Travis McGaha & Joel Ramirez Summer 2025
CIS4480

CIS4480 Midterm Exam

This is an open-note, open-calculator, open-computer, open-internet exam.

If you provide an incorrect answer, any explanation about how you arrived at your solution may lead to partial credit.

Full Name [print]: _____

PennID: _____

Exam Details & Instructions

- The exam consists of 5 questions (and a short bonus 6th question) worth 100 points total.
- You have 90 minutes to complete the exam.
- The exam is open book and open internet.
- Use of AI tools is strictly prohibited.

Pledge: I neither cheated nor helped anyone else cheat on this exam. All answers are my own. Violating this pledge may result in a failing grade.

Sign Here: _____

Advice

- There are 5 questions (with 14 parts total). Budget your time to complete all of them.
- Don't worry if there is more space than you need for answers. Extra space is provided just in case.
- Take a deep breath and relax. A bad grade on this exam is not the end of the world. You can improve your score with the Midterm "Clobber" Policy (see the course syllabus).

Please initial or write your PennID on each page to ensure they can be matched if they become separated.

The last page is blank for extra space. If you use it, indicate clearly where your answer continues and include your full name and PennID at the top.

1 Files and File Descriptors [25 Points]

Take a look at the following block of code, which forks two child process, establishes a pipe between the two children, dup2s a file descriptor, and calls execvp in both of the children. After being compiled, the program is executed in the shell. **You may assume all functions are successful.**

```
1
2  int main(){
3
4      int fd_in = open("in.txt", O_RDONLY);
5      // LINE 5
6      int pipe[2];
7      pipe2(&pipe, O_CLOEXEC);
8
9      pid_t pids[2];
10     pids[0] = fork();
11
12     if(pids[0] == 0){
13         dup2(fd_in, STDIN_FILENO);
14         dup2(pipe[1], STDOUT_FILENO);
15         close(fd_in);
16         execvp(...);
17     }
18
19     close(fd_in);
20
21     pids[1] = fork();
22     if(pids[1] == 0){
23         dup2(pipe[0], STDIN_FILENO);
24         execvp(...);
25     }
26
27     waitpid(pids[1], NULL, 0);
28 }
```

Problem continues onto the next page

(a) How many descriptors? [5 Points]

By the time we reach `line 5` (just before the pipe), **how many file descriptors are in use? Justify your answer** by listing each one by its standard name or the file it refers to.

(b) How many now? [7 Points]

By the time both children have exec'd (but have not finished running), **how many *file descriptors are open in total across both children?* Do not take into account the parent. You may assume the children do not modify their own descriptors or open any files after exec'd**

(c) Closing Time [8 Points]

You attend office hours to review the code above and sit between two other students, Mireya and Shayla. However, you notice that, at times, the program is stalled. You want to know **the minimum number of file descriptors to close** to keep the program from hanging.

Mireya says that just before calling `waitpid`, you should close the **write end** of the pipe to prevent processes from getting stuck. **The read end doesn't matter in this case.**

Shayla argues that you should **absolutely close both the write and read ends** of the pipe before calling `waitpid`.

Who is correct? Explain your reasoning.

(d) One waitpid? [5 Points]

After applying your fix and closing all file descriptors, you see that there's *only one waitpid call*. It might look like a race condition, but assume neither child exits abnormally. **If both exec'd programs are `cat`, explain why having just one waitpid is okay here.**

To help conceptualize, think of it like running this in the shell:

```
/workspace# cat < in.txt | cat
```

2 Time to Shreddddd [15 Points]

Take a look at the code below—this is a basic outline of a shredder implementation from a different version of the course, where **functionality within signal handlers is not allowed**.

Only the child process uses `sigaction`, but it doesn't install a real handler. When the parent receives a `SIGALRM`, it simply forwards that signal to the child to terminate it (as shown in part (b)). **You can assume the parent is always able to forward the signal when received.** However, as you'll see, there are some issues with this approach.

```
pid_t child_pid = 0;

void forwardAlarm(){
    kill(child_pid, SIGALRM);
    writeToStdout("Bwahaha ... tonight I dine on turtle soup\n");
}

void executeShell(int timeout){

    struct sigaction sa_ignore = {
        .sa_handler = SIG_IGN
    };

    command = getCommandFromInput();

    if (command != NULL) {
        child_pid = fork();
        alarm(timeout);
        ....
        if (childPid == 0) {
            sigaction(SIGALRM, sa_ignore);
            fork_exec(command); //assume this works
        }
        if (childPid != 0) {
            waitForChild(); //assume this works
        }
    }
}
```

(a) The Disposition of SIGALRM [5 Points]

When the parent receives a SIGALRM a problem occurs. The parent forwards the `SIGALRM` to the child but the child does not terminate from the signal. As a result, the `waitForChild()` call doesn't return (unless the child exits some other way, e.g. it exits normally).

Explain why this happens and suggest a possible fix.

(b) Waiting for Child [10 Points]

Consider the following possible implementation of the `waitForChild` function. You can assume all children run to completion and are never stopped or suspended. **You may refer to the `sigwait` man page for more details. Assume the issue from part (a) is fixed.**

```
void emptyHandler(int signal){ return; }

int blockSignals(){
    // <SIGALRM & SIGCHLD are blocked using sigprocmask>
}

void waitForChild(pid_t child_pid){
    blockSignals();

    //installs the emptyHandler for the SIGCHLD signal
    install_sigchld_handler();

    int sig;
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigaddset(&set, SIGALRM);

    while( (sigwait(&set, &sig)) == 0) ){
        if(sig == SIGCHLD){
            alarm(0); //cancel alarm just in case.
            break;
        } else {
            forwardAlarm();
        }
    }
    wait(NULL);
}
```

In lecture, we mentioned the importance of using the `SA_RESTART` flag when setting up signal handlers. However, you may have noticed that it wasn't used in this implementation. **Based on the `waitForChild` function above, explain why `SA_RESTART` is not needed here.** In other words, **why is `wait` guaranteed to return immediately?**

3 Inodes and Calculations [20 Points]

As mentioned in lecture, a lot of work in a file system involves calculating the offsets of Blocks, Inodes, and byte locations within files. Path resolution is also a key part of this process. Let's walk through a few examples together.

You may assume the following for this question.

- **Inodes Block Start:** Block 2
- **Inode size:** 64 bytes
- **Block size:** 1024 bytes
- **Block number size:** 2 bytes
- **Block pointers in inode:**
  - 12 direct
  - 1 singly indirect
  - 1 doubly indirect
  - 1 triply indirect

(a) Best Case Scenario [4 Points]

You're resolving the path: `/usr/pennstdnt/shell.txt`. In the best case, how many total blocks must be read **to find the inode number** for `shell.txt`? Assume the inode for the root directory has already been loaded. **Briefly explain your answer.**

(b) Worst Case Scenario [6 Points]

You're resolving the same path: `/usr/pennstdnt/shell.txt`. In the worst case, how many blocks must be read **to find the inode number** for `shell.txt`? Again, assume the root inode is already in memory. You may leave your answer as an expression. **Briefly explain your answer.**

(c) Locating the Inode [6 Points]

You've found that the inode number for `text.txt` is 6723. Which block contains this inode, and what is the index (or offset) of the inode within that block? Show your work.

(d) Locating the Byte [4 Points]

Using the inode from the previous part, determine **which data block of the file** (i.e., the `i`th block containing the file's actual contents) is needed to access `byte 102345`.

For example, `byte 999` is in the **0th data block of the file**, and `byte 12288` is in the **11th data block of the file** (which is the last direct block).

In addition to the block number within the file, also determine the **path required to locate that block via the inode** — whether it is accessed through a **direct pointer**, or via a **singly, doubly, or triply indirect pointer**.

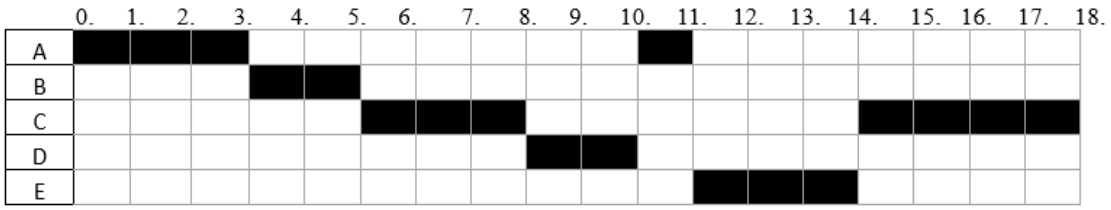
You may leave your answers as expressions if convenient. **Show your work.**

4 I want to schedule more sleep time [26 Points]

In the class, we scheduled the following processes:

Process	Arrival Time	Duration
A	0	4
B	1	2
C	2	7
D	3	2
E	4	3

Scheduling this with a quantum of 3 with round robin algorithm results in 6 context switches. See the diagram below:



In this scenario, we made the following assumptions:

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time quantum finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.
- If a process would not finish before the time quantum is up, and there are no other processes to schedule, then the process runs for another time quantum without context switching.

Let's make slight modifications to this scenario and see its impacts on the scheduler.

(a) Let's Schedule It [16]

Schedule the same tasks that we ran using round robin, **but with a time quantum of two**. We specifically want to see the time each task finishes. **Make sure to justify your answer by showing work.**

No More Assumptions

In our problem, we made several unrealistic assumptions about the nature of the scheduler and the processes it runs (For example: ”All processes do not block for I/O or any resource” and ”Context switching and running the Scheduler are instantaneous”.) Real schedulers would have to consider this in their design.

Let’s explore one advantage and one disadvantage in choosing a shorter time quantum instead of a longer time quantum for a real scheduler.

(b) Pros [5 Points]

In general (not specific to this scenario), what is one advantage to choosing a shorter time quantum instead of a longer time quantum for a real scheduler? **Justify your answer.**

(c) Cons [5 Points]

In general (not specific to this scenario), what is one disadvantage to choosing a shorter time quantum instead of a longer time quantum for a real scheduler? **Justify your answer.**

5 Threads of Sanity [12 Points]

Consider the following code that utilizes threads:

```
//global
bool flag = false;

void* thrd_fn1(void* arg) {

  if (flag) {
    printf("foo\n");
    flag = false;
  } else {
    printf("bar\n");
    flag = true;
  }

  return NULL;
}

void* thrd_fn2(void* arg) {

  flag = true;

  return NULL;
}

int main() {
  pthread_t thd1;
  pthread_t thd2;

  pthread_create(&thd1, NULL, thrd_fn1, NULL);
  pthread_create(&thd2, NULL, thrd_fn2, NULL);

  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);

  if (flag) {
    printf("flag is true\n");
  } else {
    printf("flag is false\n");
  }
}
```

Assume that the code compiles and that none of the functions fail. Additionally, there are no hardware or compiler optimizations that occur. Using only what we have learned so far about threads in class, answer the questions on the next page.

(a) False Flag [5 Points]

Is it possible for main to print ”flag is false”? If possible, describe the steps necessary in the programs execution to achieve this. If not possible, briefly justify why. **We are not looking for a formal proof nor is the output of thread 1 relevant.**

(b) Foo True [7 Points]

Is it possible for thread1 to print ”foo” and main to print ”flag is true”? If possible, describe the order of thread execution to achieve this. If not possible, briefly justify why. **Again, we are not looking for a formal proof.**

6 The Last Question [2 Points]

What is one bug that you and your partner got stuck on in Penn-Shell? What was the source of the bug? How did you fix it?

Did you learn anything from that bug? **Or, you could just write (or draw) anything you want here. :)**

Scratch Work Page