Announcements

● Assignment 3 will be released later today
● Weekly survey will go out later this evening. Submit by Sunday night to get extra credit on Assignment 3 (due next Friday).

Lecture Overview

● Multiprocessing review!
● Assignment 2 redux
Recall the `mysystem` program from a few lectures ago:

```c
static int mysystem(const char *command) {
    pid_t pid = fork();
    if (pid == 0) {
        char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
        execvp(arguments[0], arguments);
        printf("Failed to invoke /bin/sh to execute the supplied command.");
        exit(0);
    }
    int status;
    waitpid(pid, &status, 0);
    return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

Note the `exit(0)` in the child. **If `execvp` fails, what would happen if we returned instead of exited?**
exit versus return

- Recall the `mysystem` program from a few lectures ago:

```c
static int mysystem(const char *command) {
    pid_t pid = fork();
    if (pid == 0) {
        char *arguments[] = {"/bin/sh", "-c", (char *) command, NULL};
        execvp(arguments[0], arguments);
        printf("Failed to invoke /bin/sh to execute the supplied command.");
        exit(0);
    }
    int status;
    waitpid(pid, &status, 0);
    return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
}
```

- Note the `exit(0)` in the child. **If execvp fails, what would happen if we returned instead of exited?**
  - The child would return to the function that called `mysystem`, in the main function, which would allow the child to continue executing code that wasn’t meant for it.

- **Remember the difference!**
Let's go through an example that is the kind of signals problem you may see on the midterm exam.

- Consider this program and its execution. Assume that all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
    printf("pirate\n");
    exit(0);
}

int main(int argc, char *argv[]) {
    signal(SIGUSR1, bat);
    pid_t pid = fork();
    if (pid == 0) {
        printf("ghost\n");
        return 0;
    }
    kill(pid, SIGUSR1);
    printf("ninja\n"); return 0;
}
```

For each of the five columns, write a **yes** or **no** in the header line. Place a **yes** if the text below it represents a possible output, and place a **no** otherwise.

<table>
<thead>
<tr>
<th>ghost</th>
<th>pirate</th>
<th>ninja</th>
<th>ninja</th>
<th>pirate</th>
</tr>
</thead>
<tbody>
<tr>
<td>pirate</td>
<td>ninja</td>
<td>ghost</td>
<td>ninja</td>
<td>pirate</td>
</tr>
</tbody>
</table>
Let's go through an example that is the kind of signals problem you may see on the midterm exam.

- Consider this program and its execution. Assume that all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling or time slice durations.

```c
static void bat(int unused) {
    printf("pirate\n");
    exit(0);
}

int main(int argc, char *argv[]) {
    signal(SIGUSR1, bat);
    pid_t pid = fork();
    if (pid == 0) {
        printf("ghost\n");
        return 0;
    }
    kill(pid, SIGUSR1);
    printf("ninja\n"); return 0;
}
```

For each of the five columns, write a yes or no in the header line. Place a yes if the text below it represents a possible output, and place a no otherwise.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>yes!</td>
<td>yes!</td>
<td>no!</td>
<td>no!</td>
<td>no!</td>
</tr>
<tr>
<td>ghost</td>
<td>pirate</td>
<td>ninja</td>
<td>ghost</td>
<td>pirate</td>
</tr>
<tr>
<td>pirate</td>
<td>ninja</td>
<td>ghost</td>
<td>ninja</td>
<td>pirate</td>
</tr>
<tr>
<td>ninja</td>
<td>pirate</td>
<td>ghost</td>
<td>ninja</td>
<td>pirate</td>
</tr>
<tr>
<td>pirate</td>
<td>ghost</td>
<td>ninja</td>
<td>pirate</td>
<td>ghost</td>
</tr>
</tbody>
</table>
Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

List all possible outputs.
Consider this program and its execution. Assume that all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

List all possible outputs:
- Possible Output 1: 112265
- Possible Output 2: 121265
- Possible Output 3: 122165
- If the > of the counter > 0 test is changed to a >=, then counter values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)
Problem 2

Consider this program and its execution. Assume that all processes run to completion, all system and `printf` calls succeed, and that all calls to `printf` are atomic. Assume nothing about scheduling or time slice durations.

```c
int main(int argc, char *argv[]) {
    pid_t pid;
    int counter = 0;
    while (counter < 2) {
        pid = fork();
        if (pid > 0) break;
        counter++;
        printf("%d", counter);
    }
    if (counter > 0) printf("%d", counter);
    if (pid > 0) {
        waitpid(pid, NULL, 0);
        counter += 5;
        printf("%d", counter);
    }
    return 0;
}
```

List all possible outputs:
- Possible Output 1: **112265**
- Possible Output 2: **121265**
- Possible Output 3: **122165**
- If the > of the `counter > 0` test is changed to a `>=`, then `counter` values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)
  - **18 outputs** now (6 x the first number)
Problem 3

Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

What is the output of the program?

```c
static pid_t pid; // global for handler1
static int counter = 0;
static void handler1(int unused) {
    counter++;
    printf("counter = %d\n", counter);
    kill(pid, SIGUSR1);
}
static void handler2(int unused) {
    counter += 10;
    printf("counter = %d\n", counter);
    exit(0);
}
int main(int argc, char *argv[]) {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while (true) {}
    }
    if (waitpid(-1, NULL, 0) > 0) {
        counter += 1000;
        printf("counter = %d\n", counter);
    }
    return 0;
}
```
Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

What is the output of the program?

- `counter = 1`
- `counter = 10`
- `counter = 1001`

This is the only possible output based on the program’s logic.
Problem 3

Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

What are the two potential outputs of the above program if the `while (true)` loop is completely eliminated?
Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

What are the two potential outputs of the above program if the `while (true)` loop is completely eliminated?

- The output from before (the 1/10/1001) output is still possible because the child process can be swapped out just after the `kill(getppid(), SIGUSR1)` call, and effectively emulate the stall that came with the `while (true)` loop when it was present.
- Now, though, the child process could complete and exit normally before the parent process has the opportunity to signal the child via its `handler1` function. That would mean `handler2` wouldn’t even execute, and we wouldn’t see `counter = 10` (Note that the child process’s call to `waitpid` returns -1, since it has no grandchild processes of its own).
- So, another possible output: `counter = 1 / counter = 1001`
Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

Problem 3

```c
static pid_t pid; // global for handler1
static int counter = 0;
static void handler1(int unused) {
    counter++;
    printf("counter = %d\n", counter);
    kill(pid, SIGUSR1);
}
static void handler2(int unused) {
    counter += 10;
    printf("counter = %d\n", counter);
    exit(0);
}
int main(int argc, char *argv[]) {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while (true){}
    }
    if (waitpid(-1, NULL, 0) > 0) {
        counter += 1000;
        printf("counter = %d\n", counter);
    }
    return 0;
}
```

- Now further assume the call to `exit(0)` has also been removed from the `handler2` function. Are there any other potential program outputs? If not, explain why. If so, what are they?
Assume that each call to `printf` flushes its output to the console in full, and further assume that none of the system calls fail in any unpredictable way (e.g. `fork` never fails, and `waitpid` only returns -1 because there aren’t any child processes at the moment it decides on its return value).

No other potential outputs, because:

- `counter = 1` is still printed exactly once, just in the parent, before the parent fires a `SIGUSR1` signal at the child (which may or may not have run to completion).
- `counter = 10` is potentially printed if the child is still running at the time the parent fires that `SIGUSR1` signal at it. The 10 can only appear after the 1, and if it appears, it must appear before the 1001.
- `counter = 1001` is always printed last, after the child process exits. It’s possible that the child existed at the time the parent signaled it to inspire `handler2` to print a 10, but that would happen before the 1001 is printed.
Assignment 2 Info

- Does the subprocess routine that creates the children that will factor numbers actually inherit the blocked set of the parent farm process?
  - Yes! Technically, a better subprocess implementation would unblock `SIGCHLD` in the child before `execvp` is called, since `SIGCHLD` is the only blocked signal in our farm program. In our case, we get away with not doing this because we know `factor.py` doesn't spawn child processes and so it doesn't depend on receiving `SIGCHLD`, but if we want to be technically thorough, we should unblock signals in the child as discussed in lecture. You don’t need to do this though. Great question from the students who asked it!

- To see which signals are blocked, I wrote a utility function that you can use by including this file:
  - `#include "signal-utils.h"
  - To get a list of the blocked signals, call: `print_blocked_signals(sigset_t *set)`
  - Pass in `NULL` to get the list of currently blocked signals. Otherwise, pass in the set you’d like to check.

- You don’t have to use `outbuf`; `dprintf` is fine. We won’t dock points. Just understand that `outbuf` is the better thing to use since it is a C++ stream versus `dprintf` which is for C. If you do use `outbuf`, make sure you understand how to use it because it has a tricky detail.

- Late days available for current assignment!
End of Lecture 9