

# 15-213 Recitation 9: Processes, Signals, and tshlab

14 March 2016

Ralf Brown and the 15-213 staff

# WELCOME BACK!

- We hope you're rested and ready to finish tshlab

# Agenda

- Reminders
- The Birth, Death, and Afterlife(?) of Processes
- Signals
- I/O
- tshlab Tools and Hints
- Summary of Useful Functions

# Reminders

- Cache Lab grades will be available on Autolab soon
  - click 'View Source' on your latest submission to read our annotations and comments
- tshlab is due in just over a week

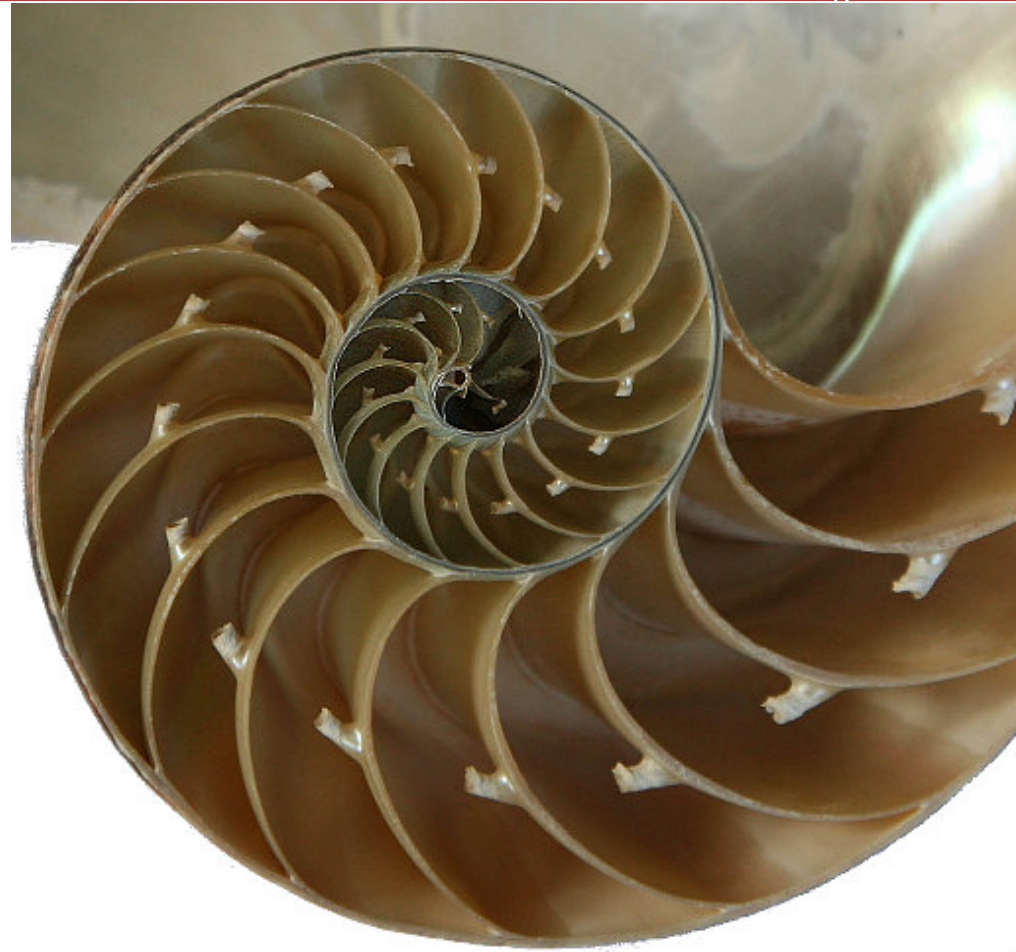


Image credit: flickr user Jitze Couperus  
CC-BY 2.0

# What Is a Process?

- An instance of an executing program
- The basic unit of execution in the operating system
  - an abstraction with properties such as
    - private state (memory, registers, etc.)
    - shared state (e.g. open files)
    - management information (process ID [pid] and process group ID [pgid])

# The Life Cycle of a Process

- 1 Creation
- 2 Loading a Program
- 3 Execution
- 4 Termination
- 5 Post-termination

## Birth of a Process: fork()

- `pid_t fork(void);`
- Clones the current process, and assigns the clone a new process ID
  - exact duplicate of the caller's state except for `%rax` and maybe `%rsp`
  - new copies of file descriptors, but the files themselves are shared
- This function returns **twice!** Both the parent and child get return values
  - child gets a return value of 0
  - parent gets the child's *pid* to be able to track its children
  - returns -1 if the cloning failed
- Either parent or child could run first after the fork

## Life of a Process: `execve()`

- `execve(const char *filename, char *const argv[], char *const envp[]);`
- Replace the state of the current process with new state loaded from the executable binary at 'filename' and start it running
  - `argv` contains the commandline arguments for the new program
  - `envp` contains the environment variables (use global `environ` from `unistd.h` to copy caller's default environment)
- This function does not return if successful!
  - returns -1 if it fails, and sets the global 'errno' to the reason for failure
- The combination of `fork()` and `execve()` is the usual way of starting a new program – create the new process and run a new binary in it



## Death of a Process: `exit()`

- `void exit(int status);`
- This C library function performs a variety of clean-up tasks, then invokes the `_exit()` system call
  - open stdio streams are flushed and closed
  - temporary files are removed
  - user-registered exit-handling functions are called
- The process is immediately terminated on `_exit`
  - the OS frees *almost* all of the process's resources

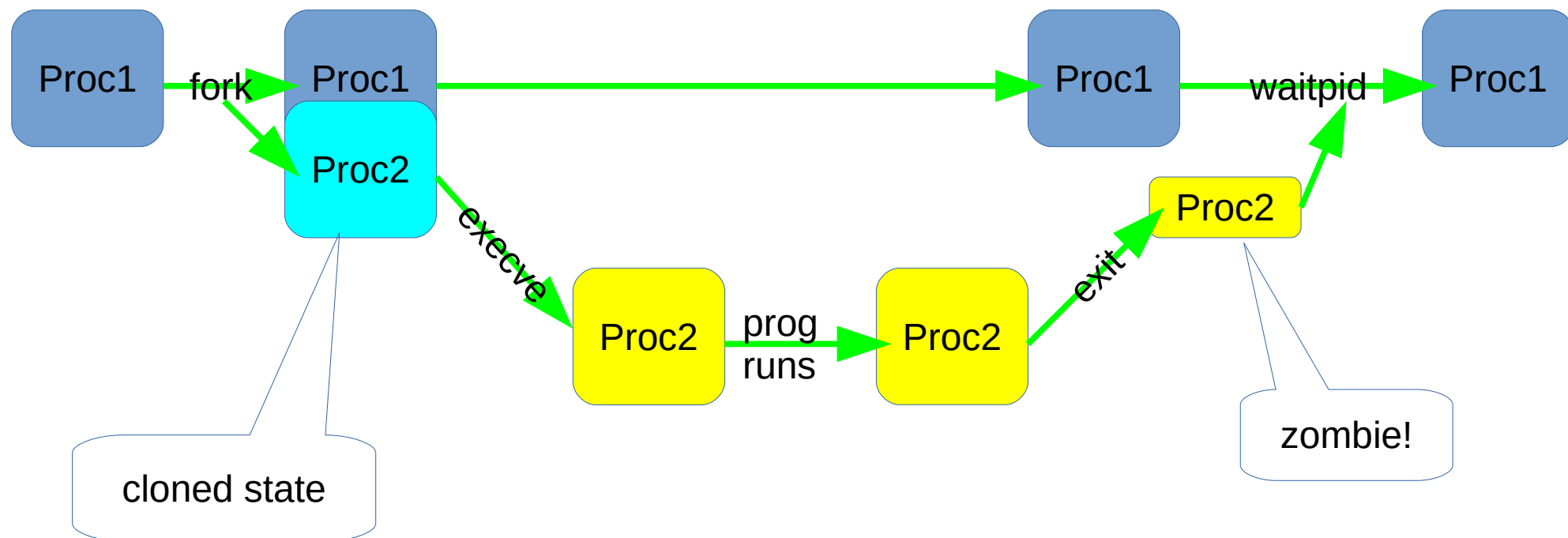
# What Happens After a Process Dies?

- The process hangs around as a *zombie* (yes, that's the technical term) until its parent retrieves its exit code
  - not runnable, but continues to use some resources, e.g. process table entry
- Once its parent has received the exit code, the zombie is *reaped* and all remaining resources are returned to the operating system
  - now it is fully dead, not just mostly dead
  - its process ID can finally be reused

# Reaping Zombies: waitpid()

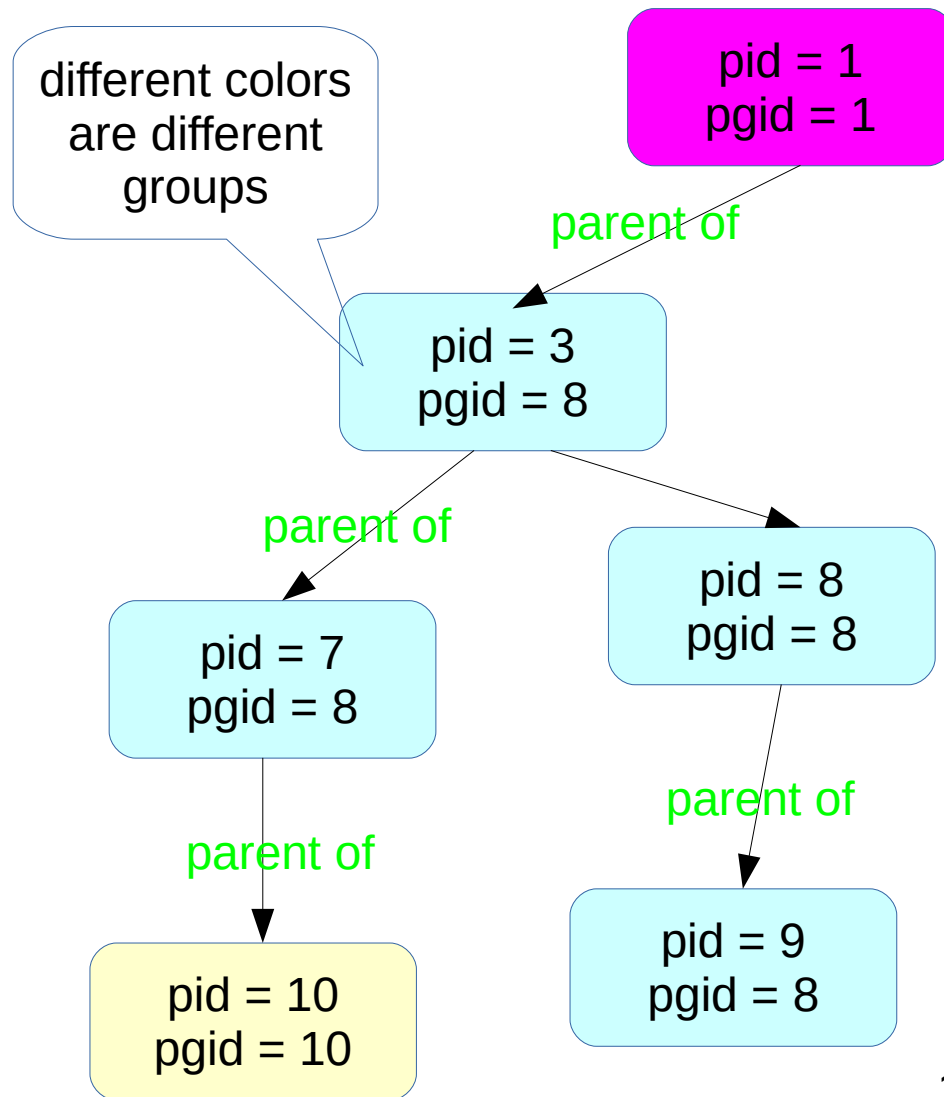
- `pid_t waitpid(pid_t pid, int *status, int options);`
- Wait for the process specified by *pid* to terminate, then return its status
  - the process must be a direct child of the calling process
  - use `pid = -1` to wait for *any* child
  - the return value is the pid of the reaped child, *status* is updated
- Options:
  - `WNOHANG`: return immediately if no children have terminated
  - `WUNTRACED`: report stopped children as well as terminated children
  - `WCONTINUED`: also report continued children
- **Note: this is *not* the way to wait for a child process in tsh!**

# Process Life Cycle



# Process Groups

- A job or application may consist of multiple processes
  - so it's useful to be able to treat them as a unit – called a *process group*
- In addition to a process ID, each process also has a process group ID (*pgid*)
- every process with a specific group ID is considered to be in that process group



## Process Groups: setpgid()

- `int setpgid(pid_t pid, pid_t pgid);`
- Set the process group ID of process *pid* to *pgid*.
  - if *pid* is zero, use the caller's pid
  - if *pgid* is zero, use the caller's pgid (so we don't have to call `getpgrp()`)
- Returns 0 if successful, -1 on failure (and sets `errno` to the reason)
- **You'll need this function for job control in `tsh`.**

# Process States

- Running
  - executing instructions on CPU
  - number of running processes is limited by number of CPU cores
- Runnable
  - would be running if we had more CPU cores
- Blocked
  - waiting for an event, e.g. completion of disk I/O, wake-up signal
- Zombie
  - terminated, but not yet reaped

# Signals

- One form of Exceptional Control Flow
- ECF is a way to react to changes in **system state** rather than program state
- Other types include
  - exceptions
  - nonlocal jumps
  - context switches
  - interrupts

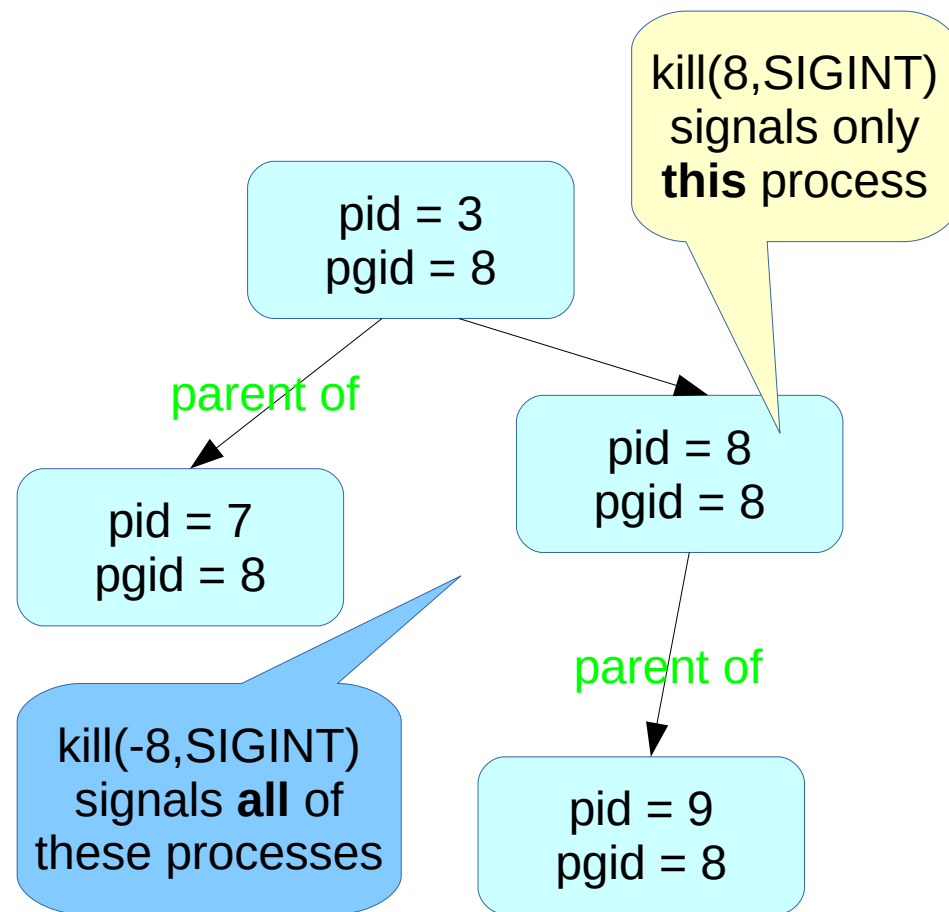


# Signals

- One form of Exceptional Control Flow
- A very basic method of inter-process communication
  - can be sent by the operating system on certain events (Ctrl-C, Ctrl-Z)
  - can be sent by another process
  - no additional information is sent beyond the existence of the signal
  - non-queuing – a parent may receive only one SIGCHLD even if dozens of children terminate and send SIGCHLD
- Possible reactions to a signal
  - ignore it
  - terminate the process (with optional core dump)
  - catch the signal by calling a user-level *signal handler* function

# Sending Signals

- `int kill(pid_t pid, int signal);`
- if *pid* is positive, send the indicated signal to the process with that ID
- if *pid* is **negative**, send the signal to every process with process group ID equal to `-pid`
- returns 0 if at least one signal was sent, -1 on error (and sets `errno`)



# Signal Handlers

- `sighandler_t signal(int signum, sighandler_t handler);`
- install a function *handler* as the signal handler for signal *signum*.
  - use `SIG_IGN` as the handler to ignore the signal
  - use `SIG_DFL` as the handler to perform the default action (kill, ignore, etc.)
- returns the previous handler or `SIG_ERR` on error (and sets `errno`)
- the handler takes an `int` argument which specifies the signal which was caught and returns nothing:
  - `void myhandler(int signum) { ... }`
  - **separate flow of control** in the same process; resumes program on return
- `SIGSTOP` and `SIGKILL` cannot be caught or ignored

# Signals Are Asynchronous

- Signals can happen **at any time**.
  - including while another signal is being handled
  - they **will** eventually occur at the worst possible point in your program!
  - if the signal handler accesses data used by the main program, concurrency bugs can occur
- We need to prevent concurrent access to common data
  - there are techniques for doing so in multi-threaded programs
  - but signals offer a much simpler solution: blocking

# Blocking Signals

- If you're in a critical part of your code, you can temporarily block *most* signals, then unblock them when it's safe again
  - SIGKILL and SIGSTOP **can't** be blocked
- signals that arrive while blocked will be delivered as soon as unblocked
  - but only one occurrence of each signal – if two SIGINTs occur while SIGINT is blocked, the program only gets one when it is unblocked
- which signals to block are specified with a signal set, type `sigset_t`
- build up the signal mask one signal at a time with `sigemptyset()`, `sigfillset()`, `sigaddset()`, and `sigdelset()`
- then use `sigprocmask()` to block the selected signals

# Blocking Signals: Building a Signal Mask

- For a mask consisting of just SIGINT and SIGHUP

```
sigset_t mask;  
  
sigemptyset(&mask);  
sigaddset(&mask, SIGINT);  
sigaddset(&mask, SIGHUP);  
sigprocmask(SIG_SETMASK, &mask,  
            NULL);
```

details of sigprocmask  
are on the next slide

- For a mask including everything **but** SIGINT

```
sigset_t mask;  
  
sigfillset(&mask);  
sigdelset(&mask, SIGINT);  
sigprocmask(SIG_SETMASK, &mask,  
            NULL);
```

## Blocking Signals: Applying the Signal Mask

- `int sigprocmask(int how, const sigset_t *mask, const sigset_t *oldmask);`
- *how* specifies how to merge *mask* with the current blocked signals
  - SIG\_BLOCK: add the signals in *mask* to the set of blocked signals
    - `blocked |= mask`
  - SIG\_UNBLOCK: remove signals in *mask* from the set of blocked signals
    - `blocked &= ~mask`
  - SIG\_SETMASK: *mask* becomes the new set of blocked signals
- if *oldmask* is not NULL, it is filled with the mask before the update
- returns 0 on success, -1 on error (and sets `errno`)

# Waiting For Signals

- `int sigsuspend(const sigset_t *mask);`
- waits for a signal which is **not** in *mask*
  - replace the current signal mask (from `sigprocmask`) with the given mask
  - suspend caller until a signal is received which would terminate the process or invoke a signal handler
  - run the signal handler
  - restore the signal mask
- This is the way to wait for a child's termination without looping while still being able to handle other events
  - just leave `SIGCHLD` and the other events out of *mask*
  - caller does not use any CPU until a signal is received



# Using Signal Blocking and sigsuspend()

```
sigset_t waitmask, newmask, oldmask;
```

```
...code...
```

```
...code...
```

```
...code...
```

```
// make set with only SIGINT
```

```
sigemptyset(&newmask);
```

```
sigaddset(&newmask, SIGINT);
```

```
// block SIGINT and get old mask
```

```
if (sigprocmask(SIG_BLOCK, &newmask,  
                &oldmask) < 0)
```

```
    Error("SIG_BLOCK error");
```

```
// critical code here won't be
```

```
// interrupted by SIGINT
```

```
// restore the original signal mask to
```

```
// unblock SIGINT
```

```
if (sigprocmask(SIG_SETMASK, &oldmask,  
                NULL) < 0)
```

```
    Error("SIG_SETMASK error");
```

What if we needed  
to wait for a SIGINT?

# Using Signal Blocking and sigsuspend()

```
sigset_t waitmask, newmask, oldmask;
// make set with everything but SIGINT
sigfillset(&waitmask);
sigdelset(&waitmask, SIGINT);
// make set with only SIGINT
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

// block SIGINT and get old mask
if (sigprocmask(SIG_BLOCK, &newmask,
                &oldmask) < 0)
    Error("SIG_BLOCK error");

// critical code here won't be
// interrupted by SIGINT
```

```
// done with critical code, now wait for
// a SIGINT
if (sigsuspend(&waitmask) != -1)
    Error("sigsuspend failed");
if (errno != EINTR)
    Error("did not get signal") ;
// sigsuspend restored the signal state
// that was active when it was called
// code here still won't be interrupted
// by SIGINT
// restore the original signal mask to
// unblock SIGINT
if (sigprocmask(SIG_SETMASK, &oldmask,
                NULL) < 0)
    Error("SIG_SETMASK error");
```

# Race Conditions

- Since signals can happen at any time, the order of operations between two processes may be indeterminate
- if results may differ depending on which process goes first, we call that a *race condition*
- if something can go wrong, it eventually **will**
  - concurrent programming requires careful reasoning about *all* possible sequences of events

# Race Conditions

```
// SIGCHLD handler installed
```

```
pid_t child_pid = fork() ;  
if (child_pid == 0) {  
    /* child runs here */  
}  
else {  
    add_job(child_pid);  
}
```

```
void sigchld_handler(int signum) {  
    int status;  
    pid_t child_pid =  
        waitpid(-1, &status, WNOHANG);  
    if (WIFEXITED(status))  
        remove_job(child_pid);  
}
```

Which runs first, add\_job or remove\_job?

# Race Conditions

```
// SIGCHLD handler installed
```

```
pid_t child_pid = fork() ;  
if (child_pid == 0) {  
    /* child runs here */  
}  
else {  
    add_job(child_pid);  
}
```

```
void sigchld_handler(int signum) {  
    int status;  
    pid_t child_pid =  
        waitpid(-1, &status, WNOHANG);  
    if (WIFEXITED(status))  
        remove_job(child_pid);  
}
```

Which runs first, add\_job or remove\_job?  
How can we ensure that add\_job goes first?

# Race Conditions

```
// SIGCHLD handler installed
sigemptyset(&mask);
sigaddset(&mask, SIGCHLD);
sigprocmask(SIG_BLOCK, &mask, NULL);
pid_t child_pid = fork() ;
if (child_pid == 0) {
    /* child runs here */
}
else {
    add_job(child_pid);
}
sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

```
void sigchld_handler(int signum) {
    int status;
    pid_t child_pid =
        waitpid(-1, &status, WNOHANG);
    if (WIFEXITED(status))
        remove_job(child_pid);
}
```

# Alarms

- You can request a SIGALARM signal to be sent at a specific time in the future with `int alarm(int numseconds);`
- Cancel a pending alarm with `alarm(0)`
  - returns the number of seconds remaining until the alarm or 0 if none
- Useful if you need to do something at a specific time even though you might not otherwise be running (e.g. waiting for a signal)

# Unix I/O

- Unix (and thus Linux) uses a unified abstraction for I/O
  - a stream of bytes with a current-location pointer
    - identified by a *file descriptor*
  - applies to files, network connections, hardware devices, etc.
- four basic operations and one optional one
  - `open()` -- open the named file or device and return a descriptor
  - `close()` -- close the given descriptor
  - `read()` -- read from the given descriptor and advance its current location
  - `write()` -- write to the given descriptor and advance its current location
  - `lseek()` -- modify the current location for the descriptor
    - not every underlying object supports seeking – you can't rewind a keyboard!



# Unix I/O: File Descriptors

- A file descriptor is a small non-negative integer
  - think of it as an index into the process's *file descriptor table*
  - a value of -1 is used to denote an error
- Every process has three file descriptors by default:
  - 0 – stdin: the standard input
  - 1 – stdout: the standard output
  - 2 – stderr: the standard error-message output
- You can call `close()` on the default descriptors, but then various standard C library functions will break....

# Unix I/O: Duplicating File Descriptors

- `int newfd = dup(int oldfd);`
  - create a duplicate of the given file descriptor and return it
- `int dup2(int oldfd, int newfd);`
  - close *newfd* if it is open
  - then make *newfd* a duplicate copy of *oldfd*
- the two descriptors can be used interchangeably -- actions on either descriptor will affect the file in the same way
  - `lseek(oldfd)` will change the location of the next `read(newfd)`
- `dup2( )` is handy for implementing I/O redirection in shells
  - use one of the default descriptors as *newfd*

# Redirecting Standard Input

```
#include <unistd.h>
int inputfd = open(infile, O_RDONLY);
if (inputfd == -1) {
    unable_to_open_file();
}
if (dup2(inputfd, STDIN_FILENO) == -1) {
    redirection_error();
}
if (close(inputfd) == -1) {
    file_close_error();
}
```

dup2 must be called  
by child process  
after fork (*why?*)

unistd.h declares names  
for all three standard  
file descriptors  
(also STDOUT\_FILENO  
and STDERR\_FILENO)

# tshlab: Tools

- **./runtrace**

- perform a pre-programmed set of actions on the shell

- **./tshref**

- reference implementation – compare with the operation of your shell

- **./sdriver**

- lets you run traces multiple times and compare against the reference shell

## tshlab: Hints

- Read the starter code and understand what it needs you to implement
  - lots of useful code – don't reinvent the wheel
- Don't use `sleep()` to *try* to solve synchronization problems
  - Especially not for forcing a child or parent to run first
- You will lose points for using tight loops to wait
  - `while (1) { ... } → 0xBADBEEF!`
  - Use `sigsuspend( )` like we've told you
- Suggested order of implementation: process creation, job control, signals/synchronization, I/O redirection
- Shell must forward `SIGINT` and `SIGSTOP` to the foreground job
  - consider using process groups

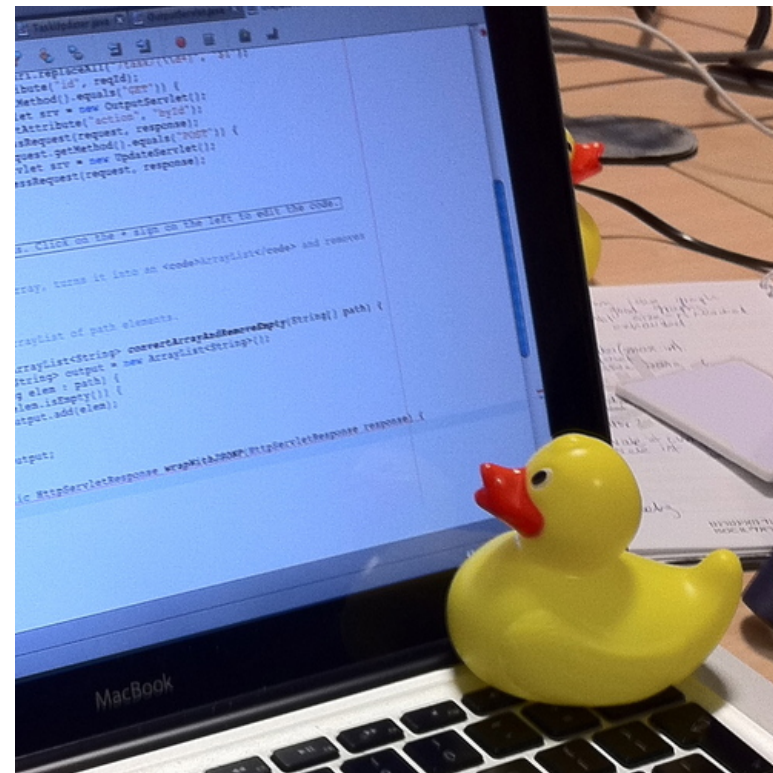
# If You Get Stuck

- *Please read the writeup.* *Please read the writeup!*
- CS:APP Chapters 8 and 10
- Do manual unit tests **before** jumping into runtrace and sdriver!
- View lecture notes and course FAQ at <http://www.cs.cmu.edu/~213>
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man 2 fork, man 2 execve, man 3 exit`
- `man 7 signal, man 2 sigsuspend, man 3 sigsetops`

# Rubber Duck Debugging

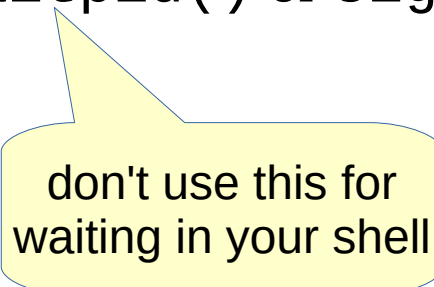
If your program still has a bug, **obtain a rubber duck**. [...] Explain to the duck using simple words why each line of each method in your program is obviously correct. At some point you will be unable to do so, either because you don't understand the method you wrote, or because it's wrong, or both. Concentrate your efforts on that method; *that's probably where the bug is*.

<http://ericlippert.com/2014/03/05/how-to-debug-small-programs/>



# Summary: Process Handling

- create a new process: `fork()`
- run another program in the current process: `execve()`
- terminate the current process: `exit()`
- wait for child process to terminate:
  - `waitpid()` or `sigsuspend(SIGCHLD)`



don't use this for  
waiting in your shell



# Summary: Useful Signal Functions

- set up handlers with `signal()`
- set up signal masks
  - `sigemptyset()` -- create a mask containing **no** signals
  - `sigfullset()` -- create a mask containing **all** signals
  - `sigaddset()` -- add a signal to the given mask
  - `sigdelset()` -- remove a signal from the given mask
- block signals with `sigprocmask()`
- check for pending blocked signals with `sigpending()`
- wait for signals with `sigsuspend()`
- send signals with `kill()`

# Appendix: Programming Style

- Properly document your code
  - header comments, overall operation of large blocks, any tricky bits
- Write robust code – **check error and failure conditions**
  - system calls can frequently fail due to external factors
- **Find your race conditions before we do**
- Formatting
  - 80 characters per line
  - consistent braces and whitespace
- No memory or file descriptor leaks
- See <http://www.cs.cmu.edu/~213/codeStyle.html> for full details