Betriebssysteme

5. Tutorium - Threading, Segmentation, MM

Peter Bohner

28. November 2024

ITEC - Operating Systems Group

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

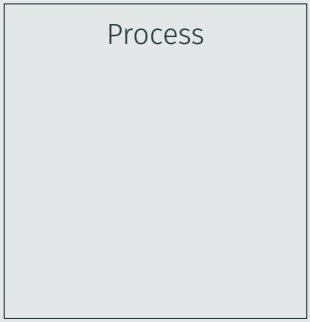
Spawn a few threads using pthreads!

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()
```

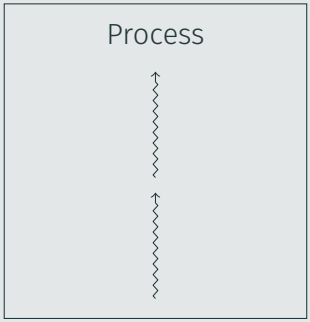


Problems?

Thread models — One To One

One To One

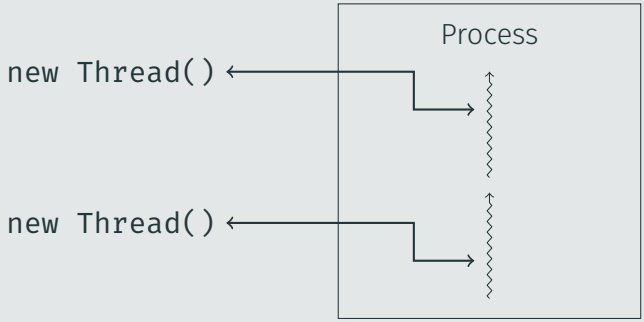
```
new Thread()

new Thread()
```



Problems?

Thread models — One To One

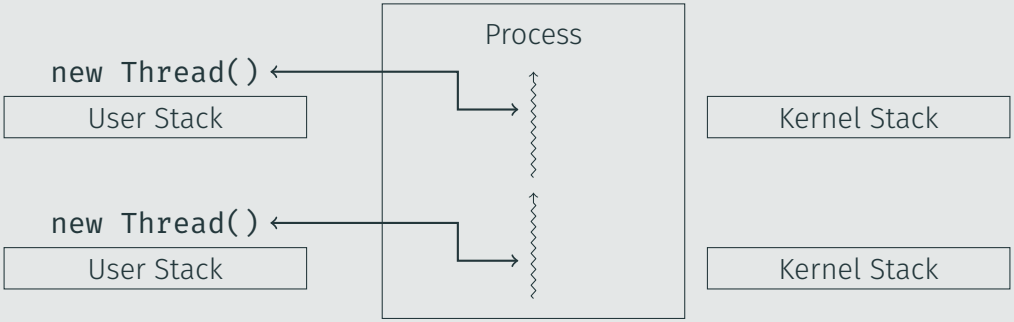
One To One



Problems?

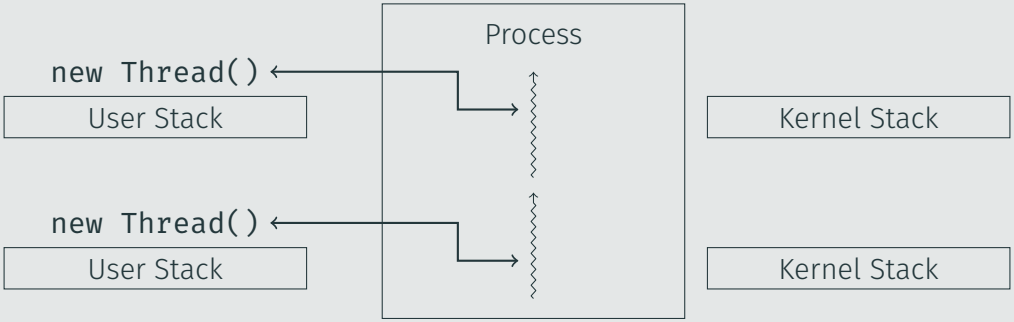
Thread models — One To One

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*

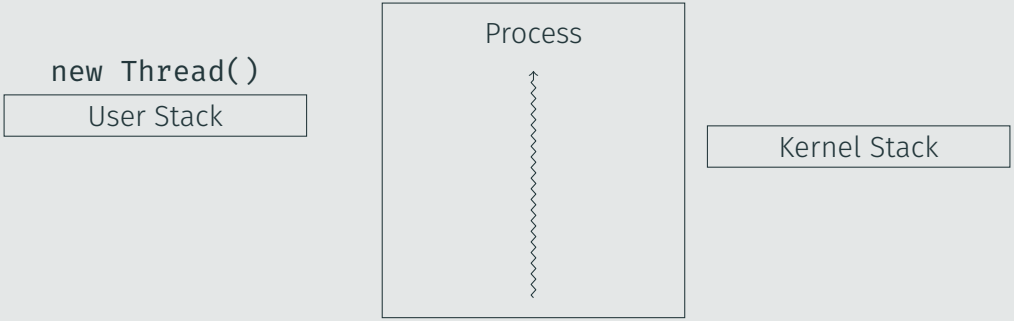
Thread models - One To One

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*!!

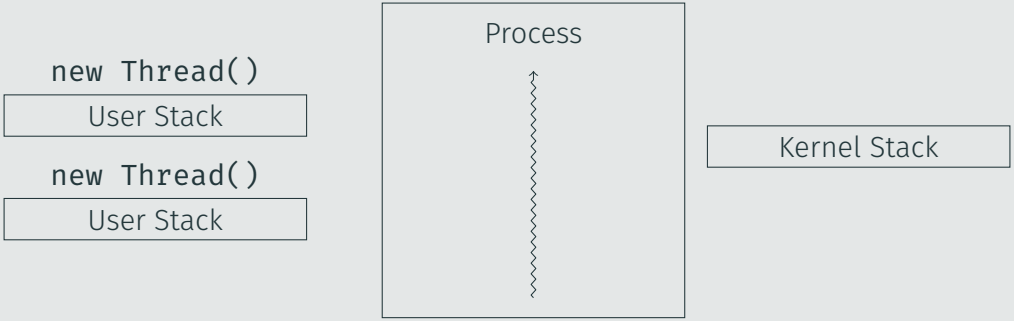
Thread models

Many to One



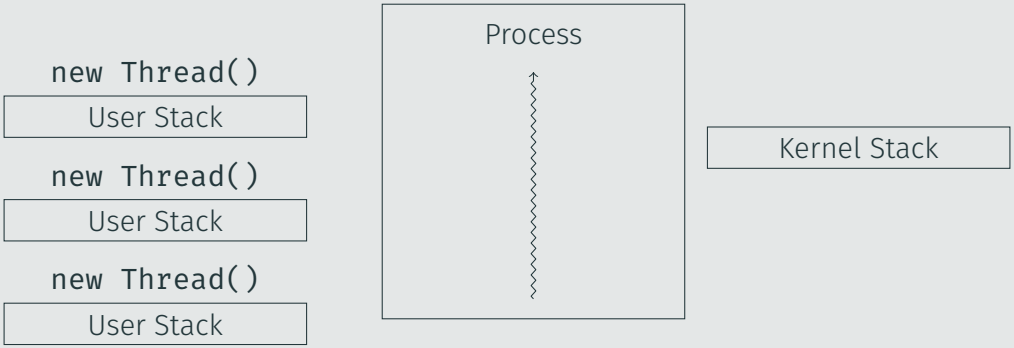
Thread models

Many to One



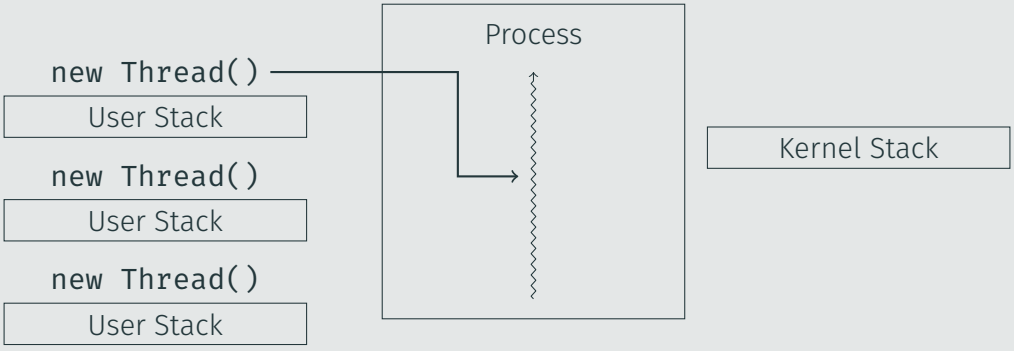
Thread models

Many to One



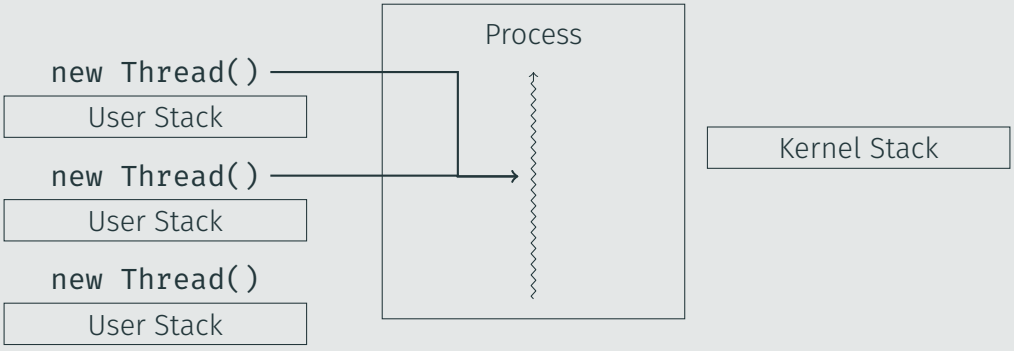
Thread models

Many to One



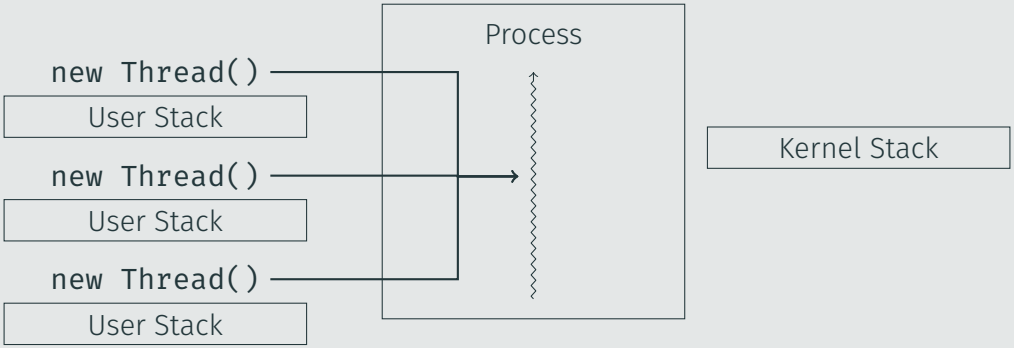
Thread models

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

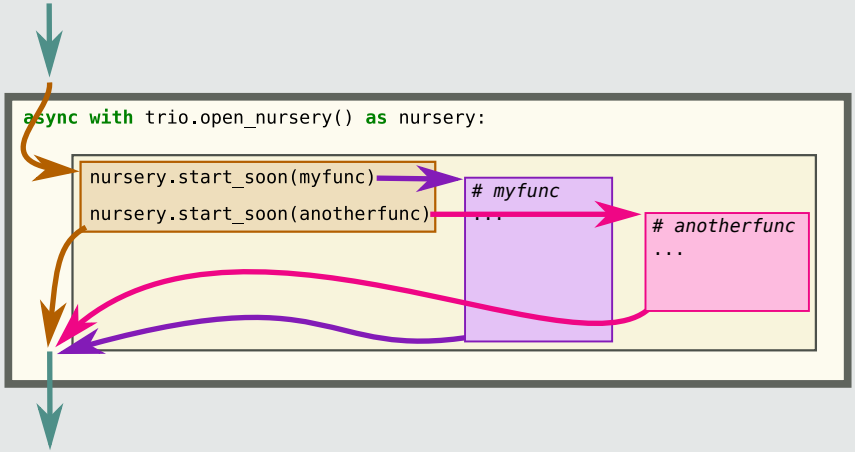
And what about threads?

- 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*

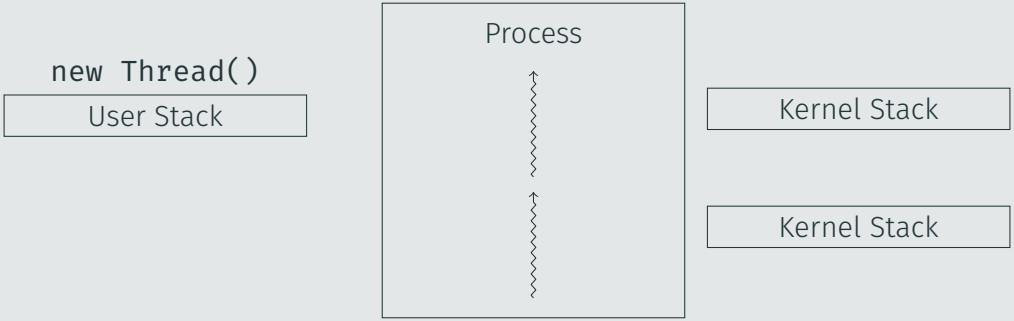
Further reading:

Notes on Structured Concurrency

ULTs and Structured concurrency in Java - Project Loom

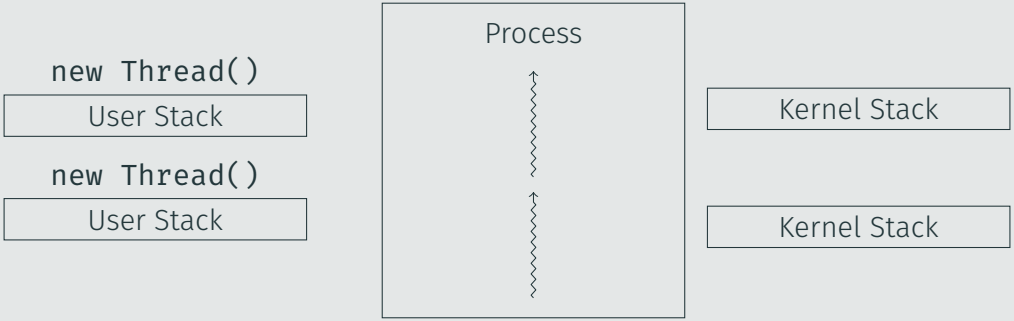
Thread models - Many To Many

Many To Many



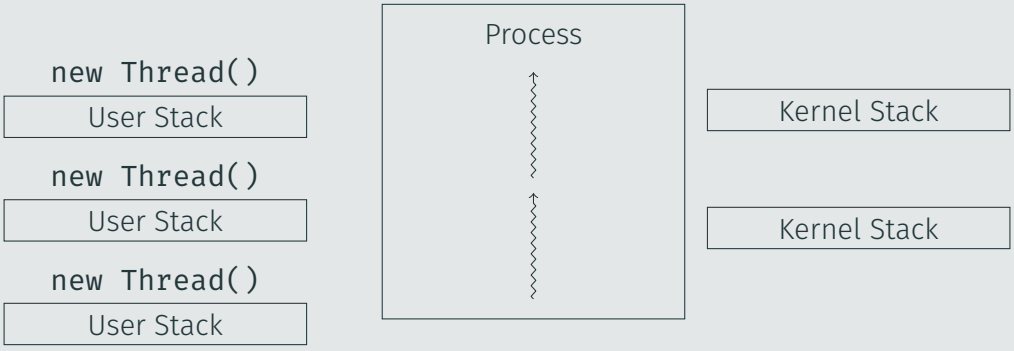
Thread models - Many To Many

Many To Many



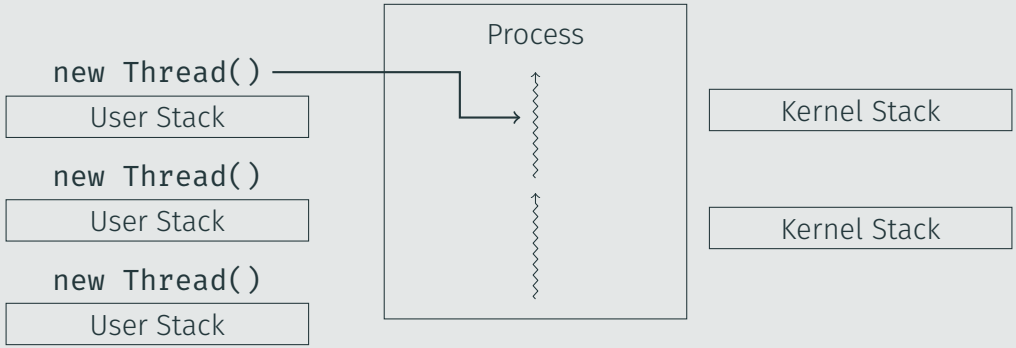
Thread models - Many To Many

Many To Many



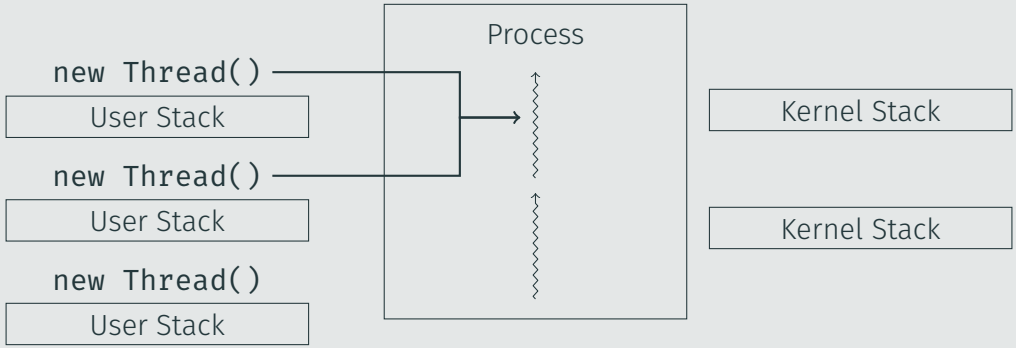
Thread models - Many To Many

Many To Many



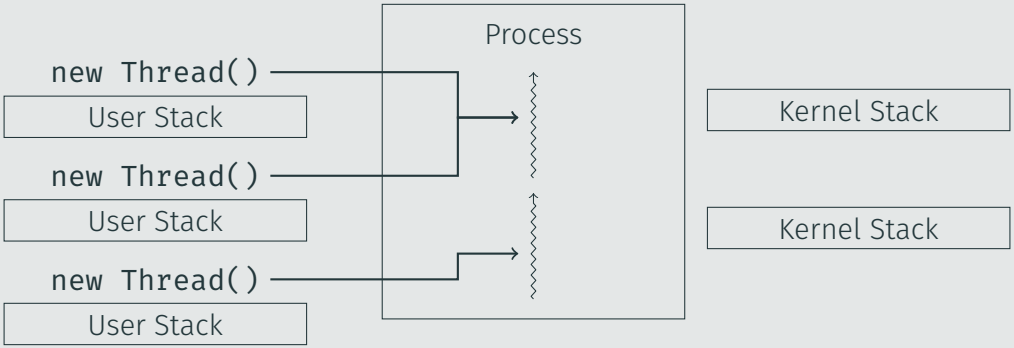
Thread models - Many To Many

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

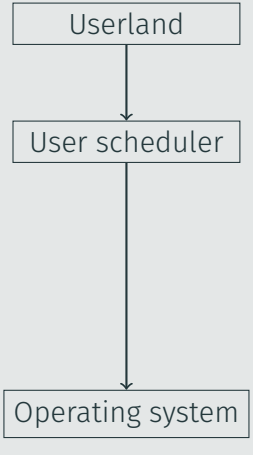
Problems and benefits of *Many To Many*?

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.

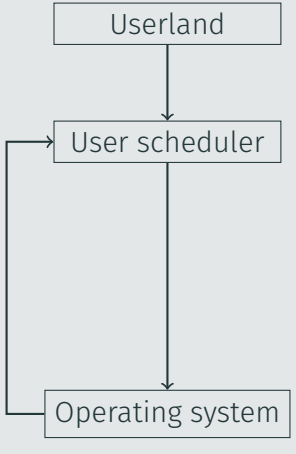


And preemption is now possible, which might complicate user code.

Thread models - Many To Many

Problems the second

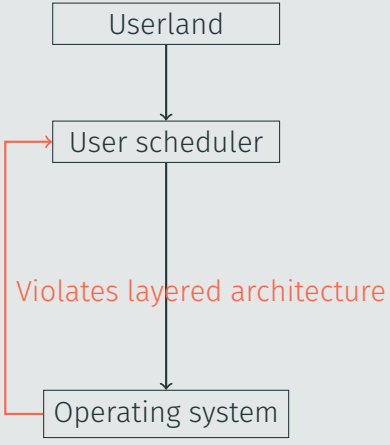
You have all attended SWT 1! So let's have a look.



Thread models - Many To Many

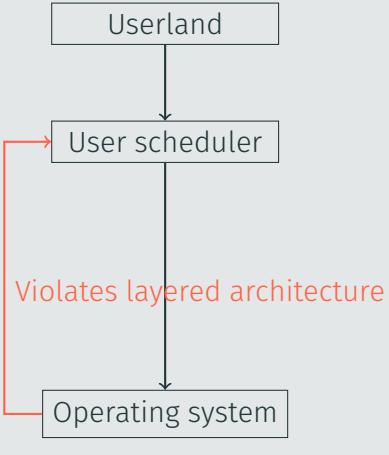
Problems the second

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?“

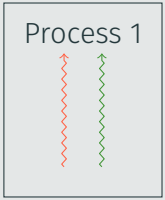
- Program structure (e.g. Structured Concurrency, channels or just easier pipelines)

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?“

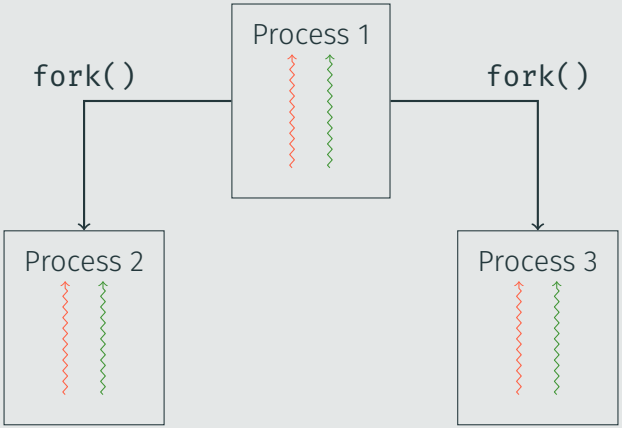
- Program structure (e.g. Structured Concurrency, channels or just easier pipelines)
- The same or higher I/O throughput if on an abstracted platform

Should a fork do this?



Threads — Forking

Should a fork do this?



Threads — Forking

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?

Threads — Forking

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
- ...

Stackframes

Stackframes

Let's look at some code

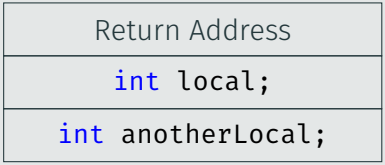
```
1 int foo(int a, int b) {
2   int local = a + b;
3   int anotherLocal = 'a';
4   return local + anotherLocal;
5 }
```

Stackframes

What is this? What does it contain for `int foo(int,int)`?

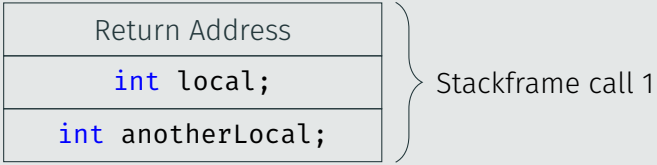
Stackframes

What is this? What does it contain for `int foo(int,int)`?



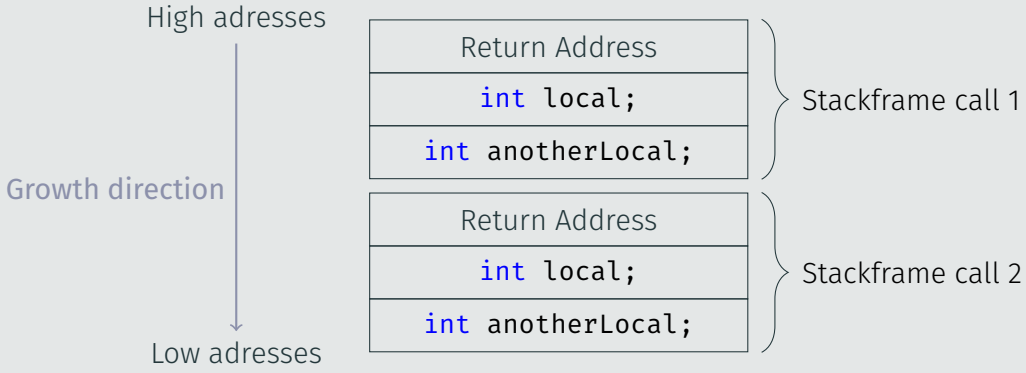
Stackframes

What is this? What does it contain for `int foo(int,int)`?



Stackframes

What is this? What does it contain for `int foo(int,int)`?



Stackframe

How are arguments passed?

Stackframe

How are arguments passed?

Registers!

How are arguments passed?

Registers! What happens for

```
1 void foo(int a, int b, int c, int d, int e, int f, int g, int h);
```

How are arguments passed?

Registers! What happens for

```
1 void foo(int a, int b, int c, int d, int e, int f, int g, int h);
```

We do not have enough registers!

Fixing...

Stackframe

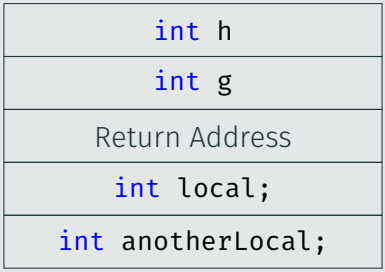
How are arguments passed?

Registers! What happens for

```
1 void foo(int a, int b, int c, int d, int e, int f, int g, int h);
```

We do not have enough registers!

Fixing...



Popping frames

Popping frames

How could you release the space? I.e. figure out by how much to decrement the stackpointer?

Popping frames

Popping frames

How could you release the space? I.e. figure out by how much to decrement the stackpointer?

The compiler surely knows how much space the local variables take up, right?

Popping frames

How could you release the space? I.e. figure out by how much to decrement the stackpointer?

The compiler surely knows how much space the local variables take up, right?

No! C has e.g. *variable-length arrays (VLA)*: `new int[someVariable]`.

$\Rightarrow$ Store the address at the start of the frame!

Popping frames

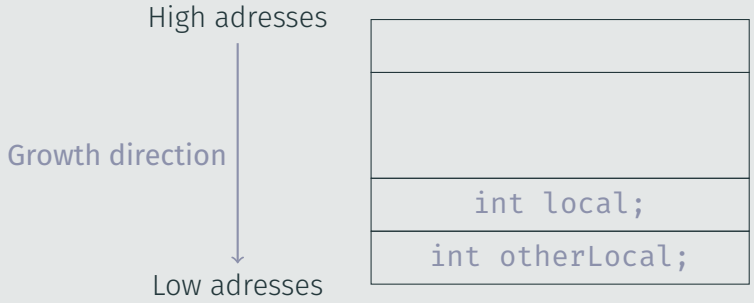
How could you release the space? I.e. figure out by how much to decrement the stackpointer?

The compiler surely knows how much space the local variables take up, right?

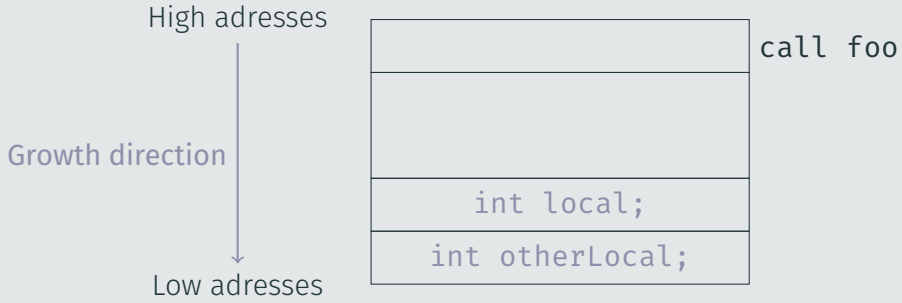
No! C has e.g. *variable-length arrays (VLA)*: `new int[someVariable]`.

⇒ Store the address at the start of the frame! Enter: `rbp`, the base pointer.

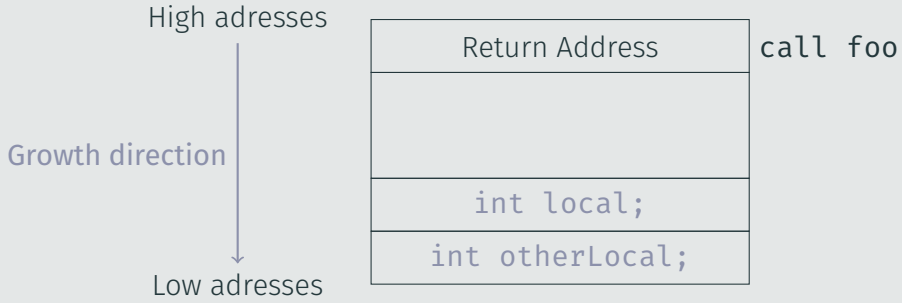
Base pointer and the function (Pro|Epi)log



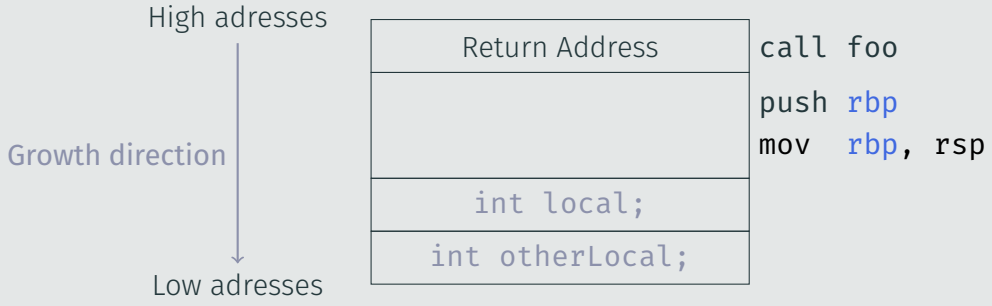
Base pointer and the function (Pro|Epi)log



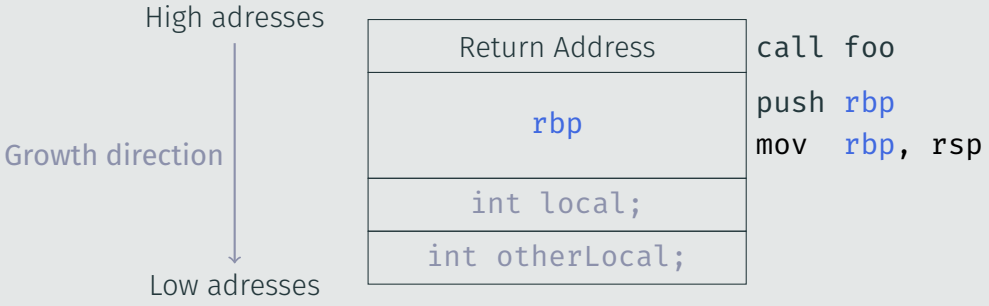
Base pointer and the function (Pro|Epi)log



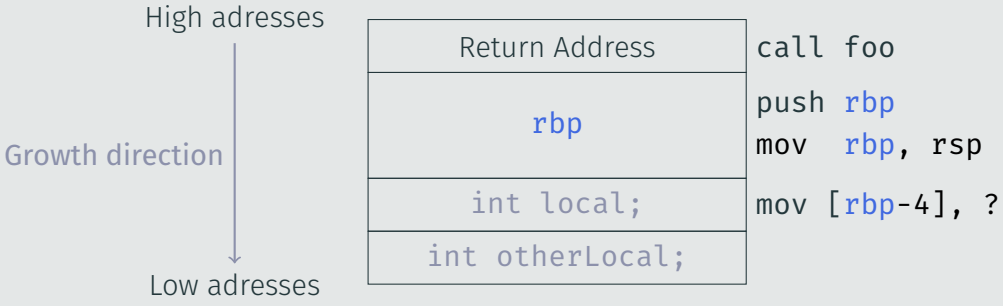
Base pointer and the function (Pro|Epi)log



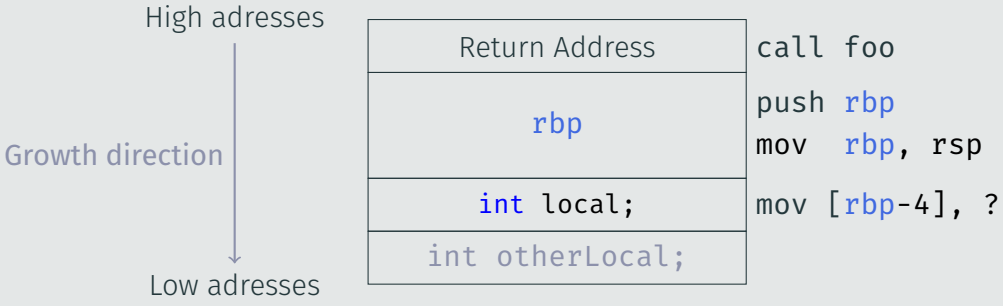
Base pointer and the function (Pro|Epi)log



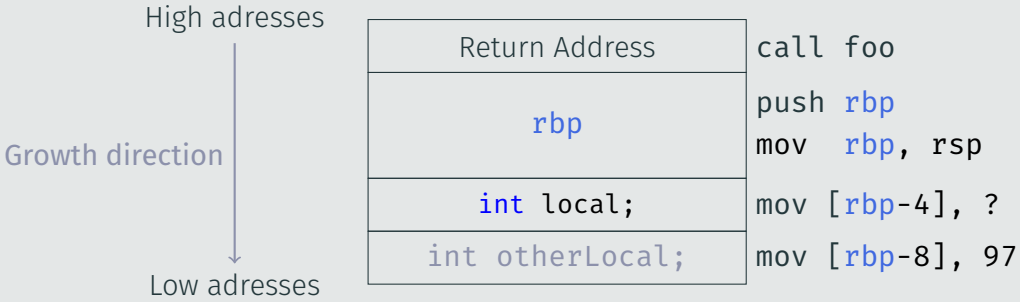
Base pointer and the function (Pro|Epi)log



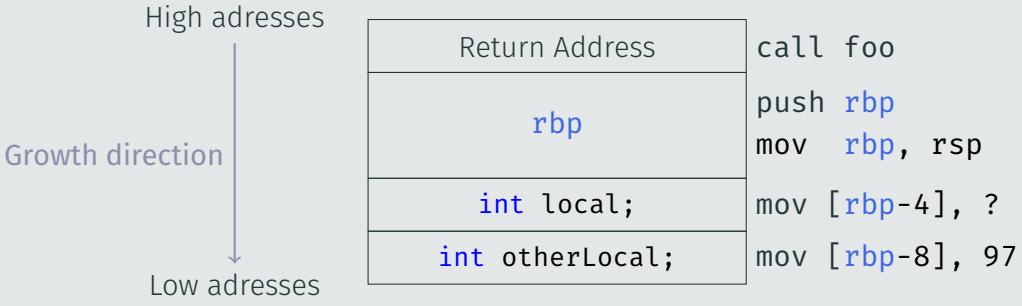
Base pointer and the function (Pro|Epi)log



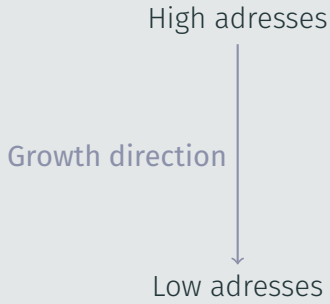
Base pointer and the function (Pro|Epi)log



Base pointer and the function (Pro|Epi)log

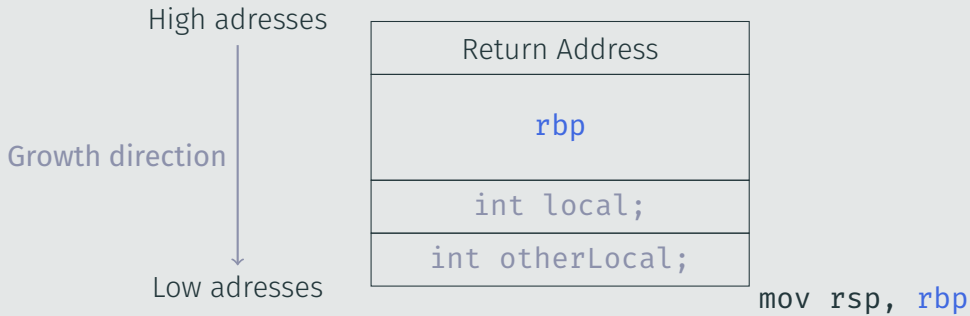


Base pointer and the function (Pro|Epi)log

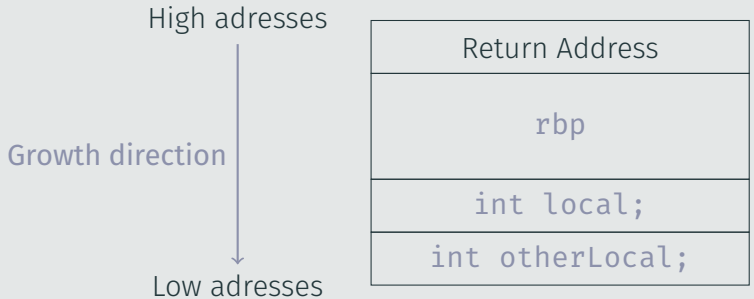


Return Address	`call foo`
`rbp`	`push rbp` `mov rbp, rsp`
`int local;`	`mov [rbp-4], ?`
`int otherLocal;`	`mov [rbp-8], 97`
	`mov rsp, rbp`

Base pointer and the function (Pro|Epi)log

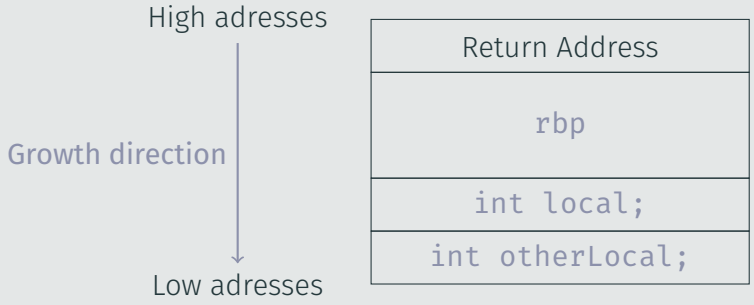


Base pointer and the function (Pro|Epi)log



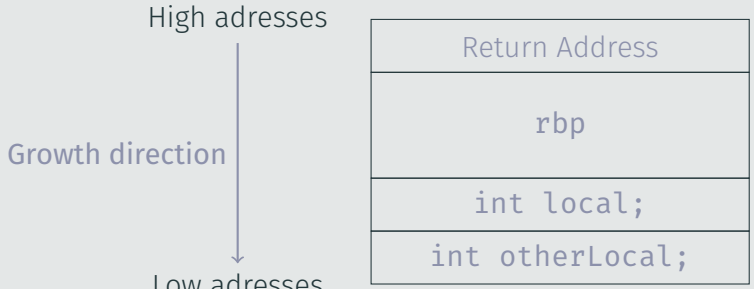

```
mov rsp, rbp
pop rbp
```

Base pointer and the function (Pro|Epi)log




```
mov rsp, rbp
pop rbp
ret
```

Base pointer and the function (Pro|Epi)log




```
mov rsp, rbp
pop rbp
ret
```

Segmentation

Segmentation

Where have you seen that word before while sadly staring at your screen?

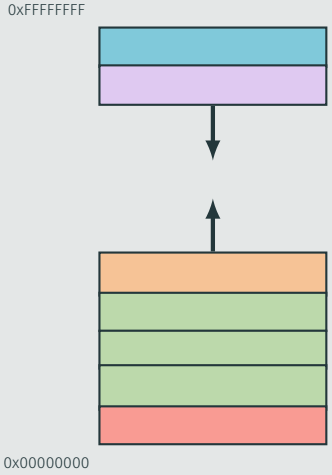
Segmentation

Where have you seen that word before while sadly staring at your screen?

```
> Segmentation fault (core dumped)
```

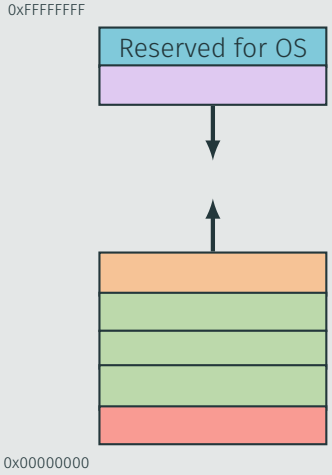
Segmentation

A few example segments



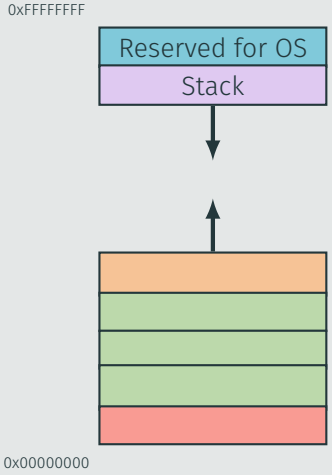
Segmentation

A few example segments



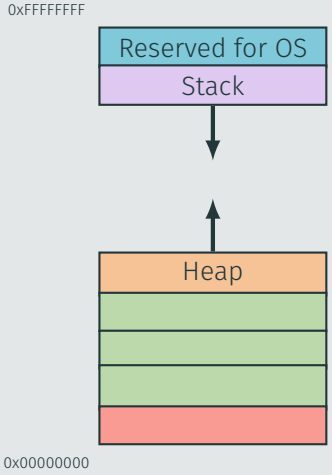
Segmentation

A few example segments



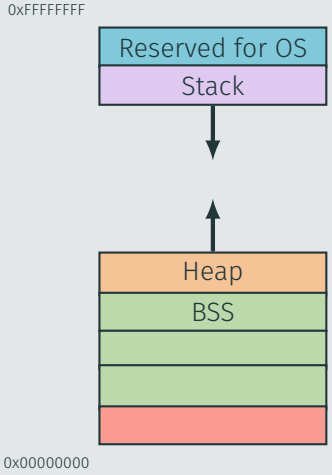
Segmentation

A few example segments



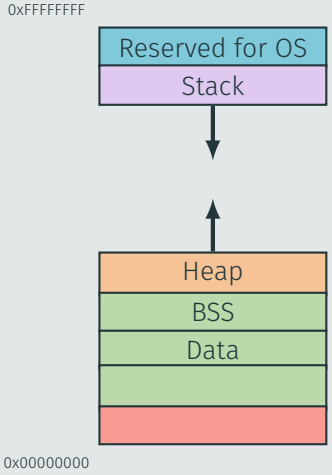
Segmentation

A few example segments



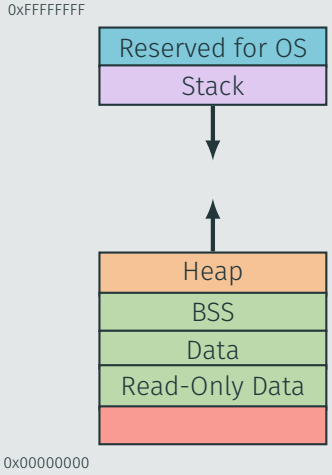
Segmentation

A few example segments



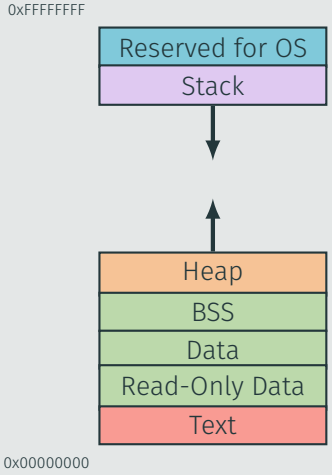
Segmentation

A few example segments



Segmentation

A few example segments



A Virtual Address

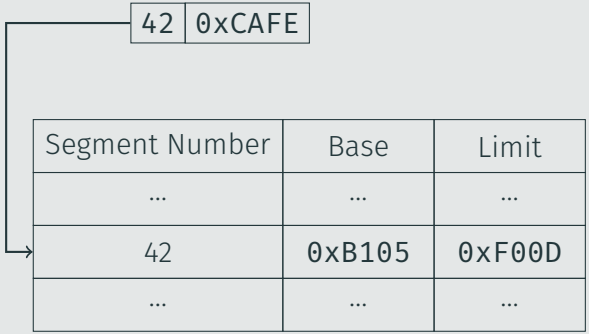
What does it look like?

42	0xCAFE

Segment Number	Base	Limit
...	...	...
42	0xB105	0xF00D
...	...	...

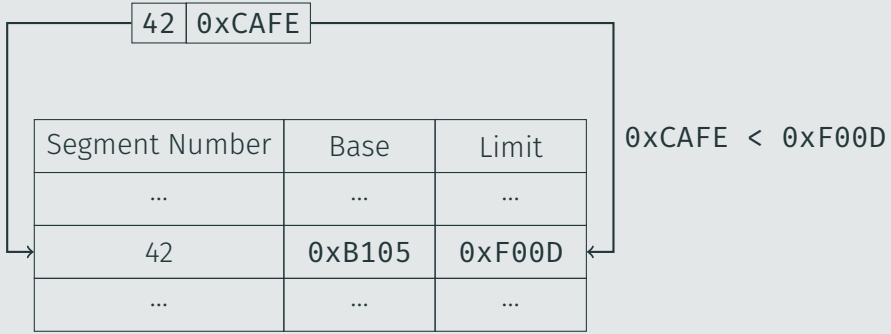
A Virtual Address

What does it look like?



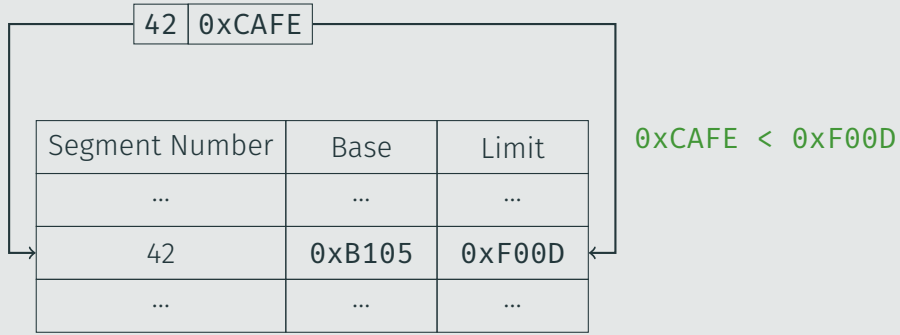
A Virtual Address

What does it look like?



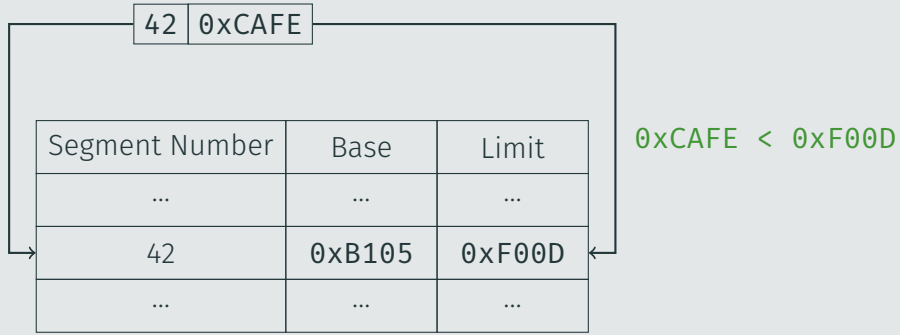
A Virtual Address

What does it look like?



A Virtual Address

What does it look like?




```
0xB105 + 0xCAFE = 0x17C03
```

And let's try it

Segments

Segment Number	Base	Limit
0	0xdead	0x00ef
1	0xf154	0x013a
2	0x0000	0x0000
3	0x0000	0x3fff

Your task

Virtual Address	Segment Number	Offset	Valid?	Physical Address
	3	0x3999		
0x2020				
		0x0204	yes	
			yes	0xf15f

And let's try it

Solution

Virtual Address	Segment Number	Offset	Valid?	Physical Address
`0xf999`	3	`0x3999`	yes	`0x3999`
`0x2020`	0	`0x2020`	no	Offset outside limit
`0xc204`	3	`0x0204`	yes	`0x0204`
`0x400b`	1	`0x000b`	yes	`0xf15f`

Memory Management Basics

Physical And Virtual Addresses

And once again: What is the difference?

Physical And Virtual Addresses

And once again: What is the difference?

- We assume no 1:1 mapping (i.e. we have virtual memory)

Physical And Virtual Addresses

And once again: What is the difference?

- We assume no 1:1 mapping (i.e. we have virtual memory)
- *All program addresses are virtual*

Physical And Virtual Addresses

And once again: What is the difference?

- We assume no 1:1 mapping (i.e. we have virtual memory)
- *All program addresses are virtual*
- Mapped to *physical* addresses as needed by the memory management unit

Fragmentation

What is *internal* fragmentation?

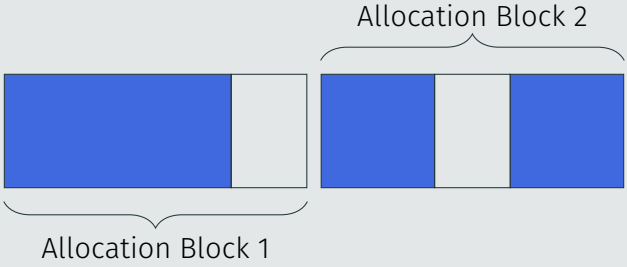
Fragmentation

What is *internal* fragmentation?

Internal, i.e. *within* a block

Fragmentation

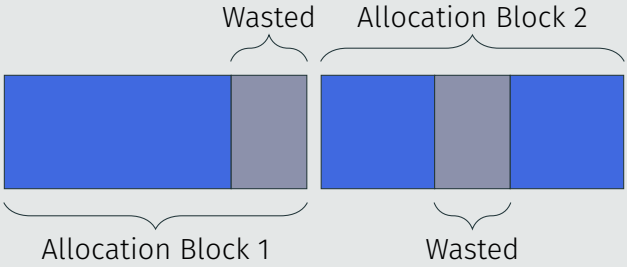
What is *internal* fragmentation?



Internal, i.e. *within* a block

Fragmentation

What is *internal* fragmentation?



Internal, i.e. *within* a block

Fragmentation

What is *external* fragmentation?

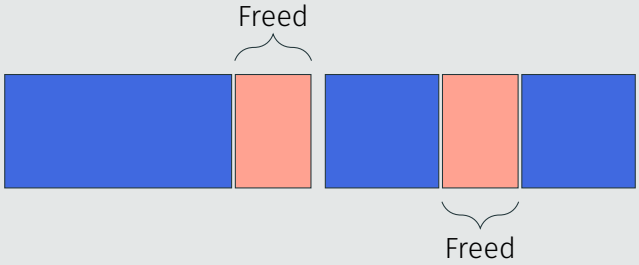
Fragmentation

What is *external* fragmentation?

External, i.e. due to *external factors* (different time-to-free)

Fragmentation

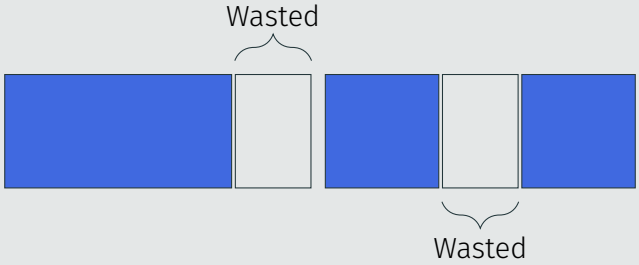
What is *external* fragmentation?



External, i.e. due to *external factors* (different time-to-free)

Fragmentation

What is *external* fragmentation?



External, i.e. due to *external factors* (different time-to-free)

Fragmentation

Can you have *both* types at the same time?

Fragmentation

Can you have *both* types at the same time?

Yes!

Fragmentation

Can you have *both* types at the same time?

Yes!

- Allocate in *chunks* by e.g. rounding up to 2^x

Fragmentation

Can you have *both* types at the same time?

Yes!

- Allocate in *chunks* by e.g. rounding up to 2^x
- Have different lifetimes

Fragmentation

Can you have *both* types at the same time?

Yes!

- Allocate in *chunks* by e.g. rounding up to 2^x
- Have different lifetimes

$\Rightarrow$ Wasteful allocations scattered throughout RAM

Fragmentation

What do we do now? This sounds bad!

Fragmentation

What do we do now? This sounds bad!

Fragmentation

What do we do now? This sounds bad!

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

$\Rightarrow$ They are all garbage now!

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

$\Rightarrow$ They are all garbage now!

- Works just fine in languages with indirections (e.g. garbage collection)

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

$\Rightarrow$ They are all garbage now!

- Works just fine in languages with indirections (e.g. garbage collection)
- Also works for segments in physical memory! How?

Fragmentation

What do we do now? This sounds bad!

This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

$\Rightarrow$ They are all garbage now!

- Works just fine in languages with indirections (e.g. garbage collection)
- Also works for segments in physical memory! How? Update base addresses in MMU

Memory allocation policies

Some common strategies

Which strategies for finding free blocks do you know?

First Fit,

Some common strategies

Which strategies for finding free blocks do you know?

First Fit, Best Fit,

Some common strategies

Which strategies for finding free blocks do you know?

First Fit, Best Fit, Worst Fit

Some common strategies

Which strategies for finding free blocks do you know?

First Fit, Best Fit, Worst Fit

First Fit

Pick the first block that is large enough

Some common strategies

Which strategies for finding free blocks do you know?

First Fit, Best Fit, Worst Fit

First Fit

Pick the first block that is large enough

Best Fit

Pick the *smallest* block that fits

Some common strategies

Which strategies for finding free blocks do you know?

First Fit, Best Fit, Worst Fit

First Fit

Pick the first block that is large enough

Best Fit

Pick the *smallest* block that fits

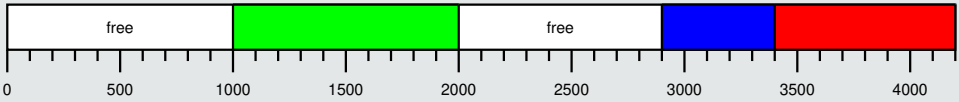
Worst Fit

Pick the *largest* block that fits

Let's try them

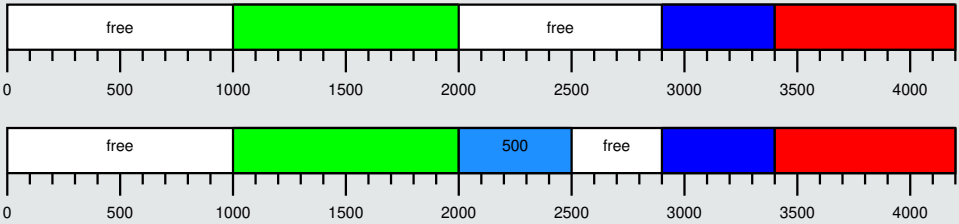
Best fit

Allocate 500, 1200, and 200, fail if not possible.



Best fit

Allocate 500, 1200, and 200, fail if not possible.

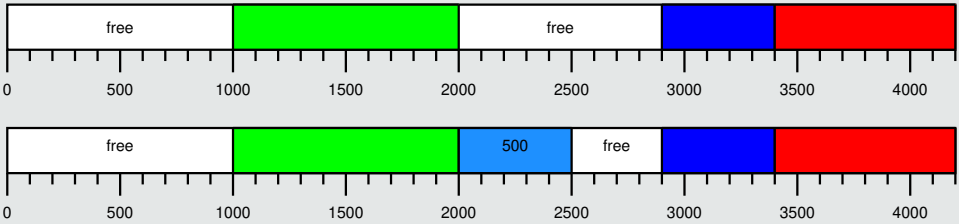


And compact it to fit the next one!

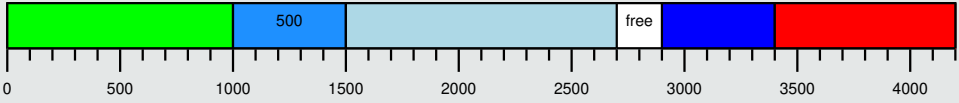
Let's try them

Best fit

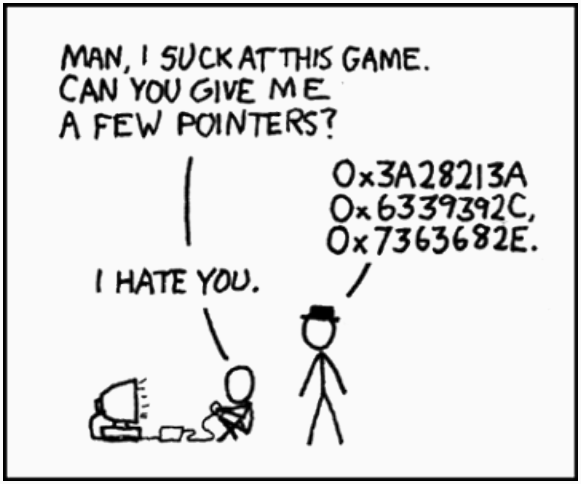
Allocate 500, 1200, and 200, fail if not possible.



And compact it to fit the next one!



Ende



XKCD 138 - Pointers

F R A G E N?

Bis nächste Woche :)