# Anita's Super Awesome Recitation slides

**15/18-213: Introduction to Computer Systems**

**Processes and Signals, 21 October 2013**

**Anita Zhang**

# …And We're Back

- Cache Lab grades are out
  - Autolab → Cachelab → Handin History
  - Look for the latest submission
  - Click "View Source" to see our comments
- Midterms went well
  - Check email for the link to view your exam
  - Email us with grading concerns
- Shell lab is due next Tuesday, October 29 2013

# An "Hour" of Fun Ahead of Us

- Basics of everything
- Processes
  - Birth, Life, Death, After
- Signals
- Sigsuspend
  - So much sigsuspend!
- I/O
- Shell Lab
  - All the hints!

# My (neighbor's) Rabbit (name is fork())

# Exceptional Control Flow

- A way to react to changes in **system state**
  - As opposed to program state
- Types
  - Exceptions
  - Process Context Switch
  - Signals
  - Nonlocal jumps

# Flavors of Exceptions

- Asynchronous
  - I/O interrupts
  - Reset interrupts
- Synchronous
  - Traps
  - Faults
  - Aborts

# Programs? What are those?

- Specification
  - Written according to this to tell users what it does
- Data and instructions stored in an executable binary file
  - Tells a computer what to do
- Binary file is **static**
  - No state, just instructions

# And Then There Were Processes!

- An **instance** of a program in execution
- Ubiquitous on multitasking systems
- A fundamental abstraction provided by the OS
  - Process IDs, Group IDs
  - Single thread of execution (linear control flow)
    - Until you have more threads (more fun ahead..)
  - Full, **private** memory space and registers
  - Various other **states**
    - Open files, private address spaces, etc.
    - Running, Zombie, etc.

# Basics of Process Control

- Four basic process control functions
  - fork()
  - exec()
    - Variations exist
  - exit()
  - wait()
    - Variations exist
- Standard on all Unix-based systems
- CS:APP provides Fork(), Execve(), Wait(), etc.
  - **Error-handling wrappers** provided for your use

# Birth: fork()

- Creates demon spawn
- OS creates an **exact duplicate** of parent's **state**
  - Virtual address space (including heap and stack)
  - Registers, except the return value (%eax)
  - File descriptors (**files are shared)**
  - Exact clone of the program!
- Result: **equal** but **separate** state
- Returns: 0 to child process, child's PID to parent
  - Returns -1 on failure
- Can return execution in an arbitrary order
  - Either child/parent may run first after fork()

# Life: execve (char* filename, char** argv, char** environ)

- **Replaces** the current process's state and context
- This is how you run programs
  - Replace current memory image with new program
  - Sets up stack
  - Start execution at the entry point
- Newly loaded program's perspective: **as if the previous program has not been run before**
  - On success, it does not return to the old program

# LIFE:
## EXECVE (CHAR* FILENAME, CHAR** ARGV, CHAR** ENVIRON)

- Arguments
  - filename
    - Absolute path of the file to run
  - argv
    - Command line arguments to the new program
  - environ
    - Environment variable
    - Information that affects the various ways a process works
    - Declaring `extern char** environ` sets it up to default
      - `#include <unistd.h>`

# Death: exit (int status)

- Terminates a process
- OS frees resources used by exited process
  - Heap, open file descriptors, etc.
  - But not exit status!
- The process becomes a **zombie**
  - Technical terminology
  - Remains in process table to await its reaping
- Zombies are reaped when their parents read their exit status
  - Done by init process if the parent has died
  - Then the PID can be reused~ :D

## Reap:
## waitpid (pid_t pid, int* status, int options)

- Waits for a child process to change state
- If a child has terminated, this allows the parent to "reap" the child
  - Frees all resources
  - Collects the exit status
  - Child is "fully" gone D:
- Only reaps direct children
  - Not grandchildren or great-grandchildren, etc
- Status pointer must be in valid memory
  - `wait()` uses it to fill in the status of the reaped child

# Reap:
# waitpid (pid_t pid, int* status, int options)

- Arguments
  - pid
    - Process ID of the child to wait for
    - -1 to wait on ANY child
  - status
    - Pointer to space to fill in the status information
    - Can be read with built-in macros
      - `WIFEXITED`
      - `WEXITSTATUS`
      - `WIFSIGNALED`
      - And more!
  - options
    - Things that make `wait()` behave differently
      - `WUNTRACED`
      - `WNOHANG`
      - And more!

# ADDITIONAL USEFULNESS: SETPGID (PID_T PID, PIT_T PGID)

- Sets the process group ID of process with process ID pid
- By default children inherit parent's group ID
- Arguments:
  - pid
    - Apply to process with ID pid
    - If 0, `setpgid()` is applied to the calling process
  - pgid
    - Set group ID to pgid
    - If 0, `setpgid()` uses `pgid = pid` of the calling process

# WHICH RUNS FIRST?

```
pid_t child_pid = fork();

if (child_pid == 0) {
   /* only child prints this */
   printf("Child!\n");
   exit(0);
} else {
   printf("Parent!\n");
}
```

- What are the possible outcomes?
  - Child!
    Parent!
  - Parent!
    Child!
- How can we get the child to always print first?

# Which Runs First?

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

if (child_pid == 0) {
   /* only child prints this */
   printf("Child!\n");
   exit(0);
} else {
   waitpid(child_pid, &status, 0);
   printf("Parent!\n");
}
```

- Use waitpid() to wait until a child has terminated
  - Exit status can be inspected using the status variable here
- Only one outcome
  - Child!
    Parent!

# Using execve()

```
int status;
pid_t child_pid = fork();
char* argv[] = {“ls”, “-l”, NULL};
extern char **environ;

if (child_pid == 0){
   /* only child comes here */
   execve(“/bin/ls”, argv, environ);
   /* will child reach here? */
} else {
   waitpid(child_pid, &status, 0);
}
```

- argv
  - Argument list
  - Convention: argv[0] is the name of the executable
- execve
  - const char *filename
  - char *argv[]
  - char const envp[]
    - environ provided by unistd.h
    - Can also specify your own

# Process States

- Running
  - Executing instructions on the CPU
  - Number bounded by number of CPU cores
- Runnable
  - Waiting to run
- Blocked
  - Waiting for an event
  - Not runnable
- Zombie
  - Terminated, not yet reaped

# What are these "Signal" things?

- Primitive form of inter-process communication
- Notifies a process of an event
- **Asynchronous** with normal execution
- Comes in several flavors
  - man 7 signal
- Sent in various ways
  - ctrl +c, ctrl+z
  - `kill()`

# Signals

- Are **non-queuing**
  - If we block SIGCHLD, and multiple SIGCHLD arrive, we only receive one SIGCHLD when we unblock
  - Can receive multiple types (ie. SIGCHLD & SIGINT)
- Options for handling signals
  - Ignore
  - Catch and run signal handler
  - Terminate (and optionally dump core)

# More on Signals

- Many have default behaviors
  - `SIGINT, SIGTERM` will terminate the process
  - `SIGSTOP` will suspend the process until it receives `SIGCONT`
  - `SIGCHLD` is sent from a child to its parent when the child changes state
- Can ignore/catch most signals, but not some
  - `SIGKILL` cannot be caught, blocked, or ignored
  - `SIGSTOP` cannot be caught, blocked, or ignored

# USEFUL SIGNAL SYSCALLS

- Setting up handlers
  - `signal()`
- Setting up signal masks
  - `sigemptyset()`
  - `sigfullset()`
  - `sigaddset()`
  - `sigdelset()`
- Blocking signals
  - `sigprocmask()`
- Waiting for signals
  - `sigsuspend()`
- Sending signals
  - `kill()`

# Signal Handlers

- Can run handler when particular signal received
  - `void  handlername (int signum) { …. }`
- **Separate flow of control** in the same process
- Resumes program upon returning
- Can be called anytime when the signal is fired
- `Signal(int signum, sighandler_t handler)`
  - When a signal is caught, runs the installed handler (or default)

# Concurrency Bugs

```
void handler(int sig)
{
    pid_t pid;
    /* Reap a zombie child */
    while ((pid = waitpid(-1, NULL, 0)) > 0)
        deletejob(pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
}
```

- What could happen between fork() and addjob()?
  - SIGCHLD
- How would you handle it?
  - Block in the right places

```
int main(int argc, char **argv)
{
    int pid;

    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

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

# Why sigsuspend()?

- What is `sigsuspend()`?
  - Used to protect critical regions from signal interruption.
  - It is especially useful for (you guessed it) "pausing" or "sleeping" while waiting for a signal.
  - Much better solution to the "sleep loop"
- Goal: to block all the way up until the instruction our process is suspended.

# ABOUT SIGSUSPEND()

- `int sigsuspend(const sigset_t *sigmask);`
  - Where `sigmask` contains a mask of signals YOU DON'T want to be interrupted by
  - Can be considered opposite of `sigprocmask()` which takes a mask of signals you want to operate on.
- Quick example: if you want to be woken up from `sigsuspend()` by SIGCHLD, it better not be in the mask you pass in!

# How to sigsuspend()

```
int main() {
    sigset_t waitmask, newmask, oldmask;

    /* set with everything except SIGINT */
    sigfillset(&waitmask);
    sigdelset(&waitmask, SIGINT);

    /* set with only SIGINT */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /* oldmask contains the mask of signals before the
     * block with newmask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        unix_error("SIG_BLOCK error");

    /* "CRITICAL REGION OF CODE" - (SIGINT blocked) */

    /* Pause, allowing ONLY SIGINT */
    if (sigsuspend(&waitmask) != -1)
        unix_error("sigsuspend error");

    /* RETURN FROM SIGSUSPEND -- (Returns to signal
     * state from before sigsuspend) */
    /* Reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        unix_error("SIG_SETMASK error");
}
```

- Points of interest
  - `sigprocmask()` fills oldmask with the signal mask from before SIG_BLOCK
  - If `sigsuspend()` returns from being awoken, it returns 1.
  - After `sigsuspend()` returns, the state of the signals returns to how it was before the call

# I/O

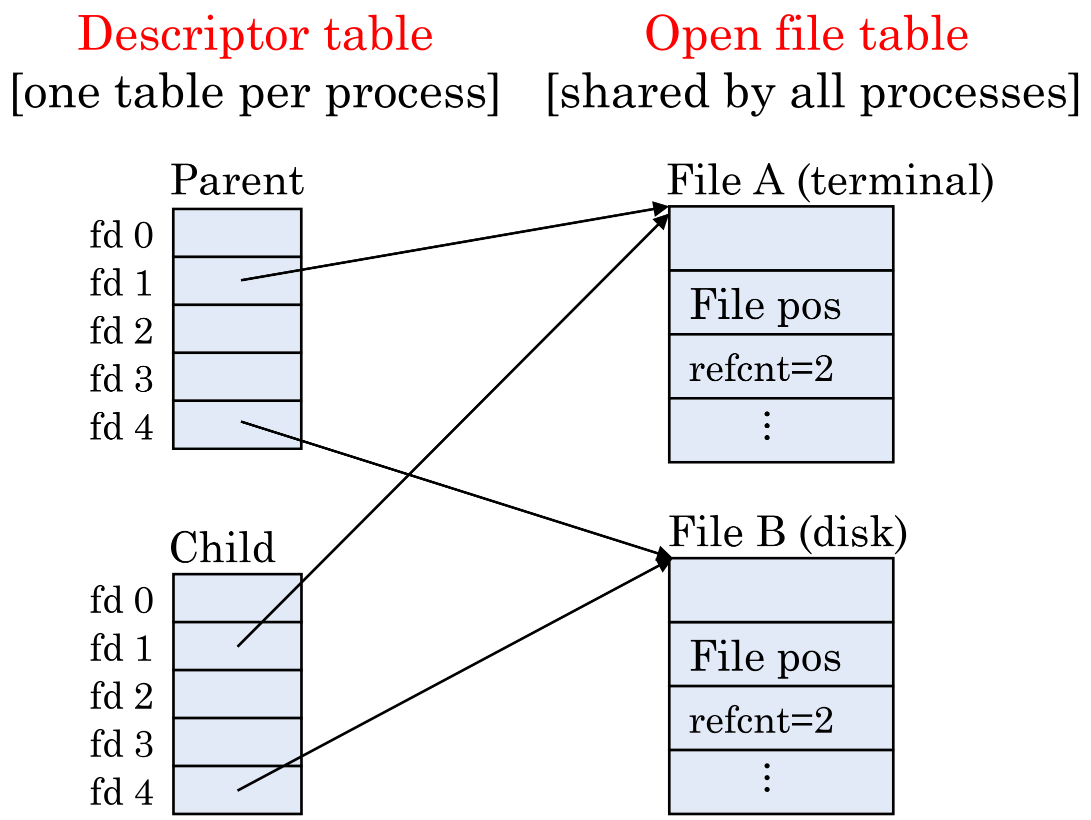
- Four basic operations (operate on file descriptors)
  - open()
  - close()
  - read()
  - write()
- What's a file descriptor?
  - Returned by open()
  - Some positive value, or -1 to denote error
  - `int fd = open("/path/to/file", O_RDONLY);`

# File Descriptors

- Every process starts with these 3 by default
  - 0 – STDIN
  - 1 – STDOUT
  - 2 – STDERR
- You can call close() on them…..
  - But you that's probably not what you want
- Every process gets its own **file descriptor table**
- All processes share open file tables

# Parent and Child After fork()

- Shamelessly stolen from lecture:

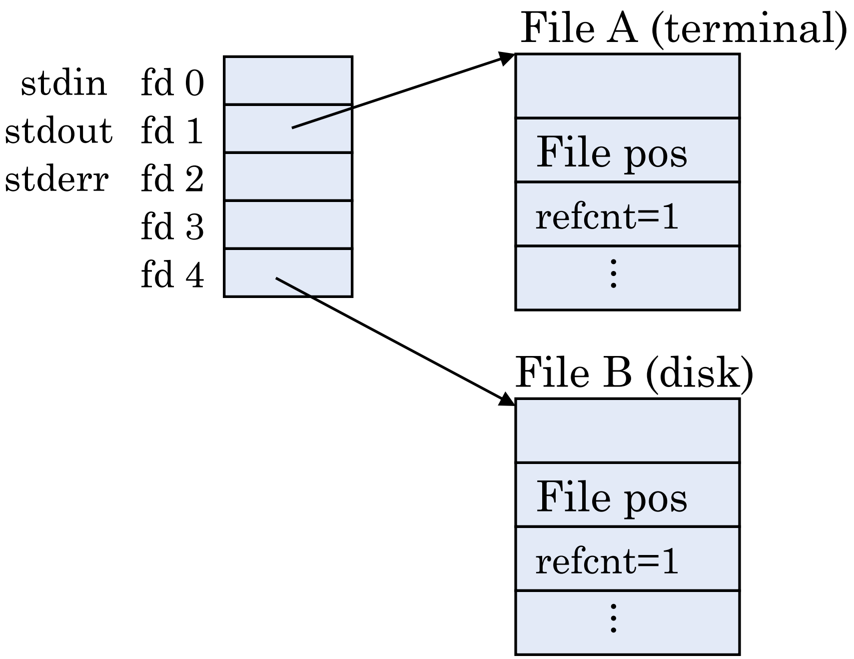


# What is dup2()?

- Copies file descriptor entries
  - Causes the entries to point to the same files as another file descriptor
- Takes the form: `dup2(dest_fd, src_fd)`
  - `src_fd` will now point to the same place as `dest_fd`

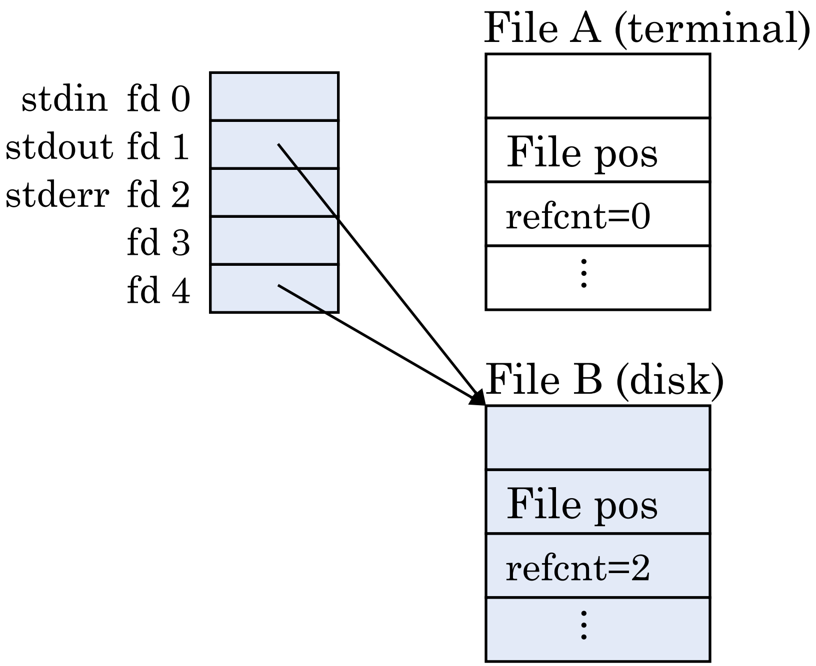
# DUP2() Super Relevant: Before

- Goal: Redirect stdout
- First, use `open()` to open a file to redirect
  - For Shell Lab: Done right before the call to `exec()` in the child process
  - This example, fd 4 is the file descriptor of the opened file



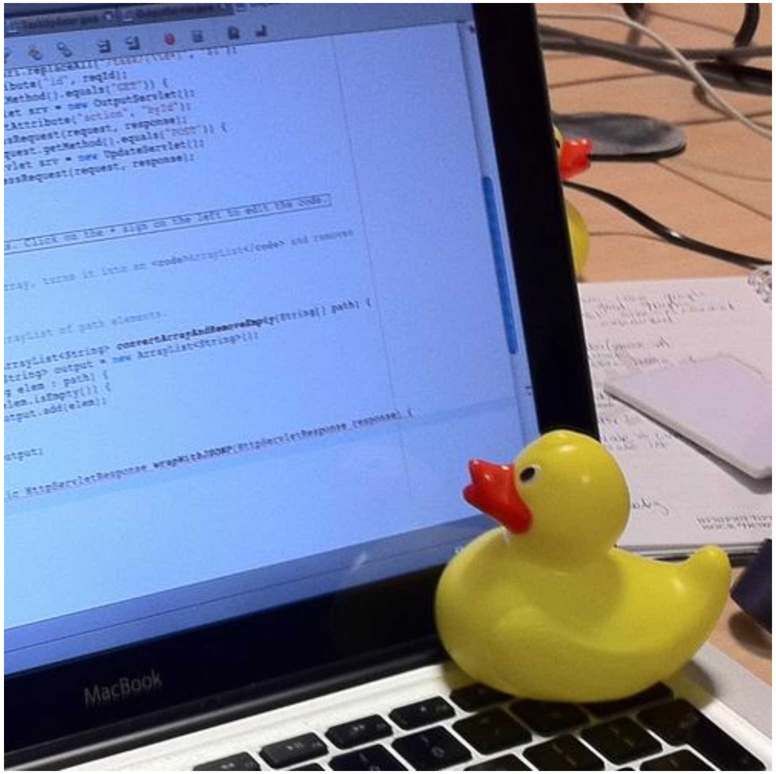
# dup2() Super Relevant: After

- To redirect, duplicate fd 4 into fd 1.
- Call `dup2(4, 1)`
  - Causes fd 1 to refer to disk file pointed at by fd 4
- Accessing fd 1 will now get you File B



# Rubber Duck Debugging

"To use this process, a programmer explains code to an inanimate object, such as a rubber duck, with the expectation that upon reaching a piece of incorrect code and trying to explain it, the programmer will notice the error."



# Has Everyone Seen the Duck?

# Shell Lab

- Race conditions
- Creating processes
- Reaping zombies
- Job control synchronization
- I/O redirection
- Managing signals
- And more!

# Shell Lab Tools

- `./runtrace`
  - Runs traces on your chosen shell (defaults to tsh)
  - Execute without arguments to see usage
- `./tshref`
  - Reference shell – experiment, run programs, etc.
- `./sdriver`
  - Used to run traces multiple times
  - Execute without arguments to see usage

# Plan of Attack

- As always, **read the handout**
  - Bundles of hints in there
- If there is one chapter to read from the textbook..
  - CS:APP: Chapter 8 – Exceptional Control Flow
  - **Tons** of examples and explanations on how to synchronize your processes
    - They're pretty much giving you the answers…
    - At least read the example code
- Suggested order: Job control/ process creation, signals and synchronization, I/O redirection
- Unit test by hand
  - Don't jump into the `sdriver` or `runtrace` too soon

# Hints

- CS:APP p.735 and p.757
  - Basic `eval()` starter codes
  - Great way to start the lab
  - Code links in the credits
- Read the starter code, understand what it wants
  - We do all the job and parsing work for you!
- Don’t use sleep() to solve synchronization issues
  - Definitely don’t use it to make a child/parent run first
  - Will lose points for using tight loops to wait
    - `while(1) { … } ← 0xBADBEEF!!!!`
    - `sigsuspend()`
      - We already told you to use it

# More Hints

- Shell must forward `SIGINT` and `SIGSTP` to the foreground job (and all its children)
  - How could process groups be useful?
- `dup2` is a handy function for I/O redirection
- `SIGCHILD` handler may have to reap multiple children per call
- Try actually running your shell
  - Can be easier to debug this way
  - Strangely satisfying to write a working shell!
  - Compare output to reference shell

# Style

- Check return values
  - You're dealing with system calls; they matter a lot
- Provided code is a good example of what we expect from you
  - Relevant comments and explanations of design
- Find your race conditions before we do
- 10 points for style. Make it count.

# This Slide Intentionally Filled

Questions?

- Fork Photo Credit
- CS:APP Error Handling Wrappers and Header
- Poking with Stick Picture
- CS:APP Code Samples
- Rubber Duck 1
- Rubber Duck Debugging on Wiki
- Florentijn Hofman's Duck