COMP 530 INTRODUCTION TO OPERATING SYSTEMS

Fall 2017
Kevin Jeffay

Homework 6, November 1

Due: 11:15AM, November 20, 2017

The "Linux Warmup Revisited Revisited" Revisted: Producer/Consumer Interaction with a Bounded Buffer Using Linux Processes

In Homework 3 you implemented your sequential C program from Homework 1 as a multithreaded *C* program using the *ST* threading package. By using threads within a single Linux address space, you were able to use global C program variables as shared memory between the threads.

In this assignment you will build the same processing pipeline but this time using Linux processes instead of threads. The goal of the assignment is to learn how to implement shared memory between Linux processes. To do this, you will learn how to use a Linux facility called a *memory mapped file*.

Specifically you are to re-implement your Homework 3 pipeline using Linux processes instead of *ST* threads (or any other form of threads). If you do this right, very little of your Homework 3 code should change. In particular, you should still use a bounded buffer "object"/abstract data type. However, inside the object/data type you should use a memory-mapped file as the shared memory for the buffer. In addition, instead of using *ST* semaphores for inter-process synchronization you should use the implementation of POSIX semaphores present in Linux.

To implement your shared buffers you'll also use a POSIX semaphore facility to synchronize access to your memory mapped files.

Ground Rules

For this assignment your challenge is to figure out what memory-mapped files are and how you can use them effectively to implement your bounded buffer object/data type. (In addition you have to figure out how to use POSIX semaphores.) Given this challenge, you may consult on-line resources but you MUST list all the sources/sites you considered in your code (you need not list the Linux man pages, just list any other sources you used other than the man pages). A sample file showing how memory mapped files are used is appended to this handout.

As part of the assignment, in your main program file, include comments that explain exactly how you are using memory-mapped files.

Grading

Submit your program electronically for grading by emailing *jacobf@cs.unc.edu* when the program is ready for grading.

You will submit three (3) files for this homework assignment. Your main program should be in a file called *HW6.c*. Your bounded buffer abstract data type should be in files *buffer.c* and *buffer.h*. It's OK to

use/create additional files (and you likely will need additional files), just give these files appropriate mnemonic names.

Important: When you email the TA that your program is ready for grading, please also include two command-line fragments that the TA can cut and paste into a script to build and execute your program. The first is a command line fragment the TA can use to get a copy of your program and the second is the verbatim command line that you used to compile your program. For example if your Homework 6 solution used the files HW6.c, Helper.c, Helper.h, buffer.c, and buffer.h then in your submission email you'd include to two command lines:

```
cp ~yourloginID/comp530/submissions/HW6/{HW6.c,Helper.c,Helper.h,buffer.c,buffer.h} .
gcc HW6.c Helper.c buffer.c
```

Note again that late submissions will not be accepted and that non-fully functional programs will receive little, if any credit.

As before, the program should be neatly formatted (*i.e.*, easy to read) and well documented. For this program, approximately 60% of your grade for a program will be for correctness and 40% for "programming style" (appropriate use of language features, including variable/procedure names, implementation of a bounded buffer abstract data type, structure of the process pipeline, termination of processes, *etc.*), and documentation (descriptions of functions, general comments, use of invariants, assertions, pre- and post conditions).

```
/*
 * This is a sample program for using memory mapping.
 * It is done by the function mmap (see man 2 mmap).
 * In this example there are two processes that use a memory mapped loction
 * to communicate.
 * The first process simply reads from stdin and puts chars in an array full.
 * The second process reads from the array and prints out each char until it
 * has read the whole array.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>  // mmap, munmap
#include <unistd.h>    // sleep
#include <sys/types.h> // kill
#include <signal.h>    // kill

#define MESSAGE_LENGTH 80
#define ERROR -1
#define SLEEP_TIME 1

/* This is the object that is mapped to memory. It has an array
 * and a count of objects in the array.
 */
typedef struct {
    char MSG[MESSAGE_LENGTH];
    int count;
} messageObject;

// Prototypes for functions
void writer(messageObject *state);
void reader(messageObject *state);
void waitForChildren(pid_t*);
pid_t forkChild(void (*func)(messageObject *), messageObject* state);
messageObject* createMMAP(size_t size);
void deleteMMAP(void*);

int main () {

    //get address of memory mapped location.
    messageObject* state = createMMAP(sizeof(messageObject));
    state->count = 0; // initialize count

    //Fork children
    pid_t childpids[2];
    childpids[0] = forkChild(writer, state);
    childpids[1] = forkChild(reader, state);

    //wait for them
    waitForChildren(childpids);

    //cleanup
    deleteMMAP(state);
    exit(EXIT_SUCCESS);
}
```

```
messageObject* createMMAP(size_t size){
  //These are the neccessary arguments for mmap. See man mmap.
  void* addr = 0;
  int protections = PROT_READ|PROT_WRITE; //can read and write
  int flags = MAP_SHARED|MAP_ANONYMOUS; //shared b/w procs & not mapped to a file
  int fd = -1; //We could make it map to a file as well but here it is not needed.
  off_t offset = 0;

  //Create memory map
  messageObject* state =  mmap(addr, size, protections, flags, fd, offset);

  if (( void *) ERROR == state){//on an error mmap returns void* -1.
      perror("error with mmap");
      exit(EXIT_FAILURE);
  }
  return state;
}

void deleteMMAP(void* addr){
      //This deletes the memory map at given address. see man mmap
      if (ERROR == munmap(addr, sizeof(messageObject))){
            perror("error deleting mmap");
            exit(EXIT_FAILURE);
      }
}

pid_t forkChild(void (*function)(messageObject *), messageObject* state){
     //This function takes a pointer to a function as an argument
     //and the functions argument. It then returns the forked child's pid.

      pid_t childpid;
        switch (childpid = fork()) {
                case ERROR:
                        perror("fork error");
                        exit(EXIT_FAILURE);
                case 0:
                        (*function)(state);
                default:
                       return childpid;
        }
}

void waitForChildren(pid_t* childpids){
      int status;
      while(ERROR < wait(&status)){ //Here the parent waits on any child.
            if(!WIFEXITED(status)){ //If the termination err, kill all children.
                  kill(childpids[0], SIGKILL);
                  kill(childpids[1], SIGKILL);
                  break;
            }
      }
}
```

```
// The execution path for writer
void writer(messageObject* state) {
    //This process reads a char & writes it until the array is full.
    while(1){
      while (MESSAGE_LENGTH == state->count){ //busy wait if condition is true
            sleep(SLEEP_TIME);
            // This is a naive/bad approach to synchronization, wouldn't a
POSIX semaphore be great!

      }
        char c = fgetc(stdin); //read in a char
        state->MSG[state->count] = c;
      state->count++;
        if (EOF == c){ //if EOF break out of loop and exit
            state->count = MESSAGE_LENGTH; //Set this condition so that the
other process can read
          exit(EXIT_SUCCESS);
        }
    }

    return;
}

// The execution path for reader
void reader(messageObject* state) {
    //simply prints a char until the array is empty.
    int localCount = 0;//used to keep track of what is next to read.
    while(1){
      while (MESSAGE_LENGTH > state->count){
            sleep(SLEEP_TIME);
            //Once again a bad way to do synchronization.
      }
            //Read in char
      char c = state->MSG[localCount];

      if (EOF == c){//break if EOF
            exit(EXIT_SUCCESS);
        }
        //print char
      putchar(c);
      //update count
      localCount++;
      //check if array is full and reset counts
      if (MESSAGE_LENGTH == localCount){
          localCount = 0;
          state->count = 0;
      }
}

    return;
}
```