Project 1A

CS111
Discussion 1A

Breaking down project 1

- Instead of putting everything in a main, you may find it useful to write functions that:
  - Parses the next command and its arguments
    - Dynamically allocating memory for arguments
    - Checking that the syntax is correct
  - Executes a command given name and arguments
    - Creating the child process
    - Performing the necessary redirection
    - Executing the process
- Keep track of the number of open files
  - You can write a function that opens files/pipes
  - It increments a counter on the number of open files
  - It resets a dynamic flag variable after opening

The verbose flag

- If this option is specified, print to stdout all options and arguments to that option before execution
  - To do this, write() to stdout, do not print()/putchar()...
  - Keep track of how many arguments have been passed to the current option

- getopt() will not recognise arguments, you will have to manually set optind as you iterate through them
  - getopt() will jump to the next option, which is a string starting with '--'

fork(2)

Goal: Create a child process

Success:
- Parent receives child PID
- Child receives 0

Error: Parent receives -1

pid_t fork(void)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {

    fork();

    printf("Hello world!\n");
    return 0;
}
```

What is the output ?

fork(2)

Goal: Create a child process

Success:
- Parent receives child PID
- Child receives 0

Error: Parent receives -1

pid_t fork(void)

Important note:
When a parent calls fork(), the child that is spawned inherits all the open file descriptors from the parent! (Keep this in mind for later)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {

    fork();

    printf("Hello world!\n");
    return 0;
}
```

What is the output ?

Hello world
Hello world

2 processes running concurrently

exec(3)

Goal: loading a new program in a calling process

Error : -1

```
int execvp(const char *file, char *const argv[]);
```

Path to new process image file

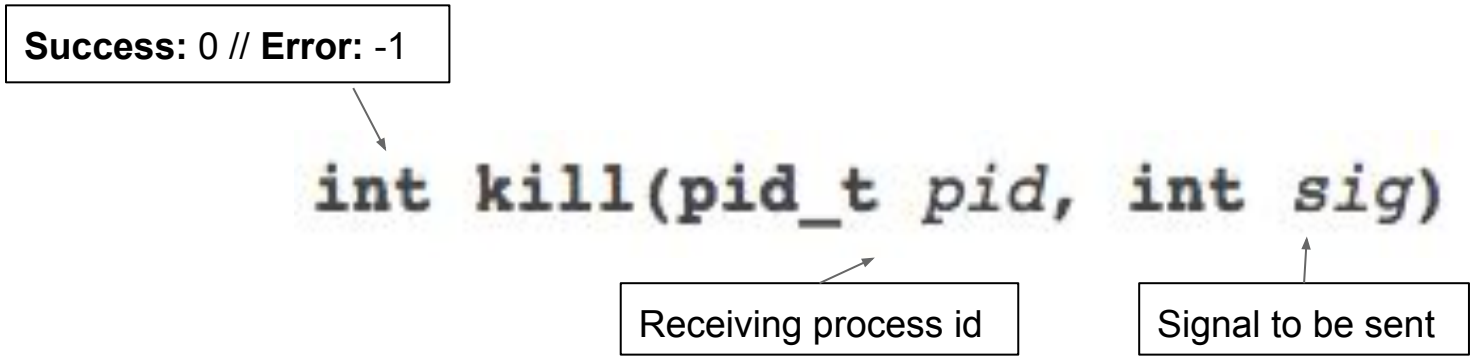
- Pointers to arguments available to new process image
- List is terminated by a NULL pointer

You know that:

- Every '--command' should be followed by at least 4 options (3 fd's + command name)
- Options for that '--command' are over when you're out of arguments or you see the next '--' in argc
- Creating a 'command' structure that holds all options for given commands could be useful
- Terminate it by NULL

kill(2)

Goal: send a signal to a process (or a group)



(pid>0)? Send to process **pid**

(pid==0)? Send to all processes in group of sender

(pid==-1)? Send to all processes in group of calling process

(pid<-1)? Send to all processes whose gpid = abs(**pid**)

What signal will you send ? SIGINT for 0x03

waitpid(2)

Goal: wait for child process to change state

pid of terminated child / **Error:** -1

Location to store exit information

pid_t waitpid(pid_t *pid*, int *status, int *options*)

- **<-1** : wait for any child process whose gpid == abs(**pid**)
- **-1** : wait for any child process
- **0** : wait for any child process whose gpid == (calling process pid)
- **>0** : Wait for the child whose pid == **pid**

- OR of these parameters (cf man):
- **WEXITSTATUS** : return exit status of the child
- **WIFSIGNALED** : returns True if child was terminated by a signal
- **WTERMSIG** : returns the number of the signal that caused the child to terminate

waitpid(2)

When will you use it ?

- To harvest the exit status of a child process
- Or the signal number that caused the process to exit
- You will only wait on previous commands!
  - Store the pid of the commands you run through fork() calls

- If an exit status is available, print it out
- If the child was terminated by a signal, return the signal's value

File flags

- There seems to be a lot of options in 1B...
  - But most of them are super simple!
- Create an open_flag variable
  - Initialize it to 0
  - Reset it to 0 every time you open a file
- You can now implement all flags with an OR operation
  - File_flag |= <option>

Pipe(2)

- Used for interprocess communication (IPC) on the same host
- When a call to pipe() is made, 2 file descriptors are returned
- One is the write end (on the left) and one is the read end (on the right)
- The idea is for process A to write(fd_out) and B to read (fd_in) : data goes from A to B!
- Note that pipes are unidirectional! If A writes to B, B **cannot** write to A. (need a 2nd pipe)

Should we call fork() before pipe() or after? Does it matter?

Pipe(2)

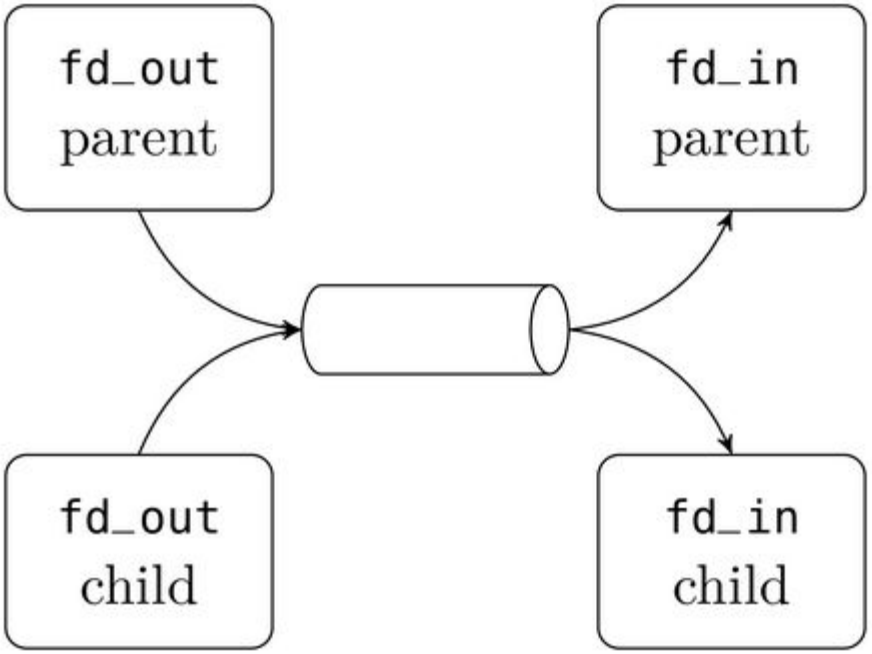
- Used for interprocess communication (IPC) on the same host
- When a call to pipe() is made, 2 file descriptors are returned
- One is the write end (on the left) and one is the read end (on the right)
- The idea is for process A to write(fd_out) and B to read (fd_in) : data goes from A to B!
- Note that pipes are unidirectional! If A writes to B, B **cannot** write to A. (need a 2nd pipe)

Should we call fork() before pipe() or after? Does it matter?

- Important: Both processes A and B need to have access to the file descriptors associated to the pipe they will communicate with.
- Remember: when process A calls fork() to spawn process B, process B inherits all fds from A
- We need to call pipe() (creating 2 new fds in process A) and then fork() (so that B also gets access)
- If we fork() first, A and B don't have the same file descriptors anymore!

Pipe(2) : child is the writer

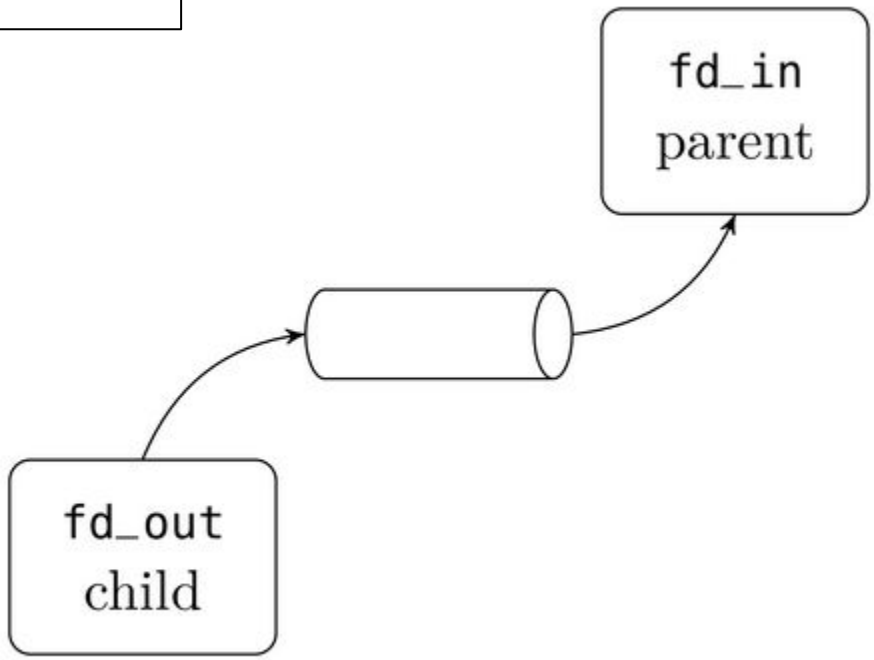
Senario: child writes to parent. Which file descriptors should we close?



Note:
This question arises because since we called pipe() and then fork(), both child and parent have access to both read and write ends of the pipes (4 fds total).
Pipes are unidirectional, so we will only use 2 of those 4. To pick the direction we have 2 fd's to close: one write end and one read end.
Closing unused file descriptors when piping is necessary (to harvest exit statuses)

Pipe(2) : child is the writer

Close 1 fd on the parent side and one on the child side



Pipe : close your file descriptors!

- In this project, you will create pipes between different children
- Just like in part A, you will have to do redirection after the call to fork()
  - To define where your child process takes input and where it outputs
  - In 1A, you did redirection with regular files
  - You can handle pipes in the same way!
- You will need to close the ends of the pipe you're not using
  - The simplest policy is to systematically close all file descriptors larger than 2 (in the child) **after redirection**

--catch

- There are 3 signal options
  - You can use either signal() (simpler, deprecated) or sigaction() (more complete and complicated) to implement these
- For --catch, just register a signal handler
  - Print out the signal number and ‘caught’ to stderr
  - And exit(signal number)
- You can pass your signal handler to signal() or sigaction
  - Both calls just aim at defining what behavior your process should have when the specified signal arises
  - The second argument of signal() is the handler
  - For sigaction you have to define a ‘sigaction’ structure
    - Pass the handler through the sa_handler field

--ignore

- Once again, you want to modify the behavior of your process if it receives a given signal
- You are doing the same thing as in --catch, the only thing that changes is the handler you specify
- In catch, it was one of your functions
  - Now you have to ignore the signal
- SIG_IGN is the disposition that should be passed as a handler
  - Second argument of signal()
  - The second argument to sigaction() is a pointer to the sigaction structure, set its sa_handler field to SIG_IGN

--default

- We now want the ‘default’ handler
- SIG_DFL

--abort

- Force a segfault
- The system should dump core via a segmentation violation
- Just like lab0! Dereference a null pointer

--pause()

- pause , waiting for a signal to arrive
- The C standard library provides a function pause()
  - It causes the the calling process to sleep until a signal is delivered
- Just call it!

getrusage()

Goal: Measure resource usage.

Fills out this struct:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

timeval

- The measured time is stored in a structure, which has 2 fields:
  - tv_sec : number of **whole** seconds of elapsed time
  - tv_usec : rest of the time in microseconds

- Time_elapsed = tv_sec + tv_usec
  - The 2 fields are not the same value in different units!

getrusage()

- You have to specify “who’s” resource usage you want to monitor
- You want to monitor every option
- The options are:
  - RUSAGE_SELF : monitor calling process (sum of all threads)
  - RUSAGE_CHILDREN : monitor all descendants that have terminated or have been waited for
  - RUSAGE_THREAD : monitor the calling thread
- Where will you use it?
  - When using catch, close, pipe, opening a file -> just the calling process
  - When executing a command -> also monitor the child process!