15-213 Recitation 8: 10/28/02

Outline

- Processes
- Signals
  - Racing Hazard
  - Reaping Children

Reminder

- L5 due: Halloween night, 11:59pm

Annie Luo

e-mail:

luluo@cs.cmu.edu

Office Hours:

Thursday 6:00 – 7:00

Wean 8402

How Programmers Play with Processes

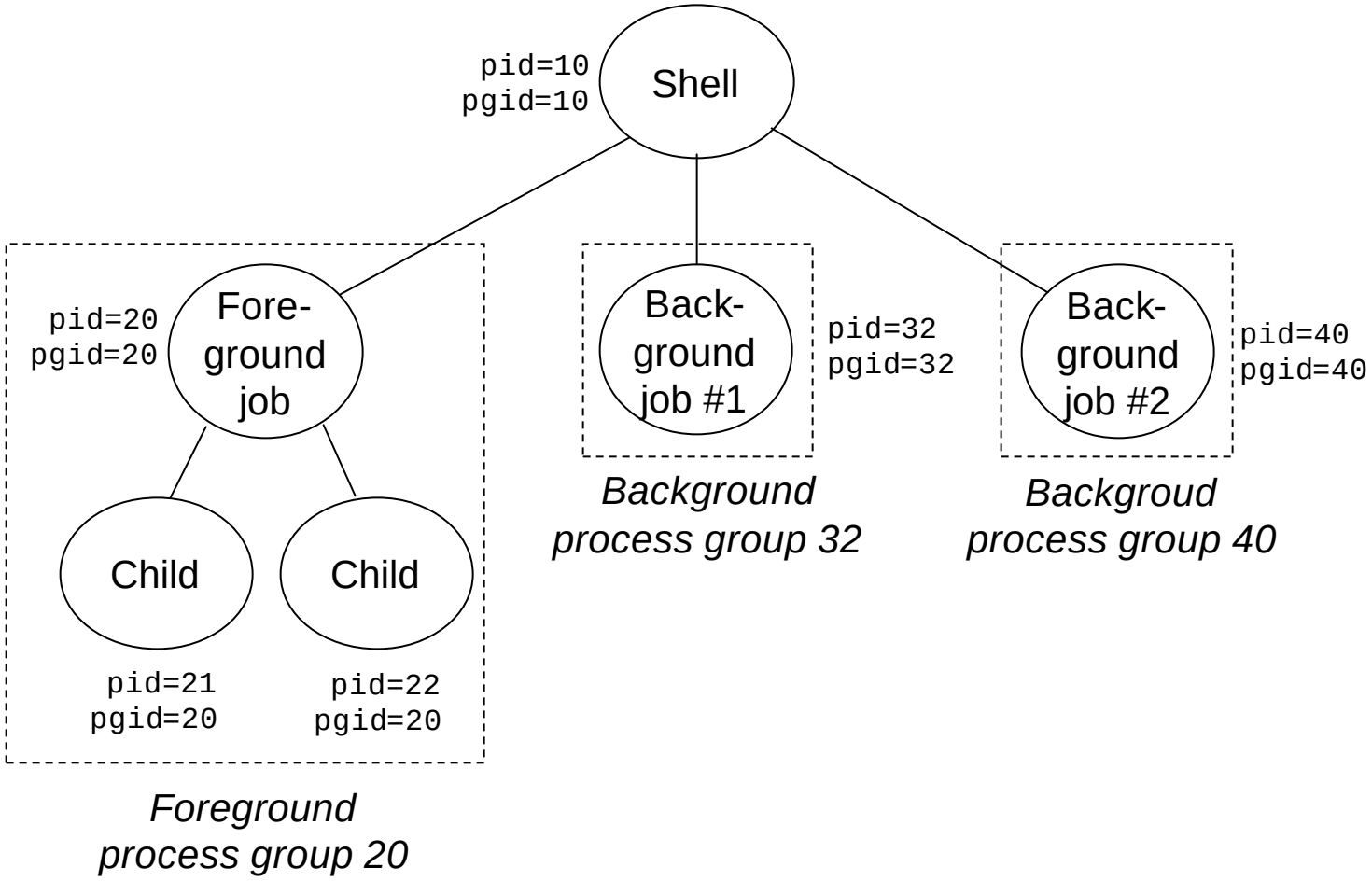
- Process: executing copy of program

- Basic functions
  - **`fork()`** spawns a new process
  - **`exit()`** terminates own process
  - **`wait()`** and **`waitpid()`** wait for and reap terminated children
  - **`execl()`** and **`execve()`** run a new program in an existing process
  - **`kill()`** send arbitrary signal to process or process group
    - **`kill(-pid, SIGINT)`** sends SIGINT to every process in pg “pid”

Process IDs and Process Groups

- Each process has its own, unique process ID
  - **`pid_t getpid();`**
- Each process belongs to exactly one process group
  - **`pid_t getpgid();`**
- Which process group does a new process belong to?
  - Its parent's process group
  - **`pid_t getppid();`**
- A process can make a process group for itself and its children
  - **`setpgid(pid, pgid);`**
  - Use **`setpgid(0,0)`** to put the child in a new pgroup other than the shell

Process Groups



Signals

- Section 8.5 in text book
  - Read at least twice, really!
- Used to reap background jobs
  - **`waitpid()`** alone is not enough
  - Background jobs become zombies
- A signal tells our process that some event has occurred in the system
- Can we use signal to count events?
  - No

Important Signals

- SIGINT
  - Interrupt signal from keyboard (ctrl-c), terminates the current foreground job
- SIGTSTP
  - Stop signal from keyboard (ctrl-z), suspends current job until next SIGCONT
- SIGCHLD
  - A child process has terminated (becomes a zombie) or stopped

Look at Figure 8.23 for a complete list of Linux signals

Signals – Sending

- A signal is sent through the kernel to a process
- When does the kernel send a signal?
  - The kernel detects a system event, e.g.:
    - Divide-by-zero (SIGFPE)
    - Termination of a child process (SIGCHLD)
  - Another process invokes a system call, e.g.:
    - **`kill(pid_t pid, int SIGINT)`**
    - **`alarm(unsigned int secs)`**

Signals – Receiving

- A process receives a signal when the kernel forces it to react to the signal

- Three default ways to react to a signal
  - The process ignores the signal
  - The process terminates (and dump core)
  - The process stops until next SIGCONT

- Can modify the default action (except SIGSTOP and SIGKILL) by executing signal handler functions
  - `signal(SIGINT, sigint_handler)`

Signal Handling Issues

- Subtle when deal with *multiple* signals

- Pending signals are *blocked* and *not queued*
  - Same type of signal currently being processed by handler
  - Not received until after the current handler returns
  - At most ONE pending signal of the same type at any time

- **`pending`** bit vector: bit *k* is set when signal type k is delivered, clear when signal received

- **`blocked`** bit vector: can be set by the program using the **`sigprocmask`** function

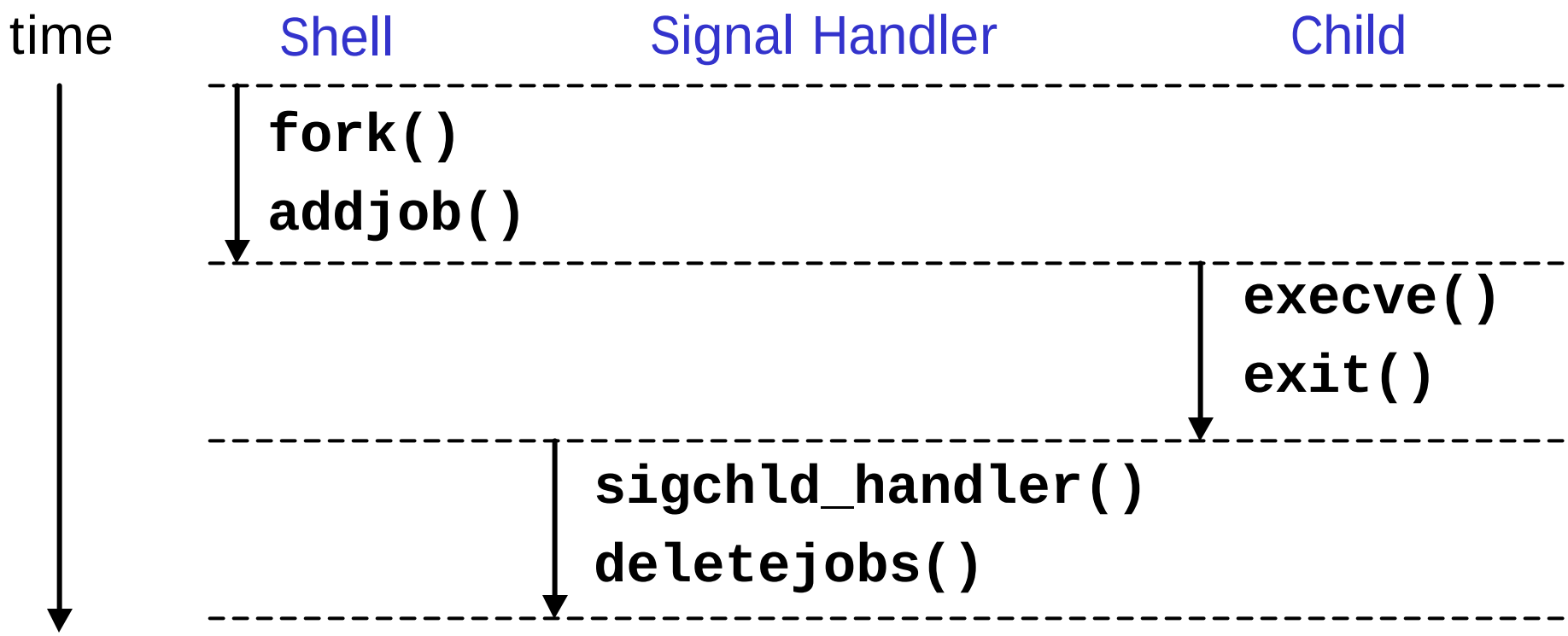
Synchronizing Parent and Children

- Preemptive scheduler run multiple programs “concurrently” by time slicing
  - How does time slicing work?
  - The scheduler can stop a program at any point
  - Signal handler code can run at any point, too
- Program behaviors depend on how the scheduler interleaves the execution of processes
- Racing condition between parent and child!
  - Why?

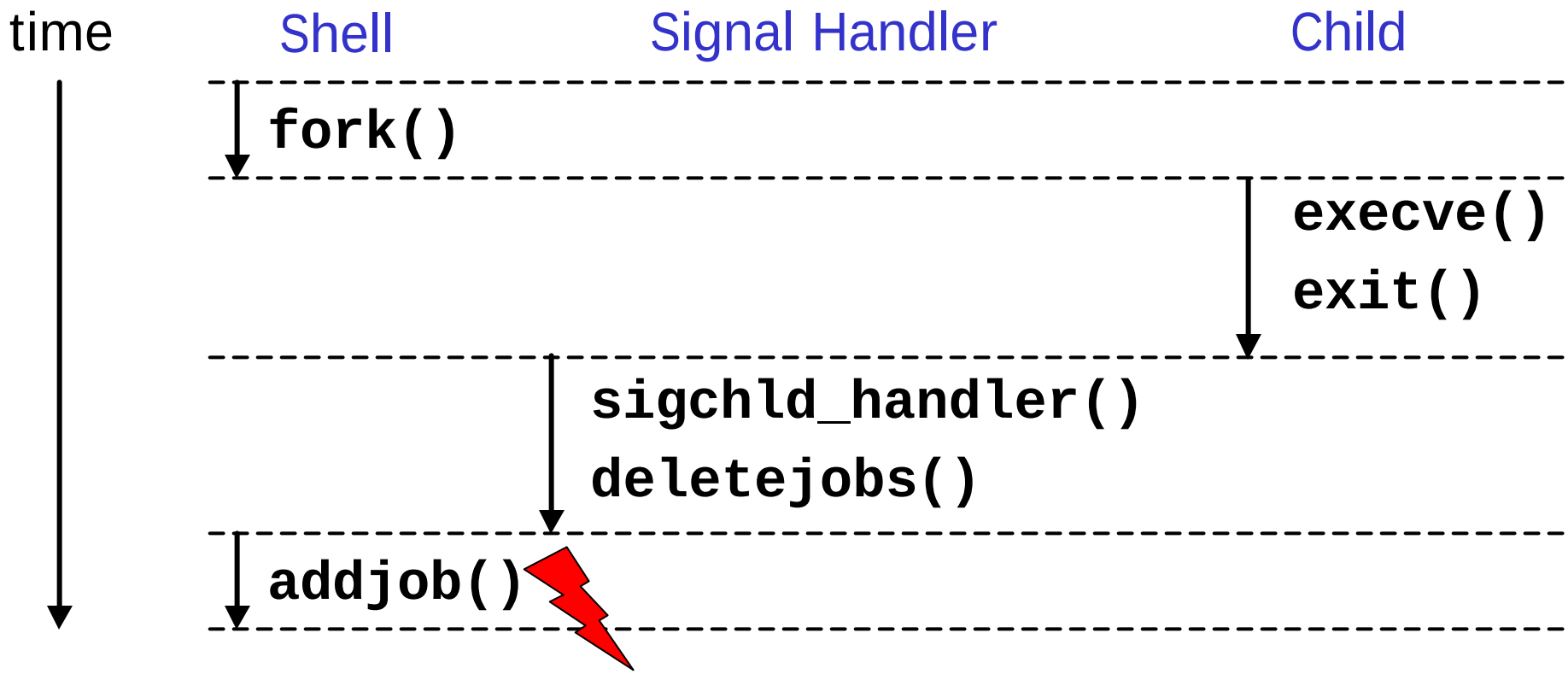
Parent & Children Racing Hazard

```
sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}

void eval() {
  pid = fork();

  if(pid == 0){ /* child */
    execve(…);
  }

  /* parent */
  /* signal handler might run BEFORE addjob() */
  addjob(…);
}
```

An Okay Schedule



A Problematic Schedule



Job added to job list *after* the signal handler tried to delete it!

Solution to P&C Racing Hazard

```
sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}

void eval() {
  sigprocmask(SIG_BLOCK, …);
  pid = fork();
  if(pid == 0){ /* child */
    sigprocmask(SIG_UNBLOCK, …);
    execve(…);
  }

  /* parent */
  /* signal handler might run BEFORE addjob() */
  addjob(…);
  sigprocmask(SIG_UNBLOCK, …);

}
```

More details see section 8.5.6 (p.633)

Waiting for Foreground Child Process

- What's the parent's behavior for foreground child?

- It is blocked:
  - In `eval()` the parent uses a busy loop checking foreground child pid
  - Parent pauses if fg child processes is still alive

Busy Loop: Pause vs. Sleep?

```
void eval() {
   … …
  /* parent */
  addjob(…);
  sigprocmask(SIG_UNBLOCK, …);
  while (fg process still alive)
    pause();
}

sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}
```

`pause()` causes the invoking process to sleep until a signal is received.

What's the problem here?

If signal is handled before pause is called, then it will not return when fg process sends SIGCHLD

Busy Loop: Pause vs. Sleep?

```
void eval() {
   … …
  /* parent */
  addjob(…);
  sigprocmask(SIG_UNBLOCK, …);
  while (fg process still alive)
    sleep(1);
}

sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}
```

Use **sleep** instead

Reaping Child Process

- Child process becomes zombie when terminates
  - Still consume system resources
  - Parent performs reaping on terminated child
  - Where to wait children processes to terminate?

- Suggested solution:
  - In **`sigchld_handler()`** use **`waitpid()`**, detecting any terminated or stopped jobs and reaping them

Waitpid

- Called in signal handler of SIGCHLD
  - Reaps all available zombie children
  - Does not wait for any other currently running children

- Waitpid in detail:

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

  `pid:` child process ID being waited to terminate

  `pid = -1:` wait for any child process

  `status`: tells why child terminated

  `options`:

  `WNOHANG`: return immediately if no children has exited (zombied)

  `WUNTRACED`: return for stopped children (status not reported)

Status in Waitpid

- **`int status;`**
  `waitpid(pid, &status, NULL)`
- Macros to evaluate status:
  - **`WIFEXITED(status):`** child exited normally
  - **`WEXITSTATUS(status):`** return code when child exits
  - **`WIFSIGNALED(status):`** child exited because of a signal not caught
  - **`WTERMSIG(status):`** gives the number of the term. signal
  - **`WIFSTOPPED(status):`** child is currently stopped
  - **`WSTOPSIG(status):`** gives the number of the stop signal

Summary

- Process provides applications with the illusions of:
  - Exclusively use of the processor and the main memory
- At the interface with OS, applications can:
  - Creating child processes
  - Run new programs
  - Catch signals from other processes
- Use **`man`** if anything is not clear!
- Coding style issues