Processes, Signals, I/O, Shell Lab

15-213: Introduction to Computer Systems

Recitation 9: 10/21/2013

Tommy Klein

Section B

Agenda

- News
- Processes
  - Overview
  - Important functions
- Signals
  - Overview
  - Important functions
  - Race conditions
- I/O Intro
- Shell Lab General

News

- Cachelab grades are out
  - Autolab->Cache Lab->View Handin History
  - Look for the latest submission
  - Click 'View Source' to read our annotations/comments
- Midterm grades were good
- Check answers to identify possible errors with the rubric
  - Email us with concerns
- Shell lab out, due Tuesday 10/29 11:59PM

Processes

- An instance of an executing program
- Abstraction provided by the operating system
- Properties
  - Private memory
    - No two processes share memory, registers, etc.
  - Some state is shared, such as open file table
  - Have a process ID and process group ID
    - pid,pgid
  - Become zombies when finished running

Processes – Important Functions

- exit (int n)
  - Immediately terminates the process that called it
  - Sets return status to n
    - Return status is normally the return value of main()
  - 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

Processes – Important Functions

- execve(char* filename, char** argv, char** environ)
  - 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
  - Environ is the environment variable
    - Contains information that affects how various processes work
    - On shark machines, can get its value by declaring: “extern char** environ;”

Fork/exec – “echo hello world”

```
extern char** environ;
int main()
{
    pid_t result = fork();
    printf(“This prints in both the parent and child!\n”);
    if (result == 0)
    {
        //Execute only in child
        char* cmd = “/bin/echo”;
        char* args[] = {cmd, “hello”,”world”};
        execve(cmd,args,environ);
        printf(“This will only print if execve failed!\n”);
    }
    else
    {
        //Execute only in parent
        printf(“In the parent!\n”);
    }
}
```

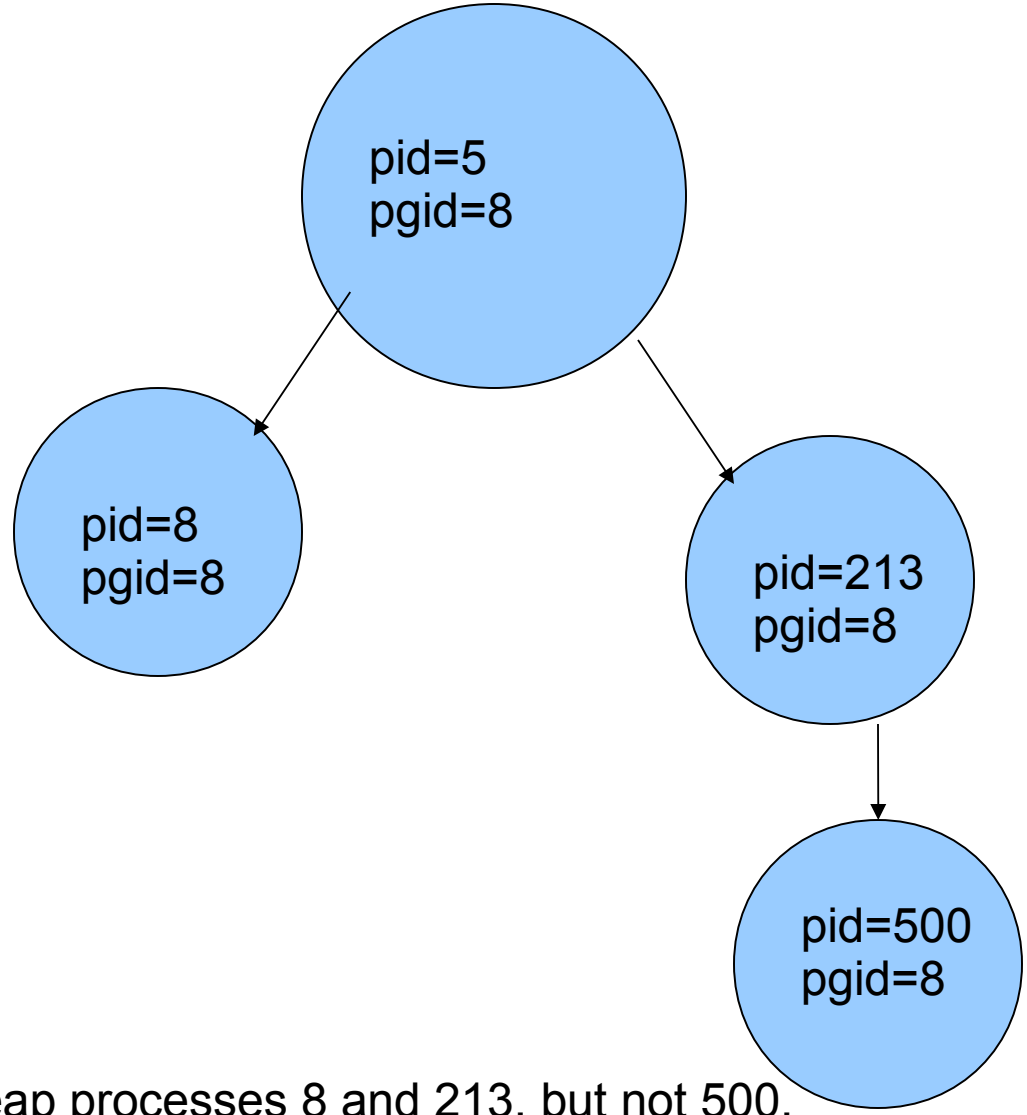
Processes – Important Functions

- waitpid(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
  - Will reap/cleanup the child
  - 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

Processes – Important Functions

- 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

Process Group Diagram



process 5 can reap processes 8 and 213, but not 500.
Only process 213 can reap process 500.

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

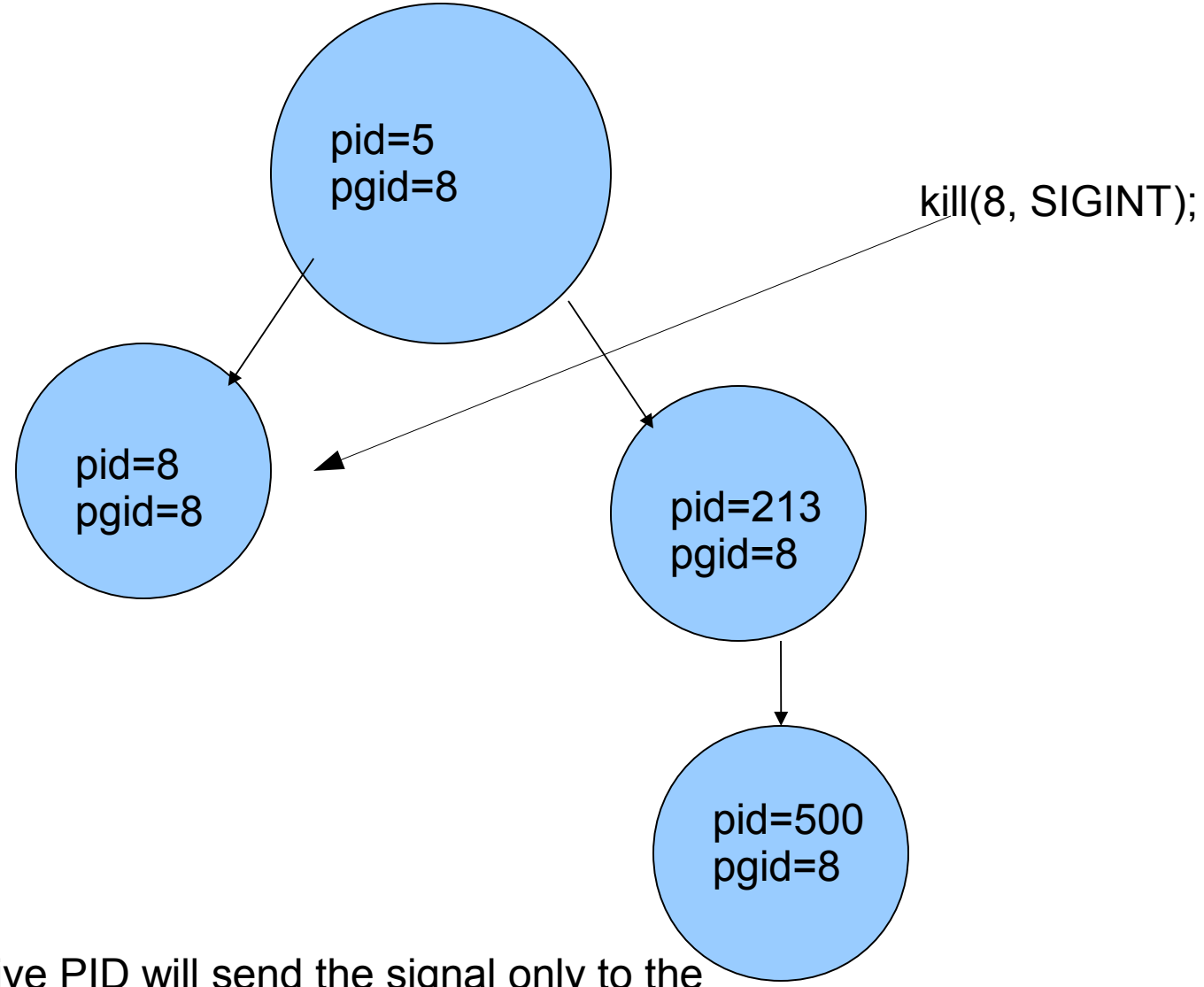
Blocked Signals

- 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

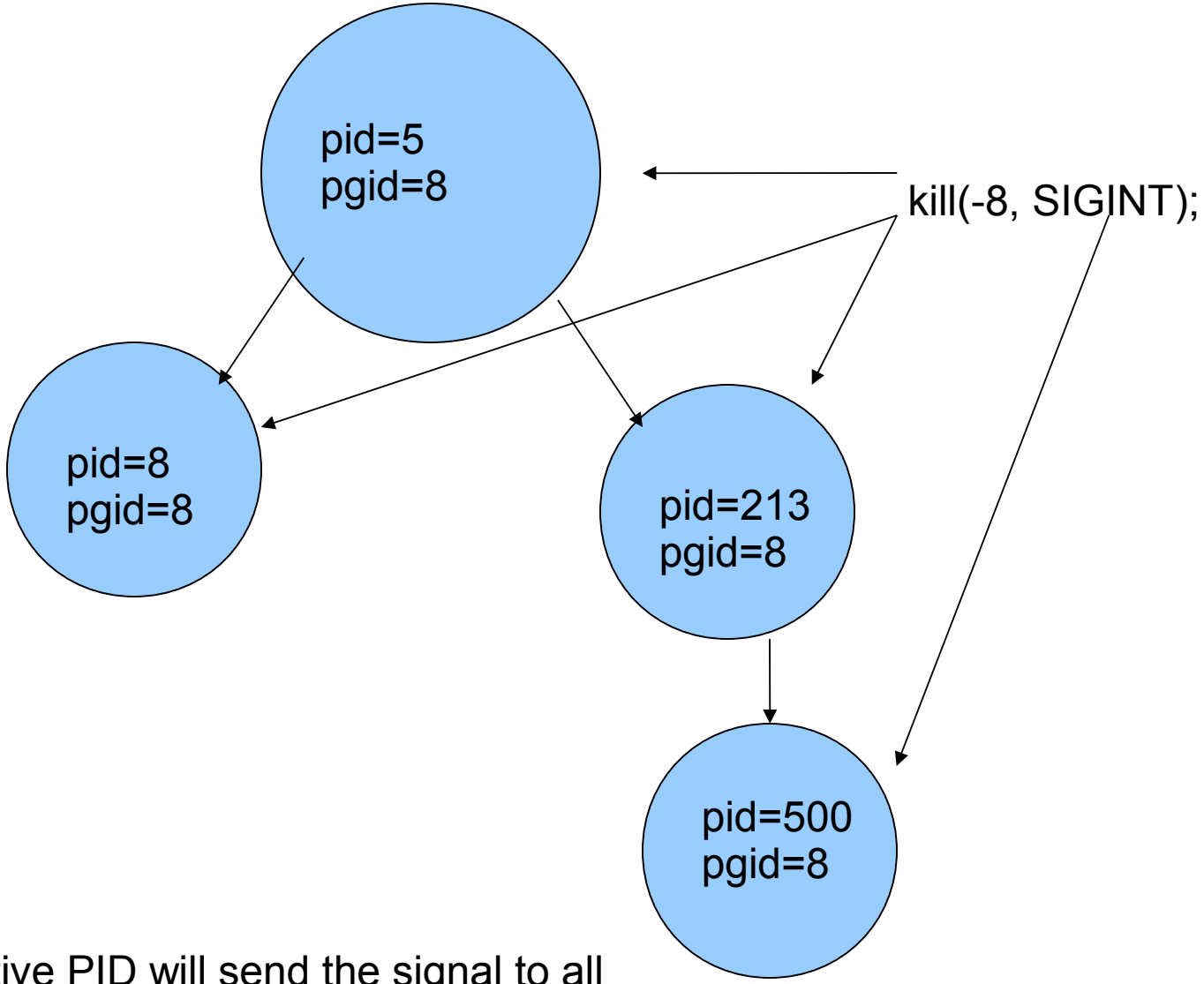
- kill(pid_t id, int sig)
  - If id positive, sends signal sig to process with pid=id
  - If id negative, sends signal sig to all processes with with pgid=-id

Kill - Process



kill() with a positive PID will send the signal only to the process with that ID.

Kill – Process Group



kill() with a negative PID will send the signal to all processes with that group ID.

Signals – Important Functions

- 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

Signals – Important Functions

- Sigsetops
  - A family of functions used to modify signal sets
  - Sigsets correspond sets of signals, which can be used in other functions
  - http://linux.die.net/man/3/sigsetops
  - Remember to pass in the address of the sets, not the sets themselves

Signals – Important Functions

- 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
  - Signal mask's old value is written into oldSet

Signals – Important Functions

- sigsuspend(sigset_t *tempMask)
  - Temporarily replaces the signal mask of the process with tempMask
  - Sigsuspend will return once it receives an unblocked signal (and after its handler has run)
  - Good to stop code execution until receiving a signal
  - Once sigsuspend returns, it automatically reverts the process signal mask to its old value

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

Race Conditions - Signals

```
int counter = 1;
void handler(int signum)
{
    counter--;
}
int main()
{
    signal(SIGALRM,handler);
    kill(0,SIGALRM);
    counter++;
    printf("%d\n",counter);
}
```

- Possible outputs?
- What if we wanted to guarantee that the handler executed after the print statement?
- Tip: you'll face a similar problem adding/removing jobs in Shell Lab...

Race Conditions – Handler After

```
int counter = 1;
void handler(int signum)
{
    counter--;
}
int main()
{
    signal(SIGALRM,handler);
    sigset_t alarmset,oldset;
    sigemptyset(&alarmset);
    sigaddset(&alarmset,SIGALRM);
    //Block SIGALRM from triggering the handler
    sigprocmask(SIG_BLOCK,&alarmset,&oldset);
    kill(0,SIGALRM);
    counter++;
    printf("%d\n",counter);
    //Let the pending or incoming SIGALRM trigger the handler
    sigprocmask(SIG_UNBLOCK,&alarmset,NULL);
}
```

Unix I/O

- All Unix I/O,from network sockets to text files, are based on one interface
- Important distinction between *file descriptors* and *open file description*
  - I/O commands such as open will generate an open file description and a file descriptor
  - A file descriptor is like a pointer to an open file description
  - Note that the open file table is at the OS-level and shared between all processes, while there is one file descriptor table per process
  - Multiple file descriptors, either from the same or different processes, can point to the same OFD

Unix I/O

```
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 do the file tables look like?
- What's the output?
- What if the process forked before opening the file?

Shell Lab Tips

- 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?
- Use sigsuspend, not waitpid, to wait for foreground jobs
  - You will lose points for using tight loops (while(1) {}), sleeps to wait for the foreground

Shell Lab Tips

- Shell requires SIGINT and SIGSTP to be fowarded to the foreground job (and all its descendants) of the shell
  - How could process groups be useful?
- dup2 is a handy function for the last section, I/O redirection
- SIGCHILD handler may have to reap multiple children per call
- Try actually using your shell and seeing if/where it fails
  - Can be easier than looking at the driver output