University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2022)

Lab#	Date	Title	Due Date	Grade Release Date
Lab09	Week 09	**Parallel Programming**	Two-Week Lab Nov. 23, 2022, Wednesday Midterm EDT	Nov. 28, 2022

The main objective of this lab will be to write a program that can have more processor's share, hence executes and finishes faster, by creating multiple child processes for doing parts of the program in parallel.

UNIX's kernel shares a processor to multiple processes based on their *niceness*. Initially, a process is given a nice number NZERO which is a non-zero number. A process could choose to run with lower priority by increasing its nice value (being more generous) and reducing its share of the processor. To raise its priority and have more share of processor, the process should decrease its niceness (being meaner). We asked you about nice value in Lec09 assignment. In this lab assignment, however, we want to practice on a better alternative. We want to create multiple processes (children) to achieve a program's task. Although the kernel still gives the same processor share to each process, our parent process including its children receives more processors share overall: 1 share for the parent and 1 share for each child. For example, a parent process with 3 children will receive 4 processor shares.

Step 1. Sequential Run

Let's write a program that does the four main math operations +, -, *, / on two input numbers:

```
hfani@bravo:~/lab09$ vi sequential.c
```

```c
#include <stdio.h>
#include <stdlib.h>
int result;
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        result = a + b;
        printf("%d + %d = %d\n", a, b, result);

        result = a - b;
        printf("%d - %d = %d\n", a, b, result);

        result = a / b;
        printf("%d / %d = %d\n", a, b, result);

        result = a * b;
        printf("%d * %d = %d\n", a, b, result);

        return 0;
}
```

The above program is a typical example of a sequential program that runs from the first line to the last run under a single process. In total, it should do 4 steps to finish.

Step 2. Parallel Run

However, a better way would be to create 4 child processes and give each one a math operation. The parent process can then wait for the children to achieve the result. If we have multiple processors (not in this course), the child processes can do the math operations in parallel. Hence, the parent process can finish after 1 step (4 operations in parallel becomes 1 step.) In this course, we have a single processor. So, there is no parallel run. But still, 4 child processes can have 4 processor shares and finishes faster.

Let's create one child process for each math operation using the fork() system call:

```
hfani@bravo:~/lab09$ vi parallel.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int result;
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        pid_t pid = fork();
        if (pid == -1){//not able to create child
                result = a + b;
                printf("parent PID %d => %d + %d = %d\n", getpid(), a, b, result);
        }
        else if (pid == 0) {//child
                result = a + b;
                printf("child01 PID: %d => %d + %d = %d\n", getpid(), a, b, result);
                exit(0);
        }
        //here pid > 0 => parent
        pid = fork();
        if (pid == -1){//not able to create child
                result = a - b;
                printf("parent PID %d => %d - %d = %d\n", getpid(), a, b, result);
        }
        else if (pid == 0) {//child
                result = a - b;
                printf("child01 PID: %d => %d - %d = %d\n", getpid(), a, b, result);
                exit(0);
        }

        //here pid > 0 => parent
        pid = fork();
        if (pid == -1){//not able to create child
                result = a / b;
                printf("parent PID %d => %d / %d = %d\n", getpid(), a, b, result);
        }
        else if (pid == 0) {//child
                result = a / b;
                printf("child01 PID: %d => %d / %d = %d\n", getpid(), a, b, result);
                exit(0);
        }

        //here pid > 0 => parent
        pid = fork();
        if (pid == -1){//not able to create child
                result = a * b;
                printf("parent PID %d => %d * %d = %d\n", getpid(), a, b, result);
        }
        else if (pid == 0) {//child
                result = a * b;
                printf("child01 PID: %d => %d * %d = %d\n", getpid(), a, b, result);
                exit(0);
        }
        wait(0);
        exit(0);
}
```

In our program, we have to check whether the child is created. Otherwise, the parent should do the task. A sample run of the above program produces:

```
hfani@bravo:~/lab09$ ./parallel 2 5
child01 PID: 3812170 => 2 + 5 = 7
child01 PID: 3812171 => 2 - 5 = -3
child01 PID: 3812172 => 2 / 5 = 0
child01 PID: 3812173 => 2 * 5 = 10
```

If we add another math operation, we have to copy-paste the same lines of code for creating a new child process. Replicating the same lines of code is not desirable (why?). Better programming would be to use a for loop that iterates based on the number of math operations and creates the child process at each iteration:

```
hfani@bravo:~/lab09$ vi parallel_forloop.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int result;
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        char math_op[4] = {'+', '-', '/', '*'};

        for(int i = 0; i < sizeof(math_op); ++i){
                pid_t pid = fork();
                if (pid == -1){//not able to create child
                        result = math_op[i] == '+' ? (a + b) :
                                 (math_op[i] == '-' ? (a - b) :
                                 (math_op[i] == '/' ? (a / b) :
                                 (math_op[i] == '*' ? (a * b) : (0))));
                        printf("parent PID %d => %d %c %d = %d\n", i, getpid(), a, math_op[i], b, result);
                }
                else if (pid == 0) {//child
                        result = math_op[i] == '+' ? (a + b) :
                                 (math_op[i] == '-' ? (a - b) :
                                 (math_op[i] == '/' ? (a / b) :
                                 (math_op[i] == '*' ? (a * b) : (0))));
                        printf("child%d PID: %d => %d %c %d = %d\n", i, getpid(), a, math_op[i], b, result);
                        exit(0);
                }
        }
        wait(0);
        exit(0);
}
```

You may wonder what **?:** operator is. This is a ternary operator (it accepts three operands) and is called the Elvis operator since it looks like Elvis Presley's hairstyle. Read more here: https://en.wikipedia.org/wiki/Elvis_operator

A sample run of the program will be the following where 4 different child processes do the math operations separately (PIDs are different):

```
hfani@bravo:~/lab09$ ./parallel_forloop 2 5
child0 PID: 3848581 => 2 + 5 = 7
child1 PID: 3848582 => 2 - 5 = -3
child2 PID: 3848583 => 2 / 5 = 0
child3 PID: 3848584 => 2 * 5 = 10
```

Step 3. Scalar Matrix Multiplication

From math, we know that we can multiply a number *a* to a matrix $A_{n\times m}$ where each element of A is multiplied by *a*, that is: $a * a_{i\times j}$. For example, $5 * \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}$ yield $\begin{bmatrix}5*1 & 5*2 & 5*3\\ 5*4 & 5*5 & 5*6\end{bmatrix} = \begin{bmatrix}5 & 10 & 15\\ 20 & 25 & 30\end{bmatrix}$. We can break the multiplication task into subtasks per row and create a child to do the multiplication for each row. In our example, we need a child to do the 5 * [1 2 3] and another one for 5 * [4 5 6].

In the below program, we first ask the user to input the size of the matrix A, and then we ask her to fill out the matrix:

```
hfani@bravo:~/lab09$ vi matrix_mul_row.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int main(int argc, char *argv[]){
        int row = 0;
        int column = 0;
        row = atoi(argv[1]);
        column = atoi(argv[2]);

        int *A = (int *) malloc((row * column) * sizeof(int));
        int i, j;
        printf("Enter the matrix elements:\n");
        for (i = 0; i < row; i++) {
                for (j = 0; j < column; j++) {
                        printf("A[%d,%d] = ", i, j);
                        scanf("%d", A + i * column + j);//matrix elements is stored in row-wise
                }
                printf("\n");
        }
```

As seen, we allocate the matrix dynamically using the heap space. So, we cannot access the matrix element using the index operator []. We have to find the element of the matrix manually. To do so, we need to know how C program stores a 2D matrix. Memory is a 1D array of bytes. C compiler must convert the 2D matrix into a 1D array. This is done by row-wise order. For instance, $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ is stored as [1, 2, 3, 4, 5, 6]. So, if you want to find the element at row i and column j, A[i, j], you have to pass i rows, each of which has 3 elements, (i * 3), and then move j columns forward. So, A[i, j] should be rewritten as *(A + (i * 3) + j).

Then, we ask for a number from the user with which we want to multiply the matrix:

```c
int n = 0;
printf("Enter a number:");
scanf("%d", &n);
```

Now, we can create a child process per row and leave it to the child to do the multiplication:

```c
for(int i = 0; i < row; ++i){
        pid_t pid = fork();
        if (pid == -1){//not able to create child
                printf("parent PID %d => \n", getpid());
                for (j = 0; j < column; j++) {
                        *(A + i * column + j) = n * (*(A + i * column + j));
                        printf("%d * A[%d,%d] = %d\n", n, i, j, *(A + i * column + j));
                }
        }
        else if (pid == 0) {//child
                printf("child%d PID: %d \n", i, getpid());
                for (j = 0; j < column; j++) {
                        *(A + i * column + j) = n * (*(A + i * column + j));
                        printf("%d * A[%d,%d] = %d\n", n, i, j, *(A + i * column + j));
                }
                exit(0);
        }
}
```

And finally, we wait() until all children are done. Then, we print out the final result:

```c
wait(0);
printf("Final matrix elements:\n");
for (i = 0; i < row; i++) {
        for (j = 0; j < column; j++) {
                printf("A[%d,%d] = %d\n", i, j, *(A + i * column + j));
        }
        printf("\n");
}

exit(0);
```

Let’s have a look at a sample run:

```
hfani@bravo:~/lab09$ cc matrix_mul_row.c -o matrix_mul_row
hfani@bravo:~/lab09$ ./matrix_mul_row 2 3
Enter the matrix elements:
A[0,0] = 1
A[0,1] = 2
A[0,2] = 3

A[1,0] = 4
A[1,1] = 5
A[1,2] = 6

Enter a number:2
child0 PID: 4098201
2 * A[0,0] = 2
2 * A[0,1] = 4
2 * A[0,2] = 6
child1 PID: 4098202
2 * A[1,0] = 8
2 * A[1,1] = 10
2 * A[1,2] = 12
Final matrix elements:
A[0,0] = 1
A[0,1] = 2
A[0,2] = 3

A[1,0] = 4
A[1,1] = 5
A[1,2] = 6
```

As seen, the children have done their work correctly by multiplying each row by 2, but when the parent prints out the final result, there is no change to the matrix! Why?

Step 4. Lab Assignment

You have to:

1) Explain the problem with the program in step 3
2) Fix the problem when creating a child process per *column*.

Hint: a possible solution would be using a file to store the results from children. However, if a single file is used and all the children wanted to write in a single file, issues raise!

The sample code for steps 1, 2 and 3 have been attached in a zip file named lab09_hfani.zip.

1.1. Deliverables

You will prepare and submit the program in one single zip file lab09_uwinid.zip containing the following items:

(90%) lab09_uwinid.zip

- (10%) problem.pdf/jpg/png =>why the program in step 3 is not working properly.
- (60%) matrix_mul_col.c =>fixed the problem in step 3 for column-based child creation
- (20%) results.pdf/jpg/png =>the image snapshot of your program run
- (Optional) readme.txt

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of uwinid, use your own account name, e.g., mine is hfani@uwindsor.ca, so, lab09_hfani.zip