Overview:

The goal of this lab is to design and integrate a comprehensive set of functions and a pipe structure within the operating system to support the capability of multiprocessing.

Major Parts:

- Synchronization: Add locks for lab1 syscall functions to ensure concurl rent access and modification of shared data structures. This will involve the use of semaphores, mutexes or other synchronization mechanisms.
- SysCalls: Implement a set of system call functions, including fork(), wait(), exit() and exec() to enable multiprocessing. These functions will be responsible for creating, terminating and managing processes and their execution.
- Pipe: Implement a pipe structure to allow communication between parent and child process. This will involve implementing the pipe() function, which creates a new pipe and returns the file descriptors for the read and write ends of the pipe, and the read() and write() functions for communicating through the pipe.

How different parts of the design interact with each other

fork(), wait(), and exit() will all interact with each other. fork() will create new processes and exit() will end these processes. wait() will be called after fork() to wait for child process to exit()

The global file struct and sys_write/sys_read will interact with the pipe buffer. The pipe buffer will be treated similarly to a file. Where the user will get a file descriptor to read/write to a file buffer.

Synchronization Issues with fork(), wait(), exit()

The goal of fork() is for the parent process to make a clone to be able to have a copy of the process. Although the instructions will be the same, the flow of execution can be different depending on the PID, where fork() returns the PID of the child process and the child process gets a PID of 0

The goal of wait() is for the parent process to halt instruction until any child process finishes. The parent process will clean up child's PCB, page tables, and kernel stack. In the case wait() is not called, we will consider how these cases will be handled

The goal of exit() is to halt the process and clean up as much resources as possible. The process will let itself be known that it is finished.

In-depth Analysis and Implementation:

We'll to implement wait(), fork(), exit(), exec() and pipes:

wait()

- Halts the execution of code until any child process is finished executing
- Must acquire spin lock to the ptable when checking child process's state
- Parent cleans up child process on wait()
- Need to free child's PCB, page tables, kernel stack
- wait() will do a sleep lock to wait for child process
    - Ex: while (child->state != ZOMBIE) sleep();
- Returns pid of a finished child process. Returns -1 if there are no child processes (whether all finished or none were called)

fork()

- Copy struct context and virtual memory
- Must acquire spin lock to the ptable when creating a new process
- Use vspaceinstall() and vspaceinstallkern()
- Will have the exact fd pointing to the same file_struct. We will have to increase the # of references
- A new entry in the process table must be created via allocproc
- User memory must be duplicated via vspacecopy
- The trapframe must be duplicated in the new process
- All the opened files must be duplicated in the new process (not as simple as a memory copy)
- Set the state of the new process to be RUNNABLE
- Return 0 in the child, while returning the child's pid in the parent

exit()

- exit() will call wake() inorder to let parent process know we are finished if they called wait
- In the case that wake() is called and the parent process never calls wait(),
- Set state to ZOMBIE and free up all memory, leave rest to the next process to clean up

exec()

- Overwrite current process and start executing the new process
- Create a new virtual address space and register state
- Call vspaceloadcode to read and load programme
- Call vspaceinitstack to Initialize the user stack
- Call vscpacewriteova to copy the argument of executing process
- Do not return on success

Pipes

```
struct pipe {

    char *buffer;

    int read_pos;

    int write_pos;

    int max_size;

    struct spinlock lock;

};
```

- Enable intercommunication between parent and child processes
- Runtime allocation using kalloc()
- Allocate 4KB as a bounded buffer for the pipe struct
- Free the pipe when the reference counts for both the read end and the write end drop to 0
- Add spinlock to the pipe struct to ensure synchronization
- Add condition variables to the pipe struct to indicate whether the pipe is open for write or read

- Implement piperead file.c to read from the pipe buffer as opposed to concrruent_readi()
- Implement pipewrite in file.c to write to the pipe buffer as opposed to concurrent_writei()
- Modify sys_write, sys_read, sys_dup, sys_close to support pipes

We'll reimplement syscalls from lab1 to support concurrency and pipes:

- Add spinlock to the global file array. Implement the logic in the syscall functions in sysfile.c to acquire the lock when they want to access the file array, otherwise, they will get blocked.
- Implement the logic to determine if the passed in file descriptor is a pipe struct. Sys_write will need to write to the pipe when there is enough space in the buffer, if not the syscall should be blocked until the buffer is freed. That means we require some mechanism, for example, a conditional variable. If the buffer is full, then the process can call sys_wait on the conditional variable to block sys_write. Or we can use spinlock such that when sys_write fail to acquire spinlock, it will be blocked. Same logic applies to sys_read, when there is nothing left to read, we will block the sys_read until there is data in buffer. For sys_dup and sys_close, we need to decrement and increment the reference count for our pipe struct.

Risk Analysis:

Unanswered questions

- In the case the parent process does not call wait(), do we it up to the next process to clean up?
- What does it mean to clean up a child process? What files do I have to know about?
- How is a pipe implemented? Will we have a file_struct in the global file array to write to a pipe or do we need something entirely separate.
- Do we need to add more to our file_info struct to be able to hold pipes?

    - When to free the lock for pipe buffer when the pipe is filled? How do we free up the space?
    - How we distinguish if we are writing or reading to/from a file struct or a pipe?

Staging of work

We'll fix the synchronization of syscalls from lab1. After that we implement fork(), exit(), wait(), exec(). Then we will implement the pipe struct. Finally, we implement sys_pipe() and modify the syscals from lab1 to support pipes.

Time estimation

- Lab1 reimplementation: 4-7 hrs
- sys_fork(): 3-5 hrs
- sys_wait(): 3-5 hrs
- sys_exit(): 3-5 hrs
- sys_pipe(): 3-5 hrs
- sys_exec(): 3-5 hrs