C Process API

Glenn Bruns
CSUMB

Creating processes with processes

From a C program, you can write code to create processes and start them running.

What if you write a function fork_bomb() that creates two new processes and has each of them run fork_bomb()?

But actually, how do you create new processes in C?

Lecture Objectives

At the end of this lecture, you should be able to:

- Be able to use the C process API

Process control in C

Sometimes applications want to create or kill processes.

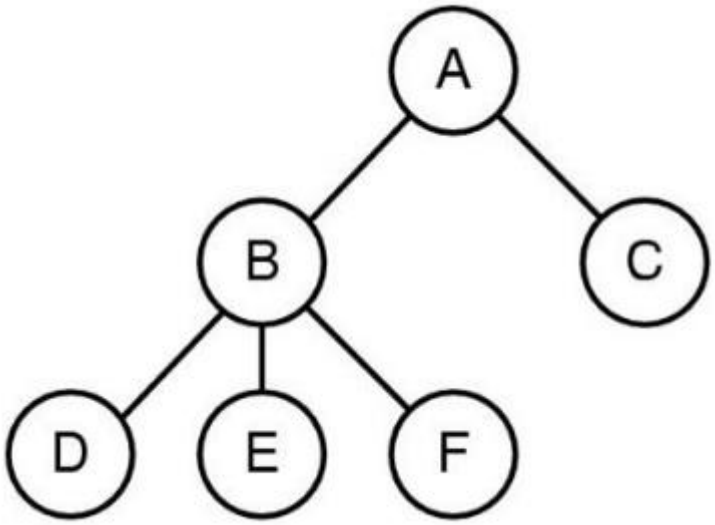
A running app is a process – so sometimes processes want to create or kill processes.

For example the shell sometimes will create and kill processes.

how the shell works:

```
forever {
   print “$”
   get user command
   run command as a separate process
   wait for the process to end
}
```

A process tree



Process A created two child processes: B and C

Process B is a "child process" of A

figure from Tanenbaum, Modern Operating Systems

Process creation with fork

Fork is weird and wonderful.

It clones the process that made the fork call.

One process makes the call; two processes see the return!

```
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  }
```

this code is from OSTEP

Exercise: what does this output?

```
int
main(int argc, char *argv[])
{
  fork();
  printf(“hello!\n”);
  return 0;
}
```

```
$ ./fork-prob4
hello!
$ hello!
```

Exercise: what is wrong with this?

```
int
main(int argc, char *argv[])
{
  int pid = getpid();
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == pid) {
    printf("child (pid:%d)\n", (int)getpid());
  } else {
    printf("parent (pid:%d)\n", (int)getpid());
  }
  return 0;
}
```

Exercise: what does this output?

```
int
main(int argc, char *argv[])
{
  fork();
  fork();
  printf("hello!\n");
  return 0;
}
```

```
$ ./fork-prob5
hello!
$ hello!
hello!
hello!
```

Wait

A **wait** call returns when one of the child processes of a process has terminated.

```
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  }
```

this code is from OSTEP

Exercise: what does this output?

```
int
main(int argc, char *argv[])
{
  printf("pid = %d\n", (int)getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else {
    wait(NULL);
  }
  printf("pid = %d\n", (int)getpid());
  return 0;
}
```

```
$ ./wait-prob1
pid = 22073
pid = 22074
pid = 22073
```

Exec

An **exec** call changes the code that a process is running.

```
7   int
8   main(int argc, char *argv[])
9   {
10      printf("hello world (pid:%d)\n", (int) getpid());
11      int rc = fork();
12      if (rc < 0) {          // fork failed; exit
13          fprintf(stderr, "fork failed\n");
14          exit(1);
15      } else if (rc == 0) { // child (new process)
16          printf("hello, I am child (pid:%d)\n", (int) getpid());
17          char *myargs[3];
18          myargs[0] = strdup("wc");   // program: "wc" (word count)
19          myargs[1] = strdup("p3.c"); // argument: file to count
20          myargs[2] = NULL;           // marks end of array
21          execvp(myargs[0], myargs);  // runs word count
22          printf("this shouldn't print out");
23      } else {                  // parent goes down this path (main)
24          int wc = wait(NULL);
25          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                  rc, wc, (int) getpid());
27      }
28      return 0;
29  }
```

this code is from OSTEP

Summary

- `fork` – make a copy of the current process
- `wait` – wait for a child process to terminate
- `exec` – code of current process is replaced