Introduction to Operating Systems

An operating system is a software that manages a computer's hardware. It provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.

A computer can be divided into 4 main components

1. Hardware
2. Operating System
3. Application Programs
4. User

The user's view of a computer varies with the interface. In this case, the OS is designed mostly for ease of use, with some attention to performance and security, and no attention to resource utilisation.

The system's point of view is that the OS is a resource allocator, and manages all the different resources on offer. It emphasizes the need to control the various I/O devices and user programs, and behaves like a control program.

The operating system is a program running at all times on a computer, called a kernel. Along with a kernel, system programs and application programs work with the kernel to provide user experience to the user.

Virtualisation

The primary way through which the operating system operates is virtualisation. That is, the OS takes a physical resource and transforms it into a more general, easy-to-use virtual form of itself. The OS is hence, sometimes referred to as a virtual machine.

The OS provides APIs to programs to help them use the resources of the computer. Because virtualisation allows many programs to run concurrently on the CPU, the OS manages the resources for these programs.

Turning the CPU into a seemingly infinite number of CPUs and thus, allowing multiple programs to run as if they are using the entirety of the CPU is called virtualizing the CPU.

Memory in modern systems is just an array of bytes, and to read or write memory, one must specify the address of where the relevant memory resides. Each process accesses its own private virtual address space, and the OS maps this address space to the physical addresses in the machine. This abstraction is called virtualizing the memory.

Concurrency

There exists a problem of concurrency when programs share memory with each other due to their non-atomic nature of execution. This is handled by the OS using policies that provide rules to which part of the program executes when.

Persistence

DRAMs in systems store memory in a volatile manner, i.e, the memory is lost when the system is powered off. Hardware and software is thus critical to store memory persistently, and is usually some kind of an I/O device.

The software in the OS that usually manages the disk is called a filesystem, which is responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system.

Unlike the abstractions provided by the OS, the OS does not create a private, virtualized disk for storage disk. It assumes that users will want to share information that is in these files. It uses the system calls of open(), write(), read(), close(), etc to interact with the filesystem.

To handle the problems with the filesystem in times of system crashes, systems usually implement a write protocol such as journaling or copy-on-write, that allows the system to recover to a reasonable state afterwards.

Abstraction: The Process

A process is a simple running program, created in lieu of implementing an abstraction to user commands. Time sharing in the CPU allows users to run as many concurrent processes as they would like, at the cost of performance. Space sharing may also be done, where a resource is divided in space among those who wish to use it.

Policies are algorithms that make decisions in the OS for process control.

The memory a process can address is called its address space. Registers are also part of the process' state, which facilitate the read/write and execution of the process' instructions.

Usually, high level policies and low-level mechanisms allow the OS to change policies without having to rethink the mechanism, and thus gives modularity.

A process is more than just information code. It is divided into several parts:

1. Text Section - Code of the process
2. Program counter - Current activity
3. Processor registers
4. Process Stack - Temporary data
5. Data Section - Global variables
6. Heap - Dynamically allocated memory

A program is a passive entity, stored as an executable file in memory. In contrast, a process is an active entity that executes instructions. A program becomes a process when it is brought into memory.

As a process executes, it changes state. The state of a process is defined by the current activity of that process. A process may be in one of the following states

1. New - Creating a process
2. Running - Instructions being executed
3. Waiting - Process is waiting for some event
4. Ready - Process is waiting to be assigned to a processor
5. Terminated - Process has finished execution

Each process is represented in the OS by a process control block or a task control block (PCB). It contains information about each process such as

1. Process state
2. Program counter
3. CPU registers
4. CPU scheduling information - process priority, pointers to scheduling queues, etc.
5. Memory management information - base and limit, page tables, etc.
6. Accounting information - amount of CPU used, time limits, etc.
7. I/O status information - list of I/O devices allocated

The Process API

1. **Fork**

The fork() system call is used to create a new process. fork() returns the process identifier (PID) of the process calling the fork(). The process created is an almost-exact copy of the calling program. The newly created process is called the child process and has its own copy of the address space, its own registers, PC, and so forth. The child process always returns a value of 0 to the fork call.

2. **Wait**

The wait() system call forces the parent to wait() for the child process to finish, adding determinism to how the parent and child execute together.

3. **Exec**

The exec() system call is used to run a different program as opposed to the calling program. It loads the code from the executable sent as an argument and overwrites its code segment and re-initializes all the current segments.

A system shell is designed like this, where when a new command needs to be executed, it is executed in the child using the fork call, and the calling process is restored after the child is done executing.

4. **Pipe**

The pipe() system call is used to connect the output of one process to the output of another, allowing a chain of commands to be executed in a string.

Introduction to Scheduling

Workload Assumptions

The processes running at a given time are collectively called the workload. The workload assumptions we make here are

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU
4. The runtime of each job is known

Scheduling Metrics

We also define metrics for scheduling, called the turnaround time, which is defined as the time at which the job completes its execution minus the time it arrived into the system. For now, the turnaround time is just the time of completion as we have assumed that all jobs arrive at a time $t = 0$.

This is a performance metric, where a fairness metric can be used as well, which provides a tradeoff between performance and fairness to processes.

The objective of multiprogramming is to maximise CPU utilisation. A process is executed till it is made to wait for some I/O request or so. Since several processes are taken in memory, while the process waits, another process executes some portion of its code.

Process execution consists of a cycle of CPU execution and I/O wait. Process execution begins with a CPU burst followed by an I/O burst, and the cycle repeats.

Whenever the CPU becomes idle, the OS must select one of the processes in the ready queue to be executed, which is decided by the CPU scheduler or the short-term scheduler.

Scheduling decisions may take place in the following cases

1. When a process switches from a running state to waiting state.
2. When a process switches from a running state to ready state.
3. When a process switches from a waiting state to ready state.
4. When a process terminates.

In non-preemptive scheduling, once the CPU has been allocated to a process, the process holds the CPU until the end of process execution or till it encounters a waiting state.

Preemptive scheduling allows processes to be preempted based on conditions such as priority and time, but can result in race conditions. Preemption also affects the design of the kernel, when processes preempt kernel processes doing important tasks. This problem is handled by the OS by waiting for a system call to complete or for an I/O block to take place before a context switch. This model is simple, but does not support real-time computing well, when tasks must be completed within a time frame.

Interrupts need to be handled carefully and be guarded from simultaneous use. To enable only single access, interrupts are disabled at entry and enabled at exit.

The dispatches is the module that gives control of the CPU to the process selected by the short-term scheduler. Its functions are

1. Switching context
2. Switching to user mode
3. Jumping to proper location in the user program for restarts

The time taken by the dispatcher to preempt a process with another is called the dispatch latency.

Scheduling Criteria

To compare two scheduling algorithms, we consider

1. CPU Utilisation
2. Throughput - Number of processes completed per time unit
3. Turnaround time
4. Waiting time - Amount of time spent in the waiting queue
5. Response time - Time between first response and request

It is desirable to maximise 1 and 2, and minimise the rest, but mostly we optimize the average measure. These have also been replaced to minimise the variance these days.

Scheduling Algorithms

First Come, First Served

With this scheme, the process that requests the CPU first is allocated the CPU first. This is managed by a FIFO queue. When a process enters the ready queue, the PCB is linked to the tail of the FIFO queue and when the CPU is free, the head of the queue is executed. This is a non-preemptive form of scheduling.

Advantages

1. Simple

Disadvantages

1. Waiting time under FCFS is quite long
2. Convoy effect occurs in dynamic scheduling where a big process refuses to get off the CPU, resulting in lower CPU utilisation.
3. Troublesome for time-shared systems as it is non-preemptive.

A Gantt chart represents a schedule of processes including the start and finish times.

Shortest Job First

This algorithm associates with each process the length of the process' next CPU burst. The one with the smallest CPU burst is assigned the CPU first. If the bursts are same FCFS is used.

Advantages

1. Optimal as minimum average waiting time is obtained

Disadvantages

1. Process burst time is not known and must be estimated
2. Cannot be implemented with short term scheduling

The next CPU burst is usually predicted as an exponential average of the measured lengths of the previous CPU bursts as

$T(n+1) = a*t(n) + (1-a)*T(n)$

Where T denotes the predicted value and t denotes the length of the CPU burst.

This algorithm can be preemptive or non-preemptive. Preemptive SJF is called Shortest Remaining Time First scheduling.

Priority Scheduling

The SJF algorithm is a special case of priority scheduling. Each process is associated with a priority, and the CPU is allocated to the process with the highest priority. Priorities can be defined internally or externally. Internally defined priorities use some measurable quantity to compute the priority, whereas external priorities depend on external criteria.

Priority scheduling can be preemptive or non-preemptive.

A major problem with this scheduling is indefinite blocking or starvation, where low priority processes can be left waiting indefinitely. A solution to this problem is to use aging, that gradually increases the priority of each waiting process at each cycle.

Round Robin Scheduling

This algorithm is designed for time shared systems, based on preemption. A small unit of time, called a time quantum or a time slice is defined. The ready queue is treated like a circular queue and each process is given a time slice to execute, after which it is preempted for the next process.

The average waiting time in this type of scheduling is long, as long processes are executed in short bursts.

The performance of this algorithm depends heavily on the size of the time quantum chosen. If the time quantum is extremely large, it is just FCFS. If it is very small, it results in a large number of context switches. Thus the quantum must be large with respect to the context switch size.

Multilevel Queue

A division is made between foreground and background processes. A multilevel queue partitions the ready queue into these two separate queues, and processes are permanently assigned to one of these queues based on their properties. Each queue implements its own scheduling algorithm.

A priority is assigned to each queue and each queue has absolute priority over lower priority queues. Each queue gets a certain portion of CPU time in which it executes. This is a preemptive type of scheduling.

Multilevel Feedback Queue

It uses the same technique as a multilevel queue, but allows processes to switch queues. If a process uses too much CPU time, it is moved to a lower priority queue. A process that waits to long in a low priority queue is moved to a higher priority queue. A multilevel queue is thus defined by the following parameters

1. Number of queues
2. Scheduling algorithm for each queue
3. Method used to determine when to upgrade a process
4. Method used to determine when to demote a process
5. Method used to determine which queue a process enters

This is the most complex algorithm.

Operating System Examples

1. **Linux**
    a. Runs a variation of traditional UNIX scheduling algorithm
    b. Algorithm did not support systems with multiple processors
    c. Resulted in poor performance when number of processes were large
    d. Started using a O(1) algorithm that ran in constant time
    e. Provided processor affinity and load balancing
    f. Poor response time on interactive processes
    g. Now uses a Completely Fair Scheduling scheme
        i. Based on scheduling classes
        ii. Each class assigned a priority
        iii. Kernel accommodates different scheduling algorithms based on class
        iv. Relative priority assigned based on a “nice” value
        v. Identifies a target latency as the time in which every runnable task must run at least once.

vi. Maintains a virtual run time for how long each task ran
vii. This runtime is associated with a decay factor based on priority, where low priority tasks have higher decay
viii. Preemption is possible

2. Windows

a. Ensures that a highest priority thread always runs
b. Dispatcher handles scheduling
c. Thread selected by dispatcher runs until it is preempted by a higher priority thread, etc.
d. Uses a 32 level priority scheme to determine the order of execution
e. Variable class has priorities 1 to 15 and realtime has the rest
f. Thread running at priority 0 is used for memory management
g. Uses a queue for each priority level
h. The different priority classes are
    i. IDLE_PRIORITY
    ii. BELOW_NORMAL_PRIORITY
    iii. NORMAL_PRIORITY
    iv. ABOVE_NORMAL_PRIORITY
    v. HIGH_PRIORITY
    vi. REALTIME_PRIORITY
i. Priorities can be altered as well
j. A thread within a priority class can have priorities as well
    i. IDLE
    ii. LOWEST
    iii. BELOW_NORMAL
    iv. NORMAL
    v. ABOVE_NORMAL
    vi. HIGHEST
    vii. TIME_CRITICAL
k. When a thread's time runs out, the thread is interrupted and if it's in the variable-priority class, its priority is lowered. Lowering the priority limits CPU consumption of bound threads.
l. When the thread is released from wait, its priority is boosted
m. This gives good response time to interactive threads to keep the I/O devices busy
n. User Mode Scheduling (UMS) was introduced to allow applications to create and manage threads independently of the kernel.
o. Fibers were used to map several user mode threads to a single kernel thread, but were of limited practical use because

    i. Not allowed to make calls to the windows API because they had to share their thread environment block.
    ii. This presented a problem if the windows API placed information about different threads in the different blocks.
p. UMS is not directly used by programmers
q. Windows provides Concurrency Runtime (ConcRT) for task based parallelism.

3. Solaris

a. Uses priority based thread scheduling
    i. Time Sharing
    ii. Interactive
    iii. Realtime
    iv. System
    v. Fair share
    vi. Fixed priority
b. Default scheduling class is time sharing
c. Multilevel feedback queues are followed for each priority class
d. Higher the priority, smaller the time size
e. The dispatch table used has the following fields
    i. Priority
    ii. Time quantum
    iii. Time quantum expired
    iv. Return from sleep
f. Realtime class has the highest priority
g. Uses system class to run kernel threads
h. Fair Share scheduling uses CPU shares instead of priorities to schedule processes
i. Global priority is taken and the process is run