Linux Kernel Development (LKD)

Session 3

Scheduling and System calls

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

2017

CISTER - Research Centre in Real-Time & Embedded Computing Systems
isep Instituto Superior de Engenharia do Porto
INESCTEC

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Scheduling Concepts

Process

- A process is an active program and related resources.
  - From the kernel's point of view, the purpose of a process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated.
- It may have one or more threads of execution
  - Each thread includes a unique program counter, process stack, and set of processor registers
- Provides two virtualisations, giving the illusion that it alone monopolises the system
  - Virtualised processor
  - Virtualised memory

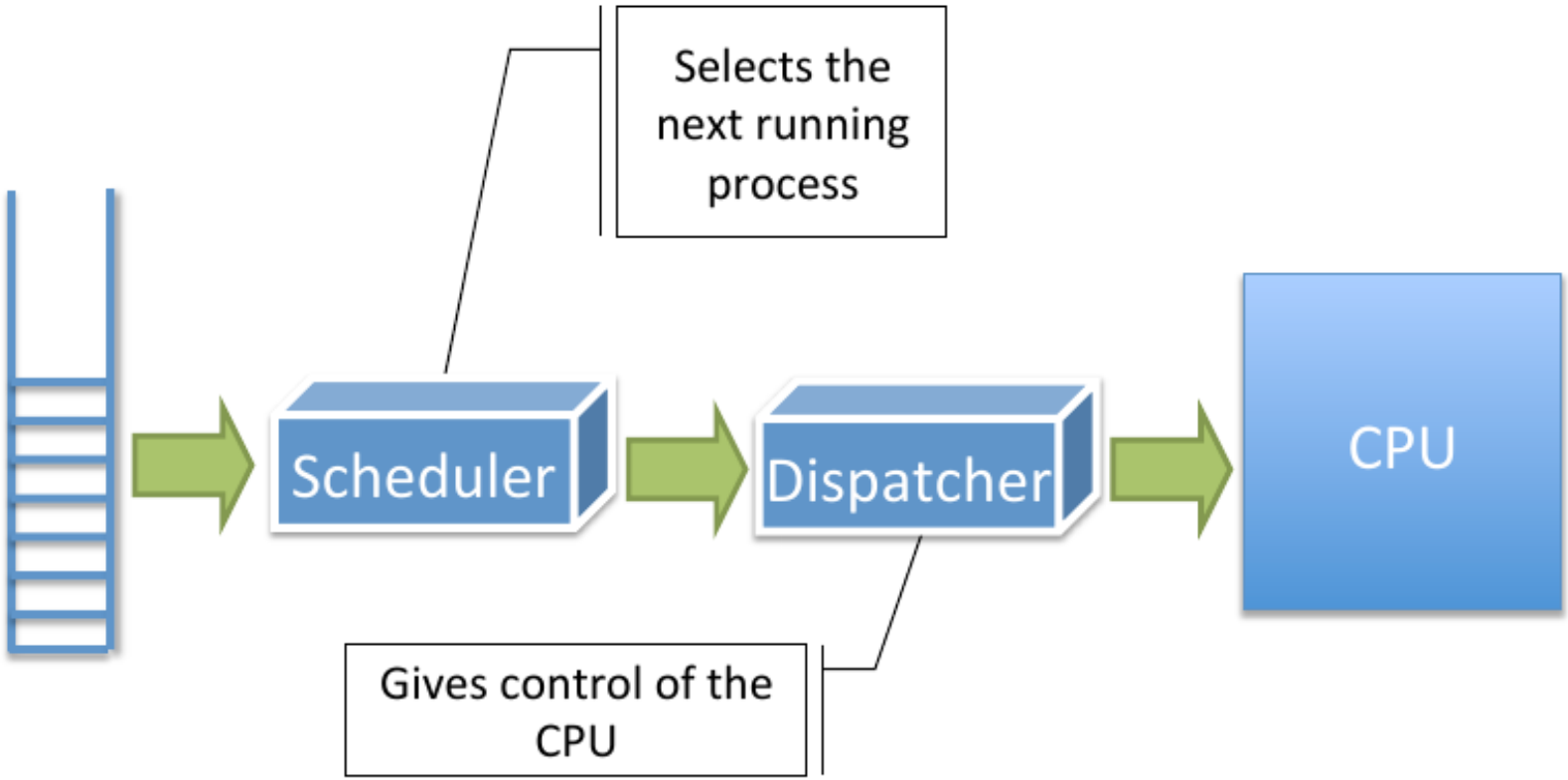
Multitasking

- Modern operating systems are able to run several processes at the same time
  - At least, this is the impression users get, even with only one CPU
- The kernel and the CPU create the illusion of multitasking by switching repeatedly between the different applications running on the system at very rapid intervals
- This gives rise to several issues that the kernel must resolve
  - Memory access: how to protect processes from one another?
  - Scheduling: which process to run and for how long?
  - Dispatching: how to switch processes?

Process scheduler and Dispatcher

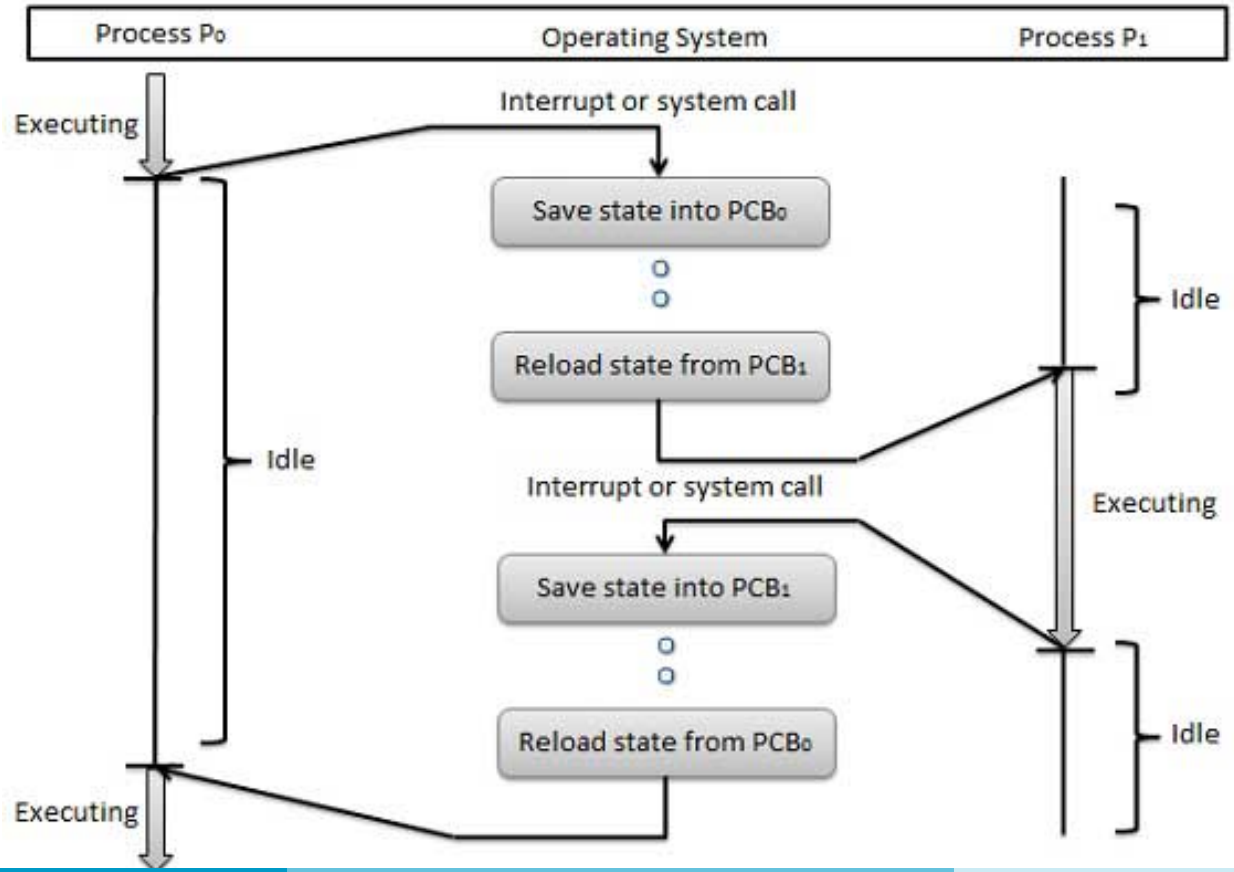
- The **process scheduler** decides which process runs, when, and for how long
  - It is the basis of a multitasking operating system
- By deciding which process runs next, the scheduler is responsible for best utilising the system
  - It divides the finite resource of CPU time between the runnable processes on a system
- It needs a scheduling policy, which defines the scheduling rules
- The **Dispatcher** is the module that gives control of the CPU to the process selected by the scheduler
  - Switching context;
  - Switching to user space.
  - ...

Scheduler and dispatcher



Context switching (I)

- The act of switching from one process to another
- The system has to:
  - Save the context of the current process
  - Restore the context of the new process

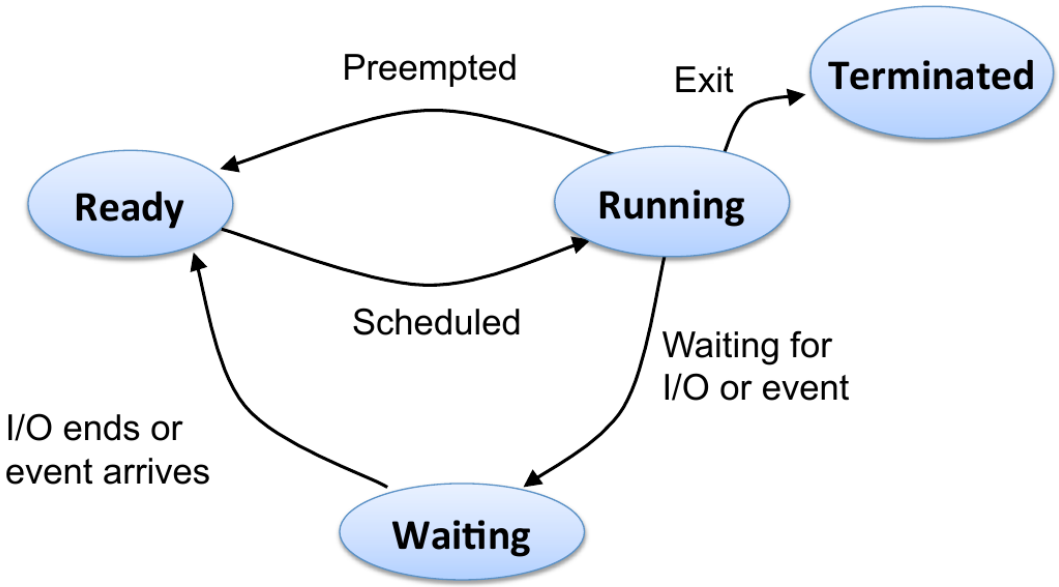


Context switching (II)

- What is the context of a process?
  - Program Counter
  - Stack Pointer
  - Registers
  - Code + Data + Stack (also called Address Space)
  - Other state information maintained by the OS for the process (open files, scheduling info, I/O devices being used, etc.)
- All this information is usually stored in a structure called **Process Control Block** (PCB)

Process State

- Multitasking implies process states.
  - Names for these states are not standardised, but they have similar functionality:
    - **Ready**: ready to run, but waiting to be scheduled;
    - **Running**: executing at the moment;
    - **Waiting**: waiting for I/O or an event;
    - **Terminated**: no longer ready to execute.



Scheduling events

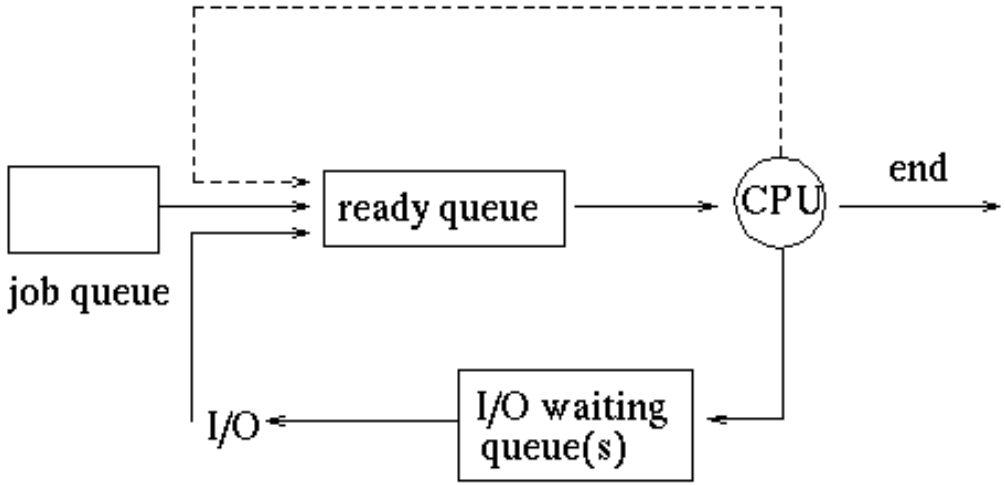
- A process switches from the **running** state to **waiting** state (e.g. I/O request);
- A process switches from the **running** state to the **ready** state (e.g. time slice expires);
- A process switches from **waiting** state to **ready** state (e.g. completion of an I/O operation);
- A process **terminates**.

Notice

All executing tasks go from **ready** to **running** state. Note that, there is only one task in the **running** state, per CPU.

Scheduling queues

- **Ready queue**: All processes that are ready for execution
- **Wait queues**: When a process is blocked in an I/O operation or waiting for an event, it is put in a device/event queue



Scheduling algorithms

Scheduling issues (I)

- **Preemptive**: The ability of the operating system to preempt or stop a currently scheduled process in favour of a “higher priority process”.
- **Non-preemptive**: Once the CPU has been allocated to a process, the process keep the CPU until it release the CPU either by terminating or by switching to waiting state. (Windows 95 and earlier)
- **Process priority**:
  - A numeric value that ranks processes based on their worth and need for processor time;
  - The general idea is that processes with a higher priority run before those with a lower priority
- **Time slice**:
  - A numeric value that represents how long a task can run until it is preempted.

Scheduling issues (II)

- Process types:
  - **I/O-bound**:
    - Spend much of their time submitting and waiting on I/O requests
    - Consequently, such processes are runnable for only short durations, because they frequently block waiting on more I/O
  - **CPU-bound**:
    - Spend much of their time executing code
    - Tend to run until they are preempted/finished because they do not block on I/O requests very often

Scheduling issues (III)

- **Starvation**: is a problem encountered in multitasking where a process is perpetually denied of necessary resources. Without those resources, the program can never finish its task;
- **Scalability**: the scheduler must scale well with a growing number of tasks.
- **Priority Inversion**: if using priorities, a low-priority task must not hold up a high-priority task;
- **Fairness**: a scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement (starvation).

Scheduling issues (IV)

- A **scheduling algorithm** is the algorithm which dictates how much CPU time is allocated to processes.
- Scheduling decisions can be taken according to:
  - Past behavior of process
  - Urgency
  - Priority
  - Origin (batch, interactive)
- It should:
  - Be fair and predictable;
  - Balance load;
  - Maximize: throughput, CPU utilization;
  - Minimize: overhead, turnaround time, waiting time and response time
- Failure to meet these goals can cause **starvation**.

Scheduling algorithms

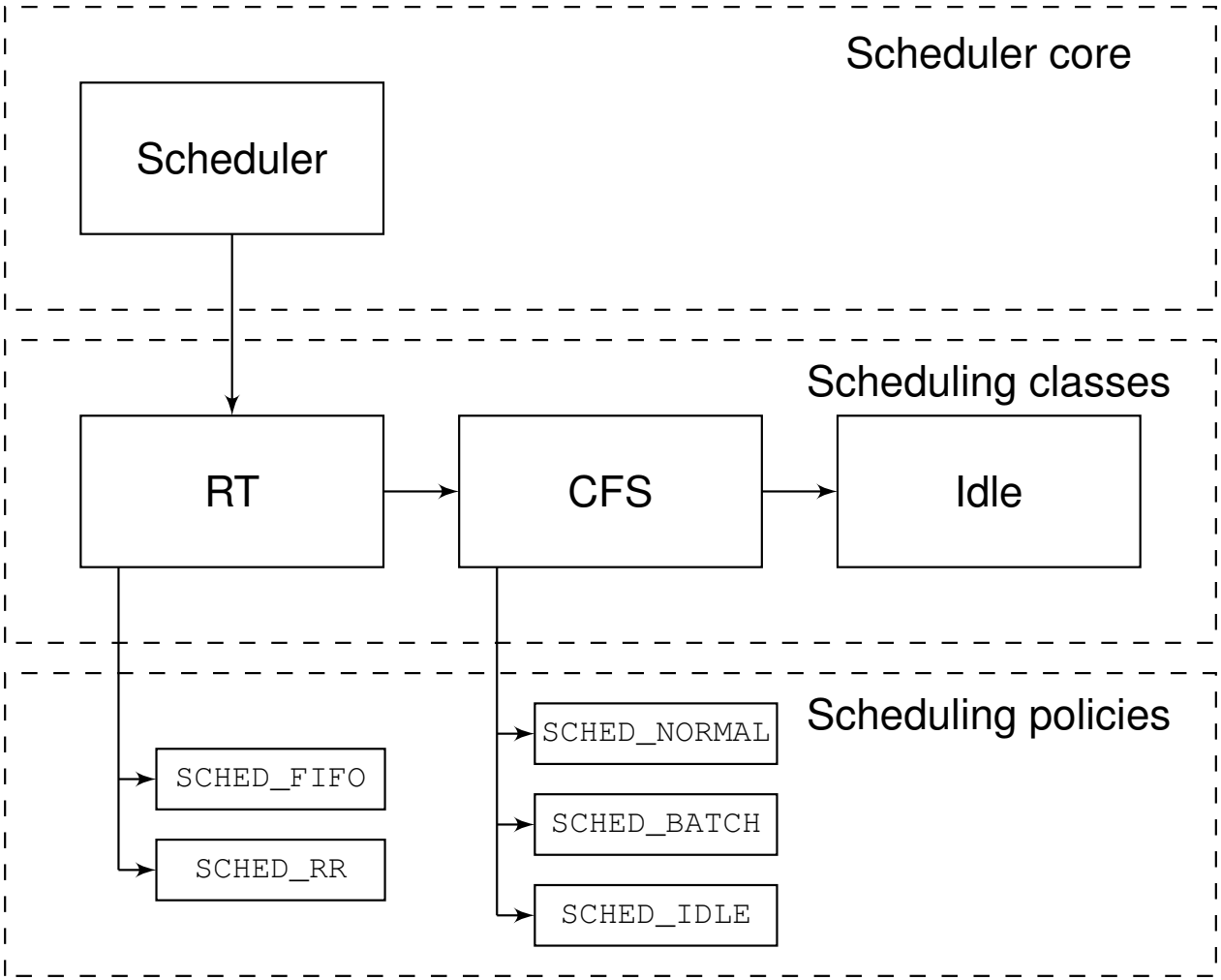
- First-Come, First-Served (FCFS);
- Shortest-Job First (SJF);
- Round-Robin (RR);
- Priority-based scheduling;
- Multi-level scheduling;
- Real-Time Scheduling.

Linux scheduling framework

Overview (I)

- The scheduler (or dispatcher) is the part of the kernel responsible, at run-time, for allocating processors to tasks for execution and the scheduling classes are responsible for selecting those tasks.
- The scheduling classes encapsulate scheduling policies.
  - These scheduling classes are hierarchically organized by priority and the scheduler inquires each scheduling class in a decreasing priority order for a ready task.
  - Linux has four main scheduling classes: Deadline (DL), Real-Time (RT), Completely Fair Scheduling (CFS) and Idle.
  - In this system, the scheduler first inquires the RT scheduling class for a ready task.
    - If DL scheduling class does not have any ready task, then it inquires the DL scheduling class.
    - If RT does not have any ready task, it proceeds by inquiring CFS and then it reaches the Idle scheduling class, used for the idle task.
    - *Every processor has an idle task in its ready-queue that is executed whenever there is no other ready task.*

Overview (II)



(It is missing the DL scheduling class).

Data structures(I)

- For every active processes in the system, Linux kernel create an instance of `struct task_struct` to manage them.
  - The kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on.
- A runqueue is a container for all processes in a `TASK_RUNNING` state.
  - Each CPU has its own runqueue; all runqueue structures are stored in the runqueues per-CPU variable of type `struct rq`.

Data structures(II)

- The Linux scheduler is modular, enabling different algorithms/policies to schedule different types of tasks.
  - An algorithm's implementation is wrapped in a so called scheduling class.
  - A scheduling class offers an interface to the main scheduler skeleton which it can use to handle tasks according to the implemented algorithm.
  - A scheduling class is an instance of `struct sched_class` data structure.

struct task_struct

- /include/linux/sched.h

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
/*
 * For reasons of header soup (see current_thread_info()), this
 * must be the first element of task_struct.
 */
struct thread_info thread_info;
#endif
/* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long state;
void *stack;
atomic_t usage;
/* Per task flags (PF_*), defined further below: */
unsigned int flags;
unsigned int ptrace;

...
};
```

struct rq

- It keeps track of all runnable tasks assigned to a particular CPU:
  - a `lock` to synchronize scheduling operations for this CPU
  - Pointers to the currently running (`curr`) and the idle (`idle`) tasks.
  - Actually, runqueue incorporates sub-runqueues per scheduling classes:`dl`, `cfs` and `rt`, for DL, CFS and RT scheduling classes, respectively.
- `/kernel/sched/sched.h`

```
struct rq {
/* runqueue lock: */
raw_spinlock_t lock;

...
unsigned int nr_running;
...

struct cfs_rq cfs;
struct rt_rq rt;
struct dl_rq dl;
...
```

struct sched_class (I)

- /kernel/sched/sched.h

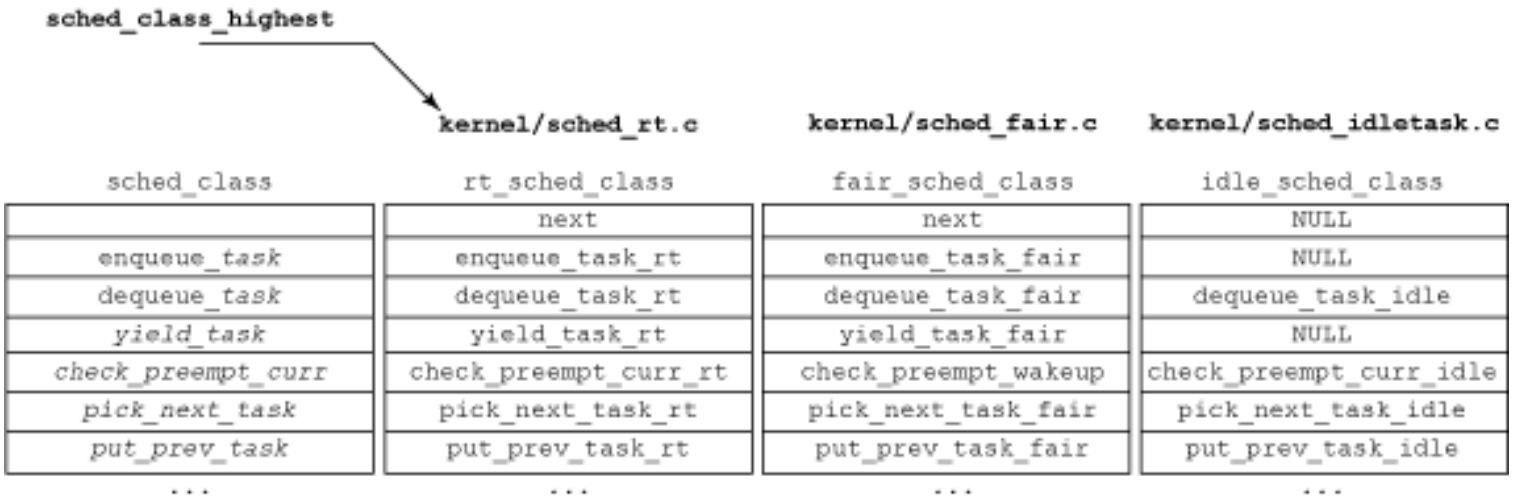
```
struct sched_class {
const struct sched_class *next;

void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);

void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);

struct task_struct * (*pick_next_task) (struct rq *rq,
struct task_struct *prev,
struct rq_flags *rf);

...
void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
...
};
```

`struct sched_class` (II)

- `next`: It is a pointer to `struct sched_class` that is used to organize the scheduler modules by priority in a linked list and the scheduler core, starting by the highest priority scheduler module, will look for a runnable task of each module in a decreasing order priority.
- `enqueue_task`: Called when a task enters a runnable state.
- `dequeue_task`: Called when a task is no longer runnable.
- `check_preempt_curr`: This function checks if a task that entered the runnable state should preempt the currently running task.
- `pick_next_task`: This function chooses the most appropriate task eligible to run next.
- `task_tick`: This function is mostly called from time tick functions;

struct sched_class (III)

- Except for the first one, all members of this struct are function pointers which are used by the scheduler core to call the corresponding policy implementation hook.
- All existing scheduling classes in the kernel are in a list which is ordered by the priority of the scheduling class.
- Idle is special scheduling classes.Idle is used to schedule the per-cpu idle task (also called `swapper` task) which is run if no other task is runnable.



The Scheduler Entry Point

- The main entry point into the process scheduler is the function `__schedule`.
- This is the function that the rest of the kernel uses to invoke the process scheduler, deciding which process to run and then running it.
- Its main goal is to find the next task to be run.
- It has two `struct task_struct` pointers, `prev` and `next` that are set with the currently executing task (which will relinquish CPU) and the next task to be executed (which will be assigned to CPU), respectively.

__schedule function

```
static void __sched notrace __schedule(bool preempt){
   struct task_struct *prev, *next;
   unsigned long *switch_count;
   struct rq_flags rf;
   struct rq *rq;
   int cpu;
   cpu = smp_processor_id();
   rq = cpu_rq(cpu);
   prev = rq->curr;
   local_irq_disable();
   rq_lock(rq, &rf);
    if (!preempt && prev->state) {
      if (unlikely(signal_pending_state(prev->state, prev))) {
         prev->state = TASK_RUNNING;
      } else {
         deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);
      }
   }
   next = pick_next_task(rq, prev, &rf);
   if (likely(prev != next)) {
      rq->curr = next;
      rq = context_switch(rq, prev, next, &rf);
   } else {
      ...
   }
}
```

`__schedule` function: main points (I)

- Since the Linux kernel is pre-emptive, it can happen that a task executing code in kernel space is involuntarily pre-empted by a higher priority task.
  - The first thing, it does is disabling the interrupts by calling `local_irq_disable`.
  - Secondly, it locks the current CPU's runqueue and, at same time.
- Next, it examines the state of the currently executing task, `prev`.
  - If it is not runnable and has not been pre-empted in kernel mode, then it should be removed from the runqueue. To remove a task from the runqueue, `deactivate_task` is called which internally calls the `dequeue_task` hook of the task's scheduling class.
  - However, if it has nonblocked pending signals, its state is set to `TASK_RUNNING` and it is left in the runqueue. This means `prev` gets another chance to be selected for execution.

`schedule` function: main points (II)

- Next, it is time to pick the next task to be assigned to the CPU calling `pick_next_task` function.
- After that, it checks if `pick_next_task` found a new task or if it picked the same task again that was running before.
  - If the latter is the case, no task switch is performed and the current task just keeps running.
  - If a new task is found, which is the more likely case, the actual task switch is executed by calling `context_switch`. Internally, `context_switch` switches to the new task's memory map and swaps register state and stack.
- To finish up, the runqueue is unlocked and pre-emption is reenabled.

Calling the Scheduler

- `__schedule` function is invoked when:
  - Whenever a task is mark to be preempted.
  - At regular times, at timer tick expiration.
  - Currently running task goes to sleep or finishes.
  - Sleeping task wakes up or newly forked tasks.

Requiring preemption

- `resched_curr` function marks the currently executing task to be preempted.
- This sets the `TIF_NEED_RESCHED` flag in the task structure, and the scheduler core will initiate a rescheduling at the next opportune moment.
- Example:
  - An interrupt occured.
  - Interrupt handler is invoked to manage the interrupt request.
  - If the `resched_curr` is invoked in the interrupt handle function.
  - When it proceeds to the IRQ Exit path, it checks:
    - If `TIF_NEED_RESCHED` flag is set , it calls `__schedule` function.

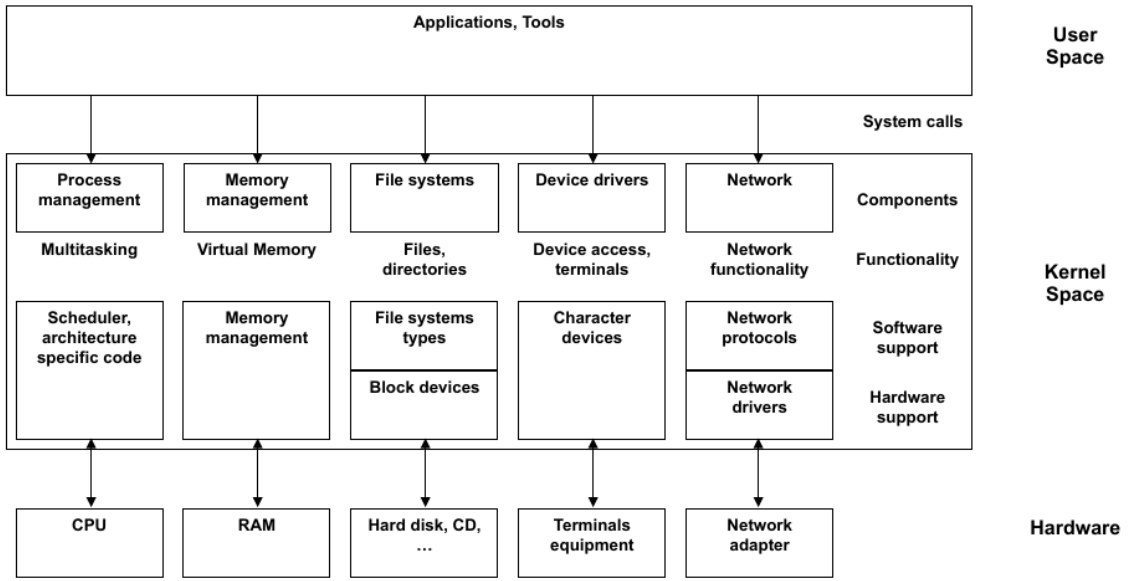
`scheduler_tick` function (I)

- The function `scheduler_tick` is called regularly by a timer interrupt, called `tick`.
- Its purpose is to update runqueue clock, CPU load and runtime counters of the currently running task.
- It calls the scheduling class hook `task_tick` of the currently executing task task update for the corresponding class.
  - at this point it can mark the current executing task to be preempted, by calling `resched_curr` function.
- load balancing is invoked if SMP is configured.

System calls

Introduction

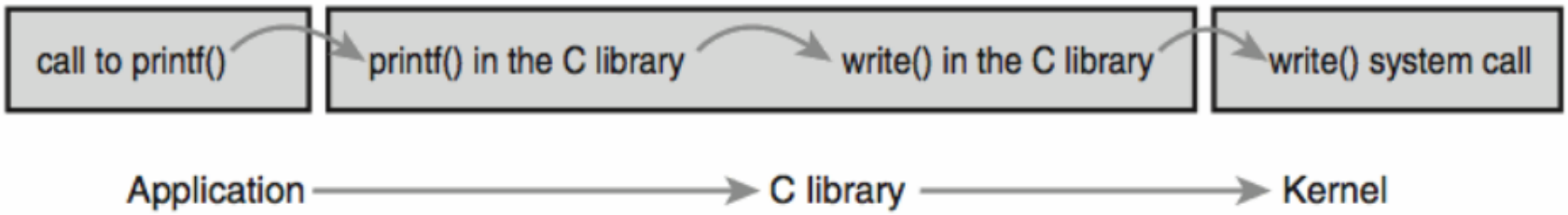
- It is not possible for user-space applications to execute kernel code directly
  - They cannot simply make a function call to a method existing in kernel-space because the kernel exists in a protected memory space
- If applications could directly read and write to the kernel's address space, system security and stability would be nonexistent



Communicating with the kernel (I)

- Typically, applications are programmed against an Application Programming Interface (API) implemented in user-space
  - An API defines a set of programming interfaces used by applications
- The C library implements the main API on Unix systems
  - Including the standard C library, the system call interface, and the majority of the POSIX API

```
#include <stdio.h>
int main(int argc, char *argv[]){
printf("LKD is cool\n");
return 0;
}
```



Communicating with the kernel (II)

- From the application programmer's point of view, system calls are usually irrelevant
  - All the programmer is concerned with is the API
- Libraries, in turn, rely on a system call interface to instruct the kernel to carry out tasks on the application's behalf
  - These interfaces act as the messengers between applications and the kernel

Tracing system calls

- The `strace` command line tool logs all system calls issued by an application and makes this information available to programmers
  - `> gcc test.c -o test`
  - `> strace ./test`

```c
#include <stdio.h>
int main(int argc, char *argv[]){
printf("LKD is cool\n");
return 0;
}
```

```
execve("./test", ["./test"], [/* 73 vars */]) = 0
...
write(1, "LKD is cool\n", 12LKD is cool
) = 12
exit_group(0) = ?
+++ exited with 0 +++
```

System call identifier

```c
#include <unistd.h>
int main(int argc, char *argv[]){
   syscall(1,1,"LKD is cool\n", 12);
   return 0;
}
```

- System call are identified by a number.
  - `> gcc test1.c -o test1`
  - `> ./test1`

How to add a new system call

Steps

1. Add a new entry to the system call table. This is located at `arch/x86/syscalls/syscall_64.tbl`.
2. Provide a function prototype in the `include/linux/syscalls.h` file.
3. Implementation of the system call function
4. Include the system call function in the Linux kernel compilation process.

System call table (I)

- Each system call is assigned a number
  - This is a unique number that is used to reference a specific system call
- The kernel keeps a list of all registered system calls in the system call table
  - This table is architecture-dependent.
    - On x86 it is defined in `/arch/x86/entry/syscalls/syscall_64.tbl`

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read sys_read
1 common write sys_write
2 common open sys_open
3 common close sys_close
...
330 common pkey_alloc sys_pkey_alloc
```

System call table (II)

- The format is: `<number> <abi> <name> <entry point>`
  - `<number>`
    - All syscalls are identified by a unique number. In order to call a syscall, we tell the kernel to call the syscall by its number rather than by its name.
  - `<abi>`
    - The ABI, or Application Binary Interface, to use. Either `64`, `x32`, or `common` for both.
  - `<name>`
    - This is simply the name of the syscall.
  - `<entry point>`
    - The entry point is the name of the function to call in order to handle the syscall.
    - The naming convention for this function is the name of the syscall prefixed with `sys_`.
    - For example, the read syscall's entry point is `sys_read`.

System call function prototype (I)

- A function prototype is a function declaration that specifies the data types of its arguments in the parameter list as well its return.
- The function prototype for our entry function will look like the following:
  - `asmlinkage long <entry point>(<list of arguments>);`
    - The curious part of this line is the `asmlinkage`.
    - This is a macro that tells to the compiler that the function should expect all of its arguments to be on the stack rather than in registers.

System call function prototype (II)

- The function prototype of syscall's entry function must be included into `include/linux/syscalls.h` file.
- The function prototype for our entry function will look like the following:
  - `asmlinkage` macro specifies the method to handle the system call argument(s) on the kernel stack.

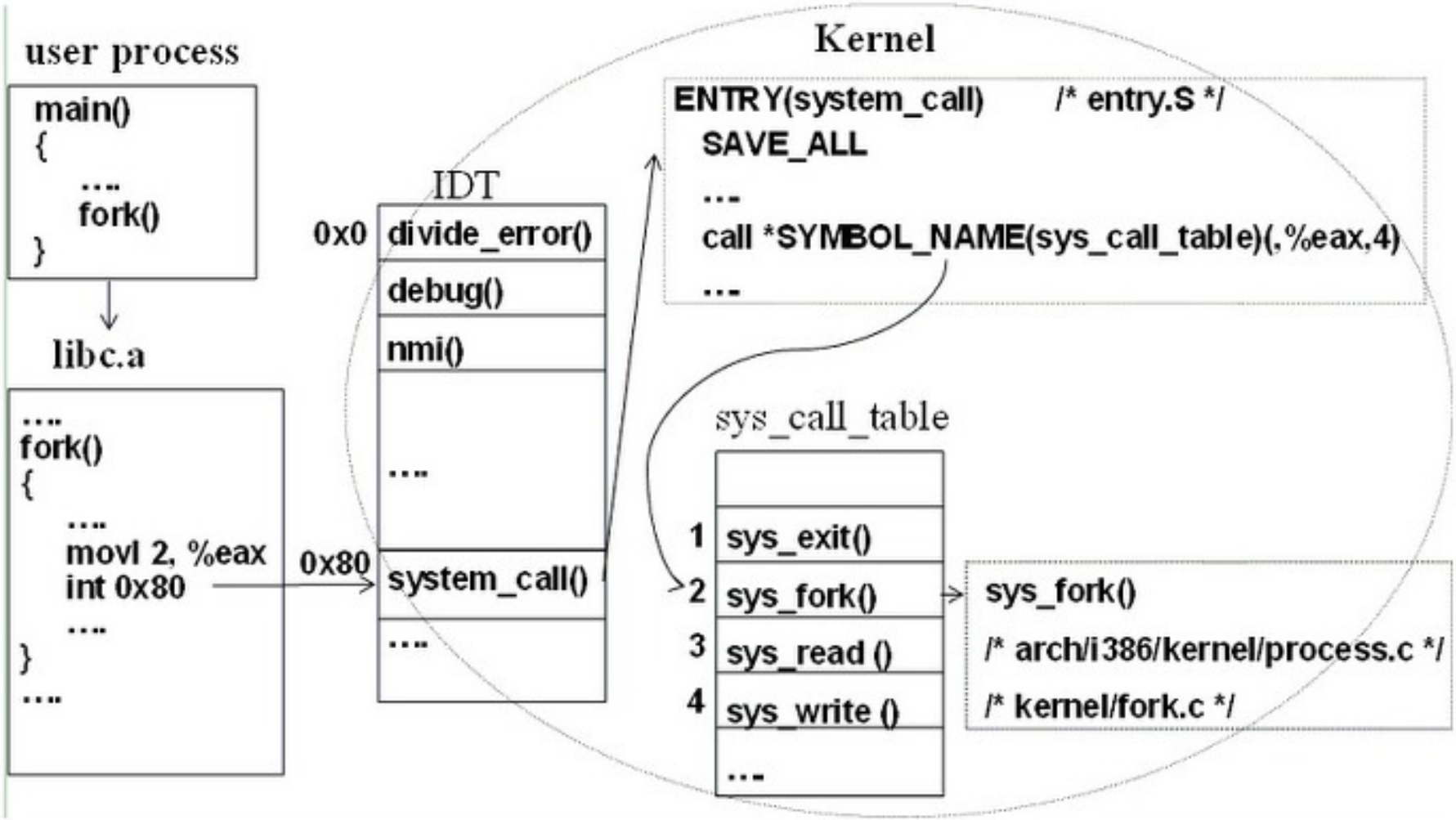
Linux naming conventions

- Defining a system call with `SYSCALL_DEFINE`*n*
  - `SYSCALL_DEFINE`*n* macros are the standard way for kernel code to define a system call, where the *n* suffix indicates the argument count. The definition of these macros (in `include/linux/syscalls.h`)
- `SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)`

System call handler (I)

- User-space applications must somehow signal to the kernel that they want to execute a system call and have the system switch to kernel mode
  - The mechanism to signal the kernel is a software interrupt
    - Raises an exception (interrupt), the system will switch to kernel mode and execute the interrupt handler, that, in this case, is actually the system call handler
    - On x86 processors it is used `int` assembly instruction, with interrupt number 128 (or 0x80):
    - On more modern processors of the IA-32 series (Pentium II and higher) two assembly language instructions (`sysenter` and `sysexit`) are used to enter and exit kernel mode quickly.
    - On x86_64 processors the `syscall` assembly instruction is used to enter into kernel.

System call handler (II)



Invoking system calls

Invoking system calls (I)

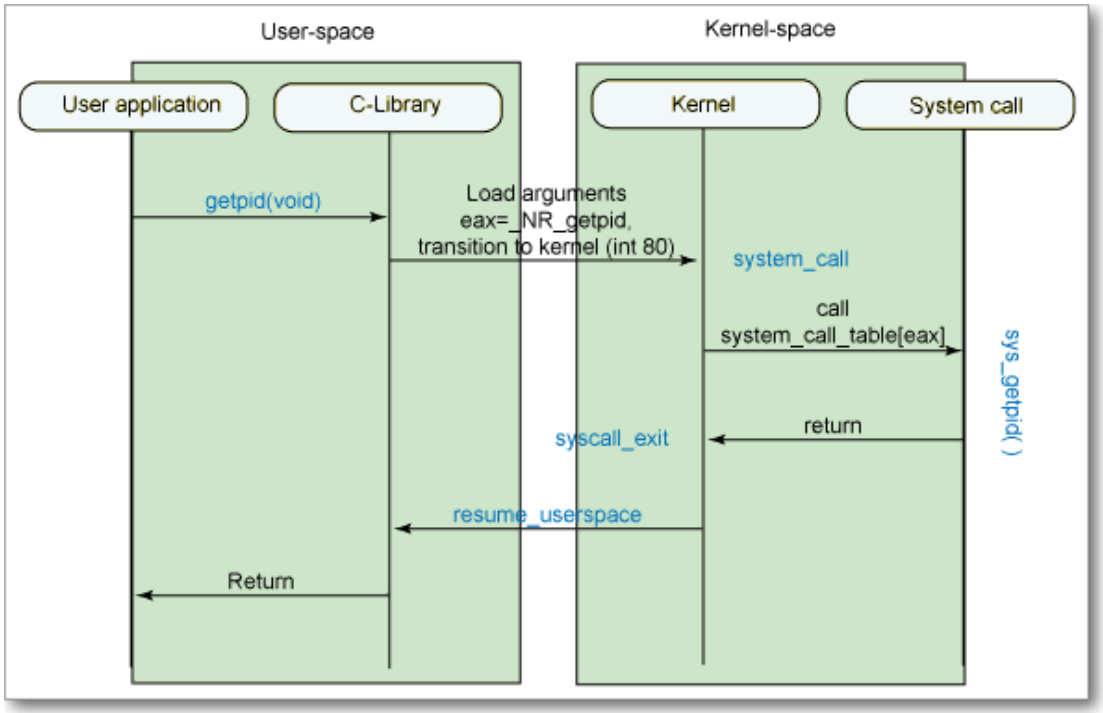
- System calls can be indirectly invoked with `syscall` function.
  - `long syscall(long number, ...)`
  - The return value is defined by the system call being invoked. In general, a 0 return value indicates success. A -1 return value indicates an error, and an error code is stored in `errno`.

Invoking system call (II)

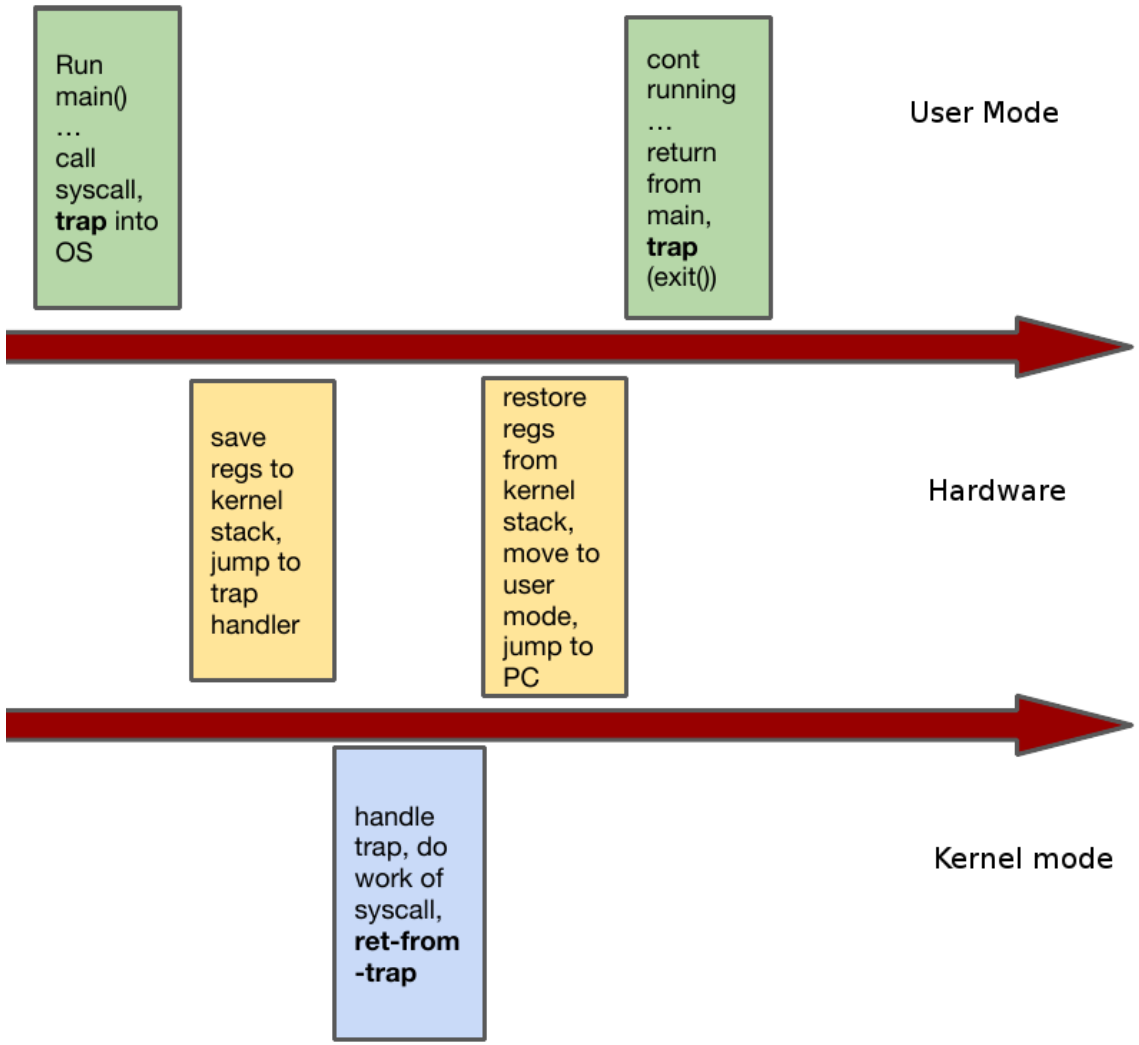
- Parameter passing on x86
  - On x86, the parameters are stored in CPU registers (`%eax`,`%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`).
    - For the system call number is used `%eax` .
    - The registers `%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi` contain, in order, the first five parameters.
    - For six or more parameters, a single register is used to hold a pointer to user-space where all the parameters are stored.
    - The return value is sent to user-space also via `%eax` register.
- System calls use kernel stack.

Invoking system call (III)

- Each system call must inform the user application whether its routine was executed and with which result
- From the perspective of the application, a normal variable is read using C programming features
- The return value is sent to user-space also via a CPU register
  - On x86, it is written into the `%eax` register



Invoking, executing and returning



Books and Useful links

Books

- *Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel, 3rd Edition*, Robert Love. Addison-Wesley Professional, 2010.
- *Professional Linux Kernel Architecture*, Wolfgang Mauerer. Wrox , 2008.
- *Linux Device Drivers, 3rd Edition*, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. O'Reilly, 2005.
- *Understanding the Linux Kernel, 3rd Edition*, Daniel P.Bovet, Marco Cesati, O'Reilly Media, 2005.

Links

- elixir.free-electrons.com/linux/v4.10/source
- www.kernel.org/doc/htmldocs/kernel-api/
- kernelnewbies.org/Documents
- lwn.net/Kernel/LDD3/