A Deep Dive into the Linux Kernel
Processes and Syscall

Ahmed Ali-Eldin

Practicalities

- Book: Linux Kernel Development 3rd Edition, Robert Love
  - Available as E-Book via the library
  - Available in hard-copy
  - Google the book, it is a great book
- Part of the course, so part of the Midterm exam
  - You are expected to understand both Minix and Linux

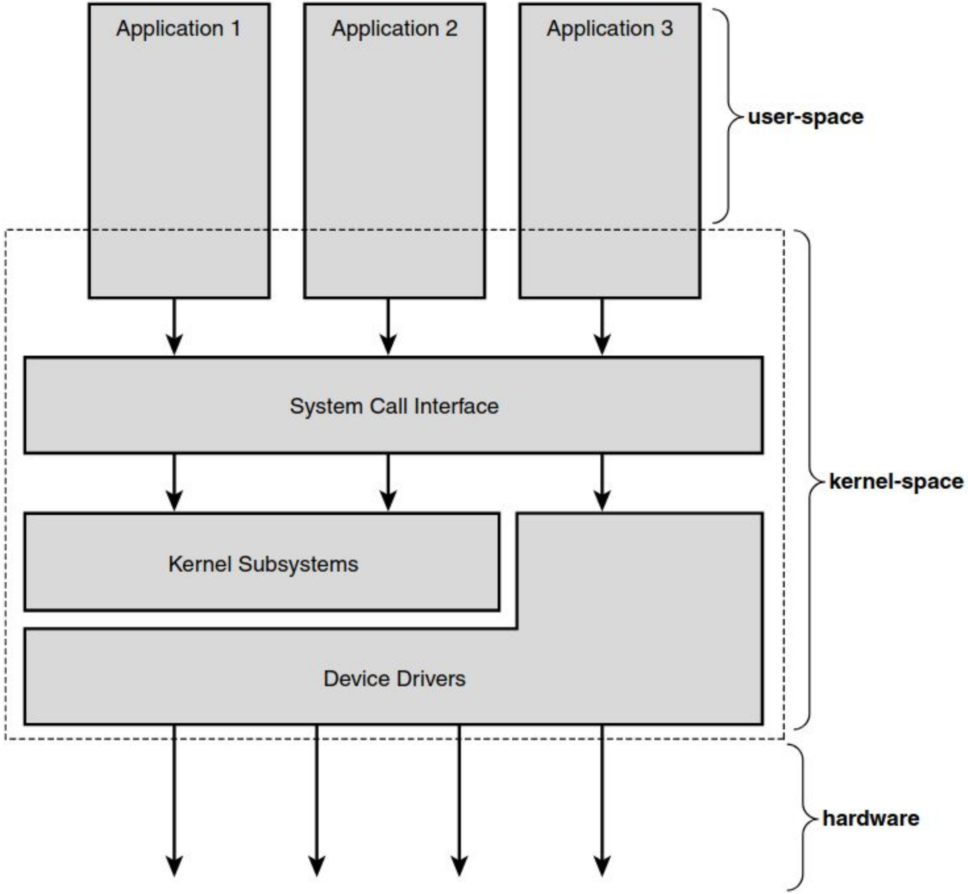
- My assumption, you know Minix, so let us look at Linux and compare

Linux very short history

- Started by an undergrad at the University of Helsinki
    - Frustrated by Minix licensing issues
    - Ported some code from a previous project, the GNU project
        - The GNU project started in 1983, to create a "complete Unix-compatible software system"
    - People took real notice in the mid-nineties
    - Today, maintained by the Linux foundation
        - Stable release by Greg Kroah-Hartman (http://www.kroah.com/)

Linux system model

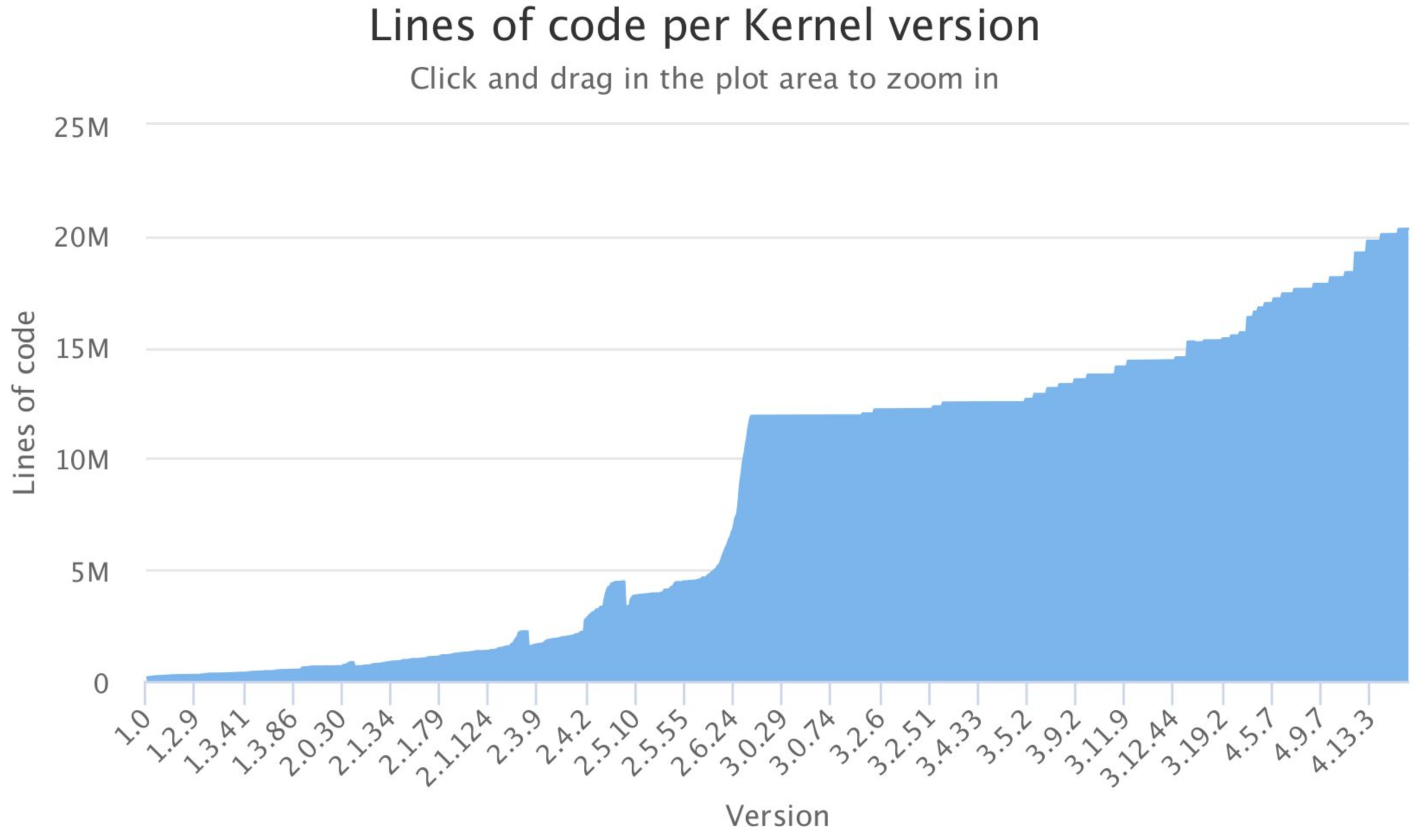
- Each processor is
  - In user-space, executing user code in a process
  - In kernel-space, in process context, executing on behalf of a specific process
  - In kernel-space, in interrupt context, not associated with a process, handling an interrupt



A beast of a different nature

- The kernel has access to neither the C library nor the standard C headers.
- The kernel is coded in GNU C.
- The kernel lacks the memory protection afforded to user-space.
- The kernel cannot easily execute floating-point operations.
- The kernel has a small per-process fixed-size stack.
- Because the kernel has asynchronous interrupts, is preemptive, and supports SMP, synchronization and concurrency are major concerns within the kernel.
- Portability is important.

Linux is a beast



Floating Point operations in the kernel

- Horrible idea
  - Floating point operations in user space
  - the kernel normally catches a trap and then initiates the transition from integer to floating point mode
  - What does this mean, varies by architecture
  - Using a floating point inside the kernel requires manually saving and restoring the floating point registers

Concurrency and Synchronization

- Race conditions can happen and will happen when developing in the kernel
  - Linux is a **preemptive multitasking** operating system.
    - Processes are scheduled and rescheduled at the whim of the kernel's process scheduler.
    - The kernel must synchronize between these tasks.
  - Interrupts occur asynchronously with respect to the currently executing code.
    - without proper protection, an interrupt can occur in the midst of accessing a resource, and the interrupt handler can then access the same resource.
  - The Linux kernel is preemptive.
    - without protection, kernel code can be preempted in favor of different code that then accesses the same resource.

Kernel Source tree

No servers

Directory	Description
arch	Architecture-specific source
block	Block I/O layer
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
firmware	Device firmware needed to use certain drivers
fs	The VFS and the individual filesystems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
samples	Sample, demonstrative code
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)
tools	Tools helpful for developing Linux
virt	Virtualization infrastructure

Linux Process Management

The Process abstraction

- Thread share the virtual memory abstraction but each receive it own virtual processor
- A program itself is not a process;
  - a process is an active program and related resources
  - Open files, address space…
- In the Linux code base, **processes are tasks**

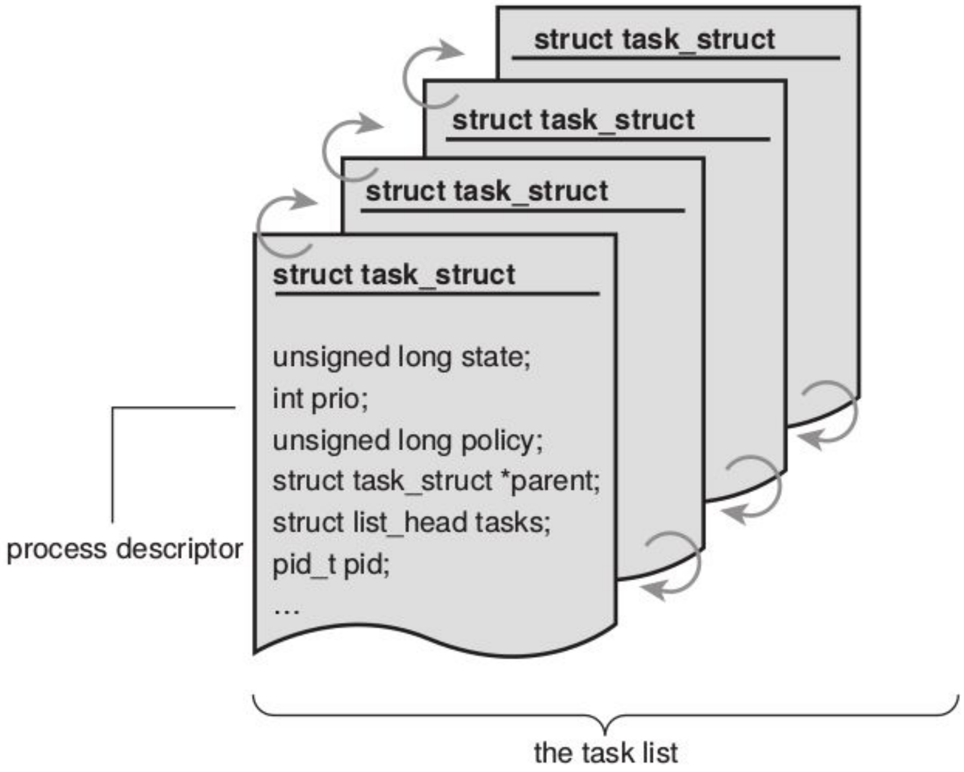
The Linux Task Structure

- A circular doubly linked list called the task list (or a task array)
- It is long with around 500 lines
  - around 1.7 kilobytes on a 32-bit machine

sched.h

```c

struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
    unsigned int wakee_flips;
    unsigned long wakee_flip_decay_ts;
    struct task_struct *last_wakee;

    int wake_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group *sched_task_group;
#endif
    struct sched_dl_entity dl;

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif

    unsigned int policy;
    int nr_cpus_allowed;
    cpumask_t cpus_allowed;

#ifdef CONFIG_PREEMPT_RCU
    int rcu_read_lock_nesting;
    union rcu_special rcu_read_unlock_special;
    struct list_head rcu_node_entry;
    struct rcu_node *rcu_blocked_node;
#endif /* #ifdef CONFIG_PREEMPT_RCU */
#ifdef CONFIG_TASKS_RCU
    unsigned long rcu_tasks_nvcsw;
    bool rcu_tasks_holdout;
    struct list_head rcu_tasks_holdout_list;
    int rcu_tasks_idle_cpu;
#endif /* #ifdef CONFIG_TASKS_RCU */
```

The Task list

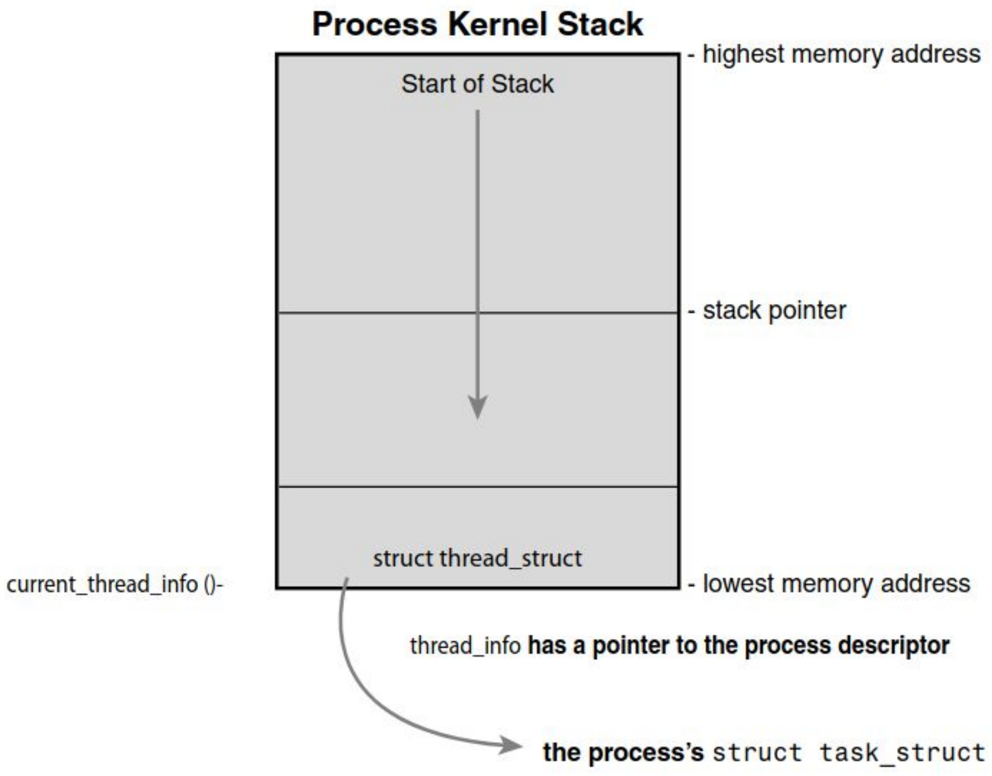


Linux Process Tree

- All processes are descendants of the init process, whose PID is one.
  - The relationship between processes is stored in the process descriptor.
  - Each *task_struct* has a pointer to the parent's *task_struct* , named parent
  - And a list of children, named children

```
	/*
	 * pointers to (original) parent process, youngest child, younger sibling,
	 * older sibling, respectively.  (p->father can be replaced with
	 * p->real_parent->pid)
	 */
	struct task_struct __rcu *real_parent; /* real parent process */
	struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
	/*
	 * children/sibling forms the list of my natural children
	 */
	struct list_head children;	/* list of my children */
	struct list_head sibling;	/* linkage in my parent's children list */
	struct task_struct *group_leader;	/* threadgroup leader */
```

Per Process Kernel Stack



Process Creation

- Unix/Linux separates creating a new process into two distinct functions: *fork()* and *exec()*

Fork and exec

- *fork()*
  - Creates a child process that is a copy of the current task
  - Differs only from the parent in its (unique) PID
  - its parent PID which is set to its original ID
  - A few other signals
- *exec()*
  - loads a new executable into the address space and begins executing it

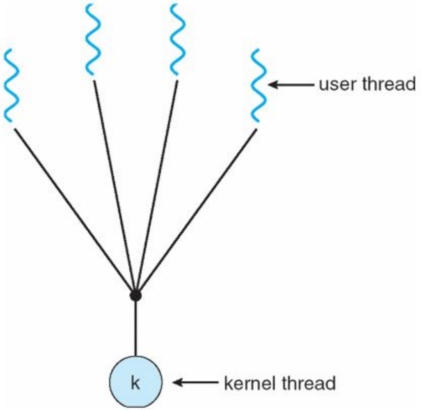
copy-on-write

- Delay or altogether prevent copying of the data
- Rather than duplicate the process address space, the parent and the child can share a single copy.
- The data, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy.

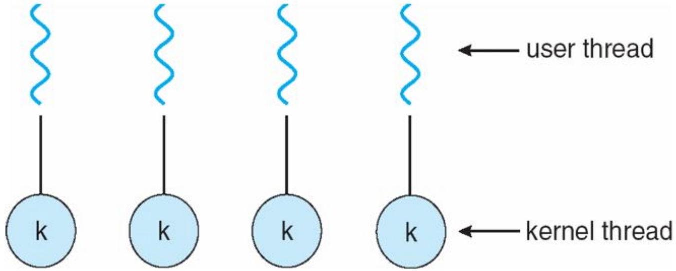
Threading in Linux

Remember..

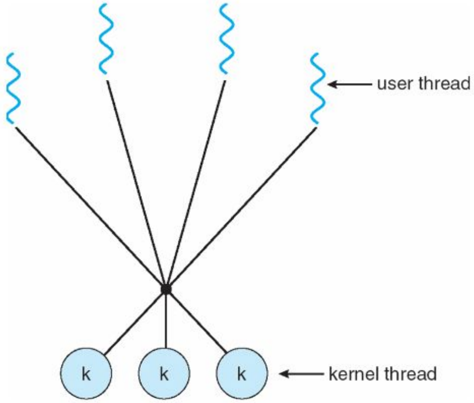
Minix

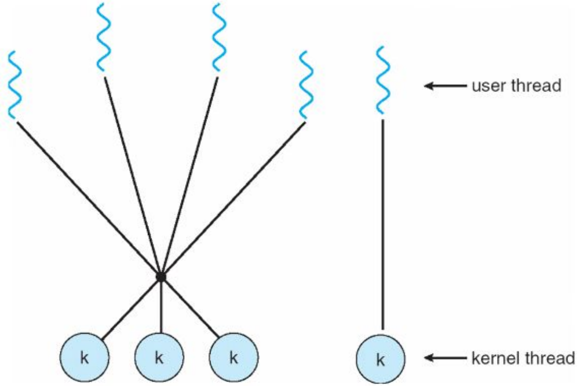


Linux



Old Solaris
(but also
GoLang!)





Side note: Why M:N in GoLang?

- Because it decouples concurrency from parallelism.
    - A 100 requests/sec to a web-server running on 4 cores

Back to Linux

- Kernel has no real threads
  - Everything is a process, i.e., kernel has no special data-structures or semantics to handle threads
  - Each thread thus has a unique *task_struct*
  - Windows, Solaris, and many other OSes have an explicit kernel support for threads, sometimes referred to as lightweight processes
  - To Linux, threads are simply a manner of sharing resources between processes
  - Threads created using *clone()* syscall

Clone() flags

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.

Kernel threads

- Special threads for the kernel to run operations in the background
- Exist only in the kernel with no corresponding user-level thread
- They are schedulable and preemptable
- To see the kernel threads running on your Linux machine
    - *ps -ef*
- More on this in later Linux lectures!

Process (and thread) termination

- Process destruction is self-induced.
  - occurs when the process calls the exit() system call
  - explicitly when it is ready to terminate
  - implicitly on return from the main subroutine of any program.
  - Involuntarily, due to a signal or an exception
  - bulk of the work is handled by *do_exit()* (defined in *kernel/exit.c*)
  - After do_exit() completes, the process descriptor for the terminated process still exists, and the process is a zombie
    - enables the system to obtain information about a child process after it has terminated

Process (and thread) termination

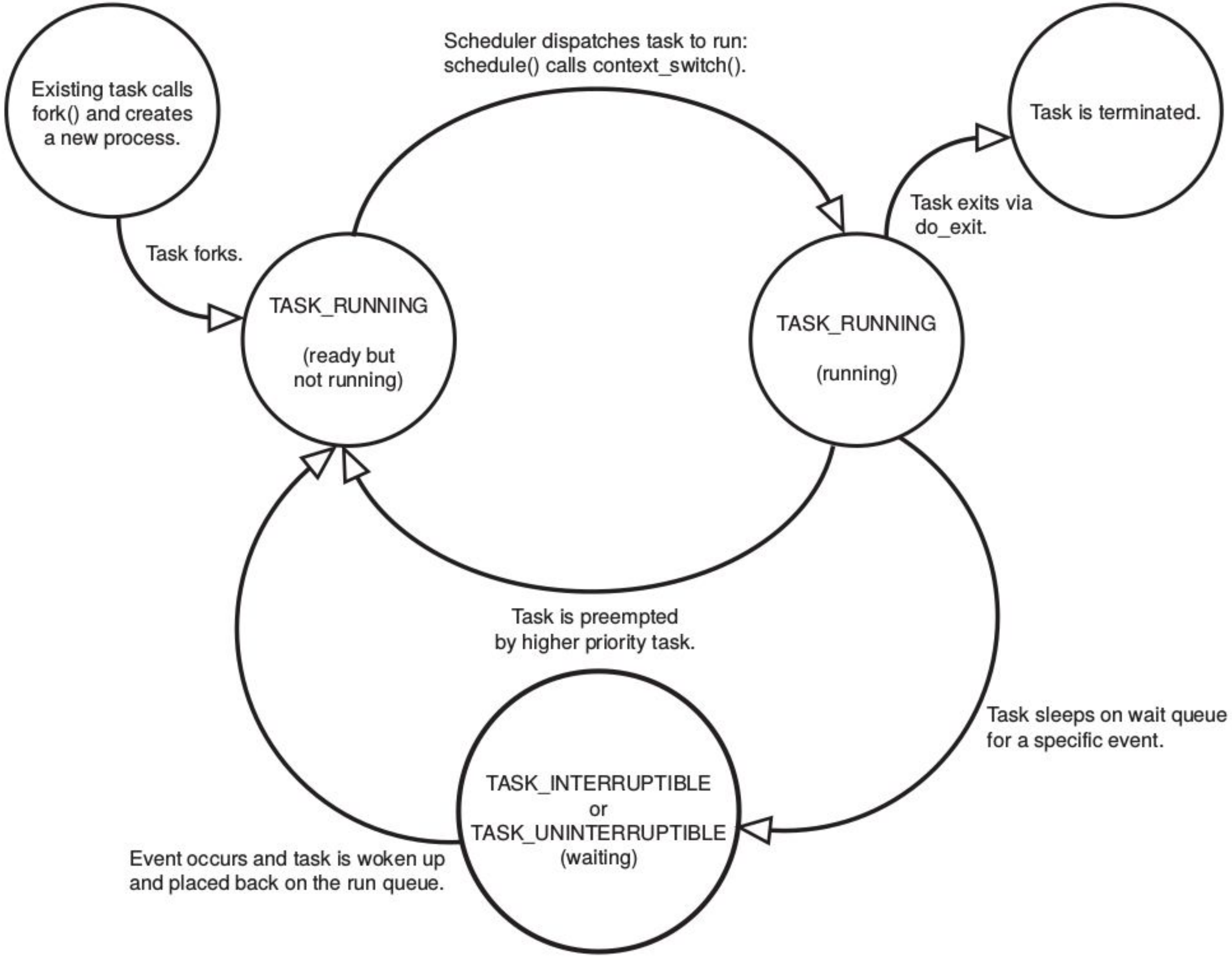
- Parent in charge of cleaning up after children
  - Remember, all tasks/processes/threads have a parent
- The acts of cleaning up after a process and removing its process descriptor are separate
- Parent has obtained information on its terminated child, or signified to the kernel that it does not care, the child’s task_struct is deallocated.

What if the parent dies/exits?

- Children are re-parented
  - either another process in the current thread group
  - or, if that fails, the init process

Process Scheduling

Process states



Multitasking

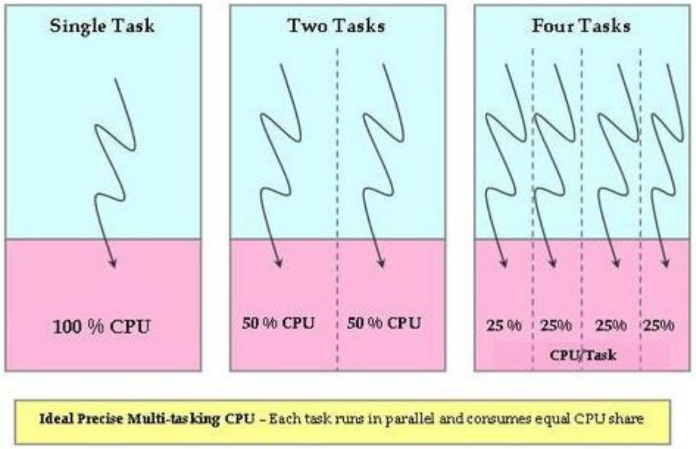
- Linux interleaves the execution of more than one process
  - On Mutli-processor machines, processes can run in parallel
- Linux uses preemptive multitasking
  - Scheduler kicks out tasks based on some algorithm
  - Usually after a given time-slice
  - This is opposite to cooperative multitasking where tasks run for as long as they wish
    - Mac OS 9 and Windows 3.1 (two ancient OSes) used cooperative multi-tasking

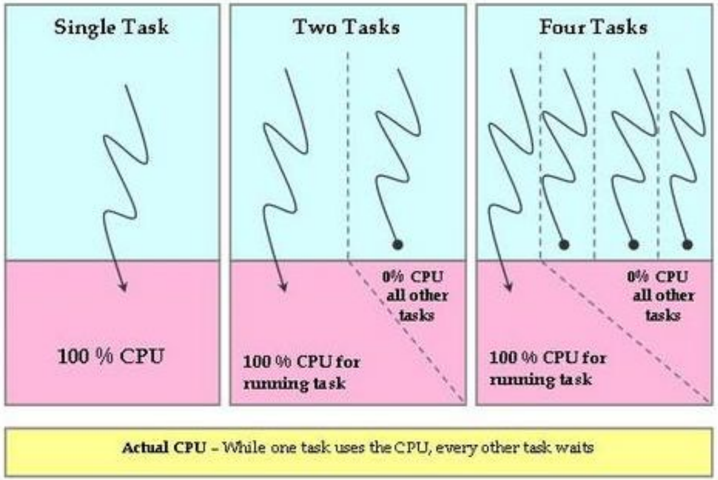
Evolution of Linux Process scheduler

- Before kernel v2.4, very naive scheduler that scaled poorly
- In v2.5, Linux introduced a new scheduler, commonly called the *O(1)* scheduler
  - A constant time algorithm to pick which process to run
  - Scaled to 100s of cores
  - But had several shortcomings with latency-sensitive applications
    - Extremely slow which made things bad for many applications
- In v2.6, introduced multiple new schedulers for the user to choose from
  - The most notable of these was the Rotating Staircase Deadline scheduler,
  - introduced the concept of fair scheduling, borrowed from queuing theory,

The Completely fair Scheduler

- Developed as part of v2.6.23, and rolled out in october 2007
  - Default scheduler today
  - Reading: https://www.linuxjournal.com/node/10267





Scheduling primer

- I/O Bound vs CPU bound processes
  - Run until blocked vs run until preempted
    - Word processor vs Matlab
  - Linux favors I/O bound processes
- Priorities
  - Nice values from -20 to 19
- Timeslices
  - CFS has a novel approach to calculate a timeslice
  - Assigning a proportion of the processor based on the current load in the system, with the nice value acting as a weight
    - Processes with higher nice values (a lower priority) receive a deflationary weight, yielding them a smaller proportion of the processor
    - Processes with smaller nice values (a higher priority) receive an inflationary weight, netting them a larger proportion of the processor.

Example: What should the scheduler do?

Consider a processor running

HEVC

H.265 - HIGH EFFICIENCY VIDEO CODING

OpenOffice.org™

Scheduler ideal scenario

- Have the word editor run fast
  - Give higher priority/CPU time
- Have the encoder use all processor when available
  - But get preempted by the word editor
- Other Operating Systems
  - Give higher prio + higher time slice to interactive apps
- Linux
  - Guarantee the text editor a certain proportion of the processor, i.e., 50% in this case
  - When the word editor blocks, run the encoder
  - When the editor wakes up, preempt the encoder

The Linux Scheduling algorithm

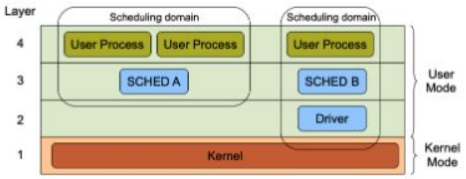
- Linux scheduler is modular
  - Huge difference from most other operating systems today
  - Multiple schedulers can be running for different processes!
    - All in parallel
  - This is the concept of scheduler classes
- Which scheduler class takes precedence controlled by a class priority
  - Base scheduler defined in *kernel/sched.c*
  - CFS is registered as the base scheduler for all normal processes
  - Let us look at the different available schedulers
  - https://github.com/torvalds/linux/blob/master/kernel/sched/sched.h

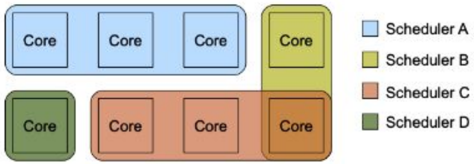
```c
#define RUNTIME_INF             ((u64)~0ULL)

static inline int idle_policy(int policy)
{
        return policy == SCHED_IDLE;
}
static inline int fair_policy(int policy)
{
        return policy == SCHED_NORMAL || policy == SCHED_BATCH;
}

static inline int rt_policy(int policy)
{
        return policy == SCHED_FIFO || policy == SCHED_RR;
}

static inline int dl_policy(int policy)
{
        return policy == SCHED_DEADLINE;
}
```

Compare to Minix 3 (from last lecture)

Multiple Schedulers

Moving Scheduler into user space presents an important scheduling opportunity to create multiple schedulers, where a scheduler could exist per user, per device type, etc. Also, this allows better utilization of a multicore system as it allows higher cpu utilization and load balancing.





Many reasons behind CFS

- Start interactive processes even if they have finished their timeslice
- Absolute time slices are a function of the timer ticks (clock speed)
  - Linux runs from embedded systems to large servers
- There are other reasons

Fair Scheduling

- Fairness, each task gets 1/n of the processor slice
- True life, context switching has a cost
  - Cache
  - Registers
  - Etc
- Instead run Round robin starting with the process that ran the least
- Each process run for for a timeslice proportional to its weight divided by the total weight of all runnable threads.
- If there are too many threads, switching cost becomes a huge issue
  - CFS defines a floor timeslice
  - Default is 1 ms

Implementation of CFS

- Time Accounting
- Process Selection
- The Scheduler Entry Point
- Sleeping and Waking Up

The Scheduler Entity Structure

- struct *sched_entity*, defined in

  <linux/sched.h>

```c
struct sched_entity {
	/* For load-balancing: */
	struct load_weight		load;
	unsigned long			runnable_weight;
	struct rb_node			run_node;
	struct list_head		group_node;
	unsigned int			on_rq;

	u64				exec_start;
	u64				sum_exec_runtime;
	u64				vruntime;
	u64				prev_sum_exec_runtime;

	u64				nr_migrations;

	struct sched_statistics		statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
	int				depth;
	struct sched_entity		*parent;
	/* rq on which this entity is (to be) queued: */
	struct cfs_rq			*cfs_rq;
	/* rq "owned" by this entity/group: */
	struct cfs_rq			*my_q;
#endif

#ifdef CONFIG_SMP
	/*
	 * Per entity load average tracking.
	 *
	 * Put into separate cache line so it does not
	 * collide with read-mostly values above.
	 */
	struct sched_avg		avg;
#endif
};
```

CFS implementation

- CFS Selection policy: Use the smallest vruntime
  - CFS uses a red-black tree to manage the list of runnable processes and efficiently find the process with the smallest vruntime
- Scheduler entry point
  - Function ***schedule()*** in *kernel/sched.c*
  - Finds highest priority scheduler class
- Sleeping and waking up
  - `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE`.