# Operating System Exercises

**Facultad de Informática, UCM**

**Units 3.1 and 3.2: Process Management and Scheduling**

**1.-**Consider the following program executed on a UNIX-like OS:

```
void main(int argc, char *argv[]){
  int i;
  for(i=1; i<=argc; i++)
    fork();

  ...
}
```

*a*) Draw the resulting process tree for argc =3

*b*) How many processes are created for argc = n ?

**2.-**Consider the following program:

```
int globalVar;

void main() {
  int localVar=3;
  pid_t pid;

  globalVar=10;
  printf("I am the original process. My PID is %d\n", getpid());
  fflush(NULL);

  pid = fork();
  if (pid ==-1) {
    perror("Can't fork()\n");
    exit(-1);
  }
  if (pid == 0 ) {
    // Child process
    globalVar = globalVar + 5;
    localVar = localVar + 5;
  }
  else {
    // Parent process
    wait(NULL);
    globalVar = globalVar + 10;
    localVar = localVar + 10;
  }
  printf("I am the process with PID %d. Mi parent is %d Global: %d Local %d\n", getpid(),
      getppid(),globalVar, localVar );
}
```

Assuming that the PID of the original process is 100 and this process is a child of *init* (PID=1), indicate what will be printed on the screen when running the code. Does the final value for the variables change from run to run (based on the order of execution of the processes)? Does the program always print the messages in the same order for the parent and the child?

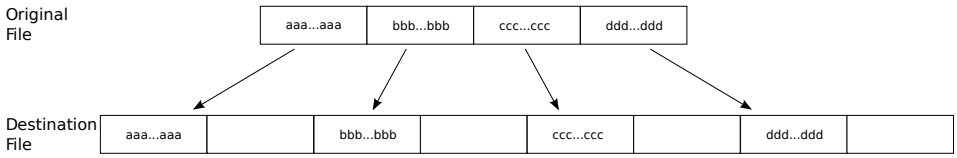
**3.-**Consider the following code snippet:

```
int a = 3;
void main() {
    int b=2;
    int p;
    for (i=0;i<4;i++) {
         p=fork();
         if (p==0) {
             b++;
             execlp("command":...);
             a++;
         }
         else {
              wait();
              a++;
              b--;
         }
     }
    printf("a=%d,b=%d\n",a,b);
}
```

*a*) How many processes are created (excluding the parent process)? What is the maximum number of processes running simultaneously?

*b*) What will the last statement of the program print on the screen ?

**4.-**An inexperienced programmer aims to write an application that creates the interspersed version of a file in parallel. The figure below illustrates the process of creating the interspersed version of a file: the first block in the file is copied onto the destination file. Then an empty block (gap) is left. Another block from the origin file is copied onto the destination file and another empty block is left, and so on.




The programmer creates a first version of the application that creates the interspersed version of a 4-block file by means of 4 processes. The source code is as follows:

```
1  #define BLOCK 4096
2  char buf[BLOCK] = "xxxxxxx...xxxxx";

4  void copy_block(int fdo, int fdd) {
5      read(fdo,buf,BLOCK);
6      write(fdd,buf,BLOCK);
7  }

9  void main() {
10 pid_t pid;
11 int fdo,fdd,i;
12 fdo = open("Origin",O_RDONLY);
13 fdd = open("Destination",O_RDWR|O_CREAT|
       O_TRUNC,0666);
14 for (i=0; i < 4; i++) {
15    lseek(fdo,i*BLOCK, SEEK_SET);
16    lseek(fdd,2*i*BLOCK,SEEK_SET);
17    pid = fork();
18    if (pid==0){
19        copy_block(fdo,fdd);
20        exit(0);
21     }
22 }
23 while (wait(NULL)!=-1) { };
24 read(fdd,buf,BLOCK);
25 lseek(fdd,0,SEEK_SET);
26 read(fdd, buf,BLOCK);
27 }
```

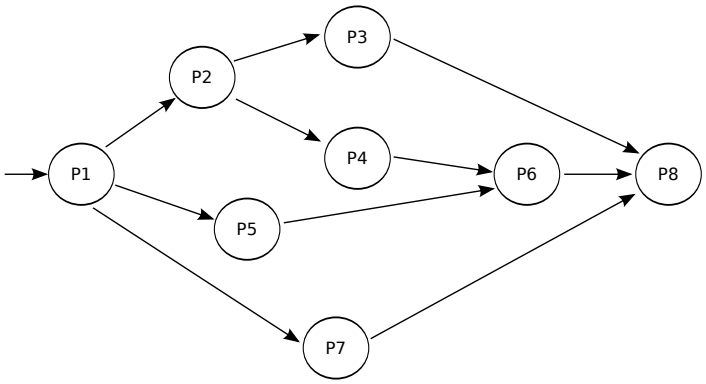
Provide an answer to the following questions assuming that the parent process has the highest priority followed by child processes in the order in which they are created (i.e., the first child process has a higher priority than that of the second child process.).

*a*) Indicate the contents of the `buf` array right after executing the statements at lines 24 and 26. Justify your answer.

*b*) Does the program create the interspersed version of the file correctly?. If not, create a parallel program (consisting of multiple processes) that works correctly without making any assumption regarding the order of execution of the various processes.

*c*) Consider that the files are stored in a UNIX file system that uses blocks of 4KB (4096 bytes) pointers of 4 bytes and inodes featuring 3 direct pointers and one simple indirect pointer. Represent a potential final state for the inode of the destination file after executing the original program.

**5.-**Use the fork(), exec(), exit() and wait() system calls to enforce the synchronization of 8 processes depicted in the graph below. To do so, write a C program (with a main() function) that serves as a master process; this process must launch each process shown in the graph separately (via fork()+exec()) but respecting the order indicated in the directed graph (e.g. P6 cannot be launched until P4 and P5 terminate).




**6.-**Consider the following code snippet:

```
int fd = -1;
char buf1[4]="aaaa";
char buf2[4]="bbbb";

int main() {
   pthread_t tid1, tid2;
   pthread_create(&tid1,NULL,thread1,NULL);
   pthread_create(&tid2,NULL,thread2,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
   close(fd);
}

void* thread1(void* arg) {
   fd=open("test",O_RDWR|O_CREAT|O_TRUNC,0666);
   write(fd,buf1,4);
}
void* thread2(void* arg) {
   while (fd==-1) {};
   write(fd,buf2,4);
}
```

Indicate whether the following statements are true or false about this multithreaded program:

*a*) Thread 2 will stay in the while loop throughout the execution of the `thread2()` function.

*b*) The write operation in thread 2 will lead to an error, since the file was not opened beforehand.

*c*) After executing the program the *test* file will store either "aaaa" or "bbbb". No other alternative is possible.

*d*) The invocation to close() in the main function cannot return -1 (i.e., the call always succeeds).

**7.-**In a single-processor system four processes are launched at different time instants. The table below indicates each process's arrival time and its execution profile (duration of a CPU burst followed by the duration of an I/O burst, followed by the duration of another CPU burst, and so on).

| **Process** | **Arrival time** | **CPU** | **I/O** | **CPU** | **I/O** |
|---|---|---|---|---|---|
| P1 | 0 | 1 | 5 | 1 | |
| P2 | 1 | 3 | 1 | 1 | 1 |
| P3 | 0 | 5 | 4 | 1 | |
| P4 | 3 | 3 | 2 | 1 | 1 |

For the different scheduling algorithms indicate the completion time and waiting time of each process, as well as the value of the following global metrics: throughput, percentage of CPU utilization, average completion time and average waiting time. Assume that at t=0, P1 is enqueued before P3 in the scheduler's run queue.

*a*) FCFS
*b*) SJF
*c*) RR with quantum=3
*d*) RR with quantum=2

**8.-**Five CPU-intensive batch processes (A,B,C,D and E) are launched at the same time on a system featuring one processor. The estimated CPU time of each process is 10, 6, 2, 4, and 8 minutes, respectively. The priority of each process is as follows: 3, 5, 2, 1 and 4, respectively, where 5 denotes the highest priority. Neglecting the time associated with context switches, indicate the completion time (or turnaround time) of each process as well as the average turnaround time observed for the following scheduling algorithms:

*a*) An ideal completely fair scheduler (RR q$\approx$ 0), which ensures an even distribution of CPU time among processes regardless of the duration of the scheduling interval considered.
*b*) A priority-based scheduler
*c*) FCFS
*d*) SJF

**9.-**Consider a single-processor system with a 3-level (A, B and C) feedback queue scheduling policy. Suppose further that processes in each priority level are scheduled in a round-robin (RR) fashion, where the quantum for each level is 2, 4 and 8, respectively (from the highest to the lowest priority level). For simplicity, on this system, the scheduler does not preempt a process if a higher priority process wakes up.

At t=0, three processes (P1, P2 and P3) are enqueued in the highest-priority run queue (level A), and no other processes are enqueued in the run queues associated with the other two levels (B and C). The processes have the following recurrent execution profiles:

P1: (3-CPU,5-I/O)  P2: (8-CPU,5-I/O)  P3: (5-CPU,5-I/O)

Using a time diagram for the first 30 time units, indicate the state of each process as well as which processes are enqueued in each priority level over time. In addition, calculate the percentage of CPU usage as well as the waiting time of each process during the first 30 time units. Consider that the scheduler increments the priority of a process when it does not use up its quantum. Suppose further that the priority of a process is decremented automatically in the event that the process consumes its entire quantum.