Interlude: Process API

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

The `fork()` System Call

- **The `fork()` system call is used to create a new process**

```
#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
    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;
}
```

rc = child의 pid값 return

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

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

The output is not deterministic; the OS scheduler determines which one runs at a given moment in time

- The process starts with printing “hello world” with its process identifier (PID)
- The newly-created process (child) by **`fork()`** is a copy of the calling process; but it does not start running at **`main()`**, which is not an exact copy
- The child process now has its own address space, registers, PC, and so forth
- The value it returns to the call of **`fork()`** is different (child: 0, parent: PID)

The `wait()` System Call

- The `wait()` system call is used to wait for a child to finish

```
#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 rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

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

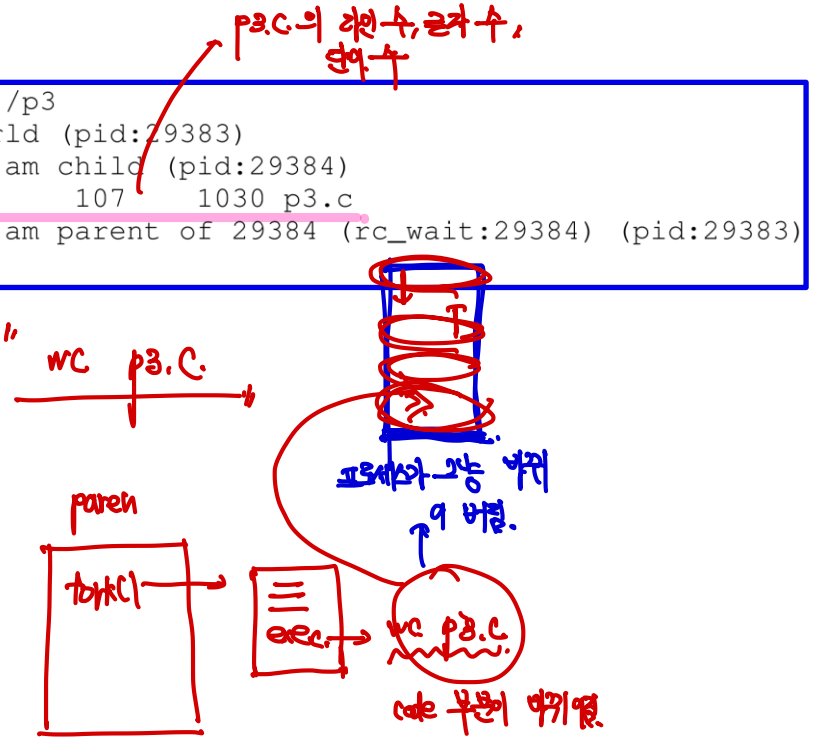
  - The parent process calls **`wait()`** to delay its execution the child finishes executing; when child is done, **`wait()`** returns to the parents
  - This **`wait()`** call makes the output deterministic

The exec() System Call

- **exec()** is to run a program that is different from calling program

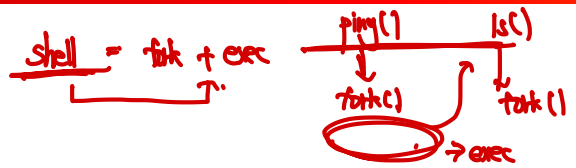
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

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

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



  - The **exec()** loads code and static data from the executable and overwrites its current code segment and static data with it (replacing the current with a new)
  - Also, the heap, stack and other parts of its memory space are re-initialized
  - A successful call to **exec()** never returns

Motivating the API



- **Separation of `fork()` and `exec()` is essential in building a shell**
  - It allows the shell to run code after the call to fork() but before the call to exec()
  - The shell is a user program that shows you a prompt and waits for you to type, then **`fork()`** and **`exec()`** with **`wait()`** when typed
  - An example of redirection

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

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

  - The redirection can be implemented by closing STDOUT (screen) and opening a file to be redirected between **`fork()`** and **`exec()`**

```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
    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);

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