FSO

Unit 1 – Operating System Concept

Operating System: set of software that allows the operation of computer systems and offers a friendly interface to users, and an efficient environment to run programs. Its functions are:

- Providing the interfaces to the users (hardware abstraction)
- Offering services (system calls)
- Managing resources such as processes, memory, files, and I/O
- Supervising resource access to avoid conflicts and unauthorized accesses

It provides services to different users:

- Application user
- Application programmer
- System programmer
- System administrator

Kernel: consists of File Systems, Memory Manager, Process Manager and Device drivers

System utilities: they extend the OS providing key utilities that aren't included in the kernel (shell, GUI, monitoring)

Kernel types:

- <u>Microkernel</u>: provides only basic hardware abstractions and services
- <u>Monolithic</u>: all kernel components are in the same address space
- <u>Hybrid</u>: a modified microkernel that includes some non-essential components whose execution speed is critical

System workload: it consists of a set of programs to be executed. This is a sequence of CPU and I/O bursts (time intervals required to perform an action). Oss must achieve that the CPU is as active as possible.

- <u>CPU Usage</u>: CPU busy time / Total time

Multiprogramming: alternative use of the CPU by running programs. When a process is waiting for an I/O operation, the CPU executes instructions from other programs (context switch). Therefore, the CPU usage increases as well as the system performance.

Historical evolution of the OS:

- <u>Basic batch systems</u>: jobs are processed sequentially, CPU is idle when performing I/O operations. Resident monitor that automatizes some tasks. No direct user-machine interactions
- <u>Multiprogrammed batch systems</u>: CPU scheduling, multiprogramming, memory, disk and protection management. No user-machine interaction
- <u>Time sharing systems</u>: direct user-machine interaction with multiprogramming. Job synchronization and communication, file systems, virtual memory, process scheduler (OS limits the CPU via context switches that rely on interrupts)
- <u>Real time systems</u>: for executing tasks with a fixed deadline
- <u>PC systems</u>: personal use, friendly user interfaces, multimedia, plug-and-play, network access
- <u>Parallel systems</u>: multiprocessor (multicore) via shared memory, reliability
- <u>Distributed systems</u>: computation is distributed along several computers via a network, resource and workload sharing
- <u>Cloud systems</u>: storage and computation as a service

Unit 2 – System Call Concept

I/O and CPU concurrency: I/O is slower than CPU. It is important that when I/O is being performed, the processor can execute useful instructions.

- <u>Device Driver</u>: OS component, software capable providing a friendly interface to use and program the Device Controller
- <u>Device Controller</u>: hardware component, capable of DMA (direct memory access)

Interrupts: an OS is an event-driven program. Th events are hardware interrupts, software interrupts and exceptions. The OS acts as a server program waiting for work commissioned by interrupts. OS processes and I/O devices perform OS service requests.

- <u>I/O interrupt</u>: generated by I/O devices
- <u>Clock interrupt</u>: the OS enters execution at certain time intervals
- <u>Hardware exception</u>: due to memory parity error, power outage
- <u>Traps</u>: used by programs to request OS services
- <u>Software exceptions</u>: generated when an invalid instruction is processed in a program

The mechanism: [CPU is busy] -> [Interruption request arrives] -> [CPU executes OS code]

Execution modes: processors have two or more execution modes, which support the OS. Processes running simultaneously share machines resources, so they need protection.

Protecting the access to the hardware prevents user programs from accessing system memory freely, monopolizing the CPU and system registers, and accessing directly to I/O devices. The mode is identified by a single bit:

- Kernel mode (0): runs any type of hardware operations, memory access and I/O devices. Instructions available only on kernel mode are called privileged instructions, and they are associated with three types of protection: I/O, memory and process protection.
- User mode (1): has a restricted instruction set

System calls: the OS provides on-demand services, an interface that defines the operation set to allow access to machine resources. This is implemented as C library functions, and each OS has its own system calls.

Service requests work as:

1. [USER MODE] Program running
2. System call (trap or software interrupt)
3. [KERNEL MODE] Identification of the service
4. Execution of the service
5. Data requested/Service result
6. [USER MODE] Next instruction

➔ POSIX stands for Portable Operating System Interface

System utilities: are programs that run as user processes and provide a more comfortable environment. They are provided as part of the OS but are not essential for machine operation.

Unit 3 – Process concept and implementation

Program: file in executable format generated by a compilation of source code

Process: a program which is running (working unit of the OS, resource consumer, basically every activity happening on a computer)

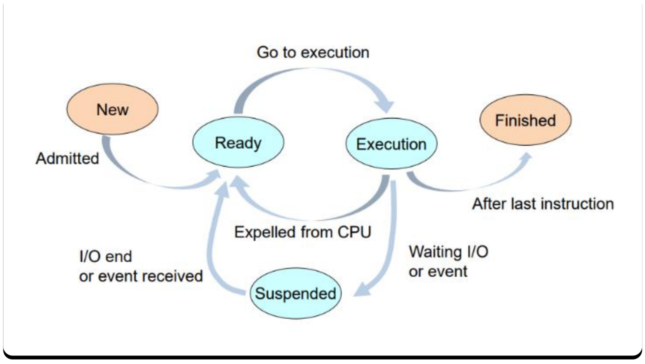
Executable files: are obtained first compiling the source code, linking it to the system or other user libraries, to incorporate external code, and finally generating an executable file. An executable file contains the code to be executed, the initialized data and library functions. With this information the OS can allocate the necessary resources for execution.

To define a process, we need to consider its attributes, behaviour and operations. The attributes are the resources and features owned by a process:

- Identity: process identifier, PID
- Runtime environment: working directory, opened file descriptors
- State: process state, machine context (PC, Stack Pointer, general purpose register values)
- Memory: memory addresses for code, data and stack
- Scheduling: CPU consumption time, priority
- Monitoring

Although the number and type of operations on processes depend on the OS, creation, communication, waiting, resource access and ending are common operations on most OS.

Process states:



Process implementation (PCB): a PCB is the data structure that supports the abstraction process. It keeps relevant information which changes during process lifetime (each OS has its own structure).

Context switch: mechanism that allows the OS to suspend the execution of the current process to start/resume another process (which is activated by an interrupt). The context of the old process is saved.

Unit 4 – Process scheduling

Lack of resources: many processes need to access a single resource the CPU, so then the OS must implement a policy to allocate those resources.

Scheduler: OS component that decides which process gets a resource at every time instant, following a certain policy.

Process types: a CPU-bound process is a process which spends most of its time performing CPU bursts. Same with I/O-bound process (but with I/O bursts). Most processes have shorter CPU bursts than I/O ones.

There are many ways to schedule a process depending on the load type:

- CPU usage: relative CPU busy time
- Throughput: jobs processed per time unit
- Turnaround time: time elapsed between a process' arrival and its completion
- Waiting time: time spent in the ready queue
- Response time: time from launching a process until the CPU executes its first instruction
- Fairness: every process gets its fair share of CPU. The opposite is Starvation.

Scheduling: the short-term scheduler decides which process from those in the ready queue is assigned to the CPU. The scheduler should act if the CPU is idle, if a process arrives to the ready queue or if there is a timer interrupt.

- Scheduling algorithms:
    - Non-pre-emptive (the process owns the CPU until it leaves)
        - First Come First Serve (FCFS): the CPU allocates processes in arrival order of the ready queue (easy to implement, waiting time not optimized, short delay for long jobs)
        - Shortest Job First (SJF): each job is associated with the length of the next CPU burst, CPU is assigned to smallest CPU burst
    - Pre-emptive (the scheduler can take a process out from the CPU)
        - Shortest Remaining Time First (SRTF): CPU is allocated to the process with less remaining time to finish its CPU burst (optimizes average waiting time, starvation risk)
        - Round Robin: every process is assigned with a CPU time packet (quantum) "q". If the CPU burst is higher than "q", the process is put out from the CPU. If there are n processes in the ready queue, each gets 1/n of the CPU time in intervals of "q" units
    - Priorities (pre-emption optional, static/dynamic): every process is associated with an integer (priority). CPU is allocated according to the job priority (lower value usually means higher priority)
    - Multilevel queue: there are several queues of ready processes, each of them having its own scheduling policy. It is required to have an inter-queue scheduling (pre-emptive priorities, CPU usage)

Unit 5 – Execution Threads

Concurrent programming: a single problem that solves a problem using simultaneous activities. Then there is parallelism and the completion time can be reduced. For example, web server and multiplayer games.

A process is an abstract entity composed by the resource assignment unit, and the CPU assignment unit. The OS can separate these units inside a process. The execution thread is the basic unit for the CPU assignment.

Execution threads inside a process share code, data and the process-assigned resources, but they also have its own attributes, the thread ID, the stack, PC and CPU registers.

Threading costs less than processes as they are easier to manage than processes and switching contexts between processes is easier, and from the programming point of view it is more natural and efficient.

Depending on where the thread support is given we can find three models:

- User level: the multithreading support is given by the programming language runtime
- Kernel level: given by the OS kernel by means of system calls
- Hybrid model: provided by both programming language and kernel, maximum flexibility and performance

To sum up, we can say that concurrency:

- Is fundamental at both application and system level
- Deals with communication, resource sharing, synchronization and CPU time reservation
- Is present at both monoprocessor and multiprocessor systems
- It can happen inside an application, inside an OS or in several applications running at once

But with producer/consumer threads that are executed concurrently, where there is shared variable access, the context switch is performed by the scheduler, so programmers don't have any control about when and how context switches happen.

A **race condition** occurs when the set of concurrent operations on a shared variable leave the variable in an inconsistent state according to the operations performed. They appear because programmers don't know when context switches happen, and because the OS doesn't know the dependencies between threads.

Race conditions are difficult to debug because they are due to thread interaction over time, being their isolated codes correct. Therefore, multithreaded programs must avoid race conditions as programmers have no control over context switches.

Unit 6 – Synchronization (Basic solutions)

The **Critical section** problem is the code zone of a process/thread where it accesses shared data. The solution is to find a protocol to synchronize the access to these sections avoiding a race condition. The critical section access protocol must verify three conditions:

- <u>Mutual exclusion</u>: if an activity is inside its critical section, no other activities can be at the same time in their critical sections
- <u>Progress</u>: if the critical section is free and there are multiple requests to enter it, the decision of which activity enters only depends on waiting activities
- <u>Limited waiting</u>: after an activity has asked to enter the critical section there is a limited number of times other activities can enter the critical section

There exist two kinds of solutions to this problem:

Software solutions (The protocol is implemented at user level)

- <u>Basic algorithm</u>: several activities share a key that indicates the critical section is busy. This doesn't verify the *mutual exclusion*, as if there is a context switch it's possible that more than one thread enters the critical section
- <u>Decker solution</u>: two activities synchronize using a variable that indicates the activity turn. It works for only two threads and doesn't verify the *progress* condition, as it imposes an alternation
- <u>Decker, Peterson and Lamport algorithms</u>: is correct software solution

Hardware solutions (machine instructions used to implement the protocol)

- <u>Disabling interrupts</u>: all context switching is avoided in the critical section (too much). This is only applicable to kernel level (privileged instructions)
- <u>Atomic instruction</u>: allows atomic (indivisible, execution can't be interrupted) evaluation and change of a variable with only one machine instruction (test-and-set), returns the value of a target variable and sets it to true. Doesn't comply with *limited waiting* as the following thread entering the critical section is unknown

All these solutions have something in common: **Active waiting**. The critical section prevents entering the critical section via forcing the execution of an empty loop. This allows for stopping an activity without calling the OS but there can be problems if you rely on priorities (thread waits for another thread which can't take the CPU), and it also wastes CPU time.

You can solve this issue using test-and-set + usleep or forcing the thread to leave the CPU. There can also be **event synchronization**, where when an activity attempts to enter the critical section, it is suspended, and a system entity is created and put at the end of the event queue. When the critical section is available, an activity is taken from the event queue to enter the critical section. System calls are needed, so there can be overhead.

Seminars Summary

C functions (simplified)

#include <stdio.h>	
printf	Prints to terminal
#include <string.h>	
strcat	Concatenates two strings
strcpy	Copies string 2 in string 1
strlen	Returns the length of str (null character ignored)
strcmp	Returns 0 if strings are equal, (>0) if str1 > str2, (<0) if str1 < str2
#include <malloc.h>	
malloc	Reserves b bits of memory and returns a pointer to the beginning of the reserved space
free	Frees the memory block pointed by p
#include <pthread.h>	
pthread_create	Creates a new thread
pthread_join	Thread waits for another to end
pthread_exit	Ends thread execution
pthread_self	Gets thread ID
pthread_equals	Compares thread ID
#include <semaphore.h>	
sem_t S	Defines a semaphore S
sem_wait P	Decrements S count by 1, if S < 0 suspends activity
sem_post V	Increments S count by 1, if S <= 0 awakes activity from S queue
pthread_mutex_lock (like P)	If mutex locked, suspends thread. Else locks mutex and thread becomes mutex owner
pthread_mutex_unlock (like V)	If there exists a suspended thread, mutex is unlocked. Else one of the suspended threads is awoken and it becomes the mutex owner

UNIX Shell (Reduced)

Basic commands	
id	Shows user ID and belonging group
su	Changes active user
who	Gives a list of active users
echo	Prints the value received
env	Prints the set of defined shell variables
cd	Changes working directory
pwd	Displays absolute path
cat	Shows file content
ls	Lists file attributes
rm	Removes files/directories
mkdir	Creates a directory
rmdir	Removes a directory
mv	Moves files/directories
cp	Copies files/directories
chmod	Sets permissions
ps	Lists current processes
kill	Sends a signal to process
sleep	Suspends shell execution
top	Shows real time stats about current active processes
Shell programming	
$0	Script name
$1..9	1^{st}..9^{th} argument
$#	Number of arguments
$?	Exit code value
Process management	
fork	Create a child process (returns 0 to the child, -1 if error, child PID to the parent)
getpid	Gets process ID
getppid	Gets parent process ID
exec	Changes process memory image to the one defined in the executable
wait	Waits until any children finish
waitpid	Waits for a child in particular
exit	Exits the process [process ends before parent -> Zombie, parent ends before process -> Orphan]
Filter commands	*(involving patterns or regular expressions)*
head	Writes the first n lines
tail	Writes the last n lines
sort	Sorts the text lines and writes the result
wc	Counts lines, words and characters
grep	Writes the lines that conform a regular expression
cut	Selects components from the processed text line
Name patterns	*(i.e. ls *.txt -> all files that end with .txt)*
*	Any substring
?	Any character

File permissions	*(chmod rwxrw-r--) -> [rwx] = user, [rw-] = group, [r--] = other*
r	Reading
w	Writing
x	Execution
Environment variables	
$PATH	Path of executable files [returns directory names separated by :]
$HOME	Path of the user home directory
$HOSTNAME	Name of PC on the network
$PWD	Working directory
$TERM	Terminal type
Directories	
.	Working directory
..	Parent directory
/bin	Common commands
/dev	I/O devices
/etc	Administration files
/usr	Application files