Operating System:

<http://web.cecs.pdx.edu/~walpole/class/cs510/fall2017/slides/1.pdf>

What is an Operating system?

* A concurrent program that controls the execution of application program and implements an interface between them and the computer hardware.
* It is an abstract machine that hides complex details of the underlying hardware by providing API to applications. Challenges include what to hide and what to expose, defining abstractions, efficient mapping to hardware.
* Resource manager, control access to shared resources, implements global allocation policies. Challenges are concurrency, security, performance.
* OS needs help from hardware to detect and prevent certain activities, and to enforce allocations.

Types of OS:

<https://www.geeksforgeeks.org/operating-system-types-operating-systems-awaiting-author/>

-->Batch OS:

This type of operating system do not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.

-->Time sharing OS:

Each task has given some time to execute, so that all the tasks work smoothly. Each user gets time of CPU as they use single system. These systems are also known as Multitasking Systems. The task can be from single user or from different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to next task.

-->Distributed OS:

These types of operating system is a recent advancement in the world of computer technology and are being widely accepted all-over the world and, that too, with a great pace. Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as loosely coupled systems or distributed systems. These systems processors differ in sizes and functions. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.

-->Network OS:

These systems runs on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that’s why these computers are popularly known as tightly coupled systems.

Real-Time OS:

These types of OSs serve the real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called response time.

Real-time systems are used when there are time requirements are very strict like missile systems, air traffic control systems, robots etc.

Functions of OS:

Security, Control over system performance, Error detecting aids, coordination between other software and users, memory management, processor management, device management, file management.

Multiprogramming – A computer running more than one program at a time (like running Excel and Firefox simultaneously).

Multiprocessing – A computer using more than one CPU at a time. Multitasking – Tasks sharing a common resource (like 1 CPU).

Multithreading is an extension of multitasking

system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system’s kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

<https://www.geeksforgeeks.org/operating-system-introduction-system-call/>

Program Counter(PC): Holds memory address of the next instruction.

Instruction Register(IR): Holds the instruction currently being executed.

General purpose registers: Hold variable and temporary results.

Arithmetic and Logic Unit (ALU) : performs arithmetic functions and logic operations.

Stack Pointer (SP) : holds memory address of a stack top , one frame for parameters & local variables of each active procedure.

Status Register: A word full of control flags/bits, Includes the mode bit to determine whether the CPU will execute privileged instructions.

Program Execution:

The Fetch/Decode/Execute cycle

- fetch next instruction pointed to by PC

- decode it to find its type and operands

- execute it

- repeat

How Can an OS Run Applications?

The OS must load the address of the application’s starting instruction into the PC

Example:

- computer boots and begins running the OS

- OS code must get into memory somehow

- fetch/decode/execute OS instructions

- OS requests user input to identify application program/file

- OS loads application (executable file) into memory

- OS loads the address of the app’s first instruction into the PC

- CPU fetches/decodes/executes the application’s instructions

How Can the OS Regain Control?

- OS needs something to interrupt the CPU and load OS instructions again

- interrupts can be generated from a timer device

- OS must register a future timer interrupt before handing control of the CPU over to an application

- When the timer interrupt goes off the hardware starts running the OS at a pre-specified location called an interrupt handler

Interrupt: Hardware event that loads an address of an instruction into the PC . Disabling interrupts must be a privileged instruction that is not executable by applications.

Interrupt handler: An instruction sequence that has been associated with an interrupt, Address of first instruction is registered with the interrupt hardware by filling in an interrupt vector.

The CPU knows whether or not to execute privileged instructions based on the value of the mode bit in the status register.

Privileged instructions are only executed if the mode bit is set - attempted execution in non-privileged mode generally causes an interrupt (trap) to occur.

How Can Applications Invoke the OS?

Special trap instruction causes a kind of interrupt

- changes PC to point to a predetermined OS entry point instruction

- simultaneously sets the mode bit

- CPU is now running in privileged mode Application calls a library procedure that includes the appropriate trap instruction fetch/decode/execute cycle begins at a pre-specified OS entry point called a system call handler

What are Privileged Instructions?

The Instructions that can run only in Kernel Mode are called Privileged Instructions .

<https://www.geeksforgeeks.org/operating-system-privileged-and-non-privileged-instructions/>

What is the difference between system call and function call?

What is the difference between System Call and Function Call? System call is a call to a subroutine built in to the system, while a function call is a call to a subroutine within the program. Unlike function calls, system calls are used when a program needs to perform some task, which it does not have privilege for.

Types of System Calls. There are 5 different categories of system calls: process control, file manipulation, device manipulation, information maintenance and communication.

How are system calls different to interrupts?

System call is a call to a subroutine built in to the system, while Interrupt is an event, which causes the processor to temporarily hold the current execution. However, one major difference is that system calls are synchronous, whereas interrupts are not.

Device (mechanical hardware)

Device controller (electrical hardware)

Device driver (software)

The Device vs. its Controller:

Some duties of a device controller:

- Interface between CPU and the Device

- Start/Stop device activity

- Convert serial bit stream to a block of bytes

- Deal with error detection/correction

- Move data to/from main memory

**Types of Interrupts:**

Timer interrupts -

Allows OS to regain control of the CPU - One way to keep track of time

I/O interrupts -

Keyboard, mouse, disks, network, etc…

Program generated (traps & faults) -

Address translation faults (page fault, TLB miss) - Programming errors: seg. faults, divide by zero, etc. - System calls like read(), write(), gettimeofday()

System calls :

System calls are the mechanism by which programs invoke the OS Implemented via a TRAP instruction

Example UNIX system calls: open(), read(), write(), close() kill(), signal() fork(), wait(), exec(), getpid() link(), unlink(), mount(), chdir() setuid(), getuid(), chown()

Are Traps Interrupts?

Traps, like interrupts, are hardware events

But traps are synchronous whereas interrupts are asynchronous traps are caused by the executing program rather than by a device that is external to the CPU

Process - a program in execution

Program - description of how to perform an activity (instructions and static data value)

Process -

A snapshot of a program in execution.

memory (program instructions, static and dynamic data values)

CPU states (registers, PC, SP)

Operating system state (open files, accounting statistics)

Process Address Space

Each process runs in its own address space that consists of:

Stack space - used for function and system calls

Data space - variables (both static and dynamic)

Instructions - the program code(usually read only)

Invoking the program multiple times results in creating of multiple distinct address spaces.

Switching among process:

Program instructions operate on operands in memory and (temporarily) in registers.

Saving all the information about a process allows a process to be temporarily suspended and later resumed.

The Scheduler :

Lowest layer of process-structured OS . handles interrupts & scheduling of processes .

Sequential processes only exist above that layer .

How Are Processes Created?

Events that cause process creation:

System startup

Initiation of a batch job

User request to create a new process

Program request to create a new process

All result in the execution of a process creation system call (fork) from another process.

Process Hierarchies

Parent process creates/forks child process - each process is assigned a unique identifying number or process ID (PID) - system calls for communicating with and waiting for child processes

Child processes can create their own child processes

How Do Processes Terminate?

Conditions that terminate processes:

Normal exit (voluntary)

Error exit (voluntary)

Fatal error (involuntary)

Killed by another process (involuntary)

All of the above are system calls!

Process Creation in UNIX

All processes have a unique process id getpid(), getppid() system calls allow processes to get their information.

Process creation fork() system call creates a copy of a process and returns in both processes (parent and child), but with a different return value exec() replaces an address space with a new program.

Process termination, signaling signal(), kill() system calls allow a process to be terminated or have specific signals sent to it.

Process Creation (fork)

Fork creates a new process by copying the calling process

The new process has its own -

- Memory address space (copied from parent)

Instructions (same program as parent!)

Data

Stack

- Register set (copied from parent)

- Process table entry in the OS

Pthreads: A Typical Thread API Pthreads: POSIX standard threads

First thread exists in main(), creates the others

pthread\_create (thread,attr,start\_routine,arg)

- Returns new thread ID in “thread”

- Executes routine specified by “start\_routine” with argument specified by “arg”

- Exits on return from routine or when told explicitly

pthread\_exit (status)

- Terminates the thread and returns “status” to any joining thread

pthread\_join (threadid,status)

- Blocks the calling thread until thread specified by “threadid” terminates - Return status from pthread\_exit is passed in “status” - One way of synchronizing between threads pthread\_yield ()

- Thread gives up the CPU and enters the run queue

Kernel-Level Threads:

OS threads: Thread-switching code is in the kernel

User-level threads: The thread-switching code is in user space

Implementing Threads:

We need a CPU in order to run a kernel thread

We need a kernel thread or process in order to run a user-level thread

We need a thread control block (TCB) structure in memory to hold thread state

It can be in user memory for user-level threads

When a thread is created, what needs to happen?

Threads share their code section. In many systems, even separate tasks that are executing the same program can share their code section, because the code sections are read-only, so whatever one task does won't affect the other task.

Each thread has its own set of register values, including the program counter. Each thread also has its own call stack. Having its own program counter and call stack (in a way, the program counter is the top of the call stack) pretty much defines a thread. Multiple threads of the same task have program counters that each point at some place in the shared code section.

Creating a thread requires allocating memory to store its stack, and to store its register values when switching between threads. It doesn't affect the code section.

What will cause a thread switch to occur?

The main distinction between a thread switch and a process switch is that during a thread switch, the virtual memory space remains the same, while it does not during a process switch. Both types involve handing control over to the operating system kernel to perform the context switch. The process of switching in and out of the OS kernel along with the cost of switching out the registers is the largest fixed cost of performing a context switch.

A more fuzzy cost is that a context switch messes with the processors caching mechanisms. Basically, when you context switch, all of the memory addresses that the processor "remembers" in its cache effectively become useless. The one big distinction here is that when you change virtual memory spaces, the processor's Translation Lookaside Buffer (TLB) or equivalent gets flushed making memory accesses much more expensive for a while. This does not happen during a thread switch.

Which is faster addressing mode?

Relative addressing

Absolute addressing

In Relative addressing mode, offset is mentioned in the instruction. In high level language, if else while generates jump instruction and they are relatively small in size. So, offset tends to be small and may fit is same page in some case. So, it is faster. In absolute address, processor needs to calculate relative distance which cause extra cycle.

Moreover, relative addressing allows position independent code, absolute addressing not.

**Is the kernel really needed?**

It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications. To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. A device driver is a computer program that enables the operating system to interact with a hardware device.In Kernel mode, the executing code has complete and unrestricted access to the underlying hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system.

Types of “Context Switch” :

Process switch

Thread switch

Kernel thread switch

User-level thread switch

Interrupt?

Function call?

What is a process and process table? What are different states of process?

A process is an instance of program in execution. For example a Web Browser is a process, a shell (or command prompt) is a process. To keep track of the state of all the processes, the operating system maintains a table known as the process table. Inside this table, every process is listed along with the resources the processes is using and the current state of the process.

Processes can be in one of three states: running, ready, or waiting.

The running state means that the process has all the resources it need for execution and it has been given permission by the operating system to use the processor. Only one process can be in the running state at any given time. The remaining processes are either in a waiting state (i.e., waiting for some external event to occur such as user input or a disk access) or a ready state (i.e., waiting for permission to use the processor). In a real operating system, the waiting and ready states are implemented as queues which hold the processes in these states.

What is a Thread? What are the differences between process and thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes.

A thread has its own program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals.

Race condition:

- Multiple accesses to the same variable, from different threads, with at least one write

- Result depends on the execution order of the threads

- Also called a data race

Critical section:

- a section of code that contains a race condition

Enforce mutual exclusion on critical section code:

- make sure only one thread can execute it at a time

What is deadlock?

Deadlock is a situation when two or more processes wait for each other to finish and none of them ever finish. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).

What are the necessary conditions for deadlock?

Mutual Exclusion: There is a resource that cannot be shared.

Hold and Wait: A process is holding at least one resource and waiting for another resource which is with some other process.

No Preemption: The operating system is not allowed to take a resource back from a process until process gives it back.

Circular Wait: A set of processes are waiting for each other in circular form.

What is Virtual Memory? How is it implemented?

Virtual memory creates an illusion that each user has one or more contiguous address spaces, each beginning at address zero. The sizes of such virtual address spaces is generally very high.

The idea of virtual memory is to use disk space to extend the RAM. Running processes don’t need to care whether the memory is from RAM or disk. The illusion of such a large amount of memory is created by subdividing the virtual memory into smaller pieces, which can be loaded into physical memory whenever they are needed by a process.

What is Thrashing?

Thrashing is a situation when the performance of a computer degrades or collapses. Thrashing occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary to in order to appreciate the benefits of virtual memory, thrashing has a negative effect on the system. As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault.

Mutex vs Semaphore

mutex and semaphore are kernel resources that provide synchronization services (also called as synchronization primitives)

Critical section is group of instructions/statements or region of code that need to be executed atomically such as accessing a resource (file, input or output port, global data, etc.).

The [producer-consumer](http://en.wikipedia.org/wiki/Producer-consumer_problem) problem:

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the entire buffer.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is signaling mechanism (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

1. Can a thread acquire more than one lock (Mutex)?

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a recursive mutex can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. What happens if a non-recursive mutex is locked more than once.

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

7. Can we acquire mutex/semaphore in an Interrupt Service Routine?

An ISR will run asynchronously in the context of current running thread. It is not recommended to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

8. What we mean by “thread blocking on mutex/semaphore” when they are not available?

Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

9. Is it necessary that a thread must block always when resource is not available?

Not necessary. If the design is sure ‘what has to be done when resource is not available‘, the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example POSIX pthread\_mutex\_trylock() API. When mutex is not available the function returns immediately whereas the API pthread\_mutex\_lock() blocks the thread till resource is available.

Interrupt Vs System Call

User programs can’t be trusted if they wish to invoke some functionality in kernel. So whenever they wish to access the OS functionality in kernel they make the use of System Calls. System Calls let the User access the kernel in controlled manner.

However an interrupt is a request from some external device for a response from the processor. There can be software interrupts (generated via a program) or hardware interrupts but the way interrupts are handled remains same.

There's an Interrupt Dispatch Table (IDT) and a System Service Dispatch Table (SSDT).

When a user-mode application call needs servicing by the kernel, e.g., it wants to open a file, for example, where it calls fopen(), eventually this request will be sent to the kernel, and this will be a system call.

There is an interrupt dispatch table (IDT), with respective functions to call when an interrupt occurs. For example, int3 is a well-known interrupt for a debugger break-point.

System call is a call to a subroutine built in to the system, while Interrupt is an event, which causes the processor to temporarily hold the current execution. However one major difference is that **system calls are synchronous, whereas interrupts are not**. That means system calls occur at a fixed time (usually determined by the programmer), but interrupts can occur at any time due an unexpected event such as a key press on the keyboard by the user. Therefore, whenever a system call occurs the processor only has to remember where to return to, but in the event of an interrupt, the processor has to remember both the place to return to and the state of the system. Unlike a system call, an interrupt usually does not have anything to do with the current program.

##### 

Difference between forking and multithreading

A fork gives us a brand new process which is a copy of the current process with the same code segment. It looks exactly like the parent process with different process id having it's own memory. Parent process creates a separate address space for the child with same code segments but executes independently of each other. Because the system issues a new memory space and environment for the child process, it is known as heavy-weight process.

While threads can execute in parallel with same context. Also, memory and other resources are shared between the threads causing less overhead. A thread process is considered a sibling while a forked process is considered a child. Also, threads are known as light-weight processes as they don't have any overhead as compared to processes (as it doesn't issue any separate command for creating completely new virtual address space). A single process can have multiple threads. For all threads of any process, communication between them is direct. While process needs some interprocess communication mechanism to talk to other processes. Thought, threads seem to be more useful for any reason, do note that changes in any thread may lead to changes in other threads of the same process. While, changes in child processes is independent as parent process has its own execution copy.

What is the Translation Lookaside Buffer (TLB)?

In a cached system, the base addresses of the last few referenced pages is maintained in registers called the TLB that aids in faster lookup. TLB contains those page-table entries that have been most recently used. Normally, each virtual memory reference causes 2 physical memory accesses- one to fetch appropriate page-table entry, and one to fetch the desired data. Using TLB in-between, this is reduced to just one physical memory access in cases of TLB-hit.

Please give a brief introduction of the terminology of Race Condition in Operating System.

1)The order of execution of instructions defines the results produced

2)Occurs when access of same data at same time .

3)Arises when several processes tries to access and manipulate the shared data (also called as Critical Section) concurrently. This concurrent access of Critical section will lead to data inconsistency and undesirable results.

Example :

Consider a kernel data structure that maintains list of all open files in the system. This list must be modified when the new file is opened or closed(adding the file to the list and removing file from the list).If two processes are opening file simultaneously then the concurrent update in this list will lead to race condition.

##### What is difference between top half and bottom half ?

Interrupt handlers run asynchronously, and must respond to time-critical inputs quickly. Therefore, interrupt handler is divided into top half and bottom half. First half was executed by the kernel immediately after a hardware interrupt, whereas bottom half are the tasks that can be deferred to execute later.

How do you write code that is sure not to run into deadlock condition?

Dead lock can happen only on all conditions hold: mutual exclusion, hold and wait, no preemptation, cycle wait. Several techniques are used to resolve deadlock problem:

dead lock prevention:

(1) Banking algorithm: model all resources into a vector and each request into q req vector, and already allocated resource into alloc vector. The algorithm runs like this: iterate through each of the req vector and deduct them from the resource remaining vector, if some of the resource will be negative then it will have some race condition, so will reject the request. If all request can be fullfilled, the request is approved

(2) Resource ordering: We can tag each kind of resource with a number. Each thread/process which request some resource which is smaller than its current holding resource will be rejected

dead lock avoidance:

(1) wait and die: newer thread will wait for older thread, however older thread will preempt currently new thread.

(2) wound wait: newer thread will be dropped when older thread is running, however older thread will wait for newer thread

dead lock detection: this solution used the wait-for graph to find if there is any cycle wait. if yes it will pick some victim thread and break the loop

How do you find out if there is a memory leak?

run tool valgrind to check memory leak

How does segmentation fault work internally?

When a processor is given a virtual address, it passes the address to MMU which will check if there is a mapping from the virtual address to physical address by looking up the page table (MMU will check TLB first though ). If there is no such mapping, the processor will take it as a page fault and subsequently check the page fault is a valid or not by checking if the address belongs to any segments the process currently has.

When the page fault is valid, which means the page resides in the swap space and needs to be swapped into memory, the processor will proceed after the page gets swapped in; otherwise, the page fault is invalid, the processor will send a segmentation fault signal to the process and kills the process by default if there is no signal handler that catches the SIGSEGV signal.

What prints the message "Segmentation Fault"?

Shell that runs the program prints the message "Segmentation Fault". Basically when the process is killed because of segmentation fault, its parent(the shell in this case) that calls wait() to wait for its termination will get its exit status. Then the parent knows the reason why the child process is terminated based on the exit status, and prints the message accordingly.

Please explain the concept inode in file system.

A basic idea: inode is a type of data-structure in a UNIX-style file system, and is used to store a file's metadata such as block location, owner, time of last change, etc.

The files in the UNIX-style file systems are stored in blocks of fixed sizes( and when I say files it includes directories too) and follow contiguous chained memory allocation). As you mentioned Inode contains metadata of the file but most importantly it has the pointer to the next location which might be the block itself containing the file or another node which may point to block or to some other node and so on (depending how large the file is, thus how many nodes it requires)

What is interrupt? How does interrupt work?

An interrupt is a signal the causes the processor to stop executing the current process, and instead process the code that is in the interrupt handler. Once that is done, processor resumes previous operation.

Interrupts can be generated by both hardware and software. Hardware interrupts happens when an input pin detect a rise or fall in the (low to high or high to low). In the case of PCs, external hardware like keyboard, disk etc. can generate hardware interrupts.

Software interrupts are generated by the current running process when asking for resources that are provided by the kernel (via system calls). The don't necessarily interrupt the CPU but only the process itself.

Why Paging is used?

Paging is a mechanism that allows memory virtualization, in other words, allows addressing more memory than physically exists. The memory manager will keep "most used pages" in RAM and swap out those less frequently used. Pages are usually in the 4K boundaries. When a reference to a non existing virtual memory region is used, the CPU will emit a trap, which then the kernel will handle, by loading/replacing the page where the memory address referenced lives, fixing the trap and continuing from the same PC address. Now the program will happily access to the referenced virtual address. This concept was introduced with protected memory models, I believe by Intel with 80286.

paging is used to remove external fragmentation that occurs in the contiguous memory allocation techniques like multiprogramming with variable number of tasks and paging is non contiguous allocation of memory i.e., program may reside in different number of pages and in order to use paging we require a page table which is used to find the frame number from the page number which is implemented by using memory management unit

As there will be more space to be allocated to the page tables we use the concept of multi level paging i.e., applying paging on the page table and also segmentation.

Thread vs process

A process is a program in execution with an associated context, address space, and thread of execution (stacks pointer, registers, program counter). A process can also have multiple threads. Threads can be viewed as processes with the key difference that they share the same address space. You can view a process as one thread with one address space. Creating a multi-threaded process comes with advantages,as threads are cheaper to create and to context switch. Also communication between threads is naturally more simple, as they can access the same memory space.

Difference between BIOS and Kernel

Kernel is one of the most important part of Operating System. We use the word kernel to mean the part of operating system that runs in the privileged mode (Sometimes even a subset of this). Kernel is closer to the hardware and often performs tasks like memory management and system calls.

Now for BIOS (Basic Input-Output System), it is the one which is responsible to provide drivers for new devices to OS. BIOS constitutes of the code that is stored in read-only memory (ROM) and some configuration data in non-volatile RAM.

BIOS provides three primary functions:

Power on self test (POST), so it knows where to load the boot program.

Load and transfer control to boot program .

Provide drivers for all devices.

The main BIOS is supplied as a chip on the motherboard. It contains everything needed to perform the above three functions. Additional BIOSes on other boards can provide access to additional devices.

Least Recently Used Cache

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes.

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.  
   Waiting Time = Turnaround Time – Burst Time

Process States:

Ready

Running

Blocked

Scheduling Policies:

First-Come, First Served (FIFO)

Shortest Job First (non-preemptive)

Shortest Job First (with preemption)

Round-Robin Scheduling Priority Scheduling

Real-Time Scheduling

First-Come, First Served (FIFO) :

Start jobs in the order they arrive (FIFO queue)

Run each job until completion

Turn around time = processing time + delay

Shortest Job First

Select the job with the shortest (expected) running time

Non-Preemptive

Round-Robin Scheduling Priority Scheduling

Effectiveness of round-robin depends on

- The number of threads, and

- The size of the time quantum.

Priority Scheduling

Assign a priority (number) to each thread Schedule threads based on their priority Higher priority threads get more CPU time

Starvation is possible!

Managing priorities Can use “nice” to voluntarily reduce your priority

Scheduler can periodically adjust a process’ priority

- Prevents starvation of a lower priority process

- Can improve performance of I/O-bound processes by basing priority on fraction of last quantum used

Multi-Level Queue Scheduling

Multiple queues, each with its own priority Equivalently:

each priority level has its own ready queue Round-robin scheduling is used within each queue Simplest approach uses statically assigned priorities

Memory Management Unit (MMU)

- Dynamically converts relocatable logical addresses to physical addresses

Multiprogramming: a separate partition per process

What happens on a context switch?

Store process base and limit register values Load new values into base and limit registers

When and why do we access a page table?

- On TLB miss faults to refill the TLB

- During process creation and destruction

What if the entry is not in the TLB?

- Go look in the page table in memory

- Find the right entry

- Move it into the TLB

Hardware TLB refill

- Page tables in specific location and format

- TLB hardware handles its own misses

- Replacement policy fixed by hardware

Software refill

- Hardware generates trap (TLB miss fault)

- Lets the OS deal with the problem

- Page tables become entirely a OS data structure!

- Replacement policy managed in software

How can we prevent the next process from using the last process’s address mappings?

- Option 1: empty the TLB on context switch New process will generate faults until its pulls enough of its own entries into the TLB

- Option 2: just clear the “Valid Bit” on context switch New process will generate faults until its pulls enough of its own entries into the TLB

- Option 3: the hardware maintains a process id tag on each TLB entry Hardware compares this to a process id held in a specific register … on every translation

Do we access a page table when a process allocates or frees memory?

- Not necessarily

Library routines (malloc) can service small requests from a pool of free memory already allocated within a process address space

When these routines run out of space a new page must be allocated and its entry inserted into the page table

- This allocation is requested using a system call

Three options of page table desgin:

- Single-level page tables

Problem: requires one page table entry per virtual page!

32 bit addresses and 4KB pages means 2^20 page table entries per process

64 bit addresses and 4KB pages means 2^52 page table entries per process!

- Multi-level page tables

Not all pages within a virtual address space are allocated

- Not only do they not have a page frame, but that range of virtual addresses is not being used

- So no need to maintain complete information about it

- Some intermediate page tables are empty and not needed

We could also page the page table

- This saves space but slows access … a lot!

- Inverted page tables

An inverted page table

- Has one entry for every resident memory page

- Roughly speaking, one for each frame of memory

- Records which page is in that frame

- Can not be indexed by page number

Page Replacement Algorithms:

The Optimal Algorithm

Select the page that will not be needed for the longest time

Problem:

Can’t know the future of a program

Can’t know when a given page will be needed next

The optimal algorithm is unrealizable

First In First Out (FIFO)

Replace the page that has been in memory for the longest time

Implementation Maintain a linked list of all pages in memory

Keep it in order of when they came into memory

The page at the tail of the list is oldest

Add new page to head of list

Disadvantage:

The oldest page may be needed again soon

Some page may be important throughout execution

It will get old, but replacing it will cause an immediate page fault

Not Recently Used (NRU)

Second Chance / Clock

Least Recently Used (LRU)

Not Frequently Used (NFU)

Working Set (WS)

WSClock

Each page table entry (and TLB entry!) has a

- Referenced bit - set by TLB when page read / written

- Dirty / modified bit - set when page is written

- If TLB entry for this page is valid, it has the most up to date version of these bits for the page

- OS must copy them into the page table entry during fault handling

Time required to read or write a disk block determined by 3 factors

- Seek time

- Rotational delay

- Actual transfer time

Disk Scheduling Algorithms

First-come first serve

Shortest seek time first

Scan " back and forth to ends of disk

C-Scan " only one direction

Look " back and forth to last request

C-Look " only one direction

Disk Space Management

How to keep track of allocated vs free blocks ?

Approach #1:

- Keep a bitmap

- 1 bit per disk block

Approach #2

- Keep a free list

Allocation Strategies

Determining which blocks make up a file:

• Contiguous allocation

• Linked allocation

• FAT file system

• Unix I-nodes

Each I-node (“index-node”) is a structure containing info about the file

- Attributes and location of the blocks containing the file

In Unix:

Hard links

- Both directories point to the same i-node

Symbolic links

- One directory points to the file’s i-node

- Other directory contains the “path”

The primary limitation of weighted round-robin queuing is that it provides the correct percentage of bandwidth to each service class only if all the packets in all the queues are the same size or when the mean packet size is known in advance.