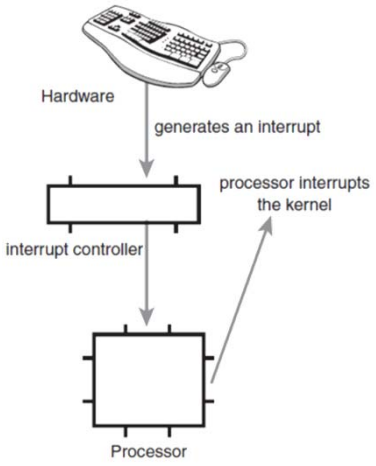
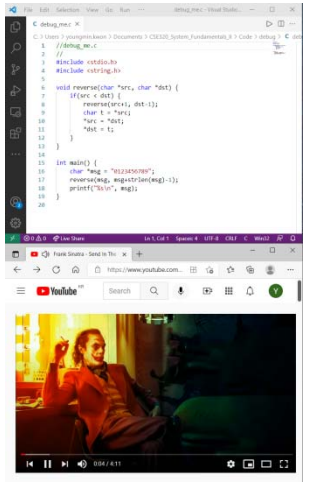
CSE320 System Fundamentals II
Exceptional Control Flow

YoungMin Kwon

Exceptional Control Flow

- Exceptional Control Flow (ECF)
  - Exceptions: Interrupt, Trap, Fault, Abort
    - Hardware timer goes off or a network packet arrives
  - Context switch: OS transfers control from one process to another
  - A process sends a signal to another process (next lecture)

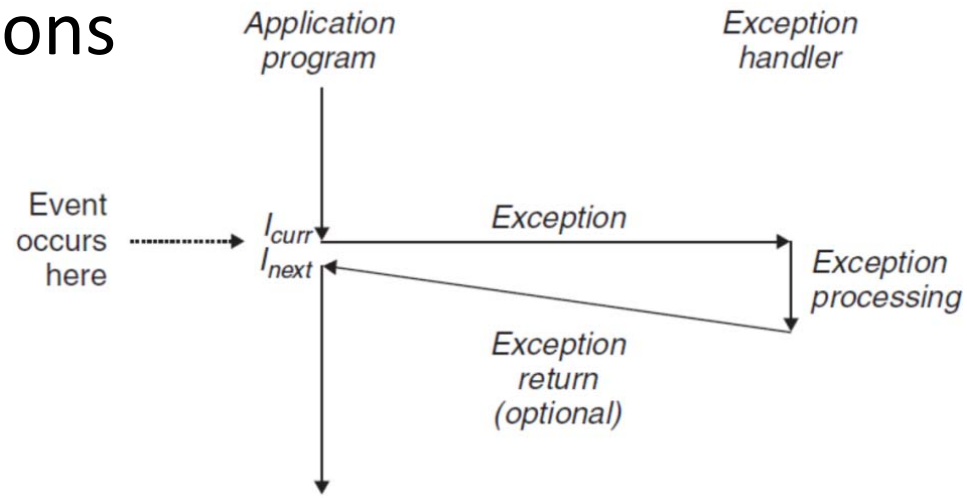




Exceptions

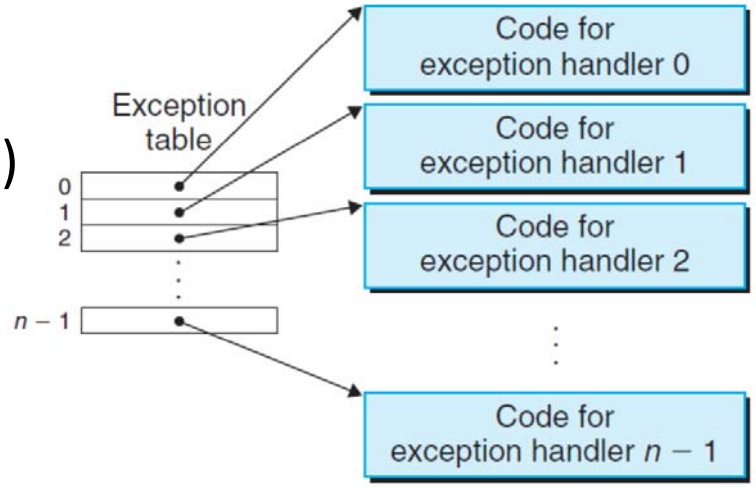
- Exception:
  - Change in the control flow due to processor's state change (event)
  - Virtual memory page fault, division by zero, I/O complete operation

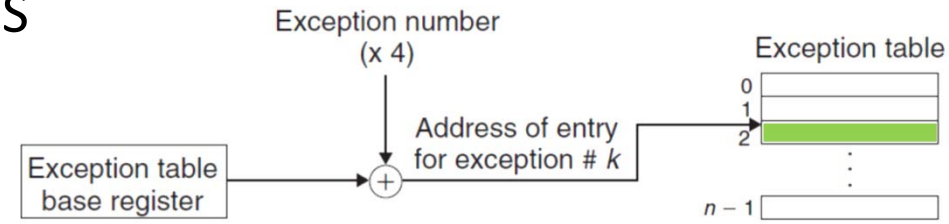
- After handling exceptions
  - Return to I_{curr}
  - Return to I_{next}
  - Abort the program



Exception Handling

- Exception number
  - Set by HW designer (division by 0)
  - Set by Kernel designer (system call)
- Exception table
  - When an event k occurred, the flow jumps to the k^{th} entry in the exception table
  - At system boot time, the OS initializes the jump table





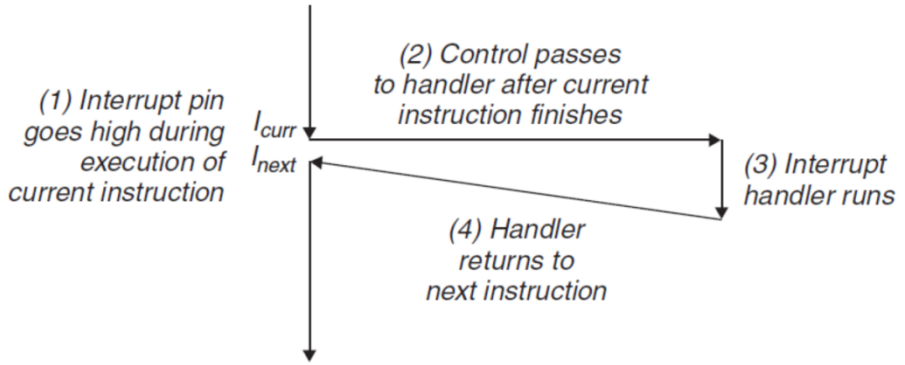
Exception Handling

- Mode switches from user mode to kernel mode if the event occurred in the user mode.
- Return address (current instruction or next instruction) and EFLAGS register are pushed onto the kernel's stack
- After the handler has processed the event the control returns to the interrupted program
- Switch mode back to the user mode if necessary

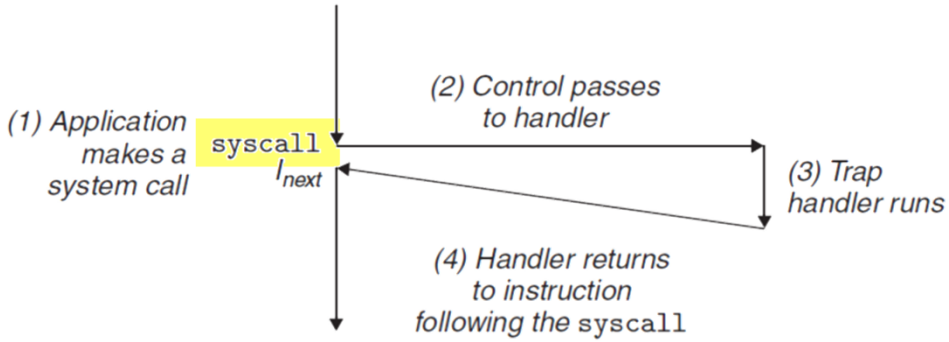
Classes of Exceptions

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Interrupt:

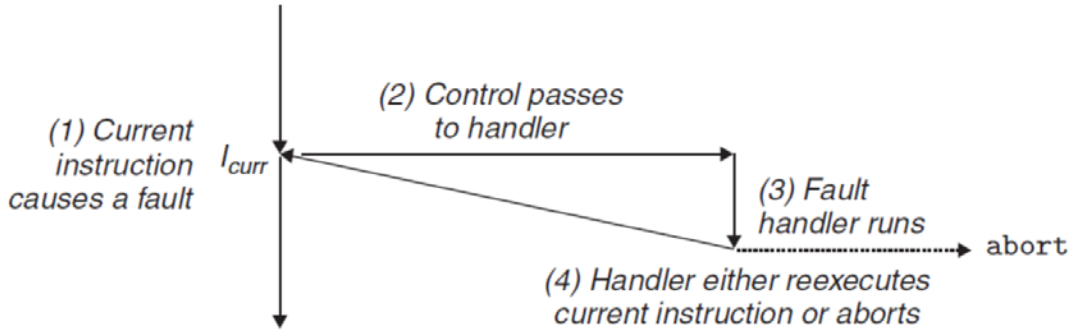


Trap:

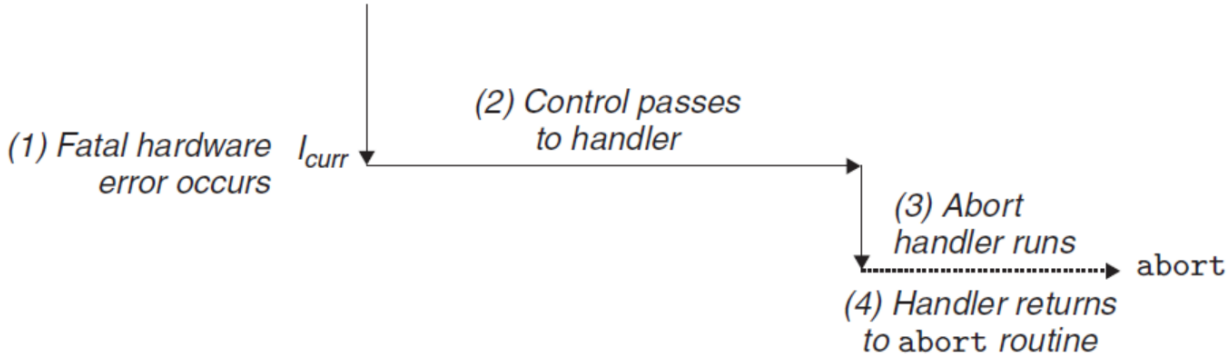


Classes of Exceptions

Fault:

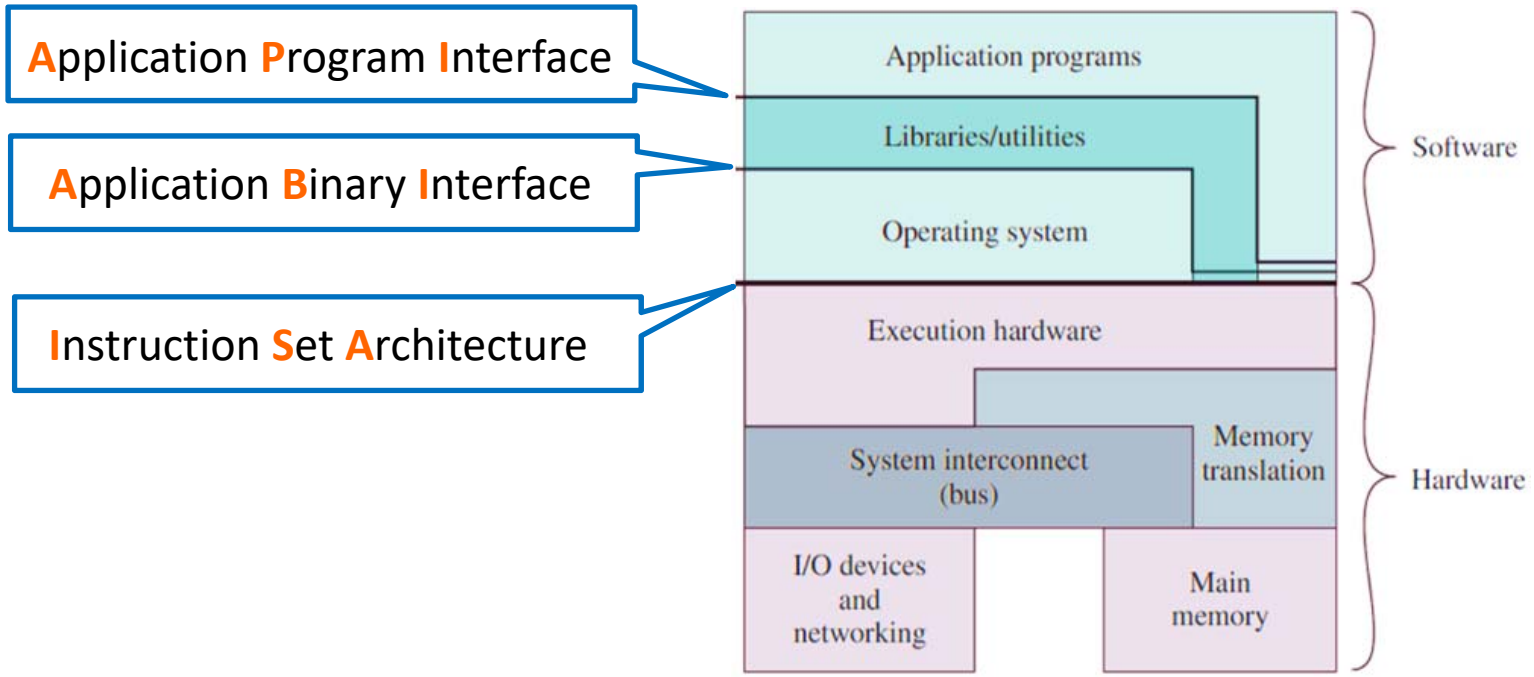


Abort:



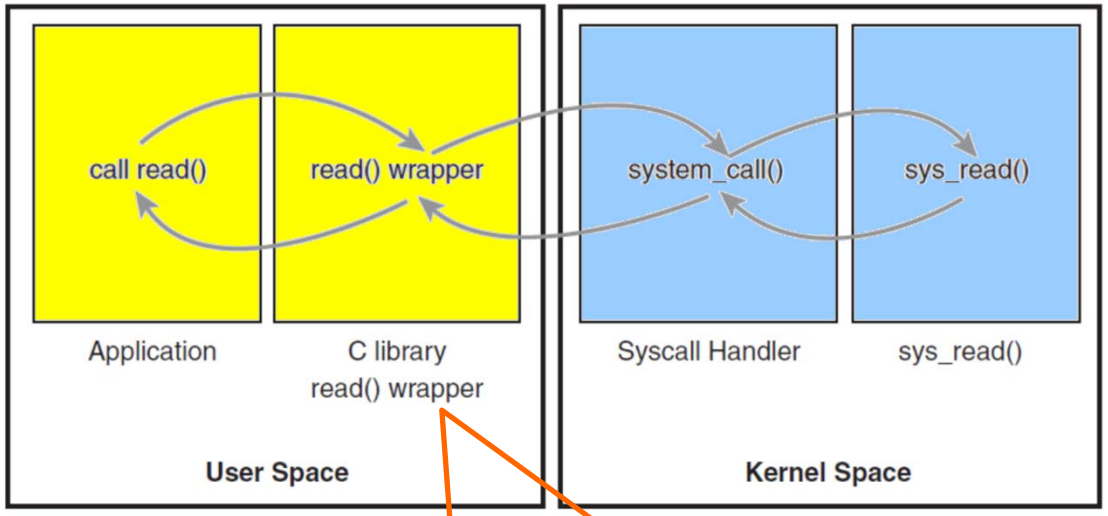
System Call

- Computer Hardware and Software Structure



System Call

- Kernel takes control of the CPU in two ways
  - Interrupt and System call
- Control flow of a system call



Invoking the system call handler and executing a system call.

POSIX provides a standard set of APIs despite the differences in system calls

System Call

- syscall:
  - System calls are provided by a trapping instruction called syscall
  - syscall number is in %rax
  - up to 6 arguments are in %rdi, %rsi, %rdx, %r10, %r8, and %r9
  - return value is in %rax; %rcx and %r11 are destroyed.

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

System Call Example

```c
#include <unistd.h>
int main() {
    write(1, "hello, world\n", 13);
    _exit(0);
    return 0;
}
```

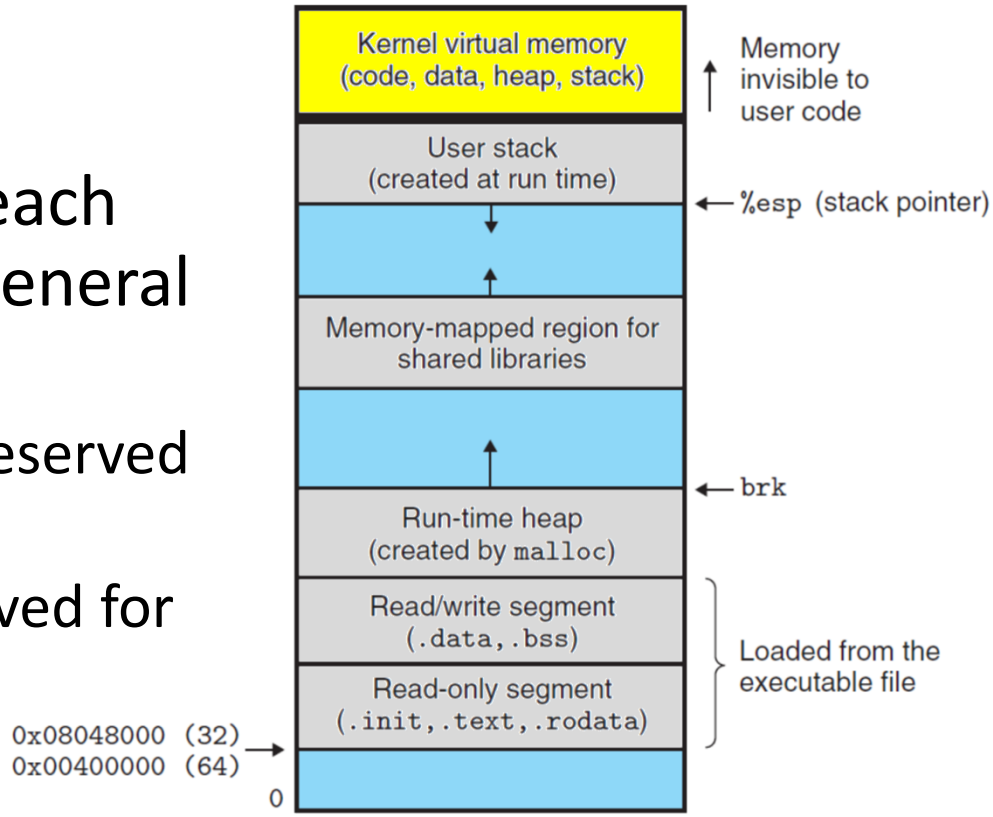
```
    .section    .rodata
.LC0:
    .string "hello, world\n"
    .text
    .globl  main
main:
    pushq   %rbp
    movq    %rsp, %rbp

    #write(1, "hello world\n", 13)
    movl    $13, %edx           #3rd param
    leaq    .LC0(%rip), %rsi    #2nd param
    movl    $1, %edi            #1st param
    movq    $1, %rax     #1: write syscall
    syscall

    #_exit(0)
    movl    $0, %edi            #1st param
    movq    $60, %rax    #60: exit syscall
    syscall

    movl    $0, %eax
    popq    %rbp
    ret
```

Processes

- Process
  - An instance of a program in execution
- Each program runs in the context of a process
  - A context includes:
    - The program's code,
    - Data in memory, stack, registers,
    - Open file descriptors…

Private Address Space

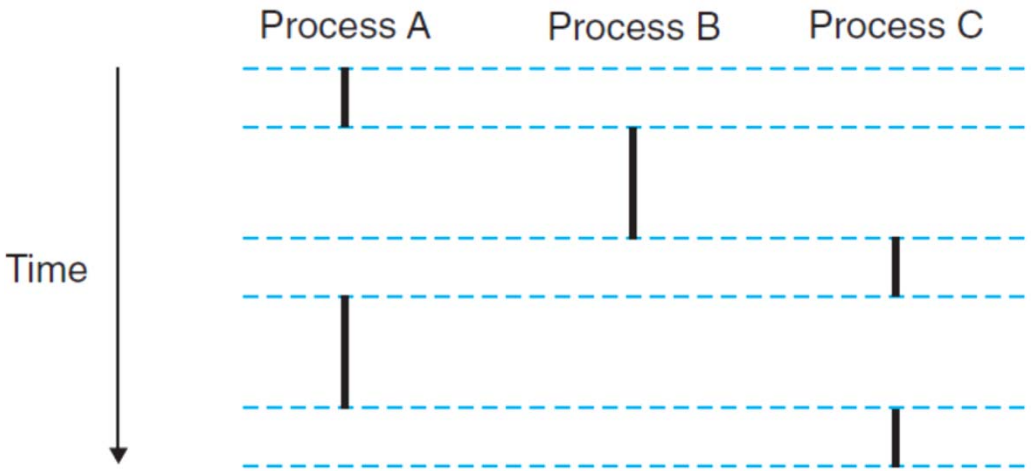
- A process provides each program with its own private address space
- The address space for each process has the same general organization
  - The bottom portion is reserved for the user program
  - The top portion is reserved for the kernel

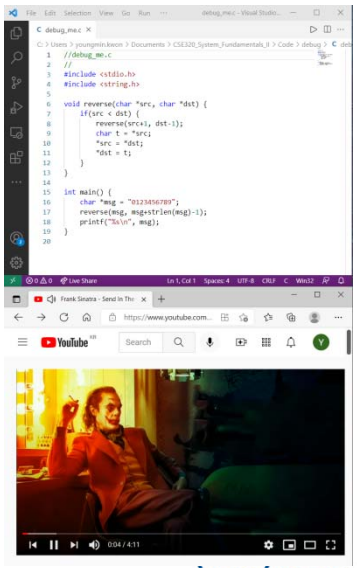


Logical Control Flow

- A process provides each program with an illusion that it has exclusive use of the processor although many other programs are running concurrently

- Each vertical bar in the figure below represents a portion of the logical control flow of a process





Concurrent Flows

- Concurrent Flow:
  - A logical flow whose execution overlaps in time with other flows
    - A and B run concurrently; A and C run concurrently.
    - B and C do not run concurrently
- Parallel flow:
  - If two flows are running concurrently on different processor cores or computers

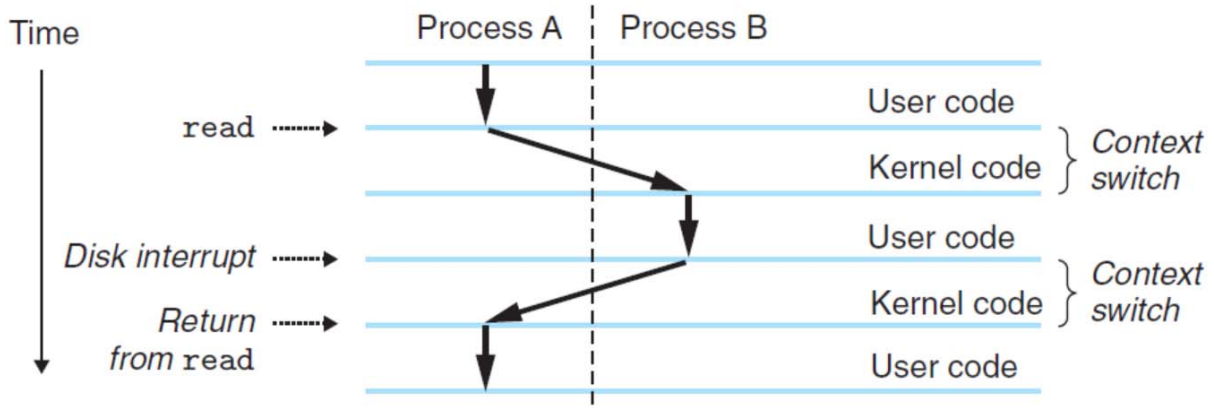
Context Switches

- Kernel maintains a context for each process

- A context consists of
  - registers,
  - PC (program counter, %rip),
  - user stack,
  - kernel stack,
  - kernel data structure (page table, process table, file table)

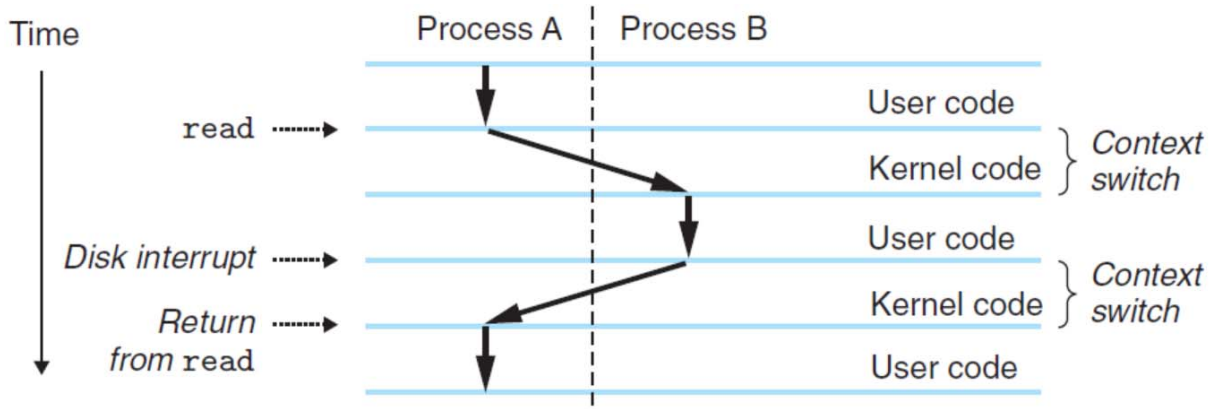
Context Switches

- Context switch
  - Save the context of the current process
  - Restore the saved context of the previously preempted process
  - Pass control to the newly restored process



Context Switches

- Context switch
  - Kernel can decide to preempt the current running process
  - Scheduler picks the next process to run
  - Kernel transfers the control to the new process through context switching



User mode and Kernel mode

- Processors typically provide a mode bit
- Kernel mode
  - When the mode bit is set
  - Can execute any instructions and can access any memory location
- User mode
  - When the mode bit is not set
  - Cannot execute privileged instructions: halt the processor, change the mode bit, I/O operation
  - Cannot reference kernel code or data

User mode and Kernel mode

- Some kernel data structures are exposed to user mode processes
  - /proc file system
  - /proc/cpuinfo, /proc/<pid>/maps, …

```
$ cat /proc/5558/maps
00400000-004e1000 r-xp 00000000 08:01 7209108            /bin/bash
006e0000-006e1000 r--p 000e0000 08:01 7209108            /bin/bash
006e1000-006ea000 rw-p 000e1000 08:01 7209108            /bin/bash
006ea000-006f0000 rw-p 00000000 00:00 0
00753000-00f8a000 rw-p 00000000 00:00 0                  [heap]
7f240fa46000-7f240fa51000 r-xp 00000000 08:01 7867530    /lib/x86_64-linux-gnu/libnss_files-2.15.so
7f240fa51000-7f240fc50000 ---p 0000b000 08:01 7867530    /lib/x86_64-linux-gnu/libnss_files-2.15.so
7f240fe5c000-7f240fe5d000 r--p 0000a000 08:01 7867532    /lib/x86_64-linux-gnu/libnss_nis-2.15.so
7f2410c90000-7f2410c92000 rw-p 00023000 08:01 7867529    /lib/x86_64-linux-gnu/ld-2.15.so
...
7fffcac50000-7fffcac71000 rw-p 00000000 00:00 0          [stack]
7fffcad4f000-7fffcad50000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
$
```

Process Control

- Obtaining Process IDs

```
pid_t getpid(void);  //get the process id
pid_t getppid(void); //get the parent process id
```

- Creating a process

```
pid_t fork(void);
```

- Terminating a process

```
void exit(int status);
```

```
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if(pid == 0)
        printf("child:  pid: %d, "
               "ppid: %d\n",
               getpid(), getppid());
    else
        printf("parent: pid: %d, "
               "ppid: %d, child: %d\n",
               getpid(), getppid(), pid);
    exit(0);
}
```

Process Control

- When a process terminates
  - The kernel does not remove it from the system immediately
  - The process is kept around in a terminated state (zombie process) until it is reaped by its parent

- Waits for its child to terminate and reap it

```
pid_t waitpid(pid_t pid, int *statusp, int options)
pid_t wait(int *statusp); // wait for any children
                          // to terminate
```

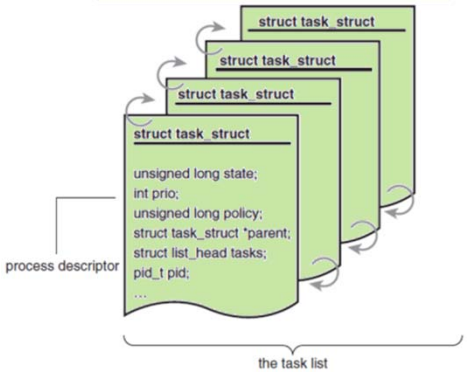
Process Control

- Putting process to sleep

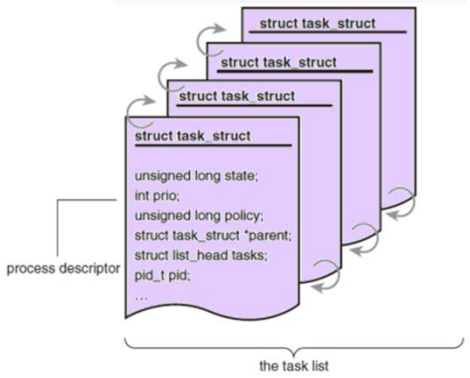
```
//sleep for sec seconds or until signaled
unsigned int sleep(unsigned int sec);

//sleep until signaled
int pause(void);
```

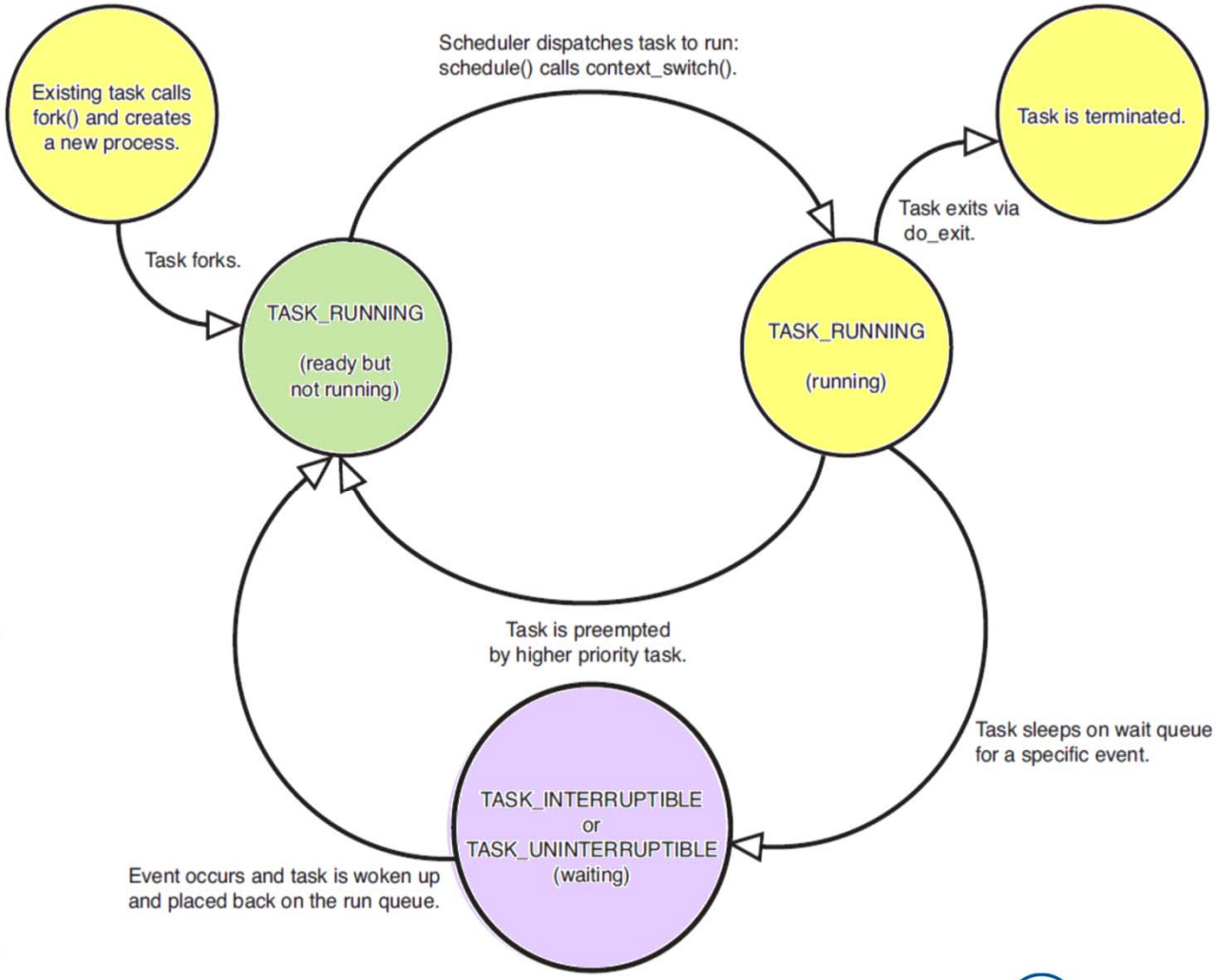
Process State

Ready Queue



Blocked Queue





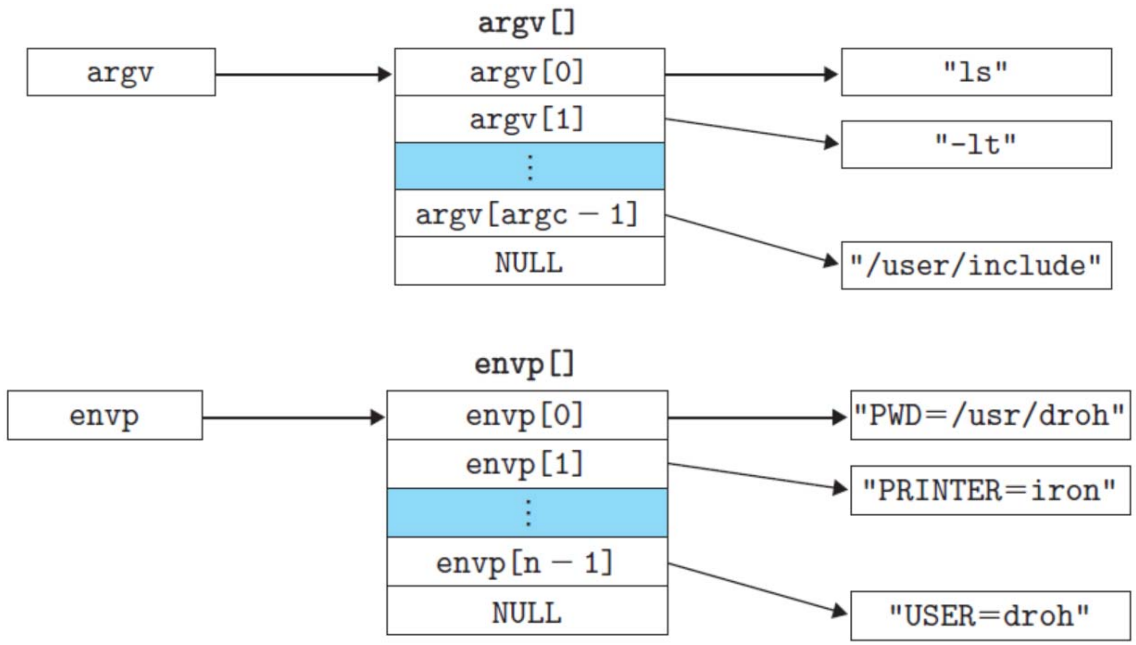
fork and waitpid

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    int status, i;
    pid_t pid;
    for(i = 0; i < 10; i++)
        if((pid = fork()) == 0)
            exit(100+i);

    while((pid = waitpid(-1, &status, 0)) > 0) {
        if(WIFEXITED(status))
            printf("pid: %d, status: %d\n", pid, WEXITSTATUS(status));
        else
            printf("pid: %d, abnormal termination\n", pid);
    }

    exit(0);
}
```

Process Control

- execve loads and runs a new program in the context of the current process

```
int execve(const char *filename,
           const char *argv[],
           const char *envp[]);
```



execve Example

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

int main(int argc, char **argv, char **envp) {
    char *args[100] = {//ls -a -l
        "/bin/ls", "-a", "-l", NULL
    };

    if (argc > 1) {
        int i;
        for (i = 0; i < argc; i++)
            printf("%d: %s\n", i, argv[i]);

        for (i = 1; i < argc; i++)
            args[i - 1] = argv[i];
        args[i - 1] = NULL;
    }

    execve(args[0], args, NULL/*envp*/);
}
```

```
> ./a.out /usr/bin/cal -m 12
0: ./a.out
1: /usr/bin/cal
2: -m
3: 12
   December 2020
Su Mo Tu We Th Fr Sa
        1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

A Simple Shell Program

```
#include "csapp.h"
#define MAXARGS   128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* Command line */

    while (1) {
        /* Read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* Evaluate */
        eval(cmdline);
    }
}
```

```c
/* eval - Evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;   /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* Child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
```

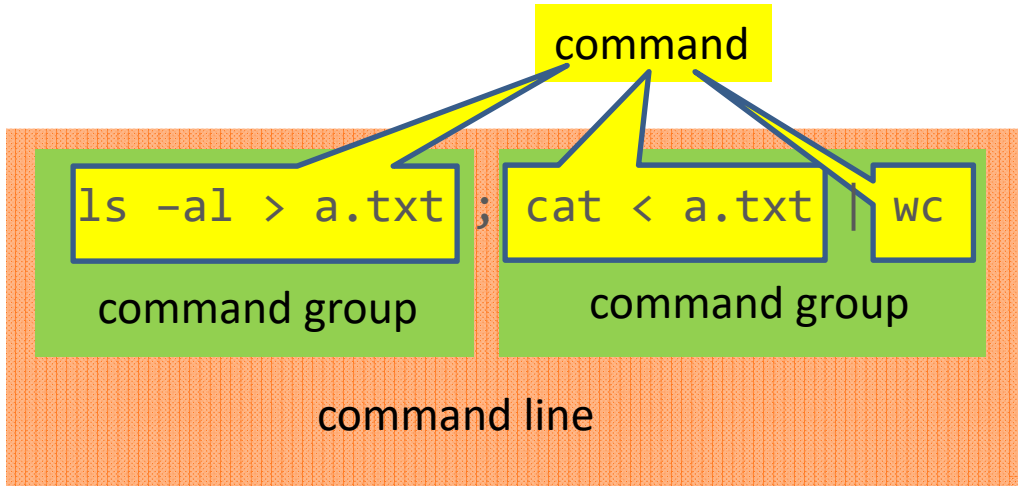
```
        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* If first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* Ignore singleton & */
        return 1;
    return 0;                     /* Not a builtin command */
}
```

```
/* parseline - Parse the command line and build the argv array */
int parseline(char *buf, char **argv)
{
    char *delim;         /* Points to first space delimiter */
    int argc;            /* Number of args */
    int bg;              /* Background job? */

    buf[strlen(buf)-1] = ' ';  /* Replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* Ignore leading spaces */
        buf++;

    /* Build the argv list */
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* Ignore spaces */
               buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)  /* Ignore blank line */
        return 1;

    /* Should the job run in the background? */
    if ((bg = (*argv[argc-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

Programming Assignment 7

- In this assignment, we will implement a shell program
  - With redirection and pipe
- Download shell.zip and finish implementing all TODO lines
- Test some commands like
  - `cat < shell.c | wc`
  - `ls –al > a.txt ; cat < a.txt | wc`
  - `ps -a | wc & ps –a | wc`
  - `ps -a | wc ; ps –a | wc`
- Due date: TBD

```c
//shell.c
//
int main(int argc, char **argv) {
    while(1) {
        //print the prompt
        prompt();
        fflush(stdout);

        //read a command line
        char cmdline[MAXLINE];
        fgets(cmdline, sizeof(cmdline), stdin);
        if(feof(stdin))
            exit(0);

        //execute the command line
        execute_cmd_line(cmdline);
    }
}
```



```
//command_line.c
//
void execute_cmd_line(char *cmdline) {
    //tokenize cmdline
    token_buf_t *tb = tokenize(cmdline);

    for(char *token = ""; token[0] != EOL; ) {
        //make a command group
        cmd_grp_t *cgrp = make_cmd_grp();

        //parse command group
        parse_command_group(tb, cgrp);

        //check the delimiter
        token = tb->read_token(tb);
        ON_FALSE_EXIT(token[0] == EOL || token[0] == ';' || token[0] == '&',
                      "delimiter should be EOL, ;, or &\n");

        //execute the command list
        cgrp->execute(cgrp, token[0] == '&'/*bg_proc*/);

        //destroy command group
        cgrp->destroy(cgrp);
    }

    tb->destroy(tb);
}
```

```
//execute a command group
static void execute_cmd_grp(cmd_grp_t *self, int bg_proc) {
    command_t **cmds = self->cmds;
    int nr_cmd = self->nr_cmd;

    //run commands
    for(int i = 0; i < nr_cmd; i++) {
        //empty command
        if(cmds[i]->argc == 0)
            continue;

        //TODO: if cmds[i] is an internal command, call its execute_internal
        //      otherwise, call its execute
    }

    //wait for child processes to finish
    for(int i = 0; i < nr_cmd; i++) {
        //internal or empty commands
        if(cmds[i]->pid == -1)
            continue;

        //TODO: if bg_proc, print the command and the pid
        //                  of the child proc running it
        //      else reap the child process
    }

    //TODO: for all terminated background child processes,
    //      reap them and print their pids.
    //hint: use waitpid with WNOHANG option
}
```

```
typedef struct command {
    //the command and its arguments
    char *argv[MAXARGS];    //command and arguments
    int argc;               //# of valid argv

    //file descriptors for stdin and stdout
    int fdin;               //file descriptor for input
    int fdout;              //file descriptor for output

    //pid of the process that runs this command
    pid_t pid;
…
} command_t;


//execute a command
static void execute_cmd(command_t *self) {
    ON_FALSE_EXIT(self->argc > 0, "empty command");
    //TODO: fork and execute self in the child process
    //child process
    //   TODO: if fdin/fdout are set for the redirection or for the pipe,
    //         dupe them to STDIN_FILENO/STDOUT_FILENO and close them
    //   TODO: execute self
    //   hint: use execvp
    //parent process
    //   TODO: if fdin/fdout are set for the redirection or for the pipe,
    //         close them
    //   TODO: copy the pid of the child process to self->pid
}
```

```c
//connect prev and curr using pipe
static void set_pipe(command_t *self, command_t *next) {
    //check if self's fdout and next's fdin are already set
    ON_FALSE_EXIT(self->fdout == STDOUT_FILENO, "fdout is already set");
    ON_FALSE_EXIT(next->fdin  == STDIN_FILENO,  "fdin is already set");

    //prepare fdin and fdout of self and next commands
    int fd_pipe[2];  //for pipe
    //TODO: set fd_pipe using pipe
    //TODO: set fdin and fdout of self and next commands
}

//set fdout of cmd
static void set_redir_out(command_t *self, char *fname) {
    //check if the self's fdout is already set
    ON_FALSE_EXIT(self->fdout == STDOUT_FILENO, "fdout is already set");

    //TODO: open fname for redirection
    //hint: use mode for the mode
    mode_t mode = S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH;

    //prepare fdout of the command
    //TODO: set fdout of cmd
}
```

```c
//set fdin of cmd
static void set_redir_in(command_t *self, char *fname) {
    //check if the command's fdin is already set
    ON_FALSE_EXIT(self->fdin == STDIN_FILENO, "fdin is already set");

    //TODO: open fname for redirection

    //prepare fdin of the command
    //TODO: set fdin of cmd
}
```

Expected Result

```
$ ./a.out
/home/ubuntu/home/cse320/shell> cat < shell.c | wc
     29      61     611
/home/ubuntu/home/cse320/shell> ls -al > a.txt; cat < a.txt | wc
     20     173    1067
/home/ubuntu/home/cse320/shell> ps -a | wc & ps -a | wc
background process ps, pid: 280393
background process wc, pid: 280394
      6      24     178
/home/ubuntu/home/cse320/shell>       4      16     120

background processs terminated, pid: 280393
background processs terminated, pid: 280394
/home/ubuntu/home/cse320/shell> ps -a | wc ; ps -a | wc
      4      16     120
      4      16     120
/home/ubuntu/home/cse320/shell> cd ..
/home/ubuntu/home/cse320> exit
$
```