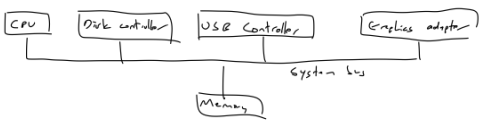
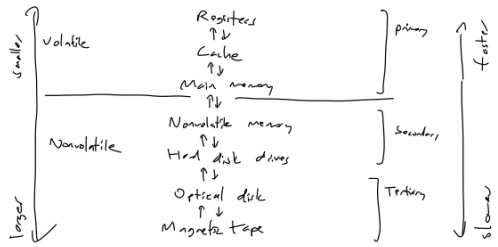
1: Introduction

General OS goals. Execute solving programs to make user tasks easier, use hardware in an efficient manner, make the computer convenient to use.
User goals. Convenience, ease of use, reliability, safety.
System goals. Ease of maintenance, flexibility, efficiency.
OS components. Kernel, system program, application program, user.
OS user view. For the user of a general-purpose operating system, the OS is designed primarily for ease of use and good performance, with no attention paid to resource utilization.
OS system view. Resource allocator (manage hardware), control program (execution)
OS definition. We have no universally accepted definition of an OS. One viewpoint states that it includes everything a vendor ships. Another definition is that the operating system is the "one program running at all times," or the kernel.
Computer startup/system boot. The bootstrap program, typically stored in a fixed point in nonvolatile EEPROM and other forms of firmware, is the initial program to run. It initiates all aspects of the system including CPU registers, device controllers, and memory contents. The bootstrap program locates the OS kernel and loads it into memory. Once the kernel is loaded, it loads everything else (system daemons, etc.).
EEPROM. Electrically erasable programmable read-only memory (EEPROM) is nonvolatile memory which stores the bootstrap program and other information that is infrequently written to.
Basic computer system organization.



Interrupts. See Chapter 12.
Main memory. This type of storage, including CPU cache and RAM is directly accessed by the CPU.
Secondary storage. The most common secondary-storage devices are hard-disk drives (HDDs) and nonvolatile memory (NVM) devices, which provide storage for programs and data.
Tertiary storage. This type of storage is slow/large enough that they are used only for special purposes (e.g. backups).
Storage hierarchy diagram.



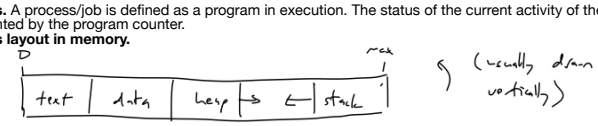
Direct memory access (DMA). See Chapter 12.
Asymmetric multiprocessing. Each processor is assigned a specific task.
Symmetric multiprocessing. This is the most common type of multiprocessing, where each processor performs all tasks. Note that each processor has its own set of registers, and a private/local cache, but share the same physical memory over the system bus.
Multiprocessor and multicore. The definition of multiprocessor has evolved over time and now includes multicore systems, in which multiple cores reside on a single chip. Multicore systems are more efficient because on-chip communication is faster than between-chip communication. Additionally, one chip with multiple cores uses significantly less power than multiple single-core chips. See Chapter 5 for an expanded definition of *multiprocessor*.
Advantages of multiprocessing. Increased throughput, economy of scale, increased reliability.
Migration of data from disk to register. Hard disk → main memory → cache → register
Cache coherency. Multiprocessor environments must provide this in hardware such that all CPUs have the most recent value in their cache.
Non-uniform memory access (NUMA). Each CPU has its own *local memory* that is accessed via a small, fast local bus. This means that access times for memory can vary greatly. The CPUs are connected via a shared system interconnect, so that all CPUs share one physical address space. With this approach, there is no contention over the interconnect, so NUMA systems can scale more effectively as more processors are added, even if NUMA systems have generally slower access to memory.
Clustered systems. Consists of two or more systems loosely coupled together; each system is typically a multicore system. Clustered systems usually share storage via an SAN to provide a high-availability service which survives failures. Some clusters for high-performance computing are designed for *parallelization*; some have a *distributed lock manager* to avoid conflicting operations.
Asymmetric clustering. One machine is available in hot-standby mode, monitoring the active server, and taking over as the active server should the other server fails.
Symmetric clustering. Two or more hosts are running applications and monitoring each other. This structure requires that more than one application be available to run.
Multiprogramming. Multiprogramming increases CPU utilization by organizing programs so that the CPU always has something to execute. The *degree of multiprogramming* refers to the number of programs that are being managed by the OS and are resident in memory at the same time. *Time-sharing* is the illusion of direct access to the system.
Multitasking. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently providing the user with a fast response time.
Kernel and user modes. The mode bit is added to the hardware to indicate the current mode (kernel or user). When the computer system is executing on behalf of a user application, the system is in user mode. When the application requests an operating system service, the system transitions from user to kernel mode to fulfill the request. Kernel mode allows the execution of privileged instructions.
Protection and security. Protection is defined as any mechanism for controlling the access of processes or users to the resources defined by a computer system. The job of *security* is to defend a system from external/internal attacks.

2: OS Structures

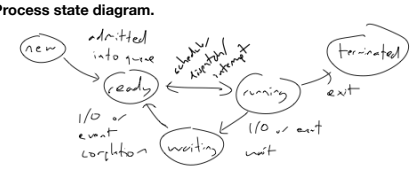
User-facing OS services. User interface, program execution, I/O operations, filesystem manipulation, communications/networking, error detection.
System-facing OS services. Resource allocation, accounting (tracking of resources), protection/security
Command interpreters. The main function of the command interpreter is to get and execute the next user-specified command. The command interpreter may contain the code to execute the command itself, or it may implement most commands through system programs (used by UNIX).
System calls. These provide an interface to the services of an operating system. The most common system APIs are the Windows API (Win32), POSIX API, and Java. The functions that make up an API typically invoke the actual system calls on the user's behalf. A system call is often implemented via a *lookup table* in the kernel.
System call categories. Process control (create/terminate process, load/execute, get/set process attributes, wait/signal events, allocate/free memory), file management (create/delete file, open/close, read/write/reposition, get/set file attributes), device management (request/release device, read/write/reposition, get/set device attributes, logically attach/detach devices), information maintenance (get/set time or date, get/set system data, get/set process/file/device attributes), communications (create/delete communication connection, send/receive messages, transfer status information, attach/detach remote devices), and protection (get/set file permissions).
System call parameter passing. Registers, block/table in memory, stack method.
System programs/services/utilities. Provide a convenient environment for program development and execution. Some are simply user interfaces to system calls, while others are more complex.
System program categories. File management (create/delete/copy/rename/print/dump/list/manipulate), status information (provide information incl. date/time/available memory/disk space/number of users, provide detailed performance/logging/debugging info), file modification, PL support, program loading/execution, communications (virtual connections among processes, users, and computer systems), background services (known as services/subsystems/daemons, *run in user context*, provide disk checking/process scheduling/error logs).

3: Processes

Process. A process/job is defined as a program in execution. The status of the current activity of the process is represented by the program counter.
Process layout in memory.



Process state diagram.



Process/task control block (PCB). A representation of a process in the OS.
PCB elements. Process state, program counter, CPU registers, CPU scheduling info (pointers to priority/scheduling queue), memory management info (base/limit registers, what's allocated to the process), accounting info (amount of CPU/real time used, job/process numbers), I/O management info.
I/O bound and CPU bound. I/O bound processes spend more time doing I/O than computations; they have many short CPU bursts. CPU bound processes spend more time doing computations; they have fewer, longer CPU bursts.
Scheduling queues. As processes enter the system, they are put into a *ready queue* where they are waiting to execute on a CPU's core. Processes that are waiting for a certain event to occur, such as completion of I/O, are placed in a *wait* or *device queue*. Finally, the *job queue* is the set of all processes in the system.
Short-term/CPU scheduler. Selects which process must execute next, and allocates the CPU.
Long-term/job scheduler. Selects which processes should be brought into the ready queue.
Medium-term scheduler. Handles swapping and decreasing the degree of multiprogramming.
Multitasking on mobile systems. Generally, mobile systems only allow one foreground with a handful of background processes; background processes have limits.
Context switching. When an interrupt occurs, the system performs a state save of the current CPU core state, and then a state restore to resume operations. The context is represented by the PCB. Some processors provide multiple sets of registers; a context switch here simply requires changing the pointer to the current register set.
Process creation. In UNIX, processes are a tree starting from the *init/systemd* process. A new process is created using the *fork* command. The return code for *fork* is zero for the child process, and the child's process identifier for the parent. The exec command is then typically used in the child to replace the process's memory space with a new program. The parent can issue *wait* to move itself off the ready queue until the child is killed.
Process termination. Some systems implement *cascading termination*, where children are killed with the parent. A child is a *zombie* if no parent has yet called *wait* on it, or an *orphan* if the parent terminates without waiting. Traditional UNIX systems assign *init* as the parent to orphans, periodically calling *wait* and releasing the orphan's identifier.
Interprocess communication. Processes are *independent* if they don't share data with other processes, or *cooperating* if it can affect/be affected by other processes.
Reasons for process cooperation. Information sharing, computation speedup, modularity, convenience.
Methods of interprocess communication (IPC). Shared memory, message passing. See below.
Producer-consumer problem. A producer produces information that must be consumed by a consumer process.
Shared memory method of IPC. Two processes read and write to a region of shared memory. The two processes must be synchronized, so that one process doesn't read to a region that the other is writing to. Unbounded or bounded buffers can be used.
Message passing method of IPC. The facility provides *send* and *receive* operations, and the message size can be fixed or variable. If processes wish to communicate, they must establish a *communication link*. This link can be implemented in a variety of ways. Two variations of message passing are direct and indirect communication.
Direct communication. Each process that wants to communicate must explicitly name the recipient/sender. A communication link is associated with exactly two processes; there is exactly one link between a pair of processes. There are *symmetric* and *asymmetric* variations of this scheme; in the *asymmetric* variation, the receiver is not required to name the sender. The disadvantage of direct communication is the limited modularity of process definitions.
Indirect communication. Messages are sent to and received from *mailboxes* or *ports*. Each mailbox has a unique identification. Two processes have a communication link if they have a shared mailbox, and they must name the mailbox. A communication link may be associated with more than two processes, and a number of different links may exist, with each link corresponding to one mailbox. Links may be unidirectional or bidirectional.
Synchronization. Message passing may be either blocking or non-blocking, or synchronous and asynchronous. With a nonblocking receive, the receiver retrieves either a valid message or a null. When both send and receive are blocking, we have a *rendezvous* between the sender and receiver.
Pipes. A pipe is a simple IPC mechanism. In implementing a pipe, there are issues that must be considered: bidirectional/unidirectional communication? Half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)? Relationship required (e.g. parent-child)? Network or same-machine?
Ordinary/anonymous pipes. Allow two processes to communicate in standard produce-consumer fashion, with write and read ends. Ordinary pipes are unidirectional, requiring two pipes for bidirectional communication. Once both processes have terminated, the ordinary pipe ceases to exist.
Named pipes. Communication can be bidirectional, and no parent-child relationship is required. Named pipes continue to exist after termination, and several processes can use it (e.g. can have several writers).
Sockets. A socket is defined as an endpoint for communication. Each process communicating over a network has a socket associated with it. A socket is identified as an IP address with a port number. All port numbers below 1024 are *well-known*. The IP address 127.0.0.1 is known as a *loopback*, since the computer is referring to itself.
Remote procedure calls (RPC). These abstract procedure calls between processes on networked systems, and use ports for service differentiation. The client-side proxy provides *stubs* for actual procedures on the server; each stub locates the port on the server and *marshals* the parameters. Similarly, a stub on the server-side receives the message and invokes the procedure. Marshaling or external data representation (XDR) addresses differences in data representation; for instance, *big-endian* systems store the most significant byte first, while *little-endian* systems store the least significant byte first.
RPC call semantics. The *at most once* semantic guarantees that a message is executed only once if the client sends a message one or more times. It is implemented by attaching a timestamp to each message, and the server must keep a history of timestamps for messages it has already processed. The *exactly once* semantic implements the *at most once* semantic, but the server must also acknowledge ("ACK") to the client that an RPC call was received and executed. The client resends the RPC call periodically until it receives the ACK for the call.
Matchmaker/rendezvous daemon. A service provided by the OS which runs on a fixed RPC port. The client sends a message to the matchmaker requesting the port number of the RPC it needs to execute. This method requires extra overhead but is more flexible than predetermining the fixed port address.

4: Threads

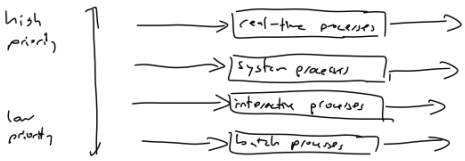
Thread. A basic unit of CPU utilization. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control; multithreaded processes have multiple threads of control.
Comparison of single-threaded and multithreaded processes.

Examples of threaded applications. An application which generates photo thumbnails on a separate thread; a web browser with dedicated threads for displaying content and retrieving network data; a word processor with a separate thread for spell checking.
Benefits of multithreaded programming. Responsiveness, resource sharing by default, economy, scalability.
Concurrency and parallelism. A concurrent system supports more than one task by allowing all tasks to make progress. In a parallel system, multiple tasks are being worked on simultaneously. It is possible to have concurrency without parallelism. *Data parallelism* focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. *Task parallelism* involves distributing not data, but tasks (threads) across multiple cores. Data parallelism and task parallelism are not mutually exclusive.
Challenges in programming for multicore systems. Identifying tasks (that can be divided), balance (ensuring that tasks perform equal value), data splitting (must be divided into separate cores), data dependency (for data accessed by more than one task), testing and debugging.
Amdahl's Law. Identifies gains that comes from allocating more cores for an app with serial and parallel components.
User and kernel threads. User threads are supported *above* the kernel and are managed without kernel support, whereas kernel threads are supported (directly scheduled) and managed directly by the operating system. User threads are *unknown* to the kernel, and kernel threads do not have to be associated with a specific process. Kernel threads are more expensive to maintain because they must be represented with a kernel data structure.
Many-to-one model. Maps many user threads to one kernel thread. The entire process will block if a thread makes a blocking system call. Few systems use this model because of its inability to take advantage of multicore systems.
One-to-one model. Maps each user thread to a kernel thread. This approach provides better concurrency than the many-to-one model, and allows for parallelism on multiprocessors. However, it is expensive and a large number of kernel threads may burden the performance of a system. *Most operating systems use this model.*
Many-to-many model. Maps many user threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. Developers may create as many user threads as necessary, and corresponding kernel threads may run in parallel on a multiprocessor. One variation of this, called the *two-level model*, allows certain user threads to be bound to kernel threads.
Asynchronous and synchronous threading. With synchronous threading, the parent thread creates and waits for one or more children to terminate before resuming. Asynchronous is the opposite.
Implicit threading. Describes creation and management of threads by the runtime environment rather than explicitly. Developers must identify *tasks*, not *threads*, which can run in parallel. Three approaches discussed in the textbook are *thread pools* (using a work queue with a pool of threads), *OpenMP* (set of compiler directives which identify parallel regions as blocks of code), and *Grand Central Dispatch* (schedules tasks or execution by placing them on a serial or concurrent dispatch queue, managed internally using POSIX threads).
***fork* and *exec* calls with threads.** The default *fork* call creates a new process with the calling thread as the only active thread. Some versions of UNIX such as Solaris have a *forkall* call which duplicates all threads. The *exec* call works the same in that the new program will replace the entire process, including all threads.
Synchronous and asynchronous signals. A signal is generated by the occurrence of a particular event, is then delivered to a process, and must be handled. A signal may be *synchronous*, or delivered to the same process that performed the operating causing the signal (examples include illegal memory access and division by 0). *Asynchronous* signals come from an event external to the running process.
User-defined signal handlers. Every signal has a *default signal handler* that the kernel runs when handling that signal. The default action can be overridden by a *user-defined signal handler* that is called to handle the signal.
Handling signals in multithreaded programs. Generally, there are four options for signal delivery to a process with several threads: deliver the signal to the thread to which the signal applies, deliver the signal to every thread in the process, deliver the signal to certain threads in the process, assign a specific thread to receive all signals for the process. The method chosen depends on the type of signal; for instance, synchronous signals should only be delivered to the thread causing the signal, while asynchronous signals should be delivered to all threads. Most multithreaded versions of UNIX allow a thread to specify which signals are accepted and which are blocked. However,

because signals only need to be handled once, they are typically delivered only to the first thread found that's not blocking it. The standard UNIX function for delivering a signal is *kill*; alternatively, the *pthreads_kill* function allows a signal to be delivered to a specific thread.
Thread cancellation. This involves terminating a thread before it has completed. For instance, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads may be canceled. The *target thread* is the thread to be canceled. *Asynchronous cancellation* is when the target thread is immediately terminated, while *deferred cancellation* (the default type) is when the target thread periodically checks whether it should terminate (at *cancellation points*), allowing itself to terminate in an orderly fashion. In the case of deferred cancellation, the *cleanup handler* is invoked to release resources acquired by the thread.
Thread-local storage (TLS). In some cases, each thread might need its own copy of certain data. Such data is called thread-local storage (TLS). Unlike local variables, TLS data is visible across function invocations.
Additional exam notes. When counting unique creation of threads, only count calls of *pthread_create*; do not consider thread created by *fork* or *exec* calls.

5: CPU Scheduling

CPU-I/O burst cycle. Processes alternate between CPU bursts and I/O bursts. Execution begins with a CPU burst, which is followed by an I/O burst, back to a CPU burst, and so on. Eventually there is a final CPU burst to terminate.
CPU scheduler. Carries out the selection process of a process in the ready queue, and allocates the CPU.
Preemptive and nonpreemptive scheduling. CPU scheduling decisions may take place under the following four circumstances: 1) running → waiting (e.g. I/O request or *wait* call), 2) running → ready (e.g. interrupt), 3) waiting → ready (e.g. I/O completion), 4) running → terminated. When scheduling takes place *only under* circumstances 1 and 4, we say the scheduling scheme is *nonpreemptive* or *cooperative*. Under *nonpreemptive scheduling*, the process keeps the CPU until released, either through terminating or voluntarily switching to the waiting state.
Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the scheduler. This function involves switching context from one process to another, switching to user mode, jumping to the proper location in the user program to resume that program.
Dispatch latency. The time it takes for the dispatcher to stop one process and start another.
Scheduling criteria. CPU utilization (in practice should range from 40-90%), throughput (# of processes completed execution per time unit), turnaround time (time to execute a particular process), waiting time (time a process has been in the waiting queue), response time (time from request submitted → first response).
First come, first serve (FCFS) scheduling. The simplest CPU-scheduling algorithm, where processes are managed in FIFO order. Downsides of FCFS include a long waiting time, and the *convoy effect*, where one CPU-bound process blocks many I/O-bound processes. Note that the FCFS scheduling algorithm is *nonpreemptive*.
Shortest job first (SJF) scheduling. This algorithm considers the length of the process's next CPU burst, and assigns the CPU to the process with the smallest next CPU burst. Since we cannot predict the future, actual SJF implementations approximate the length of the next CPU burst using exponential averaging. The average waiting time of SJF is shorter than that of FCFS. SJF has preemptive and nonpreemptive variations; the choice arises when a new process arrives at the ready queue while a previous one is still executing. The preemptive version (called *shortest remaining time first*, or SRTF) may preempt the current process if the CPU burst of the new process is shorter than what is left of the currently executing process.
Round-robin (RR) scheduling. Same as FCFS, but preemption is added to enable switching between processes. A *time quantum* or *time slice* is defined, the ready queue is treated as a circular queue, and the CPU scheduler allocates each process for a period of up to 1 time quantum. The average waiting time under RR is quite long, and so is the turnaround time compared to SJF, but RR generally provides better response time. If the time quantum is extremely, it results in a large number of context switches; if too large, the algorithm degenerates into FCFS.
Priority scheduling. The SJF algorithm is a special case of priority scheduling, where a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Priority scheduling may be non-preemptive or preemptive; the preemptive variant will preempt the running process if an incoming process has a higher priority. A major problem with priority algorithms is *indefinite blocking* or *starvation*. One solution to this is *aging*, or gradually increasing the priority of processes that wait for longer. Another is priority round-robin scheduling; see below.
Priority round-robin scheduling. This is a variant of priority scheduling which executes processes with the same priority using round robin. *This means that if only one process has a specific priority, that process finishes executing without preemption once started.*
Multilevel queue scheduling. In practice, it is often easier to have separate queues for different priority levels of processes, and to use priority scheduling to schedule the process in each distinct queue. *This approach has low scheduling overhead but is inflexible.* This approach works well with round-robin; if there are multiple processes in the highest-priority queue, they are executed in round-robin order. Another use case is to separate foreground (interactive) processes and background (batch) processes.
Multilevel queue scheduling example.



Multilevel feedback queue scheduling. Unlike normal multilevel queue scheduling, this variant allows a process to move between queues. If a process uses too much CPU time, it will be moved to a lower-priority queue. This approach leaves I/O bound processes in higher-priority queues. It also implements aging by moving long-waiting processes to higher-priority queues, to prevent starvation.
Thread scheduling. Like mentioned in Chapter 4, kernel-level threads are *unaware* of user-level threads. User-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
Contention scope. On systems implementing the many-to-one and many-to-many mapping models, the *thread library*, not the CPU, schedules user-level threads to run on an available LWP. This scheme uses *process-contention scope (PCS)*, where competition for the CPU takes place among threads belonging to the same process. When scheduling kernel-level threads onto a CPU, the kernel uses *system-contention scope (SCS)*, where competition for the CPU takes place among all threads in the system. Systems using the one-on-one mapping model only use SCS.
Multiprocessor definition. The term *multiprocessor* can be expanded to include multicore CPUs, multithreaded cores, NUMA systems, and heterogeneous multiprocessing. In the first three, processors are *homogenous* in functionality.
Asymmetric multiprocessing in scheduling. In asymmetric multiprocessing, each processor has a specific task, and scheduling is one of them. All scheduling is handled by a single processor. This approach is simple because only one core accesses the system data structures, reducing the need for data sharing.
Symmetric multiprocessing in scheduling. The standard approach for supporting multiprocessors, where each processor is self-scheduling. Threads may either be in a common ready queue shared by all cores, or each processor may have its own private queue of threads.
Multicore processors and scheduling issues. Multicore processors may complicate scheduling issues by introducing a *memory stall*, which occurs primarily because modern processors operate at much faster speeds than memory, and because of a cache miss. To remedy this situation, modern hardware designs implement multithreaded processing cores with two or more *hardware threads*. Only one thread may technically run at a time; however, if one thread stalls while waiting for memory, the core can switch to another thread. Each hardware thread maintains architectural state such as instruction pointer and register set, and appears as a logical CPU available to run a software thread. This technique is also known as *chip multithreading*.
Coarse-grained and fine-grained multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Fine-grained (or interleaved) multithreading switches between threads at the boundary of an instruction cycle. The design of fine-grained systems includes logic for thread switching, so the cost of switching threads is much smaller than coarse-grained multithreading systems, where the instruction pipeline must be flushed before the other thread begins execution.
Load balancing. Only necessary on systems where each processor has its own private ready queue of eligible threads to execute. The two general approaches to load balancing are push and pull migration. With push migration, a specific task periodically checks the load on each processor and evenly distributes the load by moving threads. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration are not mutually exclusive, meaning both can be implemented at the same time.
Processor affinity. Most operating systems with SMP support try to avoid migrating a thread from one processor no another and instead attempt to keep a thread running on the same processor, taking advantage of a *warm cache*. When the OS attempts to keep a process running on the same processor, but has no guarantee it will do so, we have *soft affinity*. In contrast, some systems support *hard affinity*, allowing a process to specify a subset of processors on which it can run. Note that processor affinity works against load balancing.
Heterogenous multiprocessing (HMP). Improves power consumption by assigning tasks to certain cores based upon the specific demands of the task. For instance, some ARM processors support a *big.LITTLE* architecture.
Real-time CPU scheduling. *Soft real-time systems* provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. *Hard real-time systems* have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all. *Interrupt latency* and *dispatch latency* affect the performance of real-time systems.
Latency. *Event latency* is the amount of time that elapses from when an event occurs to when it is serviced. *Interrupt latency* refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. *Dispatch latency* refers to the amount of time required for the scheduling dispatcher to stop one process and start another. The *conflict phase* of dispatch latency preempts any process running in the kernel, and releases resources used by low-priority processes needed by a higher-priority process. Following the conflict phase, the dispatch phase schedules the high-priority process onto the CPU.
Characteristics of priority based scheduling. Processes are considered *periodic*, where they require the CPU at constant intervals (periods). The task has a fixed processing time *t*, a deadline *d* by which it must be serviced by the CPU, and a period *p*. The *rate* of the periodic task is 1/p. Using a technique known as *admission control*, the scheduler either admits the process (guaranteeing that it will complete on time), or rejects the request.
Rate monotonic scheduling (RMS). Uses a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst.
Earliest deadline first (EDF) scheduling. Assigns priorities dynamically according to the deadline, with preemption; the earlier the deadline, the higher the priority. The process must announce its deadline to the scheduler. The process doesn't need to be periodic (can be one-time), and the CPU time required can vary. In practice, it is impossible to achieve high CPU utilization due to the cost of context switching between processes and interrupt handling.

6: Synchronization

The critical section problem. Each process has a segment of code, called a critical section, in which the process may be accessing and updating data that is shared with at least one other process. When one process is executing in its critical section, no other process is allowed to execute in its critical section. Each process must request permission to enter its critical section; this happens in the *entry section*. The critical section may be followed by an *exit section* to release resources, followed by a *remainder section*.
Requirements to solving the critical section problem. Mutual exclusion (when one process is executing in its critical section, no other process can be executing in their critical sections), progress (if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely), bounded waiting (there exists a limit on the number of times other processes are allowed to enter their critical sections after a process has requested to enter its critical section).
Preemptive and nonpreemptive kernels. A *preemptive kernel* allows a process to be preempted while it is running in kernel mode. A *nonpreemptive kernel* is the opposite, and it is free from race conditions on kernel data structures.
Pseudocode for Peterson's Solution.

```
do {
  flag[0] = true;  // indicates process is ready
  turn = 1;  // indicates whose turn it is to enter critical section
  while (flag[1] && turn == 1) do no-op;  // wait for resource
  // critical section
  flag[0] = false;
  // remainder section
} while (true)
```

Hardware instructions. The *test_and_set* and *compare_and_swap* instructions are atomic instructions implemented in hardware. Modern software-based solutions are dependent on these instructions.
Algorithmic description for test_and_set.

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;
  return rv;
}
```

Algorithmic description for compare_and_swap.

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value == expected) *value = new_value;  // only update if it matches expected arg
  return temp;  // return the original value
}
```

Atomic variables. This tool uses *compare_and_swap* to provide atomic operations on basic data types such as integers and booleans. This is used to prevent the race condition that incrementing/decrementing may cause.
Mutex locks. We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. Mutex locks may be implemented using *compare_and_swap*. The mutex lock is a *spinlock*, which wastes CPU cycles when busy waiting but has the advantage that no context is required for waiting on a lock.. *Additionally, this lock can only be released by the process which acquired it.*
Semaphores. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations *wait* and *signal*. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1, similar to a mutex lock. The implementations of *wait* and *signal* may use busy waiting, or they may suspend/resume the process. *Note that under the second implementation, semaphore values may be negative indicating the number of processes waiting.*
Monitors. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. A monitor type is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. *The monitor construct ensures that only one process at a time is active within the monitor.*
Liveness failure. A *deadlock* may occur where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. *Priority inversion* occurs in systems with more than two priorities, where a lower-priority process is preempted by an unrelated medium-priority process, while holding a lock on a resource required by a higher-priority process. Typically priority inversion is avoided by implementing the *priority-inheritance protocol*, where processes accessing resources needed by a higher-priority process inherit the higher-priority until finished with the resources.

7: Synchronization Examples

Bounded-buffer problem. The *bounded-buffer problem* contains *n* buffers, each of which can each hold one item. There is the *mutex* semaphore to protect the resource being written, the *full* semaphore indicating how many buffers are available for reading, and the *empty* semaphore indicating the number of empty spaces available for writing.
Readers-writers problem. This problem describes a case which allows multiple readers of data at the same time, but only one writer. The *rw_mutex* semaphore protects the resource being read/written to, and the *mutex* semaphore protects the *read count*, which indicates the number of processes reading concurrently.
Pseudocode for the readers-writers problem.

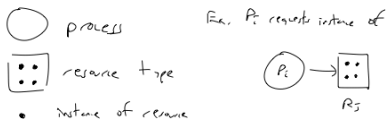
```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;

// Writer process
while (true) {
  wait(rw_mutex);
  // write to data
  signal(rw_mutex);
}

// Reader process
while (true) {
  wait(mutex);
  read_count++;
  if (read_count == 1) wait(rw_mutex);
  signal(mutex);
  // read from data
  wait(mutex);
  read_count--;
  if (read_count == 0) signal(rw_mutex);
  signal(mutex);
}
```

Dining philosophers problem. The *dining philosophers problem* involves a group of philosophers who alternate between thinking and eating at a table, but must contend for resources (forks) without causing deadlock. A basic solution using an array of binary semaphores can result in a deadlock; one solution is to only allow eating if both forks are available. *Note that even this solution may result in starvation for one of the philosophers.*
Pseudocode for a philosopher in the dining philosophers problem, using a binary semaphore array.

```
while (true) {
  wait(fork[i]);
  wait(fork[(i + 1) % 6]);
  // eat
  signal(fork[i]);
  signal(fork[(i + 1) % 5]);
  // think
}
```

8: Deadlocks

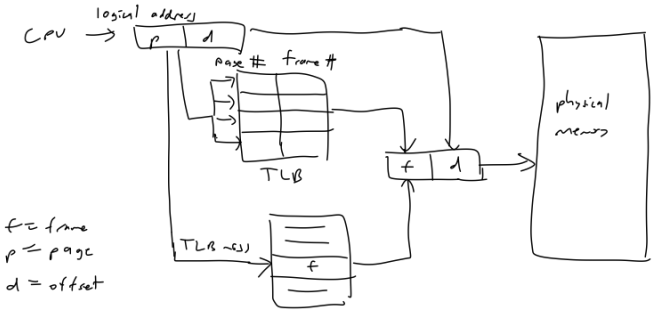
Deadlock. A deadlock is a form of liveness failure, and it is defined as a situation in which every process in a set of processes is waiting for an event that can be caused only by another process in the set. Operating systems do not provide deadlock-prevention facilities, leaving it to the programmer to ensure they design deadlock-free programs.
Livelock. A livelock is another form of liveness failure. It prevents two or more threads from proceeding, *due to a thread continuously attempting an action that fails*.
Deadlock characterization. The necessary conditions for a deadlock are mutual exclusion (only one process at a time can use a resource), hold and wait (process with resource(s) is waiting for a resource), no preemption *of the resource* (resource can be released only after task completed), circular wait: processes $P = \{P_0, P_1, \ldots P_n\}$ has a chain of dependencies that come from waiting for resources.
Resource-allocation graph. A directed graph which describes deadlocks. The graph contains vertices $P = \{P_1, P_2, \ldots P_n\}$ containing all processes and vertices $R = \{R_1, R_2, \ldots R_m\}$ containing all resource types. The graph also contains request edges $P_i \rightarrow R_j$, and assignment edges $R_j \rightarrow P_i$. *Note that processes P are interchangeable with threads T in this scenario.* If the graph contains no cycles, then there is no deadlock. If the graph contains a cycle, and there is only one instance per resource type, then a deadlock has occurred. If there are several instances per resource type, there is the possibility of a deadlock occurring.
Resource-allocation graph notation.



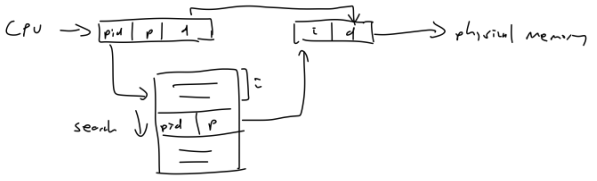
Deadlock handling methods. Systems can ignore the problem, use a protocol to prevent/avoid deadlocks (ensuring the system never enters a deadlock), or provide a protocol to detect and recover from a deadlocked state. Most operating systems ignore the problem since deadlocks occur infrequently, and it is cheaper than other approaches.
Deadlock prevention. *Deadlock prevention* provides a set of methods to constrain how requests for resources can be made, ensuring that at least one of the necessary conditions cannot hold.
Preventing the four conditions. Generally speaking, we cannot deny the *mutual-exclusion condition* because some resources are non-sharable. To prevent *hold-and-wait*, the system can require each thread to be allocated resources before it begins execution, or allow a thread to request resources only when it has none. Both may lead to low resource utilization and starvation. To prevent the *no preemption* condition, we can use the following: if a thread is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the thread must wait), then all resources the thread is currently holding are preempted. The preempted resources are added to the list of resources for which the thread is waiting. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions. To prevent *circular wait*, we can impose a total ordering of all resource types and to require that each thread requests resources in an increasing order.
Deadlock avoidance. *Deadlock avoidance* requires that the operating system be given additional information in advance concerning resources a thread will request and use during its lifetime. The simplest and most useful model requires that each thread declare the *maximum number* of resources of each type that it may need.
Safe state. A state is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock. Formally, there exists $P = \{P_0, P_1, \ldots P_n\}$ such that for each P_i, resources that P_i can request can be satisfied by currently available resources + resources held by all P_j with $j < i$. An *unsafe state* is not necessarily a deadlock, but may lead to one.
Resource-allocation graph algorithm. We can use a variant of the resource-allocation graph for deadlock avoidance. In addition to the request and assignment edges, we introduce a new type of edge, called a claim edge. *This algorithm is not applicable to a system with multiple instances of each resource type.*
Banker's algorithm. This scheme is applicable to a system with multiple instances of each resource type, but is less efficient than the resource-allocation graph algorithm. When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need.
Deadlock detection. The system may an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock.
Recovery from a deadlock. One option to break a deadlock is to abort one or more threads to break the circular wait. Another is to preempt some resources from one or more deadlocked threads. For *process termination*, we can abort all deadlocked processes or abort one process at a time until the deadlock cycle is eliminated. The second method has considerable overhead because the deadlock-detection algorithm must be invoked for every process terminated. Three issues to be addressed with *resource preemption* is selecting a victim (which resources/which processes?), rollback (usually a total rollback; abort and restart the process), and starvation (will resources always be preempted from the same process?).

9: Main Memory

Stall. Completing a memory access may take many cycles of the CPU clock, which leads to a stall, since the CPU doesn't have require data. This is undesirable because of the frequency of memory accesses. The remedy is to add a cache typically on the CPU chip for faster access.
Base and limit registers. Protecting the operating system from user processes, as well as protecting user processes from one another, must be done by the hardware. The base and limit registers establish the legal logical address space of the process. The base register is the smallest legal memory address, while the limit register specifies the range.
Binding of instructions and data to memory addresses. Binding can be done at compile-time (generating absolute code; recompilation needed when starting location changes), load time (generates relocatable code; must reload user code when starting location changes), or execution time (process can be moved during its execution from one memory segment to another, requires special hardware).
Logical and physical addresses. An address generated by the CPU is referred to as the logical address, whereas an address seen by the memory unit is the physical address. *Logical and virtual addresses are used interchangeably in the text.* The run-time mapping from virtual to physical addresses is done by a hardware device called the *memory-management unit (MMU)*. User programs never access physical addresses.
Dynamic loading. With dynamic loading, a routine is not loaded into memory until it is called. All routines are kept on disk in a relocatable load format. The advantage of dynamic loading is that a routine is loaded only when needed. *However, the main program (a significant portion) is still loaded into memory.*
Dynamic linking and shared libraries. Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run. Some systems support only static linking, in which libraries are treated like other object modules and combined by the loader into into the binary program image. Dynamic linking is similar to dynamic loading, except linking rather than loading is postponed until execution time.
Contiguous memory allocation. The memory is usually divided into two partitions: one for the OS and one for processes. The OS may be placed in either low or high memory addresses. Each process is contained in one section of memory that is contiguous to the section containing the next process.
Variable-partition scheme. The simplest method is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process. The OS must keep a table indicating which parts of memory are available and which are occupied. Free parts of memory are referred to as *holes. When a hole is allocated to a process and there's space remaining, the leftover space cannot be used until the hole is reconfigured.*
First fit strategy for dynamic-storage allocation. Allocate the first hole that is big enough.
Best fit strategy for dynamic-storage allocation. Allocate the smallest hole that is big enough. This strategy requires searching the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
Worst fit strategy for dynamic-storage allocation. Allocates the largest hole. This strategy requires searching the entire list, and produces the largest leftover hole, which may be more useful than the smallest leftover hole.
External fragmentation. Both First-fit and best-fit strategies suffer from external fragmentation. External fragmentation exists when enough memory space exists to satisfy a request, but they are not contiguous.
50-percent rule. Given N allocated blocks, another 0.5 N blocks will be lost. A third of memory is unusable.
Internal fragmentation. The memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is referred to as internal fragmentation.
Compaction. One solution to external fragmentation is compaction, which shuffles memory contents to place free memory together in one large block. Compaction is not possible if relocation is static and done at assembly/load time. It is possible only if relocation is dynamic and done at execution time.
Paging. Paging is a memory-management scheme that permits a process's physical address space to be noncontiguous. Paging avoids external fragmentation and the associated need for compaction. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
Page table. Small page tables may be stored on CPU registers. However, most modern CPUs support much larger page tables, which must be kept in main memory. In this case, a page-table base register (PTBR) points to the table.
Transition look-aside buffer (TLB). The TLB is associative, high-speed memory. When the associative memory is presented with an item, the item is compared with all keys simultaneously (parallel search), adding no performance penalty. The MMU first checks if a page number is present in the TLB. If available, its frame number is immediately used to access memory. If unavailable (TLB miss), the MMU goes to the page table. Some entries, such as those for kernel code, may be *wired down*. Some TLBs store address-space identifiers (ASIDs) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection. If the TLB does not support separate ASIDs, the TLB must be flushed every time a new page table is selected (e.g. context switch).
Full diagram of paging hardware, including TLB.



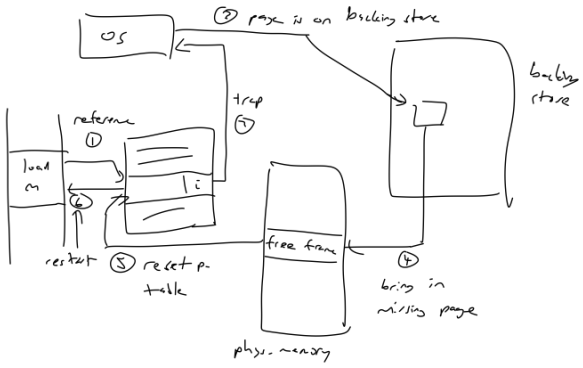
Calculating the page number size and offset. The page number has to have enough bits to represent the *number of pages in the table*, while the offset must have enough bits to represent the *page size*. If given the number of pages, take the logarithm with base 2 to find the number of bits needed for the page number. If given the page size, do the same thing to find the offset. For any address, the page number is the first few bits followed by the offset. When converting from binary back to hex, count from the *end* of the binary number.
Effective access time with TLB. TLB hit % * memory access time + TLB miss % * (memory access time * 2).
Valid-invalid bit. When the bit is set to valid, the associated page is a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space.
Page-table length register (PTLR). This is hardware provided on some systems to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process.
Shared pages. *Reentrant code* can be shared. For instance, only one copy of the standard C library needs to be kept in physical memory, as well as other heavily used programs such as compilers, window systems, databases, etc.
Hierarchical page tables. The page table itself is also paged, splitting the page number of the logical address into two parts. This scheme is also known as a *forward-mapped* page table. For systems with an address space larger than 32 bits, the hierarchical page table is generally inappropriate.
Hashed page tables. This approach is appropriate for address spaces larger than 32 bits. The hash value is the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location, to avoid collision. Each element consists of the virtual page number, the value of the mapped page frame, and a pointer to the next element in the linked list.
Inverted page tables. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Only one page table is in the system, and it has only one entry for each page of physical memory. Inverted page table usually require that an address-space identifier be stored in each entry of the table.
Diagram of the inverted page table.



Swapping. A process or a portion of the process can be swapped temporarily out of memory into a backing store. *Standard swapping* involves moving entire processes in and out of memory, including data structures associated with the process. Idle or mostly idle processes are good candidates for standard swapping. Most systems use a variation of swapping in which pages of a process can be swapped, referred to as *paging*. This approach is less costly. Swapping is generally not supported on mobile systems because mobile devices use flash memory rather than hard disks.

10: Virtual Memory

Virtual memory. The separation of user logical memory from physical memory. The logical address space can therefore be much larger than the physical address space. The requirement that instructions must be in physical memory to be executed, but an entire program is often not needed (e.g. code to handle unusual error conditions are almost never executed). Virtual address spaces that include holes are known as *sparse* address spaces.
Demand paging. An alternative strategy to loading programs is to load pages only as they are needed. This technique is commonly used in virtual memory systems. Pages that are never accessed are thus never loaded into physical memory. This system is similar to a paging system with swapping. In the extreme case, we can start executing a process with *no* pages in memory. This is known as *pure demand paging*.
Steps in handling a page fault.

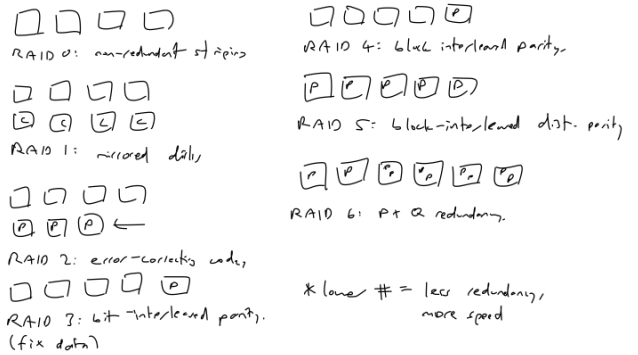


Sequence for servicing a page fault. Trap to the operating system, save registers and process state, determine that the interrupt was a page fault, check that the page reference was legal + determine location of page in secondary storage, issue a read from the storage to a free frame (wait in a queue until the read request is serviced, wait for the device seek/latency time, begin the transfer of the page), allocate the CPU core to some other process while waiting, receive an interrupt from the storage I/O subsystem, save the registers and process state for the other process, determine that the interrupt was from the secondary storage device, correct the page table and other tables to show that the desired page is now in memory, wait for the CPU core to be allocated to this process again, restore registers + process state + new page table, and resume the interrupt instruction.
Locality of reference. Programs tend to have this, which significantly lowers the likelihood o accessing several new pages of memory with each instruction, possibly causing multiple page faults per instruction.
Free-frame list. To resolve page faults, most operating systems maintain a free-frame list, a pool of free frames for satisfying such requests. Free frames must also be allocated when the stack or heap segments from a process expand. Operating systems typically allocate free frames using a technique known as zero-fill-on-demand. Zero-fill-on-demand frames are "zeroed-out" before being allocated, thus erasing their previous contents.
Effective access time with demand paging. (1 - page fault probability) * memory access time + page fault probability * page fault time.
Copy-on-write. The parent and child processes initially share the same pages. These shared pages are marked as copy-on-write pages. If either process writes to a shared page, a copy of the shared page is created.
Modify/dirty bit. When this scheme is used, each page or frame has a modify bit associated with it in the hardware. When we select a page for replacement, we examine its modify bit. If the bit is set, we must write the page to secondary storage when replacing it. Otherwise, we simply overwrite the page.
First-in, first-out (FIFO) page replacement. Simple page replacement algorithm which associates with each page the time that it was brought into memory.
Belady's anomaly. For some page replacement algorithms (FIFO), the page-fault rate may increase as the number of allocated frames increases. Giving more memory to a process does not necessarily improve its performance.
Optimal (OPT) page replacement. Replace the page that will not be used for the longest period of time. This algorithm requires knowing the future and is difficult to implement.
Least recently used (LRU) page replacement. Replace the page that has not been used for the longest period of time. This algorithm is often used as a page replacement algorithm, but may require substantial hardware assistance. There are two feasible implementations. Using *counters*, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for each memory reference. Using a *stack*, we keep a stack of page numbers. When a page is referenced, it is removed from the stack and put on the top. The least recently used page is always on the bottom of the stack. *Both OPT and LRU belong to a class of page replacement algorithms called stack algorithms, which don't suffer from Belady's anomaly.*
Reference bit. The reference bit for a page is set by the hardware whenever that page is referenced. This information is used to approximate LRU for systems which provide little hardware support.
Second chance page replacement algorithm. We can think of this algorithm as a circular queue. If the reference bit of a page is set to 1, we give the page a second chance by setting its reference bit to 0, and setting arrival time to the current time. If the reference bit is set to 0, the page is cleared.
Enhanced second chance algorithm. We consider both the reference bit and modify bit as an ordered pair. (0, 0) represents neither recently used nor modified, and is the best page to replace. (0, 1) represents not recently used but modified, and requires that the page be written out before replacement. (1, 0) represents recently used but clean, and will probably be used again soon. (1, 1) represents recently used and modified.
Counting-based page replacement. Alternative algorithms for page replacement keep a counter of the number of references that have been made to each page. For instance, *least frequently used (LFU)* requires that the page with the smallest reference count be replaced. This algorithm presents a problem when a page is used heavily during the initial phase of a process but is never used again. *Most frequently used (MFU)* is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
Fixed allocation. The simplest way to split frames among processes is to give every process an equal number of frames, or *equal allocation*. Remaining frames can be used as a free-frame buffer pool. Alternatively, we have *proportional allocation*, where we allocate available memory to each process according to its size.
Priority allocation. This is a proportional allocation scheme based on priorities rather than size. If a process generates a page fault, we either select for replacement one of its frames, or a frame from a process with a lower priority number.
Global and local replacement. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to another process. This results in varying process execution times but greater throughput. Local replacement requires that each process select from only its own set of allocated frames. This results in more consistent performance but possibly underutilized memory.
Thrashing. A process is thrashing if it spends more time paging than executing. We can fix thrashing by allocating more memory, or killing processes (reduce the degree of multiprogramming).
Locality model. As a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap.
Working-set model. This model uses a parameter Δ to define the working-set window. The set of pages in the most recent Δ page references is the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference.

11: Mass-Storage Management

Hard disk drives (HDDs). Each disk platter has a flat circular shape like a CD. We store information by recording it magnetically on the platters, and read information by detecting magnetic patterns on the platters. The surface of a platter is logically divided into circular *tracks*, which are subdivided into *sectors*. The set of tracks at a given arm position make up a *cylinder*. The transfer rate is the rate at which data flow between the drive and the computer. The

positioning time, or random-access time, consists of the time time necessary to move the disk arm to the desired cylinder (seek time), and the time necessary for the desired sector to rotate to the disk head (rotational latency).
Nonvolatile memory (NVM) devices. NVM devices can be more reliable than HDDs because they have no moving parts and can be faster, because they have no seek time or rotational latency. Additionally, they consume less power. Standard bus interfaces can cause a limit on throughput, leading to NVM devices which connect directly to the system bus (e.g. PCIe). NVM devices deteriorate with every write/erase cycle, and lifespan is measured in drive writes per day (DWPD). Algorithms to handle limitations are implemented in the NVM device controller.
Magnetic tape. This is a relatively permanent storage medium which can hold large quantities of data. Use cases include backup, storage of infrequently used data, or as a transfer medium between systems.
Connection methods. A secondary storage device is attached to a computer by the system bus or an I/O bus. Buses available include advanced technology attachment (ATA), serial ATA (SATA), eSATA, serial attached SCSI (SAS), universal serial bus (USB), and fibre channel (FC). NVM devices may use a special interface called NVM express.
FCFS scheduling. The simplest form of disk scheduling considers only the order of requests in a disk queue. *Variants of FCFS are the most common algorithms used on NVM devices.*
SSTF scheduling. This algorithm selects the request with the minimum seek time from the current head position.
SCAN scheduling. The arm starts at one end of the disk and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk.
C-SCAN scheduling. This variant of SCAN provides a more uniform wait time by returning to the beginning of the disk without servicing requests along the way. *Both SCAN and C-SCAN tend to perform better for systems that place a heavy load on the disk, and result in less starvation. However, starvation can still occur.*
LOOK and C-LOOK scheduling. In this variant of SCAN/C-SCAN, the arm only goes as far as the last request in each direction, then reverses direction immediately.
Low-level formatting. Before a storage device can store data, it must be divided into sectors that the controller can read and write. This process is called *low-level formatting*, or *physical formatting*. Low-level formatting fills the device with a special data structure for each storage location.
Steps for an operating system to record data structures. The first step is to partition the device into one or more groups. Each partition is treated as if it's a separate device. *Mounting* a file system is making the file system available for use by the system and its users. The second step is *volume creation and management*. This step is sometimes implicit as when a filesystem is placed directly within a partition. The third step is *logical formatting*, or creation of a filesystem. The OS store the initial filesystem structures onto the device, including maps of free and allocated space. To increase efficiency, most filesystems group blocks together into larger chunks, frequently called clusters.
Raw disk. Sometimes operating systems give special programs (e.g. databases) the ability to use a partition without any filesystem data structures. I/O to this device is termed *raw I/O*.
Boot block. For a computer to start running, it must have an initial program to run. The *bootstrap loader* can bring in a full bootstrap program from secondary storage. This program is stored in the *boot blocks* at a fixed location on the device. A device that has a boot partition is called a *boot disk* or *system disk*.
Host-attached storage. This is storage accessed through local I/O ports, including SATA.
Network-attached storage (NAS). This provides access to storage across a network. The storage unit is usually implemented as a storage array with software that implements the RPC interface.
Storage-area network (SAN). This is a private network (using storage protocols rather than networking protocols) connecting *servers* and *storage units*. Multiple hosts and multiple storage arrays can be attached to the same SAN. The storage arrays can be RAiD protected or unprotected.
Redundant arrays of independent disks (RAID). A variety of disk-organization techniques, collectively called RAID, are commonly used to address performance and reliability issues.
Mirroring. The simplest approach, called *mirroring*, is to duplicate every drive. The *mean time between failures (MBTF)* is dependent on the MBTF of the individual drives, as well as the *mean time to repair*.
Disk striping. This technique uses a group of disks as one storage unit. *Bit-level striping* is splitting the bits of each byte across multiple drives. In *block-level striping*, blocks of a file are striped across multiple drives. Other levels of striping, such as bytes of a sector or sectors of a block, are also possible.
RAID levels.



12: I/O Systems

I/O hardware. A device communicates with the machine via a connection point, or port (e.g. a serial port). A bus is a set of wires with a rigidly defined protocol that specifies a set of messages that can be sent. When device A has a cable that plugs into device B, which plugs into device C, which plugs into the computer, this arrangement is a *daisy chain*. A *controller* is a collection of electronics that can operate a port, a bus, or a device. A *serial-port controller* is a single chip in the computer that controls a serial port. By contrast, a fibre channel bus controller handles a complex protocol and is often implemented as a separate circuit board.
Memory-mapped I/O. The processor communicates with the controller by reading and writing bit patterns in the controller. This communication can occur through special I/O instructions or through *memory-mapped I/O*. In this case, the device-control registers are mapped into the address space of the processor. For instance, a graphics card controller has a large memory-mapped region to hold screen contents.
I/O device control registers. The *data-in register* is read by the host (driver) to get input. The *data-out register* is written by the host to send output. The *status register* contains bits that can be read by the host. The *control register* can be written by the host to start a command or to change the mode of a device.
Polling. The host repeatedly reads the busy bit (on the status bit) until the bit becomes clear. This means the host is in a loop, reading the status register over and over. This method is inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone.
Interrupts. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually through the system bus. The CPU immediately transfers execution to a fixed location which contains the starting address where the interrupt service routine is located.
Interrupt vector. A table of pointers to interrupt service routines for various devices, usually stored in low memory (usually the first 100 locations).
Interrupt-request line. A wire attached to the CPU which is sensed after every instruction. The interrupt number passed through the line acts as an index into the interrupt vector.
Nonmaskable and maskable interrupts. Most CPUs have *two* interrupt request lines. Nonmaskable interrupts are reserved for interrupts such as unrecoverable memory errors. The maskable line can be turned off by the CPU before execution of critical instruction sequences.
Interrupt chaining. Each element of the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, each handler of the corresponding list is called until one is found that can service the request.
Interrupt priority levels. These levels allow the CPU to defer handling of low-priority interrupts without masking all interrupts and enables the preemption of low-priority interrupts.
Direct memory access (DMA). This feature of computer systems allows hardware components to write to main memory without CPU intervention, or programmed I/O (one byte at a time). The OS writes a DMA command block to memory. The command block contains source and destination addresses, and the number of bytes to transfer. Then, the CPU writes the address of the command block to the DMA controller, and goes on with other work. Finally the DMA controller proceeds to operate on the bus directly. *Note that because a device and the CPU may access memory simultaneously, the memory controller must provide access to both in a fair manner. Therefore, CPU memory operations are slowed down during a DMA transfer.*
Scatter-gather method of DMA transfers. This allows multiple transfers to be initiated in a single DMA command.
Double buffering. The target address is usually stored in kernel address space. To get the DMA-transferred data to user space for thread access, a second copy operation from kernel to user memory is needed. This *double buffering* is inefficient, and operating systems have moved to using memory-mapping instead.
Cycle stealing. The DMA controller seizes the memory bus at the end of a transfer, preventing the CPU from accessing main memory. This can temporarily slow down CPU computation.
Direct virtual memory access (DVMA). Some computer architectures use virtual addresses for DMA, which undergo translation to physical addresses. DVMA can performa a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.
Application I/O interface. Devices may vary in many dimensions. A *character-stream device* transfers bytes one by one, while a *block device* transfers a block of bytes as a unit. A *sequential devices* transfers data in a fixed order determined by the device, whereas the user of a *random-access device* can instruct the device to seek to any of the available storage locations. A *synchronous device* performs data transfers with predictable response times, in coordination with other aspects of the system. An *asynchronous device* exhibits irregular or unpredictable response times not coordinated with other computer events. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot. Device speeds range from a few bytes per second to gigabytes per second. Some devices perform both input and output, but others support only one data transfer direction. Some allow data to be modified after write, but others can be written only once and are read-only thereafter.
More characteristics of I/O devices. I/O devices can be grouped by the OS into block I/O, character I/O (stream), memory-mapped file access, or network sockets.
Escape (back door). Allows for direct manipulation of I/O from an application.
Clocks and timers. A *programmable interval timer* is hardware to measure elapsed time and trigger operations. It can be set to wait a certain amount of time and generate an interrupt, and it can be set to do this once or repeatedly. Modern PCs include a *high-performance event timer (HPET)*.
Network devices. Many operating systems provide a *socket* interface that is different from the read-write-seek interface used for disks. This interface separates the network protocol from network operation, and includes *select* functionality, which replaces *read*.
Nonblocking and asynchronous I/O. When an application issues a *blocking* system call, the execution of the calling thread is suspended. *Nonblocking I/O* still "blocks" for a very short duration, but returns quickly. An example of a nonblocking call is an interface that receives keyboard and mouse input. An alternative to nonblocking calls is *asynchronous calls*, which return immediately without waiting for I/O to complete.
Kernel I/O subsystem. A *buffer* is stored memory when transferring between devices, to cope with device speed mismatch and device transfer speed mismatch. *Double buffering* is when there are two copies of the data between kernel and user. *Flow control* is another approach to indicate whether to slow down/speed up transfer. *Cached* data is always just a copy of data, and is sometimes combined with buffering.
Spooling. This approach holds output for a devices if device can only serve one request at a time (e.g. printing).
Device reservation. This approach gives exclusive access to a device; applications must be aware of deadlocks.

13: Filesystem Interface

File. A file is a collection of related information that is recorded on secondary storage. Commonly, files represent programs (both source and object forms) and data. A *text file* is a sequence of characters organized into lines. A *source file* is a sequence of functions, each of which is further organized as declarations followed by executable statements. An *executable file* is a series of code sections that the loader can bring into memory and execute.
File attributes. Name, identifier (unique tag which identifies the file within the file system), type, location (pointer to a device and the location of the file on the device), size (in bytes, words, or blocks), protection (access-control information, permissions, etc.), timestamps and user modification.
File operations. Creating a file, opening a file (returns a file handle), writing a file (specifying the open file handle and the information to be written using a write pointer), reading a file (using the open file handle and a read pointer), repositioning within a file (also known as a file seek), deleting a file (searching the directory for the named file, and releasing all file space), truncating a file (erasing the contents of a file but keeping its attributes).
Open-file table. The operating system keeps an *open-file table* containing information about all open files. Several pieces of information are associated with an open file, including file pointer, file-open count (tracks number of opens and closes), location of the file, and access rights.
Sequential access method. Information in the file is processed in order, one record after the other. This is the most common mode of access. Such a file can be reset to the beginning, or skipped forward and backward on some systems. This method is based on a tape model of a file and works well on sequential- and random-access devices.
Direct/relative access method. A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. For direct access, the file is a numbered sequence of blocks or records. Direct-access files are applicable for immediate access to large amounts of information, such as in a database.
Other access methods. On top of the direct access method, we can construct an index for the file, which contains pointers to the various blocks. With large files, the index may be too large to be kept in memory, so an index for the index file may be created.
Operations to be performed on a directory. Search for a file, create a file, delete a file, list a directory, rename a file, traverse the file system (access every directory and every file within a directory structure).
Single-level directory. All files are contained in the same directory. This approach has significant limitations, when the number of files increases or when the system has more than one user. For instance, all files must have unique names.
Two-level directory. There is a separate directory for each user. Each user has their own file directory, which list the files of a single user. A username and file name define a path name, which every file in the system has.
Tree-structured directories. We can extend the directory structure to a tree of arbitrary height. Each process has a current directory which contains files of interest to the process. For instance, the initial current directory of a user's login shell is designated when the user job starts or the user logs in. The *absolute* path name begins at the root and follows a path down to the specified file. The *relative* path name defines a path from the current directory.
Acyclic-graph directories. This approach allows directories to share subdirectories and files. A shared file or directory is not the same as two copies of the file. The most common way to implement shared files is through links, which are effectively pointers to another file or subdirectory. Problems with acyclic-graph directories include filesystem traversal and deletion. *In UNIX, links are left dangling if the original file is deleted.*

14: Filesystem Implementation

Filesystem structure. Disks provide most of the secondary storage on which file systems are maintained. This is because disks can be rewritten in place, and a disk can access directly any block of information it contains. I/O transfers between memory and mass storage are performed in units of *blocks*. *Filesystems* provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily.
Filesystem levels/layers. The *I/O control* level consists of device drivers and interrupt handlers to transfer information between main memory and the disk system. Its input consists of high-level commands such as "retrieve block 123," outputting low-level hardware-specific instructions used by the hardware controller. The *basic filesystem* needs to issue generic commands to the appropriate device driver to read and write blocks on the storage device. The *file-organization module* knows about files and their logical blocks, and includes the free-space manager. The *logical filesystem* manages metadata information, which include all filesystem structure except for actual data. Filesystem layering can introduce more operating system overhead, but comes with the benefits of abstraction.
Inode. A file control block, or *inode*, contains information about the file, such as ownership, permissions, and location.
Filesystem information on storage. A *boot control block* contains information needed by the system to boot an operating system from that volume. This is only needed if the volume contains an operating system. A *volume control block* contains volume details such as number of blocks, size of blocks, a free-block count, and more. A *directory structure* is used to organize the files. A per-file FCB (or inode) contains many details about the file, and may also be stored in the volume control block.
In-memory filesystem information. An in-memory mount table contains information about each mounted volume. An in-memory directory structure holds the directory information of recently accessed directories. The system-wide open file table contains a copy of the FCB for each open file, and other information. The per-process open file table contains pointers to appropriate entries in the system-wide open file table. Finally, buffers hold filesystem blocks when they are being read from or written to a filesystem.
Directory implementation. A directory may be implemented as a linear list of file names with pointers to data blocks, which is simple to program but is time-consuming to execute. A *hash table* may be used on top of the existing list, decreasing directory search time but introduces filename collision. Hash tables may only be used if entries are fixed size, or using the chained-overflow method.
Allocation methods. With *contiguous allocation*, each file occupies a set of contiguous blocks. This provides the best performance, but problems include finding space for the file, knowing the file size, and external fragmentation (introducing the need for compaction). With *linked allocation*, each file is a linked list of blocks. This schemes has no external fragmentation and works great for sequential access, but increases internal fragmentation and reliability issues, as well as decreasing performance (requiring many disk seeks). Finally, *indexed allocation* solves the problem of efficient direct access by bringing all pointers into an *index block*, which each file has.
Schemes for the index block. The *linked scheme* links together several index blocks in a linked list. The *multilevel index* uses a first-level index block to point to set of second-level index blocks. The *combined scheme* uses both.
Free-space list. The system maintains a free-space list to keep track of free disk space. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. Frequently, the free-space list is implemented as a *bitmap* or *bit vector*, where each block is represented by 1 bit. Another approach is to link together all free blocks, keeping a pointer to the first free block in a special location. Although traversing the list is inefficient, it is not a frequent operation. A modification of the linked list approach is *grouping*, which stores the address of the next free blocks in the first free block, plus a pointer to the next block containing free block pointers. Finally, the *counting* approach takes advantage of the fact that several contiguous blocks may be allocated or freed simultaneously. This approach keeps the address of the first free block and the count of following the free blocks; the free-space list then has entries containing addresses and counts.
Unified buffer cache. Memory-mapped I/O uses a page cache using virtual memory techniques and addresses. A *unified buffer cache* uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching.
Recovery. Methods of recovery include consistency checking (fix inconsistencies between directory structure with data blocks), as well as backup and restoration. *Log structured (journaling) file systems* record each metadata update to the file system as a transaction, similar to a database.

Appendix

Hex to binary.

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Decimal to hex.
Take the remainder of the decimal number when dividing by 16, substituting A-F for 10-15. Arrange in reverse order until the decimal number is 0.