Universidad Carlos III de Madrid	**Departamento de Informática** **Grado en Ingeniería Informática** **Sistemas Operativos** **Examen de la convocatoria extraordinaria** **21 de junio de 2011**	ARCOS

NAME:

NIA:

Exercise 1. Theory [2.5 points]:

a. Compare and contrast Shortest Job First and Round Robin scheduling algorithms in terms of produced scheduled lengths, fairness and the possibility of producing starvation.
b. What are the four conditions required for deadlock to occur?
c. Why might the direct blocks be stored in the inode itself?
d. Name some advantages and disadvantages of user-level threads.
e. Describe and compare the first fit and best fit memory management algorithms.

Exercise 2 [2.5 points]:

Write a C program consisting of a father process and several children. Every time it receives a SIGUSR1 signal, the father should create a new child process and print the number of existing children. The father must stay in an infinite loop, waiting for children to finish and for signals to arrive. Each child must live for a random number of seconds between 5 and 20 seconds and then exit.

Exercise 3 [2,5 points]:

Given a disk formatted with a Unix file system with a block size of 1KBytes. A block address is stored on 4 bytes and the i-nodes contain a reference count field and the traditional structure for mapping files on disk (10 direct addresses, 1 simple indirect address, 1 double indirect address and 1 triple indirect address).

You are required to:

a. Explain what the reference count is used for.

b. Calculate how many disk blocks are needed for storing a file of 8200 KBytes. Explain the calculus in detail.

c. Consider the absolute pathname: /usr/student/john/a.pdf. How many disk blocks are needed in order to retrieve the first block of data from the file *a.pdf*. You may assume that each directory file requires a single block and that the inode for the root directory is already cached by the OS. Briefly justify your answer.

Exercice 4 [2,5 points]:

The following code provides a solution to the producer consumer problem using semaphores.

```
int BufferSize = . . . ;

semaphore mutex = . . . ;
semaphore empty = . . .  ;
semaphore full = . . .;

producer()
{
  int item;

  while (TRUE) {
    make_new(item);
    down(&empty);
    down(&mutex);
    put_item(item);
    up(&mutex);
    up(&full);
    }
}

consumer()
{
  int item;

  while (TRUE) {
    down(&full);
    down(&mutex);
    remove_item(item);
    up(&mutex);
    up(&empty);
    consume_item(widget);
    }
}
```

Answer the following questions:

a. Initialize the values of BufferSize and all three semaphores, assuming that the buffer is supposed to have a maximum of 10 items.
b. Is the mutex semaphore necessary? Justify your answer.
c. Is it possible to simplify the program by replacing the full and empty semaphores with only one semaphore, let's call it count? Justify your answer. If possible, rewrite the program.
d. In the provided solution, is there a problem if we switch down(&empty) and down(mutex) in the producer?

Universidad Carlos III de Madrid	**Departamento de Informática** **Grado en Ingeniería Informática** **Sistemas Operativos** **Examen de la convocatoria extraordinaria** **21 de junio de 2011**	ARCOS

SOLUTIONS:

Exercise 1

a. SJF : unfair, starvation possible, shortest schedule length
RR: fair, no starvation, larger schedule length

b. Mutual exclusion: only one thread can use a resource at a time
Hold and wait: a resource can be held, then blocking whilst waiting for more resources
No preemption: resources cannot be forcibly taken away from process holding it
Circular wait: A circular chain of 2 or more processes, each or which are waiting for a resouce held by the next member in the chain.

c. Quick access to the start of the file stored in the inode itself. For small files, no direct blocks required, entire file can be in the inode. This improves the performance for small files.

d. Advantages: User level threads are more configurable, as they may use any scheduling algorithm, and they can also run on any OS (probably even DOS) since they do not require kernel support for multiple threads. User level threading is also much faster, as they don't need to trap to kernel space and back again.
Disadvantages: without kernel thread support, the threads must be cooperative. Yields are annoying, and require every thread to be well written, e.g. having enough yields and not monopolizing cpu time. The I/O must be non-blocking, then require extra checking to deal with cases that would normally block. This is not always possible (either because of lack of async I/O system calls, or polling system calls), or because events such as page faults (which are always synchronous) will block the process and all the user-level threads in side.. User level threads cannot take advantage of multiple CPUs, as the OS cannot dispatch threads to different processors, or potentially do true parallelism, because the OS only sees 1 process with 1 thread.

e. First fit faster but more fragmentation
Best fit slower and optimal fragmentation

Exercise 2

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int createChild=0;

void catchSignal (int s){
  createChild=1;
}
int child (){
int waiting;
  srandom (getpid());
  waiting=random ()%16 + 5;
  printf ("Child %d waits %d seconds\n", getpid(), waiting);
  sleep (waiting);
  _exit (0);
}
main (){
struct sigaction sa1;
int pidchild;
int countchild=0;

 sa1.sa_handler=catchSignal;
 sa1.sa_flags=0;
 sigemptyset(&(sa1.sa_mask));
 sigaction (SIGUSR1, &sa1,NULL);
while (1){
    while  (createChild==0)
      if (pidchild=wait (NULL) >0){
        contchild--;
      }
      else {
         sleep(1);
      }

    if (fork() == 0)
      child ();
    countchild++;
    createChild=0;
    printf ("There are %d children alife \n",countchild);
  }
}
```

Exercise 3

a) The reference count is the number of hard links (names in directories) presently pointing at a particular inode. When this reaches zero, it is an indication that that inode can be deleted; since nothing points at it anymore (it has no name).

b) **Nr of blocks =** 1 i-node + 10 direct blocks + 1 indirect block + 256 blocks from indirect block + 1 double indirect block + 31 blocks pointed to from double indirect block + 7934 blocks from double indirect = **8234 blocks**

c) Root directory is cached. Number of blocks to get to a.pdf is 6; The inode for a.pdf is 7th and to get the first block of the file requires 8 accesses.

Exercise 4

a)

```
int BufferSize = 10;

  semaphore mutex = 1;              // Controls access to
critical section
  semaphore empty = BufferSize;     // counts number of empty
buffer slots
  semaphore full = 0;               // counts number of full
buffer slots
```

b) The mutex semaphore is necessary for implementing the critical section. Without the critical section the race condition for the same buffer may produce an incorrect behavior.

c) It is not possible to implement the solution with one semaphore, because there are two different situations in which the program has to block. Semaphore can block only when a down operations is done and the value is 0.

d) There is a problem, a deadlock may occur. If the buffer is full, empty=0. Producer enters critical section by calling wait(&mutex) and blocks when calling wait(&empty). Consumer can never can consume the item, because it will block when calling wait(&mutex). Both would block => deadlock.