CS:3620 Operating Systems

Process API

What API does the OS provide to user programs?

- API
  - Application Programming Interface
  - Functions available to write user programs
- API provided by OS is a set of system calls
  - System call is a function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
  - Some “blocking” system calls cause the process to be blocked and descheduled (e.g., read from disk)

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
  - Programs written to the POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Ensures program portability

- Program language libraries hide the details of invoking system calls
  - The *printf()* function in the C library calls the write system call to *write()* to screen
  - User programs usually do not need to worry about invoking system calls

Process related system calls (in Unix)

- *fork()* creates a new child process
  - All processes are created by forking from a parent
  - The *init* process is ancestor of all processes
- *exec()* makes a process execute a given executable
- *exit()* terminates a process
- *wait()* causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

What happens during a fork?

- A new process is created by making a copy of parent's memory image

- The new process is added to the OS process list and scheduled

- Parent and child start execution just after fork (with different return values)

- Parent and child execute and modify the memory data independently

fork()

- On success, the PID of the child process is returned in the parent,
  - 0 is returned in the child.
  - On failure, -1 is returned in the parent, no child process is created

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main(int argc, char *argv[])
7  {
8      printf("hello world (pid:%d)\n", (int) getpid());
9      int rc = fork();
10     if (rc < 0) {          // fork failed; exit
11         fprintf(stderr, "fork failed\n");
12         exit(1);
13     } else if (rc == 0) { // child (new process)
14         printf("hello, I am child (pid:%d)\n", (int) getpid());
15     } else {              // parent goes down this path (main)
16         printf("hello, I am parent of %d (pid:%d)\n",
17                rc, (int) getpid());
18     }
19     return 0;
20 }
```

Child

Parent

Figure 5.1: **Calling `fork()` (`p1.c`)**

When you run this program (called `p1.c`), you'll see the following:

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

fork() – Non deterministic Behaviors


```
$ ./p1
hello world (pid:31752)
I am parent of 31753 (pid:31752)
I am child (pid:31753)
```


Most of the times


```
$ ./p1
hello world (pid:31752)
I am child (pid:31753)
I am parent of 31753 (pid:31752)
```

Sometimes

Waiting for children to terminate …

- Process termination scenarios
  - By calling *exit()* (exit is called automatically when end of main is reached)
  - OS terminates a misbehaving process
- Terminated process exists as a zombie
- When a parent calls *wait()*, zombie child is cleaned up or “reaped”
- *wait()* blocks in parent until child terminates (non-blocking ways to invoke wait exist)
- What if parent terminates before child? *init* process adopts orphans and reaps them

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/wait.h>
5
6   int
7   main(int argc, char *argv[])
8   {
9       printf("hello world (pid:%d)\n", (int) getpid());
10      int rc = fork();
11      if (rc < 0) {           // fork failed; exit
12          fprintf(stderr, "fork failed\n");
13          exit(1);
14      } else if (rc == 0) { // child (new process)
15          printf("hello, I am child (pid:%d)\n", (int) getpid());
16      } else {                // parent goes down this path (main)
17          int wc = wait(NULL);
18          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                 rc, wc, (int) getpid());
20      }
21      return 0;
22  }
```

p2.c

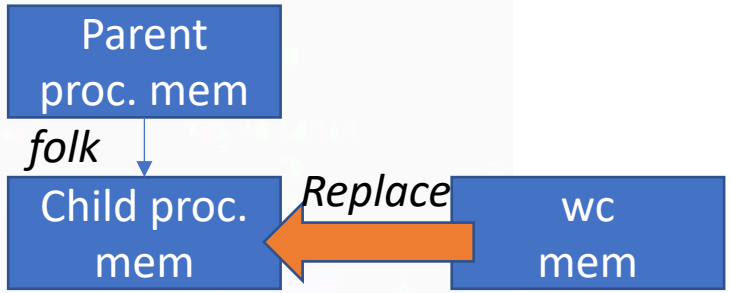
fork() + wait()


```
$ ./p2
hello world (pid:31752)
I am child (pid:31753)
I am parent of 31753 (pid:31752)
```


Always

What happens during exec?

- After fork, parent and child are running same code
  - Not too useful!
- A process can run *exec()* to load another executable to its memory image
  - So, a child can run a different program from parent
- Variants of *exec()*, e.g., to pass command line arguments to new executable



The child process executes "wc p3.c"

```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
        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;
}
```

p3.c

```
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>
```

How does a shell work?

- In a basic OS, the *init* process is created after initialization of hardware
- The *init* process spawns a shell like *bash*
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like *ls* are all executables that are simply exec'ed by the shell


```
prompt>ls
a.txt b.txt c.txt
```

More funky things about the shell

- Shell can manipulate the child in strange ways

- Suppose you want to redirect output from a command to a file

- *“ls > foo.txt”*

- Shell spawns a child, rewires its standard output to a file, then calls exec on the child

```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);

        // 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;
}
```


Figure 5.4: All Of The Above With Redirection (p4.c)

Here is the output of running the `p4.c` program:

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

Exercise

- Poll on Zoom

*ignore the extra space in printf

Exercise: Question 5


```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   int
6   main(int argc, char *argv[])
7   {
8       printf("hello world                ",                );
9       int rc = fork();
10      if (rc < 0) {           // fork failed; exit
11          fprintf(stderr, "fork failed  ");
12          exit(1);
13      } else if (rc == 0) { // child (new process)
14          printf("       I am child              "                    );
15      } else {                  // parent goes down this path (main)
16          printf("       I am parent                    ",
17                                  );
18      }
19      return 0;
20  }
```

Disclaimer

- *These lecture slides are based on a slide set by Youjip Won (Hanyang University) and Mythili Vutukuru (IIT Bombay)*