12. Case Study II UNIX (Linux)

9th ed: Ch. 6, 18

10th ed: Ch. 5, 20

Objectives

- To examine memory management in Linux
- To explore how Linux implements file systems
- To understand how Linux manages I/O devices
- To understand how a shell works

Outline

- Physical memory
- Virtual memory
- File systems
- I/O
- Start of day

Outline

- Physical memory
  - Page allocation
  - Slab allocation
- Virtual memory
- File systems
- I/O
- Start of day

Physical memory management

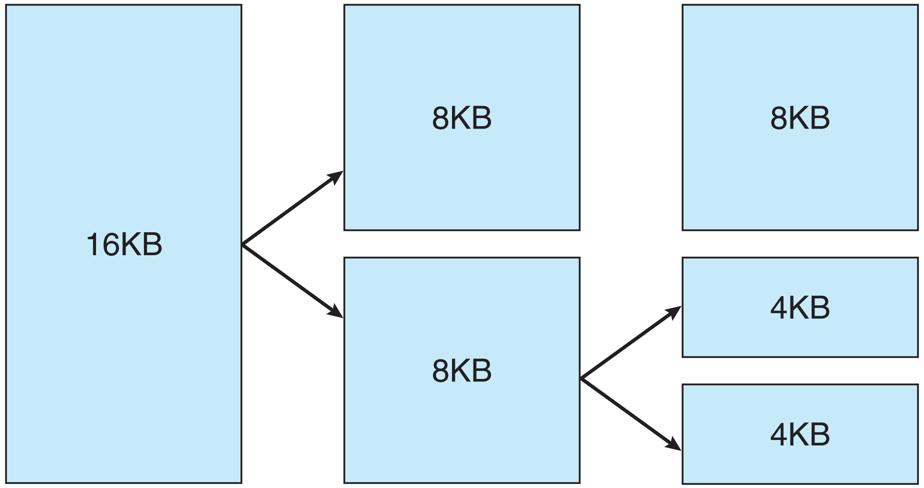
- Deals with allocation/freeing of pages, groups of pages, small blocks of memory
  - Additional mechanisms for handling **virtual memory**, memory mapped into the address space of running processes
- Splits memory into zones based on hardware characteristics
  - DMA, DMA32, NORMAL, HIGHMEM
- Architecture specific; e.g., x86_32
  - Some devices only address lower 16MB, so DMA must take place there
  - HIGHMEM is memory not mapped into kernel space, all else is NORMAL

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

- Other systems have different constraints
  - E.g., some devices can only access first 4GB (even with 64 bit addresses)
  - x86-64 has (small) 16MB DMA zone for legacy devices, and the rest is ZONE_NORMAL

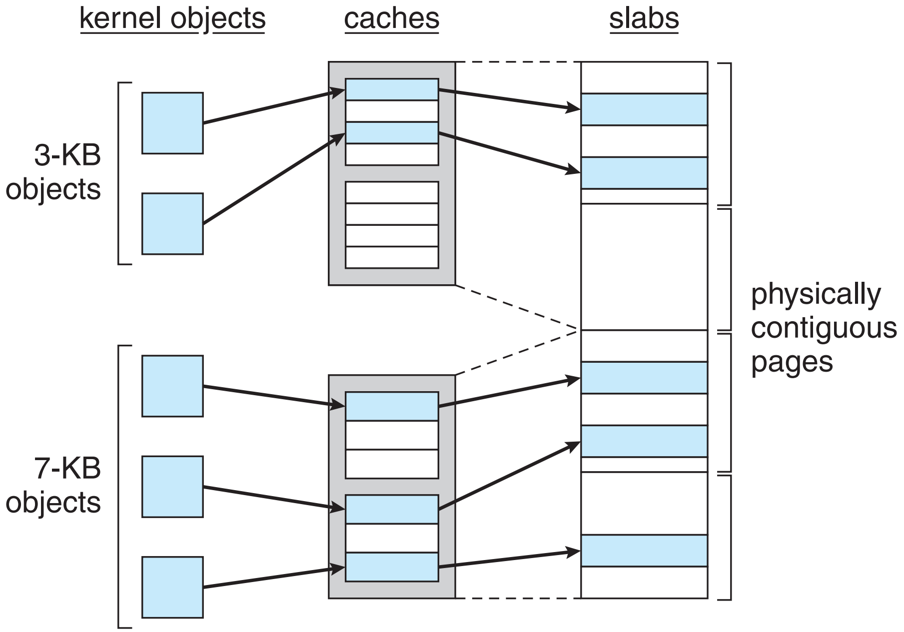
Physical page allocation

- **Page allocator** allocates and frees all physical pages
  - Can allocate ranges of physically-contiguous pages on request
- Uses a **buddy-heap algorithm** to track available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Two allocated partner regions freed together are combined into a larger region
  - If no small free region exists to satisfy a small memory request, subdivide a larger free region into two pieces to satisfy the request



Slab allocation

- Allocation in the kernel occurs either
  - **Statically**, drivers reserve contiguous memory during system boot, or
  - **Dynamically**, via the page allocator
- Uses a **slab allocator** for kernel memory
- Using **page cache**, virtual memory system also manages physical memory
  - Kernel's main cache for files
  - Main mechanism for I/O to block devices
  - Stores entire pages of file contents for local and network file I/O



Outline

- Physical memory
- Virtual memory
  - Creation
  - Running a program
- File systems
- I/O
- Start of day

Virtual memory

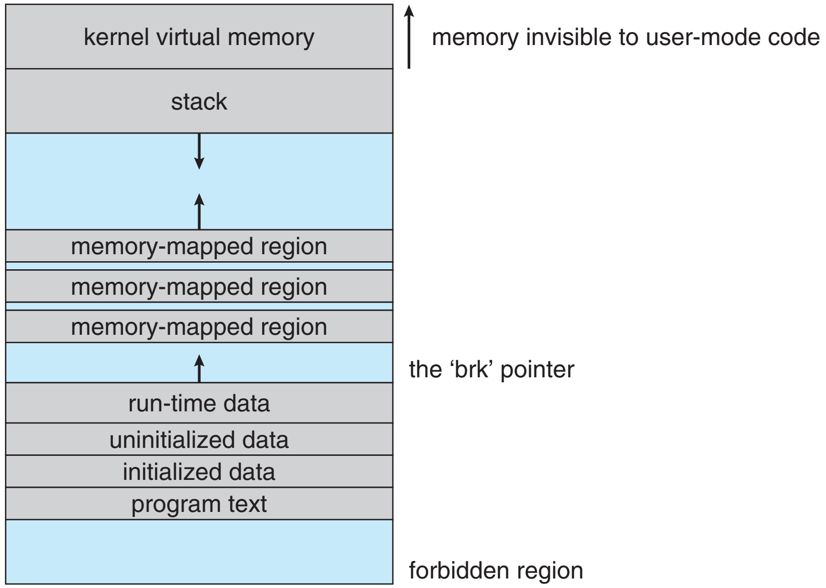
- Virtual memory system maintains each process' address space
  - Creates pages of virtual memory on demand
  - Manages loading of those pages from disk or swapping back out as required
- VM manager maintains two views of a process's address space
  - **Logical view** describes the layout of the address space, a set of non-overlapping regions, each representing a continuous, page-aligned subset of the address space
  - **Physical view** stored in the process' hardware page tables
- Virtual memory regions are characterized by
  - The **backing store**, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (demand-zero memory)
  - The region's **reaction to writes**, either **page sharing** or **copy-on-write**
- **Paging system** uses **page-out policy** to decide which pages to move to and from backing store using the **paging mechanism**

Virtual memory creation

- The kernel creates a new virtual address space for two reasons
- **A process runs a new program via *exec***
  - The existing process is given a new, completely empty virtual-address space
  - Program-loading routines populate the address space with virtual-memory regions
- **A process creates a new process via *fork***
  - New process is given a complete copy of the parent’s virtual address space
  - Kernel copies parent’s VMA descriptors and creates a new set of page tables for the child
  - Then copies parent’s page tables into the child’s, incrementing the reference count of each page covered
  - Thus parent and child address spaces initially share the same physical pages of memory
- Kernel reserves a constant (architecture-dependent) area of two regions
  - **Static region** has page table references to every available physical page to ease logical-physical translation in kernel
  - Remainder is unreserved and PTEs can be pointed to any other area of memory

Running a program

- Kernel has function table for program **loading**
  - Supports multiple binary formats, commonly ELF
- ELF-format program has a header plus several page-aligned sections
  - Pages initially mapped into virtual memory, and then faulted in to physical memory
  - ELF loader reads header and maps sections of the file into separate VM regions
- Unless **statically** linked there will be symbols defined elsewhere
  - Calling dynamic linker stubs trigger mapping of the link library into memory, resolving references
  - Shared libraries typically compiled to **position-independent code** (**PIC**) so can be loaded anywhere



Outline

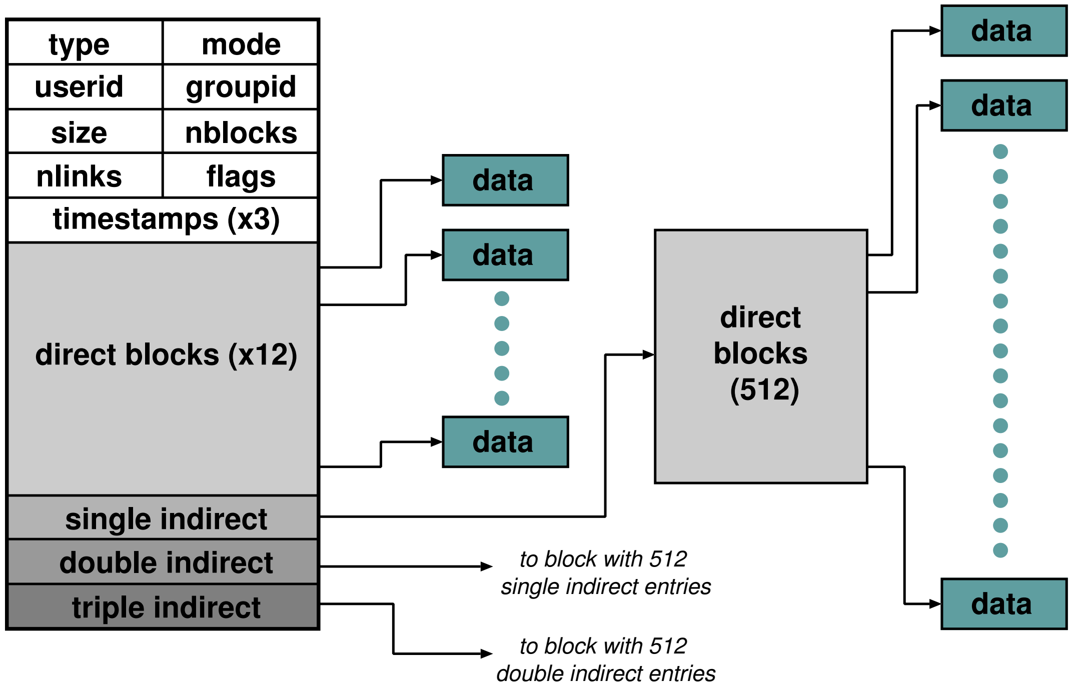
- Physical memory
- Virtual memory
- File systems
    - Implementation
    - Directories and links
    - Access control
- I/O
- Start of day

File systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
  - Devices are represented by special files
  - **proc file system** doesn't store data but computes it on demand using inode number to identify the operation
- Kernel hides details, managing different file systems via the **virtual file system** (**VFS**), an abstraction layer with four components
  - The **inode object** structure represent an individual file
  - The **file object** represents an open file
  - The **superblock object** represents an entire file system
  - A **dentry object** represents an individual directory entry
- Then manipulate those objects via a set of operations on the objects, e.g., for files include
  - int (*__open__) (struct inode *, struct file *);
  - ssize_t (*__read__) (struct file *, char __user *, size_t, loff_t *);
  - ssize_t (*__write__) (struct file *, const char __user *, size_t, loff_t *);
  - int (*__mmap__) (struct file *, struct vm_area_struct *);

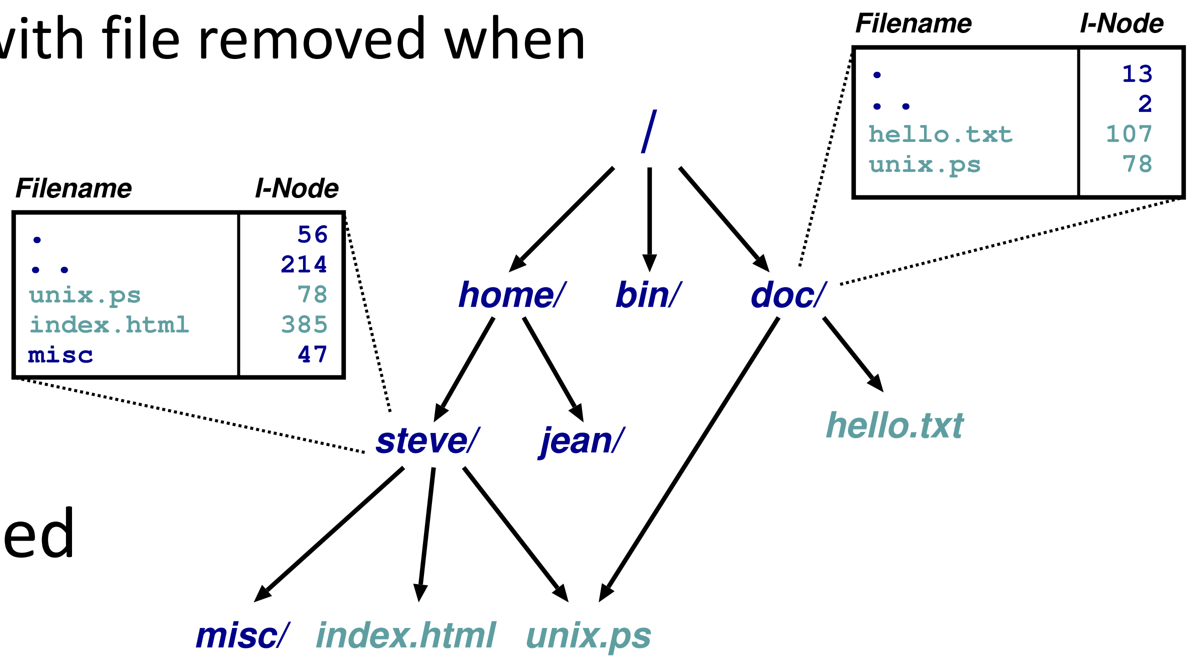
File system implementation

- UNIX file systems use **inodes** (index nodes) as FCBs
  - A **combined scheme**: the inode contains pointers to blocks, and pointers to pointers to blocks, and so on
- Alternatives include **linked schemes** where an index block points to blocks and ends with either a *null* or a pointer to the next index block



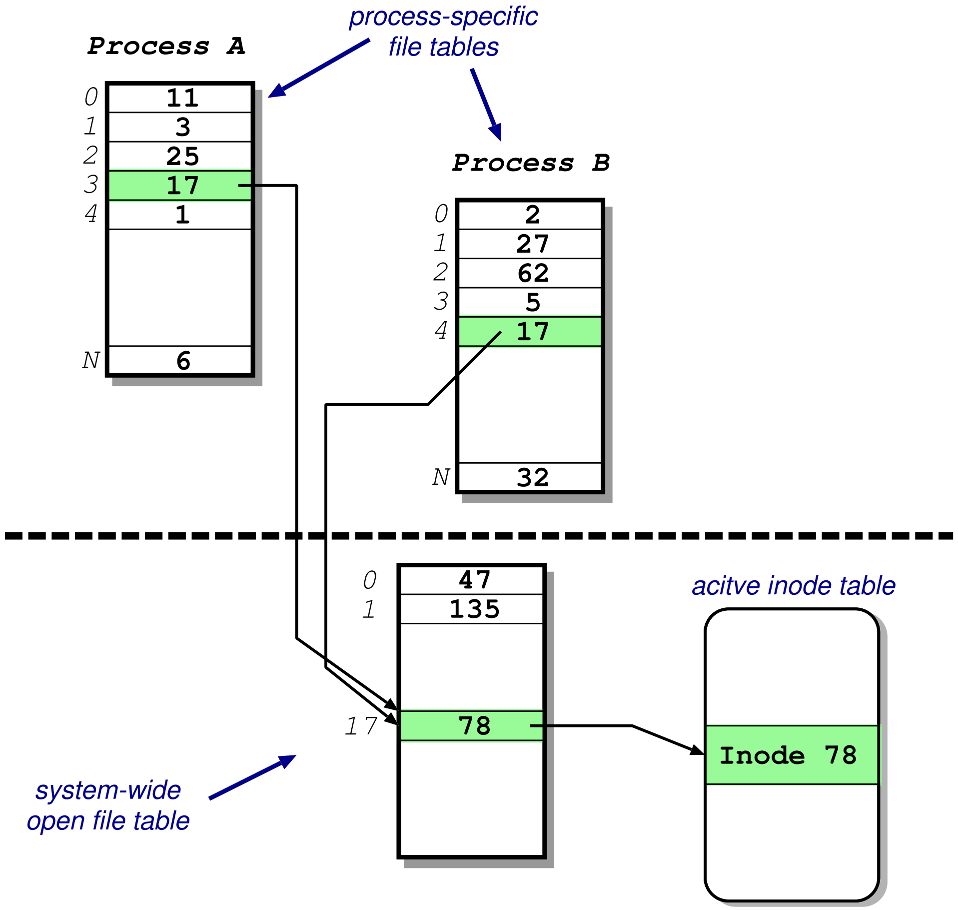
Directories and links

- Directory is just a file, itself pointed to by an inode, mapping **filenames** to **inodes**
- An instance of a file in a directory is a **hardlink**
  - Reference counted in the inode with file removed when reference count becomes zero
  - Directories cannot have more than one hardlink otherwise cycles might be created
- Alternatively, a **softlink** or **symbolic-link** is a normal file containing a filename, interpreted by the filesystem



In-memory tables

- Each process sees files as **file descriptors**
  - Index into a process-specific **open file table**
- Table entries point into a **system-wide open file table**
  - Multiple processes might operate on the same file, including deleting it
- System-wide table entries then point to in-memory **inode table**

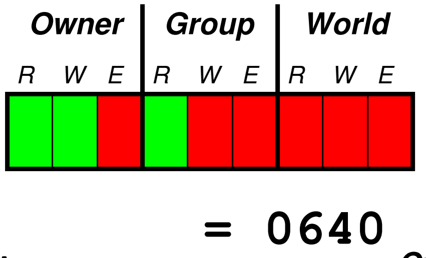


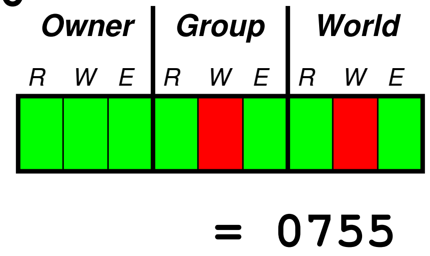
Access control

- Every object uses same mechanism: unique numeric identifiers
  - **User ID** (UID) identifies single user (set of rights)
  - **Group ID** (GID) identifies a group (rights held by one or more users)
- Processes have a single UID but one or more GIDs
  - Process UID matches object UID, then process has **user**/**owner rights**
  - Else if a process GID matches an object GID, then process has **group rights**
  - Else process has **world rights**
- Object has **protection mask** indicating R/W/X for user/group/world
  - Root UID process has automatic rights to everything
- Rights can be passed by forwarding fds down a local network socket
  - E.g., Print server is passed a descriptor for the file to be printed, avoiding the need for it to have rights to read any other of the user's files

File access control

- Access control information held in each inode
  - Three bits for each of owner, group and world
  - For files, read, write execute
  - For directories, read entry, write entry, traverse directory





- Also have *setuid* and *setgid* bits:
  - Normally processes inherit permissions of invoking user
  - *setuid/setgid* allow user to “become” someone else when running a given program
- E.g. an assessment application might have
  - A *sit-exam* application owned by the examiner with permissions 0711 plus *setuid*
  - A *test-scores* file also owned by the examiner but with permissions 0600

Outline

- Physical memory
- Virtual memory
- File systems
- I/O
  - Buffer cache
  - Device types
- Start of day

Input/Output

- Device-oriented file system accesses disk storage via two caches:
  - The **page cache** caches data, unified with the virtual memory system
  - The **buffer cache** caches metadata separately, indexed by physical disk block
- Three classes of device:
  - **Block devices** allow random access to independent, fixed size blocks of data
  - **Character devices** include most other devices, not needing the functionality of regular files
  - **Network devices** are interfaced via the kernel's networking subsystem

Buffer cache

- Maintain copies of some parts of disk in memory for speed
- Reading then involves
  - Locate relevant blocks from inode
  - Check if in buffer cache
  - If not, read from disk into buffer cache memory
  - Return data from buffer cache
- Writing is the same except final step updates the version **in the cache**
  - “Typically” prevents majority (around 85%) of implied disk transfers
  - But at risk of losing data while the update is only in the buffer cache
- Must periodically (30 seconds) flush dirty buffers to disk
  - Can cache metadata too but what problems can that cause?

Device types

- **Block devices** provide the main interface to system's disk devices
  - Block buffer cache acts as a pool of buffers for active I/O and as a cache for completed I/O
  - Request manager handles reading/writing of buffer contents to/from block device driver using **Completely Fair Queueing** (**CFQ**)
- **Character devices** do not offer random access, with driver just passing on request directly
  - Main exception are **terminal devices** where **line discipline** is responsible for interpreting information from device
  - Eg., **tty discipline** glues *stdin*/*stdout* onto terminal data/output streams
- **Network structure** complex with socket interface, protocol drivers, network device drivers
  - Also firewall management, filtering, marking etc

Outline

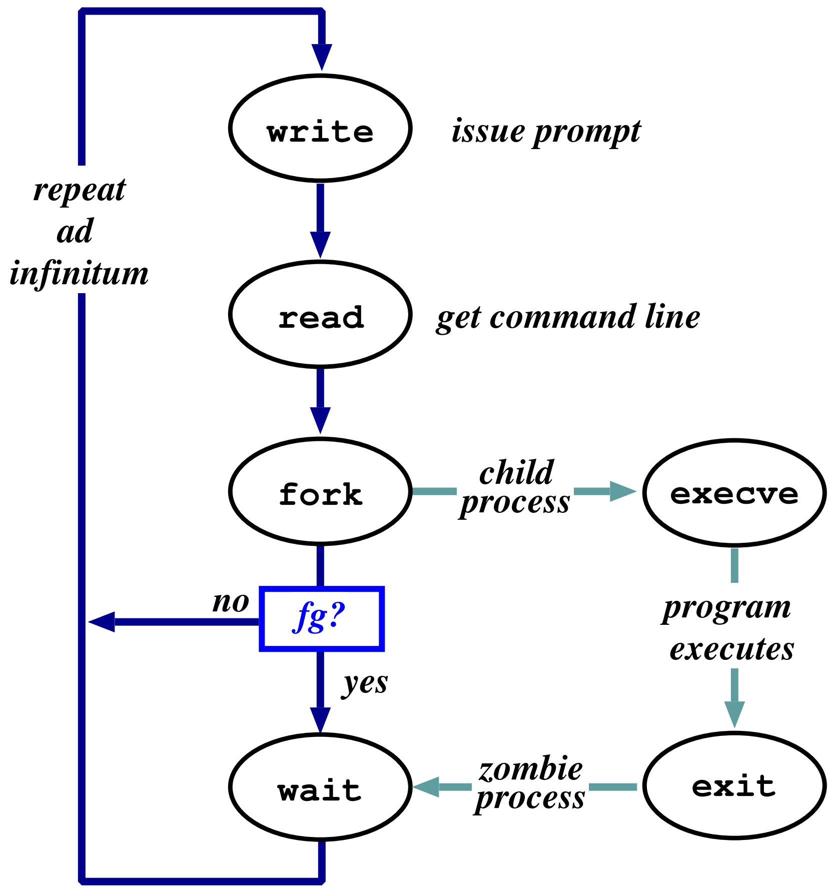
- Physical memory
- Virtual memory
- File systems
- I/O
- Start of day
  - Shell operation
  - Standard I/O

UNIX start of day

- Kernel (*/vmunix*) loaded from disk and executed, mounting root filesystem
  - Bootloader required to read from the disk
  - First process (PID=1), traditionally */etc/init*, is hand-crafted
- Proceeds by reading */etc/inittab* and, for each entry:
  - Opens terminal special file, e.g. */dev/tty0*, duplicates the resulting fd twice, and forks an */etc/tty* process
- Each tty process then:
  - Initialises the terminal, outputs the string **login:** & waits for input
  - On receiving input, *execve /bin/login*
- */bin/login* then
  - Outputs the string **password:** & waits for input
  - On receiving input, hash it and check against entyr in */etc/passwd*
  - If match, set the UID & GID, and *execve* the indicated shell
- When the shell exits, the parent *init* resurrects the */etc/tty* process which goes again

Shell operation

- Just another process – needn't understand commands, just files
  - Using CWD avoids need for fully qualified pathnames
- Command line parsing can be complex
  - Wildcard expansion (**globbing**)
  - Tilde (~) processing
  - Conventionally trailing & backgrounds forked process



Standard I/O

- Every process has three fds on creation:
  - **stdin** from which to read input
  - **stdout** to which output is sent
  - **stderr** to which diagnostics are sent
- Inherited from parent but can be **redirected** to/from a file, e.g.,

  ls >listing.txt ls >&listing.txt sh <commands.sh

- Consider: *ls >temp.txt; wc <temp.txt >results*
  - Pipeline is better, e.g. *ls | wc >results*
- Unix command lines can become very complex e.g., with many filters
  - Redirection can cause some buffering subtleties

Summary

- Physical memory
  - Page allocation
  - Slab allocation
- Virtual memory
  - Creation
  - Running a program
- File systems
  - Implementation
  - Directories and links
  - Access control
- I/O
  - Buffer cache
  - Device types
- Start of day
  - Shell operation
  - Standard I/O