

Operating Systems exercises – week 7

Memory management questions.

- 1) Explain the difference between logical and physical addresses.
- 2) Explain the difference between internal and external fragmentation.
- 3) Explain the following allocation algorithms:
 - a. First-fit
 - b. Best-fit
 - c. Worst-fit
- 4) Given memory partitions of 200K, 425K, 100K, 700K, and 300K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

Remember that space left over after a process has been placed in a partition will then become part of the free space available for later allocations.

- 5) Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.
 - a. How many bits are there in the logical address?
 - b. How many bits are there in the physical address?
- 6) Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory? Why should it or should it not?

Linux interprocess communication exercises.

1) Start the Linux virtual machine [Start, then VMware, then VMware Player, and select Linux 2.6.x kernel.vmx file]. Remember login id is user1 and password is vmware.virtual machine.

Remember to press CTRL+G to direct input to the virtual machine and CTRL+ALT to direct it to the host computer.

2) 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 interprocess communication in Linux.

Start an editor of your choice (you could use the simple editor (Applications then Accessories then Text Editor).

3) Remember that we wrote a program 2 weeks ago that that forked to produce a child process and the parent process output odd (impair) numbers while the child process output even (pair) numbers. We will use a program that repeatedly outputs the alphabet to illustrate interprocess communication.

First write a program that repeatedly outputs the alphabet using an infinite loop.

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz  
abcdefghijklmnopqrstuvwxyz.....
```

You could use a for loop like for (i=0;i<26;i++,ch++) and putchar(ch) to output the chars.

You will probably also want to produce a delay like we did last time using something like for (i=0; i < 50000000;i++);

4) Save the program as a C file – alpha.c

5) Go to the terminal application [Applications menu then Accessories and then Terminal].and use ls -l to check that the source code program is in your user directory.

6)As before you can compile it using cc with the -o option. Remember the -o option allows you to give the executable file a sensible name (otherwise you get the default a.out). So

```
cc -o alpha alpha.c
```

7) You can now use ls -l again to check that you have an executable file as well as your source code program.

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

`PATH=$PATH:./`

9) Now we can run our program. All we need to do is type the name of the executable file at the command prompt. Make sure it works properly as we will be extending and modifying it.

10) We are going to be modifying the program to use interprocess communication. So first, we need to change the program so that it now uses a `fork()` system call as we did 2 weeks ago with the parent process outputting the alphabet. The child process can do nothing for now.

11) Save, compile and run the program and make sure it works.

12) Now we will modify the program so that the parent process produces the alphabet characters and the child process actually outputs the characters to the screen. In the first version we will use a shared file.

You can use the following code as a template for your program i.e. it creates a file to be shared between parent and child process, and shows you how to write to and read from a file.

It uses the following system calls:

```
int open(const char *path, int flags, mode_t mode)
/* parameters - name of file to open; how to open the file – O_RDWR | O_CREAT is
used to open file for reading and writing and create it if it does not exist; permissions
used on file creation – S_IRUSR | S_IWUSR is used to create file with read and write
permissions for owner.
Returns file descriptor to file opened or -1 if there is an error */
```

```
int write (int fd, void *buf, size_t nbytes)
/* parameters – file descriptor of file; pointer to block of memory where nbytes of
data are written to file; a count of the number of bytes to write.
Returns number of bytes written or -1 if there is an error */
```

```
int read(int fd, void *buf, size_t nbytes)
/* parameters - file descriptor of file; pointer to block of memory where read will put
data it reads from file ; a count of the number of bytes to be read from file.
Returns number of bytes read or -1 if there is an error */
```

```
int close (int fd)
/* parameter – file descriptor for file being closed.
Returns – 0 on success */
```

Modify the code as specified in the comments, so that communication occurs between parent and child using the shared file.

```

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

#define SIZE 26

main()
{
    int i, count, fd;
    char buf[SIZE];
    char ch;

    // create or open a file called 'shared'
    if ((fd = open("shared", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)) == -1)
        printf("error opening file\n");

    if (fork()) { // parent process
        // you will have to put here your code to repeatedly produce alphabet
        // and put it into array buf rather than output to screen

        // code to write contents of buf to shared file
        if (write(fd, buf, SIZE) != SIZE)
            printf("error writing to file\n");
    }
}
else { // child process
    // code to read from shared file
    for (; ;) {
        if (read(fd, buf, SIZE) == -1)
            printf("error reading file\n");
        // you will put in code here to output contents of array buf in child process
    }
}
close(fd);
}

```

13) Look at the size of the file – you will see that it has grown and grows more everytime you run your program.

14) Now we will do the same, but using pipes to communicate between the parent and child process.

For this we can use your existing program with the reads, writes and close just as before. The difference is that instead of using open() to open/create a shared file, we will use the pipe() system call to create a pipe.

So you can delete the statement that opens a file, as we no longer need it.

Remember from the lecture that a pipe is just a buffer with 2 file descriptors associated with it `fda[0]` for reading and `fda[1]` for writing.

You will need to add the declaration

```
int fda[2];
```

The `pipe()` system call takes the file descriptor array e.g. `fda` as its parameter. You should call `pipe()` as follows,

```
if (pipe(fda) == -1)
    printf("error creating pipe\n");
```

The parent process will need to close the pipe for reading

```
close(fda[0]);
```

while the child process will need to close the pipe for writing

```
close(fda[1]);
```

Reading and writing are as before except you will need to use `fda[0]` as the file descriptor for reading and `fda[1]` for writing.

15) You can delete the file 'shared' by typing

```
rm shared
```

you can check that it no longer exists by using `ls`.

16) Save, compile and run the new version of your program. The pipe is very small and does not grow like the file.

17) Now we are going to also try the program using FIFOs or named pipes.

To create a FIFO we use the system call `mknod()`. This has the signature,

```
int mknod(char *path, mode_t mode, dev_t dev)
/* parameters are – name of FIFO to be created; bit values to define access
permissions and type of FIFO – in our case it will be S_IFIFO | S_IRUSR | S_IWUSR
– this means create a FIFO with read and write access permissions; value used when
creating special device files and so will be 0 in our case. Returns a file descriptor to
FIFO or -1 if an error */
```

However, in this case it would probably be easier to create one by typing the following command at the terminal prompt,

```
mknod fifoname p
```

This will create a FIFO in your directory.

Use `ls -l` and you should see the FIFO is present in the directory just like any normal file. Therefore, it can be opened and then read and written to by any process that has the appropriate access permissions.

You would open the FIFO using `open()` in an if statement as before. However, there is a 2 parameter version of `open()` which is more suitable for opening the FIFO and you should use that. The first parameter is the FIFO's name, the second parameter will have the value `O_RDONLY` for read access only and `O_WRONLY` for write access only.

```
if ((fd = open(fifoname, read or write access as above)) == -1)
    printf("error opening FIFO\n");
```

18) So we are going to write 2 programs that are not related to each other i.e. they are no parent and child. One of the programs will do what the parent used to do and produce the alphabet and write it to the FIFO and the other program will read from the FIFO and output the values read to the screen. So the producer program (writer) will need to open the FIFO for write access and the consumer program (reader) will need to open it for read access.

Now write 2 simple programs one that opens the FIFO you have just created for writing and repeatedly writes the alphabet to it and the other program opens the FIFO for reading and reads the alphabet and outputs that to the screen.

19) You should be able to see that it has the advantage of a pipe in being small – from looking at the directory entry using `ls -l` and also that it has the advantage of being available to non-related processes because it has a directory entry in the file system.