Processes

Operating Systems
Kartik Gopalan

References:

- Chapter 4 of OSTEP book
- Chapter 2 of the Tanenbaum's book

Process

- Informal definition:

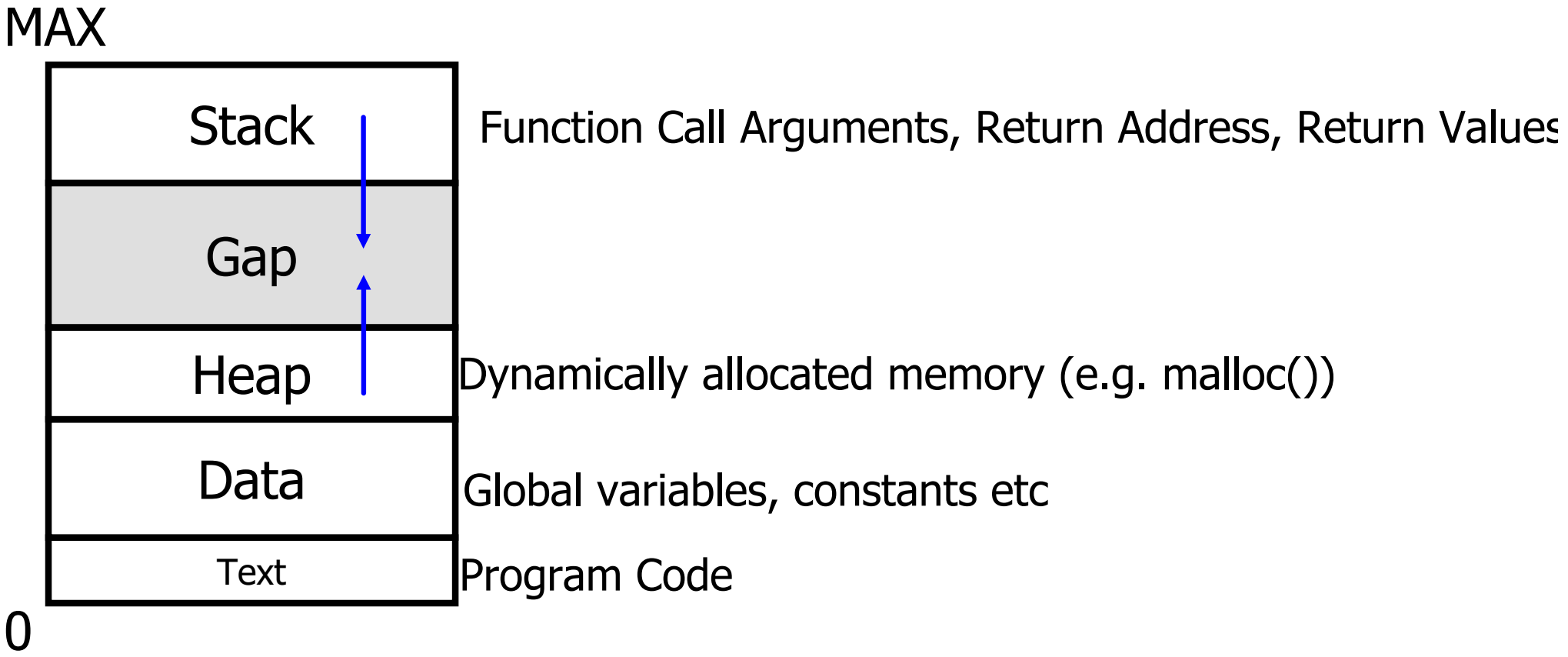
  A process is a program in execution.

- Process is not the same as a program.
  - Program is a passive entity stored in the disk
  - Process is an actively executing entity
  - Program is just one component of a process.

So what else constitutes a process?

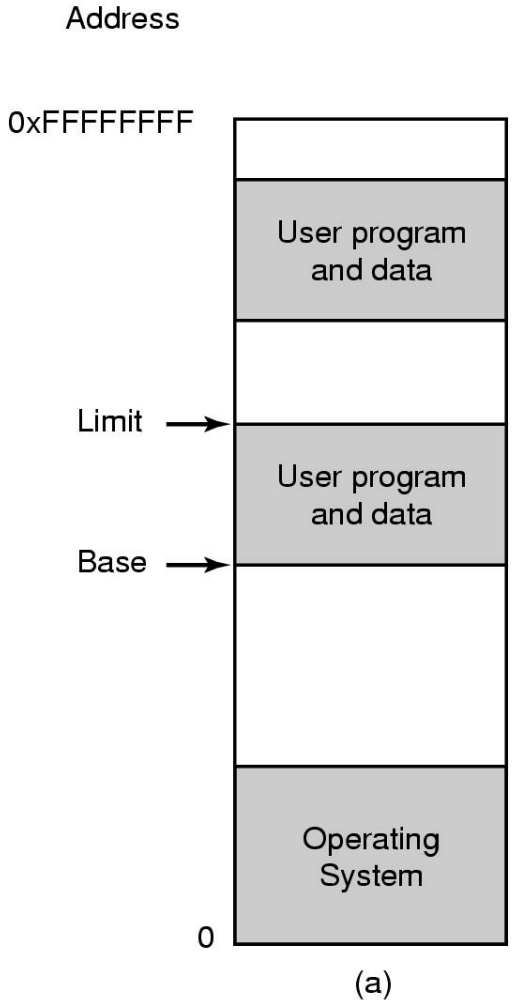
- Memory space (static, dynamic)
- Procedure call stack
- Registers and counters :
  - Program counter, Stack pointer, General purpose registers
- Open files, connections
- …….

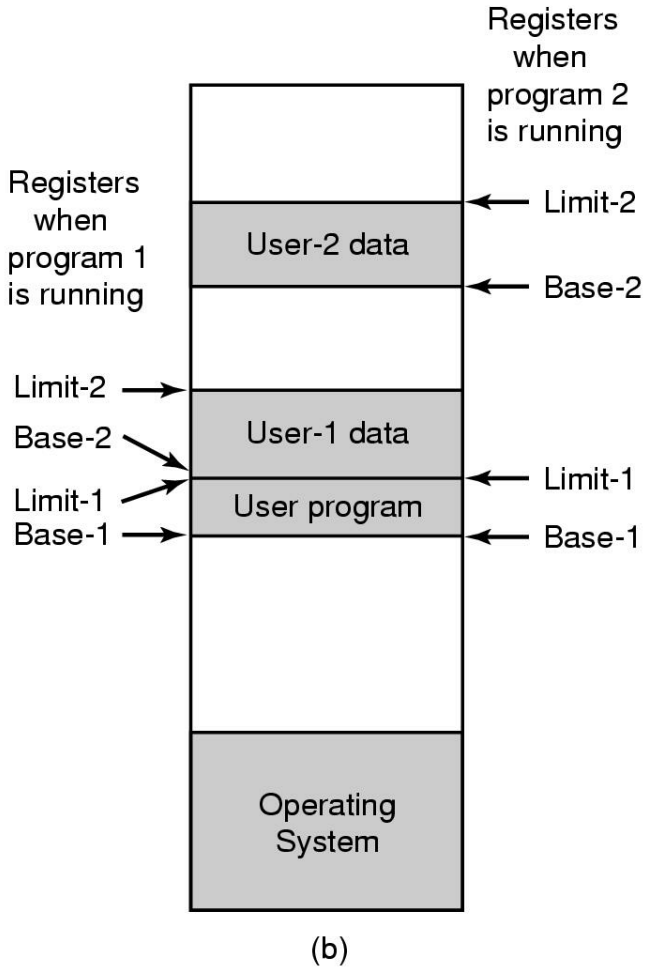
Memory Layout of a typical process



- Stack and heap grow towards each other

Multiple processes sharing main memory





Two processes running different programs

Two processes running the same program

Process Creation

- Always using fork() system call.

- When?
  - User runs a program at command line
  - OS creates a process to provide a service
    - Check the directory /etc/init.d/ on Linux for scripts that start off different services at boot time.
  - One process starts another process
    - For example in servers

Creating a New Process - fork()

Example code fork_ex.c

http://oscourse.github.io/examples/fork_ex.c

```
pid = fork();

if (pid == -1) {
        fprintf(stderr, "fork failed\n");
        exit(1);
}

if (pid == 0) {
        printf("This is the child\n");
        exit(0);
}

if (pid > 0) {
        printf("This is parent. The child is %d\n", pid);
        exit(0);
}
```

Points to Note

- fork() is called once …

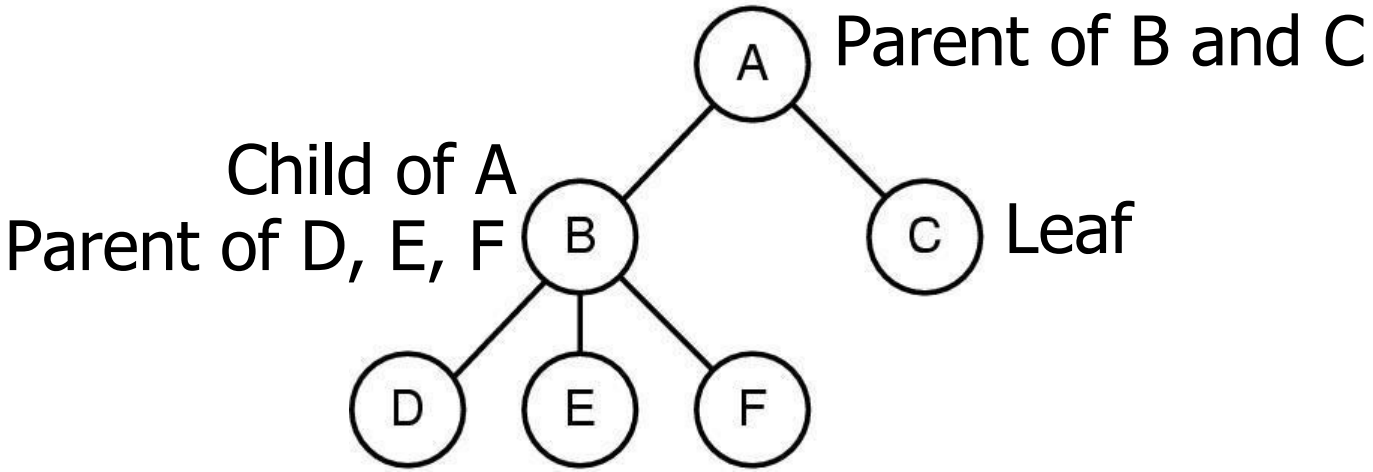
- But it returns twice!!
  - Once in the parent and
  - Once in the child

- The parent and the child are two different processes.

- Child is an exact "copy" of the parent.

- So how to make the child process do something different?
  - Return value of fork in child = 0
  - Return value of fork in parent = [process ID of the child]
  - By examining fork's return value, the parent and the child can take different code paths.

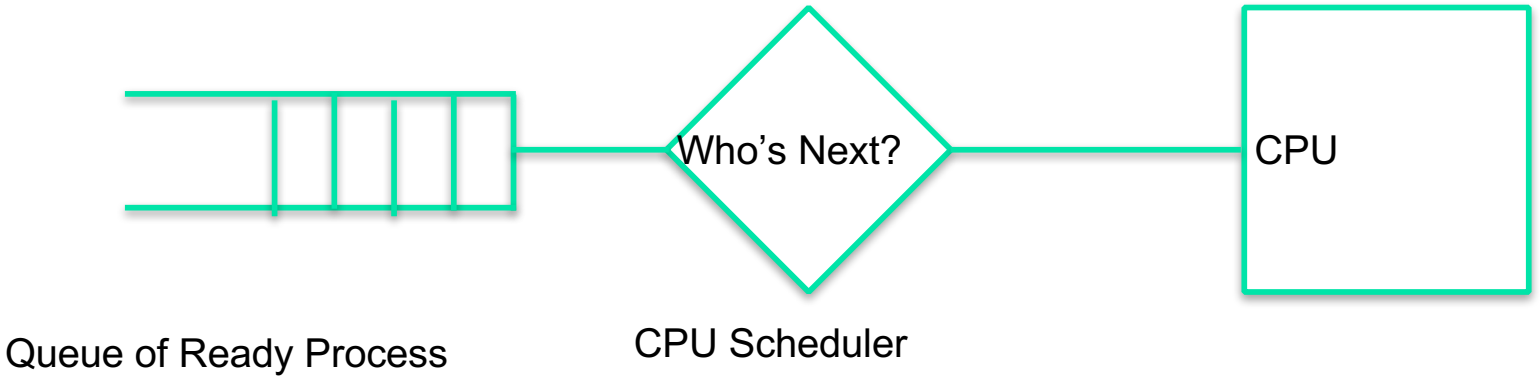
Process Hierarchy Tree



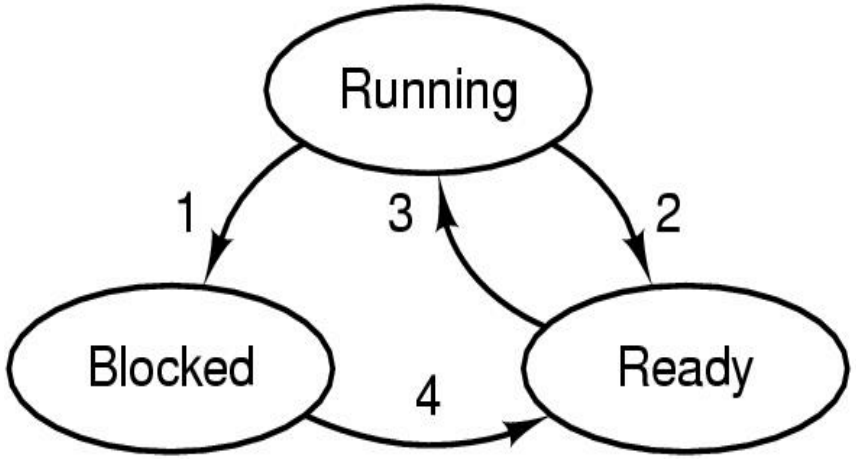
- A created two child processes, B and C
- B created three child processes, D, E, and F

CPU scheduler

- Time-shares many processes on one CPU



Process Lifecycle



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Ready
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running
  - Process is executing on the CPU
- Blocked
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue.

Typical Kernel-level data structure for each process

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

- See task_struct in Linux at http://lxr.linux.no

Examining Processes in Unix/Linux

- ps command
  - Standard process attributes

- /proc directory
  - More interesting information if you are the root.

- top command
  - Examining CPU and memory usage statistics.

The exec() system call

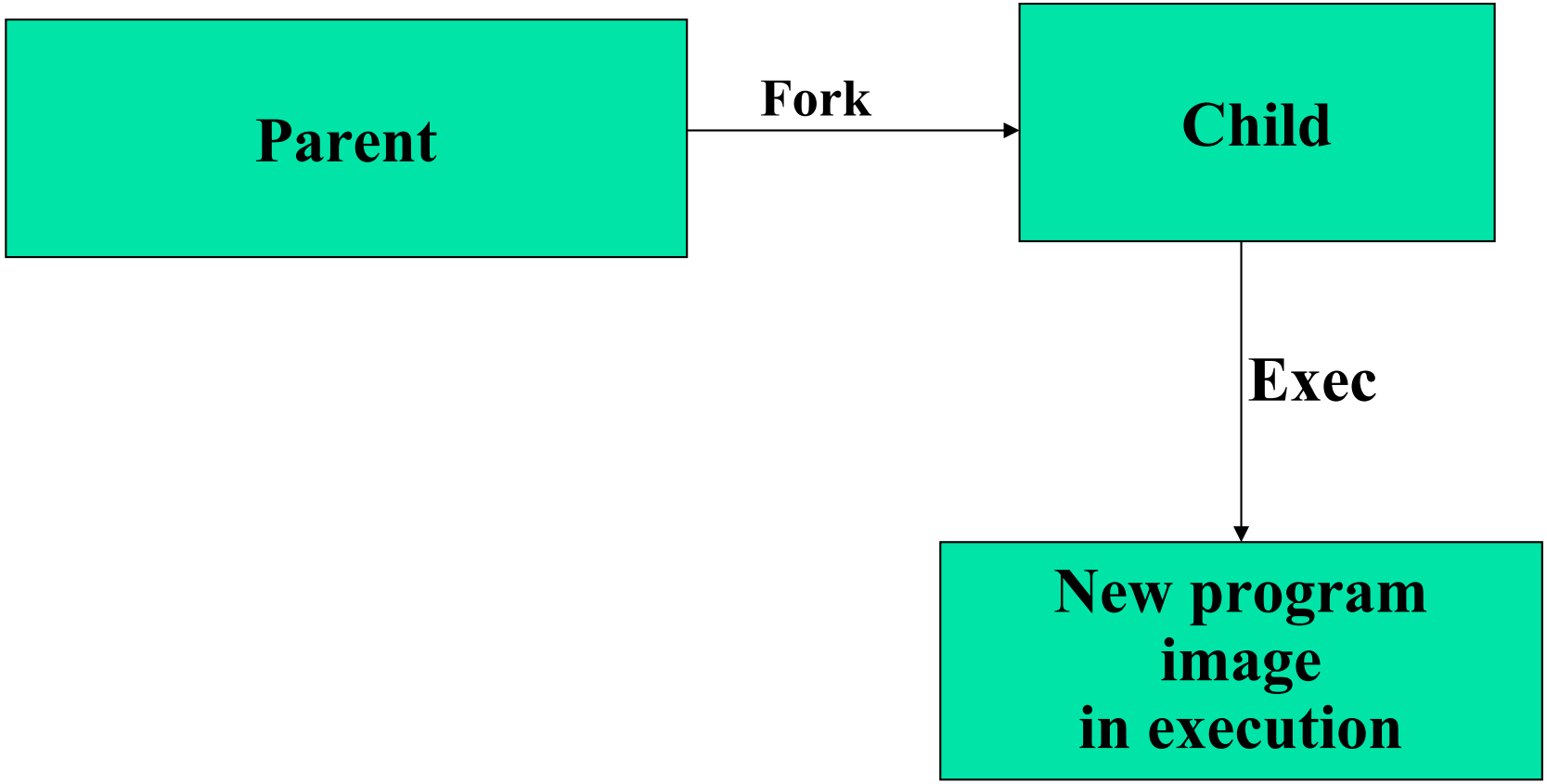
- Consider how a shell executes a command

$ pwd
/home/kartik

- How did that work?
  - Shell forked a child process
  - The child process executed /bin/pwd using the exec() system call

- Exec replaces the process' memory with a new program image.

Running another program in child – exec()



exec() — Example code exec_ex.c

http://oscourse.github.io/examples/exec_ex.c

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
}

if (pid == 0) {
    if( execlp("echo",
               "echo",
               "Hello from the child",
               (char *) NULL) == -1)
        fprintf(stderr, "execl failed\n");


    exit(2);
}

printf("parent carries on\n");
```

Properties of exec()

- Replaces current process image with new program image.
  - In the last example, parents' image was replaced by the "echo" program image.

- All I/O descriptors open before exec stay open after exec.
  - I/O descriptors = file descriptors, socket descriptors, pipe descriptors etc.
  - This property is very useful for implementing filters.

Different Types of exec()

- `int execl(char * pathname, char * arg0, … , (char *)0);`
  - Full pathname + long listing of arguments

- `int execv(char * pathname, char * argv[]);`
  - Full pathname + arguments in an array

- `int execle(char * pathname, char * arg0, … , (char *)0, char envp[]);`
  - Full pathname + long listing of arguments + environment variables

- `int execve(char * pathname, char * argv[], char envp[]);`
  - Full pathname + arguments in an array + environment variables

- `int execlp(char * filename, char * arg0, … , (char *)0);`
  - Short pathname + long listing of arguments

- `int execvp(char * filename, char * argv[]);`
  - Short pathname + arguments in an array

- More info: check “man 3 exec”

wait() system call

Helps the parent process

- to know when a child completes.
- to check the return status of child

wait( ) system call

```
if ((pid = fork()) == 0) {
        /* in the child – exec another program image */
        ret = execlp("lpr", "lpr", "myfile", (char *) 0);

        if( ret == -1 )
        fprintf(stderr, "exec failed\n");


} else {
        /* in the parent - do some other stuff */
  ......

        /* now check for completion of child */
        waitpid(pid, &status, 0);

        /* Alternative ==> while (wait(&status) != pid); */

        /* remove file */
        unlink("myfile");
}
```

Few other useful syscalls

- sleep(seconds)
  - suspend execution for certain time

- exit(status)
  - Exit the program.
  - Status is retrieved by the parent using wait().
  - 0 for normal status, non-zero for error

Orphan process

- When a parent process dies, child process becomes an orphan process.
- The init process (pid = 1) becomes the parent of the orphan processes.
- Here's an example:
  - http://oscourse.github.io/examples/orphan.c
  - Do a 'ps –l' after running the above program and check parent's PID of the orphan process.
  - After you are done remember to kill the orphan process 'kill –9 <pid>'

Zombie Process

- When a child dies, a SIGCHLD signal is sent to the parent.
- If parent doesn't wait()on the child, and child exit()s, it becomes a zombie (status "Z" seen with ps).
- Zombies hang around till parent calls wait() or waitpid().
- But they don't take up any system resources.
  - Just an integer status is kept in the OS.
  - All other resources are freed up.

References

- Chapter 4 of OSTEP book
- Chapter 2 of the Tanenbaum's book

- Man pages for different system calls
  - Try "man 2 <syscall_name>"
    - E.g. man 2 exec
  - Syscalls are normally listed in section 2 of the man page

- Google for "Linux source code"