Complete Lab Assignments of Systems Programming Course

Arif A. Balik

Undergraduate Student
Sytstems Programming
Department of Computer Science
Arel University
Büyükçekmece, İstanbul 34537
Email: arifbalik@outlook.com

June 17, 2019

Abstract

These reports covers solutions to problems and answers to questions along with optimizations asked in lab assignments. All reports includes Theory of Operation, Q&A section an optional Optimization and Appendix sections.
All the source code can be found at *github.com/arifbalik/SYSTEMS_PROG_LAB*.
This report powered by LaTeX

Lab #1

Concurrent File Access by Parent and Child Processes

This lab report include analysis of some system calls in UNIX environment based on a program which provided in System Programming course. First section answers some questions asked in the experiment paper, and second section suggests and implements some improvements and optimizations on the code.

Theory of Operation

Program first checks whether expected three argument is given or not, if not program terminates. In case of correct input from console, program opens input file and creates a new file to copy the text. Later on program calls *fork* and creates a child process. Then in both child and parent processes program reads the input file and writes into created file along with a console message based on which part of the program did that (C for child, P for parent). There is a delay between read and write operations in both processes, it causes copied file to be corrupted; for instance parent reads the character x and waits for loop to finish, in the same time child also reads the character $x+1$[i] and finishes the loop first and writes the character $x+1$[ii] in the place where x should be.

Q&A

Q1. *What is the purpose of this laboratory work?*

The purpose is to analyse the behaviour of the given program by observing and tweaking some code lines, thus getting a sense of how some system calls behave in which conditions in UNIX environment.

Q2. *What is the meaning of two parameters of the function* ***main()****?*

The first parameter **argc** stands for *Argument Count* which holds the information about how many parameter is given when program executed. The second one, **argv** stands for *Argument Value* which stores the values of given arguments during execution. The form of **argv** is called *pointer-to-pointer* which is a char pointer to another char pointer, this gives an incredible flexibility to developer, when there is a need for any kind of text processing. Simply, we can think it as two-dimensional char array.

Q3. *Suppose that you started the program using two names of files. What will be the value of the parameter* ***argc*** *in the* ***main()*** *function?*

The **argc** is always greater than 0, because the first parameter of **argv** is always the name of the executable program (unless if we use a specialised compiler). So in case of given two arguments to the program, we expect **argc** to be 3.

Q4. *What are* ***argv[0]****,* ***argv[1]*** *and* ***argv[2]****?*

As we stated earlier, **argv[0]** should be name of the program, **argv[1]** the first parameter, and **argv[2]** is the second parameter.

[i] When program calls *read* function it reads the n bytes of data and increments the cursor the latest character, therefore even when both processes store the information in their own isolated variables, they effect each other via global cursor value

[ii] 1 is an arbitrary value it can be much more than that

Q5. *What is the purpose of system calls* ***open()*** *and* ***creat()****? What is the meaning of the parameters* ***O_RDONLY*** *and* ***0666*** *in these system calls?*

The function **open()** takes mainly two parameter, first is a path name and second is flags. It opens a file and returns a number associated with opened file (*File Descriptor*) and flags determines whether it is a read only or read and write or write only etc. file. It returns -1 if anything goes wrong, and sets the global error flag for developer to further debug the problem.

The function **creat()**[iii] as name calls creates a file and return a file descriptor just like **open()** function. If file already exists, function opens the file and rewrites it. It has two parameter, which first one is path name and second one is a parameter called *mode*. It defines permissions for the file to be created.

But as said earlier **open()** function takes *mainly* two parameters, but it has one more optional parameter which is the parameter also called *mode*. Function behaves just as same as **creat()** function when the parameter flag is set as following code. The permissions can be defined via third parameter.

```
open(path, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

O_RDONLY is a constant integer, specifies that the file is opened for only reading and **0666** specifies the permissions of created file. 6 gives the permissions read, write, execute, and every digit stands for different users of system, such as owner, others.

Q6. *What system call is used for the creation of a new process in UNIX?*

fork(). Also there is **exec()** but it replaces the existing process with another one.

Q7. *What is the meaning of the variable* ***pid*** *in the program? Has this variable the same value for two processes in the program? Explain your answer in more detail.*

When we call **fork()** program duplicates itself and in order to make those two programs to do different things, developer needs to know which one is which. Therefore fork returns 0 when a child process created, but we can only see that value in the child process, and in the parent process the value of pid must be different than one in the child process and indeed parent process's pid value is a positive number.

Q8. What is the purpose of statement ***break*** *in two cycles in the in the program?*

The **read()** function returns the number of bytes read from the file which is given by developer with parameter *n_byte*. It returns *-1* if something happens during the process and sets the error flag. Therefore the program reads one byte and checks whether function read it or not, if not program breaks the loop and exits if it's in the child process or waits for child process to exit if it is in the parent process.

[iii] Interesting fact : *creat* is a typo made by Ken Thompson.

But program can't tell whether there was a problem while reading or program reached the end of the file, because read function returns *0* when it reaches the end of the file and program does not check *0* or *-1* condition, it treats same for both values. This problem will be fixed later in this report.

Q9. *Suppose that there is no system call* ***exit()*** *in the child portion of the program. What could be the result?*

Program would continue to execute following codes after the if statement, and since there is no lines of code to be executed, it would stop.

We can observe this by just listing currently running processes with **ps** command in the command line.

Q10. *Suppose that there is no system call* ***wait()*** *in the parent portion of the program. What could be the result?*

Same as *Q9*, the parent will exit after reaching the end of the code which is instant since there is also no lines of code after **wait()**. If parent finishes its execution and terminate itself before child process, child becomes an *orphan* and later will be *adopted* by **init()**. So everything should be as same for this specific program.

Q12. *What is the role of cycle "****for****" between read and write operations in both processes? What will be the result without this cycle (in both processes) when the copied file is small?*

The *for* loop in the program is just a large loop with an empty body. Since every cycle of the loop takes some execution time, it allows developer to delay the code between executions. This is a very primitive way for a program to delay since we can not exactly say how much it will take to finish it because UNIX is not a real-time operating system. But, then again it is what suppose to happen for this particular case. With small time nuances in the program, it will take different times between *read* and *write* operations both for child and parent processes and that will create a corrupted file which is the subject of this experiment.

So in the absence of those loops the program would have less uncertainty and less corruption, and also the size of the text is important because when size gets large it is more probable for collisions of processes to happen.

Q12. *Do both processes execute the same statement? If not, show which statements are executed by the parent and child processes.*

Practically they execute the same thing, but they don't. Parent starts as normal program from top to bottom, until function *fork* called. After fork, the program would continue with an identical copy of itself from the point the *fork* function called and then

the child will enter the *if* statement (line 32-41) and parent will enter the *else* statement.

Q13. *In two cycles "**for**" the same index variable **i** is used. Since the parent and child processes ron concurrently, is it possible that any of these two processes will use the value of **i** modified by the other process? Explain why yes/no.*

When a child process created, all of the parent process's variables will be copied to a new address space, so they will be independent even if they have the same name, their addresses will be different, therefore, they can not interfere. This can be proven by changing the loop in the parent section with the following code and waiting the child process to exit before parent goes into the loop, if they would have the same address for the variable ***i***, there would be no way for the parent to go into the loop because the value of the variable would be changed by the child process, and condition *i [less than] 50000* would be false.

```
for(; i < 50000; i++);
```

Optimization

Firstly, the definition of the main function was out-of-date, so it has been changed;

Listing 1: Original

```
main(argc , argv)
int argc;
char* argv[];

{
```

Listing 2: Optimised

```
int main(int argc, char** argv){
```

Some variables were unnecessary, for instance *open()* and *creat()* functions return something called *file descriptor* which will be always 3 at first (0, 1 and 2 are reserved) and 4, 5 and so on. Therefore for this program they are pretty much constants since there is no dynamic creation or opening a file, it is only done twice and never again. Thus the variables *fdrd* and *fdwt* were changed with constants 3 and 4 respectively.

Listing 3: Original

```
int fdrd, fdwt;
char parent = 'P';
char child = 'C';
```

Listing 4: Optimised

```
#define __FDRD 3
#define __FDWT 4
#define P 'P'
#define C 'C'
```

Unnecessary but (Crucial for embedded software design) the variable *pid* can be defined as a *char* (or *byte*) object since its value will be small.

Personally I prefer *while* loops instead of *for* loops by means of readability, and try to use backward looping when possible;

Listing 5: Original

```
for(;;)
{
   ...
   for(i = 0; i < 50000; i++);
   ...
   ...

}
```

Listing 6: Optimised

```
while(1){
   ...
   while(--i);
   i = WAIT;
   ...
   ...
}
```

where WAIT is a constant.

Developers often use preprocessors to switch between modes. This feature can be implemented for this program's debug feature by using *#ifdef* preprocessor wherever an information printed for user to see. For example;

Listing 7: Original

```
printf("Some Text");
```

Listing 8: Optimised

```
#ifdef DEBUG
   printf("Some Text");
#endif
```

This will increase the overall performance. Developer just needs to comment the line *#define DEBUG* in the code.

There might be endless discussions about how optimised and efficient this simple program can be, for example using *write* instead of *printf* or using *open* instead of *creat* etc.

There is more optimisation made on this program, which can be seen in the Appendix.

Appendix

```
/*
 * System Programming Course
 * Lab #1
 * Concurrent File Acces by Parent and Child Processes
 * Arif Ahmet Balik - 180303019
 * Text Editor : Sublime Text 3
 * Last Update : 7 March 2019
 * All the files can be found at github.com/arifbalik/SYSTEMS_PROG_LAB
*/

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <unistd.h>

#define _FDRD 3
#define _FDWT 4

/*
 * Uncomment to enable debug mode.
 * When disabled it saves 3000 microseconds!!!
 */
#define DEBUG
#define WAIT 10000

#define HELP "Please use the command as follows;\n" \
                        ".\\sharefile.c [input_file] [output_file]"

int main(int argc, char** argv){

        #ifdef DEBUG
                struct timeval  t1, t2;
                gettimeofday(&t1, NULL);
        #endif

        char pid, c, error;
        unsigned long i = WAIT;
```

```
if(argc > 3){
        printf("Too much parameter!" HELP "\n");
        exit(1);
}else if(argc < 3){
        printf("Not enough parameter!" HELP "\n");
        exit(1);
}
if(open(argv[1], O_RDONLY) == -1 ||
   open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0666) == -1){
        perror("Oops!");
        exit(1);
}

#ifdef DEBUG
        printf("Parent: creating child process \n");
#endif

pid = fork();
if(pid == 0){

        #ifdef DEBUG
                printf("Child process starts, id: %d\n", getpid());
        #endif
        while(1){
                error = read(_FDRD, &c, 1);
                if(error == -1){
                        perror("Oops!");
                        break;
                } else if(!error){
                        #ifdef DEBUG
                                printf("\nChild: End of the file!\n")
                        #endif
                        break;
                }
                while(--i);
                i = WAIT;
                #ifdef DEBUG
                        write(1,"\nChild have read:",17);
                        write(1, &c, 1);
                #endif
                if(write(_FDWT, &c, 1) == -1) perror("Oops!");
        }
        #ifdef DEBUG
```

```
                printf("\nExiting child process\n");
        #endif
        exit(0);
    }
    else{

        #ifdef DEBUG
                printf("Parent process starts, id: %d\n", getpid());
        #endif

        while(1){
        error = read(_FDRD, &c, 1);
        if(error == -1){
                perror("Oops!");
                break;
        } else if(!error){
                #ifdef DEBUG
                        printf("\nParent: End of the file!\n");
                #endif
                break;
        }
        while(--i);
        i = WAIT;
        #ifdef DEBUG
                write(1,"\nParent read:",13);
                write(1, &c, 1);
        #endif
        if(write(_FDWT, &c, 1) == -1) perror("Oops!");
    }

    wait(0);
    close(_FDRD);
    close(_FDWT);

    #ifdef DEBUG
        gettimeofday(&t2, NULL);
        printf ("Execution time = %f uSeconds\n",
                (double) (t2.tv_usec - t1.tv_usec));
        printf("Exiting parent process\n");
    #endif

    }
}
```

Lab #2

The Study of Processes in UNIX

This report include analysis of some system calls in UNIX environment based on a program which provided in System Programming course. First section explains some system calls, second section answers some questions asked in the experiment paper, and third section suggests and implements some improvements and optimisations on the code.

Explanation of Some System and Library Functions

Difference between system and library functions

A system call is provided by the kernel and directly handled there, and a library functions are functions within programs, usually they are in the same level as shells.

write()

write is a system call used to output data to given channel (terminal, file etc.)

It has the prototype;

```
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

where *fildes* is the channel that is where to write, *buf* is which to write and *nbytes* is how many bytes to write.

fork()

The system call *fork* creates an exact copy of current program. Takes no parameters and returns *0* for *child* process and child's id for *parent* process.

getpid()

The system call *getpid* returns the process id of currently running process.

getppid()

The system call *getppid* returns the process id of currently running process's parent process.

execl()

The system call *excl* replaces the current program with given program. It has the form;

```
int execl(const char *path, const char *arg, ...);
```

where *path* is the path for new executable file and *arg* is arguments to pass to the new process. Function only returns *-1* if there is an error. By convention the first argument to pass into new program should be the name of the program and list of arguments should end with *NULL*.

sleep()

In the labsheet, instructor specified the function *sleep* as a library function, but in most of places it reffered as a system function.[iv]

[iv] http://www.cs.miami.edu/home/geoff/Courses/CSC521-04F/Content/UNIXProgramming/UNIXSystemCalls.shtml

But then again looking at the **unistd.h** where sleep defined, it can bee seen that it uses a well documented system call **nanosleep**.

```
unsigned int sleep(unsigned int seconds)
{
        struct timespec ts;

        ts.tv_sec = seconds;
        ts.tv_nsec = 0;
        if(nanosleep(&ts, &ts) == 0)
                return 0;
        return ts.tv_sec;
}
```

It delays the system by given amount in seconds and returns *0* if the amount of time elapsed otherwise (in case of interruption) returns the amount of time left.

Theory of Operation

Part 1

mainprog.c runs first and immediately creates three childs and childs overlays themselves with the program **child.c** with the arguments *1*, *2* and *3* respectively. Then parent process prints the ID of childs and wait them until they exit.

In the **child.c** each child prints their pid values along with a *char*, that is 0 - 256 number created from its own pid value. Then it sleeps random amount of time (0 - 5 sec). Finally if the value that have been passed trough child via *argv* is checked whether it is an even or odd number, if it is even then child kills itself with the signal value *9*, if not it exits with the char value mentioned above.

Part 2

In the **procident.c** program first creates a child process than terminates itself after 3 seconds of sleep (thus child becomes orphan). In the child process, after 10 seconds of sleep, child overlays itself with **simple.c**. Also as mentioned in the previous lab report, the variables can not effect each other after fork unless intended.

In the **simple.c** child reports its parent (**init()**) and exits.

In the **procmemory.c** program creates a bunch of global and local variables and shows their virtual addresses along with the starting addresses of the program, that is *etext*, *edata* and *end*.

INFORMAL NOTE : Honestly I did not understand why the variables have the same addresses in both child and parent processes. I tried to change the value of i in the child process and read it both from child and parent, and as expected they yield different results, but then again their addresses are same. My expertise is on mostly embedded

systems and bare-metal C, so I can't understand the concept of Virtual Memory how matter I try. I will read more documents thou.

etext	0x100000DF4
edata	0x100001070
end	0x100003000
main	0xD4EDB90
showit	0xD4EDD30
cptr	0xD4EE068
buffer1	0xD4EE070
i	0x52712A4C
buffer2	0x527129D8

Q&A

Q1. *What is the purpose of this laboratory work?*

The purpose is to get an understanding of some UNIX system calls, dynamics between child and parent processes during switching between programs and understanding the Virtual Memory.

Q2. *How mony child processes created by the parent process in* ***mainprog.c****?*

Three child processes created.

Q3. *In* ***mainprog.c****, a few statements are executed by each created child process. Show these statements.*

```
sprintf(value, "%d", i);
execl("child", "child", value, 0);
```

Q4. *In* ***mainprog.c****, to what program is child process switched? Show and explain the corresponding statement. Does a child process return to the* ***mainprog*** *after finishing another program (explain, why yes/no)?*

Child processes switched to **child.c**.

Child does not returns to **mainprog.c** because when **execl** called, it replaces the current program with **child.c**, there is no way that child can return.[v]

Also, in the program either process kills itself or exits.

[v] Unless **mainprog.c** called again in **child.c** using **execl**

Q5. *Which statements are NOT executed by the parent process in* ***mainprog.c****?*

Statements shown in **Q3.**

Q6. *What is the purpose of system call* ***wait()*** *in* ***mainprog.c****? What results are extracted by the parent process from this system call?*

The function **wait** puts parent process into sleep until child processes exits. It returns the ID of child process who terminated and fills the variable which given as a parameter with the exit status given by the child when terminating.

Q7. *Is it possible for a child process to continue its work after the parent process terminates? What will the parent be for such a child? Prove your answer based on the results of the Part 2 of this lab work.*

Yes. When parent terminates before child process, then that process called *orphan*, and **init** function takes over the role of parent which has a pid of 1.

As we can see from the output of the program **procident.c**, parent terminates before the child, and child still runs and outputs its parent process id which is 1.

```
Child: my ID = 5219, i = 2
Child: my parent ID = 5218
Parent: my ID = 5218, i = 0
Parent terminating...
Arif-MacBook-Pro:lab2 celimless$
 Child after sleping: my ID = 5219
 Child after sleping: my parent ID = 1
NEW PROGRAM simple IS STARTED BY TH CHILD PROCESS
Child: my ID = 5219
Child: my parent ID = 1
Child: terminating...
```

Q8. *What is the purpose of directive* ***#define*** *in the program* ***procmemory.c****?* **#define** is a preprocessor, it has no use as a compiled program but it gives developer the ability to create more modular and readable code. In the **procmemory.c** compiler simply pastes the contents wherever it sees the definition **SHW_ADR**. Also the definition behaves as a function but has no effect on program as such. For example, if developer calls this definition *SHW_ADR("x",x)*, compiler changes that line as follows;

```
printf("ID %s \t is at virtual address: %8X\n", "x", &x)
```

Q9. *What is the meaning of the variables* ***etext****,* ***edata*** *and* ***end*** *in* ***procmemory.c****?*

*Why are these variables declared with the word **extern**?*

The variables **etext**, **edata** and **end** represents segments of the program, such as variable environment, text segment etc. They are provided by linker and dangerous to use as stated by Linux man page. For instance macOS does not allow to access does variables directly, it rather provides some functions which returns the values of those variables. **extern** keyword gives variable the ability to extend itself throughout the source files. But it only declares the variable, not defines it. In order to use the variable, it has to be defined somewhere in the source program, or in a library.

***Q10.** Suppose that a variable i was declared and assigned some value in the parent process before the creation of a child process. Will this variable be accessible to the child process? Will the parent process see the change made in this variable by the child process?*

Every variable is copied and transferred to new locations with their last values in the process of fork. So the child could access to the variable but it would be isolated from parent's variable even thou they are in the same source file, have the same name, and same definition etc.

***Q11.** Is **malloc()** a system call or a library function? Can you guess it looking at your program?*

Knowing whether is a function a system call or not is not possible by looking at the source file but few comments can be made. **malloc** used to allocate memory, so it has to deal with low level sources but as in **printf** - **write** relationship, **malloc** may use some other system calls to accomplish the task of allocation. According to the *Linux Programmer's Manual* **malloc** is not a system call.[vi]

***Q12.** What is the meaning of two parameters of the function **main()** in the program **child.c**? What is **argv[1]** in this program?*

The first parameter **argc** gives the number of arguments given as an input when executable file started, and second one gives the values of those parameters. **argv**[**1**] is the number of the child which is the variable *i* in the **mainprog.c**

***Q13.** What is the purpose of system calls **getpid** and **getppid**?*

getpid returns the value of current process and **getppid** returns the value of parent process of the current process. It is usually used to create unique file names, but can be used to make child and parent processes to do different things in the same source file.

[vi] http://man7.org/linux/man-pages/man2/syscalls.2.html

Q14. *Is it possible for a process to use more than one program?*

No. A process can run one program at a time. But it can use multithreading which is not same as running multiple programs. It can also trigger other programs to run them, but then again it does not mean the process runs that program.

Q15. *Is it possible for two or more processes to use the same program?*

Yes but by saying *same* it should be noted that two processes will use same source code and will run two identical programs at the same time separately.

Q16. *What is the purpose of the system calls of the* ***exec*** *family? Does this system call return any result (in case it succeeds or fails)?*

exec system calls used to alter the current program and replace it and naturally they have no return values unless there is an error, which will return -1.

Optimization

Programs are well written, not much to be done. In the **mainprog.c** there is an unnecessary use of if and while condition.

Listing 9: Original

```
while((w = wait(&status)) &&
                    w != -1){
        if(w != -1) ...
}
```

Listing 10: Optimised

```
while(!(w = wait(&status))){
    ...
}
```

Because return of the function **wait** will be always positive when child successfully terminates there is no need to use *!= -1*. Also the condition of *w != -1* is checked three times, two inside the parenthesis of while loop and one inside the loop.

Lab #3

Advanced Study of Processes and Files in UNIX

This report include advance analysis of some system calls and files in UNIX environment based on programs which provided in System Programming course. First section explains theory of operation of Part 2 of lab ssheet , second section answers some questions asked in the experiment paper, and third section suggests and implements some improvements and optimisations on the code.

Theory of Operation

Part 1

Part 1 is already given in UML format.

Part 2

The program in part to starts with creating a child process to replace itself with the **echo** program along with a text message. At the same time parent process writes the current date by calling **date** command via **execl**

Q&A

Part 1

Q1. *Why has the child process the ability to copy file 1 that is opened by the parent process?*

Unix has a file description structure which keeps the current position in the file, path of the file, file descriptor etc. When the function **fork** called, it copies this structure to child, thus child can access the files those opened before the **fork**

Q2. *Explain why files 2 and 3 can have different size and contents (after copying)?*

Because of the amount of bytes are different that child and parent reads from the file.

Q3. *Which of the processes will terminate first?*

It depends on which process read the last byte. It is highly probable the child process finishes first because it reads more data but first they work on the same file so size does not matter too much, and parent always waits for the child to terminate.

Q4. *Assume that the block "Waiting for the child" in the parent part of the program put immediately after the block "Create new file 2". What will the result be?*

File 2 would be empty because parent would wait for the child to terminate, hence child wont terminate until it reads all the file. After termination of child, parent would try to read and get *END OF FILE* from **read** function.

Q5. *What is the effect of large value of N?*

Because of the uncertainty mentioned in the last lab sheets, it can cause the sizes of the *file 2* and *file 3* to be too different from each other.

Q6. *What is the effect of the small value N?*

Vice versa.

Part 2

Q1. *What is the purpose of this laboratory work?*

The purpose is to gain a complete understanding the mechanism of processes and multithreading behaviours of the programs during concurrent file access.

Q2. *What is the purpose of system calls of* ***exec*** *family?*

exec family of functions replaces the current program with a new program. It allows developer to switch between programs during execution.

Q3. *What is the purpose of system calls* ***wait*** *and* ***sleep****? Is there any difference between them?*

As the name calls, the function **wait**, keeps the program locked until a signal is given, which is the child process's exit function. The function **sleep** just locks the system for given amount of time in seconds.

Q4. *What is the difference between the functions* ***printf*** *and* ***perror****?*

perror prints a custom message along with error messages by looking at error flags. **printf** (print formatted) prints a text with a given format, printf has nothing to do with errors.

Q5. *What is the difference between system calls* ***exec*** *and* ***system****?*

exec overlays the program, replaces it with a program given as a parameter, **system** passes the given argument to the shell. **system** basically creates a child and calls **exec** functions.

procexec/system.c - The Linux Programming Interface (Listing 27-9, page 586)

```
switch (childPid = fork()) {
    case -1: /* fork() failed */
        status = -1;
        break;    /* Carry on to reset signal attributes */

    case 0: /* Child: exec command */
```

Q6. *Suppose that the child process in the program at the step 3 terminated before*

the parent. What will be the state of the child and how long this state will exist?

In such case the child can not terminate and called zombie and it will remain until the parent process terminates.

Q7. *Is it possible for a parent process in the program at the step 3 to terminate before child process? What will be the new parent for the child process.*

Such a child is called **orphan**, and **init** takes control over the process and becomes its new parent.

Q8. *Why the order of printing messages by the processes in step 3 is not always correct?*

Because parent does not wait for child to terminate to print its message, in some cases that causes parent to finish first and print the message before child.

Optimization

Part 2

It defeats the purpose of the program but, it is very pleasing to optimise entire program down to a single line of code.

```
system("echo Todays date is && date '+Date = %D Time = %H:%M'");
```

Appendix

Part 1 Source Code

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

#define HELP "./part1 [file1] [file2] [file3]"

#define OPEN 1
#define CREATE 0
#define OPEN_FILE(path, o_c, fd)        fd = ((o_c) ? open(path, O_RDONLY) :\
                                                open(path, O_CREAT|O_WRONLY|O_TRUNC, 0666));\
                                        if(fd == -1){\
                                                perror("Error while file creation/opening");\
                                                exit(1);\
                                        }



#define PARENT_READ_BLOCK 10
#define CHILD_READ_BLOCK 20

#define WAIT 1

int read_write(int fd_read, int fd_write, int size){
        char* block;
        int i;
        short bytes;

        block = (char*)malloc((char)size);

        bytes = read(fd_read, block, size);
        switch(bytes){
                case 0:
                        printf("%s File 1: EOF\n",
                                (size == PARENT_READ_BLOCK) ? "Parent" : "Child");
                        goto eof;
                break;
                case -1:
                        perror("Error during read from File 1");
                        goto error;
                break;
                default:
                        if(bytes == size){
                                i = WAIT;
                                while(i--);
                                if(write(fd_write, block, size) != size){
                                perror("Error during write into File 3");
                                goto error;
                                }
                                return 1; //ready for the next sequence
                        }else if(bytes > 0){
                                write(fd_write, block, bytes);
                                printf("%s File1: EOF with %d bytes\n",
                                (size == PARENT_READ_BLOCK) ? "Parent" : "Child", bytes);
                        }else{
                                printf("Error during read File 1. bytes: %d\n",bytes);
                                goto error;
                        }
                        goto eof;
                break;
        }

        error:
        free(block);
        exit(1);

        eof:
        free(block);
        return 0;
}

int main(int argc, char** argv)
{
        short fd1, fd2, fd3;

        if(argc != 4){
                printf("Invalid argument count. Help\n"HELP"\n" );
                exit(1);
        }

        OPEN_FILE(argv[1], OPEN, fd1);

        if(!fork()){
                OPEN_FILE(argv[3], CREATE, fd3);
                while(read_write(fd1, fd3, CHILD_READ_BLOCK));
                close(fd3);
                exit(1);
        }

        OPEN_FILE(argv[2], CREATE, fd2);

        while(read_write(fd1, fd2, PARENT_READ_BLOCK));

        close(fd2);
        wait(0);
        close(fd3);

}
```

Lab #4

Understanding Threads in a UNIX System

This report include analysis of threads in UNIX environment based on programs which provided in System Programming course. First section explains theory of operation, second section answers some questions asked in the experiment paper, and third section suggests and implements some improvements and optimisations on the code.

Theory of Operation

Single Thread Application

For the first part, there is a single threaded application which does nothing but running a loop, calling two functions and calculating the amount of time it took to execute the program. Program executes in a very standart sequence as expected.

Following figure shows the timeline of the program. First column shows the time which functions occurred and the other columns show which one occurred. The number in square brackets is the value of the variable **r1**

Timeline (singlethread.c)

0 sec	main [0]		
3 sec	main [0]		
6 sec	main [0]		
9 sec		func1 [0]	
13 sec		func1 [0]	
17 sec		func1 [0]	
21 sec		func1 [0]	
25 sec		func1 [0]	
29 sec		func1 [0]	
33 sec		func1 [0]	
37 sec		func1 [0]	
41 sec		func1 [0]	
45 sec			func2 [0]
48 sec			func2 [0]
51 sec			func2 [0]
54 sec			func2 [0]
57 sec			func2 [0]
60 sec			func2 [0]
63 sec			func2 [0]

As seen in the output of the program, **main** thread runs 3 times with 3 second intervals, **func1** runs 9 times with 4 second intervals and **func2** runs 7 times with 3 second intervals.

Multi Thread Application

In **multithread.c**, program creates two threads as **func1** and **func2** respectively. Then it runs just as same as single threaded application. But observe the output of the program;

First as they execute simultaneously, their occurrence is somewhat random, it can be observed by running the program several times. And because they work at the same time, the total time of execution is half the size of single thread application. Also the

reason of the randomness in the occurrence is because the delay time of each functions is different.

There is no **race condition** (race condition will be discussed in the **Q&A** section in further detail) because **mutex** is implemented, but then again this program can hardly create a race condition without mutex, since the occurrence of threads is relatively separate.

Timeline (multithread.c)

0 sec	main [101]		
0 sec			func2 [101]
0 sec		func1 [101]	
3 sec			func2 [101]
3 sec	main [102]		
4 sec		func1 [102]	
6 sec	main [103]		
6 sec			func2 [103]
8 sec		func1 [103]	
9 sec			func2 [103]
12 sec		func1 [103]	
12 sec			func2 [103]
15 sec			func2 [103]
16 sec		func1 [103]	
18 sec			func2 [103]
20 sec		func1 [103]	
24 sec		func1 [103]	
28 sec		func1 [103]	
32 sec		func1 [103]	

The program has been tested without waiting threads to terminate. The expected result should be that after **main** function terminates, all the threads will be lost and total execution time is 9 seconds since the loop in the main function runs 3 times wirh 3 second delay. The following output shows the results.

Timeline (multithread.c) without wait

0 sec	main [101]		
0 sec			func2 [101]
0 sec		func1 [101]	
3 sec			func2 [102]
3 sec	main [102]		
4 sec		func1 [102]	
6 sec			func2 [102]
6 sec	main [103]		
8 sec		func1 [103]	
9 sec			func2 [103]

Indeed, results proves the assumption.

Multi Process Application

In **multiprocess.c** the only difference is creation of threads is done by processes. But outcomes are not even close.

Following output shows the results of running the program.

Timeline (multiprocess.c)

0 sec	main [101]		
0 sec		func1 [0]	
0 sec			func2 [0]
3 sec			func2 [0]
3 sec	main [102]		
4 sec		func1 [0]	
6 sec			func2 [0]
6 sec	main [103]		
8 sec		func1 [0]	
9 sec			func2 [0]
12 sec		func1 [0]	
12 sec			func2 [0]
15 sec			func2 [0]
16 sec		func1 [0]	
18 sec			func2 [0]
20 sec		func1 [0]	
24 sec		func1 [0]	
28 sec		func1 [0]	
32 sec		func1 [0]	

First noticeable thing is that the values for **r1** is not the same with **main**. That is because processes copied the variables to new spaces so their addresses are no longer the same with **main**.

And the other interesting behaviour is when **main** function terminates without waiting anything to terminate, the threads continues their execution. Again that is because processes created their own environment which is isolated enough for threads to continue. So we would get the same output as above.[vii]

Q&A

Q1. *What is a program thread?*

Threads are some kind of process, **dependently**[viii] and concurrently running within the parent process.

> To Linux, a thread is just a special kind of process. (Linux Kernel Development, 2011, p. 29)

Q2. *What is the purpose of using threads in programs?*

As seen in the section **Theory of Operation**, it reduces the amount of time for a program to do a certain job and increases performance.

Q3. *Is there any difference between processes and threads (explain)?*

Yes. Threads are sharing the same address space as parent process, and threads naturally can only work locally.

Q4. *When is the real parallelism of thread execution is possible?*

In fact, there is no such thing, but when kernel decides to run the thread in a different **physical** core, then thread has the possibility to achieve parallel computation. Otherwise kernel just schedules the codes in an order to simulate parallelism.

Q5. *What thread standard is implemented in UNIX Systems?*

POSIX. Hence the name **p** in the first character of functions related to threading.

Q6. *Are threads available in Windows operating systems?*

Yes. By definition, an operating system should be able to run multiple programs simultaneously, this includes threads as well.

[vii] that is, when childs wait for the threads to terminate to terminate themselves
[viii] by means of memory

Q7. *How many threads are there in a program when it just starts (that is, at the very beginning)?*

There is only one called *main* thread.

Q8. *Is it possible to change the number of threads in a program during its execution (explain)?*

Yes. For threads to have meaning program has to have the ability to create and terminate threads. In UNIX this can be achieved with the function **pthread_create**.

Kernel hold all the information of threads and processes in a structure called **task_struct** which is located in the kernel **include/linux/sched.h**, this allows kernel to dynamically add and remove threads.

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup
         * (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;
```

Q9. *How can a new thread be created in a program?*

Using **pthread_create** in the standart library **pthread.h**.

Q10. *Assume that the main (primary) thread in a program have created two new (secondary) threads and wants to continue its work only after these two new threads finish their work. Which function call should be used by the main thread for this purpose?*

pthread_join

Q11. *How will a thread, after having been created by the main function, learn what routine (function) it should use to perform its work? Is this routine a part of the main function or it should be specified outside the main function?*

It is given by the main program as a pointer to function during its creation via the first argument of the function **pthread_create**. It can locate anywhere as long as it is compiled with the main program.

Q12. *Is it possible for two or more threads to use the same routine (function)?*

Yes, because kernel calls the function from different sources, but it can be dangerous as they try to access same variable.

Q13. *Suppose that main thread, after the creation of a number of new threads, terminates. What will be with the created secondary running threads in this case?*

It will terminate since it has no environment to exist after main terminates.

Q14. *Is it possible to run a multithreaded program on a multiprocessor or a few computers connected to a network?*

Threads meant to work on multiprocessor, they are more efficient that way but they can't work on different machines.

Q15. *Suppose that two or more threads use (read and/or write) the same global data defined in the program. What can happen if you don't undertake same precautions? What should these precautions be?*

When multiple threads tries to access and change a variable, something called **race condition** occurs. That may create some unpredictable results. For example let one thread have an *if* condition that checks value of a variable and inside of the *if* condition it changes value of that variable, but at the same time, when the thread checks the variable and goes into the *if* condition, some other thread may change the value of that variable and cause critical damage.

To prevent such conditions UNIX offers a method called **thread locking** and it can be enabled by using a set of functions, such as **pthread_mutex_lock** which locks the access to resources in the region between **pthread_mutex_lock** and **pthread_mutex_unlock**.

For example in the **multithread.c** the variable **r1** can be safe from race conditions with the following code;

```
pthread_mutex_lock(&lock);
r1 = 100 + j;
pthread_mutex_unlock(&lock);
```

Q16. *Is programming with threads easier than programming without threads (explain)?*

It depends. For developer, it is easier to think and act safely in a single threaded program. But in real world a lot of applications requires multithreading, for example, it would be almost impossible to stream data from ethernet and drive an LCD display smoothly with a single threaded application because both of those peripherals requires

complex and long protocols. So, how hard it is to work with threads becomes more and more irrelevant as they become unavoidable with the needs of modern technology.

Optimization

By giving up the microsecond resolution, time calculation can rid of heavy arithmetic operations.

Original

```
(float)((1000000*time2.tv_sec + time2.tv_usec) -
(1000000*time1.tv_sec + time1.tv_usec))/1000000
```

Optimised

```
time2.tv_sec - time1.tv_sec
```

There are other optimisations. Check Appendix for more.

Appendix

singlethread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock; //we need this for func1 and func2 to work

int main(int argc, char **argv){
        char* text = (char *)malloc(20);
        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        for (int j = 1; j < 4; ++j){
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif

                sleep(3);
        }


        func1();
        func2();
        gettimeofday(&time2, 0);
        printf("Main: Total elapsed time = %ld seconds\n", time2.tv_sec - time1.tv_sec);
        printf("Main thread terminated...\n");

        free(text);
        return 0;
}
```

multithread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock;

int main(int argc, char **argv){
        pthread_t td1, td2;
        int p;
        char* text = (char *)malloc(20);

        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        p = pthread_create(&td1, NULL, (void *)func1, NULL);

        if(p != 0){
                perror("Thread 1 creation problem");
                exit(1);
        }

        p = pthread_create(&td2, NULL, (void *)func2, NULL);

        if(p != 0){
                perror("Thread 2 creation problem");
                exit(1);
        }

        for (int j = 1; j < 4; ++j){
                pthread_mutex_lock(&lock);
                        r1 = 100 + j;
                pthread_mutex_unlock(&lock);
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif
                sleep(3);
        }

        //pthread_join(td1, NULL);
        //pthread_join(td2, NULL);

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        delta = (time2.tv_sec - time1.tv_sec);
        pthread_mutex_unlock(&lock);
        printf("Main: Total elapsed time = %f seconds\n", delta);
        printf("Main thread terminated...\n");

        return 0;
}
```

multiprocess.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <pthread.h>
#include "timing.h"

void func1(void);
void func2(void);


int r1 = 0, r2 = 0;
struct timeval time1, time2;
pthread_mutex_t lock;

int main(int argc, char **argv){
        pthread_t td1, td2;
        int pid, c_pid;
        int p;
        float delta;
        char* text = (char *)malloc(20);

        #ifdef DRAW_TIMING_DIAGRAM
                system("clear");
        #else
                printf("Single threaded process starts here...\n");
        #endif

        gettimeofday(&time1, 0);

        pid = fork();

        if(pid < 0){
                perror("Error during child1 creation");
                exit(1);
        }
        if(pid == 0){
                p = pthread_create(&td1, NULL, (void *)func1, NULL);

                if(p != 0){
                        perror("Thread 1 creation problem");
                        exit(1);
                }
                pthread_join(td1, NULL);
                exit(1);
        }

        pid = fork();

        if(pid < 0){
                perror("Error during child2 creation");
                exit(1);
        }
        if(pid == 0){
                p = pthread_create(&td2, NULL, (void *)func2, NULL);

                if(p != 0){
                        perror("Thread 2 creation problem");
                        exit(1);
                }
                pthread_join(td2, NULL);
                exit(1);
        }

        for (int j = 1; j < 4; ++j){
                pthread_mutex_lock(&lock);
                        r1 = 100 + j;
                pthread_mutex_unlock(&lock);
                #ifdef DRAW_TIMING_DIAGRAM
                        sprintf(text, "\t|main[%d]\t|\t\t|\n",r1);
                        time_line(text);
                #else
                        printf("Main function of the process works: %d\n", j);
                #endif
                sleep(3);
        }

        while ((c_pid = wait(0)) > 0);

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        delta = (time2.tv_sec - time1.tv_sec);
        pthread_mutex_unlock(&lock);
        printf("Main: Total elapsed time = %f seconds\n", delta);
        printf("Main thread terminated...\n");

        return 0;
}
```

func1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "timing.h"

void func1(void){
        int i;
        char* text = (char *)malloc(20);
        for (int i = 1; i < 10; ++i){
                #ifdef DRAW_TIMING_DIAGRAM
                        pthread_mutex_lock(&lock);
                        sprintf(text, "\t|\t\t|func1[%d]\t|\n",r1);
                        pthread_mutex_unlock(&lock);

                        time_line(text);
                #else // Original
                        printf("Function func1 prints and then sleeps 4 s: %d\n", i);
                #endif
                sleep(4);
        }
        free(text);
        return;
}
```

func2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include "timing.h"

void func2(void){
        int i;
        char* text = (char *)malloc(20);;
        for (int i = 1; i < 8; ++i){
                #ifdef DRAW_TIMING_DIAGRAM
                        pthread_mutex_lock(&lock);
                        sprintf(text, "\t|\t\t|\t\t|func2[%d]\n",r1);
                        pthread_mutex_unlock(&lock);
                        time_line(text);
                #else // Original
                        printf("Function func2 prints and then sleeps 3 s: %d\n", i);
                #endif
                sleep(3);
        }
        free(text);
        return;
}
```

timing.h

```
#define DRAW_TIMING_DIAGRAM
extern struct timeval time1, time2;
extern int r1;
extern pthread_mutex_t lock;

static void time_line(const char *text){

        pthread_mutex_lock(&lock);
        gettimeofday(&time2, 0);
        printf("|%ldsec", (time2.tv_sec - time1.tv_sec));
        printf("%s",text);
        pthread_mutex_unlock(&lock);
}
```

Lab #5

Programming Operations on Files and Directories in UNIX

This reports covers some properties of system calls which relate to directory and file operations on UNIX environment. It explains its theory of operation and answers some questions asked in the lab sheet.

Theory of Operation

The programs waits for a command line argument to work, that is path to be examined, if not, program returns an error and exits. After, programs immidiatly gathers information about the path, and checks whether path is a directory or file. Then if it is a file than program prints its information and exits, if it is a directory than it tries to print every file and directory in that directory along with their sizes, finally it closes file and exits.

Q&A

Q1. *What is the purpose of this lab work?*

The purpose of this laboratory work is to understand basic file system operations such as reading names and sizes of files and directories.

Q2. *What is the purpose of the system call stat()? What arguments are used in this system call?*

stat() system call is used to gather information about the file or directory.
The prototype of this function is as follows;

```
int stat(const char *path, struct stat *buf);
```

The first argument is path (directory or file) which function will gather information for, and the second one is the structure that function will fill the information with.

Q3. *What are main fields in the structure stat? In what header file is this structure defined?*

Based on *The Open Group Base Specifications Issue 6* structure is as follows;

```
dev_t     st_dev     ID of device containing file
ino_t     st_ino     file serial number
mode_t    st_mode    mode of file (see below)
nlink_t   st_nlink   number of links to the file
uid_t     st_uid     user ID of file
gid_t     st_gid     group ID of file
dev_t     st_rdev    device ID
                        (if file is character or block special)
off_t     st_size    file size in bytes
                        (if file is a regular file)
time_t    st_atime   time of last access
time_t    st_mtime   time of last data modification
time_t    st_ctime   time of last status change
blksize_t st_blksize a filesystem-specific preferred I/O block
                size for this object.  In some filesystem types,
                this may vary from file to file
blkcnt_t  st_blocks  number of blocks allocated for this object
```

it contains various information about file or directory such as size, time of last access, mode of file etc.

This structure can be accessed through *sys/stat.h*

Q4. *How can you learn that an item is a regular file or a directory?*

After calling stat system call, developer can mask the *st_mode* and *S_IFMT* with *&* operator and after this mask operation developer can check various options. All options are listed below;

```
S_IFBLK
Block special.
S_IFCHR
Character special.
S_IFIFO
FIFO special.
S_IFREG
Regular.
S_IFDIR
Directory.
S_IFLNK
Symbolic link.
S_IFSOCK
Socket.
```

Q5. *What is the purpose of the constants S_IFREG and S_IFDIR? In which header file these constants are defined?*

Those constant are *regular file* and *direction* respectively. It is used to determine the type of the path.

Those constants can be accessed through *sys/stat.h*

Q6 *What function can be used for opening a directory? What is the returned result of this function when the function call fails?*

opendir can be used to open a directory. When function fails it returns *NULL*[ix] and sets the *errno*.

Q7. *How can you learn the size of a file in a directory?*

Size and many other features of a file or directory can be learned by using structure *stat*. For size information developer can look at *st_size* in *stat*.

Optimization

As asked in the lab sheet, owner of the file and searching all the sub directories will be implemented.

Owner of the file can be found in structure *stat*, the field is *st_uid*. Following code prints the information of the users for a given path.

```
struct passwd *ps = getpwuid(info.st_uid);
 printf("Owner: %s\n", ps->pw_name);
```

And following function can recursively searches for sub directories;

[ix] In Linux/UNIX a system call returning *NULL* or a positive number when something goes wrong is too rare.

```
void list_dir(char *base_path, const int root)
{
    int i;
    char path[1000];
    struct dirent *dp;
    struct stat info;
    DIR *dir = opendir(base_path);
    char cur_path[BUFSIZ + 1];
    struct passwd *ps;

    if (!dir){
        return;
    }


    while ((dp = readdir(dir)) != NULL){

        if (strcmp(dp->d_name, ".") != 0 &&
                        strcmp(dp->d_name, "..") != 0){
            for (i=0; i<root; i++) {
                if (i % 2 == 0 || i == 0)
                    printf("|");
                else
                    printf(" ");

            }
        sprintf(cur_path, "%s/%s",base_path, dp->d_name);
            if(stat(cur_path, &info) < 0) return;
        ps = getpwuid(info.st_uid);
            printf("|--%s, Size: %ld, Owner: %s\n",
                dp->d_name, info.st_size, ps->pw_name) ;

            strcpy(path, base_path);
            strcat(path, "/");
            strcat(path, dp->d_name);
            list_dir(path, root + 2);
        usleep(100000); // to make it look coooool!
        }
    }

    closedir(dir);
}
```

Full source code is given in the next section.

Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/dir.h>
#include <string.h>
#include <pwd.h>


void  list_dir(char *base_path, const int root);

int main(int argc, char **argv)
{


    DIR *dp;
    struct dirent *dir;
    struct stat info;
    char pathname[BUFSIZ+1];
    struct passwd *ps;

    if(argc != 2){
        printf("Please give a path. \n
                        Example: ./[name] [path]\n");
        exit(1);
    }

    if(stat(argv[1], &info) < 0) exit(1);

    if((info.st_mode & __S_IFMT) != __S_IFREG &&
       (info.st_mode & __S_IFMT) != __S_IFDIR ){
        printf("%s is not a directory or file\n", argv[1]);
        exit(1);
    }

    switch((info.st_mode & __S_IFMT)){
        case __S_IFREG:
            ps = getpwuid(info.st_uid);
            printf("File name : %s, Size : %ld, Owner: %s\n",
                 argv[1], info.st_size, ps->pw_name);
            exit(1);
        break;
        case __S_IFDIR:

             list_dir(argv[1], 0);
        break;
    }

    closedir(dp);

    return 0;
}

void list_dir(char *base_path, const int root)
{
    int i;
    char path[1000];
    struct dirent *dp;
    struct stat info;
    DIR *dir = opendir(base_path);
    char cur_path[BUFSIZ + 1];
    struct passwd *ps;

    if (!dir){
        return;
    }


    while ((dp = readdir(dir)) != NULL){

        if (strcmp(dp->d_name, ".") != 0 && strcmp(dp->d_name, "..") != 0){
            for (i=0; i<root; i++) {
                if (i % 2 == 0 || i == 0)
                    printf("|");
                else
                    printf(" ");

            }
        sprintf(cur_path, "%s/%s",base_path, dp->d_name);
            if(stat(cur_path, &info) < 0) return;
        ps = getpwuid(info.st_uid);
            printf("|--%s, Size: %ld, Owner: %s\n", dp->d_name, info.st_size, ps->pw_name) ;

            strcpy(path, base_path);
            strcat(path, "/");
            strcat(path, dp->d_name);
            list_dir(path, root + 2);
        usleep(100000); // to make it look coooool!
        }
    }

    closedir(dir);
}
```

Lab #6

Using Shared Memory with Semaphores for IPC

This report covers some properties of IPC, shared memory and semaphores. It explains its theory of operation and answers some questions asked in the lab sheet.

Theory of Operation

There is a main process, it initializes a shared memory, two semaphores and two child process, two child process then overlays themselves with **producer.c** and **consumer.c** respectively. Then producer program after locking the semaphore, fills an array and sets current position of array in the shared memory, then unlocks the semaphore and waits for consumer program to unlock the semaphore after it prints the message. This process repeats given amount of time, then both process exits and parent program destroys both shared memory and semaphores.

Q&A

Q1. *What is the purpose of this lab work?*

To understand shared memory concept and semaphores thus IPC.

Q2. *Describe the operation of a producer-consumer system in general*

There is a shared memory for both consumer and producer programs. Both of these programs tries to access this memory concurrently, producer puts given amount of text in memory and consumer prints them, and they collaborate with semaphores to avoid race condition.

Q3. *How many processes do implement a producer-consumer system in this lab work?*

There are three processes, one is the main process and other two are producer and consumer.

Q4. *What are the roles (duties) of each of the processes in this lab work?*

The cons_prod_parent process creates producer and consumer as well as the shared memory and two semaphores. Producer attaches the shared memory and semaphores (so does the consumer) and tries to fill the shared memory with some messages, and consumer prints those messages. Finally cons_prod_parent detaches the shared memory and semaphores after waiting the termination of both processes.

Q5. *Will the producer-consumer system in this lab work, operate normally if the parent process initially creates the producer and the consumer processes, and only after that it will create shared memory and semaphores? Explain your answer.*

No. Because right after producer and consumer starts they attach already created semaphores and shared memory to themselves, and since there are no semaphores and

shared memory in present, function **semget** and **shmget** will throw error.

Q6. *Initially the parent process and its two child (the producer and the consumer) use the same executable program* ***prod_cons_parent****. How is it done that later the producer and the consumer execute different executable programs?*

After fork operation, each process immediately forced to call the function **execl**, thus each process is overlaid by programs **producer.c** and **consumer.c**.

Q7. *Suppose that the producer wants to put a message into the shared message queue when this queue is full (that is. there are no empty slots). What will the producer do in this case? Explain in detail, using the source text of the producer.*

Suppose the length of queue is 10. When producer produces the 9th message and puts it in memory it increments the tails by one and takes the reminder of the increment divided by length, so it becomes 0 and on the next turn it overrides the first message.

So in fact, message queue will never be full, since its last index will be always empty.

Following codes shows the lines for this operation.

```
strcpy(memptr->buffer[memptr->tail], local);
memptr->tail = (memptr->tail + 1) % N_SLOTS;
```

Q8. *Suppose that the consumer wants to extract a message from the shared message queue when this queue is empty (that is. there are no messages in the queue). What will the consumer do? Explain in detail. using the source text of the consumer.*

It will do nothing and wait for the producer. Because in the enum the first parameter is *AVAIL_SLOTS*. If we change it and make the first one *TO_CONSUME*, it wont wait for producer anymore and will print empty string.

Q9. *Why do we need semaphores for the producer-consumer system? What can happen if no semaphores are used in this system?*

We need semaphores because we want to control the flow of our program, otherwise we cant predict what might happen, consumer might face with empty array situation, or race condition may occur.

Q10. *What can you tell about the values of variables* ***head*** *and* ***tail*** *when the message queue is empty?*

They are both zero, since they point nothing.

Q11. *What can you tell about the values of variables* ***head*** *and* ***tail*** *when the message queue is full?*

They both reset themselves to zero as explained in *Q7.*

Q12. *Suppose that the sleep time in the producer process is always zero, but the sleep time in the consumer is not zero. That is. the producer is capable to produce messages very fast. Can it really do this in the lab work, if semaphores are used? What will the producer do producing of messages in this case? Will it be blocked or not? If yes. then in what statement? Explain your answer with the of the source text.*

It will be slightly faster since it was waiting between 0 and 6 seconds in each cycle. But it still won't be as fast as it can be because it will be blocked by the consumer. The following code line keeps the producer waiting until consumer grants access.

```
semid = semget(myparid, 2, 0);
```

Q13. *Suppose that the parent process does not remove the shared memory and the semaphore set at the end. What will happen in this case? How can you see that the shared memory and the semaphores are not removed? Can you remove them manually? How?*

For semaphores,this is OS dependent and this behavior is not specified in Linux but it may cause resource leaks. For the shared memory, if not destroyed it will stay in the memory. And both of these can be destroyed manually by using **iprcm** command, and can be listed with **ipcs** command.

Q14. *The shared memory is the fastest mechanism of IPC. Explain Why*

Because otherwise data, signal, message should be physically carried in the memory by OS. In shared memory, all a process or thread needs to do is to access the shared variable.

Q15. *Explain the main disadvantage of shared mechanism of IPC.*

Synchronization is up to the developer. Developer must be careful, otherwise program can leave semaphores, shared memories which are not destroyed behind, and can cause race conditions etc.

Q16. *Is it always necessary to use semaphores with shared memory? Explain (see Introduction to OS textbooks).*

No. Developer can implement a custom signaling algorithm, or use mutex. Semaphore is just a method to lock access to shared memory by other processes, any method that can do this is applicable.

Q17. *Is it possible to the shared memory mechanism of IPC in UNIX for communication between processes run on different computers in a network? Explain.*

Yes. Since processes can run on different computers (locally), and since abbreviation of IPC is *Inter Process Communication*, it is possible.

Optimization

None.

Appendix

None.