11. Case Study I UNIX (Linux)

9th ed: Ch. 18

10th ed: Ch. 20

Objectives

- To know a little of the history of UNIX from which Linux is derived
- To understand some principles upon which Linux's design is based
- To examine the Linux process model and lifecycle
- To describe how Linux schedules processes, provides kernel synchronization, and provides inter-process communication

Outline

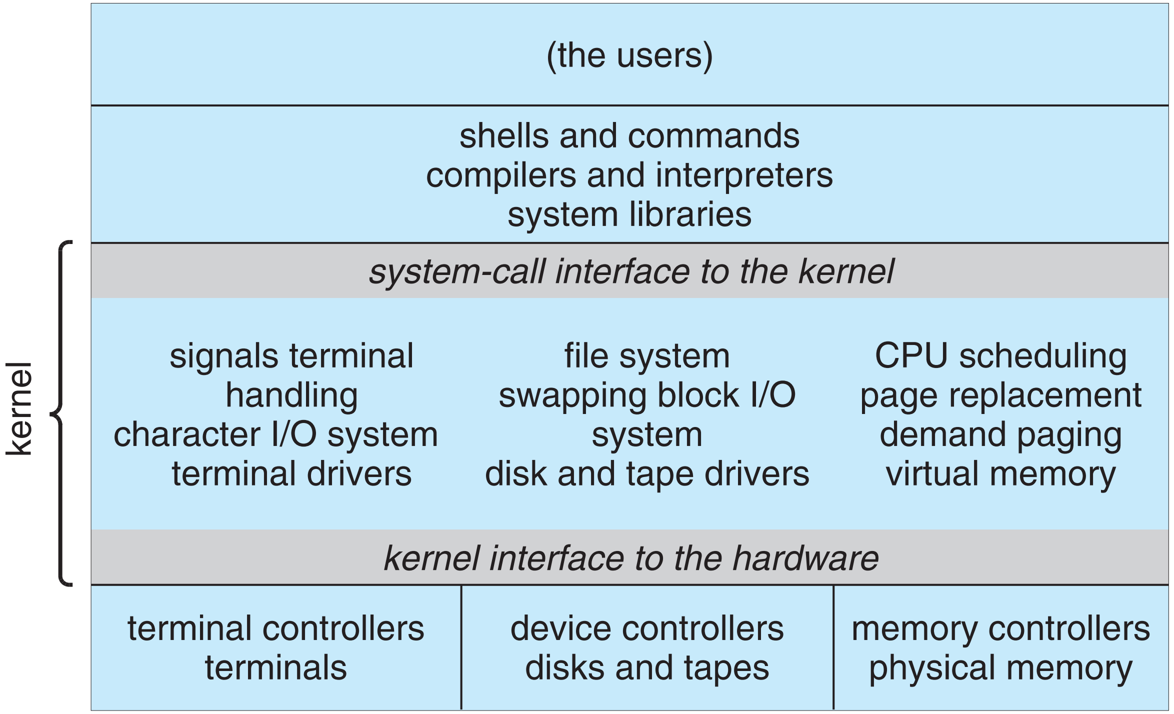
- UNIX / Linux
- Processes
- Tasks

Outline

- UNIX / Linux
  - History
  - Components
  - Kernel modules
- Processes
- Tasks

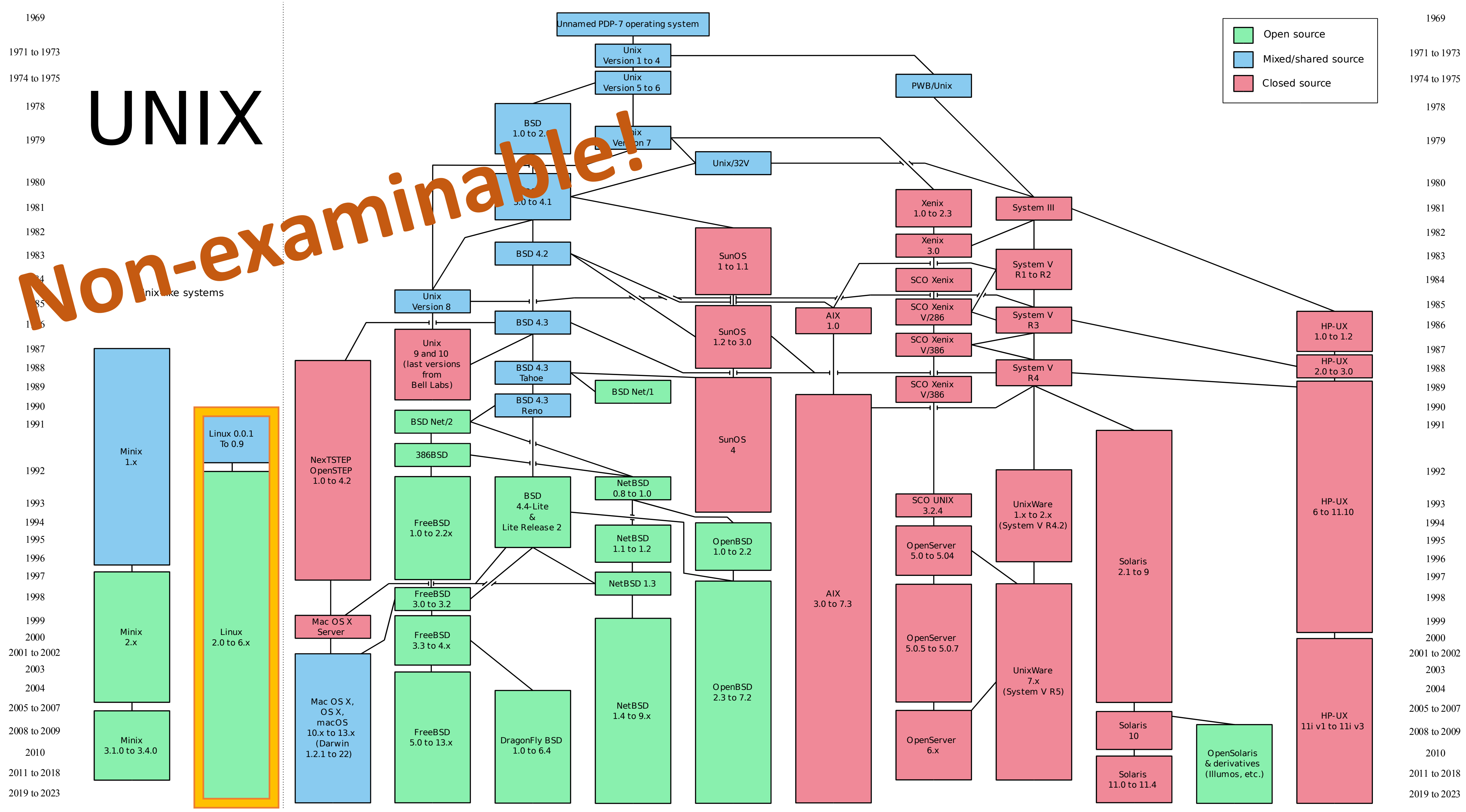
UNIX key feature

- Separation of kernel from user space
  - Only essential features inside the OS – editors, compilers etc are just applications
- Processes are the units of scheduling and protection
  - Command interpreter (shell) just another process
- All I/O looks like file operations
  - In UNIX, everything is a file



UNIX

Non-examinable!



By Eraserhead1, Infinity0, Sav_vas - Levenez Unix History Diagram, Information on the history of IBM's AIX on ibm.com, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1801948

UNIX history

- Developed in 1969 by Thompson & Ritchie at Bell Labs
  - A reaction to Multics which was rather bloated
  - Focus on (relative) ease-of-use due to e.g., interactive shell
  - In 1973 re-written from ASM to (portable) C even though performance critical
- Development continued through 1970s, 1980s
  - Notably, 1976 release of 6^{th} edition ("V6") included source code, so features could easily be added from other OSs
- From 1978 two main families
  - **System V** from AT&T and **BSD** from University of California at Berkeley
  - Introduction of POSIX standard, attempting to re-unify
  - Addition over time of, e.g., virtual memory, networking
  - Notably, 4.2BSD in 1983 included TCP/IP stack funded by DARPA
- Most common UNIX today is Linux

Linux history

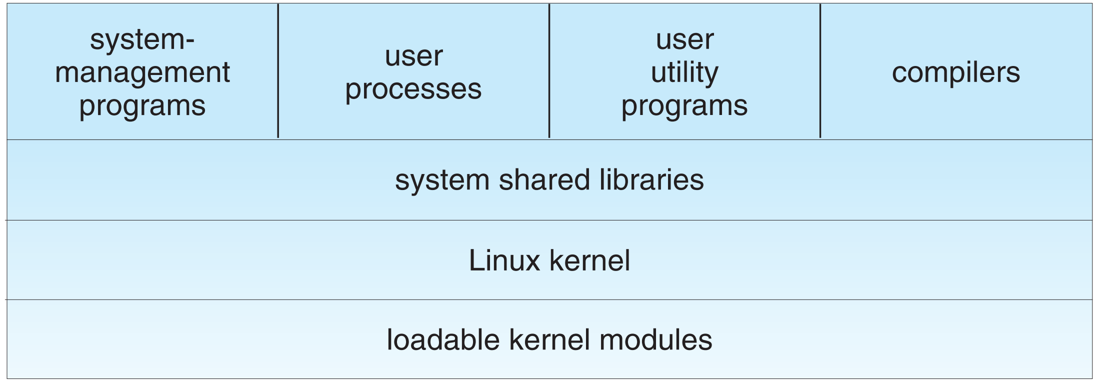
- A modern free OS based on UNIX standards
  - Originally a small self-contained kernel in 1991 by Linus Torvalds, release open-source
  - Designed for efficiency on common PC hardware but now runs on a huge range of platforms
  - Kernel entirely original but compatibility gives an entire UNIX-compatible OS, for free
  - Different distributions provide package management, support, configurations, tools, etc
  - Odd-number kernels are development kernels, even numbered are production
- Version 0.01, May 1991
  - No networking, Intel 80386-compatible processors and PC hardware only, extremely limited device-drive support, supported only the Minix file system
- Version 1.0, March 1994
  - TCP/IP plus BSD-compatible socket interface and device-driver support for IP on Ethernet
  - Enhanced file system and SCSI controller support for high-performance disk access
  - Linux 1.2, March 1995, was the final PC-only Linux kernel
- Development continues at pace

Linux design principles

- Multiuser, multitasking system with a full set of UNIX-compatible tools
  - File system adheres to traditional UNIX semantics
  - Fully implements the standard UNIX networking model
  - Designed to be POSIX compliant, achieved by at least two distributions
- Main design goals are speed, efficiency, and standardization
  - Constant tension between efficiency and security
- Supports Pthreads and a subset of POSIX real-time process control
- Linux programming interface has SVR4 UNIX semantics, not BSD

Components of a Linux system

- As most UNIX implementations, there are three main pieces
  - Most important distinction is between kernel and the rest
- The **kernel** is responsible for maintaining the important abstractions of the operating system
  - Executes in kernel mode with full access to all the physical resources of the computer
  - All kernel code and data structures share the same single address space
- **System libraries** define standard functions apps use to interact with the kernel
  - Implement much OS functionality that does not need kernel privileges
- **System utilities** perform individual specialized management tasks
  - Rich and varied user-mode programs



Kernel modules

- Sections of kernel code that can be compiled, loaded, and unloaded independently
  - Implement, e.g., device drivers, file systems, or networking protocols
  - Interface enables third parties to write and distribute non-GPL components
  - Enable a Linux system to be set up with a standard, minimal kernel, without extra device drivers compiled in
- Dynamic loading/unloading requires **conflict resolution**
  - Kernel must manage modules trying to access same hardware
  - E.g., reservation requests via kernel before granting access

Outline

- UNIX / Linux
- Processes
  - Management
  - Properties
  - Context
  - Threads
- Tasks

Process management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations
    - The *fork* system call creates a new process before *exec* runs a new program
    - Under UNIX, a process encompasses all the information that the OS must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups:
    - Identity
    - Environment
    - Context

Process properties

- Identity
  - **Process ID** (PID) uniquely identifies and is used to specify the process
  - Process **credentials** in the form of a User ID and one or more Group IDs
  - Support for emulation gives **personality** – not traditional but allows slightly modified semantics of system calls
  - **Namespace** gives specific view of file system hierarchy – typically shared but can be unique
- Environment, inherited from parent as two null-terminated vectors
  - **Argument vector** listing command-line arguments used to invoke the running program
  - **Environment vector** lists *NAME*=*VALUE* pairs associating named variables with arbitrary values
  - Flexible way to pass information between user-mode components, giving per-process customisation
- Context
  - The (constantly changing) state of a running program at any point in time

Process context

- Most important part is the **scheduling context**
  - Required for the scheduler to suspend and restart the process
  - Also includes accounting information about current and past resources consumed
- An array of pointers into kernel file structures called the **file table**
  - I/O system calls use indexes into this table, the **file descriptor** (**fd**)
- Separately, **file-system context** applies to requests to open new files
  - Current root and default directories for new file searches are stored here
- **Signal-handler table** defines per-process per-signal signal handling routine
- **Virtual-memory context** describes full contents of process' private address space

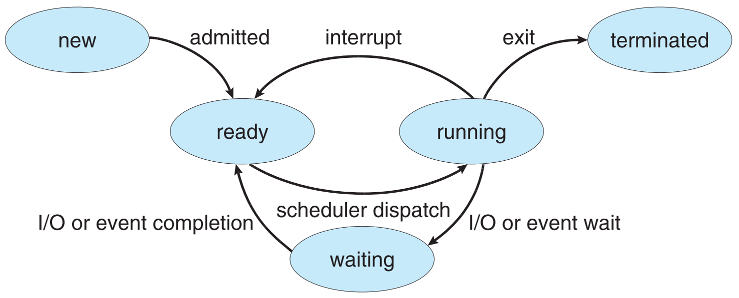
Processes and threads

- The same internal representation
  - A thread is just a new process that shares its parent's address space
- Both called **tasks** by Linux, distinguished only when created via *clone*
  - *fork* creates a new task with an entirely new task context
  - *clone* creates a new task with its own identity, but sharing parent's data structures
- *clone* gives control over exactly what is shared between two threads
  - File system, memory space, signal handlers, open files

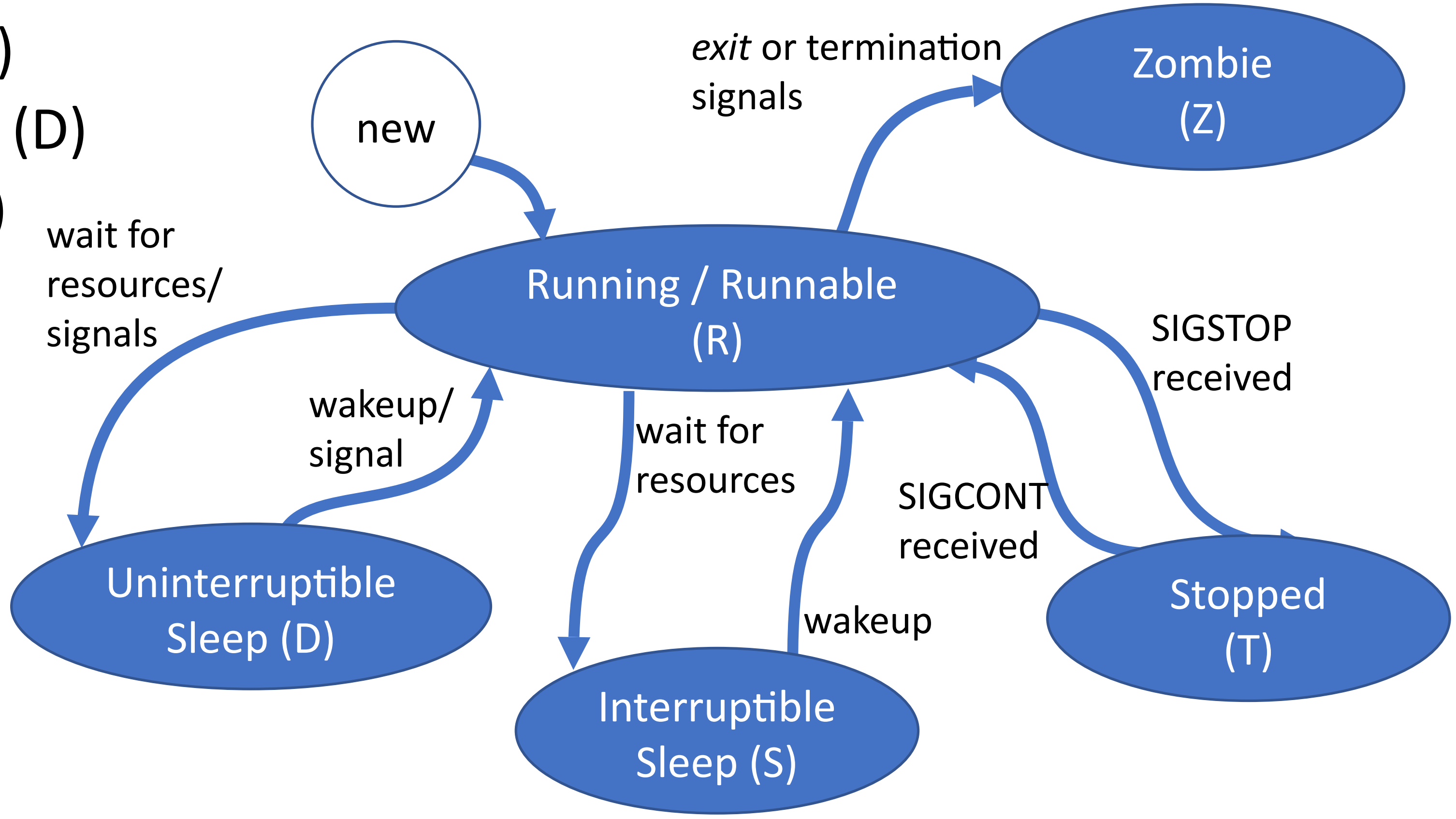
Outline

- UNIX / Linux
- Processes
- Tasks
  - Lifecycle
  - Scheduling
  - Synchronisation
  - Interrupt handlers
  - IPC

Task lifecycle



- Five states:
  - Running/Runnable (R)
  - Uninterruptible Sleep (D)
  - Interruptible Sleep (S)
  - Stopped (T)
  - Zombie (Z)



Task scheduling

- Allocation of CPU time to different tasks
  - As well as processes, in Linux this includes various kernel tasks
  - Those requested by a running process and those executed for a device driver
- Traditional UNIX scheduling uses fixed time slices and priorities to boost/penalise
  - Quantum 100ms, round-robin within priority levels
  - Priority set from process' base priority, average length of process' run queue, and *nice* value
- Worked ok for early time-sharing systems but did not scale or provide good interactive performance for current systems

Completely Fair Scheduler (CFS)

- Since 2.6.23 – no more time slices
  - Start by assuming every task out of N tasks should have 1/N of the CPU
  - Adjust based on **nice** value from -20 to +19: smaller is higher priority giving higher weighting
  - Run task *j* for a time slice $t_j \propto w_j / \sum_i w_i$
- Actual length of time given a task is the **target latency**
  - Interval during which time every runnable task should run at least once
  - E.g., target latency is 10ms, two runnable tasks of equal priority, each will run for 5ms
  - If ten runnable tasks, each runs for 1ms – but what if 1000 runnable tasks?
  - To avoid excessive switching overheads, **minimum granularity** is the minimum length of time for which a process will be scheduled
- CFS scheduler maintains per-task virtual run time in variable *vruntime*
  - Scheduler picks task with lowest *vruntime*; in default case, the same as actual run time
  - Lower priority means higher decay rate of *vruntime*
  - Implemented as red-black tree with left-most bottom-most value (lowest *vruntime*) cached

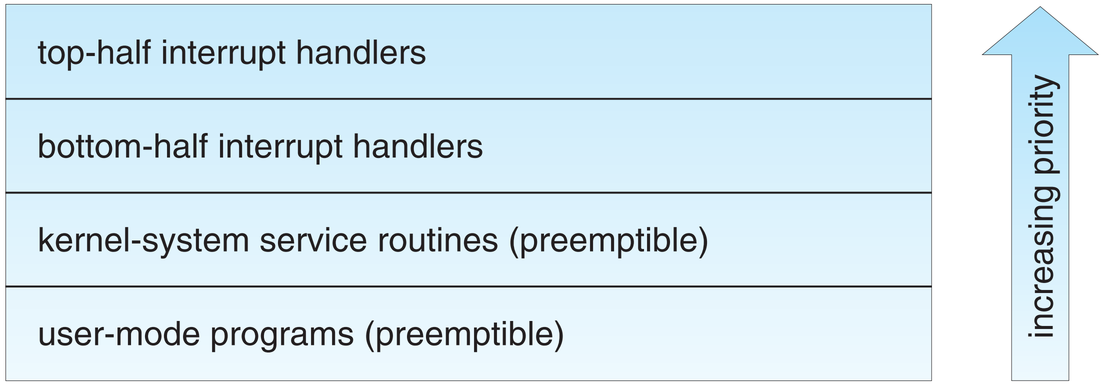
Kernel synchronisation

- Kernel-mode execution requested in two ways:
  - Process requests an OS service, explicitly via a system call or implicitly e.g. when a page fault occurs
  - A device driver delivers a hardware interrupt causing the CPU to start executing a kernel-defined handler for that interrupt
- Need guarantees that kernel's critical sections run without interruption by another critical section
  - Before 2.6, kernel code is **non-preemptible** so timer interrupt sets *need_resched*
  - After 2.6, either **spin locks** or **enable/disable pre-emption**

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Interrupt handlers, top and bottom

- Want long critical sections to be able to run without disabling interrupts for long periods of time
- Split interrupt service routines into a **top half** and a **bottom half**
  - **Top half** is a normal interrupt service routine, run with recursive interrupts disabled
  - **Bottom half** is run, with all interrupts enabled, by a miniature scheduler that ensures bottom halves never self-interrupt



- This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

Inter-Process Communication

- **Signals**
  - Process-to-process
  - Limited number, carry no information other than which signal has occurred
- **Wait queues**
  - Used inside the kernel
  - Process puts itself on wait queue for an event, and informs scheduler that it is no longer eligible for execution
  - All waiting processes are woken when the event completes
- **Pipes**
  - Just another type of *inode* in the VFS
  - Each pipe has a pair of wait queues for reader and writer to synchronise
- **Shared memory**
  - Fast but no synchronisation mechanism – need to be provided
  - Persistent object, like a small independent address space

Summary

- UNIX / Linux
  - History
  - Components
  - Kernel modules
- Processes
  - Management
  - Properties
  - Context
  - Threads
- Tasks
  - Lifecycle
  - Scheduling
  - Synchronisation
  - Interrupt handlers
  - IPC