CS 201P Computer Security Winter 2020

Process, OS Interfaces, and getting hands-on with UNIX and C

17 January 2020

Aftab Hussain
University of California, Irvine

UNIX

C

OS Interfaces

A process usually alternates between
running in the user space and the kernel space

Since there may be multiple user processes, the kernel uses the CPU’s hardware protection mechanisms to ensure that each process executing in the user space can access only its own memory.

fork()

```
int pid  = fork();
if (pid > 0){
      printf(“parent: child=%d\n”, pid);
      pid = wait ();
      printf(‘child %d is done\n”, pid);
}     else if(pid == 0){
            printf(“child: exiting\n);
            exit();
}     else {
            printf(“fork error\n”);
}
```

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){
      printf("parent: child=%d\n", pid);
      pid = wait ();
      printf('child %d is done\n", pid);
}     else if(pid == 0){
            printf("child: exiting\n);
            exit();
}     else {
            printf("fork error\n");
}
```

```
int pid  = fork(); *  // a new child process is created with same memory contents as the parent process.
if (pid > 0){
      printf("parent: child=%d\n", pid);
      pid = wait ();
      printf('child %d is done\n", pid);
}     else if(pid == 0){
            printf("child: exiting\n);
            exit();
}     else {
            printf("fork error\n");
}
```

At this point we have two processes running.

> fork returns 0 to the child process.

> fork returns the pid of the child process in the parent process.

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){ // the parent process
      printf("parent: child=%d\n", pid);
      pid = wait ();
      printf('child %d is done\n", pid);
}     else if(pid == 0){ // the child process
            printf("child: exiting\n);
            exit();
}     else {
            printf("fork error\n");
}
```

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){ // the parent process
      printf("parent: child=%d\n", pid);
      pid = wait (); // returns the id of an exited child process
      printf('child %d is done\n", pid);
}     else if(pid == 0){ // the child process
            printf("child: exiting\n);
            exit();
}     else {
            printf("fork error\n");
}
```

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){ // the parent process
      printf("parent: child=%d\n", pid);
      pid = wait (); // returns the id of an exited child process
      printf('child %d is done\n", pid);
}     else if(pid == 0){ // the child process
            printf("child: exiting\n);
            exit(); // exits the calling process, releasing resources
}     else {
            printf("fork error\n");
}
```

So what’s the output? Say child pid = 1234.

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){ // the parent process
      printf("parent: child=%d\n", pid);
      pid = wait (); // returns the id of an exited child process
      printf('child %d is done\n", pid);
}     else if(pid == 0){ // the child process
            printf("child: exiting\n);
            exit(); // exits the calling process, releasing resources
}     else {
            printf("fork error\n");
}
```

```
int pid  = fork();     // a new child process is created with same memory contents as the parent process.
if (pid > 0){ // the parent process
      printf("parent: child=%d\n", pid);
      pid = wait (); // returns the id of an exited child process
      printf('child %d is done\n", pid);
}     else if(pid == 0){ // the child process
            printf("child: exiting\n);
            exit(); // exits the calling process, releasing resources
}     else {
            printf("fork error\n");
}
```

Output:

parent: child=1234
child: exiting
parent: child 1234 is done

OR

child: exiting
parent: child=1234
parent: child 1234 is done

While the memory contents of the child and parent processes are *initially* the same, changing a variable in one does not affect the other.

exec()

exec() replaces the calling process's memory with
a new memory image loaded from a file
stored in the file system.

When exec succeeds, it does not return to the calling program.

Ref:

xv6 - a simple, Unix-like teaching operating system, Cox et al., PDOS, CSAIL, MIT