5950 Section 9 - C++ Smart Pointers and Fork

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

C++ Smart Pointers

C++'s smart pointers can be used to automatically manage memory if used properly.

- ***std::unique_ptr*** – `.get()`, `.release()`, `.reset()`
- ***std::shared_ptr*** – `.get()`, `.use_count()`, `.unique()`
- ***std::weak_ptr*** – `.lock()`, `.use_count()`, `.expired()`

1) "Smart" LinkedList

Consider the `Node` struct below. Convert the `Node` struct to be "smart" by using `shared_ptr`s.

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}

  ~Node() { delete value; }

  T* value;
  Node<T>* next;
};
```

After the conversion, we should be able to get rid of the destructor and the following program that uses this `Node` struct should have no memory leak. (Note that we never called `delete` ourselves!) Try checking that your "smart" node doesn't leak memory!

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
  shared_ptr<Node<int>> head =
                      shared_ptr<Node<int>>(new Node<int>(new int(351),
                nullptr));
  head->next = shared_ptr<Node<int>>(new Node<int>(new int(333), nullptr));
  shared_ptr<Node<int>> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Processes & IPC

Process and Threads:

- A process has a virtual address space. Each process is started with a single thread but can create additional threads.
- A thread contains *a* sequential execution of a program and is contained within a process.

Process Functions:

There are a variety of functions commonly used with processes:

- `pid_t fork()`
  - Creates a new process, returning 0 to the newly created child process and the pid of the child process to the parent process.
- `void exit(int status)`
  - Exits the currently running process with specified status
- `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
- `pid_t wait(int* wstatus)`
  - Waits for any child process to exit. Gets their status through the output parameter `wstatus`.
- `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.

Process and Files:

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.

2) Fork Pipe

Consider the incomplete program below. This is a simplified version of some of the lecture code, where we are trying to write a program that makes use of fork(), exit(), waitpid(), execvp() and pipe() to fork a process running the numbers program and feed in user input from the parent process. Fill in the necessary blanks below to complete the program.

```
// writes the contents of the specified string to the specified fd
void wrapped_write(string to_write, int fd);

int main (int argc, char** argv) {
  // create a pipe to send input to program
  int in_pipe[2];
  pipe(________);

  pid_t pid = fork();

  if (pid == 0) {
     // child
     close(______________); // close write end

     // replace stdin with read end of pipe
     dup2(_______________, STDIN_FILENO);

     close(_____________); // close read end since it has been duplicated

     // exec the program "./numbers" with no command line args
     string command(_______________);
     char* args[] = {_________________________________________};
     execvp(_______________, ___________________);

     // should NEVER get here
     return EXIT_FAILURE;
  } else {
     close(__________________________); // close read end

     // write inputs to the pipe
     string inputs = "30\n40\n50\n6";
     wrapped_write(to_echo, __________________);

     // close pipe so that exec'd
     // program knows there is no more piped contents to read
     close(____________________);

     // wait for child to finish
     waitpid(__________________);
  }

  return EXIT_SUCCESS;
}
```