Operating system Definitions:

The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can view a computer system as consisting of hardware, software and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating simply provides an environment within which other programs can do useful work.

OS : user’s view of the computer

The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization—

From the computer’s point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator such as CPU time, memory space, file-storage space, I/O devices.

A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices.

The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

operating system is the one program running at all times on the computer—usually called the kernel. (Along with the kernel, there are two other types of programs: system programs, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system

Mobile operating systems often include not only a core kernel but also middleware—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent

mobile operating systems—Apple’s iOS and Google’s Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name a only few).

computer powered up (reboot)

it needs to have an initial program(bootstrap program) to run.

it is stored within the computer hardware in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term firmware.

It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system.

the bootstrap program must locate the operating-system kernel and load it into memory

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes, or system daemons that run the entire time the kernel is running.

On UNIX, the first system process is “init,” and it starts many other daemons.

Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

The occurrence of an event is usually signalled by an interrupt from either the hardware or the software.

Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.

Software may trigger an interrupt by executing a special operation called a system call

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located.

The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler.However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of

pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low

memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or interrupt vector, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device

The interrupt architecture stores the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state—for instance, by modifying

register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

Memory:

computers run most of their programs from rewritable memory, called main memory (RAM)

Because ROM cannot be changed, only static programs, such as the bootstrap program are stored there.

EEPROM can be changed but cannot be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

A typical instruction–execution cycle

first fetches an instruction from memory and stores that instruction in the instruction register.

The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register.

After the instruction on the operands has been executed, the result may be stored back in memory.

memory unit does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or

what they are for (instructions or data).

we want the programs and data to reside in main memory permanently. But this is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a volatile storage device that loses its contents when power is turned off.

Thus, most computer systems provide secondary storage

The most common secondary-storage device is a magnetic disk

Others include cache memory, CD-ROM, magnetic tapes, and so on.

Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility.

I/O Structure

Storage is only one of many types of I/O devices within a computer. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus.

Each device controller is in charge of a specific type of device. Depending on the controller, more than one device may be attached. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) controller.

A device controller maintains some local buffer storage and a set of special-purpose registers.

The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

operating systems have a device driver for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as “read a character from the keyboard”). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read.

For other operations, the device driver returns status information.

interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, direct memory access (DMA) is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU.

Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective.

The I/O subsystem consists of several components:

* A memory-management component that includes buffering, caching, and spooling
* A general device-driver interface
* Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

**System Architecture**

Single-Processor Systems

On a single processor system, there is one main CPU capable of executing a general-purpose instruction set including instructions from user processes all single processor systems have other special-purpose processors as well like DMA.

All of these special-purpose processors run a limited instruction set and do not run user processes.

All of these special-purpose processors run a limited instruction set and do not run user processes.

In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously.

Multiprocessor Systems

Multiprocessor systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

* Increased throughput.
* Economy of scale.
* Increased reliability.

fault tolerant, they can suffer a failure of any single component and still continue operation.

Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

The HP NonStop (formerly Tandem) system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of multiple pairs of CPUs, working in lockstep. Both processors in the pair execute each instruction and compare the results. If the results differ, then one CPU of the pair is at fault, and both are halted. The process that was being executed is then moved to another pair of CPUs, and the instruction that failed is restarted.

multicore: multiple computing cores on a single chip.

They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips.

while multicore systems are multiprocessor systems, not all multiprocessor systems are multicore each core has its own register set as well as its own local cache. Other designs might use a shared cache or a combination of local and shared caches

**Deadlock**

A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

Deadlock problems can only become more common, given current trends, including

* Larger numbers of processes,
* Multithreaded programs,
* Many more resources within a system, and
* An emphasis on long-lived file and database servers rather than batch systems.

A process utilizes a resource in the following sequence:

* Request: The process requests the resource.
  + If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
* Use: The process can operate on the resource
* Release: The process releases the resource.

The request and release of resources are system calls. Examples are the request() and release() device, open() and close() file, and allocate() and free() memory system calls

For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource.

A **system table** records whether each resource is free or allocated.

For each resource that is allocated, the table also records the process to which it is allocated.

If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

A deadlock can occur only if **four** necessary conditions hold **simultaneously** in the system:

* **Mutual exclusion**: resource must be non-sharable.
  + i.e. only one process at a time can use the resource.
* **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources
* **No pre-emption**: a resource can be released voluntarily by the process holding it, after that process has completed its task.
* **Circular wait:** A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

**To prevent deadlocks**, we must ensure that at least one of the necessary conditions never holds.

There are three principal methods for dealing with deadlocks:

* Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
  + Operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs.
* Allow the system to enter a deadlocked state, detect it, and then recover.
* Ignore the problem altogether and pretend that deadlocks never occur in the system.
  + This solution is used by most operating systems, including Linux and Windows.

**Avoiding deadlocks:** banker’s algorithm

A method for avoiding deadlocks, rather than preventing them, requires that the operating system have prior information about how each process will utilize system resources.

The banker’s algorithm, for example, requires prior information about the maximum number of each resource class that each process may request. Using this information, we can define a deadlock avoidance algorithm.

If a system does not employ a protocol to ensure that deadlocks will never occur, then a detection-and-recovery scheme may be employed.

A deadlock detection algorithm must be invoked to determine whether a deadlock has occurred.

If a deadlock is detected, the system must recover either by terminating some of the deadlocked processes or by preempting resources from some of the deadlocked processes.

Where preemption is used to deal with deadlocks, three issues must be addressed: selecting a victim, rollback, and starvation.

In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur, and the selected process can never complete its designated task.

Researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems.

The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

**Resource-Allocation Graph**

Deadlocks can be described in terms of a directed graph called a system resource-allocation graph.

This graph consists of a set of vertices V and a set of edges E.

The set of vertices V is partitioned into two different types of nodes:

P = {P1, P2, ..., Pn}, the set consisting of all the active processes in the system, and

R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

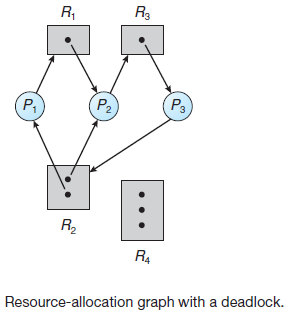
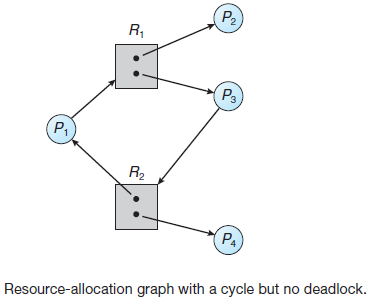
Pi ? Rj(request edge) it signifies that process Pi has requested an instance of resource type Rj and

is currently waiting for that resource.

Rj ? Pi(assignment edge) ; it signifies that an instance of resource type Rj has been allocated to process Pi .

Given the definition of a resource-allocation graph, it can be shown that,

* If the graph contains no cycles, then no process in the system is deadlocked.
* If the graph does contain a cycle, then a deadlock may exist.
  + If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.



**Recovery From Deadlock**

Once a deadlock exists there are two options for breaking a deadlock.

**Process Termination**

Abort one or more processes to break the circular wait and reclaims all resources allocated to the terminated processes.

* Abort all deadlocked processes
  + Very expensive as the deadlocked processes may have computed for a long time, which must be discarded and probably will have to be recomputed later.
* Abort one process at a time until the deadlock cycle is eliminated.
  + Overhead: after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Issues with Aborting any process

* If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
* If the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Many factors for choosing the victim (process to abort):

* Priority of the process
* How long the process has computed and
* How much longer the process will compute
* How many and what type of resources the process has used (for example,
* Whether the resources are simple to preempt)
* How many more resources the process needs in order to complete
* How many processes will need to be terminated
* Whether the process is interactive or batch

**Resource Preemption**

Successively pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Issues need to be addressed:

* Selecting a victim to minimize cost.
* Rollback. What should be done with victim process?
  + We must roll back the process to some safe state and restart it from that state.
* Starvation. Victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim.
  + We must ensure that a process can be picked as a victim only a (small) finite number of times.