Introduction to IPCs: Shared Memory

Objective

- To introduce process communication
- To better understand the use of shared memory to communicate between processes.

Description

Many processes can live their lives without having to interact with other processes (independent processes). For many reasons, including convenience, computation speedup, and information sharing, some processes choose to interact with each other. Some will use/consume data produced by other processes (cooperating processes). A programmer who expects her/his program to communicate with other programs can build their communication tools. They should provide a medium of communication and provide the tools to guarantee proper access to the data:

- Buffer must be created with controlled access (read, write, or both)
- Data must be protected from errors caused by the difference in the speed between the producer and the consumer processes:
    - Data is protected from being overwritten before it's read
    - Data protected from being read multiple times

Modern operating systems provide the programmer with tools to facilitate correct communication between processes. These tools abstract a buffer and provide the proper methods to access the data. Some of these tools are designed to work with communicating processes across computer systems and others are intended to work on the same machine. Communication between processes on a single machine is known as Inter-Process Communication (IPC) and some of the tools used in IPC are: shared memory, pipes, and message queues.

Inter Process Communication through shared memory is when two or more processes use a common block of memory to communicate. By writing to the shared memory segment, changes made by one process can be viewed by other processes.

For a process to use shared memory, the following needs to be done:

- Create a shared memory segment or use an already created shared memory segment (`shmget()`). A shared memory segment is identified through a numeric key (i.e., int).
- Attach the shared memory segment to the process's address space (`shmat()`)
- When done, detach the shared memory segment from the process's address space (`shmdt()`)
- To perform control operations on a shared memory segment use (`shmctl()`)

Check out the man pages for the following system calls:

```
int shmget(key_t key, size_t size, int  flag);
void * shmat(int shmid, const void *addr, int flag);
int shmdt(const void *addr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

-

The following are shell commands that can be used to obtain the status of system-wide shared memory resources:

```
>ipcs
>ipcs -m (Shows only shared memory)
```

For more information on `ipcs`:

```
>man ipcs
```

The **`ipcrm`** command removes a message queue, a semaphore set or a shared memory:

```
>ipcrm  -M key
>ipcrm  -m id
```

For more information on ipcrm:

```
>man ipcrm
```

Submission:

- *One compressed file that contains all files (C, text and pdf files if needed)*
- *Make sure to do the following:*
  - *Check the success of every system call you use*
  -

1. Run the following programs: (20 Points)

Download and execute `printLine.c` and `changeLine.c` that use a shared memory to print a character multiple times. Please use the system command `ipcs` before and after every instruction (below) and comment on the information retrieved by `ipcs`.

Follow the instructions below to understand its behavior of the 2 files:

a. Compile both files and generate 2 executable files:
```
>gcc printLine.c -o printLine
>gcc changeLine.c -o changeLine
```
b. Run the system command `ipcs` and comment on its output (i.e., Shared Memory keys).
c. Which program should you run first: `printLine` or `changeLine`? And why?
d. Run the `printLine` executable file in one shell and observe its performance.
e. In a different shell, run `changeLine` multiple times (`changeLine <character> <number>`). After each time, please run the system command `ipcs` and comment on its output.

Please include text answers to parts b-e in a separate text file or as comments in your source file.

2. Simulate a producer and a consumer in a single program: (25 Points)

Write a single C program where simple items (e.g., int) are produced and consumed. The user uses the keyboard to control the production and consumption of items by entering:

- 'p': the program will produce an item if the buffer has space. The program prints a <Buffer is full> message if the buffer is full.
- 'c': the program to consume the oldest item in the buffer. The program prints a <Buffer is empty> message if the buffer is empty.
- '0': the program to terminate.

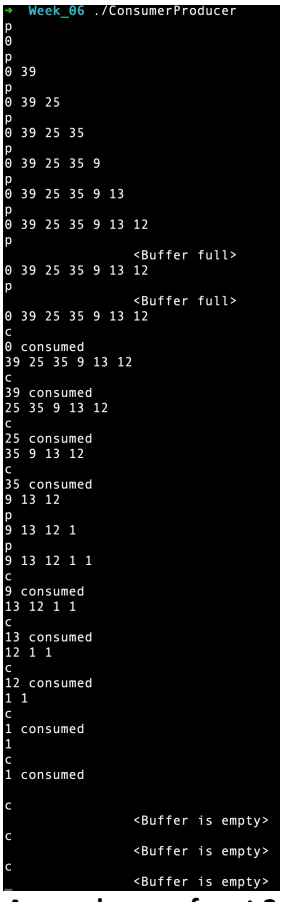
Use a bounded buffer so the production part of the code will write to and the consumer part will read from. Production should be limited to the buffer size after which no new items should be produced. If the buffer is empty, then nothing can be consumed. To facilitate this, use 2 variables/pointers (in, out) to synchronize the use of the buffer:

- When `in = out`: buffer is empty of new items, and the reader should stop.
- When `in+1 = out`: buffer is full, and the writer should stop.

For more details on synchronizing production and consumption, refer to ***slides 4-6*** in "***Lec_05 Process Communication*** ". Use the following structure in your code:

```
#define MAX_SIZE 8
typedef struct bufferStruct {
         int in;
         int out;
         int content[MAX_SIZE];// will hold items produced (single int)
 }bufferStruct;
```

It represents the bounded buffer and the 2 variables used to synchronize production and consumption.



A sample run of part 2

3. Using shared memory to communicate between producer and consumer processes: (25 Points)

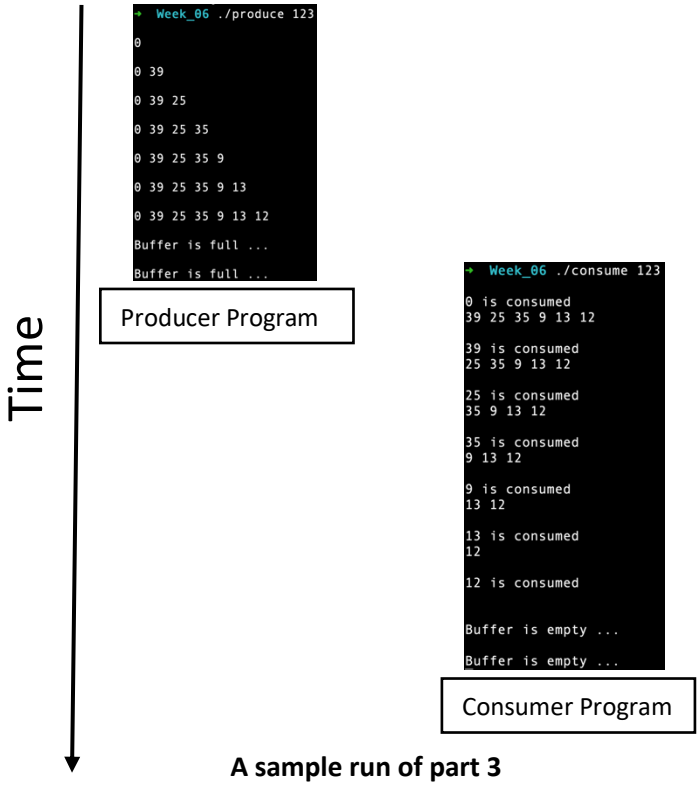
Extend part 2 above by writing 2 separate programs: producer and consumer programs that use a shared buffer. The producer program (produce.c) will produce simple items (e.g., int) and place them in a shared buffer. The consumer program (consume.c) will read and consume these items (i.e., process them). Use the structure (below) to model the shared memory that will facilitate sharing the items between the producer and the consumer processes:

```
#define MAX_SIZE 8
typedef struct bufferStruct {
         int in;
         int out;
         int content[MAX_SIZE];// will hold ASCII code of characters
}bufferStruct;
```

Ensure the structure is saved in a separate header file `buffer.h` that you can include in your `consume.c` and `produce.c` files.

The programs will run independently and will do the following:

- `produce.c`: Whenever the user hits enter in the producer process, it generates a random value (int) and stores it in the shared buffer until it is full. If the user hits enter after that, a "`buffer is full …`" error message should be printed.
- `consume.c`: Every time the user hits enter in the consumer process, it consumes the oldest item in the buffer and prints it to the screen until the buffer is empty. If the user hits enter after that then a "`buffer is empty …`" error message should be printed.
- Any time the user enters '0', the program terminates (producer and/or consumer).



A sample run of part 3

4. Using shared memory to copy a file: (30 Points)

Write 2 programs, CP1.c and CP2.c, that will use shared memory to copy data from a source to a destination file. The 2 files are managed by different processes (CP1 and CP2): the source file is managed by one process (CP1), and the destination file is managed by another (CP2). **Assume that the buffer used to read/write the files is too small to contain the entire file so multiple read and write calls (looped) are necessary to process the entire file.** Find detailed requirements below.

The programs do the following:

CP1.c:

1. Creates a shared memory that will be used as a temporary buffer.
2. Opens the source file, which was passed as a command line argument, to read data from.
3. Reads a chunk of data from the source file and writes it to the shared memory. Waits for CP2 to read the shared memory before it writes the next part of the source file to the shared memory. The cycle continues until the file is fully copied to shared memory.

CP2.c:

1. Opens an already-created shared memory that will be used as a temporary buffer.
2. Opens the destination file, which was passed as a command line argument to write data to.
3. Reads a chunk of data from the shared memory and writes it to the destination file. Waits for CP1 to write the next part of the source file before it reads it from the shared memory and writes to the destination file. The cycle continues until the file is fully copied and EOF is reached.
4. After writing is complete, detaches and deletes shared memory.