CSC 4730 Fall 2023 Project 7

You will write a "named pipe" that shares data between two processes. Call these "parent" and "child", both of which you will write. The project employs AT&T Unix shared memory, Unix signals and named semaphores.

Objectives

As a result of this project, you will have:

- Designed and implemented a named pipe - a classic producer / consumer data structure.
- Written synchronization software employing named semaphores.
- Written software the responds to Unix signals.
- Written software that uses AT&T Unix shared memory.

Deadlines

You and your assigned partner have 7 days to complete this assignment plus one grace day. If you have not turned anything in by 11:59 PM on the eighth day, you will win a grade of zero. Therefore, to receive partial credit, hand in something before the project expires!

Overview

A pipe is an example of a producer / consumer data structure. It has finite size but infinite capacity. That is, an infinite amount of data can pass through the pipe despite the fact that its implementation uses only a small amount of memory.

You have used standard pipes in the shell and in Project 1 in which you wrote a program that makes use of them. In this case, you will be writing your own.

A named pipe differs from the pipes you have used before in that those were anonymous. There is no name to the pipe in `"ls | wc"`. A named pipe is a name that exists, even if only temporarily, on a file system so that two completely unrelated tasks can find it and connect to each other.

The name is used as a place to rendezvous. Like "meet me at such and such a place."

Named shared memory (the memory in which the pipe and synchronization controls will be implemented) along with named semaphores live on the file system but will not be visible with commands like "ls". This is of no consequence.

Being a producer / consumer, the pipe must be guarded and controlled by a pair of semaphores. Read and internalize chapter 31 of OSTEP. This chapter provides

a number of broken implementations followed by a working implementation of a producer / consumer using semaphores.

The semaphores in this project are named semaphores which are much harder to work with than anonymous semaphores. We cannot use anonymous semaphores completely and entirely because of Apple who decided to remove them from the OS.

Because the shared memory is a resource that must be open and closed explicitly, you will implement Unix signals handlers to catch the `SIGINT` signal (which corresponds to ^C) to ensure that your program will end gracefully.

Steps in the implementation of the parent include but are not limited to:

- The parent creates the named pipe which includes:
  - creating and attaching to a named shared memory region
  - creating and initializing named semaphores
- Launching the child which accesses the above.
- Entering a loop in the parent which pumps data into the pipe as fast as it can.
- Entering a loop in the child which **slowly** reads from the pipe.
- when a `SIGINT` (^C) is received by the parent, it sends a `SIGINT` to the child and both programs shut down gracefully

Includes

Be prepared to include a great many header files. You are specifically *not* being told which includes to use because no one will give you this information in the real world. You must learn to figure these things out on your own.

Suggested steps

These steps are suggestions. You are free to deviate as you see fit.

Step 1 - signals Create a program that does nothing except wait for `SIGINT`. This code can then be leveraged by both the parent and the child.

- You'll need a `struct sigaction`.
- You'll need to initialize it properly.
- You'll need to call `sigaction()`. Notice that the name of the `struct` is the same as the name of the function.

Then write a handler for the signal:

```
void my_handler(int s)
{
    printf("<WHICH PROGRAM HERE> caught signal 0x%x\n",s);
    keep_going = false;
}
```

When the signal (in this case caused by ^C) is received, the `keep_going` flag turns false which should cause your main loop to terminate.

Step 2 - shared memory and pipe initialization Next, I added the code to create a shared memory region. I did this using a C++ class and implemented it in a separate include and code file so that the code could be shared between the parent and child.

Here is the class I created, you can choose something different:

```
#pragma once
/* Many includes go here */

#define SHMEM_NAME  "p7"

class SharedMem {
public:
    SharedMem(std::string name);

    int OpenSharedMem(bool truncate = false);
    void CloseSharedMem();
    void *Map();
    void UnMap();

    void *s_ptr;
    int fd;
    std::string name;
};
```

In my implementation:

- the constructor merely sets some state including the canary values.
- `OpenSharedMem()` opens the shared memory region. The parent will truncate the shared memory. The child will not.
- `CloseShMem()` gracefully shuts down the shared memory region. It also checks the canaries to ensure no buffer overrun or underrun took place. If over or underruns occurred, it will print a message.
- `Map()` arranges for the shared memory region to appear within the address space of the process. Think of this as the `alloc()`. The shared memory must be "opened" first.

- `UnMap()` as the name implies, is like `free()`.

Apple does not support pthreads' own semaphores for reasons that are known only to them. Instead you must use named semaphores via `sem_open()` and `sem_closed()`. Initializing these and taking them down is the job of the parent. The child just needs to open, use and close them.

WARNING WARNING WARNING

The named semaphores and named shared memory regions are managed by the kernel. If you do not initialize them correct and especially do not clean them up correctly, you will have to reboot your machine (Mac) or restart WSL (Windows).

In developing this project, I myself had to reboot my Mac about a dozen times and I had a clue about what I was doing.

Environment

You must use WSL on Windows or the Terminal on the Mac.

Provided makefile

If you create main source code files named parent.cpp and child.cpp, you could be able to leverage the included makefile. Then it is easy to build your programs like so:

```
$ make all
```

There is also `make clean` to rebuild everything.

What to Hand In

Execute a `make clean`. Then zip the entire xv6 folder. Test the zip file to ensure you have done this correctly. Hand in the zip file.

Partners

- You must use only the partner I assign to you.
- Only 1 person should hand in code. The code should clearly state who the partners are in a new text file `partner.txt` in the main directory.
- The non-code-submitting partner must submit a text file "partner.txt" that states who the partners are.
- **Failure to list partners correctly as described above removes 5 points from your grade.**

"Stupid Professor, this seems like a stupid and unnecessary requirement that I will ignore." Someday you will join a team. The team may have requirements that seem stupid to you. Violation of the requirement can result in a poor performance review.

No raise for you.

Or, "They're (i.e. you are) just not fitting into our team culture."

No job for you.

Grading

The class will receive a rubric when grading is complete that describes errors and penalties.

Both partners get the same grade without exception.

Comments about the Parent

The parent:

- Does not use a random sleep period - it emits data into the pipe as quickly as it can. Correct implementation of the named pipe will cause the parent to sleep when the pipe is full and be reawakened when there is room in the pipe. It will have no knowledge of when or often this happens.
- Is responsible for truncating the shared memory region. The child does not truncate.
- as part of exiting, sends a `SIGINT` to the child using the system call `kill`.
- sends the string: "Line: 554 ABCDEFGHIJKLMNOPQRSTUVWXYZ\n" where the 554 is from a counter of lines written to the pipe.
- prints "p_index: 38 Parent wrote: Line: 554 ABCDEFGHIJKLMNOPQRSTUVWXYZ\n"
  - `p_index:` followed by a number tells you where the head pointer is in the circular buffer. This number should never equal the size of the pipe.
  - `Parent wrote: ...` is what the parent wrote (see above).
- Must completely take down the named pipe and its semaphores and the shared memory region. Failure to do this right means a reboot is in your near future.

Comments about the Child

The child:

- Uses random numbers. Between each full line received (as signified by seeing a newline), the child will sleep for a minimum of 2500 microseconds and a maximum of 125000 microseconds. This means the child will be slower at draining the pipe than the parent is at filling the pipe.
- Does not initialize the semaphores in the named pipe because this is the job of the parent.
- Closes the shared memory region but does not clean up the named pipe as this is the job of the parent.

Comments about the Named Semaphores

The functions you must use include:

- `sem_open()`
- `sem_close()`
- `sem_unlink()`
- `sem_wait()`
- `sem_post()`

If all is done correctly, you will not need `semctl()`.

I have deleted a hint about permissions - getting them completely wrong will result in a deduction.

Messing up named semaphores will require rebooting your system since these are persistent OS managed resources.

Comments about the Shared Memory

The parent sets this up and takes it down. The child uses it.

Shared memory functions you will use include:

- `shm_open()`
- `ftruncate()`
- `shm_unlink()`
- `mmap()`
- `munmap()`

Think about how shared memory works.

In my implementation the named semaphores also live in shared memory.

Constants

- The size of the pipe must be 1024 bytes. That is, the most number of bytes the pipe can contain at one time is 1024.
- The canary must have the value 5077604. Why? Because your manager said so. Following specifications is an important skill for your future.

Output

You must implement an addition the semaphores needed to control the named pipe, you must also declare and use a named semaphore for controlling access to `cout`. This is needed to ensure that output from the parent and child do not collide. Look up how a semaphore can be used as a `mutex`.

In total I am using four named semaphores.

About the names of the named things

These are simply unique character strings such as "p7_empty_sem" which I use for the empty semaphore.

Output from my implementation

You must emit similar output.

```
 % ./parent
Shared memory is initialized.
Shared memory is now mapped at parent address: 0x10090c000
Pipe controls are initialized.
Parent has launched child.
p_index: 35 Parent wrote: Line: 1 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 70 Parent wrote: Line: 2 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 105 Parent wrote: Line: 3 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 140 Parent wrote: Line: 4 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 175 Parent wrote: Line: 5 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 210 Parent wrote: Line: 6 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 245 Parent wrote: Line: 7 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 280 Parent wrote: Line: 8 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 315 Parent wrote: Line: 9 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 351 Parent wrote: Line: 10 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 387 Parent wrote: Line: 11 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 423 Parent wrote: Line: 12 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 459 Parent wrote: Line: 13 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 495 Parent wrote: Line: 14 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 531 Parent wrote: Line: 15 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 567 Parent wrote: Line: 16 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 603 Parent wrote: Line: 17 ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
p_index: 639 Parent wrote: Line: 18 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 675 Parent wrote: Line: 19 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 711 Parent wrote: Line: 20 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 747 Parent wrote: Line: 21 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 783 Parent wrote: Line: 22 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 819 Parent wrote: Line: 23 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 855 Parent wrote: Line: 24 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 891 Parent wrote: Line: 25 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 927 Parent wrote: Line: 26 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 963 Parent wrote: Line: 27 ABCDEFGHIJKLMNOPQRSTUVWXYZ
Child attached to shared memory and named pipe.
Child has installed signal handler.
p_index: 999 Parent wrote: Line: 28 ABCDEFGHIJKLMNOPQRSTUVWXYZ
c_index: 34 Child read:   Line: 1 ABCDEFGHIJKLMNOPQRSTUVWXYZ
p_index: 11 Parent wrote: Line: 29 ABCDEFGHIJKLMNOPQRSTUVWXYZ
c_index: 69 Child read:   Line: 2 ABCDEFGHIJKLMNOPQRSTUVWXYZ
^CChild caught signal 0x2

Child heading to exit.
Parent caught signal 0x2
NamedPipe::Producer() - sem_wait pipe_lock: Interrupted system call
Child has been sent a SIGINT signal
Pipe controls closed.
Shared memory has been taken down.
perrykivolowitz@DAEDALUS pk_named_pipe_v2 %
```

Notice the parent pumped out a ton before the child even began to run. Thereafter, there was a close synchronization between parent and child.

Interrupted system calls

Signals can interrupt system calls. Look through the sample above.

Notice:

```
NamedPipe::Producer() - sem_wait pipe_lock: Interrupted system call
```

A `sem_wait()` is supposed to wait forever but it was interrupted.

You must check the return value of every semaphore function, indeed every system call. Use `man` to look up correct returns versus error return values.