

Operating Systems exercise – week 5

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.
- 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.
- 3) Show that, if the wait and signal operations are not executed atomically, then mutual exclusion may be violated.
- 4) What does execute "atomically" mean?
- 5) Define the wait-(postpone) operation, wait(S).
- 6) Define the wakeup operation signal(S).
- 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.

Deadlock 1:

- 1) Is it possible to have a deadlock involving only one single process? Explain your answer.
- 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.
- 3) List types of resources we might consider in deadlock problems on computers.
- 4) Define deadlock.
- 5) What are the four necessary conditions needed before deadlock can occur?
- 6) True/False: In a system resource allocation graph (SRAG), if there are no cycles, then no deadlock exists.
- 7) True/False: In an SRAG, if there are cycles, then deadlock exists.
- 8) Give an example of a simple resource deadlock involving 3 processes and 3 resources. Draw the Resource Allocation Graph that illustrates this.

Experiments with processes in Linux.

Start VMPlayer as before – Start/VMware/VMware Player. Open the Linux 2.6.x kernel.vmx file. Remember login id is user1 and password is vmware.

Also remember that CTRL+G makes keyboard and mouse actions go to the virtual machine, whereas CTRL+ALT will make it go to the normal PC.

Start the terminal [Applications menu then Accessories and then Terminal].

1) There is a simple command that we can use to find out something about the processes running on the system.

The command ps (process status) gives a lot of information about the processes on the system. At the prompt type,

```
ps
```

This should give you a list of processes that are owned by you. There are probably 2 processes on the list. The last column gives the name of the command(program) that the process represents. For the first process it will probably say ‘bash’, which is the name of the shell program you are using. The second process is probably ‘ps’, which of course is the name of the command we have just run.

The first column PID shows the process identity number – the unique number assigned to your process by the system when it is created. The next column shows the terminal that that process is associated with. TIME is the amount of CPU time that has been used by that process so far. Probably 0 at the moment.

2) Type,

```
ps -e
```

This gives you a listing of all the processes on the system. You can use the scrollbar to look back at a few of them. You will not understand most of them, but you will notice some processes that were started by Gnome (the GUI we are using), including the terminal application that you are using. Of particular interest is the process called init which has the process id of 1. This is the first process that was run on the system and is the ultimate parent (or root of the tree of processes).

3) Type,

```
ps -f
```

The -f gives a fuller listing of information about the processes. You will see the parents PID and also the start time of the process (STIME).

4) Type,

```
ps -Lf
```

This gives some information about threads. The column headed LWP gives the id. Of any threads and NLWP gives the number of threads associated with the process.

5) Type,

```
ps -eLf
```

You will get all the processes and threads on the system. Again if you use the scrollbar you will find a few processes that have more than one thread.

Select one of them and write down the PID of the process and the thread id of the threads that belong to that process. You should notice, the same PID but different thread ids.

6) Look at the manual pages for ps

```
man ps
```

You will find many more options for ps. Try a few and see what you get.

7) Normally when a user enters a command in the shell, the shell process executes a fork to create a child process which then calls some version of `exec()` system call to copy the code/data space, etc of the command's program into the child process. The child process then executes, while the shell suspends itself waiting for the child to finish. Once finished the shell prompt will then appear again.

If you put `&` after a command, then the command will run in the background. This means that instead of suspending itself when it has forked to produce a child process, the shell will continue and return a prompt.

`Sleep n` where `n` is a number of seconds is a program that simply suspends itself for `n` seconds and then terminates. So going

```
sleep 5
```

will create the sleep process and the shell will suspend itself (no prompt) for 5 seconds. Try it.

Now try

```
sleep 5&
```

The sleep process is running in the background and the prompt re-appears immediately. You will also see something like [1] 2345. The number in [] is the number of the background processes you have running. The other number is the PID of the sleep process that is being run in the background. What is the PID of your sleep process?

8) We can look at the sleep process using ps by typing ps at the prompt immediately after we have set sleep running. So type in

```
sleep 60&
ps
```

You will see output that is reporting to you that the sleep 5 process has completed or Done. You will also see the standard ps output, but this time you will see the sleep process listed.

You can always terminate a process that is running in the background if you know its PID (you can find it if you don't know by using ps and noting the PID number given there). You can then use the kill command kill followed by the PID number.

9) Try the command

```
top
```

Look at the various columns. What information is it giving you about the processes on the system? Use man to find out more about this command.

10) 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());
    }
```

11) Save this as a C file – so you could call it forktest1.c

12) Go to the terminal application and use `ls -l` to check that it is in your user directory. The listing should show that it is not executable i.e. no 'x' in the list of modes of access – just 'rw'.

13) You can compile it using `cc` with the `-o` option. The `-o` option allows you to give the executable file a sensible name (otherwise the default is called `a.out`). So

```
cc -o forktest1 forktest1.c
```

14) Now use `ls -l` again and you should find a file called `forktest1` as well as `forktest1.c`. The new file `forktest1` should show that it is executable.

15) Before we execute this program, we need to change the `PATH` variable so that the system looks at the current working directory for executables as well as standard locations. So we need to type,

```
PATH=$PATH:./
```

16) Now we can run our program. All we need to do is type the name of the executable file at the command prompt. So,

```
forktest1
```

What behavior do you get? What is the PID of the child process?

Execute the program several times. Are the 2 messages always executed in the same order?

17) Use your `forktest.c` program as the basis, but modify it by deleting the `printfs` 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.

18) Now save and compile your program (`forktest2.c`). Run it as before.

```
forktest2
```

The problem is that the list of numbers produced is printed to the screen so quickly that you cannot see the CPU being switched between the processes. You can terminate the process by using CTRL+C.

19) 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.

20) Compile and run this new program. What you will see is the result of the 2 processes being scheduled onto the CPU.

21) 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.

22) Save (as forktest4.c), compile and run this program. What output do you get? You could try using other commands.

23) 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.

24) Save, compile and run your program. What output do you get? Does it differ from