COEN 177: Operating Systems
Lab assignment 3: Pthreads and inter-process Communication – Pipes

Objectives

1. To develop multi-process application programs
2. To demonstrate the use of pipes as an inter-process communication (IPC) mechanism
3. To demonstrate the use of pthreads

Guidelines

In Lab2, you have learned to develop multi-processing and multi-threading 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 address space of the parent process. This new process is a child process, and it inherits the execution context of the parent (e.g. open files). Unlike with threads, parent and child processes do not share their address space. Therefore, processes must use an IPC mechanism to exchange information. The kernel provides three such IPCs: Pipes, Shared Memory, and Message Queues.

Pipes are the traditional Unix IPC. The | symbol is used in the command line to denote a pipe. Try the following commands:

- 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]: set to the upstream file descriptor of the new pipe
- P[0]: set to the downstream file descriptor of the new pipe

Typically Process A (parent) calls pipe() and forks twice to create B and C. Each process then closes the ends of the pipe it does not need. Process B would close the downstream end, process C would close the upstream end, and process A would close both ends. Processes B and C then exec() other programs, retaining their file descriptors. To redirect existing file descriptors (e.g. stdin, stdout, stderr), you will need to use int dup2(int OldFileDes, int NewFileDes). Note that stdin, stdout, and stderr are file descriptors 0, 1, and 2 respectively.

With shared memory, one process creates a shared segment using id = shmget(key, MSIZ, IPC_CREAT | 0666); where MSIZ is the size of the desired memory segment. Other process(es) can then get the ID of the created segment with id = shmget(key, MSIZ, 0). Using the same key, each process attaches to itself the shared segment using shm_ptr = shmat(id, 0, 0) , then processes can begin to read and write at the shared memory address. Before exiting, all processes attached to a segment should detach from it with shmdt(shm_ptr) and then the process that created it should call shmctl(id, IPC_RMID, 0) to remove the segment entirely.
With message queues, one process creates a message queue using id = msgget(key, IPC_CREAT | 0644). Other process(es) get the ID of the created segment using id = msgget(key,0). Processes send and receive messages from a queue using v = msgsnd(msid, ptr, length, IPC_NOWAIT) and v = msgrcv(msid,ptr,length,type,IPC_NOWAIT), respectively.

In this lab, you will practice use of pipes for IPC. Include the following libraries.

```
        #include <sys/types.h>
        #include <sys/ipc.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. Compile and run the following program

```
/*Sample C program for Lab assignment 3*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
//main
int main() {
   int fds[2];
   pipe(fds);
   /*child 1 duplicates downstream into stdin */
   if (fork() == 0) {
      dup2(fds[0], 0);
```

```
        close(fds[1]);
        execlp("more", "more", 0);
    }
    /*child 2 duplicates upstream into stdout */
    else if (fork() == 0) {
        dup2(fds[1], 1);
        close(fds[0]);
        execlp("ls", "ls", 0);
    }
    else {  /*parent closes both ends and waits for children*/
        close(fds[0]);
        close(fds[1]);
        wait(0);
        wait(0);
    }
return 0;
}
```

Step 2. Compile and run the following program

```
/*Sample C program for Lab assignment 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. Modify the program in Step 2 so that the writer process passes the output of the “ls” command to the upstream end of the pipe. You may use dup2(fds[1],1); for redirection and execlp("ls", "ls", 0); to run the “ls” command.

Step 4. Write a C program that uses the same system calls to implement the following shell command:
cat /etc/passwd | grep root

Producer – consumer with pipes

Step 5. 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 to the buffer while the consumer removes messages from the buffer. Pipes provide an easy solution to this problem because of their built-in synchronization capability. Thus, as a programmer, you do not have to worry about whether or not the buffer is empty or full to produce or consume messages.

Write a C program to implement the producer-consumer message communication using pipes. The program must allow for an arbitrarily large number of messages (up to maxint, or use a while loop) to be passed before termination.

Pthread_create()

Step 6. Compile and run the following program, then list how many threads are created and what values of i appear in the output.

```
#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 %ld with iteration %d\n",  (long)pthread_self(), *(int *)arg);
 return 0;
}
```

Write down your observations. You may notice a bug in the program (depending on system load), where multiple threads may print the same values of i. Why could this happen?

Step 7. [Bonus] What is a fix for the program in Step 6 that maintains concurrent execution of the thread functions? Write a program to demonstrate that fix.

Requirements to complete the lab

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

Be sure to retain copies of your .c and .txt files. You will want these for study purposes and to resolve any grading questions (should they arise)

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

```
//Name:
//Date:
```

```
//Title: Lab3 -
//Description:
```