Systems 1 – CS 6013

Computer Architecture and Operating Systems

Lecture 13: Inter-process Communication (IPC) / Unix Shell

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2022

*(adapted from slides by Scott Brandt at UC Santa Cruz and other general sources, including previous MSD slides)

Lecture 13 – Topics

- IPC – Inter-Process Communication
- Unix Shell Assignment Topics

Reminders

- Unix Shell is Due Friday, Feb 16
  - Have you started?

Misc: sizeof() vs strlen() - and memory addr's


```
void doit( int i[20] ) {
    // i param above is BAD form, but
    // compiler swallows it...
}   // (See nm command below)

int main()
{
   int   i1[10];
   int * i2 = new int[10];
   const char * cPtr = "hello";
   doit( i1 ); // Why does this work?
}
```

```
sizeof( i1 ):   40
sizeof( i2 ):   8
sizeof( cPtr ): 8
strlen( cPtr ): 5
                          HEX                DEC
Address i1: 0x7ffd2b8903e0 140725333853152
Address i2:       0xe08040          14712896
Address cPtr:     0x400ac4           4197060
```

```
> nm a.out | grep doit | c++filt  # What are '#','>'?
   000000000040082d T doit( int* )  # Type of doit?
```

- Sizes / Addresses of above variables?

Misc - File Descriptors and Dup2


```
void dup_cin_example()
{
  int fd = open( "output.txt", O_WRONLY | O_CREAT | ... );
  // DUP2( FD1, FD2 )  dup2 name is bad... should be rename
  //   Make FD1 become a copy of FD2, then:
  //   Close FD2
  dup2( fd, STDOUT_FILENO );
  cout << "This is a test, this is only a test.\n";
  cout << "Let's hope this works!!!\n";
  close( fd );
}
```

Miscellaneous

- POSIX
  - **Portable Operating System Interface**
  - A set of standard operating system interfaces based on the Unix operating system.

Miscellaneous

- Want to read / write a single integer:

```
int x[ 1 ]
write( OUT, x, 4 );
```

- However, an array of one integer doesn't make a lot of sense… so how do we fix this (make it better)?

```
int x;
write( OUT, x, 4 ); // Compiler Error
```

- Parameter 2 is not valid…
  - What type does `write` expect for param 2?
  - A pointer… so?

```
write( OUT, &x, 1 );
```

- Question:

```
void * vs int *
sizeof( int * )
sizeof( void * )
```

Miscellaneous


```
void printError( int rc, char *message ) {
    if( rc < 0 ) {
        perror( message );
        exit( 1 );
    }
}
```

- What's wrong with "`printError()`"?
  - Doesn't do what it claims to do… How to fix?
  - Rename to `checkForError()`

(still more) Miscellaneous

- `int fds[ 2 ]`
- `pipe( fds );`
- Assuming we are going to use this pipe to write from the parent to the child:
  - In Parent:
    - write to `fds[ 1 ]`
    - What about `fds[ 0 ]`?
      - We should just close it (to make sure we don't accidentally use it).
      - `close( fds[ 0 ] )`
  - In Child:
    - read from `fds[ 0 ]`
    - What about `fds[ 1 ]`?
      - Again, just close it.

- How to write from child to parent?

```
int child_to_parent_fds[ 2 ];
int parent_to_child_fds[ 2 ];
pipe( child_to_parent_fds );
pipe( parent_to_child_fds );
fork()
```

- Blocking vs Non-blocking calls?

  eg: `read()`
  - See `fcntl()`

Using pipes on the command line

- `sort data.txt | uniq`
- `cat code.cpp | head -20 | tail -10`
- <program A> | <program B> [| …]
  - Standard Out from program A is sent to the Standard In for program B.
  - In what order are these programs executed (run)?
    - All programs that are connected this way via pipes are run at the same time!
- Note: Turns out that the | (pipe), just like everything else, is a file.

Shell Assignment

- <program A> | <program B>
  - Parse the entire command line, which gives us?
    - Two Commands.
      - technically two `struct Command`
  - Don't run each command as you parse it… run them all at the same time. How do we do this?
    - loop over each command and `fork()/exec()` it.
  - But before we `exec()` (or `fork()`) the programs, what do we need to do with their inputs / outputs?
    - Connect the output of 1st to the input of the 2nd (etc.)
    - What does this mean?
      - Create pipes. How many?
      - One for each *pipe* on the command line (or more specifically the number of pipes is equal to the # of commands - 1).

Command Line File Redirection

- Redirect input: <
- Redirect output: >

```
# echo "hello" > file.txt
```

- Standard Out is no longer the screen – it is file.txt.

```
# nl < file.txt
```

- Standard In is no longer the keyboard – it is file.txt.
- What is the # on the above example lines?
  - The prompt character. Note on most Unix systems, a # prompt means you are running as the root user – so be careful.

Recall

- Inter-process Communication (IPC) enables processes to communicate with each other to share information
  - Pipes (half duplex)
  - FIFOs (named pipes)
  - Stream pipes (full duplex)
  - Named stream pipes
  - Message queues
  - Semaphores
  - Shared Memory
  - Sockets
  - Streams

Named Pipes (aka FIFOs)

- Not to be confused with the FIFO scheduler.
- Full duplex.
  - Data can flow in both directions.
- Can now connect any pair of processes (including parent-child or child-child).
- A FIFO is a named pipe.
  - By giving the pipe a name, we can actually refer to it as if it were a file*!
    - *It is a file.

Named Pipes (FIFO) – command line

- Note: `mkfifo` is also a unix command.
- You can therefore use it in the shell (aka command line).
- Try:
  1. `mkfifo /tmp/mypipe`

     ls: `prw-------  1 dav wheel Feb 8 18:46 /tmp/mypipe`

     1. Then, in one console: `ls -l > /tmp/mypipe`
     2. In a second console: `cat < /tmp/mypipe`
- Note, you might need `>!`
  - Depending on your shell…
- Let's try this as a group… Log into shell.cs.utah.edu.

Named Pipes (via Code)


```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo( const char *path, mode_t mode );
```

- This creates a "FIFO special file" with name `path`. Here, `mode` sets the FIFO (file) permissions.

- Can also use `mknod()` to create a FIFO.
  - This is the old / non-standard way to make a FIFO

`const char *` vs `char *`

`string s;`

- *Where* is the actual data for this string?

`s.c_str()`

- What type of data does `c_str()` return (and why)?
  - `const char *`, So you can't change its data under the covers
- How are the `args` stored in the Command struct?
  - `vector< const char * >`
- What is the 2nd parameter to exec()?
  - `argv`, which is…
  - `char *` (or perhaps better to think of it as a `char[]`)
- Can't pass a `const char *` to a function that takes in `char *`

`const char *` vs `char *`

- How to turn a `const char *` into a `char *`?
- You use `const_cast` to remove const from a variable, though you should almost never do it…
  - The exception being when using older routines that were written before `const` was a thing.

```
char * arg = const_cast<char*>( command.argv[1] );
```

- This un-consts a single `char *`. I'll leave it as an exercise to the reader on how to convert an array of `const char *` into an array of `char *`

Assignments

- Code Review?
  - Anyone want me to review their lab code?
- Unix Shell
  - Questions?

~ Fin ~