CIS 5480 Recitation 1 - Fork, Wait, and Signals

Welcome to recitation!!! 😃

Name: _____ **PennID: _____**

1 Debugging Resources, Tools, and Techniques

In this class, it is very helpful to be comfortable reading the man pages as well as using valgrind. The man pages tell you all the behavior of system calls and C library functions, and valgrind is useful for catching many types of runtime memory errors.

1.1 The Man Pages

Let's try looking up a couple of different man pages for different functions in C.

- Type man read in the terminal. Do you see what you expect?

- What do you need to type in order to see the correct man page for the C function for read?

- What about man printf? Is it in the same section as read? Why or why not?

Although not all are necessary yet, the main sections that we will use are section 2, 3 and 7. Section 7 is a bit special in that it is the "Miscellaneous" section full of specific details denoting behavior and functionality. For example, in future assignments you will need to use man 7 signal a fair amount. Unfortunately, LLMs still don't have a fair grasp of this section or explain it well, so you'll have to manually check this section quite often.

1.2 valgrind

valgrind will analyze your code during a specific execution of your program, meaning with non-deterministic programs you may get different output each time. The following command will run valgrind on **`program`**:

```
$ valgrind ./program
```

Check out the man pages for more info on additional flags to use with valgrind.

2 Address Space Layout, Arrays vs Pointers, and Permissions

2.1 Where Does This Live?

For this question, we will review the segments of memory in a typical C program. Let's go ahead and get up to speed.
Consider the following code:

```
#include <stdio.h>

int g = 7;

int main(void) {
    char arr[] = "hello";
    char *ptr = "world";

    printf("%c %c %d\n", arr[0], ptr[0], g);
    return 0;
}
```

Question: For each item below, identify where it exists in the process address space layout (stack, heap, .data/.bss, or read-only text/rodata), and briefly justify.

- The variable arr

- The characters that make up the string stored in arr (i.e., the memory backing "hello")

- The variable ptr

- The characters that ptr points to (i.e., the memory backing "world")

- The variable g

3 wait(), waitpid(), and Blocking Behavior

In lecture, we saw that wait() and waitpid() allow a parent to observe child state changes. Whether or not these calls block has major performance and correctness implications.

3.1 Predict Before Running

Consider the following program:

```
int main() {
  pid_t pid = fork();
  if (pid == 0) {
    sleep(5);
    printf("Child done\n");
    return 0;
  }
  printf("Parent waiting...\n");
  wait(NULL);   //   wait for any child to finish
  printf("Parent done\n");
}
```

Question:

• Which line(s) could appear first? Which can never appear first?

• Suppose we remove the wait call in the parent. What happens to the child?

3.2 Busy Waiting vs Blocking

Now consider this version:

```
int status;
while (waitpid(-1, &status, WNOHANG) == 0) {
  printf("Still waiting...\n");
}
printf("Done waiting\n");
```

Question:

• What does it mean when waitpid returns 0 in this loop?

• You realize that the child you are waiting for might never return. How can you terminate it?

4 Signals Galore

4.1 Not Handling Signals in a Signal Handler

Signal handlers can interrupt one another; this can be dangerous. To fix this, you can set the **`sigset_t sa_mask`** field of a **`sigaction`**, which automatically applies when the handler is running.

```
struct sigaction sa_sigusr1 = {.sa_handler = handler_a}
struct sigaction sa_sigusr2 = {.sa_handler = handler_b}

// Fill in the (4) lines that make this safe.




sigaction(SIGUSR1, &sa_sigusr1, NULL);
sigaction(SIGUSR2, &sa_sigusr2, NULL);
```

4.2 Reasoning About Atomicity

Consider the following program:

```
sig_atomic_t received = 0; // set to 1 inside a SIGINT handler
sigset_t block;
sigset_t suspend_mask;

sigemptyset(&block);
sigemptyset(&suspend_mask);
sigaddset(&block, SIGINT);

sigprocmask(SIG_BLOCK, &block, NULL);

while (!received) {
  sigsuspend(&suspend_mask);
}
```

Questions:

- What is the signal mask of the process:
    - immediately before calling sigsuspend?
    - while inside sigsuspend?
    - immediately after sigsuspend returns?

- Suppose SIGINT arrives **after** the while condition is checked but **before** sigsuspend is called. Why does this pattern still work correctly?