Disclaimer
This is a collection of definitions and simple procedures that were taught in the OS & Networks section of the Systems (SYST) module, initially produced for my own revision needs. This is by no means complete, may contain errors, and you should revise through all materials given in the module. **I could not be held responsible for any deviations of this collection from the original course content.**

However, if you spot any problems in this collection, please do contact me through the email address on the top-right corner, so I can correct them for future readers.

This collection is based extensively on (public) lecture and practical materials produced by module leaders of VIGR, most recently by Dr N. Audsley. You should consult copyright holders and the department before using it for any purposes other than personal academic revisions. Source: 2015-2016 materials from http://www-module.cs.york.ac.uk/syst/os .

Computer System
Four components: Hardware, OS, Application Programs, Users.

Operating System
It is domain specific (embedded, PC, etc.).
It manages hardware and software of the system, provides a stable and consistent environment for execution of applications.
It acts as resource (CPU, memory, etc.) allocator, control program (to prevent errors and improper use) and abstract of the hardware.
Interrupt driven, Direct Memory Access (DMA) allows direct write from device to CPU memory for concurrency.

Multiprogramming
Organize jobs so that CPU is always busy, switch to other tasks when the current task blocks.
Multi-tasking: switch jobs frequently to allow responsive interactions with the user. Chooses a job to execute when several are ready. Swap processes in and out of memory when memory not large enough, and use virtual memory. Prevent individual applications from taking over the whole system or violate security protections.
To implement this, **Dual Mode CPU** is used.

Dual Mode CPU
With hardware support, CPU operates in two modes: privileged Kernel mode (access to all memory and instructions) and normal User mode (access to allocated memory and non-privileged instructions only).
System calls between the OS and applications switch the CPU mode and then run the respective programs. User process finishes, calls system call, OS executes system call (and do other things), and finally returns system call to the calling process to execute again.
User applications are prevented from accessing OS memory space.
If a user application runs for too long, a countdown timer interrupts the computer, CPU mode returns to Kernel and OS takes over. This prevents crashes due to program timeout.

System Calls
OS provides services, accessible via system calls from user applications, through a system call interface.
Example kernel services: program execution, I/O with file or device, file-system manipulation, communications between processes or computers, error detection in hardware and software, resource allocation, security.
Ways of parameter passing: pass parameters in registers (simple, limited length); stored in memory and pass a pointer in register (Linux); program push and OS pop a stack.

OS Structure
Policy-Mechanism Separation: layered approach for OS. Policy defines the functionality, while mechanism implements the functionality. Allows flexibility for policy changes.
Simple: single job, single memory space (MS-DOS).
Monolithic: entire OS operating in kernel space. Good performance but difficult to maintain (Unix). Some implementations of Linux kernel utilise loadable kernel modules to reduce size of kernel and improve extensibility.
Microkernel: migrate functionality from kernel to user space (Mach). Inter-module communications through kernel. Flexible and easy to maintain and extend. However inefficient due to amount of communications.
Hybrid: mostly monolithic, but contain microkernel features (Windows).

Process Management
A **process** is the *activity* of executing a program after loaded in memory.
A process can have different states: new, running, waiting, ready, terminated. It can be created in many ways, but will be always allocated memory and **Process Control Block** (PCB), provided registers and arguments (if any).
Process Control Block contains information for OS process management: state, process ID, program counter, CPU register content, memory and I/O information. It is reclaimed when process is terminated.
When a process is created or returned from a block, it is ready and placed in a ready queue. CPU decides which of the ready processes to run out of the ready queue subject to multi-tasking capacity.
When a process blocks when performing I/O requests, it is placed in waiting state, and returns to ready state when request complete.
When a process is terminated by others or exits (normal/error) on its own, it is in terminated state.
Context Switch: OS switches CPU between processes. When executing process P_0, OS receives an interrupt or system call, then saves P_0 state into PCB_0. It then loads P_1 state from PCB_1, and runs P_1. When P_1 returns, it saves P_1 state into PCB_1, and loads P_0 state from PCB_0, and resumes P_0.

Linux Process Management
When system starts, the first process **init** is created with ID 1 and PCB 1, used as the root of the process tree.
Recursively, a parent can create a child process with **fork**. As an exact copy, the position of execution (program counter) and memory values are inherited by the child in another memory location, but the child has a different ID and PCB.
For the child to then execute a different program, **exec** is done to replace the existing code

with new code. Copying of memory space may not be needed as the child often immediately **exec** after parent's **fork**. **Copy-on-write** semantics is used so that memory space copy is only performed if the child attempts to write to the address space.
The recursive **fork** (and **clone**) calls create a process hierarchy.
Processes finish with **exit**. A parent may (**wait**) or may not wait for its child to finish. It may also use **abort** to terminate its child. Child with no **wait**ing parent is a “zombie”, and child with parent terminated without **wait**ing is an “orphan”.
Requirements for persistence of parent and child process relations are different in different operating systems.

Differentiating Concurrency, Parallelism and Multi-threading
Applications can be constructed from multiple processes. For example, a multiprocess web browser will not hang if a certain web page crashes.
(Process) Concurrency: improve responsiveness (e.g. when part of the process blocked), resource sharing, economy (more efficient than context-switching) and scalability .
Parallelism: Data parallelism (split data across cores performing same operations), Task parallelism (distribute thread across cores, each thread performs unique operation).
Concurrency arranges tasks so that they *could* be performed in parallel. Concurrency provides potential for parallelism, when hardware permits (multiple cores, etc.).
Amdahl’s Law: calculate the speed up of parallelism:

$$speedup \leq \frac{1}{S+\frac{(1-S)}{N}}$$

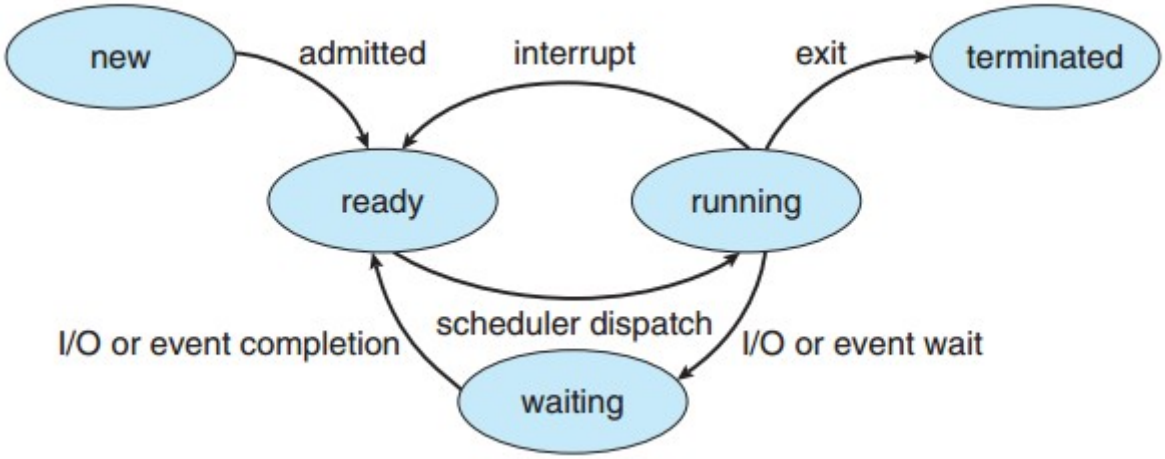
*where **S** is the serial portion percentage of program, **N** is the number of cores.*
Note that serial portion has disproportionate effect on performance gained by adding cores.
Multi-threading: concurrent **thread**s created within *one* process, sharing address space. Thread creation is efficient due to the new thread sharing the same memory space.
OS kernels are generally multi-threaded.

Threads
A process contains a single control path through the code, but can have multiple threads with different thread ID, sharing same code and data section, OS resources, etc.
User Threads: threads implemented by user-level library. Use a thread table within process to keep track of thread state and context. Thread management available without support from the OS, therefore kernel knows nothing about that process's threads.
Advantages: fast and low overhead, application defined management.
Disadvantage: one thread performing system call will block the *entire* process.
Kernel Threads: thread management supported by the OS. Thread table in OS.
Advantages: no single thread blocking problem due to scheduling handled by the OS; parallelism possible for multiple threads on different CPUs.
Disadvantages: less efficient and more overhead.
Threads in Linux: **clone**, behaves similarly to **fork**, but the new child will share instead of duplicate the parent's address space.
In practise, both processes and threads are created by **fork** calling **clone**, distinctions between them are handled with parameters to decide whether parent and child share the

same address space, etc.
In general, after **fork** calls, **clone** + **exec** = child process, **clone** without **exec** = kernel thread.
If a thread calls **fork**, only the calling thread should be copied in kernel threading, but all threads under the process will be copied in user-level threading.

Hybrid Threading
Define maximum number of kernel threads per user process, multiplex several user threads onto several kernel threads, each kernel thread have several user threads taking turns to use the kernel thread. With three possible models:
Many-to-One: essentially user threads approach, with its advantages and disadvantages, used on systems without kernel thread support.
One-to-One: each user-level thread mapped to a kernel thread, essentially kernel threads approach. High concurrency and overhead.
Many-to-Many: OS creates a sufficient number of kernel threads, application specifies how many user threads use a kernel thread, and distribute them unevenly if required.

Process Scheduling
Assignment of the CPU among the ready processes.
Choose one or more (when multi-processor) ready processes, perform context switch, move them to the running state.
Dispatcher: responsible for giving control of the CPU to a process.



An example of Policy-Mechanism separation. Policy decides which ready process to choose, while Mechanism specifies how dispatcher switches context, CPU mode and jump to program start, as well as the dispatch latency (overhead).
Scheduler switches from a running process to another, either done non-preemptively when the running process switches to waiting state or terminates, or done preemptively when the running process is interrupted.
Goal: maximise CPU utilization and process throughput, minimise turnaround time, waiting time and response time.

Scheduling Queue and Schedulers
Scheduling based on the states of processes. Several queues such as ready queue, I/O wait queue are established. OS needs to locate the PCBs of running processes efficiently.
CPU executes a process until it makes an I/O request (I/O queue), runs out of time slice,

forks a child (child executes) or interrupted.
Short-term scheduler: very frequently selects process from ready queue to send to CPU, must be fast, also called CPU scheduler.
Long-term scheduler: selects what new processes to be added to ready queue, manages multiprogramming, not used frequently, also called job scheduler.
Need to balance the amount of I/O-bound (lots of I/O waits and short CPU bursts) processes and CPU-bound (few long CPU bursts) processes scheduled.
Medium-term scheduler used by some OS's to swap in/out partially executed programs. Long-term and medium-term schedulers provide options for short-term schedulers.

Scheduling Policies
Most processes have a sequence of alternating CPU bursts and I/O bursts. And among the CPU bursts, many short ones and a few long ones.
First-Come First-Served (FCFS): schedule processes in the order they arrive in the ready queue, i.e. FIFO queue. Not optimal due to convoy effect (short processes delayed behind long processes).
Shortest-Job-First (SJF): schedule process with shortest length of next CPU burst first. Can be done non-preemptively (always wait for current process to finish) or preemptively (interrupt current process when a shorter process arrives – **Shortest-Remaining-Time-First** (SRTF)). Optimal, but can only estimate the length of each process's next CPU burst. Can be done with length of the previous CPU burst and exponential averaging:
Predicted next burst length $\boldsymbol{p_{n+1}} = \boldsymbol{k}\ \boldsymbol{t_n} + (1 - \boldsymbol{k})\ \boldsymbol{p_n}$, where $\boldsymbol{t_n}$ is the previous CPU burst length, $\boldsymbol{p_n}$ is the prediction for the previous burst and $\boldsymbol{k}$ is a constant weighing between the two.
<u>If burst sizes are unknown</u>:
Priority: associate a priority (integer) with each process, assign CPU to the process with the highest priority.
SJF is a special case (priority = next CPU burst time).
Potential problem of starvation (low priority processes hard to run), requires ageing of processes.
Round Robin (RR): Allocate a small unit of CPU time (time quantum) to each process, and preempty processes running out of this time to the end of the ready queue.
For the *n* processes in the ready queue with time quantum *q*, give each process 1/n of total CPU time in chunks of at most *q* time units, so that no process waits more than (*n* – 1)*q* time units.
If processes finish early, proceed to next one immediately. If no other processes waiting when a process is placed back into the ready queue, it can start running again immediately.
Higher turnaround time than SJF, but better responses. Need to adjust quantum size to have good turnaround time and low overhead. Too large means similar to FCFS, too small means too much overhead.

Multilevel Scheduling
Partition ready queue into separate queues, split CPU time between the queues, and each queue has its own scheduling policy to distribute its share of CPU time between its processes.
Multilevel-feedback-queue scheduler: processes can move between queues due to ageing, etc. Defined by the number of queues, their scheduling algorithms, methods to

determine when to promote/demote a process, and which queue to enter a process in. For example, move a process to another queue if it doesn't finish within a set amount of time.

Thread Scheduling
For kernel threads, the OS has knowledge of processes and threads, and makes scheduling decisions on both processes and threads.
For user threads, the OS can schedule processes but has no knowledge of process threads, so the process has to schedule its threads, and makes its down scheduling policy (hierarchical scheduling).
Process scheduling and thread scheduling can have combined effects. Applications should be allowed to re-order the thread scheduling queues at any time with any policy.

POSIX Threads (pThreads)
Can define scheduling priority and policy (which of the same priority threads to run and how do they share the CPU) for each individual thread.
Make a set of queues, one queue for each priority level, and add threads to the queues of their priority. Pick the ready thread with highest priority to run. Use either RR or FIFO policy to determine how long to run this thread. Real-time threads usually given the highest priority to run without interference.
Two independent scheduling levels for user threads: system level which is managed by OS, and process level which is decided by local thread scheduler.
Two contexts for kernel threads: process context (OS decides which process to run, and then decides which **one** thread of the process to run – treat processes on thread-level), and system context (OS decides which process to run, and schedule all threads of the process in the same block – multi-threaded processes effectively get more CPU time).

Multiprocessor Scheduling
Run multiple threads on multiple CPUs/cores at the same time, need to decide which processes to dispatch and on which CPU to put each thread.
Can use a single ready queue that is shared between all CPUs (requires synchronisation), or per-CPU ready queue (requires deciding which queue to add a process/thread).
Load balancing: keep all CPUs busy with even loads and scheduling overheads. Two approaches: push (a process checks queues periodically and move threads) and pull (CPU steals from other queues when idle). Many systems use both approaches.
Affinity: thread states could be present in CPU cache when rescheduled, scheduling a thread to the same CPU it ran on before can improve performance by reusing cache. This can be done automatically by the OS or on hints from the user.

Linux Process Scheduling
Two ready queues: Active (for new threads, separate queue for each priority level, threads moved to Expired when running out of time quanta); Expired (when Active empty, swapped with Active so all ready threads get a new time quanta).
Higher priority processes get larger time quanta. A bitmap is kept for the active queue, one bit per priority level to track if queue empty on that priority level.
For multiprocessor scheduling, assumes Symmetric Multiprocessor (SMP) architecture, memory shared amongst all CPUs. Per-CPU Active and Expired queues used, and balanced periodically. Each process has a bit mask for affinity that can be changed by processes and preserved over rebalancing.

Inter-Process Communication (IPC)
Processes / threads coordinate their actions by communicating via shared memory or message passing. Synchronisation over shared data to maintain data consistency.
Access to shared data must be atomic (indivisible).
In high-level languages, atomicity is provided by critical section access to shared data (monitors, shared types, etc). All CPU-level instructions are indivisible as interrupts must be between instructions. Each HLL instruction consists of several CPU instructions, therefore race conditions possible. To prevent this, data must be accessed under **mutual exclusion**, which must be implemented at instruction level.
Critical section solution: mutual exclusion (only one process going through a critical section at the same time); progress (must eventually allow waiting processes to enter their critical sections); bounded waiting (limited waiting time for a process to enter critical section after requesting, assuming that processes running at a non-zero speed).
Implementation:
Turns based solution: a *turn* variable used, process blocks until its *turn* to access the critical section, and hand over the *turn* when exiting critical section. Atomic update of the *turn* variable required.
Synchronization hardware helps to implement this mutual exclusion by allowing test and modify the content of a word atomically.
TestAndSet (set the CPU register value to that in target address, and set the target address value to true) instruction can be used to implement mutual exclusion directly or through implementing a mutex. The mutex can then be used to implement language constructs such as semaphores and protected objects.
To prevent a spin-lock when waiting (repeatedly checking the flag, wasting resources), put the process into a waiting queue, and runs this process again to test whether the lock is released when another process writes to the same lock (regardless of what it writes to the lock).

Implementing IPC
Mutex between threads: pThreads can obtain lock and give up lock on a piece of data. They can also attempt to lock an object, do critical section if successful and do something else if unsuccessful (another thread is using the data); useful if multiple locks are needed, but possible deadlock if a process holds one lock and waits for another.
Conditional Synchronisation: shared data is access in mutual exclusion when certain condition holds.
Semaphores: can be used to implement conditional synchronisation. It contains a zero or positive integer value that is changed atomically. **wait** decrements the value when positive, and **signal** increments the value. It is used to indicate a condition.
Mutex is a special form of semaphore where the value can only be 0 or 1.
In POSIX, unnamed semaphores for use within a process and named semaphore for persistent use between processes. Linux semaphores are not POSIX compliant and are system-wise resources managed by the OS in a semaphore table.
Thread conditional synchronisation: requires three elements (shared data, conditional variable and mutual exclusion of accesses to them). Threads atomically block and test condition under mutual exclusion, if condition is false, thread blocks on the variable and mutex is released; if condition is true, it proceeds to critical section. If another thread changes the condition, the blocking thread is waken up and rechecks the condition.
Signals: data-less communications between processes, or between processes and the

OS. OS uses signals to tell application about some event like mouse movement. It functions like a software interrupt, and the process must use a handler when receives them (e.g. Java's *EventListener*).
System-defined set of signals (31) and user-defined signals (>31).
Synchronous Signal: occurs at the same place every time in the program, caused by the program itself.
Asynchronous Signal: occurs at any time from a source other than the process, for example, another process (e.g. SIGKILL), certain events (e.g. SIGINT on CTRL-C) or an alarm clock (e.g. SIGALRM).

Deadlock Handling
Deadlock: A set of blocked processes each holding a resource (mutual exclusion data) and waiting to acquire a resource held by another process in the set.
Occurs when mutual exclusion is applied to a *resource*, that is currently held by a *process*, who needs to hold additional resources currently held by other processes to *proceed*, when no pre-emption of resource is possible and a circular wait exists between these processes.
Resource Allocation Graph is used to represent the relations between resources and requesting processes. Arrows represent requests and resource allocations. If circularities exist in the graph, a deadlock may exists if either: only one instance per resource type, or several instances per resource type but not allocated correctly.
To avoid a deadlock, we either: prevent deadlocks from ever occurring; detect and recovery from deadlocks; or just ignore the problem (used by most OS's).
Prevention: if a process cannot hold all resource it requires, then it must not hold any resource (possible starvation), and impose a total ordering of all resource types so that requests to resources must be made in an increasing order of enumeration (resource hierarchy). It can prevent deadlocks at a cost of low resource utilisation and system throughput.
Avoidance: alternatively each process declare the maximum number of resources of reach type it may need, and dynamically examine resource-allocation state to prevent circular-wait conditions. This ensures that the system remains in a safe state (there exists a safe sequence of all processes so that all processes can get their required resources and complete). Deadlock possible in unsafe state, but no deadlocks in safe state.
Claim edges (as a priori, indicates that a process may request a resource) are added to the resource allocation graph. When a resource is released by a process, its relation with the process returns to a claim edge. Unsafe states can be observed from potential changes to claim edges.
Banker's Algorithm[1]: check if granting a resource request will keep a system in safe state.
If a system does not have deadlock prevention or avoidance, then it must be able to detect and recover from deadlocks. It must maintain a wait-for graph, and periodically invoke an algorithm to search for cycles in the graph ($O(n^2)$). A variation of Banker's Algorithm can be used for this purpose.
Recovery from deadlock can be achieved by terminating all or one of the deadlocked processes to eliminate the cycle in the wait-for graph. The order of termination can be determined by assessing the characteristics of the processes. Alternatively, we can reclaim a held resource, but need to resource and hold process must be returned to some safe

1 Algorithm available at https://en.wikipedia.org/wiki/Banker%27s_algorithm , it is easier than it looks.

state first (rollback), which is sometimes difficult.
Combination of prevention, avoidance and detection can be used to optimally handle deadlocks.

Contiguous Memory Management
Allocate a single contiguous block of physical memory to each process, so that all of a process is in physical memory. Several processes can be within the physical memory, and processes can be swapped in / out of memory.
In physical memory allocation, the main memory is divided into several partitions: low memory partition containing resident OS and interrupt vectors, and one or more user process partitions above the low memory.
Fixed partitions allocation:
Equal-size: equal size of memory for all processes, any process with size less than or equal to this size can be loaded into an available partition. If memory full, OS can swap processes out of partitions to make space. This is inefficient as smaller processes still occupy entire partitions (**internal fragmentation**) and larger processes may not fit in a single partition.
Unequal-size: assign each process the smallest partition size that will fit, and queue for each partition.
Dynamic Partitioning:
Make partitions of variable length and number, and allocate a partition of exactly the required memory size to each process. This will however produce holes in memory (**external fragmentation**), so compaction must be implemented to shift processes so that all free memory is in one block.
Otherwise, to allocate free holes to processes, three approaches can be used:
First-fit: allocate the first large enough hole (regardless of how much it is larger than required). It is the fastest, but it could cause up to half of available memory lost in holes or unused process memory.
Best-fit: allocate the smallest hole that is large enough, requires full search of holes. Slow, but has lowest external fragmentation.
Next-fit: scan holes from the location of previous allocation and choose the next large enough hole. External fragmentation varies, best at providing block of contiguous free memory.
Buddy System: Treat the entire available memory space as a single block, then split it in 2^U pieces, where *U* is a incrementing positive integer. Allocate a block of 2^U pieces to the process when the required memory is between 2^{U-1} and 2^U (i.e. a piece that is just large enough before further splitting), otherwise, increment *U* (split further) and try again. The system maintains a list of holes of size 2^i. However, be careful that a pair of neighbouring 128K block and 64K block *cannot* be merged to allocate 192K memory to a single process.

Logical Addressing for Contiguous Memory
Assign logical addressing to process memory space so that the process does not need to know its physical location in memory to work, and can be easily swapped in and out of physical memory (relocatable).
Binding of start memory address is needed. For compile time-binding, code must be recompiled every time the start location changes. For load-time binding, code is reloaded with start position every time memory location changes. If program must be moved from one memory segment to another (in execution-time binding), hardware support is needed.

Logical address is generated by the CPU, also known as virtual address. It is the same as physical address for compile-time and load-time binding, but different for execution-time binding.
Hardware support (e.g. **Memory-Management Unit** – MMU) is used to translate logical addresses to physical memory addresses, it also prevents a process from accessing memory addresses belonging to other processes.

Memory-Management Unit (MMU) for Contiguous Memory
User program deals with logical addresses, which are then translated by the MMU to be used to access physical memory.
Simple: MMU simply adds an offset (held in a relocation register) to each logical address. To provide memory protection, it also maintains the range of logical addresses assigned to each process in the limit register, and if a user process requests a logical address that is larger than the limit register, an error is triggered and the request is not sent to the relocation register.

Segmentation
Contiguous allocation for all processes is difficult, so we split memory allocation to each process in segments instead.
From a *user view*, a process has a collection of segments: code, static data, heap, stack, etc. They can be placed in different segments in the physical memory space.
A segment table is maintained, each entry of the table represents the **base** and **limit** for each segment. Logical address consists of a tuple of **segment number** and **offset**.

Paging
A process can be broken into pieces that do not all need to be loaded into the main memory at the same time. This provides opportunity for multi-programming.
Paging eliminates the need to allocate partitions of contiguous memory to processes, thus reducing internal and external fragmentation. It is also a key enabling technique for **virtual memory**.
The **physical** memory is divided into small fixed-sized blocks called **frames**. Typically the size is small (e.g. 4KB) to reduce internal fragmentation (still exist due to process size often not a multiple of frame size, but minimal).
The **logical** memory is divided into blocks of same size called **pages**.
We then allocate pages to frames at run-time, therefore contiguous allocation no longer needed due to convenient relocation. Different pages can be spread out into available non-contiguous memory frames.
Logical-to-physical address translation: each page is assigned a page number and a page offset. The page number is used to lookup the base memory address in a page table, and the offset is then added to the base address to produce the physical memory address for the page.
Each logical address is of $n+m$ bits, the LSB m bits are the offset, so page size = frame size = 2^m; the MSB n bits are page number, so 2^n entries in the page table. Each page table entry contains t bits, corresponding to 2^t frames in physical memory. t might be larger or smaller than n depends on the sizes of the logical and physical address space.
To translate a logical address, take the MSB n bits, enquire the page table, find start physical address $k \times 2^m$ (the k^{th} frame in memory), and physical address for the referenced byte is $(k \times 2^m + m)$.

Implementation of page tables: keep page table in main memory. Each data/instruction access requires two memory accesses: page table access and memory content access. To reduce the slow-down caused by this, **Translation Look-aside Buffer** (TLB) hardware is used.
Implementation of memory protection: attach a protection bit to each frame, and a valid/invalid bit to each entry of the page table, which indicates whether an associated page is in the process's logical address space. Prevent access if invalid.
Hierarchical page tables: allow logical address space to be broken into multiple page tables to allow easier addressing in large memory scenarios. A logical address will now include several segments of page numbers to index multiple levels of page tables.
To combine segmentation into paging, the logical address space is broken into a number of segments, and each segment is broken into a number of pages. A logical address is firstly mapped by segmentation unit, then by paging unit. Some segments may be directly accessible with fewer levels of paging to improve access speed.

Virtual Memory
Separation of user logical memory from physical memory.
As only part of a program needs to be in memory for execution, separation allows the logical address space to be much larger than the physical address space. This reduces memory space constraint on programs, aids portability, allows more process to be loaded faster into memory, and provides library sharing between processes.
A **virtual address space** can be provided to programs that is much larger than the physical memory space. Empty space can be provided in the allocated virtual space to allow stack/heap growth and dynamically linked libraries (mapping multiple logical memory spaces to the same physical memory frame).
Demand paging: brings a page into memory only when it is needed, more efficient. When a page is needed, there is a reference of instruction or data address to it, and it can be brought into memory on demand with hardware assistance, from **swap space** on disk.
When a referenced page is not in main memory, then the corresponding page table entry will be set to invalid, trigger a **page fault**.
Page fault: a non-maskable interrupt (requires context switch), prompts the OS to bring the appropriate page into main memory and to detect invalid memory references.
If the memory reference is checked to be invalid, then OS aborts; otherwise the OS will find an empty physical frame, swap the requested page into the frame, update page table and internal OS tables, and restart the instruction that triggered the page fault.
To measure the performance of demand paging,
Effective Access Time (EAT) = (1 – *p*) x *memory access time*
+ *p* x (*page fault overhead* + *swap out time* + *swap in time*)
where *p* is the page fault rate percentage.
Page faults disproportionally affect the system performance.
To improve performance, disk swap space can be loaded rather than loading through file system I/O due to larger allocation chunk size and less management book-keeping. In older BSD Unix, the entire process image is copied to swap space at a process's load time. We could also discard rather than paging out (write to swap) when freeing a frame, but still need to write to swap space if pages are modified in memory but not yet reflected in the file system. Copy-on-write can be deployed to allow parent and child process to initially shared the same pages, and pages are copied on demand when they are modified by either the parent or the child.

When no memory frames are free to be used for a page fault, **page replacement** needs to be done.

Page Replacement
Need to swap out some frames when no free frames are available. Select a victim frame with a **page replacement algorithm**, swap the required page into this frame, and update page and frame tables. Page replacement goal: achieve a low page-fault rate. Need to choose suitable page replacement algorithms:
Belady's Anomaly: sometimes more frames gives more page faults, FIFO particularly suffers from this anomaly.
Optimal approach: evict the page that won't be used for the longest time in future, but it is impossible to predict the future.
First-in-first-out (FIFO): evict the oldest page.
Least recently used (LRU): replace the page that has not been referenced for the longest time, requires additional storage to store this information, large overhead.
Second Chance Algorithm: a form of LRU, use a reference bit for each page, construct a circular list of pages. If a page to be replaced has reference bit 1, then set reference bit to 0 and leave page in memory, and try to replace the next page in circular order; if reference bit is 0, replace that page.
Frame locking: the OS may require some page to remain in physical memory frames, such as OS codme, control structures and I/O buffers, therefore requires frame locking. It marks a frame not to be replaced, requires a lock bit (usually supported by OS as software).

Page Replacement and Allocation of Frames
Each process needs minimum number of pages, which may vary depends on process priority. When moving a frame, several pages may be involved due to indirect reference for the source and the destination.
Global replacement (Linux): a process due to replace a page selects a replacement frame from the set of all frames, and one process can take a frame from another. Problem arises when the page fault behaviour of a process depends on the behaviour of other processes.
Local replacement: each process selects from only its own set of allocated frames. Page fault behaviour is independent for each process, allows more consistent process execution time.
Working Set Model is used to decide how many frames to allocate a process, which uses locality. A working set approximates the locality of a process. It defines a working set window which contains the most recent page references for a process. A page in active use will have been accessed in the working set window and hence kept in the working set; while a page not in active use will have been dropped from the working set sometime after its last reference. The length of the working set window must be approximated with reference bits, which is difficult to implement accurately.
Page-Fault Frequency Scheme (PFF): attempt to equalize the fault rate among all processes, and to have an acceptable system-wide page-fault rate. A process loses a frame when actual rate too low, and gains a frame when actual rate too high.
For a process page fault rate increases as working set changes. Not having sufficient frames allocated will cause a high page-fault rate. Active processes have to steal frames from other active processes, result in repeated page swapping. When total localities size

of running processes exceed the memory size, **thrashing** happens – more time is spent by OS to swap pages than executing useful processes, severely reduces system performance.
when average utilisation is reduced as a result of high page-fault rate, OS utilisation monitoring can cause thrashing by adversely increasing multiprogramming when observing low utilisation.
Global replacement causes thrashing more often due to the ability of processes stealing frames from others, while local replacement limits thrashing to individual processes.

Files
Provide persistent storage of large amounts of data. Files are the fundamental abstraction for I/O, to / from applications.
In relation with previous components, programs and files are stored on disk; program pages read into memory frames, pages swapped into / out from disk; file blocks read into OS pages as an I/O cache; OS copies part of file data to process's page so process can read an amount smaller than a page/block.
From user view: a file is a contiguous logical space, can be of different types, and may not be stored as a contiguous file. Files may have simple structures or complex structures (built with appropriate information inserted into simple structures).
From OS view: files have no structure but are only sequences of machine words or bytes.
OS provides file management to allow applications to manipulate files: basic operations (e.g. create, read), protection, persistence (e.g. backup) and naming, provided in the form of file management functions.
Type: indicates the function of the file. Information provided as extensions in Windows to allow OS to invoke appropriate application for each file type; provided as a magic number at start of file for Linux.
Attributes: name (human-readable), type, location, size, protection, date / time / identification information. Attributes are kept in the directory structure.
Organisation: how files are organised and accessed in physical media, good organisation allows fast access, ease of update, etc. **Sequential** and **direct access** methods possible.
Sequential access: read a file from beginning until reaching the desired position, then rewind to beginning. Slow and difficult to update, used for tapes.
Direct access: based on random access devices such as hard disks, no sequential ordering of the data in the file, used when rapid access required; useful for conventional user files and swapping, more expensive than tapes. Although not all parts of a file can be accessed in uniform time.
Directories: a file that organises files, owned by the OS. It is a *name space* for file names, as all names in a directory need to be unique. It stores information about files, and map between file names and files, and provides user interactions with the files.
Single-level directory for all users, requires all names unique; or multi-level, directory per user, so each user must keep their file names unique.
Tree-structured directories: efficient searching and grouping, no sharing of files. Requires remembering state of “current” directory. Relative and absolute pathnames possible.
Acyclic-graph directories: allows sharing of files and directories. Includes different file names for the same file (aliasing) and multiple path names for the same file. When deleting a file, must delete all references. Need to avoid cycles.
Protection: prevent unauthorised modifications to a file: per-user and per-file access

control. UNIX implementation: bit mask of rights by user, group and others to read, write and execute.

Secondary Storage
File systems reside on secondary storage mediums like disk. It stores programs, data and swapped out pages for memory management. Need to store logical files efficiently on physical disks, allow fast read / write, and map high level file system operations to low level disk operations.
Control structures:
On disk: boot control block, partition control block, directory structure, per-file **File Control Block** (FCB): permissions, dates, owner, group, size, blocks.
In memory: partition table, directory structure, system-wide and per-process open-file table.
To open a file, search directory structure in kernel memory to locate directory structure and file control block on disk.
To read a file, lookup per-process open-file table, then system-wide open-file table, then locate data blocks and file control block on disk.
File space allocation strategies:
Contiguous: each file occupies a set of contiguous blocks on disk. Simple, only need to store location and length, easy random access, but wasteful of space due to fragmentation. Allocation is difficult as contiguous space needs to be found.
Linked: each file is a linked list of disk blocks, each block contains data and pointer to next block. Simple, only need to store starting address, no waste of space, but no random access possible. Allocation is easy.
Indexed: an index block stores all pointers, producing an index table. This allows random access for linked structure, but large tables needed for large files. This overhead can be solved by multi-level indexing. Allocation is easy.
To manage free space, and reuse space from deleted files, a bit vector is deployed. Each bit that is 1 means the indexed block is free, 0 means occupied. Allows quick locating of free space. Used in swap file systems.
Alternatively, to save overhead space, use a linked list, and maintain a pointer to the first free block. Read overhead involved in traversing, but this is uncommon in operations. However, harder to get contiguous free space easily.

File Systems and Sectors
A file is stored as a number of equal-sized **blocks**. Block size is usually the same as the size of a hard drive disk **sector**.
A **file system** is used by the OS to map files and file blocks to disk sectors. It remembers where on the disk it has mapped the file. Disk also contains information about what is on the disk.
Multiple file systems may reside on separate partitions on the disk. This information is stored as a **partition table** on the disk, at a standard location (e.g. sector 0) to be looked for by the OS at boot.
Each file system has a sector (or several sectors) describing the file system, at standard locations (e.g. after the partition table). This is also called the *superblock*.

Linux file system
Provides simple file operations, e.g. create, open, mount, stat, lseek and truncate. It uses

a tree hierarchy, all files have distinct absolute paths. However, a cycle is possible with symbolic links.
It uses a multi-level indexing scheme (**i-node**). Each i-node contains attributes and file block address. Other than mode type field, no distinction between files and directories.
i-node content: i-node stores the first few disk addresses for the file, which may be all data for a small file. For larger files, i-node stores links to further i-nodes which contain disk addresses. This is a flexible system, but complex and expensive to store.
On disk level, each i-node stores 13 block pointers. First 10 are direct pointers to 512B blocks of file data, then single, double and triple indirect pointers.
Directory: a flat file of fixed-size entries. Each entry in a directory contains a file name and an i-node number. That i-node contains information about type, size, time, ownership and what disk blocks are contained in this i-node. When provided a file path, OS walks through a chain of i-nodes to locate the requested file.
Symbolic links: reference single files and directories via multiple different paths.
Logic file system: the user view of the file system, arrange many physical file systems into a convenient form for user to give single name space. Able to amount physical file systems to directories. Mount table is kept by the OS. The initial file system is the root partition, which is mounted at "/". Identified as root partition in the boot process.
Ext 2/3/4 File System: general purpose file systems for Linux, with POSIX i-node scheme. Mostly static allocation of i-nodes to disk sectors. Use bitmaps to keep track of used / unused disk sectors. Attempts to keep related information together when allocating disk blocks.
Ext 3/4 journalling: attempts to maintain consistency of files stored in both disk and main memory cache with log-based transaction.
Virtual File Systems (VFS): provides uniform access to all file systems, including special files. VFS is used for logical file systems, which provides the same API to be used for different types of file systems. OS file commands are mapped to VFS interface operations, and then performed on local file systems. Contains cache for previous loaded directory entries, file i-nodes and contents. VFS also abstracts I/O devices and their device drivers.

Application I/O Interface
OS provides a common interface based on files for different hardware devices, that may vary in many different ways (e.g. access mode, speed, read / write). Hardware devices are controlled in same manner as files (e.g. create, write).
Blocking I/O: process suspended until I/O completed, easy to use, but insufficient for some needs.
Process reading a file from disk: I/O system determines which disk, translates name to device representation, physically reads data from disk into buffer, makes data available to requesting process, and returns control to the process.
Non-blocking I/O: I/O call returns as much as available. Often buffered.
Asynchronous I/O: process runs while I/O executes, difficult to use. I/O must signal process when it's completed.

Virtualisation
Abstract hardware of a single computer into several different execution environments. Create several **virtual machines** (VM) on which OS's and applications can run.
Components:
Host: underlying hardware system.

Hypervisor: creates and runs virtual machines by providing interface that is identical (except paravirtualization) to the host. Also known as virtual machine manager (VMM).
Guest: process provided with virtual copy of the host, usually an OS.
Benefits: protect hardware host system from VM malfunctions, and protect virtual machines from each other; able to easily clone, freeze, suspend and resume VM's; can also run different OS's on the same hardware, useful for consolidation.
Challenge: it is not safe for guest system kernels to run in kernel mode of the host CPU, but protection to guest systems requires kernel mode. Options: paravirtualization or hardware support (VT-x, etc).
Trap-and-emulate: switch between *virtual user* and *virtual kernel* modes by trapping privileged instructions made in user mode, then check whether it is a valid instruction made by guest, if valid then execute the privileged instruction, finally return control to guest in user mode. This makes guest privileged instructions slower, but can be improved by hardware support in CPU.
Some instructions are not safe to be executed by guest OS, hence are translated into safe alternatives and executed by the hypervisor.
Page tables: each guest has its own page tables, so nested page tables are needed to for the guest to map their virtual page tables to physical tables, and for the hypervisor to map these physical tables to the host page table.
Hypervisor types:
Type 0: hardware-based solutions, VM creation and management via firmware. Guest OS not aware of virtualisation.
Type 1: OS-like software built to provide virtualisation, running in kernel mode, contains hardware device drivers. Also general purpose OS that provides standard functionality and hypervisors, treat guest OS's just like processes, but with special handling of privileged instructions. Guests generally not awake of virtualisation .Example: VMware ESX and KVM on Linux.
Type 2: software on standard OS's that provide hypervisor features. No change to OS required, CPU not aware of virtualisation. Worse performance due to usually not using hardware support. Example: Parallels Desktop.
Paravirtualization: modified guest OS to work with hypervisor for performance. Able to achieve good performance on older machines that do not provide hardware virtualisation support.
Other types include programming-environment (e.g. Oracle Java), emulators and application environment (e.g. BSD Jails).
CPU scheduling: when not dedicated to guests, each physical CPU usually runs several virtual CPUs of several guests. CPU has to be overcommitted. Hypervisor must schedule CPU time for virtual CPUs. The scheduling can be weighted to favour certain guests.
CPU scheduling would cause guests not getting all CPU cycles they expect, which may cause response time and clock problems. Some hypervisors provide additional features to coordinate time and other issues.
Live migration: move running guests between systems without interruption. This involves two hypervisors establish connection, creating a copy of guest on the target hypervisor, and copy memory and data of guests over. Copy read-only memory pages first, and then read-write pages – which requires repeated copying to get a clean copy. When copying finishes clean, guest can be switched over very quickly to the target hypervisor.

Networks

Internet: channel of information exchange between connected computers. It is a hierarchy of networks.

Intranet: portion of internet, managed separately. Security policies can be enforced on boundaries of intranets.

Different protocols connect different networks together. Enabling efficient communication, resource sharing and high reliability systems.

Connectivity: **nodes** connected by physical mediums called **links**. Usually links can transfer data in both directions (*duplex mode*). Styles include point-to-point and multiple-access.

Components:

Host: nodes that uses a network.

Switch: nodes inside the network that are attached to multiple point-to-point links.

Router/**gateway**: a special node, works similar to a switch, that connects two or more networks.

Address: unique identification of the recipient node for a message.

Route: how can switches and routers forward a message to its destination based on its address.

Sending modes: unicast (one-one), multicast (one-many), broadcast (one-all). Networks must support addresses of all three.

The same network link is usually shared by multiple hosts by **multiplexing**, which combines multiple data streams to send through the same link between two nodes.

Multiplexing division: frequency-division (transmit each flow at a different frequency); synchronous time-division (divide time in equal-sized quanta and distribute to flows in a round-robin fashion).

Disadvantage for synchronous time-division: hosts that do not need to transmit still occupies quanta, and resizing quanta difficult.

Switching methods: circuit switching (break message down into packets, and send packets of the same message down the same circuit, more reliable but can waste bandwidth); packet switching (break message down into packets, and packets of the same message can go different routes, less reliable but less wasteful).

Errors: packet-level (packet lost due to bit error or switch congestion, difficult to distinguish between delay and lost); node/link level (hardware or software problem, can eventually be corrected, packet-switched network mat be able to route around failed links).

Network performance:

Bandwidth: how many bits per second transmitted through the link, depends on physical and software capabilities and network load. Maximum bandwidth in practise: *throughput*.

Latency: **processing** delay (examining packets) + **queuing** delay (at switches) + time needed to **transmit** a packet (packet length / bandwidth) + speed-of-light **propagation** delay (distance / speed). Usually round-trip-time (RTT) for messages more important.

Network Protocols

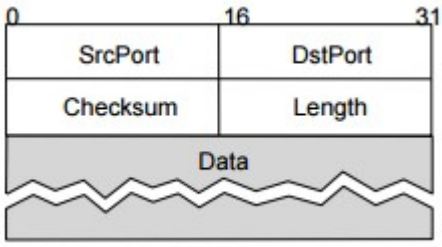
Protocols regulate how computers pass messages, specify sequence and format of messages exchanged. They deal with difference in hardware, transparency, scheduling, security, fault tolerance, etc.

Each protocol contains several layers to reduce design complexity, and improve portability and support for change. Each layer offers services required by higher levels, and shields higher levels from implementing lower level details. Each layer has an associated

protocols, which has two interfaces: service interface, called by the level above to perform operations on this protocol; and peer-to-peer interface to exchange messages with peer at the same level.
Data & control pass from higher to lower layers, across network, then up through layers on other machine. Each layer will alter or pad/unpad the message before sending it to the next layer.
Protocol design issues: identification of messages, data transfer modes (simple, half-duplex, full-duplex), error control (detection and correction), message order preservation and preventing fast sender from swamping slow receiver.
Connection-Oriented Service: establish connection then send data in order. Good for file transfer, authentication, voice, etc. More complex for multi-hop switching. Example: TCP.
Connectionless Service: pass all data independently, transmit before route is established, message may not arrive in order. Good for email, client-server etc, but cannot maintain timeliness. Example: UDP.
7-layer OSI Reference Model: standardised interconnection model. Application, Presentation (plus security), Session (with error handling), Transport, Network, Data link, Physical. Each lower layer adds a header in front of the higher layer message.
TCP Reference Model: simplified OSI. Layer 7: Application; no presentation and session layers, handled in application; Layer 4: Transport; Layer 3: Internet; Layer 2+1: Host-to-network.
Example services:
Application: TELNET, FTP, SMTP, DNS.
Transport: TCP (**Transmission Control Protocol**), UDP (**User Datagram Protocol**).
Internet: IP (**Internet Protocol**).
Host-to-network: ARPANET, LAN.
Comparison: different number of layers; OSI supports connectionless and connection-oriented communication in network layer and connection-oriented in transport layer, whereas TCP/IP supports connectionless in network layers, and both in transport layer.
TCP/IP model is practically designed only for TCP/IP protocol, and does not distinguish between physical and data-link layers.
Middleware can be added to OSI to replace session and presentation layers, and added to TCP/IP to the same position to provide security and fault tolerance, etc.
In computer systems, Application layer runs in user process, which calls Socket API to reach Transport and Network layers, which run in the OS. The OS interacts with network adapter through CPU and memory to reach Host-to-network layer.

Transport Layer
Internet layer provides best-effort service. Transport layer provides assurance of stability, which may involve sending duplicate copies of a message, dropping and re-ordering messages, delay sending and size limiting. Stability means guaranteed message delivery in the same order, with at most one copy, and supports arbitrary message length and receiver flow control.

User Datagram Protocol (UDP)
Connectionless transport (Layer 4) protocol, unreliable & unordered datagram.
Unreliable process-to-process level service extended from host-to-host service. No flow control, and checksum is optional.



End points are identified by ports, which are only interpreted on a host. Transport layer routes each incoming message to correct port. This allows multiplexing amongst applications for same service.
Compared to simple IP on Internet layer, UDP provides port interface, which is useful for many higher level applications such as name servers and time servers. It makes transport service-oriented and removes need for sender to know much about receiver / network topology. Routers use IP, have knowledge of topology and connections, while hosts using TCP do not know for transparency reasons.

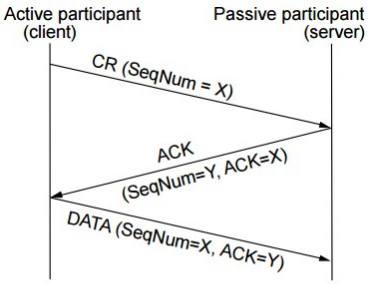
Transmission Control Protocol (TCP)
Connection-oriented service protocol, with flow control (prevent sender from overrunning receiver) and congestion control (prevent sender from overrunning network). Full duplex and point-to-point. It uses ports like UDP, and provides reliable byte-stream (TCP sends bytes written by application in segments, which are read in bytes by another application).
Requirements: explicit connection establishment and termination (connect many different hosts), adaptive time-out mechanism (different RTTs), handle very old packets (network delays), accommodate different node capacities, and handle network congestions.
Segment format:

Bits	0-15	16-31	32-63	64-95
Name	SrcPort	DstPort	SequenceNum	Acknowledgement
Usage	Source Port	Destination Port	Sequence Number	Acknowledgement Field

Bits	96-99	100-102	106-111	112-127
Name	HdrLen	0	Flags	AdvertisedWindow
Usage	Size of header	Zeros Reserved	ACK = acknowledgement; SYN = synchronise seq. nos.; FIN = no more data from sender; etc.	Window size for flow control.

Bits	128-143	144-159	160-191	192-beyond
Name	Checksum	UrgPtr	Options	Data
Usage	Checksum of header and data	Urgency	Variables	Variable length data, requires TCP segment including header to fit in data payload of IP datagram.

Connection establishment: a three-way handshake. A server listens for client connection requests. The client issues **Connection Request** (CR), with which port it wishes to connect to; if the port is waited by the server, server sends **ACK**, indicating next data expected; the client then acknowledges connection, and synchronises sequence numbers in first data sent out.



Notes:
1. *ACK messages have ACK bit set in TCP segment header flags*
2. *ACK messages have own Seqnum and are also acknowledged*

Flow Control: A sliding window is a bound on un-ACKed segments. Based on the knowledge of which data has been sent and acknowledged, as well as the buffer sizes of both sides, the sender and the receiver each creates a window stretching from last data acknowledged (X) to X+ send/receive buffer size. The receiver will inform the sender of its available window. The larger the window, the more flexible the connection is,

but also greater the risk of failures remain undetected.

On sender's side: assign sequence number to each frame. Maintain three variables:
- **Send Window Size** (**SWS**): maximum number of unacknowledged frames sender can transmit.
- **Last Acknowledgement Received** (**LAR**): sequence number, advanced when an ACK arrives.
- **Last Frame Sent** (**LFS**): sequence number.

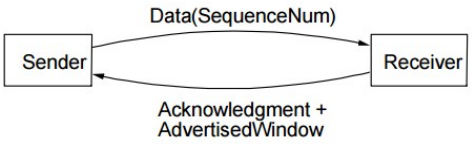
Invariant: **LFS** – **LAR** <= **SWS**, retransmit frame if ACK not received within a given time.
Buffer maintained up to SWS number of frames. Transmission failure detection time is therefore bounded.

On receiver's side: to support out of order receive and drop out of date messages, need to maintain three variables:
- **Receive Window Size** (**RWS**).
- **Largest Frame Acceptable** (**LFA**): how many messages make up the largest allowed frame.
- **Last Frame Received** (**LFR**): last frame received but not delivered.

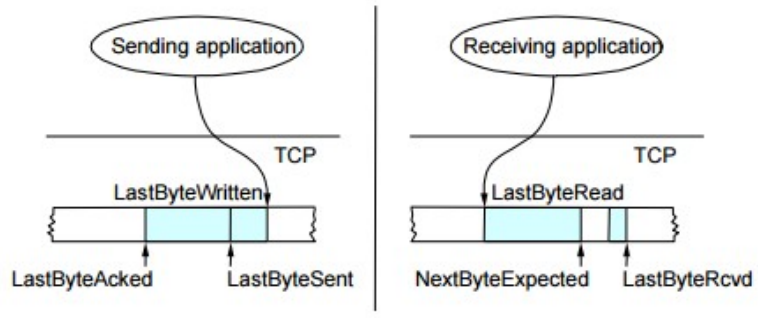
Invariant: **LFA** – **LFR** <= **RWS**.
When a frame of **SeqNum** arrives from sender:accept if LFR < **SeqNum** <= LFA, drop frame otherwise.
ACKs are sent in cumulative order: if later frames arrive before expected earlier frames, always wait until receiving earlier packets then acknowledge them in order – unless SeqNum of the frames outside the window, then these too-far-into-future frames will be rejected.
Buffer maintained up to RWS frames.

Error handling:
If RWS = 1, all frames after an error are discarded, and the sender must resend the last N frames – N depending on transport delay.
If RWS > 1, frames after an error are buffered, and sender needs to selectively repeat missed frame.
The greater transmission delay and likely errors, the larger the buffer needed.

Implementation of flow control in TCP:
Acknowledgement message uses the ACK bit, and indicates the acknowledged segment number in Acknowledgement field. In the same message, a new AdvertisedWindow can be sent. AdvertisedWindow equals how many more bytes can be sent than those already received / acknowledged.



Sender side:
LBA <= LBS, LBS <= LBW.
Buffer size: MaxSendBuffer.
LBS – LBA <= AdvertisedWindow
EffectiveWindow = AdvertisedWindow – (LBS – LBA)
LBW – LBA <= MaxSendBuffer.



Block when (LBW – LBA) + *proposed send size* > MaxSendBuffer. Persist when AdvertisedWindow = 0. Always ACK arriving data.
Receiver side:
LBRead < NBE, NBE <= LBRcvd + 1.
Buffer size: MaxRcvBuffer.
LBRcvd – LBRead <= MaxRcvBuffer
AdvertisedWindow = MaxRcvBuffer - (LBRcvd – LBRead).

Wrap Around Protection: limited number of bits available for sequence number. Wrapping around could cause repetitions of SeqNum in short periods of time for fast networks. Either extend length of sequence number or advertise wrap around events.
Utilisation: window sizes are crucial to network utilisation. Small window sizes cannot handle long delays plus high bandwidth, and small sequence number length could cause wrap around to happen frequently, causing problems.
Retransmission: need a timer for the sender to retransmit unacknowledged messages after time-out. Timer needs to be adaptive to suit changing network conditions.
Examples:
Weighted (moving) average: measure sample RTT for each segment, compute weighted average with constants based on sample and previous estimate, and set time-out twice of the current average.
Jacobson-Karels Algorithm: use the mean deviation (differential) of actual RTTs from the estimated RTT, and set time-out value based on variance. Algorithm is only as good as granularity of clock.
Karn-Partridge Algorithm: multiply time-out by constant after each retransmission (progressive incrementation), when early ACK occurs, use Jacobson-Karels to set a new time-out.

Intradomain Routing
Forwarding: the router forwards a packet arriving on input link to an appropriate output link.
Routing: The Network Layer determines the path taken by a packet from a sender to a receiver. Algorithms used for this process are *routing algorithms*. Routing is needed for forwarding packets in a connectionless network, and for establishing virtual circuits in a connection-oriented network. Routers need to construct their forwarding tables through routing algorithms.
Because most protocols are only designed for up to hundreds of nodes (interior gateway protocols for **intradomain** routing), a hierarchical routing structure based on domains is required to perform **interdomain** routing.
Static algorithms (calculated once) are insufficient due to them not dealing with node/link failures and changes of nodes/links. Dynamic and distributed algorithms are therefore required (however difficult to construct).
Flooding: send packet through all possible routes until received by destination, simple and no topology knowledge required, but inefficient and could cause packets to loop in cyclic structures. Can use TTL reduction at each hop to mitigate the loop problem.
Source routing: send packet with a complete list of nodes predefined. Does not require network support, but must have topology knowledge. Used when end user needs control of route.
Forwarding table routing: each router has a forwarding table of next-hops for each

possible destination. Constructed from a **spanning tree graph** that reaches all nodes with no loop, with edges indicating costs. Each packet needs to contain its destination.
Modern routing algorithms consider latency, bandwidth and current load of network, in addition to routing with fewest hops.
Distance-Vector Routing Algorithm: each node stores an array (a vector) containing the currently believed distances to all other nodes (initially ∞ except immediate neighbours). Vectors are distributed to a node's immediate neighbours, until information in the table stabilises as the table converges.
Link-State Routing Algorithm: **reliable flooding** used to propagate a node's knowledge of how to reach its directly connected neighbours to all nodes, then each node can have total knowledge of the network's topology and calculate shortest routes by using Dijkstra's Algorithm. It stabilises quickly, does not generate much traffic and reacts quickly to changes, but does not scale well in large networks due to amount of information required.
Link-state Packets (**LSP**): transmitted in reliable flooding, contains the ID of the LSP creator node along with a list of its immediate neighbours and the cost to reach them, as well as a sequence number and a time-to-live field.
Reliable flooding: the originator of a packet send out LSP through all directly connected links, and each node will forward the received LSP to all links, except for the link receiving the LSP. Use acknowledgements and retransmissions; consult the sequence number to not forward old LSPs; remove packets running out of TTL; and periodically update link-state information.
The Forward Search Algorithm: implementation of Dijkstra's Algorithm. For routing table, construct two lists: Tentative and Confirmed, with entries (Destination, Cost, NextHop). Initialise the Confirmed list with an entry for “myself”. Then for each neighbour of each node, calculate cost. If that neighbour is not on either list, add (Neighbour, Cost, NextHop) to Tentative; if that neighbour is on Tentative and costs less than the current cost, replace the current entry. If Tentative list is empty then stop; otherwise pick an entry from Tentative with lowest cost to move to Confirmed, and repeat for next node in the table.

Interdomain Routing
Open Shortest Path First Protocol (OSPF): link-state routing protocol that adds some features to the basic link-state algorithm, including authentication, additional hierarchy and load balancing.
OSPF Packets: contain link state OSPF messages, including a “hello” message, and messages for requesting, sending and acknowledging the receipt of link-state messages. Contains one for more **link-state advertisements** (LSA) that provide information about how to reach networks.
Link-state advertisements: routers advertise one or more of their directly-connected networks, or the cost of reaching other directly-connected routers.
Broadcast routing: the Network Layer provides a service of delivering a packet sent from a source node to all other node in the network.
It is inefficient for the sender to simply send a separate copy of a packet to each node, therefore controlled flooding (with list of already seen sequence numbers) is used. **Reverse Path Forwarding** (RPF) is also deployed so that a packet is only flooded if it arrives on the link corresponding to the shortest path to flood source.
The spanning tree for broadcast routing is built from a centred-based algorithm (construct the tree from centre node – rendezvous point).

Multicast routing: enables a single source node to send a copy of a packet to a subset of all nodes on the network. Used for pushing software updates and streaming. Need to identify receivers correctly from a group-shared spanning tree (subset of full routing spanning tree) or a source-based tree (RPF).
Network Layer provides **Internet Group Management Protocol** (**IGMP**) and **Multicast Routing Protocol** for multicast routing.
Network services should be independent of router technology. The Transport Layer should be shielded from routing, and a uniform numbering plan should be provided to Transport Layer.

Packet Switching
Build indirectly connect networks to provide broader network coverage with **switches**. Switches (on Network Layer) take packets arriving at some input port and forward them to the appropriate output port (towards its destination).
Challenges: find the right route, contention (packets arrive too fast), congestion (so fast that not enough buffer for packets arriving, need to discard packets).
Switches interconnect several links to form large networks, creating a star topology with good scalability, composability and large geographic scope.
Identifying switch path:
Assuming that nodes are identified with globally unique addresses (such as IPv4), and each port of the switch has a unique number, possible approaches:
Connectionless approach: each packet contains the full destination path, and use a forwarding table (destination-switch_port) to determine the path. All packets can be immediately forwarded and packets can be routed around failed switches or links, but the sending host cannot know whether the destination is able to receive, and packets may arrive out of order.
Connection-oriented approach: use **virtual circuits** (VC) from established source-destination connections. Either **permanent** (static) virtual circuits (PVC) set by administrator, or **switched** virtual circuits (SVC) set by sender appropriately signalling the network.
Establishing PVC: each switch stores incoming interface (port), **virtual circuit identifier** (VCI) from incoming and for outgoing packets, and outgoing interface in a **virtual circuit table**. VCIs are not global, but only for a give link, and VCI values must be unique with each link.
Establishing SVC: The signalling host will send a setup message into the network, including the address of the receiver. The setup message will travel through the network along a certain route, and each switch enroute will create a new entry in its VC table, including the incoming port of the message, the outgoing port, and an incoming VCI generated by the switch itself. When the setup message reaches the destination, the receiver will send an acknowledgement to the last switch, which contains the VCI for the last hop, picked by the receiving host. And then each switch on the returning trip will send a similar acknowledgement to the previous node on the route, until an acknowledgement is received by the signalling host.
Evaluation of Connection-oriented approach: Each packet only need to include a VCI that is unique to each link, instead of the full receiver address, reducing overhead. Established VCs provides assurances of route, handshake and sufficient resource. However, at least 1 RTT of delay before any data can be sent, link failure could invalidate the established route, and need to recycle VCs no longer needed.

Dealing with congestion: not a big problem for the connection-oriented approach, as a VC may not be allowed to establish if switches enroute are congested. Connectionless approach cannot avoid congestion, and must provide congestion recovery.

Networked Storage

Virtual file system is connected to all storage devices. It provides the system with the same interface for both local and remote file systems. The virtual file system can be networked to access remote storage devices.

A local user can mount networked file systems as exported by the server in all or parts. Users do not share name spaces as files have different names from different parts of the system. Name resolution for remote path is now iterative (resolved partly locally, the rest resolved remotely).

For security reasons, networked file system cannot export directories it has mounted, and the client must explicitly mount such a directory from the server originally hosting it (no multi-level mounting).

Requirements: transparency (same API as local file systems), efficiency and speed, reliability in network failures and scalability.

Middleware: provides a transparent layer between applications and transport protocols.

Remote Procedure Call (RPC): higher level of abstraction than sockets. Client program calls a procedure in another server, can be either synchronous (client blocks until reply) or asynchronous. Parameters are passed by value.

Dispatcher: used in RPC servers and clients to map incoming calls or replies to relevant procedures. Each procedure in interface has a number or name, which is used by the dispatcher to map calls or replies.

Threading issues: single-threaded client must make RPC calls serially, each call blocks 2 * network delay + processing time; multi-threaded client can make RPC calls to different servers concurrently; multi-threaded server can serve different requests to different threads concurrently, which improves responsiveness.

Failure recovery:

Server: unique ID for each callable procedure. For each RPC, which contains an ID, the ID can be used to identify failed server process. New ID created when procedures restart, and calls to stale IDs are aborted by the server.

Client: client may fail after making a call but before receiving response. The server must roll back or do nothing.

Failure call semantics:

Best Effort: send request only once, don't know why if request times out; unreliable network may still duplicate calls; state of server not reliable.

At-Least-Once: client retries up to n times, and if successful, procedure executed at least once depending on failure modes.

At-Most-Once: guarantees remote procedure executed no more than once (but may be only partially complete). Server must track request identifiers and discard duplicates, and buffer and retransmit replies until acknowledged by client.

Transactional: guarantees procedure executed once or not at all (no partial completion). Server must provide atomic transactions for each RPC between consistent states.

Example: NFS (collection of protocols, based on RPC, one-copy semantics).

NFS accesses file system via VFS. VFS maps OS-independent file ids onto internal file ids, tracks available local and remote file systems and passes requests. IDs are termed file handles, which are identifiers passed between server and client for file operations.

Caching: store recently used data and directories locally for repeated access, can be stored on client (harder to maintain consistency, low network load) or server (faster, no consistency problem, high network load).
Cache flush policies: write through to server (write in parallel, reliable but poor performance), delayed write (periodic write or write when cache replaced, consistency problem) and write on close (problem if file opened for long periods of time).
Cache validation: checks whether client copy is still valid (TTL expiry or recent update) with modification timestamp and validation timestamp. Accept if $T - T_c < t$ or $T_m$client = $T_m$server, where t is freshness internal (set adaptively based on file update frequency), T_c is last cache validated time and T_m is last modification time.
Synchronisation: it is inefficient to propagate all changes to server immediately, but instead session semantics is used (modifications saved when file is closed). However, concurrent access remains a problem. In NFS, file locks are used, and measures are in place to prevent irresponsive client or server crashes from locking down a file.

Role of OS in Security
OS should ideally provide authenticity (identify user, with password etc.), confidentiality (protection information from attacker, e.g. memory protection, file protection and I/O device protection), integrity (no alteration of operations under attack) and availability (attack cannot prevent users from accessing the system).
UNIX:
File permission set by owner or root (not delegable) to protect files.
Each process has real user ID (RUID, process starter), effective user ID (EUID, executing user) and saved user ID (SUID for restoration); and similar scheme for group IDs.
Processes can however change ID (and their file access permissions) by calling executable files with setuid and setgid bits, which will run with the permission of the file's owner.
Root as system administrator has ID 0, which is able to access any file. **fork** and **exec** will by default inherit the three ID's. A program owned by root, and executed by a normal user will run at the permissions of the normal user, but if the setuid bit is set to root, the process will run as root, which is dangerous.
Common system attacks take over a machine by remotely running a program with setuid(root) operation, make the program malfunction in a way that it forks another process (e.g. shell) with full root privileges. Apart from running as few applications with root as possible, this is both an issue for OS and for applications.
Buffer overflows can be used to overwrite application's return address, and insert new code to gain root privilege. Hardware protection of certain parts of memory, and use of memory-safe functions in applications can help prevent the issue.
Other system threats include spyware installed along with legitimate software, malware, botnets, social engineering, etc.
Sandboxed environments or virtual machines can be used to test untrusted code.