Shell Lab: Exceptional Control Flow

CSE251, Spring 2019

Recitation 5: Wed, Apr 24th, 2019

Changmin Yi

ulistar93@unist.ac.kr

Reference : CMU 15-213: Intro to Computer Systems Fall 2015
Recitation 9 - Celeste Neary

Shell Lab

- Shell Lab is out!
- Due Thursday, May. 2, 11:59PM
  - **Please don't be late!**
- **The grader will run every 6am/noon/6pm on 4/24 – 5/2**
- **The final grader will run 00:05 5/3**

Shell Lab

```
Makefile
myint*
myint.c
myspin*
myspin.c
mysplit*
mysplit.c
mystop*
mystop.c
README.md
README.txt
sdriver.pl*
shlab.pdf
trace01.txt
trace02.txt
trace03.txt
trace04.txt
trace05.txt
trace06.txt
trace07.txt
trace08.txt
trace09.txt
trace10.txt
trace11.txt
trace12.txt
trace13.txt
trace14.txt
trace15.txt
trace16.txt
tsh*
tsh.c
tshref*
tshref.out
```

- There are a lot of files, but there is an only one file you should implement.
  tsh.c

- You will implement…

  - `eval`: Main routine that parses and interprets the command line. [70 lines]
  - `builtin_cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`. [25 lines]
  - `do_bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
  - `waitfg`: Waits for a foreground job to complete. [20 lines]
  - `sigchld_handler`: Catches SIGCHILD signals. 80 lines]
  - `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [15 lines]
  - `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [15 lines]

Shell Lab Testing

- You can run your shell by
  - ./tsh
    (you should 'make' first whatever you run)
  - tsh is a shell as like Unix shell

Unix

```
[cs20141494@uni06 shlab]$ /bin/ps
  PID TTY          TIME CMD
 6986 pts/35   00:00:00 ps
16199 pts/35   00:00:00 bash
```

tsh

```
[cs20141494@uni06 shlab]$ ./tsh
tsh> /bin/ps
  PID TTY          TIME CMD
 6987 pts/35   00:00:00 tsh
 6988 pts/35   00:00:00 ps
16199 pts/35   00:00:00 bash
tsh>
tsh> [cs20141494@uni06 shlab]$
```

run ./tsh

exit tsh
"ctrl + d"
= "^d"

Shell Lab Testing

- You can run your shell by
  - ./tsh
- The possible commands are
  - /bin/ls
  - /bin/ps
  - /bin/echo
  - jobs
  - bg
  - fg
  - kill
  - quit
- Please do not try in tsh>
  - vi, emacs, more, less

Shell Lab Testing

- And you can test your shell by
  - sdriver.pl

```
unix> ./sdriver.pl -h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
Options:
  -h            Print this message
  -v            Be more verbose
  -t <trace>    Trace file
  -s <shell>    Shell program to test
  -a <args>     Shell arguments
  -g            Generate output for autograder
```

  - For example

```
unix> ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

  or

```
unix> make test01
```

  (both are same)

Shell Lab Testing

- And also there is reference answer shell !
  - So your ./tsh should be work same as ./tshref
- The reference answer is
  - tshref.out

```
make[1]: Entering directory `/afs/cs.cmu.edu/project/ics/im/labs/shlab/src'
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (26252) ./myspin 1 &
```

Shell Lab

- While doing the implementation,
- You need to consider the hazards that already told in lecture
  - Race conditions
    - Hard to debug so start early (and think carefully)
  - Reaping zombies
    - Race conditions
    - Handling signals correctly
  - Waiting for foreground job
    - Think carefully about what the right way to do this is
- **Hints** in the instructions will be helpful when you work I recommend reading it carefully

Processes

- Four basic States
  - Running
    - Executing instructions on the CPU
    - Number bounded by number of CPU cores
  - Runnable
    - Waiting to be running
  - Blocked
    - Waiting for an event, maybe input from STDIN
    - Not runnable
  - Zombie
    - Terminated, not yet reaped

Processes

- Four basic process control function families:
  - fork()
  - exec()
    - And other variants such as execve()
  - exit()
  - wait()
    - And variants like waitpid()
- Standard on all UNIX-based systems

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts (asynchronous)
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as Ctrl-C (SIGINT), divide-by-zero (SIGFPE), or the termination of a child process (SIGCHLD)
  - Another program called the `kill()` function
  - The user used a `kill` utility

Signals

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal

- Receiving a signal is non-queuing
  - There is only one bit in the context per signal
  - Receiving 1 or 300 SIGINTs looks the same to the process
- Signals are received at a context switch
- Three possible ways to react:
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt

Signals

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal

- Blocking signals
  - Sometimes code needs to run through a section that can't be interrupted
  - Implemented with `sigprocmask()`
- Waiting for signals
  - Sometimes, we want to pause execution until we get a specific signal
  - Implemented with `sigsuspend()`
    -> What different with `wait()` and `waitpid()`?
- Can't modify behavior of SIGKILL and SIGSTOP

Signals

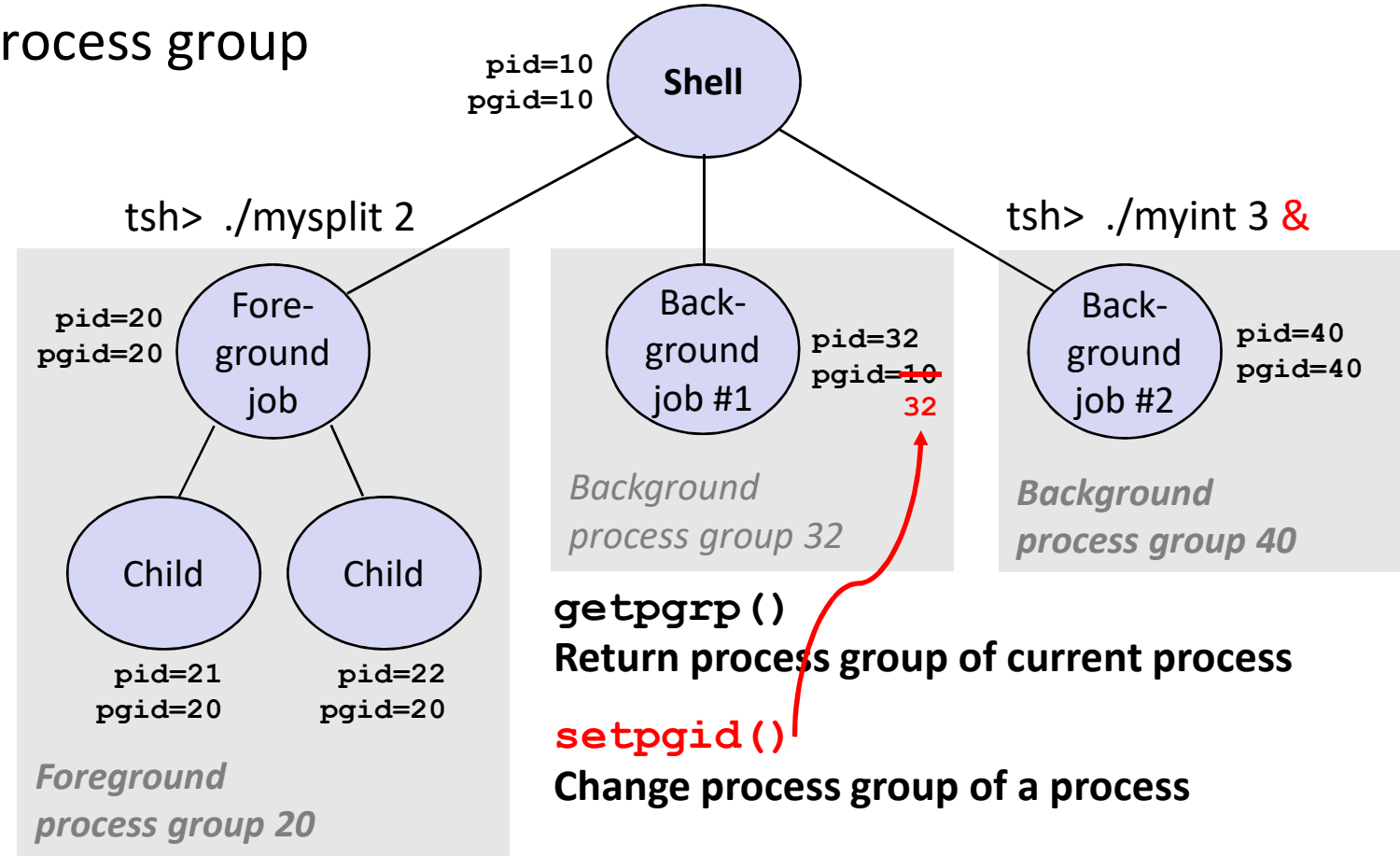
- Signal handlers
  - Can be installed to run when a signal is received
  - The form is `void handler(int signum){ … }`
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired

Signals

- `int sigsuspend(const sigset_t *mask)`
  - Can't use wait() twice – use sigsuspend!
  - Temporarily replaces the signal mask of the calling process with the mask given
  - Suspends the process until delivery of a signal whose action is to invoke a signal handler or terminate a process
  - Returns if the signal is caught
    - Signal mask restored to the previous state
  - Use sigaddset(), sigemptyset(), etc. to create the mask

Signal Examples

- Every process belongs to exactly one process group
- Process groups can be used to distribute signals easily
- A forked process becomes a member of the parent's process group



Signal Examples

```
// sigchld handler installed

pid_t child_pid = fork();

if (child_pid == 0){
   /* child comes here */

   execve(……);
}
else{

   add_job(child_pid);

}
```

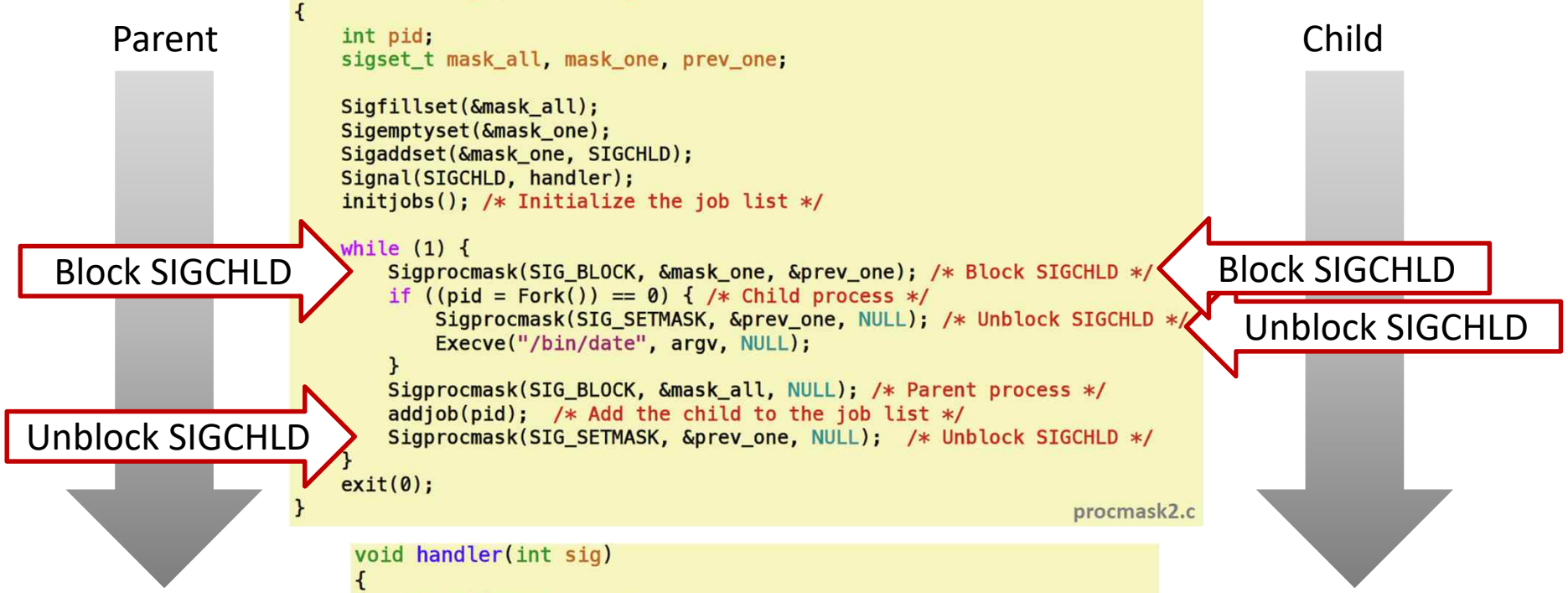
```
void sigchld_handler(int signum)
{
    int status;

    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 we block signals in this code to guarantee correct execution?

Corrected Shell Program without Race




```c
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

procmask2.c

```c
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

Async-Signal-Safety

- **Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.**
- **Posix guarantees 117 functions to be async-signal-safe**
  - Source: "`man 7 signal`"
  - Popular functions on the list:
    - `_exit, write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** on the list:
    - `printf, sprintf, malloc, exit`
    - Unfortunate fact: `write` is the only async-signal-safe output function

Safely Generating Formatted Output

- Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v) /* Put long */`
  - `void sio_error(char s[]) /* Put msg & exit */`

```c
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    Sio_puts("So you think you can stop the bomb with ctrl-
c, do you?\n");
    sleep(2);
    Sio_puts("Well...");
    sleep(1);
    Sio_puts("OK. :-)\n");
    _exit(0);
}
```

sigintsafe.c