Betriebssysteme

Tutorium 4

Peter Bohner

22. November 2024

ITEC - Operating Systems Group

Shortest Job First

Pitfalls - How would you implement this?

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length!

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length! How do you solve this?

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length! How do you solve this?
- Predict the *future* based on *past* behaviour

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length! How do you solve this?
- Predict the *future* based on *past* behaviour
- Does this work?

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length!
  How do you solve this?
- Predict the *future* based on *past* behaviour
- Does this work?
  „THERE IS AS YET INSUFFICIENT DATA FOR A MEANINGFUL ANSWER.“
  ~ Isaac Asimov, „The Last Question“ (Comic)

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length!
  How do you solve this?
- Predict the *future* based on *past* behaviour
- Does this work?
  „THERE IS AS YET INSUFFICIENT DATA FOR A MEANINGFUL ANSWER."
  ~ Isaac Asimov, „The Last Question" (Comic)
  You need some balanced initial value. Not *that* big of a deal with preemption though. Why?

Shortest Job First

Pitfalls - How would you implement this?

- You would need to have future knowledge to figure out the job length!
  How do you solve this?
- Predict the *future* based on *past* behaviour
- Does this work?
  „THERE IS AS YET INSUFFICIENT DATA FOR A MEANINGFUL ANSWER."
  ~ Isaac Asimov, „The Last Question" (Comic)
  You need some balanced initial value. Not *that* big of a deal with preemption though. Why?
  Interrupt the process after the estimated time is over.

Priorities

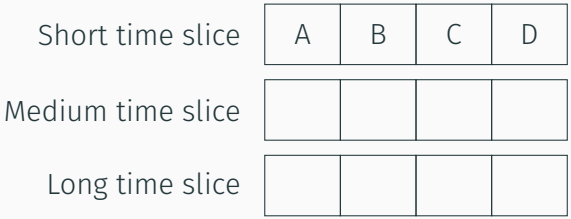
What is priority scheduling? Why would you use it?

Priorities

What is priority scheduling? Why would you use it?

- Each process is assigned a priority
- The process with the highest priority is chosen

Multi-Level Feedback Queues



How it works

- All processes start in the highest queue
- When they use up their timeslice and are preempted, they descend
- If they block before, they stay in the level (optionally: Are moved up)

$\Rightarrow$ I/O bound processes rise to the top and react quickly, CPU bound processes get longer timeslices but less often

Multi-Level Feedback Queues

How it works

- All processes start in the highest queue
- When they use up their timeslice and are preempted, they descend
- If they block before, they stay in the level (optionally: Are moved up)

$\Rightarrow$ I/O bound processes rise to the top and react quickly, CPU bound processes get longer timeslices but less often

Multi-Level Feedback Queues

How it works

- All processes start in the highest queue
- When they use up their timeslice and are preempted, they descend
- If they block before, they stay in the level (optionally: Are moved up)

$\Rightarrow$ I/O bound processes rise to the top and react quickly, CPU bound processes get longer timeslices but less often

A Flawless Scheduling Algorithm?

Consider the waiting time

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

How could you fix this?

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

How could you fix this?

- E.g. reset the whole thing after a given interval, so all start in the highest level again

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

How could you fix this?

- E.g. reset the whole thing after a given interval, so all start in the highest level again
- ”Boost” processes that waited for a long time

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

How could you fix this?

- E.g. reset the whole thing after a given interval, so all start in the highest level again
- ”Boost” processes that waited for a long time

What metrics does it optimize?

- Utilization? Turnaround time? Throughput? Waiting time? Response time?

A Flawless Scheduling Algorithm?

Consider the waiting time

- A few I/O-bound jobs could saturate the CPU!

$\Rightarrow$ The lower-level processes starve

How could you fix this?

- E.g. reset the whole thing after a given interval, so all start in the highest level again
- ”Boost” processes that waited for a long time

What metrics does it optimize?

- Utilization? Turnaround time? Throughput? Waiting time? Response time?
- Prefer I/O bound, prefer short jobs, group the rest based on their needs

Process switching

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space
- And a bit of housekeeping:

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space
- And a bit of housekeeping: Open files, scheduling priorities, ...

$\Rightarrow$ These things make up the PCB - The

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space
- And a bit of housekeeping: Open files, scheduling priorities, ...

$\Rightarrow$ These things make up the PCB - The Process Control Block

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space
- And a bit of housekeeping: Open files, scheduling priorities, ...

$\Rightarrow$ These things make up the PCB - The Process Control Block

Where are the PCBs stored? In user or kernel space?

Process switching — PCB

What does the kernel need to do when switching processes?

- Adjust Instruction, Base and Frame Pointer – okay

The process shouldn't notice it was interrupted! What else do you need to adjust?

- General purpose register
- Address space
- And a bit of housekeeping: Open files, scheduling priorities, ...

$\Rightarrow$ These things make up the PCB - The Process Control Block

Where are the PCBs stored? In user or kernel space?

- Kernel space! Users shouldn't be able to modify them

Process switching — PCB

Are the PCBs always valid?

Process switching — PCB

Are the PCBs always valid?

No! Some parts (registers, PC, etc) only when the process is not running. Why?

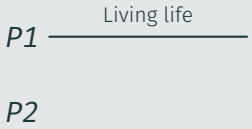
Process switching — PCB

Are the PCBs always valid?

No! Some parts (registers, PC, etc) only when the process is not running. Why? They are saved when it is switched out.

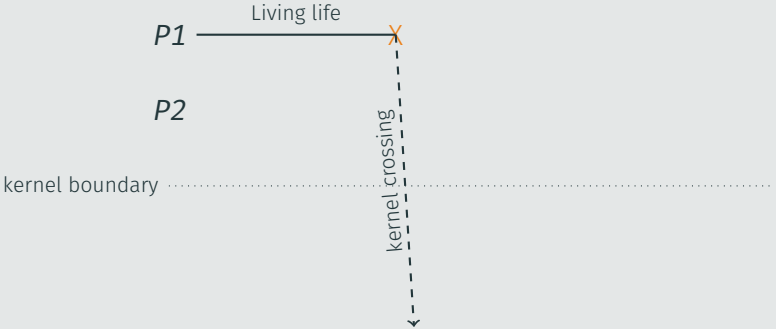
Process switching

In pictures



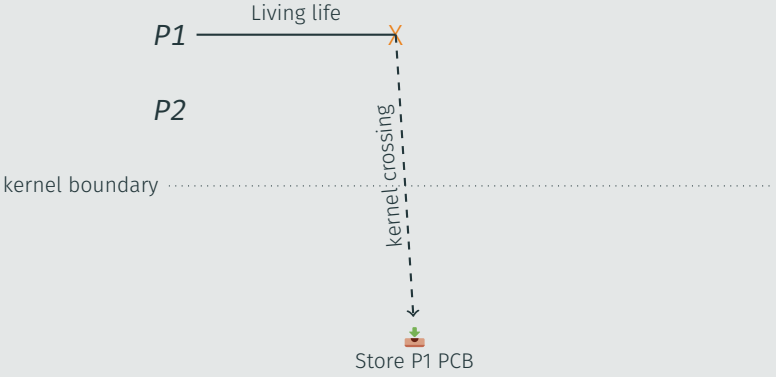
Process switching

In pictures



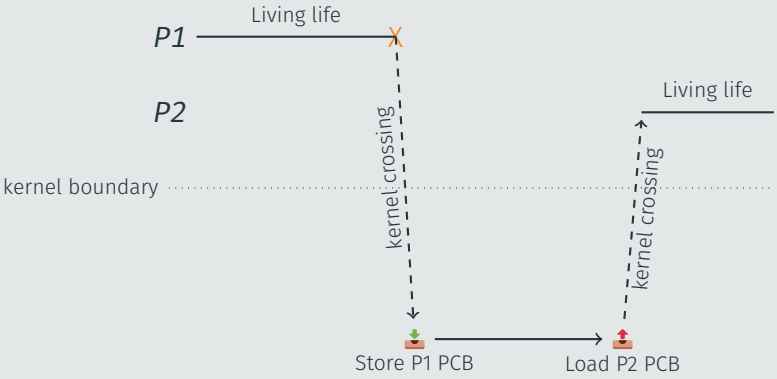
Process switching

In pictures



Process switching

In pictures



Process switching

As text

1. Transition to the kernel (How?

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)
2. Save the context of the process. What was that again?

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)
2. Save the context of the process. What was that again? Registers, Program counter, Stack pointer, etc.

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)
2. Save the context of the process. What was that again? Registers, Program counter, Stack pointer, etc.
3. Where do you save it to? A per-process kernel stack typically (as it is often moved there automatically) and then move it to the PCB.

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)
2. Save the context of the process. What was that again? Registers, Program counter, Stack pointer, etc.
3. Where do you save it to? A per-process kernel stack typically (as it is often moved there automatically) and then move it to the PCB.
4. Restore the context of the next process

Process switching

As text

1. Transition to the kernel (How? Interrupt, Exception, Syscall)
2. Save the context of the process. What was that again? Registers, Program counter, Stack pointer, etc.
3. Where do you save it to? A per-process kernel stack typically (as it is often moved there automatically) and then move it to the PCB.
4. Restore the context of the next process
5. Leave kernel mode and transfer control to the PC of the next process

Threads

Threads

What are processes, address spaces and threads? How do they relate to each other?

Threads

What are processes, address spaces and threads? How do they relate to each other?

- A thread is an *entity of execution*, the personification of control flow

Threads

What are processes, address spaces and threads? How do they relate to each other?

- A thread is an *entity of execution*, the personification of control flow
- A thread lives in an address space, i.e. all the addresses that it can can access and the data that is stored there

Threads

What are processes, address spaces and threads? How do they relate to each other?

- A thread is an *entity of execution*, the personification of control flow
- A thread lives in an address space, i.e. all the addresses that it can can access and the data that is stored there
- Thread + Address Space = Process

Spawn a few threads using pthreads!

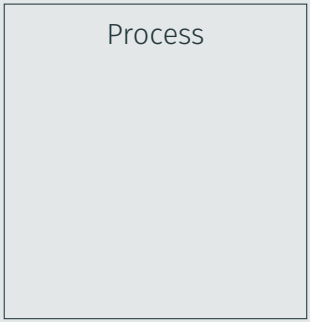
Write a small program that creates five threads using the pthread library. Each thread should print its number (e.g., *`Hello, I am 4`*) and the main program should wait for each thread to exit.

Thread models — One To One

One To One

new Thread()

new Thread()



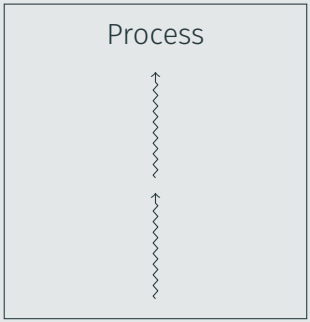
Problems?

Thread models — One To One

One To One

```
new Thread()
```

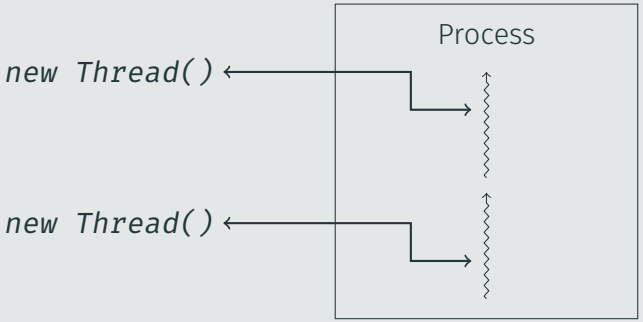
```
new Thread()
```



Problems?

Thread models — One To One

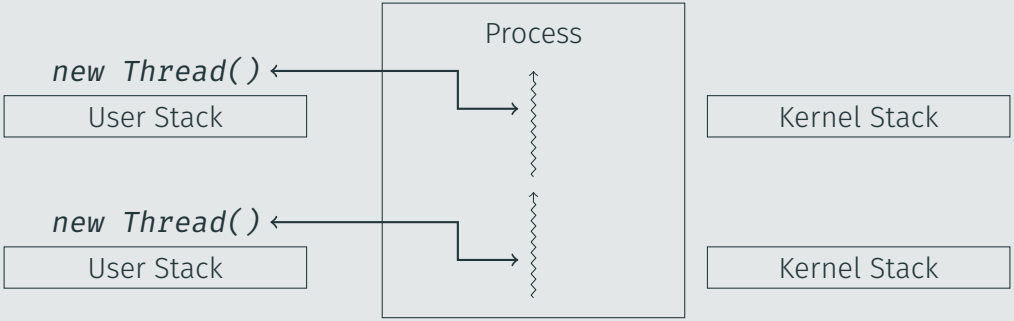
One To One



Problems?

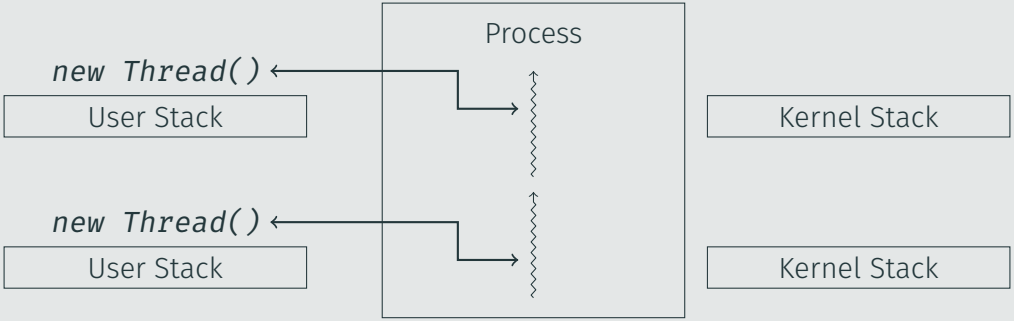
Thread models — One To One

One To One



Thread models — One To One

One To One



Problems?

Thread models - One To One

Problems and benefits of *One To One*?

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads
+ Can piggy-back onto the OS scheduler

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads
+ Can piggy-back onto the OS scheduler
- Must piggy-back onto the OS scheduler

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads
+ Can piggy-back onto the OS scheduler
- Must piggy-back onto the OS scheduler
- Relatively high overhead due to context switches

Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads
+ Can piggy-back onto the OS scheduler
- Must piggy-back onto the OS scheduler
- Relatively high overhead due to context switches
- Relatively high overhead *when creating one*

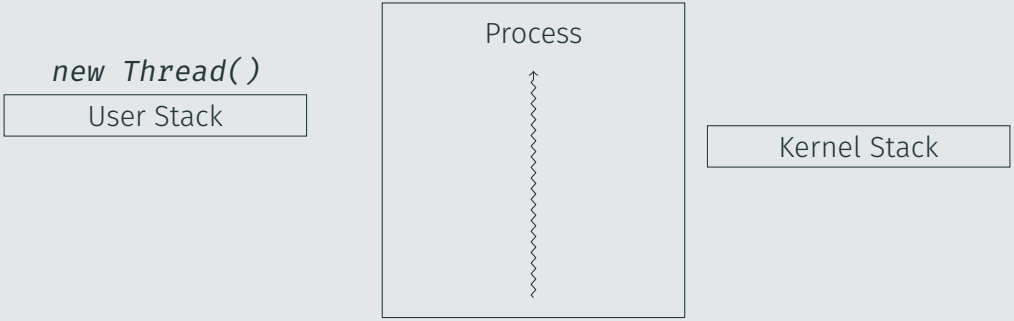
Thread models - One To One

Problems and benefits of *One To One*?

+ Scales with core count
+ Conceptually easy — the OS does the hard stuff
+ Blocking does not affect other threads
+ Can piggy-back onto the OS scheduler
- Must piggy-back onto the OS scheduler
- Relatively high overhead due to context switches
- Relatively high overhead *when creating one*!!

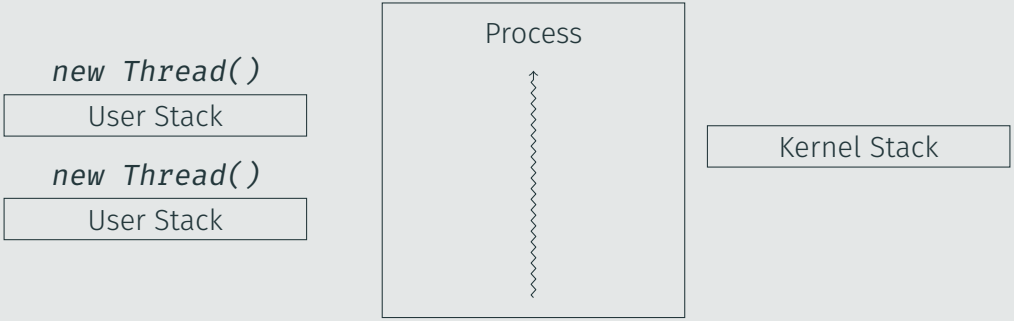
Thread models

Many to One



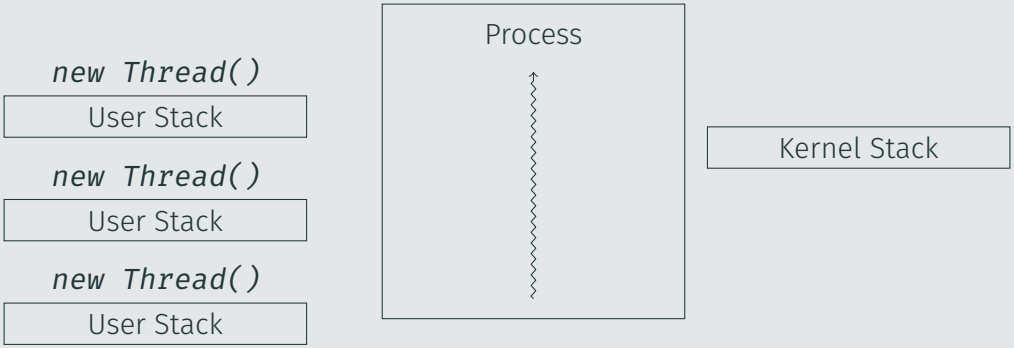
Thread models

Many to One



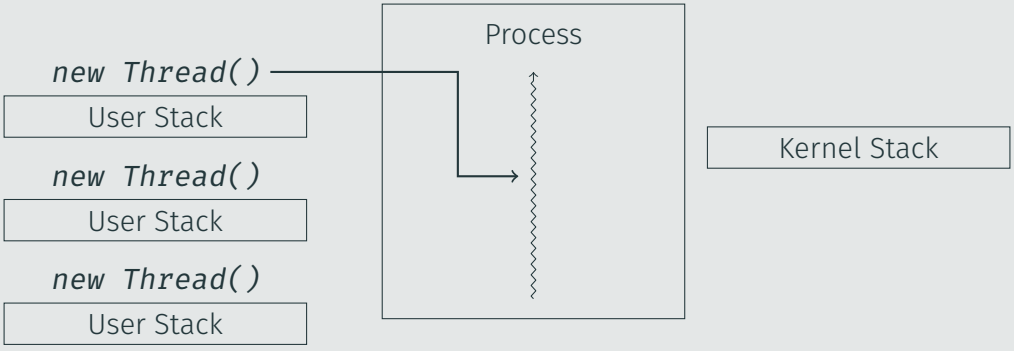
Thread models

Many to One



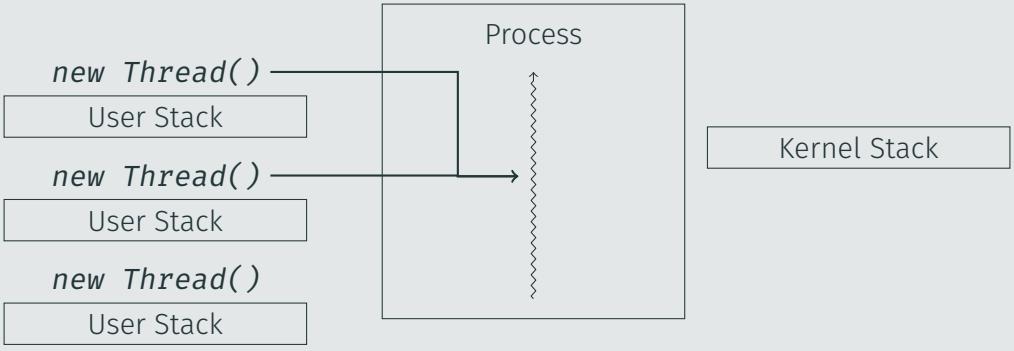
Thread models

Many to One



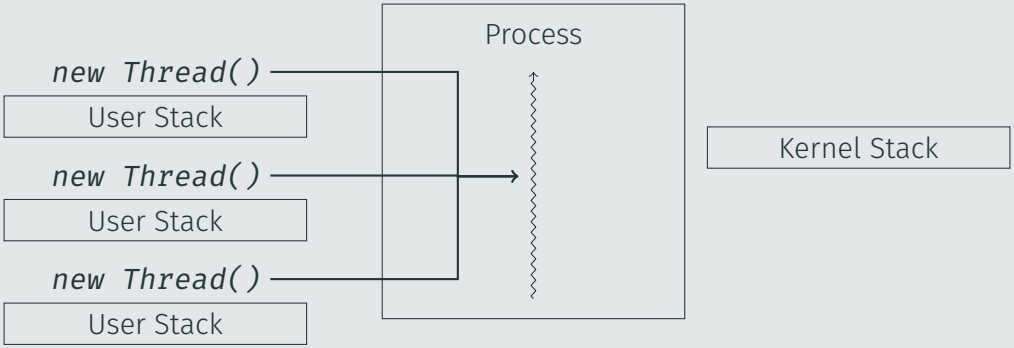
Thread models

Many to One



Thread models

Many to One



Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

+ Scales with core count?

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- Can only use one core

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- − Can only use one core
- + **Conceptually easy — the OS does the hard stuff?**

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- Can only use one core
- Harder to implement — the OS doesn't help you much

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- − Can only use one core
- − Harder to implement — the OS doesn't help you much
- \+ **Blocking does not affect other threads?**

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- Can only use one core
- Harder to implement — the OS doesn't help you much
- Blocking *does* affect other threads

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- − Can only use one core
- − Harder to implement — the OS doesn't help you much
- − Blocking *does* affect other threads
- \+ **Can piggy-back onto the OS scheduler?**

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- Can only use one core
- Harder to implement — the OS doesn't help you much
- Blocking *does* affect other threads
- Can *not* piggy-back onto the OS scheduler

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- Can only use one core
- Harder to implement — the OS doesn't help you much
- Blocking *does* affect other threads
- Can *not* piggy-back onto the OS scheduler
- Must piggy-back onto the OS scheduler?

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- − Can only use one core
- − Harder to implement — the OS doesn't help you much
- − Blocking *does* affect other threads
- − Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- – Can only use one core
- – Harder to implement — the OS doesn't help you much
- – Blocking *does* affect other threads
- – Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- – **Relatively high overhead due to context switches?**

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- – Can only use one core
- – Harder to implement — the OS doesn't help you much
- – Blocking *does* affect other threads
- – Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- \+ Low overhead during context switches

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- – Can only use one core
- – Harder to implement — the OS doesn't help you much
- – Blocking *does* affect other threads
- – Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- \+ Low overhead during context switches
- – **Relatively high overhead *when creating one*?**

Thread models - Many To One

Problems and benefits of *Many To One*?

Do they improve anything?

- – Can only use one core
- – Harder to implement — the OS doesn't help you much
- – Blocking *does* affect other threads
- – Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- \+ Low overhead during context switches
- \+ Low overhead when creating one

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. *`nodejs`* using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. *`nodejs`* using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another
- Selection: One or more are executed (i.e. an *`if`*-statement)

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. *`nodejs`* using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another
- Selection: One or more are executed (i.e. an *`if`*-statement)
- Iteration: A block is executed more than once (i.e. a loop)

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. `nodejs` using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another
- Selection: One or more are executed (i.e. an `if`-statement)
- Iteration: A block is executed more than once (i.e. a loop)
- Recursion: A block calls itself until an exit condition is met (i.e. recursion!)

Do you know any keyword in C which *doesn't* quite adhere to that but can instead totally spaghettify your control flow?

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. *`nodejs`* using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another
- Selection: One or more are executed (i.e. an *`if`*-statement)
- Iteration: A block is executed more than once (i.e. a loop)
- Recursion: A block calls itself until an exit condition is met (i.e. recursion!)

Do you know any keyword in C which *doesn't* quite adhere to that but can instead totally spaghettify your control flow? `goto`

Thread models - Many To One

Do you know a programming language / runtime using that?

E.g. *`nodejs`* using its „event loop“

A small excursion - Structured Programming

Control flow should fall into one of four patterns:

- Sequence: One block is executed after another
- Selection: One or more are executed (i.e. an *`if`*-statement)
- Iteration: A block is executed more than once (i.e. a loop)
- Recursion: A block calls itself until an exit condition is met (i.e. recursion!)

Do you know any keyword in C which *doesn't* quite adhere to that but can instead totally spaghettify your control flow? *`goto`*

Famous paper by a proponent of Structured Programming:
„Go To Statement Considered Harmful“ by Edsger W. Dijkstra

Thread models - Many To One

And what about threads?

- Can outlive the methods they were spawned in

Thread models - Many To One

And what about threads?

- Can outlive the methods they were spawned in
- Can use variables and fields after they went out of scope in a method

Thread models - Many To One

And what about threads?

- Can outlive the methods they were spawned in
- Can use variables and fields after they went out of scope in a method
- Can split up or transfer their control flow arbitrarily

Thread models - Many To One

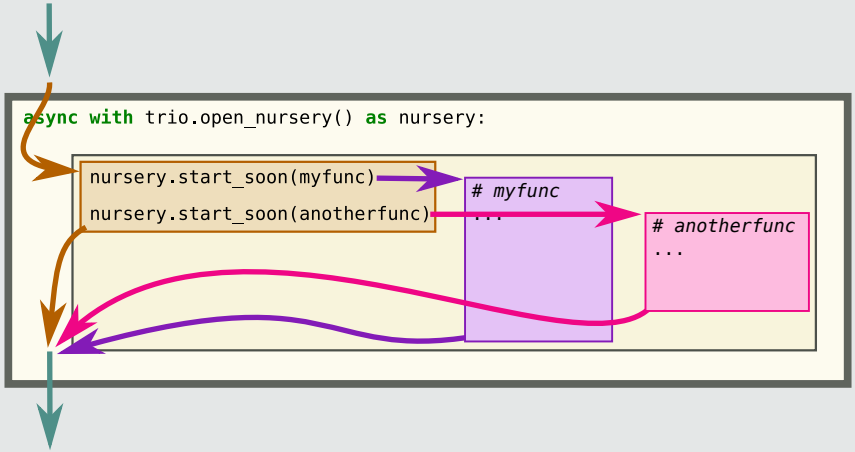
And what about threads?

- Can outlive the methods they were spawned in
- Can use variables and fields after they went out of scope in a method
- Can split up or transfer their control flow arbitrarily

So that might sound familiar...

Thread models - Many To One

Structured Concurrency



Taken from vorpus.org

Thread models - Many To One

Nice, but what does this have to do with ULTs?

Thread models - Many To One

Nice, but what does this have to do with ULTs?

- Spawning lots of threads for small operations *is too slow otherwise*

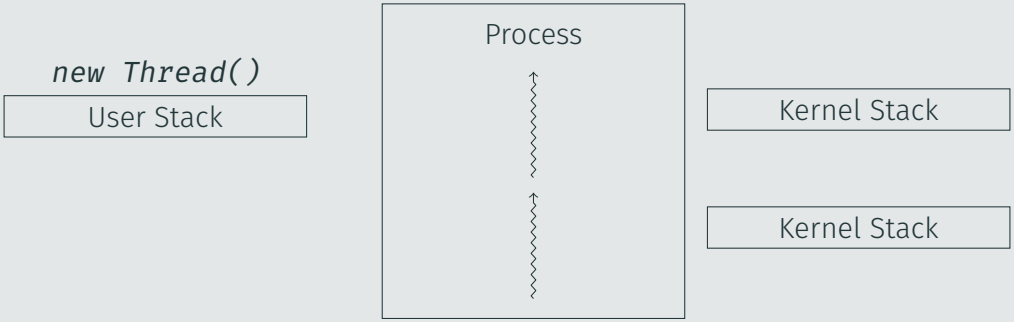
Further reading:

Notes on Structured Concurrency

ULTs and Structured concurrency in Java - Project Loom

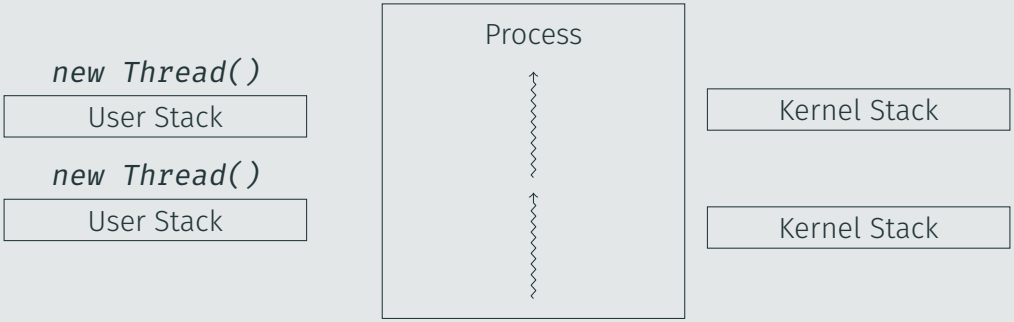
Thread models - Many To Many

Many To Many



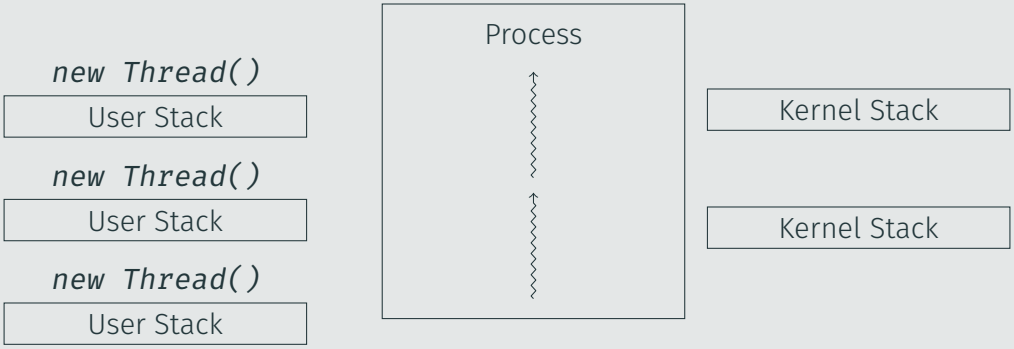
Thread models - Many To Many

Many To Many



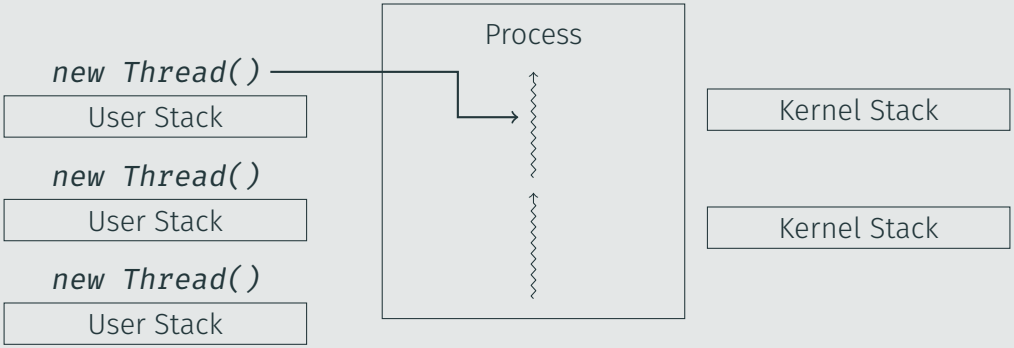
Thread models - Many To Many

Many To Many



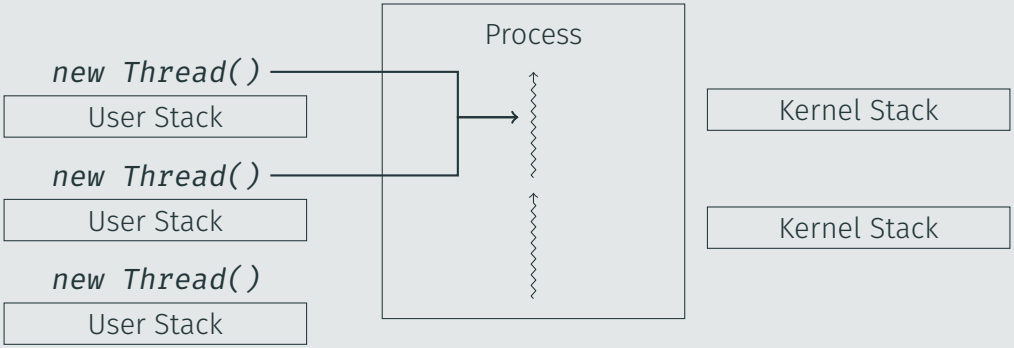
Thread models - Many To Many

Many To Many



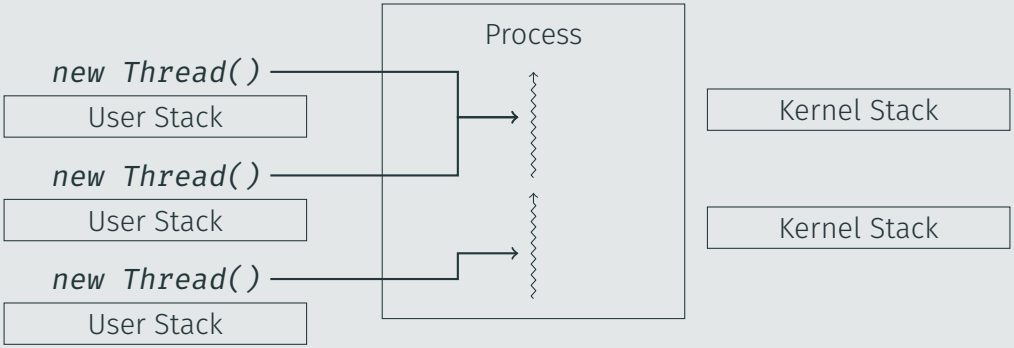
Thread models - Many To Many

Many To Many



Thread models - Many To Many

Many To Many



Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count?

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
+ **Conceptually easy — the OS does the hard stuff?**

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ **Blocking does not affect other threads?**

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
+ Can piggy-back onto the OS scheduler?

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
− Can *not* piggy-back onto the OS scheduler

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

- \+ Scales with core count
- − Harder to implement — the OS doesn't help you much
- \+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
- − Can *not* piggy-back onto the OS scheduler
- − Must piggy-back onto the OS scheduler?

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
− Can *not* piggy-back onto the OS scheduler
+ Can implement its own scheduler

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
– Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
– Can *not* piggy-back onto the OS scheduler
+ Can implement its own scheduler
– **Relatively high overhead due to context switches?**

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

- \+ Scales with core count
- – Harder to implement — the OS doesn't help you much
- \+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
- – Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- \+ Low overhead during context switches (unless you need to interact with the kernel threads, e.g. to schedule between them)

Thread models - Many To Many

Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

+ Scales with core count
− Harder to implement — the OS doesn't help you much
+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
− Can *not* piggy-back onto the OS scheduler
+ Can implement its own scheduler
+ Low overhead during context switches (unless you need to interact with the kernel threads, e.g. to schedule between them)
− **Relatively high overhead *when creating one*?**

Thread models - Many To Many

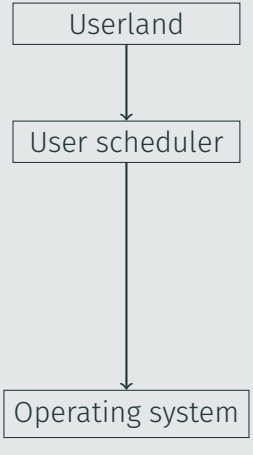
Problems and benefits of *Many To Many*?

Important: The kernel *knows about the user level scheduler*

- \+ Scales with core count
- − Harder to implement — the OS doesn't help you much
- \+ Blocking does not affect other threads as the kernel informs the user scheduler (`upcalls`) and does *not* pause the whole kernel level thread
- − Can *not* piggy-back onto the OS scheduler
- \+ Can implement its own scheduler
- \+ Low overhead during context switches (unless you need to interact with the kernel threads, e.g. to schedule between them)
- \+ Low overhead when creating one

Problems the second

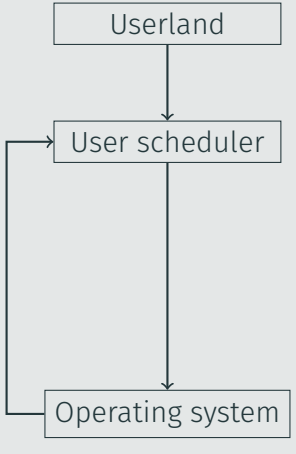
You have all attended SWT 1! So let's have a look.



And preemption is now possible, which might complicate user code.

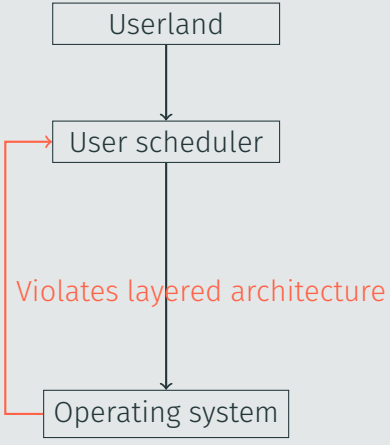
Problems the second

You have all attended SWT 1! So let's have a look.



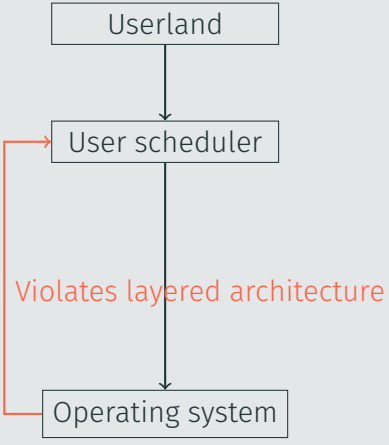
Problems the second

You have all attended SWT 1! So let's have a look.



Problems the second

You have all attended SWT 1! So let's have a look.



And preemption is now possible, which might complicate user code.

Thread models - One To One

What events can trigger a context switch?

Thread models - One To One

What events can trigger a context switch?

- Voluntary:

Thread models - One To One

What events can trigger a context switch?

- Voluntary: yield, blocking system call

Thread models - One To One

What events can trigger a context switch?

- Voluntary: yield, blocking system call
- Involuntary:
  - interrupts, exceptions, syscalls

Thread models - One To One

What events can trigger a context switch?

- Voluntary: yield, blocking system call
- Involuntary:
  - interrupts, exceptions, syscalls
  - end of time-slice

Thread models - One To One

What events can trigger a context switch?

- Voluntary: yield, blocking system call
- Involuntary:
  - interrupts, exceptions, syscalls
  - end of time-slice
  - high priority thread becoming ready

Thread models - Many To One

What events can trigger a context switch?

Thread models - Many To One

What events can trigger a context switch?

- Most libraries only support *cooperative scheduling*

Thread models - Many To One

What events can trigger a context switch?

- Most libraries only support *cooperative scheduling*
- Why is switching with preemption, interrupts, blocking system calls hard?

Thread models - Many To One

What events can trigger a context switch?

- Most libraries only support *cooperative scheduling*
- Why is switching with preemption, interrupts, blocking system calls hard? Kernel is not aware of the ULTs and will return where it came from — *but not call out to the scheduler and carry on*
- The benefit of platforms: How can `Java` (using Project Loom) or `Node.js` switch on most of the above?

Thread models - Many To One

What events can trigger a context switch?

- Most libraries only support *cooperative scheduling*
- Why is switching with preemption, interrupts, blocking system calls hard?
  Kernel is not aware of the ULTs and will return where it came from — *but not call out to the scheduler and carry on*
- The benefit of platforms: How can `Java` (using Project Loom) or `Node.js` switch on most of the above?
  You can not execute syscalls directly, but need to call library methods!
  Suspension points can be inserted there.

Thread models - Many To One

„Jobs are either I/O-bound or compute-bound. In neither case would user-level threads be a win. Why would one go for pure user-level threads at all?“

Thread models - Many To One

„Jobs are either I/O-bound or compute-bound. In neither case would user-level threads be a win. Why would one go for pure user-level threads at all?“

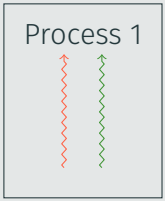
- Program structure (e.g. Structured Concurrency, channels or just easier pipelines)

Thread models - Many To One

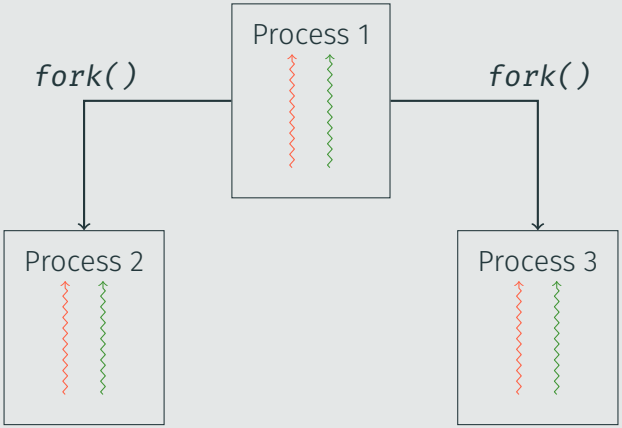
„Jobs are either I/O-bound or compute-bound. In neither case would user-level threads be a win. Why would one go for pure user-level threads at all?“

- Program structure (e.g. Structured Concurrency, channels or just easier pipelines)
- The same or higher I/O throughput if on an abstracted platform

Should a fork do this?



Should a fork do this?



Who is aware of the fork?

- The thread that executed the fork and the child

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else?

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Can you foresee any problems?

- If you copy the thread it might do weird things

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Can you foresee any problems?

- If you copy the thread it might do weird things
- Locks are memory and will be copied as well! What if a thread holds a critical lock when it is forked?

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Can you foresee any problems?

- If you copy the thread it might do weird things
- Locks are memory and will be copied as well! What if a thread holds a critical lock when it is forked?

Do you need thread duplication for our shell example?

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Can you foresee any problems?

- If you copy the thread it might do weird things
- Locks are memory and will be copied as well! What if a thread holds a critical lock when it is forked?

Do you need thread duplication for our shell example?

No, we *exec* anyways.

Threads — Forking

Who is aware of the fork?

- The thread that executed the fork and the child
- Who else? Nobody!

Can you foresee any problems?

- If you copy the thread it might do weird things
- Locks are memory and will be copied as well! What if a thread holds a critical lock when it is forked?

Do you need thread duplication for our shell example?

No, we *`exec`* anyways.

Summary

`fork` is not as simple as it once was. Is it still a good abstraction?

Kernel Mode Threads — Kernel Level Threads

What is the difference?

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

- Sometimes the OS needs to do some housekeeping that doesn't belong in a system call

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

- Sometimes the OS needs to do some housekeeping that doesn't belong in a system call
- Free some pages, swap something in and out of memory

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

- Sometimes the OS needs to do some housekeeping that doesn't belong in a system call
- Free some pages, swap something in and out of memory
- Flush pages from the disk cache to the hard disk

Kernel Mode Threads — Kernel Level Threads

What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

- Sometimes the OS needs to do some housekeeping that doesn't belong in a system call
- Free some pages, swap something in and out of memory
- Flush pages from the disk cache to the hard disk
- Perform VFS Checkpointing

Kernel Mode Threads — Kernel Level Threads

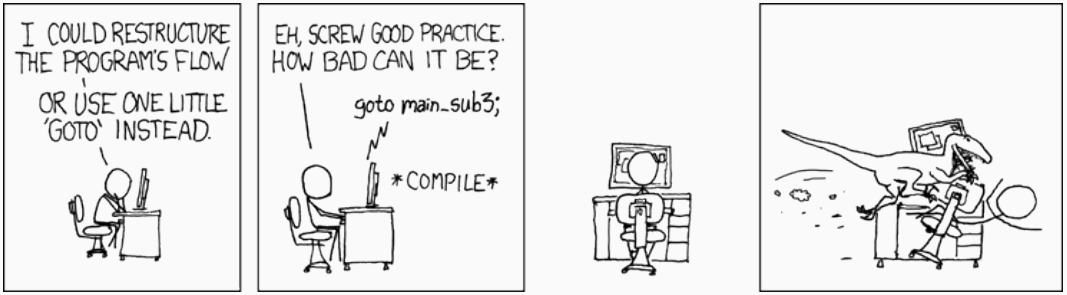
What is the difference?

- Kernel Level Thread: The kernel knows about and manages the thread
- Kernel Mode Thread: The thread runs in *kernel mode*

Why would you need kernel *mode* threads?

- Sometimes the OS needs to do some housekeeping that doesn't belong in a system call
- Free some pages, swap something in and out of memory
- Flush pages from the disk cache to the hard disk
- Perform VFS Checkpointing
- ...

F R A G E N?



XKCD 292 - Goto

Bis nächste Woche :)