

Operating Systems exercise – week 5 – Complete answers.

Process synchronisation:

1) What is the meaning of the term busy waiting? What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

Answer.

- A process is waiting for an event to occur and it does so by executing instructions.
- A process is waiting for an event to occur in some waiting queue (e.g., I/O, semaphore) and it does so without having the CPU assigned to it.
- Busy waiting cannot be avoided altogether.

2) In the textbook, it is mentioned that disabling interrupts frequently could affect the system's clock. Explain why it could, and how such effects could be minimized.

Answer.

Timekeeping is usually based on interrupts generated by a clock crystal. After a fixed number of oscillations, which equal the passage of an amount of time, an interrupt occurs and the computer increases its stored time. If interrupts are disabled, these clock ticks can be lost, and the time value is not incremented.

3) Show that, if the wait and signal operations are not executed atomically, then mutual exclusion may be violated.

Answer.

Suppose the value of semaphore $S = 1$ and processes P1 and P2 execute wait(S) concurrently.

- a. T0: P1 determines that value of $S = 1$
- b. T1: P2 determines that value of $S = 1$
- c. T2: P1 decrements S by 1 and enters critical section
- d. T3: P2 decrements S by 1 and enters critical section

4) What does execute "atomically" mean?

Answer.

Executed as a group, with no context switches possible until all of the statements in the group are completed.

5) Define the wait-(postpone) operation, wait(S).

Answer.

While semaphore is nonpositive, wait; when semaphore becomes positive, subtract one and exit.

6) Define the wakeup operation signal(S).

Answer.

Add 1 to the semaphore.

7) Suppose we want to execute the statements S1, S2, and S3 in sequence, but that S2 has to be executed exclusively for one process at a time. Write the code needed using semaphores.

Answer.

Assume $Q = 1$ initially.

S1;

wait(Q);

S2;

signal(Q);

S3.

Deadlock 1:

1) Is it possible to have a deadlock involving only one single process? Explain your answer.

Answer.

No. This follows directly from the hold-and-wait condition.

2) Consider the traffic deadlock depicted in Figure 8.11 (see handout).

a) Show that the four necessary conditions for deadlock indeed hold in this example.

b) State a simple rule that will avoid deadlocks in this system.

Answer.

a) Each section of the street is considered a resource.

- Mutual-exclusion —only one vehicle on a section of the street.

- Hold-and-wait — each vehicle is occupying a section of the street and is waiting to move to the next section.

- No-preemption — a section of a street that is occupied by a vehicle cannot be taken away from the vehicle unless the car moves to the next section.

- Circular-wait —each vehicle is waiting for the next vehicle in front of it to move.

b) Allow a vehicle to cross an intersection only if it is assured that the vehicle will not have to stop at the intersection.

3) List types of resources we might consider in deadlock problems on computers.

Answer.

CPU cycles, memory space, files, I/O devices, tape drives, printers.

4) Define deadlock.

Answer.

A situation where every process is waiting for an event that can be triggered only by another process.

5) What are the four necessary conditions needed before deadlock can occur?

Answer.

This answer is not available as it may be in the portfolio

6) True/False: In a system resource allocation graph (SRAG), if there are no cycles, then no deadlock exists.

Answer.

True.

7) True/False: In an SRAG, if there are cycles, then deadlock exists.

Answer.

False; a deadlock is possible.

8) Give an example of a simple resource deadlock involving 3 processes and 3 resources. Draw the Resource Allocation Graph that illustrates this.

Answer.

Experiments with processes in Linux – answers.

9) Now we are going to enter, compile and run some C programs in order to illustrate some of the things we have been saying about processes in Linux.

Leave the terminal application running as we will need that later. Start an editor of your choice (you could use the simple editor (Applications then Accessories then Text Editor)).

Now type as a template the following simple C program that creates a child process using the fork system call.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    pid_t val;

    printf("PID before fork(): %d\n", (int) getpid());
    if (val = fork())
        printf("PID after fork(): %d – so who am I?\n", (int) getpid());
    else
        printf("PID after fork(): %d – so who am I?\n", (int) getpid());
}
```

16) Use your forktest.c program as the basis, but modify it by deleting the printf's that occur under the if and the else that output the PIDs and replace them with code for the parent to output the series of odd (in French 'impair') numbers in an infinite loop (e.g. 1, 3, 5, 7, 9,...) and for the child process to output the series of even (in French 'pair') numbers in an infinite loop (e.g. 2, 4, 6, 8, 10,...). You could do this by using a for loop under the relevant branch of the if..else statement.

Answer:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
```

```

pid_t val;
int i;

    if (val = fork())
        for (i=1; i > 0; i=i+2)
            printf("%d\n",i);
    else
        for (i=2; i > 0; i=i+2)
            printf("%d\n",i);

}

```

18) To solve the above problem we need to introduce a suitable delay into the processing of the outputting of the odd and even numbers. So modify your code to introduce a delay. You could for example add the following code before each printf statement in the for loops that output the odd and even numbers:

```

    for (n=0; n < 50000000; n++);

```

That is right 50 million. Modern processors are so fast that we need something like this to slow things down enough for us to see what is going on.

n must be declared as an int.

Answer:

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    pid_t val;
    int i,j;

    if (val = fork())
        for (i=1; i < 1000; i=i+2){
            printf("%d\n",i);
            for (j=0; j < 50000000; j++);
        }
    else
        for (i=2; i < 1000; i=i+2){
            printf("%d\n",i);

```

```

        for (j=0;j < 500000000; j++);
    }
}

```

20) Now we want to run as a child process not just programs that are copies of the parent process but new programs as well. As we saw in the lecture we do this using a version of the `exec()` system call.

So now go back to the `forktest1.c` program and modify that by making the child process load a new program over its process space using `exec`. The simplest version of `exec` to use is `execl()`. This will load up the program whose pathname (location) is given in the first parameter with the second and subsequent parameters being the command line strings for the program (includes command name itself as the first command line string. So to execute the `ls` command you need to call `execl` as,

```
execl("/bin/ls", "ls", "-l", 0);
```

This needs to go in the code that is to be executed by the child process.

Answer:

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
    pid_t val;

    if (val = fork())
        printf("parent");
    else
        execl("/bin/ls", "ls", "-l", 0);
}

```

21) Up till now the parent process has continued to execute while the child process has executed. We can use the `wait()` system call to make the parent process suspend itself and wait until it is woken up by the termination of the child process.

So modify `forktest4.c` to make the parent process wait for the child. You will need to use the `wait()` system call.

You will need to add to the #include list

```
#include <sys/wait.h>
```

The wait() system call takes a pointer to an int as a parameter. It is the location where the exit status of the child will be placed when it terminates. So you will need to add an int declaration,

```
int status;
```

to your code. Then you can call wait() as

```
wait(&status);
```

This needs to go in the code that is to be executed by the parent process.

The & in the parameter gives the address of the variable status to wait() rather than the value.

Answer:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
main()
{
    pid_t val;
    int status;
```

```
        if (val = fork()) {
            printf("parent");
            wait(&status);
        }
        else
            execl("/bin/ls", "ls", "-l", 0);
    }
```