System calls for process management

Mythili Vutukuru
CSE, IIT Bombay

API for process management

- What API does OS provide to user programs to manage processes?
  - How to create, run, terminate processes?
- 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 higher CPU privilege level
  - 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 context switched out (e.g., read() from disk), while others (e.g., getpid() to get PID) can return immediately

Portability of code across OS

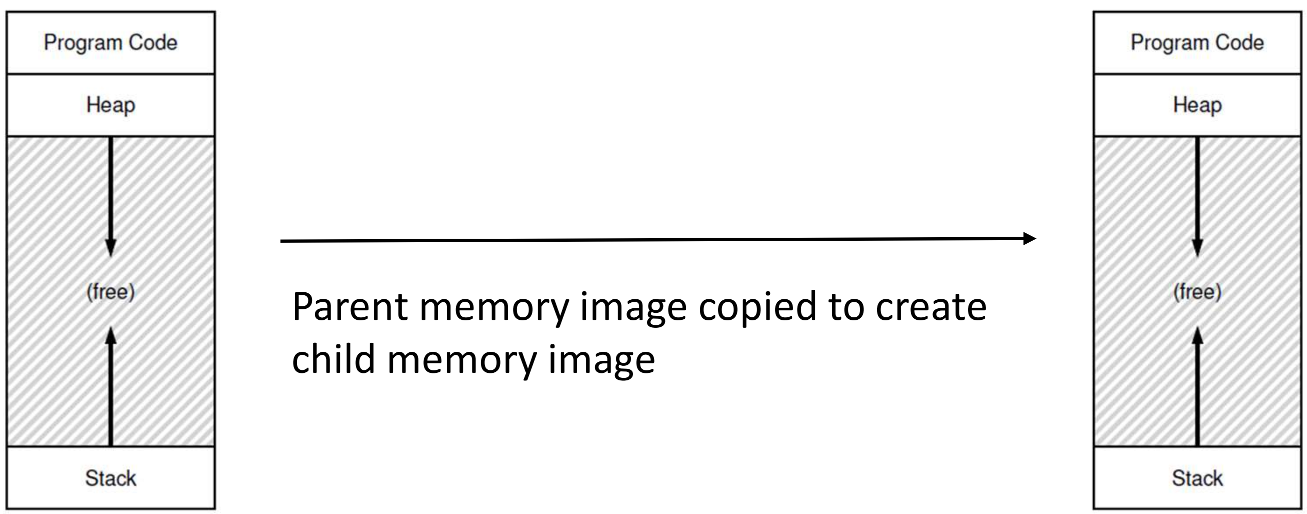
- POSIX API: standard set of system calls (and some C library functions) available to user programs, defined for portability
  - Programs written using POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Program may still need to be recompiled for different architectures
- Program language libraries hide the details of invoking system calls
  - The `printf` function in libc calls the `write` system call to write to screen
  - User programs usually do not need to worry about invoking system calls
- ABI (application binary interface) is the interface between machine code and underlying hardware: ISA, calling convention, …

Process related system calls (in Unix)

- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - OS starts `init` process after boot up, which forks other processes
  - The `init` process is ancestor of all processes, including shell/terminal
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants of the above system calls exist in language libraries with different arguments

Process creation: fork

- Parent process calls "fork" system call to create (spawn) a new process
- New child process created with new PID
- Memory image of parent is copied into that of child
- Parent and child run different copies of same code



Parent memory image copied to create child memory image

Image credit: OSTEP

What happens after fork?

- Parent and child resume execution in their copies of the code
- Child starts executing with a return value of 0 from fork
- Parent resumes executing with a return value equal to child PID
- Parent and child run independently
- Any changes in parent's data after fork does not impact child

```
int ret = fork()
if(ret == 0) {
  print "I am child"
}
else if(ret > 0) {
  print "I am parent"
}
```

Child resumes here

Parent resumes here

```
int ret = fork()
if(ret == 0) {
  print "I am child"
}
else if(ret > 0) {
  print "I am parent"
}
```

```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;
}
```

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

Image credit: OSTEP

Example code with fork

- Parent and child run independently and print to screen
- Order of execution of parent and child can vary

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>
```

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

Image credit: OSTEP

Example code with fork

- What values of x are printed?
- Parent and child both start with their own independent copies of variable x in their memory images
- Child increments its copy of x, prints 2
- Parent decrements its copy of x, prints 0

```
int ret = fork()
int x = 1
if(ret == 0) {
  print "I am child"
  x = x+1
  print x
}
else if(ret > 0) {
  print "I am parent"
  x = x -1
  print x
}
```

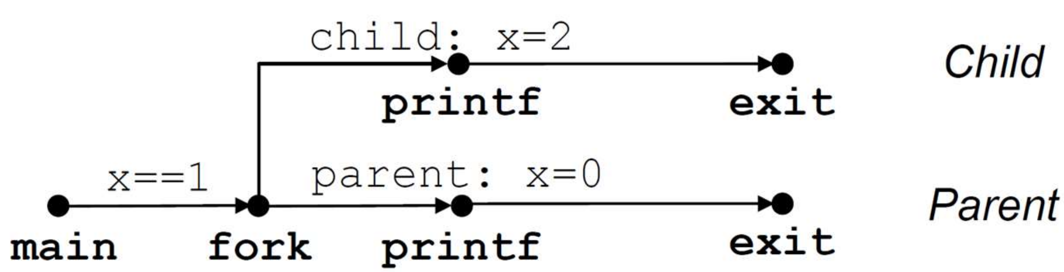


Image credit: CSAPP

Example code with nested fork

- Total 4 processes (1 parent + 3 child)
- Hello printed 4 times

fork()
fork()
print hello
exit

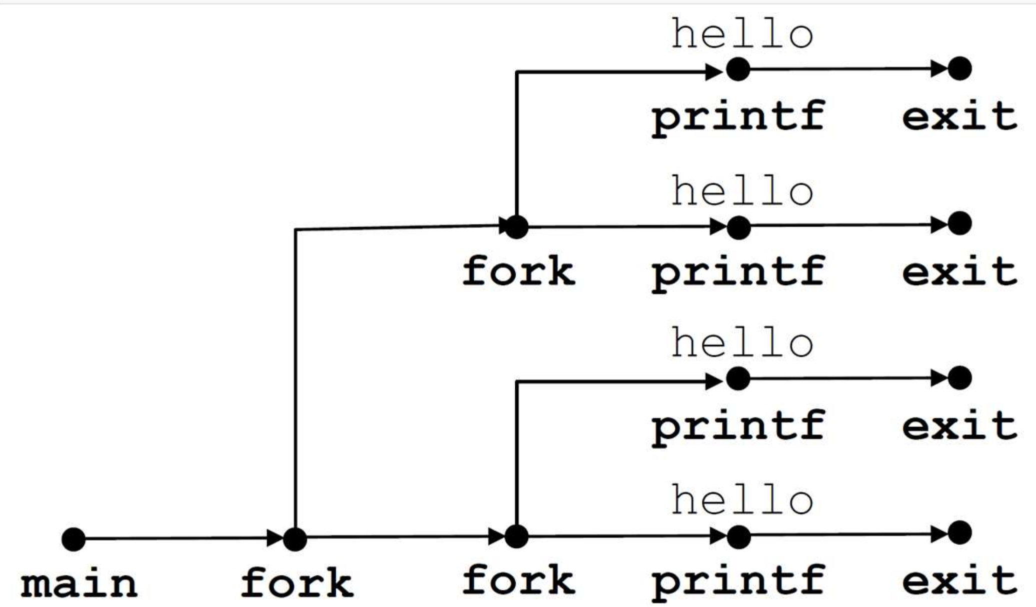


Image credit: CSAPP

Exit system call

- When a process finishes execution, it calls exit system call to terminate
  - OS switches the process out and never runs it again
  - Exit is automatically called at end of main
- Exiting process cannot clean up its memory, and memory must be freed up by someone else (why? More on this later.)
- Terminated process exists in a zombie state
- How are zombies cleaned up?

Wait system call

- Parent calls wait system call to reap (clean up memory of) a zombie child
- Wait cleans up memory of one terminated child and returns in parent process
- If child still running, wait system call blocks parent until child exits
- If child terminated already, wait reaps child and returns immediately
- If parent with no child calls wait, it returns immediately without reaping anything

```
…
int ret = fork()
if(ret == 0) {
  print "I am child"
  exit()
}
else if(ret > 0) {
  print "I am parent"
  wait()
}
…
```

More on wait

- Wait system call variant waitpid reaps a specific child with a given PID, while regular wait reaps any terminated child
  - Read man pages for more details on arguments to waitpid and wait
- Wait system call “reaps” one dead child at a time (in any order)
  - Every fork must be followed by call to wait at some point in parent
- What if parent has exited while child is still running?
  - Child will continue to run, becomes orphan
  - Orphans adopted by init process, reaped by init when they terminate
- If parent forks children, but does not bother calling wait for long time, system memory fills up with zombies
  - Common programming error, exhausts system memory

```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;
}
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

Image credit: OSTEP

Example code with fork and wait

- Order of printing of child and parent is deterministic now
- Why? Parent waits until child prints and exits, then prints

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

Image credit: OSTEP

Exec system call

- Isn't it impractical to run the same code in all processes?
  - Sometimes parent creates child to do similar work..
  - .. but other times, child may want to run different code
- Child process uses "exec" system call to get a new "memory image"
  - Allows a process to switch to running different code
  - Exec system call takes another executable as argument
  - Memory image is reinitialized with new executable, new code, data, stack, heap, …

```
…
int ret = fork();
if(ret == 0) {
  exec("some_executable")
}
else if(ret > 0) {
  print "I am parent"
}
…
```

```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;
}
```

Figure 5.3: Calling **fork()**, **wait()**, And **exec()** (**p3.c**)

Image credit: OSTEP

Example code with exec

- Many variants of exec system call (execvp used in example), which differ in the arguments provided (read more in man pages)
- If exec successful, child gets new memory image, never comes back to the code in old memory image after exec
  - Print statement after exec doesn't run if exec successful
- If exec unsuccessful, reverts back to original memory image

```
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>
```

Image credit: OSTEP

Shell / Terminal

- After bootup, the `init` process is first process created
- The `init` process spawns a shell like `bash`
- All future processes are created by forking from existing processes like `init` or shell
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like `ls, echo, cat` are all readily available executables that are simply exec-ed by the shell

Example shell code

- How does the shell run a user command?
- Read input from user
- Shell process forks a child process
- Child process runs exec with “echo” program executable as argument, calls exit when done
- Parent shell calls wait, blocks till child terminates, reaps it, goes back for next input

```
$echo hello
hello
$
```

```
do forever {
   input(command)

   int ret = fork()

   if(ret == 0) {
      exec(command)
   }
   else {
      wait()
   }
}
```

More on shell and commands

- Some commands already exist as programs written by OS developers and compiled into executables
  - Shell runs such command by simply calling exec in child process
- Some commands are implemented directly in shell code itself
- Think: why doesn’t shell exec command directly? Why fork a child?
  - Do we want the shell program code to be rewritten fully?
- For “cd” command, “chdir” system call used to change directory of parent process itself, no child process is forked. Why?
  - Every process has a current working directory
  - Do we want to change directory of some child process or shell itself?

Foreground and background execution

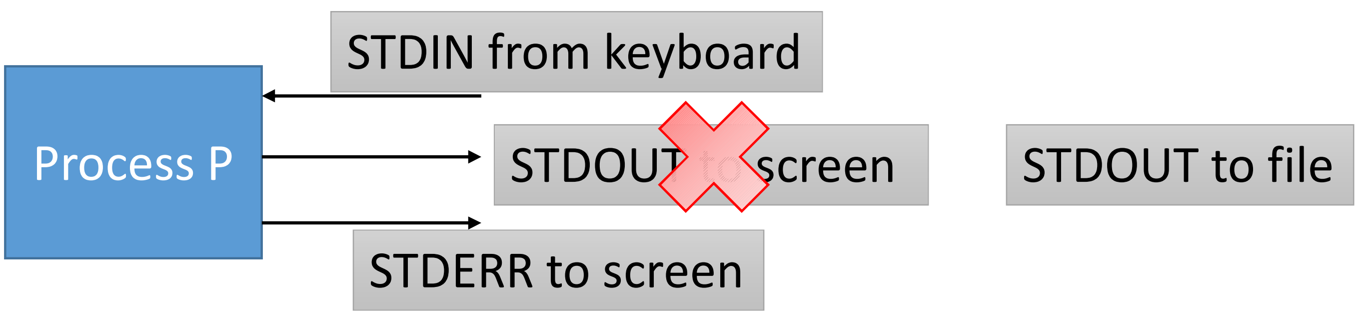
```
$sleep 10 &
$
```

- By default, user command runs in foreground, shell cannot accept next command until previous one finishes
- Background execution: when we type command followed by &
  - Shell starts child to run command, but does not wait for command to finish
- Background processes reaped at a later time by shell
  - When? Periodically? When next input is typed?
  - How? There is a way to invoke wait where parent is not blocked even if child has not exited (explore it on your own)
- It is also possible to run multiple commands in the foreground
  - One after the other serially (next command starts after previous finishes)
  - Or, all start at same time in parallel
  - Explore how such things can be done in the standard Linux shell

I/O redirection

```
$ls > foo.txt
$
```

- Every process has some I/O channels (“files”) open, which can be accessed by file descriptors
  - STDIN, STDOUT, STDERR open by default for all processes
- Parent shell can manipulate these file descriptors of child before exec in order to do things like I/O redirection
- E.g., output redirection is done by closing the default STDOUT and opening a regular file in its place



```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>
```

Open uses the first available file descriptor (STDOUT in this case)

Image credit: OSTEP

Shell commands with pipes

- Shell can also "pipe" the output of one command into another, by connecting STDOUT of one child to the STDIN of another child via a pipe (a communication mechanism provided by kernel)

```
$ cat foo.c | grep factorial
```

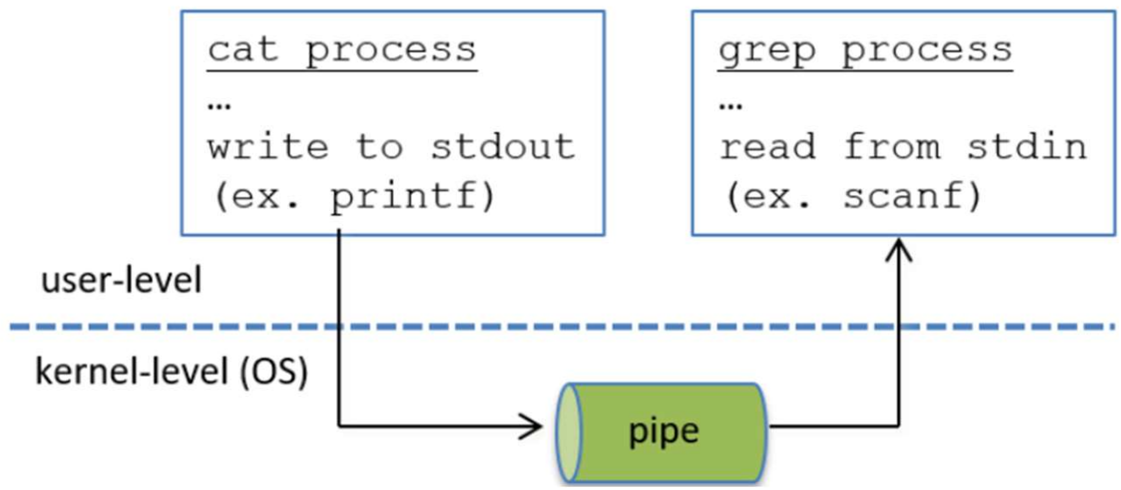


Image credit: Dive Into Systems