Travis McGaha & Joel Ramirez Spring 2025
CIS4480

CIS4480 Midterm Exam Solutions

This is a closed calculator/computer exam. You may consult one double-sided sheet of notes. No other references are allowed.

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 closed book. No textbooks, phones, laptops, wearable devices, or any other electronics are allowed beyond the permitted notes.
- You may use one 8.5" x 11" double-sided sheet of paper for notes.
- Turn off and put away all electronic devices and noise-making items.
- Remove all hats, headphones, and watches.

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, Files, and Files [17 Points]

Take a look at the following block of code, which forks a child process, establishes a pipe between the parent and child, and transmits the entire contents of a file through the pipe using the function `write_file_pipe`.

You may assume that your PennShell is the only running process that references the controlling terminal prior to it executing the following code as a **single job.**

```
1  const char *argv[] = {"./myprogram", NULL};
2
3  int main(){
4
5      char *file_name_str = "my_file.txt"; // question 2.3 getline(....);
6      int file_fd = open(file_name_str, O_RDONLY);
7
8      int pipe_arr[2];
9      pipe2(pipe_arr, O_CLOEXEC);
10
11     pid_t child = fork(); //Line 11
12
13     if(child == 0){
14         close(file_fd);
15         dup2(pipe_arr[0], STDIN_FILENO); //line 15
16         execvp(argv[0], argv);
17         exit(EXIT_FAILURE);
18     }
19
20     close(pipe_arr[0]);
21
22     write_file_pipe(pipe_arr[1], file_fd);
23
24     close(file_fd);
25     close(pipe_arr[1]);
26
27     waitpid(-1, NULL, 0);
28     printf("Finished waiting for child. \n");
29
30 }
```

Problem continues onto the next page

(a) How many descriptors? [5 Points]

The **job** is executed by your PennShell **in the foreground**. By the time we reach `line 11` (i.e. right before the fork but haven't executed it), **how many file descriptor(s) are in use by this foreground job? Justify your answer.**

> **Solution.** At the start, we have three in usage, STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO. With the open call, we add one more for the file. And finally, the pipe requires two. So in total, 6.

(b) How many can write? How many can read? [6 Points]

The **job** is executed by your PennShell **in the foreground**. By the time `line 11` (i.e. the fork) is executed successfully, **how many *processes* are allowed to write to the terminal? How many are able to read from it? Justify your answer. Assume the parent and child have not gone further than the fork.**

> **Solution.** First, we count the total number of processes running within this session:
>
> 1. **Penn-Shell** (the shell)
> 2. **The Parent process** (which is the original job)
> 3. **The Child process** (which is forked from the original job)
>
> Since the child process is created via `fork()`, it inherits the same Process Group ID as its parent.
>
> By default, any process within the same session can print to the terminal. Therefore, a total of three processes can write to it, including Penn-Shell, the parent process, and the child process. For example, executing `cat file.txt &` runs without issue because background processes can write to the terminal.
>
> Since the job is running in the foreground, the entire process group retains terminal control via `tcsetpgid`, allowing them to read from the terminal without issue. However, this terminal control handoff is managed by the initial shell process (Penn-Shell). Consequently, Penn-Shell itself is not allowed to read from the terminal unless it explicitly regains control using its own `tcsetpgid` call.
>
> Thus, only two processes—the original job (the parent) and the child—can read from the terminal.
>
> *NOTE:* **This exam does not focus on extremely minute details. Even if the PennShell implementation ignores or blocks `SIGTTIN` and/or `SIGTTOU`, this does not mean it will still be able to read from the terminal. While the process may ignore or block `SIGTTIN` sent by the kernel, the `read` operation will still fail with an `EIO` error.**
>
> If the response states that "three processes can read because PennShell ignores/blocks `SIGTTIN`," full credit will still be awarded even if it is wrong.

(c) get(in)line PennShell Partner [6 Points]

Your PennShell partner wants to make your life harder! (Go figure.) Instead of hardcoding which file to open, they suggest using `getline(...)` to prompt the user for a filename. They then modify the code accordingly and run the job **in the background**.

What should happen when this job is run? **Explain.**

> **Solution.** The key here is knowing that using `getline` you are reading from STDIN which by default refers to the terminal for the session. When a background process attempts to read from the terminal that it does not have control of (via `tcsetpgrp`) then it is sent a SIGTTIN from the kernal and it is stopped. As long as you said it stops then you get full points!

2 A Set of Signals is Called a `sigset_t` [16 Points]

As a senior software engineer at your highly lucrative company, you're responsible for mentoring an intern. Additionally, your team is in charge of developing a ***New Operating System***. The ***New Operating System*** makes preemption of a task more easily controllable by user code by having the kernel send two new signals, `SIGSLEEP` and `SIGWAKE`, **at any time**. These signals cause a process to suspend itself immediately (`SIGSLEEP`) or resume execution (`SIGWAKE`) ***under all circumstances***. While these signals can be blocked, they ***cannot be 'ignored'***.

A `SIGWAKE` is **always preceded** by a `SIGSLEEP`. If a process receives a `SIGSLEEP`, it is guaranteed to receive a `SIGWAKE` at some unknown future time. If both SIGWAKE and SIGSLEEP are pending signals (blocked), they are not delivered to the process and are discarded. Additionally, **the only way to continue a process suspended via SIGSLEEP is with a SIGWAKE.**

Your intern writes this code to demonstrate that signals work in the new operating system. The code overview is as follows:

- Install a handler for `tested_signal`.
- Block the signal, `tested_signal`.
- Send the process that signal using `kill`.
- Unblock the signal.
- Install a handler for `SIGALRM`.
- Call `wait_for_alarm` to wait for the alarm that the `tested_signal`'s handler registered.

```
void signal_handler(int sig); //Calls ALARM(10);
void alrm_signal_handler(int sig); //Writes to the terminal "Alarm caught!".

void test_signal_mask(int signal) {
    sigset_t new_set;
    sigemptyset(&new_set);
    sigaddset(&new_set, signal);
    sigprocmask(SIG_BLOCK, &new_set, NULL);
    kill(0, signal); // Send the signal to itself
    sigprocmask(SIG_UNBLOCK, &new_set, NULL);
}

//This function should RETURN ONLY IF and WHEN A SIGALRM IS RECEIVED.
// An alarm should be registered (by the tested signal's handler) to make this function return.
void wait_for_alarm() {
    sigset_t set;
    sigfillset(&set); //Adds all signals to the set
    sigdelset(&set, SIGALRM);// Removes SIGALRM from the set
    sigsuspend(&set); // suspend execution with a specified mask till a signal is recieved
}

int main(){
    int tested_signal = SIGINT; // Can be most signals. NOT SIGALRM, SIGSLEEP or SIGWAKE
    install_handler(tested_signal, signal_handler); // installs handler with sigaction
    test_signal_mask(tested_signal);
    install_handler(SIGALRM, alrm_signal_handler);  // installs handler with sigaction
    wait_for_alarm();
}
```

Your intern's code is supposed to always print 'Alarm caught!' and then terminate. However, after executing your intern's code, sometimes 'Alarm caught!' is correctly written to the terminal and exits; at other times, the program is terminated seemingly at random, and sometimes it hangs after printing 'Alarm caught!'."

What seems to be the issue(s) in the intern's code? **Explain and give the possible fix(es). You can assume that the code compiles and that the only signals blocked in the handlers are the signals that triggered them.**

Solution. This is more of a walkthrough. A detailed description was not expected on the exam. There are three possible outcomes; let's examine them:

'Alarm caught!' is correctly written to the terminal and exits. If something can execute correctly without modifying the code, that strongly indicates a ***race condition***. The key factor here is the ***timing*** of execution. The other two outcomes provide more insight into this behavior.

"...the program is terminated seemingly at random." The only sources of *randomness* in this context are `SIGWAKE` and `SIGALRM`. While kernel/OS scheduling introduces some variability, the alarm is set for 10 seconds—a long duration from the computer's perspective—so it is reasonable to assume that the program reaches `sigsuspend` before the alarm expires. If the program terminates at random, only one signal can be responsible: `SIGALRM`.

`SIGSLEEP` and `SIGWAKE` do not terminate processes; they only stop or resume them. However, `SIGALRM` has a default disposition that terminates the process upon reception. A handler is installed for `SIGALRM`, but the crucial question is: **when is it installed?**

Since the handler is installed *after* the alarm is set, there exists a period where the alarm is active, but the default disposition has not yet been overridden. Additionally, the interval between `SIGSLEEP` and `SIGWAKE` is both unknown in duration and timing. A `SIGSLEEP` could arrive immediately after a `SIGWAKE`, creating a delay longer than the alarm interval.

Fix: Install the `SIGALRM` handler **before** calling `ALARM(10)`. This ensures that the default termination behavior is overridden by the time the alarm is set.

"...sometimes it hangs after printing 'Alarm caught!'" In this scenario, the `SIGALRM` handler is correctly installed, the `alarm(10)` executes as expected, and `SIGALRM` is received. So, what causes the program to hang?

The only way a process can hang in this case is on `sigsuspend`. However, `SIGALRM` should allow the program to exit from `sigsuspend`. The issue arises when the `SIGSLEEP - SIGWAKE` interval is longer than the alarm interval and occurs *before* the program enters `sigsuspend`.

If the alarm triggers before reaching `sigsuspend`, the signal is handled, and the program proceeds. But because `sigsuspend` blocks everything except `SIGALRM`, and `SIGALRM` has already been received, the program will never receive another one. As a result, it remains indefinitely suspended.

Fix: Block `SIGSLEEP` and `SIGWAKE` **before** calling `ALARM(10)` and unblock them **after** exiting `sigsuspend`. This prevents unwanted delays that might cause the alarm signal to be handled too early.

Expected student solution:

The presence of a race condition prevents `SIGALRM` from being properly synchronized with the rest of the program. The desired behavior is for `SIGALRM` to trigger its handler rather than its default termination behavior, which explains the random terminations. To ensure this, the `SIGALRM` handler should be installed **before** calling `ALARM(10)`, ideally at the same time as installing the handler for the tested signal. *continues on next page.*

Additionally, the program might hang if it reaches `sigsuspend` after the alarm has been set, the handler installed, and the alarm triggered. This situation only occurs if `SIGSLEEP` and `SIGWAKE` introduce a delay longer than 10 seconds between setting the alarm and entering `sigsuspend`.

Fix: Block `SIGSLEEP` and `SIGWAKE` before calling `ALARM(10)` and unblock them after exiting `sigsuspend`. With these changes, the program functions correctly.

3 Extending the Linux File System ext2 [24 Points]

We have seen that filesystems, such as ext2, are more complex than they initially appear. In this question, you will design an enhanced version of ext2 called *PennExt*, focusing on secure block management.

(a) Why Wipe Blocks? [3 Points]

Before you begin designing PennExt, it is crucial to consider the implications of leaving old data intact on newly allocated blocks. When implementing a kernel, your job isn't to just write a kernel that maintains processes and files, but also to ensure the security within processes and file accesses.

What security and privacy risks could arise if blocks are not wiped before being reallocated to a new file?

> **Solution.** Allocating blocks that were previously used by other files without clearing them poses a security and privacy risk, as those blocks may still contain sensitive information. This risk is especially concerning in environments where multiple users share the same file system or storage device.
>
> Without a deep understanding of the file system interface, one could speculate that it might be possible to increase the size of a file without writing to it and then read existing data left in those blocks. Regardless of the specific mechanism, users should never be able to access data that does not belong to them (or more specifically, do not have permission to read from).

(b) Free Blocks [7 Points]

In a basic design, free blocks could be tracked with a bitmap spanning multiple blocks. However, you are restricted from introducing any new data structures and must rely on ext2's existing reserved inodes (10 in total, with the second inode always being the root).

How can you leverage these reserved inodes to manage free blocks? Where exactly would these block references be stored, and how would you maintain and update them?

> **Solution.** Since there are nine free reserved inodes, one or more of them could be used to track blocks that have become unused. Typically, inodes track data blocks associated with regular files, such as 'generic.txt'. However, nothing prevents the creation of a hidden file (e.g., `unusedblocks.inode`) managed by the kernel and file system.
>
> This special file would store the block numbers of unused blocks. When a block becomes unused, its number would be appended to this file. When an unused block is needed, the required number of free block numbers can be read from this file, allocated to a new file, and then removed from the list. The data blocks of this special file can function as singly indirect blocks containing references to unused data blocks, mimicking an inherent aspect of the filesystem without the need to add any superfluous data structures or abstractions.

(c) When Writing, How Long Do You Wait? [3 Points]

Files can be truncated or deleted in several ways, such as by calling `OPEN` with the `O_TRUNC` flag, which reduces a file's size by truncating it, or by using the `ftruncate` function, which you may not have encountered before.

In your design, blocks must be wiped using `erase(block_number)` before being returned to the free list. Since there is only a single structure tracking free blocks, blocks must be erased and re-added to the free list immediately; otherwise, they will be lost.

Assume that when a file is truncated (reducing its number of blocks) or deleted, its blocks are **immediately erased and returned to the free list**. **Why might this approach be suboptimal?**

> **Solution.** This approach is suboptimal because it significantly increases overhead when a file is truncated or resized to a smaller size. The waiting time for `I/O` operations increases further if blocks need to be cleared before being added to the unused-block structure. File systems and kernels must remain responsive and efficient and not block for longer than absolutely necessary. Additionally, users should not experience any noticeable delays due to these behind-the-scenes operations, which should remain abstracted away.

(d) But, is there a solution? [9 Points]

You can mitigate the trade-off by using the kernel **when it is idle**. The kernel can execute a `synchronize()` function that synchronizes the file system so that all unused blocks that previously held file data are wiped and added to your free list design from **part (b).**

What would you need to add to your design in (part b) to make this `synchronize()` feasible? Explain how your approach alleviates the drawbacks from the previous question.

> **Solution.**
>
> In **part b**, the design used a reserved inode (or a set of them) to track unused data blocks. The data blocks of these reserved inodes acted as singly indirect blocks, storing the block numbers of available but uncleared blocks.
>
> To introduce the functionality of ***clearing*** unused blocks in the background—either when the kernel or file system is idle—and synchronizing the file system efficiently, two separate inodes (or sets of inodes) must be used to maintain two distinct lists:
>
> 1. Blocks that have been recently marked as *free/unused* but are still uncleared (i.e. *dirty*).
> 2. Blocks that have been ***cleared/empty/ready-to-use***.
>
> To achieve this, an additional hidden file, `dirty_unused_blocks.inode`, can be introduced. This file's data blocks serve as singly indirect blocks containing the block numbers of *dirty* (uncleared) unused blocks. The original hidden file from **part b** can be renamed to `cleared_unused_blocks.inode`, where its data blocks continue to store the numbers of cleared and available blocks, meaning they are ready to be allocated to files.
>
> When a block is newly marked as unused and is still dirty, it is appended to `dirty_unused_blocks.inode`. This approach significantly reduces overhead, as it simplifies to appending data to a file rather than immediately clearing blocks.
>
> ***continues on next page.***

When the kernel or file system is idle (or if the number of cleared and unused data blocks reaches zero), `synchronize()` can be executed. During this process:

1. The data blocks in `dirty_unused_blocks.inode` are read to extract block numbers.
2. Those blocks are cleared.
3. The corresponding entries are removed from `dirty_unused_blocks.inode`.
4. The cleared block numbers are added to `cleared_unused_blocks.inode`, making them available for allocation.

Allocating these blocks to files then simply requires:

1. Reading block numbers from `cleared_unused_blocks.inode`.
2. Removing them from the file.
3. Adding references to these blocks in another inode's data block list.

And with that, the system efficiently manages unused block clearing while minimizing performance overhead. **(We were not looking for submissions that were this verbose, this solution is this long for explanatory purposes in case you wanted to see a more complete response.)**

(e) How do we know what these inodes do? [2 Points]

Finally, before your new system design hits production, you need to indicate the functionality these inodes now implore. Where in the file system would you need to mark this change? **Briefly** explain your answer. *Hint: In which block?*

Solution. We can indicate the change in the super-block, as this is where the information **about** the file system is maintained, including the functionality of the reserved inodes.

4 Threads are cooooooooooooool [10 Points]

Consider the following code.

```c
int global_counter = 0;

void* thread_main(void* arg) {
  while (global_counter < 2) {
    global_counter += 1;
  }
  return NULL;
}
```

Assume that two threads are running the function thread_main and return from the function normally. Additionally, there are no hardware or compiler optimizations that occur.

(a) Minimum [4 Points]

What is the minimum value that global_counter can be after both threads finish? **Briefly justify your answer.**

Solution. The minimum possible value of `global_counter` is 2. This can be deduced from the fact that exiting the `while` loop requires `global_counter` to be at least 2. In a strictly sequential execution, `thread A` would first increment `global_counter` to 2 and exit, after which `thread B` would increment it to 3 before exiting. However, it is possible for `global_counter` to remain at 2 due to interleaving operations between the two threads.

To understand this, consider breaking down the increment operation into its fundamental steps:

1. Load the value of `global_counter` from memory into a register.
2. Increment the value stored in the register.
3. Store the incremented value back into memory at the address of `global_counter`.

Now, consider the following interleaving of instructions between `thread A` and `thread B`, both starting when `global_counter` is initially 1:

```
# Thread A                  # Thread B
LOAD R1, global_counter
                            LOAD R2, global_counter   # Both threads load 1
ADD R1, R1, 1
                            ADD R2, R2, 1             # Both increment to 2
STORE global_counter, R1
                            STORE global_counter, R2  # Both write back 2
```

Both threads independently read the initial value (1) into their respective registers. They each increment their local copy to 2 and then write it back to memory. Since both store the same value (2), the final result in memory remains 2 instead of 3. As a result, both threads exit, leaving `global_counter` at its minimum possible final value of 2.

(b) Maximum [6 Points]

What is the maximum value that global_counter can be after both threads finish? **Briefly justify your answer.**

Solution. Building off the explanation in the previous part, the maximum possible value of `global_counter` is 3. We can see this by coming up with a possible interleaved sequence of instructions between the two threads. Consider the following interleaving of instructions between `thread A` and `thread B`, both starting when `global_counter` is initially 1 and both are at the entry point of the while loop:

```
# Thread A                  # Thread B
LOAD R1, global_counter
ADD R1, R1, 1
STORE global_counter, R1
                            LOAD R2, global_counter   # Thread B load 2
                            ADD R2, R2, 1             # Increment to  3
                            STORE global_counter, R2  # Thread B writes back 3
                                                      # Then it exits.
```

Unlike the previous case, `thread A` completes its increment operation first, updating `global_counter` to 2 before `thread B` executes its load instruction. When `thread B` then loads the value from memory, it sees the updated value (2), increments it to 3, and writes it back. Since `thread B` was the last to update `global_counter`, the final stored value in memory is 3. This interleaving represents the maximum possible final value of `global_counter`.

Again, we weren't looking for justifications that were this verbose. It's here more for completeness.

5 I want to schedule more sleep time [30 Points]

Let's introduce a new scheduling algorithm, **Shortest Remaining Time First (SRTF)**, which is a preemptive version of **Shortest Job First (SJF)**.

Shortest Remaining Time First works as follows:

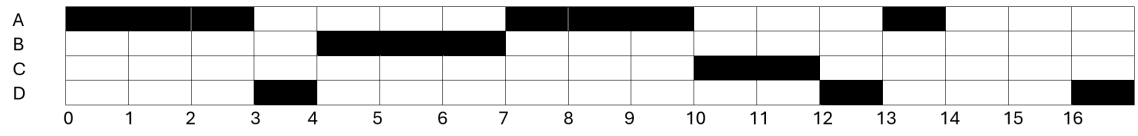
- The first task received becomes the current running task.
- The current running task is the only one that is making progress and thus it's "remaining time" gets smaller (since it is currently being executed)
- The current running task is pre-empted if a new task with shorter remaining time arrives. Otherwise, the run current task runs to completion or till it willingly gives up the CPU.

Consider the table of tasks below:

Task	Arrival Time	Run Time
A	0	7
B	2	3
C	6	2
D	1*	1*

Note that task D is a special "reoccurring" task. It runs for only 1 unit of time before it gets blocked on I/O (e.g. "finishes"). However, after 3 time units of not running it is received as a task again.

If we were to schedule this with the round robin algorithm with a quantum of 3, we get:



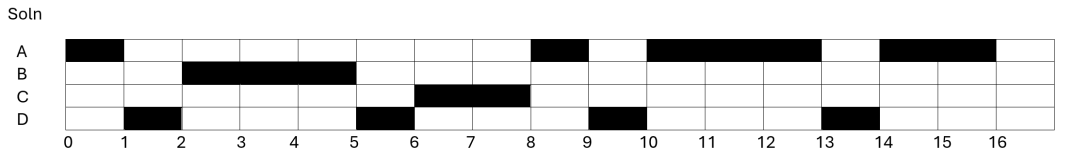
Note how there is a long time between the first time and second time task D ran; it simply became "*runnable*" after 3 time units of not running and was re-added to the queue until its turn.

(a) Let's Schedule It [20]

Schedule the same tasks that we ran with round robin, but this time with SRTF. You can make the following assumptions if needed:

- A task can be scheduled to run as soon as it arrives.
- If there are two or more tasks that the scheduler could run with the same remaining time left, then it chooses the one that comes first alphabetically (e.g. if A and C tied, A would run).

Solution.



(b) Pros [5 Points]

In general (not specific to this scenario), what is one way SRTF is better than round robin? **Please justify your answer.**

> **Solution.** SRTF can be more efficient than Round Robin in scenarios where tasks arrive in bursts with gaps between arrivals — especially when task lengths vary.
>
> If all jobs arrive at the same time and are of equal length (e.g., 9 units), SRTF simply runs each job to completion before moving on. With two tasks, for example, it completes the first in 9 ticks and the second in another 9 ticks, totaling 18 ticks.
>
> Round Robin, however, slices time into fixed quanta (e.g., 3 units). This forces the CPU to frequently switch between tasks, even when it's unnecessary. In the same scenario, Round Robin would complete the first task in 15 ticks and the second in 3 more, still totaling 18 ticks — but with significantly more context switches. This dramatically increases overhead, especially when the quantum is small or when task lengths are highly variable. SRTF reduces this overhead by minimizing the number of context switches, making it more efficient in such cases.

(c) Cons [5 Points]

In general (not specific to this scenario), what is one reason SRTF may be worse than a round robin? **Please justify your answer.**

> **Solution.** One critical reason SRTF is worse than round robin is that ***starvation*** is possible. If there is a Task A that takes 10 quantum to run, but the scheduler receives a multitude of short 1-5 quatumns to run consistently, then there is a good chance that the lengthy job would never run as it has a lower priority that those with a shorter run time.
>
> ***There are multile reasons why it could be worse; however, this it probably the most important reason why it is worse.***

6 The Last Question [3 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. :)

> **Solution.** I cri everi time

Scratch Work Page