

Lecture 5

Systems Programming:
Unix Processes Creation
Pipes

Unix Process Creation

Creation

Management

Destruction

examples are in

`~mark/pub/51081/processes`

Process Attributes

- Process ID:

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
pid_t getpid(void);
```

- Every unix process has an associated process id (pid)
- each new process is assigned a new unique unused pid
- The pid is a 32bit unsigned integer, which usually ranges from 0 to 32767
- pids roll over after 32767 and assignment begins again at 0, issuing *unused* pids

Process Ids and init

- Every process on the system has a parent, with the exception of pid 1: init
 - the init process “hangs around”, it is responsible for the initialization and booting of the system, and for running any new programs, like the login program, and your shell
 - Init executes /etc/rc* files during initialization, and is the ultimate parent of every subsequent process in the system
 - If init is killed, the system shuts down
- Any process’s parent id (ppid) can be obtained with the `pid_t getppid(void)` call.

Death and Destruction

- All processes usually end at some time during runtime (with the exception of init)
- Processes may end either by:
 - executing a `return` from the main function
 - calling the `exit(int)` function
 - calling the `_exit(int)` function
 - calling the `abort(void)` function
 - generates SIGABRT signal, core dumps and then exits
- When a process exits, the OS delivers a termination status to the parent process of the recently deceased process

Environments

- All processes by default inherit the environment of their parent process
- The environment can be obtained through the `char * environ[]` variable.
- `char * getenv(const char * name)` will return the associated value for the *name* passed in:
 - `char * path = getenv("PATH");`
- `int setenv(const char * name, const char * value, int overwrite)` will set an environment variable
- *examples: environ.c*

The Spawn

- `exec()`
- `fork()`
- `system()`
- `clone()`

The exec() Functions:

Out with the old, in with the new

- The exec() functions all *replace* the current program running within the process with *another* program
- bring up an xterm:
 - `exec sleep 5` #what happens and *why*?
- There are two families of exec() functions, the “l” family (list), and the “v” family (vector)
- Each exec() call can choose different ways of finding the executable and whether the environment is delivered in the form of a list or an array (vector)
- The environment, open file handles, etc. are passed into the exec'd program
- What is the return value of an exec() call?

The execl... functions

- `int execl(const char * path, const char * arg0, ...);`
 - executes the command at `path`, passing it the environment as a *list*: `arg0 ... argn`
 - thus, the `execl` family breaks down `argv` into its individual constituents, and then passes them as a *list* to the `execl?` function (the *l* stands for *list*)
- `int execlp(const char * path, const char * arg0, ...);`
 - same as `execl`, but uses `$PATH` resolution for locating the program in *path*, thus an absolute pathname is not necessary
- `int execlenv(const char * path, const char * arg0, ... char * envp[]);`
 - allows you to specifically set the new program's environment, which *replaces* the default current program's environment
- *examples: params.c, execl.test.c, execlenv.test.c, environ2.c, execlp.test.c, sash.c*

The execv... functions

- `int execv(const char * path, char *const argv[]);`
 - executes the command at `path`, passing it the environment contained in a single `argv[]` *vector*
- `int execvp(const char * path, char *const argv[]);`
same as `execv`, but uses `$PATH` resolution for locating the program in *path*
- `int execve(const char * path, char *const argv[], char * const envp[]);`
 - note that this is the only system call of the lot
- *examples: execv.test & myecho.c*

fork()

- fork() *creates* a *new* child process
- the OS copies the current program into the *new process*, resets the program pointer to the start of the new program (child fork location), and both processes *continue execution independently as two separate processes*
- The child gets its own *copy* of the parent's:
 - data segments
 - heap segment
 - stack segment
 - file descriptors

fork() Return Values

- fork() is the one Unix function that is called *once* but returns *twice*:
- If fork() returns 0:
 - you're in the *new* child process
- If fork() returns > 1 (i.e., the pid of the new child process)
 - you're back in the parent process
- *examples: fork1.c, forkio.c*

Waiting on Our Children

- Unlike life, parents should always hang around for their children's lives (runtimes) to end, that is to say:
 - Parent processes should always wait for their child processes to end
- When a child process dies, a SIGCHLD signal is sent to the parent as notification
- The SIGCHLD signal's default disposition is to ignore the signal
- A parent can find out the exit status of a child process by calling one of the wait() functions

Waiting on Our Children

- Parent processes find out the exit status of their children by executing a wait() call:
 - `pid_t wait(int * status);`
 - `pid_t waitpid(pid_t pid, int * status, int options);`
- Wait() *blocks* until it receives the exit status from a child
- Waitpid can wait on a specific child, and doesn't necessarily block (WNOHANG)
- Waiting allows the parent to obtain the *return value* from the child's process
- *examples:*
 - *childdeath echo hi*
 - *forkandwait echo hello world*
 - *forkandwait sleep 10*

waitpid()

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

- pid can be any of 4 values:
 - < -1: wait for any child whose gpid is the same as pid
 - == -1: waits for any child to terminate
 - == 0: waits for a child in the same process group as the current process
 - > 0: waits for process *pid* to exit
- The following macros work on status:
 - WIFEXITED(status): true if process exited normally
 - WIFSIGNALED(status): true if process was killed by a signal
 - *examples: forkandwait2 sleep 15*

Problem Children: Orphans and Zombies

- If a child process exits before it's parent has called `wait()`, it would be inefficient to keep the entire child process around, since all the parent is going to want to know about is the exit status:
 - A *zombie* is a **child** process that that has exited *before it's parent's has called `wait()`* for the child's exit status
 - A zombie holds nothing but the child's exit status (held in the program control block)
 - Modern Unix systems have `init (pid == 1)` *adopt* zombies after their parents die, so that zombies do not hang around forever as they used to, in case the parent never did get around to calling `wait`

Problem Children: Orphans and Zombies

- If a parent process dies *before* it's child, the child process becomes an *orphan*
 - An orphan is a **child** process whose parent is no longer living
 - An *orphan* is immediately “adopted” by the init process (pid == 1), who will call wait() on behalf of the deceased parent when the child dies
- *examples: myzombie.c, myorphan.c*

vfork() and Copy On Write

- When a process forks, the entire current process (plus segments, environment, etc.) is copied over to the new process
- When that new process called `exec()`, the entire address space is replaced (overlaid) with the new environment of the `exec`'ing program
- Efficiency question: If you know you're going to call `exec` immediately after `fork`, why have `fork` spend time copying the entire address space over when we know it's just going to get overwritten immediately on the `exec()` call?
- Answer: `vfork()` doesn't copy entire address space

system()

- `int system(const char * cmd)`
- `system()` forks a child process that exec's `/bin/sh`, which in turn runs the command `cmd`
- As such, it has the following qualities:
 - it's easy and familiar to use
 - it's inefficient
 - because it uses system variables and executes from a shell, it can be a security risk if the command is `setuid` or `setgid`
- example: `system("ls -la /usr/bin");`

Sessions and Process Groups

- A process group is a group of related processes, that share some common interest, as all the processes in a pipeline do:
 - `ls -l | sort | wc -l`
- A session is a further abstracted group of related process groups or individual processes, such as all the jobs in a given terminal shell session
- Sessions are generally created during login, and process groups are managed by the job processing capabilities of a given shell

Priorities and Being Nice

- The scheduler recognizes processes of three different scheduling policies:
 - SCHED_FIFO (unalterable real-time processes)
 - SCHED_RR (alterable real-time processes)
 - SCHED_OTHER (conventional, time-shared)
- Processes with SCHED_OTHER policy are assigned a default dynamic priority of 0, and can voluntarily lower their priority by incrementally raising their “niceness” value, up to 10 (range is -20 to +19, effectively 1 – 40 in terms of process priority)
- *example: mynice.c*
- *gcc -O0 -g -o mynice mynice.c*
- *mynice [nice]*

Beginner's Guide to Writing a Shell

- Define a buffer to hold a command entered from the command line
- Create a forever loop that forever prompts for a new command
- Block on a read (fgets, etc.) and allow the user to enter a command
- Parse the command into parameters for exec
- fork() a child process
- have the child process exec() the parsed command
- have the parent wait on the child process to finish

Debugging Multiple Processes

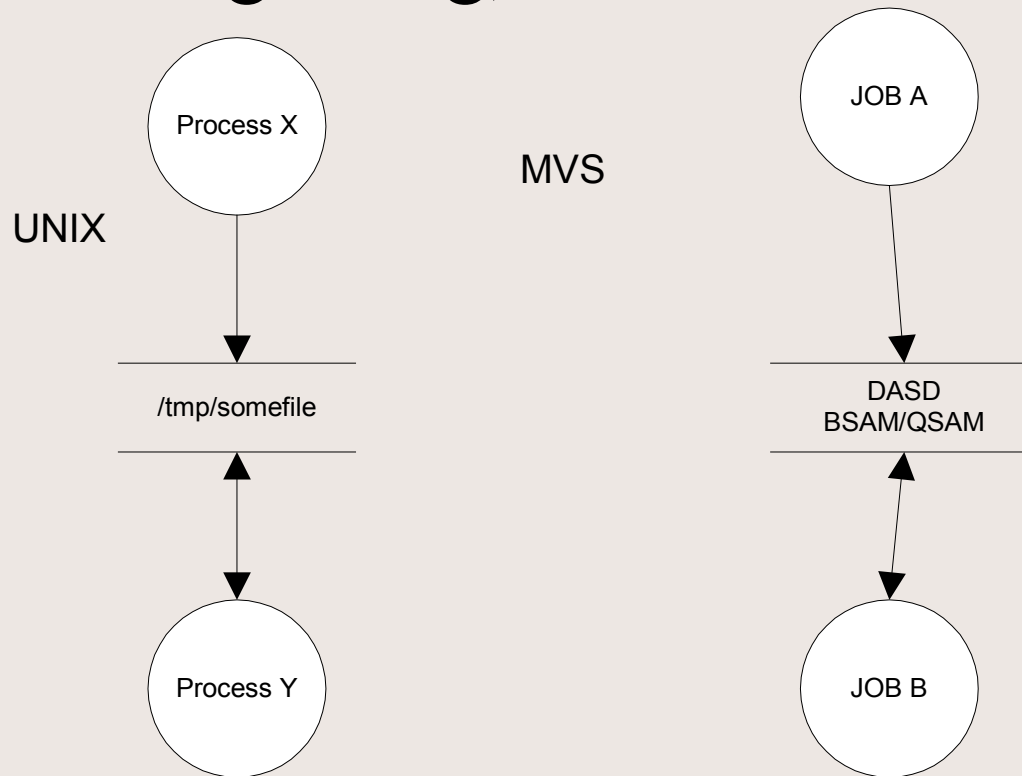
- Debugging processes that fork can be a little tricky, because whereas once you had one process, now you have two.
- Which process will gdb debug? Answer: the parent
- How do you debug the child process?
 - With another gdb (ddd) session:
 - Add a sleep() call at the start of the child code
 - run gdb (ddd) on the program, and set a breakpoint right after the sleep() call in the child section
 - run the first gdb session on the parent
 - after the fork(), attach to the child process and then issue the “continue” call in the child gdb session
- *example: forkdebug.c*

Pipes

Interprocess Communication using
pipes

Motivation: Batch Sequential Data Processing

- In the beginning, there was a void...



Px > /tmp/somefile
Py < /tmp/somefile

Batch Sequential Data Processing

- Stand-alone programs would operate on data, producing a file as output
- This file would stand as input to another stand-alone program, which would read the file in, process it, and write another file out
- Each program was dependent on its version of input before it could begin processing
- Therefore processing took place sequentially, where each process in a fixed sequence would run to completion, producing an output file in some new format, and then the next step would begin

Pipes and Filters Features

- Incremental delivery: data is output as work is conducted
- Concurrent (non-sequential) processing, data *flows* through the pipeline in a stream, so multiple filters can be working on different parts of the data stream simultaneously (in different processes or threads)
- Filters work *independently and ignorantly* of one another, and therefore are plug-and-play
- Filters are ignorant of other filters in the pipeline --there are no filter-filter interdependencies
- Maintenance is again isolated to individual filters, which are loosely coupled
- Very good at supporting producer-consumer mechanisms
- Multiple readers and writers are possible

What is a pipe?

- A pipe is an interface between two processes that allows those two processes to communicate (i.e., pass data back and forth)
- A pipe connects the STDOUT of one process (writer) and the STDIN of another (reader)
- A pipe is represented by an array of two file descriptors, each of which, instead of referencing a normal disk file, represent input and output paths for interprocess communication
- Examples:
 - `ls | sort`
 - `ypcat passwd | awk -F: '{print $1}' | sort`
 - `echo "2 + 3" | bc`

How to create a pipe (lowlevel)

- `#include <unistd.h>`
- `int pipe(int pipefd[2]);`
- `pipefd` represents the pipe, and data written to `pipefd[1]` (think `STDOUT`) can be read from `pipefd[0]` (think `STDIN`)
- `pipe()` returns 0 if successful
- `pipe()` returns `-1` if unsuccessful, and sets the reason for failure in `errno` (accessible through `perror()`)
- *examples: `pipe2.c`*

Pipe One-Niner, Come in

- Pipes are half duplex by default, meaning that one pipe is opened specifically for unidirectional writing, and the other is opened for unidirectional reading (i.e., there is a specific “read” end and “write” end of the pipe)
- The net effect of this is that across a given pipe, only one process does the writing (the “writer”), and the other does the reading (the “reader”)
- If two way communication is necessary, two separate pipe() calls must be made, or, use SVR5’s full duplex capability (stream pipes)
- *examples: fullduplex.c (compile and run on linux and solaris (SVR5))*

Traditional Pipes

- How would you mimic the following command in a program:
 - `$ ls /usr/bin | sort`
- Create the pipe
- associate stdin and stdout with the proper read/write pipes via `dup2()` call
- close unneeded ends of the pipe
- call `exec()`
- *example: `ls_sort.c`*

Pipes the easy way: popen()

- The simplest way (and like system() vs. fork(), the most expensive way) to create a pipe is to use popen():
 - `#include <stdio.h>`
 - `FILE * popen(const char * cmd, const char * type);`
 - `ptr = popen("/usr/bin/ls", "r");`
- popen() is similar to fopen(), except popen() returns a pipe via a FILE *
- you close the pipe via pclose(FILE *);

popen()

- When called, popen() does the following:
 - creates a new process
 - creates a pipe to the new process, and assigns it to either stdin or stdout (depending on char * type)
 - “r”: you will be reading *from* the executing command
 - “w”: you will be writing *to* the executing command
 - executes the command cmd via a bourne shell
- *example: pipe_echo.c*

Meanwhile, back at the ranch...

- One thing is in common between all the examples we've seen so far:
 - All our examples have had *shared file descriptors*, shared from a parent processes forking a child process, which *inherits* the open file descriptors as part of the parent's environment for the pipe
- Question: How do two entirely *unrelated* processes communicate via a pipe?

FIFOs: Named Pipes

- FIFOs are “named” in the sense that they have a name in the filesystem
- This common name is used by two separate processes to communicate over a pipe
- The command `mknod` can be used to create a FIFO:
 - `mkfifo MYFIFO` (or “`mknod MYFIFO p`”)
 - `ls -l`
 - `echo “hello world” >MYFIFO &`
 - `ls -l`
 - `cat <MYFIFO`

Creating FIFOs in code

- `#include <sys/types.h>`
- `#include <sys/stat.h>`
- `int mkfifo(const char * path, mode_t mode);`
 - path is the pathname to the FIFO to be created on the filesystem
 - mode is a bitmask of permissions for the file, modified by the default umask
- `mkfifo` returns 0 on success, -1 on failure and sets `errno` (`perror()`)
- `mkfifo("MYFIFO", 0666);`
- *examples: reader.c, writer.c*