Welcome to CS170 section!!!

Lab 3 Intro

Nov 18, 2021
So now that we have multiple processes, how can they talk to each other?

KOS' memory

<table>
<thead>
<tr>
<th>main-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCESS 3</td>
</tr>
<tr>
<td>PROCESS 2</td>
</tr>
<tr>
<td>PROCESS 1</td>
</tr>
</tbody>
</table>

- User_Base
- User_Limit

- load_user_program(char *fn)
- ...
- ...
- run_user_code Limit registers[1]
- ...
- ...
- etc.
Pipes!

KOS memory

main-memory

User_Base
User_Limit

load_user_program(char *fn) I

run_user_code Lin registers[] I

I

etc.
Multiple processes can all write to and read from the same pipe

```c
Friendly process
write(3, "Hi!");
```

```c
Antisocial process
write(3, "Bye!");
```

```c
Listening process
char buf[100];
read(4, buf);
```
int main(int argc, char **argv)
{
    int pipe_desc[2];
    char *string;
    char read_buffer[4096];
    int child_status;

    // Create a new pipe
    pipe(pipe_desc);

    pid_t child_id = fork();
    if(child_id != 0) {
        // Parent will be the writer, so it can close the read end
        close(pipe_desc[0]);

        // Send the child a string
        string = "a string made by the parent\n";
        write(pipe_desc[1],string,strlen(string));

        // Close the pipe to let the read end know we are done
        close(pipe_desc[1]);

        // Wait for the child to exit
        wait(&child_status);
    } else {
        // Child will be the reader, it can close the write end
        close(pipe_desc[1]);

        // Zero out a buffer to read into
        memset(read_buffer,0,sizeof(read_buffer));

        // Read string from the parent
        read(pipe_desc[0],read_buffer,sizeof(read_buffer));

        close(pipe_desc[0]);
    }
    exit(0);
}
int main(int argc, char **argv)
{
    int pipe_desc[2];
    char *string;
    char read_buffer[4096];
    int child_status;

    // Create a new pipe
    pipe(pipe_desc);

    pid_t child_id = fork();
    if(child_id != 0) {
        // Parent will be the writer, so it can close the read end
        close(pipe_desc[0]);

        // Send the child a string
        string = "a string made by the parent\n";
        write(pipe_desc[1],string,strlen(string));

        // Close the pipe to let the read end know we are done
        close(pipe_desc[1]);

        // Wait for the child to exit
        wait(&child_status);
    } else {
        // Child will be the reader, it can close the write end
        close(pipe_desc[1]);

        // Zero out a buffer to read into
        memset(read_buffer,0,sizeof(read_buffer));

        // Read string from the parent
        read(pipe_desc[0],read_buffer,sizeof(read_buffer));

        close(pipe_desc[0]);
    }
    exit(0);
}
```c
int main(int argc, char **argv)
{
    int pipe_desc[2];
    char *string;
    char read_buffer[4096];
    int child_status;

    // Create a new pipe
    pipe(pipe_desc);

    pid_t child_id = fork();
    if(child_id != 0) {
        // Parent will be the writer, so it can close the read end
        close(pipe_desc[0]);

        // Send the child a string
        string = "a string made by the parent\n";
        write(pipe_desc[1],string,strlen(string));

        // Close the pipe to let the read end know we are done
        close(pipe_desc[1]);

        // Wait for the child to exit
        wait(&child_status);
    } else {
        // Child will be the reader, it can close the write end
        close(pipe_desc[1]);

        // Zero out a buffer to read into
        memset(read_buffer,0,sizeof(read_buffer));

        // Read string from the parent
        read(pipe_desc[0],read_buffer,sizeof(read_buffer));
        close(pipe_desc[0]);
    }
    exit(0);
}
```
In Lab 3, you have to implement the following system calls:

```c
int pipe(int pipefd[2]);

int dup(int oldfd);

int dup2(int oldfd, int newfd);

int close(int fd);
```

And probably modify at least the following system calls:

```c
ssize_t write(int fd, const void *buf, size_t count);

ssize_t read(int fd, void *buf, size_t count);

pid_t fork(void);

void exit(int status);
```
PIPE

From the man page (typing ‘man 2 pipe’ into the terminal):

```c
int pipe(int pipefd[2]);
```

Pipe(0) creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe. `pipefd[0]` refers to the read end of the pipe. `pipefd[1]` refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see pipe(7).
DUP, DUP2

From the man page (typing ‘man dup’ into the terminal):

```c
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

The `dup()` system call creates a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor.

After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description (see `open(2)`) and thus share file offset and file status flags; for example, if the file offset is modified by using `lseek(2)` on one of the file descriptors, the offset is also changed for the other.

The two file descriptors do not share file descriptor flags (the close-on-exec flag). The close-on-exec flag (FD_CLOEXEC; see `fcntl(2)`) for the duplicate descriptor is off.

dup2()

The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being reused.
From the man page (typing ‘man close’ into the terminal):

```c
int close(int fd);
```

`close()` closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see `fcntl(2)`) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If `fd` is the last file descriptor referring to the underlying open file description (see `open(2)`), the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using `unlink(2)`, the file is deleted.
Questions

What is a file descriptor?

What does this code do? `write(3, “Hey! What’s up?”);`
Questions

What is a file descriptor?

A file descriptor is an integer that the process corresponds to a place to read or write (the console or a pipe in kos).

In other words, it is an index into the process’ file descriptor table.

What does this code do? `write(3, “Hey! What’s up?”);`
Questions

What is a file descriptor?

A file descriptor is an integer that the process corresponds to a place to read or write (the console or a pipe in ko/s).

In other words, it is an index into the process’ file descriptor table.

What does this code do? `write(3, “Hey! What’s up?”);`

It writes “Hey! What’s up?” to the location referred to by file descriptor 3.
KOS' memory

main-memory

User_Base
User_Limit

load_user_program(char* fn);

run_user_code(int registers[]);

etc.

PROC 1 CONTROL BLOCK

registers[1] = ...
save pointer
base = 0
limit = memory size
file descriptor table

standard in
standard out
standard err
pipe