# Computer Systems

# 1 Introduction

## 1.1 Networks

The internet is essentially a network of networks. Communication infrastructure enables distributed applications. Project 2 will likely be a multi-threaded server. A protocol is used to establish connection first, and then the data is transferred from the client to the server. We need to use the port number and the I.P. address of the server for communication.

## 1.2 Operating Systems

The OS provides a much higher level interface. The OS plays referee, preventing people from stepping on each other. The CPU scheduler provides the illusion that multiple programs can run on one CPU at the same time. Virtual memory provides an address that is larger than the size of the physical memory. It also provides a file system, so the programmer doesn't have to think about the sectors on the disk.

### 1.2.1 Process Management

**Exam Questions:** Define processes and multi-programming; managing processes in Unix/Linux; the user-kernel distinction; system calls in Unix/Linux; threads

A process is a program execution; a program is static, a process is dynamic. For example, like cooking, the recipe itself is static and is like a program but the act of cooking itself is dynamic and therefore is like a process. Processes have three segments: the stack, the data and the text (the actual code). The stack is where the variables are and the the heap is where all the mallocs are. Ctrl-Z doesn't kill the process, it just puts it to sleep (it keeps running in the background). Every process that runs has a parent as it had to come from somewhere.
If several processes are to be active at the same time, there has to something that has to keep them in check; enter the **kernel**. Note: the kernel is **not** itself a process.
CPUs have two modes: the user mode and the kernel mode. The program status

word (PSW) gives the current mode. The code running in user mode cannot issue privileged instructions but code running in kernel mode can. The user-kernel distinction is the core of the security in a given OS. To go into the kernel mode from the user, like if we have to print something in C, the trap bit is initiated and then the code enters the kernel mode, and the code then runs in kernel mode. System calls are used to allow users to ask for the kernel to execute privileged instructions or access privileged memory locations. Examples are `open, read, write, close, fork, exec, exit, wait` etc. **Need to know these commands.** There are systems calls for everything from process control to file management, device management, communication etc. `printf` in C is not a system call, it call `write()`, therefore providing that extra level of abstraction. For example, if we call `read()` the program gives the control to the kernel, which does everything and then the controls returns to the program.

`fork()` creates a new process with the exact copy of the parent. If `fork()` returns 0, then the child process was created. Exec* is also a systems call and is used to run other programs, it replaces the address space of the process. Also note that this entire program is non-deterministic, as in we don't know which is going to print out first, the parent or the child. This is because they both are separate processes and the processor may decide to put one ahead of the other depending on the circumstances.

### 1.2.2 Interrupts

When a hardware device needs the attention of the CPU e.g. if it has finished doing its task, it makes an interrupt. When an interrupt occurs, the CPU's hardware takes the values in the program counter and the program status word registers, and saves them in privileged parts of the memory reserved just for this purpose. The interrupt handler must do several tasks:

- save the rest of the status of the current process
- service the interrupt
- restore what it saved
- execute a *return for interrupt* or similar instructions to restore whatever the hardware saved when the interrupt occurred

The *interrupt vector* is an area of memory that stores the program counter, the PSW and on some machines the stack pointer for use when the interrupt occurs, this address is wired into the CPU.

# 2 Process Management

**Exam questions:**

- Difference between user and kernel mode?

- Why have the distinction?
- What are the security mechanisms?
- What are the three main purposes of an OS? Provide an environment for the users to run their programs on hardware in an efficient manner; to allocate separate resources of the computer as needed, the allocation process should be as efficient as possible; as a control program it serves the purpose of supervision of the user programs to see if they are running properly without any errors and management of the operation and control of the system I/O
- What is the purpose of a system call? They allow user-level processes to request services of the OS.
- In Linux, what system calls have to be executed by a command interpreter or shell in order to start a new process? A `fork()` system call is followed by an `exec()` call needs to be performed to start a new process. The `fork()` call clones the currently executing process, while `exec()` call overlays a new process based on a different executable over the calling process.

Remember a program has a program code, global data, heap and stack. What is going on when we use a signal? The basic cycle of execution is Start; fetch next instruction; execute instruction; loop back to fetch next instructions until there are no more left and then finally finish the program. To extend further: whenever an instruction is executed, there is a check to see if there was an interrupt issued.

Any given interrupt has a different Interrupt Priority Level (IPL), and the CPU only services the interrupts that have a higher IPL than the process currently running. The OS executes the interrupt at the IPL of the interrupt handled. Usually we put the clock at the top followed by networks, tapes and then disks. **True interrupts** come from hardware outside the CPU and **Pseudo-interrupts** come from inside the CPU, like a divide-by-zero error. Each type of pseudo-interrupt has its own entry in the interrupt vector. For hardware syscall instructions, examples are TRAP, and INT 2E. A system call instruction causes a **synchronous** except (or *traps*), other examples are divide-by-zero, bus error, page fault etc. Interrupts are **asynchronous** exceptions, examples are timer, disk, ready, network etc. On a system call, exception or interrupt the hardware/OS enters kernel mode with *interrupts disabled*; saves PC, then jumps to the appropriate exception handler in the kernel. A systems call runs in kernel mode, but it *checks for permissions first.*

## 2.1 Threads

**Definition:** a sequential execution stream within the process (sometimes called a "lightweight process"). Threads are the basic unit of CPU utilization - including program counter, register set and stack space.

Some OS's provide functionality for sharing by dividing the notion of a process into two components: a thread and a container for the thread. Thus, a process has one container, but may have more than one thread, and each thread can perform computation (almost) independently of the other threads in the process. *Note: only one thread can run at a time.* An example is a word process or a multi-threaded server which creates a new thread to deal with each of the requests it gets. For any given program, information such as the code, data and the files are shared but each thread has its own stack and register. The data shared by threads is address space and memory - code and data section; contents of memory (global variables, heap); open files; child processes; signal and signal handlers. Threads have their own copy of: program counter, registers, stack (local variables, function call stack), state (running/waiting). Threads can easily communicate with each other, and there is less overhead when compared with making multiple processes. Look at the `pthread` library for creating threads in C. Global variables are shared across threads. *Thread switches can occur at any point.* Thus, another thread could modify shared data at any time; therefore there is a need to *synchronize* threads to avoid such problems.

## 2.2 Process Scheduling

**Exam questions:**

- Process states (and the use of PCB)
- Context switching
- The use of scheduling algorithms
- Explain the performance characteristics of particular algorithms

A process can be in one of three states: **running:** actually using the CPU, **ready:** runnable; temporarily stopped to let another process tun and **blocked** unable to run until some external event happens. Note that a process **cannot** go from being ready to blocked. Saved states of processes are stored in the kernel in the *process table* which has a slot for every process. Some entries are:

- process id
- process state
- user id and other privilege information
- start addresses and sizes of memory areas
- CPU time used and other accounting information
- priority
- working directory

- list of open files and information about them
- space for saved copies of registers (including PC, PSW and SP)

To create a process first memory is allocated for it; the process control block is initialized; the process is inserted into the Ready queue; sometimes running time is estimated; its memory requirements are sometimes analyzed and finally, the its first part is loaded into memory.

When an interrupt occurs and a running thread is set to ready, and the CPU comes back after the interrupt the highest priority thread is restored, sometimes this is not the thread that was running before, if this is the case, then this event is called a *context switch* and the previously running thread was preempted. *Processes and threads are not aware of interrupts or context switches.*

For any scheduling lgorithm it must be fair: every process/thread should get its fair share of the CPU; throughput: should maximize the number of jobs processed per time; response time: should minimize waiting for interactive users; turnaround time: should minimize waiting for batch jobs.

FCFS works well when job times are similar but it is unfair to short jobs. Shortest first has a good turnaround time but it may starve long jobs. R.R. keeps a list of ready threads and allocates a quantum of time to each of them. If a thread is still running at the end of its quantum, it is put back in the queue. If we have a short quantum we have good response time, poor throughput, good for interactive systems; if we have long quantum we have poor response time, good throughput, thus great for batch systems.

A S.R.R. works like R.R. except it doesn't put jobs in the ready queue until demand for the CPU is "acceptably low". Thus, S.R.R. can approximate either F.C.F.S. or R.R.

There is also priority scheduling which assumes that each thread has a priority and it always runs the thread with the highest priority. Note that the process's absolute priority doesn't matter as much as its priority as relative to the processes its being compared with.

# 3 Memory Management

**Exam Questions:**

- Fragmentation
- The mapping between physical and virtual memory
- Paging: page faults, page tables, page replacement algorithms
- Working set

The main functions of a memory manager are: to keep track of allocated and free parts of memory, to allocate memory, to protect memory from unwanted access and to simulate bigger main memory by automatically moving data between the disk and memory. There are different types of memory from smaller & faster to larger & slower: register, L1 cache, L2 cache, main memory, local secondary storage and remote secondary storage. The parts of the disk used to store process information on the disk are called the *swap space* as total size of all processes may exceed total memory size. As processes are added and removed from memory, if there is a smaller process in memory than the previous one, the too-small space becomes unusable and over time a lot of these *holes* start to appear, this problem is called *external fragmentation.* One way to solve this is to move all the processes into one contiguous area in memory, this is called *memory compaction.* This is not done as memory speed is very low as compared with the CPU speed. The program is unaware of the fact that the address it has and the physical address of a memory location are different as the OS adds a base register (an offset) to the addresses generated by the program. Base and limit registers guarantee security if the addresses generated by a program are always checked with the limit register and an exception raised if outside limit; and the base and limit registers are only modifiable in kernel mode.

A program assumes that all its code and data are in main memory when running and the code and data are stored in contiguous locations. These assumption also cause external fragmentation. Virtual memory is applied via **paging** or **segmentation.** Paging relies on the difference between physical and virtual/logical addresses and uses a *memory management unit* (MMU) to do the translation. CPU's typically provide a way to access physical addresses directly without MMU in kernel mode. MMU is usually in the CPU or close to it. The virtual address space is the set of addresses a program on a machine may generate. *Each process has its own virtual address space.*

A paged system divides both physical and virtual addresses into pages. It maps a virtual page to a physical page, also called a **page frame.** When a virtual is moved multiple times, it may be loaded into different page frames. Since the memory allocated to a process is an integral multiple of page size, there is almost always going to be some space wasted, this is called **internal fragmentation.** In a paged systems an address is *pageNumber * pageSize + offset*. The page size in bytes is: 2 raised to the power of (BitsInTheOffset). The information required to do the mapping is recorded in a page table and *each process has its own page table.*

Usually a page table entry has:

- a physical page number

- a valid bit to check whether the virtual page is in memory
- a referenced bit, whether the page was referenced before
- a modified bit
- read, write and execute permission bits

The physical page will only be constructed by the MMU if valid bit is true and the permissions permit memory access. It will then set the referenced bit and also the modified bit if it was a write. If either conditions isn't met is raises a *page fault* exception, to be handled by the kernel.

# 4 Memory Management

**bits in offset = size of page**

If the valid bit is zero and the OS must:

- suspend the process
- free up a page frame if required
- load the virtual page
- cause MMU to map that virtual page
- restart the process at the *same* instruction

Now we need two memory accesses: the PTE and to access the data. To optimize we use **translation lookaside buffer (TLB)** which serves as a cache holding copies of recently accessed PTE's. To avoid overlapping of TLB entries it is cleared at every context switch so that there is no conflict like process 1 and 2 having same virtual addresses pointing to different physical addresses.

A scheme used was splitting the page into two, for code, heap at the start and the stack at the end, but this can't handle dynamic linking and shared memory operations. Thus, a *page table directory* is always in memory when a process runs whose entries point to page table fragments which may be in memory or in disk.

For a split system we must know the physical address of the each half of PT and their lengths. For directory, we want to know the its physical address, its length and fragment length. A *page table base register* in the CPU has the first word of the currently running process. In most computers, the kernel is actually a part of the address space of every process, a virtual address with high bit set is system space and user space, otherwise. There is 1 PT for system space and 1 for each user processes.

The page fault handler must be kept in memory. If the TLB misses the lookup shouldn't go through the MMU in a usual way. If the chosen page to be replaced has been modified (modified bit changed) then the OS must first write the changes to disk before taking it out of the memory. All page replacement algorithms assume that the near future will be the same and the near past. Two techniques are used: **spatial locality:** locations around a recently referenced location are likely to be used; **temporal locality:** if a location has been referenced lately, it will be referenced again.

Page replacement policies: **optimal** which chooses the page whose reference is going to be the furthest in the future, and established the upper bound on how good an algorithm can be; **random** which picks a page at random; established the upper bound on how bad an algorithm will be; in some cases can't do better than this.

The FIFO algorithm is simple but disregards locality and throws out even heavily used pages. The LRU algorithm picks the least recently used page, but it requires us to keep time stamps, which doubles memory accesses, so it isn't used. The LFU keeps a counter for each page and adjusts them at each clock interrupt but a highly used page is left in memory, even if no longer required. The clock algorithm slowly sweeps through memory clearing referenced bits, if a bit is already clear, the page is marked for replacement. A second and minute hand is used in practice for resetting and checking; they are separated by fixed page numbers or milliseconds.

A free page pool can be used by having a list of clean, ready pages which are filled in and then the next page to be removed is found; this almost always improved performance. Paging in a dirty page requires two steps: copy to disk and write new. The page write daemon can find dirty old pages, write them to disk and put them in the pool.

Smaller memory size lead to higher faults. If the processes in memory is too high then we get a lot of page faults and the waiting queue will be longer and CPU will be underutilized; this is called **thrashing**. If the OS knows about this, it should avoid bringing in new processes and take steps to end it.

The principle that a process shouldn't run until the set of pages it needs regular access to are in memory is called **the working set principle.**. Also called the minimum number of pages necessary to get it to start. Whenever a program goes from one "section" to another, we will see a sharp rise in the working set size; we should accommodate for this by having the "normal" page fault rate band wide enough.

### 4.1 Process Creation

Create processes by issuing a system call with the new process and its attributes or issue a call to clone the current process. Fork is good as it allows for environment setting before execution and can use all information from the parent to do this. Fork is bad as the cloning work done is wasted. `vfork()` borrows the address space of the parent instead of creating a new one, but child can screw with parent's address space size and contents. At `fork()` time the OS only copies the PT and sets it read-only and copy-on-write. If some tries to write to read-only page, the COW bit is checked and if set:

- copies the page
- sets the write permission bits and resets the COW bits in both PTE's
- restarts the faulting instruction

## 5 File Systems

**Exam questions:**

- The purpose of a file system? Application in Linux/Unix
- File permissions
- The use of file descriptors
- File allocation methods including FAT and indexed (use of i-node)

The layout is (high-level to low) application programs, logical file system, file-organization module, basic file system, I/O control and devices. Collection of bytes stored on disk is called a file. Unix always views files as *sequence of bytes.* Some file attributes are:

- Name - only info kept in human-readable form
- unique id
- type
- pointer to location
- size
- protection - who can read, write, execute
- time, data and user id - data for protection, security and usage

An inode contains all file info in Unix like: type, owner, group, permissions, time of last access, last modification time, last inode change, size and disk addresses. In Unix a directory is just a file maintained by the kernel. We use B-trees for speed mapping of inode and file names. A Unix filename is just a sequence of names with slashes which is converted to inode number by looking at each component. The kernel caches the pathname translations for speed.

The same inode (file) may be referred to from more than 1 directory entry which then belongs to each of them equally, each reference is called a **link**. `ln` in Linux. Hard links can only point to regular files, not directories as each directory has its own set of inodes. Symbolic link is `ln -s` and is a special kind of file that contains a filename, whenever it's opened, the kernel sees the link kind in the inode and opens the filename instead.

## 5.1 Permissions

For `chmod` u is user; g is group and o is others. For the number 4 is for read, 2 for write and 1 for execute; from LHS of the number we have the owner, group and other. The format is `rwx`, each are set in binary. Each user is part of one or more groups; each group has a unique id and name separate from user id and name. If a user is part of more than one groups, one group is primary, the rest, secondary. User info is in `/etc/passwd` and group info is in `/etc/group`. The OS must know which subjects are authorized to perform what operations on which objects. Subjects are users or processes executing on their behalf; objects are files, areas of memory etc. There is not a full list as it would be too long so OS's keep the info with the objects or the subjects. An application is the **Access control list or ACL** which stores the info with the file itself and the ACL is searched to find the user and his/her permissions.

A process/user can look into a directory if the 'x' is set, checks 'w' to create and delete files. When a mask is created, the unmask number is subtracted from the `chmod` number. The unmask facility is a mechanism and default unmask is a policy. When the `setuid` bit is on the program is called a setuid program and can do commands that the user of that id can do; same for setgid. The `su` command is substitute user and `sudo` is superuser do which just runs the command line program as setuid to root.

A file is an abstract data type with: create, write, read, reposition within file, delete, truncate, open (fi) which searches the directory structure on disk for entry fi and move the content of memory to memory and close (fi) move to the content of entry fi in memory in directory structure on disk. In Unix the input/output stream is referred to by a file descriptor: a small integer. *The kernel uses the file descriptor to index into a table representing the process's open files.* It is impossible to do operations on a file with the kernel's permission. Unix system call for files are `open, read, write, close`: checks if operation permitted and then returns descriptor, read a given number of bytes into a given

buffer and then return bytes read, writes n bytes to given file from buffer and make descriptor unavailable. Files are created in Unix at zero size and write beyond the current size.

The kernel maintain a *current offset*: distance from start of file to next byte to read/write. Access files randomly using `lseek` by changing the current offset. The Unix kernel keep s three main kinds of tables for files:

- open file table: global table containing one entry for each open without a close, and each entry with current offset
- descriptor tables: small per-process table, maps descriptors of that process to open file table entries
- active inode table: global table with information about every active file

The per-process file descriptor table is copied when a new process is created via `fork()`. Every program opened by the shell can count of `stdin, stdout, stderr` descriptors open.

In Unix the drive is divided into partitions, which the kernel uses as a virtual disk. A partition may serve as a component of swap space or hold a file system; which is a tree-structured hierarchy of directories. Unix allows one file system to be mounted on top of a directory is some other file system, the root directory of the mounted file system conceptually replaces the mounted-on directory; they are flagged as such in inodes. Every file has a system file `/etc/fstab/, /etc/vfstab` that describes the configuration of its file systems.

## 5.2 Structure of File Systems

A Unix file system has: the boot block(0) which has bootstrap code, the super block (1) which summary info for the file system, blocks for fixed number of inodes and blocks for file data. There are different file allocation methods: contiguous, linked, FAT, indexed and multi-level indexed allocation. FAT is similar to linked allocation but the linked list are stored in a separate table called DFAT, which speeds up direct access and the FAT can be cached. Now FATXX means there are XX bits for each disk address.

For a two-level table: the inode points to a 2nd level index block and each 2nd level index block points to a data block. The inode contains pointers to the first 10 blocks of a file, and blocks beyond are accessed by a single, double or triple indirection. `ext3` **is the standard for Linux.** Although to the users it appears as a hierarchical tree, actually the kernel manages using an abstraction layer called the *Virtual File System (VFS).* The Linux VFS is object-oriented and is composed of: a set of definitions that define what a file object is allowed

to look like, the inode-object and the file-object structures represent individual files, the file system object represents the entire file system; a layer of software to manipulate those objects.

The device-oriented file systems accesses disk through two caches: data is cached in the page cache, which is unified with the virtual memory system; metadata is cached in the buffer cache, a separate cache indexed by the physical disk block. Linux splits all devices into three class:

- block devices allow random access to completely independent, fixed size data blocks
- character devices include most other devices; they don't need to support the functionality of regular files
- network devices are interfaced via the kernels networking subsystem

The *buffer cache* is the cached disk blocks holding inodes and indirection blocks in main memory of blocks that were just referenced. All disk I/O in Unix is disk to buffer cache. If recent accesses to a file were sequential, Unix will prefetch the next block i.e. start reading it into the buffer cache ahead of time, this is called **read-ahead**. Write system calls copy data from user address space to the buffer cache, blocks affected are marked "delayed write". A daemon process that wakes up every 30 seconds and schedules all such blocks to be written out to disk; the mark is deleted when the write is completed, this is called **write-behind.** Blocks are only deleted or *evicted* when their space is needed to cache some other block. If evicted block is marked as delayed write, it's written to disk first.

A unified buffer ache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O. The advantages of this cache are: reduced number of disk reads required; no need to require read/write requests to be aligned; files with short lifetimes usually require no accesses; if a file is updated several times, only the final version needs to written to disk; scheduling the writing of many blocks at the same time gives more scope for the disk scheduler; no need to lock user pages in memory during I/O transfer. Disadvantages are: disk accesses required an extra pass over the data like disk to buffer and then to user; delayed writes maybe lost if the computer crashes.

A Unix pipe is an inter-process communication channel that can be accessed with read and write operations. Each pipe has a buffer; a read on empty or full pipes cause the process concerned to suspend; and read/write fails on broken pipes. Good thing about pipes is they are seldom written on disk and kept in the memory cache instead. The `pipe` system call creates a pipe and returns two descriptors: for reading and writing to the pipe. For x — y, x is write and y is read. Each sub-process has access to only one end of the pipe; the shell closes the other before the exec.

# 6 Synchronization

**Exam questions:**

- Race conditions
- Mutual exclusion (protecting the critical section)
- Techniques to deal with mutual exclusion: algorithms, semaphores, pthread mutex
- Practical example like producer-consumer problem

If a single program wants to use more than one CPU, it must be divided into more than one thread of execution. The two ways parallel programs are structured are: the threads communicate via shared data structures, usually in main memory; the threads send messages directly to each other, only feasible in distributed systems. Virtual memory systems can map same physical addresses of memory to different virtual addresses to share info.

To avoid problems one process should be updating a freeslot at any time. *Only one process may be executing code in a given critical section at any time and the sections mutually exclude each other.* If a process is in its critical section, no other process can interfere with it called completeness; progress: if no other process is executing the critical section then it is free to be used and starvation: if a process wants a critical section, it is eventually going to get it. Mutual exclusion can be reached with interrupts by raising the CPU's priority level so that all interrupts are blocked.

The **test-and-set** assembly instruction can test and set a value in a single atomic operation, the hardware prevents all accesses to the word/value from other CPUs between the read and write. **Spinlocks** are busy-waiting synchronization methods, since they can waste CPU time, one must ensure that time in them is bounded and short.

## 6.1 Semaphores

A semaphore is an integer value that can be accessed by primitives `wait(s)` and `signal(s)`. The former decrements s if ¿ 0, and invoking process must be delayed otherwise; the latter increments s in an atomic operation. They allow mutual exclusion for a number of processes.