

Assignment 3

Handed Out: October 28, 1999 Due: November 19, 1999

File Copy Via Inter-Process Communication (50 Marks)

In this assignment we will be exploring different methods of inter-process communication (IPC). Specifically, we will look at the following methods:

- Signals
- Semaphores
- Pipes
- UNIX Domain Sockets

The first two methods are used for synchronization of access to a shared resource. With the first two methods we will use shared memory to facilitate the passing of information between the processes. The remaining two methods manage synchronization and information flow in a single mechanism.

A classic operating system problem is that of coordinating a producer and consumer process. This assignment will involve a specific instance of the producer/consumer problem called the *bounded buffer* problem, with a buffer size of one block (see below). The producer *produces* a block and stores the value in a common buffer that can *only hold one block*. The consumer obtains the value from the storage location and *consumes* it. To guarantee integrity, each value *produced* must be *consumed* (not lost via overwriting by a speedy producer with a slow consumer) and no value should be *consumed* twice (such as when a consumer is faster than the producer).

For this assignment you will write a single program, in four steps.

The Program Synopsis

Your program will be named "**ipcopy**" (short for Inter-Process Copy). The program is called as follows:

```
ipcopy <source> <destination>
```

where both <source> and <destination> are mandatory parameters; each is a pathname (absolute, relative or filename are all valid). You are only to copy regular files, so neither parameter may be a directory or special file. All files are to be handled as binary files, i.e. do not treat \n or \r as special characters, and be aware that each byte in the file may take on any value in the range 0...255. If the named destination file already exists, overwrite it (if possible); otherwise create a new file with the desired name. Your program does all appropriate error checking.

Part A – Synchronization using Semaphores, Data Communication via Shared Memory

Set up a shared memory buffer that both child and parent processes can access, and use it to pass blocks between the producer and consumer. You must ensure that data doesn't get over-written by the producer before the consumer reads it, and that the consumer doesn't read the same block twice. To achieve this, create a semaphore which protects access to the shared memory, and use the "mode" member of Block to indicate that the buffer is full (*i.e.* the consumer should read it) or empty (*i.e.* that the producer should write it). If the producer wants to write to the buffer, it first gains access via the semaphore. If "mode" is empty, then the producer writes its data, sets mode to full, and releases the semaphore. If mode is full, then the producer releases the semaphore and sleeps for some random amount of time before trying again. If the consumer wants to read the buffer, it gains access via the semaphore. If mode is full, then the consumer reads the data, sets mode to empty, and releases the semaphore. If mode is empty, the consumer releases the semaphore and sleeps a random amount of time. For sleeping, use `usleep()` with a parameter in the range of 100 to 10000. Enclose all code specific to Part A within `"#ifdef SEMAPHORE"` blocks.

Part B – Synchronization using Signals, Data Communication via Shared Memory

As in Part A, set up a shared memory buffer. This time there is no mode member in the block structure. Instead, set up appropriate signal handlers so the producer and consumer can signal each other when it is their turn. For example, after the producer finishes writing the buffer, it should signal the consumer that the buffer can be read. It then waits for a signal from the consumer that the buffer has been read. The consumer reads the buffer when it receives a signal from the producer, and signals the producer when the buffer has been read. Use `SIGUSR1` for both signals. You may find it useful to set up the producer's signal handler before the fork, and have the consumer signal the producer when it has set up its signal handler and is ready to start. Enclose all code specific to Part B within `"#ifdef SIGNAL"` blocks.

Part C – Data Communication via Pipes

For this part you do not need shared memory, as the pipe itself conducts the data. The producer should use `write()` to feed blocks into the pipe, and the consumer should use `read()` to read blocks from the pipe. Enclose all code specific to Part C within `"#ifdef PIPE"` blocks.

Hint: You may find it easiest to do this part first!

Part D – Data Communication via UNIX Domain Sockets

For this part, create a UNIX Domain socket connection between the producer and the consumer. The producer should be the server. Since the consumer (child) may run before the producer (parent) is completely set up, the consumer should retry the socket connection at one-second intervals until the connection succeeds, or a maximum number of attempts (10) has elapsed. The producer then writes blocks into the socket, and the consumer reads data from the socket. Enclose all code specific to Part D within `"#ifdef SOCKET"` blocks.

Hint: You may find it easiest to do this part second!

Program Design

The program begins by initializing any necessary data structures, then calling `fork()` to create a child process. The parent will be the "producer", and the child the "consumer". The producer opens the source file, reads it one block at a time (see below for definition of a block), and transfers it to the consumer via the appropriate IPC channel. The consumer receives data from the producer one block at a time, and writes it to the destination file.

Data Structures

Your program will copy the file one "block" at a time. You must use the following data structures/definitions:

```
#define BLOCK_SIZE 2048
#ifdef SEMAPHORE
typedef enum (AccessEmpty, AccessFull) AccessMode ;
#endif
typedef struct {
    unsigned char data[BLOCK_SIZE] ;
#ifdef SEMAPHORE
    AccessMode    mode                ;
#endif
    unsigned long size                ;
} Block ;
```

This data structure is designed to facilitate transfer of data between the parent and child processes. Your program should create a variable of this type, either as a local variable or by allocating memory via `malloc()`. The data from the file being copied is stored in the "data" member of the Block structure. The member variable "size" will always be 2048, except perhaps when transferring the last block of the file. The consumer, upon receiving a block with `size < BLOCK_SIZE` will consider this the last block and terminate.

Use of the "mode" member of Block

The "mode" member exists only when using semaphores to allow the producer to notify the consumer when new data is ready for reading, and allows the consumer to signal to the producer when the data has been read. It works like this:

- Upon program initialization, it should be set to `AccessEmpty` to signal that there is no data in the buffer
- After data is written to the buffer (only done if `mode == AccessEmpty`), the producer sets `mode` to `AccessFull`.
- After reading from the buffer (only done if `mode == AccessFull`), the consumer sets `mode = AccessEmpty`.

Version Specific Data Parameters

```
typedef struct {
```

```

#ifdef SEMAPHORE && SIGNAL
    /* put any declarations here you need for the semaphore & signal versions */
#endif
#ifdef SEMAPHORE
    /* put any declarations here you need for the semaphore version */
#endif
#ifdef SIGNAL
    /* put any declarations here you need for the signal version */
#endif
#ifdef PIPE
    /* put any declarations here you need for the pipe version */
#endif
#ifdef SOCKET
    /* put any declarations here you need for the sockets version */
#endif
} Parms ;

```

Use this structure to define any version specific parameters you need. For example, for the pipe version, you will place any file descriptors you require here. The first section is for any parameters which are used by both the signals and semaphores version (i.e. any shared memory parameters).

Special Functions

Your program will declare and implement the following functions:

```

void InitIPCpre(Parms *p);
void InitIPCpost(Parms *p, int parent);
void SendBlock(Block *theBlock, Parms *p);
void ReadBlock(Block *theBlock, Parms *p);
void CleanupIPC(Parms *p, int parent);

```

You should use these functions to hide the specifics of your IPC method. This means that most (all?) of your conditional compilation directives will be in these functions. The function `InitIPCpre()` is to be used for any initialization which needs to be done before `fork()`, and `InitIPCpost()` for any which needs to be done after `fork()`. Note the latter has a Boolean parameter to specify whether the parent or child is running it. You may not change their definitions. A simple **pseudo-code** version of the program is as follows:

```

int main(int argc, char *argv)
{
    Parms myParms ;
    Block myBlock ;
    int    pid      ;

    process command line parameters ...

    InitIPC(&myParms);
    if ((pid = fork()) == 0) { /* child/consumer */
        InitIPCpost(&myParms, False);
        while (not LastBlock)

```

```

    {
        ReadBlock(&myBlock, &myParms);
        write destination file ...
    }
} else { /* parent/producer */
    InitIPCpost(&myParms, True);
    while (not eof(source))
    {
        read from source file ...
        SendBlock(&myBlock, &myParms);
    }
}
CleanupIPC(&myparms, pid); /* both parent and child do this */
}

```

Use of Global Variables

You will refrain from using any global variables except for variables used by signal handlers to communicate with ReadBlock() or SendBlock(). Marks will be deducted for any unnecessary use of a global variable. You are free of course to use local static variables in functions.

Conditional Compilation

You will write one set of code which can be compiled into four different programs by using conditional compilation. The program is compiled as follows:

Part A	<code>gcc -DSEMAPHORE</code>	<code>ipcopy.c -o ipcopy</code>
Part B	<code>gcc -DSIGNAL</code>	<code>ipcopy.c -o ipcopy</code>
Part C	<code>gcc -DPIPE</code>	<code>ipcopy.c -o ipcopy</code>
Part D	<code>gcc -DSOCKET -lsocket</code>	<code>ipcopy.c -o ipcopy</code>

As always, you should hand in a printed version of your program, as well as submitting it electronically on CDF using "**submit -N a3 csc209h ipcopy.c**". You can overwrite a previous submission by adding the "**-f**" switch to the submit command. No external documentation is required, but your program should be well documented.

If you wish to use multiple source files you may, but you must 1) include a makefile, and 2) use tar to create a single file, `ipcopy.tar`, containing all your source files plus the makefile. Your makefile must have four targets named "semaphore", "signal", "pipe" and "socket" which make Parts A—D respectively.

The assignment will be marked with 10 marks for each of Parts A—D, with a further 10 marks for the code common to all parts. The marks will be allocated 30% for style, and 70% for correct operation.