15-213 Recitation 8

Introduction to Computer Systems

Fahim Dalvi

24 October, 2013

Today

- Control Flow
  - Signals
- Unix I/O
- Shell lab

But first...

- Tarek is going to hold a workshop on control flow
  - Will provide a lot of examples that will solidify your ideas
  - Will help you both in the lab and in the exams

“Process”

- An instance of an executing program
- Important characteristics
  - Private memory → Two processes do not share memory/registers
  - Have a process ID and group ID
  - On “dying”, every process is a zombie
    - It must be reaped by someone
  - Processes may share some things, like file tables

Important functions

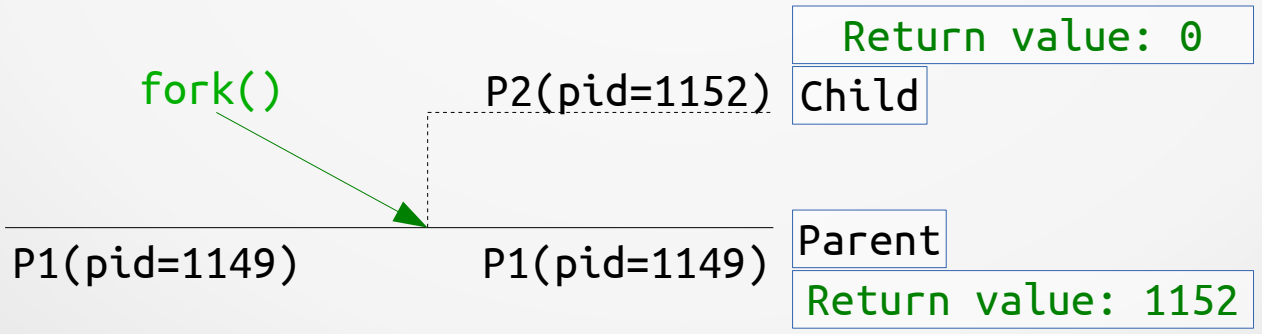
- exit()
- fork()
- execvp()/execl()/execlp()/execle()/execv()/execve()
- waitpid()
- setpgid()

exit()

- Takes exactly one argument, an integer
- Immediately terminates the process that called it
- Sets 'return status' to the input argument
- Leaves a zombie to be reaped by the parent with *wait()* or *waitpid()*

fork()

- Clones the current process
- Returns twice (one in the parent, one in the child)
- Return value in child is 0, child's pid in parent
- Returns -1 in case of failure



exec()

- `execvp(char* filename, char** argv)`
  - Replaces current process with a new one
  - Does not return (or returns `-1` on failure)
  - `filename` is the name of the program to run
  - `argv` are like the command-line arguments to main for the new process
    - `argv[0]` is always the "command name", actual arguments start from `argv[1]` onwards.

Demo time!

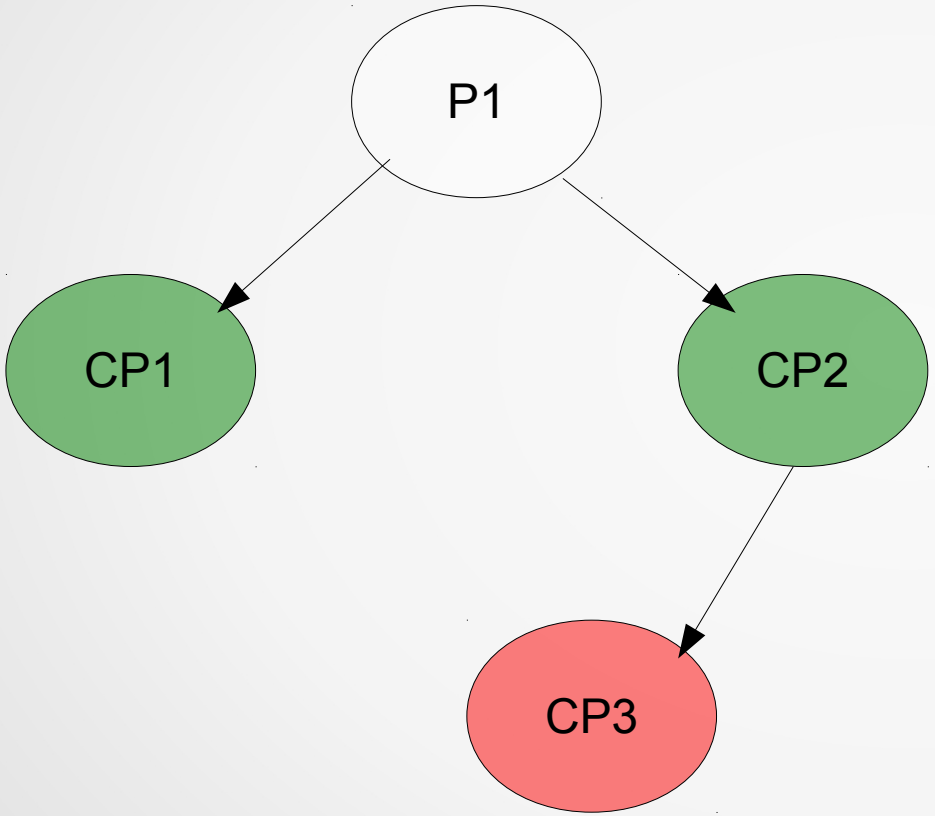
```
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t result = fork();
    printf("Print 1!\n");
    if (result == 0)
    {
        char* cmd = "/bin/echo";
        char* args[] = {cmd, "hello","world","from","child"};
        execvp(cmd,args);
        printf("Print 2!\n");
    }
    else
    {
        printf("Print 3!\n");
    }
    return 0;
}
```

waitpid()

- wait(pid_t pid, int* status, int options)
  - Returns when the process specified by pid terminates
  - pid must be a direct child of the invoking process
  - If pid=-1, will wait for any child to die
  - Writes information about child's status into status
  - Options variable modifies its behavior
  - options = WUNTRACED | WNOHANG
  - Returns pid of the child it reaped
  - Required by parent to kill zombies/free their resources

Reaping



P1 can only reap CP1/CP2, but not CP3

setpgid()

- setpgid(pid_t pid, pit_t pgid)
  - Sets the pgid of the given pid
  - If pid=0, setpgid is applied to the calling process
  - If pgid=0, setpgid uses pgid=pid of the calling process
  - Children inherit the pgid of their parents by default

Signals

- Basic communication between processes
- Sent several ways (kill command/function, ctrl-c, ctrl-z)
- Many 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
- Possible to ignore/catch most signals, but some can't
  - SIGKILL is unstoppable SIGINT
  - SIGSTOP is unstoppable SIGSTP

Blocked 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
- Process will only track that it has received a blocked signal, but not the number of times it was received

Signals – Important Functions

- kill()
- signal()
- sigprocmask()

kill()

- kill(pid_t id, int sig)
  - If id positive, sends signal sig to process with pid=id
  - If id negative, sends signal sig to all processes with with pgid=-id

signal()

- signal(int signum, sighandler_t handler)
  - Specifies a handler function to run when signum is received
  - sighandler_t means a function which takes in one int argument and is void (returns nothing)
  - When a signal is caught using the handler,its default behavior is ignored
  - The handler can interrupt the process at any time, even while either it or another signal handler is running
  - Control flow of the main program is restored once it's finished running
  - SIGKILL,SIGSTOP cannot be caught

sigprocmask()

- sigprocmask(int option, const sigset_t* set, sigset_t *oldSet)
  - Updates the mask of blocked/unblocked signals using the handler signal set
  - Blocked signals are ignored until unblocked
    - Process only tracks whether it has received a blocked signal, not the count
    - Getting SIGCHILD 20 times while blocked then unblocking will only run its handler once
  - option: SIG_BLOCK,SIG_UNBLOCK,SIG_SETMASK
  - The 'set' can be modified using sigemptyset() and sigaddset()

Race Conditions

- Race conditions occur when sequence or timing of events are random or unknown
- Signal handlers will interrupt currently running code
- When forking, child or parent may run in different order
- If something can go wrong, it will!
  - Must reason carefully about the possible sequence of events in concurrent programs

Demo time!

```
#include <stdio.h>
#include <signal.h>

int counter = 1;

void handler(int signum)
{
  counter--;
}

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

- What are the possible outputs?
- What if we wanted to guarantee that the handler executed after the print statement?
- Heads up: You will need to do this at some point in shell lab

Unix IO

- Each process maintains a table of its “open” files, and references to them
- This table is copied over on forking
  - Hence, files that are open in one process would remain open in the forked process

Demo time!

```
int main()
{
    int fd = open(“ab.txt”, O_RDONLY);
    char c;
    fork();
    read(fd,&c,1); //Read one character from the file
    printf(“%c\n”,c); //Print the character
}
```

- Assume the file ab.txt contains “ab”
- What's the output?
- What if the process forked before opening the file?

Shell Lab

- Due: 4th November, Monday
- Requires you to implement a 'shell' aka what you see when you login to the unix/shark machines
- Writeup has a page full of hints
  - Read these very carefully to avoid wasted effort

Shell Lab

- There's a lot of starter code
  - Look over it so you don't needlessly repeat work
- Use the reference shell to figure out the shell's behavior
  - For instance, the format of the output when a job is stopped
- Be careful of the add/remove job race condition
  - Jobs should be removed from the list in the SIGCHILD handler
  - But what if the child ends so quickly, the parent hasn't added it yet?