Universidad Carlos III de Madrid	**Departamento de Informática** **Grado en Ingeniería Informática** **Sistemas Operativos** **Examen de la convocatoria ordinaria** **20 de enero de 2011**	ARCOS

NOTES:

- Exam time: **3 hours**.
- You can NOT use either notes or calculators.
- The mobile phones have to be turned off during the exam (turned off and not only silenced)

NAME:
NIA:
GROUP:

Exercice 1. Theory and small questions [3 points]:

Q1. When does a process enter the zombie state?
A.- When his father dies and he has not finished yet.
B.- When his father dies without calling `wait` for him.
C.- When the process dies and his father has not called `wait` for him.
D.- When the process dies and his father has not finished yet.

Explain why.

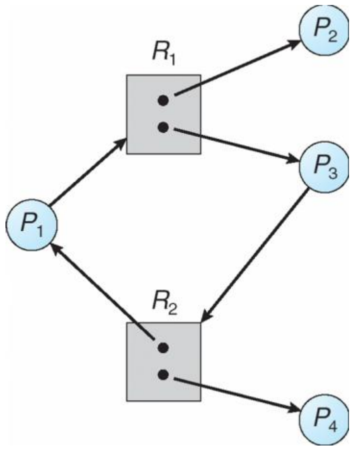
Solution: C
A zombie is created only after a process dies and stays in that state until a parent picks up the exit code with the wait operation.

Q2. Which of the following scheduling policies is best suited for a time-sharing multiprogramming system?
A.- Shortest Job First.
B.- Round-Robin.
C.- Priorities.
D.- FIFO.

Explain why.

Solution: B. It is the only policy that guarantees interleaving of concurrently executing programs.

Q3. Is the following system in a deadlock? Why or why not?



Solution: No deadlock. P2 and P4 eventually release the resource R 1 and R2 and P1 and P3 gain access to the resources.

Q4. Is the following system in a safe state? Why or why not?

	Allocation	Request	Availability
	A B	A B	A B
P1	0 0	5 1	1 5
P2	3 0	2 3	
P3	1 0	1 4	

Solution: safe state. P3 executes => Availability (2 5). P2 executes => Availability (5 5). Finally P1 executes.

Exercice 2 [2,5 points]:

1. Eratostene sieve is a algorithm of generating a series of consecutive prime numbers. In this exercise you are required to implement the Eratostene sieve with ***processes and pipes***. The algorithm works in the following way :

- A father creates N children.
- The father connects to the child 0.
- Child i connects to children `i-1` and i+1 creating a pipeline.
- The father generates a series of consecutive numbers 2, 3, 4, 5,..... and sends them one by one to child 0. The sequence is terminated by the number -1, which indicates that all the children have to finish.
- Each child stores the first number to receive in a variable `local_prime`. Subsequently, if the number it receives is not a multiple of `local_prime`, it forwards it to the next child.
- When a child receive a -1, it forwards it to the next children and terminates
- The father waits for the termination of all children before it exits.

Answer:

```
#define N 10

int main()
{
  int p[N][2];
  int i;
  // CREATE PIPES
  for (i=0;i<N;i++)
    pipe(p[i]);

  for (i=0;i<N;i++){
    if (!fork()) {
      int local_prime, y;
      close(p[i][1]);
      close(p[i+1][0]);

      // FIRST NUMBER TO BE RECEIVED IS A PRIME NUMBER
      read(p[i][0], &local_prime, 4);
      printf(" CHILD %d received PRIME NUMBER %d¥n", getpid(), local_prime);

      while (1) {
              read(p[i][0], &y, 4);

              // FORWARD A NUMBER ONLY IF IT IS NOT MULTIPLE OF THE LOCAL PRIME
   NUMBER
              if ((y%local_prime) &&(i < N-1))
                      write(p[i+1][1], &y, 4);
              // FINISH WHEN RECEIVING -1
              if (y == -1)
                      break;
      }
      exit(0);
    }
  }

  // SENDING A STREAM OF NUMBERS
  for (i=2;i<N*10;i++)
     write(p[0][1],&i,4);
  // SENDING -1 TO TERMINATE
  i=-1;
  write(p[0][1],&i,4);

  // WAIING FOR THE CHILDREN TO TERMINATE
  for (i=0;i<N;i++){
    int status,pid;
    pid=wait(&status);
    printf(" CHILD %d terminated. ¥n", pid);
  }
}
```

Exercice 3 [2 points]:

An array of 100 elements is processed for 200 iterations in the following way. In each iteration, for each index between 1 and 98 the average of the item and its two neighbors items is calculated and stored in the same item, i.e . **v[i]= (v[i-1]+v[i]+v[i+1])/3.** The neighbor of index 0 is considered to be 99 and viceversa.

The program is optimized by executing it in two threads: one half in **thread0** and another half in **thread1**. In each iteration the calculations are first done in an auxiliary array **vaux** local to each thread, so that the calculated values do not overwrite the current values. At the end of the iteration the result is written to the v array.

Summarizing each thread performs the following operations:

1. Calculate the average in the auxiliary array **vaux.**
2. When the calculation is finished, wait for the other thread to finish, before copying the vaux to v. In this way it is avoided that one thread overwrites necessary values.
3. Once a thread copies the data to the array v it must wait that the other thread also finishes copying the data to v before starting a new iteration.

Below you are given the main program structure. You are required to add the necessary variables and synchronization at marks I, II, III, IV and V.

```
#define SIZE 100
#define NUMITER 200
pthread_attr_t attr;
pthread_t idth[2];
float v[SIZE];

I  //ADD THE NECESSARY VARIABLES




void *thread0(void *num) {
  int i,j;
  float vaux[SIZE/2];
```

II **Thread 0**//ADD THE NECESSARY VARIABLES

```
  for (j=0; j<NUMITER; j++) {
        vaux [0]= (v[SIZE-1]+v[0]+v[1])/3;
        for (i=1; i<SIZE/2 ; i++)
                vaux [i]= (v[i-1]+v[i]+v[i+1])/3;
```

III **Thread 0**//ADD THE NECESSARY SYNCHRONIZATION OPERATIONS

```
             }
  pthread_exit(0);
}
void *thread1(void *num) {
   int i,j;
   float vaux[SIZE/2];
```

IV **Thread 1**// ADD THE NECESSARY VARIABLES

```
    for (j=0; j<NUMITER; j++) {
        for (i=SIZE/2; i<SIZE-1 ; i++)
            vaux [i-SIZE/2]= (v[i-1]+v[i]+v[i+1])/3;
        vaux [SIZE/2-1]= (v[SIZE-2]+v[SIZE-1]+v[0])/3;
```

V **Thread 1** // ADD THE NECESSARY SYNCHRONIZATION OPERATIONS AND COPY THE DATA TO THE ORIGINAL ARRAY

```
    }
   pthread_exit(0);
}

int main(){
  int i;
  initArray(); // initialize the array values
  pthread_mutex_init (&mtx1, NULL);
  pthread_mutex_init (&mtx2, NULL);
  pthread_attr_init(&attr);
  pthread_create(&idth[0],&attr,thread0,NULL);
  pthread_create(&idth[1],&attr,thread1,NULL);
  for (i=0; i<2; i++)
      pthread_join(idth[i],NULL);
   return(0);
}
```

Solution

```
I
pthread_mutex_t mtx1,mtx2;
pthread_cond_t varcond1,varcond2;
int contfin1=0, contfin2=0;

II
 int init1=0, init2=0;

III
 pthread_mutex_lock (&mtx1);
   contfin1++;
   //The one waiting will reinit contfin1 for the next iteration
   if (contfin1==1) init1=1;
  //Wait for the other to finish
   while (contfin1!=2)
     pthread_cond_wait(&varcond1, &mtx1);
   for (i=0; i<SIZE/2 ; i++)
     v[i]=vaux[i];
   pthread_cond_signal(&varcond1);
   if (init1){
     contfin1=0;
```

```
    init1=0;
  }
 pthread_mutex_unlock (&mtx1);

 pthread_mutex_lock (&mtx2);
  contfin2++;
  if (contfin2==1) init2=1;
 //Wait for the other to copy the data
  while (contfin2!=2)
     pthread_cond_wait(&varcond2, &mtx2);
  pthread_cond_signal(&varcond2);
  if (init2){
    contfin2=0;
    init2=0;
  }
 pthread_mutex_unlock (&mtx2);
```

IV

```
 int init1=0, init2=0;
```

V

Same as thread 0 except:

```
    for (i=SIZE/2; i<SIZE ; i++)
      v[i]=vaux[i-SIZE/2];
```

Exercise 4 [2,5 points]:

In a UNIX file system the block size have 1024 bytes, the block pointers are 2 bytes long, and the inodes contain:

- 16 direct block pointers.
- 2 simple indirect block pointers.
- 2 double indirect block pointers.

Answer the following questions:

a. How many disk accesses are necessary for reading a file of size 1 Mbytes? Assume that once a block is accessed from disk it is kept in cache.

b. What is the theoretical maximum size of a file?

c. What is the maximum possible size of the whole file system?

d. Which are the advantages and disadvantages of increasing the block size? Hint: Refer to the question b and c and to the efficiency of disk space utilization.

Solution:

a. Necessary read data blocks:
10 MB/1024 = (10*2^20)/2^10 = 10*2^10 bloques de datos

Simple indirection blocks direct:
(1024/2)*1024= (2^10/2)*2^10 = 2^19 = 512 KB

Double indirection blocks direct:
(1024/2)*(1024/2)*1024 = (2^10/2)*(2^10/2)*2^10 = 2^28 = 256 MB

Total:

10*2^10 data blocks + 1 access to inode block, 2 accesses to simple indirect blocks + 1 access to double indirect blocks

b. (16*2^10)+ 2*((2^10/2)*2^10) + 2*((2^10/2)*(2^10/2)*2^10) =
16KB + 2*512KB + 2*256MB = 16KB + 1MB + 512MB

c. You can direct in the FS maximum 2^16 blocks. The maximum size: 2^16*1KB = 64 MB

d. Consequences:
- Increases the file size
- Increases the device size of the file system
- Improves the performance (less disk accesses to a file)
- Increases the probability of internal fragmentation.