Lecture 10 — Pipes and Shared Memory

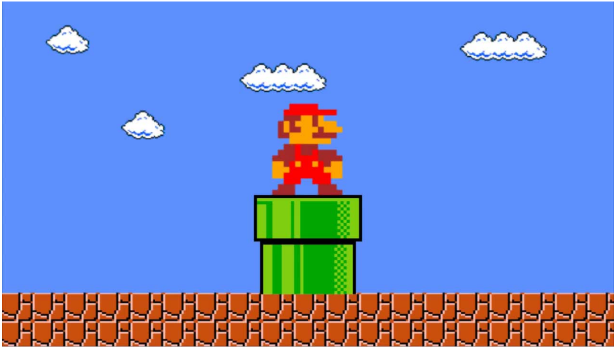
Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 13, 2019

In addition to message passing, we should talk about pipes and shared memory.

In UNIX, we can create a *pipe* to set up communication.



The producer writes in one end; the consumer receives on the other.

This is unidirectional, so if bidirectional communication is desired, two pipes must be used (going in different directions).

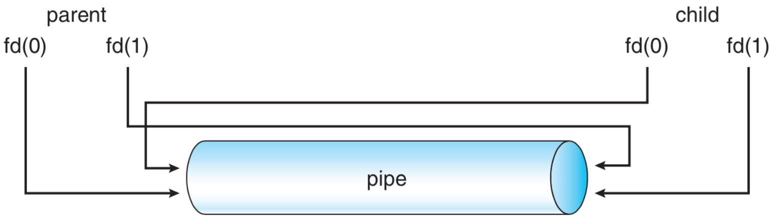
Constructing a UNIX Pipe

The method is `pipe` and it is constructed with the call:

```
pipe( int fileDescriptors[])
```

where `fileDescriptors[0]` is the read-end; and `fileDescriptors[1]` is the write-end.

Yes, `fileDescriptors` means that UNIX thinks of a pipe as a file even though it is in memory.


parent
fd(0)
fd(1)
child
fd(0)
fd(1)
pipe

The pipe is a block of main memory interpreted as a circular queue.

Each entry in the queue is fixed in size and usually one character.

The sender may place the message into the queue in small chunks.

The receiver gets data one character at a time.

The sender and receiver need to know when the message is finished.

Solutions: termination character, or declared length at the start.

Named Pipes

A UNIX pipe may be stored on disk.

When this happens, we call it a named pipe.

Unless we make it a named pipe, a pipe exists only as long as the processes are communicating.

Regular pipes require a parent-child process relationship.
Named pipes do not.

Named pipes are also bidirectional, but one direction at a time.

UNIX Command Line Pipes

You may have worked with pipes on the UNIX command line.

A command like `cat fork.c | less` creates a pipe;.

It takes the output of the `cat` program and delivers it as input to `less`.

Pipe Code Example

Use `fork` to spawn a new child process and then setting up a communication pipe between the parent and child.

We will send a message "Greetings" from the parent to the child.

```
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

if (pipe(fd) == -1) {
  fprintf(stderr,"Pipe failed");
  return 1;
}
```

Pipe Code Example, Continued

```
/* fork a child process */
  pid = fork();

   if (pid < 0) {
     /* error occurred */
     fprintf(stderr, "Fork␣Failed");
     return 1;
   }
  if (pid > 0) { /* parent process */
     /* close the unused end of the pipe */
     close(fd[READ_END]);

     /* write to the pipe */
     write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

     /* close the write end of the pipe */
     close(fd[WRITE_END]);

   }
```

`READ_END` is defined as `0` in a `#define` directive.
`WRITE_END` is defined as `1` in a `#define` directive.

Pipe Code Example, Continued

```
  else { /* child process */
    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read␣%s",read_msg);
     /* close the write end of the pipe */
     close(fd[READ_END]);
  }
  return 0;
}
```

Does the output match what's supposed to happen?

Or are there extra characters?

Creating a Named Pipe

If we wanted to create a named pipe, the system call is `mkfifo`.

Sometimes a named pipe is called a FIFO.

As it is a file, it can be manipulated with the usual UNIX file system calls: `open`, `read`, `write`, and `close`.

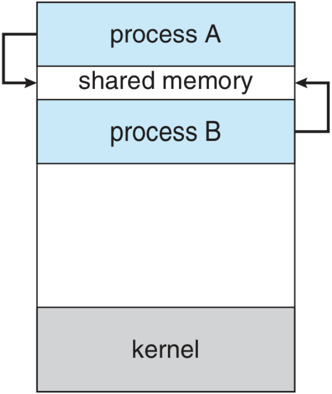
Shared Memory

Conceptually, the idea of shared memory is very simple.

A particular region of memory is designated as being shared between multiple processes, all of whom may read and write to that location.

To share an area of memory, the OS must be notified.

Normally, a region of memory is associated with exactly one process (its owner).



The kernel is only involved in the setup and cleanup of that shared area.

Whose Turn Is It Anyway?

When a section of memory is shared, there exists the possibility that one process will overwrite another's changes.

To prevent this sort of problem, we will need a mechanism for co-ordination...

A subject we will return to later.

Share a Key

Suppose we want to share a section of memory.

We need to obtain a key that identifies a specific memory segment.

Keys are just integer values, so we would like them to be unique.

One method is to generate the key with the "file to key" function found in `sys/ipc.h`:

```
key_t ftok( char *pathname, int proj )
```

The key is generated from the given file name (`pathname`) and the value `proj`.

Not Private Key Encryption

There is a very small risk of duplicate numbers.

Another way we can get a key is using the constant `IPC_PRIVATE`.

If we give the constant in where a `key_t` is expected then a guaranteed unique key is returned.

Workflow for Shared Memory

- Create a new shared memory segment – `shmget`.
- Attach the shared memory segment – `shmat`.
- Then the process can use the shared memory.
- Detach – `shmdt`.
- Delete the shared memory segment, done by one process only – `shmctl`.

Create

To create a shared memory segment, or get a reference to an existing one, we use `shmget`.

```
int shmget( key_t key, size_t size, int shmflg );
```

The first argument is the key, which can be either the result of a `ftok()` call or the constant `IPC_PRIVATE`.

`size`: how many bytes of memory are to be shared.

`shmflg`: access permissions (UNIX standards, eg 600)
  Optional: `IPC_CREAT`, `IPC_EXCL`

Return value: the integer ID of the shared memory segment.

Attach

```
void* shmat( int shmid, const void* shmaddr, int shmflg );
```

`shmid`: ID of the shared memory segment.

`shmaddr`: where it should go; always use `NULL`.

`shmflg`: optionally, `SHM_RDONLY`

Return value: standard C pointer with the address of shared memory.

But how do we know what the shared memory segment ID is?

This is the key for what?

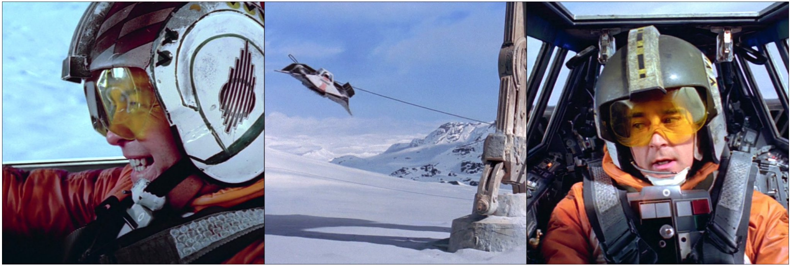
If we created the segment ourselves, we obviously know where it is.

But presumably you want some other process to have it as well.

If two processes use the same input values for `ftok()` they will get the same result, so that's one method.

Or, if a parent attaches a shared memory segment and then calls `fork()`, the child inherits the shared memory segments, so it's is already set up.

Detach Cable!



Detach Segment

When we are done with a segment we can detach from it with `shmdt`.

```
int shmdt( const void* shmaddr );
```

`shmaddr`: the address returned by the attach call

If we forget, it happens at process termination (but don't forget!)

Delete Shared Memory

```
int shmctl( int schmidt, int cmd, struct shmid_ds *buf )
```

This function can do a lot more than delete it, such as modify properties of the data structure that is used to control shared memory.

The command is `IPC_RMID` (“remove ID”).

We must leave the last argument as `NULL` for this deletion.

Deletion may be deferred!

Shared Memory Example

```c
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

int main( int argc, char** argv ) {
    int shmid = shmget( IPC_PRIVATE, 32, IPC_CREAT | 0666  );

    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        waitpid( pid, NULL, 0 );
        void* mem = shmat( shmid, NULL, 0 );
        printf("The msg received from the child is %s.\n", (char*) mem );
        shmdt( mem );
        shmctl( shmid, IPC_RMID, NULL );
    } else if ( pid == 0 ) { /* Child */
        void* mem = shmat( shmid, NULL, 0 );
        memset( mem, 0, 32 );
        sprintf( mem, "Hello World" );
        shmdt( mem );
    }
    return 0;
}
```

Consult the Map

Altenative: `mmap()`

An alternative approach for shared memory involves the use of `mmap()`, a function nominally used to map a file into memory.

But we can also use this for IPC!

```
void* mmap( void* address, size_t length, int protection, int flag,
   int fd, off_t offset );
```

`address`: where you want the mapped region to go; use `NULL`.

`length`: how many bytes to map.

`protection`: rules for how memory can be used.

`flag`: mode for mapping.

`fd`: file descriptor of the file to map.

`offset`: how far from the start of the file mapping begins.

Protection Flags

Valid values are `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and `PROT_EXECUTE`.

They can be combined with the bitwise OR operator.

Whatever flags you choose have to be consistent with how the file was opened with `open`.

What’s the point of PROT_NONE, if all things are forbidden?

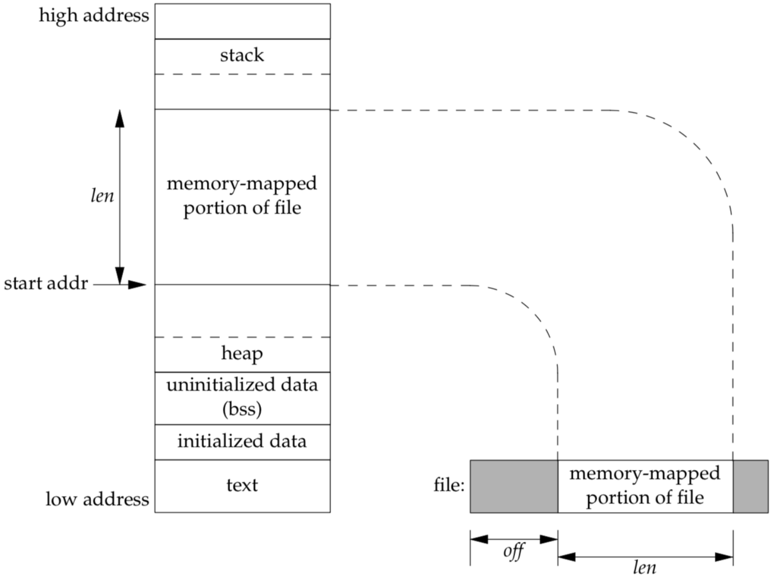
Flags

Flags can be one of two options: `MAP_PRIVATE` or `MAP_SHARED`.

Private: modifications are not visible to other processes mapping the same file and not written out to the underlying file.

Shared: modifications are visible to other processes and written out to the file… but maybe not instantly.

Memory Mapped File



If we wish to change the protection rules for a section, we use `mprotect`.

```
int mprotect( void* address, size_t length, int prot );
```

`address`: the memory to modify protection of.

`length`: the size of said memory.

`port`: the new protection rules.

```
int msync( void* address, size_t length, int flags );
```

`address`: the memory to synchronize.

`length`: how many bytes to synchronize.

`flags`: mode for synchronization; use `MS_SYNC` (blocking).

Unmap

```
int munmap( void* address, size_t length );
```

`address`: the memory to unmap.

`length`: how many bytes to unmap.

A segment would be unmapped automatically when a process exits, but as always it is polite to unmap it as soon as you know that you are done with it.

Memory Mapping Example

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main( int argc, char** argv ) {

    int fd = open( "example.txt", O_RDWR );

    struct stat st;
    stat( "example.txt", &st );
    ssize_t size = st.st_size;
    void* mapped = mmap( NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0 );
```

```
    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        waitpid( pid, NULL, 0 );
        printf("The␣new␣content␣of␣the␣file␣is:␣%s.\n", (char*) mapped);
        munmap( mapped, size );
    } else if ( pid == 0 ) { /* Child */
       memset( mapped, 0, size ); /* Erase what's there */
       sprintf( mapped, "It␣is␣now␣Overwritten");
       /* Ensure data is synchronized */
       msync( mapped, size, MS_SYNC );
       munmap( mapped, size );
    }

    return 0;
}
```

The Example is... Flawed

The example works acceptably in the sense that we successfully overwrite the data with the new data and the parent process sees the change.

But things get weird if we tried to write fewer bytes than the original message.

In general, the mapped area size cannot change.

Linux has `mremap` but this is not portable...

But this would be great for something like sorting an array, wouldn’t it?
The sorted array is the same size as the input and we could share the work...