CS 503 Spring 2014

Lab 2: Interprocess Communication Using Bounded Buffers In Xinu

Due: Tuesday 03/04/14, 11:59 PM

1. Objectives

The objective of this lab is to become familiar with interprocess communication and the need for process synchronization. Specifically, you will work with a pipe abstraction used to transfer data between processes. The first part of the lab requires you to create a new pipe service in Xinu; the second part requires you to implement a corresponding program using the Xinu pipe facility.

---

2. Readings

1. UNIX Inter-Process Communication here
2. Chapters 6, 7, 8 in the XINU book (Process Management, Coordination of Current Processes and Message Passing).

---

3. Interprocess Communication in Unix

3.1 Unix pipes

In Unix, a pipe is a unidirectional communication channel between two processes (i.e., the pipe serves as a temporary buffer that allows one process to deposit data in the pipe and another process to extract data from the pipe. Data stored in a pipe is always trnasferred sequentially (first-in-first-out). A sender uses write() to deposit data to a pipe, and a receiver uses read() to extract data from a pipe. The following Unix system calls are used with pipes: `pipe(), read(), write(), and close().`

For a quick reference on Unix pipes, you may refer here.
If you are not familiar with pipes, it may be a good idea to write a few test programs that use UNIX pipes before attempting to create the pipe facility for Xinu. Any test programs you write are for your own edification and will not be graded.

3.2 Readers and writers (sample problem - not graded)

Write a random word `generator` program which outputs a sequence of words to a UNIX pipe (have it make a random selection of words from Dictionary (right click to download). Arrange a `reader` process at the other end of the pipe to reads words from the pipe, count the number of words that start with each vowel, and otherwise discard the words. As an extra challenge, arrange for the generator to communicate with a `controller` process periodically (every 1000 words). Have the generator list each

vowel with a count of the words that have been generated which start with the vowel, and have the controller provide feedback that specifies new probabilities (i.e., if the count of words beginning with "a" is high, have the controller lower the probability of selecting words that begin with "a".

---

4. Adding pipes to Xinu (50%)

4.1 Overview

Download a new version of XINU
In the first part of the lab, you will implement pipe primitives for interprocess communication in Xinu. Your mechanism will allow processes to open a maximum of `NPIPE` (set to 10 by default) pipes. A process can create a pipe and pass the ID of the pipe to another process; the two processes can then use the pipe to send a receiver a stream of data -- one process sends data and another receives it. To implement you pipe facility you must add a set of system calls to Xinu.
To operate with pipe primitives, one should follow the sequence:

1. Create a pipe
2. Connect the processes that need to communicate with the pipe
3. Perform communication through the pipe
4. Release (i.e., terminate) the pipe

A pipe between two processes should be constructed as a classical bounded-buffer. One process (called a `producer`) fills the buffer with data, and the other process (called a `consumer`) empties the buffer. Each pipe has a fixed-size buffer of `PIPE_SIZ` bytes (256 bytes by default).

4.2 Design

Your pipe facility must insure that the producer cannot add data to a pipe if the buffer is full, and the consumer cannot remove data from a pipe if the pipe is empty. We strongly recommend that you use semaphores to control access to the pipe (two semaphores are necessary and sufficient).
A pipe in Xinu is uniquely identified by the pipe ID.

A pipe can be in one of the following states:

`PIPE_FREE`: A pipe is in the "free" state when pipe has not been allocated by a process. Data can neither be read nor written to a free pipe. All the pipes are in the "free" state when the system starts.

`PIPE_USED`: A pipe has been created by using the `pipcreate()` function (see below) but no process has yet connected to either end of the pipe.

`PIPE_CONNECTED`: Two processes have connected to the pipe by calling `pipconnect()` (see below).

`PIPE_OTHER`: You can design other state(s) for your own pipe to help you implement our requests.

Remember that a Xinu pipe is a means of communication between only two processes - producer process and a consumer process.

4.3 System Calls

You are required to implement the following set of system calls (Refer to `prototype.h` for adding new system calls to Xinu).

1. syscall pipcreate()

This system call should be implemented in a file `pipcreate.c` in `xinu-14spring-linksys/system` directory. A process (not a producer process nor consumer process) creates a pipe using this system call, which returns the pipe ID `pipid32` if the process is able to find an available pipe from system and setup the appropriate state variables. If success, it puts the pipe in "USED" state. You need to think about the appropriate state variables required for a pipe, and also about where this state needs to be stored. The system call returns `SYSERR` if the process is unable to allocate a pipe because the all pipes are not available.

2. syscall pipdelete(pipid32 pip)

This system call should be implemented in a file `pipdelete.c` in directory `xinu-14spring-linksys/system` . The `pipdelete()` system call is used release a pipe; the call a pipe ID as an argument. It should set the state of the pipe to "free" and return `OK`. After the call, the pipe should be ready for reallocation (i.e. the call should clear the buffer and reset associated variables).

3. syscall pipconnect(pipid32 pip, pid32 writer, pid32 reader)

This system call should be implemented in a file `pipconnect.c` in directory `xinu-14spring-linksys/system`. The `pipconnect()` system call is used to connect two processes to an existing available pipe. It takes as its arguments a pipe ID, a process ID for writer, and a process ID for reader. The system call returns `OK` if the pipe is successful connected with two processes. In any case, if the arguments are invalid, the pipe was already connected to other processes, or the state of the pipe does not permit connection, the system call returns `SYSERR`.

4. syscall pipdisconnect(pipid32 pip)

This system call should be implemented in a file `pipdisconnect.c` in directory `xinu-14spring-linksys/system`. This system call is used to disconnect two processes from a specified pipe. The call returns `OK`, if the pipe is empty. Otherwise, it would prevent further coming data to prepare another **`pipdisconnect`** call, and return `SYSERR`.

5. syscall pipwrite(pipid32 pip, char *buf, uint32 len)

This system call should be implemented in a file `pipwrite.c` in directory `xinu-14spring-linksys/system`. This system call is used by a process to write data to the bounded-buffer of an connected pipe. It takes as arguments the pipe ID, a pointer to the buffer from which data is to be written and the length of data to be written. The call returns `SYSERR` if the pipe is not in connected state, or this process is not writer. Otherwise, the call returns the number of bytes written to the pipe. If the bounded-buffer is full and there is no room for additional data to be written, the calling process needs is blocked. (Hint: use Xinu's semaphore primitives). If there is space for data to be written, the system call copies the data from buf to the pipe (bounded-buffer), else the process should wait for another writing opportunity. If the writer could not successfully write all data in buf, the system call return `SYSERR`.

6. syscall piprea(pipid32 pip, char *buf, uint32 len)

This system call should be implemented in a file `pipread.c` in directory `xinu-14spring-linksys/system`. This system call is used by a process to read data from a connected pipe specified by the pipe ID `pipid32`. The system calls reads up to `len` bytes from the connected pipe and saves it in `buf`. The system call returns the actual number of bytes read or `SYSERR`, if the read is unsuccessful. The calling process blocks if there is no data to be read from a connected pipe.

Implementation Hints

One can declare the buffer and space for other variables by defining global variables. As an alternative, one can use getmem function to allocate the buffer and other state variables for the pipe dynamically. If dynamic allocation is used, your implementation must place limits on the number of pipes a give process can allocate. If static allocation is used, the system can pre-allocate an array of pipe structures.

To use the semaphore mechanism in Xinu to enforce synchronization between a producer and consumer processes. you should look at functions `semcreate, semdelete, signal, signaln, wait, semcount and semreset`. Refer to the textbook for the details about the calls.

Remember that you are actually designing parts of an operating system. You should implement **any reasonable condition check** within your code to make sure your system won't block if some one happens to use your API in an impertinent way. Do not expect too many details from here!

---

5. More details (special requests) for your pipe mechanism (30%)

This part maybe modified according to students' feedback. **Please check this part when every time you are confused, thanks. If you could not get answer, ask it on Piazza.**

`pipcreate():`

The process who creates a pipe is the owner of this new pipe.

`pipconnect(pipid32 pip, pid32 writer, pid32 reader):`

Writer process and reader one should not be same.

`pipwrite(pipid32 pip, char *buf, uint32 len):`

The writer process should write data into the pipe continually, until it finishes its job. Exceptions are illustrated in `pipdisconnect` and `pipdelete`.

`pipread(pipid32 pip, char *buf, uint32 len):`

TBD.

`pipdelete(pipid32 pip):`

Only owner of the pipe can call this system call (see `pipcreate()` to learn more about owner).

When owner deletes the pipe, writer and reader processes should be terminated and the pipe would be clean up (initialized).

If a pipe owner process is killed, its pipe(s) should be deleted.

`pipdisconnect(pipid32 pip):`

Only reader or writer can call this system call.

When one side is disconnected from the pipe (call this function), the other one should disconnect from the pipe also and clean the pipe. (This pipe can be still reconnected)

If any side process is killed, both of processes should be disconnected from the pipe.
Every time a reader/writer wants to read/write the pipe, make sure whether it is possible.

---

6. Adding Shell test commands (20%)

6.1 Shell commands

Add two shell commands `gen` and `search`. `gen` produces words from a set. Note: that the XINU file system is not exposed in the version of XINU provided, so it would be a good idea to use a wordlist.h (right click to download) file for generating words. `gen` is used to dump words on one end of a pipe and `search` will be used to search words beginning with vowels and maintain word counts for each vowel.

6.2 Usage

```
xsh $ gen | search
```

`gen` should periodically (after 5 secs) print the number of words it generated
`search` should periodically (after 10 secs) print the word count for each vowel

Individually
```
xsh $ gen
xsh $ search
```
Should say invalid use of the program.

`gen` and `search` should be listed as new commands supported by the XINU shell.
Note that `gen` and `search` those two commands (processes) should run together, you need to create a pipe for the data transition.

Implementation hints

The shell/ directory within the implementation of XINU provides the commands for current shell implementation. `shell.c lexan.c` is a good place to start looking on how to add commands. Consider | as a token and create reader writer processes on encountering the symbol. A good understanding of `shell.c` would be required to proceed. The rand() and srand() primitives are present, again an understanding of the code is required for usage.

Turn-in Instructions

Electronic turn-in instructions:

1. Go to `xinu-14spring-linksys/compile` directory and do `make clean`.
2. Go to the directory of which your `xinu-14spring-linksys` directory is a subdirectory (NOTE: please do not rename `xinu-14spring-linksys`, or any of its subdirectories.) E.g., if `/homes/abc/xinu-14spring-linksys` is your directory structure, go to `/homes/abc`
3. Type in the following command: **`turnin -c cs503 -p lab2 xinu-14spring-linksys`**

Notes

1. Please start with a fresh copy of `xinu-14spring-linksys` codebase
   `tar zxvf /u/u3/503/xinu-14spring-linksys.tar.gz`
2. ALL debugging output should be turned off before you submit your code.
3. No system call functionality should be provided in `xinu-14spring-linksys/system/main.c` file. It will be replaced with a different `main.c` during testing. But your code for part 5 should be included in `main.c`
4. Please place a short README describing what you have done in your submitted tarball.
5. Submissions with only partial output should contain a **detailed** README of what works, and design decisions adopted.

Back to the CS 503 web page