0 Contents

1 C

1.1 fixed size types

- int8_t, int16_t, int32_t, int64_t
  fixed integers with bit count
- intmax_t max size int on platform
- intptr_t signed integer with the size of an address on this platform
- uint8_t, uintptr_t unsigned versions
- size_t
  this is used in containers, the reson for this is that *this has the max size that for example an array can be.*
  This is unsigned!

1.2 Addition of pointers

If you try to add or subtract 2 pointers to get the amount of sizeof(t) difference, then you can only do this with the exact same type, something like signed int and unsigned int will not work!

```
int32_t *y = 100;
int32_t *x = 120;
ptrdiff_t z = x - y; // z == 5
uint32_t *u = 120;
ptrdiff_t p = u - y;  // Error: Different ptr types
```

1.3 Index Operator on Pointers

You can index on pointers like an array, this can be used to get elements on any object.
Note that you have to manually make sure to stay within the bounds of that object, as otherwise you will have *undefined behavior.*

```
int32_t x = 0;
int32_t *y = &x;
y[0] = 0x42;       // same as: x = 0x42;
(&x)[0] = 0x42;    // same
0[&x] = 0x42;      // same
100[200] = 0x42;   // Error: no address
```

1.4 Padding

When you mix and match different types of different sizes inside of a struct, then the compiler will include padding based on the bigger type:

```
struct  {
char c;      // Offset 0
int32_t x;   // Offset 4 --> Padding
char d;      // Offset 8
} t;         // sizeof t == 12
# structure matters!!
struct  {
char c;      // Offset 0
char d;      // Offset 1
int32_t x;   // Offset 2 --> Padding
} t;         // sizeof t == 6
```

1.5 Forwards Declaration

```
struct Folder;
// Forward-Deklaration
struct File {
struct Folder *parent;
// OK: all pointer types
//
have same size
char name[256];
// OK: fixed size array
};
// --> Type complete
struct Folder {
struct File * file[256]; // OK: fixed size array
};
// --> Type complete
```

2 Operating System APIs

2.1 Basic features of an Operating system

- abstraction and portability
  - define generic APIs that work on all (as many as possible) devices
  - define abstractions that we don't care about -> how are files stored on the disk?
- Isolation and Resource Management
  - Isolate each usecase from each other -> posx
  - Runtime (make it blazingly fast)
  - Memory Management
  - Secondary Storage handling
- Security

2.1.1 Limitations of Portability

While the operating system can define standards, there are often things that we as developers need to consider, for example, while the operating system can define how a user will interact with the keyboard etc, if said device *doesn't have this input*, then your application will not work... Eg. An application meant for touch on the desktop might work, but not properly, and the OS can't really help there.

2.1.2 Limits of Isolation

Often, you want some form of interoperability, or you are basically forced to use that.
For example an application might want the focus of the keyboard, but then a popup appears.
If the application continues taking the focus, then the application now breaks the user experience.

2.2 Processor Privilege

Modern operating systems define a range of instructions that only the kernel is allowed to perform.
This is done to protect the operating system from attacks that might be exploitable via these privlieges.
In this case you run in *user mode*, this is also why anti-cheats are often running in kernel mode,
in this mode, the anti-cheat can access any memory all the time for whatever reason the anti-cheat would like to do so.
Should an application break the rule of user-space, the operating system will be notified and can then kill or otherwise restrict the application.

2.3 Kernel

2.3.1 MicroKernel

Microkernel is the idea that only the *absolutely necessary operations need to be in the kernel*, this means that often,
things like drivers run in the userspace, eg. Radv would be in userspace.
Features:

- Reliability
  less code is easier to maintain, which means a more stable operating system.
- Analysability
  less code is easier to bisect, meaning that bug hunting is easier.
- Performance
  Because drivers are now in a lower priority environment, they can no longer directly access hardware.
  This means that you will have a significant performance hit, which is also the reason that *linux is not a microkernel!*.

In the real world, there is no *real microkernel*, they usually add the necessary functionality of drivers and leave it at that.

2.3.2 Monolithic Kernel

Monolithic kernels have all the base functionality included. This means that you will not need to supply basic functionality to the kernel,
just to get a functional operating system.

- Performance
  Since drivers have direct access to hardware, this means they can run faster!
- Security
  Since drivers have direct access to hardware, this means that misconfigured or malicious hardware,
  can easily infiltrate the kernel
- Reliablity
  More code means more possible bugs, and in the kernel this is worse than in the usermode.

2.3.3 Unikernel

This is a kernel that is made for one specific application, which means *it is an application!*.

- Performance
- Seurity
- Reliablity
- Only one Usecase

2.3.4 Running an instruction in Kernel Mode

When you want to run an instruction in the kernel mode, then you need to do a *syscall*.
The processor will then switch into kernel mode (if the os has given the privelege) and run the instruction in kernel mode.

2.3.5 Syscall (SVC on ARM)

There is only one function to run something in kernel mode, this means that we have to use *codes* instead.
Eg. a syscall with code 60 would be the exit code for a program. -> plox kill me
NOTE: Syscall also doesn't take arguments, therefore you need to place the arguments in registers.
This is exactly why you had to place all these things into registers, when you wanted to print a simple "hello world" in assembly.
The implication: output and input are kernel mode!!

2.4 ABI vs API

*A*pplication *B*inary *I*nterface

- concrete interfaces
- calling conventions
- projection of datastructures

*A*pplication *P*rogramming *I*nterface

- abstract interfaces
- platform/OS independent aspects

2.4.1 ABI in Linux

Calling Conventions for syscall are different for different linux kernels!
This means that you need to compile applications for each kernel!
To counter problems that will appear with this, there is a standard called *Linux Standard Base*, which defines a set of conventions to use.

2.4.2 API in Linux

The proper solution is to use APIs instead, which can be done with languages such as C (and tomorrow Rust).
This means that you *no longer use syscall*, you instead use *C functions*, which work on every kernel, not just on one.

2.5 POSIX

In general, every OS has its own ABI and API.
The unix API has been developed alongside the C API, this lead to the ISO standard.
However, at some point there were multiple standards, which meant the compatability was wrecked again.
Instead, the POSIX standard API was defined, which meant that if you wrote your program POSIX compliant, then it will run on any POSIX OS.

2.5.1 POSIX Conformity

- MacOS: since version 10.5
- Linux, not certified, but somewhat POSIX conform
  Bad: not everything is standard, but we all know that sometimes you either go your own way, or nothing happens -> matrix
- Windows: no :)
- BSD: yes

2.6 Man Pages

Man pages provide information about a POSIX system, it *is made of 9 parts:*

1. Executable Programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually fond in /dev)
5. File formats and conventions, e.g. /etc/passwd
6. Games lol
7. Miscellaneous (including macro packages and convenstions)
8. System administration commands (usually only for root)
9. Kernel routines (not standard)

2.7 Shell

- Made of an ouput and input stream.
- many different shells, bash, dash, zsh, fish, nu
- doesn't need special rights or prerequisites
- Made to call OS functions via text

2.7.1 Arguments in shell

- All arguments are considered strings -> its just an IO stream
- Spaces usually seperate the arguments
- "\" usually used to escape characters -> like space

These arguments are then passed to C or C++ program in the array called *(char **argv)*, and the *(int argc)* variable is the count of arguments.
The first entry in argv is the programname

2.7.2 Env Vars

- Key-Value pair
- Example: MOZ_ENABLE_WAYLAND=1
- can be set for the shell in .zshrc/.bashrc etc
- can be set for environment in xdg-config/environment.d, /etc/environment or .profile, /etc/profile
- Order is root first then local, if variable is set twice, then you will overwrite it!

To use variables in C: getenv, putenv, setenv, unsetenv
Technically, there is a variable called *environ*, which is an array of null pointers to strings which is 0 terminated.
However, this variable *is not defined!!*

getenv

```c
//definition
char * getenv (const char * key)

char *value = getenv("PATH");
// value = "/home/ost/bin:/home/ost/.local/bin"
// returns nullptr -> 0 if variable not set
```

setenv

```c
// definition
int setenv(const char *key, const char *value, int overwrite);

int ret = setenv("HOME", "/usr/home", 1);
// returns code 0 if ok, error code otherwise
// sets variable with value, overwrite if not 0
// error if variable doesn't exist!
```

unsetenv

```c
// definition
int unsetenv(const char *key);

int ret = unsetenv("HOME");
// returns 0 if ok, error code otherwise
// removes the env variable
```

putenv

```c
//definition
int putenv (char * kvp)

int ret = putenv("HOME=/usr/home");
// adds env variable pair
// returns 0 if ok, error code otherwise
// if variable already exists -> overwrite
```

In general, use env vars as a flag to configure things, not as a config that you *need* to configure.
For other operating systems this is done via other management solutions like windows -> registry.... haah get shit on windoof users

3 Filesystems

Essentially an API that makes sure that applications do not need to understand/know how the hardware works

3.1 Files

Files have 2 parts:

- Data
  Sequence of bytes that represent the file

- Metadata
  visible for users: size, date, owner, filetype hint
  not visible for users: place on hardware, connection of blocks on hardware

3.1.1 FileTypes

- ending after . -> .pdf
- File-endings have pretty much no point, they are just for the user
- File-endings are used as a hint to open said file with a specific program
- Type can be deducted via *magic numbers* inside of the file

General Advice:

- Data is trash unless proven otherwise
- Validate ALL data
- General advice: when decoding a file, continuously check if the file is really a pdf, or whatever you expect, test it to be.
  A few lines may not be enough to prove that it really is of said type.

3.2 Directories

- Essentially a file with a special type
- Each directory other than root directory has exactly one parent -> tree
- root directory is often / -> penguinOS

3.2.1 Special directories

- . -> reference on self
- .. -> parent reference
  for root this would just be itself..
- $PWD -> working directory. getcwd in C
- chdir / fchdir -> cd in C

Example for getting the current working directory in C:

```c
int main (int argc, char** argv) {
  char *wd = malloc (PATH_MAX);
  getcwd (wd, PATH_MAX);
  printf ("Current WD is %s", wd);
  free (wd);
  return 0;
}
```

3.2.2 Paths

- absolute path /home/dashie/../dashie/.zshrc
  The reason for the .. in the middle is that a canonical path is an absolute path, but without either .. or . in the middle.
- relative path ../ai-app/ai-app.tex
- canonical path /home/dashie/.zshrc
  can be received with "realpath".

3.2.3 Max Path

POSIX systems can have different max path lengths, these are defined in "limits.h".
Macros:

- NAME_MAX max length of filename (exclusive 0 termination)
- PATH_MAX max length of path (inclusive 0 termination)
- _POSIX_NAME_MAX minimal value of NAME_MAX according to posix
- _POSIX_PATH_MAX minimal value of PATH_MAX according to posix

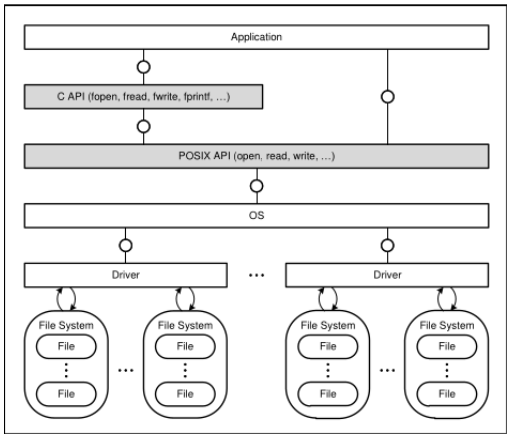
3.2.4 Rights

There are 3 permission categories, each having 3 groups, *read-write-execute* with *owner-group-other*
Technically there is one more bit, the sticky bit, but this is not really used anymore.
110-110-110 -> owner and group can do all, other can't do anything
Note that this can also be done with numbers -> 7 would be all -> 111

```
drwxr-xr-x dashie dashie 884 B  Fri Nov 11 11:15:56 2022  Desktop
drwxr-xr-x dashie dashie 778 B  Mon Feb 20 11:03:11 2023  Documents
drwxr-xr-x dashie dashie 274 B  Tue Mar  7 14:13:01 2023  Downloads
drwxr-xr-x dashie dashie 146 B  Tue Dec 20 12:33:04 2022  Games
drwxr-xr-x dashie dashie 1.2 KB Mon Mar  6 13:35:10 2023  gits
```

These rights are also stored as Macros in POSIX -> "sys/stat.h"
These can be chained with | -> S_IRWXU | S_IRGRP

4 File API



The main difference with the POSIX API to the C API, is that the POSIX API gives us raw data, without any interpretation of the data, while the C API supports more specific things such as sockets, decoding etc.

4.1 POSIX File API

General Info:

- All file functions are declared in <unistd.h> and <fcntl.h>
- error codes can be checked with "errno"
- raw data
- should only be used for binary data, not for anything that needs interpretation

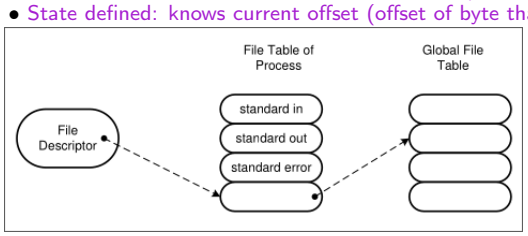
4.1.1 Usage of errno

```
if (chdir("docs") < 0) { // type is int
  if (errno == EACCESS) { // EACCESS defined in the function documentation
    printf ("Error: %s\n", strerror (errno));
    // or you can use perror
    perror ("Error"); // this makes use of the standard error stream
  }
}
```

Note, not all function set this flag, and should be used immediately, as other function will overwrite it.
Codes for the error are directly defined in the function documentation.
Returns the address of a string, which describes the code in text.
perror is the same as strerror but with a special error stream.

4.1.2 File-Descriptor

- valid within a process
- indexed in a table of all open files in a process
- Process file table is indexed in the global table of all open files
- Receives data in order to identify physical file (correct hardware with correct driver)
- State defined: knows current offset (offset of byte that will be read next)



In each process open file index for a process there are 3 predefined file descriptors

```
// STDIN_FILENO = 0  -> standard input
// STDOUT_FILENO = 1 -> standard output
// STDERR_FILENO = 2 -> standard error
```

4.1.3 Opening files with POSIX API

```
int open (char *path, int flags, ...)
```

- O_RDONLY: Read only
- O_RDWR: read and write
- O_CREAT: create file if not exists, needs another parameter for access rights
- O_APPEND: set offset to end of file before each write access
- O_TRUNC: set length of file to 0

4.1.4 Close files with POSIX API

```
int close(int fd)
```

deallocates file descriptor fd, which can now be used by other functions.
returns 0 if ok, and -1 for error

4.1.5 Usage of open and close

```
int fd = open("filename.file", O\_RDONLY);
if (fd < 0) {
  // le error handling
}
// do something with file
close(df);
```

4.1.6 Read data with POSIX API

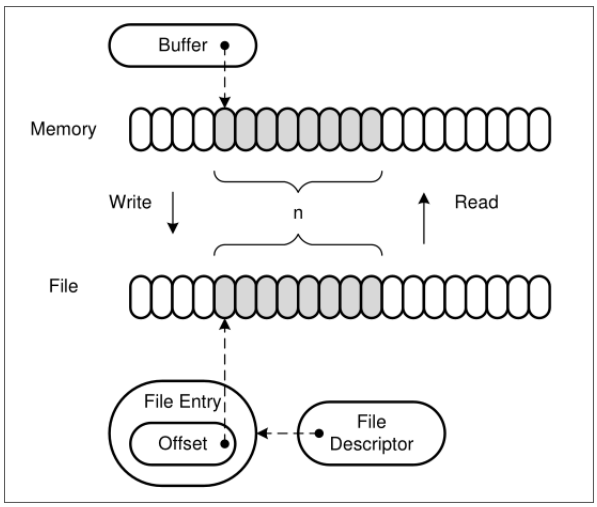
```
ssize_t read (int fd, void *buffer, size_t n)
// ssize_t is a signed size_t
```

- tries to copy the next n(parameter) bytes to current offset from fd to the buffer
- returns count of read bytes or -1 when error
- blocks thread until: n bytes are copied, error occurs, end of file has been reached
- increments offset of fd by the amount of read bytes

4.1.7 Write data with POSIX API

```
ssize_t write (int fd, void *buffer, size_t n)
// ssize_t is a signed size_t
```

- tries to copy the n bytes from the buffer to the offset on fd
- returns count of written bytes or -1 on error
- blocks thread until n bytes are written, error occurs, or the end of file has been reached
- increments offset of fd by the amount of bytes written



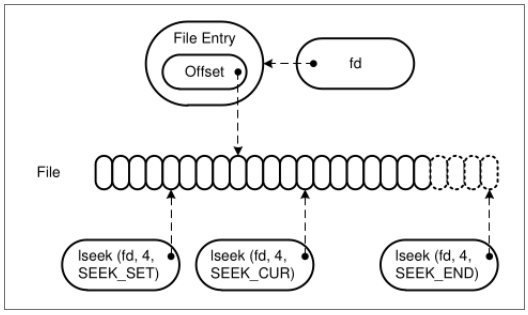
4.1.8 Jump in a file with POSIX API

```
off_t lseek (int fd, off_t offset, int origin)
```

- parameter offset that should be the new offset of fd
- origin = SEEK_SET for start of file
- origin = SEEK_CUR for current offset
- origin = SEEK_END for end of file
- returns new offset or -1 on error

Example usage:

```
lseek (fd, 0, SEEK_CUR) return current offset
lseek (fd, 0, SEEK_END) return size of file
lseek (fd, n, SEEK_END) go beyond end of file, which will make write put 0s in this space if called.
```



4.1.9 pread and pwrite in POSIX API

```
ssize_t pread (int fd, void *buffer, size_t n, off_t offset)
ssize_t pwrite (int fd, void *buffer, size_t n, off_t offset)
```

These are alternatives to write and read, that do not change the offset, however, this means that you will need to define where we are currently!

4.1.10 windoof proprietary paths

- \instead of / because fuck you
- \also needs to be escaped if you want to write it
- root directory per disk instead of per system
- C is the default disk, A and B were reserved for floppy disks
- functions in windoof:
  - open -> CreateFile
  - read -> ReadFile
  - write -> WriteFile
  - lseek -> SetFilePointer
  - close -> CloseHandle

4.1.11 Example for reading and writing data in POSIX API

```
#define N 32
char buf[N];
char spath[PATH_MAX];
char dpath[PATH_MAX];

/* get paths from somewhere */

int src = open(spath, O_RDONLY);
int dst = open(dpath, O_WRONLY | O_CREAT, S_IRWXU);
ssize_t read_bytes = read(src, buf, N);
write(dst, buf, read_bytes);
close(src);
close(dst);
```

4.2 C Stream API

Idea: Operating systems do things differently, even something as simple as a newline is handled differently, so we need an API that can translate this to the correct symbol:

```
// Windows: \r \n = 13d 10d = 0Dh 0Ah
// Linux: \n = 10d = 0Ah
// Mac OS: \r = 13d = 0Dh (before Mac OSX, now just like penguinOS)
```

- OS independent
- stream-based: symbol-oriented

- can be buffered or unbuffered
  dependent on the implementation, transparent for applications
- normally buffered for files
  independently transfers data-blocks between files and buffers
- Has a file Position indicator
  - for buffered streams: defines position in buffer
  - for unbuffered streams: is the offset in the file-descriptor

4.2.1 FILE datastructure

- has information about a stream
- should not be used directly, instead only per pointers that are created via the C-API
- should not be copied, pointer can be used as ID by the API
- three predefined standard-streams
  - FILE *stdin
  - FILE *stdout
  - FILE *stderr

4.2.2 open file with C-API

```
FILE * fopen (char const *path, char const *mode)
// flags
// "r": like O_RDONLY
// "w": like O_WRONLY | O_CREAT | O_TRUNC
// "a": like O_WRONLY | O_CREAT | O_APPEND
// "r+": like O_RDWR
// "w+": like O_RDWR | O_CREAT | O_TRUNC
// "a+": like O_RDWR | O_CREAT | O_APPEND
```

- creates FILE-Object and stream for the file
- returns pointer to created FILE-Object or 0 on error

4.2.3 close file with C-API

```
int fclose (FILE *file)
```

- calls fflush
- closes stream defined by file parameter
- removes file from memory
- returns 0 when ok, otherwise EOF

4.2.4 flush of file with C-API

```
int fflush (FILE *file)
```

- writes content to write from memory into file (if content exists)
- will automatically be called when the buffer is full or file is closed
- returns 0 when of, otherwise EOF

4.2.5 Conversion from POSIX-API to C-API

```
FILE * fdopen (int fd, char const *mode)
// like fopen, but instead of path, we use a file descriptor

int fileno (FILE *stream)
// returns file-descriptor for the stream, or -1 on error
```

4.2.6 read from file with C-API

```
int fgetc (FILE *stream)
```

- reads the next byte from stream *as unsigned char* and returns it as int
- increments the file-position indicator by 1

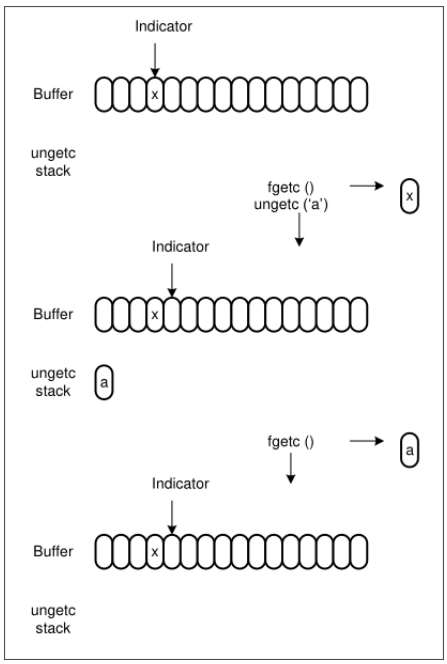
```
char * fgets (char *buf, int n, FILE *stream)
```

- reads to n-1 symbols from stream, or until newline or EOF
- adds a 0 and creates a null terminated string
- returns buffer(string) or 0 on error
- increments the file-position indicator by the amount of read symbols

4.2.7 "un"read from file with C-API

```
int ungetc (int c, FILE *stream)
```

- puts c(parameter -> file-descriptor) back to the stream into a special *unget stack*
- the next fget call will prefer the symbols in the unget stack
- no change in the file itself
- unget stack has a minimum size of 1 -> works at least once, can work multiple times depending on implementation
- returns c(parameter -> file-descriptor) or EOF on error



4.2.8 Write to file with C-API

```
int fputc(int c, FILE *stream)
```

- converts file-descriptor c in unsgined char and writes it to stream
- returns either c or EOF
- increments the file-position indicator by 1

```
int fputs (char *s, FILE *stream)
```

- writes the symbols from the string s until the 0 termination symbol into the stream
- the 0 termination will not be written
- returns EOF on error

4.2.9 End of file and Error in File C-API

```
int feof (FILE *stream)
// returns 0 when end of file has NOT been reached

int ferror (FILE *stream)
// retuns 0 when NO error occurred

// Example usage:
int return_value = fgetc (stream);
if (return_value == EOF) {
  if (feof (stream) != 0) {
    /* EOF reached */
  } else if (ferror (stream) != 0) {
    /* error occurred, check errno */
  }
}
```

4.2.10 Manipulation of file-position indicator with C-API

```
long ftell (FILE *stream)
// returns the current file-indicator
// POSIX extension of ftello with return type off_t

int fseek (FILE *stream, long offset, int origin)
// set the file-position indicator like lseek
// POSIX extension of fseeko with off_t as type for the offset

int rewind (FILE *stream)
// reset the stream
// equivalent to fseek(stream, 0, SEEK_SET) and clear the error state
```

4.3 Ext2

4.4 Ext4

5 Processmodels

5.1 Base

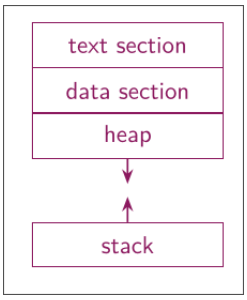
Whenever we run a program, there are at least 2 actors: the program itself and the OS.
The communication between OS and program runs on the C-API (or soon rust as well :))
Each program only knows itself and the OS.
Systems like this are called *monoprogramming*

5.1.1 Improvement with "async"

This is what javascript does, it is only pseudo parallel in the sense that there is *efficient process switching*, however we are still at 1 process at a time!
There are multiple names for a process in this case -> task, job, or simply process.
Here we still want to ensure that each program may think that is "the only running program",
this means that the operating system *must provide each service individually* -> memory, IO, etc.

5.2 Process

Each process has:
- Text section: image of the program in the memory (binary)
- data section: global variables of the program
- memory for heap
- memory for stack



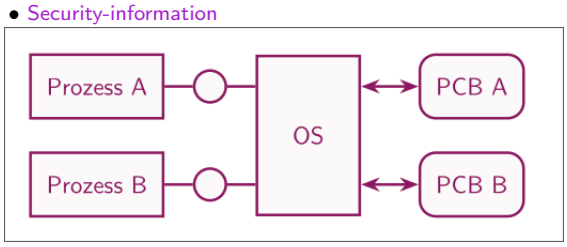
5.2.1 Program vs Process

Process
- active: actually executes instructions
- executes instructions in serial manner
- may perform actions for multiple programs -> according to POSIX

Program
- passive: only says what to do
- can be run in parallel -> multiple processes execute different tasks for the program

5.2.2 Process Control Block (PCB)

The operating system stores information about a process in this block.
- ID for process -> PID
- Memory for the state of the processor
- scheduling information
- data for synchronization and communication between processes
- filesystem relevant information -> current open files
- Security-information



5.2.3 Interrupts, Processes and Context Switch

For every interrupt the Operating system saves the current state of the process in the PCB.
The following will be saved:
- Registers
- Flags
- Instruction Pointer
- MMU-Configuration -> Page-Table-Pointer

After that, the interrupt handler will be called (which is run by the OS), which can switch context if needed -> e.g. switch to another process.
As soon as the handler is done, the new context PCB is recovered.

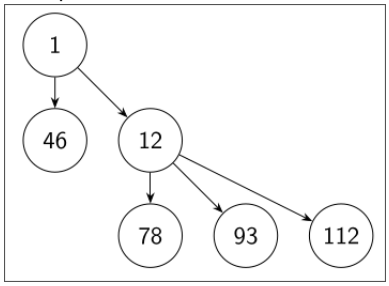
5.2.4 Creation of a process

POSIX:
1. create process
2. load a program into this process and put it into ready mode

Windoof:
both at once... (proprietary)

5.3 Process Hierarchy

In POSIX, each process has exactly 1 parent, other than the root process.
Each process can have unlimited amount of child processes!



5.3.1 fork function

The fork funcion crates an exact copy of the parent process.

```
pid_t fork(void)
// not to mistake with :(){ :|:& };:
// kekw
```

- the forked process has its own ID -> PID
- the forked process has the parent ID saved as well
- the return of this function will be handled in *both processes*
  in parent: return will be either the ID or -1 for error
  in child: returns 0

Example of usage:

```
int main() {
  pid_t new_pid = fork()
  // from here on both processes run this.

  if (new_pid > 0) {
```

```
    // this is the parent process
    // only the parent will run here
  } else if (new_pid == 0) {
    // this is the child process
    // only the child will run here
  } else {
    // error handling for parent process if forking failed
  }
}
```

5.3.2 exit function

```
void exit(int code)
```

This is a return method to get out of a process.
The code is simply the error handling code that you would like to pass to the parent process.

5.3.3 wait function

```
pid_t wait(int* status)
```

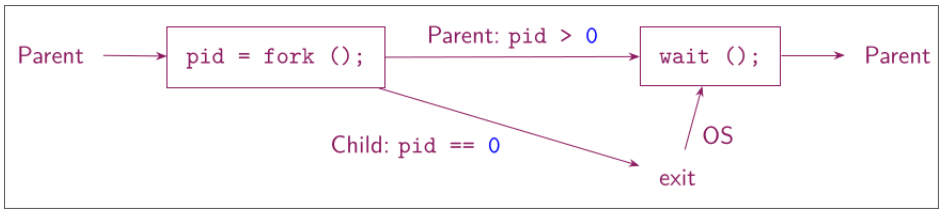
- blocks process until *one of his child processes ends*
- the status will cover the return code of the child process
- status usually handled with macros:
  WIFEXITED(*status) != 0 -> if child exited properly
  WEXITSTATUS(*status): exit code of child
- will return -1 if error
  ECHILD: no child anymore to wait for (if false you still have children)
  EINTR: was interrupted by signal

5.3.4 waitpid function

```
pid_t waitpid (pid_t pid, int *status, int options)
```

Like wait, but you can choose the child to wait for with pid.
- pid > 0: waits for child with this pid
- pid == -1: waits for any child -> like wait
- pid == 0 and pid < -1 enables waiting for processes of a specific process group

5.3.5 Examples



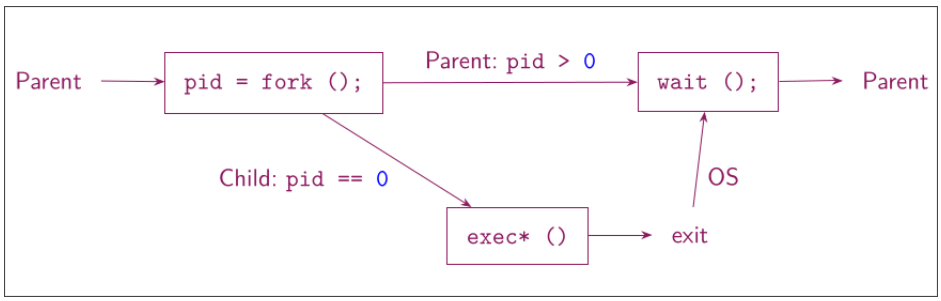
Worker example:

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

void spawn_worker(int a) {
  if (fork() == 0) {
    // we are in child
    printf("%d",a);
    // print number of process
    exit(0); // return code ok
  }
}

int main() {
  int pid = 0;
  for (int i = 0; i < 10; i++) {
    spawn_worker(i);
    // create 10 child processes
  }

  // do something in parent

  do {
    pid = wait(0);
    // status code is not saved!
  }

  while (pid > 0 || errno != ECHILD);
  // run until no more children!
}
```

5.3.6 exec functions

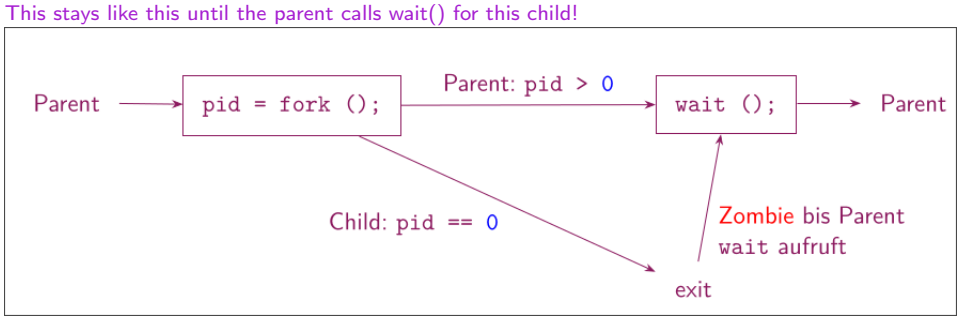
There are 6 versions of exec -> execl, execle, execlp, execv, execve, execvp
Each exec function *replaces the program in the current process with another*.
Parameters:
- execl*: binary and args as list -> execl(path0, arg0, arg1, ...)
- execv*: binary and args as vector -> execv(path, argv)
- exec*e: allow array for environment variables as parameter, in the other variants the env-vars stay the same
- exec*p: search for filename via environment variable PATH, the others use absolute or relative paths

the * means any character -> functions are a family



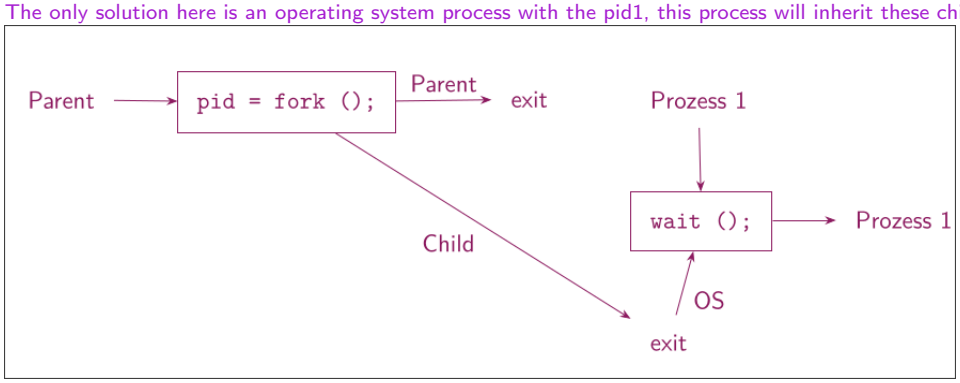
5.3.7 Zombie Processe

This refers to child processes that have ended, but aren't removed yet.
This means the PCB etc are all still there, but it doesn't do anything.
This stays like this until the parent calls wait() for this child!



5.3.8 Orphan Process

This is even worse than regular zombie processes. This means that the parent process is now dead, therefore no process can wait for this.
The only solution here is an operating system process with the pid1, this process will inherit these child processes and will then continuously end all of them.



Note, it might be that the parent process is stuck/has an error, in this case the child processes will be passed to 1 for ending them, while the parent is killed.

5.3.9 function sleep

```
unsigned int sleep(unsigned int seconds)
```

- waits "closesly" for the amount of seconds provided
- can be interrupted by the OS (example: signals)
- returns the amount of seconds that would remain to sleep
  example if interrupted after 5 seconds, returns entered time - 5 seconds

5.3.10 function atexit

Here you can pass functions that will be executed before the process exits.
This is usefull for resources that the OS doesn't know/care about.

```
int atexit(void (*function)(void))
```

- atexit can be called multiple times to register multiple functions
- same function can be registered multiple times
- functions will be called in reverse order to registered order!

5.3.11 function to read pid

These functions simply return the pid of the current process or the parent.

```
#include <stdio.h>
#include <unistd.h>
int main() {
  pid_t my_pid = getpid();
  pid_t my_parent_pid = getppid();
  // do something with pid
}
```

6 C Toolchain

The toolchain handles the *precompilation, compilation, assembling and linking* of executables.

6.1 Precprocessor

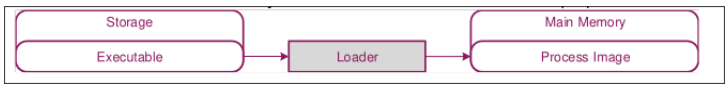
The preprocessor translates Macros and header files into actual code.
The output will then be passed to the translation unit and the compiler.

6.2 Linker

After the individual assembly files have been assembled, the linker takes all object files and creates one executable from them.

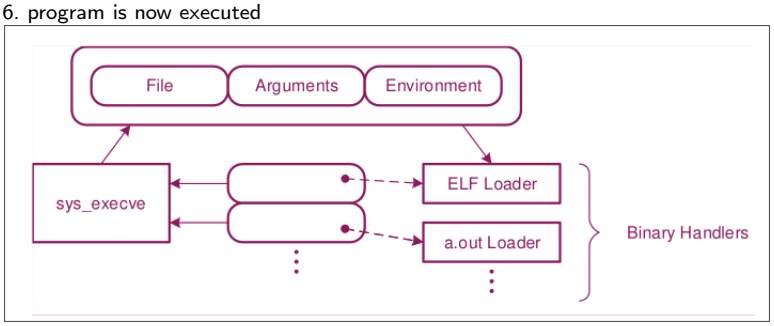
6.3 Loader

The loader takes executables and dynamic libraries and loads them into ram.
Note that static libraries are basically inside the executable.



6.3.1 Penguin Loader

The loader under linux has the following steps:
1. exec* have syscalls on sys_execve
2. searches for file, checks for permission and opens specified file
3. counts and copies arguments and environment variables
4. provides request for each registered "binary handler"
5. Binary handler try to load data and interpret data one by one
6. program is now executed

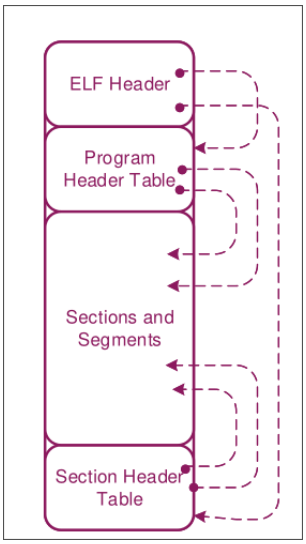


6.4 Executable Linking Format ELF

- Binary format, which specifies a program as binary
  windoof and penguin have some more than just this
- two possible formats
  Linking view
  Execution view
- Used for linker and loader
  Object-File: Linking-view -> something.o Programs: Execution-view -> no ending? Shared-Objects: dynamic libraries (Linking and Execution View!) -> wlroots.so
- item 4

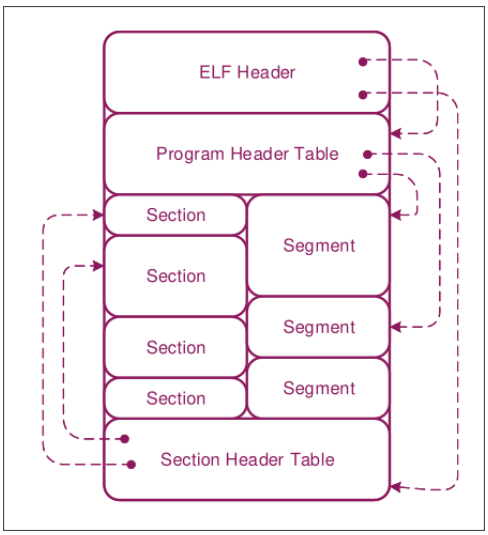
6.4.1 ELF Structure

- Header
- Program Header Table: (only necessary in execution view)
- Segments: (only necessary in execution view)
- Section: Header Table (only necessary in linking view)
- Sections: (only necessary in linking view)



6.4.2 Segments and Sections

- Segments and sections occupy the same memory space
  Compiler specifies space for sections
  Linker specifies space for segements
- Linker combines segments with the same names and defines segments
- Compiler defines sections



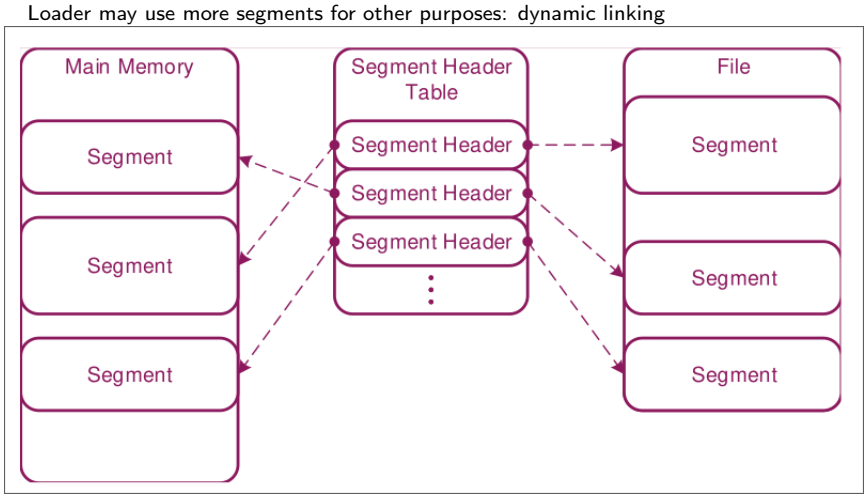
6.4.3 Header of an ELF file

- 52 byte: describes the structure of the file
- Type: Reallocatable, executable, shared object?
- 32 or 64 bit
- Encoding: little or big endian

- Machine: i386, arm, intel 64 etc
- Entrypoint: Address at which the program should start -> main()
- relative address, count and size of the entries for the program header table
- relative address, count and size of the entries for the section header table

6.4.4 Program Header Table and Segments

- Program Header Table (or Segment Header Table): Table with n entries
- Each entry is 32 bit and describes a segment
  - Segment-Type and Flags
  - Offset and size of file
  - Virtual address and size in memory -> possible addition: physical address
- Segments are used at runtime by the loader
  Loader loads specified segments to memory
  Loader may use more segments for other purposes: dynamic linking



6.4.5 Sections

- Section Header Table: Table with m entries (usually :m != n | not equal sections to segments)
- Each entry (40bytes) describes a section
  - Reference to string table
  - Type and Flags
  - Offset and Size of file
  - Specific information based on section type
- Sections will be used by linker
  - collects and concatenates(same name) sections of all object files
  - generates executable

Section Types

- SHT_PROGBITSData defined by program, linker does not interpret this
- SHT_SYMTABSymbol Table
- SHT_STRTABString Table
- SHT_REL/RELARelocation Information
- SHT_HASHHashtable for symbols
- SHT_DYNAMICInformation for dynamic Linking
- SHT_NOBITSSetion without data in file

Section Attributes

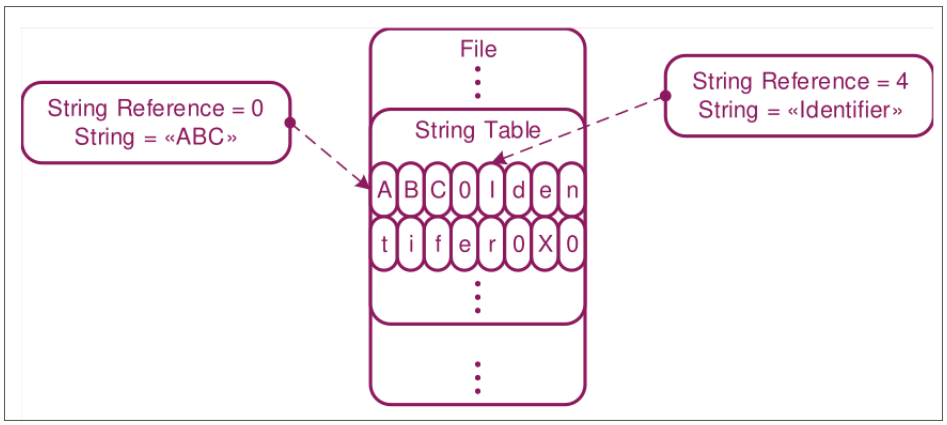
- SHF_WRITE data of this section should be writable during execution
- SHF_ALLOC data of this section should be in memory during execution
- SHF_EXECINSTR data of this section display machine code

Special Sections

- .bss: uninitialized data
- .data: initialized data
- .data1: initialized data
- .debug: debug information
- rodata: read only data
- .rodata1: read only data
- .text: executable instructions
- .symtab: Symbol-Table
- .strtab: String-Table

6.4.6 String Table

- Section of a file that has null terminated strings in a row
- Strign reference -> offset of string (starts at 0 and increases per char)
- typically has names of symbols
- typically does NOT include string literals like "henlo birb!", this is usually in .rodata
- item 4

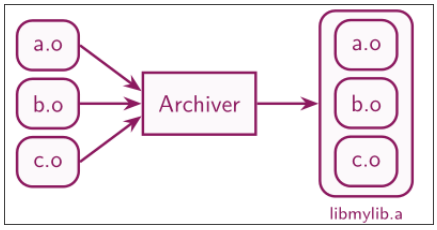


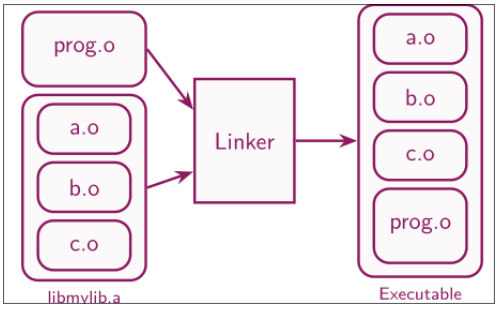
6.4.7 Symbols

- Symbol Table: each entry has 1 symbol
- Symbol: 16 bytes
  - Name: 4 bytes, reference in string table
  - Value: 4 bytes, depends on symbol type, could be an address for example
  - Size: 4 bytes, size of symbol (for example length of function)
  - Info: 4 bytes types, binding attributes (local, global, weak), reference to section header

6.5 Static Libraries

- Static libraries are archives of object files
- will be produced with the "ar" tool
- named lib<something>.a -> note the ending
- will only be referenced if included in compilation -> clang -lmylib
- static libraries are included in the binary and will receive a static address





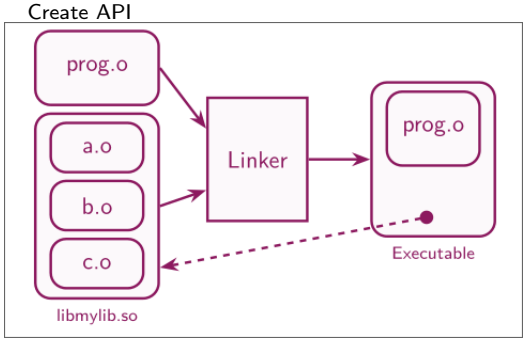
More information:

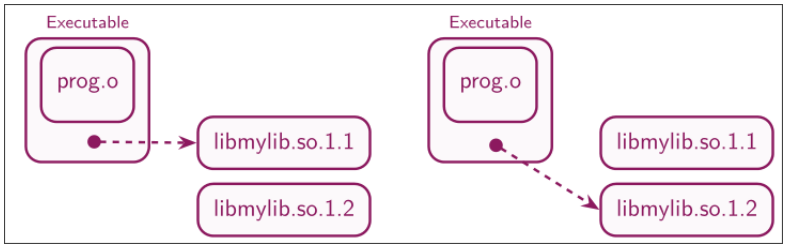
- In the early days, there were only static libraries
- Will never cause a program to suddenly stop working due to updates -> it is included in binary!
- easy to implement and use
- Must be recompiled with program if changes are made
- increases binary size
- functionality can't be increased with updates to library for binaries -> again must be recompiled
- plugins impossible or only very hard

6.6 Dynamic Libraries

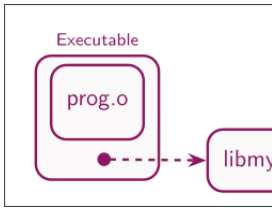
- loaded at runtime
- harder to implement
- executable receives only a reference to library
- library can be updated independent of binary
- if binary might stop working on library updates
- Plugins are relatively easy to create

  Create API





6.6.1 Delayed Loading



This is essentially lazy loading for libraries, only the libraries that will be used right then and there are loaded in order to save memory.

6.7 POSIX API: dynamic libraries

6.7.1 dlopen

opens a dynamic library and returns a handle to it

```
void* dlopen(char* filename, int mode)
```

Mode specifies how to handle the file:
- RTLD_NOW: all symbols will be bound to the library on load
- RTLD_LAZY: symbols will be bound if needed
- RTLD_GLOBAL: symbols can be bound used during binding of other object-files
- RTLD_LOCAL: symbols can't be used for other object-files

6.7.2 dlsym

Returns the address of a symbol from the specified library (handle for library)

```
void* dlsym(void* handle, char* name)
```

There is no type information, you only receive an address.
This means you can't know if it is a function, variable or whatever.
Example:

```
typedef int (*func_t)(int);                 // our function returns int and takes int
handle = dlopen ("libmylib.so", RTLD_NOW); // open dynamic library
func_t f = dlsym (handle, "my_function");  // take function from library and store in f
int *i = dlsym (handle, "my_int");         // take variable from library and store in i
(*f)(*i);                                   // call function
// Note we have no information about either f or i, we just assume they are function and variable respectively!
// You have to read API and documentation yourself to make sure that is the case!
```

6.7.3 dlclose

Closes the dynamic library via the specified library (handle to library)

```
int dlclose(void* handle)
```

Returns 0 on success

6.7.4 dlerror

Returns a null terminated string as error if an error occurred.

```
char* dlerror()
```

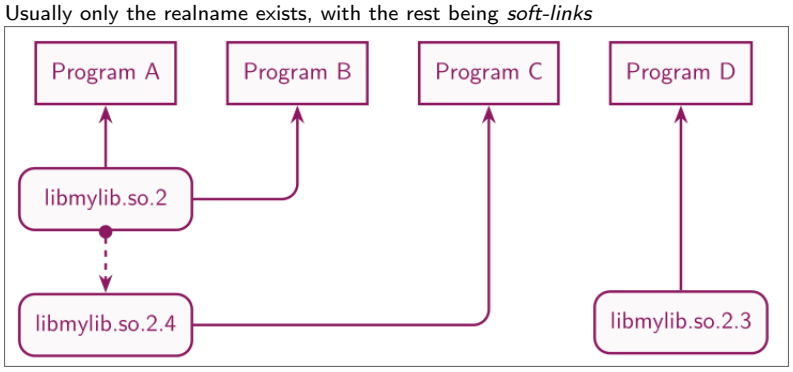
6.7.5 Automatic Loading of ELF files

As long as there is a reference to the library in the executable ELF file, libraries will automatically be loaded by need.

6.8 Naming of Shared Objects

- Linker-Name: libmylib.so
- SO-Name: libmylib.so.2
- Real-Name (filename): libmylib.so.2.1

The tool "ldconfig" properly creates these files for you.
Usually only the realname exists, with the rest being *soft-links*



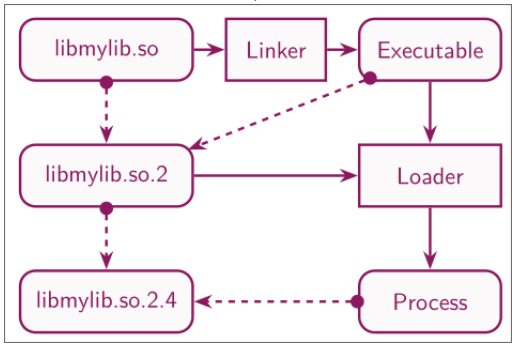
As one can see, you should always use the SO name (or linker (not recommended)) at the least, if you link against the real name, then your application will break on each update...

6.8.1 Updates

- Real name never changes
- SO-name: update on feature increases -> new API
- Real-Name: update on bugfixes

6.9 Shared Objects with Linker and Loader

Linker uses the base name, the executable the SO name and the process will finally use the real name.



6.10 Creating Static Libraries:

```
// compile
clang -c f1.c -o f1.o
clang -c f2.c -o f2.o

// create archive
ar r libmylib.a f1.o f2.o
```

6.11 Creating Dynamic Libraries:

```
// compile
clang -fPIC -c f1.c -o f1.o
clang -fPIC -c f2.c -o f2.o

// create image -> .so file
clang -shared -Wl,-soname,libmylib.so.2 -o libmylib.so.2.1 f1.o f2.o -lc
```

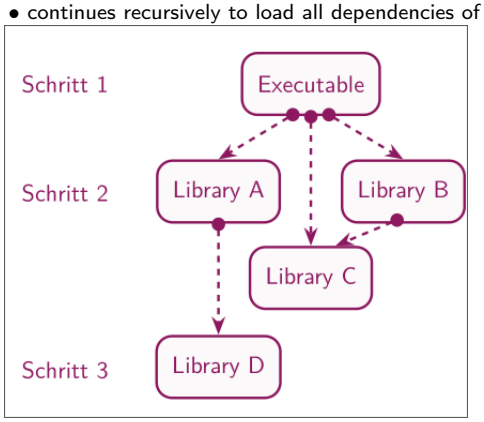
Important, the linker will prefer dynamic libraries, you need to override the default to use static versions if both exist

6.12 Using libraries

```
// static libraries
clang main.c -o main -L. -lmylib

// dynamic libraries
clang main.c -o main -lmylib

// dynamic library that will be called with dlopen
clang main.c -o main -ldl
```

6.12.1 ld-linux.so effective loader

- Can be used as an executable with –list flag
- will be called by the OS indirectly
- finds and loads all other shared objects one by one
- continues recursively to load all dependencies of loaded shared objects

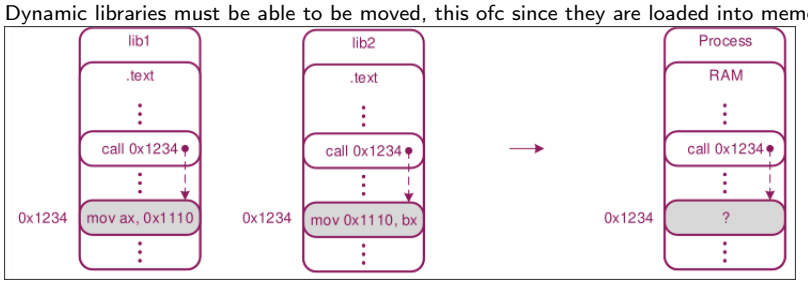


6.13 Shared Object Facts

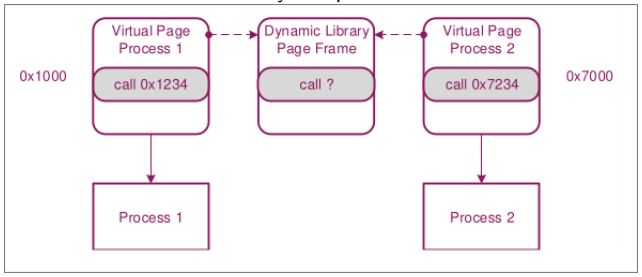
- Referenced Shared Objects are stored inside the executables
  - readelf -d shows the content of dynamic section
  - type of the entries is NEEDED
- ldd shows all shared objects, even indrectly used ones
  - executes executables and instructs loader to show trace
  - should only be used on trusted executables!!
- Pretty much all Executables (C) need at least two shared objects
  - libc.so -> standard C library
  - ld-linux.so -> ELF Shared Object Loader

6.14 Dynamic Library Implementation

Dynamic libraries must be able to be moved, this ofc since they are loaded into memory.



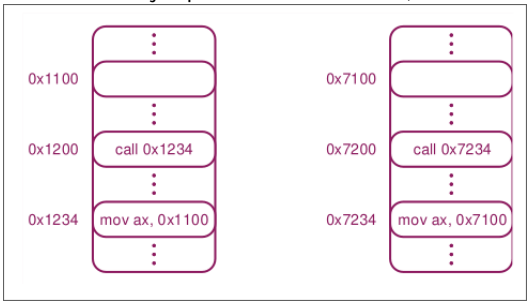
This however means that only one process at the time can load the library unless you load it twice, which is not something you want.



The solution to this is saving the address of the library which can be seen with 2 implementations below:

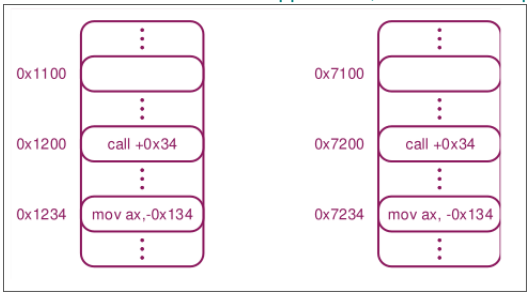
6.14.1 Position-Dependent Code

This means we jump to absolute addresses, in this case to absolute addresses of our dynamic library.



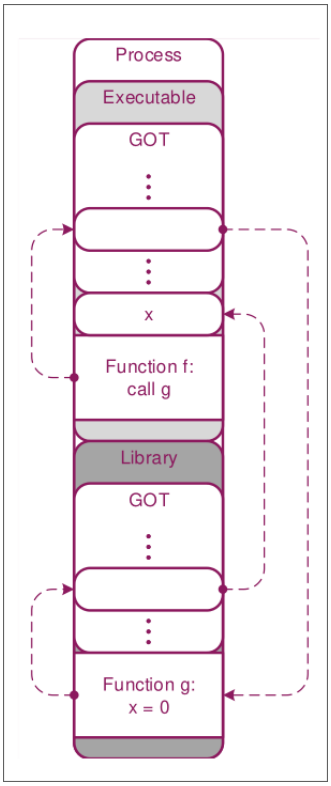
6.14.2 Position-Independent Code

This means we jump relatively to our current instruction pointer inside of our program.
Note that the CPU has to support this, 32bit for example does not.



6.14.3 Global Offset Table

In each program/library we have a global offset table, which shows the relative distances to libraries etc that you need/want.
This way you can use the same dynamic library that is loaded into memory, even though another process is also using it.



6.15 PLT Process Linkage Table

Implements lazy binding!

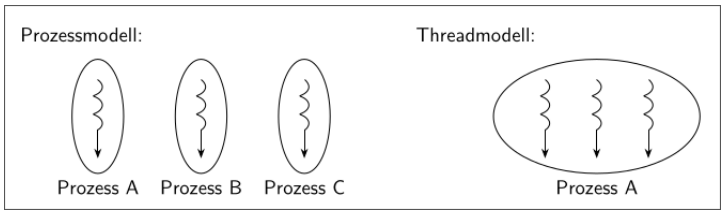
- one function per entry
- PLT entry has call instruction in GOT entry
- GOT points to proxy function
- Proxy function finds link to proper function and overwrites own entry
- Benefit: eliminates check for "is it loaded" after initial load (is expensive!!)

7 MultiThreading

7.1 Threads

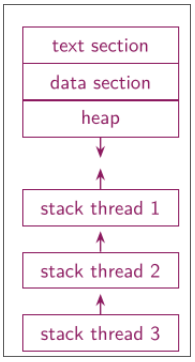
- Threads are parallel activities inside of a process
- Threads have equal access to all resources in the process
  - code(text secion)
  - global variables (data section)
  - heap
  - open files
  - MMU-data

7.2 Process-Model vs Thread-Model



7.3 Thread as stack and context

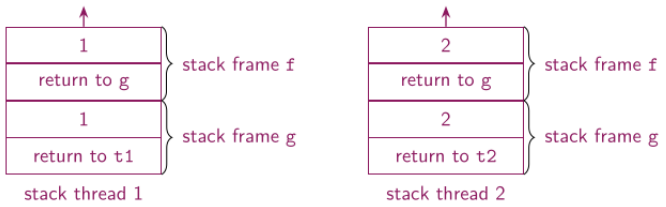
- Each thread needs its own context and its own stack, since it will have its own function call stack
- Information about call stack will usually be placed in a Thread-Control-Block
- In Linux, each process has a copy of the Process-Control-Block of its own process instead!!



Example of two threads that call the same function:

```
1 void f (int x) { /* ... */ }
2
3 void g (int x) { f (x); }
```

```
1 void t1 (void) { g (1); }
2
3 void t2 (void) { g (2); }
```

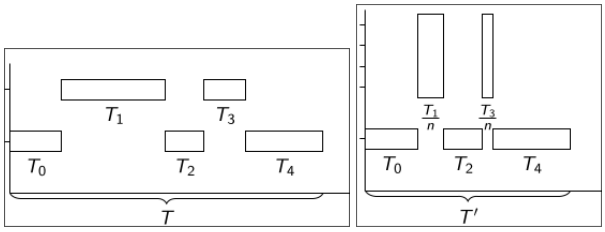


7.4 Parallization of algorithms

Algorithms are by default serial, however, there are some parts that you could run in parallel.
Example:

```
// this can be split
for (int i = 0; i < n; ++i) {
  sum += a [i];
}

// into this
for (int i = 0; i < n / 2; ++i) {
  sum0 += a [i];
}
for (int i = n / 2; i < n; ++i) {
  sum1 += a [i];
}
sum = sum0 + sum1;
```

7.4.1 Time difference



This shows the non parallizable parts on the bottom and the others on top.
As it can be seen the top parts are shrinking by large margins since they can be done in parallel!

7.4.2 Speedup Factor

$$f \leq \frac{T}{T'} = \frac{T}{T_s + \frac{T - T_s}{n}}$$

$$s = \frac{T_s}{T}$$

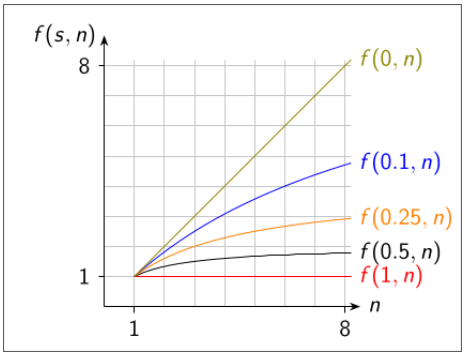
$$f \leq \frac{T}{T_s + \frac{T - T_s}{n}} = \frac{T}{s \cdot T + \frac{T - s \cdot T}{n}} = \frac{T}{s \cdot T + \frac{1 - s}{n} \cdot T}$$

$$\boxed{f \leq \frac{1}{s + \frac{1-s}{n}}}$$

Legend:
- f: speedup factor
- T: Total time
- T': Time saved
- T_s: total time that must be done in serial fashion!
- n: Count of processors
- s: relation of serial time to total time

  Note this is often just a guess, not a calculation.
  This is because we often can't say how long something would really take!

7.5 Amdahls Rule



f(s,n)!!

- best case: s==0 -> everything is parallelizeable
- worst case: s==1 -> nothing is parallelizeable -> user input

If you keep increasing the processor count, then you will eventually go towards the limit of 1:

$$\lim_{n\to\infty} \frac{1-s}{n} = 0$$

$$\lim_{n\to\infty} s + \frac{1-s}{n} = s$$

$$\lim_{n\to\infty} \frac{1}{s + \frac{1-s}{n}} = \frac{1}{s}$$

8 POSIX Threading API

8.1 Creating a thread

```
int pthread_create (
  pthread_t *thread_id,
  pthread_attr_t const *attributes,
  void * (*start_function) (void *),
  void *argument
)
```

- Creates a thread and returns 0 on success, otherwise errorcode
- The ID of the new thread will be stored in thread_id
- attributes is an opaque object, with which you can specify things like the stack-size
- start_function is the first instruction the new thread will execute
- the thread will place the parameter argument inside of that function
- the programmer must ensure the start_function and the argument are compatible
- You must specify who is responsible for garbage collection of the data structure
- Don't delete the stack of the thread during its execution!

8.1.1 Example for pthread

```
struct T {
  int value;
};

void *my_start(void * arg) {
  struct T* p = arg;
  printf ("%d\n", p->value);
  free (arg);
  return 0;
}

void start_my_thread (void) {
  struct T* t = malloc (sizeof(struct T));
  t->value = 109;
  pthread_t tid;
  pthread_create (
    &tid,
    0, // default attributes
    &my_start,
    t
  );
}
```

The default attributes are handled by the operating system!
They are used when you pass 0 -> nullptr to this function

8.1.2 Attributes for pthread

Attributes are handled the following way:

```
pthread_attr_t attr;                        // variable for attribute
pthread_attr_init(&attr);                   // initialize attribute
pthread_attr_setstacksize(&attr, 1 << 16); // set the stacksize of attribute
pthread_create(..., &attr, ...);            // create attribute
pthread_attr_destroy(&attr);  );            // delete attribute
```

The number 0 means that we pass default attributes

8.2 pthread_exit

```
void pthread_exit(void *return_value)
```

- ends the thread and returns the return_value
- this is the equivalent to the return of the start_function with a return value

8.2.1 Lifetime of a thread

A thread lives until:
- the thread executes a return statement
- the thread calls pthread_exit
- another thread calls thread_cancel
- the thread is killed otherwise

8.3 pthread_cancel

```
int pthread_cancel(pthread_t thread_id)
```

- Sends a demand to the OS to kill a thread with the thread_id
- The function doesn't wait for the thread to be killed
- The return code is 0 when the thread exists, otherwise ESRCH errorcode

8.4 pthread_detach

```
int pthread_detach(pthread_t thread_id)
```

- Removes the memory which a thread has allocated, as soon as the thread is terminated
- Doesn't end the thread, in case the thread has not ended
- the return value is 0 if the thread exists, otherwise ESRCH

8.5 pthread_join

```
int pthread_join(pthread_t thread_id, void **return_value)
```

- waits until the thread_id is dead
- Takes the return value of the thread and stores it in the return_value parameter
- return_value can be NULL -> 0, in this case it will not store the return value
- calls pthread_detach
- the return value is 0 if the thread exists, otherwise ESRCH

8.6 pthread_self

```
pthread_t pthread_self(void)
```

Returns the ID of the current running thread

9 Thread-Local Storage (TLS)

The problem here is that there are certain things that are stored globally, take errno for example.
If this would be thread overreaching, then another thread might overwrite errno, therefore interfering with another thread that is running somewhere else.
This leads to race conditions where multiple threads want to access -> read/write this variable.
TLS simply stores these variables per thread, resolving this issue.
This is complicated to implement for the OS, but for many programming languages this is simply handled with an attribute.
The idea is that you create a TLS-Variable which will be saved in a global variable as the key.
This key can then be used in the other threads with special variables. see below:

9.1 pthread_key_create

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*))
```

- creates a new key in the parameter key
- pthread_key_t is an opaque datastructure
- The OS stores each key for each thread with a value of type void*
- the value of this void* is initialized with NULL -> 0
- The OS calls the destructor for this value if it is not NULL
- returns 0 if ok, error code otherwise

9.2 pthread_key_delete

```
int pthread_key_delete(pthread_key_t key)
```

- removes a key and the associated values on all threads
- the key can't be used after calling this function
- the program must remove all memory that was allocated
- returns 0 if ok, error code otherwise

9.3 pthread_setspecific and pthread_getspecific

```
int pthread_setspecific(pthread_key_t key, const void * value)
void* pthread_getspecific(pthread_key_t key)
```

- writes or reads the value that is associated with the key for this thread
- typically used as a pointer for memory space -> []

9.4 Example with TLS

```
// setup
typedef struct {
  int code;
  char *message;
} error_t;

pthread_key_t error;
void set_up_error (void) {
  pthread_setspecific(error, malloc (sizeof(error_t)));
}

// read and write in the thread
void print_error (void) {
  error_t * e = pthread_getspecific(error);
  printf("Error %d: %s\n", e->code, e->message);
}

int force_error (void) {
  error_t * e = pthread_getspecific(error);
```

```
  e->code = 98;
  e->message = "file not found";
  return -1;
}

// main and thread
void *thread_function (void *) {
  set_up_error();
  if (force_error() == -1) { print_error(); }
}

int main (int argc, char **argv) {
  pthread_key_create(&error, NULL);
  pthread_t tid;
  pthread_create(&tid, NULL, &thread_function, NULL);
  pthread_join(tid, NULL);
}
```

9.5 Important note for Threads

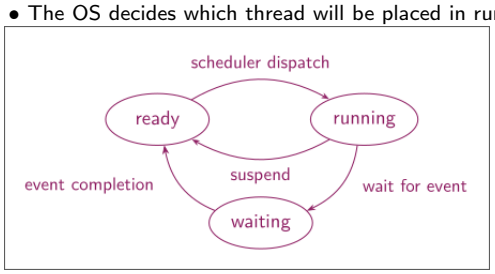
If one thread crashes or otherwise does something that it shouldn't, it could potentially break the entire program, this is the sacrifice that you are going to accept if you use threads over processes!

10 Scheduling

Note, in the lecture the word processor is equivalent to the word core. Kinda correct, but still...

10.1 States of a Thread

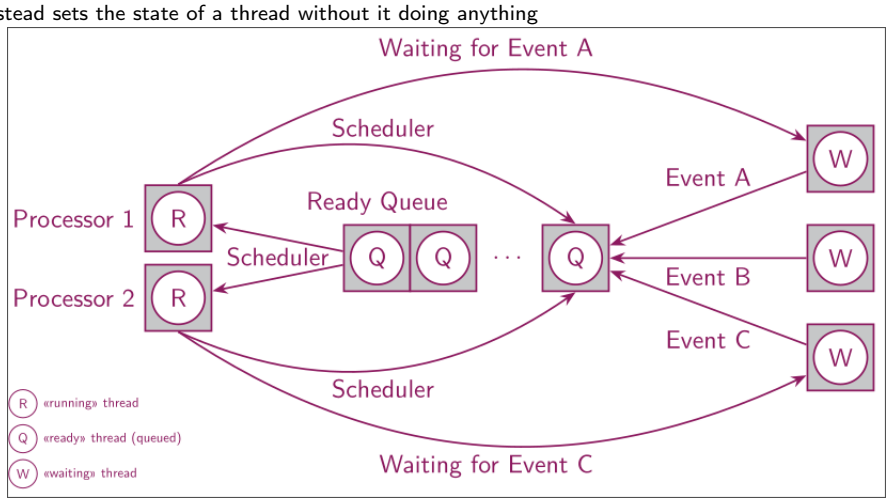
- There is at max only one running thread per processor core, this thread is in state *running*
- All other threads are in state *ready*
- A thread that is waiting with phtread_join is in state *waiting*
- Changes of states in a thread are done by the OS
- The OS decides which thread will be placed in running mode on each core



10.2 Waiting threads and Ready-Queue

Waiting Threads

- Threads that are waiting are not running at all -> they are NOT waiting in a loop!
- The OS instead sets the state of a thread without it doing anything



Ready Queue

- All threads that are ready are in ready queue
- Ready queue is not always a queue
  Scheduling might be done in a different way.
- new threads are usually placed in ready queue directly
  depending on the OS they might actually be placed in waiting...
- Threads *can't directly jump into running from waiting*

10.2.1 Powerdown-Mode

- Should no thread be ready to run, then the OS will put the core into standby
- The sleeping core will then be awoken by the OS on interrupt and continues operation
- Busy-wait, aka waiting in a loop hinders standby! This is why we should not do it!

10.2.2 Types of Threads

- I/O-heavy thread
  communicates often with I/O devices and doesn't compute much -> IO bound
- processor-heavy
  doesn't communicate much with IO and computes pretty much exclusively
- While the difference might be vague, good systems make a difference for these to optimize performance

10.3 Types of Concurrency

- Cooperative
  Each thread decides itself when it leaves the core -> stops running
- Preemptive
  The Scheduler decides when the thread will be removed from the core

10.3.1 Preemptive Multithreading

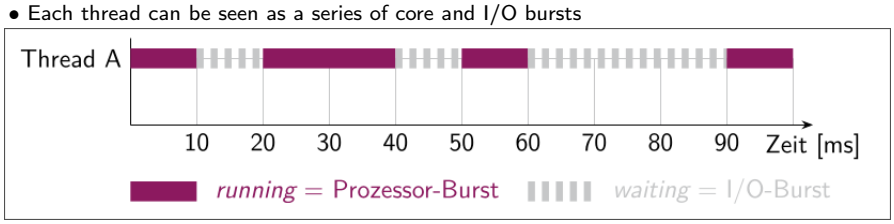
The current thread runs until:
- waits for IO operations
- waits for a resource or another thread
- system timer interrupt occurs
- leaves core on its own
- another thread with higher priority by scheduler gets into ready state
- a new thread with higher priority by scheduler gets created

10.4 Parallel, "quai"parallel and Concurrent Execution

- parallel
  all threads run parallel in realtime -> n threads require n cores
- "quasi"parallel
  N threads are running on less than n cores -> this means threads will have to be scheduled
  you only think they are runnin in parallel, they are actually not
- Concurrent This is an overall term that can refer to either of the above!

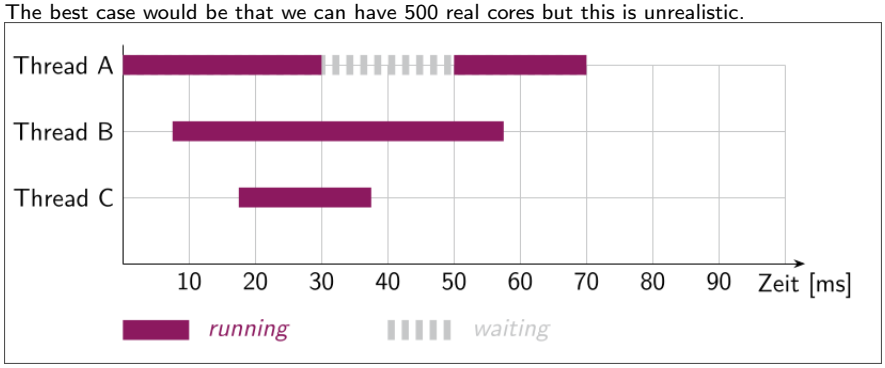
10.5 Bursts

- Core-burst
  Interval in which a thread will occupy a core fully in a parallel system
- I/O Burst
  Interval in which a thread will not need the core until next event will wake the thread up again -> input event
- Each thread can be seen as a series of core and I/O bursts



10.6 Parallel Execution as an ideal

This means that we will never have to switch threads, this was indeed the case when there were no threads to be had.
Aka inteli5 with 4 cores and that's it. This however means that there will always be wasted potential as some cores will be 100% occupied with crap.
The best case would be that we can have 500 real cores but this is unrealistic.



10.7 Requirements for Scheduler

Note that the requirements for embedded systems and consumer devices are very different.
In embedded systems you might not want a scheduler, as you know which processes will be running on there, on consumer devices you NEED scheduling as ANY process can be run on there.
Requirements from a users perspective:
- Waiting Time
  time spent by a thread in ready queue
- Response Time
  time spent until response is ready after request
- Turnaround Time
  time spent from start to end of a thread

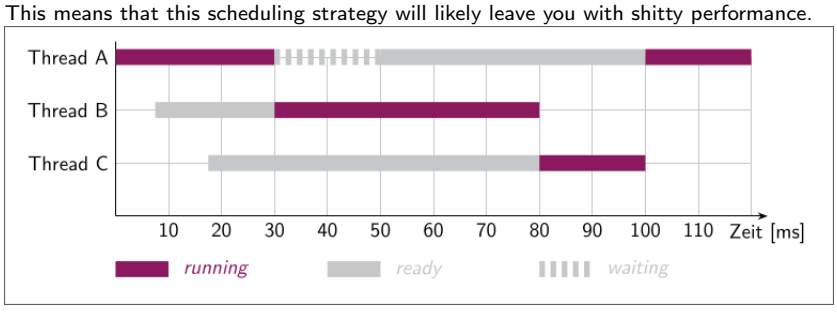
Requirements from a system perspective:
- Throughput
  Amount of threads that can be run per interval
- Processor Utilization
  Core usage percentage

10.7.1 Conflict of interest

The response and turnaround time will be increased by scheduling at all, but not scheduling will decrease throughput and core usage.
E.g. In order to gain concurrency, you need to sacrificy some responsiveness -> last thread might need to be removed before new thread can work.
Depending on the OS, or the setting of it, one or the other can be prioritized.

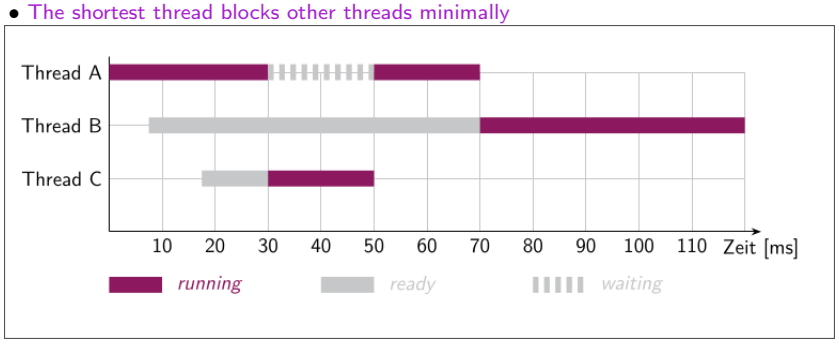
10.7.2 First Come First Serve(FCFS) Scheduling

This is a non-preemptive scheduling policy that does not care about thread priority or thread-task-size (how long will the task take?)
This means that this scheduling strategy will likely leave you with shitty performance.



10.7.3 Shortest Job First(SJF) Scheduling Strategy

- Chooses smallest task thread first
- If task size is ambiguous(more than one thread with smallest size) chooses via FCFS
- Can be used cooperatively or preemptively
- The shortest thread blocks other threads minimally

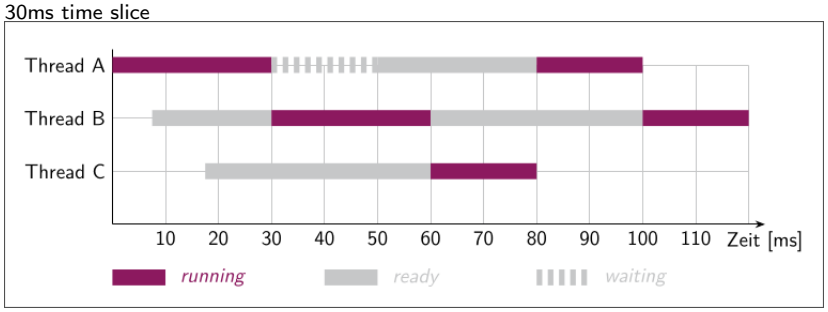


The problem with this is knowing the processor-burst -> task time
There is no real way of knowing it and can therefore only be guesses.

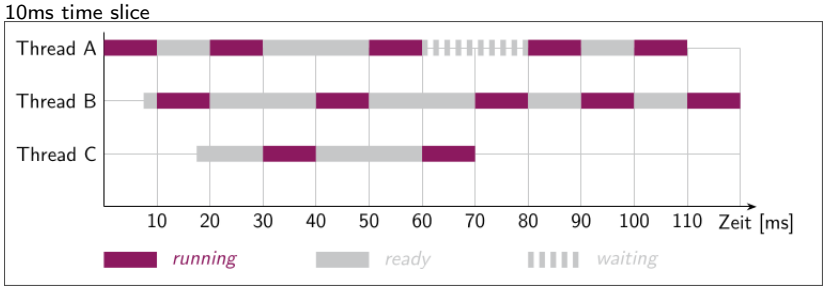
10.7.4 Round-Robin Scheduling

- Scheduler defines time slice of 10 to 100ms
- The default is FCFS, but a thread will only run as long as max time se by scheduler
- time slice size will affect performance heavily

30ms time slice



10ms time slice



10.7.5 Priority-Basd Scheduling

- Each thread will receive a priority
- Threads with higher priority will be chosen over others
- Same priority will be deal with via FCFS
- SJF is a special case of this scheduling -> shorter task == higher prio
- Priorities range based on OS -> 0 to 7 or 0 to 4096 (nice level)
- not to mistake with nice! -> Linux nice is per *PROCESS*: -20 highest prio, 0 default, +19 lowest
- On some OS 0 is the highest priority

10.7.6 Starvation

This can happen with some scheduling systems like priority based scheduling. Threads with lower priorities might not be able to run at all due to higher priority threads. This can be improved with *aging*, aka making sure threads at least get some sort of time no matter the prio.

10.7.7 Multi-Level Scheduling

The OS might specify different levels for scheduling like layers: background/foreground.
This means that there will be a mix of scheduling policies based on the layer the process is in, or on the priority the process is in.
This helps make sure that each part of the system gets a proper amount of time to run.
Queues can also be split -> high prio queue, low prio queue etc.

10.7.8 Multi-Level Scheduling with Feedback

- Per priority one queue
- Threads of higher priority queues will be preferred
- exhausts a thread its time slice, then it will be placed in a lower priority -> priority - 1
- Time slices increase in lower priorities
- The effect: threads with lower bursts -> task lengths are preferred

11 Scheduling in POSIX

11.1 Nice Value

- Each process has a nice value
- Value ranges from -20 to +19
- Lower values == higher priority
- Default value is 0
- Only root can go below 0 with processes by default

11.1.1 Nice Utility

On Linux there is the "nice" tool that prints the current nice level or sets a new nice level:
nice -n -10 command-to-run -> sets the priority 10 lower and runs command
nice -> prints the nice level of process

11.1.2 Nice function

```
int nice(int i)
```

- adds i to the nice value of the process
- returns the new nice value or -1 on error (check below....)

The -1 can be an error or a nice value! Here the code to show how to get the difference:

```
errno = 0;
if (nice(i) == -1 && errno != 0) {
// Error
}
else {
// -1 is nice value
}
```

11.1.3 Get Priority

```
int getpriority(int wich, id_t who)
int setpriority(int wich, id_t who, int prio)
```

- can change priority of processes, groups or users
- which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
- who: ID of process, group, or user
- getpriority() returns the nice value of the process
- setpriority() sets the value of nice of the process
  returns 0 on sucess, -1 on error

11.2 Setting Priority of Threads

```
int pthread_getschedparam (pthread_t thread, int * policy,
struct sched_param * param);

int pthread_setschedparam (pthread_t thread, int policy,
const struct sched_param * param);

int pthread_attr_getschedparam (const pthread_attr_t * attr,
struct sched_param * param);

int pthread_attr_setschedparam (pthread_attr_t * attr,
const struct sched_param * param);

// note the sched_param is a struct that is not really defined
// the only thing the posix standard says is that is needs a sched_priority field.

//example usage
pthread_attr_t a;
pthread_attr_init (&a);

struct sched_param p;
pthread_attr_getschedparam (&a, &p);

// here we set the new priority of the schedule priority
// set p.sched_priority

pthread_attr_setschedparam (&a, &p);

pthread_create (&id, &a, thread_function, argument);
pthread_attr_destroy (&attributes);
```

12 Mutexes and Semaphores

12.1 Producer and Consumer Problem

This is a simple implementation to the consumr and producer problem.

```
#define BUFFER_SIZE 16
struct item { ... };
struct item buffer [BUFFER_SIZE];

int writer = 0;
int reader = 0;

// producer thread
while (1) {
  produce_item (&buffer[writer], ...);
  int next_w = (writer + 1) % BUFFER_SIZE;
  // wait if reader is still reading next index
  while (next_w == reader) {}
  writer = next_w;
}

// consumer thread
while (1) {
  // wait if writer is writing to next index
  while (reader == writer) {}
  consume (&buffer[reader]);
  reader = (reader + 1) % BUFFER_SIZE;
}
```

Problems with this:

- Blocking wait: CPU is still used while in while loop -> useless
- Does not work is multiple producers/consumers are here -> write/reader index is not atomic!
- Count of elements available is unknown

If we now want to check for the amount of elements, then we can't do this immediately with –counter or ++counter
this instruction is not atomic!

12.1.1 Non-Atomic instructions

```
// an instruction like
--counter;

// will turn into this assembly code
// mov rax, [counter]
// inc rax
// mov [counter], rax
```

After any of these assembly instructions, the CPU can be interrupted to swap to another thread etc.
This means that the instruction in C is not done, leading to a potential data race!

t_0 :	Producer	`mov rax, [counter]`	rax = 5
t_1 :		`inc rax`	rax = 6
t_2 :	Consumer	`mov rax, [counter]`	rax = 5
t_3 :		`dec rax`	rax = 4
t_4 :	Producer	`mov [counter], rax`	counter = 6
t_5 :	Consumer	`mov [counter], rax`	counter = 4

This is a typical update lost problem, overwrite of an update.

12.1.2 Critical Section

This refers to the code segment that causes the issues -> here it is the counter increment and decrement, as well as any other non-atomic instruction.

12.2 Synchronization

- Mutual Exclusion:
  during a critical section, one wants other threads to not interfere with this thread -> only one thread accesses this segment
- Progress: If another thread wants to enter a critial section that is already being used, then there must be a decision on which thread enters the section next(in finite time!)
- finite wait: There must be a number N that defines how many times a thread can be overtaken, before getting a guarantee to enter the critical section

12.3 Atomic Instructions

An atomic instruction is an instruction that will be done with 1 processor cycle -> can therefore not be interrupted.

```
mov [2000], rax ; atomic, naturally aligned, no shifting necessary
; in other words, this address can be divided by 8
mov [2009], rax ; not atomic -> 2 writes necessary as not divideable by 8
; the reason: processors can write one entire byte.
; here it would write to 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016
; this means it has to write parts of 2 addresses!! 2016 is a new byte
; 2008-2015 byte 1, 2016-2023 byte 2
```

- Modern CPUs give no guarantee that *ANY* instruction definitely is atomic
- Modern CPUs will reshuffle instructions as it pleases -> this is no problem for singular threads, but can fuckup multithreaded programs
- In other words, multithreading is only possible with cooperation with hardware manufacturers

12.3.1 Naive Solution: Disable interrupts

- during critical section no context switching is allowed
- Ok with a single core? But not with multi core -> not thread safe
- Dangerous: What happens when this thread breaks? Aka infinite loop? Or some other error? The thread can't be killed!!!

12.3.2 Usage of Special Instructions

- Hardware guarantee that these instructions can't be done at the same time(even with multicore/multiCPU systems)
- Locks can be done based on this system
- Functions: Test-And-Set / Compare-And-Swap
- In other words, YOU CANT IMPLEMENT IT IN SOFTWARE, YOU MUST USE THESE HARDWARE FUNCTIONS

Test-And-Set:
This reads the value of from an address and sets the value to 1

```
// NOT REAL CODE
// only example of how it works
int test_and_set (int *target) {
  int value = *target;
  *target = 1;
  return value;
}

// could then be used like this:
while (tas (&lock) == 1) {}
// do something
lock = 0;
```

Compare-And-Swap:
Reads the value of the address, checks if the value is according to the expected value, and overwrite the value if it did.

```
// NOT REAL CODE
// only example of how it works
int compare_and_swap (int *a, int expected, int new_a) {
  int value = *a;
  if (value == expected) {
    *a = new_a;
  }
  return value;
}

// usage
while (cas (&lock, 0, 1) == 1) {}
// do something
lock = 0;
```

12.4 Semaphore

The semaphore is the actual implementation of this lock!
A semaphore has a counter with a number that is *always positive!*
You can interact with them via special functions.
Special Functions:

- Post: increments the value of the sempahore by 1
- Wait: if counter is 0, thread will go to waiting until another thread increments the counter again
  if counter is higher than 0, decrement and continue execution
- These functions are *atomic!*]

12.4.1 Producer and Consumer with Semaphores

```
// producer
while (1) {
  // wait if consumer is in next index
  wait (free);
  produce_item (&buffer[w], ...);
  post (used);
  writer = (writer + 1) % BUFFER_SIZE;
}

// consumer
while (1) {
  // wait if writer is in current reader index
  wait (used);
  consume (&buffer[r]);
  post (free);
  reader = (reader + 1) % BUFFER_SIZE;
}
```

12.5 Usage of Sempahores: sem_init

```
// definition
int sem_init (sem_t *sem,
              int pshared,
              unsigned int value);
```

Initializes a sempahore.
pshared == 0 > semaphore can only be used in this thread
pshared == 1 > semaphore can be used in other threads
Usage:

```
// note tis is typically a global variable!!
sem_t sem;
int main (int argc, char **argv) {
  sem_init (&sem, 0, 4);
  return 0;
}
```

Altnernative use:

```
// here  we create a struct that will be passed to the thread as parameter
struct T { sem_t *sem; ... };
int main (int argc, char **argv) {
  // instead of global variable, we now have a parameter
  sem_t sem;
  sem_init (&sem, 0, 4);
  // struct to pass to thread
  struct T t = { &sem , ... };
  return 0;
}
```

12.5.1 sem_wait and sem_post

```
int sem_wait (sem_t *sem);
int sem_post (sem_t *sem);
```

- Implement post and wait
- return 0 on success, else -1 and error code in errno
- in case of error, the semaphore will not be changed/mutated

12.5.2 sem_trywait and sem_timedwait

```
int sem_trywait (sem_t *sem);
int sem_timedwait (sem_t *sem,
                   const struct timespec *abs_timeout);
```

- sem_trywait: same as sem_wait but cancels instead of blocking if decrement not possible -> value of semaphore == 0
- sem_timedwait: waits until either timeout reached, or wait successful

12.5.3 sem_destroy

```
int sem_destroy(sem_t *sem);
```

Removes all memory that has been allocated for the semaphore.

12.6 Mutexes

A mutex has 2 states: acquired and released:

- Acquire:
  if count == 0, set count to 1 and continue
  if count == 1, block thread until count == 0
- set count to 0
- These are just locks

12.6.1 Usage

Very easy to use:

```
acquire (mutex);
++counter;
release (mutex);
```

Note, mutexes are also fences, this means that noone, neither the CPU or the OS may shuffle the instruction order in a mutex.

12.6.2 Mutexes in POSIX

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

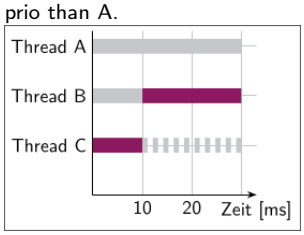
attributes are optional
- protocol: for example PTHREAD_PRIO_INHERIT: Mutex uses priority inheritance
- pshared: mutex can be used in other threads if set to anything other than 0
- type: Allows recursive usage or not -> with error code or without
- prioceiling: Returns minimum priority for threads that can stop the mutex

Usage:

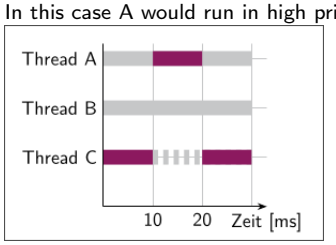
```
// acquire (blocking)
int pthread_mutex_lock (pthread_mutex_t *mutex);

// attempt to acquire (non-blocking)
int pthread_mutex_trylock (pthread_mutex_t *mutex);

// release
int pthread_mutex_unlock (pthread_mutex_t *mutex);

// clean up
int pthread_mutex_destroy (pthread_mutex_t *mutex);

// usage in main
pthread_mutex_t mutex; // globale Variable
int main () {
  pthread_mutex_init (&mutex, 0);
  // run threads and wait for them to finish
  pthread_mutex_destroy (&mutex);
}

// usage in thread
void * thread_function (void * args) {
  while (running) {
    // Enter critical section
    pthread_mutex_lock (&mutex);
    // Perform crit. section, e.g. ++counter
    // Leave critical section
    pthread_mutex_unlock (&mutex);
  }
}
```

12.6.3 Priority Inversion

Problem: A thread with high priority needs a resource within a mutex, the current holder of said mutex is a thread of low priority.
There is also a medium priority thread that is waiting to be scheduled.
High prio thread now needs to give up it's place in order to wait for resource, but instead of getting the resource from A, B is run instead since it has higher prio than A.



Solution: Priority Inheritance
This means that should thread C have to wait for a resource, then the thread holding this mutex will run with the same priority as the thread asking for the resource.
In this case A would run in high prio, meaning that B will not be run:



Legend:
- purple: running
- gray-dashed: waiting for resource
- gray: regular waiting

13 Communication and Synchronization

14 Programs and libraries

15 Graphical Overlays

C++ Advanced

Fabio Lenherr

summary

Table of Contents