CS310 Operating Systems

Lecture 7: Process – fork() system call

Ravi Mittal
IIT Goa

Acknowledgements !

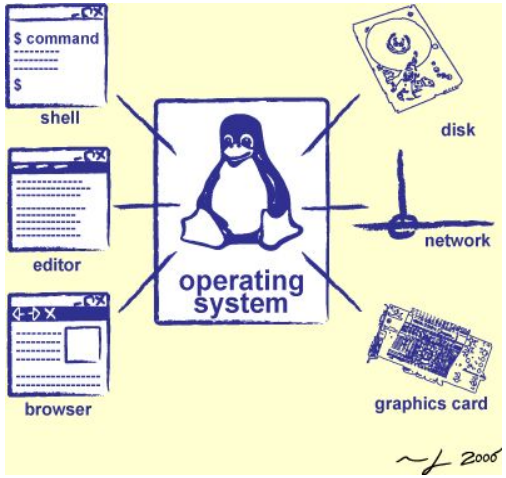
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Class presentation: University of California, Berkeley: David Culler, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion 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

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

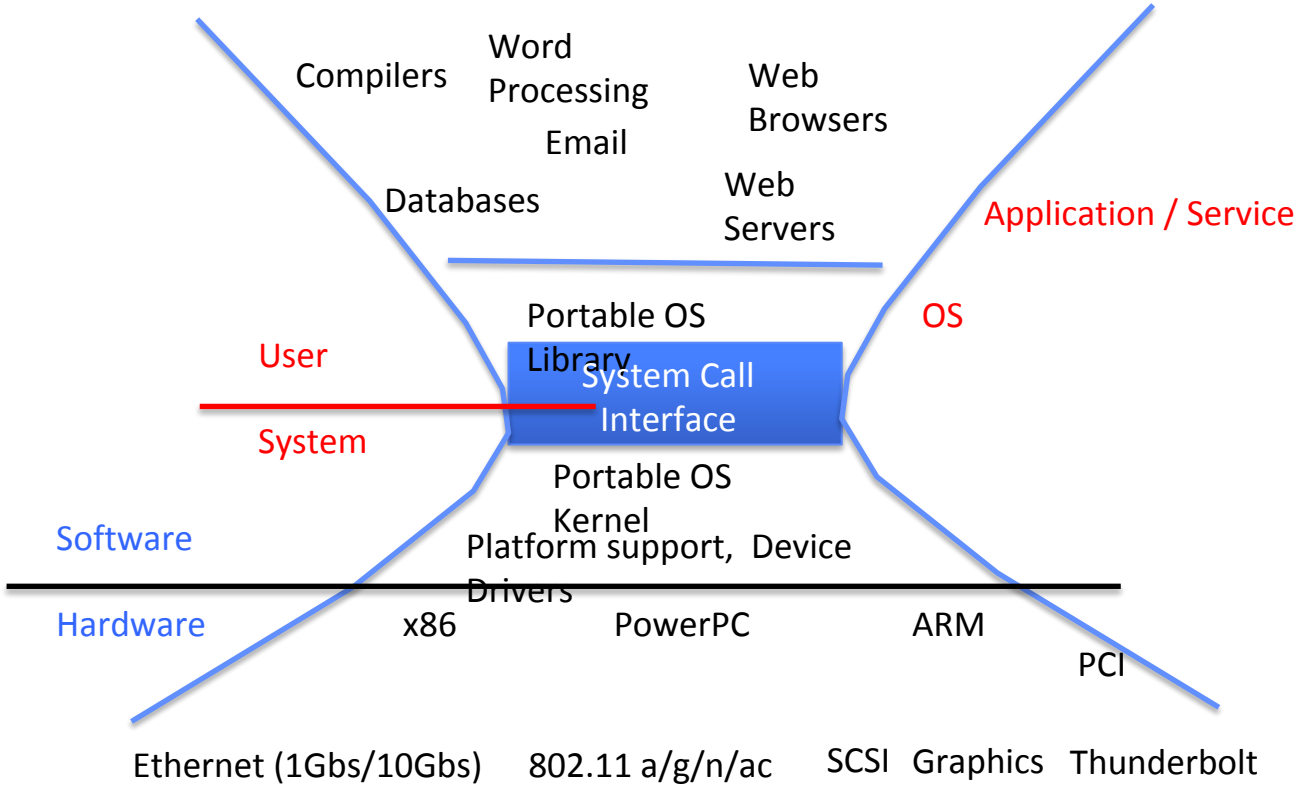
We will study..

- Fork() system call

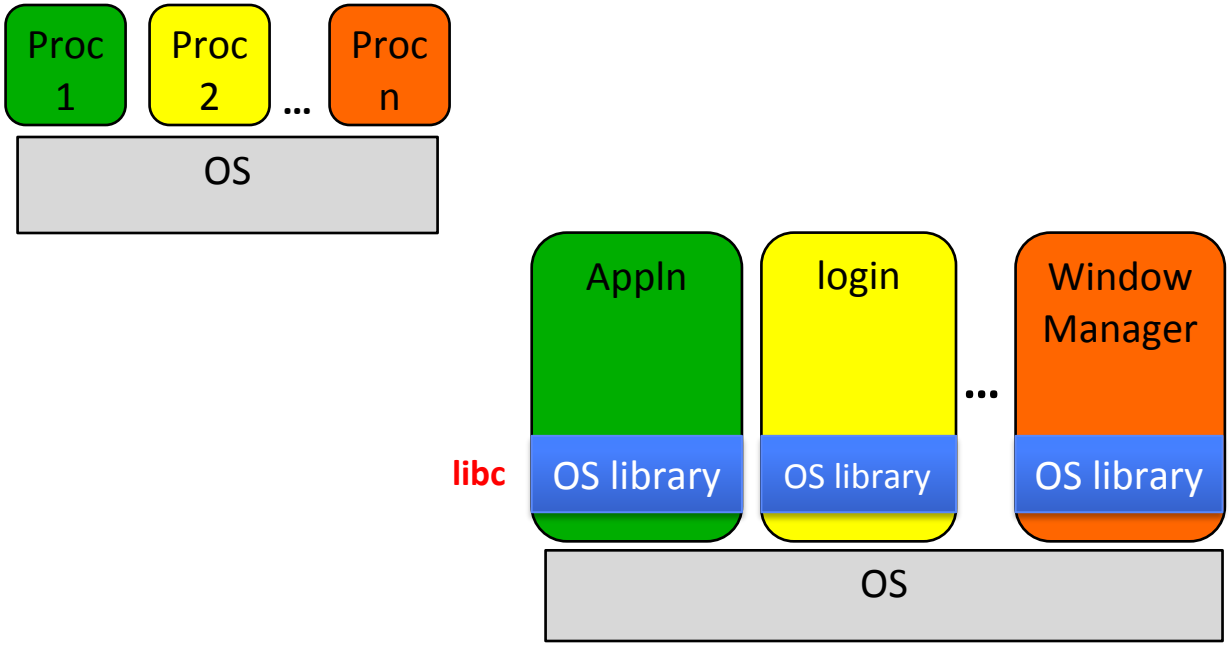


We have studied so far …

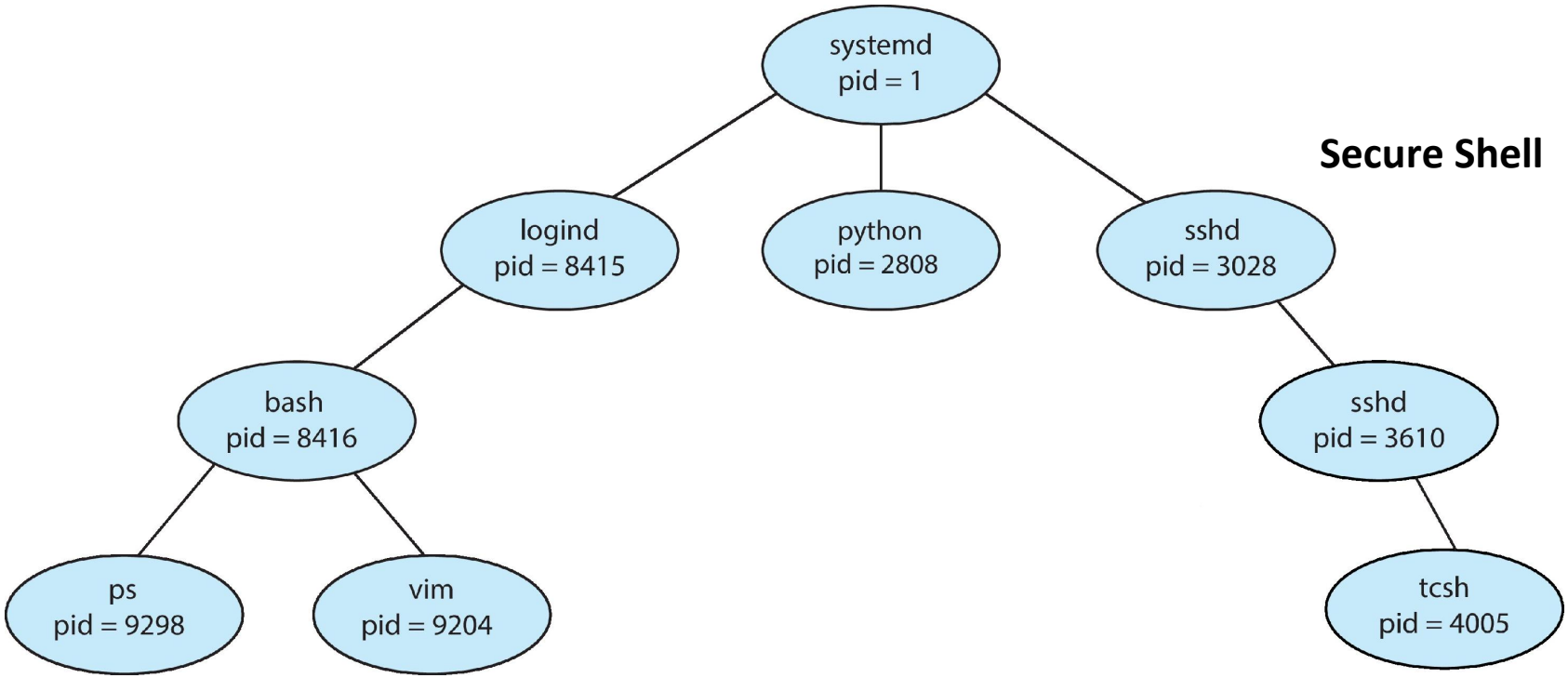
System Calls ("Syscalls")



OS Library Issues Syscalls



A Tree of Processes in Linux



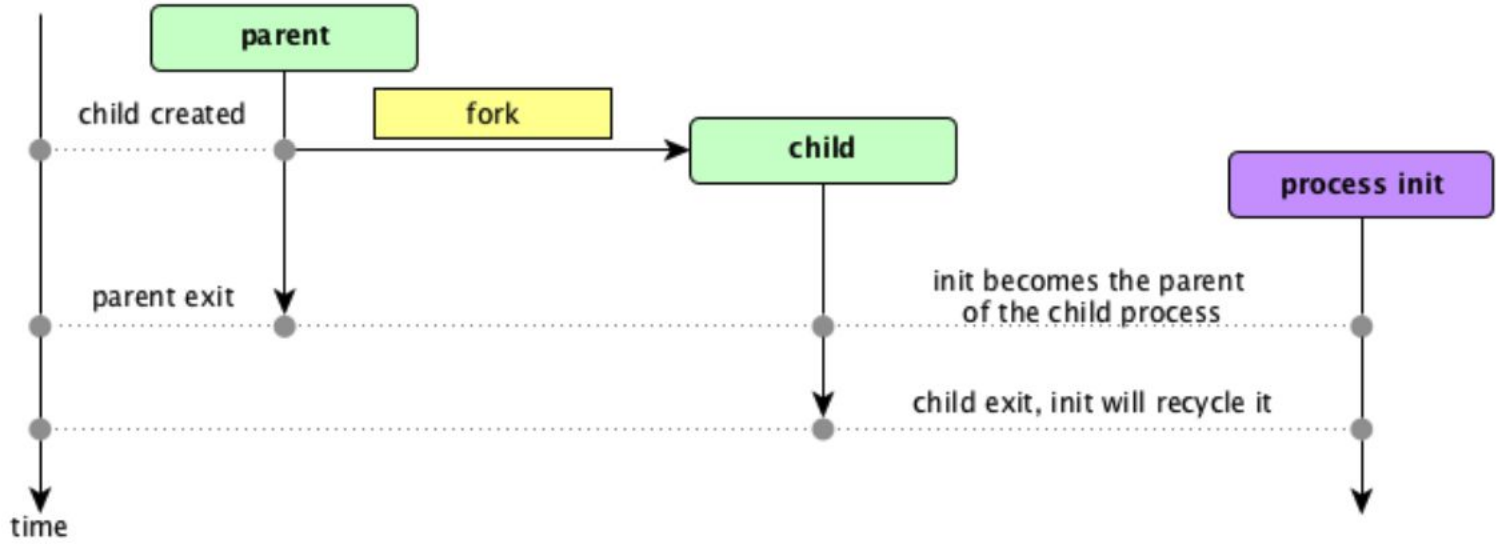
ps -el command gives details of all active processes in the system

Process Creation

- During execution, a process may create several new processes
  - Creating Process: Parent Process
  - New Processes: Children Processes
- Each of the children process may create other processes
  - Forming Tree of processes
- The `systemd` process (which always has a pid of 1) serves as the root parent process for all user processes
  - `Systemd` process in Unix is called `init` process

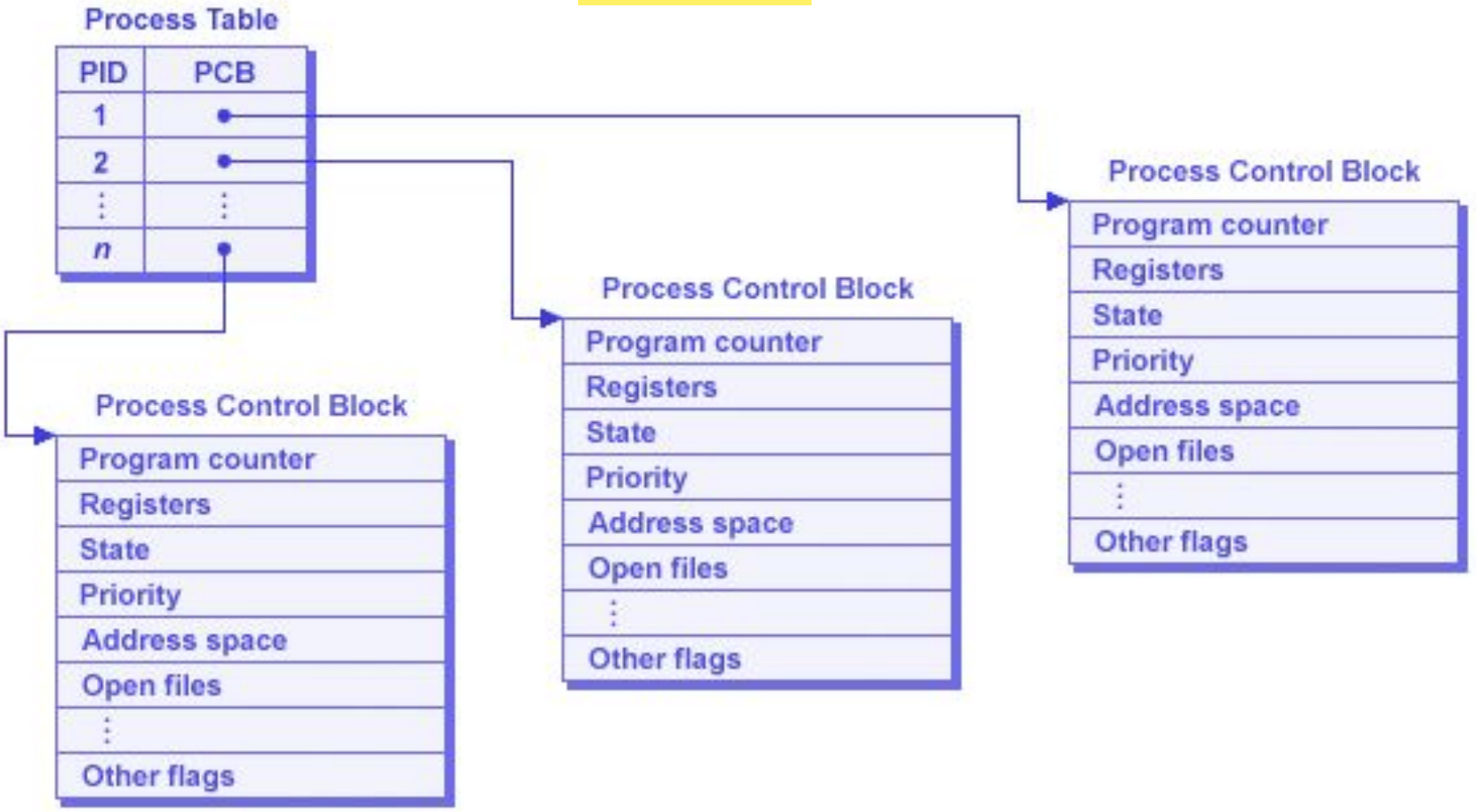
Process creation and termination – snapshot - 1

In case the parent dies, `init` becomes parent of the child



Another Data structure – Process Table

Used in Linux



System Calls

Process Related System Calls (in Unix)

- 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

System Calls – fork()

Creating a Process (Linux)

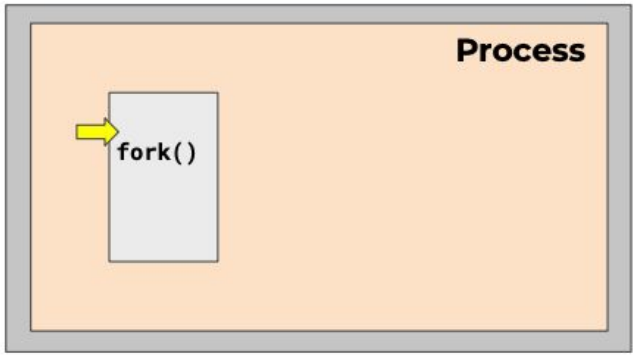
- All processes are created using fork system call
  - Creates an exact copy of the current process
  - Both processes continue in parallel from the statement that follows the fork call
  - The only difference is in the return value
    - Parent: Child process ID (“pid” , non–zero)
    - Child: 0
      - Child can get parent ID via getppid()
    - Failure: -1

Actions during a fork

- A new process is created by making a copy of parent's memory image
  - In a separate address space
- The new process is added to the process list and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify the memory data independently
- Fork copies variables and registers from the parent to the child

Creating a process

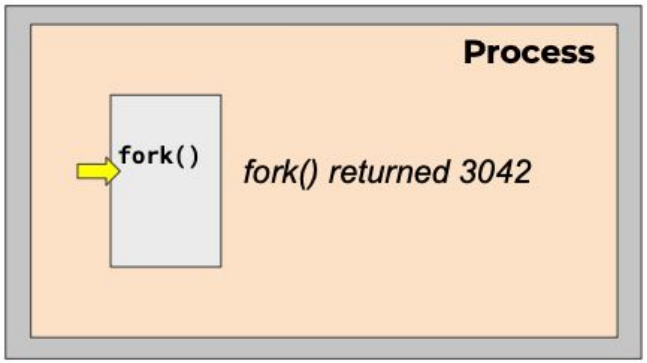
- Just one process – Parent process



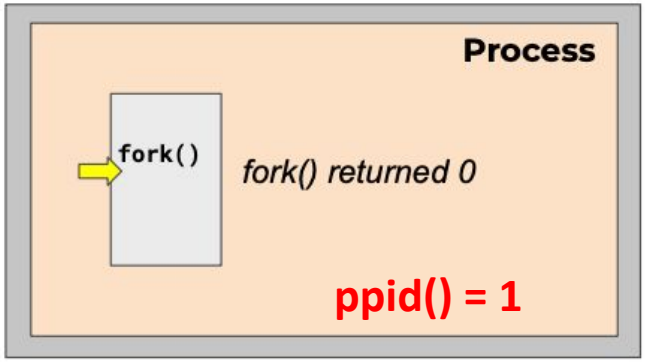
Creating a process

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

id = 1

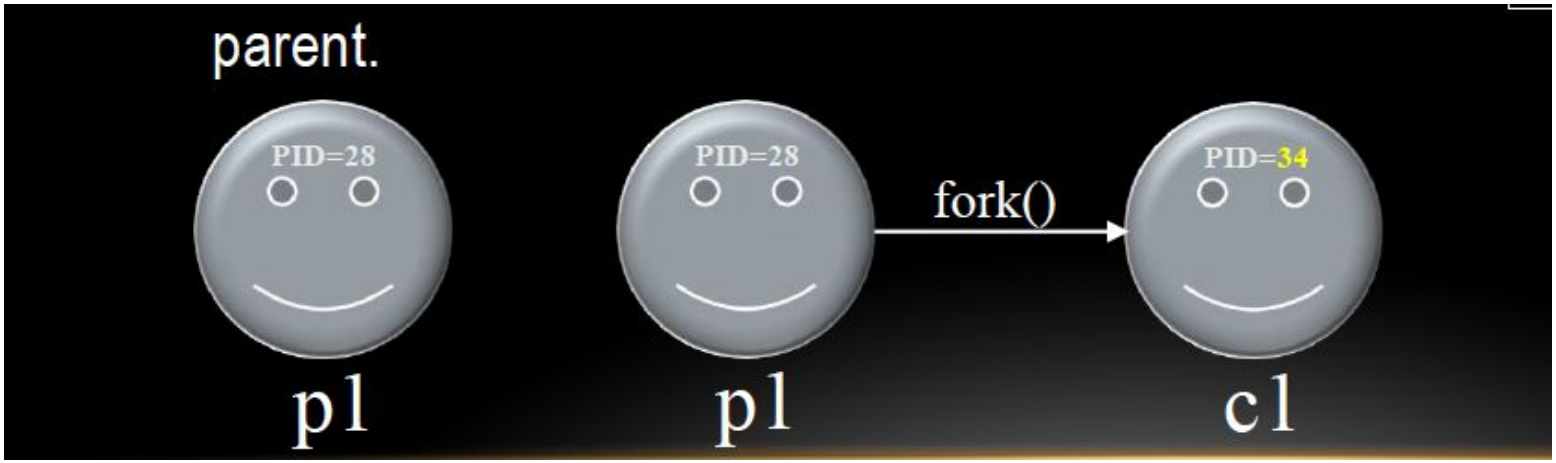


id = 3042



Fork() is fun..

- The fork() is one of the those system calls, which is called once, but returns twice



p1.c

```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
    fprintf(stderr, "fork failed\n");
    exit(1);
} else if (rc ==0) {
    //child new process
    printf("hello I am child process with (pid = %d) \n", (int) getpid());
} else {
    //parent
    printf("hello I am parent of %d (pid: %d) \n", rc, (int) getpid());
}
return 0;

}
```

```
(base) Ravis-MacBook-Pro-2:Cprograms ravimittal$
./a.out

hello world (pid:22305)
hello I am parent of 22306 (pid: 22305)
hello I am child process with (pid = 22306)
```

Notes on Fork()

- Note that the child isn't an exact copy
  - It has it's own copy of the address space
  - It's own registers, PC, etc
  - Value it returns is different from it's parent
- The output of p1.c is not deterministic
  - After child process is created – there are two active processes
  - On a single CPU, it depends on scheduler which one gets scheduled first. For example, in another run we might get:

```
Ravis-MacBook-Pro-2:Cprograms ravimittal$ ./a.out
hello world (pid:22394)
hello I am child process with (pid = 22395)
hello I am parent of 22395 (pid: 22394)
```

Class Summary

- fork() and exec() combination is a powerful way to create an manipulate processes
- In the next class we will study exec() system call

Next Class

System Calls – **exit, wait and exec**

System Calls – exit() and wait()

Process termination

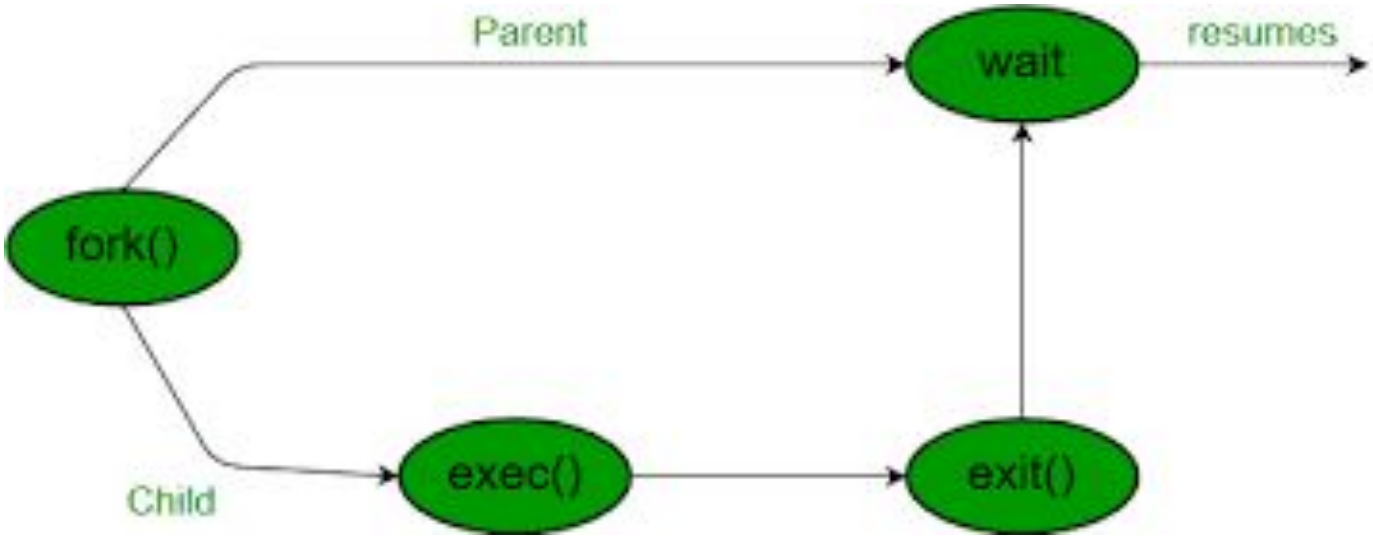
- Multiple ways for a process to get destroyed
  - Process issues and *exit()* call – Voluntary
  - The parent process issues a *kill()* call - Involuntary
  - Process receives a terminate signal - Involuntary
    - When it did something illegal !
- On death
  - Reclaim all of process's memory regions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

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
  - If descriptor was destroyed immediately, this information could not be gotten
  - After getting this information, the process descriptor can be removed
    - no more remnants of the process

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

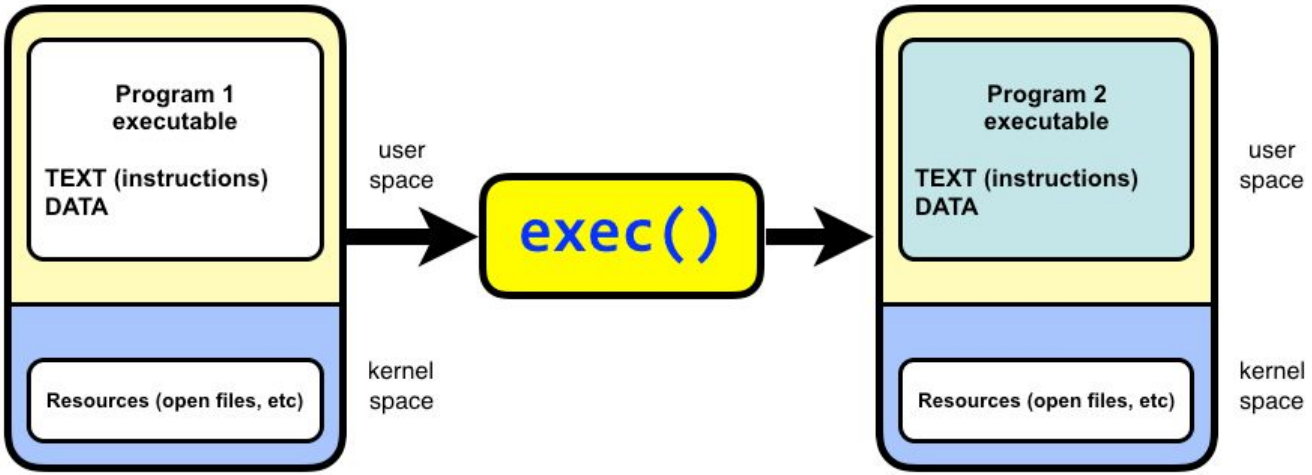
Parent process waits for the child to finish

```
(base) Ravis-MacBook-Pro-2:cp ravimittal$ ./p2
hello world (pid:13858)
hello, I am child (pid:13859)
hello, I am parent of 13859 (wc:13859) (pid:13858)
```

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
- 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

exec() system call

- We consider system call: execvp()
- execvp system call requires two arguments
  - The first argument is a character string that contains the name of a file to be executed
  - The second argument is a pointer to an array of character strings. More precisely, its type is **char ****

execvp() system call

- When **execvp()** is executed, the program file given by the first argument will be loaded into the caller's address space
  - duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character
- The second argument will be provided to the program. It's argument vector. The first argument points to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer.

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

More on Process APIs

- kill() system call is used to send **signals** to a process
  - To pause, die, and other imperatives
- In shells, certain keystroke combinations are configured to deliver a specific signal to the currently running process
  - control-c sends a SIGINT (interrupt) to the process (normally terminating it)
  - control-z sends a SIGTSTP (stop) signal thus pausing the process in mid-execution
    - Can use bg command to resume