CIS 3800 Spring 2024: Midterm

Feb 29, 2024

First Name : _____Adam_____

Last Name : _____Blank_____

Penn ID : _____

Please fill in your information above, read the following pledge, and sign in the space below:

I neither cheated myself nor helped anyone cheat on this exam. All answers on this exam are my own. Violation of this pledge can result in a failing grade.

Sign Here : _____

Exam Details & Instructions:

- There are 8 questions made of 10 parts (and a short bonus) worth a total of 100 points.
- You have 120 minutes to complete this exam.
- The exam is closed book. This includes textbooks, phones, laptops, wearable devices, other electronics, and any notes outside of what is mentioned below.
- You are allowed one 8.5 x 11 inch sheet of paper (double sided) for notes.
- Any electronic or noise-making devices you do have should be turned off and put away.
- Remove all hats, headphones, and watches.
- **Your explanations should be more than just stating a topic name. Don't just say something like (for example) "because of threads" or just state some facts like "threads are parallel and lightweight processes". State how the topic(s) relate to the exam problem and answer the question being asked.**

Advice:

- Remember that there are 8 questions made up of a total of 10 parts (and a short bonus question). Please budget your time so you can get to every question.
- Do not be alarmed if there seems to be more space than needed for an answer, we try to include a lot of space just in case it is needed.
- Try to relax and take a deep breath. Remember that we also want you to learn from this. A bad grade on this exam is not the end of the world. This grade also can be overwritten by a better grade with the Midterm “Clobber” Policy (details in the course syllabus)

Please put your PennID at the top of each page in case the pages become separated.

If you need extra space, the last page of this exam is blank for you as scratch space and to write answers. If you use it, please clearly indicate on that page and under the corresponding question prompt that you are using the extra page to answer that question. Please also write your full name and PennID at the top of the sheet.

Question 1 {15 pts}

For worse or better Python is becoming an increasingly popular programming language and C is used less than it used to be used in the past. Despite this, operating system interfaces are largely written in C, so for python to support some of the functionality it needs, it must invoke some of the system calls we have seen in this class.

Consider the python function subprocess.check_output(). You are not expected to know python for this course, so here is some pseudo-code for how it works

```
string check_output(string program_name)

int main() {
  string output = check_output("grep -r comrade");
  // do something with the output afterwards...
}
```

Put into words: the check_output function runs the specified program and returns to you a string containing the stdout and stderr output for the specified program.

You may use this blank space for scratch work. The questions begin on the next page.

PennID: _____

Question 1 {15 pts}

Which of the system calls below are likely needed to implement the function check_output and how many times do we need each to be called?

System call	Number of times called
fork()	**1**
waitpid()	**1**
pipe()	**1 or 2**
execvp()	**1**
kill()	**0**
signal()	**0**

For each system call, please briefly (3 sentences max) justify why we need it that many times (or why it is not needed in the case of 0)

Reminder: we have an appendix at the end of the exam containing part of the man page for most of these functions

We need to call fork once to fork a process to exec the program we want to get the output from.

Waitpid is needed once to know when process is done

Pipe is needed once to get the stdout of the process. We could use the same pipe and have stderr also print to that pipe or make a separate pipe for stderr output from the process that is executed.

Execvp is needed once so that we can run the program we want to execute.

Kill() is not needed, and neither is signal(). the question mentions nothing about needing to handle any signals.

PennID: _____

Question 2 {15 pts}

Pipes are one method we have seen for Inter Process Communication (IPC). A student wrote the following short program to send a message from the child to the parent, and from the parent to the child, using a pipe. Unfortunately, this implementation is flawed. In a few sentences describe the following:

• Identify a possible flow of execution which results in undesired behavior (Note: desired output would be "Hello! (from parent)" and "Howdy! (from child)")

• Explain why this undesired behavior occurs

• Describe the conceptual misunderstanding of pipes in this code and propose a solution to achieve consistent desired behavior (Note: you do not have to write the code itself, just describing it at a high level is fine)

```
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  int pid = fork();

  if (pid == 0) {
    // Child
    write(pipe_fds[1], "Hello!", 6);
    char str[7];
    ssize_t chars_read = read(pipe_fds[0], str, 6);
    if (chars_read != -1) {
      str[chars_read] = '\0';
      printf("%s (from child)\n", str);
    }
    exit(EXIT_SUCCESS);
  }
  // Parent
  char str[7];
  ssize_t chars_read = read(pipe_fds[0], str, 6);
  if (chars_read != -1) {
    str[chars_read] = '\0';
    printf("%s (from parent)\n", str);
  }

  write(pipe_fds[1], "Howdy!", 6);
  return EXIT_SUCCESS;
}
```

Write your answer on the next page.

Question 2 continued {15 pts}

In a few sentences describe the following:

• Identify a possible flow of execution which results in undesired behavior (Note: desired output would be “Hello! (from parent)” and “Howdy! (from child)”)

• Explain why this undesired behavior occurs

• Describe the conceptual misunderstanding of pipes in this code and propose a solution to achieve consistent desired behavior (Note: you do not have to write the code itself, just describing it at a high level is fine)

The possible issue is that the child could read what it wrote to the pipe. Pipes are meant to be used as a uni-directional channel of inter-process-communication. However, in this example we are using a single pipe to communicate in both directions (parent to child and child to parent). Since the child reads from the same pipe it writes to, the child could read what it wrote to the pipe, thus the data does not go to the parent.

Additionally in this case where the child reads what it wrote, the parent will block waiting for EOF to be indicated from reading the pipe and never terminate. This is because it should have been unblocked from reading what the child wrote, but since it didn’t read it then it will wait for something to be written or EOF. EOF will not be received since the parent still has a write end to the pipe open.

To fix this we should create two pipes instead of just one. One of the pipes would be used for sending data from child to parent and the other for parent to child.

Question 3 {15 pts}

In class we talked about critical sections and how accessing shared resources in the process and the signal handler at the same time can cause that resource to enter an invalid state.

In our examples in class, we demonstrated this by trying to modify a shared global data structure. However, critical sections are possible even just with modifying an integer variable.

Consider this code that declares a global integer variable and a way for that variable to be incremented.

```
int global_counter = 0;

void increment() {
  global_counter++;
}
```

Despite the modification of the integer being 1 line of code, it is actually composed of multiple steps. First we need to load the variable global_counter from memory into a register, increment the value and then store it back into memory.

We can rewrite the increment function in two different ways to better show this is happening:

Assembly pseudo code	C pseudo expansion

`void increment() {`
`  LOAD register global_counter`
`  ADD register, register, 1`
`  STORE register global_counter`
`}`
	`void increment() {`
	`  int local_counter = global_counter;`
	`  local_counter = local_counter + 1;`
	`  global_counter = local_counter`
	`}`

Part 1 {7 pts}

Suppose the function increment() is executed twice, once in a signal handler and once in the main() function of a process. (see the example code below)

```
void handler(int signo) {
  increment();
}

int main() {
  signal(SIGALRM, handler);
  alarm(1);  // set an alarm for 1 second
  increment(); // might be interrupted by alarm handler
  sleep(5);  // try and give time to make sure alarm goes off
             // at some point
  printf("%d\n", global_counter); // print global counter;
}
```

Part 1 continues onto the next page.

Part 1 continued {7 pts}

In the code on the previous page, it's possible for the signal handler to interrupt `main` executing the `increment` function and for us to get an output different than expected (which would be 2).

What is the minimum possible value that global_counter could be left at after both the signal handler and main() have called increment? Please explain how that value is possible

You should assume that none of the functions fail and that the alarm goes off while main is executing the increment function.

Hint: we highly suggest thinking about the assembly and C pseudo code of `increment()`

Note: This answer space is big, but you do not have to use this whole space. We are giving it just in case you need it and because we want the next problem to start on the next page.

1

It is possible that the increment in main() is interrupted and mid increment, the signal handler goes off. This means we could have 1 as the possible value after both increments return.

Consider that first 0 is loaded into a register in the main thread, then the signal handler goes off. The handler increments the global variable to 1 and we return to the main thread which still has 0 in a register. The main thread does not see the updated global variable and thus also writes 1 to the variable in memory.

So even if we increment twice, we may get 1 as the result.
(The explanation does not need to be this detailed, this is just to make it clear why 1 is possible)

PennID: _____

Part 2 {8 pts}

Consider the following solution that tries to fix the critical section:

```
void handler(int signo) {
  increment();
}

int main() {
  signal(SIGALRM, handler);
  alarm(1);  // set an alarm for 1 second
  signal(SIGALRM, SIG_IGN);
  increment(); // might be interrupted by alarm handler
  signal(SIGALRM, handler);
  sleep(5);  // try and give time to make sure alarm goes off
             // at some point
  printf("%d\n", global_counter); // print global counter;
}
```

It is possible that the alarm goes off while main invokes increment, but still under this solution we would not get the expected output (2). In a few sentences, answer the following:

- Why may we not get 2 as our printed output?
- At a high level describe how we would need to change the program to fix this problem? (you do not need to write any code for this)

You should assume that none of the system calls fail and that the alarm goes off while main is executing the increment function.

We may still get one printed if the alarm goes off while the signal is ignored and so the signal is discarded, so the increment from the alarm doesn't run.

a fix would be to instead block the signal temporarily instead of ignoring it. That way when the signal is unblocked, we will still get the alarm.

Question 4 {10 pts}

Oh no! The C standard library implementation of the sleep() function was found to be buggy, and so we need to re-implement it ourselves using the functions we have learned in this class. Nate and Seungmin each come up with their own implementation of sleep().

Nate's implementation:

```
void handler(int signo) {
  // do nothing
}

void sleep(unsigned int seconds) {
  signal(SIGALRM, handler);
  alarm(seconds);
  sigset_t suspend_set;
  sigfillset(&suspend_set);
  sigdelset(&suspend_set, SIGALRM);
  sigsuspend(&suspend_set);
}
```

Seungmin's implementation:

```
bool sleep_done = false;

void handler(int signo) {
  if(signo == SIGALRM) {
    sleep_done = true;
  }
}

void sleep(unsigned int seconds) {
  sleep_done = false;
  signal(SIGALRM, handler);
  alarm(seconds);

  while (!sleep_done) {
    // waiting for alarm to go off so sleep can finish...
  }
}
```

Both of these work, but one of these implementations is generally considered better than the other. Which implementation and why? **Put your answer on the next page**

Note: you should be familiar with sigsuspend and signal, but we've included part of the man page for it in the appendix if you need it.

Remember what we say on the front page about the requirements for an explanation.

Question 4 {10 pts}

Note: This answer space is big, but you do not have to use this whole space. We are giving it just in case you need it and because we want the next problem to start on the next page.

Nate's is better since it doesn't busy-wait for the alarm to go off. Instead it suspends execution and thus does not consume CPU cycles while sleeping. Seungmin's will busy-wait as it loops in a while-loop, consuming CPU time for no reason when you could suspend like nate's solution.

Question 5 {14 pts}

On some systems it may make sense to have multiple pieces of hardware that make up our file system and have different block allocation schemes on each. In this case, some files are stored using contiguous allocation, but for others an Inode-based approach is used. The creator of this system chose this design in order to balance file access times and the overheads of managing larger blocks of data. In particular, minimizing the amount of time we need to seek (reposition) within disk and the number of times we want to access disk in general.

Part 1{7 pts}

The system needs to frequently access some small files on the system. These files will always take up about the same amount of space. These files are usually read from, but sometimes written to. Which allocation scheme (contiguous or Inodes) would work best for storing these files? Please justify your answer.

Contiguous allocation. Since the blocks are stored next to each other we do not need as many seeks to read the whole contents of the file. We also do not run into the issue of growing files and thus needing to move files since we usually are just reading the file.

Part 2 {7 pts}

The system also stores many very large files that stretch several blocks. These files will be frequently modified: contents overwritten, contents re-arranged, the file extended, and the file shortened. Which allocation scheme (contiguous or Inodes) would work best for storing these files? Please justify your answer.

Inodes. This allows for easy reallocation of files in the system and also avoids issues of fragmentation. Since continuous allocation requires the blocks are next to each other, growing a file can be a pain and we may not be able to fill holes resulting from fragmentation.

PennID: _____

Question 6 {10 pts}

We are reading a FAT with the following contents:

Index	Entry
0	SPECIAL / METADATA
1	9
2	4
3	8
4	0xFFFF
5	0xFFFF
6	0
7	0xFFFF
8	0xFFFF
9	7

For this FAT, we will follow what we will do with PennFAT:

- 0 and 0xFFFF are special values. 0 indicates the block is unused, and 0xFFFF indicates that it is the current end block of a file.
- The FAT starts counting from index 1, with the first block being the first block that comes after the FAT stored on disk

We read the root directory and get the following directory entries:

File Name	First Block #
A	2
B	5
C	3

For each of the blocks below, fill in either:

- EMPTY if the block is unused.
- The name of the file the block belongs to.
    - This includes the Root Directory itself.

Your answers go in the box below:

FAT	Root1	A1	C1	A2	B1	EMPTY	Root3	C2	Root2

PennID: _____

Question 7 {10 pts}

We have written a short program that uses fork() and manipulates file descriptors.
Here is the program:

```
int main() {
  pid_t pid = fork();
  if (pid == 0) {
    close(STDOUT_FILENO);
    printf("HELLO!\n");
    return EXIT_SUCCESS;
  }

  dup2(STDOUT_FILENO, STDIN_FILENO);
  write(STDIN_FILENO, "pan", strlen("pan"));

  return EXIT_SUCCESS;
}
```

What are the possible outputs of this program? You can assume that all system calls succeed.
Please justify your answer:

fork creates copies of file descriptors, so parent and child processes have different copies of STDOUT fd.

Child process closes STDOUT, so "HELLO!" is never printed

dup2 replaces STDIN with STDOUT in the parent process, so "pan" is written to STDOUT

Question 8 {10 pts}

Suppose that we have an array of floats that is very large, roughly 1,000,000 of them, and we want to perform several computations on each of those floats.

```
float arr[1000000];

// function declarations. Assume each function is very long
float compute1(float f);
float compute2(float f);
float compute3(float f);
```

We have two different ways of performing the computations on each of the floats:

The first implementation:

```
for (int i = 0; i < 1000000; i++) {
   arr[i] = compute1(arr[i]);
   arr[i] = compute2(arr[i]);
   arr[i] = compute3(arr[i]);
}
```

And the second implementation:

```
for (int i = 0; i < 1000000; i++) {
   arr[i] = compute1(arr[i]);
}
for (int i = 0; i < 1000000; i++) {
   arr[i] = compute2(arr[i]);
}
for (int i = 0; i < 1000000; i++) {
   arr[i] = compute3(arr[i]);
}
```

Our L1 cache is too small to hold the entire array, and the instruction cache cannot hold the code for a single one of the compute functions.

How does the speed of these two implementations compare? Please justify your answer.

The 1st implentation is faster for this operation due to the array data already being in the cache when the functions execute (temporal locality)

Question 9 {1 pt} **<u>all submissions will get this point</u>**

This course has you program in C! Describe your outfit in terms of C code 

If you don't want to do that, then put anything here! What's your favourite thing about C programming? Anything you want to show us or want us to know?

```
int main() {
  char** drip = NULL;
  *drip = "Cause a segmentation fault";
  exit(EXIT_FAILURE);
  pid_t self = getpid();
  kill(self, SIGKILL);
  abort();
}
```

Appendix

Waitpid man page

SYNOPSIS

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Description

This system call is used to wait for state changes in a child of the calling process and obtains information about the child whose state has changed.

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call.

The value of options is an OR of zero or more of the following constants:

WNOHANG

return immediately if no child has exited.

WUNTRACED

also return if a child has stopped.

If wstatus is not NULL, waitpid() stores status information in the int to which it points. This integer can be inspected with the following macros

WIFEXITED(wstatus)

returns true if the child terminated normally, that is, by calling exit() or by returning from main().

WIFSIGNALED(wstatus)

returns true if the child process was terminated by a signal

RETURN VALUE

on success, returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

execvp man page

SYNOPSIS

```
int execvp(const char *file, char *const argv[]);
```

DESCIRPTION

replaces the current process image with a new process image. This causes the program that is currently being run by the calling process to be replaced with a new program specified by the argument file and, that program will have the arguments specified by argv. The process will have a newly initialized stack, heap, and data segments.

RETURN VALUE

does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly.

Returns -1 on error

pipe man page

SYNOPSIS

```
int pipe(int pipefd[2]);
```

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

dup2 man page

SYNOPSIS

```
int dup2(int oldfd, int newfd);
```

DESCRIPTION

The dup2() system call creates a copy of the file descriptor oldfd, using the file descriptor number specified in newfd. If the file descriptor newfd was previously open, it is silently closed before being reused.

kill man page

SYNOPSIS

```
int kill(pid_t pid, int sig);
```

DESCRIPTION

The kill() system call can be used to send any signal to any process group or process. In normal usage, signal sig is sent to the process with the ID specified by pid.

signal man page

SYNOPSIS

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

DESCRIPTION

signal() sets the disposition of the signal signum to handler, which is either SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal handler").

sigsuspend

SYNOPSIS

```
int sigsuspend(const sigset_t *mask);
```

DESCRIPTION

sigsuspend() temporarily replaces the signal mask of the calling thread with the mask given by mask and then suspends the thread until delivery of a signal whose action is to invoke a signal handler or to terminate a process.

If the signal terminates the process, then sigsuspend() does not return. If the signal is caught, then sigsuspend() returns after the signal handler returns, and the signal mask is restored to the state before the call to sigsuspend().

This page is intentionally left blank.