Processes, Signals, I/O, Shell Lab

- 15-213: Introduction to Computer Systems
- Recitation 9: Monday, Oct. 21, 2013
- Marjorie Carlson
- Section A

Agenda

- **News**
- **Shell Lab Overview**
- **Processes**
  - Overview
  - Important functions
  - Concurrency
- **Signals**
  - Overview
  - Important functions
  - Race conditions
- **I/O Intro**

News

- **Midterm grades were good! Go you!**
  - The exams will be viewable soon if they're. not already.
  - E-mail us with concerns.

- **Cachelab grades are out. I've annotated your code while grading for style.**
  - **Autolab → Cache Lab → View handin history**
    Click on your most recent submission, then **View Source**
    View as Syntax Highlighted Source
  - Style matters! http://www.cs.cmu.edu/~213/codeStyle.html

- **Shell lab out, due Tuesday Oct. 29 11:59 p.m.**

Agenda

- **News**
- **Shell Lab Overview**
- **Processes**
  - Overview
  - Important functions
  - Concurrency
- **Signals**
  - Overview
  - Important functions
  - Race conditions
- **I/O Intro**

Processes

- **An instance of an executing program**
- **An abstraction provided by the operating system**
- **Properties:**
  - Private memory. No two processes share memory, registers, etc.
  - Some shared state, such as the open file table
  - A process ID (pid) and a process group ID (pgid)
  - Become zombies when finished running

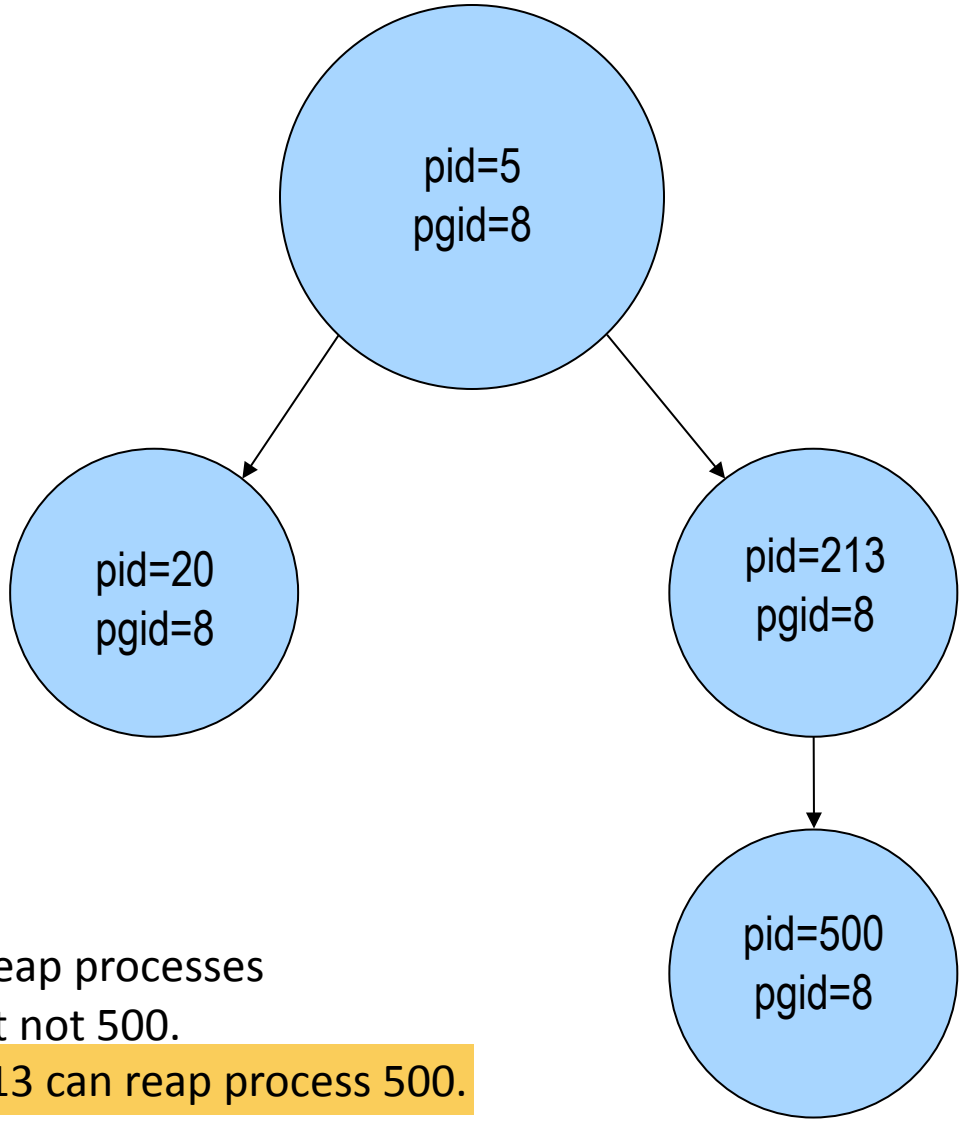
Birth of a Process: fork

- `pid_t fork()`
  - Clones the current process.
  - The new process is an **exact duplicate** of the parent's state. It has its own stack, own registers, etc.
  - It has its own file descriptors (but the files themselves are shared).
  - **Called once, returns twice** (once in the parent, once in the child).
  - Return value in child is 0, child's pid in parent. (This is how the parent can keep track of who its child is.)
  - Returns -1 in case of failure.
  - After the fork, **we do not know** which process will run first, the parent or the child.

Birth of a Process: a digression

- `int setpgid(pid_t pid, pit_t pgid)`
  - Sets the process group ID of the process specified by `pid`. (Returns 0 on success, -1 on failure.)
  - If `pid` = 0, setpgid is applied to the calling process.
  - If `pgid`= 0, setpgid sets the pgid of the specified process to the pid of the calling process.
  - By default, children inherit the pgid of their parents.
    - (When won't you want children to inherit the parent's pgid? Hmm…)

Birth of a Process: a digression



Process 5 can reap processes
20 and 213, but not 500.
Only process 213 can reap process 500.

Life of a Process: exec

- **`int execve(const char *filename, char** argv, char** envp)`**
  - Replaces the current process with a new one. This is how we run a new program.
  - **Called once; does not return** (or returns -1 on failure).
  - `argc`, `argv` are like the command-line arguments to main for the new process
  - envp is the environment variable
    - Contains information that affects how various processes work
    - On Shark machines, can get its value by declaring `extern char** environ;`

Death of a Process: exit

- **`void exit(int status)`**
  - Immediately terminates the process that called it.
  - `status` is normally the return value of `main()`.
  - The OS frees the resources it used (heap, file descriptors, etc.) **but** not its exit status. It remains in the process table to await its reaping.
  - Zombies are reaped when their parents read their exit status. (If the parent is dead, this is done by `init`.) Then its pid can be reused.

Reaping of a Process: wait

- **`pid_t waitpid(pid_t pid, int* status, int options)`**
  - The `wait` family of functions allows a parent to know when a child has changed state (e.g., terminated).
  - `waitpid` returns when the process specified by `pid` terminates.
    - `pid` must be a direct child of the invoking process.
    - If `pid = -1`, it will wait for *any* child of the current process.
  - Return value: the pid of the child it reaped.
  - Writes to `status`: information about the child's status.
  - `options` variable: used to modify waitpid's behavior.
    - `WNOHANG`: keep executing caller until a child terminates.
    - `WUNTRACED`: report **stopped** children too.
    - `WCONTINUED`: report **continued** children too.

Yay Concurrency!

```
pid_t child_pid = fork();

if (child_pid == 0) {
   printf("Child!\n");
   exit(0);
}

else {
   printf("Parent!\n");
}
```

- **Outcomes:**
  - Child!
    Parent!
  - Parent!
    Child!

```
int status;
pid_t child_pid = fork();

if (child_pid == 0) {
   printf("Child!\n");
   exit(0);
}

else {
   waitpid(child_pid, &status, 0);
   printf("Parent!\n");
}
```

- **Outcome:**
  - Child!
    Parent!

Race Conditions

- **Race conditions occur when the sequence or timing of events is random or unknown.**
  - When you `fork`, you don't know whether the parent or child will run first.
  - Signal handlers will interrupt currently running code. (We'll get to this in a sec.)

- **If something can go wrong, it will!**
  - You must reason carefully about the possible sequence of events in concurrent programs. (A big theme of the second half of this course!)

Agenda

- **News**
- **Shell Lab Overview**
- **Processes**
  - Overview
  - Important functions
  - Concurrency
- **Signals**
  - Overview
  - Important functions
  - Race conditions
- **I/O Intro**

Signals

- Signals are the basic way processes communicate with each other. They notify a process that an event has occurred (for example, that its child has terminated).
- They are sent several ways: Ctrl-C, Ctrl-Z, a "kill" instruction.
- Signals are asynchronous. They aren't necessary received immediately; they're received right after a context switch.
- They are non-queuing. If 100 child processes die and send a SIGCHLD, the parent may still only receive one SIGCHLD.

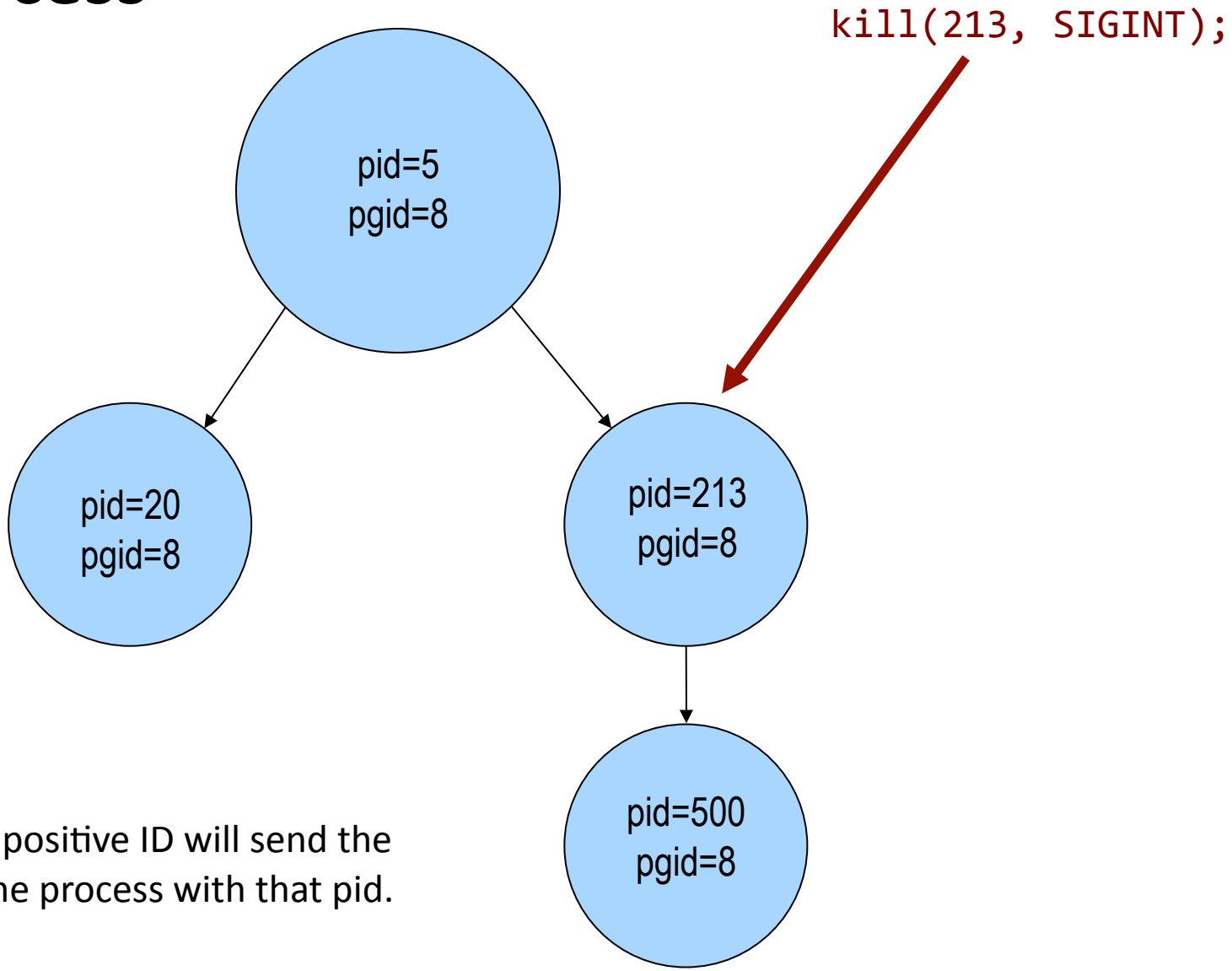
Signals

- **Many signals have default behaviors**
  - SIGINT, SIGTERM will terminate the process.
  - SIGSTP will suspend the process until it receives SIGCONT.
  - SIGCHLD is sent from a child to its parent when the child dies or is suspended.
- **But we can avoid the default behavior by catching the signal and running our own signal handler.**
  - SIGKILL and SIGSTOP can't be modified.
- **We can block signals with `sigprocmask()`.**
- **We can wait for signals with `sigsuspend()`.**

Sending a Signal

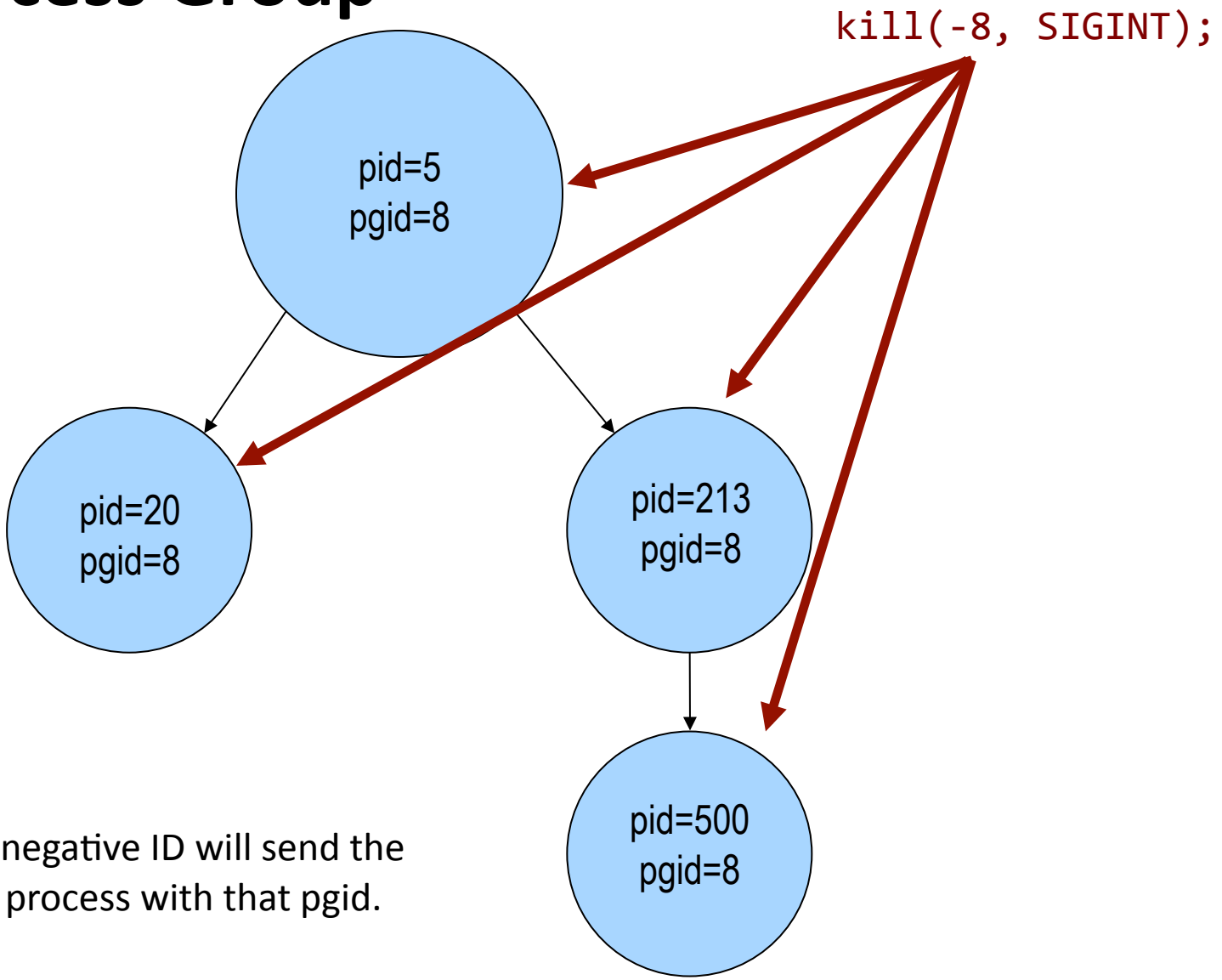
- `int kill(pid_t pid, int sig)`
  - Despite the name, this is used to send **all** signals between processes – not just SIGKILL.
  - If `pid` is positive, this sends signal `sig` to the process with pid = `id`.
  - If `pid` is negative, this sends signal `sig` to **all** processes with with pgid = `-id`.

Kill: Process



`kill()` with a positive ID will send the signal only to the process with that pid.

Kill: Process Group



`kill()` with a negative ID will send the signal to all the process with that pgid.

Handling a Signal

- **`signal(int signum, sighandler_t handler)`**
  - `signal` installs a signal `handler` that will run whenever a particular signal (the `signum`) is received.
  - The handler is just a function you write that takes one int (the signum) and is void (returns nothing).
  - Now, whenever you receive that signal, the handler will interrupt the process and execute –even if it or another signal handler is currently running.
  - Control flow of the main program is restored once the handler has finished.
  - This creates a separate flow of control in the same process, because you don't know when the handler will get called!

Handling a Signal

```
void handler1(int sig) {
   pid_t pid;
   while ((pid = waitpid(-1,NULL,WNOHANG)) > 0)
      deletejob(pid);
   if (errno != ECHILD)
      unix_error("waitpid error");
}

int main() {
   int pid;
   Signal(SIGCHLD, handler);      // initialize the handler
   initjobs();                    // initialize a jobs list

   while(1) {
      /* child process */
      if (pid = Fork()) == 0) {
         Execve("/bin/date",argv,NULL);
      }
      /* parent process */
      addjob(pid);
   }
   exit(0);
}
```

Do not use this code! It has a concurrency bug!

Blocking Signals

- Processes can choose to block signals using a signal mask.
- While a signal is blocked, a process will still receive the signal but keep it pending. No action will be taken until the signal is unblocked.
- Still nonqueuing: the process will only track that it has received a blocked signal, but not the number of times it was received.
- This allows you to ensure that a particular part of your code is never interrupted by a particular signal.

Blocking Signals: Relevance to You

- **When you fork a new process in shell lab:**
  - CHILD:
    - exec; run, then terminate (thus sending a SIGCHLD to parent).
  - PARENT:
    - Add the process to the job queue.
    - When you receive SIGCHLD, go to the SIGCHLD handler, which removes the child from the job queue.
- **Common race condition:**
  - The child could terminate before the parent has added it to the job queue. Then the SIGCHLD handler tries to remove something from the job queue that isn't there, and worse, it's then added to the job queue and never removed.
  - Solution: block SIGCHLDs during the critical section.

Back to This Code…

```
void handler1(int sig) {
   pid_t pid;
   while ((pid = waitpid(-1,NULL,WNOHANG)) > 0)
      deletejob(pid);
   if (errno != ECHILD)
      unix_error("waitpid error");
}

int main() {
   int pid;
   Signal(SIGCHLD, handler);      // initialize the handler
   initjobs();                    // initialize a jobs list

   while(1) {
      /* child process */
      if (pid = Fork()) == 0) {
         Execve("/bin/date",argv,NULL);
      }
      /* parent process */
      addjob(pid);
   }
   exit(0);
}
```

Do not use this code! It has a concurrency bug!

Blocking Signals: Important Functions

- **`sigsetops`**
  - A family of functions used to create and modify sets of signals. E.g.,
    - `int sigemptyset(sigset_t *set);`
    - `int sigfillset(sigset_t *set)`
    - `int sigaddset(sigset_t *set, int signum);`
    - `int sigdelset(sigset_t *set, int signum);`
  - These sets can then be used in other functions.
  - http://linux.die.net/man/3/sigsetops
  - Remember to pass in the *address* of the sets, not the sets themselves

Blocking Signals: Important Functions

- **`int sigprocmask(int option, const sigset_t set, sighandler_t *oldSet)`**
  - `sigprocmask` updates your mask of blocked/unblocked signals, using the sigset `set`.
  - option: `SIG_SETMASK`, `SIG_BLOCK`, `SIG_UNBLOCK`
  - The signal mask's old value is written into `oldSet`.
  - Blocked signals are ignored until unblocked.
    - **Remember**: a process only tracks whether it has received a blocked signal, not how many times.
    - If you block SIGCHLD and then 20 children terminate, when you unblock you will only receive **one** SIGCHLD and run the appropriate handler **once**.

Blocking Signals: Important Functions

- `int sigsuspend(const sigset_t *mask)`
  - `sigsuspend` suspends your process until it receives a particular signal. It's good way to wait for something to happen. This is the right way to ensure your processes do their thing in the right order.
  - It *temporarily* replaces the process's signal mask with `mask`, which should be the signals you **don't** want to be interrupted by. (This is the opposite of `sigprocmask`.) So if your goal is to be "woken up" by a SIGCHLD, your mask should **not** contain SIGCHLD.
  - `sigsuspend` will return after an **un**blocked signal is received and its handler run.
  - Once `sigsuspend` returns, it automatically reverts the process signal mask to its old value.

Race Conditions

```
int counter = 1;
void handler(int signum) {
    counter--;
}

int main() {
    signal(SIGALRM, handler);
    kill(0, SIGALRM);
    counter++;
    printf("%d\n",counter);
}
```

- **Possible outputs?**
- **What if we wanted to guarantee that the handler executed after the print statement?**

Race Conditions

```
int counter = 1;
void handler(int signum) {
    counter--;
}

int main() {
    signal(SIGALRM, handler);
    sigset_t alarmset, oldset;
    sigemptyset(&alarmset);
    sigaddset(&alarmset, SIGALRM);

    //Block SIGALRM from triggering the handler
    sigprocmask(SIG_BLOCK,&alarmset,&oldset);

    kill(0,SIGALRM);
    counter++;
    printf("%d\n",counter);

    //Let the pending or incoming SIGALRM trigger the handler
    sigprocmask(SIG_UNBLOCK,&alarmset,NULL);
}
```

Agenda

- **News**
- **Shell Lab Overview**
- **Processes**
  - Overview
  - Important functions
  - Concurrency
- **Signals**
  - Overview
  - Important functions
  - Race conditions
- **I/O Intro**

Unix I/O

- **All Unix I/O, from network sockets to text files, are based on one interface.**
- **A file descriptor is what's returned by open( ).**

```
int fd = open("/path/to/file", O_RDONLY);
```

- **It's just an int, but you can think of it as a pointer into the file descriptor table.**
- **Every process starts with three file descriptors by default:**
  - 0: STDIN
  - 1: STDOUT
  - 2: STDERR.
- **Every process gets its own file descriptor table, but all processes share the open file table and v-node table.**

Unix I/O: A Handy Function

- **`dup2(dest_fd, src_fd)`**
  - src_fd will now point to the same place as dest_fd.
  - You can use this to redirect output from STDOUT to a location of your own choosing.
  - This is handy for implementing redirection in your shell.
    - Use open() to open the file you want to redirect to.
    - Call dup2(what-you-just-opened, 1) to cause fd to refer to your file instead of STDOUT.

Shell Lab Tips

- **There's a lot of starter code in the book; look it over.**
- **Read the handout well, especially the "Hints" section.**
- **Use tshref to figure out the required behavior.**
  - For instance, the format of the output when a job is stopped.
- **Be careful of the add/remove job race condition!**
- **Use `sigsuspend`, not `waitpid`, to wait for foreground jobs**
  - `waitpid` should only occur once in your code.
  - You will lose points for using tight loops (`while(1) {}`) or `sleep` to wait for a signal.

Shell Lab Tips

- **Shell requires SIGINT and SIGSTP to be forwarded to the foreground job of the shell (and all its descendants).**
  - How could process groups be useful?
- **Remember the SIGCHILD handler may have to reap multiple children per call!**
- **We provide drivers:**
  - ./runtrace
  - ./sdriver
- **BUT start by actually using your shell and seeing if/where it fails.**
  - The last page of the handout is a guide to what functionality we test.