Kernel Build System, Process Management and System Calls

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa
`pbs@isep.ipp.pt`

Instituto Superior de Engenharia do Porto

P.PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

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

Outline

The Linux Kernel Build System

Introduction

- The Linux kernel has a **monolithic architecture**, which means that the whole kernel code runs in kernel space and shares the same address space.
- But, **Linux is not a pure monolithic kernel**
  - It can be extended at runtime using loadable kernel modules.
- However, to load a module
  - The kernel must contain all the kernel symbols used in the module.

Modules

There is the need to choose at kernel compile time most of the features that will be built in the kernel image and the ones that will allow you to load specific kernel modules once the kernel is executing.

Compiling Linux kernel (I)

1. Get the Linux kernel source code
2. Configure the Linux kernel features and modules
   - The Linux kernel source code comes with the default configuration.
     - The kernel configuration is kept in a file called `.config` in the top directory of the kernel source tree.
     - If you have just expanded the kernel source code, there will be no `.config` file, so it needs to be created.
     - It can be created from scratch, created by basing it on the default configuration.
   - You can adjust it to your needs using a kernel configuration tool (`make menuconfig`).
3. Compile the kernel
   - Running `make` will cause the kernel build system to use the configuration file (`.config`).
     - If the kernel build finished without any errors, you have successfully created a kernel image.

Compiling Linux kernel (II)

4. Install modules
   - `make modules_install` command will install all the modules that you have built and place them in the proper location in the filesystem for the new kernel to properly find.
     - Modules are placed in the `/lib/modules/KERNEL_VERSION` directory, where `KERNEL_VERSION` is the kernel version of the new kernel you have just built.
5. Install Linux kernel
   - `make install` command will:
     - The kernel build system will verify that the kernel has been successfully built properly.
     - The build system will install the static kernel portion into the `/boot` directory and name this executable file based on the kernel version of the built kernel.
     - Any needed initial ramdisk images will be automatically created, using the modules that have just been installed during the `modules_install` phase.

Compiling Linux kernel (III)

6. Update the Bootloader
   - `update grub2` command will:
     - The bootloader program will be properly notified that a new kernel is present, and it will be added to the appropriate menu so the user can select it the next the machine is booted.

Components

- **Configuration symbols**:
  - Compilation options that can be used to **compile code conditionally** in source files and to decide which objects to include in a kernel image or its modules.
- **Kconfig files**:
  - Define each **config symbol** and its attributes, such as its type, description and dependencies.
  - Programs that generate an option menu tree (for example, `make menuconfig`) read the menu entries from these files.
- **.config file**:
  - It stores each config symbol's selected value.
- **Makefiles**:
  - Normal GNU makefiles that describe the relationship between source files and the commands needed to generate each make target, such as kernel images and modules.

Configuration Symbols (I)

- Configuration symbols are the **ones used to decide which features will be included in the final Linux kernel image**.

```
config SMP
bool "Symmetric multi-processing support"
---help---
...
config X86_MCE_INJECT
depends on X86_MCE
tristate "Machine check injector support"
---help---
...
```

  - In the source code as well as in the `Makefile` they will be referred as `CONFIG_SMP` and `CONFIG_X86_MCE_INJECT`.
  - The **`CONFIG_`** prefix is assumed but is not written.
- Two kinds of symbols are used for conditional compilation:
  - **Boolean symbols**
    - They can take one of two values: **true (y)** or **false (n)**.
  - **Tristate symbols**
    - They can take three different values: **yes (y)**, **no (n)** or **module (m)**.

Configuration Symbols (II)

- Default configuration

```
config PM_DEVICE
bool
default n
...
```

- Dependencies and help

```
config PM
bool "Power Management support"
...
config PM_DEBUG
bool "Power Management Debug Support"
depends on PM
...
```

- Menus

```
menu "XPTO device support"
config XPTODEVICES
...
endmenu
```

`Kconfig` Files (I)

- Configuration symbols **are defined in files known as `Kconfig` files**.
- Each `Kconfig` file can describe an arbitrary number of symbols and can also include other `Kconfig` files.
  - Compilation targets that construct configuration menus of kernel compile options, such as `make menuconfig`, read these files to build the tree-like structure.
  - The contents of `Kconfig` are parsed by the configuration subsystem, which presents configuration choices to the user, and contains help text associated with a given configuration parameter.
- The configuration utility (`make menuconfig`) reads the `Kconfig` files starting from the `arch` subdirectory's `Kconfig` file.
- Typically, there is one `Kconfig` file per directory.

`Kconfig` Files (II)

```
config HAVE_ATOMIC_IOMAP
        def_bool y
        depends on X86_32

config X86_DEV_DMA_OPS
        bool
        depends on X86_64 || STA2X11

config X86_DMA_REMAP
        bool
        depends on STA2X11
```

Kconfig syntax for defining config Macros and their dependencies

```
source "net/Kconfig"

source "drivers/Kconfig"

source "drivers/firmware/Kconfig"

source "fs/Kconfig"

source "arch/x86/Kconfig.debug"

source "security/Kconfig"

source "crypto/Kconfig"
```

This Kconfig file includes other Kconfig files, which are defined in others directories

`.config` File

- The output of this configuration exercise is written to a configuration file named `.config`, located in the **top-level Linux source directory** that drives the kernel build.

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 5.3.8-moker Kernel Configuration
#

#
# Compiler: gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
#
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=70400
CONFIG_CLANG_VERSION=0
CONFIG_CC_CAN_LINK=y
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_CC_HAS_WARN_MAYBE_UNINITIALIZED=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_EXTABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y
```

Makefile (I)

- The Makefile **uses information from the `.config` file** to construct various file lists used by `kbuild` tool to build any built-in or modular targets.
- It is responsible for building two major products:
  - `vmlinux` (the resident kernel image)
  - `modules` (any module files).
- It builds these goals by **recursively descending into the subdirectories** of the kernel source tree.
  - Each subdirectory **has a `Makefile`** which carries out the commands passed down from above.
- The start point is an arch `Makefile` with the name `arch/$(ARCH)/Makefile`.
  - The arch `Makefile` supplies architecture-specific information to the top `Makefile`.

Makefile (II)

```
#
# Makefile for the linux kernel.
#

obj-y     = fork.o exec_domain.o panic.o \
            cpu.o exit.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
            extable.o params.o \
            kthread.o sys_ni.o nsproxy.o \
            notifier.o ksysfs.o cred.o reboot.o \
            async.o range.o smpboot.o ucount.o

obj-$(CONFIG_MULTIUSER) += groups.o

ifdef CONFIG_FUNCTION_TRACER
# Do not trace internal ftrace files
CFLAGS_REMOVE_irq_work.o = $(CC_FLAGS_FTRACE)
endif

# Prevents flicker of uninteresting __do_softirq()/__local_bh_disable_ip()
# in coverage traces.
KCOV_INSTRUMENT_softirq.o := n
# These are called from save_stack_trace() on slub debug path,
# and produce insane amounts of uninteresting coverage.
KCOV_INSTRUMENT_module.o := n
KCOV_INSTRUMENT_extable.o := n
# Don't self-instrument.
KCOV_INSTRUMENT_kcov.o := n
KASAN_SANITIZE_kcov.o := n

# cond_syscall is currently not LTO compatible
CFLAGS_sys_ni.o = $(DISABLE_LTO)

obj-y += sched/
obj-y += locking/
obj-y += power/
obj-y += printk/
obj-y += irq/
obj-y += rcu/
obj-y += livepatch/
```

All these files (with".c" extension)will be unconditionally included in the compilation process

All these directories will be unconditionally included in the compilation process. All these directory have to have a file called "Makefile"

Makefile (III)

```
obj-$(CONFIG_UID16) += uid16.o
obj-$(CONFIG_MODULES) += module.o
obj-$(CONFIG_MODULE_SIG) += module_signing.o
obj-$(CONFIG_KALLSYMS) += kallsyms.o
obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
obj-$(CONFIG_CRASH_CORE) += crash_core.o
obj-$(CONFIG_KEXEC_CORE) += kexec_core.o
obj-$(CONFIG_KEXEC) += kexec.o
obj-$(CONFIG_KEXEC_FILE) += kexec_file.o
obj-$(CONFIG_BACKTRACE_SELF_TEST) += backtracetest.o
obj-$(CONFIG_COMPAT) += compat.o
obj-$(CONFIG_CGROUPS) += cgroup/
obj-$(CONFIG_UTS_NS) += utsname.o
obj-$(CONFIG_USER_NS) += user_namespace.o
obj-$(CONFIG_PID_NS) += pid_namespace.o
obj-$(CONFIG_IKCONFIG) += configs.o
```

This file will be included in the compilation process only if the CONFIG_KALLSYMS option is set

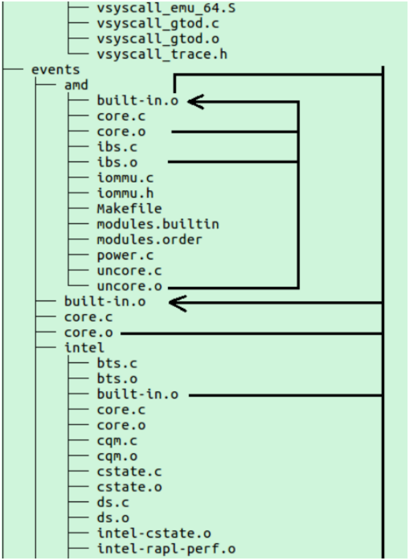
This directory will be included in the compilation process only if CONFIG_CGROUPS option is set. cgroup directory has to have a Makefile

Makefile (IV)

- Compile a **built-in object**: `obj-y`
  - `obj-y += foo.o`: This tells `kbuild` that there is one object in that directory, named `foo.o`.
  - `foo.o` will **be built from `foo.c` or `foo.S`**.
  - Then, it **is merged into one `built-in.o` file**.
- Compile a **loadable module**: `obj-m`
  - `obj-m += foo.o`: This tells `kbuild` that there is one object in that directory, named `foo.o`.
  - This specifies object files which are built as loadable kernel modules.
- `obj-$(CONFIG_FOO) += foo.o`: depends on the `CONFIG_FOO` value.
  - `CONFIG_FOO=y`: built-in kernel code.
  - `CONFIG_FOO=m`: compiled as a module.
  - `# CONFIG_FOO is not set`: it is not compiled.

Makefile (V)

- All object files **are combined into a `built-in.o` object file per directory**.
- All `built-in.o` files are **included into the `built-in.o` file of the parent directory**.



Makefile (VI)

- All `built-in.o` files **are then linked and the resulting file `vmlinux` is located at the root of the source code directory**.

```
                                            kernel/built-in.o    certs/built-in.o
                                            mm/built-in.o        fs/built-in.o
                                            ipc/buit-in.o        security/built-in.o
                                            crypto/built-in.o    block/built-in.o
head_64.o                                   lib/built-in.o       lib/lib.a
head64.o                                    usr/built-in.o       drivers/built-in.o
ebda.o                                      sound/built-in.o     firmware/built-in.o
platform-quirks.o     init/built-in.o       net/built-in.o       virt/built-in.o

    |                       |                    |                   |

$(head-y)               $(init-y)                    $(core-y)  $(drivers-y)
                                                     $(net-y)   $(libs-y)
                                                     $(virt-y)
    |                       |                           |          |
    |                       |                           |          |
    \                       /                           \          /
     \                     /                             \        /
      KBUILD_VMLINUX_INIT                          KBUILD_VMLINUX_MAIN
             \                                             /
              \___________________________________________/
                                  |
                               vmlinux
```

Code: conditional compilation

```
#ifdef CONFIG_DEBUG_PAGEALLOC
  /*
   * Need to access the cpu field knowing that
   * DEBUG_PAGEALLOC could have unmapped it if
   * the mutex owner just released it and exited.
   */
   if (probe_kernel_address(&owner->cpu, cpu))
     return 0;

#else
  cpu = owner->cpu;
#endif
  ...
#ifdef CONFIG_RT_MUTEXES
  ...
   void rt_mutex_setprio(struct task_struct *p, int prio{
     ...
   }
#endif
```

Code: conditional inclusion

- It is possible to control **pre-processing itself with conditional statements that are evaluated during pre-processing**.
- This provides a way **to include code selectively, depending on the value of conditions evaluated during compilation**.
  - For example, to make sure that the contents of a file `hdr.h` are included only once, the contents of the file are surrounded with a conditional like this:

```
#ifndef HDR_H
#define HDR_H

/* contents of hdr.h go here */

#endif
```

    - The first inclusion of `hdr.h` defines the name `HDR_H`.
    - Subsequent inclusions will find the name defined and skip down to the `#endif`.

Process Management

Process Representation

- Linux is a **multi-user and multitasking operating system**, and thus has to manage multiple processes from multiple users
- **A process is an instance of execution that runs on a processor**.
- Processes are more than just the executing program code.
  - They also include a **set of resources** such as open files and pending signals, internal kernel data, processor state, a memory address space with one or more memory mappings, one or more threads of execution, and a data section containing global variables.
  - The data structures used to represent individual processes **have connections with nearly every subsystem of the kernel**

Process identification

- Linux allow users to identify processes by means of a number called the **Process ID** (or PID)
- PIDs are numbered sequentially
- The PID of a newly created process is normally the PID of the previously created process increased by one
- There is an upper limit on the PID values
- When the kernel reaches such limit, it must start recycling the lower, unused PIDs
- Each process has its own PID and they also have a **TGID (thread group ID)**.

Thread identification

- When a new **process is created**, it appears as a **thread where both the PID and TGID are the same number**.
- When a **thread/process starts another thread**, that started thread gets its own PID (so the scheduler can schedule it independently) but it **inherits the TGID from the original thread**.
- When a **thread/process starts another process**, that started process **gets its own PID and TGID**.

```
                       USER VIEW
 <-- PID 43 --> <----------------- PID 42 ----------------->
                     +---------+
                     | process |
                    _| pid=42  |_
                  _/ | tgid=42 | \_ (new thread) _
       _ (fork) _/   +---------+                  \
      /                                        +---------+
+---------+                                    | process |
| process |                                    | pid=44  |
| pid=43  |                                    | tgid=42 |
| tgid=43 |                                    +---------+
+---------+
 <-- PID 43 --> <--------- PID 42 --------> <--- PID 44 --->
```

Process vs Thread identification

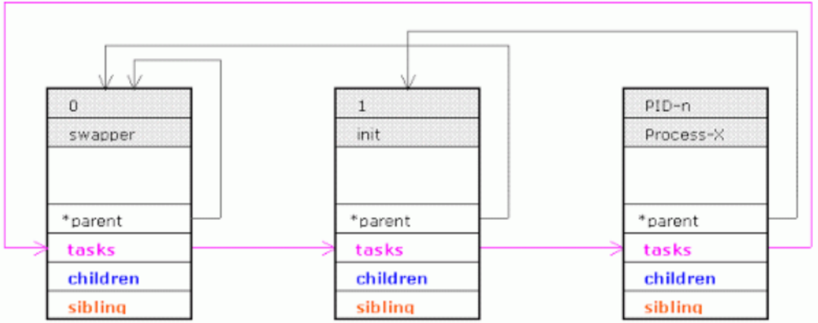
- From the schedule point of view, the Linux kernel **does not differentiate threads and processes**.
- Both are managed by `struct task_struct` data structure.
  - Defined in `include/linux/sched.h`

struct task_struct data structure

```
struct task_struct {
  ...
  /* -1 unrunnable, 0 runnable, >0 stopped: */
  volatile long   state;
  ...
  int    prio;
  ...
  const struct sched_class *sched_class;
  ...
  unsigned int   policy;
  ...
  struct list_head  tasks;
  ...
  int    exit_state;
  ...
  pid_t    pid;
  pid_t    tgid;
  ...
  struct task_struct  *real_parent;
  struct task_struct  *parent;
  ...
  char     comm[TASK_COMM_LEN];
  ...
};
```

Process hierarchy

- All processes are **descendants of the `init` process**, whose Process ID (PID) is one.
  - The kernel starts `init` in the last step of the boot process
- Every process has exactly **one parent**
  - Likewise, every process has zero or more **children**
  - Processes that are all direct children of the same parent are called **siblings**

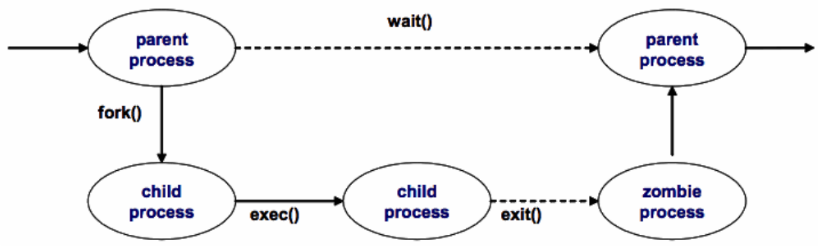


Process lifecycle (I)

1. `fork` function creates a child process that is a copy of the current task.
   - It differs from the parent only in its PID (which is unique) and certain resources and statistics, such as pending signals, which are not inherited.
2. `exec` function loads a new executable into the address space and begins executing it.
3. When a process terminates, by invoking `exit` function, the kernel releases the resources owned by the process and notifies the child's parent of its demise.
4. After process completes, the process descriptor for the terminated process still exists, but the process is a `zombie` and is unable to run.
5. After the parent has obtained information on its terminated child the child's `task_struct` is deallocated.

Process lifecycle (II)

- The standard behavior of the wait function is to suspend execution of the calling task until one of its children exits, at which time the function returns with the PID of the exited child.

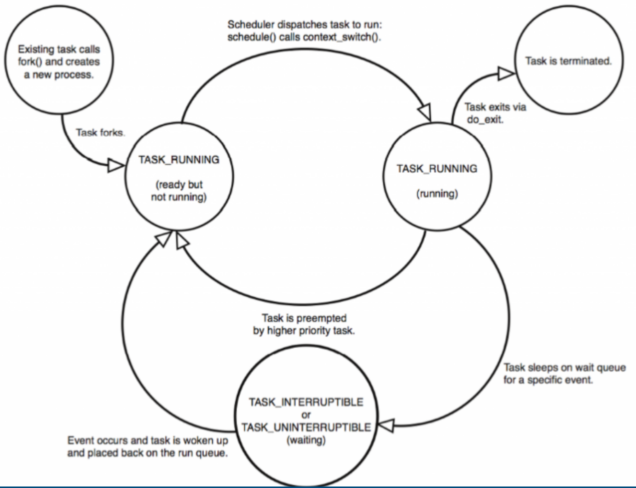


Parentless task/process

- If a **parent exits before its children**, some mechanism must exist to re-parent any child tasks to a new process
  - Otherwise, parentless terminated processes would forever remain zombies
- The solution is to **re-parent a task's children** on `exit` to either another process in the current thread group or, if that fails, the `init` process
- `init` routinely calls `wait` on its children, cleaning up any zombies assigned to it

Task state

- Every task **has its own state** that shows what is currently happening in the task.



`state` field (I)

- Range of values for `volatile long state` field of the `struct task_struct` data structure:
  - -1: **unrunnable**;
  - 0: **runnable**;
  - >0: **stopped**.
- Defined in `/include/linux/sched.h`

```
/* Used in tsk->state: */
#define TASK_RUNNING   0x0000
#define TASK_INTERRUPTIBLE  0x0001
#define TASK_UNINTERRUPTIBLE  0x0002
#define __TASK_STOPPED   0x0004
#define __TASK_TRACED   0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD   0x0010
#define EXIT_ZOMBIE   0x0020
#define EXIT_TRACE   (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED   0x0040
#define TASK_DEAD   0x0080
...
```

`state` field (II)

- `TASK_RUNNING`
  - The task is either executing on a CPU or waiting to be executed. This is the only possible state for a task executing in user-space.
- `TASK_INTERRUPTIBLE`
  - The task is blocked until some condition becomes true. A typical example of a `TASK_INTERRUPTIBLE` process is a process waiting for keyboard interrupt.
- `TASK_UNINTERRUPTIBLE`
  - Identical to `TASK_INTERRUPTIBLE` except that the task does not wake up and become runnable if it receives a signal.
- `__TASK_STOPPED`
  - Process execution has stopped; the task is not running nor is it eligible to run. This occurs if the task receives some (such as `SIGSTOP` or other) signal or if it receives any signal while it is being debugged.
- `__TASK_TRACED`
  - The process is being traced by another process, such as a debugger, via `ptrace`.

`exit_state` field

- `EXIT_ZOMBIE`
  - A process always switches briefly to the zombie state between termination and removal of its data from the process table.
- `EXIT_DEAD`
  - It is the state after an appropriate wait system call has been issued and before the task is completely removed from the system.

`policy` field

- The `policy` field **holds the scheduling policy** applied to the process.
- Range of values for `int policy` field of the `struct task_struct` data structure.
- Defined in `/include/uapi/linux/sched.h`

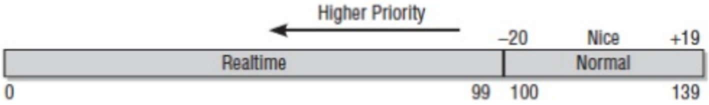
```
/*
* Scheduling policies
*/
#define SCHED_NORMAL  0
#define SCHED_FIFO  1
#define SCHED_RR  2
#define SCHED_BATCH  3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE  5
#define SCHED_DEADLINE  6

/* Can be ORed in to make sure the process is reverted back to SCHED_NORMAL on
      fork */
#define SCHED_RESET_ON_FORK 0x40000000
```

Scheduling policies

- Handled by the Completely Fair Scheduler (CFS).
  - `SCHED_NORMAL`: is used for normal processes. `SCHED_BATCH` and `SCHED_IDLE` can be used for less important tasks.
    - `SCHED_BATCH` is for CPU-intensive batch processes that are not interactive. Tasks of this type are disfavored in scheduling decisions.
    - `SCHED_IDLE` tasks will also be of low importance in the scheduling decisions, but this time because their relative weight is always minimal. Tasks running as `SCHED_IDLE` would only run when the processor would otherwise be idle
    - Note that `SCHED_IDLE` is, despite its name, not responsible to schedule the `idle task`.
- Handled by the RT.
  - `SCHED_RR` implements a round robin method.
  - `SCHED_FIFO` uses a first in, first out mechanism.
- Handled by the Deadline
  - `SCHED_DEADLINE` it is an implementation of the Earliest Deadline First (EDF) + Constant Bandwidth Server (CBS) scheduling algorithms.

Kernel Representation of Priorities

- The static priority of a process can be set in userspace by means of the `nice` command, which internally invokes the nice system call.
    - The `nice` value of a process is between -20 and $+19$ (inclusive).
    - Lower values mean higher priorities.
- The kernel uses a simpler scale ranging from 0 to 139 inclusive to represent priorities internally.
    - Lower values mean higher priorities. The range from 0 to 99 is reserved for real-time processes.
    - The nice values $[-20, +19]$ are mapped to the range from 100 to 139.



__schedule function (I)

- Defined in `kernel/sched/core.c`

```
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;
  ...
  next = pick_next_task(rq, prev, &rf);
  ...
  if (likely(prev != next)) {
    rq->nr_switches++;
    rq->curr = next;
    ...
    rq = context_switch(rq, prev, next, &rf);
  } else {
    ...
  }
  ...
}
```

`__schedule` function (II)

- Scheduler core function.
- The **main means of driving** the scheduler and thus **entering this function are**:
  - Explicit blocking: mutex, semaphore, waitqueue, etc.
  - The executing task is marked to be preempted.
    - To drive preemption between tasks, the scheduler marks the executing task to be preempted in timer interrupt handler `scheduler_tick`.
  - Wakeups do not really cause entry into schedule.
    - They add a task to the run-queue and that's it.
  - At task execution termination (invoking `exit` function).

`scheduler_tick` function

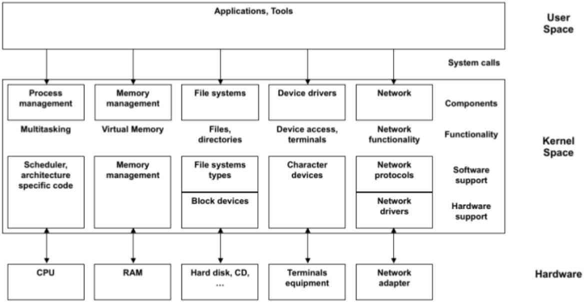
- This function **gets called by the timer code**, with `HZ` frequency.
  - It is called with interrupts disabled.
- Defined in `kernel/sched/core.c`

```
void scheduler_tick(void){
  int cpu = smp_processor_id();
  struct rq *rq = cpu_rq(cpu);
  struct task_struct *curr = rq->curr;
  struct rq_flags rf;

  sched_clock_tick();

  rq_lock(rq, &rf);

  update_rq_clock(rq);
  curr->sched_class->task_tick(rq, curr, 0);
  calc_global_load_tick(rq);
  psi_task_tick(rq);

  rq_unlock(rq, &rf);

  perf_event_task_tick();
  ...
}
```

System Calls

Introduction

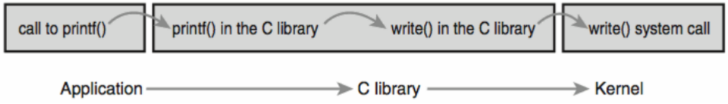
- It is **not possible** for user-space applications to execute kernel code directly
  - They **cannot simply make a function call to a method existing in kernel-space** because the kernel exists in a protected memory space
- If applications could directly read and write to the kernel's address space, **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("MOKER 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`

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

  - `strace ./test`

```
execve("./test", ["./test"], [/* 113 vars */]) = 0
...
write(1, "MOKER is cool\n", 14MOKER is cool
) = 14
exit_group(0) = ?
+++ exited with 0 +++
```

System call identifier

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

```
#include <unistd.h>
int main(int argc, char *argv[]){
 syscall(1,1,"MOKER is cool\n", 14);
 return 0;
}
```

  - `./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 __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0 common read    __x64_sys_read
1 common write    __x64_sys_write
2 common open    __x64_sys_open
 ...
```

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.

System call function prototype

- 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:
    - `long <entry point>(<list of arguments>);`
- The function prototype of syscall's entry function must be included into `include/linux/syscalls.h` file.

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`)

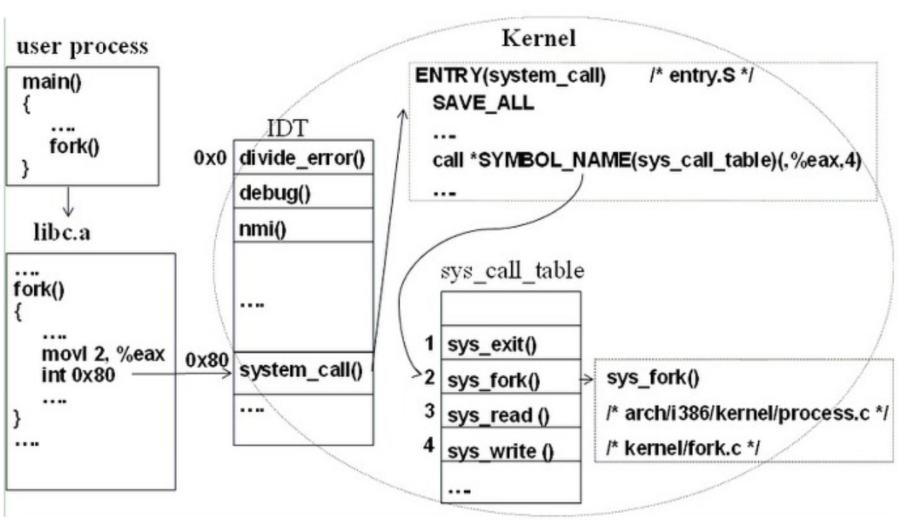
```
#ifndef SYSCALL_DEFINE0
#define SYSCALL_DEFINE0(sname)      \
 SYSCALL_METADATA(_##sname, 0);     \
 asmlinkage long sys_##sname(void);    \
 ALLOW_ERROR_INJECTION(sys_##sname, ERRNO);   \
 asmlinkage long sys_##sname(void)
#endif /* SYSCALL_DEFINE0 */
```

- `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 function**.
    - On x86 processors it is used an 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

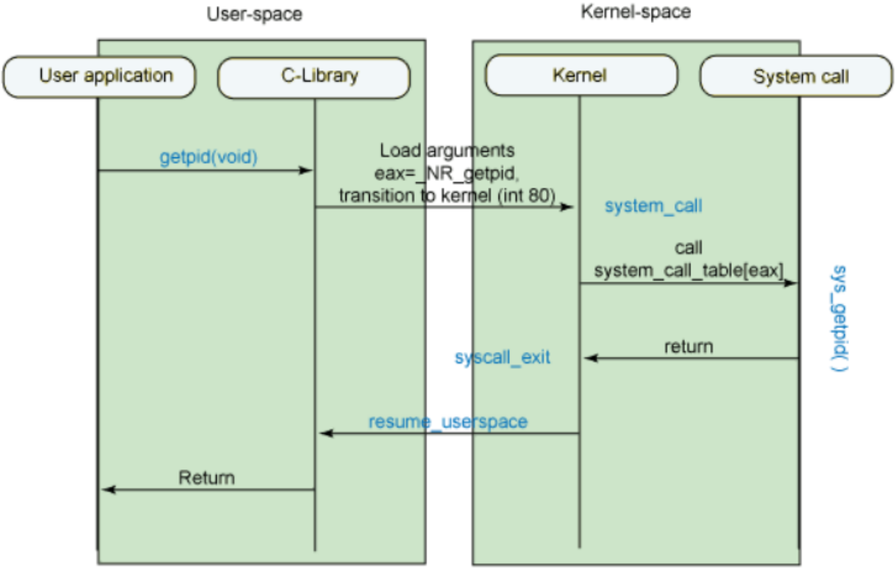
`syscall`

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

Parameter passing on x86

- For the system call number is used `%eax`
- Without `asmlinkage`
  - 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.
- With `asmlinkage`
  - Parameters are passed via kernel stack
- The return value is sent to user-space also via `%eax` register.

Invoking, executing and returning (I)



Invoking, executing and returning (II)

- System calls use kernel stack.