**2025-04-24-os3-en**

0:09  
So last week, if I'm not mistaken, we ended up here and we talked about devices and interrupts and we explained how interrupt works to notify the CPU that something has happened in the device, that the data transfer is complete, that somebody, somebody pressed a key on a keyboard or whatever the device needs to report to the operating system, to the CPU.

0:34  
So let's end the part regarding devices.

0:37  
Yeah, that's very important and also very bulky part.

0:40  
Unfortunately, we didn't have time for more details and let's switch for processing.

0:45  
That's another very important topic in the operating systems before we started a few terms just to clarify the details.

0:53  
So first of all you already have some intuitive notion about what programme is, so let let me formalize it a bit.

1:02  
A program is basically a collection of instructions and data, usually written encoded into one file or a bunch of files depending on the operating system.

1:14  
On most operating systems like Linux or Windows it usually is and can be 1 file.

1:21  
Some other systems like Mac OS prefer to use a bunch of files, so they they basically say that one application is 1 directory.

1:31  
But that's just a technical detail.

1:34  
It's always encoded in some in in file or multiple files on the on the hard drive.

1:40  
And usually this is a production of some compiler and linker.

1:43  
So you you write your program in AC or whatever language you prefer, then you compile it and the compiler yields one or multiple executable files.

1:55  
The program just sits there.

1:57  
So the program does nothing until you start it.

1:59  
And when you start a program and you execute a program, it becomes a process.

2:05  
So basically process is something like a running program.

2:09  
Technically process is much more because process is an entity of operating system.

2:13  
So it has some additional attributes, it has some, it has some additional resources.

2:19  
Most importantly, it has its own address space.

2:22  
It has its own memory where the data and the instructions from the file are loaded before you can execute them.

2:29  
Obviously, because you cannot execute code on the hard drive, you need to load the code into the main memory so the CPU can execute the instructions.

2:38  
So basically the program becomes live when you create a process out of it and it becomes live by an activity.

2:45  
And by this activity, of course mean that the CPU starts executing the instructions which were previously stored in the file.

2:54  
Yeah, this step isn't that easy as it sounds, because there are many tiny steps.

2:59  
For instance, if you have dynamically linked libraries, remember dynamically linked libraries from two or three weeks later before.

3:07  
So yeah, you need to make sure of the operating system, the operating system loader needs to make make sure that all the libraries or the dynamically linked libraries are present in memory as well, so they can be interlinked with this executable.

3:22  
There are some relocations, there are some tiny fixes in the code that needs to be done after it's loaded into the memory.

3:30  
So yeah, this needs to be handled before the program can be executed.

3:35  
But these details were covered in the runtimes, at least the most important details.

3:41  
Within a process there should be at least one thread.

3:45  
And the thread is like a the stream of the stream of instructions being executed.

3:49  
That's the activity which is being performed by the CPU.

3:55  
The thread is also the basic unit of kernel scheduling.

3:59  
So the kernel, when it when it handles multiple processes, multiple programs running simultaneously within the same operating system on the same host, it schedules, not processes, but it usually schedules threads.

4:12  
Obviously there may be multiple threats within one process.

4:16  
That's why we didn't make this distinction.

4:18  
Yeah, normally or usually, there is at least one threat within within a process.

4:26  
If you'd have no threat within a process, then this no long, this is no longer valid.

4:31  
Nothing is running in the process, then the process isn't alive.

4:35  
So it's not a process.

4:37  
And sometimes we can make a even finer distinction about what can be scheduled, what can be executed.

4:43  
We can talk about fibers or possibly tasks.

4:46  
Fiber is like a lightweight threat, but this is usually not implemented within the operating system.

4:53  
Fibers are usually implemented in in libraries and they usually use some a form of cooperative scheduling.

4:59  
So usually fibers are like a lightweight abstraction above threats.

5:03  
But some some libraries or some operating systems don't even recognize fibres.

5:08  
So this is just another term that you might need to remember just to get you the full picture.

5:14  
But we won't get into the fibres in the terms of operating systems because usually fibres are implemented outside operating systems in libraries.

5:23  
What's the most important part is remember how the data are laid out in memory when the process is started.

5:31  
So basically the code and the static data are loaded or initialized when the process is start.

5:38  
This is the part that is loaded from the file when the process is executed.

5:42  
Some part of this might be also loaded from the file.

5:45  
The other part is just zeroed or initialized.

5:48  
And then we have the stack and heap, which also needs to be initialized when the process starts.

5:53  
And the important part, you, you should remember right away that each thread has its own stack.

5:58  
So if we have multiple threads, there need to be multiple stacks present.

6:03  
And the tricky part is, yeah, if we have a like a huge address space, we can place the code and the static data at the beginning and heap at the bottom and we just say, OK, let let the heap grow upwards and let the stack grow downwards.

6:18  
It's just a convention.

6:19  
We can do it the other way.

6:21  
So in this case, if we have one, one stack and one heap, it's fine because they are growing one against another.

6:27  
And if, if you have like a 64 bit address space, then it's fine because they will never intercept these, these two barriers will never, never touch.

6:38  
Well, yeah, they technically might.

6:40  
But if you have a 64 bit address space, it would take a real effort to to do that.

6:45  
And, and also it would mean that you have some enormous machine with, with memory that is unimaginable because 64 bits is a lot.

6:55  
So normally it's, it's, it's not what happens.

6:57  
But if you have multiple stacks, what might happen is that one stack grows and it hits the other stack.

7:04  
That might happen if these stacks are too close together.

7:08  
So first of all, stacks are usually designed to store small amounts of data.

7:12  
If you, for whatever reasons, place a large amount of data on the stack, so you create a huge local variables, you're not programming very well.

7:21  
That's the rule #1 So usually stacks should be small.

7:25  
Second, if we have a 64 bit address space, we can still place these stacks quite far apart because we have enough room for that.

7:35  
So there might be terabytes in between.

7:37  
So usually this shouldn't happen either.

7:40  
And also it's a good idea to place some cushion some some specific area just right above the the second stack.

7:48  
And this area is like a special marked.

7:51  
So when anybody access this area, it means that the first stack has overflown and the operating system usually shuts down the application.

7:58  
So it's like a memory that shouldn't be touched.

8:01  
Yeah, we can do that.

8:02  
I will explain it later when we talk about virtual memory, but there should be some safeguards which prevents 1 stake hitting the other.

8:10  
So that's that's one thing to remember.

8:12  
Let me take a closer look on the distinction between the process and the threat.

8:16  
So they are they're both important entities of the operating system and the the important things to remember process is mainly the code loaded in memory.

8:26  
So the one process is 1 code.

8:29  
All threads share the same code.

8:31  
Yeah, usually they are executing different parts of this code, but for instance, two different threads can execute the same function within the same code.

8:39  
OK, so they share everything.

8:41  
They also share the global memory.

8:44  
So they have one memory space and all the threads one process, All the threads of the process share the same memory space, which is a bit tricky because if these two threads or multiple threads are running simultaneously, they can also simultaneously update or overwrite the same memory, which is potentially dangerous.

9:04  
You need to make some steps.

9:06  
You need to make some effort to do this properly so there are no mistakes.

9:11  
There are incidental mistakes.

9:13  
And also any other system resources.

9:15  
This, this, this includes mainly the file handlers, also network sockets or whatever network connections your application makes, any operating system primitives for UI synchronization primitives, etcetera.

9:29  
On the other hand, threat is just a position in the code, right?

9:33  
But it's not just the position.

9:34  
The position together with the programme counter of course also includes the CPU state, because if there is a code to be executed, it always alters the CPU state.

9:46  
So the entire CPU state is part of the threat context.

9:51  
If threat is switched to another CPU core, then the CPU state must be saved and then restored on another CPU core whenever threat is rescheduled.

10:01  
And by the CPU state that this mainly concerns the the CPU registers but some other stuff and of as you already know the stack is also property of a thread.

10:12  
So if we have multiple threads, we have multiple stacks.

10:16  
Technically if you just remember where each variable resides, then you can deduce right now easily that each thread executes its code on its own local variables.

10:29  
So within local variables, within function arguments, each thread has its own copy.

10:34  
It's like recursion, right?

10:36  
So basically each thread has its own stack, so all the local variables are unique to that particular thread.

10:42  
On the other hand, global variables or heap variables can be shared, are shared in fact, among all the threads.

10:49  
So we need to make some synchronization efforts if we need to access them.

10:55  
OK, How to create a process?

10:58  
That's the question, right?

10:59  
So it's actually quite easy.

11:01  
I'm not going into the details of the technical stuff.

11:04  
So basically there are operating system calls, There are some procedures, some functions you can call and create a process in Windows.

11:12  
It's not that difficult.

11:13  
It's only tedious because you need to initialize many individual structures.

11:20  
These structures hold some additional properties, some some options, how the process should be created and what the process can do and what, what are the privileges of the process, etcetera, etcetera.

11:30  
But the bottom line is there is one system.

11:33  
This is a function which actually it translates to a system call or even multiple system calls, and this creates the new process.

11:41  
This needs to be executed somewhere.

11:43  
So always one process is creating another process.

11:47  
That's important thing to remember.

11:49  
Actually the same goes in any other system.

11:52  
Always one process creates another process.

11:55  
So basically technically you need one process or multiple processes which are responsible for executing applications.

12:02  
For instance, in Windows you usually have this desktop right at the front face of the windows or the which also implements the the the sidebar or or the bottom bar where the start menu is and you somehow select an icon or something and and double click it.

12:18  
At that very moment, the process that displays this desktop is responsible for finding the appropriate application that corresponds to that icon and execute it using this system call.

12:30  
OK, but it's the process that creates another process.

12:38  
What's also important is that each process yields some sort of a handle.

12:44  
It has some identification.

12:46  
We can shorthand it to PID, like a process ID, but there is some handle of the process and we can use it later, for instance, to wait until the process finishes.

12:58  
So when you execute and one process executes another process, it gets an handle.

13:02  
And this handle can be used for various things.

13:04  
And the one of the most important thing is wait for the process to die.

13:08  
Because usually if you, if you like, execute a script or execute a batch application, you want to know when the computation finishes, right?

13:17  
Yeah, sometimes you execute an application which tends to run for a long time.

13:21  
For instance, you execute a Firefox or Chrome or whatever is your favorite browser.

13:27  
In that case it usually stays on because everybody right uses.

13:32  
You usually have your browser open all the time with 27 different tabs, usually with forgotten stuff, etcetera.

13:40  
So yeah, it's possible that the the process may run for a long time, but normally, normally more usually to let's say our computation perspective or programming perspective, you execute something like solution to your homework.

13:54  
And the solution to your homework tends to load some data, load some input, perform some computation, yield the output to a file or to the standard output or whatever, and then terminate.

14:04  
So it's a good idea for the parent process to wait for the child process to terminate.

14:11  
In Linux, it's very much the same.

14:12  
So you can see there is a similar function, similar procedure to create another process.

14:20  
And also this procedure somehow yields PID.

14:23  
I'm, I'm going into the details later.

14:25  
So you get the PID of the new process.

14:27  
So in the end, you can wait for the new, for the newly created process.

14:31  
That's it.

14:32  
OK, that's the common that are the common part.

14:36  
What's slightly different is that here you are creating a new process as a new entity.

14:42  
The old process is completely intact.

14:46  
It gets the handle in here somewhere.

14:49  
Yeah, this structure is filled with some information about the newly created process and you can get the handle out of the structure, but that's it.

14:57  
This, this is the only communication that happens after the creation in Linux.

15:02  
It's somewhat different or actually this is not a specific of Linux.

15:05  
This is how all Unix systems work, right?

15:09  
So there is a special functions of special operation of the operating system called fork.

15:15  
And fork tends to duplicate processes.

15:18  
So you have one process, the parent process that wants to execute another.

15:22  
So first step of this is that the parent process fork itself, it duplicates itself.

15:30  
Yeah, it's awkward, but it's like how this how this works.

15:34  
It's a similar to let's say cellular division in the Organism.

15:38  
One process is split in two.

15:41  
And the tricky part or the the funny part of fork is the new 2 processes share everything including the memory.

15:51  
So technically this is like a creating a new threat.

15:54  
It looks like creating a new thread, but it really creates a new process.

15:58  
But both processes are running now and they share everything, including the memory, including the stack, including the program counter.

16:08  
So those two processes after the fork just happen to end up here on this line of code.

16:17  
And the only distinction or the only difference is then one of these processes is given the parent process is given the ID of the child process.

16:28  
OK, so basically this value, the P value, which is PID, yeah, if the fork fails it, it returns some error code like -1 or something.

16:40  
And then you need to then the fork didn't work and you're still in one process and you need to handle the error.

16:47  
Unfortunately, if fork fails, then you're doomed, because if the if the operating system cannot perform fork, you're done.

16:55  
You can just reboot the machine or something, because you're screwed everything.

17:00  
If you type a command in your shell and you want to execute some trivial command in your shell, it creates a new process, it calls fork.

17:08  
So if you cannot write a simple program in your shell, the root cannot log in.

17:13  
You cannot fix anything in that machine.

17:15  
The only course of action is to reboot the system.

17:19  
OK, so let's skip the error handling.

17:21  
It's not important.

17:23  
So these two actually these two if else branches are for the two newly created processes.

17:32  
If P is 0 then I know this is the newly created process because zero is a special value.

17:39  
No I no process in the system has has PID zero.

17:43  
The first process created during the bootstrap the init process has PID 1 so no process ever has PID 0.

17:52  
So if I'm if this 4 grits on zero I know it's not valid PID.

17:57  
So this is the, this is the branch of the newly created process.

18:02  
And the exec is a special command that say hey, throw everything, everything away, discard all the memory of me, disclose or close or open files or sockets, everything and glowed a completely new process in my set in my memory space.

18:23  
So this is like an execution, but within existing process.

18:27  
That's the only distinction from the from the windows.

18:30  
If you didn't get an, the else part goes for the original process, so the original process gets a valid PID.

18:37  
So P is not zero.

18:38  
So here it can do something with that valid PID, like wait for the newly created process for instance.

18:44  
If you didn't get the details regarding fork, it's not that important.

18:49  
It's just like a nuance that some operating systems take a different approaches to this, to this creation of new processes.

18:58  
But in the end, the same part is there is some function that basically creates new process.

19:04  
Yeah, I forget to mention one important thing.

19:07  
New process is always tied to a program and program is executable file somewhere in the file system.

19:14  
So yeah, the path, the executable file and possibly some command line arguments or the command line together.

19:21  
This is like these two strings together.

19:23  
Basically is given to the create process.

19:28  
So somehow I need to give the create process or the exec the path to the newly to the program that should be executed.

19:35  
So this is this is the connection between process and program.

19:41  
Yeah, I, I talked about processes, creating processes.

19:43  
So usually we the original process is denoted parent and the newly created process is denoted child.

19:51  
Because we like analogies from real world, right?

19:56  
And the child, at least in the nooks, holds also identifier of its parent.

20:01  
It's a good idea that a child know who his parent is.

20:04  
That's also an analogy from real world, right?

20:08  
A child notifies its parent when he dies.

20:14  
That's a handy tool.

20:15  
As a parent myself, I would very much appreciate this feature in real world.

20:19  
Unfortunately it doesn't work that way.

20:21  
But in the end, what can we do, at least in our world of computer science of of computers, at least in Linux, a child sends a signal to the parent when it dies.

20:36  
Technically speaking, the operating system sends the signal to the to the parent, but when the child dies.

20:42  
But, but that's no difference.

20:44  
And another thing is that a dead child becomes a zombie until somebody asks how it died, until somebody asks for its exit code.

20:56  
The exit code holds the the value, the reason of death usually.

21:02  
So this is like a benefit we have in in Linux that we can wait for the for the termination.

21:08  
We can we get notification when a process dies and we can also ask its exit code, even, even if it's long debt.

21:15  
And then again, it's a good idea to ask your dead children how they died because otherwise you can, can end up the zombies can, can bottle up in the system.

21:26  
And and yeah, they, they take resources.

21:28  
So it's usually good idea to quickly ask your dead child how he died.

21:32  
So, so the resources of the zombie process are released.

21:37  
Yeah.

21:38  
Killing a process doesn't kill children.

21:40  
That's again an analogy from the real world, because usually when you kill parents, the children live on.

21:45  
So this same goes in in operating system, which also means that killing a huge sub tree of processes.

21:53  
Because if you have a complex application, the application itself might execute multiple processes you have like the main process which corresponds to the loaded program, but then again it executes additional processes that run on background for instance.

22:09  
So usually if you want to kill the whole application then you need to kill multiple processes.

22:15  
You need to kill the process itself and all spawn children.

22:19  
And that's not possible easily because you can kill the parent, but when you kill the parent then the children's are orphaned and readapted by the in it process.

22:28  
And the in it process in Linux is always there.

22:30  
That's the first process started at the beginning.

22:33  
It has PID 1.

22:35  
So all the the children are somewhat rehashed, regrouped under the under this in it process.

22:42  
It's like an orphanage for all the processes.

22:46  
By the way, this also, this is also how demonization works.

22:50  
If you you want to create a demon, do you know what the demon is in the in the Linux terminology?

22:56  
It's like a system service or it's like a service that tends to run in the background.

23:01  
OK, So what do you do if you want to create a demon?

23:06  
You have like some initialization exec process for the demon.

23:11  
The exec process spawns the demon and then the demon.

23:18  
The child process kills its own parent when the parent is dead.

23:26  
The demon process.

23:27  
The child process is is adapted by the, by the by the init process and it it can do all the stuff it needs to do in the background without any direct connection to its parent.

23:39  
OK, so this is how demonisation works in the in Linux and another another.

23:47  
The implication is that if you want to kill the whole subtree of the processes, you need to pay.

23:53  
You need to put in some extra effort.

23:56  
And it's a bit tricky because technically a process can be spawning other processes.

24:00  
Why are you killing its children?

24:03  
So you need to be quick about it or you need to suspend the whole tree before you start killing.

24:08  
Yeah, it's always better.

24:09  
It's always easier to kill someone when when someone is asleep than than to kill someone who's running, right?

24:15  
That's like one of the basic rules of, of this kind of business, right?

24:21  
So technically you need to do some extra effort to to kill the whole subtree of processes if you want to.

24:29  
In Windows the situation is similar, but a bit more complicated.

24:32  
The trouble is that originally the basic structure which is called Win 32 process entry doesn't does not contain the PPID, does not contain the parent process ID, which makes it slightly more difficult to track who's the parent of which process.

24:50  
Right.

24:50  
There is no actually there is no special relationship between the parent and the child.

24:55  
So yeah, you can get this information from some extended system information structure.

25:00  
So yeah, there is a way how to get to that, at least on modern windows.

25:05  
And it's a bit just a bit tedious to do that, but you can do it.

25:10  
But then again, killing a process doesn't kill all its sub trees.

25:14  
If you if you kill a process and the whole subtree is killed in the in the process manager, for instance, yeah, you might do that often your Firefox or whatever browser you use just go go crazy, do some during some update or something just goes wacky and you need to kill it completely.

25:34  
So you just find that the the browser in your process manager and then you, you, you press the button, kill the process or end task or whatever, whatever its name and the whole subtree of all the processes is killed.

25:48  
But that's the job of the task manager.

25:50  
Again, there is special effort needs to be done and this special effort is implemented in this process manager.

25:57  
And this process manager is an application like any other.

25:59  
It just has very powerful privileges so it can kill other processes.

26:04  
So this is about parent child relations.

26:10  
Let's move on and let's just take a look about on how threads are created.

26:16  
About threads that that's much more easier because threads are normal part of parallel programming.

26:21  
So you you need to create threads all all the time.

26:24  
Actually usually you don't even create threads on your own.

26:28  
Creating threads is usually job for some parallel library you using.

26:32  
It's a very low level stuff, even for programmers, but even programmers sometimes need to create a thread.

26:41  
All operate, all common programming languages have direct adaptations, direct abstractions regarding the threats in C + + C#, Java, all these languages.

26:54  
Thread is basically a class.

26:55  
So you're basically creating a new instance of a new class, new object.

27:01  
And the only thing that you need to remember is that this new object is given a piece of code to execute, which is usually a function.

27:10  
Yeah, it might be something else like Lambda or delegate or whatever programming language you're using.

27:15  
But technically it's bottom line is it's some sort of a function or method that's being executed.

27:21  
Plus optionally it may get some additional parameters and this it's like a function call.

27:28  
Technically the difference is this new function is called, is executed in the newly created thread.

27:35  
But then again it's very similar to function call.

27:38  
So over here once the thread is created, it immediately executes.

27:42  
So here in this part of code, the main thread which executes the main function and this newly created thread T are running possibly concurrently.

27:56  
Why?

27:56  
I'm saying carefully possibly because there is no guarantee that these two threads will ever run concurrently.

28:03  
It depends on your system resources.

28:05  
If I have a very old or some very low tech PC, then this code might still run sequentially.

28:12  
In this case T and main thread might actually run somewhat semi concurrently.

28:19  
So they will like share the CPU and for a while they will be running the first thread, for a while they will be running the second thread, and then we will be alternating on the CPU.

28:30  
So that's also possible.

28:31  
But if I have a normal modern CPU with multiple cores, it's very likely that these two threads will be really executed in parallel until I call T join, which is like wait for death.

28:45  
It's very similar to the to wait for a single object or wait PID for the processes.

28:52  
So basically here I'm waiting for T to terminate.

28:56  
This will block the main thread.

28:58  
The main thread will be suspended until thread T finishes it's job, finishes this code and terminates.

29:08  
So this is how threats are being created and terminated.

29:14  
And the most important part, the most important job the operating system does is scheduling.

29:21  
So it needs to perform, it needs to handle all the processes and make sure that they share the CPU in some, in some manner.

29:33  
The scheduling algorithms, yeah, are quite complex to explain the best scheduling algorithms would take much more than this course.

29:40  
Actually, it would take like a dissertation if you want, if you're into this stuff.

29:44  
So it's quite complicated, especially now when we have multiple CPU cores.

29:50  
By the way, just to clarify a few things, Multitasking term stands for executing multiple processes within the same operating system.

30:01  
Multitasking is something which is quite old.

30:05  
Actually it's not that old because I still remember an operating system that didn't have multitasking.

30:10  
Would you venture a guess what that system would be?

30:13  
I was using an operating system that didn't have multitasking.

30:21  
Yeah, actually all the windows had multitask.

30:23  
All the windows I knew.

30:24  
I'm not sure about all the windows.

30:26  
All the windows I knew had multitasking.

30:28  
Even 3.1.

30:34  
Yeah, MSDOS.

30:35  
That was the only system that was available when I was young and that didn't have multitasking, not at least not in the traditional sense, because in MS-DOS there was only one application running at any given time.

30:49  
Yeah, you can like a suspend application, keep it resident in memory and execute another application, but it's not real multitasking.

30:58  
So multitasking is concurrent execution of multiple processes, real concurrent.

31:05  
And it can be done in single on a single CPU core, it can be done.

31:09  
You just need to make sure that all these processes are somehow sharing the CPU core evenly in a time multiplexing.

31:17  
Which means each time for time slice, for a short period of time, the CPU is allocated for one process only.

31:24  
But this time is kind of short.

31:26  
It's for few millisecond or few, few tenths of milliseconds, so that it creates an illusion that these processes are really running currently, but in fact they share the CPU in a very fast, orderly fashion.

31:41  
Multi processing on the other hand is a utilization of multiple CPU cores, which is the job for current operating systems because usually you have multiple CPU cores.

31:52  
I'm not sure if you can even buy less than two or maybe 4 cores nowadays if you're buying modern and mainstream CPUs.

32:01  
Of course, I'm not talking about specialized CPUs like Arduinos or CPUs for your washing machines, but normal CPUs for your normal laptops.

32:09  
I'm not even sure if you can even buy CPU with less than two or less than 4 cores.

32:14  
So this is a normal thing that all the operating systems needs to be doing and actually they need to be doing these two things simultaneously.

32:21  
Usually you have many, many more processes in an operating system.

32:28  
By the way, if you have access to some Linux server, you can easily check how many processes are running on that Linux server.

32:35  
It would be thousands usually.

32:37  
Normally, yeah, it depends on what survey it is, etcetera, how many users are using this server.

32:42  
But technically it's no, no problem if there are 1000 processes.

32:47  
So yeah, you usually don't have 1000 cores.

32:49  
So you need to balance all these processes among all the available cores.

32:53  
And that's quite a juggling act.

32:55  
You, you need to do this properly.

32:57  
Yeah, sometimes you need also to set affinity.

33:00  
Sometimes you want to prefer that some processes are running, running on some cores only.

33:05  
Sometimes you have a very specific processes that needs to run all the time or that needs to consume a lot of resources or they need to respond in real time to something.

33:15  
So you can allocate CPU cores forever for these, for these processes by setting affinity that specific process runs on a specific core or set of cores.

33:26  
Yeah, and all these things are very, very, very challenging.

33:30  
Another thing you need to understand before we get to the scheduling algorithms themselves is what's involved in the multitasking.

33:39  
When multitasking happens, what you need to do, what you need to be able to do, is to suspend one process, actually suspend one threat.

33:47  
I've previously mentioned that we are scheduling at the level of threats, not processes.

33:51  
So you need to take one thread away from the CPU core and put another threat in its place.

33:58  
What that involves.

33:58  
This involves something which is called context switch, changing the context.

34:04  
The context itself holds everything that's necessary for this, for the thread to run on the CPU core, most importantly the registers.

34:13  
That's the most part.

34:15  
If you remember at least the registers, that would be good, right?

34:18  
But well, if you remember that there are registers and maybe something else that that would be good because usually there is something else.

34:25  
For instance, if there are some additional units, let's say 87 Co processor, yeah, probably we don't have these anymore.

34:33  
Sometimes we do for some specialised CPUs.

34:36  
But if there are some additional units, you need to save the state of these additional units as well.

34:41  
By the way, if I'm talking about registers, I'm not talking only about the 8 or 16 registers.

34:47  
I've, I've shown you there are many more registers like vector registers or specialized CPU registers, system registers.

34:54  
So there are many more registers than you you've seen.

34:57  
And also it needs to adjust the virtual memory so the address space of each process is unique.

35:05  
I haven't talked about it, about this in detail.

35:08  
We will.

35:08  
We will cover it in two weeks or so, or in three weeks, sorry, there will be vacation.

35:14  
But basically it needs to switch to the virtual memory of the new thread.

35:20  
If the threads are of the same process, this doesn't have to be changed because both threads have the same virtual memory space.

35:27  
But if these two threads are from different processes, this needs to be adjusted so that the new thread, the newly executed thread, will see the virtual address space of its own process, not of the old thread.

35:38  
Of course, that would be dangerous.

35:41  
By the way, this doesn't involve memory caches.

35:44  
Memory caches are transparent.

35:46  
So yeah, they can affect a performance.

35:49  
Because if you have one thread heavily crunching some data on a CPU core, and then you put another thread in its place and it starts to crunch a complete the different numbers, complete the different data, then yeah, it will have different data set in the caches.

36:05  
So this data set wouldn't be very useful to the new thread.

36:08  
So it it kind of needs to flush everything out of the cache and load new data to the cache.

36:13  
But this will happen automatically, transparently, it cannot access the data of another thread in the caches.

36:19  
Caches are transparent.

36:21  
They work on an on a physical memory, not on the virtual memory.

36:24  
So yeah, caches, switching threads on CPU cores may affect the caches, may affect performance, but they don't they don't pose a threat from the perspective of the operating system.

36:36  
So yeah, context, which basically means that I need to store everything listed in here somewhere in memory.

36:42  
Every CPU thread, every, every operating system thread entity has its own space somewhere in the operating system kernel where everything involving this can be stored somehow and from where it can be restored.

36:58  
So basically you need to save it for the old thread and load it for the new thread.

37:03  
And it's quite costly.

37:04  
This context switch cost hundreds or maybe thousands of instructions because just just you need to save the registers, right?

37:13  
So count how many registers you have and for each of these registers you need to you need to perform store or load instruction.

37:21  
So there are plenty.

37:25  
And now let's go to the scheduling small, let's say diversion before we get to the real scheduling, not real scheduling, poor choice of words.

37:36  
Before we get to the, let's say, scheduling in mainstream operating systems.

37:40  
I've got, let's say a bit of a teaser that scheduling can be quite complicated if you have quite specific requirements.

37:49  
This is a once just to put you any at ease.

37:54  
This is just one slide.

37:55  
I'm talk, I'm not going to talk about real time systems.

37:58  
That is a very specific domain of operating systems.

38:01  
But sometimes the scheduling can be very difficult, especially if you need to to maintain some guarantees in real time operating systems, in real time scheduling, you need to, let's say meet some demands that are made outside of the operating system from the external environment, especially if you're doing scheduling or handling processing in some sensitive operating system, in some sensitive machine that controls mission critical systems like self driving car or a nuclear nuclear power plant, which is, by the way, do you know where the nearest nuclear power plant is?

38:41  
Where the not, not the power plant where the the nearest fission react reactor is.

38:50  
Yeah, somewhere over there, a few meters next to us, a few 100 meters next to us.

38:54  
So yeah, if something goes terribly wrong, well, don't worry.

38:58  
It has a power outage of which can probably boil water in a kettle, but that's it.

39:04  
So it, it's not very powerful, but I've been there and, and we've overloaded it once and yeah, it makes a huge noise when the real time system decides to release the graphite rods that control the reaction and they just fall down and stop the reaction.

39:20  
So that's the stuff I'm talking about.

39:21  
If something bad happens, if some bad event, not not necessarily completely bad, but when some event happens, we need some guarantees like the deadline.

39:30  
We need to make sure that within some time something will really happen, right?

39:36  
For instance, if you overload the reactor and the operating system gets a notification that the power outage or the reaction monitor or whatever is out of scale, it needs to within a few milliseconds or a few 100 milliseconds, it needs to release the magnetic clamps that holds the rods and it has to be done.

39:57  
It's it's a deadline which which needs to be met if you have a self driving car and it's it's approaching an obstacle.

40:05  
On the road, it needs to deploy brakes right away or within some reasonable amount of time.

40:11  
Yeah.

40:12  
Also that there might be additional constraints like release time, for instance.

40:17  
There is some time delay that needs to be, let's say, observed before the action is taken.

40:23  
A good example would be a an airbag deployment.

40:27  
I'm not sure if you've ever been in a car accident.

40:29  
I hope not, but if you if you at least observe it in some test or something, you may have noticed that the the airbag isn't deployed immediately.

40:38  
If a car crash is detected, the airbag needs to be deployed as I don't know 1050, I'm not expert, but with some delay so that the body of the driver or the the the next two driver, the the passengers are moved by the momentum into the right position, right?

41:02  
Actually there is some additional mechanisms involved.

41:05  
You know, if two cars crashed or if a car crashed into an obstacle, there are some deformation zones and the the engineers have calculated how much actually time it takes before the head of the driver gets to the right position above the the steering wheel.

41:23  
And at that very moment, the, the deployment of the Arabic must commence because if it if it's sooner, then it, it will blow up and then the head will move into the deflating airbag, which is which was which would be no help at all.

41:39  
So it these two forces need somehow to work right together.

41:43  
Furthermore, there are some additional techniques like your seatbelts.

41:47  
They have some specialized pressured gas valves and and and containers which are released so that your your seatbelts are loosened up just a bit so that the seat belt like gently releases you into the airbag.

42:05  
These two things need to be coordinated, coordinated together.

42:09  
So technically what I'm speaking about is that all these things needs to be put into a synergy and it must happen in this precise order.

42:20  
By the way, when we're talking about deadlines, there are some different types of deadlines.

42:24  
The the most basic distinction is hard and soft deadline.

42:28  
Hard deadline is a deadline that needs to be met.

42:30  
And if it isn't met, it doesn't matter anymore.

42:33  
For instance, if you don't release the, the graphite rods into the core of the of the nuclear power plant and you don't do it in time, then you you shouldn't bother anymore because there is no power plant to speak of, right?

42:47  
If you don't deploy brakes before the obstacle, then it doesn't matter because the, the CPU that computes that drives the car no longer exists because it's been demolished by the by the obstacle.

42:59  
So this is deadline that really needs to be meet.

43:03  
But many systems have lighter, lighter concerns like soft deadlines.

43:08  
For instance, if you play a movie or if you send an SMS.

43:12  
Yeah.

43:13  
Do you guys know what SMS is still OK?

43:17  
Just kidding, I hope so.

43:19  
So if you send an SMS for instance, there is a soft deadline.

43:22  
How long does it should it take for the for the SMS to be delivered?

43:27  
It's usually within seconds, but it's a soft deadline.

43:30  
So if this doesn't happen within the deadline, it it bothers the the authors of the of the software, it bothers the, the vendors that bothers the company that runs your cellular network.

43:46  
But in, in, if it doesn't happen too often, the people wouldn't complain.

43:51  
So it's like, yeah, you can sometimes if the system is really overloaded, you can sometimes relax these deadlines, but you shouldn't do that very often.

44:01  
If you're playing a movie, if you have AVCL or whatever player you prefer, so you're playing a movie which you just downloaded from a torrent or whatever, then you can, the player itself needs to deliver the images, the frames of the of the movie in some timely fashion.

44:22  
So it needs to meet a soft deadline for each of the frame to decode it, to properly render it, to place it into the right buffer for the in the video card, etcetera.

44:32  
Yeah, if sometimes this deadline isn't meet you, you can see it in in your player as a hiccup as a as a small distortion in the in the movie.

44:43  
But if it happens once or twice during the movie, you will probably won't complain, right?

44:49  
If it happens every second, it would be problem you you wouldn't you did you wouldn't like to watch that movie.

44:56  
So this is a very specific things that needs to be sometimes observed.

45:00  
But fortunately for us, let's focus on normal systems when these hard constraints or more difficult constraints aren't required.

45:09  
First of all, we need to perceive how each threat or scheduling unit is maintained within the system.

45:19  
So basically there are few states a scheduling unit may be in.

45:25  
When a thread or process is created, there is some initialisation state, yeah, all the data structures needs to be filled, etcetera, etcetera.

45:32  
That's boring.

45:33  
And afterwards it gets into the ready state.

45:36  
In ready state, it's ready to be executed immediately.

45:39  
So it has everything it needs, it has data loaded in the memory, it's basically ready to run immediately, right?

45:46  
But usually most of the processes, most of the threads are in the ready state.

45:50  
Because usually you have much more threads, much more processes then you have CPU cores.

45:55  
When it happens that CPU core becomes available, it switched to the running state.

46:02  
So the running state is the instruction of that particular thread are actually being executed by one of the cores.

46:11  
And this might end by two possibilities.

46:14  
One possibility is that the threat is has run out of time.

46:20  
This running state wouldn't be there, couldn't be there forever.

46:24  
So each thread is given a slight time slice, a short period period of time for how long it can run.

46:31  
And once this time is depleted, it can either be switched back to the ready state, or it's also possible that the thread performs some operation, that is, that requires some additional resources from the operating system.

46:46  
Let's say loading data from a file.

46:50  
That's a costly operation because loading data from a hard drive is like 6 orders of magnitude slower than executing instructions.

47:00  
So it's it's, it would be a waste if, if this threat will be running just waiting for the data, if it just wastes millions and millions of instructions or millions and millions of cycles of the CPU while it just wait.

47:14  
So it's turned into a boxed state and it's somewhat temporarily, temporarily removed from the scheduling.

47:22  
So it's not planned for running for for some time.

47:26  
Once the data are loaded, when the conditions for blocking are resolved, then it's turned from the block state back into the ready state and it can be scheduled.

47:36  
It can be executed again.

47:39  
Once all the instructions of the thread are finished are are concluded are the thread terminates.

47:45  
It's actually, this is for a I'm not distinguishing between threats and processes too much right now.

47:53  
Yeah, if threat terminates, it can be joined.

47:56  
If process terminates it, it becomes a zombie and it can be asked for for exit code.

48:01  
So but the the principle is same for both.

48:04  
So it's turned into some special terminated state.

48:07  
And in in this state, it waits until somebody asks how the process or how the threat has has terminated.

48:16  
So there are multiple ways how to implement the scheduling.

48:20  
The simplest way is cooperative scheduling.

48:24  
By cooperative it means that the threat, when it's switched to the running state, it must voluntarily yield the control.

48:33  
So the threat itself is responsible for monitoring how much time it's spent on the CPU.

48:39  
And if the threat deems it has enough, it doesn't have anything more, any work that's very important right now to do or it has deemed it, it it took too long to compute something.

48:54  
So then it can decide to yield the CPU to another threat.

49:00  
Trouble with that is that if you have a malfunctioning threats, malfunctioning programs or malicious programs or just programs written by lazy programmers who don't want to yield the CPU, that's, that's probably the most often scenario, then the operating system has no means, no additional means how to take the threat from the CPU.

49:22  
So this is a very, let's say, naive way how to look at the at the multitasking.

49:29  
This is usually something which is employed in parallel libraries, not in a, not in normal operating systems, but in special specialized libraries.

49:39  
Well, you assume that all the threads, all the units are cooperating somehow in the same effort.

49:45  
They just divided the work.

49:46  
So they can run the work, They can compute the work on multiple cores, but they are computing together, They are all in this together.

49:54  
So they want to yield willingly, because they need to share the data with with their fellow threads, their fellow computing units.

50:02  
But normally in operating systems, we usually tend to use pre emptive scheduling.

50:08  
By the way, I know only one operating system.

50:11  
I've used only one operating system that used cooperative scheduling and that was Windows 3.1.

50:16  
It has multitasking, but cooperative multitasking.

50:19  
So yeah, processes programs in in running in Windows 3.1 needed to yield explicitly if they wanted to allow other programs to run.

50:30  
So usually we are doing pre emptive scheduling which means each thread is given only a time slice that that's what I was talking about previously.

50:40  
This is how this usually works.

50:42  
We we have a short time slice, short periods of time which are given to each to each scheduling unit.

50:48  
And the operating system has some hardware timer, uses some hardware timer that uses interrupt to monitor what is what is being, what is running on each CPU core and to change what is running on CPU core.

51:02  
So basically imagine, imagine it as this, there is a timer that can take, I'm not sure every millisecond or something like that, OK or every few milliseconds.

51:13  
And this timer every time it takes it executes because it's an uninterrupted.

51:18  
So it executes an interrupt handler written by the operating system which invokes the scheduler.

51:25  
And the scheduler takes a look at the the thread currently running on the CPU core and says OK, you can still run or it says no, you you've been running too long.

51:36  
And if it's if it has been running too long then it blocks.

51:42  
Sorry, it it consumed it whole time slides, so it blocks, it interrupts the thread and it makes a context switch to another thread.

51:54  
OK, so it can swap which threads are running on the CPU core.

51:59  
This is basically how the preemptive scheduling works.

52:04  
So remember what's the important part here?

52:07  
You need to remember that we need external help.

52:10  
There needs to be some external source of some external timer that helps us with the scheduling.

52:15  
Otherwise we wouldn't be able to interrupt the running process.

52:19  
Interrupt the running thread.

52:22  
Unfortunately, deciding how long the thread should run on the CPU and which thread should run on the CPU next is a difficult task.

52:32  
The mainly it's difficult because we have many objectives we want to fulfill and these objectives are not entirely compatible.

52:39  
For instance, we we obviously need to optimize the usage of CPU.

52:44  
We really want to use all the CPU cores and if we have some workload to do then we really want to compute.

52:51  
So we don't want leave any CPU core idle.

52:55  
If we have work to do, we need to allocate the CPU cores fairly.

53:00  
This is one of the most important parts.

53:02  
So you want to give equal share or as equal as possible share to each of the processes, each of the threads which are currently able to run.

53:14  
Otherwise some of the processes might feel like left behind or something.

53:21  
You need to maximize throughput, which means throughput is number of processes completed per time unit.

53:28  
So you want, you really want this to scale so that many processes can be processed together and finish all together within the operating system.

53:37  
And also you want to minimize the turn around time.

53:39  
So for each of these processes, you want to minimize the time that takes for the process to finish.

53:46  
Yeah, this these two kind of go somewhat together.

53:49  
And also you need to minimize waiting time, so or minimize response time.

53:55  
So for each application, for each process you want to minimize the time it spends outside of the CPU core, especially if these applications are interactive.

54:06  
Unfortunately, individual applications have individual requirements.

54:10  
If you have something that handles UI, such application can sleep most of the time or be blocked most of the time and only needs to wake up when something happens in the UI, when the user clicks a mouse or where the user presses a button.

54:26  
OK, so at this at this time at this point, the application needs to wake up and do some stuff and then it can sleep again.

54:34  
However, there are also different other types of applications which tend to compute a lot, tend to crunch numbers.

54:42  
These applications want to run as long as possible on on the CPU core without possibly any interruptions.

54:49  
So yeah, you can see that they have different requirements.

54:52  
And also these computing applications don't bother themselves that they don't care about response time because it doesn't matter whether they sleep for a millisecond or or one second, it doesn't concern them.

55:05  
So yeah, these are like a basic requirements and all these requirements, I wouldn't say they are all met, but the operating systems are trying to address all of these issues when designing scheduling algorithms, how to do this, how to, let's say try to bring some fairness and and some some additional stuff into the scheduling process.

55:30  
So first of all, let's talk about priority.

55:33  
Each process, sorry, each threat, each scheduling unit has, it's a priority.

55:38  
By priority, I mean something like a number.

55:42  
Imagine a number, and this number expresses the importance of the process or the importance of the threat.

55:50  
The more important process, the more important threat, the sooner or the more often it should be scheduled.

55:58  
Or it might be scheduled before lower priority processes.

56:02  
OK, so this is the notion of priority.

56:05  
Usually we divide priority into two parts, a static priority, which is like a part of this number which is assigned when the program starts, when the threat starts.

56:17  
Yeah, you can do that using operating system tools or or even command line tools.

56:21  
And you can just adjust this number to denote the importance of the process or or the threat within the process.

56:28  
Usually the process have has one number, threads has another number, and these numbers are somehow somehow combined together.

56:37  
And then there's a dynamic priority which is changing over time.

56:42  
And this dynamic priority is there to add fairness to the scheduling.

56:45  
So basically, the dynamic priority tends to grow when the process or the threat hasn't been scheduled for a long time and decreases when the process or the threat takes some computation time, spend some time on the CPU.

57:03  
One possibility how to look at it is that the priority, the dynamic priority is increased for all ready processes or for all ready units of scheduling every once in a while.

57:15  
So every once in a while, everyone who's waiting for the CPU is giving a slight nod, slight bump in the in the dynamic priority.

57:24  
And once a process or a threat is executed, is assigned to a CPU, this dynamic priority is reset to 0 or to some lower value.

57:33  
So it's decreased.

57:35  
Yeah.

57:35  
And this usually works.

57:37  
The whole priority is usually combination of these two numbers.

57:41  
So it's like having 2 numbers which you, you can just add them up or something.

57:49  
And how use this numbers to actually plan for something?

57:56  
So there are many algorithms that can be implemented.

57:58  
The most most straightforward one, yeah, just to have something in comparison is first come first serve.

58:06  
So yeah, that's a boring algorithm.

58:08  
Just to have some basic baselines.

58:11  
You have a single queue where the processes are stored.

58:15  
Yeah, the processes are stored on one end and taken from the other.

58:19  
So let's say the processes are put in from the tail and and taken from the head and the head process is running on the CPU.

58:27  
Yeah, this in this implementation without anything, this was usually used in batch processing for large computers like 40 years ago.

58:35  
And the head process is running on the CPU until there is no it has anything else to do.

58:41  
And when it has anything else to do, it's just removed from the queue and that's it.

58:45  
Yeah, that's not very, not very interesting.

58:48  
You can also have instead of regular queue, you can have some other queues like priority queue where the processes are ordered somehow.

58:58  
For instance, you can take shortest job first.

59:02  
This minimizes throughput because the the processes will be the shortest processes will be executed first.

59:09  
So you can process more and more processes in a in a unit of time.

59:15  
Unfortunately this requires 2 things.

59:17  
First of all, this is not a good idea to do to do generally because long processes would starve.

59:24  
And 2nd, you usually don't know how long a process would take.

59:29  
That's an information which is like, yeah, you can, you can detect this information exposed, but not a priori.

59:35  
So usually you don't have this kind of information.

59:39  
Yeah, you can do this the other way around, but that's even worse.

59:43  
So what can you do in the terms of pre emptive scheduling?

59:46  
That's more interesting that those were just the baselines for non pre emptive scheduling.

59:51  
So the the easiest stuff you can do is a round Robin scheduling.

59:54  
Yeah, round Robin actually was originally designed as a method for signing partitions or or signing some documents which are inflammatory.

1:00:06  
So if you're signing some for something like a petition, then you don't want to be first on the list, right?

1:00:13  
Because the first one on the list should be persecuted.

1:00:17  
By the authorities, right?

1:00:18  
So they devise a method like a round tape where isn't clear who was the 1st on the on the list who signed it first.

1:00:28  
This was the way how to obfuscate this so everybody's on the list, but there is no clear first name.

1:00:35  
So the the round Robin technique is that you have multiple things, in this case the units of scheduling.

1:00:42  
You still have a queue and you assign the first process in the queue to the CPU, but only for the time slice.

1:00:49  
If it completes then it's easy, you just throw it away and take another.

1:00:53  
But if it gets blocked or if it runs out of time, you just take it and put it to the queue again.

1:01:00  
So you are rotating the units of scheduling possibly evenly on this, on the available CPU core.

1:01:08  
That's the easiest thing you can do.

1:01:11  
If all these units of scheduling have the same or similar requirements, it might work fine.

1:01:17  
Actually if you're implementing A trivial operating system, like if you're implementing a core kernel core for the operating systems course, this is what we did because you don't really need to do anything elaborate just to make it somehow work.

1:01:34  
This would be like the first implementation for your operating system.

1:01:39  
Unfortunately, if some of these, some of these units of scheduling have special requirements, especially for interactivity, for low, low latency, this wouldn't work very well for them.

1:01:53  
Because usually these UI processes or UI threats will get to the CPU.

1:02:01  
Then they will notice that nothing happened from the point of user interaction.

1:02:06  
So they will immediately terminate, immediately yield the CPU and will be put in the queue again.

1:02:12  
When they will, when they spend a long time before they get the CPU again, they don't use the CPU, they need to use the CPU when somebody, something happens in the UI.

1:02:22  
And if they're waiting in the queue and something happens in the UI, it will take them long time before they can respond to that.

1:02:30  
So yeah, it will work, but it will get these interactive processes somewhat sluggish.

1:02:36  
So what we need to do is somehow divide the priorities, divide the types of processes.

1:02:43  
This is 1, let's say, like a virtual algorithm.

1:02:46  
I'm not sure if this was ever implemented in a real world.

1:02:50  
A similar algorithm was used, or algorithm that somewhat emulates this was used in older versions of Windows.

1:02:56  
I'm not sure what exactly Windows are using for scheduling nowadays, but let's just imagine that we have multiple of these round Robin queues.

1:03:05  
And this is like a queue for the interactive processes and this would be like a queue for the heavily, heavily computing processes.

1:03:14  
And of course you might have multiple queues in between.

1:03:17  
So this design has some additional rules.

1:03:22  
Of course, the rule, the first rule is where should we pick our process?

1:03:28  
Because if we have multiple queues, we need some rules which queue to choose first.

1:03:32  
So we search them from top to bottom.

1:03:36  
If the first queue is non empty, then we take process from the first queue.

1:03:40  
If the first queue is empty, we take the second queue etcetera, etcetera.

1:03:44  
We take the first non empty queue from the top and we take the head process from that queue.

1:03:49  
So this is who will be scheduled next.

1:03:52  
OK, when the process is blocked so or yields its computation slice.

1:04:02  
So when the process ends prematurely, let's say, then the process is moved one queue upwards.

1:04:12  
Yeah.

1:04:12  
If the process is blocked, it's taken temporarily out of the scheduling.

1:04:16  
And then when it returns, it's it's placed one, one position upwards, 11Q upwards.

1:04:22  
If the process yields its time slice, if it doesn't want to compute right now, it's also moved one slice upwards.

1:04:30  
OK.

1:04:32  
And then the if the process takes all the time, if if it consumes its time slice, it's it is moved one one way down.

1:04:45  
OK, so the processes that tend to wait for interactions are moved upwards.

1:04:52  
the OR which are blocked are moved upwards so they can interact immediately when they get unblocked.

1:04:58  
The processes that tend to compute a lot that they take all the time, all the time slice to compute, they are moved downwards.

1:05:06  
And also it might be a good idea to change the time slice for these for these queues.

1:05:11  
For instance, let's say the time slice for the processes in the top queue would be like a millisecond or very few milliseconds.

1:05:18  
On the other hand, for the low lowest queue, it could be easily 50 or 100 milliseconds because these processes like to compute.

1:05:26  
So let them compute.

1:05:28  
Let's give them the CPU core for a longer period of time.

1:05:31  
These processes want just to check something, so they don't want to compute that much.

1:05:36  
OK, this is basically how this works.

1:05:40  
Yeah, it works much better than a regular one queue round Robin.

1:05:45  
But then again, having this many queues is somewhat tedious to manage.

1:05:50  
And also it doesn't always work, as I said, because for instance, if you have a process that just yields its own time slice, it can end up very high in this hierarchy.

1:06:01  
But if it doesn't block, if it just yields, it might end up in the top queue.

1:06:06  
And it's always executed from the top queue.

1:06:08  
So there needs to be some additional mending rules for that.

1:06:11  
So if a process is executed multiple times, it it it's given the rest or something.

1:06:17  
So yeah, it's not entire entire algorithm, but it's just an idea of how the scheduling may be augmented to give some some differentiation among the scheduling units to give you a glimpse about a real world, a real world scheduling algorithm.

1:06:37  
This is a simplified description of a completely fair scheduler, CFS, which is currently the default.

1:06:46  
Last time I checked, it was the default scheduler in the Linux kernel.

1:06:50  
So this is something which is really being used in a real world operating system.

1:06:56  
So the CFS works as follows.

1:07:00  
Each process has one important number, which is called virtual runtime.

1:07:08  
In a very rough sense, it's like how much time this process consumed, how much time this process was running on the CPU.

1:07:15  
Actually by process, yeah, I'm talking about processes, because just ignore that.

1:07:23  
It's the unit of scheduling in Linux, it's the unit of scheduling.

1:07:28  
It's process, but that doesn't matter.

1:07:30  
So each scheduling unit has this number which says how long it actually take the CPU, how long it actually run.

1:07:41  
And all these scheduling units are placed in one red black tree.

1:07:46  
So it doesn't matter, it's red black.

1:07:48  
The important is it's a balanced tree and we can very fast make a search for a process for the scheduling unit with for a unit of scheduling with the smallest virtual runtime.

1:08:02  
So normally what we do is we take the process that the unit of scheduling with the smallest virtual runtime and that's the one that is going to be executed.

1:08:13  
How long there is a maximum execution time which is repeatedly computed as there's the time slice and it's repeatedly computed as the total waiting time divided by current number of processes.

1:08:30  
So you kind of try to guess from the waiting time how long you should get, what's the even portion of the time so that every once in a while all the processes will get its fair timeshare.

1:08:45  
OK, so if the average waiting time is one second and there are 20 processes waiting for this particular core, then each process will be given 50 milliseconds, its maximum execution time.

1:09:00  
And yeah, this value is recalculated adaptively, continuously, because you need to adjust it based on how many processes are there, etcetera.

1:09:10  
So the longer each thread, each scheduling unit waits, the greater time slice it gets.

1:09:20  
It makes sense because if the system is overloaded, we want to increase the time slices so that we are lowering the overhead and making the processes really utilize the CPU.

1:09:31  
And the algorithm itself is quite simple.

1:09:33  
As I said, we take the left most node of the RB three, so the process with the smallest a virtual execution time, virtual runtime.

1:09:42  
And this is the one process that will be executed on the current core.

1:09:47  
If it completes the execution and it's easy, it's just removed from the arbitrary and and it's removed from the scheduling.

1:09:55  
The same goes if it's blocked, then it's temporarily removed from scheduling and if it reaches the maximum execution time it's stopped or interrupted or contact switch then it's it's re entering the tree with a new time key and the new time key is the old time key plus how much time actually the process spent on the CPU.

1:10:17  
So it's easy, you're just counting time.

1:10:19  
It's just time accounting, nothing else.

1:10:22  
As you can see this this makes sense when you want when you want to prioritize new processes.

1:10:29  
If a new process, new thread enters the scene, it has virtual time 0 because it's new, it hasn't run yet.

1:10:38  
So it's it's prioritized over all the old processes which are already there.

1:10:42  
And we just usually accumulated quite some time running on the CPU.

1:10:47  
So new processes, new new threats are given priority and they're running for some time before they they're enter the normal herd that that's been there for for some time.

1:10:59  
However, to make this let's say even because if this would be true, if this whole algorithm would be true as this, then threats that has computed a lot wouldn't get the CPU anymore, at least not if there are any other processes available to run.

1:11:18  
So to make this working, there is one additional condition that that fixes it.

1:11:24  
And it says that the virtual runtime decays is decreased over time.

1:11:30  
So every once in a while all the units of scheduling in this tree have their keys decreased by some constant.

1:11:40  
OK, that's it, no additional things.

1:11:43  
So as you can see, the rules are quite simple.

1:11:46  
And even with these simple rules, yeah, quite simple, there are some additional things, some some corner cases that needs to be covered, right.

1:11:54  
But basically with these two rules, how to compute the time slice and how to select easily the process that can be executed, that's enough to have a fair schedule.

1:12:08  
Yeah, of course, this is a like a dark magic because how exactly the virtual time the case I haven't I haven't explained and I do not even know.

1:12:22  
I can just say that how fast this decays is also affected by the threat or the process priority.

1:12:30  
So the priorities I talked about earlier are somehow incorporated in this.

1:12:35  
So processes with higher priorities have higher decay constants than processes with lower priorities.

1:12:43  
OK, so that's pretty much it.

1:12:47  
Regarding the scheduling, there are a few more things about processes.

1:12:52  
One of them is inter process communication.

1:12:55  
You already know that some processes need to communicate with other processes, right?

1:13:00  
Otherwise there wouldn't be, we couldn't have features like that.

1:13:03  
One process kills another process.

1:13:06  
Actually, you've seen the simplest interactions between multiple processes already.

1:13:11  
I'm assuming all of you are attending the Linux classes, am I right?

1:13:17  
Yeah.

1:13:17  
Cool.

1:13:18  
So what happens when I write something like this?

1:13:36  
Yeah, I know this is a sorry about that.

1:13:38  
It's it's mind teaser.

1:13:40  
I used to do this all the time because you shouldn't call cat to with a grab.

1:13:45  
The grab can load its own file, but I I can't resist.

1:13:48  
This is like AI.

1:13:49  
Hope your Linux teachers see that and their head explode or something.

1:13:56  
But technically it doesn't matter.

1:13:57  
I'm just what I'm doing here.

1:13:59  
Do you remember the do you, have you seen these two common mind tools?

1:14:05  
You haven't seen a cat before?

1:14:06  
Yeah.

1:14:06  
OK.

1:14:07  
In the class.

1:14:07  
I've definitely seen it out.

1:14:12  
Well, this is an animal and this is a fruit, so no.

1:14:17  
So cat, what does cat do?

1:14:18  
What's the content of the file?

1:14:22  
So we just take the file and print it to the standard output, Right.

1:14:27  
And grab is a filter.

1:14:29  
It's given a pattern is given usually a regular expression pattern, and it filters out the lines of the file that meets this pattern.

1:14:40  
Basically, we don't need to go to the details.

1:14:43  
That's not important.

1:14:44  
But what what happens here from the perspective of operating systems and not really this is a process, this is an application.

1:14:54  
Actually this is a program which becomes a process to be completely accurate.

1:15:00  
Same for this.

1:15:01  
So these are two programs which are executed simultaneously and what what does this do?

1:15:11  
What is this?

1:15:13  
It's a pipe.

1:15:13  
That's right.

1:15:15  
And what does pipe do exactly?

1:15:24  
So this will create a process.

1:15:27  
This process has standard input, standard output and error output.

1:15:31  
This will create a process.

1:15:33  
So yeah it has like 1 input and two outputs right?

1:15:44  
The input is connected to the input of your terminal.

1:15:48  
The first process is this input connected to the input of your terminal and this pipe takes this.

1:15:55  
This is city out connects this to.

1:16:00  
This creates this magical interconnection.

1:16:05  
And yeah, this is still connected to your console.

1:16:08  
This is still connected to your console, and this is still connected to your console because there are no additional things.

1:16:14  
OK, these three things are merged together to the to our console.

1:16:20  
So basically this is the basic interaction between two programs we can imagine and and it uses standard input and output.

1:16:29  
So it's, it's kind of lame, but it's very useful if you're doing something on the command line because you just create enough individual applications, you you pipe them together and then you have like a processing pipelines for some textual loading or something.

1:16:46  
Actually pipes are more generic because pipe is a well actually this pipe is not this pipe.

1:16:54  
Pipe is also an entity of operating system and it it works in a similar way.

1:17:01  
Actually it's like 2 processes tend to share a pipe.

1:17:05  
So they create another link which looks like this.

1:17:09  
One process can write to it and another process can read to it, but it's separated from the standard input and output.

1:17:15  
So when 2 processes needs to communicate, it's it's easier if they just negotiate a separate pipe which will have the same intention, which actually you will use a very similar instruction, similar, very similar functions.

1:17:27  
One will just print to it and the other just will read from it.

1:17:31  
Yeah, but it will be a separated channel from from standard input and output.

1:17:35  
That's usually more practical.

1:17:36  
Actually it works very similarly with with TCP circuits.

1:17:40  
If you have ATCP circuit, it's also a pipe of some sort where you can write or from from which you can read.

1:17:48  
So it's it's very similar.

1:17:50  
By the way, having a pipe is almost identical to having a socket over the TCP loop back.

1:17:57  
And do you remember what the loop back is?

1:18:00  
Sorry, yeah, every, every computer has a like a virtual network device inside, inside the operating system, which actually all the processes of this operating system can connect to it and it creates appearance of an Internet connection, but it actually just resends the data back.

1:18:20  
So if two processes use loop back interface, they are just connecting within within the same computer.

1:18:28  
The packets are not going anywhere outside the computer.

1:18:31  
It's just but they they can use the same networking mechanisms as if they were communicating with outside world.

1:18:38  
So sometimes it's good for debugging or it's a good if you have a, let's say, more complex application that involves multiple processes, multiple programs.

1:18:47  
And this program is to communicate.

1:18:49  
And sometimes you want to run them on the same computer some months, sometimes you want to run them on multiple computers.

1:18:55  
Another way how to processes can can communicate.

1:18:59  
Yeah, but sorry, so this is what I'm talking about.

1:19:02  
This is the IPC, this is the communication between processes.

1:19:06  
Another way is having a shared memory.

1:19:10  
Shared memory is more tricky to establish.

1:19:12  
So yeah, I'm not explaining how to create shared memory.

1:19:15  
But when you do, you basically have like a 2 pointers in two processes and each process points to its own data buffer.

1:19:25  
But these two data buffers are somewhat overlapped.

1:19:28  
So if one process writes something into this buffer, it magically appears in another process.

1:19:33  
Technically they are using the same memory.

1:19:36  
OK.

1:19:37  
And finally, final part is signals that that's something that's more typical for Linux or POSIX systems.

1:19:47  
So in in Linux kernel or Linux application a signal is kind of a call or an interrupt that one process can send to another process.

1:20:00  
In a sense the signals are somewhat an extension of CPU exceptions.

1:20:05  
And also some CPU exceptions translate into signals.

1:20:08  
So if something happens, something bad happens to your application and sorry, exception occurs, then some of these exceptions are translated into signals.

1:20:19  
The signal handler itself is a is a function.

1:20:23  
So you you can create a few functions and this and then say, hey, I'm attaching this function as a signal handler for this signal.

1:20:30  
And when this signal happens, this function will be called.

1:20:34  
But it will be called immediately like an interrupt outside of the normal flow of instructions of your main thread.

1:20:41  
It will be called within some service thread provided by the kernel.

1:20:47  
OK, so yeah, these functions then can be called at any time, at any moment during any piece of your normal regular code.

1:20:56  
Let me just give you a few examples.

1:20:58  
Yeah, I have a Just by the way, you may have noticed that there are some additional information sometimes hidden in the notes beneath the slides.

1:21:09  
If you haven't noticed that, check check it out, because there are some important things so I don't have to rewrite them to the to the whiteboard.

1:21:20  
By the way, there are some links, usually there are some additional links or some additional explanations that might be important.

1:21:26  
But the the this is a list of the most important POSIX signals.

1:21:32  
I'm not sure if you can read it.

1:21:39  
Hopefully now you can.

1:21:42  
And so one thing you should know about signals is that if you have no signal handler or you don't handle it properly, a signal will terminate your application.

1:21:52  
So unhandled signals terminate your application.

1:21:55  
Think of it this way, all the default handlers have something like exit or terminate in them.

1:22:03  
Sick term is actually sent by the operating system or by other application.

1:22:07  
If somebody wants to kill you politely, if somebody just asks you to die gently and quietly, they will send you sick term.

1:22:18  
So please just drop that and we'll be, we'll be fine.

1:22:21  
There is a more violent version of that which is called sick kill, and that's actually a signal you cannot stop.

1:22:28  
You cannot have a handler for that.

1:22:30  
So if somebody sends you sick kill, you are dead immediately.

1:22:34  
This is what what's used by the tools for killing the processes.

1:22:39  
If the processes start to misbehave or become deadlock or something, then you can send them sick kills to terminate misbehaving processes.

1:22:48  
Yeah, you need, of course, you need sufficient privileges.

1:22:51  
So usually a root user is sending sick kills to the misbehaving processes.

1:22:57  
Another very important signal which you might have encountered, and if not, I'm sure many of you will encounter it during your 6th home assignment, is 6F6F is shorthand for segmentation violation and basically this has happened if you perform invalid memory access.

1:23:14  
If you're touching memory, you shouldn't be touching 111.

1:23:19  
Quick example what you can do if you have a.

1:23:32  
You have a new pointer and you try to dereference it.

1:23:39  
This just blow your application away, throwing the 6F signal which will ultimately result in ending your application.

1:23:48  
Because this is not allowed operation, you have new pointer, which means this pointer is isn't pointing anywhere in particular and you're trying to access that value.

1:23:58  
That cannot be done.

1:23:59  
No can do.

1:23:59  
Sorry Sir, your application has been terminated.

1:24:03  
It also happens if you try, if you have a rogue pointer that wanders in a part of memory which isn't allocated for instance, then the operating system says, hey, this well operating system with cooperation with memory protection, which we will cover next time, hopefully will say, hey, this memory is protected.

1:24:22  
You cannot access this memory success.

1:24:27  
By the way, there are other things.

1:24:29  
Sick child is sent to parent process when child process terminates.

1:24:32  
I've already told you that.

1:24:34  
So this is one of the signals.

1:24:36  
And for instance sick int is sent to you if you press CTRL C on your on your keyboard.

1:24:43  
So CTRL C translates also into a signal, and if this signal isn't properly handled, it kills the application.

1:24:50  
So if you learn that control C sometimes terminates your misbehaving script or something.

1:24:57  
Yeah, it does, because it sends a signal.

1:25:00  
There are also user signals.

1:25:02  
These two signals are designed for user defined purposes.

1:25:05  
So if you want to communicate among processes, you have two signals which are completely harmless.

1:25:11  
Well, they also terminate the process if there are no handlers.

1:25:14  
But but they are harmless from the perspective they have no any additional meaning.

1:25:19  
You can use them for whatever you need to do within their inter process communication.

1:25:27  
Yeah, I think so.

1:25:29  
There are only, I'm not sure 1632 signals, I don't remember, but there are only very few signals.

1:25:36  
So you can have too many of them.

1:25:38  
OK.

1:25:39  
Oh by the way, seek I'll, yeah, this is one example I skipped accidentally.

1:25:43  
Seek I'll is when an illegal, not I'll but illegal instruction is executed.

1:25:50  
This is actually direct translation of a CPU exception, because remember if you execute an invalid OP code, invalid instruction, it throws out an exception software interrupt and this software interrupt is translated by the kernel into a signal which is translated back and sent back to the process itself.

1:26:15  
OK, unfortunately I have exactly three 2 1/2 minutes, which is enough time to say you few more things.

1:26:24  
But I have a completely new topics to start and it would be very irresponsible of me to start a new topic right away and hurry up through the introduction within two minutes.

1:26:38  
So unfortunately I will have to let these two minutes go.

1:26:43  
Are there any questions before we conclude for today?

1:26:48  
No more questions, no questions at all.

1:26:51  
OK, so thank you for coming.

1:26:52  
I hope next time not next week, remember next week our vacation, the other week are also vacation.

1:26:59  
So we will see each other in three weeks.

1:27:01  
And next time I hope just that your ranks will be replenished again and we will meet in a greater numbers.

1:27:08  
Thank you and enjoy your vacation.

1:27:10  
And remember after three, after these three-week.

1:27:14  
We will have a longer session.

1:27:17  
Yeah, I, I assume I will make a short break somewhere in the middle of this session because I don't want to keep you here too long.

1:27:25  
So we will interrupt it in the middle somewhere.

1:27:28  
But we need to prolong it slightly to fit all the data, all the materials in.

1:27:33  
So thank you for coming and see you in three weeks.