CS110 Lecture 08: Concurrency and Race Conditions

Principles of Computer Systems

Winter 2021

Stanford University

Computer Science Department

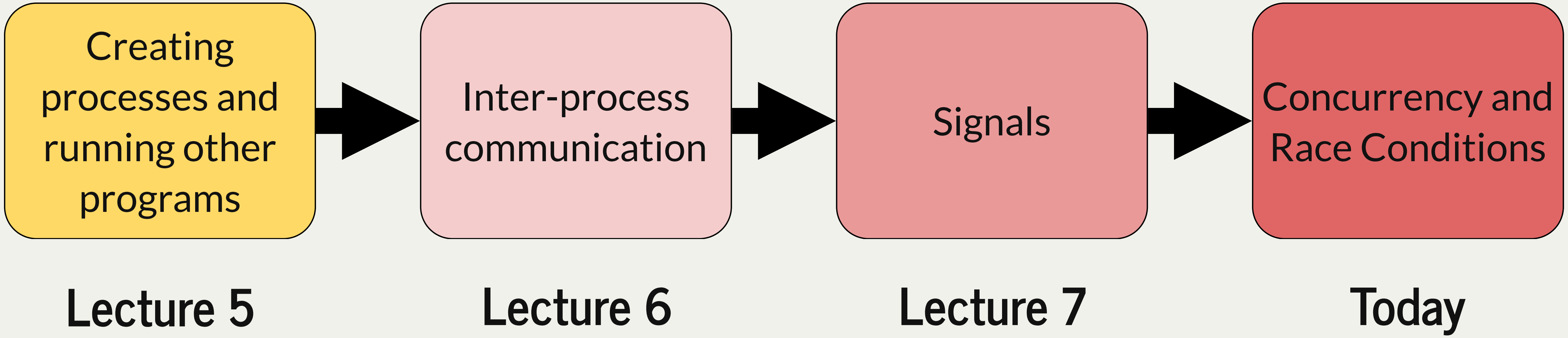
Instructors: Chris Gregg and

Nick Troccoli

https://comic.browserling.com/53

CS110 Topic 2: How can our programs create and interact with other programs?

Learning About Processes


Creating processes and running other programs
Inter-process communication
Signals
Concurrency and Race Conditions
Lecture 5
Lecture 6
Lecture 7
Today

Today's Learning Goals

- Understand what a race condition is and how they can cause problems in programs
- Learn about the race condition checklist for identifying and avoiding race conditions
- Learn more about sigsuspend and how it helps us avoid race conditions

Plan For Today

- Recap: Signals
- Race Conditions and Atomicity
- Revisiting sigsuspend
- More Practice: Race Conditions

Plan For Today

- **Recap: Signals**
- Race Conditions and Atomicity
- Revisiting sigsuspend
- More Practice: Race Conditions

Signals

A **signal** is a way to notify a process that an event has occurred

- There is a list of defined signals that can be sent (or you can define your own): SIGINT, SIGSTOP, SIGKILL, SIGCONT, etc.
- A signal is really a number (e.g. SIGSEGV is 11)
- A program can have a function executed when a type of signal is received
- Signals are sent either by the operating system, or by another process
  - e.g. SIGCHLD sent by OS to parent when child changes state
- You can send a signal to yourself or to another process you own

Sending Signals

The operating system sends many signals, but we can also send signals manually.

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

// same as kill(getpid(), signum)
int raise(int signum);
```

- **kill** sends the specified signal to the specified process (poorly-named; previously, default was to just terminate target process)
- **pid** parameter can be > 0 (specify single pid), < -1 (specify process group abs(pid)), or 0/-1 (we ignore these).
- **raise** sends the specified signal to yourself

waitpid()

Waitpid can be used to wait on children to terminate *or change state*:

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

- **pid:** the PID of the child to wait on, or -1 to wait on any of our children
- **status:** where to put info about the child's status (or NULL)
- the return value is the PID of the child that was waited on, -1 on error, or 0 if there are other children to wait for, but we are not blocking.

The default behavior is to wait for the specified child process to exit. **options** lets us customize this further (can combine these flags using |):

- **WUNTRACED** - also wait on a child to be stopped
- **WCONTINUED** - also wait on a child to be continued
- **WNOHANG** - don't block

Signal Handlers

We can have a function of our choice execute when a certain signal is received.

- We must register this "signal handler" with the operating system, and then it will be called for us.

```
typedef void (*sighandler_t)(int);
...
sighandler_t signal(int signum, sighandler_t handler);
```

- **signum** is the signal (e.g. SIGCHLD) we are interested in.
- **handler** is a function pointer for the function to call when this signal is received.
- (Note: no handlers allowed for SIGSTOP or SIGKILL)

Signal Handlers

A signal can be received at any time, and a signal handler can execute at any time.

- Signals aren't handled immediately (there can be delays)
- Signal handlers can execute at any point during the program execution (eg. pause main() execution, execute handler, resume main() execution)
  - **Goal:** keep signal handlers simple!
- Similar to hardware interrupts -- POSIX brings that model to software

Signal Handlers

Key Idea: a signal handler is called if *one or more* signals of a type are sent.

- Like a notification that "one or more signals of this type are waiting for you!"
- The kernel tracks only *what* signals should be sent to you, not *how many*

Solution: signal handler should clean up as many children as possible, using **WNOHANG**, which means don't block. If there are children we *would have* waited on but aren't, returns 0. -1 typically means no children left.

```c
static void reapChild(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    numDone++;
  }
}
```

Do Not Disturb

The **sigprocmask** function lets us temporarily block signals of the specified types. Instead, they will be delivered when the block is removed.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- To add signals to the blocked list, **how = SIG_BLOCK**, **set** points to the signals to add
- To remove signals from the blocked list, **how = SIG_UNBLOCK**, **set** points to the signals to remove
- To set the whole blocked list, **how = SIG_SETMASK**, **set** is the location of the new blocked list
- In all cases, **oldset** is where to store the old blocked list (or NULL).

Do Not Disturb

sigset_t is a special type (usually a 32-bit int) used as a bit vector. It must be created and initialized using special functions (we generally ignore the return values).

```
// Initialize to the empty set of signals
int sigemptyset(sigset_t *set);

// Set to contain all signals
int sigfillset(sigset_t *set);

// Add the specified signal
int sigaddset(sigset_t *set, int signum);

// Remove the specified signal
int sigdelset(sigset_t *set, int signum);
```

```
static void imposeSIGCHLDBlock() {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGCHLD);
    sigprocmask(SIG_BLOCK, &set, NULL);
}
```

Plan For Today

- Recap: Signals
- **Race Conditions and Atomicity**
- Revisiting sigsuspend
- More Practice: Race Conditions

Concurrency

Concurrency means performing multiple actions at the same time.

- Concurrency is extremely powerful: it can make your systems faster, more responsive, and more efficient. It's fundamental to all modern software.
- When you introduce multiprocessing (e.g. **`fork`**) and asynchronous signal handling (e.g. **`signal`**), it's possible to have concurrency issues. These are tricky!
- Most challenges come with shared data - e.g. two routines using the same variable.
- Many large systems parallelize computations by trying to eliminate shared data - e.g. split the data into independent chunks and process in parallel.
- A **race condition** is an unpredictable ordering of events (due to e.g. OS scheduling) where some orderings may cause undesired behavior.

Off To The Races

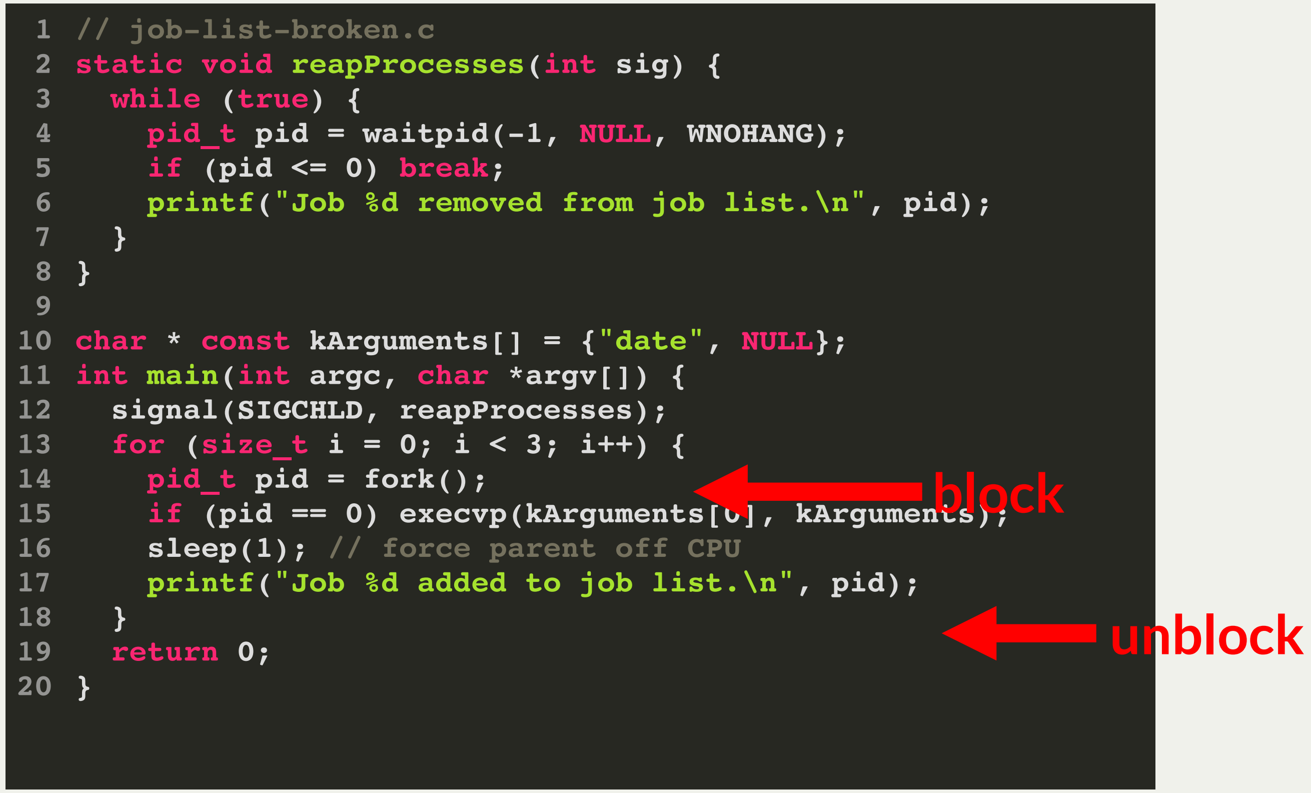
```c
// job-list-broken.c
static void reapProcesses(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    printf("Job %d removed from job list.\n", pid);
  }
}

char * const kArguments[] = {"date", NULL};
int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);
  for (size_t i = 0; i < 3; i++) {
    pid_t pid = fork();
    if (pid == 0) execvp(kArguments[0], kArguments);
    sleep(1); // force parent off CPU
    printf("Job %d added to job list.\n", pid);
  }
  return 0;
}
```

```
myth60$ ./job-list-broken
Sun Jan 27 03:57:30 PDT 2019
Job 27981 removed from job list.
Job 27981 added to job list.
Sun Jan 27 03:57:31 PDT 2019
Job 27982 removed from job list.
Job 27982 added to job list.
Sun Jan 27 03:57:32 PDT 2019
Job 27985 removed from job list.
Job 27985 added to job list.
myth60$ ./job-list-broken
Sun Jan 27 03:59:33 PDT 2019
Job 28380 removed from job list.
Job 28380 added to job list.
Sun Jan 27 03:59:34 PDT 2019
Job 28381 removed from job list.
Job 28381 added to job list.
Sun Jan 27 03:59:35 PDT 2019
Job 28382 removed from job list.
Job 28382 added to job list.
myth60$
```

Symptom: it looks like jobs are being removed from the list before being added! How is this possible?

Off To The Races



```
 1 // job-list-broken.c
 2 static void reapProcesses(int sig) {
 3   while (true) {
 4     pid_t pid = waitpid(-1, NULL, WNOHANG);
 5     if (pid <= 0) break;
 6     printf("Job %d removed from job list.\n", pid);
 7   }
 8 }
 9
10 char * const kArguments[] = {"date", NULL};
11 int main(int argc, char *argv[]) {
12   signal(SIGCHLD, reapProcesses);
13   for (size_t i = 0; i < 3; i++) {
14     pid_t pid = fork();
15     if (pid == 0) execvp(kArguments[0], kArguments);
16     sleep(1); // force parent off CPU
17     printf("Job %d added to job list.\n", pid);
18   }
19   return 0;
20 }
```

Issue: the signal handler is being called before the parent adds to the job list.

Solution: block SIGCHLD from lines 14-17 to force the parent to always add to the job list first.

This is called a **critical section** - a piece of code that is indivisible. It cannot be interrupted midway by our other code.

Off To The Races

```c
// job-list-fixed.c
char * const kArguments[] = {"date", NULL};
int main(int argc, char *argv[]) {
  signal(SIGCHLD, reapProcesses);

  // Create set with just SIGCHLD
  sigset_t set;
  sigemptyset(&set);
  sigaddset(&set, SIGCHLD);

  for (size_t i = 0; i < 3; i++) {
    sigprocmask(SIG_BLOCK, &set, NULL);
    pid_t pid = fork();
    if (pid == 0) {
      sigprocmask(SIG_UNBLOCK, &set, NULL);
      execvp(kArguments[0], kArguments);
    }
    sleep(1); // force parent off CPU
    printf("Job %d added to job list.\n", pid);
    sigprocmask(SIG_UNBLOCK, &set, NULL);
  }
  return 0;
}
```

- This is called a **critical section** - a piece of code that is indivisible. It cannot be interrupted midway by our other code.
- If something is **atomic**, it means it cannot be interrupted by something else.
- Code that executes while a signal is blocked is *atomic with respect to that signal's handler* - the handler executes before or after the code, but never during.

Race Conditions and Concurrency

Race conditions are a fundamental problem in concurrent code.

- Decades of research in how to detect and deal with them
- They can corrupt your data and violate its integrity, so it is no longer consistent
- Critical sections can prevent race conditions, but there are two major challenges
    - Figuring out where to put critical sections
        - E.g. You have a global linked list. A signal handler prints out the list. Your main code inserts and deletes from the list. You need to make sure every update to the list executes atomically, so a signal handler never sees a bad pointer.
    - Structuring your code so critical sections don't limit performance
        - E.g. if your code spends most of its time in critical sections, then signals may be delayed for a long time (making your program less responsive).

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What global variables are used in both the main code and signal handlers?

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings.** How can we use signal blocking and atomic operations to force the correct ordering(s)?

Plan For Today

- Recap: Signals
- Race Conditions and Atomicity
- Revisiting sigsuspend
- More Practice: Race Conditions

Waiting For Signals - where we left off last time

```
 1 // simplesh
 2
 3 // The currently-running foreground command PID
 4 static pid_t foregroundPID = 0;
 5
 6 static void waitForForegroundProcess(pid_t pid) {
 7   fgpid = pid;
 8   sigset_t empty;
 9   sigemptyset(&empty);
10   while (fgpid == pid) {
11     sigsuspend(&empty);
12   }
13 }
14
15 static void executeCommand(char *command, bool inBackground) {
16     // ...(omitted for brevity)...
17     if (inBackground) {
18         printf("%d %s\n", pidOrZero, command);
19     } else {
20         waitForForegroundCommand(pidOrZero);
21     }
22 }
23
24 static void reapProcesses(int signum) {
25     while (true) {
26         pid_t result = waitpid(-1, NULL, WNOHANG);
27         if (result <= 0) break;
28         if (result == foregroundPID) foregroundPID = 0;
29     }
30 }
```

- The last thing we did in our simplesh implementation last time was to fix the spin-block by adding a sigsuspend inside the while loop on line 11 in the code.
- But, there is still a race condition!
  - On line 20, we are in the parent process. But, the child process has already begun. What if the child finishes before waitForForegroundProcess gets called (perfectly likely)?
  - foregroundPID will be set to 0 in reapProcesses, then waitForForegroundProcess gets called...

Deadlock

Deadlock is a program state in which no progress can be made - it is caused by code waiting for something that will never happen.

E.g. waitForForegroundProcess loops until foregroundPID is set to 0. But it never will be set to 0 in this case!

We could force this to happen if we put a sleep(1) as the first line of waitForForegroundProcess, and run a very short program in the foreground (e.g., date)

So, what we need to do is to use our signal blocking idea to ensure that the signal handler does not get called until we are ready (much like in the job list example).

```c
 1 static pid_t fgpid = 0; // 0 means no foreground process
 2 static void reapProcesses(int sig) {
 3   pid_t pid;
 4   while (true) {
 5     pid = waitpid(-1, NULL, WNOHANG);
 6     if (pid <= 0) break;
 7     if (pid == fgpid) fgpid = 0;
 8   }
 9 }
10
11 static void waitForForegroundProcess(pid_t pid) {
12   fgpid = pid;
13   sigset_t empty;
14   sigemptyset(&empty);
15   while (fgpid == pid) {
16     sigsuspend(&empty);
17   }
18   unblockSIGCHLD();
19 }
20
21 int main(int argc, char *argv[]) {
22   signal(SIGCHLD, reapProcesses);
23   while (true) {
24     // ... (left out for brevity)
25     blockSIGCHLD();
26     pid_t pid = forkProcess();
27     if (pid == 0) {
28       unblockSIGCHLD();
29       execvp(argv[0], argv);
30       printf("%s: Command not found\n", argv[0]);
31       exit(0);
32     }
33
34     if (isbg) {
35       printf("%d %s\n", pid, command);
36       unblockSIGCHLD();
37     } else {
38       waitForForegroundProcess(pid);
39     }
40   }
41   printf("\n");
42   return 0;
43 }
```

- There are a lot of blocking / unblocking calls here, and a sigsuspend, too. Let's go through them:
- Starting in main(), we see that we block SIGCHLD before we fork (line 25). We must block here to avoid the race condition
- We unblock in the *child* (line 28) because child processes inherit their blocked signals from the parent. The child does not need a blocked SIGCHLD.
- We unblock in the parent on line 36 because for a background process, we can accept signals after this point.
- In waitForForegroundProcess, we call sigsuspend *without unblocking*. This is because sigsuspend itself will unblock! It sets the signal mask to empty, meaning that it has unblocked. When sigsuspend wakes up, it re-blocks (see next slide)
- We unblock on line 18 because we no longer need the signal blocked (though we will re-block when we start a new process in main again).

```
 1 static pid_t fgpid = 0; // 0 means no foreground process
 2 static void reapProcesses(int sig) {
 3   pid_t pid;
 4   while (true) {
 5     pid = waitpid(-1, NULL, WNOHANG);
 6     if (pid <= 0) break;
 7     if (pid == fgpid) fgpid = 0;
 8   }
 9 }
10
11 static void waitForForegroundProcess(pid_t pid) {
12   fgpid = pid;
13   sigset_t empty;
14   sigemptyset(&empty);
15   while (fgpid == pid) {
16     sigsuspend(&empty);
17   }
18   unblockSIGCHLD();
19 }
20
21 int main(int argc, char *argv[]) {
22   signal(SIGCHLD, reapProcesses);
23   while (true) {
24     // ... (left out for brevity)
25     blockSIGCHLD();
26     pid_t pid = forkProcess();
27     if (pid == 0) {
28       unblockSIGCHLD();
29       execvp(argv[0], argv);
30       printf("%s: Command not found\n", argv[0]);
31       exit(0);
32     }
33
34     if (isbg) {
35       printf("%d %s\n", pid, command);
36       unblockSIGCHLD();
37     } else {
38       waitForForegroundProcess(pid);
39     }
40   }
41   printf("\n");
42   return 0;
43 }
```

- sigsuspend() is an interesting system call:
  - sigsuspend, *atomically* both adjusts the blocked signal set *and* goes to sleep until a signal is received. When some unblocked signal arrives, the process gets the CPU, the signal is handled, the original blocked set is restored, and sigsuspend returns.
  - Remember, "atomically" means that it does this without creating a race condition. Nothing else can happen (i.e., the signal cannot be called) between the process going to sleep and the signals being unblocked. The operating system is in control here, and it puts the process to sleep (which is what we want) and *then* unblocks the signals.
  - Notice that this happens *inside* the while loop. If we had attempted to unblock and then pause (for instance), there could still be a race condition between those two lines of code.
  - We need to unblock (line 18) because when sigsuspend wakes up, it re-blocks the signals (why?)

Plan For Today

- Recap: Signals
- Race Conditions and Atomicity
- Revisiting sigsuspend
- **More Practice: Race Conditions**

Practice Problem 1

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 output orders, Place a **yes** if the text represents a possible output, and place a **no** otherwise.
  - ghost ninja pirate
  - pirate ninja
  - ninja ghost
  - ninja pirate ninja
  - ninja pirate ghost

Practice Problem 1

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 output orders, Place a **yes** if the text represents a possible output, and place a **no** otherwise.
  - ghost ninja pirate. **yes**
  - pirate ninja. **yes**
  - ninja ghost. **no**
  - ninja pirate ninja. **no**
  - ninja pirate ghost. **no**

Practice 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

Practice 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

- The grandparent that starts with counter 0 exits last, because it waits on its child; the last output is 5
- The parent that starts with counter 1 exits second to last, after waiting for its child; the second-to-last output is 6
- The parent that starts with counter 1 outputs first
- The second parent (1) output and the two child (2) outputs are up to the scheduler

Practice 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.)

Practice 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)

Practice Problem 3

- Consider the following program. 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).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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;
}
```

- What is the output of the program?
- What are the two potential outputs of the above program if the **`while (true)`** loop is completely eliminated?
- Describe how the two processes would need to be scheduled in order for each of the two outputs to be presented.
- 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?

Practice Problem 3

- Consider the following program. 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).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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;
}
```

- 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

Practice Problem 3

- Consider the following program. 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).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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;
}
```

- 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—via its **`handler1`** function— has the opportunity to signal the child. That would mean **`handler2`** wouldn't even execute, and we wouldn't expect to see **`counter = 10`**. (Note that the child process's call to **`waitpid`** returns **`-1`**, since it itself has no grandchild processes of its own).

- So, another possible output would be:

```
counter = 1
counter = 1001
```

Practice Problem 3

- Consider the following program. 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).

```c
static pid_t pid; // necessarily global so handler1 has
                  // access to it
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?
- 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.

- Note that the child process either prints nothing at all, or it prints a **`10`**. The child process can never print **`1001`**, because its **`waitpid`** call would return **`-1`** and circumvent the code capable of printing the **`1001`**.

Overview: Signals and Concurrency

- Concurrency is powerful: it lets our code do many things at the same time
  - It can run faster (more cores!)
  - It can do more (run many programs in background)
  - It can respond faster (don't have to wait for current action to complete)
- Signals are a way for concurrent processes to interact
  - Send signals with kill and raise
  - Handle signals with signal
  - Control signal delivery with sigprocmask, sigsuspend
  - Preempt running code
  - Making sure code running in a signal handler works correctly is difficult
  - *Race conditions* occur when code can see data in an intermediate and invalid state (often KABOOM)
- Assignments 3 and 4 use signals, as a way to start easing into concurrency before we tackle multithreading
- Take CS149 if you want to learn how to write high concurrency code that runs 100x faster

Recap

- Recap: Signals
- Race Conditions and Atomicity
- Revisiting sigsuspend
- More Practice: Race Conditions

Next Time: Introduction to Threads