CG2271 Real Time Operating Systems

Lab 7 - Processes in Unix

1. Introduction

In this lab we will look at process creation, coordination, communication and termination in Unix. Unix is a general operating system with many variations, including Linux, BSD Unix, AIX, Solaris and MacOS X. There are also real-time variants of Unix like RTLinux, and everything that you learn in this lab is applicable there too.

2. Instructions

Please submit a HARD COPY of your answer book to your tutor at the start of the NEXT lab session, i.e. at the start of the session for Lab 8 There will also be a demonstration at the start of Lab 8 worth 5 marks, and this lab is worth 25 marks in total.

3. Processes in Unix

In Unix, processes are created using the fork() call. The table below shows the important calls that you need to know. To use these you will need to include the "unistd.h" header file.

Call	Description
pid_t fork()	Creates a new child process. Returns 0 to the child process, and the process ID (pid) of the child to the parent.
pid_t wait(int *status)	Waits for a child to terminate. Returns the termination status of the child in "status", as well as the terminated child's pid.
void waitpid(pid_t pid)	Wait for a child specified by "pid" to terminate.
pid_t getpid()	Returns the process ID of the current process.
pid_t getppid()	Returns the process ID of the parent process.

4. Basic Process Creation

Type out the program below in a Unix or Linux system (Windows will NOT work!!) calling it lab7a.c, compile using "gcc lab7a.c –o lab7a" and execute it using "./lab7a", then answer the questions that follow in the answer book provided:

```
// lab7a.c Introduction for fork and wait

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

#define PARENT_DELAY    0
#define CHILD_DELAY     0
```

```
int main()
{
    pid_t pid;
    int i, j, k, status;

    k=15;

    if(pid=fork())
    {
        printf("Process ID of child: %d\n", pid);

        for(i=0; i<10; i++)
        {
            printf("Parent: i=%d k=%d\n", i, k);
            k++;
            // Delay
            sleep(PARENT_DELAY);
        }
    }
    else
    {
        for(i=10; i<50; i++)
        {
            printf("Child: i=%d, k=%d\n", i, k);
            sleep(CHILD_DELAY);
        }
    }
```

Question 1 (5 marks)

The fork() call returns the process ID (pid) of the child to the parent, and 0 to the child. Using Google or any other resources, describe what a pid is and why it is useful.

The program will print out the pid of the parent process's parent. Using "ps" or any similar UNIX command, determine who the parent of lab7a's parent process is.

Question 2 (2 marks)

Look at the values of k printed by the parent and the child. Are the values of k independent between the child and parent? Why or why not?

Now set PARENT_DELAY to 1 and CHILD_DELAY to 2, compile and run the program, and answer the following question:

Question 3 (3 marks)

Why do you think the output of the parent and child become mixed? After lab7a's parent process completes and you return to the shell, does the child continue to execute? Why?

The wait() function call causes a parent process to wait until at least one of its child processes has ended. The wait() function is used in this way:

```
int status;
…
if(fork())
 {
        // parent
        …
        // Wait for child to end.
        wait(&status);
}
else
{
        // child
        …
}
```

Question 4 (3 marks)

Modify your lab7a program so that it does not quit to the shell before the child process ends. Describe what changes you made.

5. Communication between Child and Parent

Unix provides a mechanism called a "pipe" to allow a parent and child to communicate. To create a pipe, first declare a two-element integer array, then pass in the array to the function "pipe", as shown here:

```
int fd[2];
…
pipe(fd);
```

If this call fails it will return a -1. However we will assume that it succeeds, and when it does, fd[0] will be used for reading from the pipe, and fd[1] will be used for writing to the pipe. Both fd[0] and fd[1] are called "file descriptors" when used in conjunction with pipes.

Important: The process that is READING from the pipe should close fd[1], and the process that is WRITING to the pipe should close fd[0], in order for the pipe to work correctly. Use the "close" function, NOT "fclose", to close ends of a pipe. So to close the input end, use:

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

To write to the pipe, use the C "write" function:

```
write(int file_desc, void *buf, size_t size);
```

To read from the pipe, use the C "read" function:

```
read(int file_desc, void *buf, size_t size);
```

In both cases, file_desc is the file descriptor to write to/read from, buf is a pointer to a buffer (not necessarily of type void *), and size is the number of bytes to write/read.

Type out the following program and call it lab7b.c. Then compile using "gcc lab7b.c –o lab7b" and execute it using "./lab7b", and answer the questions that follow:

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

int main()
{
    int fd[2];
    char string[]="Hello child!";
    char buffer[80];
    int pnum=128, cnum;

    pipe(fd);

    if(fork())
    {
        close(fd[0]);
        write(fd[1], &pnum, sizeof(pnum));
        write(fd[1], string, strlen(string)+1);
    }
    else
    {
        close(fd[1]);
        read(fd[0], &cnum, sizeof(cnum));
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent sent message: %s and %d\n", buffer, cnum);
    }
}
```

Question 5(2 marks)

What output do you see on the screen? Describe in a single statement what this program does.

Question 6 (2 marks)

What do you think the sizeof() function returns?

6. Process Creation Challenge

We will now create a program that generates 16384 random integers, and return the number of prime numbers amongst the integers. Our strategy would be to split the list into two, with the parent counting prime numbers in the 0 to 8191 sublist, and the child counting in the 8192 to 16383 sublist. The parent finally prints out the number of prime numbers in the file.

Use the skeleton given below to start with. Call your program lab7c.c. Note you must compile your program with "gcc lab7c.c –lm –o lab7c". The "-lm" is important as we need to bring in the math library where the sqrt function resides.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define NUMELTS 16384

// IMPORTANT: Compile using "gcc lab7c.c .lm -o lab7c".
// The "-lm" is important as it brings in the Math library.

// Implements the naive primality test.
// Returns TRUE if n is a prime number
int prime(int n)
{
    int ret=1, i;

    for(i=2; i<=(int) sqrt(n) && ret; i++)
        ret=n % i;

    return ret;
}

int main()
{
    int data[NUMELTS];

    // Declare other variables here.

    // Create the random number list.
    srand(time(NULL));

    for(i=0; i<NUMELTS; i++)
        data[i]=(int) (((double) rand() / (double) RAND_MAX) * 10000);

    // Now create a parent and child process.
            //PARENT:
                // Check the 0 to 8191 sub-list
                // Then wait for the prime number count from the child.
                // Parent should then print out the number of primes
    // found by it, number of primes found by the child,
                // And the total number of primes found.
           // CHILD:
                // Check the 8192 to 16383 sub-list.
                // Send # of primes found to the parent.
}
```

Question 7 (3 marks)

Cut and paste your completed program into the answer book, and demonstrate your working program to your tutor at the start of the lab session for Lab 8.