THE UNIVERSITY OF SYDNEY

COMP2017 / COMP9017 Week 9 Tutorial

More processes, Shared Memory and IPC Communication

Inter-process communication and shared memory

After being introduced to `fork` and how the operating system creates and manages processes. We will now look at how processes can communicate to each other. Inter-process communication can be managed through one of the two methods.

- Message Passing
- Shared Memory

Message Passing

Message passing is a form of communication between processes. Processes communicate by sending messages between each other through a communication channel. The messages that are passed between processes utilise a common protocol that is dependent on the domain.

This kind of approach is facilitated through the use of `pipes`. This creates an I/O channel between two processes that allow them to send messages between each other. As with any I/O channel, the data has to interpretable by both processes.

`pipe()`

The `pipe` function creates a unidirectional pipe for the current process. This is very useful in combination with `fork`. Due to the nature of fork, it will clone the current process, including the current file descriptors that have been created. When using `pipe` in conjunction with fork, we can facilitate a message passing channel between both processes, a single pipe will form a one way channel of communication where two pipes.

These pipes are typically called anonymous pipes.

More info: `man 2 pipe`

Usage of pipe

The standard way of communicating between parent and child processes is through a `pipe`. A pipe is a data flow through the operating system kernel: one end is writable, and anything written there will show up on the other end of the pipe. Pipes (as well as all other file descriptors) are preserved across actions like `fork` and `exec`, allowing a parent and child process to share a pipe.

A call to create a pipe looks like this:

```
int pipefd[2];
if (pipe(pipefd) < 0) {
    perror("unable to create pipe");
}
```

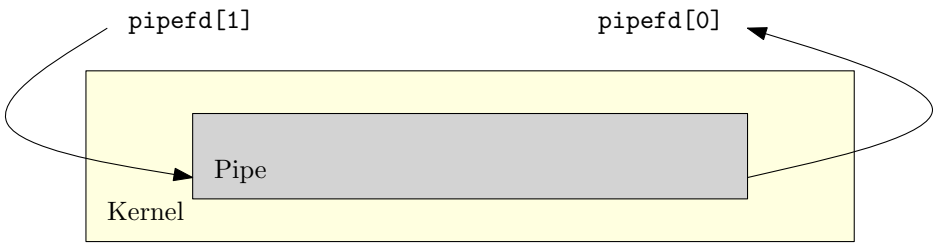

Figure 1: Contents of the pipe are buffered by the kernel. Data written to `pipefd[1]` appears on `pipefd[0]`

.

The integers `pipefd[0]` and `pipefd[1]` are the file descriptors for the pipe. Calls to `read()` and `write()` may *block* until input/output is possible. (Blocking means a function which may not return for a while, i.e. blocks the code at that line). For example, trying to read from a pipe will block until there is data to read. Writes to a pipe may block when the kernel's buffer allocated for the pipe is full.

More message passing with `mkfifo`

The `pipe` function creates anonymous pipes which are accessible through the file descriptor table on a process. As you will discover, this kind of pipe can be used between processes that belong to the same process tree (assuming they have copied the pipe). Many system architectures that focus on message passing require some method of advertising the communication interface.

`mkfifo` function allows us to create a `named` pipe to send data to our process. You can easily send data to a named pipe through your terminal (redirecting output from echo to the pipe). A process can open the pipe to read or write through regular IO functions.

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

#define BUF_SZ (256)
```

```c
#define CHANNEL_NAME ("comm")

int main() {
        if((mkfifo(CHANNEL_NAME, S_IRWXU | S_IRWXG)) >= 0) {
                int fd = open(CHANNEL_NAME, O_RDONLY);
                if(fd > 0) {
                        FILE* read_channel = fdopen(fd, "r");
                        char buf[BUF_SZ];
                        while(fgets(buf, BUF_SZ, read_channel) != NULL) {
                                puts(buf);
                        }
                        fclose(read_channel);
                }
        } else {
                fprintf(stderr, "Unable to open pipe");
        }
        return 0;
}
```

Pre-Tutorial Questions

Question 1: Some pipes

Write a program which does the following:

- Creates a pipe, and forks itself into a parent and child process.
- In the parent process, closes the read end of the pipe using `close()`, and then writes a message to the child over the pipe using the `write()` system call. Include a call to `printf` before writing to the pipe, so that you can watch what's happening. The parent should `close()` the pipe when it's finished with it.
- In the child process, closes the write end of the pipe using `close()`, and then waits for a message from the parent by trying to `read()` from the pipe. After receiving this message, the child should use `printf` to write a message out before exiting.
- Try playing around with the timing of things, to convince yourself that the call to `read()` really does block until the parent writes something. For example, include a call to `sleep(1)` before the parent writes to the pipe.

An example program might have the following output:

```
Parent: Sending a card to the child.
Child: thank you!
```

Extend the communication between the parent and the child to incorporate sending of data between the two processes.

```
Parent: The password is "bobafett"
Child: I will use the password "bobafett"
```

Question 2: What's the time?

You are required to create a program where the parent will ask the time from the child. Prior to the launching process forking, your program should create two sets of `pipes`.

Command line usage:

```
./tell_me_the_time
```

Output:

```
Parent: Hi! Do you know what time it is?
Child: The time is 8:30 !
Parent: Thank you!
```

You can use the following snippet to retrieve the current time set on your computer.

```c
struct tm* tm_info = localtime(&t); //statically allocated memory
char buf[256];
strftime(buf, 256, "%H:%M%p", tm_info);
printf("%s", buf);
```

Tutorial Questions

Question 3: Reading the output of a process

There's nothing special about file descriptors 0, 1, or 2, besides the fact that these are where `stdin`, `stdout`, and `stderr` go. The `close()` system call can remove an entry from the file descriptor table, freeing up a number for reuse. The `dup()` (short for *dup*licate) system call makes a copy of a file descriptor into the lowest available index in the table. Using these together allows the programmer to replace what "standard out" is:

fd	Destination
0	Terminal
1	Terminal
2	Terminal
3	(empty)
4	(empty)

Program start

fd	Destination
0	Terminal
1	Terminal
2	Terminal
3	`pipefd[0]`
4	`pipefd[1]`

`pipe(pipefd)`

fd	Destination
0	Terminal
1	(empty)
2	Terminal
3	`pipefd[0]`
4	`pipefd[1]`

`close(1)`

fd	Destination
0	Terminal
1	`pipefd[1]`
2	Terminal
3	`pipefd[0]`
4	`pipefd[1]`

`dup(pipefd[1])`

Table 1: The file descriptor table over the course of the program.

After the process table has been rearranged like this, any writes to standard out will actually go through the pipe instead. Since the file descriptor table is preserved across the system calls `fork` and `exec`, any program which is now executed will be writing to the pipe, instead of the terminal, for its standard output.

Write a program which reads back the contents of the `ls -l` command through a pipe, by following these steps:

1. Create a pipe, and fork off a child process.
2. In the child process, close the read end of the pipe, and replace file descriptor 1 with the write end of the pipe. Then use the `execlp` function to replace the child with `ls -l`.
3. In the parent process, close the write end of the pipe, and convert the pipe to a file stream by using `fdopen()` (see the note below). Then read the contents of the pipe line by line using `fgets`, giving each line a custom prefix (so you know the program is working).

The output of the program might look something like this:

```
Line 1: -rwxr-xr-x  1 admin  staff   8988 10 Apr 09:43 a.out
Line 2: -rw-r--r--  1 admin  staff  25005 10 Apr 12:29 pipe.pdf
Line 3: -rw-r--r--  1 admin  staff    680 10 Apr 12:31 test.c
```

To convert a file descriptor to a file stream (`FILE *`), use the `fdopen()` function:

```
FILE* fp = fdopen(pipefd[0], "r");
```

Now the stream `fp` can be used just like any other file-like object we've used up to this point in the course. In particular, `fscanf`, `fgets` and so on will work. Converting to a file stream hides many of the "ugly" parts of dealing with file descriptors: we rely on the C standard library to do the heavy lifting for us.

Shared memory

Shared memory is commonly associated with threads, since threads typically access to memory within the same process. However processes can allocate memory to be shared between other processes either within the same family of processes or independently.

When sharing memory between processes we can us the `POSIX` functions `shm_open` and `mmap`. `mmap` allows the programmer to create an allocation and apply rules to the region of memory (including the region to shared) while `shm_open` is used for providing a name for the shared memory for independent processes to use. `mmap` versatility allows the programmer to specify how memory is to be used. In regards to shared memory, specifying `MAP_SHARED` when `mmap` is called, provides other processes visibility to the same region of memory.

`mmap()` and `shm_x()`

`mmap`

`mmap` allows for creation of a memory mapping. Given a starting address, mmap will will create a new memory mapping and depending on the flags and this is where this function becomes very versatile and overly used.

`man 2 mmap` for more details

`shm_open` and `shm_unlink` (or `close`)

The `shm_open` function operates very similar to `open` as it returns a file descriptor after execution. `shm_open` function is used in conjunction with `ftruncate` and `mmap` to allow shared memory between independent processes.

This can be thought of as two independent processes reading and/or writing to the same file during their lifetimes.

Using mmap with files

`mmap` is a general function allows memory mapping of files which maps an processes memory to a file. When we operate on this area of memory it will cause that segment to be read or written to.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#define SOME_DATA (24)

int main(int argc, char** argv) {
  if(argc != 2) {
    //Need two arguments
    return 1;
  }

  char* block = NULL;
  int fd = open(argv[1], O_RDONLY);
  struct stat stat_b;
  fstat(fd, &stat_b);
  block = mmap(NULL, stat_b.st_size, PROT_WRITE|PROT_READ,
          MAP_PRIVATE, fd, 0);

  if(block == MAP_FAILED) {
    perror("MMAP Failed");
    close(fd);
    return 1;
  }
  //Read some bytes
  for(size_t i = 0; i < SOME_DATA, i++) {
    printf("%c", block[i]);
  }
  printf("\n");
  munmap(block, stat_b.st_size);
  close(fd);
}
```

Sharing between parent and child

In the previous tutorial we saw how the process is cloned and the data is copied to the other process on fork. When sharing between parent and child we can resort to using anonymous shared memory instead of file backed shared memory. This is similar to calling malloc but we will be sharing memory between both processes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#define DATA_SIZE (6)

int* give_data() {
  int* data = malloc(sizeof(int)*DATA_SIZE);
  for(int i = 0; i < DATA_SIZE; i++) { data[i] = i; }
  return data;
}

void read_share(int* d) {
  for(int i = 0; i < DATA_SIZE; i++) { printf("%d\n", d[i]); }
}

int main() {
  int* d = give_data();
  int* shared = mmap(NULL, DATA_SIZE*sizeof(int), PROT_READ|PROT_WRITE,
          MAP_ANON|MAP_SHARED, -1, 0);
  memcpy(shared, d, DATA_SIZE*sizeof(int));
  free(d);
  pid_t p = fork();
  if(p == 0) {
    printf("Child\n");
    read_share(shared);
    for(int i = 0; i < DATA_SIZE; i++) { shared[i] = i + 10; }
    munmap(shared, DATA_SIZE*sizeof(int));
  } else if(p > 0) {
    sleep(2);
    printf("Parent\n");
    read_share(shared);
    munmap(shared, DATA_SIZE*sizeof(int));
  }
  return 0;
}
```

Sharing between independent processes

Previous example showed anonymous memory mapping between parent, however a portable memory mapping implementation will require file-backing.

Similar to the previous example, prior to executing the mmap function, you will ensure you have a file descriptor that mmap will map to.

```
int fd = shm_open("/<name>")
ftruncate(fd, <size of data>);
```

After this has been executed we can then run mmap like so:

```
mmap(NULL, <size of data>, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);
```

- We have two techniques of communicating between processes, what are the pros and cons between both processes?
- Why must we use `shm_open` when sharing with unrelated processes?
- What problems do we face with processes reading and writing to the same space of memory? How could we solve this?
- What would happen if we tried to `mmap` a file that is larger than physical memory?
- What flag could we use to deal with this and what issues would you encounter?
- If you check your `/dev` directory and find `shm` directory, what would be the utility of this directory if you where to create a file there?

Question 4: What's the time (shared memory)

Change your program from `What's the time` to use shared memory instead. Your program can use one of the prior shared memory examples as a base to work from. After writing to the shared region of memory, your program can signal the other process, notifying it to read the data.

Extension: Instead of using software signal, you can use a `semaphore` to synchronise between the two processes.

Question 5: Money in the bank

You will construct a system in which one process maintain a list of bank accounts, each bank account will hold a balance, name, card and pin number. The other part of the system involves ATM processes. Each ATM process will allow a user to interact with it (you can choose to specify the number of processes).

Each ATM has the following functions which are delivered through a named pipe.

- `BALANCE <card number> <pin>`
- `DEPOSIT <amount> <card number>`
- `WITHDRAW <amount> <card number> <pin>`

The ATM and Bank processes have a segment of shared memory, which can be accessed. You can initially set this up so they are part of the same process tree, however, aim to have independent processes interact with the named shared memory segment.

Once you have implemented your processes and can confirm the above operations work correctly, attempt to set up a scenario where two ATMs withdraw and deposit over 100 times for each process. Observe the final result and see if your machine produces the expected result. If you did not get your expected result, what do you think could have caused this and how could you fix it?

Question 6: Multi-process Messages

You are to create a message server where it will maintain a history. Your multi-process program will know the maximum number of users that it can maintain and a buffer history size.

You will need to use a conjunction of `shm_open` and `mmap` to solve this problem. Implement the following commands.

```
NAME <sets my name>
LIST <will show other processes names>
CHECK <checks for messages>
MESSAGE <a 256 character string to be sent>
```

Build a client application that will interact with the shared memory and try and potentially work with your friends and see if they can make a compatible client for your server. As an **extension**, when a message has come from server, notify all clients to read the messages that have been sent.