Operating Systems
Ludovic Apvrille

I. Introduction

An *operating system* is a program that manages a computer's hardware. It acts as an intermediary between the computer user and the computer hardware. It's designed to be convenient and/or efficient.

A *computer system* can be divided into 4 components:

- The **hardware** (Central Processing Unit, memory, input/output devices): the basic computing resources for the computer.
- The **application programs** (spreadsheets, compilers, web browsers): the way in which those resources are used to solve users' computing problems.
- The **operating system** controls the hardware and coordinates its use among various application programs for the various users.
- The **users**: persons authorized to run processes.

The main 3 purposes of an OS:

- Provide environment for users to execute programs in a convenient and efficient way.
- Allocate recourses to solve a given problem.
- Prevent errors of user programs and control I/O devices.

Protection of the hardware: prevent illegal instructions, illegal use of devices, illegal memory access and overusing the CPU resources.
How? Dual mode: user mode & monitor mode

An **interrupt** is a signal sent to prevent the occurrence of an event. When the CPU is interrupted, it stops the current task and transfers execution to a fixed location where the service routine for the interrupt is located. On completion, the CPU resumes the interrupted computation.

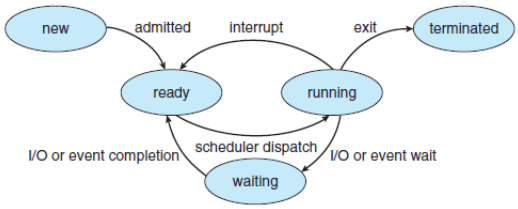
A **system call** provides the means for a user program to ask the operation system to perform tasks reserved for the OS (fork, exec, exit, wait).

A **virtual machine** is a completely isolated operating system installation within a normal host operating system.

A **container** is a package of an application and all its dependencies to execute it in any environment and isolate it from other applications.

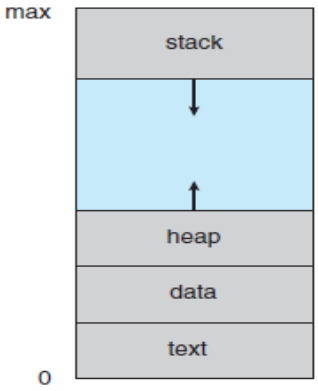
II. Processes

A **process** is a program in execution. It's more than the program code (text section). It also includes the current activity.
The OS must periodically gain control to ensure CPU fairness between processes and prevent a process stucking the system (e.g. hardware timer is set before the process is given the CPU).



Process Data:

- Program code: text section (static)
- Current activity: program counter, next instruction
- Stack: function params, return addresses, local variables
- Heap: data section



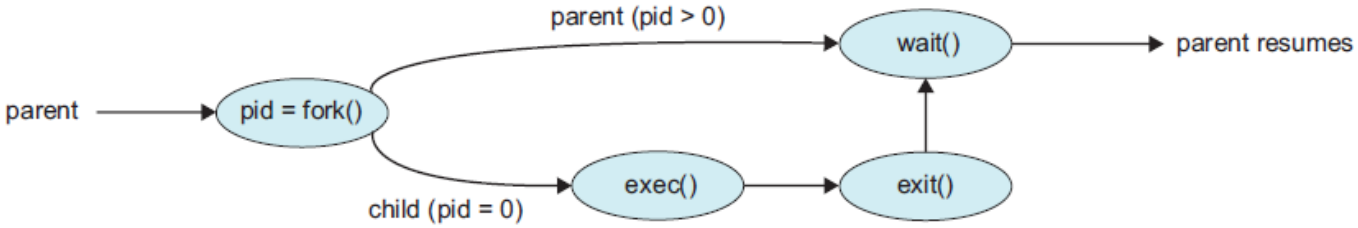
A Process Control Block contains many pieces of information about a specific process: process state, program counter, CPU registers, CPU-scheduling information, memory-management information.

1. How to launch the first user process? Boot sequence.
2. How to manage processes from a programmer's point of view? APIs.
3. How to schedule processes? Scheduling policies.
4. How can processes communicate with each others? Signals, shared memory, message passing.

Controlling Processes

- Create a new process: int main(int argc, char * argv[]), fork()
- Terminate a process: return from main, call to exit(), call to abort()

fork(): creates a new process (child) by duplicating the calling one (parent).



vfork(): the parent is suspended until the child makes a call to execve() or exit().

III. Scheduling

CPU scheduling is the basis of multiprogrammed OS. By switching the CPU among processes, the OS can make the computer more productive.

Scheduling Criteria:

- Fairness: give each process the same amount of CPU.
- Balance: keep all parts of the system busy.
- CPU utilization: keep the CPU busy.
- Throughput: maximize the number of completed processes per time unit.
- Turnaround time: minimize time between submission and termination.
- Response time: respond to interactions as fast as possible.
- Meeting deadlines: ensure tasks will be completed before a given time.

Nonpreemptive scheduling: the same process is given the CPU until it terminates.
Preemptive scheduling: the same process can run only for a predefined period.

Scheduling Algorithms

- First-Come, First-Served: single queue, easy to program, nonpreemptive.
- Shortest-Job-First: easy to implement, nonpreemptive, optimal when all processes are ready
- Shortest-Remaining-Time-Next: preemptive version of SJF
- Round-Robin: the ready queue is treated as circular queue; each process is assigned a time quantum.
- Priority-Based: the process with higher priority is chosen (can be dynamic: $p_{n+1} = q/t_n$.
- Group-Based, Fair-Share, Lottery, Multiprocessor, etc.

IV. Memory Management

Goal? prevent processes to access unauthorized memory (used by other processes, used by OS)
How? Dual mode & MMU
Memory Management Unit: run-time mapping from virtual addresses (generated by CPU at compilation time) and physical addresses (of the RAM).

Issues:

- Process admittance: OS estimates the required memory and allocates it.
- Dynamic allocation: a process may require more or release memory space
- Process termination: OS must release all allocated memory

Keep track of allocation units: linked lists
Allocation algos:

- First Fit: scan the list until a large enough space is found
- Best Fit: allocate the smallest hole that is big enough (search the entire list unless it's ordered)
- Worse Fit: allocate the largest hole

Swapping: a process can be temporarily swapped out of memory to a backing store (e.g. disk) and then brought back into memory to continue execution.
Swap out: when memory occupied is over threshold, memory allocation failed
Swap in: when a process is ready, a large amount of memory is freed

Segmentation of Memory is a memory-management scheme where a logical address space is represented as a set of segments. Each segment has a segment-name (usually segment number) and an offset.
A C compiler may create different segment for each of these parts: code, global variables, heap, stack, library.

Fragmentation: as processes are loaded and removed from memory, the total memory available is enough to satisfy a new process but it's fragmented into a large number of non-contiguous holes (blocks of wasted memory between two processes).

Paging is a memory-management scheme that allows the logical address space to be non-contiguous in physical memory.
How? Breaking physical memory into fixed-sized blocks called frames and logical memory into pages of the same size. When a process is to be executed, its pages are loaded into any free frames.

Memory Protection

- OS updates the address table
- MMU detects addresses having no correspondence

How segmentation/page faults happen?

- The address is invalid (outside the process address space) → process is stopped: seg fault
- Segment/page has been swapped out: the OS must make a swap in

Page replacement policies: Clock (Windows), NRU, LRU, FIFO.

V. Input/Output

OS is the interface between devices and other parts of the system: sends commands to devices, gets information from them, handles errors.

An I/O port typically consists of four **registers**: status, control, data-in, data-out.

Interface is the hardware circuit between a group of I/O ports and the device controller. Two types: custom for specific device (keyboard, disk) and general (USB, SATA)

Device Controller

- Interprets high-level commands received from I/O interface
- Interprets signals coming from the device

e.g. disk controller: receives high-level command (write this to disk) and sends low-level orders (position of disk head on the track)

Memory Transfer: from main memory to device memory and vice versa.

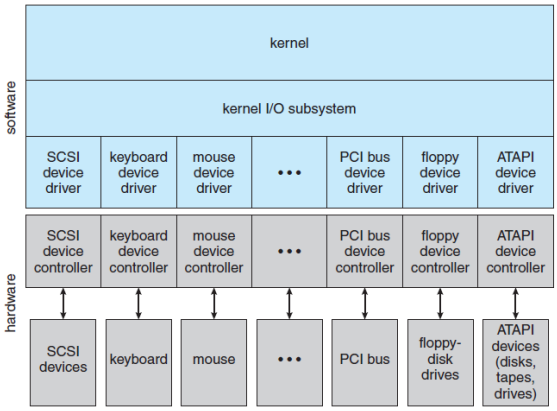
For large data transfer (disk) it's wasteful to watch the status bit and transfer one byte of data at a time. **D**irect **M**emory **A**ccess controller is used instead to transfer large data.

Characteristics of devices

- Character stream (one-byte transfer at time) vs. block stream
- Sequential access (data transfer in fixed order) vs. random access
- Synchronous (predictable response time) vs. asynchronous
- Sharable (used by different processes) vs. dedicated

Kernel I/O subsystem

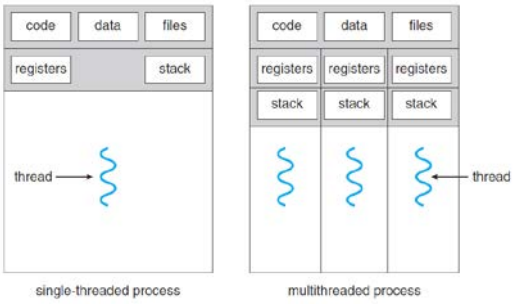
I/O scheduling, buffering, caching, access control, error management.



Scheduling: classification of requests by device and applying scheduling on requests.
Buffering: manipulating large set of data more efficiently.
Caching: main memory can be used to increase I/O operations.
Access Control: spooling (serving one job at a time: printer), exclusive device access.
Error Handling

VI. Threads

A **thread** is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register ser, and a stack. It shares with other threads of the same process its code section, data section, and other recourses.



A process with more than one thread of control can perform more than one task at time.

Benefits

- Responsiveness: an application can continue executing while one activity is blocked
- Resource Sharing: threads share several resources, e.g. open files, open network connections, ...
- Performance: thread creation/switching/destruction is faster than for processes
- Scalability: multiple threads of a process can be run on different processing cores.

→ User threads: all management is done at user level, kernel is not aware of threads.
→ Kernel threads: creation, management, scheduling is done by kernel

Multithreading Models

- Many-to-One Model: maps many user-level threads to one kernel thread
  - Efficient, Scheduling can be customized
  - One thread can bock the whole process via a blocking syscall

- One-to-One Model: maps each user thread to one kernel thread
  - Best concurrency
  - Performance drawbacks: creating many threads
- Many-to-Many Model: multiplexes many user threads to a smaller or equal number of kernel threads
  - Tradeoff between performance and concurrency

Inter-process – Inter-thread communication

- Pipes: one-way data stream routed by kernel.
- Signals: software interrupts, e.g. signal to stop a process (CTRL+C). 31 signals under Linux.
  - Process 1 sends a signal (kill(pid, sig)), process 2 executes a specific function to handle it (signal(sig, handler)).
- Network Sockets: bidirectional communication with address (IP + Port) and protocol(TCP,UDP).

IPC

Each object (shared memory segment, semaphores, message queues) is referred in the kernel by a non-negative integer.

Shared Memory: two ore more processes may share a given region of memory to exchange data (create shmget, control shmctr, attach to ots address space shmat, detach from its address space shmdet).

VII. Synchronization

Concurrent or parallel process execution can contribute to issues involving the integrity of shared data.
Synchronization to know for a process/thread at which execution point is another process/ thread.
→ Ensure shared data consistency.

A **critical section** is a segment of a process code in which the process may be changing common variables.
Critical sections must satisfy these requirements:

- Mutual exclusion: at most one process at a time can execute his critical section.
- Machine independence: no assumptions about speed or number of CPUs.
- Progress: process running outside a critical section may not block other processes.
- Bounded waiting: process should be guaranteed to enter critical section within a finite time.

Deadlock: a situation in which a process waits for a resource that will never be available. 3 methods to deal with deadlocks:

- Prevent or avoid them, ensuring that the system will never enter a deadlock state.
- Allow to enter in a deadlock state, detect it, and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system.

Implementing Critical Sections

<table>
<tr><th>Software Approaches</th><th>Hardware Approaches</th></tr>
<tr><td>Lock variables
• Reads the value of shared variable
• If 0, set it to 1 and enters critical section
• If 1, waits until its equal to 0</td><td rowspan="2">Test and Set Lock instruction: reads the content of the memory at address lock, stores it in register Rx, and sets the value at address lock to 1.</td></tr>
<tr><td>Petersons Solution: use two variables int turn; and boolean flag[2]; the first to indicate whose turn it is, and the second to indicate if process i is ready to enter critical section.</td></tr>
</table>

Limits of these approaches:
If a lower priority process is in critical section and a higher priority process busy waits to enter this critical section, the lower priority process never gains CPU → Higher priority processes can never enter critical section

Semaphore is a counter shared by multiple processes. Processes can increment/decrement the counter in an atomic way.

Mutex: Mutual Exclusion: can be locked or unlocked. Only one process/thread at a time can lock a mutex.

Check Producer/Consumer example!