COEN 177: Operating Systems
Lab 3: Inter-Process Communication and Pthreads

Objectives

1. To demonstrate the use of pipes and shared memory as inter-process communication (IPC) mechanisms.
2. To demonstrate the use of the pthread library.

Guidelines

In Lab2, you have learned to develop multiprocessing and multithreading applications using `fork()` and `pthread_create()` respectively. With `fork()`, the kernel creates and initializes a new process control block (PCB) with a new address space that copies the entire content of the parent process's address space. The new process is a child process, and it inherits the execution context of the parent (e.g. open files). Unlike threads, processes do not share their address space. Therefore, processes use an IPC mechanism to exchange information. The kernel provides three IPCs: Pipes, Shared Memory, and Message Queues.

Pipes are traditional Unix inter-process communication. The | symbol is used in the command line to denote a pipe. Try the following commands at the command line:

- who | sort
- ls | more
- cat /etc/passwd | grep root

Pipes can be created in a C program using `int pipe(int P[2])` system call, where:

- P[1]: file descriptor on the upstream (input) end of the pipe
- P[0]: file descriptor on the downstream (output) end of the pipe

Typically Process A (parent) creates a pipe and forks twice, creating B and C. Each process then closes the ends of the pipe it does not need; process B closes the downstream (output) end, process C closes the upstream (input) end, and process A closes both. Processes B and C can then execute other programs using exec, as file descriptors are retained. In this case, you will need to use `int dup2(int OldFileDes, int NewFileDes);` to overwrite the standard input and standard output file descriptors (0 and 1 respectively, which by default point to the terminal) with the file descriptor of the appropriate end of the pipe. This sends process B’s output to the input of process C.

With shared memory, one process creates a shared segment using the `id = shmget( key, MSIZ, IPC_CREAT | 0666);` system call, then, other processes can get the id of the segment with that key using `id = shmget( key, MSIZ, 0 );`. Note that key is a key_t variable that has the same value between all processes and MSIZ is the size of the shared segment. All involved processes attach to the segment using `ctrl = shmat( id, 0, 0);`, which returns a void* to the shared segment which can then be written to or read from. When a process is done with the shared segment, it detaches from it using `shmdt(ctrl);`.

With message queues, one process creates a message queue using `id = msgget( key, IPC_CREAT | 0644);` system call. Then, other processes can get the id of the queue using `id = msgget( key,0 )` system call. Processes can send messages to the queue and receive messages from the queue using the `v = msgsnd( msid, ptr, length, IPC_NOWAIT);` and `v = msgrcv( msid,ptr,length,type,IPC_NOWAIT );` system calls, respectively.

In this lab, you will use pipes and shared memory for IPC. Include the following libraries for IPC.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

C Program with pipe IPC

Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment

Step 1. [10%] Compile and run the following program and write your observations. Then, modify it to pass the -l option to the ls command.

```
/*Sample C program for Lab 3*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
```

```
//main
int main() {
    int fds[2];
    pipe(fds);
    /*child 1 redirects stdin to downstream of pipe */
    if (fork() == 0) {
        dup2(fds[0], 0);
        close(fds[1]);
        execlp("more", "more", 0);
    }
    /*child 2 redirects stdout to upstream of pipe */
    else if (fork() == 0) {
        dup2(fds[1], 1);
        close(fds[0]);
        execlp("ls", "ls", 0);
    }
    else {  /*parent closes both ends and wait for children*/
        close(fds[0]);
        close(fds[1]);
        wait(0);
        wait(0);
    }
return 0;
}
```

Step 2. [10%] Compile and run the following program and write your observations.

```
/*Sample C program for Lab 3*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
// main
int main(int argc,char *argv[]){
   int  fds[2];
   char buff[60];
   int count;
   int i;
   pipe(fds);
   if (fork()==0){
       printf("\nWriter on the upstream end of the pipe -> %d arguments \n",argc);
       close(fds[0]);
       for(i=0;i<argc;i++){
           write(fds[1],argv[i],strlen(argv[i]));
       }
       exit(0);
   }
   else if(fork()==0){
       printf("\nReader on the downstream end of the pipe \n");
       close(fds[1]);
       while((count=read(fds[0],buff,60))>0){
           for(i=0;i<count;i++){
               write(1,buff+i,1);
               write(1," ",1);
           }
           printf("\n");
       }
       exit(0);
    }
   else{
      close(fds[0]);
      close(fds[1]);
      wait(0);
      wait(0);
   }
return 0;
```

}

Step 3. [10%] Modify the program in Step 2. so that the writer process passes the output of “ls” command to the upstream end of the pipe instead of writing its arguments to it. You may use dup2(fds[1],1); for redirection and execlp("ls", "ls", 0); to run the “ls” command.

Step 4. [15%] Write a C program that implements the shell command: `cat /etc/passwd | grep root`

Producer – consumer with pipes

In Computer Science, the producer–consumer problem is a classic multi-process synchronization example. The producer and the consumer share a common fixed-size buffer. The producer puts messages into the buffer while the consumer removes messages from the buffer. Pipes provide a perfect solution to this problem because of their built-in synchronization capability. So as a programmer, you do not have to worry about whether the buffer is empty or full to produce or consume messages.

Step 5. [15%] Write a C program that implements some form of producer-consumer message communication using pipes.

Step 6. [15%] Write a C program that implements some form of producer-consumer message communication using a shared memory segment.

Pthread_create()

Step 7. [10%] Compile and run the following program, then list how many threads are created and what values of i are passed. You may observe a bug in the program where multiple threads may print the same value of i. Regardless of whether you observe the issue, how could it happen?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *go(void *);
#define NTHREADS 10
pthread_t threads[NTHREADS];
int main() {
    int i;
    for (i = 0; i < NTHREADS; i++)
        pthread_create(&threads[i], NULL, go, &i);
    for (i = 0; i < NTHREADS; i++) {
    printf("Thread %d returned\n", i);
        pthread_join(threads[i],NULL);
    }
    printf("Main thread done.\n");
    return 0;
}
void *go(void *arg) {
 printf("Hello from thread %d with iteration %d\n",  (int)(unsigned long) pthread_self(), *(int *)arg);
 return 0;
}
```

Step 8. [15%] What is the fix for the program bug in Step 7? Write a program to demonstrate your fix.

Requirements to complete the lab

1. Show the TA the correct execution of the C programs.
2. Submit your answers to questions, observations, and notes through Camino
3. Upload the source code for all your programs as .c file(s) to Camino.

Please start each program/text file with a descriptive block that includes the following information at minimum:

```
# Name: <your name>
# Date: <date of lab>
# Title: Lab2 – Step <step#> - <task>
# Description: This program computes <complete description here>
```