Lab 2: Interprocess Communication Using Bounded Buffers In Xinu (100 pts)

- CS 503 Fall 2017
- Due: Sunday, Oct. 22 2017, 11:59 PM

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. This lab requires you to create a new pipe abstraction which can be used to create a commincation layer between processes from within the XINU shell.

Readings

- UNIX Inter-Process Communication here
- Chapters 6, 7, 8 in the XINU book (Process Management, Coordination of Current Processes and Message Passing).
- Chapters 14, 15 in the XINU book (Device Independent I/O).

Interprocess Communication in UNIX

In UNIX, a pipe is a uni-directional (one-way) 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 transferred sequentially (First-In-First-Out). A sender uses `write()` deposit data to a pipe, and a receiver uses `read()` to extract data from a pipe.

Examples of UNIX system calls used with pipes are: `pipe()`, `read()`, `write()`, and `close()`.

Device Independent I/O

The hardware installed on your computer forms your devices and the OS communicates with these devices using device drivers.

In order to faciliate communication, most OS implementations allow for a single interface for these devices to communicate with processes. This simple interface hides the complex functionality and hardware specific details from user processes.

The simple abstraction is similar to filesystem API and it uses simple calls such as read(), write(), getc() and putc(). For example, “tty” is a simple device which provides shell I/O functionality.

Pipe-based Communication within Xinu

Overview

First, download a new version of XINU, which supports both Galieo and QEMU. Please do not use your old copy from previous assignment(s). Please use the following command to download:

```
tar zxvf /u/u3/cs503/xinu-fall17/xinu-x86-qemu.tar.gz
```

In this lab, you will implement pipe primitives for interprocess communication in XINU. We have pre-allocated 10 devices (PIPELINE0, PIPELINE1, …, PIPELINE9) to be used for pipe communication. Each created pipe will be bound to one of the above mentioned devices.

The pipe mechanism will be developed in a similar fashion as it exists in the UNIX shell. The XINU shell (xsh) will create pipes between created processes as demanded by the user. Consider a simple example below.

```
xsh $ gen | count
```

In this example, the XINU shell creates two processes (a) `gen` and (b) `count` and connects them through a pipe such that `gen` sends data and `count` receives it. A pipe between two processes should be constructed as a classical bounded-buffer. One process (called a producer/writer) fills the buffer with data, and the other process (called a consumer/reader) empties the buffer. To facilitate the connection between two processes, the Xinu shell may need to do the following: (1) replace `stdout` of `gen` into a pipeline producer and (2) replace `stdin` of `count` into a pipeline consumer. Note, in Xinu's context, `stdin` refers to `prdesc[0]` and `stdout` refers to `prdesc[1]` in `struct procent`.

To implement the pipeline mechanism, you will need to add a set of system calls to Xinu. To operate with pipe primitives, one should follow the sequence:

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

Each pipe has a fixed-size buffer of bytes (1024 bytes by default).

Design Workflow

First, you must scan the input entered by the user on XINU shell to ascertain whether the entered string requires a pipe or not. All primitives to do so have already been provided in `lexan.c` and `shell.c`.

If a pipe is required, the shell process should create the pipe using `pipcreate` and connect the processes using `pipconnect`. The details of these system calls have been provided in the next section.

Then the shell may need to replace stdin and stdout. Simple pseudo code is provided below:

```
// These all may need to be done by shell()

// 1. create a pipe
did32 devpipe = pipcreate();

// 2. connect a pipe
pipconnect(devpipe, writer_pid, reader_pid);

// 3. replace stdout of the writer process
proctab[writer_pid].prdesc[1] = devpipe;

// 4. replace stdin of the reader process
proctab[reader_pid].prdesc[0] = devpipe;
```

You can create yet another system call, which accesses `proctab` and replaces its `prdesc` (i.e., similar to what `dup2()` does in Linux). For this assignment, however, it is OK to simply access them from shell(). You can think about why such a direct access is not an elegant solution and thus should be avoided in Xinu.

As stdin and stdout are now replaced with a pipeline device (i.e., one of pipeline devices, `PIPELINE0`, `PIPELINE1`,…, and `PIPELINE9`) all following I/O will be redirected pipe I/O calls: `pipread` for `read`, `pipwrite` for `write`, `pipgetc` for `getc`, and `pipputc` for `putc`. More details can be found in the next section.

Your pipe facility must ensure that the producer (or writer) cannot add data to a pipe if the buffer is full, and the consumer (or reader) cannot remove data from the pipe if it 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 (i.e., `pipid32`) and it 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 XINU pipe is a means of communication between only two processes - producer (writer) process and a consumer (reader) process.

System Calls

You are required to implement the following set of calls, i.e., `pipcreate, pipconnect, pipdisconnect, pipread, pipwrite, pipgetc, pipputc`. Amongst these calls, `pipcreate, pipconnect, pipdisconnect and pipdelete` are additional system calls called by user processes.

Each of the PIPELINE device has been configured to call `pipread` on a call to `read(dev-id, ...)`. Similarly, it calls `pipwrite` on a call to `write(dev-id, ...)`, `pipgetc` on a call to `getc(dev-id, ...)`, and `pipputc` on a call to `putc(dev-id, ...)`. Please refer to `/config/Configuration` as well as its dynamically generated version `/config/conf.c`.

- syscall `did32 pipcreate()`:

  This system call should be implemented in `xinu-fall2017/system/pipcreate.c`.

  A process creates a pipe using this system call, binds it to one of the available devices (PIPELINE0 to PIPELINE9) and returns the device ID `did32` if the process is able to find available device from system. In addition, it has to setup the appropriate state variables and semaphores for the pipe.

  If successful, it puts the pipe in PIPE_USED state. You need to think about the appropriate state variables required for a pipe, and also about where these states need to be stored.

  The system call returns SYSERR if the process is unable to allocate a pipe because all the pipes are unavailable.

- syscall `status pipdelete(did32 devpip)`

This system call should be implemented in `xinu-fall2017/system/pipdelete.c`.

This system call is used to release a pipe; the call takes device ID as an argument. It should set the state of the pipe to PIPE_FREE and return OK.

After the call, pipe should be ready for reallocation (i.e. the call should clear the buffer and reset associated variables).

- syscall `status pipconnect(did32 devpipe, pid32 writer, pid32 reader)`

  This system call should be implemented in `xinu-fall2017/system/pipconnect.c`.

  This system call is used to connect two processes to a previously created pipe. It takes as its arguments a device ID, a process ID for the writer, and a process ID for the reader.

  The system call returns OK if the pipe is successfully 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.

- syscall `status pipdisconnect(did32 devpipe)`

  This system call should be implemented in directory `xinu-fall2017/system/pipdisconnect.c`.

  This system call is used to disconnect two processes from a specified pipe.

  The call returns OK, if the pipe is in reasonable state. Otherwise, it would and return SYSERR.

- devcall `devcall pipputc(struct dentry *devptr, char ch)`

  This device call should be implemented in `xinu-fall2017/system/pipputc.c`.

  This call writes the single character onto the buffer associated with this pipe from the character `ch`.

The call returns 1 if successful or SYSERR if unsuccessful. The calling process blocks in case no space.

- devcall `uint32 pipwrite(struct dentry *devptr, char *buf, uint32 len)`

  This device call should be implemented in `xinu-fall2017/system/pipwrite.c`.

  This system call is used by a process to write data to the bounded-buffer of connected pipe. It takes as arguments the device 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 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 to be blocked. (Hint: use Xinu's semaphore primitives).

  If there is space for data to be written, the system call copies the data from buffer (buf) to the pipe, else the process should wait for another writing opportunity. If the writer could not successfully write all data in buffer (buf), the system call return SYSERR.

  As can be observed, you can reuse your implementation of `pipputc`.

- devcall `devcall pipgetc(struct dentry *devptr)`

  This device call should be implemented in `xinu-fall2017/system/pipgetc.c`.

  This call reads a single character from the buffer associated with this pipe and returns a character.

  The call returns SYSERR if unsuccessful. The calling process blocks in case there is no character to read.

- devcall `uint32 piperead(struct dentry *devptr, char *buf, uint32 len)`

This device call should be implemented in `xinu-fall2017/system/piperead.c`.

This call reads data from a connected pipe specified by the device ID. The system call 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.

As can be observed, you can reuse the implementation from `pipgetc`.

Implementation Hints

The shell/ directory within the implementation of XINU provides the commands for current shell implementation. `shell.c` and `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.

To understand the primitives used by the processes to send and receive messages, have a look at `/shell/xsh_gen.c` (writer) and `/shell/xsh_count.c` (reader). (NOTE: You should not make any modification to `xsh_gen.c` and `xsh_count.c`.)

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 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 for enforcing synchronization between producer (or writer) and consumer (or reader) processes, you should look at functions `semcreate, semdelete, signal, signaln, wait, semcount and semreset`. Refer to the textbook for details about the calls.

You may wonder the relationship between devide ID (`did32`) and pipeline ID (i.e., `pipid32`). Pipeline ID is an internal index used to access internal pipline data structures, and it is never exposed to any user application (including shell, reader, and writer processes). On the contrary, device ID is the same as `dvnum` stored in `struct dentry` , and it can be handed over to a user application. For this assignment, the shell process may obtain device ID and use it to replace stdin and stdout, as shown in the previous pseudo code.

To clarify your understanding, below shows the code snippet of a reference implementation, which convers between device ID and pipeline ID. In fact, if you understand how `conf.h` is generated from `Configuration` , this may not be even necessary — `dvminor` of `struct dentry` can be used for `pipid32` .

```
pipid32 did32_to_pipid32(did32 devpipe) {
    ASSERT(devpipe >= PIPELINE0 && devpipe <= PIPELINE9);
    return devpipe - PIPELINE0;
}

did32 pipid32_to_did32(pipid32 pip) {
    ASSERT(pip >= 0 && pip <= 9);
    return PIPELINE0 + pip;
}
```

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.

More details for pipe mechanism

- `pipcreate()` : The process who creates a pipe is the owner of this new pipe.
- `pipconnect(did32 devpipe, pid32 writer, pid32 reader)` : Writer and reader processes should not be same.
- `pipwrite(struct dentry *devptr, 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.

- `pipdelete(did32 devpipe)` : 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 stop reading or writing and the pipe would be cleaned up (reset). If a pipe owner process is killed, its pipe(s) should be deleted.

- `pipdisconnect(did32 devpipe)` : Only processes (reader or writer) connected to the pipes can call this system call. When one side is disconnected from the pipe (call this function), the other side should finish its job, disconnect from the pipe and clean the pipe. (This pipe can be still reconnected) If writer calls disconnect, reader should finish its reading (until the pipe is empty) and clean up the pipe. If reader calls disconnect, writer would directly clean up the pipe. If any side process is killed, both of processes should be disconnected from the pipe.

Sample Tests

To test whether your code works properly, you can perform simple tests from within the XINU shell such as:

```
xsh $ echo a | count
count: 1 [a]
```

```
xsh $ echo a bb ccc | count
count: 1 [a]
count: 2 [bb]
count: 3 [ccc]
```

```
xsh $ echo a b c | count | count
count: 6 [count:]
count: 1 [1]
count: 3 [[a]]
count: 6 [count:]
count: 1 [1]
count: 3 [[b]]
count: 6 [count:]
count: 1 [1]
count: 3 [[c]]
```

```
xsh $ gen | count
count: 4 [save]
count: 4 [scan]
count: 7 [scanner]
count: 6 [screen]
count: 10 [screenshot]
count: 6 [script]
count: 6 [scroll]
count: 8 [security]
count: 6 [server]
count: 9 [shareware]
count: 5 [shell]
count: 5 [shift]
count: 8 [snapshot]
count: 6 [social]
count: 8 [software]
count: 4 [spam]
count: 7 [spammer]
count: 11 [spreadsheet]
count: 6 [status]
count: 7 [spyware]
count: 13 [supercomputer]
count: 4 [surf]
count: 6 [syntax]
count: 6 [backup]
count: 9 [bandwidth]
count: 6 [binary]
count: 3 [bit]
count: 6 [bitmap]
count: 4 [bite]
count: 4 [blog]
```

Changes in Reference Implementations

Provided below is a reference of the number of changes required and the files in which the changes are required to complete this assignment. Please treat this as a reference and you are not bound to follow this but your functionality should be the same as the one mentioned above.

```
include/pipe.h         | 23 +++++++++++++++++++----
shell/shell.c          | 40 +++++++++++++++++++++++++++++++++++++---
system/initialize.c    | 14 ++++++++++++--
system/kill.c          | 10 ++++++++--
system/pipconnect.c    | 26 ++++++++++++++++++++++++-
system/pipcreate.c     | 40 +++++++++++++++++++++++++++++++++++++---
system/pipdelete.c     | 30 ++++++++++++++++++++++++++++-
system/pipdisconnect.c | 31 +++++++++++++++++++++++++++++-
system/pipgetc.c       |  9 ++++++---
system/pipputc.c       |  6 ++++--
system/piread.c        | 47 +++++++++++++++++++++++++++++++++++++++++++---
system/piputils.c      | 12 ++++++++++++
system/pipwrite.c      | 47 +++++++++++++++++++++++++++++++++++++++++++---
13 files changed, 307 insertions(+), 28 deletions(-)
```

Turn-in Instructions

- Electronic turn-in instructions:
  - Go to xinu-fall2017/compile directory and do make clean.
  - Go to the directory of which your xinu-fall2017 directory is a subdirectory (NOTE: please do not rename xinu-fall2017, or any of its subdirectories.) e.g., if /homes/abc/xinu-fall2017 is your directory structure, go to /homes/abc
  - Type in the following command:

    ```
    turnin -c cs503 -p lab2 xinu-fall2017
    ```

Notes

- Please start with a fresh copy of xinu-fall2017 codebase:

```
tar zxvf /u/u3/cs503/xinu-fall17/xinu-x86-qemu.tar.gz
```

- ALL debugging output should be turned off before you submit your code.