FSO REVIEW

Important definitions

- **Shell**: It is a program that interprets commands and runs them.
  Its syntax allows to make and run shell scripts as programs.
  - It has comments, variables, flow control, functions, etc.
  - The shell is able to:
    - Manage the file system
    - Run programs
    - Control programs I/O
    - Maintain system variables
    - Monitor & manage processes

    > ### Shell script example

```
#/bin/bash
echo The number of arguments is: $#
echo The command is $0
echo We are going to list  all the arguments: $@
echo know we will list the arguments
for i in $@
do
echo $i
done
# This script first shows the introduced arguments
#and then in a for loop it shows them one by one.
```

- **Kernel**: It is the core of the computer OS and has complete control over everything in the system. It is one of the first programs loaded on start-up (after the bootloader). It handles the start-up as well as the I/O requests from software, memory and peripherals suchs as keyboards or monitors.

- **User**: Identity known by the system that has a password, a privilege set, a home directory, etc.
  - Every user has a username and a number (UID).
  - There are system users.
  - The predefined user *root* has the highest privilege.
  - A group is a set of users
    - Every group has a name and number (GID)
    - There are predefined groups: adm, man, etc.

- **System Call**: Os mechanism to provide services on-demand. It is the inferface offered by the OS to request services and access hardware & software system resources. **It makes the execution mode to change from user mode to kernel mode.**
  System calls are useful to request OS services and to access protected resources.

- **Software interrupt or trap:** Interruption caused by an instruction or *system call*. Useful to perform an OS intervention, it implies a change to kernel mode.

Process handling

Process creation

```
pid_t fork(void);
```

Creates a *child* process which is a “clone” from the *parent*
The new process inherits most of his *parent* attributes:

- Memory Image
- UID, GID
- Current Directory
- Opened file descriptors

However, their PID is different, the *child* process has a PPID which is the *father*‘s PID. Both processes execute concurrently and independently from each other.

```
// Program example that creates a new process
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
    printf("Process %ld creates another process\n", (long)getpid());
    fork();
    printf("Process %ld with parent %ld\n", (long) getpid(),(long)getpid);
    sleep(5);
    return 0;
}
```

```
 exec()

 #include <unistd.h>
int execl(const char *path, const char *arg0, ...,
const char *argn, char * /*NULL*/);

int execlp(const char *file, const char *arg, ... ,
const char *argn, char * /*NULL*/);

int execle(const char *path, const char *arg, ...,
const char *argn, char * /*NULL*/, char * const envp[]);
int execv(const char *path, char *const argv[] );

int execvp(const char *file, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);
```

The exec() call is used change the code executed by the process.

- It changes the process memory image to the one defined on the exec() call. The executable file is specified by its name or by absolute path.
  Pid, PPID, working directory and root directory remain unchanged (As well as other process attributes)

Waiting

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait and waitpid stop the calling process execution until a child ends or the calling process receives a signal.

- **wait(int *status)**
  - Suspends the execution of the caller until **any** of its children finishes. If there is a zombie child it returns inmediatly.
  - status is not a null pointer, It contais information about the child termination.

```
#include <stdio.h>
#include <sys/types.h>
#include <errno.h>
int main(void)
{
    int status;
    pid_t pid=fork();
    switch (pid) {
    case -1:
        printf(" The child process could not be created \n");
        break;
    case 0:
        printf("I am the child process with PID %ld and my parent is %ld\n",(long)getpid(), (long
)getppid());
        sleep(20);
        printf("I have finished \n");
        break;
        default:
        printf("I am the parent process with PID %ld and my child is%d.Waiting ...\n", (long)getp
id(), pid);
    if (wait(&status) != -1) {
        printf("My child has ended ok \n");
    }
    return 0;
}
```

- **waitpid(pid_t pid, int *status, int options)**
  - Waits for a particular child
  - **pid**: Pid of the child to wait (if -1 waits for first child to end)
  - **status**: Contains information about child termination, as in wait();
  - **options**:
    - Value 0: blocking (comon value)
    - WNOHANG: non-blocking

Process scheduling

There are many processes that require to use the CPU at the same time but the CPU can attend only one process at a time. This means that we have to schedule the access to the CPU.

Scheduler: OS component that dedices what process gets a particular resource at every time instant, following a certain policy.

Short, medium and long-term schedulers

Short-term scheduler: Chooses a process from the ready queue and assigns it to the CPU.
Medium-term scheduler: Controls which processes should be in memory and which should be in the swapping area.
Long-term scheduler: Controls which processes arrive to the ready queue.

Criteria

CPU Utilization: Relative CPU busy time
Busy_CPU_Time/ Total_Time
Throughput: Number os ended processes per time unit
Number_Ended_Processes/Total_Time
Turnaround Time: Elapsed time between arrival and completion of a process.
Time_end - Time_Arrival
Waiting time: Time spent by a process on the ready queue.
Response time: Time spent since a processs is launched until the CPU starts executing it.
Multiprogramming: Concurrrent execution of several processes.

Scheduling algorithms

Objective of the short-term scheduler:
It should act if the CPU is idle, a process arrives to the ready queue or there is a timer interrupt.

Scheduling policies can be preemptive or non preemptive

- Non-preemptive: The process is on the CPU untl it voluntarily leaves
  - Less context switches.
- Preemptive: The scheduler can take out a process from the CPU
  - Requires implementing time sharing and real time.

FCFS (First-Come, First-Served)

- Non-preemptive: When a process is assigned to the CPU it stays until it finishes or needs I/O access.
- The processes are allocated into the CPU by arrival order to the ready queue.
- Easy to implement.
- Disadvantages:
  - Waiting time not optimised.
  - Convoy effect: delay for long jobs.
  - Not suitable for interactive systems.

SJF (Shortest-Job-First)

- Non preemtive
- Each job is associated with the length of the next CPU burst.
- CPU is assigned to the process with smaller CPU bursts

SRTF (Shortest-Remaining-Time-First)

- Preemptive
- The CPU is allocated to the process with less CPU burst time remaining.
- Optimizes average waiting time.

- Disadvantages:
  - Predicting the duration of the next range of CPU
  - Starvation risk on long jobs.

Priority Scheduling

- Preemptive
- Every process is associated with an integer number which designates the priority.
- Processes are allocate to CPU according to priority. (Usually the lower the number the higher the priority)

Round-Robin(RR) or Circular Scheduling

- Preemptive
- Every process is assigned with a CPU time usage or “quantum”, q
- If the CPU burst is greater than q the process performs its cpu burst until it reaches his time usage and then gets expelled from the CPU.
- If there are n processes in queue, each one gets 1/n of the CPU time in invervals of q units.

Multilevel Queue

- Several ready queues.

  - Every queue has its own scheduling policy
  - Requires inter-queue scheduling -> Queue priority

    > We have to distinguish between the internal priority of a queue with its priority with respect to the other queues.