...And We’re Back

- Shell lab is due next Thursday, March 28 2013

- Midterms available at the ECE Course Hub
  - HH 1112

- How was break?
AN “HOUR” OF FUN AHEAD OF US

- Basics of everything
- Processes
  - Birth, Life, Death, Reap
- Signals
- Brief I/O
- Shell Lab

It gets OS real from here.
My (neighbor’s) Rabbit
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
  - 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.
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**)
- Result: **equal** but **separate** state
- Returns: **0 to child process, child’s PID to parent**
- Can run execution in an arbitrary order
  - Either child/parent may run first after fork()
**LIFE: exec()**

- **Replaces** the current process’s state and context
- This is how you run programs
  - Replace current running memory image with that of the new program
  - Set 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
- A family of functions
  - For details: man 3 exec
**DEATH: exit()**

- 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: WAIT()

- 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:
- Variations exists
  - Details: man 2 wait
Which Runs First?

```c
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?**

```c
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()

```c
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()`
  - `kill utility`
POKING WITH A STICK ANALOGY

- Gotta be in recitation for this one...
Signals

- Are **non-queuing**
- Options for handling signals
  - Ignore
  - Catch and run signal handler
  - Terminate (and optionally dump core)
  - Details: man sigaction
- Blocking signals
  - sigprocmask()
- Waiting for signals
  - sigsuspend()
- Can’t modify behavior of SIGKILL and SIGSTOP
**Signal Handlers**

- Can be installed to run when a particular signal is received
  - `void handler (int signum) { .... }`
- Separate flow of control in the same process
- Resumes normal flow of control upon returning
- Can be called anytime when the appropriate signal is fired
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");
}

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);
}

What could happen between fork() and addjob()?
- SIGCHLD

How would you handle it?
- Block in the right places
I/O

- Four basic operations
  - `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
- All processes share v-node tables
  - Contains the stat structure with info about a file
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 fast
HINTS

- **CS:APP p.735 and p.757**
  - Basic eval and job management starter codes
  - Great way to start the lab
  - Code links in the credits
- **Read the starter code, understand what it wants**
  - There are hints in there too
- **Don’t use sleep() to solve synchronization issues**
  - Definitely don’t use it to make a child/parent run first
  - Google or man pages for sigsuspend()
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