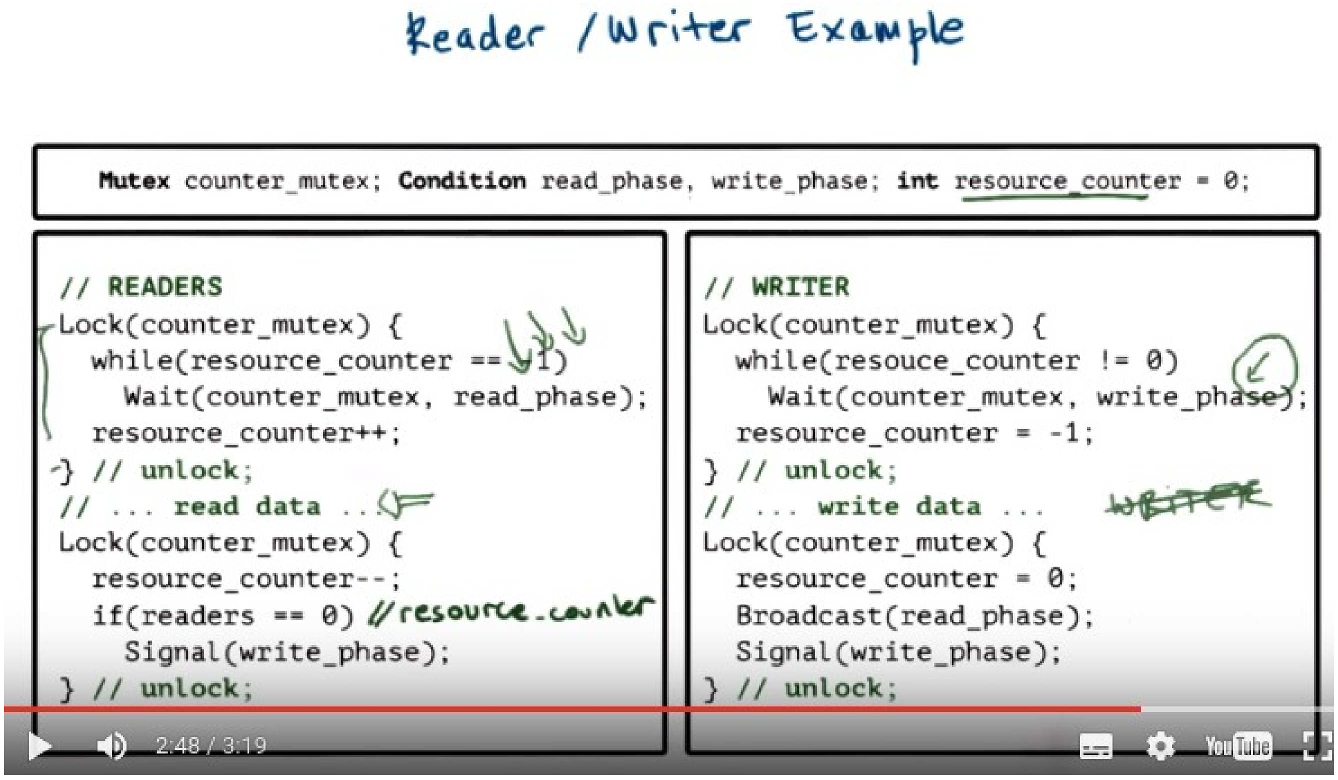
```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

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

24. So let's say the readers, our readers start finishing the accesses. So as they finish the access they will first lock the counter_mutex and this is why it was important that the writer release the mutex, because otherwise none of the readers would have been able to basically exit the real critical section, the reading of the files. So to perform these updates to the proxy variable, and to reflect the fact that nobody's currently reading the file. So reader exists the read phase. So it will lock the mutex. It will decrement the counter. And it will check the value of the resource_counter. So, once the reader decrements resource_counter, it checks to see whether it's the last reader. So, this really should be it checks whether resource_counter is, has reached the value 0. The very last reader that is exiting this read phase will see that resource_counter has reached 0, it's the last reader. And then it will signal the

write_phase condition variable. It makes sense to generate this signal and to notify a potential writer that currently there are no readers performing read operations. And given that only one writer can go at a time, it really doesn't make sense to use a broadcast. So, this write_phase will be received ultimately over here. And the one writer that was waiting on that write_phase condition of the area above will be woken up. What will happen internally, that writer will be removed from the wait queue that's associated with the write_phase condition variable. And the counter mutex will be reacquired before we come out of the wait operation. Now as we explained earlier, the very first thing that we have to do is, we have to go ahead and check the statement resource_counter one more time. So we, we go out of the wait statement, but we're still in the while loop. We have to check whether resource_counter is indeed still zero. The reason for this is that internally in the implementation of this wait operation, removing the thread from the queue that's associated with the write_phase condition variable, and acquiring this counter_mutex mutex, are two different operations. And in between them, it is possible that another, either another writer or another reader has basically beat the writer that was waiting to the mutex. And so when we come out, yes we have the mutex, but somebody else has already acquired it, changed the resource_counter value to reflect the phase that maybe there is a reader or a writer currently accessing the file, and then released the mutex. But it, actually, there is another thread that's currently in one of these code blocks that we wanted to protect in the first place.


Reader / Writer Example

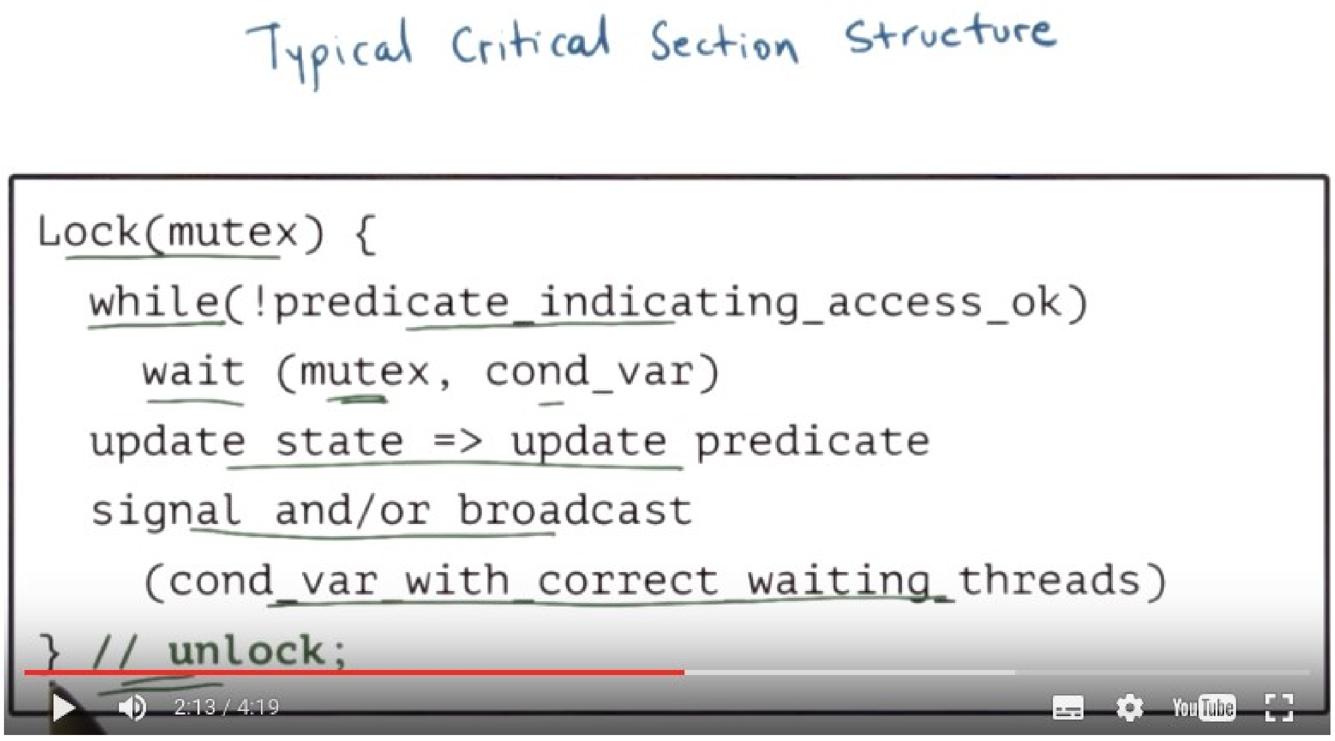
```
Mutex counter_mutex; Condition read_phase, write_phase; int resource_counter = 0;
```

```
// 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;
```

25. So, while the writer is executing in this write_phase, let's say we have another writer that came in. So this writer is now pending on the read_phase (應該是 write_phase, 見下) variable. So, we currently have basically one writer here, one writer pending on this write_phase. And let's say we have another writer (應該是 reader), reader that came in and it's pending on this read_phase variable. So we have two threads that are waiting on two different variables, and then a writer that's actually in the critical section. When this writer completes, it starts exiting the critical section. So it will reset the resource_counter value, doesn't make sense to decrement, only one writer at a time could, could be in

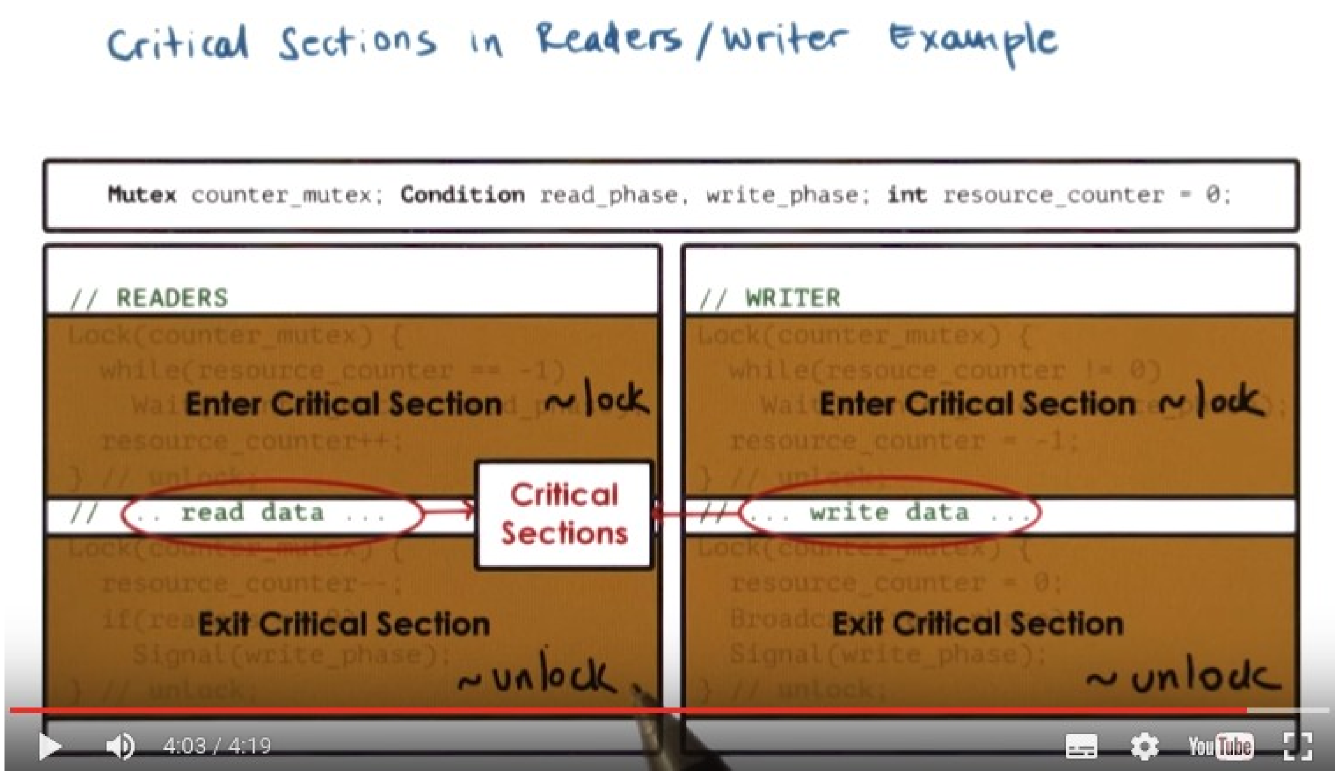
there so it's either negative 1 or 0. And then here in this code, we did two things. We broadcast to the read_phase condition variable. And we'll signal to the write_phase condition variable. We signal to the write_phase condition variable again because only one thread at a time is able to proceed with our write operation. We broadcast to those threads that are reading on the read_phase. So potentially multiple threads will be woken up. Because it makes sense, we allow multiple readers, we allow multiple threads to be in a read_phase, it makes sense to use the broadcast. So, let's say that we have multiple readers waiting on the read_phase when we issued this broadcast. Now, this phase here requires a mutex. So, when these threads that were waiting on the read_phase are waking up from the wait, they will one at a time acquire the mutex, check is resource_counter negative 1? No, it is not negative 1, right? We just reset it to be 0. So, I'll increment the counter. So, incrementing the counter, when the first thread wakes up, the first, the resource_counter will be 1. And then the first thread of the waiting ones will release the mutex and start reading the data. The remaining threads that were waiting on the read_phase will also one at a time come out of wait statement, check to see whether resource_counter is negative 1, and now it will have some positive value. And so, they too will increment the resource_counter and will come out. So, the waiting threads which were woken up from the broadcast statement will be coming out of this wait operation one at a time, but ultimately, we will have multiple threads in the read_phase at the same time. So, this is why broadcast is useful. Yes, indeed, only one thread at a time can really be woken, can really execute this piece of code. But we do want multiple threads to be woken up so that multiple threads can ultimately reach this read_phase once this writer completes. The other thing that's worth noting is here we use both broadcast and signal. Whether the reader's will be really the first ones to execute, or the writers, that we really don't have control over. It really depends on how the scheduler ends up scheduling these various operations. So, the fact that we first called broadcast versus signal, this is really just implicit that the readers are given some kind of priority over the writers. We have no control over that.


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;
```

26. Looking in the code from the previous morsel, it may seem complex, but you will soon become experts at turning on these critical sections. To make things more simple, let's generalize the structure of each of the code blocks corresponding to the entry points and the exit points in the actual, the real

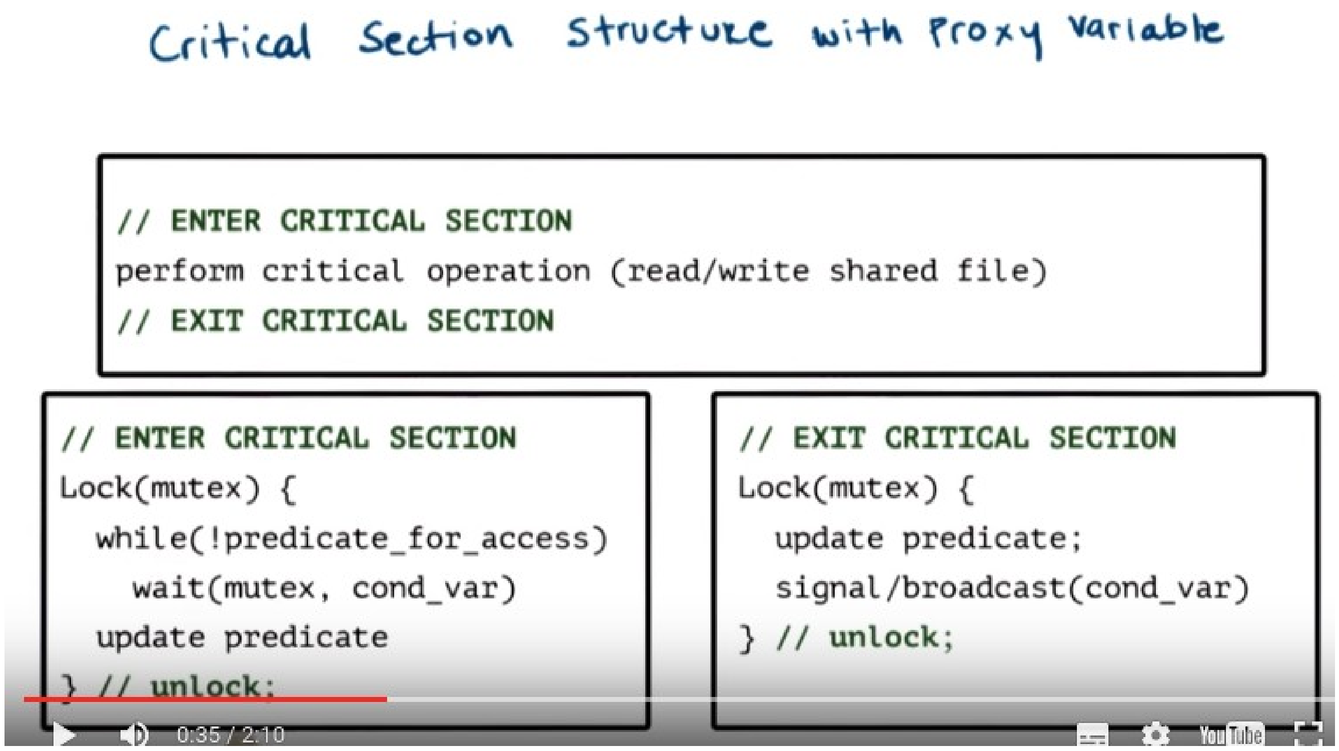
critical section code that the readers and writers are performing. So these code segments here correspond to the enter critical section and the exit critical section codes. When we consider read data as the operation that the real critical section, given that what we really wanted to protect was the file. So this is what we want to control, and then we structure these code blocks to, to limit the entry phase into this code and then the exit phase. Internally, each of these clearly, it represents a critical section code as well. Since there is a lock operation and then there is a shared resource. The resource counter that is updated in both of these cases. If we closely examine each of the blocks that we highlighted in the previous example. We will see that they have the form as follows. First we must lock the mutex. Then we have to check on a predicate to determine whether it's okay to actually perform the access. If the predicate is not met, we must wait. The wait itself is associated with the condition variable and a mutex, and this is the exact same mutex that we locked. When we come out of the wait statement, we have to go back to check the predicate. So we have to have the while loop. The wait statement has to be in a while loop. When we ultimately come out of the while loop, because the predicate has been made. We potentially perform some type of update to the shared state. These updates potentially impact something about the predicates that other threads are waiting on, so we should need to notify them by basically notifying the appropriate condition variables. We do that via a signal and a broadcast. And ultimately we must unlock the mutex so here the lock construct we set in Burrell's paper, the unlock operation is implicit. In some threading environments there is an explicit unlock api that, that we must call. Returning to the readers writers example the real critical sections in this case, are the read operation and the write operation, right? These are the operations that we want to control. And we want to protect, the very least we want to make sure that there isn't a situation where there is a concurrently a write operation where, while others are reading. As well as that there are no concurrent writes. So the code blocks that precede and follow each of these critical sections both on the reader's side that we outlined before and the corresponding ones on the writer's side. These we call the Enter Critical Section blocks and the Exit Critical Section blocks. Every single one of these blocks uses the same mutex, counter_mutex. And so only one thread at a time will be able to execute within these code blocks, except that in these code blocks we only manipulate the resource_counter variable. So only one one thread at a time can manipulate the resource_counter variable. However potentially multiple threads at a time can be performing concurrently a read file operation. So it looks like these enter critical section blocks are sort of like a lock operation that we have to perform before we want to access a resource that we want to protect. That we want to control what kinds of accesses to that shared resource we allow. And then similarly when we are done with manipulating that shared resource, these exit critical sections are sort of for like are, unlock. In the plain mutex case, we have to unlock a mutex in order to allow some other thread to proceed, otherwise the threads will be blocked indefinitely. Same here, when we finish this critical section we have to execute this portion of the code in order to allow other threads, writers in this case, to actually go ahead and gain access to the file.


Critical Sections in Readers/Writer Example
Enter Critical Section ~lock
Critical Sections
Exit Critical Section ~unlock
4:03 / 4:19

```
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;
```



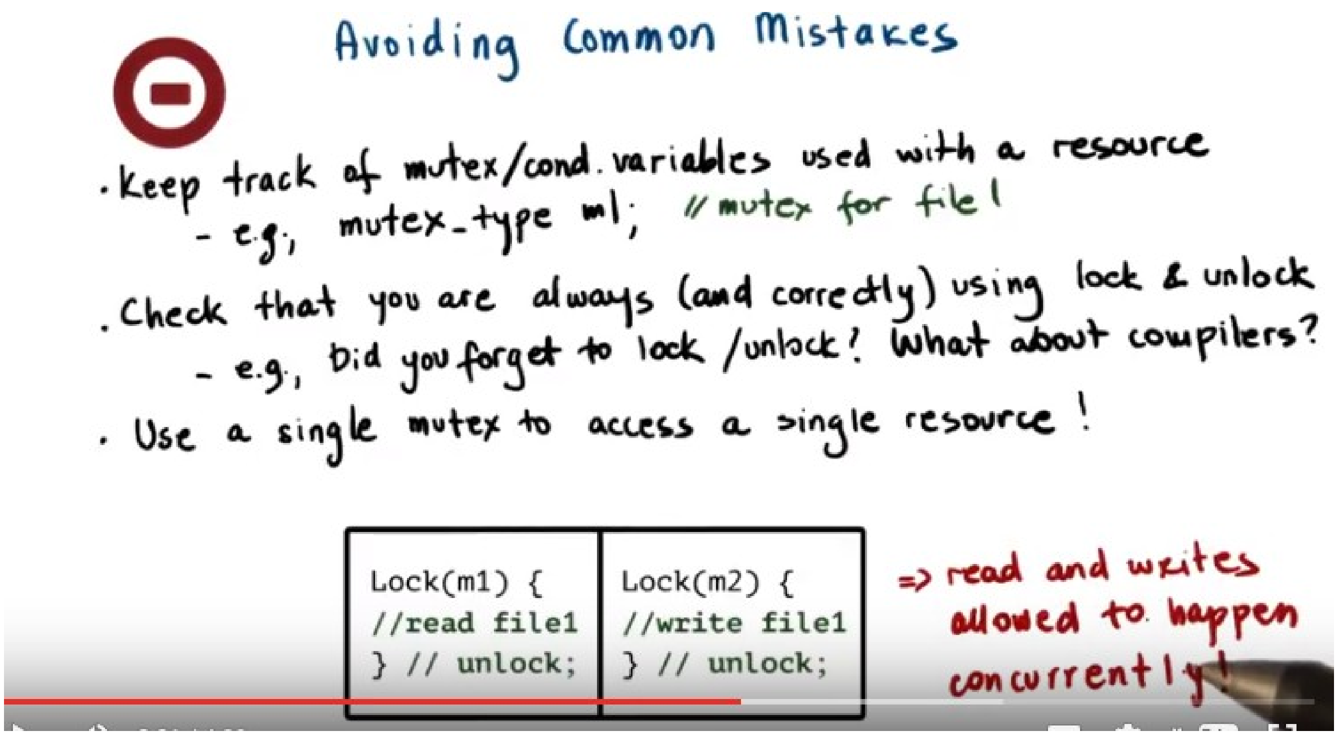
```
// ENTER CRITICAL SECTION
perform critical operation (read/write shared file)
// EXIT CRITICAL SECTION
```

```
// 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;
```

27. So for examples like the reader writer example we have basically this common building blocks. The actual operations to the shared resource, the shared file, so the reads and writes of the shared file in this case, have to be protected with these code blocks enter critical section and exit critical section. Each of

these code blocks internally follows basically the critical section structure that we outlined just couple of screens before, where they lock a mutex, they check for a predicate. If the predicate isn't met, they wait. The wait is in a while loop, and if the while is okay, a predicate potentially gets updated. When we're done with the actual operation, the exit critical section code. Again, in one of these lock constructs, we have to update the predicate, so up here we really have nothing to wait on. We just want to update the predicate and then signal and broadcast the appropriate condition variable. The mutex is actually held online within these enter critical section and exit critical section codes. So if you see it is unlocked on the end and this allows us to then basically control the access to the proxy variable, but to allow more than one thread to be in the critical section at a given point of time. This lets us benefit from mutexes, because mutexes allow us to control how a shared resource is accessed. However, this type of structure also allows us to deal with a limitation that mutexes present because they allow only one thread at a time to access that resource, so with this structure, we will be able to implement more complex sharing scenarios. So in this case the policies that either multiple threads of type reader can access the file, or one thread of type writer can access the file. The default behavior of mutexes alone doesn't allow us to directly enforce this kind of policy. Mutexes only allowed mutual exclusion policy.



Avoiding Common Mistakes

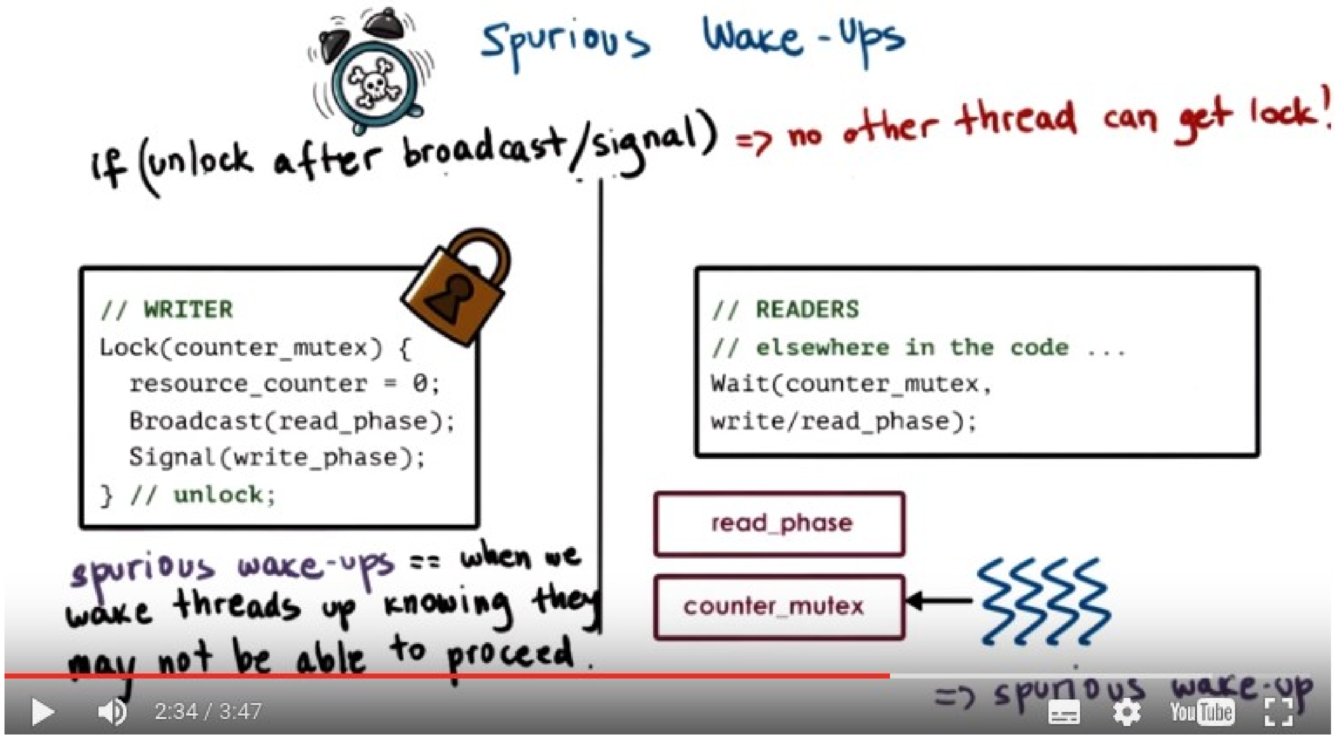
- Keep track of mutex/cond. variables used with a resource
  - e.g., mutex_type m1; // mutex for file 1
- Check that you are always (and correctly) using lock & unlock
  - e.g., Did you forget to lock/unlock? What about compilers?
- Use a single mutex to access a single resource!

Lock(m1) { //read file1 } // unlock;	Lock(m2) { //write file1 } // unlock;

=> read and writes allowed to happen concurrently!

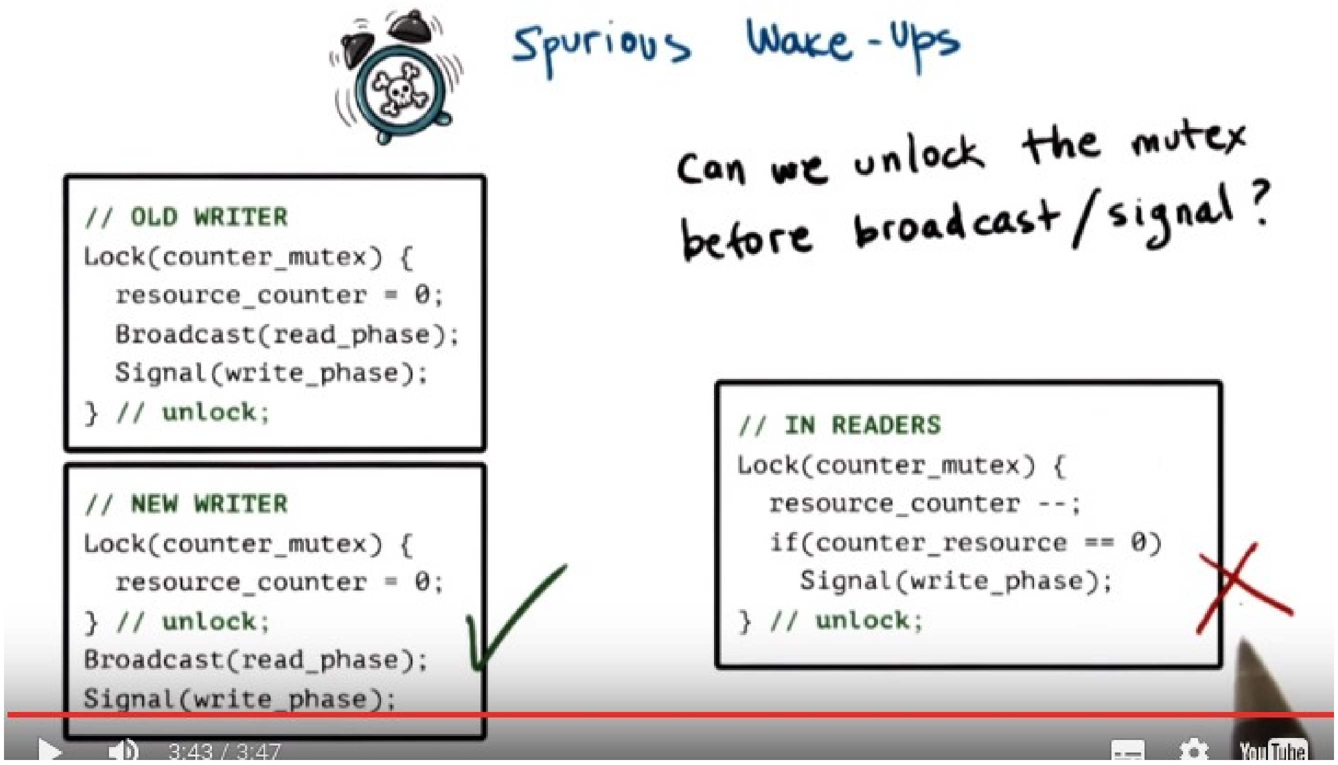
28. Now let's look at some frequent problems that come up when writing multi-threaded applications. First, make sure to keep track of the mutex and condition variables that are specifically used with a given shared resource. What that means, for instance, is that when defining these variables make sure 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.


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
2:34 / 3:47

```
// 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);
```



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

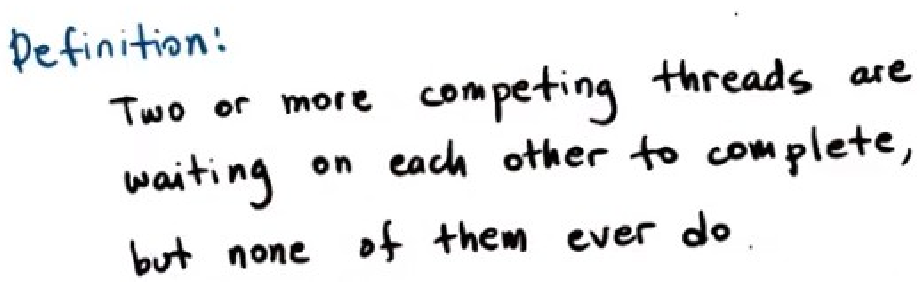
```
// NEW WRITER
Lock(counter_mutex) {
  resource_counter = 0;
} // unlock;
Broadcast(read_phase);
Signal(write_phase);
```

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

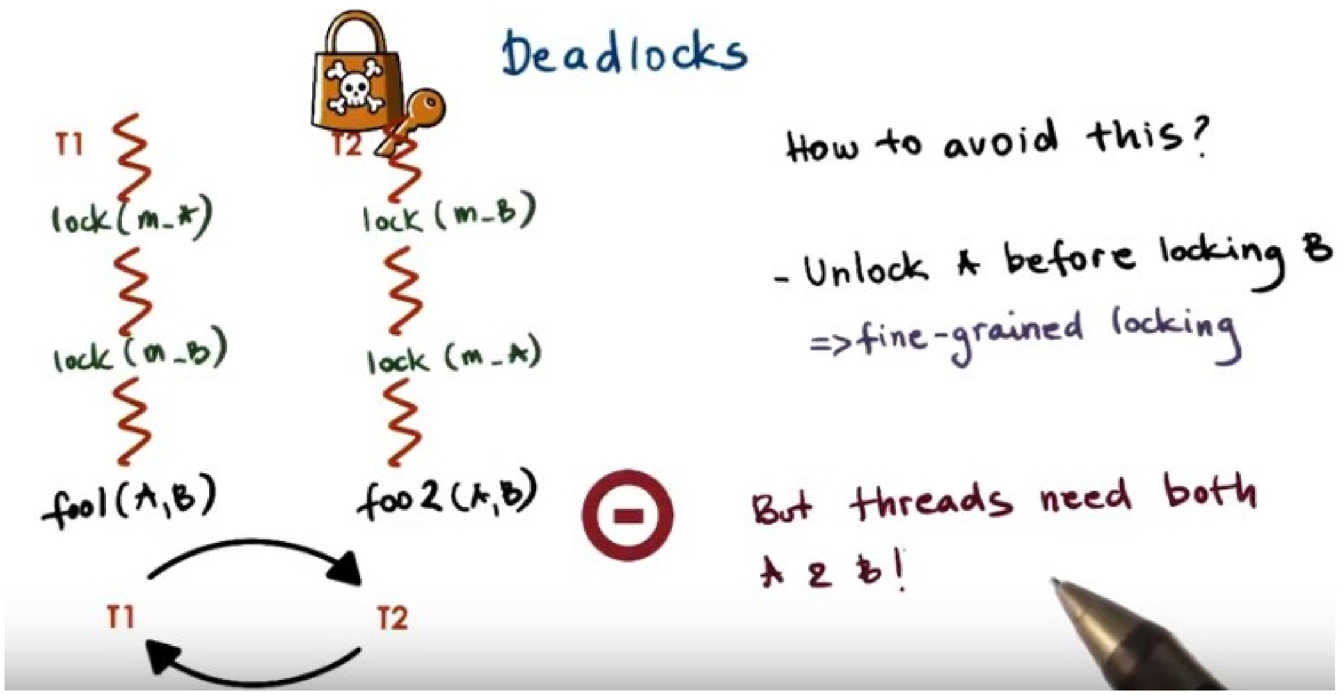
上圖右邊的 counter_resource 即 resource_counter.

29. 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 reacquire 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, the 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. So we cannot unlock and then continue accessing the shared resource (因為右邊的代碼中已先將 resource_counter--了, 所以它不可能為0). That will affect the correctness of the program. Therefore, this technique of unlocking before we perform the broadcast or signal doesn't work in this particular case or in similar cases.


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

30. One of the scariest problems related to multithreading is deadlocks. An informal definition of a deadlock is that, it is a situation in which competing threads, at least two or more, they're each waiting on each other to complete. However, none of them ever do because each waits on the other one. So thus, the execution of the process overall of all of these threads is stuck and it cannot continue. The threads are, we call, deadlocked. We can use the visual example to help explain a deadlock using our toy shop example. So, imagine that two workers in the toy shop are finishing toy orders that involve a train, and each worker will need a soldering iron and a solder wire to finish their toy. The problem is, there's only one of each of those. So, let's say the first worker grabs the soldering iron first, and the second worker grabs the solder wire. And because both workers are stubborn, they're unwilling to give up either one of the items that they've grabbed, so none of the toys will ever get made. The workers remain continuously stuck in a deadlock. In practice, the deadlock example can be described with the following situation.



Deadlocks

T1 lock(m_A) lock(m_B) foo1(A,B)

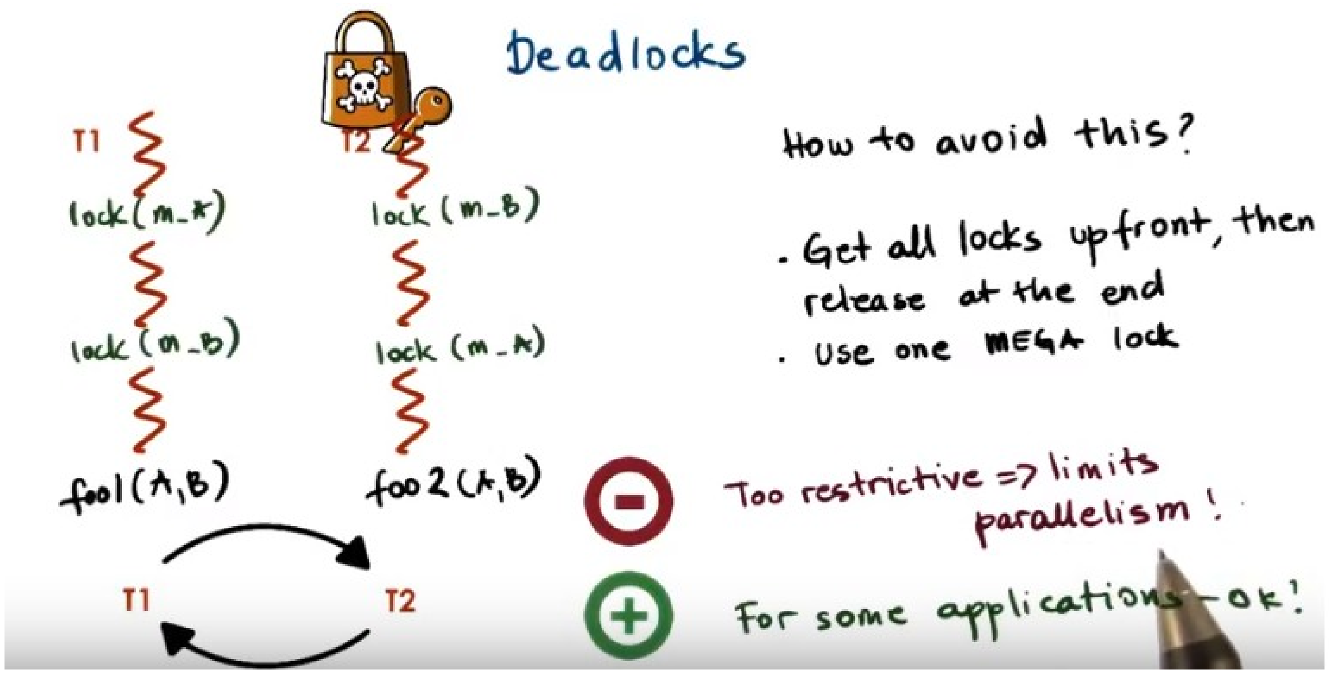
T2 lock(m_B) lock(m_A) foo2(A,B)

T1 T2

How to avoid this?

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

But threads need both A & B!



Deadlocks

T1 lock(m_A) lock(m_B) foo1(A,B)

T2 lock(m_B) lock(m_A) foo2(A,B)

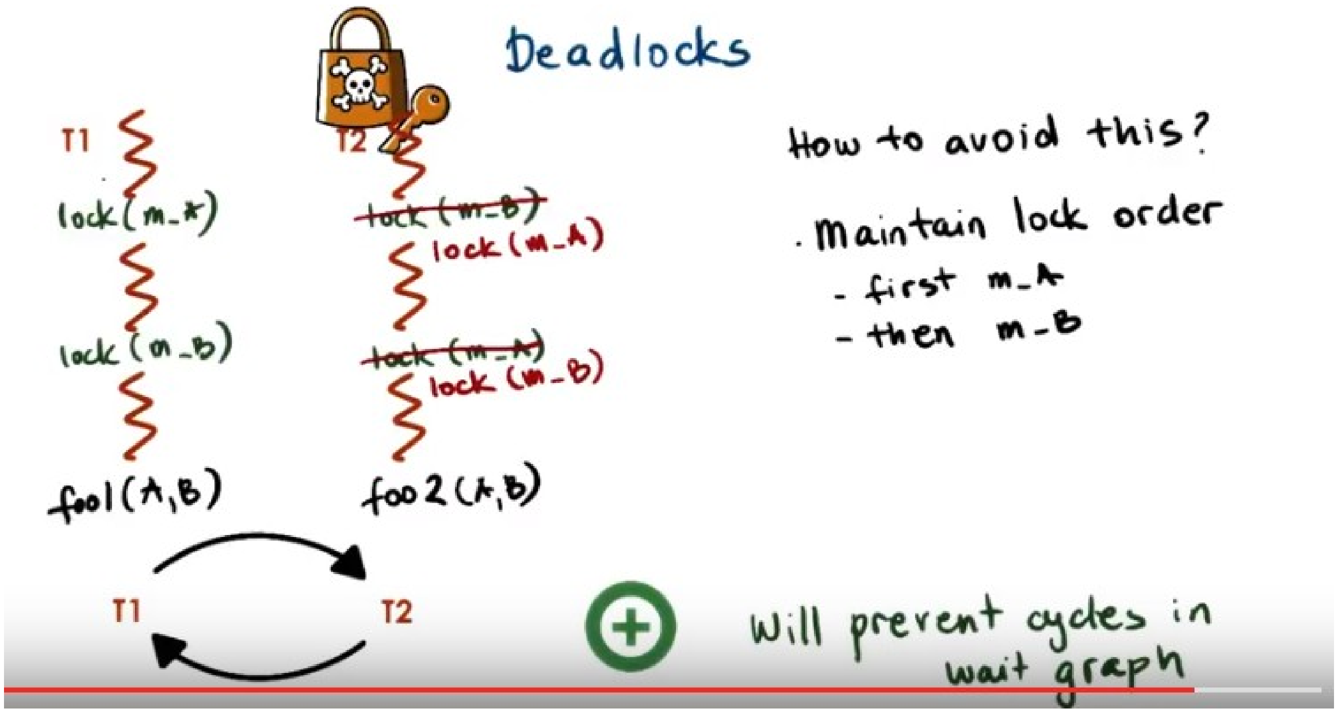
T1 T2

How to avoid this?

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

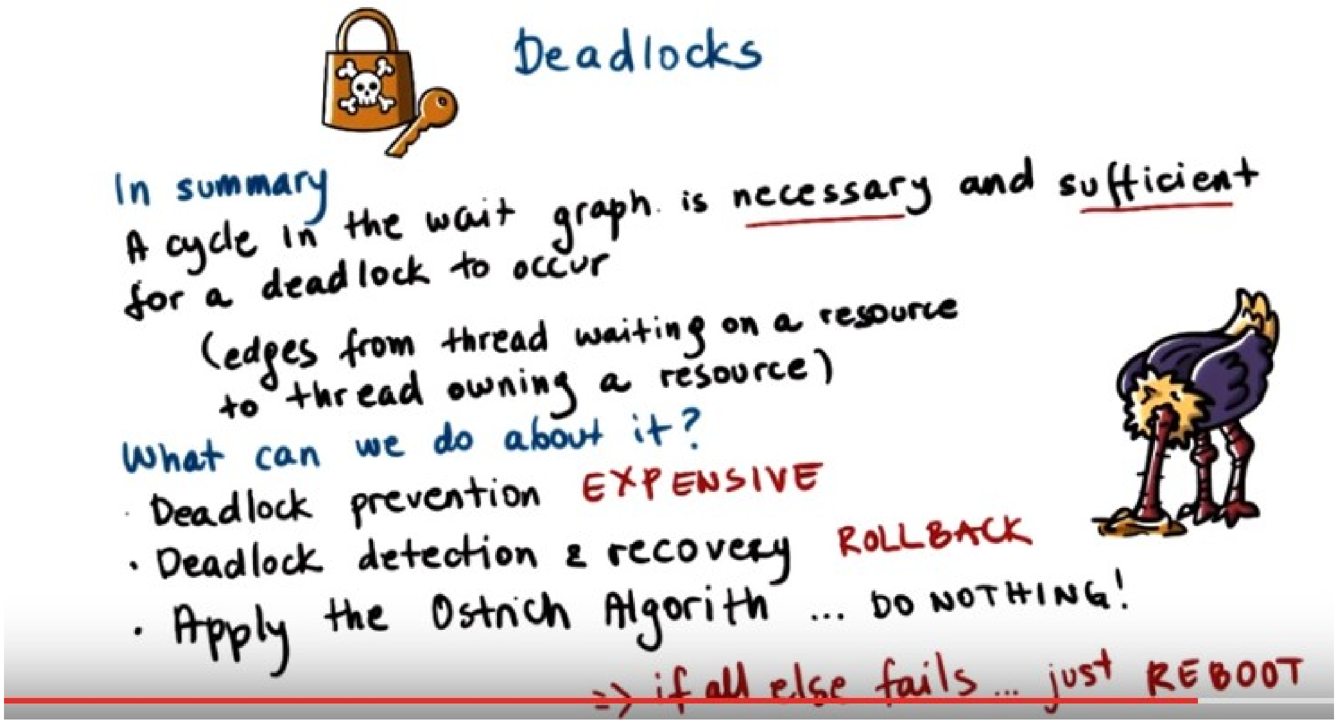
Too restrictive => limits parallelism!

For some applications - OK!



31. So we have two threads, T1 and T2. They need to perform some operation involving some variables, A and B. And, in fact, the two threads don't even need to perform the same operation on A and B. But let's say they both need to access these shared variables, A and B. Before performing these operations, they must lock mutexes that protect the shared variables A and B. And let's say T1 first locks the mutex for A and then locks the mutex for B. And in the case of T2, T2 first locks the mutex for B and then locks the mutex for A. And this is where the problem is. The two threads are waiting on each other in a cycle. Neither one of them will be able to get to the foo operation. Neither one of them, in fact, will be able to execute this second lock operation. They'll keep waiting on each other. And we'll have a dead lock. So how can we avoid these situations? One way to avoid this situation would be to unlock the mutex for A before locking the mutex for B, or the other way around. We would call this fine-grained locking. The problem with this is that, it won't work in this case since the threads need both A and B. So, they have to have both locks. They need both variables, after all. Another possibility would be to get all the locks up front and then release them at the very end. Or maybe we just end up using one mega-lock, some mutex m sub A,B. This solution may work for some applications. However, in other cases it can be very restrictive because it limits the level of parallelism that can exist in the system, and the last and the really most accepted solution is to maintain a lock order. If we force everyone to first get the mutex for A and then the lock for B, the problem will not occur. If we investigate this a little bit more, we see that the problem is that T1 is waiting on something that T2 has. And T2's waiting on something that T1 has. So we have a cycle basically. And in this case, we have two threads, so it's easy to reason about the situation and to determine that the order in which these locks are being acquired can result potentially in a cycle. In principle, this type of analysis is a little bit more difficult to illustrate than to determine, but what we're trying to really show is that if there is a cycle in this kind of wave graph in which we draw a line between two threads. If one thread is waiting on a resource that the other thread has, then a cycle indicates a dead lock. Maintaining a lock order will prevent such cycles from occurring. So will ensure that there will be no deadlocks in the

code. So to enforce this kind of maintain lock order principle in the example before, T2 would have to get the mutexes, the mutex for A first and then the mutex for B, and there no way that in that, in this code a cycle would occur. There's no way that a deadlock can happen. So, consider this. If thread 1 is waiting on the lock, is about to execute the operation lock m of B, it means it already has acquire the mutex m of A. If thread 1 already has this mutex then thread two cannot have it. So, thread two must be somewhere before this lock operation and its execution (由此可知, 若一個 thread 沒有 acquire lock, 則它會被堵在 lock 語句之前). That also means that thread 2 could not have acquired the m_b mutex, it doesn't have it, and therefore there's no cycle. Thread 1 is not going to end up in a situation in which it has to wait on thread 2 for that particular lock. So it will be able to acquire it and continue. So this type of technique will always work. The only potential challenge is that in complex programs that use many shared variables and potentially many synchronization variables, so lots of mutexes, take some effort to make sure that these are indeed ordered. And everywhere used in the same order. But as a technique this is foolproof. And it guarantees that it will prevent deadlocks from happening.



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?

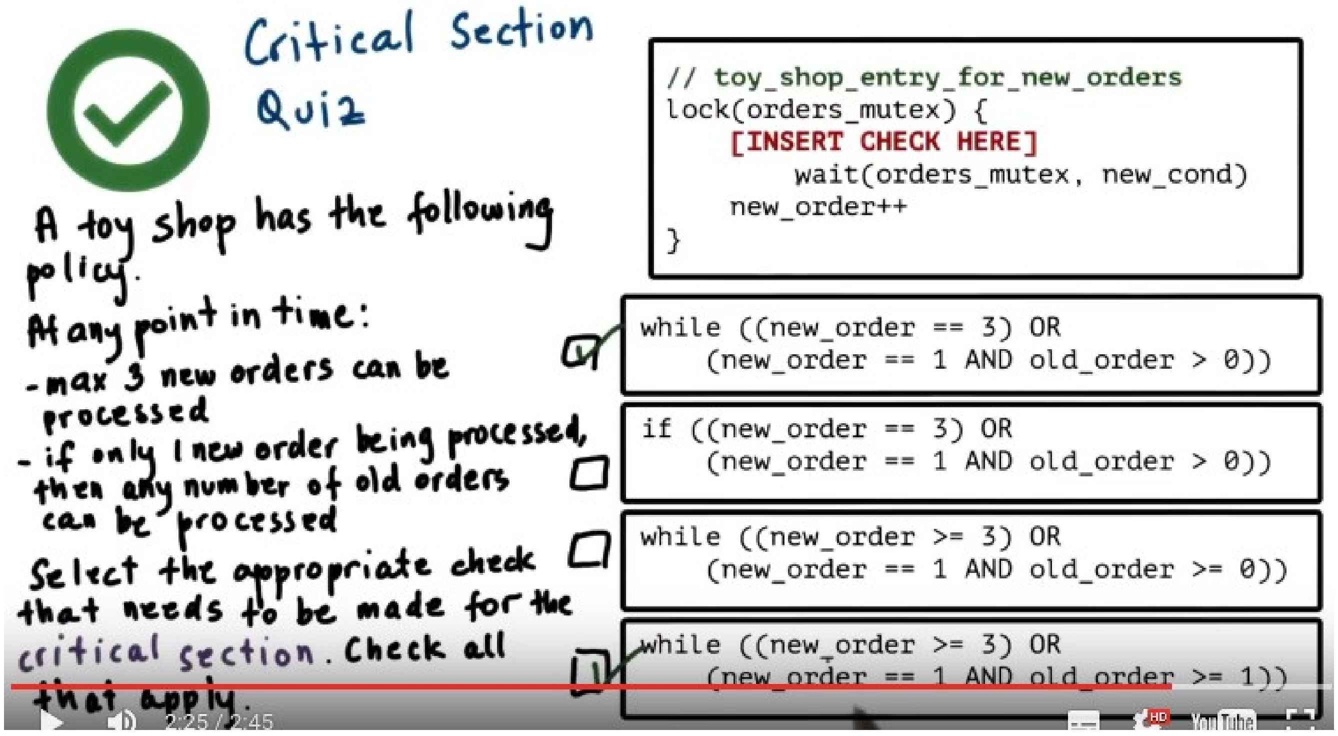
· Deadlock prevention EXPENSIVE
· Deadlock detection & recovery ROLLBACK
· Apply the Ostrich Algorith ... DO NOTHING!

=> if all else fails ... just REBOOT

32. There is more that goes into dealing with deadlocks. Detecting them, avoiding them, and recovering from them. But for the sake of this class, remember that maintaining order of the deadlocks will give you a deadlock-proof solution. So in summary, a cycle in the wait graph is necessary and sufficient for a deadlock to occur. And this graph itself is one where the edges are from the thread that's waiting on a resource to the thread that owns a resource. So what can we do about this? First we can try to prevent deadlocks. Every single time a thread is about to issue a request for a lock, we first have to see if that operation will cause a cycle in this graph. If that is the case, then we must somehow delay that operation. And in fact, we may even need to change our code so that the thread first releases some resource and only afterwards it attempts to perform that lock request. Clearly this can be very expensive. The alternative of completely preventing deadlocks is to detect deadlocks, and if you detect that they have occurred, to have mechanisms to recover from them. We do these kinds of things based on, basically, analysis of the graph to determine whether at any given point of time, any cycles have

occurred. This maybe isn't as bad as monitoring and analyzing every single lock request to see whether it will cause future deadlocks. But it's still expensive in that it requires us to have ability to roll back the execution so that we can recover. And the only way that that can be possible is if we have maintained enough state during the execution so that we can figure out how to recover. And in some cases, these rollback mechanisms are essentially impossible to perform. If we have inputs and outputs that came from external sources, we don't have ways to roll back their execution. And finally we have the option to apply the Ostrich Algorithm. And as sophisticated as that sounds, the Ostrich Algorithm is that, will basically just hide like an ostrich with his head in the sand, and do nothing. This is even more optimistic than this rollback based mechanism, in which we were letting the system execute, allowing it to cause a deadlock, and then recovering from him. Here we're essentially just hoping the system will never deadlock, and if we're wrong, then we'll just reboot. The truth is that although we know that lock order will help remove any cycles in the graph, when we have complex codes, when we have codes from different sources, it's really hard to guarantee that all of the mutexes will be acquired in exactly the same order across multiple source files. And so deadlocks are a reality. These types of techniques are possible, exist, and they can help us deal with deadlocks, however they're rather expensive in terms of the overheads they impose on the execution time of the system. So typically they're applied only in really performance-critical systems.

33. 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.



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++
}
```

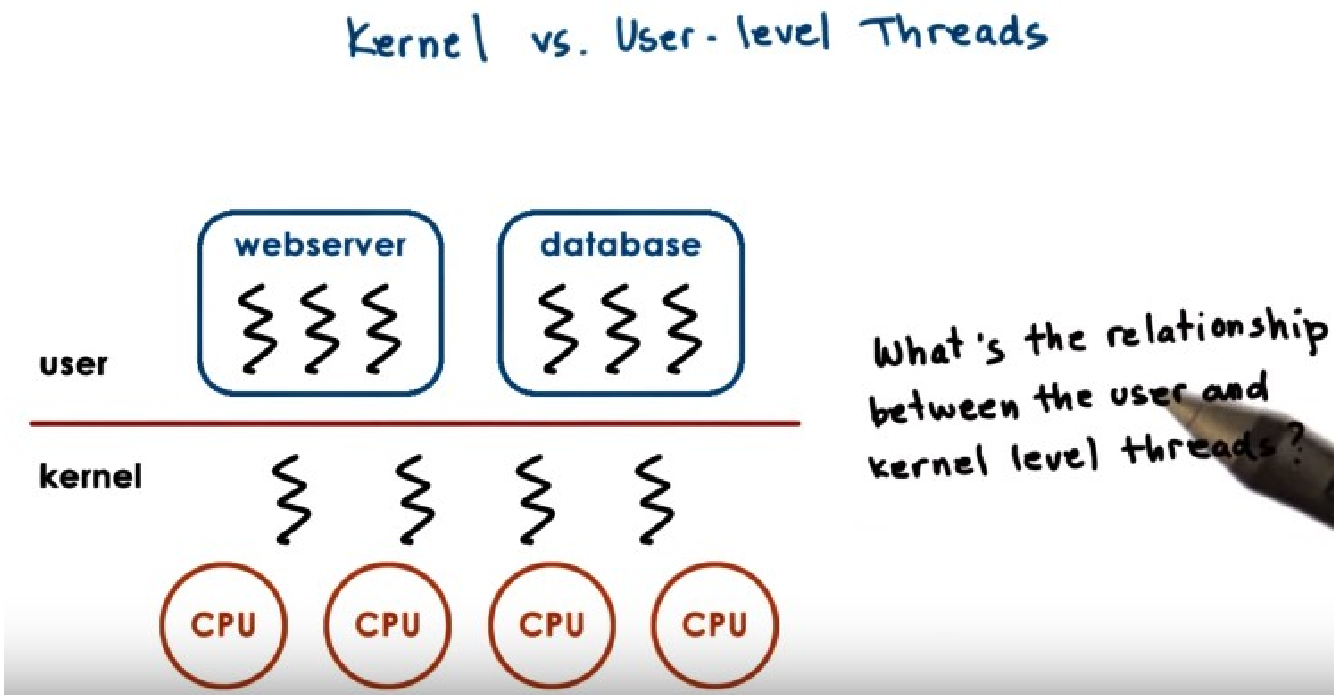
[✓]
```
while ((new_order == 3) OR
    (new_order == 1 AND old_order > 0))
```

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

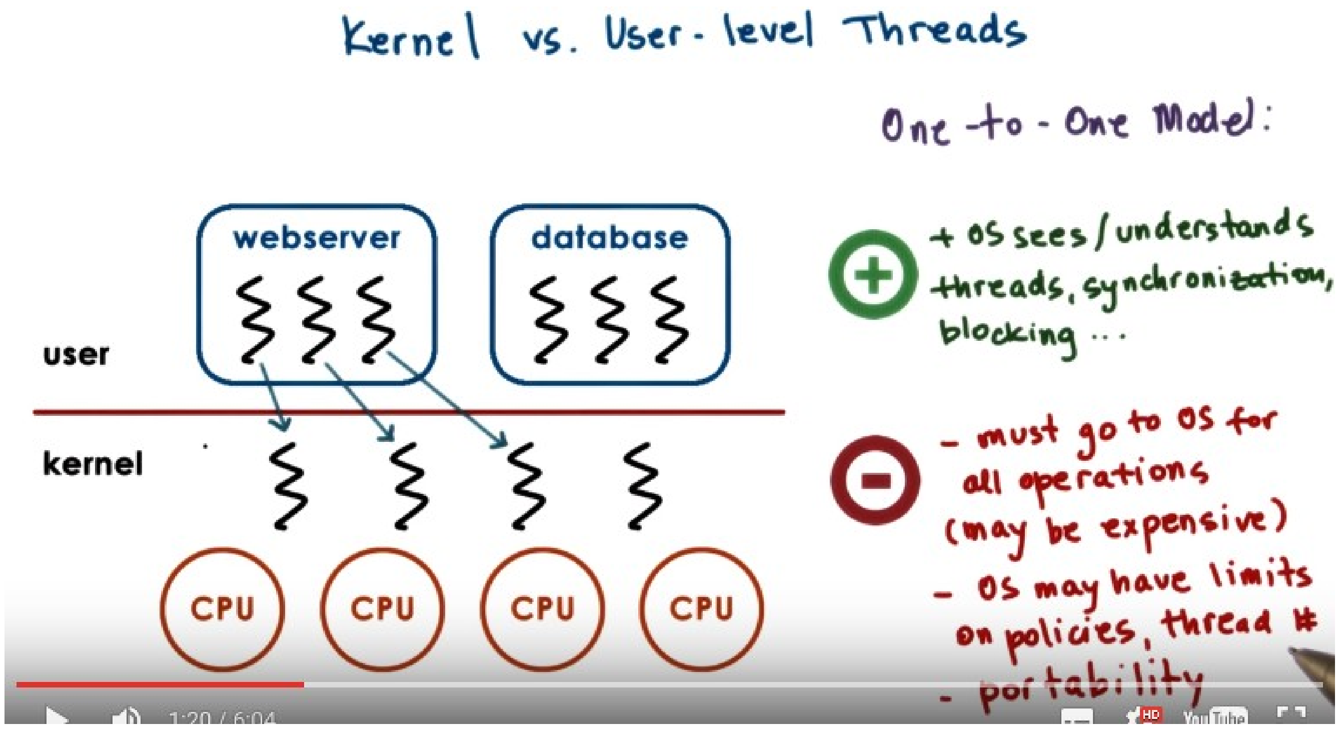
[ ]
```
while ((new_order >= 3) OR
    (new_order == 1 AND old_order >= 0))
```

[✓]
```
while ((new_order >= 3) OR
    (new_order == 1 AND old_order >= 1))
```

34. 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.


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

35. 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 multithreaded. 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 will decide how these kernel level threads will be mapped onto the underlined physical CPUs, and which one of them will execute at any given point of time. Some of these kernel level threads may be there to directly support some of the processes. So, they may execute some of the user-level threads and other kernel level threads may be there just to run certain OS level services like daemons, for instance. At the user level, the processes themselves, are multi-threaded, and these are the user level threads. For a user level thread to actually execute, first, it must be associated with a kernel level thread. And then, the OS level scheduler must schedule that kernel level thread onto a CPU. So let's investigate a little more, what is the relationship that exists between user level threads and kernel level threads. In fact, there's several possible relationships between the user level threads and the kernel level threads. And we will now look at three such models.

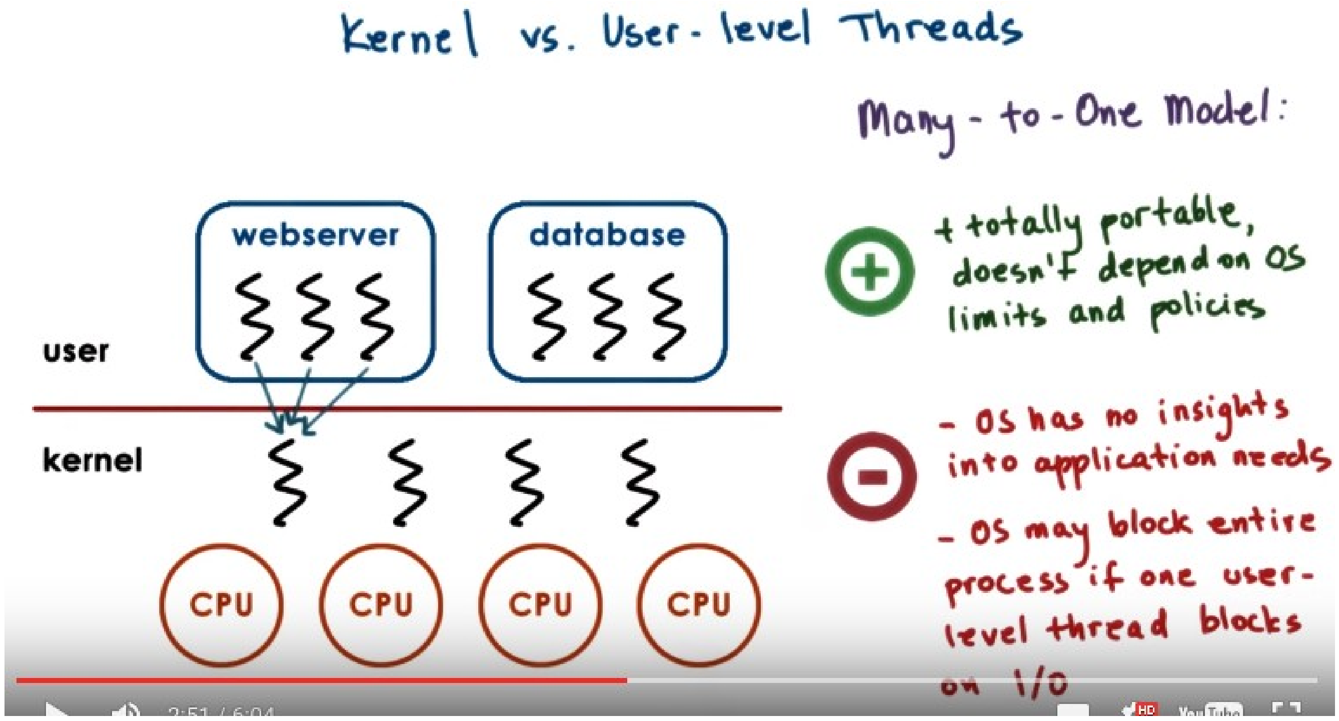


Kernel vs. User-level Threads

One-to-One Model:

+ OS sees/understands threads, synchronization, blocking ...

- must go to OS for all operations (may be expensive)
- OS may have limits on policies, thread #
- portability

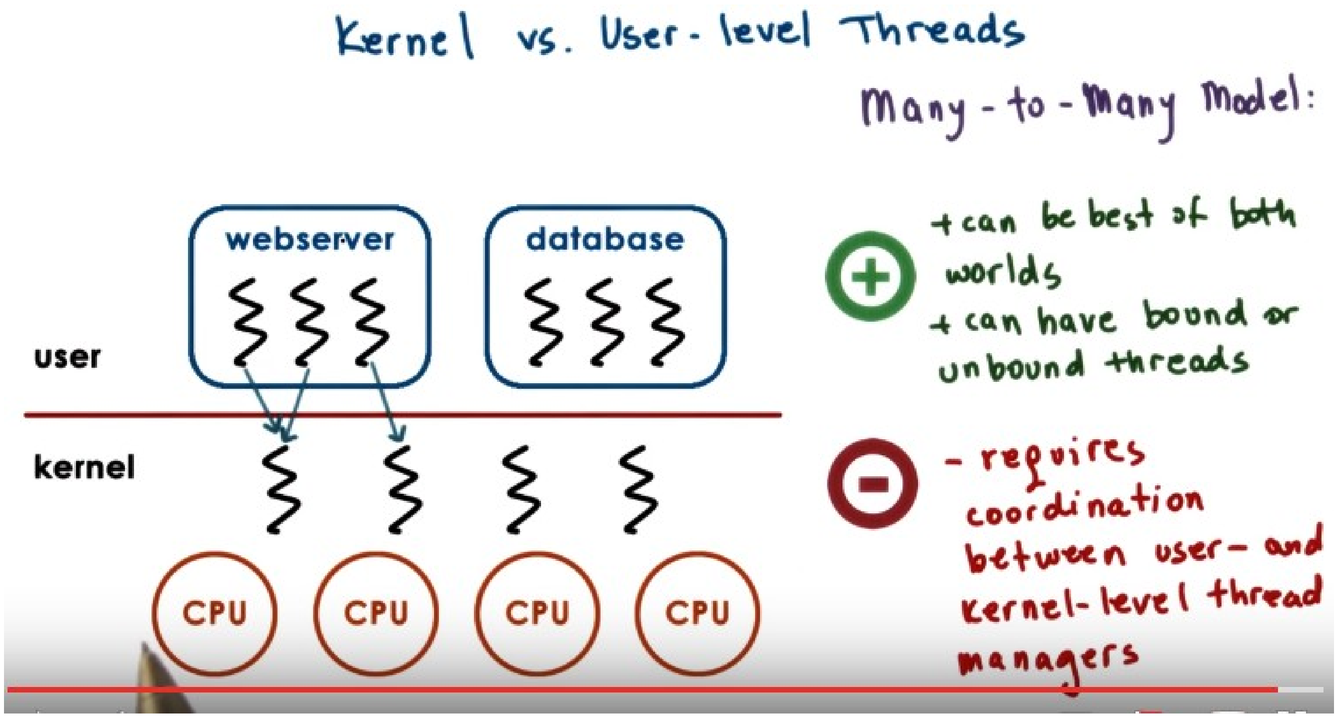


Kernel vs. User-level Threads

Many-to-One Model:

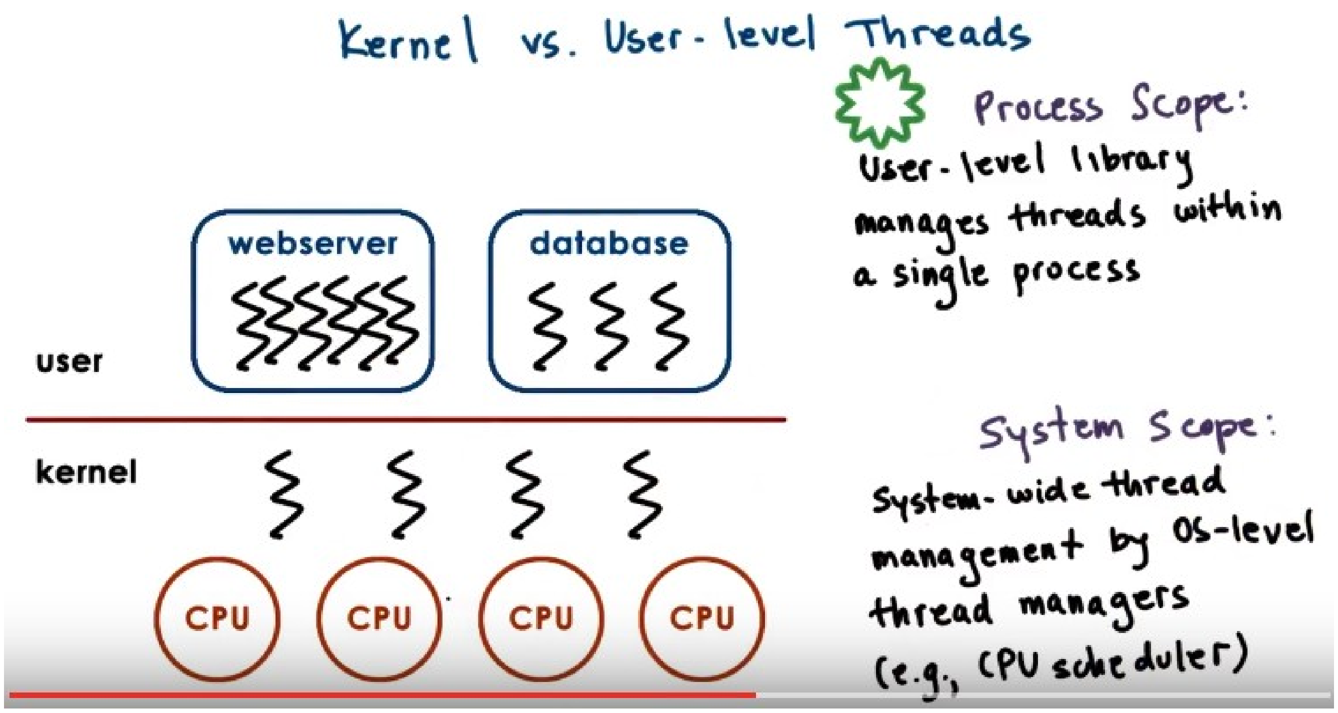
+ totally portable, doesn't depend on OS limits and policies

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


Kernel vs. User-level Threads
Many-to-Many Model:
webserver
database
user
kernel
CPU
CPU
CPU
CPU
+ can be best of both worlds
+ can have bound or unbound threads
- requires coordination between user- and kernel-level thread managers

36. The first model is a One-to-One Model. Here each user-level thread has a kernel-level thread associated with it. When the user process creates a new user-level thread, there is a kernel-level thread that either is created or there is available kernel-level thread, then a kernel-level thread is associated with user-level thread. This means that the operating system can see all of the user-level threads. It understands that the process is multithreaded, and it also understands what those threads need in terms of synchronization, scheduling, blocking. So as the operating system already supports these mechanisms in order to manage its threads, then the user-level processes can directly benefit from the threading support that's available in the kernel. The downside of this approach is that, for every operation we have to go to the kernel, so we have to pay the cost of a system call, of crossing the user to kernel boundary. This we discussed already could be expensive. This model also means that since we're relying on the kernel to do the thread-level management synchronization, scheduling et cetera, we are limited by the policies that are already supported at the kernel level. So for instance if the kernel doesn't support a particular scheduling policy or if the kernel has a limit on the number of threads that can be supported, the process is restricted to operate within those bounds. This basically affects the portability, so if a particular process, a particular application, has certain needs about how its threads should be managed, we're limited to running that kind of process only on the kernels that provide exactly that kind of support. The second model is the Many-to-One Model. Here all of the user-level threads are supported, are mapped onto a single kernel-level thread. What this means is that at the user level there is a thread management library that decides which one of the user-level thread will be mapped onto the kernel-level thread at any given point of time. That user-level thread, of course, will run only once the kernel-level thread is actually scheduled on the CPU. The benefit of this approach is that it's totally portable. Everything will be done at the user-level thread library, scheduling, synchronization, et cetera, and so we don't rely on any specific kernel-level support. Similarly we're not limited by the specific limits and policies that are available in the kernel. Also because all of the thread management is done at the user level by the user-level threading library, we don't have to make system calls, we don't have to rely on user kernel transitions, in order to make

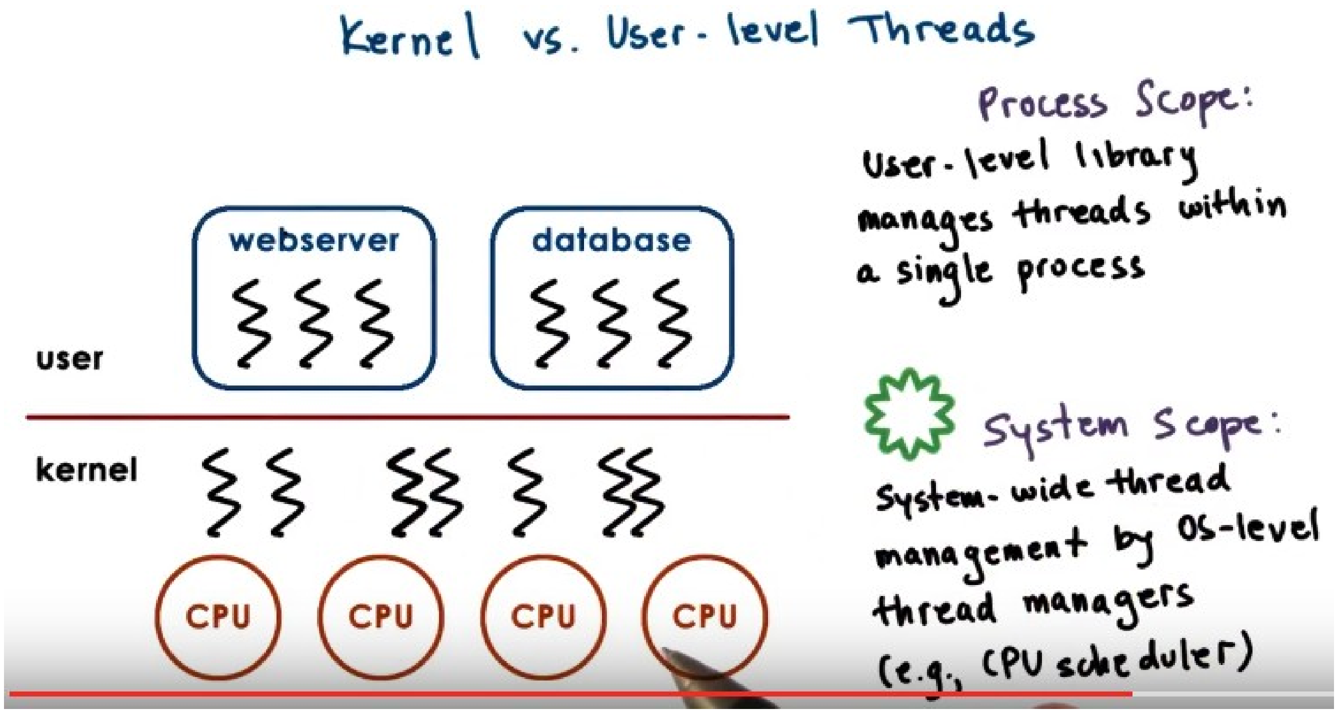
decisions regarding scheduling, synchronization, blocking, et cetera. So this is very different than what we saw in the one-to-one model. The problem with this approach is, however, that the operating system has really no insight into the application needs, it doesn't even know that the process is multithreaded. What the OS sees is just a kernel-level thread, so the real danger is that when the user-level library schedules one user-level thread onto the kernel-level thread, and let's say this user-level thread makes a request for an I operation that's blocking. The kernel level scheduler will see that the kernel-level thread block, and it will basically block the entire process. So the fact that there may be other user-level threads that have useful work to do and the process overall can make some progress, that's hidden from the operating system, from the kernel, and the whole process is forced to wait. This is obviously going to have some implication on performance. Finally there is the Many-to-Many Model. The Many-to-Many Model allows some user-level threads to be associated with one kernel-level process, others perhaps to have a one-to-one mapping with a process, so sort of the best of both worlds. The kernel knows that the process is multithreaded since it has assigned multiple kernel level threads to it. And also, if one user-level thread blocks an I/O, and as a result the kernel-level thread is blocked as well, the process overall will have other kernel-level threads onto which the remaining user-level threads will be scheduled. So the process overall can make progress. The user-level threads may be scheduled onto any of the underlying kernel-level threads, so they're unbound. Or we can have a certain user-level thread that's basically mapped one-to-one permanently onto a kernel-level thread. We called this the bound mapping. The nice thing about this is that if we have certain user-level threads that somehow even the kernel should be able to treat differently, to ensure that they have better priority or are more responsive to certain events that are happening in the kernel, we have a mechanisms to do this by these bound threads. For instance, if the kernel sees that there is some user input, and we have a particular user level thread that's designated to run the user interface for this process, the kernel has a way to immediately schedule the corresponding kernel-level thread as a result of that, also the corresponding user-level thread. So this ability to provide this kind of one-to-one, permanent mapping in the many-to-many model is another benefit of the model. There are still some cons with this model. And that's, in particular, because now we require some coordination between the kernel-level thread management and the user-level thread management, which we didn't see in the other cases. In the one-to-one model, pretty much everything goes up to the kernel-level manager, and in the many-to-one model, pretty much everything goes up to the user-level thread manager that's part of the thread's library. In the many-to-many model there's often the case where we require certain coordination between the kernel and user-level managers, mostly in order to take advantage of some performance opportunities.



Kernel vs. User-level Threads

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

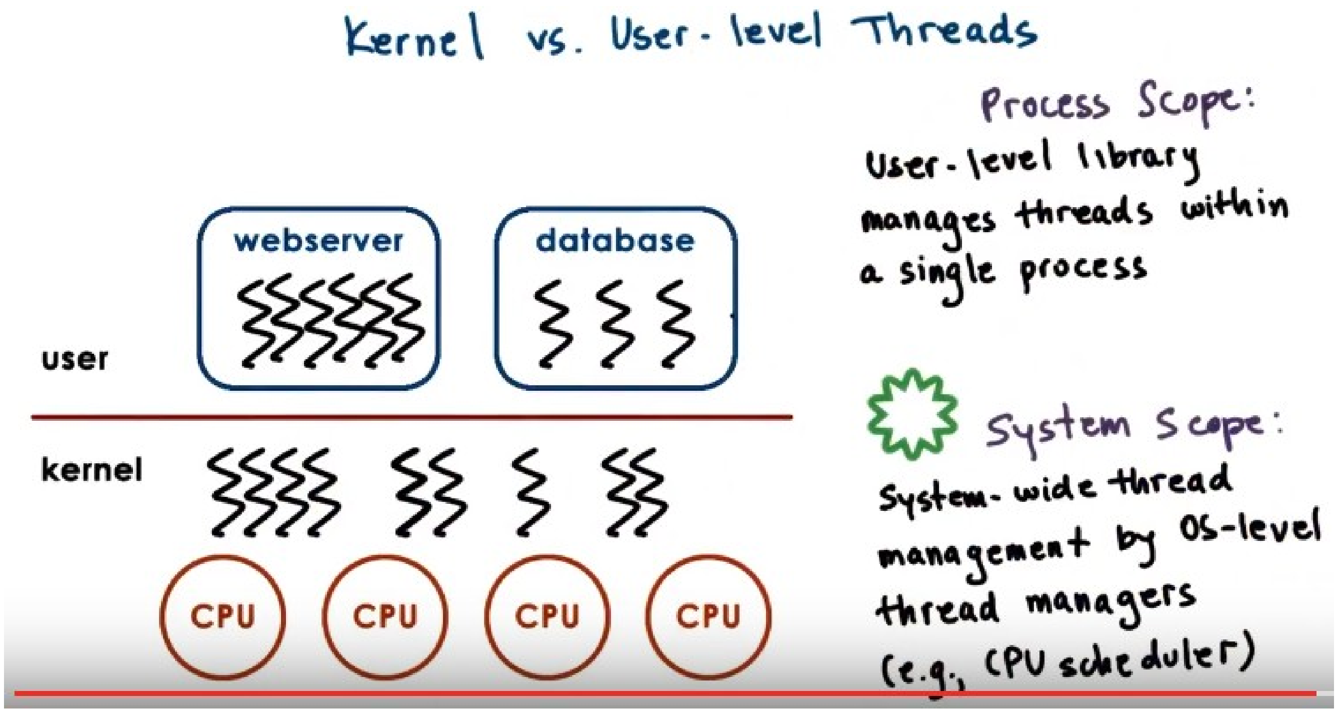
System Scope:
System-wide thread management by OS-level thread managers (e.g., CPU scheduler)



Kernel vs. User-level Threads

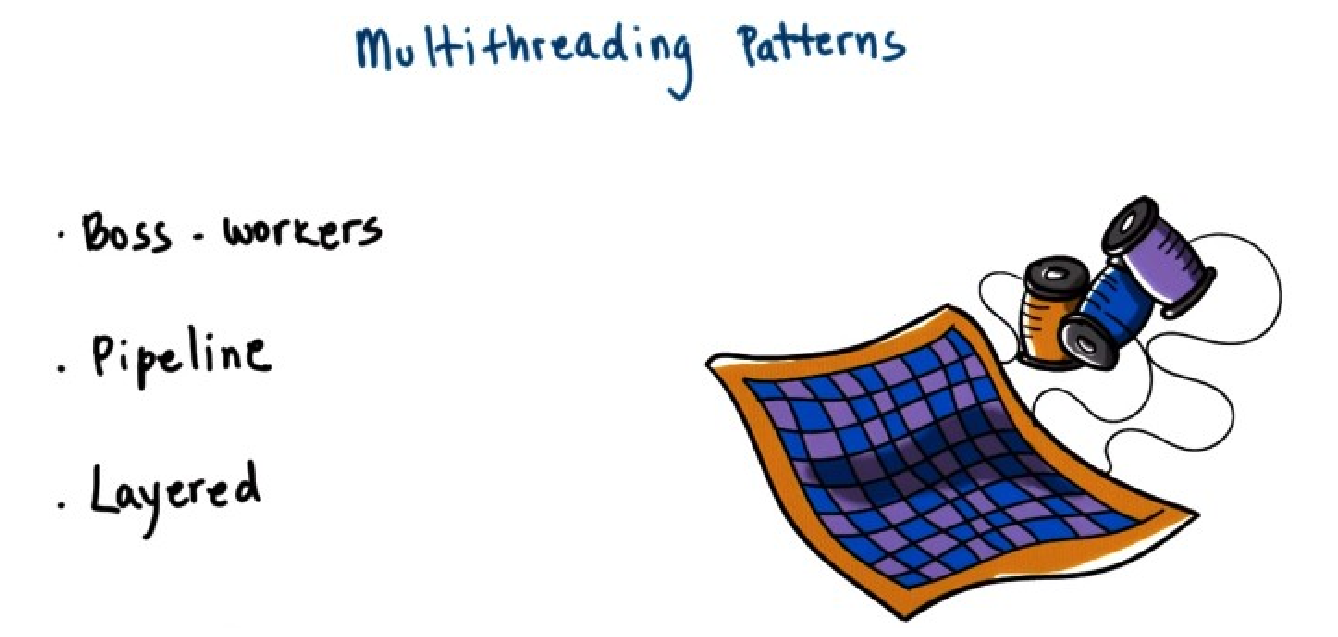
Process Scope:
User-level library manages threads within a single process

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


Kernel vs. User-level Threads
webserver
database
user
kernel
CPU
CPU
CPU
CPU
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)

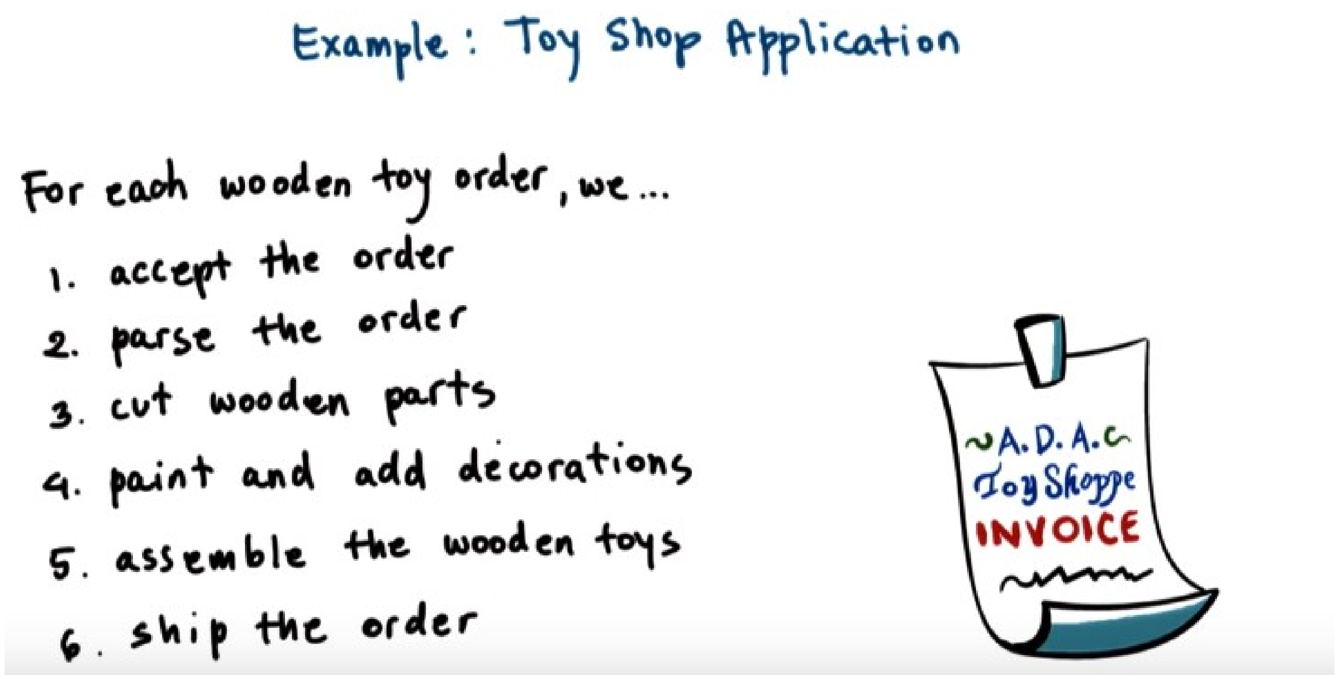
(上圖的 kernel 部分應該畫得不對)

37. We will discuss later some of the implications on implementation that are there because of the interactions between the user-level threads and the kernel-level threads. But for now, you need to understand that there are different levels at which multi-threading is supported, at the entire system or within a process. And that each level affects the scope of the thread management system. At the kernel level we have system-wide thread management that's supported by the operating system-level thread managers. What this means is, that the operating system thread managers will look at the entire platform we're making decisions, as to how to allocate resources to the threads. This is the system scope. On the other end at user level, a user-level thread library that's linked to the process manages all of the threads that are within that single process only. So the management scope is process wide. Different processes will be managed by different instances of the same library. Or even different processes may link entirely different user-level libraries. To illustrate the effects of having a different scope, let's take a look at the following situation. Let's say the web server has twice as many threads as the database. If the user-level threads have a process scope, the operating system doesn't see all of them. So at the operating system level, the available resources will be maybe managed 50/50 among the two different processes. That means that both the web server and the database will be allocated equal share of the the kernel level threads, so two each. And then the OS level scheduler will manage these threads by splitting the underlying CPUs amongst them. The end result of that however, is that the webserver's user level threads, will have half of the amount of the CPU cycles that's allocated to the database threads. Now if we have a System Scope, the user-level threads all of them, will be visible at the kernel level. So the kernel will allocate to every one of its kernel-level threads. And therefore, to every one of the user-level threads across the two applications, in equal portion of the CPU. If that happens to be the policy that the kernel implements. As a result, if we have a situation in which one process has more user-level threads than the other, this process will end up receiving a larger share of the underlying physical resources. Since very one of its user level threads will get equal share of the physical resources as the user level threads in the other process.



Multithreading Patterns

- Boss - workers
- Pipeline
- Layered



Example: Toy Shop Application

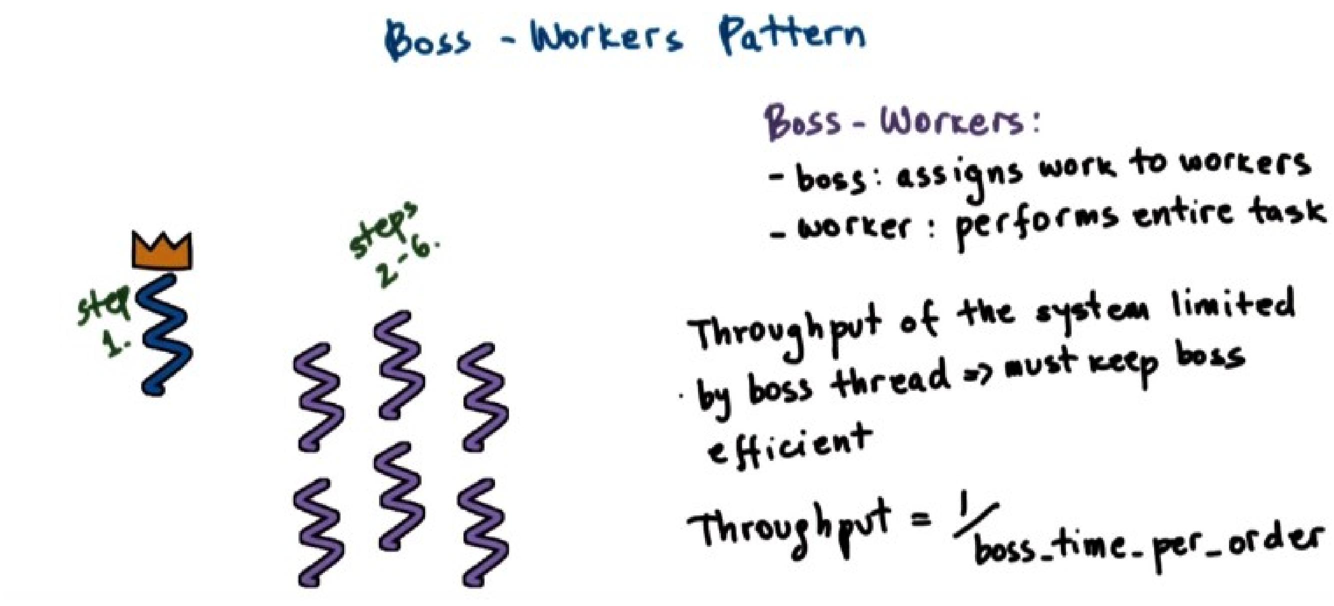
For each wooden toy order, we...

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

A.D.A.C
Toy Shoppe
INVOICE

38. Before we conclude this lesson, let's discuss some useful multithreading patterns for structuring applications that use threads. We will look at the boss-workers pattern, the pipeline pattern, and the layered pattern. Before we start, let's take a look at the toy shop application. We will describe these pattern in the context of this application. In this application, for each toy order we receive, and let's say we're sticking to wooden toy orders, we have to perform the following steps. First, we have to accept the order from the customer. Then, we have to parse the order to see what it's for. Then, we have to start cutting the wooden parts for the toy. Then, we need to paint and add decorations for the toy parts. Then, all those parts need to be put together to assemble the wooden toy. And finally, we need to ship

the order. Depending on the multithreading pattern, these steps will be assigned differently to the workers in the workshop.



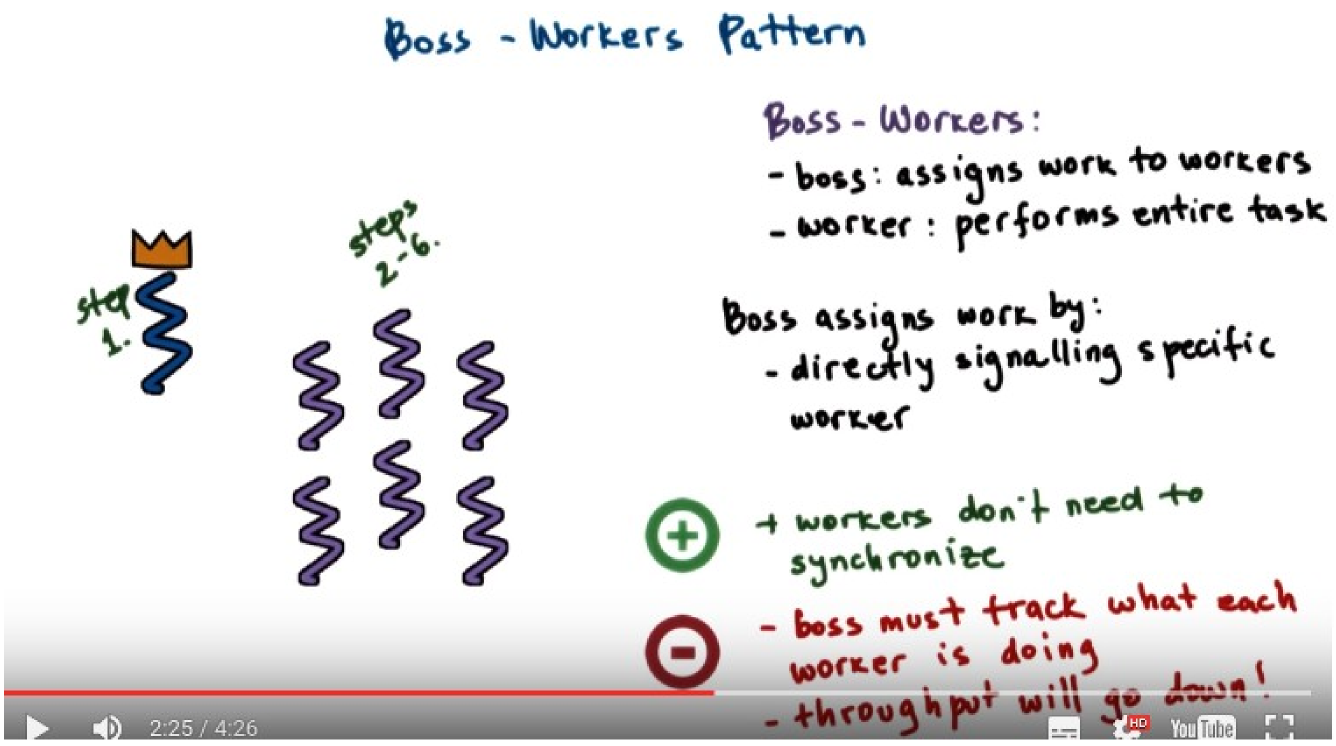
Boss - Workers Pattern

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

throughput: 生產率



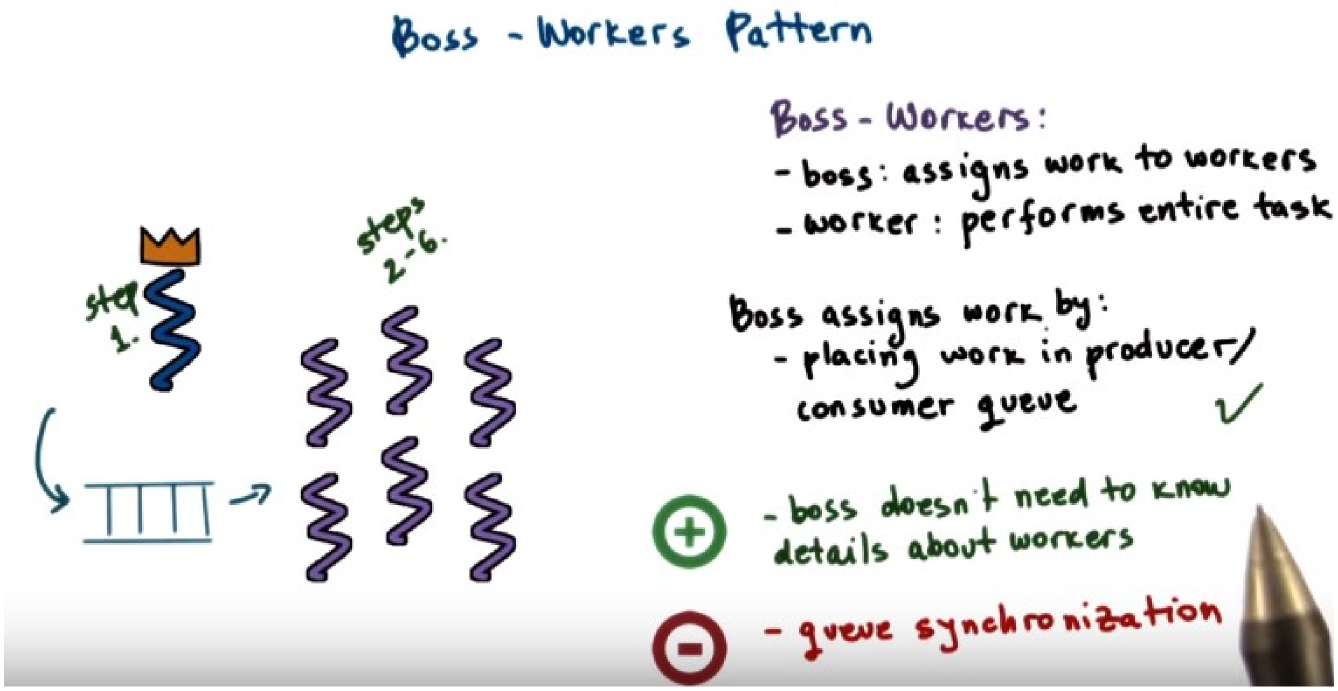
Boss - Workers Pattern

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

Boss assigns work by:
- directly signalling specific worker

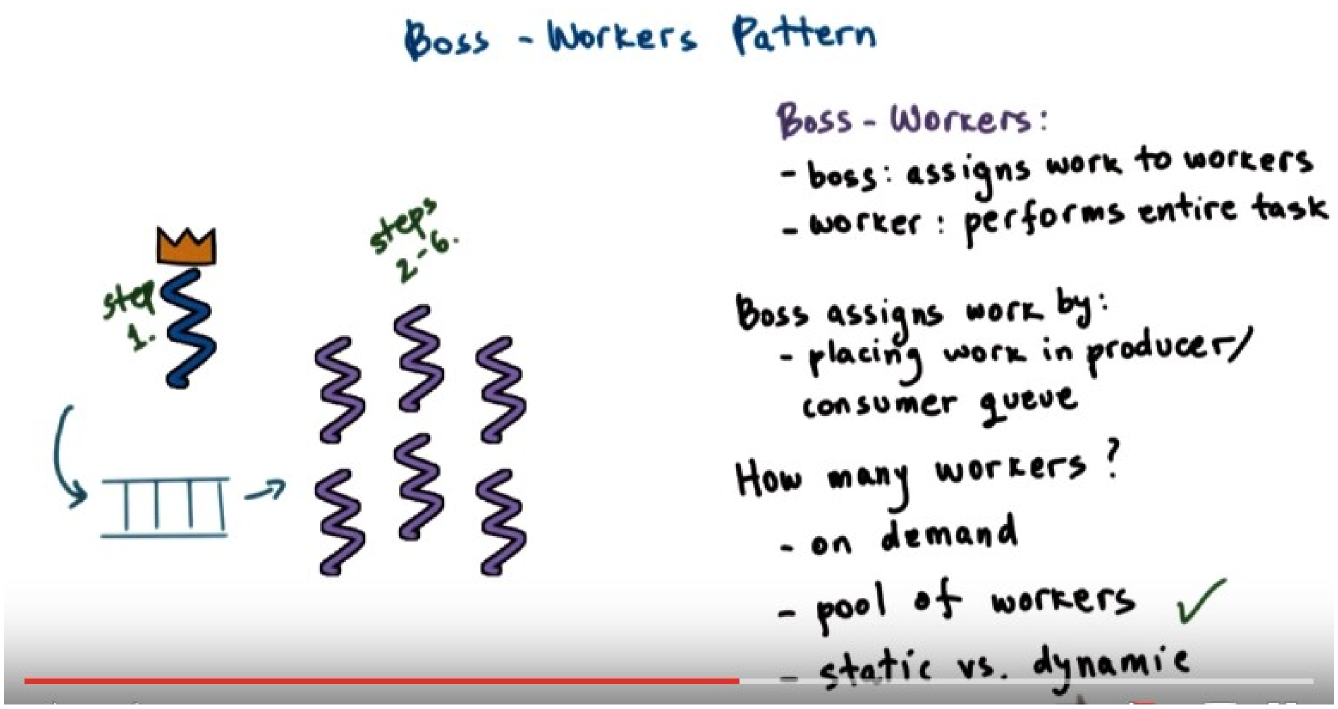
+ workers don't need to synchronize
- boss must track what each worker is doing
- throughput will go down!

synchronize: 使同步,使合拍


Boss - Workers Pattern
Boss - Workers:
- boss: assigns work to workers
- worker: performs entire task
step 1.
steps 2-6.
Boss assigns work by:
- placing work in producer/ consumer queue ✓
- boss doesn't need to know details about workers
- queue synchronization

39. We will first look at the boss-workers pattern. This is a popular pattern that's characterized by one boss thread and then some number of worker threads. The boss is in charge of assigning work to the workers, and the workers are responsible for performing the <u>entire</u> task that's assigned to them (注意每個 worker 幹的活都是樣的, 且每個 worker 都要幹所有的活, 即每個 worker 都把一個完整的現俱做出來). Concerning our toy shop example, that means that the very first step, the step where we accept an order will be performed by the boss. The boss will accept an order and then immediately pass it on to one of the workers. Each of the workers will perform steps two through six, so we'll parse the order, cut the pieces, stain the pieces, and assemble the wooden toy and ultimately ship the order. Since we only have one boss thread that must execute on every single piece of work that arrives in the system, it means that the throughput of the system overall is limited by the boss' performance. Specifically, the throughput of the system is inversely proportional to the amount of time the boss spends on each order. So, clearly, that means that we must keep the boss efficient if we want to make sure that the system overall is performing well. In our toy shop example, the boss thread just picks up an order from the customer and immediately passes it to the workers. It doesn't really look to see what it's for. That's why each of the workers starts with step two. So in that way we're trying to limit the amount of operation that's required from the boss on each order. So how does the boss pass work to one of the workers? One way is for the boss to keep track of exactly which workers are free, and then hand off work to those workers. So, it's specifically signalling one particular worker. This means that now the boss will have to do more for each order, because now it has to keep track of which of the workers are available. And will also have to wait for that particular worker to accept the order from the boss when, when its being passed, sort of like a handshake. The positive of this approach is that the workers don't need to synchronize amongst each other in any way. The boss will tell them what they need to do, and they don't have to care about what the other workers do. The downside, however, is that given that the boss now has to keep track of what the workers are doing, the throughput of the system will go down. Another option is to establish a queue between the boss and the workers. This could be similar to a producer/consumer queue, where the boss is the only producer that produces work requests, so toy

orders for the workers. And then the workers are the consumers that are picking up work from this queue, picking up orders from this queue and then proceeding with the steps that they need to perform. The upside of this approach is that the boss now doesn't really need to know about what each worker is doing and whether it's free. It also doesn't have to wait for a worker to explicitly do, a handshake when it's passing off a work item to one of them. The boss just accepts an order, so performs steps one, places the order on the shared queue, and can go back to picking up the next order. Whenever one of the worker becomes free, it looks into the queue, at the front of the queue ideally, and picks up any pending work requests. The downside is that now the workers, as well as the workers and the boss amongst each other, have to synchronize their accesses to the shared queue. All of the worker threads may contend to gain access to the front of the queue, and any one work item can only be assigned to one worker thread. And also the workers and the boss may need to synchronize when they need to compare the front and the end pointer of this queue, for instance, when they need to determine that a queue is full or that a queue is empty. Despite of this downside, this approach of using a shared queue among the boss and the workers when passing work among them, still results in lower time per order that the boss needs to spend. So it results in better throughput of the system overall. So that's why we tend to pick this particular model when a building multithread applications using this pattern

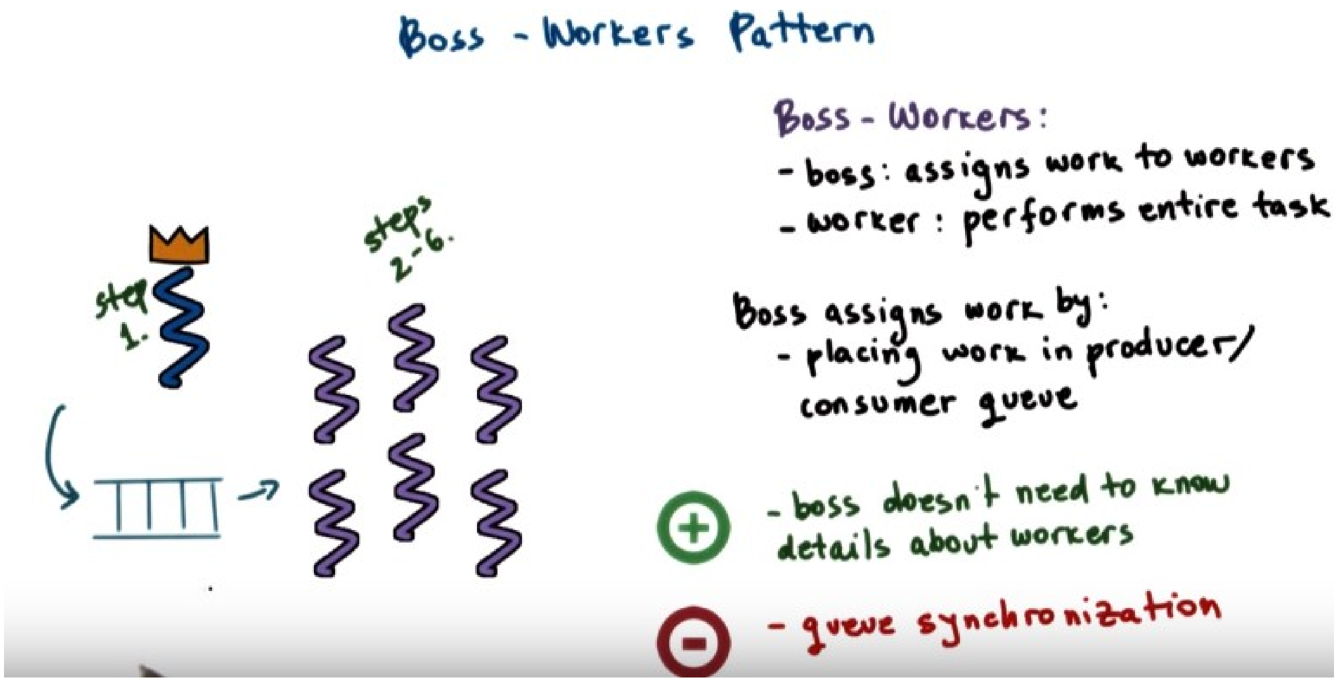


Boss - Workers Pattern

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

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

How many workers?
- on demand
- pool of workers ✓
- static vs. dynamic



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

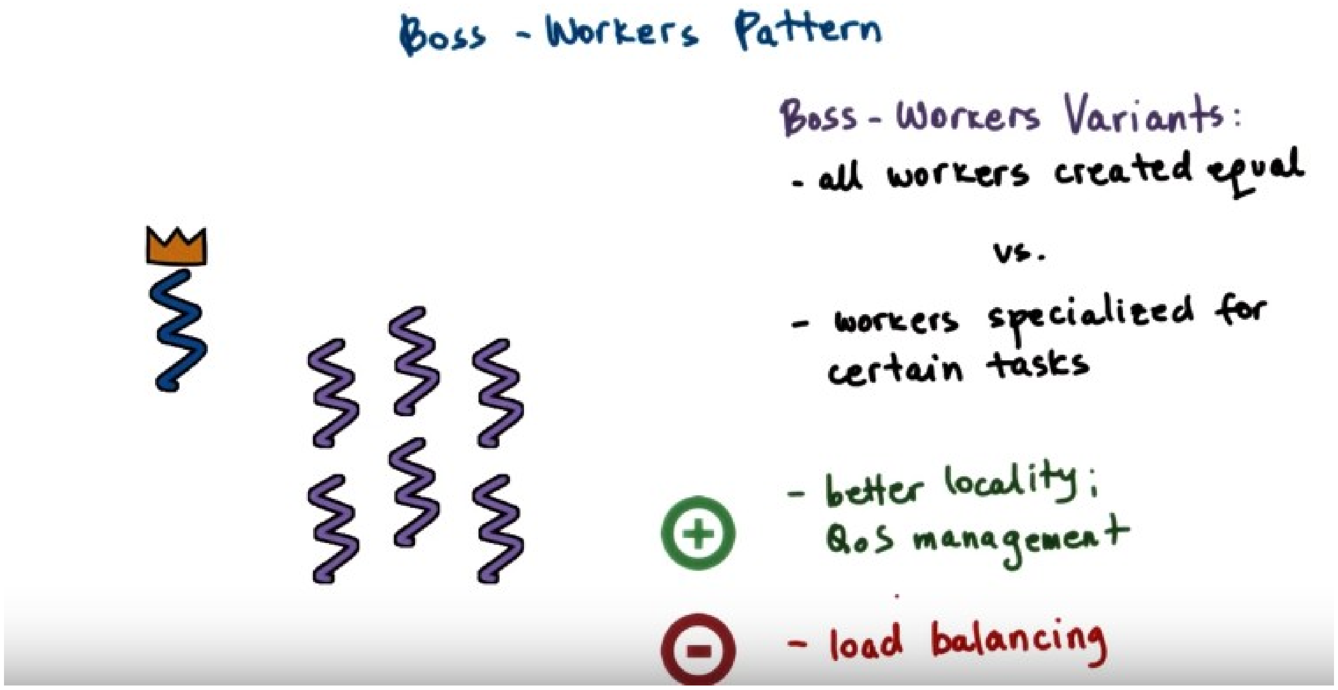
Boss assigns work by:
- placing work in producer/consumer queue

⊕ - boss doesn't need to know details about workers

⊖ - queue synchronization

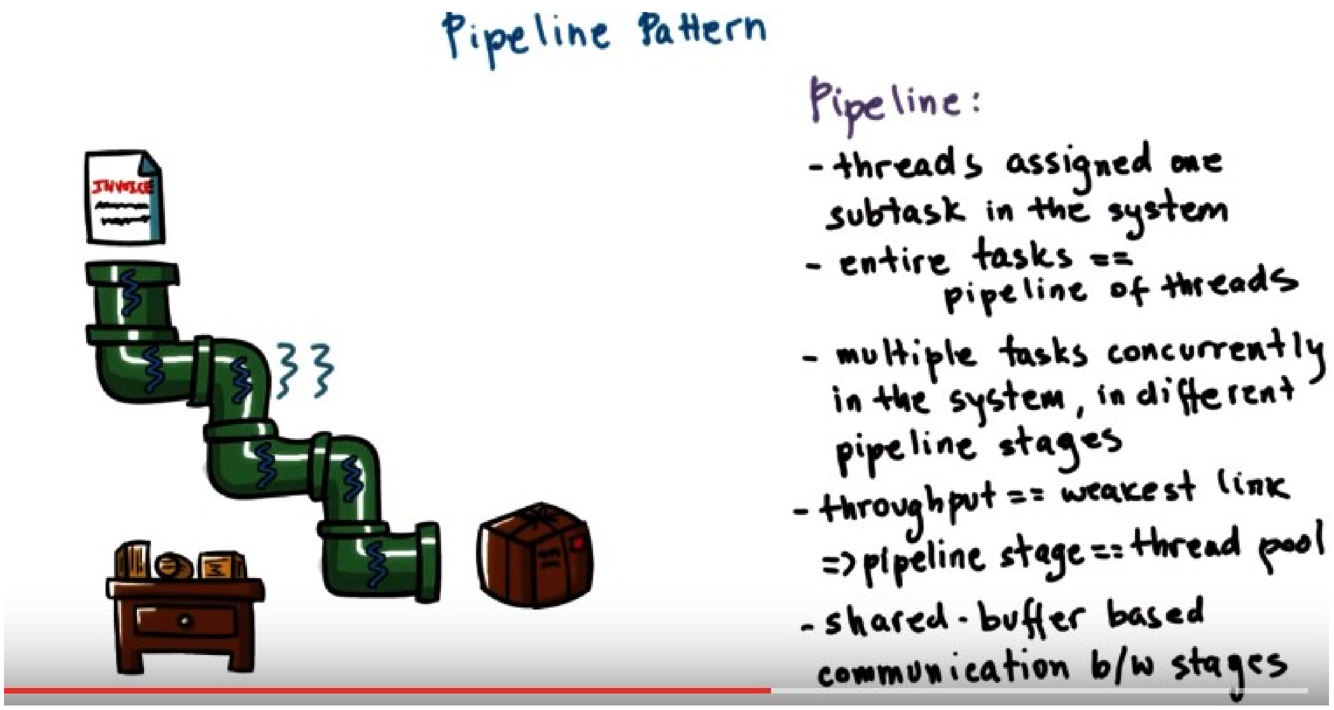
40. So if we use this queue structure, the performance of the system overall will depend on whether or not the boss thread has to wait when inserting work requests, toy orders into this queue. If the queue is full, the boss will have to wait, the time that it spends per order will increase, and overall, the throughput of the system will go down. Clearly, if we have more threads(即 workers), it's less likely that the queue will be full, but arbitrarily increasing the number of threads will add some other overheads in this system. So the question is, how many workers is enough? Well, we can add more workers dynamically on demand. Whenever we have yet another order, we go and call yet another worker to join the crew. This clearly can be very inefficient if we have to wait a long time for a worker to arrive. A more common model, therefore, is to have a pool of workers that's created up front. With such a pool of workers, or pool of threads since a worker is directly supported by a thread, we don't have to wait for a new thread to be created or a new worker to arrive every single time we start seeing that order start piling up on the queue. The question is, though, how do we know how many workers, how many thread to pre-create in this pool? A common technique is to use this pool of workers or pool of threads model, but as opposed to statically deciding what the size of the pool should be, is to allow the pool to be dynamically increased in size. Unlike the purely on demand approach, these increases won't happen one thread at a time, rather we'll create several threads whenever we determine that the pool size needs to be adjusted. So this tends to be the most effective approach of managing the number of threads in the boss-worker pattern. So to summarize, so far we saw that the boss-workers model has these features. A boss assigns work to the workers. The workers, every one of them, performs the entire task. The boss and the workers communicate via shared producer consumer queue, and we use a worker pool based approach to manage the number of threads in the system where we potentially adjust the size of the pool dynamically. The benefit of this approach is in its overall simplicity. One thread assigns work to all others. All other threads perform the exact same task. The negatives of this approach include the overhead related to the management of the thread pool, including synchronization for the shared buffer. Another negative of this approach is the fact that it ignores locality. The boss doesn't keep track of what any one of the workers was doing last. If we have a situation in which a

worker just completed performing a similar type of task or identical type of task, it is more likely that that particular worker will be more efficient at performing that exact same task in the future. Or maybe it already has some of the tools that are required for building that particular type of toy nearby on its desk. But if the boss isn't paying attention to what the workers are doing, it has no way of making that kind of optimization.


Boss - Workers Pattern
Boss - Workers Variants:
- all workers created equal
vs.
- workers specialized for certain tasks
- better locality; QoS management
- load balancing

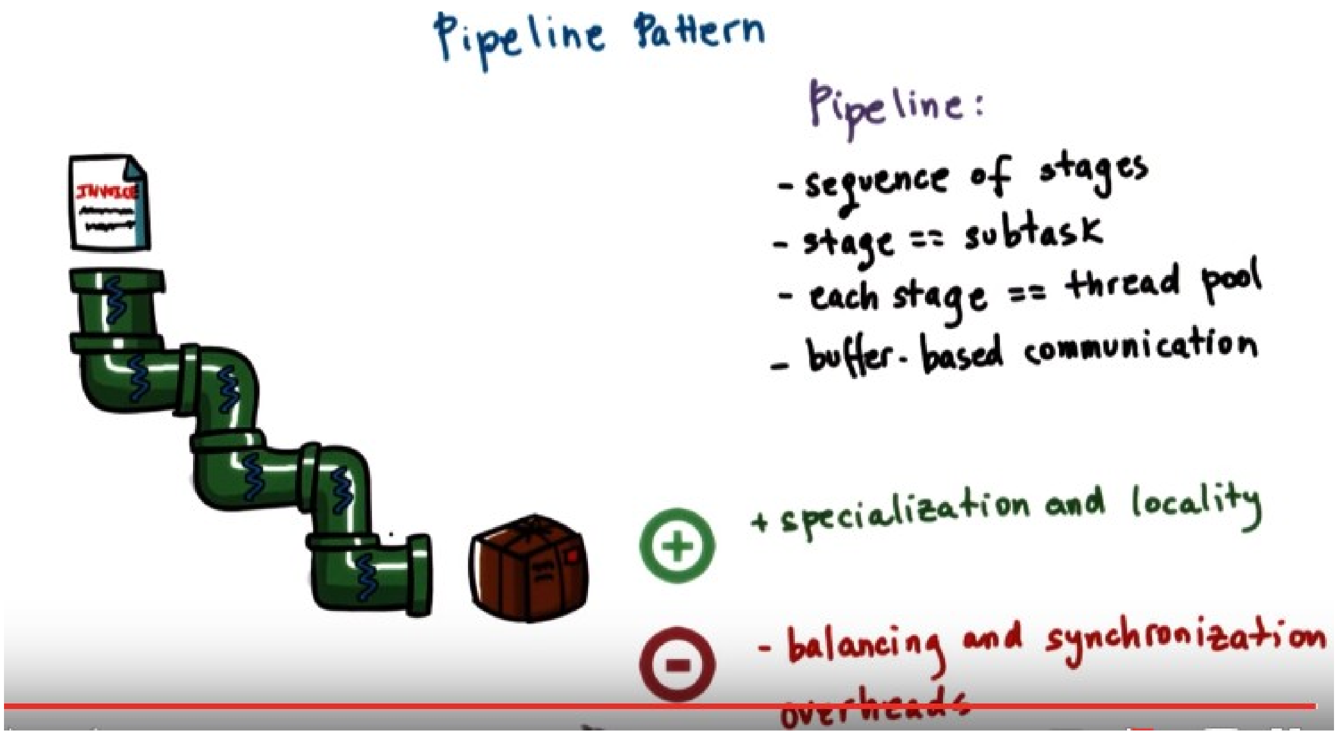
QoS: Quality of Service

41. An alternative to having all workers in the system perform the exact same task, so all workers be equal, is to have different workers specialized for different sets of tasks. In the context of the toy shop example, this may mean that we have workers specialized for different types of toys. Or it could mean that we have workers specialized for repairs versus for brand new orders, or even we can specialize the workers to deal with different types of customers. One added stipulation in this case is that now the boss has to do a little bit of more work for each order. Because in addition to just accepting the order it has to take a look at it, and decide which set of workers it should actually be passed to. Now the fact that the boss has to do a little bit of more work per order is likely offset by the fact that each of the worker threads will now be more efficient, because they're specialized for the task. So overall we can achieve better performance. The real benefit of this approach is that it exploits locality. Each of the threads, by having to do a subset of the tasks, it ends up probably accessing only a subset of the state, and therefore that state is more likely present in the cache. And we already talked about the benefits of having state present in cache. It is much faster to access than if we have to go to memory. Also, by being able to assign different workers to different types of tasks or different types of customers, we can do better quality of service management. So we can assign more threads to those tasks or those customers that we need to give higher quality of service to. This main challenge in this variant of the boss-workers model comes from the fact that it is now much more complicated to do the load balancing. How many threads should we assign for the different tasks? This is not necessarily a question that has a unique answer regardless of the amount of flow, the number of requests that are coming in the system, the kind of hardware, so the kind of tools that these threads use. So it ends up being a more complicated type of question.


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:

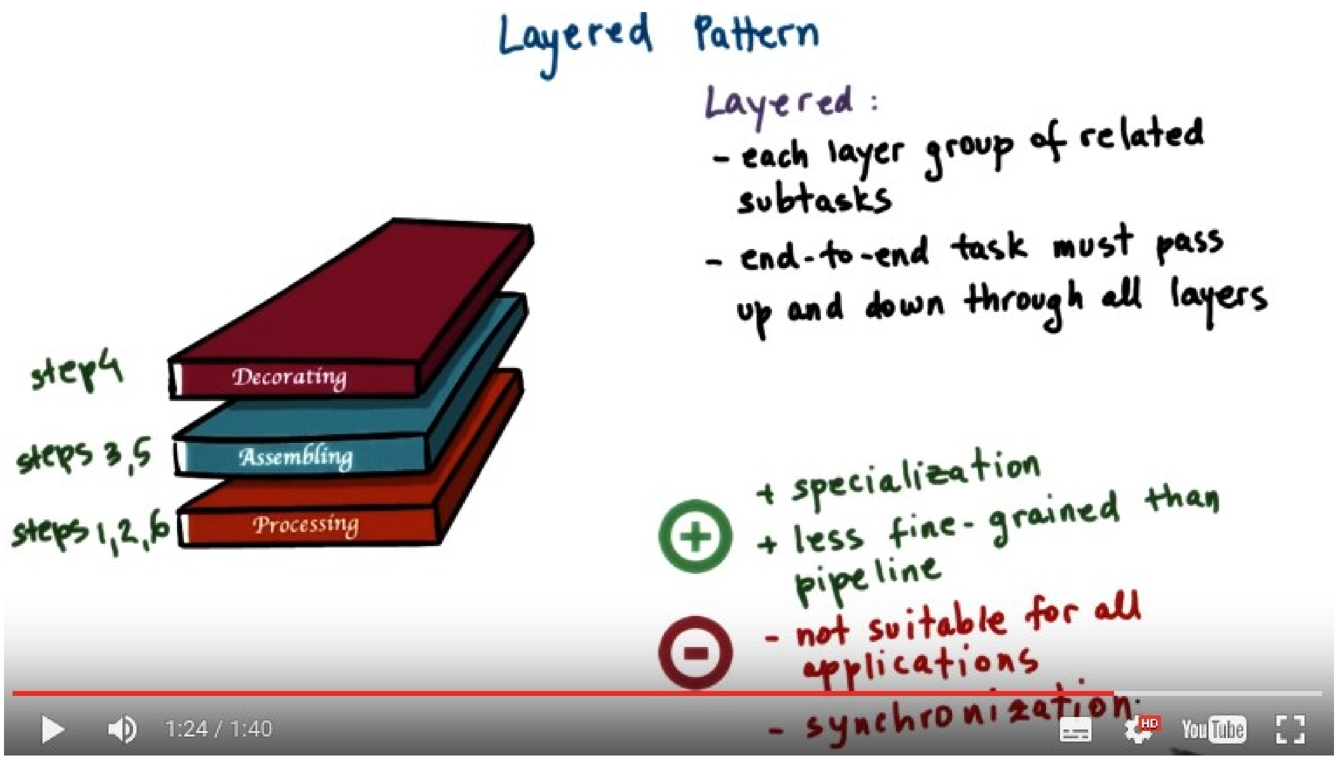
- sequence of stages
- stage == subtask
- each stage == thread pool
- buffer-based communication

+ specialization and locality

- balancing and synchronization overheads

42. A different way to assign work to threads in a multithreaded system is using this pipeline approach. In this pipeline approach, the overall task, the processing of the toy shop, is divided into subtasks, and each of the subtasks is performed by a separate thread. So threads are assigned subtask in the system, and then the entire applications, so the entire complex task is executed as a pipeline of threads. In the context of our toy shop example, for instance, what this would mean is that given that we have six steps

in the toy order processing, we can have six workers and every single one of the workers will be assigned one steps to process. At any given point of time we can have multiple tasks concurrently in the system. So, (此句誤導, 不看)multiple toy shop orders being processed by different workers. It just every single one of those orders, every single one of those tasks, will be in a different stage in the pipeline. So, one worker can be accepting one toy order, another worker can be performing the parsing of the order for another toy, a third worker can be cutting the wooden pieces for yet another toy and so forth. The throughput of the system overall will clearly be dependant on the weakest link, the longest stage in the pipeline. Ideally, we would love it if every single stage of the pipeline takes approximately the same amount of time, but sometimes it just may not be possible. So the way to deal with this is using the same thread pull technique that we saw before. If one of the pipeline stages for instance, the wood cutting stage, takes more amount of time longer. Then we can assign multiple threads to this particular stage. Let's say it takes three times longer to perform this stage in the pipeline compared to all the other stages in the pipeline. If we assign three different threads, three different workers, to this stage, then overall, every single stage in the pipeline will approximately be able to process the same number of subtasks, the same number of toy orders over a period of time. So the system overall will still be balanced. The best way to pass work among these different pipeline stages is using a shared-buffer based mechanism (即 the queue based approach) similar to the producer consumer shared buffer that we saw in the previous pattern. The alternative to that would be to require some explicit communication between threads in the pipeline, and this will mean that a thread in an earlier stage will potentially have to wait until a thread in the next stage is free. The shared-buffer based approach, 即 the queue based approach, helps correct for any kind of small imbalances in the pipeline. For instance, with direct communication in the context of our toy shop, it would require workers directly to hand off the output of their processing, so cut pieces etcetera to the next worker, to the next thread. Where as in the shared buffer base communication you can think of a worker leaves the output of its processing, so the piece is on the table and the next worker picks them up whenever he or she's ready. In summary, a pipeline is a sequence of stages where a thread performs a stage in the pipeline and that's equivalent to some subtask in the end-to-end processing. To keep the pipeline balanced a stage can be executed by more than one thread. And we can use the same thread pool management technique that we described in the Boss-Workers model to determine what is the right number of threads per stage. Passing partial work products or results across the stages in the pipeline should be done via a shared buffer based communication. This provides for some elasticity in the implementation and avoids stalls due to temporary pipeline imbalances. A key benefit of the approach is the fact that it allows for highly specialized threads and this leads to improved efficiency. Like what we saw in the variant of the boss-worker model, when threads perform a more specialized task, it's more likely that the state that they require for their processing is present in the processor cache and that kind of locality can ultimately lead to improve performance. A negative of the approach is the fact that it is fairly complex to maintain the pipeline balanced over time. When the work load pattern changes, so we have more toys arriving or when the resources of the pipeline change, so a worker slows down or takes breaks, we'll have to rebalance the entire pipeline to determine how many workers to assign to each stage. In addition to that, there is more synchronisation, since there are synchronisation points at multiple points in the end to end execution.

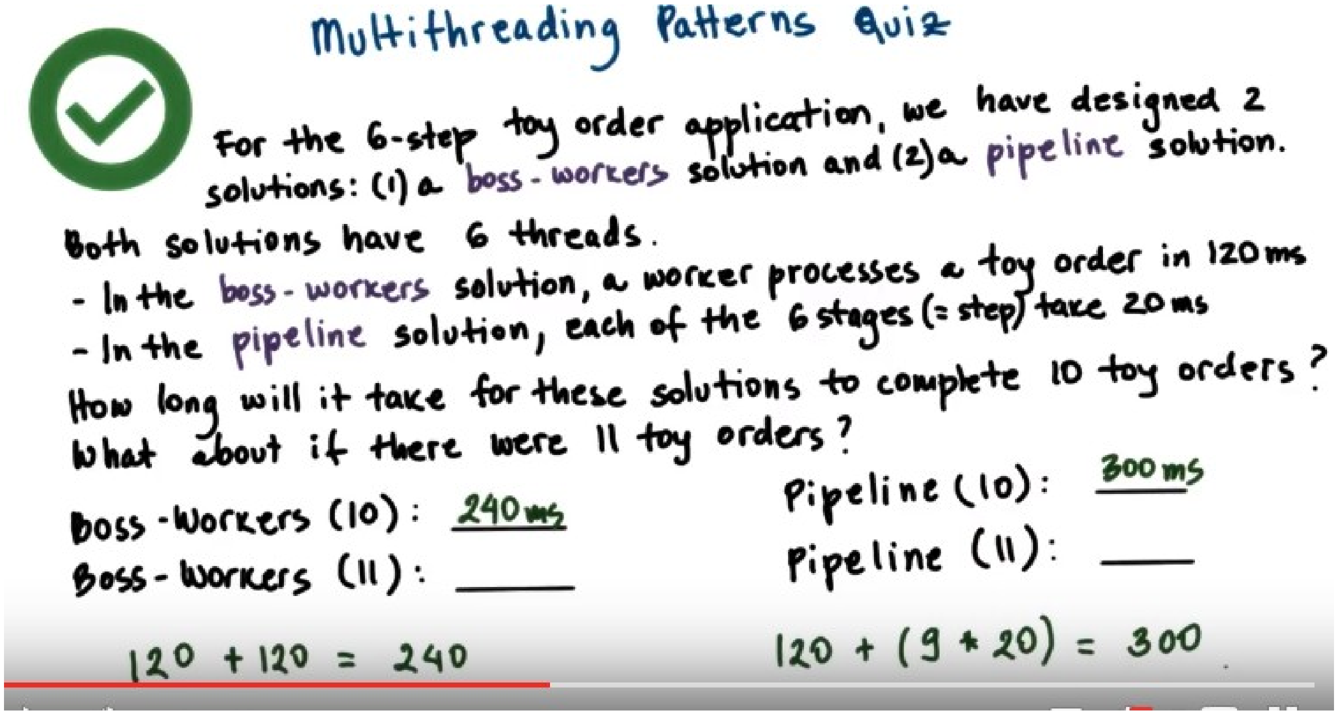


Layered Pattern
Layered :
- each layer group of related subtasks
- end-to-end task must pass up and down through all layers
step4
steps 3,5
steps 1,2,6
Decorating
Assembling
Processing
+ specialization
+ less fine-grained than pipeline
- not suitable for all applications
- synchronization
1:24 / 1:40

43. Another multithreading pattern is what we call a layered pattern. Let's return to the toy shop example. Steps one, two and six, all of them deal with order processing, accept the order, parse the order, ship the order. Steps three and five deal with cutting and assembling the wooden pieces for the toy. Step four deals with decorating or painting the toy. A layered model is one in which each layer is assigned a group of related tasks, and the threads that are assigned to a layer can perform any one of the subtasks that correspond to it. End to end, though, a task must pass up and down through all the layers. So, unlike in the pipeline pattern, we must be able to go in both directions across the stages. The benefit of the approach is that it provides for specialization and locality, just like what we saw in the pipeline approach. But it's less fine-grained than the pipeline, so it may become a little bit easier to decide, how many threads should you allocate per layer? The downsides are that this approach, this pattern, may not be suitable for all applications since, in some cases, it may not make sense for the first and the last step in the processing to be grouped together, to be assigned to the same thread. You may not be able to get any benefits from specialization in that case. The other potential issue with the approach is that a synchronization it requires is a little more complex than what we saw before, since every single layer needs to coordinate with both the layers above and below, to both receive inputs, as well as pass results.

44. Let's take a quiz now in which we will compare the performance of some of the multithreading patterns that we saw. For this quiz we will look at an extremely simplified performance calculation for the toy order application. And we will compare two solutions. One, which is implemented via the boss-workers pattern. And then the second one that's implemented via the pipeline pattern. For both solutions, we will use six threads. We'll assume, also, that in the boss-workers solution, a worker takes 120 milliseconds to process a toy. For the pipeline solution, we will assume that each of the six stages, where a stage is a step from this application, take 20 milliseconds. The question then is, how long will

it take for each of these solutions to complete ten toy orders? You should ignore any time that's spent waiting in the shared queues in order to pass orders from the boss to the workers or across the pipeline stages. And assume infinite processing resources, like tools or work areas. Then, you should answer, what if there were 11 toy orders? How long will it take each of these solutions to process the 11 orders? You should write your answers below, expressed in milliseconds.



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 120 ms
- In the pipeline solution, each of the 6 stages (= step) take 20 ms

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

Boss-Workers (10): 240 ms

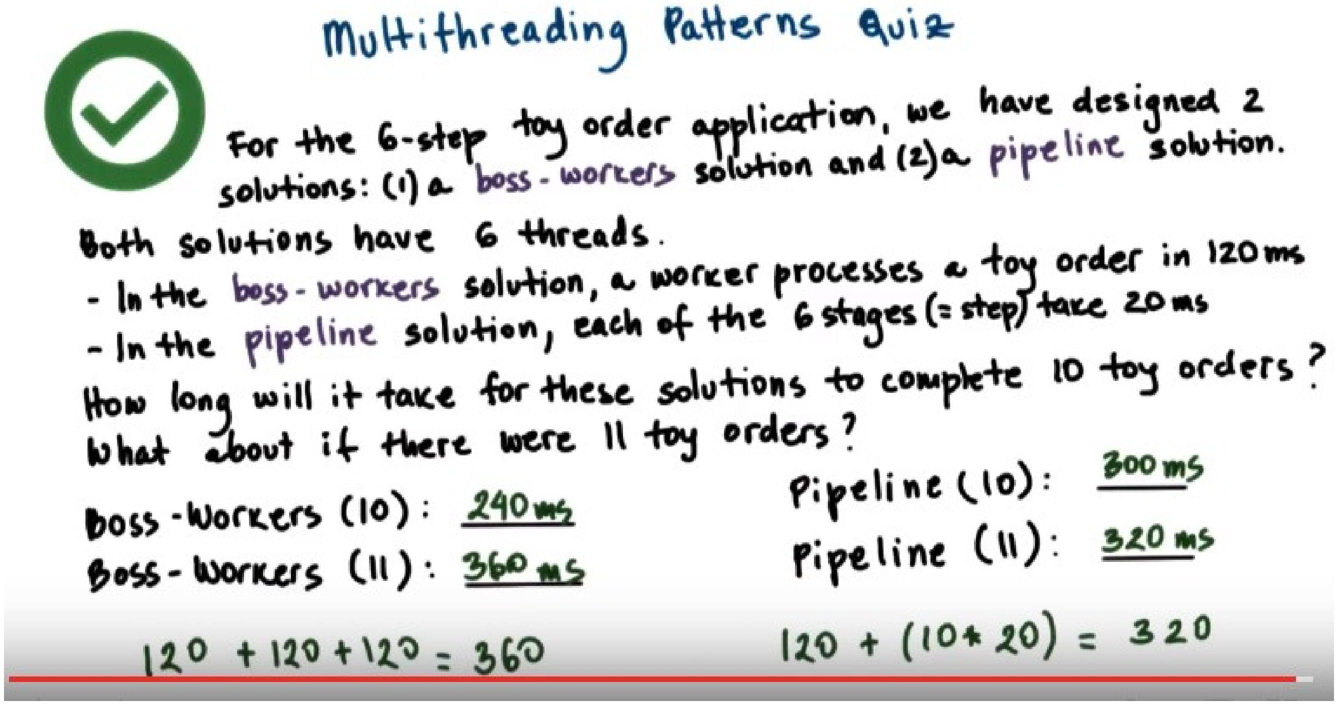
Boss-Workers (11): ______

Pipeline (10): 300 ms

Pipeline (11): ______

120 + 120 = 240

120 + (9 * 20) = 300



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 120 ms
- In the pipeline solution, each of the 6 stages (= step) take 20 ms

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

Boss-Workers (10): 240 ms

Boss-Workers (11): 360 ms

Pipeline (10): 300 ms

Pipeline (11): 320 ms

120 + 120 + 120 = 360

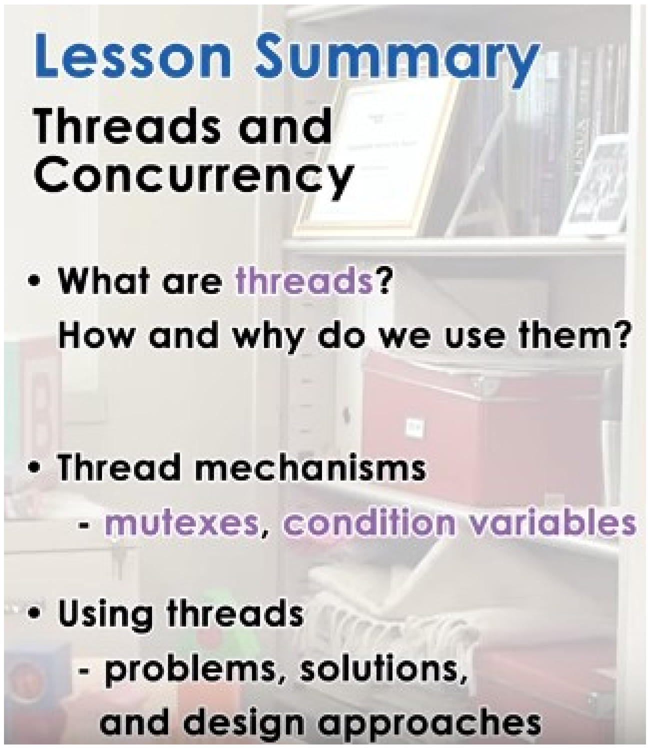
120 + (10 * 20) = 320

From Undacity Instructor Notes:

Boss-Worker Forumla:
time_to_finish_1_order * ceiling (num_orders / num_concurrent_threads)

Pipeline Forumla:
time_to_finish_first_order + (remaining_orders * time_to_finish_last_stage)

45. 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. (for the boss-workers case) 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.



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

46. In this lesson, we talked about threads, how operating systems represent threads, how threads differ from processes, and why they're useful in the first place. We spent some time talking about several mechanisms related to multithreading, and in particular, about mutexes and condition variables which are needed for synchronization. We also spent some time, at the end, talking about certain challenges, solutions, and design approaches that are related to threads and multithreading.

47. As the final quiz, please tell us what you learned in this lesson. Also, we'd love to hear your feedback on how we might improve this lesson in the future.