An OS is a program that acts as an intermediary between a user of a computer and the computer hardware

**OS is:**

**Resource allocator**: decides between conflicting requests for efficient and fair resource use

**Control program**: controls execution of programs to prevent errors and improper use of computer

**Kernel**: the one program running at all times on the computer

**Bootstrap program**: loaded at power-up or reboot stored in ROM or EPROM to load kernel

**I/O and CPU can execute concurrently**

Device controllers inform CPU that it is finished w/ operation by causing an interrupt. Device controllers have buffers to store data

Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines

Incoming interrupts are disabled while another interrupt is being processed

**Trap** is a software generated interrupt caused by error or user request

OS determines which type of interrupt has occurred by **polling** or the **vectored** interrupt system

**Polled interrupts do not specify which device generated an interrupt while a vectored interrupt specifies which device generated an interrupt**

**System call**: request to the operating system to allow user to wait for I/O completion

Device-status table: contains entry for each I/O device indicating its type, address, and state

OS indexes into the I/O device table to determine device status and to modify the table entry to include interrupt

**Main memory** – random access, volatile

**Secondary storage** – extension of main memory that provides large non-volatile storage

**Disk** – divided into tracks which are subdivided into sectors. Disk controller determines logical interaction

**Caching** – copying information into faster storage system

Multiprocessor Systems: Increased throughput, economy of scale, increased reliability. **Can be asymmetric or symmetric**

**Context Switch:** Done when CPU switches from one process to another. Saves state of currently executing process and loads state of process to be executed

Timesharing – CPU switches jobs so frequently that each user can interact with each job while it is running (interactive computing)

Dual-mode operation allows OS to protect itself and other system components – User mode and kernel mode

**Protection** – mechanism for controlling access of processes or users to resources defined by the OS

**Security** – defense of a system against attacks

User IDs (UID), one per user, and Group IDs, determine which users and groups of users have which privileges

User Interface (UI) – Can be Command-Line (CLI) or Graphics User Interface (GUI) or Batch. These allow for the user to interact with the system services via system calls (typically written in C/C++). Other system services that a helpful to the user include: program execution, I/O operations, file-system manipulation, communications, and error detection

OS operation services are: resource allocation, accounting, protection and security

Most system calls are accessed by Application Program Interface (API) such as Win32, POSIX, and Java

Each system call is associated with a unique number and the System call interface maintains a table indexed according to these numbers

Parameter passing can be done in registers, address of parameter stored in a block, pushed onto the stack by the program and popped off by the OS

**Process control system calls**: end, abort, load, execute, create/terminate process, wait, and allocate/free memory

**File management system calls**: create/delete files, open/close file, read, write, get/set attributes

**Device management system calls**: request/release device, read, write, and logically attach/detach devices

**Information maintenance system calls**: get/set time, get/set system data, get/set process/file/device attributes

**Communications system calls**: create/delete communication connection, send/receive, and transfer status information

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface such that each uses functions (operations) and services of only lower-level layers

Virtual machine: uses layered approach, treats hardware and the OS kernel as though they were all hardware. **Host creates the illusion that a process has its own processor and own virtual memory**

**Application failures** can generate core dump file capturing memory of the process

**Operating system failure** can generate crash dump file containing kernel memory

Each process contains a program counter, stack, and data section.

**Text section**: program code itself

**Stack**: temporary data (function parameters, return addresses, local variables)

**Data section**: global variables

**Heap**: contains memory dynamically allocated during run-time

**Process Control Block (PCB)**: contains information associated with each process: process state, PC, CPU registers, scheduling information, accounting information, I/O status information

**Types of processes:**

**I/O Bound**: spends more time doing I/O than computations, manyshort CPU bursts

**CPU Bound**: spends more time doing computations, few verylong CPU bursts

**When CPU switches to another process, the system must save the state of the old process (to PCB) and load the saved state (from PCB) for the new process via a context switch**

◦ **Fork** () system call creates new process

▪ **Exec** () system call used after a fork to replace the processes' memory space with a new program

**Advantages of multiprocessing:** include:

Increased throughput (more work done in less time)

Economy of scale (less cost due to peripheral sharing)

Increased reliability (failure of one processor will not halt the system)

**Inter process communication (IPC)**: shared memory or message passing

Message passing may be blocking or non-blocking

**Blocking** is considered synchronous. Blocking send has the sender block until the message is received. Blocking receive has the receiver block until a message is available

**Non-blocking** is considered asynchronous. Non-blocking send has the sender send the message and continue. Non-blocking receive has the receiver receive a valid message or null

**Multi-threading models**: Many-to-One, One-to-One, Many-to-Many

**Many-to-One**: Many user-level threads mapped to single kernel thread

**One-to-One**: Each user-level thread maps to kernel thread

**Many-to-Many**: Many user-level threads mapped to many kernel threads

Processes have copies of all code and data whereas for Threads, code and data are shared

**Two models of IPC**

**-Shared memory:** Process share a block of memory. Writing and reading to the block should be mutually exclusive i.e one at a time

**-Message passing:** Processes share a mailbox to send and receive messages. Message passing may be either blocking or non-blocking

**Blocking** is considered **synchronous**

**Blocking send** has the sender block until the message is received

**Blocking receive** has the receiver block until a message is available

**Non-blocking** is considered **asynchronous**

**Non-blocking** send has the sender send the message and continue

**Non-blocking** receive has the receiver receive a valid message or null

Issues include: thread cancellation, signal handling (synchronous/asynchronous), handling thread-specific data, and

**Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue**

**Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU**

To solve critical section problem each process must ask permission to enter critical section in entry section, follow critical section with exit section and then execute the remainder section

**Peterson's Solution**: solution for two processes

Two processes share two variables: int turn and Boolean flag [2]

Turn: whose turn it is to enter the critical section

Flag: indication of whether or not a process is ready to enter critical section

Flag[i] = true indicates that process Pi is ready

◦ Algorithm for process Pi:

do {

flag[i] = TRUE;

turn = j;

while (flag[j] && turn == j)

//critical section

flag[i] = FALSE;

remainder section

} while (TRUE);

Modern machines provide atomic hardware instructions: Atomic = non-interruptible

Solution using Locks:

do {

acquire lock critical section

release lock

remainder section

} while (TRUE);

Solution using Test-And-Set: Shared Boolean variable lock, initialized to FALSE

Solution using Swap: Shared bool variable lock initialized to FALSE; Each process has local bool variable key

**Semaphore**: Synchronization tool that does not require busy waiting. Standard operations: wait () and signal () ← these are the only operations that can access semaphore

**Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (most OSes do not prevent or deal with deadlocks).** Can cause starvation and priority inversion (lower priority process holds lock needed by higher-priority process)

Monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization. Java adds inbuilt monitors

**Only one process may be active within the monitor at a time. Contains wait and signal functions to notify control process synchronization**

Process execution consists of a cycle of CPU execution and I/O wait

CPU scheduling decisions take place when a process:

Switches from running to waiting (nonpreemptive)

Switches from running to ready (preemptive)

Switches from waiting to ready (preemptive)

Terminates (nonpreemptive)

**The dispatcher module gives control of the CPU to the process selected by the short-term scheduler**

**Dispatch latency**- the time it takes for the dispatcher to stop one process and start another

**PROCESS SCHEDULING**

FCFS, SJF, Shortest-Remaining-Time-First (preemptive SJF), Round Robin, Priority

**Determining length of next CPU burst: Exponential Averaging:**

tn = actual length of nth CPU burst

τn+1 = predicted value for the next CPU burst

α, 0 ≤ α ≤ 1 (commonly α set to 1/2)

Define: τn+1 = α\*tn + (1-α)τn

Priority Scheduling can result in starvation, which can be solved by aging a process (as time progresses, increase the priority)

In Round Robin, small time quantum’s can result in large amounts of context switches

Time quantum should be chosen so that 80% of processes have shorter burst times that the time quantum

Multilevel Queues and Multilevel Feedback Queues have multiple process queues that have different priority levels

In the Feedback queue, priority is not fixed → Processes can be promoted and demoted to different queues as they wait in the queue

Feedback queues can have different scheduling algorithms at different levels

Multiprocessor Scheduling is done in several different ways:

Asymmetric multiprocessing: only one processor accesses system data structures → no need to data share

Symmetric multiprocessing: each processor is self-scheduling (currently the most common method)

Processor affinity: a process running on one processor is more likely to continue to run on the same processor (so that the processor's memory still contains data specific to that specific process)

**CPU utilization** – keep the CPU as busy as possible

**Throughput** – # of processes that complete their execution per time unit

**Turnaround time** – amount of time to execute a particular process

**Waiting time** – amount of time a process has been waiting in the ready queue

**Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

**Preemptive**: CPU is emptied/ stopped from executing current process

**POSIX:** **is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.**

**MULTIPROCESS: C**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main (int args, char \*argv[])

{

pid\_t fork\_return;

pid\_t pid;

pid=getpid();

fork\_return = fork ();

// When fork () returns -1, an error happened.

If (fork\_return==0)

// When fork () returns 0, we are in the child process.

{

printf(“\n\nThe values are Child ID = %d, Parent ID=%d\n”, getpid(), getppid());

execl(“/bin/cat”, “cat”, “-b”, “-v”, “-t”, argv[1], 0);

}

else

// When fork () returns a positive number, we are in the parent process

// and the return value is the PID of the newly created child process.

{

wait(NULL);

printf(“\nChild Completes “ );

printf(“\nIn the Parent Process\n”);

printf(“Child Id = %d, Parent ID = %d\n”, getpid(), getppid());

}

return 0;

}

**Processor affinity** – process has affinity for processor on which it is currently running

**Soft affinity: Process can go to any processor**

**Hard affinity: Process can only go to a specific processor**