Week 12 Module 23 Processes and Threads

Introduction 1.2

In this module, we're going to be talking about processes and threads and what is needed for an operating system to manage the processes and the threads. We're going to take a look at some concurrency issues. We're going to take a look at some deadlocks. And we're going to talk about how to take care of all those issues.

What is a Process? 1.3

So, the process is made up of the code, data and context. And we're talking about the process being a running program in a system state, and we're gonna talk about what system states mean, in detail, in the next topic. But for now, we'll simply say that when the program starts, it has to have all the information that it needs in order to run; specifically, the code as well as any data it may need initially. And then the operating system needs to keep track of a lot of information about the running program, and that's what we're gonna call that the context. The code all has to be run in sequential memory; you know this, you've done some programming. You've done quite a bit of programming and you're gonna have to recognize that the code starts from point one and proceeds sequentially through the entire program. And so that code needs to be loaded into sequential memory. The process is a way for the operating system to keep track of the state of the running program, and the resources assigned to that running program.

5 State Process Model 1.4

We'd like to talk about the five state process model, and we're going to be talking about each individual state. Now what is a state? Well a state is one condition that a program might be in and the process is going to spend quite a significant amount of time in each state. So each state has to be recognized and has to be recorded by the operating system. So let's go through the five different states that a process could be in, and you'll have a better understanding of why the process is in each of these states.

So, the first state is the new state and all processes start out in a new state. Now we have to recognize that the process is going to spend quite a significant amount of time in each of these states. So, why would a process spend a significant amount of time in the new state. We have to think about the process being created and the procedure that we would go through to create a process inside the operating system. And the new state is when that process is being executed by the operating system. Now one thing that we need to recognize is that a state is an atomic element. And if the operating system can complete everything atomically, then there's really no reason to have a state. So, the new state happens because the process is being created and the code for that process, one of the reasons is the code for that process will have to be loaded from a secondary storage device into main memory. And that's an IO operation and I operate generally take an enormous amount of time, and we'll see this in a later model, but for now we just say that an IO operation takes a really long time. And the operating system doesn't want to sit in wait for the IO operation to complete; it wants to go on to do other things. But it needs to keep track of where this process is in the act of creation and the way to do that is the new state. So all processes in the new state are in the act of being created and their code being loaded into main memory.

Once the process finishes loading and can be run, then the process moves not to the running state, unfortunately, but to the ready state. And the ready state is that point where the operating system has all the processes which actually have all their parts ready to go. These are only waiting for a processor to

become available so that we can load the process onto the processor and actually let it run. Remember from our hardware discussion, that the processor is the only location where code can actually be run in the system. So, the processes that are on the ready state have everything that they need in order to run but they're not quite running yet because there's no processor available.

Once the processor becomes available, the process is going to move from the ready state into the running state and then it will actually be running on the processor and doing some real work. This is called a dispatch operation. Once the processor either times out, or the process makes a blocking system call, or the process ends altogether, the process is going to stay in the running state until one of those three things happens and then it's going to move to the ready state, the block state, or the exit state accordingly. Now we already understand what the ready state is we've seen that just a second ago. But what's the block state?

Well what happens if the process makes some sort of request which is going to take a really significant amount of time. The operating system doesn't want to stand and wait, waiting for that process to do what it's asking to do. And so it moves the process into the block state to indicate that this process is no longer ready to run; it is not running but it's not quite done yet. And the process is going to wait in the block state until whatever action it's requested finishes, and then it's going to move from the block state instead back into the ready state.

And then eventually from the ready state to the running state and from the running state back to the ready state when it times out. And this process will happen from ready to running to blocked, ready to running, to ready to running, and so on and so forth until the process finally finishes. And when it's done the process moves to the exit state. And in the exit state the process is just waiting to be cleaned up and terminated. And there's one key element that has to be returned back to the parent process, which we'll talk about later and linkages, and that is the return value. Remember return zero from int main, well that has to be given back to the parent process. And a process that in the exit state has not yet returned back its element, its return value, to the parent process and that's why we have the five states that we have.

Suspension 1.5

We also add to the concept of the states with the idea that the process may be suspended. Now suspension is a bit of an odd notion in that the process is sort of still ready to run but it isn't actually running or runnable yet. In suspension, the process will be completely removed from main memory; so, obviously its code can't run because this code isn't loaded in main memory. So, in suspension we completely take the process out of main memory and move it to a secondary storage device or even maybe a tertiary storage device, if we really don't have any memory. Why would we have this?

Well one of the biggest reasons for having a concept of suspension is that we can free up main memory. If we get to the point where main memory is starting to fill up, with so many processes or with such large processes, that we no longer have a lot of processes that are ready to run we might get into a condition where the process list has nothing on the ready state; everything is in the block state and there's no more memory available to allocate to new processes. So, we aim to free up that memory by suspending one or more processes and this is done by what's known as the medium term scheduling algorithm. And the medium term scheduling algorithm will choose which process to suspend, and then it will take that process and put it on the secondary storage device. Now, this doesn't mean that the process is terminated because we can certainly reload all the code data and context from the secondary

storage device back into main memory at a later time and then run that process, just as if nothing ever happened. So it's very possible for any of your programs your 'Hello World' style programs from previous modules, that could have been suspended at some point. We might have suspended the process just simply for debugging purposes; we might have suspended the proper process at the request of the user doing a control Z on a Unix system.

But the point is that we need to add States to our five state process model to recognize the possibility that the process is suspended. And we actually need to add two different suspended states, because we need to recognize if a process is suspended and ready to run or if it's suspended and blocked for some reason. And the movement into and out of the suspended states is relatively easy. If the process was in the block state than during suspension it would be moved to the block suspended state, and if it was in the ready state during suspension it would be moved to the ready suspended state. If it some point in time in the block suspended state, the event that we were waiting for occurs then the process would be moved from the block suspended state to the ready suspended state. So, we evolve our concept of the process model from a five state process model into a seven state process model, where we add these two suspended states.

Process Image – The PCB 1.6

So, the operating system is going to need to record quite a bit of information about the process and in total the process is going to be stored in what we know as a process image. The operating system will keep a table known as the process table with pointers to the various process images for those processes that are running inside the system. And one of the components that are in the process image is what's known as the PCB, or in some cases you might consider the PCB to be the actual process image. It includes all the information that the operating system needs in order to run and manage the process. So, the PCB is a really critical component of the operating system; it's usually implemented as a simple struct with quite a bit of information.

So, what's in there? Well we've got memory tables; we've got what's allocated to the process. We've got what secondary storage is allocated to the process. We've got IO tables, which are what devices the process has access to and where it is inside those devices. We've got file tables with all the file handles, so anytime we opened a file in our previous programs that we create an entry in the file table and that was stored inside the PCB in the process image. And then we've got stack pointers, and all of these components make up the process image, which includes all the code data and of course this is the context.

Contents of a PCB 1.7

So, if we take a deeper look into the contents of the PCB, what we're going to see is a lot of the context information for the process. We're going to see things like the numeric identifiers. The numeric identifiers that we have just a few examples of, are the process identifier. Now that's one process identifier is assigned to each process in the system and this is a simple word, it's only sixteen bytes, sorry… sixteen bits large and the PID has to be unique for all the processes. So, you're starting to come to the idea that the maximum number of processes in the system is 65,535 because that's the maximum number of process identifiers that we have.

We're also going to keep track of the parent process identifier. Now each process is going to be created by a parent, and the parent has certain responsibilities and rights over that child process. The parent can terminate the child. The parent can receive the child's information when the child finishes; the return

zero value so to speak. So, the parent process identifier needs to be recorded as part of the information that we're going to use to manage that process.

We're going to record the user identifier, of the person who started that process. The UID is important because it leads to the security perimeter restrictions that are placed onto that program. What can this program do is based on who activated this process. And we're going to talk about registers, inside the PCB; we have a number of registers, now those are for when the process is either in the blocked or the ready state. When it's in the running state remember the process is actually running, so the registers are constantly changing. But those registers have to be recorded every time we pull the process out of the running state and they need to be reloaded back into the CPU whenever we restore the process to the running state.

We're going to have the stack pointers. So, any stack pointers that this process might have in this case is probably only going to be one stack pointer later on, we're going to see where there's a possibility of having more. Scheduling: so how long has this process been on the CPU, how much time has it spent, how long has it been waiting, when should reschedule it to wait, and so on and so forth. Any linkages: if this process is linked to another process in terms of a pipe, so that the output of this process goes to the input of another process, this has to be managed via the PCB. Any into process communication systems that might be enabled inside this process, so any communications that the process is having with other processes in the system. And the resources in use, so IO devices or files and how much memory is being used. That makes up the contents of the entire PCB. and that's what the operating system really uses to manage the process. The PCB. might be moved from state to state, or the PCB. might coincide with the process image, but either way we're going to keep all this information about the process in order to manage it.

Modes 1.8

One of the other components that the operating system needs to keep track of is what mode a process might be in. And what we're really talking about here is what processor mode the process could be using. For the most part processes are going to use user mode, and that's normal, but in certain instances a process may have to be created that has more permissions, to access more parts of the system then a user mode would be allowed to access. User mode has a lot of restrictions; it's not allowed to directly access system hardware, it's not allowed to access main memory outside of its own bounds. So, the operating system keeps track of what processes are in user mode, and what processes are in kernel mode and there are very few that are in kernel mode. Now let's be clear kernel mode has absolutely no restrictions; you can do absolutely anything that the system can do when in kernel mode and kernel mode should really be restricted to only those processes that need kernel mode. In years past kernel mode was allowed for almost any process that was running on the system, and if you survived or use computers through the late 90's and early 2000's you know how dangerous that was. The era of Windows 98 and Windows ME constantly crashing and blue screening and the power switch solution to that was partly a result of having every process access to kernel mode.

Now we have a much better solution: we use user mode only for those for those processes that are going to not need kernel mode. And the processor knows the difference between user mode in kernel mode as a result of one register. In the Intel architecture, we call this the program status word register the PSW, and that one register has one bit to indicate whether it's in user mode or in kernel mode. Switching between the modes is easy if you're going from kernel mode to user mode because you simply modify the register for the program status word and it changes into user mode from kernel

mode. But switching back from user mode to kernel mode is a bit more difficult. For this the processor, itself needs to recognize when the operating system is going to run and it needs to move back into kernel mode automatically, by itself. And we'll talk about when this happens there are certain events that occur that cause the processor to move back into kernel mode and when that happens, of course, the processor is going to switch to running the operating system.

Process Switching 1.9

So, a process that's in the running state will stay in the running state until something happens, and we need to understand what happens to cause the process to come out of the running state, and how that might affect user mode and kernel mode. Well first off, a process switch is going to occur during one of the events either an interrupt a trap or a blocking system call.

During an interrupt, some piece of hardware has indicated that it needs immediate servicing, and so the operating system must be invoked in order to take care of that hardware. Now the hardware could be any piece, we can think of a wireless device; a wireless access card where it has a small amount of memory and the buffer is starting to fill up. If the buffer actually gets full then any new data that's coming in through the wireless card is going to be lost, because the buffer is full. So, the solution to this is for the network card to send an interrupt to the CPU. And what happens in an interrupt is the CPU will finish executing the instruction that it's running, and then it will switch to running the operating system. And the operating system has a specific point that it uses, it's called the Interrupt Service Routine, the ISR. The ISR runs whenever an interrupt occurs; the processor knows to switch to the operating system's ISR, and of course since it's switching from a running program to the kernel; it switches from user mode to kernel mode and it runs the ISR.

We could also have a trap; now this is a situation where the process has done something that causes it to need the operating system, even though it doesn't know it needs the operating system. And we can also have a blocking system call; these are obvious like the program is trying to open up a file. If the program is trying to open up a file, it's going to need to invoke components of the operating system and so the responsibility switches. The process will switch to running the operating system and when it does that the operating system will take care of opening up the file and reading in the data. Now a process which involves quite a lot of steps, most importantly, we need to save the context of that process. We're gonna save that in the PCB. And then we need to move that PCB into the appropriate queue whether it's the blocked queue or the ready queue to indicate what we're going to next do with this process, and then we need to service whatever was requested; deal with the interrupt, deal with the blocking system call or deal with the trap. And then choose another process that's ready to run so we're going to look to the ready queue and choose one of the processes on the ready queue to move into the running state. And then we're going to actually run that new process by restoring its context. This gets a little bit more complex if we talk about multi processors, and the idea of multiprocessor we might have more than one process on the running state at any given time. and so that's going to have some effect on performance.

Threads 1.10

So now that we've got the idea of a process down, let's make it a little bit more complex. Because while a simple program might be running one cohesive execution cycle, or execution unit, the idea of having a large program run as only one execution thread just doesn't make sense anymore. So, it was invented and we now use this concept known as a thread. Now we recognize that there are still going to be processes in the system, but the idea, the definition of a process, will doubt change to mean a unit of

resource ownership. The idea of a process now no longer means an execution path, we're now talking about a unit of resource ownership.

So, all the files that are open inside the process, all the memory that's owned by the process, all the IO devices for the process, all of that that can be owned is owned by the process. But there's no real execution happening in the process, and what I mean by that is the execution is offloaded to a concept known as a thread. So, the process becomes the unit of resource ownership but the thread becomes the unit of execution. And we might have multiple threads that run inside a single process and each thread has its own code. Although the code is memory and the memory is owned by the process. So, there might be multiple threads that are running the same code, or the threads might be running different code. All the threads inside a process share all the resources of the process and that includes all the memory, so threads can share the same memory unlike processes; processes can't share memory they're restricted because they're in user mode. But threads all exist inside of a process and the operating system is going to recognize that multiple threads are running inside a process, and it's going to be able to execute any one of the individual threads inside the process.

What is Where in the Multithreaded Environment? 1.11

So, if we now modify the definition of a process, we need to change where things are stored inside that process. So if we take a look, the PCB still has most of the stuff that we would recognize: the memory allocation, all the memory and IO tables, all the files all the linkages, all that stuff. But the thread now needs to keep track of some of the thread specific things like the context, the stack, the variables that are specific to that thread. All of that needs to be offloaded outside of the PCB and inside maybe a TCB, a thread control block, to keep track of each of the individual threads. Now the fun part about threads is that they all have access to the resources, the memory and all the rest of the stuff, from the process, and that means that we have a great level of communication between all the threads inside of a process. But the process needs to recognize, or the threads really need to recognize that they exist inside of a process and there are some concurrency issues that come up in relation to that. Now what about your old programs, like your 'Hello World' programs that you've written for previous modules, do they have threads? Well they did; they had one thread, the main thread and that main thread could have, although it didn't create other threads to do other tasks each individual task inside the system can be or inside the process rather, can be handled by one thread and each of the threads do their own sort of work. But they all work together on one project one process and they all share the materials of that process.

Reasons for Multithreading 1.12

I'd like to take a look at some reasons why we would use multithreading, and some of these are rather obvious and you're to see them in in your regular day to day use of your computer. Sometimes you might have foreground and background activity; the perfect example I love to use is Microsoft Word, because it's one that we've all used a million times. And Microsoft Word is broken down into a really large number of threads, but just the few that we might pay attention to are that we might have a thread that's taking in input from the keyboard, and it takes in whatever the user types and it stores it in main memory in what we would call a document. And there's another thread that takes the that main memory from the document and puts it onto the screen; print it in the appropriate way, formats it, makes it look nice based on the font. And then there might be another thread that goes through and checks the document and underlines everything in green or red according to whether or not it's spelled correctly. And then another thread that goes through and checks the grammar. And then we might have another thread that sits in the background and waits for ten minutes, and when that ten minute timer is

up that thread will save the document in some temporary location just in case your computer loses power. So, there's a lot of foreground and background activity that's happening inside something like Microsoft Word.

We might also look at Microsoft Excel, because it's another one that we've used quite frequently. And the act of changing one cell inside Microsoft Excel can cause a cascading change throughout the entire spreadsheet when that one cell changes there's threads that go out and change all the other threads that are related to, all the other cells, that are related to that one cell. So, then again there's a lot of foreground background activity.

There's asynchronous in synchronous processing. So, if processing has to be done or if processing can be done synchronously we can create multiple threads to deal with the synchronous processing. So, if we have a very large data set and we'd like to process portions of it separately, we can process those portions separately using multiple threads. And a block that happens in one thread will not obscure the other thread from running. We might also have asynchronous processing, where the threads need to do some sort of work cohesively so that when one thread ends the next thread begins. We might have infrequent tasks, where one thread doesn't run very often; as my example of Microsoft Word and the saving every ten minutes is an example of infrequent tasks.

Speed reading, here we have a problem where each thread that does a read operation on the file system is going to be blocked because that's going to take a very long time, so the operating system is going to block that thread. Now the interesting part about threads is that blocking one thread, doesn't block the rest of the threads in the same process. So, we can have one thread whose responsibility it is to read all the data from the file system and bring it into main memory, and another thread whose responsibility it is to process that data, and this is called Speed Reading. So, we're pausing only to deal with the IO operation but at the same time when the IO operation is causing us to pause, we're continuing to process data that's already been brought in. And then it also creates a more modular program structure where we can break down a larger program into smaller portions and we can deal with each of the portions independently, as if they were basically their own program. But they're still working on one goal which is to deal with this memory or to deal with this process as a whole.

Performance Example 1.13

Here we have an example of a file server. and the file server does some sort of an IO operation for eighty percent of its time. So, what this is we'll take in a request for some file, and then we'll process that request, whatever that means, and then we'll go off and receive the file, recover the file from them from the secondary storage device, and return it to the requester. So, you might think of this is like an old 1990's style HTTP web server so we take the request, we process the request; that's going to take let's say about two milliseconds. And then we'll deal with the IO operation of actually recovering the file from a secondary storage device and returning it to the to the requester, which maybe takes eight milliseconds. So, in total the processing time takes ten milliseconds. And that's just a perfect example; we've got an IO operation that takes eighty percent.

If we do this without threats, we do this with just one process that means we've peaked out, we've maximized, our utilization at one hundred transactions per second; each transaction is going to take us ten milliseconds and of course there's a thousand milliseconds in a second, so thousand divided by ten means we can get one hundred of these transactions in a single second. But let's take a look if we add in the idea of threads, and let's create a thread for each transaction. When we do that, we can now

synchronously process the CPU and the IO operation. In other words, if we created two threads, then the first thread's CPU could be done while the second thread was doing an IO operation. And then the second thread could do its CPU time while the first thread is doing the IO operation. So, we can sort of go back and forth using the CPU as much as we can and using the IO completely. Now if we do this, we recognize that the IO can't be done synchronously; we have to we have to do that asynchronously. We have to do that so that one IO operation is completed and then the next IO operation can start.

The IO device can run multiple requests at the same time, so we can service multiple requests at the same time, that means we have to do it asynchronously. Well what that means is that we have these little eight second eight millisecond blocks that were chopping up this one thousand millisecond second into these eight millisecond blocks, and that means we can use only one hundred twenty-five transactions per second. Now that doesn't mean we're not doing the CPU time, of course we're still doing that two milliseconds of CPU time, but it means that we're doing it while the IO operation is happening for one of the other threads for the other thread. So, ultimately we've increased our capability by twenty-five percent, because what we've effectively done is hidden the CPU time behind the IO delay, and that works out great.

But we can actually do better because in this example of a web server, it's pretty obvious that a lot of the requests are going to come in to the exact same file; they're going to be for the exact same file. And if we're dealing with each request independently, then the CPU time and the IO delay don't result in any cache; they don't result in anything being retained for the next request. But if we can keep track of each of the requests and when we bring in a file, we store that file in main memory, then in the next request we don't have to do the IO operation; we can avoid doing that eight milliseconds, that expensive IO operation, we can completely avoid that by recovering the file directly from the cache. So, this is one of the benefits of threads that were exploiting by having a shared memory space, which you can't do with processes; a shared memory space where we can store these files temporarily. Now at some point if we don't ever need the file, we have to throw it out of the cache and if we use up too much memory, we may have to throw away some of the files that aren't being used very often. So, there's an extra bit of processing, but even if we increase the processing time by twenty-five percent and increase it then say, do two and a half milliseconds of CPU time, the benefit here is that we're decreasing the amount of IO delay.

Now, let's say for example we get seventy-five percent hit rate on the cache; now that's abnormally high granted, but it's possible, if we have a web server that's popularly serving one web page. When that happens, we recognize that the IO delay drops to zero for those seventy-five percent of the operations of the transactions, the IO delay will drop to zero. Now the twenty five percent that remains we're still going to have to do that eight milliseconds of IO delay, and unfortunately now we've wasted an additional half a millisecond for the CPU time to sort of look into the cache and realize that it's not even there. So, this extends that, but it reduces the overall time because now on average we're going to have two and a half milliseconds of IO delay for every transaction but then, correction two and a half milliseconds of CPU delay for every transaction, but the IO delay is reduced to only two milliseconds for every transaction. And it's not really two milliseconds for every transaction, it's eight milliseconds for those transactions that we have to do the IO delay, but that's only seventy-five percent of the time. So, we can say that on average each transaction uses two milliseconds of IO delay, so with two and a half milliseconds of CPU and two milliseconds of IO delay, we have four and a half milliseconds total and that means that we can get four hundred transactions in a single second.

So, we've gone from one hundred transactions in a second to now four hundred transactions in a second just by implementing threads and creating a cache and getting a pretty good cache hit rate, but it's possible. If we want to put some money into the problem, maybe we can add a second processor and set up symmetric multiprocessing so that that CPU delay can be done concurrently, in which case the only slowdown here is the IO operation. We're IO bound, meaning that we still have those two milliseconds of IO delay that we can't get rid of, and that two milliseconds means that we're kind of limited at five hundred transactions per second. But with a very small investment, we can go from one hundred transactions per second to five hundred transactions per second, and with no investment at all we can go from one hundred to four hundred transactions per second just by implementing threads.

Thread States/Operation 1.14

So, if we take a look at the concept of a thread now, we recognize that the thread really only has three states. It has ready, it has running, and it has blocked, and all of those makes sense in the same concept that we talked about them in the process level notion. A thread can be ready, meaning it has everything it needs to in order to run it's just waiting for a processor. It has running, meaning it's actually on the processor and doing some work, and it has block, meaning it's waiting for something to happen.

Threads don't really have a new and an exit state, because we don't need to load code. For example, our old example of loading the code and that was why we needed the new state, we don't have that anymore because the thread already has all of its code loaded in main memory. We're just creating a new thread and how does a new thread get created? Well the operating system creates a TCB to recognize this thread, initializes the linkages initializes the TCB, and then it starts to run. So, it's really an atomic element creating a new thread can be done atomically, and the thread just begins running inside the context of the process. So, we don't need a new state and likewise, we don't even exit state because when a thread completes there's nothing returned back to the running process; the thread just records that it's done.

Now obviously, if it's the last thread or the main thread, if you would, then that needs to be returned back so we still have the concept of new and exit state in the idea of a process, but in the idea of a thread the new and the exit state aren't really there anymore. Likewise, the idea of suspension is really a process level concept; if we've removed all of the main memory the none of the threads are able to run. So in suspension, we have to recognize that all of the threads have been blocked or all the threads have been stopped altogether. But at the same time, there's no real recognition of the suspended state inside of a thread; that's more of a process level concept.

What are the Downsides? 1.15

So we've talked about the upsides of threads and we certainly understand how threads can be great, but of course there's got to be some sort of a downside. And what's the downside? Well the biggest one is that all the threads have access to the same memory of the process; well that was an upside too, wasn't it?

The biggest issue here is that we have concurrency problems; that if all the threads are trying to access the same memory that of the process, then it is possible that the threads could interfere with one another, unintentionally. And there's a lot of examples of this and we'll go over some of the concurrency examples in the next module, but ultimately what we're looking at is that whenever a thread is accessing memory or it really any resource that's common to the process, it needs to do so very carefully and it needs to keep track of where it is inside the execution cycle. Usually what we do with

this is some sort of a mutual exclusion lock, or a some or for to indicate the thread is in execution and maybe has been stopped.

Now why was it stopped? Well could be any of the reasons that we talked about for a context switch. So, any time we could switch a process, we could also switch a thread; so, from one running thread to the other. And unfortunately, if the thread is making some changes to the memory, it might not be complete with all of the changes; it might do only half of the changes leaving that memory in a really inconsistent state, and that's one of the concepts that we know of as concurrency. This is what we call concurrency.

The other problem of course is deadlock, but deadlock can happen as well in processes and we will talk about deadlocks in a later module. The other problem that we have with threads is that programmers really like them, so they create a lot of threads. And if we create too many threads we lose track, or it's very easy to lose track I should say, of what each thread is doing and that causes a lot of confusion. And two threads might interact with each other in a way that we didn't really intend because we're not being careful with how many threads we're creating.

Implementation of Threads 1.16

So, there's a lot of ways to implement threads and a lot of these are leftovers from an older era where the operating systems didn't actually support the threads, but programmers wanted to use threaded concepts inside their programs. So, the first way to implement a thread is what we know as a kernel level thread. Now this is very different from a kernel thread, and this is very different from kernel mode. This is the concept where the operating system recognizes that a thread has been created and the operating system will create a TCB for that thread and the operating system can choose to run that thread or can choose to block that thread and all the states that go along with that thread.

The other main way of doing this is what we know is user level thread, and this is really a leftover of a bygone era although that still very much in use today. The biggest downside of a user level thread, actually the way that a user level thread works is that the operating system doesn't recognize that the threads are in use. So, it sees this thread as a process, let's say, so let's take for example an operating system that doesn't support threads. Well how can we run threads? We can have a library of code inside of our program, which creates or simulates creating of threads and switching between these threads. So, what would be required would be some scheduling algorithms and some ability to create new threads and destroy threads, inside a user's program. But all of this is a simulation and the operating system doesn't really know that multiple threads, if you want to call them that, are being created here. Instead what it does is recognize that one process is being created. And the biggest downside to that is that if any of those threads, and I use quotations, those threads cause a blocking system call then all of those threads are going to be blocked because the OP running system just sees it as a single process. So, that's the idea behind a user level thread. The advantage of a user level thread is that you can choose to do whatever scheduling algorithm you want; you're not at the behest of the operating system. in a kernel level thread the operating system decides when your threads are going to run, but in a user level thread you, the programmer, decide when that thread is going to run and how often it's going to be run. So, you could have three threads and give priority to one of the three threads to run more frequently than the other two, and you can't do that with kernel level threads easily.

The hybrid approach takes benefits from each. So, the hybrid approach, which was first implemented by an operating system known as Solaris or best implemented by an operating system the Solaris, what

they did was they created both kernel level threads and they created a user level thread. And they tied the two together by means of what was known as a lightweight process, and the lightweight process is simply a container. So, we can put a number of user level threads inside of a lightweight process and we can choose to run that lightweight process on one kernel level thread, so when a kernel of a thread becomes available the lightweight process runs. And all of the user level threads in that process are run according to the scheduling library of that lightweight process. So, you as the programmer would create a lightweight process; you'd creates a number of user level threads. You'd assign the user level threads that you wanted to run to the lightweight process, but you can create any combination of user level threads and lightweight processes. So, if you had a very important user level thread, you would create one lightweight process for that one user level thread and any time that lightweight process ran that one user level thread would be the only one running. Then you could take three other user level threads, that are not critical, and put them on one lightweight process. So that when that lightweight process runs any of the user level threads inside could run. Of course, the same downside occurs as with user level threads, if one of the user level threads in the lightweight process blocks; all of the threads in the lightweight process will be blocked. As far as thread scheduling is concerned, there's both a global scheduling algorithm that chooses to run the kernel level threads and then there's a local scheduling algorithm, which you can implement you can manage yourself as the programmer, that would choose which of the user level threads associated with that lightweight process would be chosen to run.