Lab 5 – Semaphore &
Shared memory programming

Notice:
- Submitted file name: lab5.tar.bz2
- Student can modify source code
- All cheats will get ZERO if detected

1 Introduction

1.1 Goal
Writing a C program to calculate sum of prime numbers of integers using semaphore and shared memory techniques.

1.2 Prerequisite
- C programming
- Multi-process programming
- Semaphore programming
- Shared memory programming
- Makefile

1.3 Description
Input:
- Integer numbers

Output:
- Sum of prime numbers of those integer numbers

Implement the above program by using shared memory to share their result and using semaphore to solve conflict problem when these processes read from or write to shared memory at the same time.

1.4 Implementation guide
- When running, program will using fork function to duplicate to 2 processes: parent process and child process.
- These two processes works as work pool model (as illustrated in the following figure)
- Integer numbers on the command line are stored in an array.
- These two processes use shared memory to share their data. The shared memory segment includes 2 integer numbers, the first one denotes the next available element index of array, the second one denotes sum of prime numbers of all previous integers.
- Before getting an element in array to process, process reads the first number in shared memory to get the element index. After reading that element, that process increases the index by 1 and writes the new value to shared memory, index of array is increased by 1.
- After that, the process calculates sum of prime numbers of the element it has read, then it accumulates to the total sum (by reading the second number in shared memory), then it writes the new total sum into shared memory.
- After processing all elements of input array, process that processes the last element prints out the result of input numbers.
- To prevent inconsistent result of reading and writing between these two processes, we use a semaphore to synchronize these processes.

### 1.5 Programming techniques

#### 1.5.1 Parsing command line's arguments

A command line may have some arguments, in the program we can get these ones by using `argc` & `argv` arguments of function main.

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

With the above declaration, `argc` counts number of arguments we pass to the command line (including executable file name), and `argv` contains strings corresponding to those arguments in the order.

For example, when we execute command:

```
sum 2 5 7 9
```

to calculate all numbers we pass as arguments, the above command will result in `argc=5`, and `argv={“sum”, ”2”, ”5”, ”7”, ”9”}`.

#### 1.5.2 Using shared memory

Referring to lab slides.

#### 1.5.3 Using semaphore

Referring to lab slides.

### 2 Requirements

Program syntax:

```
sumprime [positive integer number]+
```
Arguments of the program are positive integer numbers, the number of arguments must be greater than or equal to 1.
The result of the above program is printed on the screen.

3 Examples
$ ./sumprime 3
  3
$ ./sumprime 6
  5
$ ./sumprime 8
  6
$ ./sumprime 3 6 8
  14