Processes

IAAS Lectures

University of Stuttgart

Brian Setz, MSc

brian.setz@iaas.uni-stuttgart.de

Institute of Architecture of Application Systems

Introduction - Lab Sessions

- Lab Sessions
  - Focus on Linux, understand how Linux functions
  - Interactive examples and demonstrations
  - Assignments at the end of every lab

- Grading: pass / resubmit / fail
  - Max. 2 fails to pass the lab → requirement for exam
  - Not handing in on time → fail
  - Not handing in as PDF → fail

- Material from labs can (will) be covered on the exam

Format

- 7 exercises (Lab sessions on Friday at 14:00-15:30 **according to lab schedule slide**) → Exercises in **<u>groups of two</u>**
  - Requires (root) access to Linux (supported: GNU/Linux Debian)
  - No laptop / PC? Find a partner (possibly use AWS Free Tier)
  - Apply theory from lectures in a Linux environment
  - Copying exercises is forbidden, your responsibility to keep your answers private
    - Cite your sources
- **Deadline: see the deadline slide for the exact time and date**
  - Submission: ILIAS → Operating Systems (neue PO) → 0 Exercises → Submission
  - Write a short, coherent report, include lab number, names + student numbers, submit as <u>PDF</u> before the deadline
  - Not only results, include all the steps you took to get to the results
- **Before Monday 28.10.2019 23:59:59, send email to Brian with the <u>two names + student numbers</u> of your group members →** brian.setz@iaas.uni-stuttgart.de

Introduction – Preliminary Schedule Labs @ 14:00-15:30

Date Lab	Lab Topic
18.10.2019 @ 14:00	*No lab*
25.10.2019 @ 14:00	**1: Introduction, Linux, Virtualization**
08.11.2019 @ 14:00	**2: Process Management**
15.11.2019 @ 14:00	*No Lab*
22.11.2019 @ 14:00	*No Lab*
29.11.2019 @ 14:00	**3: Process Scheduling**
06.12.2019 @ 14:00	Q&A
13.12.2019 @ 14:00	**4: System Calls + Interrupts**
20.12.2019 @ 14:00	*Q&A*
10.01.2020 @ 14:00	**5: Synchronization + Time**
17.01.2020 @ 14:00	**6: Memory Management**
24.01.2020 @ 14:00	Q&A
31.01.2020 @ 14:00	**7: Virtual File System & Block I/ O & I/O Systems**
07.02.2020 @ 14:00	Q&A

Lectures

Introduction – Preliminary Schedule Deadlines

Date Lab	Lab Topic
07.11.2019 @ 23:55:00	Deadline Lab 1
21.11.2019 @ 23:55:00	Deadline Lab 2
12.12.2019 @ 23:55:00	Deadline Lab 3
09.01.2019 @ 23:55:00	Deadline Lab 4
23.01.2019 @ 23:55:00	Deadline Lab 5
30.01.2019 @ 23:55:00	Deadline Lab 6
13.02.2019 @ 23:55:00	Deadline Lab 7

Additional Notes

- Only 1 submission per group
- Grading (pass / fail / resubmit) will be done within 2 weeks
- Read assignments and instructions carefully
  - Always include all steps
  - When implementation is required, include code
  - Not clear? Send an email
- Questions

Agenda

- Previously
  - Unix-Like History, GNU/Linux, Linux Kernel, Terminal
  - Virtual Machine, Virtual Box, Debian
  - Exercises Lab 1
- Today
  - Processes
  - Process descriptor and task structure
  - Process creation
  - Threads vs. Processes
  - Process termination
  - Crash-Course Kernel Modules

Processes

Processes Definitions

- Process → performs tasks in the operating system
- Program → Set of machine code instructions and data

- A process is a program that is executing

- Processes are isolated, have their own address space and cannot interact with other processes except through system calls

- Processes use many different system resources, needs to be tracked and managed

- Kernel has the task to manage processes and their resources

Kernel Process Management

- For each process, the kernel manages at least:
  - Open files
    - File descriptor count
  - Pending signals
    - SIGINT → interrupt process, SIGKILL → kill process, SIGCHLD → Child process stopped
  - Internal kernel data
    - Mappings to kernel addresses
  - Process state
    - Created, running, waiting, exited
  - Memory address space (mappings)
    - Virtual Memory (memory sandbox), page table
  - One or more threads
    - Track children, thread group IDs

Process Identifiers

- In Linux processes are identified by a numeric value called the process ID
- pid → process ID
  - Each process has a unique identifier
  - Defined as an `int` in the `task_struct`, max value: 32 768 (`short int`)
  - `cat /proc/sys/kernel/pid_max`
  - Limits process count → relevant for servers
- ppid → parent process ID: The process identifier of the parent process
- tgid → thread group ID: The ID of the process that started the thread
- tid → thread ID: The ID of the thread, unique for each thread
- Process has one thread? pid == tgid == tid
- Process has multiple threads? Same tgid for all threads, tgid == pid of first thread, unique tid per thread

init is the grandfather of all processes.

All processes running in Linux can trace their relationship back to init.

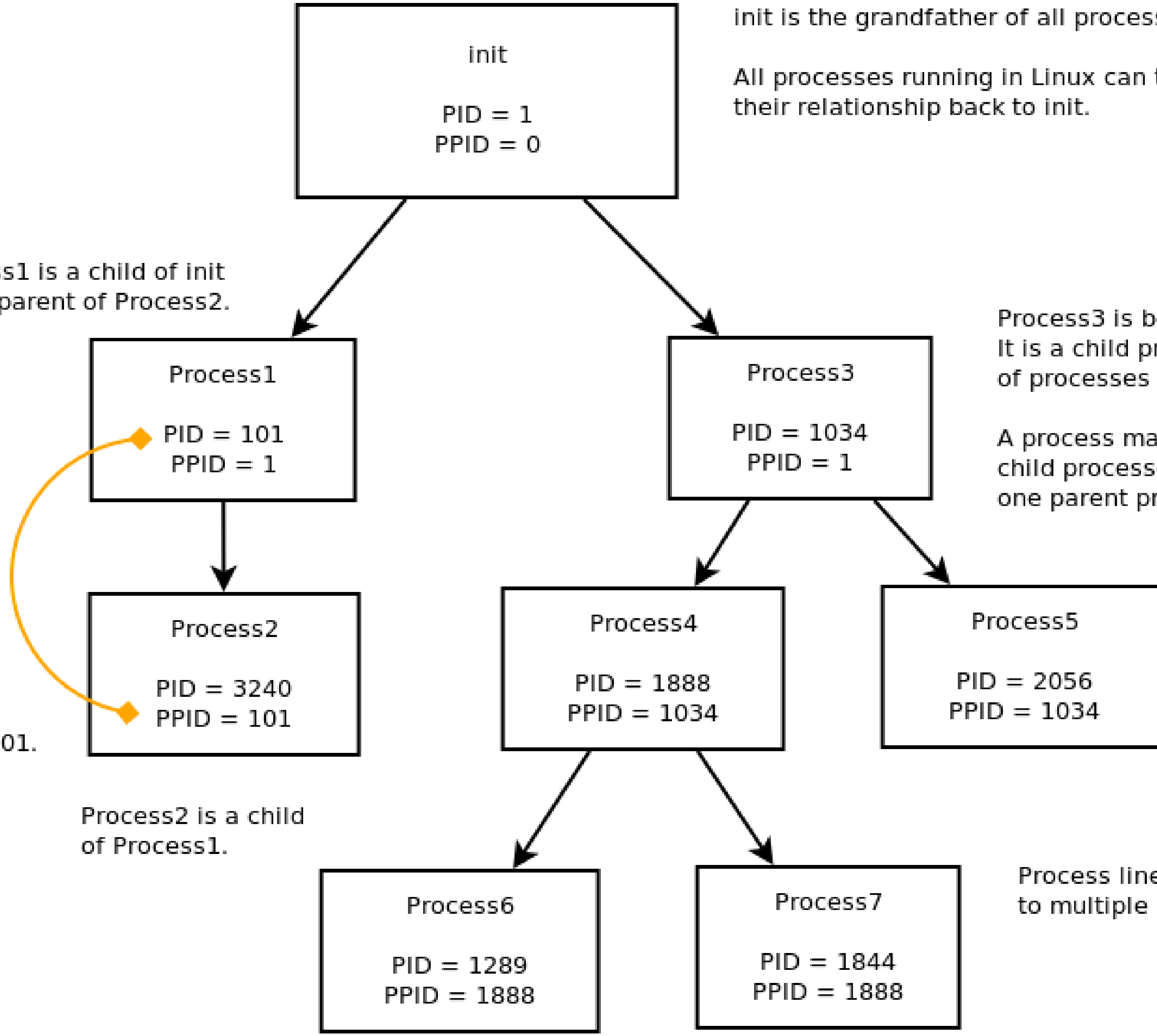
Process1 is a child of init and a parent of Process2.

Process3 is both a child and a parent. It is a child process of init and a paren of processes 4 and 5.

A process may have any number of child processes, but it may have only one parent process.

The PPID of a child is the same as the PID of its parent.

Process2 has a PPID of 101 because its parent is Process1, which has a PID of 101.



Process2 is a child of Process1.

Process lineage may extend to multiple levels.

Each PID is unique, but duplicate PPIDs are allowed since a parent process may have several child processes.

Lectures

Threads

- Thread → objects of activity within a process, execution flow
  - Program counter
  - Stack
  - Registers

- The Linux kernel schedules <u>tasks</u> NOT processes OR threads
  - Traditional UNIX → one process = one thread
  - Modern Linux → one process = at least one thread

- Linux threads are processes (tasks) that **can** share resources (e.g. memory, file descriptors, signals) → described in the process descriptor

Processes in Linux

(process id, thread group id)

- `pstree` command to view (sub) process tree

Shell

```
briansetz@debian:~$ pstree -pgh 1765
gnome-terminal-(1765,1765)─┬─bash(1827,1827)───pstree(2147,2147)
                           ├─{dconf worker}(1798,1765)
                           ├─{gdbus}(1782,1765)
                           └─{gmain}(1781,1765)
```

pstree process

Terminal Emulator

GNOME inter-process communication

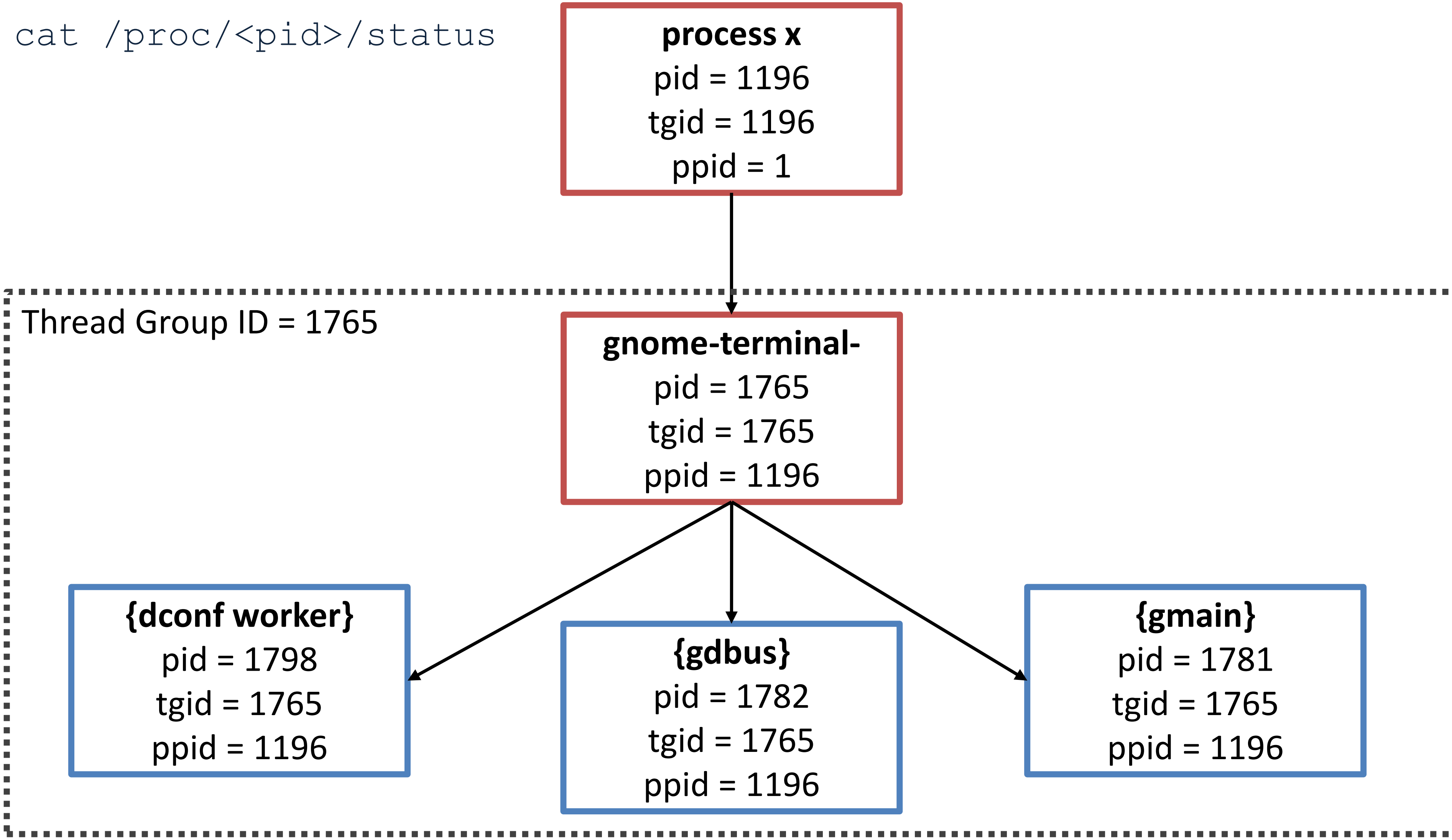
GNOME application settings

GNOME main event loop

- { <name> } → child thread of the parent

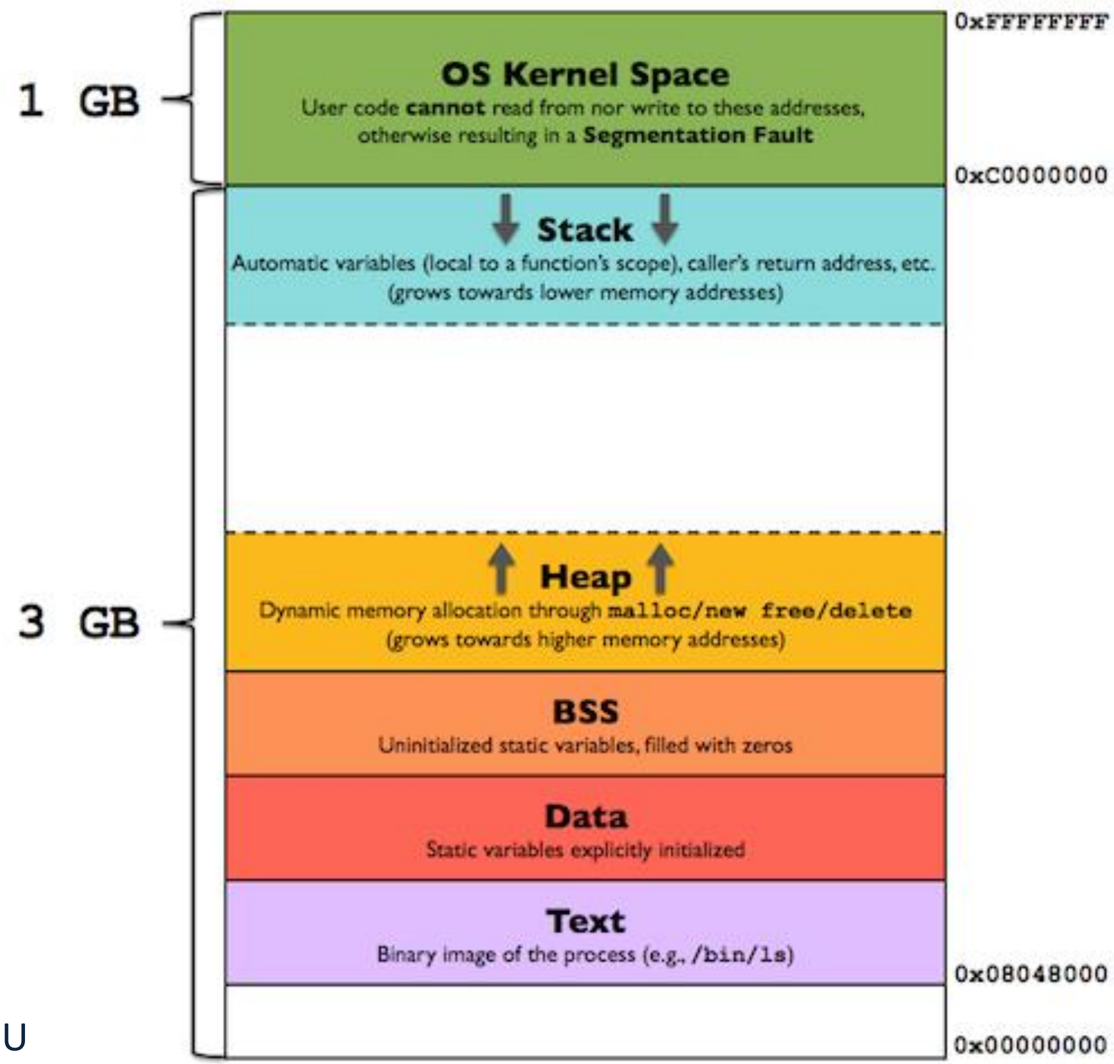
Process IDs

- `cat /proc/<pid>/status`



Program Structures

- Process → a program (= object code) that is executing
  - Program code (.text section)
  - Init. Global variables (.data section)
  - Uninit. Global variables (.bss section)
- Stack
  - Automatic variables (local)
  - Automatically allocated and freed
  - Managed by OS
- Heap
  - Dynamic allocations (global)
  - Managed by program
- OS Kernel Space
  - Shared between processes
  - Used as entrypoints to the kernel for the CPU



Intermezzo: struct

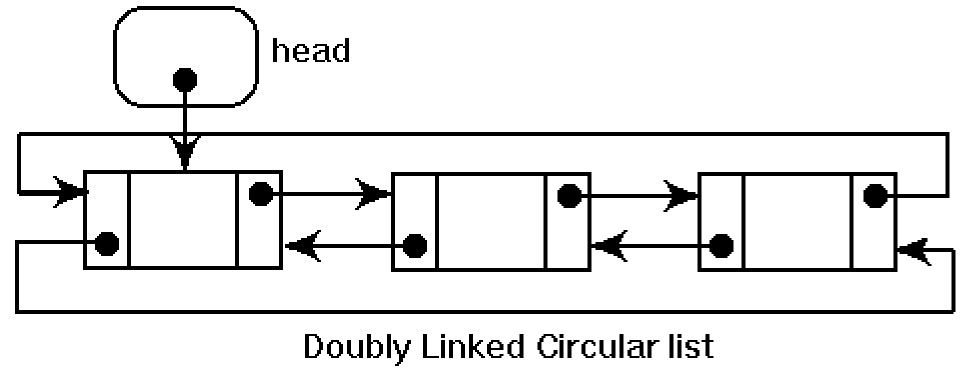
- Composite data type in the C programming language
  - Linux Kernel = 97% written in C

```
struct operating_system {
        int a = 1;
        char b[10];
        float *c;
};
```

- Contiguous block of physical memory
  - Only need one pointer to the start to access all the members
  - Int → 2 or 4 bytes, char → byte array (10 bytes), pointer → 4 or 8 bytes
- Great for data structures, many structs in the kernel

Process Descriptor (1)

- Kernel stores task (=process) list → list of processes → `tasks` vector
  - Circular doubly linked list → item: process descriptor



- Process Descriptor → contains all data that describes the executing program
  - `struct task_struct` → pid, priority, state, parent process (task_struct) …….
  - Kernel data structure (exists in kernel space)
- `task_struct` process descriptor
  - https://github.com/torvalds/linux/blob/master/include/linux/sched.h#L593

Process Tree

- /sbin/init process → PID = 1
  - Reads init scripts, executes startup programs, completes boot process

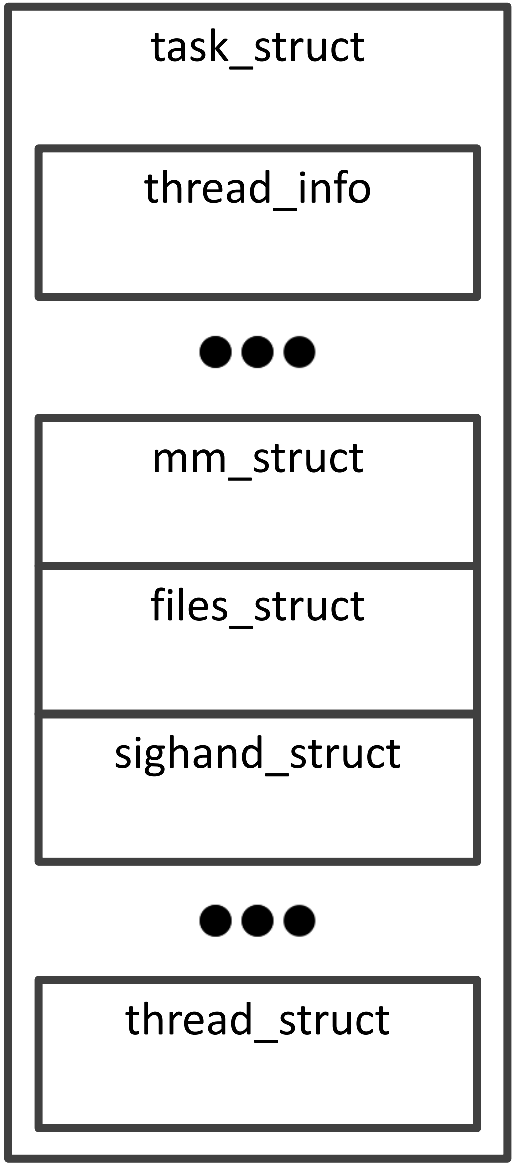
- Process has 1 parent
  - Defined in the `task_struct` as parent → pointer to another `task_struct`
    - `struct task_struct *real_parent;`
  - Also 0-n children in `task_struct` → pointer to list of `task_struct`
    - `struct list_head children;`

- Ways to iterate over processes:
  - Traverse the tree using parent and children using `real_parent` and `children`
  - Iterate the process list (circular doubly linked list) using `for_each_process()`
    - https://github.com/torvalds/linux/blob/master/include/linux/sched/signal.h#L565

Lectures

task_struct

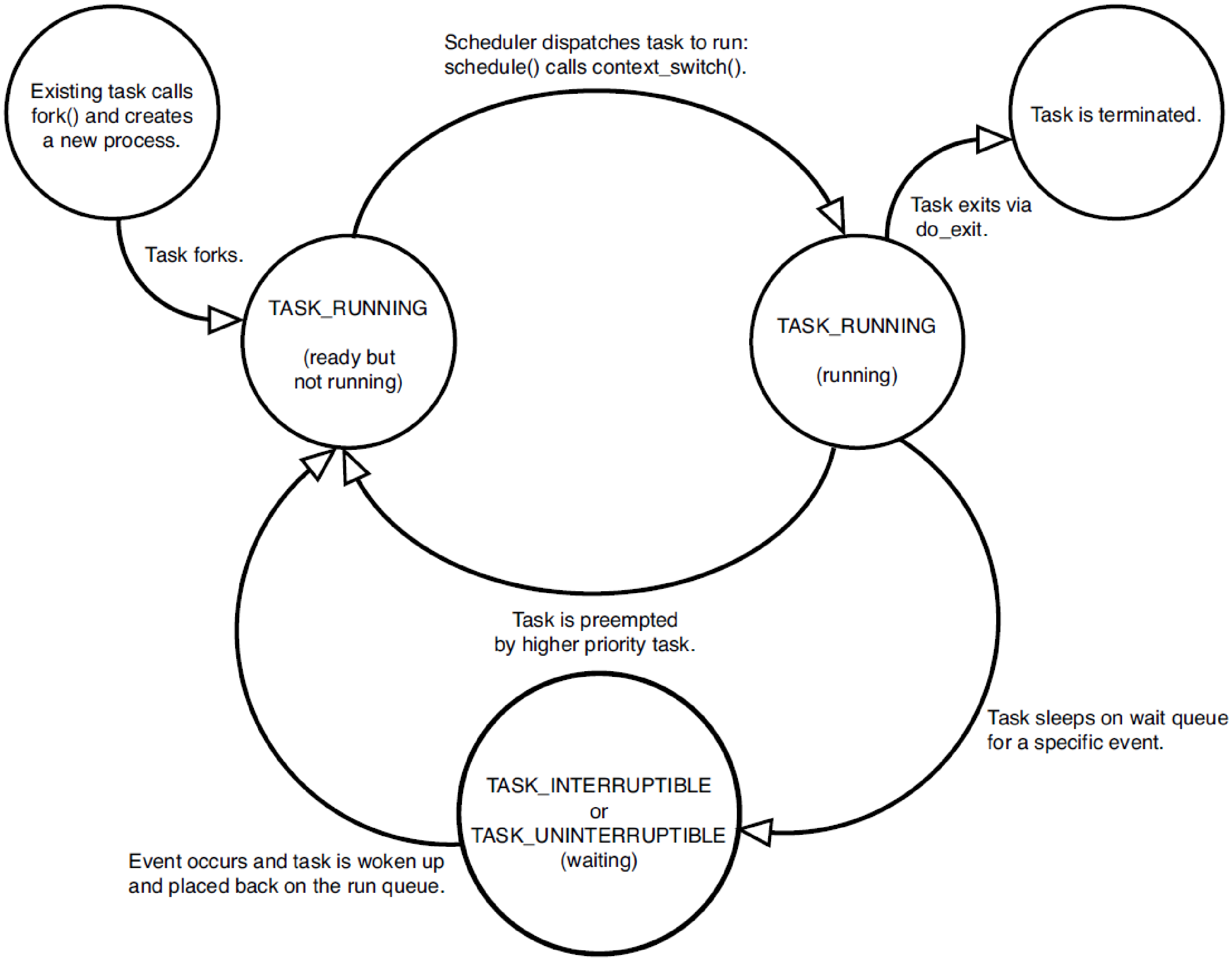
- struct thread_info
  - Low level thread data, pointer to task_struct
- struct mm_struct
  - Memory mappings
- struct files_struct
  - Open files
- struct sighand_struct
  - Signal handlers
- Many, many others
  - 1000’s LoC
- struct thread_struct
  - Registers, e.g. stack pointer, data segment



Process Descriptor (2)

- `struct thread_struct`
  - Processor architecture dependent → size of struct depends on processor architecture
    - Therefore this struct is at the end of the `task_struct`
  - https://github.com/torvalds/linux/tree/master/arch /<processor>/include/asm/processor.h
  - Some architecture have vastly different features (e.g. more registers)
    - i386 → 16 registers
    - x86-64 → 40 registers

Process State



Process State Flags

- TASK_RUNNING
  - Currently running or waiting in the run-queue
  - All processes in user-space have this state
- TASK_INTERRUPTIBLE
  - Asleep, waiting for some condition, event, or signal
- TASK_UNINTERRUPTIBLE
  - Asleep, but cannot be woken up by signals
- TASK_STOPPED
  - Execution stopped, after receiving signal (e.g. SIGSTOP)
- TASK_ZOMBIE
  - Execution stopped, waiting for parent process to collect exit code

Process Context

- Process executes program code from executable file
  - Executed within program's address space
  - Occurs in user-space

- Program executes system call or triggers exception
  - Enters kernel-space
  - Kernel executes on behalf of process → "in process context"

- All access to the kernel is through system calls and exception handlers

Process Creation

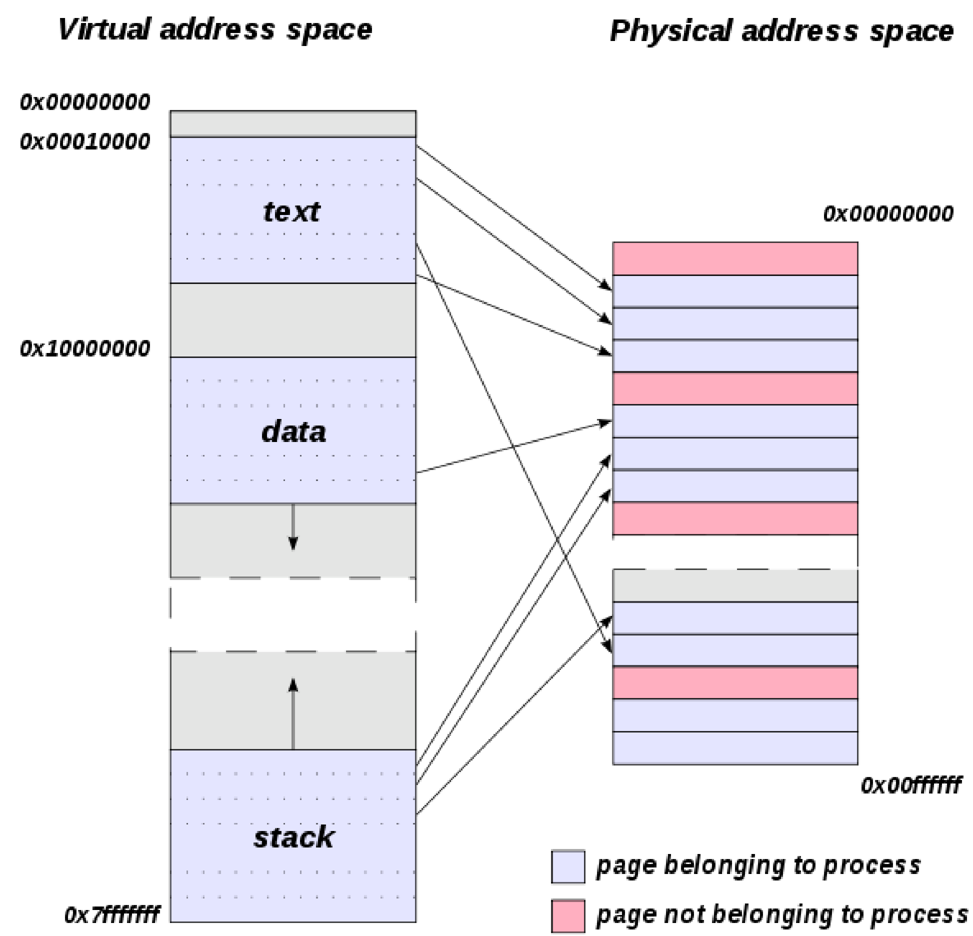
- Traditional *Spawn* mechanism to create process (`posix_spawn()`)
  - Create new process (descriptor)
  - Create new address space
  - Read executable
  - Begin execution

- Unix
  - Separated in `fork()` and `exec()` functions
    - fork() → Create child process that is a full copy (duplicates process) of parent process (PID, PPID, some resources, not copied), does not share resources
    - exec() → Loads new executable in address space (replaces process) and begins execution

Process Creation Linux

- Linux Way → Copy-on-write
  - Instead of copying all resources on `fork()`, the copying of data is delayed

- Page → block of contiguous virtual memory with fixed length described by a single entry in the page table
- Page table → per process table that maps virtual to physical addresses
- Page table structure is copied, not pages themselves
- Only when child writes to page, the page is copied (copy-on-write)
- CPU vs Memory trade-off



Child process creation

- To create a child process
- Call `clone()` → `do_fork()` → `dup_task_struct()`
  - Creates new kernel stack, `thread_info` and `task_struct` (identical to parent)
  - Some values in `task_struct` are reset (e.g. stats)
  - Child process state = TASK_UNINTERRUPTIBLE
- Call `copy_process()` → `copy_flags()`
  - Update flags for process (e.g. super user privileges flag)
- Call `alloc_pid()`
  - Assign PID
- Share open files, file system information, signal handlers, process address space, namespace (usually shared between threads in same process)
- Return pointer to new child process → `exec()` → child runs first

Threads

- Multiple threads of execution → concurrency and parallelism
  - Share memory address space, files, other resources

- Linux has no explicit concept of threads
  - Thread is just a process that happens to share resources with other processes
  - MS Windows has explicit constructs for threads (lightweight processes)
  - TGID → Thread Group ID (parent PID) and TID → Thread ID
    - `getpid()` always returns the TGID, despite its name!
- Threads created via `clone()`, resource sharing indicated by passed flags
- Kernel Threads → standard processes executing only in kernel space
  - Do not have own address space (shared with kthreads), do not switch to user-space
  - Created only by other kernel threads (parent is `kthreadd`)

Flag	Meaning
`CLONE_FILES`	Parent and child share open files.
`CLONE_FS`	Parent and child share filesystem information.
`CLONE_IDLETASK`	Set PID to zero (used only by the idle tasks).
`CLONE_NEWNS`	Create a new namespace for the child.
`CLONE_PARENT`	Child is to have same parent as its parent.
`CLONE_PTRACE`	Continue tracing child.
`CLONE_SETTID`	Write the TID back to user-space.
`CLONE_SETTLS`	Create a new TLS for the child.
`CLONE_SIGHAND`	Parent and child share signal handlers and blocked signals.
`CLONE_SYSVSEM`	Parent and child share System V `SEM_UNDO` semantics.
`CLONE_THREAD`	Parent and child are in the same thread group.
`CLONE_VFORK`	`vfork()` was used and the parent will sleep until the child wakes it.
`CLONE_UNTRACED`	Do not let the tracing process force `CLONE_PTRACE` on the child.
`CLONE_STOP`	Start process in the `TASK_STOPPED` state.
`CLONE_SETTLS`	Create a new TLS (thread-local storage) for the child.
`CLONE_CHILD_CLEARTID`	Clear the TID in the child.
`CLONE_CHILD_SETTID`	Set the TID in the child.
`CLONE_PARENT_SETTID`	Set the TID in the parent.
`CLONE_VM`	Parent and child share address space.

Lectures

fork() vs. clone()

- `fork()` → POSIX standard
  - Creates a new child process, no shared resources
  - In Linux, `fork()` calls `clone()` with specific parameters
    - `clone(…,SIGCHLD, 0);`
  - Child process execution continues from where `fork()` was called
  - No parameters

- `clone(fn,stack,flags,args)` → Linux specific = not portable
  - Creates a new child process, shared resources (depending on flags), e.g. new thread:
    - `clone(…,CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD, 0)`
  - Child process execution starts from the function that is passed to `clone()`
  - Many parameters, **note**: stack grows down, set stack pointer to highest address

Lectures

Process Termination

- Process terminates → kernel releases resources and notifies parent
- Steps
  - In `task_struct` set PF_EXITING flag
  - Call `del_timer_sync()` to remove kernel timers
  - Call `exit_mm()` to release `mm_struct` (if not shared → destroyed)
  - Call `exit_sem()` to stop Inter Process Communication (IPC)
  - Call `exit_files()` / `exit_fs()` to decrease file descriptor usage count
  - In `task_struct` set `exit_code`
  - Call `exit_notify()` to notify parent, in `task_struct` set `exit_state` EXIT_ZOMBIE
  - Now process is unrunnable (`schedule()` called to reschedule another process), only memory usage is from `task_struct`, `thread_struct`.
  - Wait for parent to collect exit information → remaining memory is freed

Parentless Child

- What if parent process exits before child process? → reparent
  - Another process in same thread group
  - init (PID = 1) process
- This process has to be executed for each (soon-to-be) parentless child
- Example `ptrace()` → process tracing
  - Temporarily changes process parent to the process that is ptracing
  - Parent can exit before child

Useful Terminal Commands

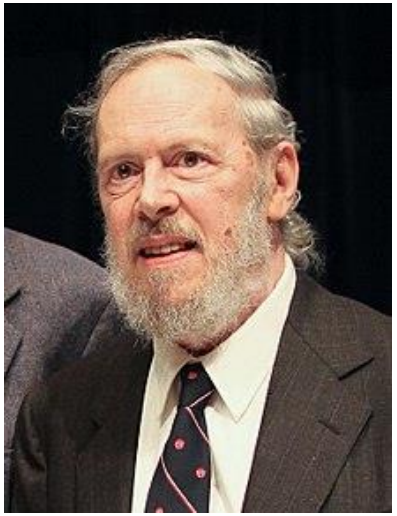
- `ps` command
  - Information about currently running processes
- `top` command
  - Interactive process information
- `pstree` command
  - Display the process tree for a given process ID
- `pgrep` command
  - Get PID by name
- `kill`, `pkill`, `killall` commands
  - Send (termination) signals to process by PID or name
- `cat /proc/<pid>/status`
  - Print process information

Mini Crash Course Kernel Modules

Introduction to C

- C is a general-purpose, imperative programming language
- 1972 – Dennis Ritchie (also co-creator of Unix) creates C language at Bell Labs
- 1978 – Publication of The C Programming Language book
- 1983 – American Nation Standards Institute begins standardization
- 1988 – ANSI C standard completed

- Key features
  - Structured programming
  - Lexical (static) variable scope
  - Recursion
  - Static type system



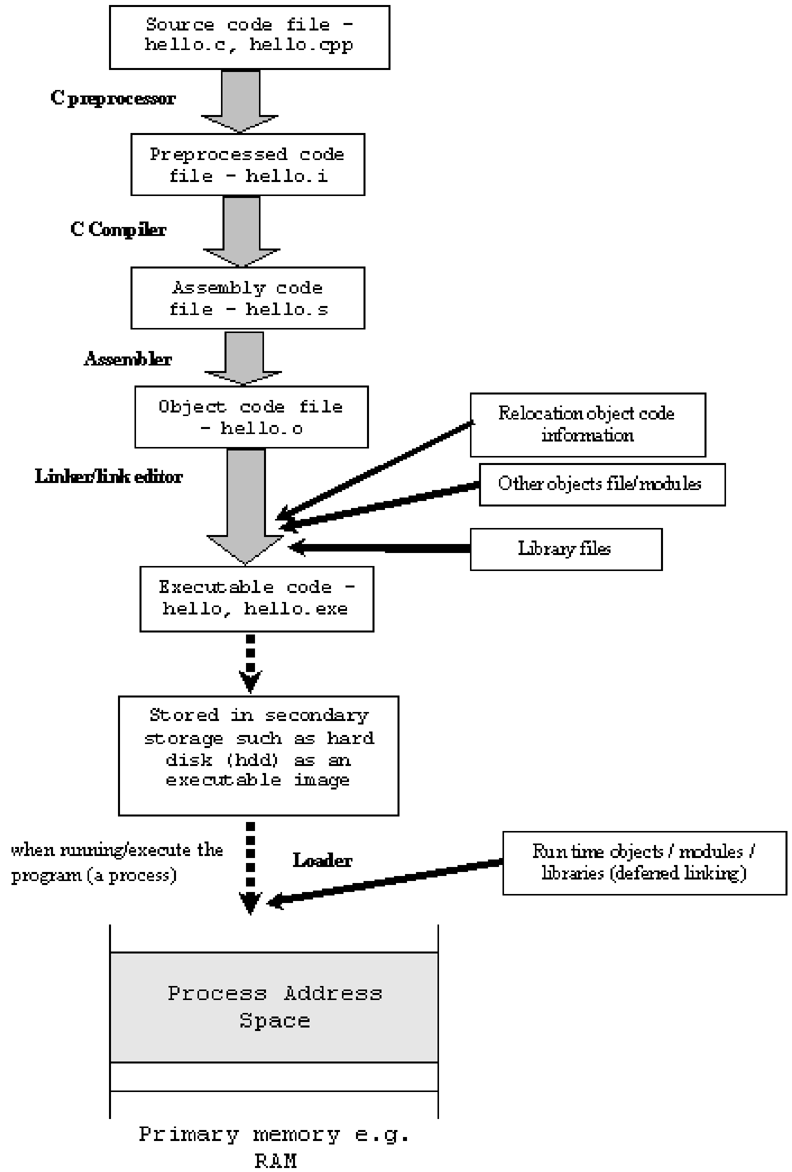
Dennis Ritchie

Pre-processor, compiler, assembler, linker, loader

- GNU C Compiler → `gcc` command
  - `apt install build-essential` (as root)
  - Runs the pre-processor, compiler, runs the assembler, optionally runs the linker

- Pre-processor → include files, conditional compilation, macros
- Compiler → take pre-processed source code, generate assembler code
- Assembler → translate assembler to machine code, generate object file
- Linker → takes object files and libraries, combines into one executable
- Loader → loads the program in memory (`exec()` system call), create process

Pre-processor, compiler, assembler, linker, loader



Hello World

- hello.c

System header file, declares interface to the system

Function definition. Main entry point, one per application

Write to stdout, declared in stdio.h

Return value of the function, exit code

```
#include <stdio.h>

int main( void )
{
        printf("Hello World!\n");
        return 0;
}
```

- To compile: `gcc -w -wError -o hello hello.c`

Compiler flag, all warnings enabled

Compiler flag, warnings to errors

Output executable filename

Files to compile

Makefile

- makefile → organize code compilation, `make` command
  - Defines a set of tasks to be executed

```
CC=gcc
CFLAGS=-w -Werror
TARGET=hello

all:
        $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c
```

CC, CFLAGS, and TARGET are constants

Make target named ‘all’

Translates to: `gcc -w -wError -o hello hello.c`

- Execute `make <target>`, to execute a specific target
- Execute `make`, first target will be executed

Kernel Module

- A kernel module is an object file (.ko) containing code to extend the running kernel
  - Support hardware
  - Add new filesystems
  - Introduce system calls
- Implemented in C
  - Access to kernel space and kernel functions and macros
    - `task_struct`, `for_each_process()`, `thread_info`, etc.
  - No user space libraries (e.g. standard C library)
- Compiled, assembled, but not linked until runtime (by the kernel)

Hello Kernel

- hello-module.c

```
#include <linux/module.h>
#define M_AUTHOR "Brian Setz <brian.setz@iaas.uni-stuttgart.de>"
#define M_DESC "Hello Module"
int init(void)
{
  printk(KERN_INFO "init(void)\n");
  return 0;
}
void exit(void)
{
  printk(KERN_INFO "exit(void)\n");
}
module_init(init);
module_exit(exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR(M_AUTHOR);
MODULE_DESCRIPTION(M_DESC);
```

Header file required for all modules

Definitions for module information

printk is kernel's version of printf

If not 0, installation is considered failed

Kernel levels defined in linux/kern_levels.h

Define module entry point

Define module exit point (clean up, etc.)

If not GPL, kernel is marked as tainted

Module information

Compile and Install

- `apt install module-assistant`
  - `m-a prepare` → installs kernel headers, when you update kernel, rerun

- `make` → builds the kernel object file (assuming the Makefile is there)
- `modinfo <x>.ko` → inspects the kernel module

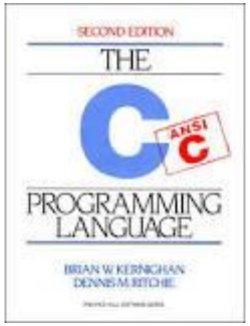
- `insmod <x>.ko` → installs the kernel module

- `cat /var/log/kern.log` → printk logs to the kern.log

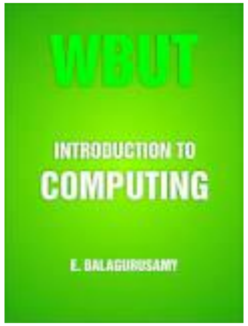
- `rmmod <x>` → uninstalls the kernel module

Resources

- Books
  - The C Programming Language (ANSI C), Kernighan & Ritchie



  - Programming in ANSI C, Balagurusamy (8^{th} edition, 2018)



- Many online resources and courses
  - C keywords: https://www.programiz.com/c-programming/list-all-keywords-c-language
  - Linux Kernel Module programming: https://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf

Summary

- Process vs. Thread
- Process Descriptor
- Process State
- Process Creation
- Process Termination
- Mini Crash Course C / Kernel Modules

Exercises

Exercises

Always include a full description (e.g. input / output, steps you take) of the performed tasks, implementation of code, in addition to the answers.

1. Use only the `ps` command to **(a)** list all processes (including UID) with parent process ID (PPID) 1. **(b)** Filter the list of processes such that only processes running as username root remain. **(c)** For the remaining processes, provide a 2-3 sentences describing their functionality.

Download `OS_Lab2.zip` and unzip it.

2. Open the `fork` directory and inspect the files. Compile the source file into a binary using the `make` command. **(a)** Run the binary and show the process tree using `pstree` and the parent PID. **(b)** Explain the number of running processes based on the source code in `fork.c`. **(c)** Do these processes share memory or other resources? Why (not)?
3. Open the `clone` directory. **(a)** Implement an application in C that uses a `fork()`, a `clone()` system call to create a process, and a `clone()` system call to create a thread. **(b)** For each of the create processes / threads print and show the TGID and PPID. **(c)** Explain why the TGIDs and PPIDs are as shown.
4. Open the `process-module` directory and inspect the files. **(a)** Implement the `list_processes` function such that it outputs a list of all processes including the executable name, process ID, and thread group ID. **(b)** Show the output of the kern.log related to your module. **(c)** Which parts of the kernel module execute in kernel space?

Lectures