CS310 Operating Systems

Lecture 8: Process - System Calls – exit, wait, exec

Ravi Mittal
IIT Goa

Acknowledgements !

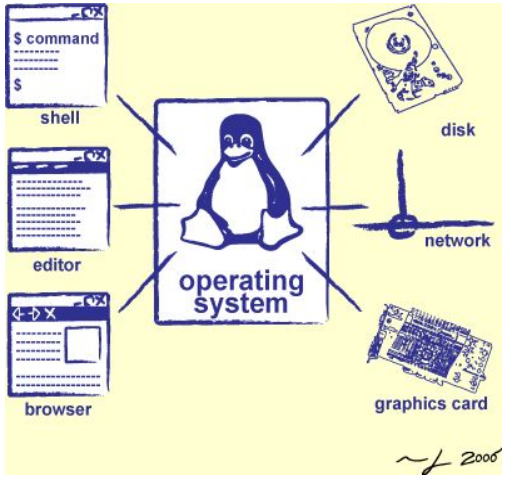
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Class presentation: University of California, Berkeley: David Culler, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner
  - Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
    - Chapter 5: Process APIs
    - Programs are taken from this chapter
  - CS 423 Operating System Design, Uinv of Illinois, Prof Fagen-Ullmschneider
  - CS351 University of Washington

Read the following:

- Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 1, Kernel and Processes
    - Chapter 4
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
  - Chapter 5: Process APIs

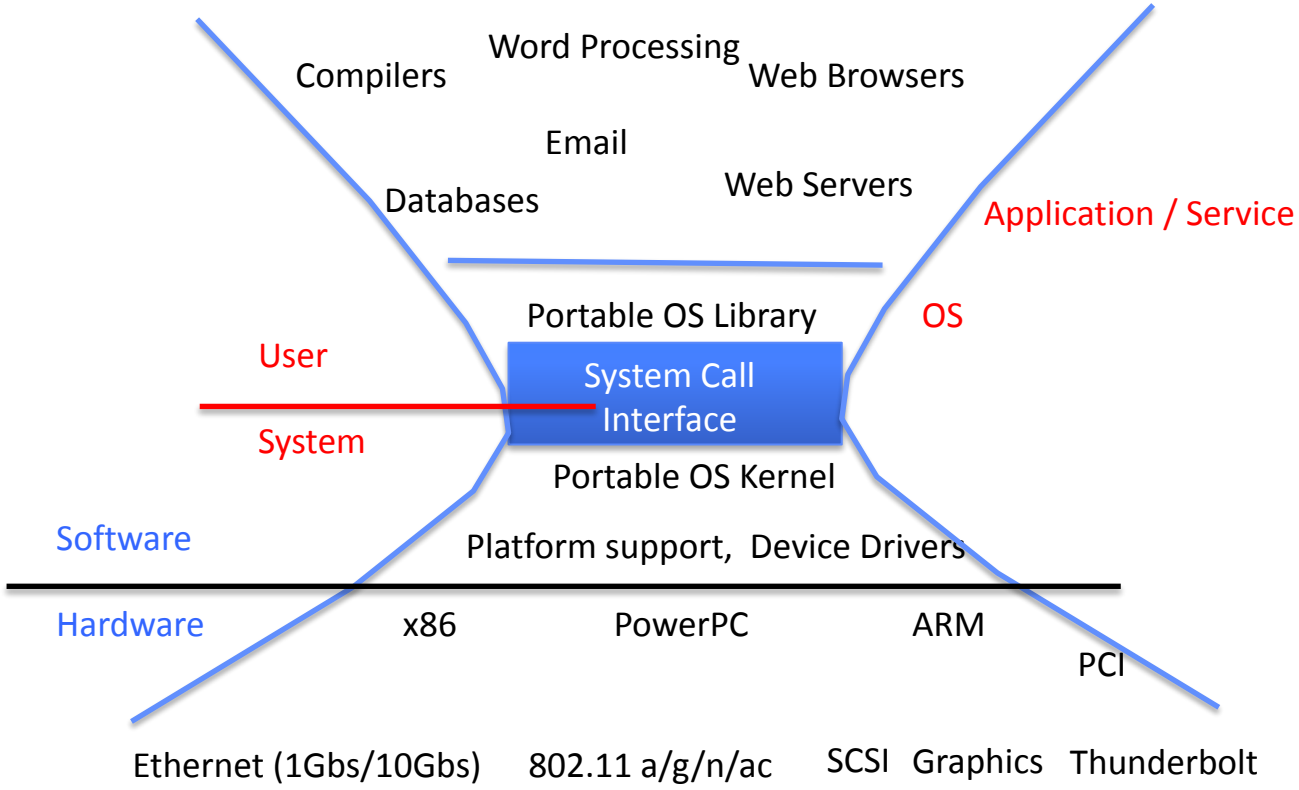
We will study..

- exit() system call
- wait() system call
- exec* sytem calls

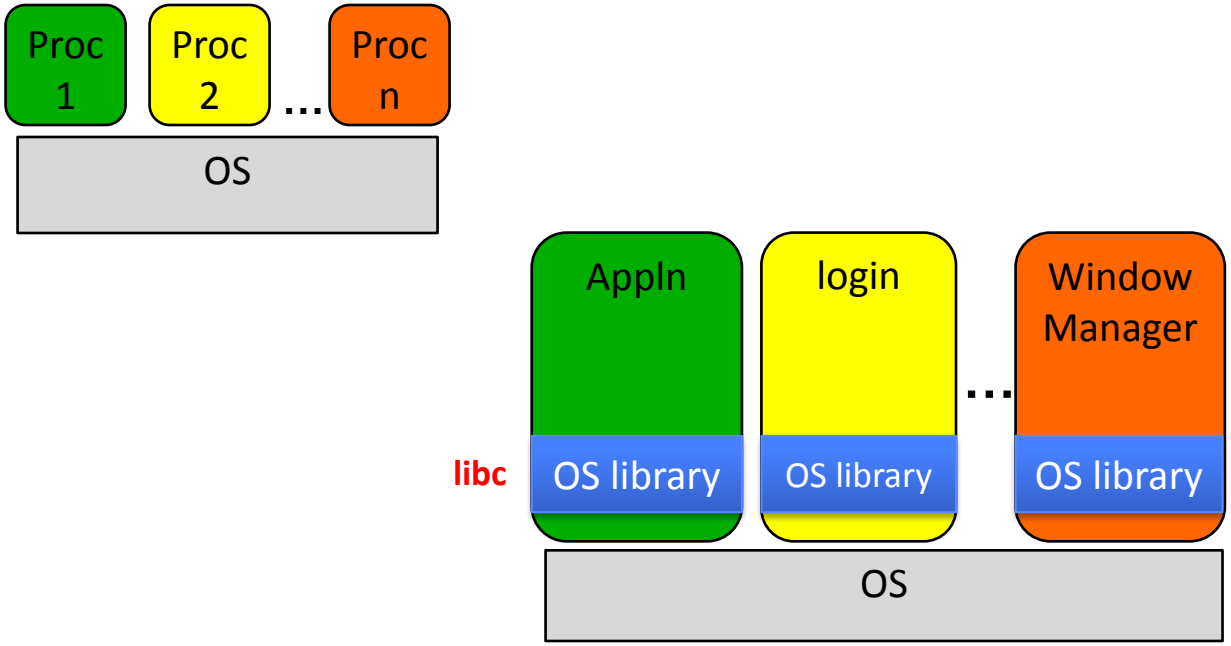


We have studied so far …

System Calls ("Syscalls")



OS Library Issues Syscalls



System Calls

Process Related System Calls (in Unix)

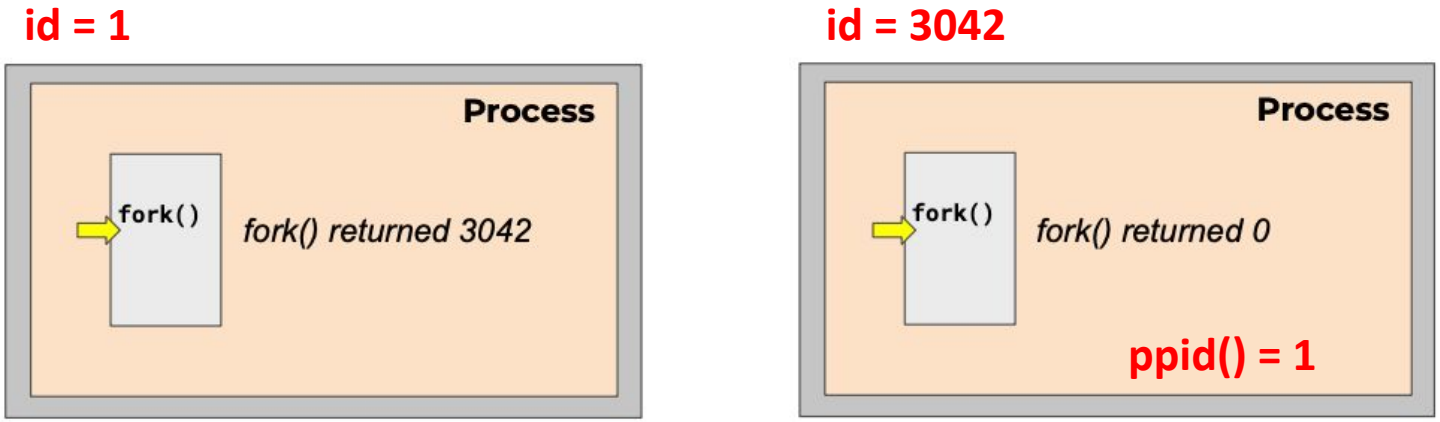
Last class

- 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

- There are many variants of the above system calls with different arguments

Creating a process

- Just one process – Parent process
- Initially there is one process – init with id = 1



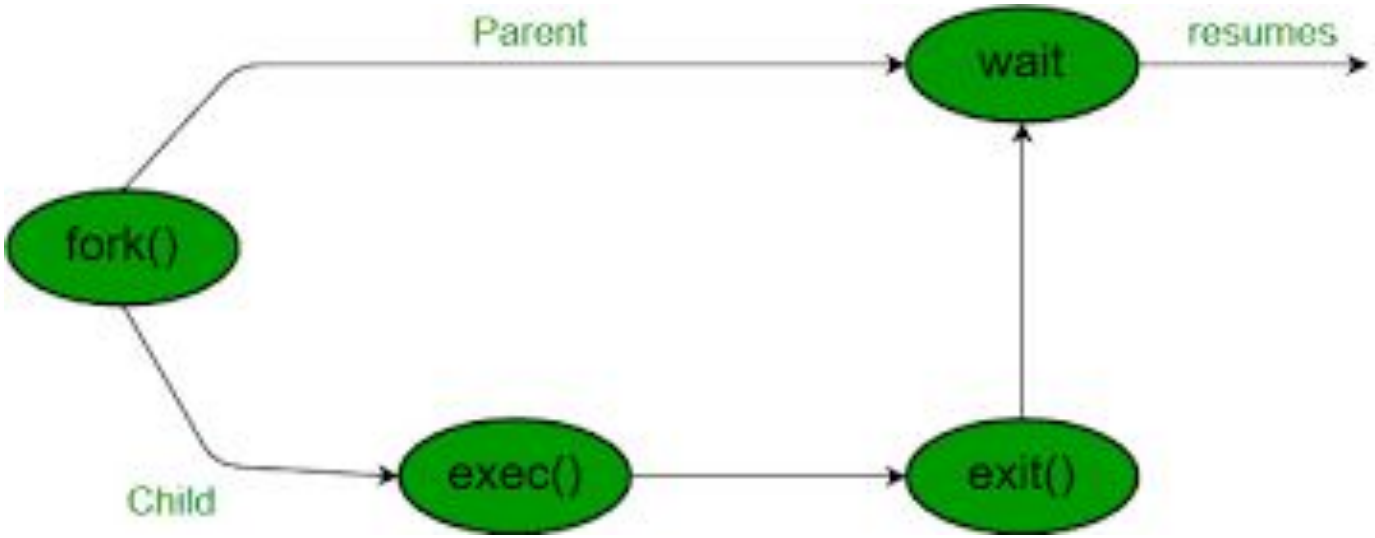
System Calls – `exit()` and `wait()`

Process termination

- Multiple ways for a process to get destroyed
  - Process issues and `exit()` call – Voluntary
  - The parent process issues a `kill()` call - Involuntary
  - Process receives a terminate signal - Involuntary
    - When it did something illegal !
- On death
  - Reclaim all of process's memory regions
  - Make process un-runnable
  - Put the process in the *zombie state* (will discuss it later)
  - However, do not remove its process descriptor from the list of processes

Waiting for children to die with `wait()`

- The parent can wait for the child to die by executing the wait system call
- It is quite useful for a parent to wait for a child process to finish what it has been doing
  - on success, **returns** the process ID of the terminated child; on error, -1 is **returned**.



`wait` and `waitpid` syscalls

- A terminated process's information is collected via a call of a wait operation by its parent
- Operations:
  - `wait`
    - blocks the calling process until a child process terminates, returning the child's pid
      - If caller has no children, wait immediately returns –1 (error)
    - the termination status (return value) of the child may be obtained via the argument
  - `waitpid`
    - permits a caller to wait for a particular child, identified by its pid

wait() system call

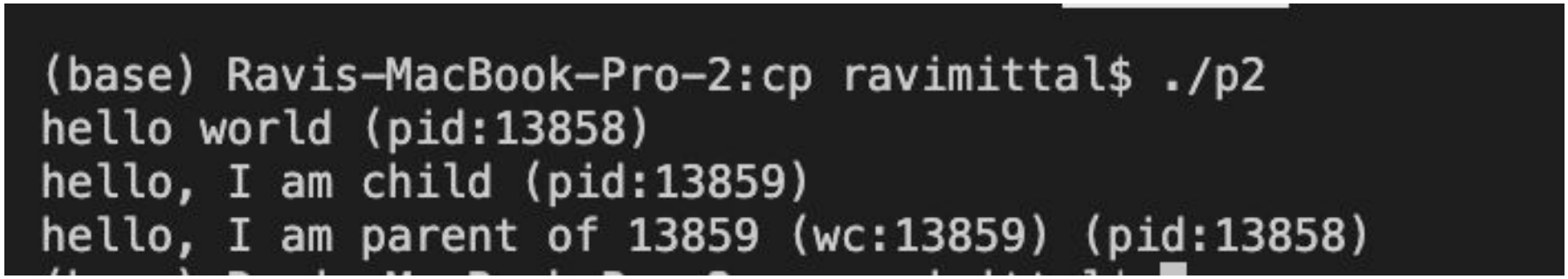
p2.c

```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());
        sleep(1);
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
```

wait() system call

Parent process waits for the child to finish

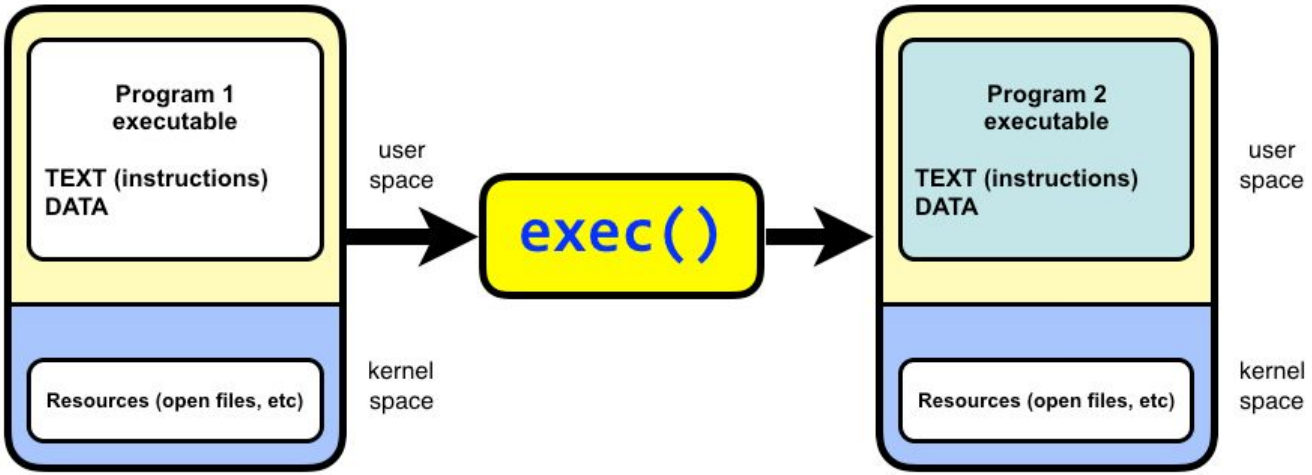


System Calls –exec family

Executing a new program

- After fork, parent and child are running same code
  - Not too useful!
- A common use of fork is to launch a new executable program
- A process can run exec() to load another executable to its memory image
  - So, a child can run a different program from parent
  - It can also be shell script
- The exec system call replaces the current process image with a new image
  - *If exec succeeds, it never returns*
- exec requires you to specify the file you program to run

exec() system call



- The exec family of system calls replaces the program executed by a process
- When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program
- All open file descriptors remains open after calling exec
  - unless explicitly set to close-on-exec

The exec( ) System Call

- There's no a syscall under the name `exec()`. By `exec()` we usually refer to a family of calls:
  - `int execl(char *path, char *arg, ...);`
  - `int execv(char *path, char *argv[]);`
  - `int execle(char *path, char *arg, ..., char *envp[]);`
  - `int execve(char *path, char *argv[], char *envp[]);`
  - `int execlp(char *file, char *arg, ...);`
  - `int execvp(char *file, char *argv[]);`

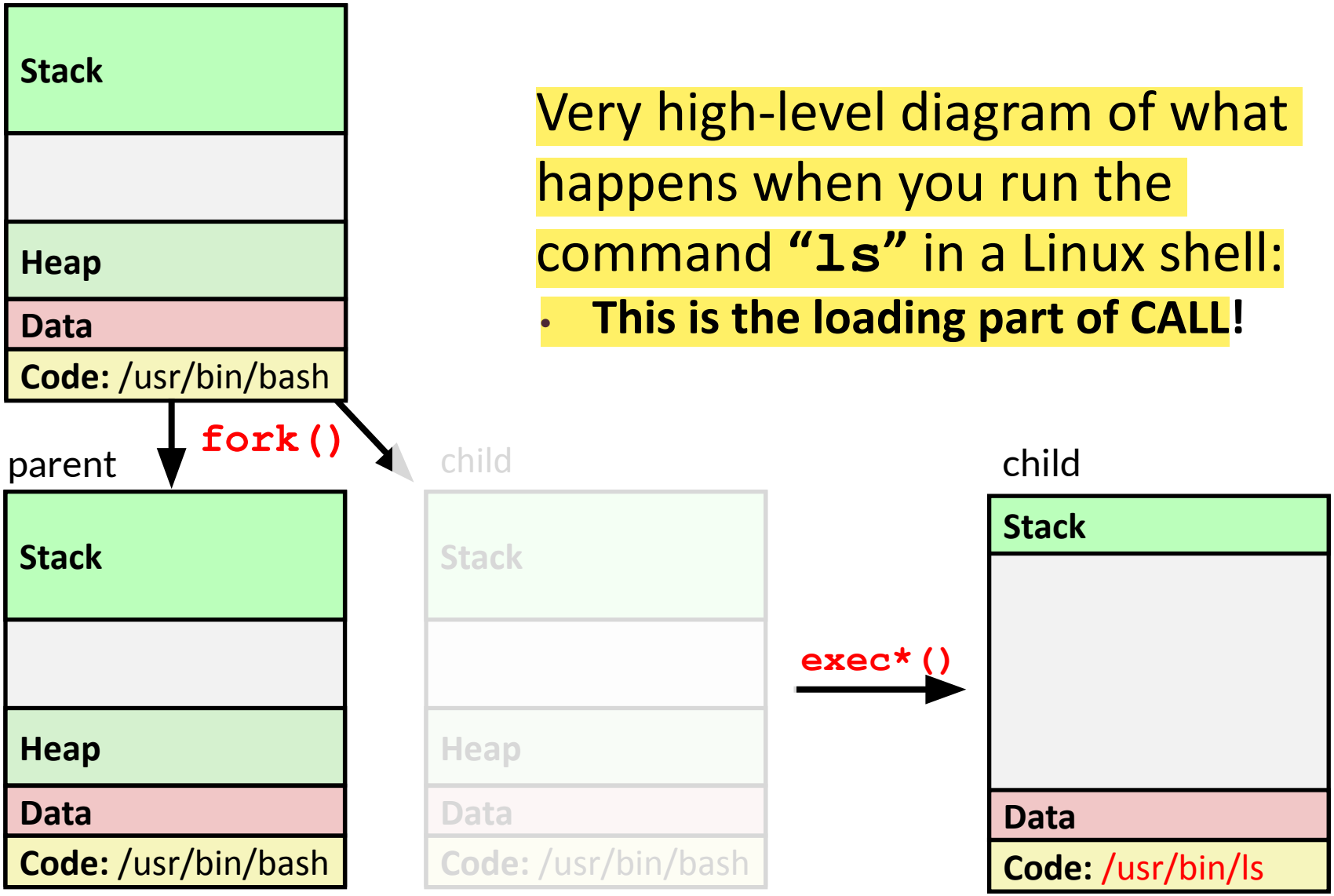
- Here's what *l*, *v*, *e*, and *p* mean:
  - ***l*** means an argument list,
  - ***v*** means an argument vector,
  - ***e*** means an environment vector, and
  - ***p*** means a search path.

execvp() system call

```
#include <unistd.h>
int execvp(const char *file, char *const argv[]);
```

- *file:* Used to construct a pathname that identifies the new process image file
  - If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file
- *argv:* An array of character pointers to NULL-terminated strings
  - These strings constitute the argument list available to the new process image
  - *argv[0]* must point to a filename that's associated with the process being started
  - the last member of this array is a NULL pointer

Exec-ing a new program



Very high-level diagram of what happens when you run the command "`ls`" in a Linux shell:

- **This is the loading part of CALL!**

p5.c

Example: Using execvp( )

```
1
2  #include <unistd.h> // execvp()
3  #include <stdio.h>  // perror()
4  #include <stdlib.h> // EXIT_SUCCESS, EXIT_FAILURE
5
6  int main(void) {
7    char *const cmd[] = {"ls", "-l", NULL};
8    execvp(cmd[0], cmd);
9    perror("Return from execvp() not expected");
10   exit(EXIT_FAILURE);
11 }
12
```

```
(base) Ravis-MacBook-Pro-2:cp ravimittal$ ./p5
total 536
-rw-r--r--@ 1 ravimittal  staff      42  6 Sep 10:50 clients.txt
-rw-r--r--@ 1 ravimittal  staff  142187  5 Sep 21:33 fig11_02.c
-rwxr-xr-x  1 ravimittal  staff   12784  3 Sep 09:12 p1
-rw-r--r--@ 1 ravimittal  staff     486  3 Sep 08:24 p1.c
-rwxr-xr-x  1 ravimittal  staff   12872  2 Sep 21:02 p2
-rw-r--r--  1 ravimittal  staff     655 30 Aug 12:11 p2.c
-rwxr-xr-x  1 ravimittal  staff   13044 30 Aug 13:21 p3
-rw-r--r--@ 1 ravimittal  staff     968 31 Aug 09:26 p3.c
-rw-r--r--@ 1 ravimittal  staff     262 10 Aug 16:00 p4.c
-rwxr-xr-x  1 ravimittal  staff   12644  6 Sep 22:07 p5
-rw-r--r--@ 1 ravimittal  staff     268 30 Aug 13:14 p5.c
-rwxr-xr-x  1 ravimittal  staff   12644 30 Aug 13:14 pexe
-rwxr-xr-x  1 ravimittal  staff   13036  5 Sep 22:17 pf
-rw-r--r--@ 1 ravimittal  staff     281  5 Sep 21:50 pfile1.c
-rw-r--r--@ 1 ravimittal  staff     933  5 Sep 22:17 pfile2.c
```

p3.c

```c
#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 (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
```

Program Output

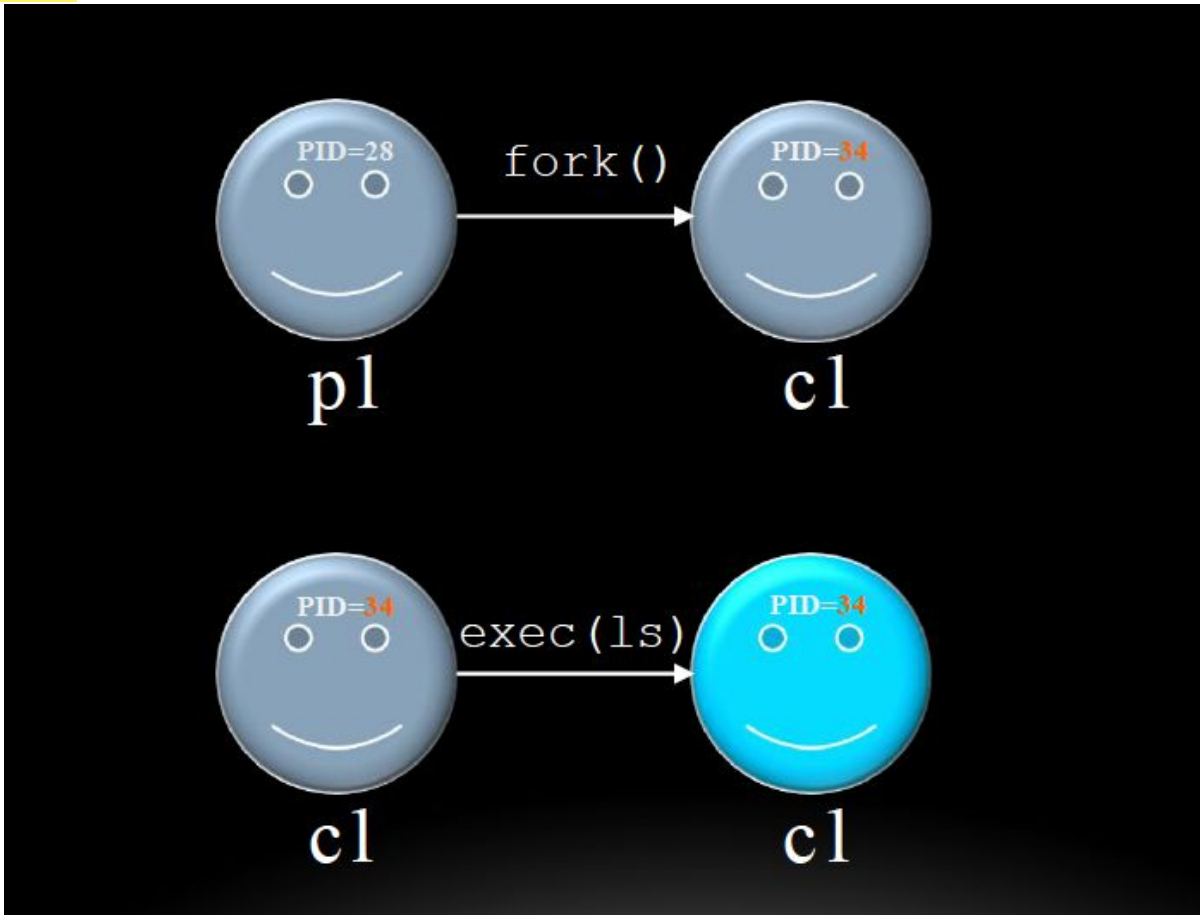
```
hello world (pid:25155)
hello, I am child (pid:25156)
      32     123     966 p3.c
hello, I am parent of 25156 (wc:25156) (pid:25155)
```

exec() – more info

- Upon success, exec() never returns to the caller
- A successful exec replaces the current process image, so it cannot return anything to the program that made the call
- If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions
- As a new process is not created, the process identifier (PID) does not change
  - However, machine code, data, heap, and stack of the process are replaced by those of the new program

fork() and exec() combined

- Often after doing fork() we want to load a new program into the child



Zombies

- A terminated process still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid of 1)
    - **Note:** on recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (*e.g.* shells, servers) we need *explicit* reaping

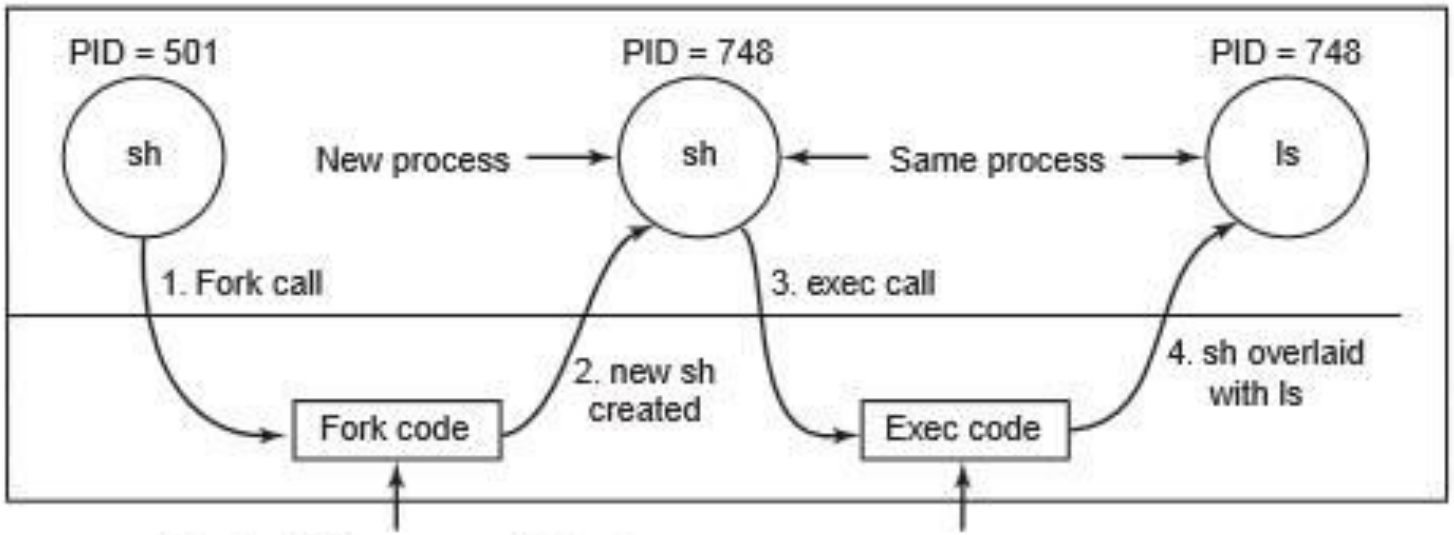
Zombie State

- Why keep process descriptor around?
  - Parent may be waiting for child to terminate
    - via the *wait()* system call
  - Parent needs to get the exit code of the child
    - this information is stored in the descriptor (PCB)
  - If descriptor was destroyed immediately, this information could not be gotten
  - After getting this information, the process descriptor (PCB) can be removed
    - no more remnants of the process

Class Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Two-process program:
    - First `fork()`
    - **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }
  - Two different programs:
    - First `fork()`
    - **if** (pid == 0) { execv(…) } **else** { */* parent code */* }
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

fork() AND exec() combined (1/2)



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

exec() – More info

- The exec()call replaces a current process' image with a new one (i.e. loads a new program within current process)
- The new image is either regular executable binary file or a shell script
- There is not a syscall under the name exec(). By exec()we usually refer to a family of calls:
  - int execl(char *path, char *arg, ...);
  - int execv(char *path, char *argv[]);
  - int execle(char *path, char *arg, ..., char *envp[]);
  - int execve(char *path, char *argv[], char *envp[]);
  - int execlp(char *file, char *arg, ...);
  - int execvp(char *file, char *argv[]);
- Here's what l, v, e, and p mean:
  - **l** means an argument list,
  - **v** means an argument vector,
  - **e** means an environment vector, and
  - **p** means a search path.