

System Level Programming

Daniel Gruss

2023-10-08

Course Organization

Motivation

Last year, you took introductory C/C++ courses

- Einführung in die Strukturierte Programmierung
- Softwareentwicklung Praktikum

Time to apply your knowledge...

- Interaction with the operating system (Posix API)
- Processes, Threads
- Memory management

Learning Goals

Learn how C and C++ does things

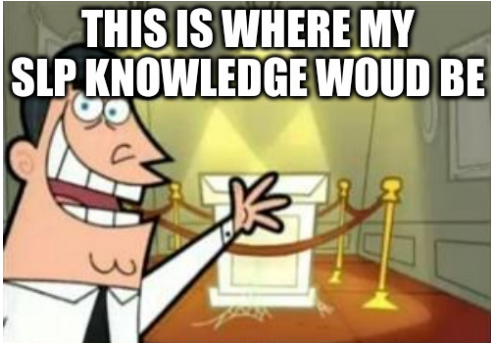
- Learn how the operating system manages your programs
- Learn to read and understand code
- Practice writing, fixing and adapting code snippets
- Practice or learn debugging!

Side effect:

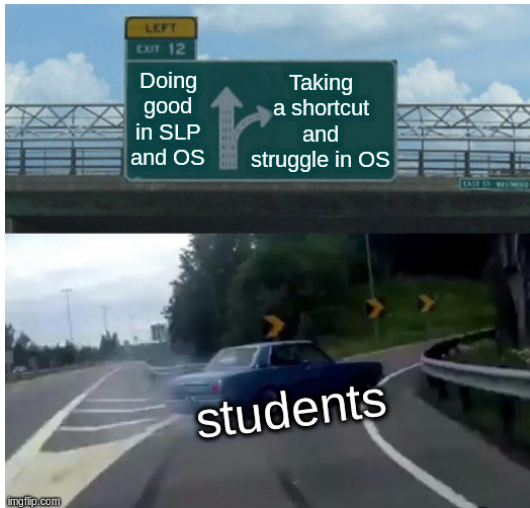
- Preparation for the operating systems course

Skillset Goals

- SLP is your booster for OS
- Everything, **Everything**, we teach, will matter for OS.
- Seemingly useless staring at addresses? You will need this.
- Seemingly useless knowledge about program behavior? You will need this.
- Seemingly useless hours of debugging? You will need it this much.
- Statistics show:
 - without SLP (or barely passed) → average grade 5 in OS
 - with a good grade in SLP → average grade 1-2 in OS



Take this course seriously



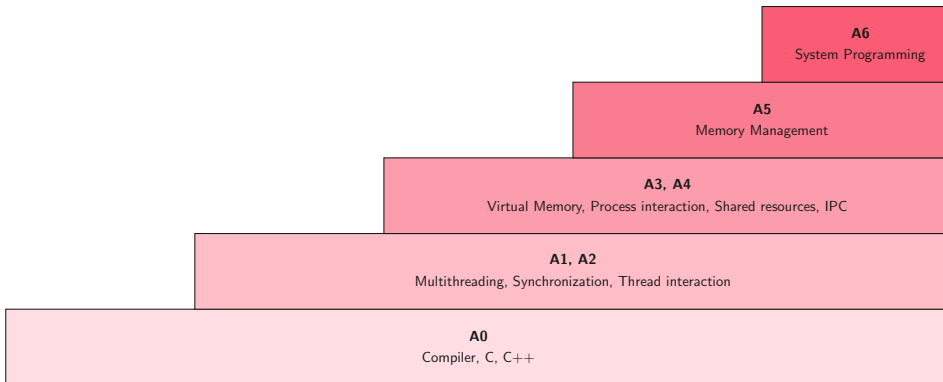
Registration and Related Issues

- Registration **closed**
- You obtain a grade if you are enrolled
 - as soon you submit a single assignment.
 - **A0 does not count** → self-assessment

You will receive an email containing information

- on your GIT repository, and
- on your account in the test-system
- You will work individually on all assignments.
- Mandatory exam

Course Outline - Assignments

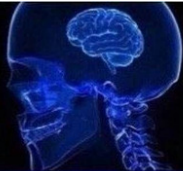


Course Outline - Lectures

Three types of lectures

- Regular lectures
 - Theory
 - Examples
- Assignment presentations
 - Kick offs
 - Organisational details
 - Some basic theory
- Weekly question hours (0.5hr)
 - Discord!
 - for current + next assignment
 - Multiple tutors present

**DO IT
BY YOUR OWN**



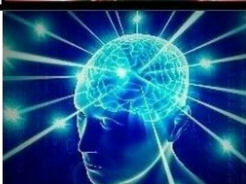
**LOOK AT
ANOTHER'S
CODE BASE**



**GET THE CODE
BASE BY EMAIL**



**COPY THE
CODE BASE**





Student: copied code

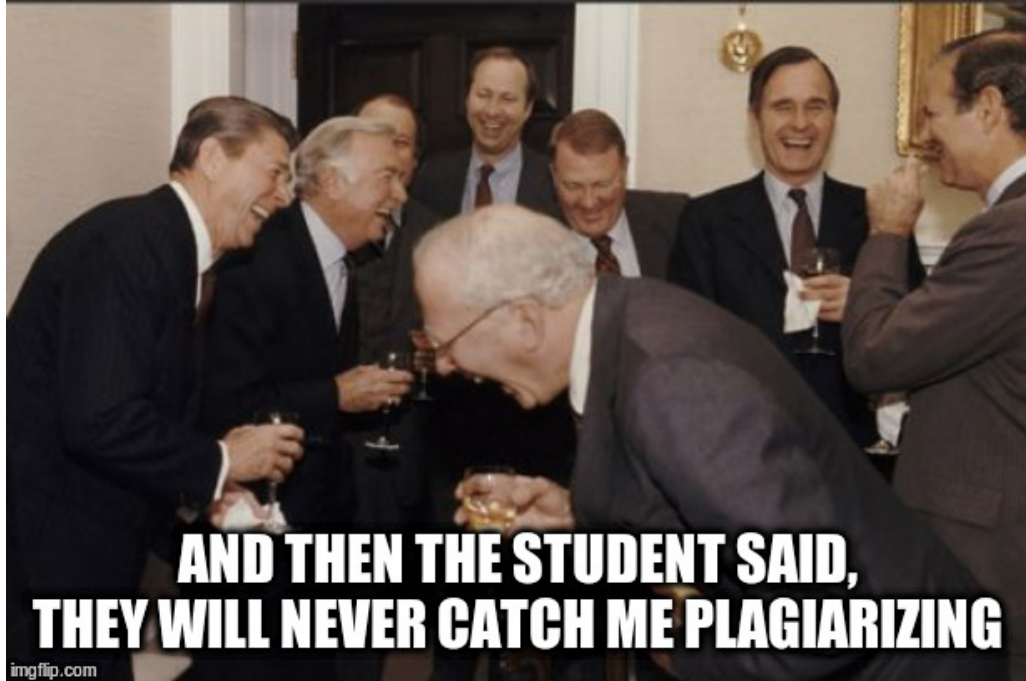
Tutor: negative grade

Student:



Plagiarism is strictly forbidden, so keep in mind that

- Every assignment will be checked
 - ...once all assignments are completed.
- Different names for variables → will have no effect!
- Shuffling code snippets → will have no effect!
- NO EXCEPTIONS!
- All people involved have to take the consequences



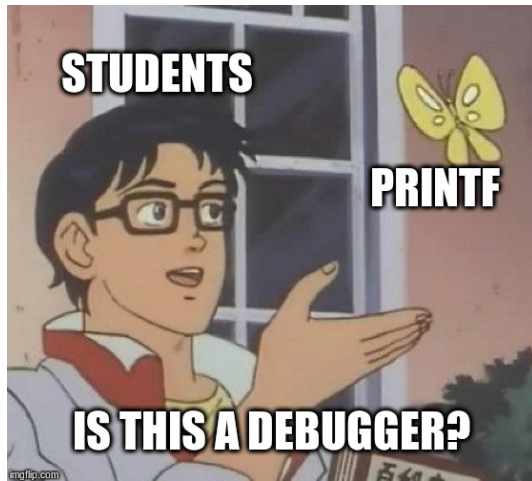
**AND THEN THE STUDENT SAID,
THEY WILL NEVER CATCH ME PLAGIARIZING**

Working on Assignments

What are your tasks

- Read the assignment [rules](#)!
- Join the [IAIK Discord: https://discord.gg/DCpzjqWBD3](https://discord.gg/DCpzjqWBD3)
- Pull from upstream before you begin.
- Understand the assignment specification,
- Have an in-depth understanding of your solution, and
- implement your solution yourself.
- Do not remove tags, after the deadline!!!
- pro advice: use gdb for debugging and valgrind for memory checks

Debugging using a debugger



Assignment grading contd'

Each assignment graded individually with the help of the test system

- 105 points reachable
- stable solutions that are in line with the rules
- If you are not sure about something: *ask*

Your submissions are tested automatically

- Subset of tests is revealed (=sanity checks)
- Passing all sanity checks does *not* mean 100% on all tests

Assignment Grading contd'

Interviews

- during the semester, after:
 - A1, A2
 - A3, A4
 - A5, A6
- you select a time slot, but get a **random** tutor
- points can be lost, but additional points can be awarded

You may have to code something or be asked about many your own code with small variation

Assignment Grading contd'

Magic coins

- A0 rewards you with up to 100 coins when completed
- Assignment handed in an hour early: +1 Coin
- For each 10 min late: -1 Coin
- Max 48 hrs for a late submission
- Coins can be converted into bonus points
- Exchange rate: 1pt/50coins

Exam and Overview of grading

- mandatory
- 30 pts reachable
- $\geq 50\%$ of points needed

Positive grade:

- Exam: ≥ 15 pts
- Assignments: ≥ 55 pts
- but overall score has to be over $\geq 50\%$

In numbers:

- Grading (max. 135 points):
 - ≥ 118 points $\rightarrow 1$
 - ≥ 101 points $\rightarrow 2$
 - ≥ 84 points $\rightarrow 3$
 - ≥ 75 points $\rightarrow 4$

Working Environment

We recommend to use Linux

- e.g., [Ubuntu](#)
- use gcc/g++, gdb and valgrind

Support Channels & Feedback

Support

- [Course website](#)
- Discord: [IAIK Discord](#)
- [studø](#)

registration	
Number of exam dates per semester	continuous assessment
Statistical evaluation of exam results	Details
Further information	
Recommended reading	
Online information	[online information]
	[course materials]
	[e-learning course]
Note	

Give us feedback

- Anytime you think something could be improved
- Evaluation at the end of the course

Changes this year

- exercise interviews during the semester
- no second chance for exercises or exercise interviews
- second chance exam still exists

Code-Fixing Challenge (A0)

The Challenge

- Not mandatory and for **self-assessment only!**
- Self-assessment – **max. 1 hour.**
- No grading, but coins as reward
- You can quit after A0, without getting graded
- The challenge is open on **Thursday (today) from 7pm to 8pm.**
- Pull from upstream

Multithreading (A1)

Assignment 1 Overview

“sea rescue game”

- an ASCII computer game
- because of a lazy tutor, you get a version without threads → not really playable
- TASK: fix it and make it fun to play

Synchronization (A2)

A2-First step

- Pull from upstream
- Try `mkdir build && cd build; cmake ..; make` and execute
- It will not work ;-)
- Fix it

- Changing core functionality/output of the program → 0 points
- Parts you may and should modify are marked with **TODO BEGIN** and **TODO END**
- Do not make unnecessary changes

A2-What do we need?



- Locks:
 - Mutex
 - Semaphore
 - Condition variable
- Use Posix locks!
- Hint: there will be lectures on this topic

A2-Typical errors

- So, how to lock correctly?
- You need to hold the lock as long as you need the shared resource
- Carefully keep track of the sequence you've locked
- Always should be the same sequence

A2-Typical errors contd

Will work, but has a very bad performance. Maybe nothing can happen simultaneously because of the way it is locked.



A2-Typical errors contd

THREAD 1

```
// ...  
lock(harddisk);  
lock(floppy);  
copySomething(floppy, harddisk);  
unlock(floppy);  
unlock(harddisk);  
// ...
```

A2-Typical errors contd

THREAD 1

```
// ...  
lock (harddisk);  
lock (floppy);  
copySomething (floppy , harddisk);  
unlock (floppy);  
unlock (harddisk);  
// ...
```

THREAD 2

```
// ...  
lock (floppy);  
lock (harddisk);  
copySomething (floppy , harddisk);  
unlock (harddisk);  
unlock (floppy);  
// ...
```

A2-Typical errors contd

Results in a deadlock.



Program, Process, Thread

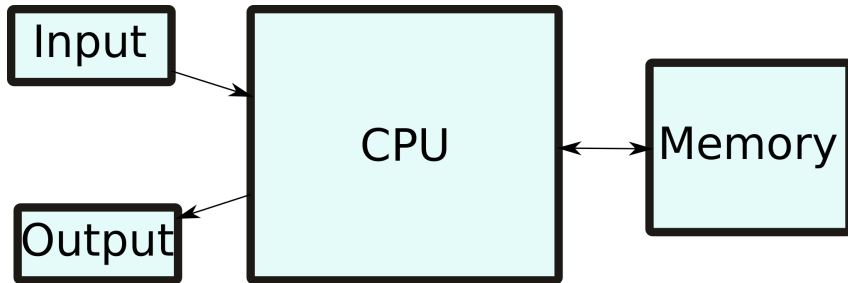
- A program: a binary file containing code and data
 - actions: write, compile, install, load
 - resources: file
- A thread: an execution context
 - actions: run, interrupt, stop
 - resources: CPU time, stack, registers
- A process: a container for threads and memory contents of a program
 - actions: create, start, terminate
 - resources: threads, memory, program

Abstractions

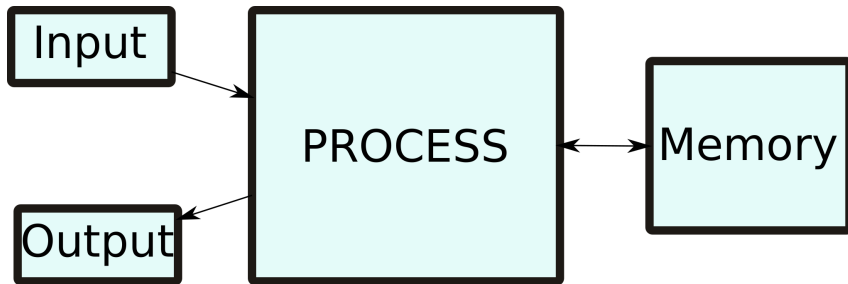
- Process: abstraction of a computer
- File: abstraction of a disk or a device
- Socket: abstraction of a network connection
- Window: abstraction of a display

→ Abstractions hide many details but provide the required capabilities

CPU vs. Process



CPU vs. Process



Program, Process, Thread

- Once a program is loaded in memory, OS can start it(s first thread) by
 - setting up a stack and setting the stack pointer and
 - setting the instruction pointer (of the first thread) to the programs first instruction
- Process is an instance of a program

Threads

Process can have multiple threads

- same program code and data
- own stack
- own registers (including instruction pointer)

Program, Process, Thread

- **Program:** a binary file containing code and data
 - a mold for a process
- **Thread:** an execution context
 - a sequence of instructions
 - if part of a process: restricted to the boundaries of a process
- **Process:** a container for threads and memory contents of a program
 - an instance of a program
 - restricted to its own boundaries and rights

Process Resources

A process is a container.

- Process ID
- Filename
- Program file
- File descriptors
- Memory
- Accounting
- Threads
- Child processes?

Thread Resources

A thread is a unit for execution.

- Thread ID
- Thread state (Running, Sleeping, ...)
- A set of register values
- A stack

Process and Thread Interaction

Load program, create process, ...

- 1 initial thread
- executes the `main()`-function
- it's not a "main"-thread
- process may start further threads if required (how?)

ELF Header:

```
Magic:    7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:    ELF64
Data:    2's complement, little endian
Version:  1 (current)
OS/ABI:   UNIX - System V
ABI Version: 0
Type:    DYN (Shared object file)
Machine:  Advanced Micro Devices X86-64
Version:  0x1
Entry point address: 0x1050
Start of program headers: 64 (bytes into file)
Start of section headers: 14680 (bytes into file)
Flags:    0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 11
Size of section headers: 64 (bytes)
Number of section headers: 29
Section header string table index: 28
```

43:	0000000000001000	0	FUNC	LOCAL	DEFAULT	11	_init
44:	0000000000001200	1	FUNC	GLOBAL	DEFAULT	14	__libc_csu_fini
45:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterTMCloneTab
46:	0000000000004000	0	NOTYPE	WEAK	DEFAULT	23	data_start
47:	0000000000004010	0	NOTYPE	GLOBAL	DEFAULT	23	_edata
48:	0000000000001204	0	FUNC	GLOBAL	HIDDEN	15	_fini
49:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__stack_chk_fail@@GLIBC_2
50:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_
51:	0000000000004000	0	NOTYPE	GLOBAL	DEFAULT	23	__data_start
52:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
53:	0000000000004008	0	OBJECT	GLOBAL	HIDDEN	23	__dso_handle
54:	0000000000002000	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
55:	00000000000011a0	93	FUNC	GLOBAL	DEFAULT	14	__libc_csu_init
56:	0000000000004018	0	NOTYPE	GLOBAL	DEFAULT	24	_end
57:	0000000000001050	43	FUNC	GLOBAL	DEFAULT	14	_start
58:	0000000000004010	0	NOTYPE	GLOBAL	DEFAULT	24	__bss_start
59:	0000000000001155	65	FUNC	GLOBAL	DEFAULT	14	main
60:	0000000000001135	32	FUNC	GLOBAL	DEFAULT	14	_Z8isdouble0i
61:	0000000000004010	0	OBJECT	GLOBAL	HIDDEN	23	__TMC_END__
62:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_registerTMCloneTable
63:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@@GLIBC_2.2

Process Creation

- at boot time (kernel threads, init processes)
- **at request of a user (how?)**
 - also: start of a scheduled batch job (cronjob, how?)

Process Creation at request of a user

via Syscall!

- UNIX/Linux: `fork` (exact copy)
- Windows: `CreateProcess` (new image)



WHAT DOES THE FORK DO?

Process Creation via fork (on Unix / Linux)

```
pid_t fork(void);
```

The fork() function shall create a new process. The new process (child process) shall be an **exact copy** of the calling process (parent process) **except** as detailed below:

- unique PID
- copy of file descriptors
- semaphore state is copied
- shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources.
- parent and the child processes shall be capable of executing independently before either one terminates.
- ...

fork Return Value

```
pid_t fork(void);
```

Upon successful completion, `fork()` shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the `fork()` function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and `errno` shall be set to indicate the error.

Fork

```
pid_t child_pid;
child_pid = fork();
if (child_pid == -1) {
    printf("fork failed\n");
} else if (child_pid == 0) {
    printf("i'm the child\n");
} else {
    printf("i'm the parent\n");
    waitpid(child_pid, 0, 0); //
        wait for child to die
}
```

- child does not know the parent
- parent knows the child
- parent waits for child to die
(waitpid)

I never knew my real dad



I never knew my real dad



I never knew my real dad



VIA GIPHY VIA

We never knew our real dad



```
int execl(const char *pathname, const char *arg, ... /* (char *) NULL */
);
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
int execl_e(const char *pathname, const char *arg, ... /*, (char *) NULL,
char * const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

```
int execlpe(const char *file, char *const argv[], char *const envp[]);
```

- replace running process by process defined by `file`
- pass `argv`
- use `envp` for environment variables (PATH etc.)

Process Termination

- Normal exit (return value: zero)
- Error exit (return value: non-zero)
- Fatal error (e.g. segmentation fault)
- Killed by another process

Process Hierarchies

Some operating systems have hierarchies:

- implicit hierarchy from forking
- process groups in UNIX/Linux
- doesn't exist in Windows

Implicit parent-child hierarchy on Unix/Linux:

- when parent dies, all children, grand-children, grand-grand-children, . . . , die aswell
- UNIX/Linux also cheats a bit: parent process typically inherits a processes' children, etc.

Process/Thread State

```
git grep TODO | sort
```

- sort has to wait for input
- what does the sort do in the meantime?
 - loop and check (busy wait)
 - sleep and get woken up
- blocking the process makes sense
- do we actually block the process?

Processes vs. Threads

- Threads are more lightweight than processes
- Less independent than processes
- No protection

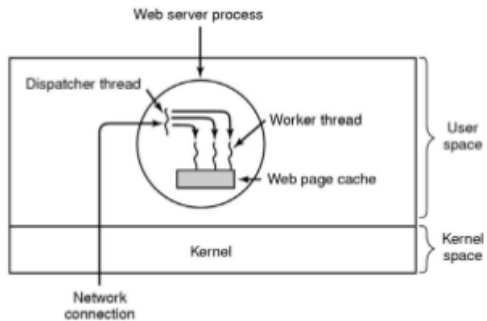
Why threads?

- Things should happen in parallel - even within one application
- Example: text processing
 - typing
 - spell checking
 - formatting on screen
 - automatically saving
- Some of these things may block
 - wait for mouse-click / keyboard press
 - wait for disk
 - etc.

Why threads

- Make programming easier
 - Split tasks in different blocks
 - Like with processes
 - But they cooperate easily because of the shared address space
 - Consumes less memory
 - Attention: do not confuse *shared memory* (between processes) with *shared address space* (between threads)
- Switching between threads can be faster
 - No need to reconfigure memory
- May achieve better performance

Example



Example

```
while (TRUE)
{
    get_next_request (&buf);
    handoff_work (&buf);
}
while (TRUE)
{
    wait_for_work (&buf);
    look_for_page_in_cache (&buf, &page);
    if (page_not_in_cache (&page))
        read_page_from_disk (&buf, &page);
    return_page (&page);
}
```

Without Threads

Without threads,

- just one thread
- complicated program structure
- read content from disk may block process
- non-blocking read (polling!) decreases performance

Non-Blocking Read

```
while (TRUE) { // VERY simplified
  get_next_event(&buf);
  if (is_request_event(&buf)) {
    if (page_not_in_cache(&page)) {
      request_page_from_disk(&buf, &page);
      save_request_in_table(&buf);
    } else {
      return_page(&page);
    }
  } else if (is_disk_event(&buf)) {
    find_request_in_table(&buf);
    mark_request_as_done(&buf);
    return_page(&page);
  } else if (is_...
```

Non-Blocking Read

- Finite-state-machine!
- Actually simulates threads
- Better: **use multithreading**

How to use multithreading?

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```



**WHAT KIND OF SORCERY IS
THIS?!**

Function Pointer

- `void * (*start_routine) (void *)`
- looks wrong...
- `void* (*start_routine) (void*)`
- much better...

Function Pointer

- `void* (*start_routine) (void*)`
- `start_routine` is the function pointer **name**
- type: `void* (*) (void*)`
- `(*)` indicates **this is a function pointer**
- takes a `void*`
- returns a `void*`

Let's make a function pointer for `main`

```
int main(int argc, char *argv[])
```

- Function pointer: (*)
- +argument parenthesis: (*) ()
- +return type: int (*) ()
- +first argument: int (*) (int)
- +second argument: int (*) (int, char* [])

Function Pointer

- `void* (*start_routine) (void*) = &main;?`
- type doesn't match... what now?
- cast:

```
void* (*start_routine) (void*) = (void* (*)(void*))&main;
```

Dangerous though ;)

How to use multithreading?

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

The `pthread_create()` function starts a new thread in the calling process. The new thread starts execution by invoking `start_routine()`; `arg` is passed as the sole argument of `start_routine()`.

The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using `pthread_attr_init` and related functions. If `attr` is `NULL`, then the thread is created with default attributes.

Before returning, a successful call to `pthread_create()` stores the ID of the new thread in the buffer pointed to by `thread`; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

How to use multithreading?

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

- pthread_t = thread ID
- pthread_t*? call by reference

How do pthreads terminate?

The new thread terminates in one of the following ways:

- It calls `pthread_exit`, specifying an exit status value that is available to another thread in the same process that calls `pthread_join`.
- It returns from `start_routine()`. This is equivalent to calling `pthread_exit` with the value supplied in the return statement.
- It is canceled (see `pthread_cancel`).
- Any of the threads in the process calls `exit`, or the main thread performs a return from `main()`. This causes the termination of all threads in the process.

How do pthreads terminate?

```
void pthread_exit(void *retval);
```

- The `pthread_exit()` function terminates the calling thread and returns a value via `retval` that (if the thread is joinable) is available to another thread in the same process that calls `pthread_join`.
- After the last thread in a process terminates, the process terminates as by calling `exit` with an exit status of zero; [...]

Waiting for threads

```
int pthread_join(pthread_t thread, void **retval);
```

- The `pthread_join()` function waits for the thread specified by `thread` to terminate. If that thread has already terminated, then `pthread_join()` returns immediately.
- If `retval` is not `NULL`, then `pthread_join()` copies the exit status of the target thread into the location pointed to by `retval`. If the target thread was canceled, then `PTHREAD_CANCELED` is placed in the location pointed to by `retval`.

Killing threads

```
int pthread_cancel(pthread_t thread);
```

- The `pthread_cancel()` function sends a cancellation request to the thread `thread`.

Take Aways

- Processes divide resources amongst themselves (except processor time)
- Threads divide processor time amongst themselves (and a few resources)
- Sometimes processes are more appropriate, sometimes threads

