Operating Systems HW2
2018-2019

Author: Nemanja Nedic
A.M. 1115201400124

Contents

Intro

Compile & Run

Makefile is provided for compilation.
Other goals: make clean, make count

Before running make sure shm segment keys 0x1111 and 0x2222 are available. If not a bash script that cleans your IPCs for you is provided. Make sure its executable (chmod +x **./cleanIPC.sh**). You should run it just to be safe.

Run:
```
./hw2 -n num_of_processesP -i num_of_iterations
```

Organization of directories and files

Inside each file you will find the following:

main.c: General outline of the program. Such as: Init Shared memory, fork child processes, wait for them to finish, and prints as mentioned in the handout.

ErrorCodes.h: error codes used to handle irregular termination

Globals.h: global variables and their definitions

InQueue.h: everything needed to send messages to Process C.

OutQueue.h: everything needed to send messages to Processes P.

SharedMem.h: operations on shared memory segments and semaphores.

Design Outline

NOTE: All of the requirements specified by the assignment handout have been implemented.

I use two shared memory segments. One for sending messages to C (InQueue) and one for sending messages to P (OutQueue). Discussed below.

Both types of processes use System V semaphores to control access to the queues as taught in class.
Once process C is done, it will write a special message to the queue for each Process P to read and terminate.

InQueue

Messages sent from P are of dynamic size. That is why along with the pid of the P Process that is sending the message to C, we have to include the size of the message that follows as well.

InQueue shared memory segment fits many messages. In order to keep track of those and to control write access to the Shared Memory, we keep a Data Structure called InQueueHeader at the start of the Shared Memory Segment. For specific details about each member to the Data Structures consult the image below.

```c
/**
 * This header exists before each message data block. It is there to provide information about the data and its size.
 */
typedef struct InMessageHeader {
    size_t message_size;    //size of the string message including \0
    pid_t pid;
} InMessageHeader;

/**
 * Processes P's write to this Queue and process C reads from it.
 * The InQueueHeader is included in the start of the Shared Memory segment.
 * Upon initialization the start_ptr and end_ptr both point after the InQueueHeader.
 * When a write is performed:
 *  The writer will check:
 *  -If some other process is already writing to the Queue (writeSem is down). If yes then wait (writeSem up).
 *  -If there is enough space to write the payload, by trying to decrease the freeSpace
 *  semaphore by (sizeof(InMessageHeader)+msg_size).
 *  After successfully writing the data the msgNum semaphore will be upped by 1 as well and the end_ptr will be moved.
 * When a read is performed:
 *  -The reader will check if there are available messages to read (down the writeSem by 1)
 *  -When the message becomes available, the data is copied by the C process, and the freeSpace semaphore semval is
 *   increased in order to represent that the space has been read, and freed. With this action the start_ptr will also be moved.
 */
typedef struct InQueueHeader {
    volatile int read_ptr; //points to where the allocated space of InQueue starts, in bytes. Set to 0 at start.
    volatile int write_ptr;   //points to where the allocated space of InQueue ends, in bytes. Set to 0 at start.
    key_t semkey_writeSem;  //down when someone is writting to the queue, upped once they finish. Set to 1 at start.
    key_t semkey_msgNum;   //upped when msg inserted, downed when msg removed from queue. Set to 0 at start.
    key_t semkey_freeSpace; //value of this sem reveals the available space in queue, in bytes.
} InQueueHeader;
```

OutQueue

Messages sent from C are of set size, since they always send a pid and a MD5 hash. That makes things a bit easier than with InQueue.
Again, the OutQueueHeader is included at the start of the second shared memory segment, and all messages are written right after it in the memory.

```c
typedef struct OutMessage {
    pid_t pid;
    char message[MD5_HASH_SIZE+1];
} OutMessage;


typedef struct OutQueueHeader {
    volatile int first;  //next message to be read from consumer. Set to 0 at start.
    volatile int last;  //last message in the queue. Set to -1 at start.
    key_t semkey_read;  //binary semaphore that controls who reads next from the queue. Set to 1 at start.
    key_t semkey_msgNum;//semaphore that gives the number of messages available for reading in queue. Set to 0 at start.
} OutQueueHeader;
```