Unix, Linux, Android

Table of Contents

- History
  - MULTICS
  - PDP-11 UNIX
  - Portable UNIX
  - Berkeley UNIX
  - POSIX
  - MINIX
  - Linux
- Linux Overview
  - Linux Goals
  - Interfaces
- Kernel Structure
  - I/O Component
  - Interdependence
  - System call interface
- Processes
  - Signals
  - Implementation
  - Process Descriptor
  - Executing `ls`
  - Kernel Threads and `clone`
- Scheduling
  - Linux O(1) Scheduler
  - Completely Fair Scheduler (CFS)
- Boot Process
  - Dynamic Loading
  - `init`
- Memory Management

  - System Calls
  - Implementation
  - Physical memory
  - Paging Scheme
  - Memory-Allocation
  - Representation of Virtual Address Space
  - Paging
  - Page Frame Reclaiming Algorithm

History

MULTICS

- 1940s-1950s: book time on a computer
- 1960s: batch systems where you leave your punched cards at the machine room
  - long delay between submitting and getting the output
  - debugging immensely difficult
- timesharing invented at Dartmouth College and MIT
- **MULTICS: Multiplexed Information and Computing Service**: developed by Bell Labs and GE, before Bell Labs pulled out
- Ken Thompson: Bell Labs researcher writes stripped-down MULTICS in assembly on a PDP-7
- Brian Kernighan referred to it as **UNICS (Uniplexed Information and Computing Service**, which eventually became UNIX

PDP-11 UNIX

- UNIX was moved to PDP-11, which was a powerful machine and large memory, with memory-protection hardware, allowing multi-user support
- Thompson rewrote UNIX in high-level language he designed, B. But it lacked structures making it insufficient.
- Dennis Ritchie designed successor C and a compiler
- Thompson and Ritchie then rewrite UNIX in C, publishing landmark paper The UNIX Time-Sharing System in 1974
- Bell Labs, owned by AT&T, as a regulated monopoly was not allowed to be in the computer business, so licensed UNIX to universities for a modest fee. At the time, most universities had PDP-11s, and the OS that came with them was terrible, so UNIX came in at the right time.

Portable UNIX

- porting UNIX to a new machine was made much simpler once it was written in C. The process involves:
  - writing a C compiler for the new machine
  - writing device drivers for the new machine's I/O devices
  - small amount of machine-dependent code for interrupt handlers/memory management in assembly
- processing of porting to an Interdata machine revealed lots of assumptions implicitly made by UNIX about integers being 16 bits, pointers, being 16 bits, etc. UNIX had to be cleaned up to make it portable.
- a portable C compiler was implemented, allowing easy modification for any machine with moderate effort
- AT&T was broken up in 1984 by the US government, and accordingly established a computer subsidiary, releasing commercial UNIX product (System III/V)

Berkeley UNIX

- source code for UNIX was available, so Berkeley heavily modified it, with funding from ARPA
- 4BSD (Fourth Berkeley software distribution) introduced many improvements
  - virtual memory and paging
  - longer file names
  - reimplemented file system with improved performance
  - increased signal handling reliability
  - introduced networking causing TCP/IP protocol stack to become de facto standard in UNIX world
- added a number of utility programs: `vi`, `csh`, Pascal and Lisp compilers, …
- some vendors subsequently based their UNIX off Berkely UNIX rather than System V

POSIX

IEEE 1003.1-2017

- late 1980s: 4.3BSD and System V Release 3 were both in widespread use and were somewhat incompatible, with each vendor additionally complicating matters with their own non-standard enhancements

- prevented commercial success as software vendors could not package UNIX programs and guarantee that they would run on any system, as was the case with MS-DOS
- attempts were made to standardise, e.g. by AT&T issuing the SVID (System V Interface Definition), however this was only relevant to System V vendors and ignored by BSD
- IEEE Standards Board came together with industry, academia, and government to produce POSIX, a Portable Operating System, defined in standard 1003.1
- 1003.1 defines a set of library procedures that every conformant UNIX system must supply. Most procedures invoke a system call (e.g. `open`, `read`, `fork`), but some can be implemented outside the kernel
- as a result a vendor writing a program using only procedures defined by 1003.1 knows that this program will run on every conformant UNIX system
- approach taken by IEEE was unusual in that it took the intersection of System V and BSD, rather than the union, such that 1003.1 looks lie a common ancestor of both OSs
- related documents standardise threads, utilities, networking, …
- C has also been standardised by ISO/ANSI

MINIX

- modern UNIX is large and complicated, the antithesis of the original conception of UNIX
- 1987: MINIX was produced as a UNIX-like system that was small enough to understand, with 11,800 lines of C and 800 lines of assembly
- MINIX used microkernel design, providing minimal functionality in the kernel so that it is reliable and efficient. Memory management and file system become user processes, while the kernel handles message passing between processes and not much else
  - Kernel: 1600 lines of code + 800 lines of assembler
  - I/O device drivers: 2900 lines of C, also in kernel
  - File system: 5100 lines of C
  - Memory manager: 2200 lines of C
- microkernels vs monolithic:
  - the former is easier to understand and maintain, with a highly modular structure
  - highly reliable: a crash of a user mode process does much less damage than a kernel-mode crash
  - lower performance due to extra switches between user/kernel mode
- 2004: direction of MINIX development changed to focus on building an extremely reliable and dependent system that could automatically repair faults
  - all device drivers were moved to user space, each running as a separate process

  - kernel was under 4000 lines of code
- has been ported to ARM, so it is available for embedded systems

Linux

- many features were requested for MINIX but these were often denied due to the goal of keeping the system small
- 1991: Linus Torvalds wrote Linux as a UNIX clone, intending for it to be a full production system
  - monolithic, with the entire OS in the kernel
  - initially 9,300 lines of C + 950 lines of assembly
- 1994: v1.0, 165,000 lines of code
- 1996: v2.0, 470,000 lines of C, 8000 lines of assembly
- 2013: ~ 16 million lines of code
- version `A.B.C.D`:
  - `A`: kernel version
  - `B`: major version
  - `C`: minor revision, e.g. support for new drivers
  - `D`: minor bug fixes and security patches
- issued under GPL (GNU Public License), permitting you to use, copy, modify, and redistribute the source and binary code freely. All derivatives of the Linux kernel may not be sold/redistributed in binary form only, they must be shipped with source code
- 1992: Berkeley terminated BSD development, releasing 4.4BSD
  - FreeBSD was based off this
  - Berkeley issued software under an open source license
  - AT&T subsidiary sued Berkeley, keeping FreeBSD off the market for long enough for Linux to become well established
  - had this not happened, Linux would have an immature OS competing with mature and stable system

Linux Overview

Linux Goals

- UNIX: interactive system for multiple processes and multiple users, designed by programmers for programmers

Goals

- simplicity
- elegance
- consistency
- power
- flexibility
- avoid useless redundancy

Examples

- files should just be a collection of bytes, not with different classes
- principle of least surprise: e.g. `ls *A` and `rm *A` should be predictable and
- small number of basic elements should be able to be combined infinite ways as needed

Interfaces

- operating system runs on bare hardware, controling the hardware and provides system calls interface to programs
- system calls allow users to create and manage processes, files, resources
- programs make system calls by putting arguments in registers and issuing trap instructions. As there is no way to write a trap instruction in C, a standard library is provided with one procedure per system call
- standard library is written in assembly but can be called from C
- POSIX specifies the library interface, not the system call interface
- Linux also supplies standard programs, some of which are specified by 1003.2, which get invoked by the user

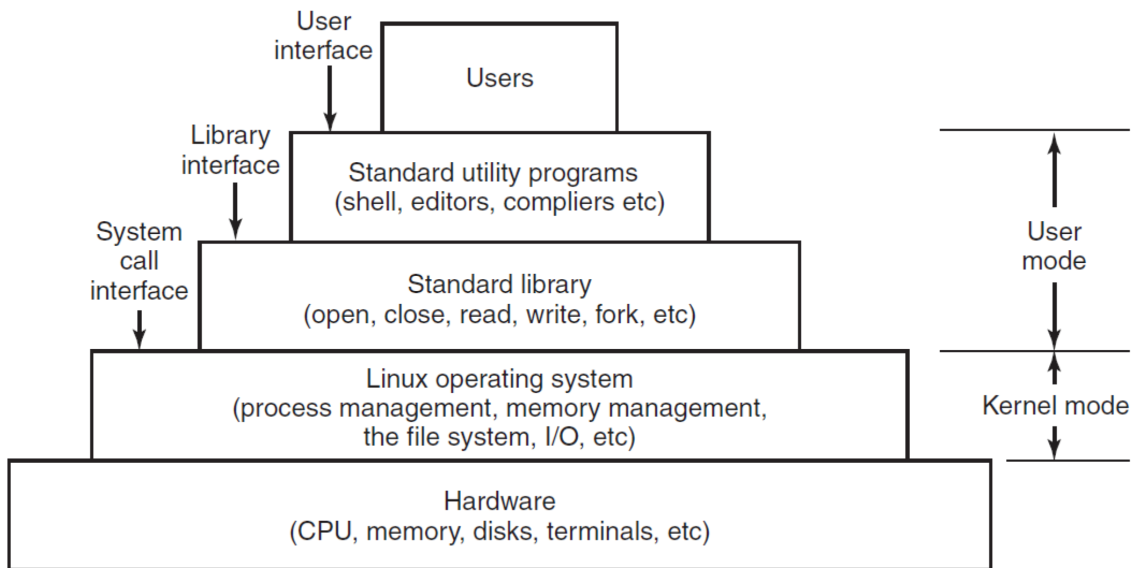

Figure 10-1. The layers in a Linux system.

Figure 1: linux-interfaces

- GUIs are supported by X windowing system (X11/X), defining communication and display protocols for window manipulation on bitmap displays
- X server: controls devices and redirects I/O
- GUI environment built on top of low-level library `xlib` which contains functionality to interact with the X server
  - graphical interface extends functionality of X11 by enriching window view: e.g. buttons, menus, …
  - `xterm`: terminal emulator program, providing basic command-line interface to OS

Kernel Structure

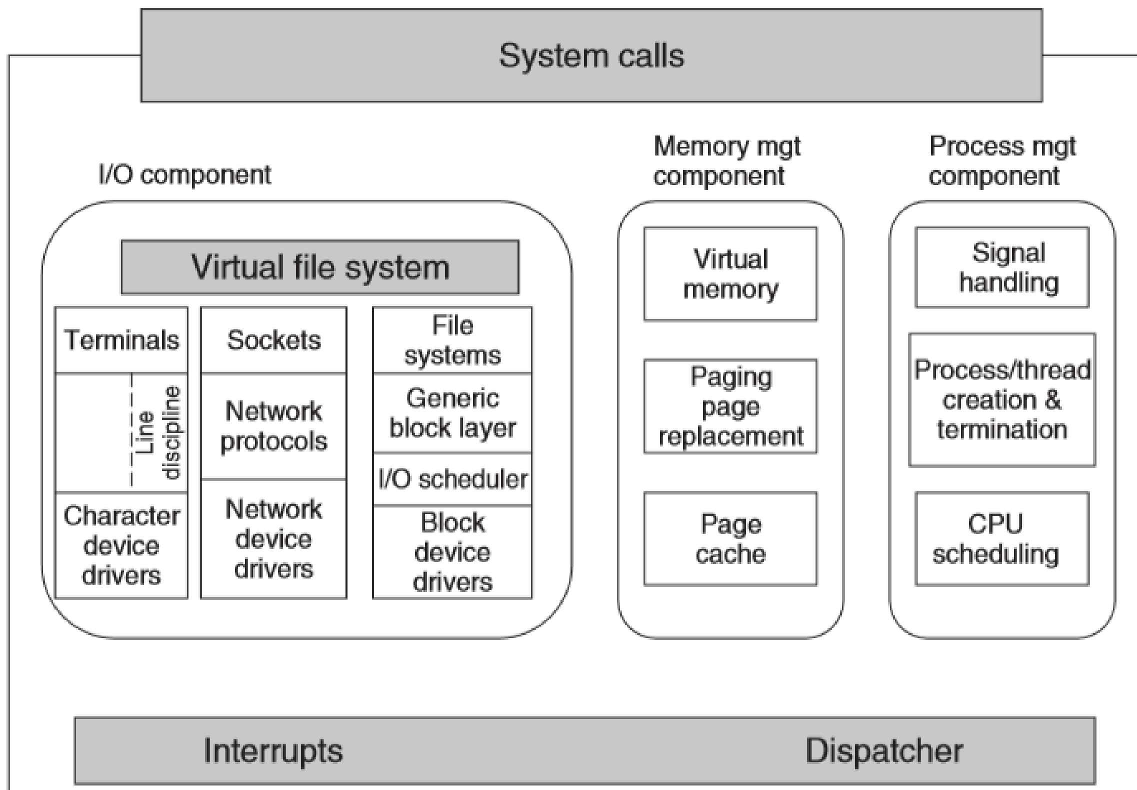

Figure 2: linux-kernel

- interrupt handlers: primary way of interacting with devices
- dispatching occurs as the result of an interrupt: code stops running process, saves its state and starts the appropriate driver
  - written in assembler

I/O Component

- **I/O component**: responsible for interacting with devices, network, storage I/O operations
- I/O operations are integrated under a **Virtual File System (VFS)** layer
- at the low-level all I/O operations pass through a device driver
- drivers are either character/block-device, depending on whether they are random access or not, and network devices, while a form of character devices are sufficiently different to consider them a different category

- **character devices** used in 2 ways:
  - every keystroke: e.g. `vim`
  - line oriented: e.g. `bash`. The character stream is passed through a line discipline
- **network devices**:
  - above network drivers is full functionality of hardware router
  - above routing code is protocol stack (including IP/TCP)
  - above this is the socket interface, allowing programs to create sockets
- **block devices**:
  - I/O scheduler orders/issues disk-operation requests per some system policy
  - file system: Linux has multiple file systems concurrently, so to abstract away architectural differences, a generic block-device layer is used by all file systems

Interdependence

- 3 components are interdependent
- e.g. page caches may be used to hide latencies of accessing files: block device dependent on memory manager
- e.g. virtual memory relies on swap area: memory manager dependent on I/O

System call interface

- all system calls come here, causing a trap which switches execution from user mode into protected kernel mode, passing control to one of the kernel components

Processes

- each process runs a single program, starting with 1 thread of control
- **process group**: ancestors, siblings, descendants
- **daemon**: background process e.g. `cron` daemon for scheduling jobs
- created by `fork`
- **pipe**: channel for two processes to communicate, one process can write a stream of bytes for the other to read
  - when a process tries to read an empty pipe, the process is blocked until data is available
- **zombie state**: process exits, and parent has not waited for it. When parent calls `wait` for it, the process terminates

Signals

- **signal**: software interrupt for one process to send signal to another process
- processes catching signals must specify signal-handling procedure
- when a signal arries, control abruptly switches to the handler. After finishing, control returns to the previous location
- processes can only send signals to members of its **process group**
- `sigaction`: syscall for a process to announce signal-handler for a particular signal
- `kill`: syscall for a process to signal another related process
  - uncaught signals kill the recipient
- `alarm`: syscall for a process to be interrupted after a specific time interval with `SIGALRM`
- `pause`: suspends process until next signal arrives

Implementation

- every process has a user part running the user program
- when a thread makes a syscall it traps to kernel mode and begins running in kernel context, with a different memory map, and full access to machine resources
- this is still the same thread, but with more power and its own kernel mode stack and program counter
- kernel represents any execution context (thread, process) as a **task** via `task_struct`,
  - multithreadd process has one task structure for each user level thread
- kernel is multithreaded, having kernel-level threads
  - executing kernel code
  - not associated with any user processes
- `task_struct`s are in memory for each process at all times, stored as a doubly linked list
- there is also a hashmap from PID to the address of the task structure for quick lookup. This uses chaining in case of collisions.

Process Descriptor

- **scheduling parameters**: process priority, CPU time used, time sleeping
- **memory image**: pointers to text, data, stack, page tables
- **signals**: mask indicate which signals are caught/ignored/delivered
- **machine registers**: when a trap occurs, machine registers are stored ere

- **syscall state**: info about the current system call
- **file descriptor table**: table mapping between file descriptor and file's i-node
- **accounting**: keep track of CPU time used by the process
- **kernel stack**: fixed stack, for use by kernel part of process
- **miscellaneous**: process state, event being waited for, PID, parent PID, UID, GUID

Executing `ls`

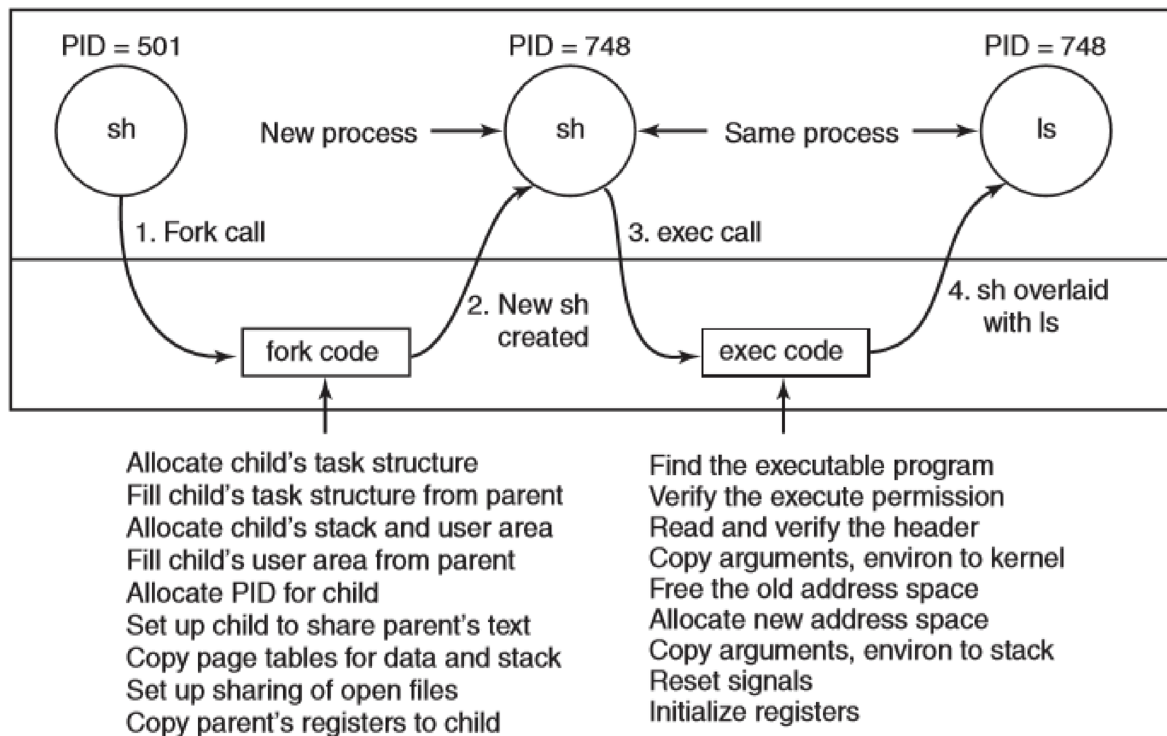

Figure 10-8. The steps in executing the command *ls* typed to the shell.

Figure 3: executing-ls

Kernel Threads and `clone`

- kernel threads in Linux differ from the standard UNIX implementation
- in UNIX, processes are resource containers, and threads units of execution. Processes contain 1+ threads, sharing address space, open files, signal handlers, alarms, …

- `clone`: introduced in Linux in 2000, blurring the distinction between threads and processes, allowing parameters to be specified as process specific or thread specific

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Whether a `clone` call creates a new thread in the current process or in a new process is dependent on `sharing_flags`, which is a bitmap allowing fine-grained control over what gets shared.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Figure 10-9. Bits in the *sharing_flags* bitmap.

Figure 4: sharing-flag-bitmap

- divergence from UNIX, meaning Linux code using `clone` is no longer portable to UNIX
- Linux stores in `task_struct` both the process identifier PID and task identifier TID
- when `clone` creates a new task sharing nothing with the creator, and PID is set to a new value
- otherwise `clone` creates a task receives a new TID but inherits the PID

Scheduling

- As Linux threads are kernel threads, scheduling is based on threads, not processes
- runnable tasks are stored in the **runqueue**, associated with each CPU
- tasks which are not runnable are stored in the **waitqueue**
- 140 priority values:
  - 0: highest priority
  - 139: lowest priority
- 3 classes of threads:
  - **real-time FIFO**: highest priority 0-99, not preemptable except by a newly readied real-time FIFO thread with higher priority
  - **real-time round robin**: highest priority 0-99, preemptable by the clock, having an associated time quantum
  - **timesharing**: lower priority 100-139, conventional threads

- NB these are not actually real-time threads, in terms of a deadline guarantee. The naming is for consistency with standards
- `nice(value)`: syscall to adjust static priority of a thread, where value ranges from -20 to +19

Linux O(1) Scheduler

- historically popular scheduler, but the heuristics were complex and imperfect, producing poor performance for interactive tasks
- constant time to select/enqueue tasks
- the runqueue uses two arrays, active and expired, storing the heads of a linked list, each corresponding to 1 of 140 priority levels
- scheduler selects task from highest-priority list in active array
- after the task's quantum has expired, it is moved to the expired list, possibly with a different priority level
- a task that blocks is placed on the waitqueue until it is ready again, at which point it is enqueued on the active array
- when there are no more tasks in the active array, the pointers are swapped to make the expired array the active array
- higher priority levels are assigned higher quanta to get processes out of the kernel quickly
- interactivity heuristics: dynamic priority is continuously recalculated to reward interactive threads and punish CPU-hogging threads (-5 to +5 penalty)

Completely Fair Scheduler (CFS)

- uses a red-black tree as the runqueue
- tasks are ordered based on amount of CPU time they have spent, `vruntime`, measured in nanoseconds
- left children have had less time on CPU and will be scheduled sooner than right children
- schedule the task which has had the least CPU time (typically left-most node)
- CFS periodically increments `vruntime` of the task, with the effective rate adjusted according to the task's priority: low priority tasks have time pass more quickly. This avoids maintaining separate runqueues for different priorities
- selection of a node: constant time $O(1)$
- insertion of a node: $O(\log n)$

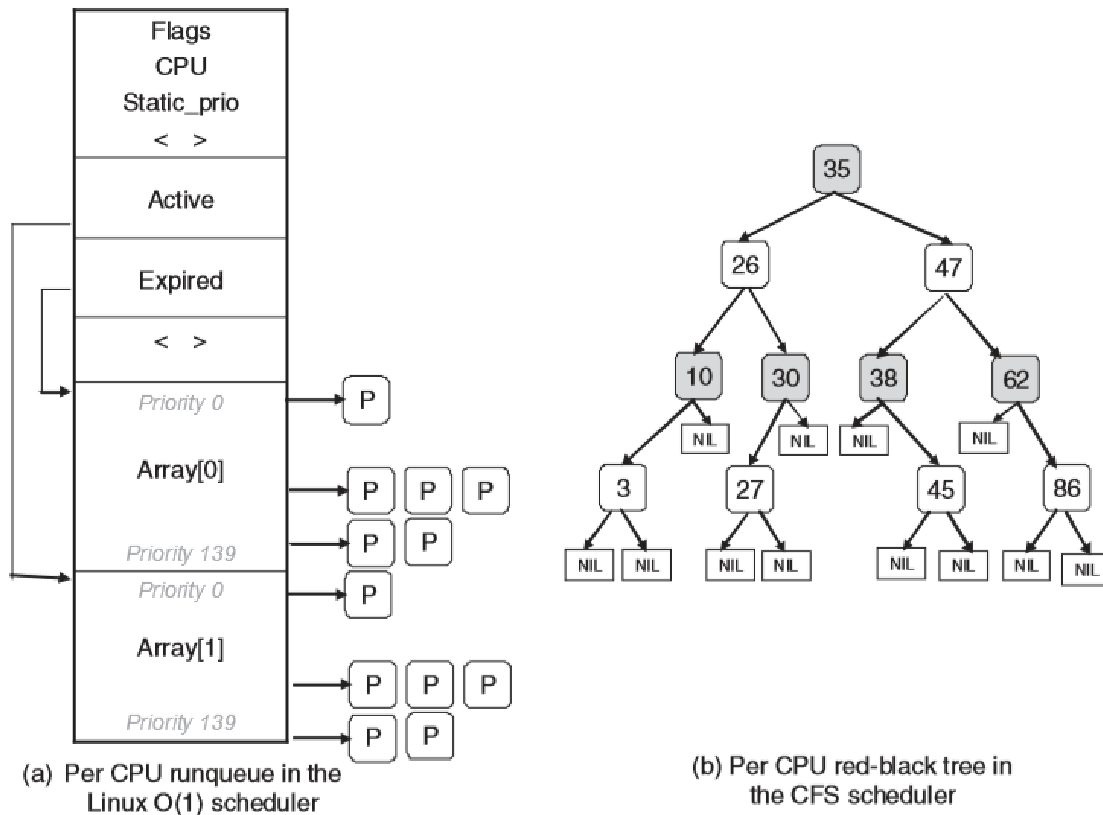

Figure 10-10. Illustration of Linux runqueue data structures for (a) the Linux O(1) scheduler, and (b) the Completely Fair Scheduler.

Figure 5: linux-scheduling

Boot Process

- BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialisation
- Master Boot Record first sector of the boot disk, is read into a fixed memory location and executed
- execution of MBR program loads a standalone `boot` program from the boot device
- `boot` is then run
  - copies itself to a fixed high memory address, freeing low memory for the OS
  - reads root directory of the boot device
  - reads in OS kernel and jumps to it. Kernel is running
- kernel start-up code is written in assembly, so is highly machine dependent

  - setting up kernel stack
  - identify CPU
  - calculate RAM present
  - disable interrupts
  - enable MMU
  - call C-language `main` procedure to start main part of the OS

- kernel data structures are allocated: page cache, page tables
- autoconfiguration: probe for present devices and add them to a table

  - device drivers can be loaded dynamically

- set up process 0 (idle process), set up its stack, and run it

  - initialisation: e.g. program real-time clock
  - mount root file system
  - create `init` (process 1)
  - create `page` daemon (process 2)

- `init` checks if it is single/multiuser:

  - single: `fork` a process executing the shell, and wait for it to exit
  - multi:

    * `fork` a process that executes system initialisation shell script `/etc/rc`,
    * read `/etc/ttys` to list terminals and their properties
    * `fork` a copy of itself for each terminal, then execute `getty`

- if someone tries to login, `getty` executes `/bin/login`, which requests credentials
- if authenticated, `login` replaces itself with the shell

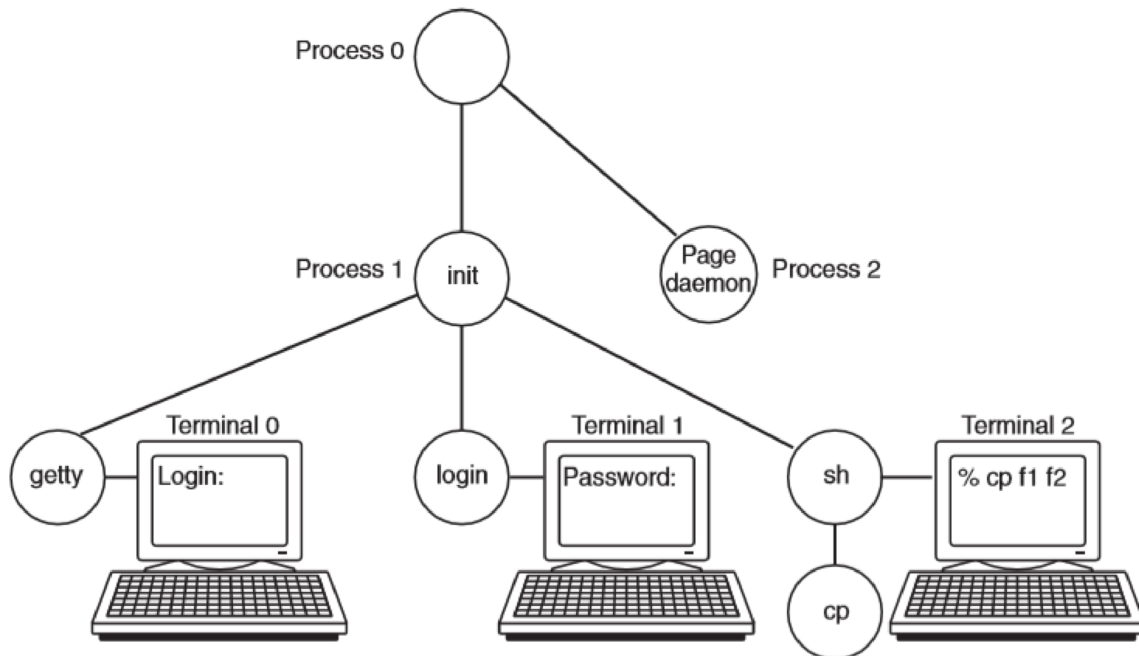

Figure 10-11. The sequence of processes used to boot some Linux systems.

Figure 6: linux-booting

- terminal 0: `getty` is waiting for input
- terminal 1: user has typed login name, so `getty` has overwritten itself with `login`
- terminal 2: successful login has occurred, so `login` has replaced itself with `/bin/sh`, which has printed the prompt. The user has typed `cp f1 f2`, causing the shell to `fork` off a child process executing `cp`. The shell is blocked, waiting for the child to terminate.

Dynamic Loading

- traditional UNIX: static linking of drivers
- dynamic loading allows you to ship a single binary for multiple configurations, with the system automatically loading the drivers it needs, possibly obtaining them over the network
- downside: this creates security vulnerabilities

`init`

Wiki: Init

Memory Management

- each process has an address space with three logical segments: text, data, stack
- **text**: machine instructions that form the program's executable code, read only
- **data**: variables, strings, arrays. Two parts: initialised and uninitialised data
  - initialised data: variables and compiler constants that have an initial value when the program starts. Similar to program text, bit patterns produced by the compiler
  - **Block Started by Symbol (BSS)**: uninitialised data
  - data segment can be increased in size by system call `brk`
    * heavily used by `malloc`
    * **heap** is dynamically allocated memory area
- **stack**: starts at/near top of virtual address space and grows downward
  - programs don't manage the size of the stack explicitly
  - if the stack grows below the bottom of the stack segment, a hardware fault occurs and the OS lowers the bottom of the stack segment by 1 page
  - when a program starts it contains environment variables and command line arguments
- **shared text segments**: allows two processes running the same program can share the same piece of text in physical memory
- some computer hardware allows separate address spaces to be used for text (in one) and data + stack (in the other), doubling the available address space
- **memory-mapped files**: ability to map a file onto a portion of a process' address space so that it can be read/written as if a byte array, making random access much easier
  - shared libraries are accessed by mapping them in this way
  - two or more files can map in the same file at the same time

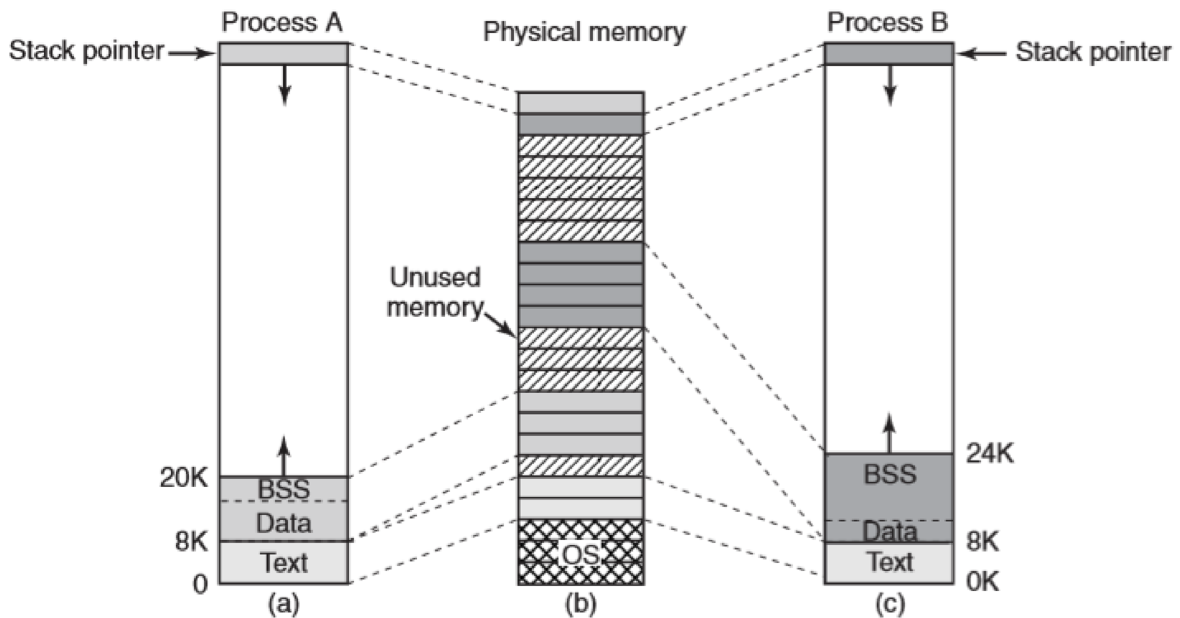

Figure 10-12. (a) Process *A*'s virtual address space. (b) Physical memory. (c) Process *B*'s virtual address space.

Figure 7: virtual-address-space-linux-memmgmt

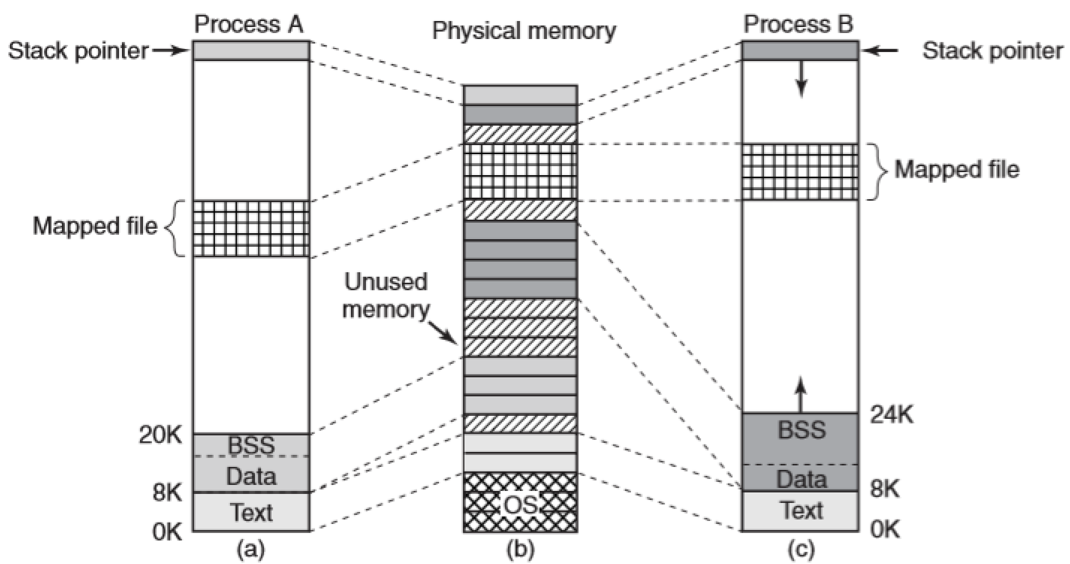

Figure 10-13. Two processes can share a mapped file.

Figure 8: mapped-file-linux

System Calls

- POSIX doesn't specify system calls for memory management as they were considered too system dependent. Instead programs need to use `malloc`, defined in ANSI C.
- most Linux systems have system calls for managing memory:
  - `brk`: change data segment to new address
  - `mmap`: map a file in

Implementation

Duke: linux memory management

- each process on a 32-bit machine has 2 segments of the address space: user and kernel.
- 3GB: private user segment individual to a process, including text, data, and stack
- 1GB: every process maps the same kernel segment into its address space, storing a small stack, kernel data structures, and mappings to directly access physical memory. This eliminates the need for address translation.
- the kernel segment is only accessible in kernel mode. If an attempt to access an address over (and including) `0xC00000000`, this will produce a fault

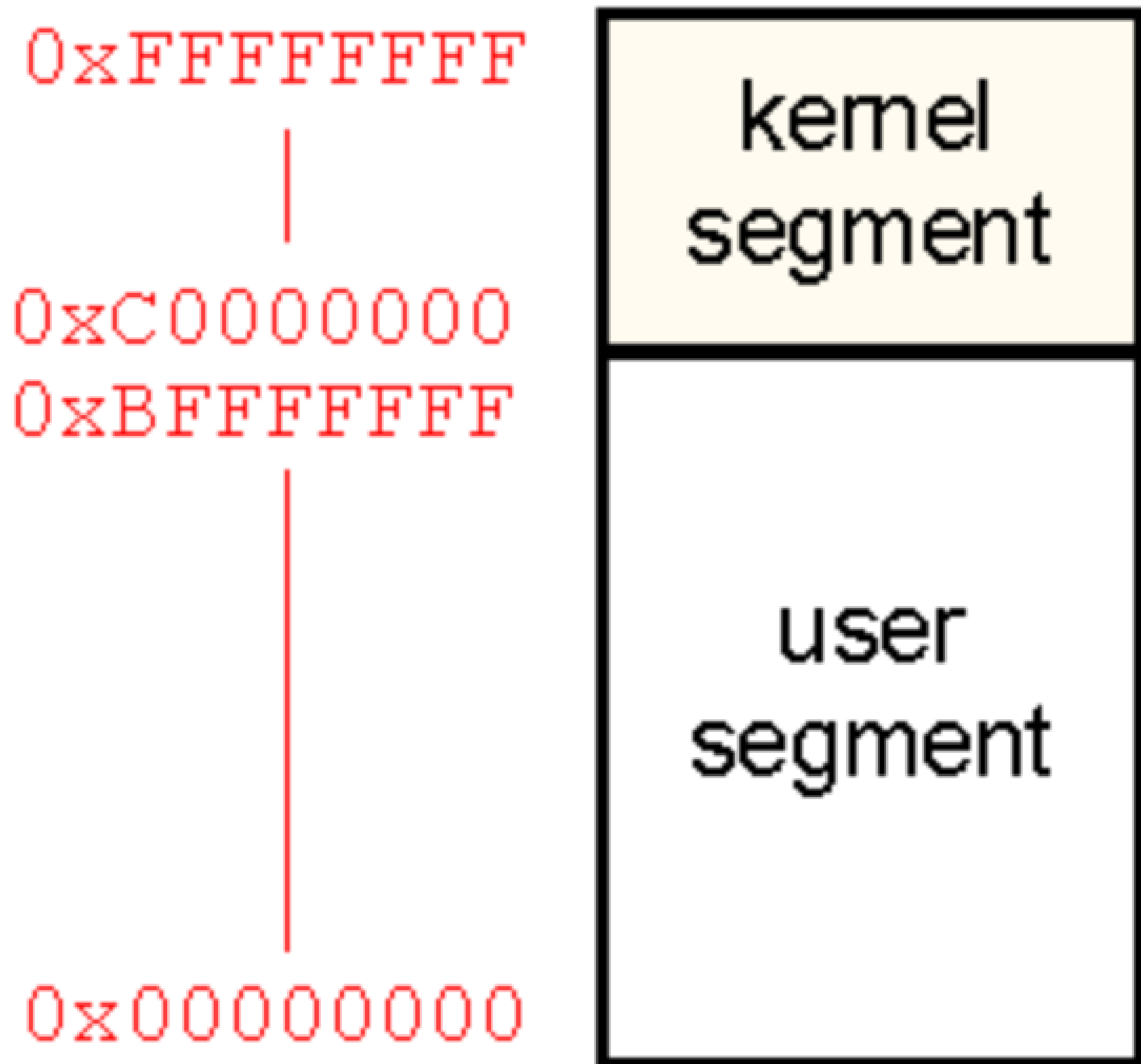

Figure 9: virtual-address-space-linux-user-kernel

- 64-bit x86 machines: only use 48 bits for addressing, for a theoretical limit of 256TB of addressable memory
  - Linux divides this between kernel and user space, of 128TB each
- address space gets created when a process is created, and is overwritten on an `exec`

Physical memory

- Linux uses **nodes** to allow it to support **Non-Uniform Memory Access (NUMA)**, where access time for different memory locations may vary. Physical memory is partitioned into nodes.
- in the case of **Uniform Memory Access**, physical memory is represented under a single node

For each node, Linux distinguishes between **zones** of memory, resulting from differences idiosyncracices of hardware which require them to be handled differently. Memory allocation can then be performed for each zone separately.

- **pinned**: memory that doesn't get pages out
- the kernel and **memory map** are pinned, while the rest of memory is divided into page frames
- a page frame can be a page for: (or else on the free list)
  - text
  - data
  - stack
  - page-table
- **memory map**: map maintained by the kernel representing the usage of physical memory
  - `mem_map`: an array of page descriptors (`page`) for each physical page frame in the system
- **zone descriptor**: one for each zone. It stores:
  - memory utilisation
  - array of free areas, where the ith element identifies the head of a list of page descriptors of blocks of 2^i free pages (used in the buddy scheme)
- page descriptor `page`: contains pointer to address space that it belongs to
  - if free, it has an additional pair of pointers that allow it to form a doublyy linked list with other free page frames

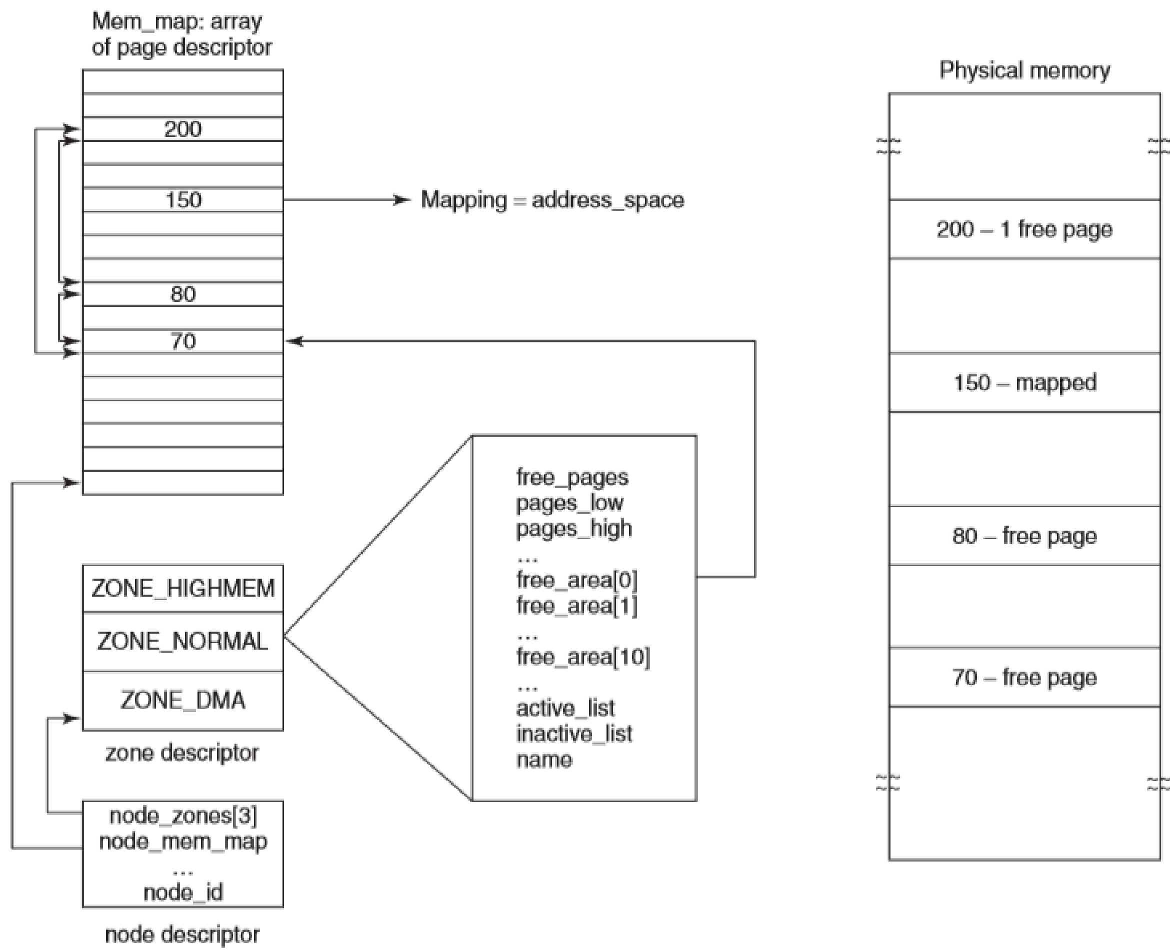

Figure 10-15. Linux main memory representation.

Figure 10: linux-memory-representation

Paging Scheme

- Linux uses a 4-level paging scheme for efficient paging
- virtual address is broken into 5 fields, each used to index the appropriate page of the table

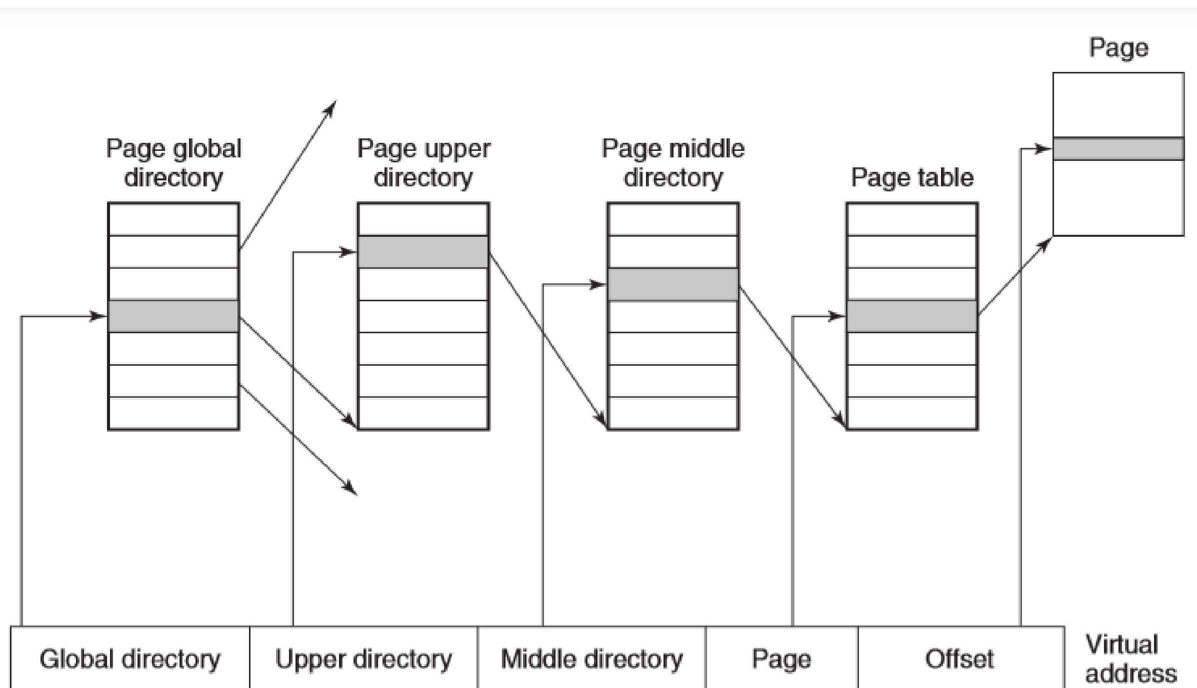

Figure 10-16. Linux uses four-level page tables.

Figure 11: linux-page-table

Memory-Allocation

- **page allocator**: allocates new page frames of physical memory using **buddy algorithm**
  - request for memory (in number of pages) is rounded up to the next power of 2
  - chunk of memory is repeatedly split until the chunk size matches this power of 2, which can then be allocated
- the array of free areas in the zone descriptor is used to store lists of blocks of size 2^i, allowing you to quickly locate such a block by indexing the array
- this results in a lot of internal fragmentation (e.g. 65 page chunk requested would yield a 128-page chunk)
- **slab allocator**: second memory allocation which takes chunks from buddy algorihtm and carves them into smaller slabs for separate management
  - slabs maintain **object caches** which can be used for storing objects frequently created/destroyed by the kernel

Representation of Virtual Address Space

The virtual address space can be broken into **areas** that are runs of consecutive pages sharing protection and paging properties, e.g. text segment, mapped files

- `vm_area_struct`: describes an area, including:
  - protection mode (read/write),
  - pinned/pageable,
  - growth direction (up/down),
  - private/shared between processes,
  - whether it has backing storage on disk: e.g. text segment: uses executable binary as backing storage, memory-mapped file: uses disk file as backing storage. The stack doesn't have backing storage assigned until they need to be paged out
- `mm_struct`: top-level memory descriptor, with information about all virtual-memory areas in an address space, information about different segments, users sharing the address space

There are 2 ways to access of an area of an address space via this top-level memory descriptor:

- linked-list: useful when all areas need to be accessed, or the kernel is trying to find a virtual memory region of a specific size to allocate
- red-black tree: gives fast lookup when a specific virtual memory needs to be accessed

Paging

- early UNIX used **swapper process** to move entire process between memory and disk
- Linux: demand-paged system, no prepaging, no working set
- paging is implemented by both kernel and **page daemon** (process 2), which runs periodically, checking if there are sufficient free memory pages, and if not, it starts to free some
- pages with backing storage are paged to their files on disk
- pages without backing storage are paged to the **swap area** (either paging partition or fixed-length paging file)
- paging to a separation partition is more efficient:
  - no mapping between file blocks/disk blocks is needed
  - physical writes can be of any size, not just file block size
  - page is always written contiguously to disk

Page Frame Reclaiming Algorithm

- idea: keep some pages free so that they can be claimed as needed
  - requires continual replenishment of the pool
- page types
  - *unreclaimable*: may not be paged out, e.g. kernel mode stacks
  - *swappable*: must be written to the swap area before the page can be reclaimed
  - *syncable*: must be written back to the disk if dirty
  - *discardable*: can be immediately reclaimed
- page daemon `kswapd` is started by `init` at boot for each node
- each time it awakens, `kswapd` checks if there are enough free pages available. If free pages falls below a threshold, it initiates PFRA
- for each run, only a target number of pages is reclaimed, typically a maximum of 32, to control I/O pressure
- approach: reclaim easy pages, then harder ones
  - discardable and unreferenced pages can be reclaimed immediately by moving to the zone's free list
  - pages with a backing store that haven't been referenced recently are next
  - then come shared pages that no user seems to be using much
  - pages that are invalid, absent from memory, shared, locked, or being used for Direct Memory Access are skipped
- uses a clock-like algorithm within a category to select old pages for eviction
- pages get categorised by two flags: *active/inactive* and *referenced/not referenced*

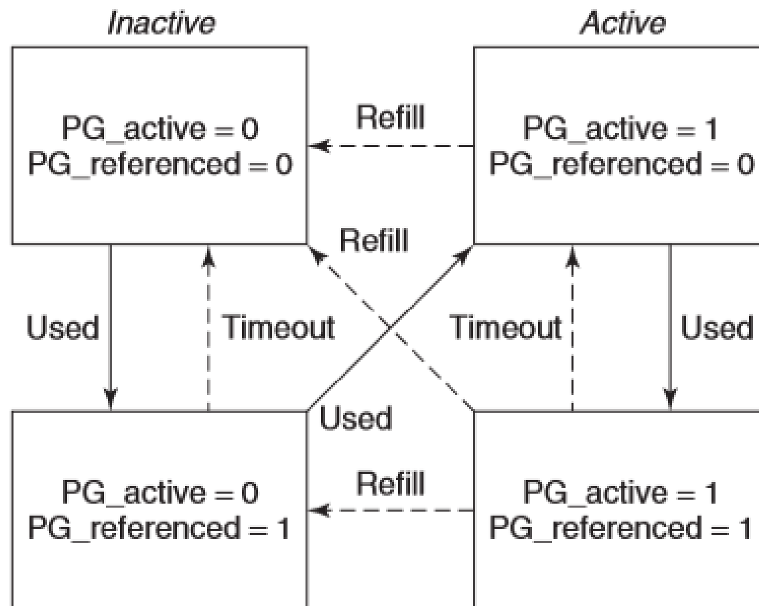

Figure 12: page-states-linux

- when PFRA first scans pages, it clears the reference bits
- on the next scan of pages, if the page has been referenced it is advanced to another state where it is less likely to be reclaimed
- pages on the inactive list which have not been referenced since last inspected are the best eviction candidates
- `pdflush`: a set of background daemons that wake periodically to write very old dirty pages back to disk
  - can also be explicitly awakened when the available memory falls below a threshold to write dirty pages from the page cache back to disk