Linux Kernel Development (LKD)

Session 2

Kernel Build System and Process Management

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

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.

The Linux Kernel Build System

- Four main components:
  - **config 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**: 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** and **tristate**.
  - Boolean symbols can take one of two values: true or false.
  - Tristate symbols can take three different values: yes (y), no (n) or module (m).

Configuration Symbols (II)

- Dependencies and help

```
config PM
bool "Power Management support"
  ...
  ---help---
  ...

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

- 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 4.12.4-cister Kernel Configuration
#
CONFIG_64BIT=y
CONFIG_X86_64=y
CONFIG_X86=y
CONFIG_INSTRUCTION_DECODER=y
CONFIG_OUTPUT_FORMAT="elf64-x86-64"
CONFIG_ARCH_DEFCONFIG="arch/x86/configs/x86_64_defconfig"
CONFIG_LOCKDEP_SUPPORT=y
CONFIG_STACKTRACE_SUPPORT=y
CONFIG_MMU=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.
  - 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`.
    - `foo.o` will be built from `foo.c` or `foo.S`.
    - 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 (II)

- The top Makefile reads the `.config` file, which it is the output of the kernel configuration process.
  - 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.
  - It includes an arch Makefile with the name `arch/$(ARCH)/Makefile`.
    - The arch Makefile supplies architecture-specific information to the top Makefile.
    - Each subdirectory has a Makefile which carries out the commands passed down from above.

Makefile (III)

```
#
# 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 (IV)

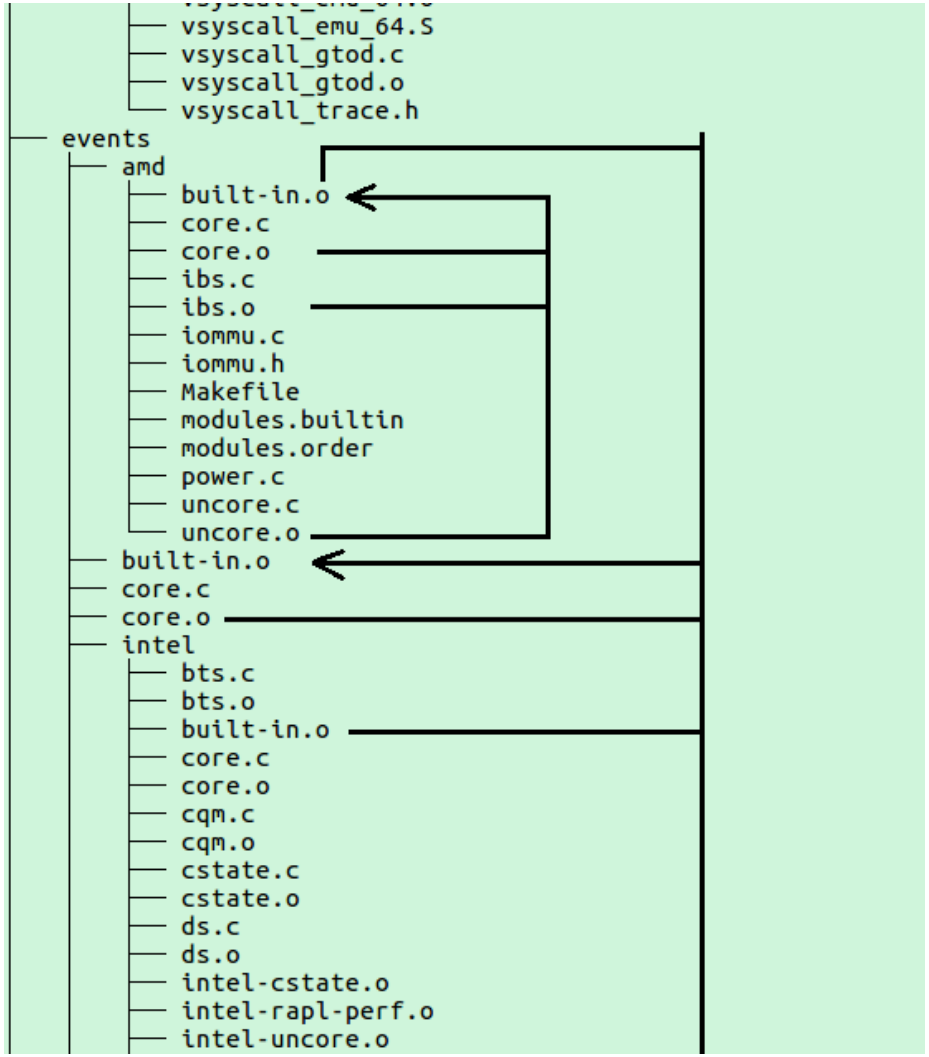
```
obj-$(CONFIG_UID16) += uid16.o
obj-$(CONFIG_MODULES) += module.o
obj-$(CONFIG_MODULE_SIG) += module_signing.o
obj-$(CONFIG_KALLSYMS) += kallsyms.o
obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
obj-$(CONFIG_CRASH_CORE) += crash_core.o
obj-$(CONFIG_KEXEC_CORE) += kexec_core.o
obj-$(CONFIG_KEXEC) += kexec.o
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 (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
- All `built-in.o` files are then linked and the resulting file `vmlinux` is located at the root of the source code directory.



Coding: 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
```

Coding: conditional inclusion

- It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing.
  - 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

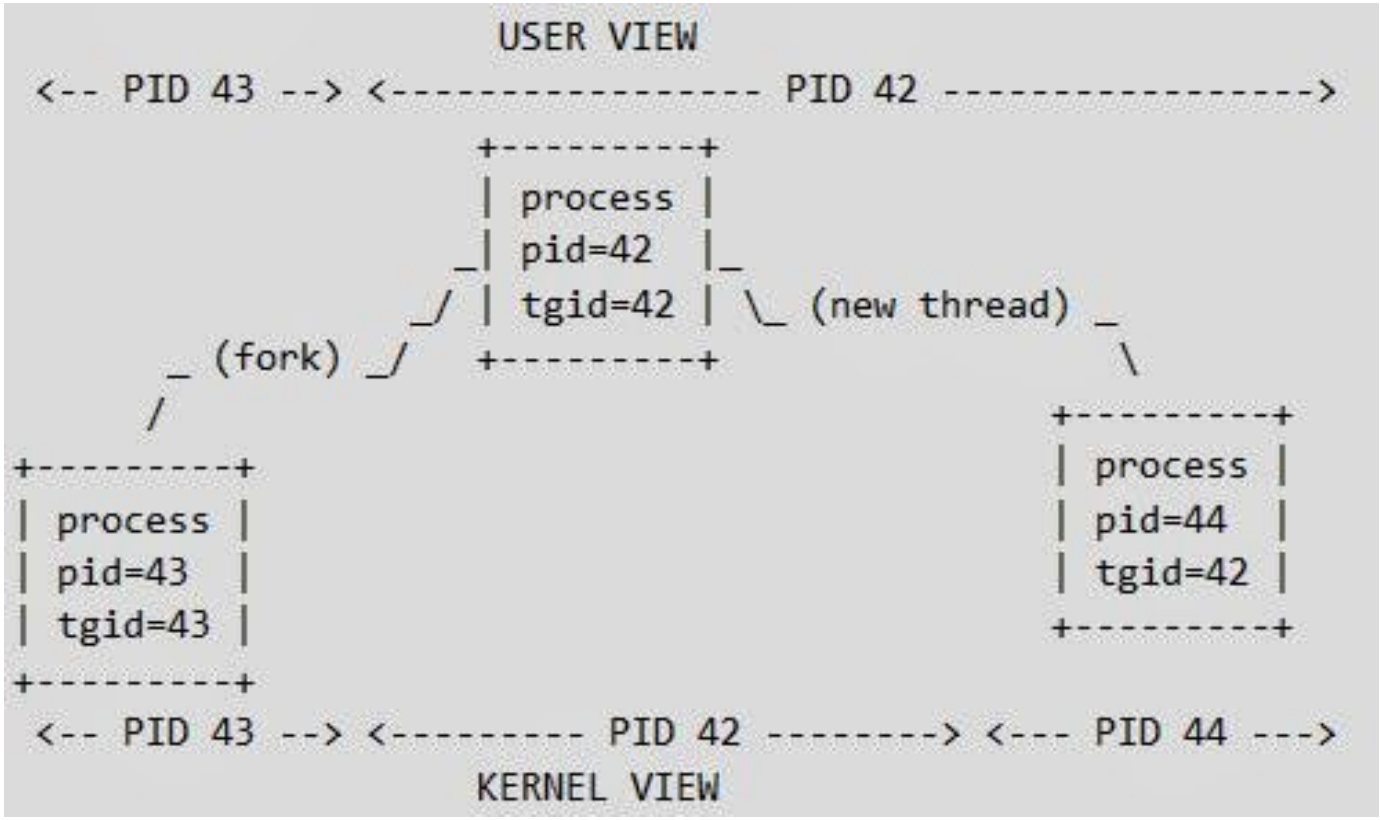
  There is an upper limit on the PID values
  - When the kernel reaches such limit, it must start recycling the lower, unused PIDs

Thread identification (I)

- Each process has it's own PID and they also have a TGID (thread group ID).
- 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.

Thread identification (II)

- From the schedule point of view, the Linux kernel does not differentiate threads and processes.
  - Both are managed by `struct task_struct` data structure.



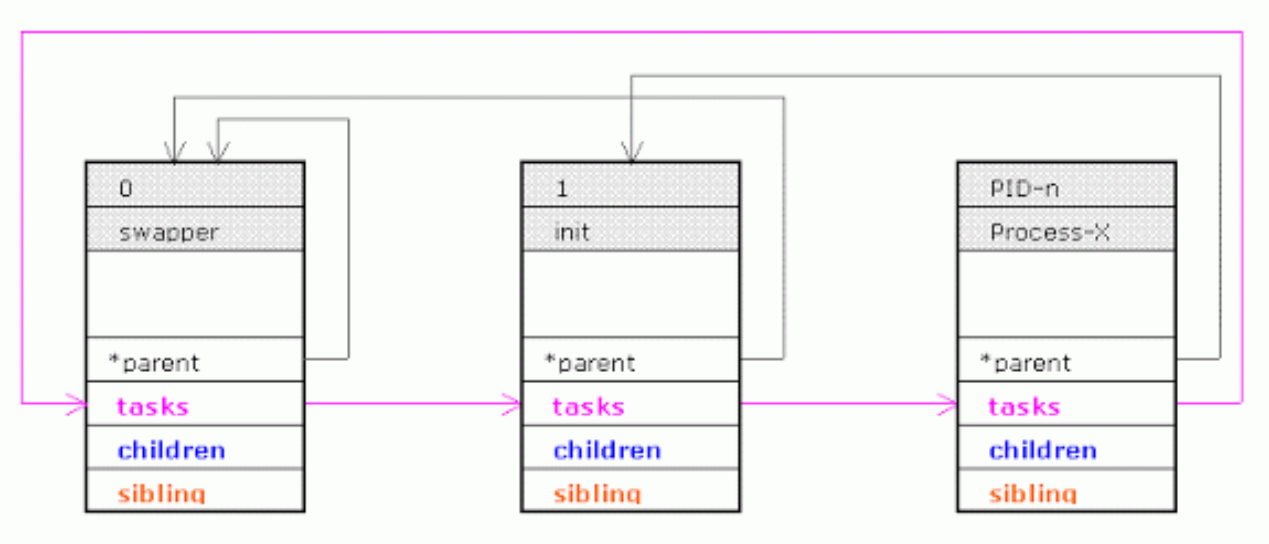
`struct task_struct` data structure

- The Linux kernel uses an instance of `task_struct` data structure (defined in `include/linux/sched.h`) to manage each process.

```
struct task_struct {
...
/* -1 unrunnable, 0 runnable, >0 stopped: */
volatile long state;
...
int prio;
...
const struct sched_class *sched_class;
...
unsigned int policy;
...
pid_t pid;
pid_t tgid;
...
};
```

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

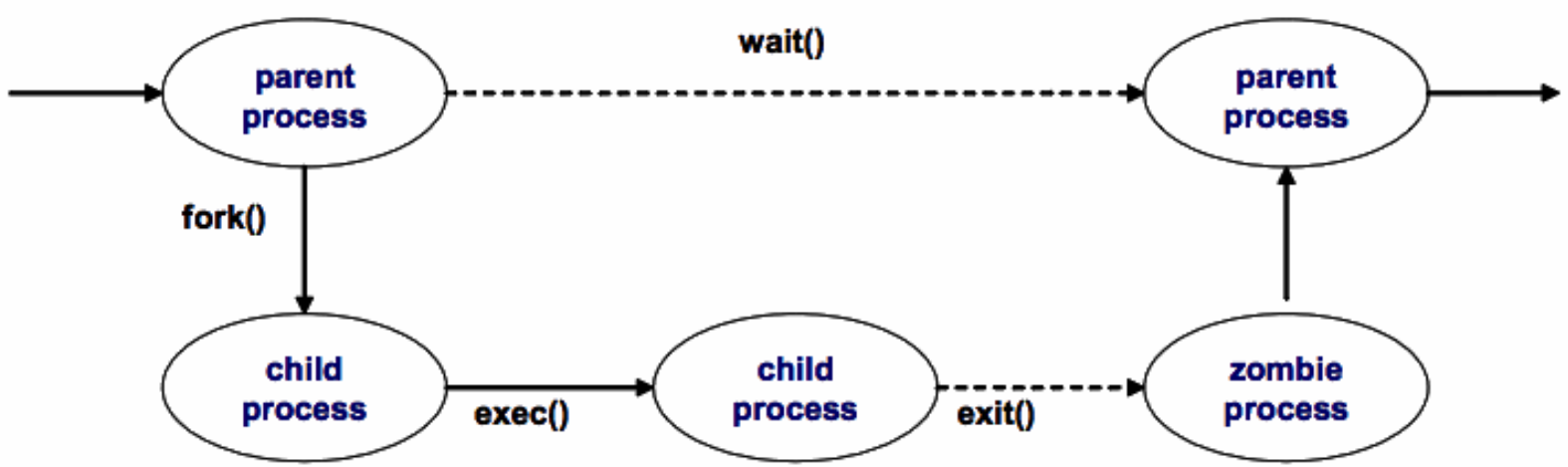


Process lifecycle (I)

- The first, `fork` , creates a child process that is a copy of the current task.
  - It differs from the parent only in its PID (which is unique), its PPID (parent's PID, which is set to the original process), and certain resources and statistics, such as pending signals, which are not inherited.
- The second function, `exec` , loads a new executable into the address space and begins executing it.
- 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.
- After process completes, the process descriptor for the terminated process still exists, but the process is a zombie and is unable to run. After the `parent` has obtained information on its terminated childthe 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.

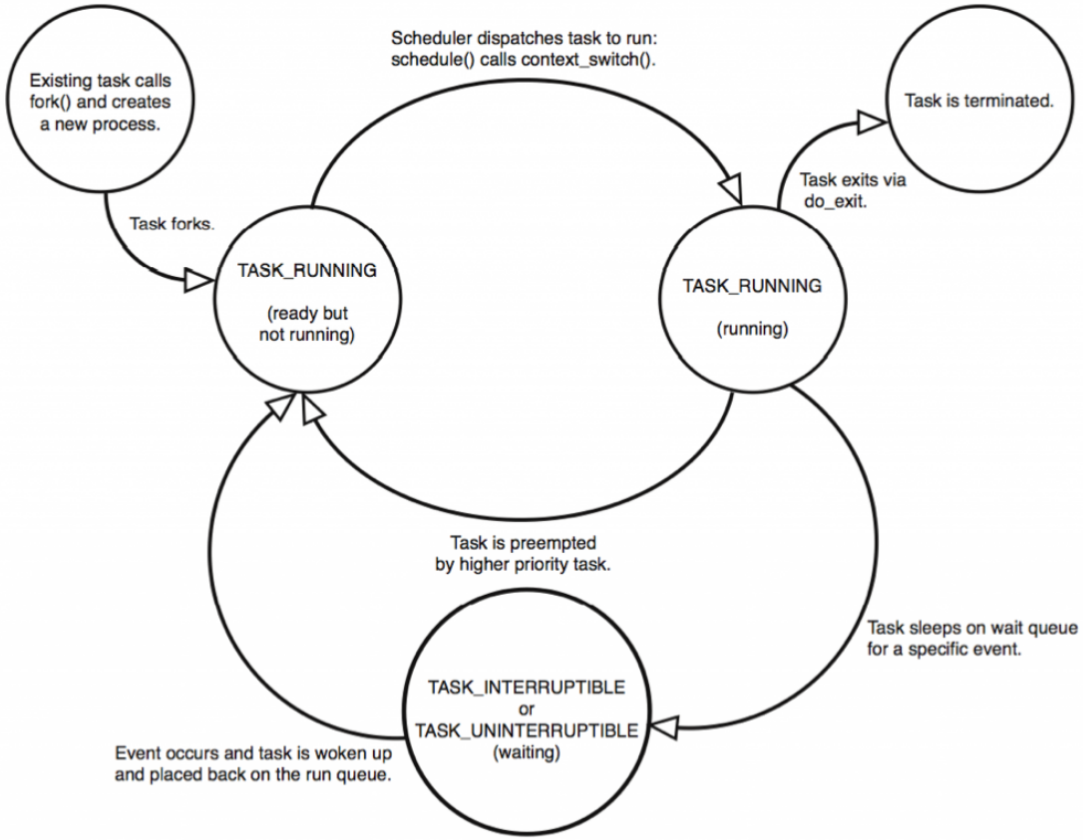


Parentless task

- 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 (I)

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



Task state (II)

- 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 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define __TASK_STOPPED 4
#define __TASK_TRACED 8
/* Used in tsk->exit_state: */
#define EXIT_DEAD 16
#define EXIT_ZOMBIE 32
#define EXIT_TRACE (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_DEAD 64
...
```

Task state (III)

- `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`.

Task exit state

- Range of values for `int exit_state` field of the `struct task_struct` data structure.
  - `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

- 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`

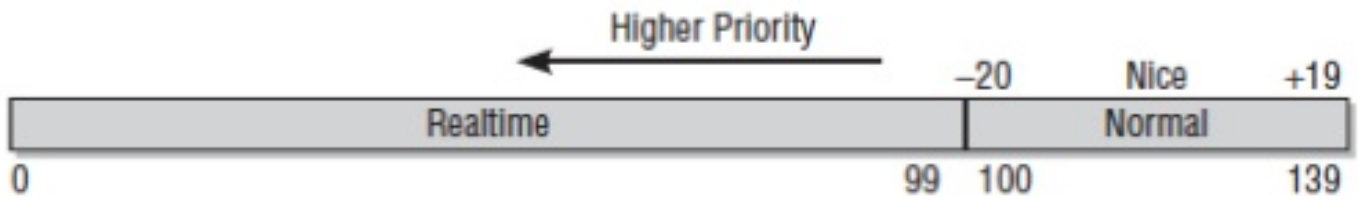
```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL 0
#define SCHED_FIFO 1
#define SCHED_RR 2
#define SCHED_BATCH 3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE 5
#define SCHED_DEADLINE 6

/* 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 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.
    - 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.
- Handleded 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.
  - Why this strange range was chosen is shrouded in history.
- 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;
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 = context_switch(rq, prev, next, &rf);
} else {
rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
rq_unlock_irq(rq, &rf);
}
...
}
```

`__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`

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
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);
cpu_load_update_active(rq);
calc_global_load_tick(rq);
rq_unlock(rq, &rf);
perf_event_task_tick();
...
}
```

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/