

Processes, Signals, I/O, Shell Lab

15-213: Introduction to Computer Systems

Agenda

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

Processes

- An instance of an executing program
- Abstraction provided by the operating system
- Properties
 - Have a process ID(pid) and process group ID(pgid)
 - Private state – memory, registers, etc.
 - Shared state - such as open file table
 - Become zombies when finished running(why?)

Process: fork()



- Prototype:
pid_t fork(void);
- Clones the current process. The new process gets a new pid, but the same pgid.
- The new process is an exact duplicate of the parent's state. It has its own stack, own registers, etc.
- It has its own file descriptors (but the files themselves are shared).
- Called once, returns twice (once in the parent, once in the child).
- Return value in child is 0, child's pid in parent. (This is how the parent can keep track of who its child is.)
- Returns -1 in case of failure.
- After the fork, we do not know which process will run first, the parent or the child.

Process: `execve()`

- Prototype:
*int execve(const char *filename, char *const argv[],
char *const envp[]);*
- Replaces the current process with a new one. The binary corresponding to '*filename*' will be run by current process.
- Called once; does not return (or returns -1 on failure).
- `fork() + execve()` creates a new process and runs a new binary on it. This is the usual way of running a new process.

Process: `exit()`

- Prototype:
`void exit(int status);`
- Immediately terminates the process that called it. The process goes to Zombie state.
- `status` is normally the return value of `main()`.
- The OS frees the resources (heap, file descriptors, etc.) but not its exit status. It remains in the process table to await its reaping.
- Zombies are reaped when their parents read their exit status. (If the parent is dead, this is done by `init`.) Then its `pid` can be reused.

Process: waitpid()

- Prototype:
*pid_t waitpid(pid_t pid, int *status, int options);*
- The wait family of functions allows a parent to know when a child has changed state (e.g., terminated).
- waitpid returns when the process specified by pid terminates.
- *pid* must be a direct child of the invoking process.
- If *pid* = -1, it will wait for any child of the current process.
- Return value: the pid of the child it reaped.
- Writes to *status*: information about the child's status.
- *options* variable: used to modify waitpid's behavior.
 - WNOHANG: keep executing caller until a child terminates.
 - WUNTRACED: report stopped children too.
 - WCONTINUED: report continued children too

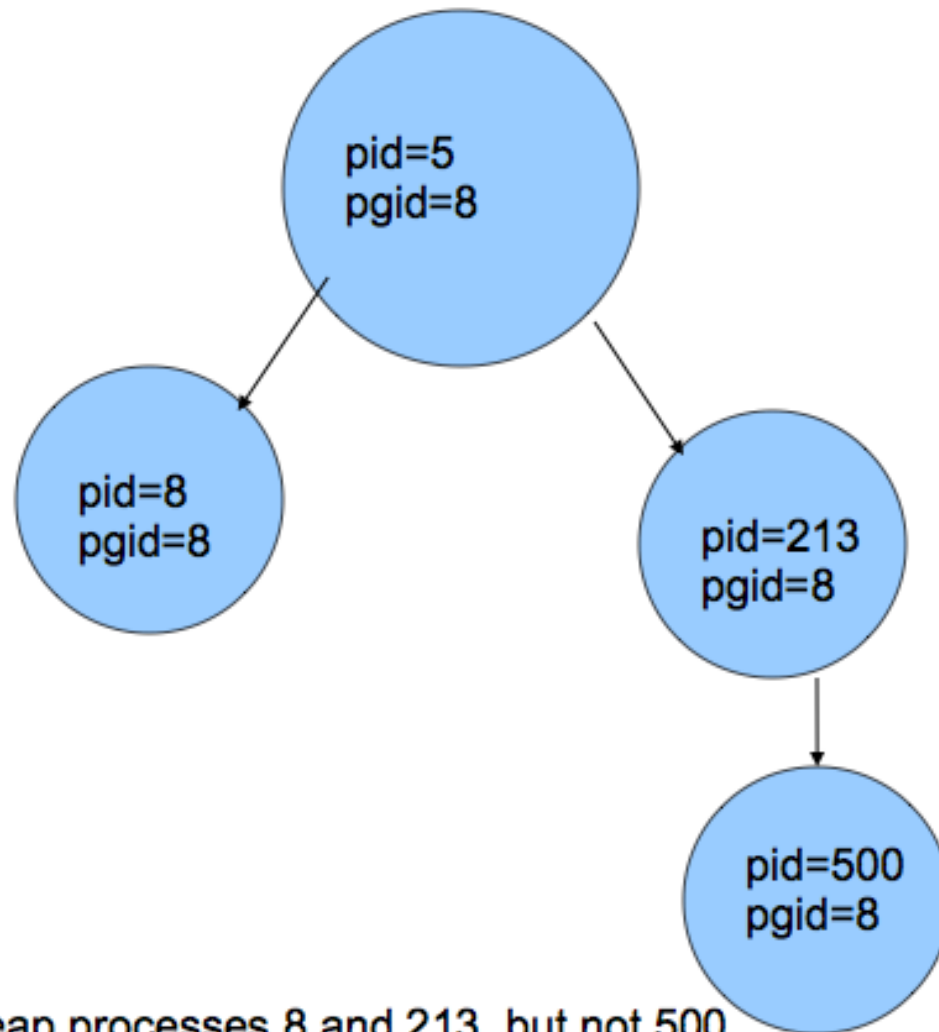
Processes – setpgid()

- Prototype:

`setpgid(pid_t pid, pid_t pgid)`

- Sets the process group id(`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.

Concurrency!

```
pid_t child_pid = fork();  
  
if (child_pid == 0) {  
    printf("Child!\n");  
    exit(0);  
}  
  
else {  
    printf("Parent!\n");  
}
```

Output?

Concurrency!

```
pid_t child_pid = fork();  
  
if (child_pid == 0) {  
    printf("Child!\n");  
    exit(0);  
}  
  
else {  
    printf("Parent!\n");  
}
```

Two possible Outcomes:

- Child!
Parent!
- Parent!
Child

Concurrency!

```
pid_t child_pid = fork();

if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}

else {
    printf("Parent!\n");
}
```

Two possible Outcomes:

- Child!
Parent!
- Parent!
Child

```
int status;

pid_t child_pid = fork();

if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}

else {
    waitpid(child_pid, &status, 0);
    printf("Parent!\n");
}
```

Only one possible Outcome:

Child!
Parent!

Agenda

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

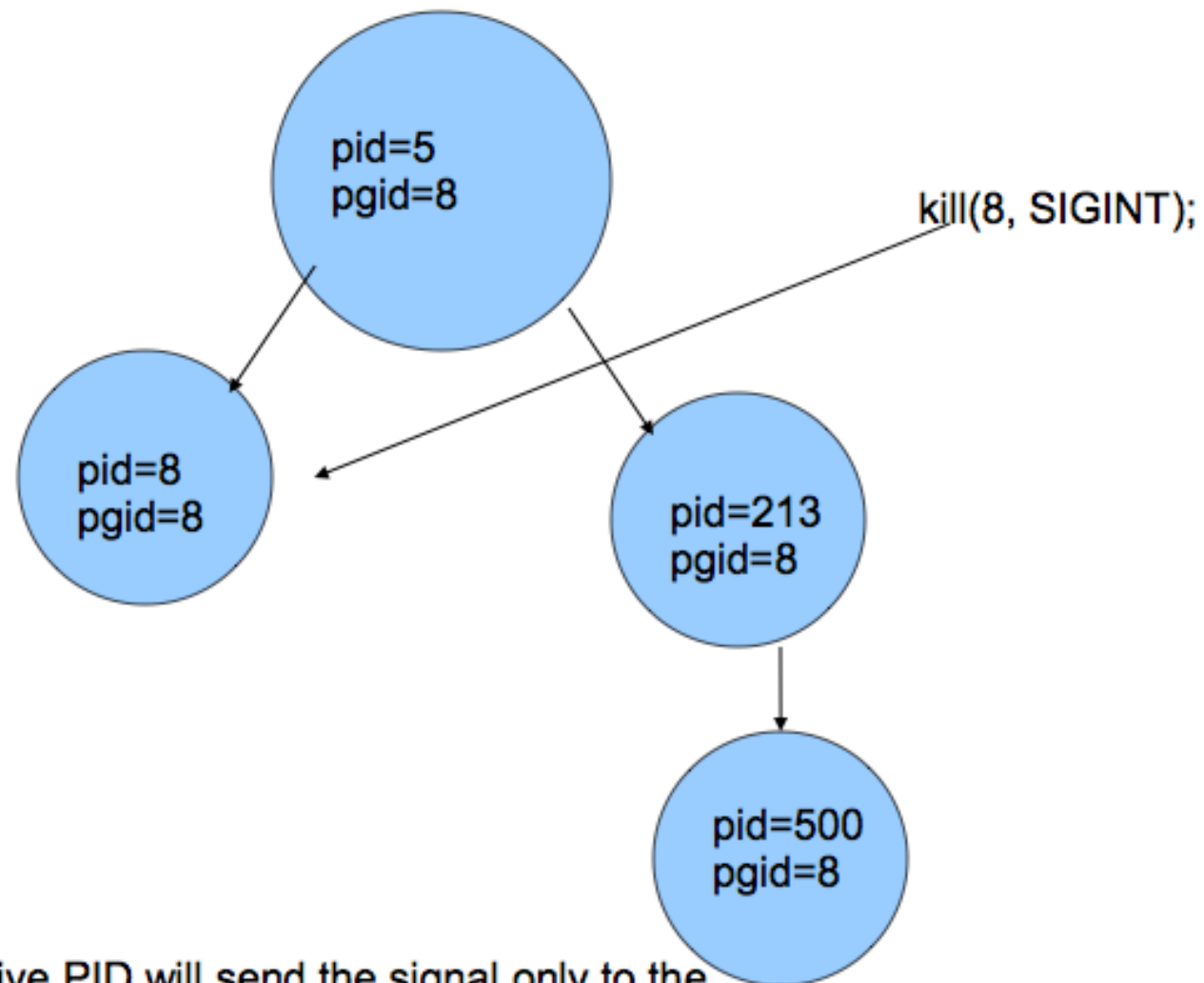
Signals

- Signals are the basic way processes communicate with each other. They notify a process that an event has occurred (for example, that its child has terminated).
- They are sent several ways: Ctrl-C, Ctrl-Z, kill()
- Signals are asynchronous. They aren't necessarily received immediately; they're received right after a context switch.
- They are non-queuing.
 - There is only one bit in the context per signal
 - If 100 child processes die and send a SIGCHLD, the parent may still only receive one SIGCHLD
- Three possible ways to react when a signal is received:
 - Ignore the signal (do nothing)
 - Terminate the process (with optional core dump)
 - Catch the signal by executing a user-level function called signal handler

Sending a signal

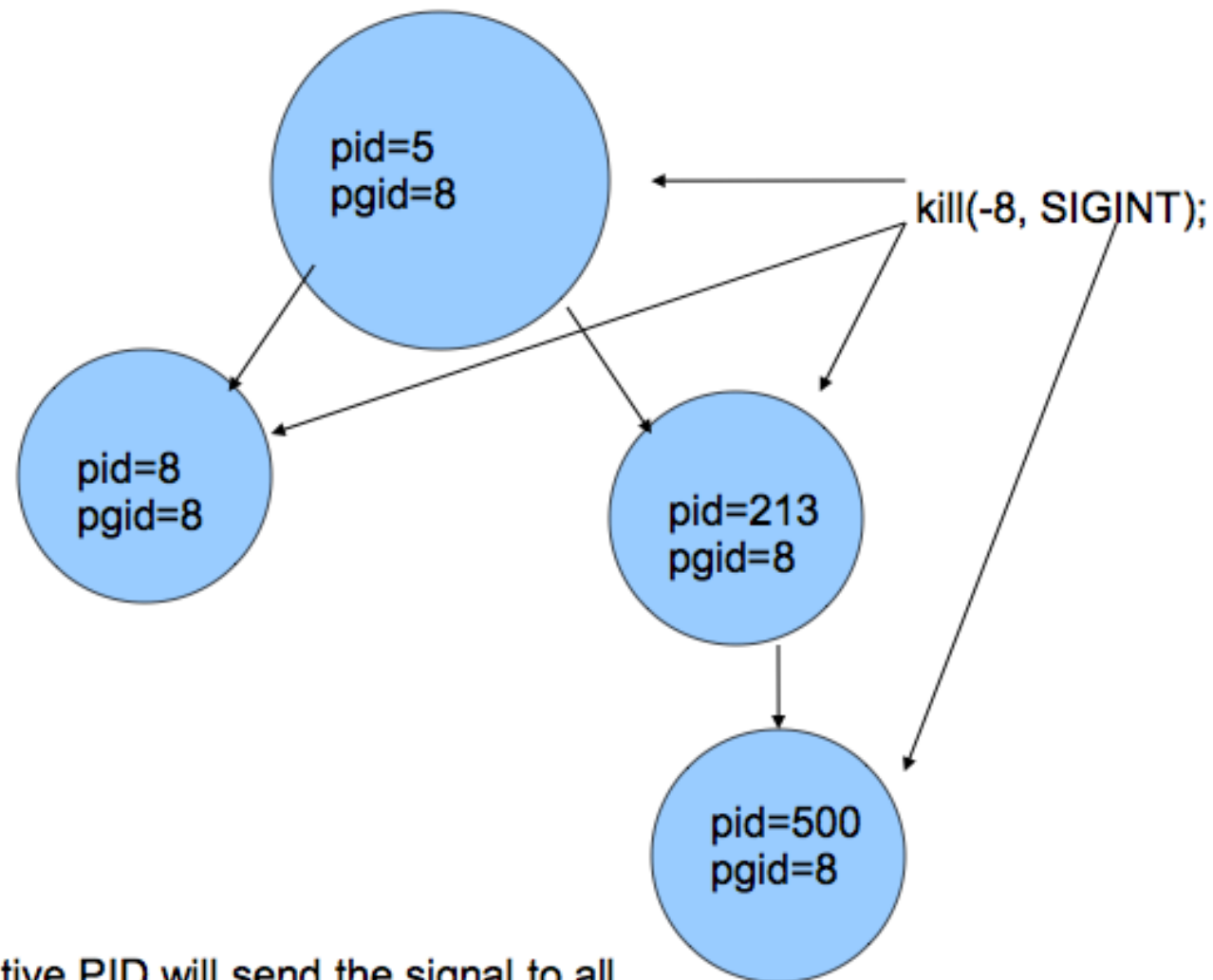
- `kill(pid_t id, int sig)`
 - If `id` positive, sends signal `sig` to process with `pid=id`
 - If `id` negative, sends signal `sig` to all processes with `pgid=-id`

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.

Handling signals

- `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

Caveat

- Remember Signals are received asynchronously.
- Signal handlers can be called anytime when the program is running.
- Concurrency bug?
 - What if `main()` and `signal_handler()` access a common data?
 - A typical scenario in your shell lab
- Solution: Block Signals

Signals (contd..)

- **Blocking Signals**
 - Processes can choose to block signals using a signal mask
 - While a signal is blocked, the signal will be still delivered to the process but keep it pending
 - No action will be taken until the signal is unblocked
 - Implemented using `sigprocmask()`
- **Waiting for Signals**
 - Sometimes, a process needs to wait for a signal to be received.
 - Implemented using `sigsuspend()`

Blocking Signals – 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 SIGCHLD 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

Waiting for Signals – sigsuspend()

- `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

Signals – sigsetops

- A family of functions used to create and modify sets of signals. E.g.,
 - *int sigemptyset(sigset_t *set);*
 - *int sigfillset(sigset_t *set)*
 - *int sigaddset(sigset_t *set, int signum);*
 - *int sigdelset(sigset_t *set, int signum);*
- These sets can then 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

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

```
// sigchld handler installed
```

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

```
if (child_pid == 0){
```

```
    /* child comes here */
```

```
    execve(.....);
```

```
}
```

```
else{
```

```
    add_job(child_pid);
```

```
}
```

```
void sigchld_handler(int signum)
```

```
{
```

```
    int status;
```

```
    pid_t child_pid =
```

```
        waitpid(-1, &status, WNOHANG);
```

```
    if (WIFEXITED(status))
```

```
        remove_job(child_pid);
```

```
}
```

- Does add_job() or remove_job() come first?
- Where can signals be blocked to ensure correctness?

Agenda

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

Unix I/O

- All Unix I/O, from network sockets to text files, are based on one interface.
- A file descriptor is what's returned by `open()`.
`int fd = open("/path/to/file", O_RDONLY);`
- It's just an int, but you can think of it as a pointer into the file descriptor table.
- Every process starts with three file descriptors by default:
 - 0: STDIN
 - 1: STDOUT
 - 2: STDERR.
- Every process gets its own file descriptor table, but processes share the open file table and v-node table.

Unix I/O – dup2()

- Prototype:
int dup2(int oldfd, int newfd);
- *newfd* becomes a copy of *oldfd*;
- Read/write on *newfd* will access the file corresponding to *oldfd*.
- This is handy for implementing I/O redirection in shell lab.

Unix I/O – Practice Problem

```
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?

Agenda

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

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
- 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 forwarded 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
- SIGCHLD 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

Questions?