5950 Section 9 - C++ Smart Pointers and Fork

Welcome back to recitation! We're glad that you're here :)

Processes

Process vs. Threads:

- A process has a virtual address space.
- Threads are contained within processes. Every process has at least one thread.
- Threads within the same process access the same virtual memory.
- But new processes have to copy over the entire address space from the process it was created from (much slower than creating a thread!)

Process Functions:

There are a variety of functions commonly used with processes:

- `pid_t fork()`
  - Child process returns 0, parent process returns the pid of the child process
- `void exit(int status)`
  - Exits the currently running process with specified status
- `pid_t wait(int* wstatus)`
  - Waits for **any** child process to exit. Gets their status through the output parameter `wstatus`.
- `pid_t waitpid(pid_t child, int* wstatus, int options)`
  - Waits for the **specified** child process to exit. Gets their status through the output parameter `wstatus`. Options can be specified, leave as 0 for default
- `execvp(char* file, char* argv[])`
  - Executes a specific command/program with specified arguments
  - argv must have `NULL`/`nullptr` as it's last value
  - `argv[0]` should have the same values as file
- `pipe(int pipefds[2])`
  - OS creates a pipe to support IPC and initializes `fd[0]` and `fd[1]` to contain the file descriptors to read from (`fd[0]`) and write to (`fd[1]`) the pipe.

Exercise:

1) Fork and Loop

Consider the program below. How many times is :) printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
      fork();
  }
  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

With each iteration, every existing process calls fork. Additionally, newly created child processes keep the same value of i, since it's copied from the parent. That means the number of processes doubles at each iteration, for 4 iterations (which makes 2^4 processes when the loop terminates). Since the program prints after the loop's termination, the number of prints is equal to the number of processes after the for-loop terminates. Therefore, there are 16 :).

2) Processes and Wait

Consider the program below. What are the possible outputs of this program?

```
int main(int argc, char* argv[]) {
  pid_t pid =  fork();
  if (pid == 0) {
     pid =  fork();
     if (pid == 0) {
       cout << "my\n";
     } else {
       cout << "mind\n";
     }
      exit(EXIT_SUCCESS);
  }
  pid =  fork();
  if (pid == 0) {
     cout << "skies\n";
      exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "of\n"; // "\n" is similar to endl
}
```

Each print statement only comes after all processes have been created. The only "synchronization" is the parent waits for the second child to exit before the parent prints. Therefore, "of" must come after "skies," though "of" does not have to immediately follow "skies." As long as that is maintained all orderings of the four print statements is possible

Communication between Processes:

Since processes have separate memory, processes use pipes to share information.

- Pipes are unidirectional. They have a read end and a write end.
- EOF is not come from the read end of a pipe until all accesses to the write end have been closed and there is nothing left to be read
    - A bug may be caused by not closing all the write ends after your process is done with them. If the reading process needs to read EOF in order to continue with the program, it will get stuck.

In addition to using pipes, once can use files to communicate between processes. Just as with a pipe, there is one instance of a particular file on the system. However, each process can have their own file descriptors to access that file/pipe. This means that if one process were to close a file, it could still be open in another process.

File Descriptor Redirection:

Processes do not share file-descriptor tables.

- A child process will copy over the parent's file-descriptor table.
- If the parent adds a new entry to the file descriptor table after forking, the child's table will not contain that new entry.

It is possible to have a file descriptor point to a different location (file or terminal) using dup2.

- Function signature: `dup2(old_fd, new_fd)`
- old_fd is the file descriptor that new_fd will be redirected to. (In other words, new_fd will change so that it points to the location that old_fd is pointing to, while old_fd does not change)
- After calling dup2, if you close old_fd, new_fd remains open
    - This is because there is one instance of the file (or pipe) systemwide, and we are only closing a single access point to the instance.

3) dup2
Consider the following program. What gets written to the file, and what gets printed to the terminal?

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd); // fd is redirected
     wrapped_write(fd, "dust"); // helper function to write string to a
fd
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";
}
```

begin.txt contains:
star
platinum

what was printed:
dust
crusader