Programming project# 1 – Part 1 – WARM-UP programming
2/21/2023

Total points: 50 programming project points

Programs containing compilation errors will receive failing grades. Those containing run-time errors will incur a substantial penalty. You should make a serious effort to complete all programs on time. Programming assignments are individual. You should complete them with your own effort. If you need help, you should come to my office hours or contact me by email. You may discuss concepts with others in the class, but not specific program code**.** The total points of this programming project are 100. However, the 100 points are NOT same as 100 points in exams, quizzes, or quizzes. **If you are not sure about the grading policy, refer to the course syllabus.**

NOTE: Each program should be placed in its own directory. Also, you need to include a Makefile in the same directory as the source code to allow the instructor to compile your program automatically. Similarly, your half-page write-up in plain text format (NO PDF or MS-word format!) should be placed in the same directory. You will submit a single "ZIP" file that includes all the subdirectories where your programs and documentation are located. The name of ZIP file should be *yourlastname_yourFirstname_prog1part1*.zip. The single ZIP file should contain C/C++ source files and Makefiles in the following subdirectories appropriately: ./1, ./2, ./3, and ./4 ./5

- You might want to use the following Unix command to create a single zip file:
  % zip –r ***yourlastname_yourFirstname_prog1part1*.zip** ./1 ./2 ./3 ./4 ./5

You are strongly recommended to use the online book titled "Advanced Linux Programming" as a reference.

NOTE: The textbook in our course is "Operating System Concepts. 10th Edition"!

Part 1:

Warm-up exercises (typing existing code!)

[1] (8 points) Creating a separate process using the Linux/Unix fork() system call.

```c
#include <sys/types.h>      /* needed to use pid_t, etc. */
#include <sys/wait.h>       /* needed to use wait() */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>         /* LINUX constants and functions (fork(), etc.) */

int main()
{
 pid_t pid;

 pid = fork();

 if (pid < 0)
  {
   printf("A fork error has occurred.\n");
   exit(-1);
  }
 else
  if (pid == 0) /* We are in the child. */
   {
    printf("I am the child, about to call ps using execlp.\n");
    execlp("/bin/ls","ls",(char *) 0);
   /*  If execlp() is successful, we should not reach this next line. */
    printf("The call to execlp() was not successful.\n");
    exit(127);
   }
  else  /* We are in the parent. */
   {
    wait(0);               /* Wait for the child to terminate. */
    printf("I am the parent.  The child just ended.  I will now exit.\n");
    exit(0);
   }

 return(0);
}
```

Create a subdirectory called "1". Change to the subdirectory.

(1) **Type** in the C program shown in the figure above and compile/run/test the program on IT Linux system.

(2) Change the variable name "pid" to "cid" in the typed C program. This change will make some clarification.

(3) **Add** additional code to check the return value/error code of execlp() and wait() function calls. Without the additional code, you automatically lose 2 points. You can find how to check the return values of execlp() and wait() by using manual page for the functions by running the following commands on Linux terminal:

   a. **man execlp**
   b. man wait

Explain in detail how APIs (fork(), execlp(),wait()) work in a separate plain text file (NO PDF or MS-Word).

Deliverables: source code, Makefile, written document **(in plain text; at least 100 words in total; NO MS-word or PDF format!)**

DO NOT include any *.o or executable files.

)

[2] (8 points) C program illustrating POSIX shared-memory API


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

   /* create the shared memory object */
   fd = shm_open(name,O_CREAT | O_RDWR,0666);

   /* configure the size of the shared memory object */
   ftruncate(fd, SIZE);

   /* memory map the shared memory object */
   ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

Figure 3.16 Producer process illustrating POSIX shared-memory API.

The producer, shown in Figure 3.16, creates a shared-memory object named OS and writes the infamous string "Hello World!" to shared memory. The program memory-maps a shared-memory object of the specified size and allows writing to the object. The flag MAP_SHARED specifies that changes to the shared-memory object will be visible to all processes sharing the object. Notice that we write to the shared-memory object by calling the sprintf() function and writing the formatted string to the pointer ptr. After each write, we must increment the pointer by the number of bytes written.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int fd;
/* pointer to shared memory obect */
char *ptr;

    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = (char *)
     mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Figure 3.17 Consumer process illustrating POSIX shared-memory API.

Create a subdirectory called "2". Change to the subdirectory.

(1) **Type** in the C programs in Figure 3.16 (Producer process illustrating POSIX shared-memory API) and Figure 3.17 in the textbook (Consumer process illustrating POSIX shared-memory API)

(2) Add additional code to check the return value of ALL API calls including shm_open(), mmap(), and shm_unlink(), ftruncate(). In case that an error is detected, print out an appropriate message and terminate the execution of the program right away. You can find out how to check errors either by checking manual pages or by checking sample code in week5-examples-v2.1.

(3) compile/run/test the program on IT Linux system. Explain in detail how APIs (shm_open(),ftruncate(),mmap(),shm_unlink()) work in a separate text file.

- Compilation:
  - Compile the code with "-lrt" option. More details can be found by running "man shm_open".

Deliverables: source code, Makefile, written document (**in plain text; at least 200 words in total; NO MS-word or PDF format!)**

)

Change the name of the shared memory object from "OS" to something else to avoid possible conflict. (e.g., OS_yourULID)

It turns out that there is an error in the code in Figure 3.17 in textbook (consumer process using shared memory)

```
ptr = (char *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)


====>

ptr = (char *) mmap(0, SIZE, PROT_READ,   MAP_SHARED, fd, 0)
```

i.e., please get rid of PRO_WRITE option when you call mmap() in consumer code.

[3] (8 points) pid values (Fig 3.34 in the textbook)

Create a subdirectory called "3". Change to the subdirectory. **Type** in the C program in Fig 3.34 (shown in the following) and compile/run/test the program under a Linux operating system. **Record** the values of pid at lines A,B,C, and D. Explain in detail how APIs (getpid() and fprintf) works in a separate file.

Deliverables: source code, Makefile, written document (**in plain text; at least 50 words**)

DO NOT include any *.o or executable files.

Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

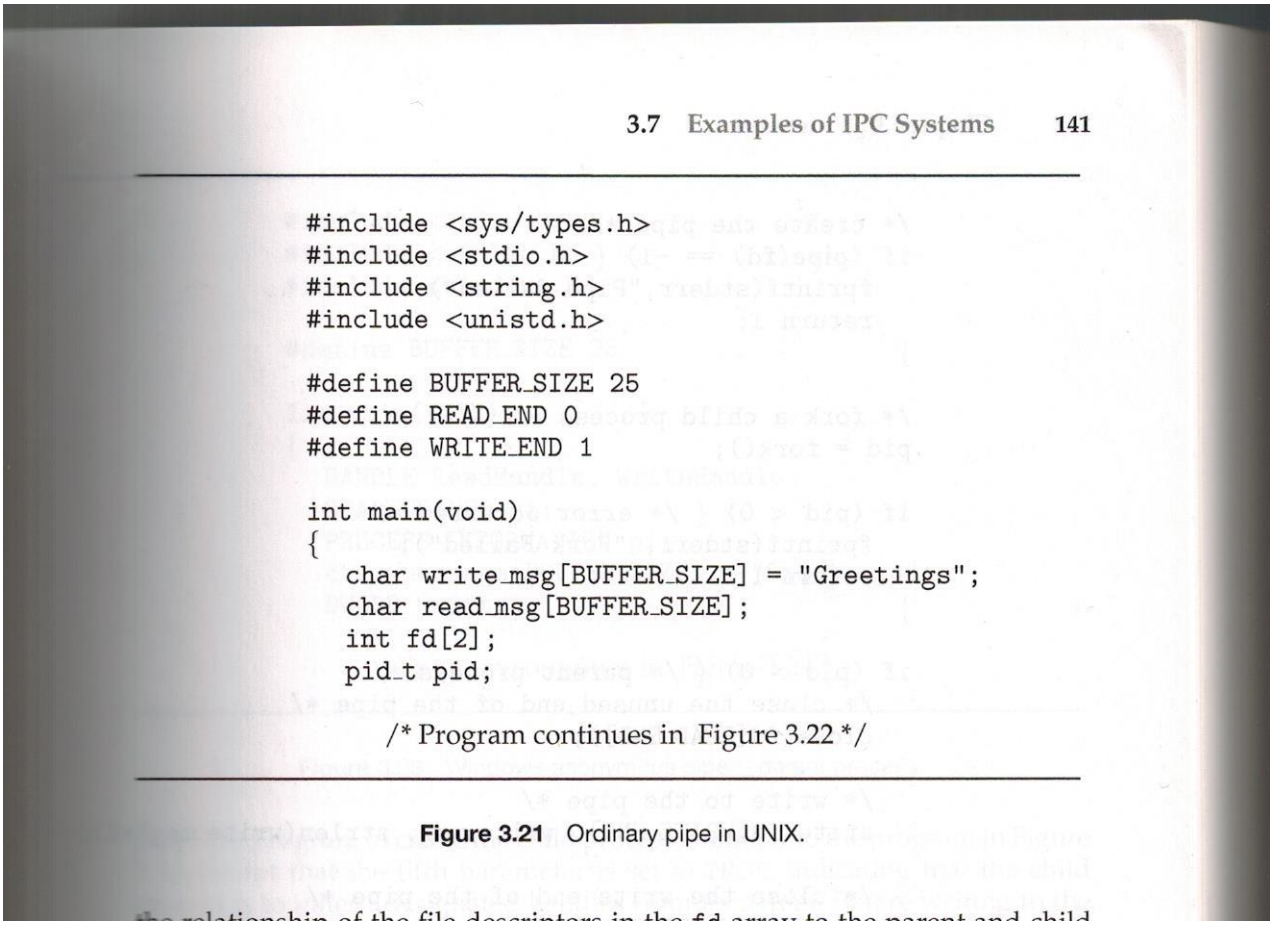
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Figure 3.34 What are the pid values?

[4] (8 points) Ordinary pipes in Linux/Unix (Figures 3.21 and 3.22 in the textbook)

Create a subdirectory called "4". Change to the subdirectory. **Type** in the C programs in Figures 3.21 and 3.22 and compile/run/test the program under a Linux operating system. **Add** additional code to check the return value/error code of every function called in this program (exceptions: printf(), close(), exit(). Without the additional code, you automatically lose 3 points.

Explain in detail how APIs (pipe(), write(), close()) work in a separate file.



3.7 Examples of IPC Systems 141

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Greetings";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

      /* Program continues in Figure 3.22 */
```

Figure 3.21 Ordinary pipe in UNIX.

```
/* create the pipe */
if (pipe(fd) == -1) {
  fprintf(stderr,"Pipe failed");
  return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
  fprintf(stderr, "Fork Failed");
  return 1;
}

if (pid > 0) { /* parent process */
  /* close the unused end of the pipe */
  close(fd[READ_END]);

  /* write to the pipe */
  write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

  /* close the write end of the pipe */
  close(fd[WRITE_END]);
}
else { /* child process */
  /* close the unused end of the pipe */
  close(fd[WRITE_END]);

  /* read from the pipe */
  read(fd[READ_END], read_msg, BUFFER_SIZE);
  printf("read %s",read_msg);

  /* close the read end of the pipe */
  close(fd[READ_END]);
}

return 0;
}
```

Figure 3.22 Figure 3.21, continued.

...the SECURITY_ATTRIBUTES structure to allow

Deliverables: source code, Makefile, written document (in plain text; at least 100 words; **NO MS-word or PDF format!)**

DO NOT include any *.o or executable files.

[5] (8 points) Create a subdirectory called "4". Change to the subdirectory. Type in message_send.c and message_rec.c by using the two source code on https://users.cs.cf.ac.uk/Dave.Marshall/C/node25.html. Compile, run, and test two programs. Note that the two programs need to run concurrently in two separate terminal windows "as demonstrated in class".

Explain in detail how message-related APIs work.

Deliverables: source code, Makefile, written document (in plain text; at least 100 words; **NO MS-word or PDF format!)**

DO NOT include any *.o or executable files.