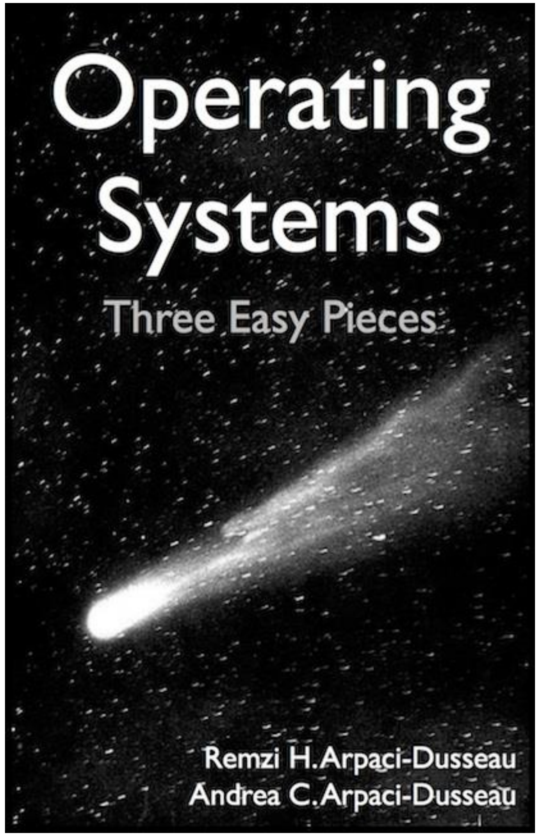
CMSC 125: Operating Systems

- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org

Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

5. Interlude: Process API

Operating System: Three Easy Pieces

Process creation and control in Unix

- Use of `fork()` and `exec()` system calls
- A process can wait for a process it has created using `wait()` system call

The fork() system call

- Create a new process
  - The newly-created process has its own private copy of the **address space**, **registers**, and **PC**.

p1.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

Calling fork() example (Cont.)

Result (Not deterministic)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

- The “original” p1 process prints its PID which is 29146 – the parent process
- Then a call to `fork()` is made which creates a new process with PID 29417 that is (almost) similar to the parent process – the child process
- Notice that the child process began execution after the call to `fork()` instead of at the start of `main()`
- The child process has its own private **address space**, **registers**, and **PC** which is different from parent process
- `fork()` returns **zero(0)** to the child process and the **child PID** to the parent process
- The order of execution of the parent and child processes is **non-deterministic**, dependent on the **scheduler**

The wait() system call

- Allows the parent process to <u>wait</u> for the child process to finish first before continuing/terminating
- This system call won't return until the child has run and exited

p2.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

The wait() system call (Cont.)

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

- Adding the call to `wait()` on the code block for the parent process guarantees that the child process will finish first before the parent process – thus **deterministic** order of execution
- Even if right after the call to `fork()` the parent process is selected by the scheduler, it will not proceed with its execution because of the `wait()` call which will not return until the child process has finished

The exec() system call

- Run a program that is different from the calling program
- The child process will have different **code**, **static data**, **address space**, **registers**, **PC**
  - In the previous examples, **code** and **static data** of the child process are the same as the parent process after a call to `fork()`

p3.c

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

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {                    // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {            // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");         // program: "wc" (word count)
        myargs[1] = strdup("p3.c");       // argument: file to count
        myargs[2] = NULL;                 // marks end of array
        …
```

The exec() system call (Cont.)

p3.c (Cont.)

```
…
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {                          // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

- `exec()` **loads** the code and static data, given the name of the executable (ex. `wc`), into the child process' **address space** which originally contains the code and static data of the parent process copied during the call to `fork()`
- A successful call to `exec()` never returns – why?

Why separate fork() and exec()?

- Essential for building a **shell** – an "interface" program that accepts commands from users to access the services of the OS
  - Ex. BASH and ZSH in Linux, Command Prompt and Powershell in Windows
- With this separation, the shell can run some code *after* the call to `fork()` but *before* the call to `exec()`
  - It allows the shell to alter the **environment** of the about to be run program by `exec()`, enabling more features (ex. **Pipes** and **I/O redirection**) to be implemented

Why separate fork() and exec()? (Cont.)

- Example: The output of `wc`, instead of being shown in the screen is saved (**redirected**) to a text file

```
prompt> wc p3.c > newfile.txt
```

- To implement this, the shell closes the STDOUT of the process after `fork()` then `open()`'s `newfile.txt` before calling `exec()`
- This works because the `open()` system call starts from zero when looking for available file descriptors to use. Since STDOUT (fd=1) was closed, the call to `open()` will assign fd=1 to `newfile.txt` and will be treated as the STDOUT after the call to `exec()`
- **Pipes** works in a similar manner but uses in-kernel data structures for Interprocess Communication (IPC) through the `pipe()` system call
  - Example: The STDOUT of `ps` is connected to the STDIN of `grep` through a pipe (the vertical bar syntax in BASH)

```
prompt> ps -A | grep cpu.elf
```

Implementing STDOUT redirection to a file

p4.c

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

int
main(int argc, char *argv[]){
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
        …
```

Implementing STDOUT redirection to a file(Cont.)

p4.c

```
        …
        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");          // program: "wc" (word count)
        myargs[1] = strdup("p4.c");        // argument: file to count
        myargs[2] = NULL;                  // marks end of array
        execvp(myargs[0], myargs);         // runs word count
    } else {                           // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Result

```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

Process Control and Users

- There are other process-related system calls
- `kill()` – used to send **signals** to a process
  - `SIGINT` – pressing CTRL+c
  - `SIGSTOP` – pressing CTRL+z
  - `SIGCONT` – `fg` command
- Signals subsystem enable the delivery of events to processes
  - Processes and **process groups** can have signal handlers (via `signal()` system call) to perform a specific action whenever a specific signal is received
- Processes are associated to **users** in order to provide ownership and control, as well as limited security and protection
  - `getuid()` – system call to return the user id of the user of the process calling it

Useful programs and commands

- `man`
- `top/htop`
- `ps/pstree`
- `kill/killall`
- `fg/jobs`
- `/proc`