Lecture 3
Processes I

Prof. Yinqian Zhang

Spring 2022

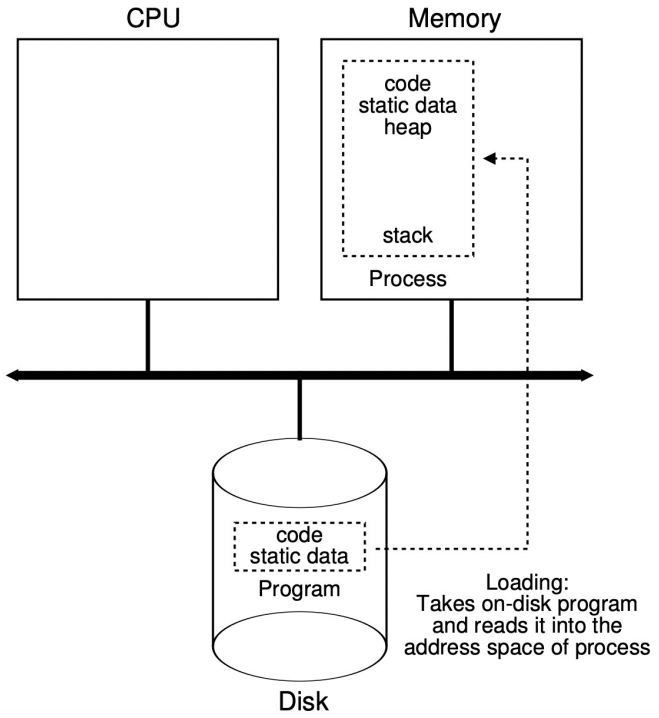
Outline

- Process and system calls
- Process creation

Process and System Calls

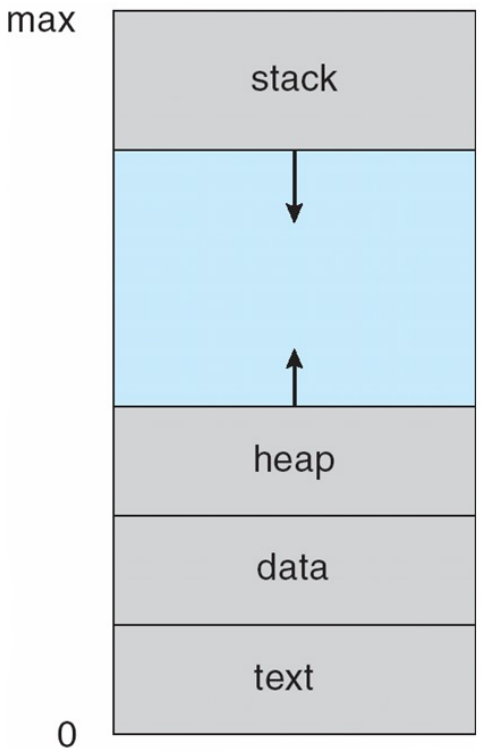
What Is a Process

- Process is a program in execution
- A program is a file on the disk
  - Code and static data
- A process is loaded by the OS
  - Code and static data are loaded from the program
  - Heap and stack are created by the OS



What Is a Process (Cont'd)

- A process is an abstraction of machine states
  - Memory: address space
  - Register:
    - Program Counter (PC) or Instruction Pointer
    - Stack pointer
    - frame pointer
  - I/O: all files opened by the process



Process Identification

- How can we distinguish processes from one to another?
  - Each process is given a unique ID number, and is called the process ID, or the PID.
  - The system call, getpid(), prints the PID of the calling process.

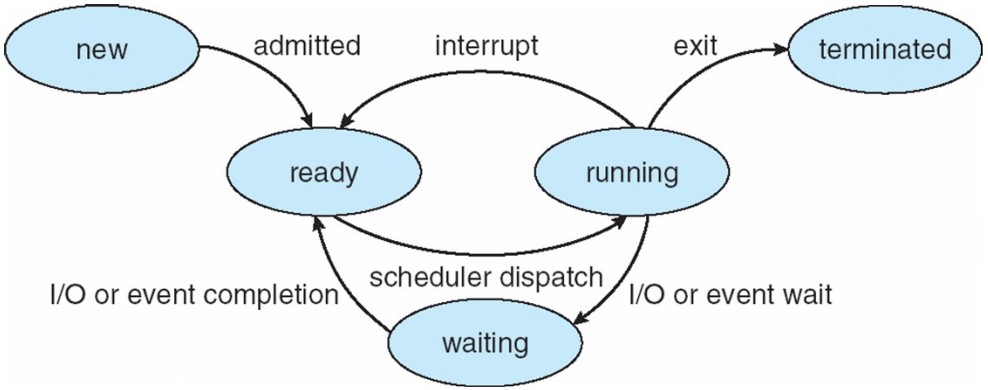
```
// compile to getpid
#include <stdio.h>   // printf()
#include <unistd.h>  // getpid()

int main(void) {
    printf("My PID is %d\n", getpid() );
}
```

```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

Process Life Cycle

```
int main(void) {
    int x = 1;
    getchar();
    return x;
}
```

Process State

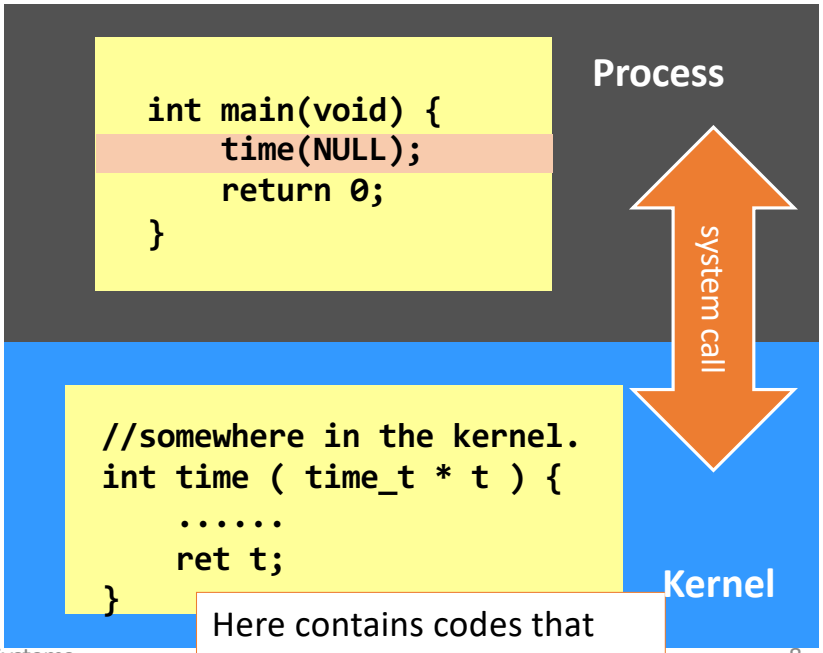


System Call: Process-Kernel Interaction

- System call is a function call.
  - exposed by the **kernel**.
  - abstraction of kernel operations.

```
int add_function(int a, int b) {
    return (a + b);
}

int main(void) {
   int result;
   result = add_function(a,b);
   return 0;
}

// this is a dummy example…
```

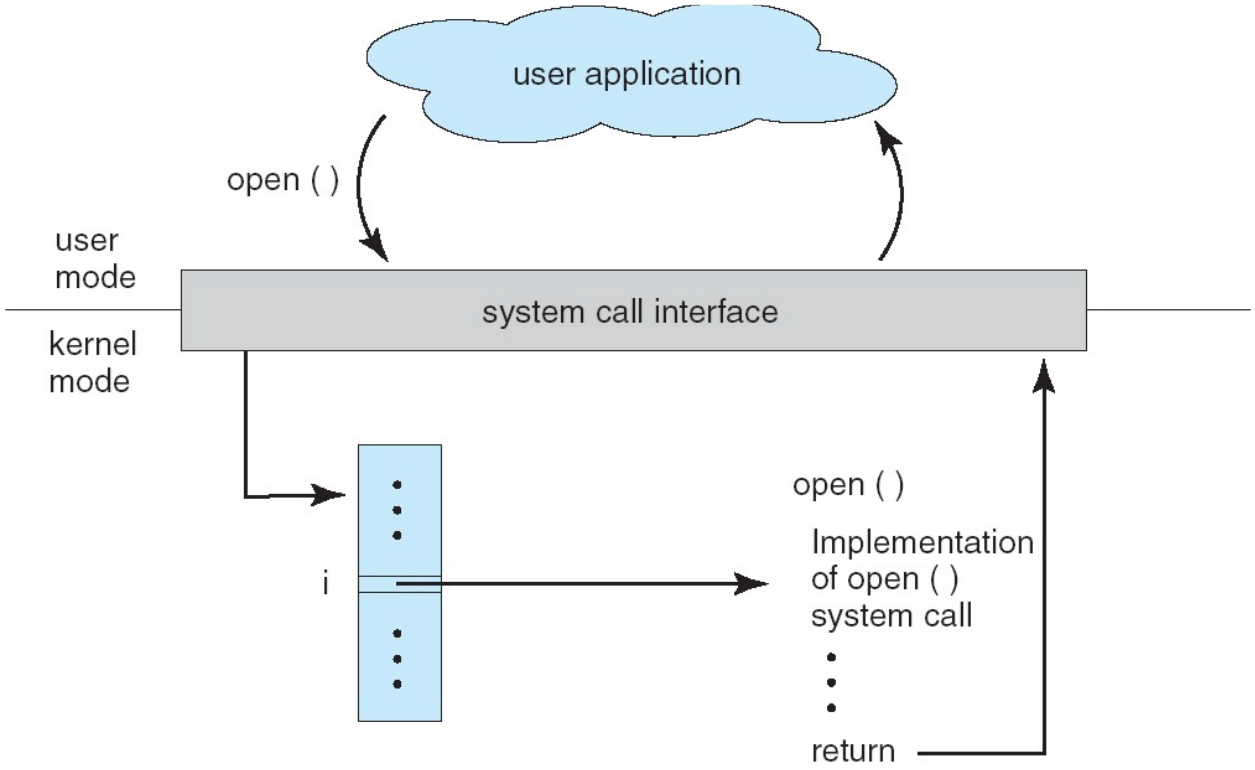
This is a function call.



Here contains codes that access the hardware clock!

System Call: Call by Number

- System call is different from function call
- System call is a call by number



System Call: Call by Number

- User-mode code from xv6-riscv

```
int main(void) {
    ……
    int fd = open("copyin1", O_CREATE|O_WRONLY);
    ……
    return 0;
}
```

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* user/usys.S */
.global open
open:
    li a7, SYS_open
    ecall
    ret
```

System Call: Call by Number

- Kernel code from xv6-riscv

```
/* kernel/syscall.h */

#define SYS_open 15
```

```
/* kernel/file.c */

uint64 sys_open(void) {
        ……
        return fd;
}
```

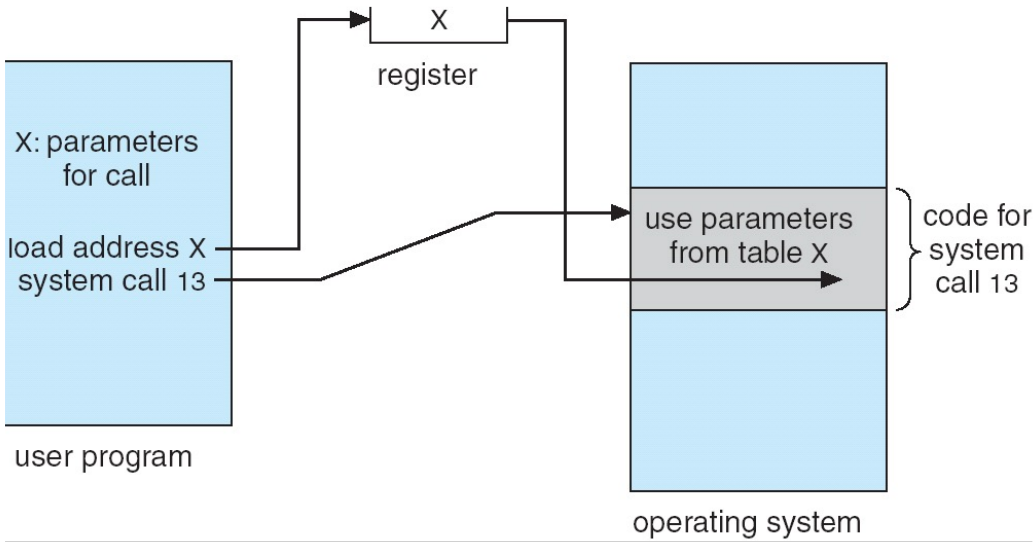
```
/* kernel/syscall.c */

static uint64 (*syscalls[])(void) = {
        …….
        [SYS_open] sys_open,
        …….
}

void syscall(void) (
        struct proc *p = myproc();
        num = p->trapframe->a7;
        p->trapframe->a0 = syscalls[num]();
}
```

System Call: Parameter Passing

- Often, more information is required than the index of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - **Registers**: pass the parameters in registers
    - In some cases, may be more parameters than registers
    - x86 and risc-v take this approach
  - **Blocks**: Parameters stored in a memory block and address of the block passed as a parameter in a register
  - **Stack**: Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
    - Block and stack methods do not limit the number or length of parameters being passed

System Call: Parameter Passing

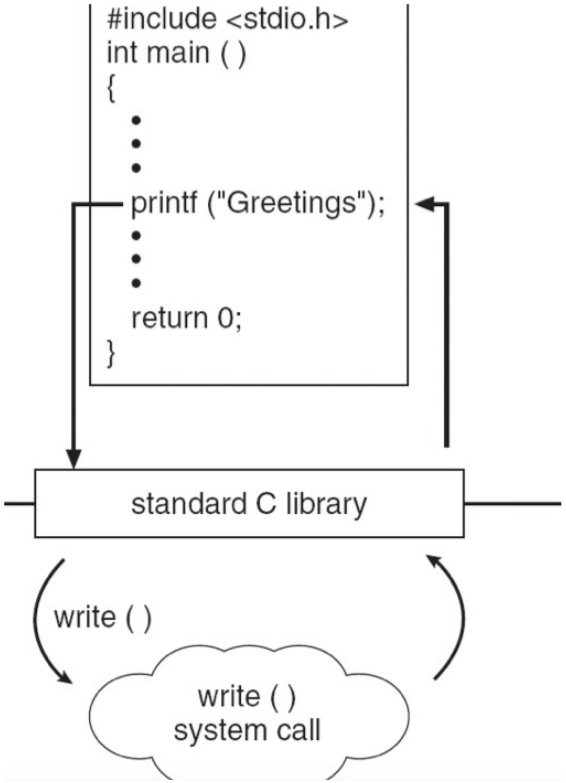
- Example: parameter passing via blocks



System Call v.s. Library API Call

- Most operating systems provide standard C library to provide library API calls
  - A layer of indirection for system calls

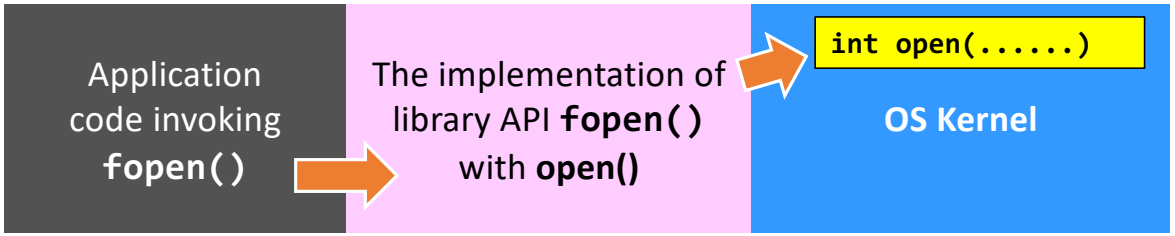
Name	System call?
`printf() & scanf()`	No
`malloc() & free()`	No
`fopen() & fclose()`	No
`mkdir() & rmdir()`	Yes
`chown() & chmod()`	Yes



System Call v.s. Library API Call

- Take **fopen()** as an example.
  - **fopen()** invokes the system call **open()**.
  - **open()** is too primitive and is not programmer-friendly!

Library call	`fopen("hello.txt", "w");`
System call	`open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);`



Process Creation

Process Creation

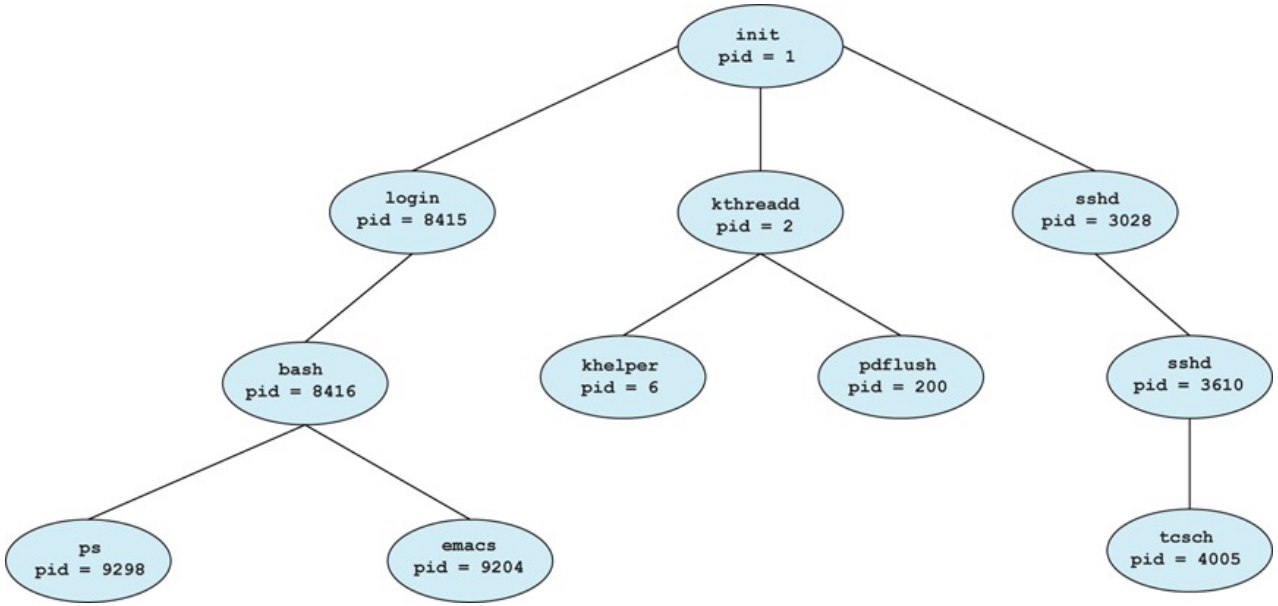
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont'd)

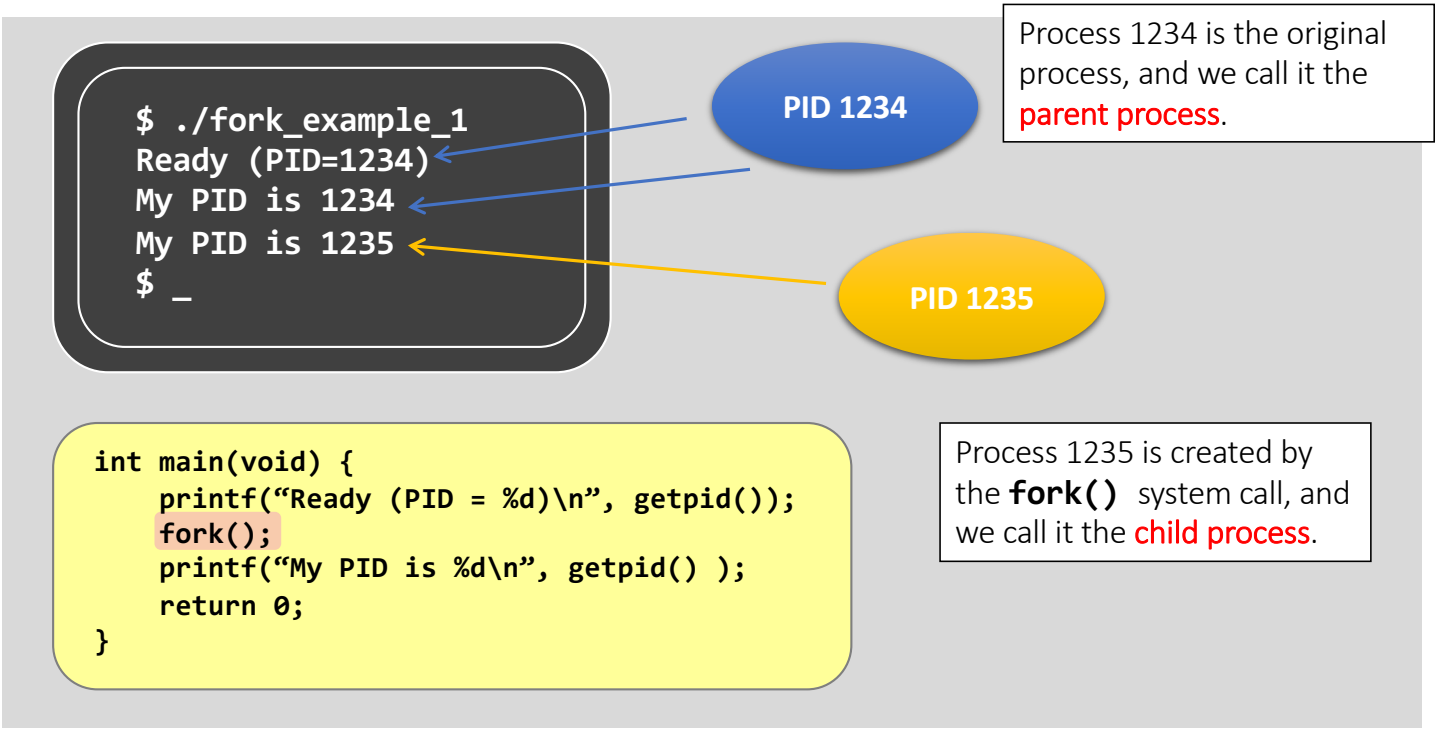
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process
  - exec system call used after a fork to replace the process' memory space with a new program

Process Creation (Cont'd)

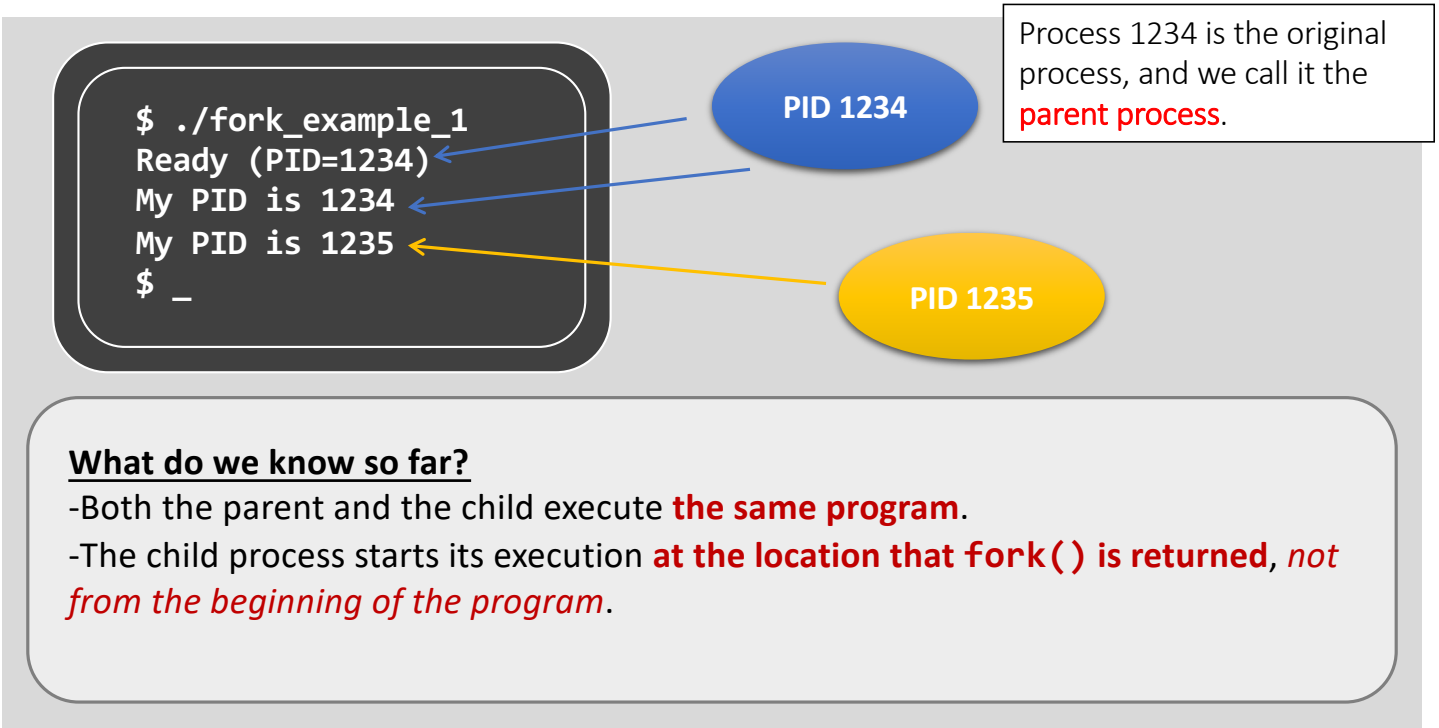
- A tree of processes in Linux



Creating Processes with fork() System Call



Creating Processes with fork() System Call

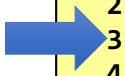


What do we know so far?

-Both the parent and the child execute **the same program**.

-The child process starts its execution **at the location that `fork()` is returned**, *not from the beginning of the program*.

Creating Processes with fork() System Call



```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
```

PID 1234

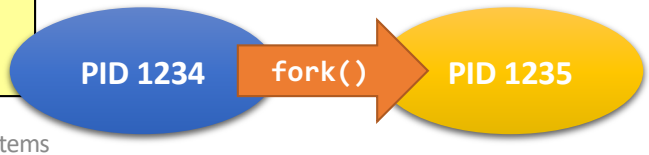
Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

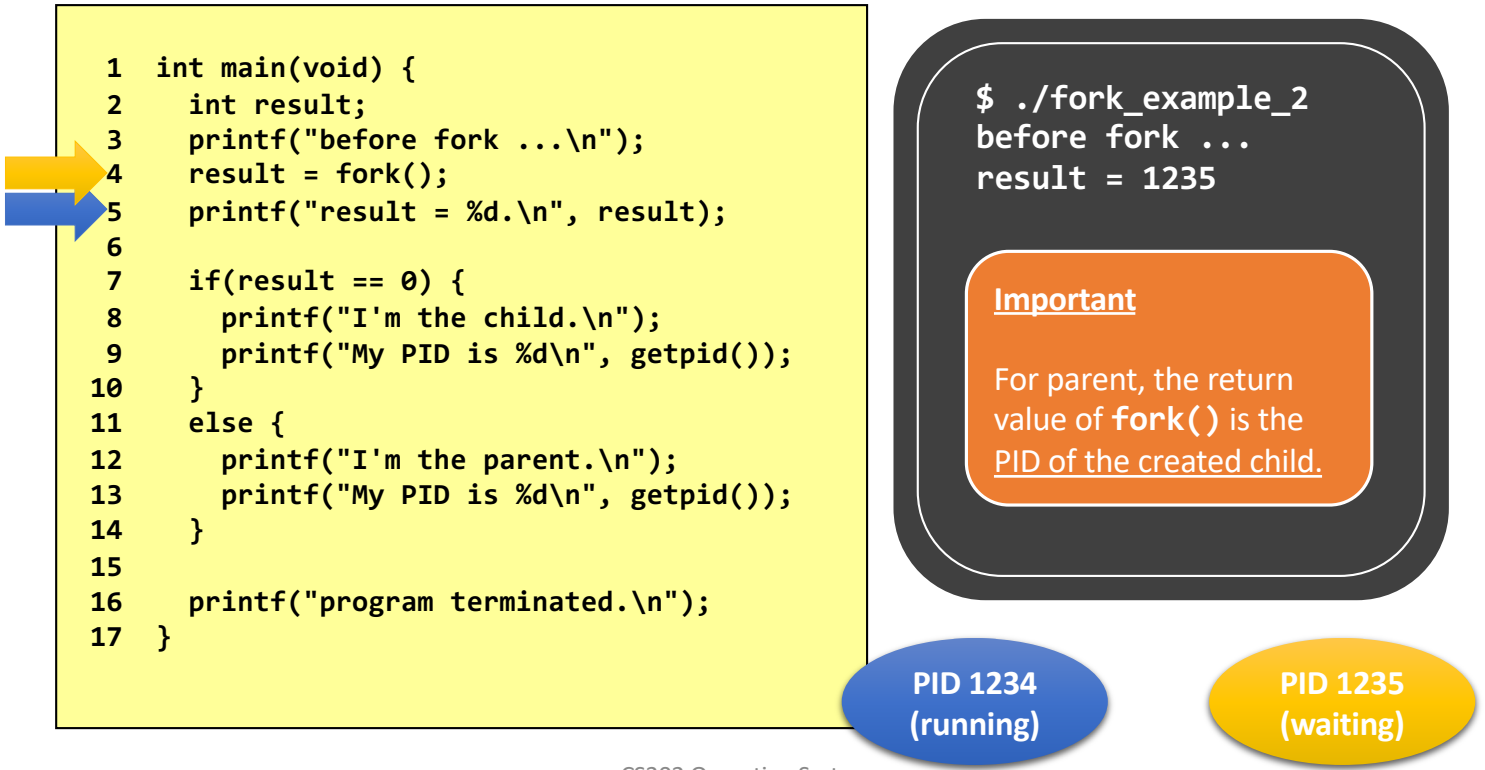
```
$ ./fork_example_2
before fork ...
```

Important

- Both parent and child need to return from fork().
- CPU scheduler decides which to run first.



Creating Processes with fork() System Call



Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
```

PID 1234 (running)

PID 1235 (waiting)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
```

PID 1234 (stop)

PID 1235 (waiting)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
```

Important

For child, the return value of **fork()** is **0**.

PID 1234 (stop)

PID 1235 (running)

Creating Processes with fork() System Call

```
1  int main(void) {
2    int result;
3    printf("before fork ...\n");
4    result = fork();
5    printf("result = %d.\n", result);
6
7    if(result == 0) {
8      printf("I'm the child.\n");
9      printf("My PID is %d\n", getpid());
10   }
11   else {
12     printf("I'm the parent.\n");
13     printf("My PID is %d\n", getpid());
14   }
15
16   printf("program terminated.\n");
17 }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
```

PID 1234 (stop)

PID 1235 (running)

Creating Processes with fork() System Call

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```

PID 1234 (stop)

PID 1235 (stop)

fork() System Call

- `fork()` behaves like "cell division".
  - It creates the child process by **cloning** from the parent process, including all user-space data, e.g.,

Cloned items	Descriptions
Program counter [CPU register]	That's why they both execute from the same line of code after `fork()` returns.
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file "fd", then the child will also have file "fd" opened automatically.

fork() System Call

- `fork()` does not clone the following…

Distinct items	Parent	Child
Return value of `fork()`	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Parent.
Running time	Cumulated.	Just created, so should be 0.
[Advanced] File locks	Unchanged.	None.

fork() System Call

- If a process can only duplicate itself and always runs the same program, it's not quite meaningful
  - how can we execute other programs?

- **exec()**
  - The **exec*()** system call family.

exec()

- **execl()** – a member of the **exec** system call family (execl, execle, execlp, execv, execve, execvp).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$ ./exec_example
before execl ...
```

Arguments of the execl() call

1st argument: the program name, "**/bin/ls**" in the example.
2nd argument: argument[0] to the program.
3rd argument: argument[1] to the program.

exec()

- **execl()** – a member of the **exec** system call family (and the family has 6 members).

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

```
$./exec_example
before execl ...
exec_example
exec_example.c
```

What is the output?

The same as **the output of running "ls" in the shell.**

exec()

- Example #1: run the command "/bin/ls"

```
execl("/bin/ls", "/bin/ls", NULL);
```

Argument Order	Value in above example	Description
1	"/bin/ls"	The file that the programmer wants to execute.
2	"/bin/ls"	When the process switches to "/bin/ls", this string is the **program argument[0]**.
3	NULL	This states the end of the program argument list.

exec()

- Example #2: run the command `"/bin/ls -l"`

```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

Argument Order	Value in above example	Description
1	`"/bin/ls"`	The file that the programmer wants to execute.
2	`"/bin/ls"`	When the process switches to `"/bin/ls"`, this string is the **program argument[0]**.
3	`"-l"`	When the process switches to `"/bin/ls"`, this string is the **program argument[1]**.
4	`NULL`	This states the end of the program argument list.

exec()

- The exec system call family is not simply a function that "invokes" a command.

```
int main(void) {

  printf("before execl ...\n");

  execl("/bin/ls", "/bin/ls", NULL);

  printf("after execl ...\n");

  return 0;
}
```

WHAT?!
The shell prompt appears!

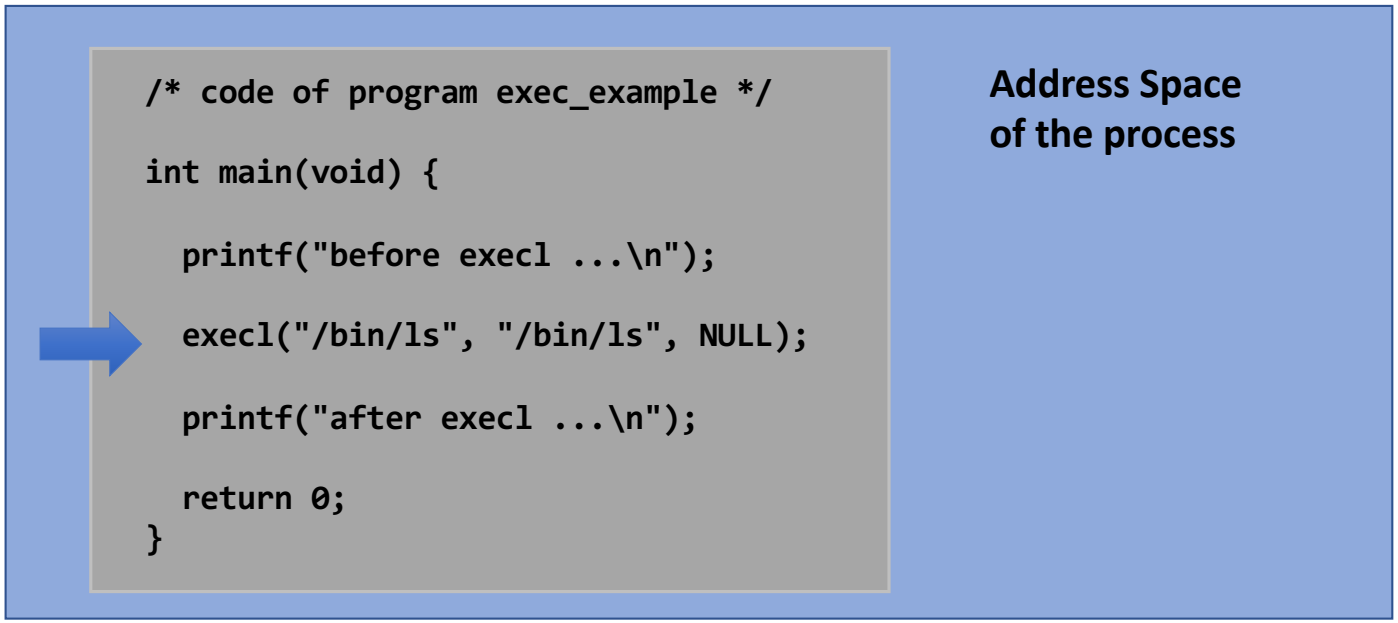
```
$./exec_example
before execl ...
exec_example
exec_example.c
$ _
```

The output says:
(1) The gray code block **is not reached**!
(2) The process is **terminated**!

WHY IS THAT?!

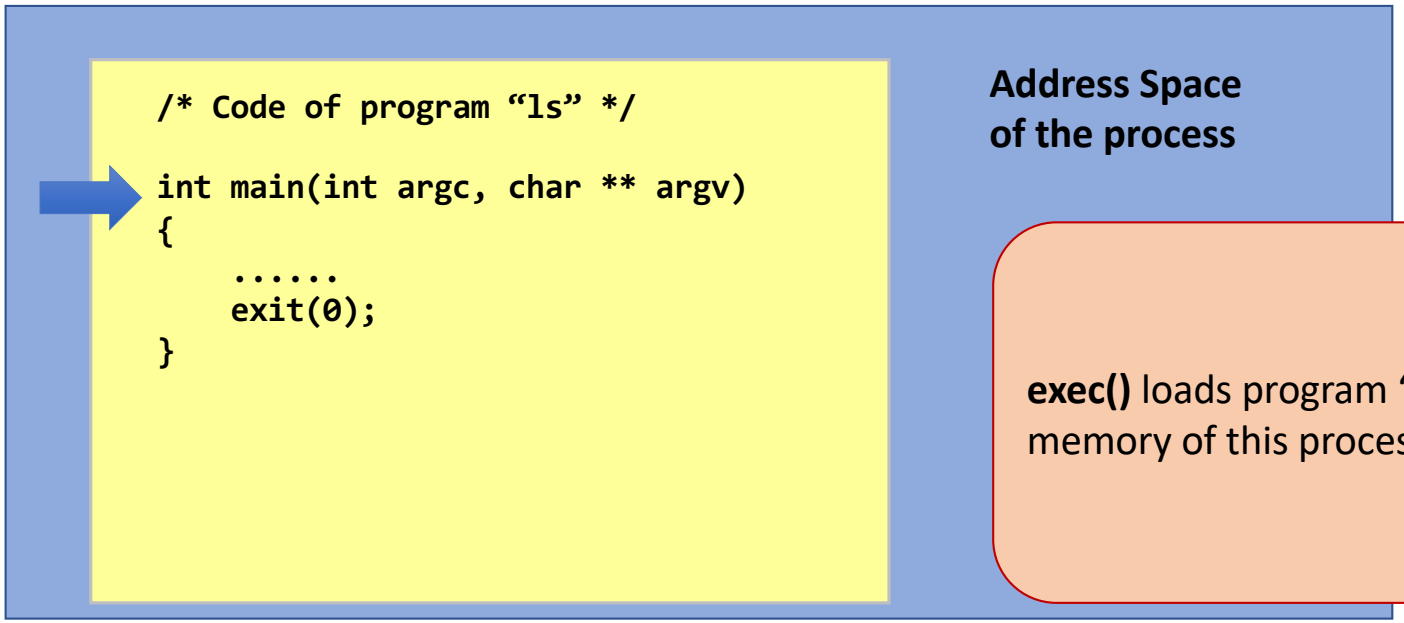
exec()

- The exec system call family is not simply a function that "invokes" a command.



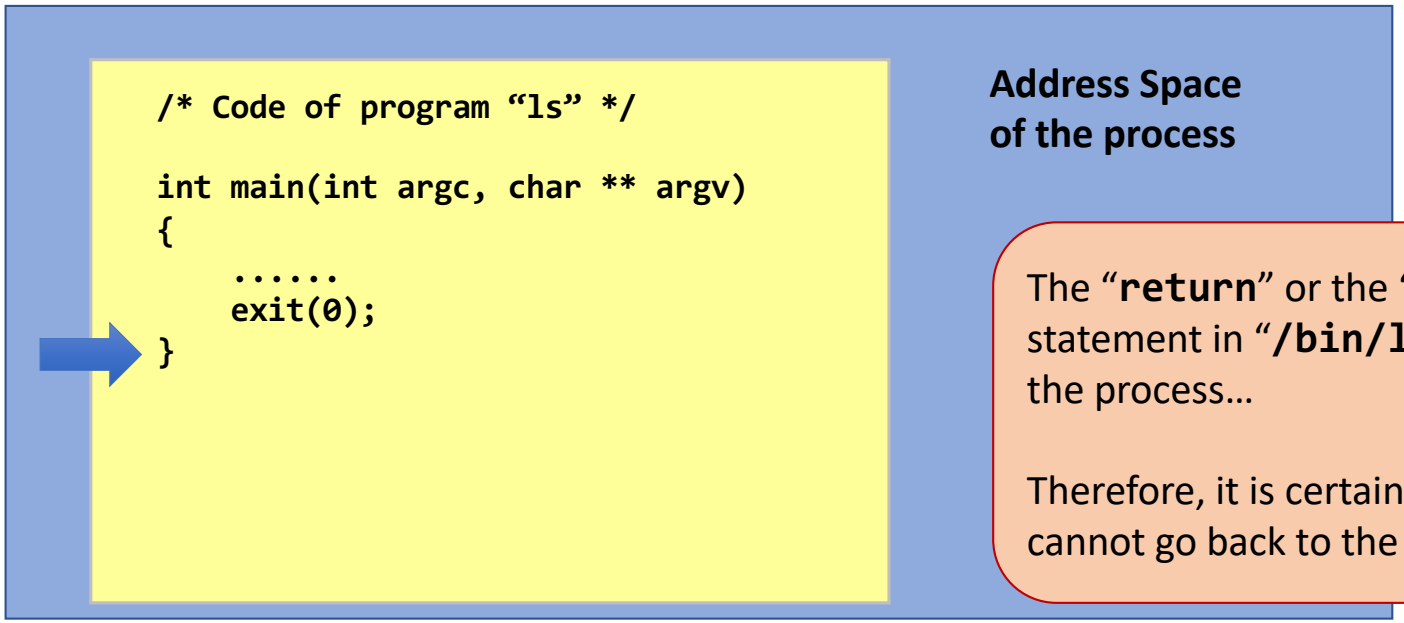
exec()

- The exec system call family is not simply a function that "invokes" a command.



exec()

- The exec system call family is not simply a function that "invokes" a command.



exec() Summary

- The process is changing the code that is executing and never returns to the original code.
  - The last two lines of codes are therefore not executed.
- The process that calls an exec* system call will replace user-space info, e.g.,
  - Program Code
  - Memory: local variables, global variables, and dynamically allocated memory;
  - Register value: e.g., the program counter;
- But, the kernel-space info of that process is preserved, including:
  - PID;
  - Process relationship;
  - etc.

CPU Scheduler and fork()

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11    else {
12      printf("I'm the parent.\n");
13      printf("My PID is %d\n", getpid());
14    }
15
16    printf("program terminated.\n");
17  }
```

Parent return from fork() first

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
My PID is 1234
program terminated.
result = 0
I'm the child.
My PID is 1235
program terminated.
$ _
```

Child return from fork() first

```
$ ./fork_example_2
before fork ...
result = 0
I'm the child.
My PID is 1235
result = 1235
program terminated.
I'm the parent.
My PID is 1234
program terminated.
$ _
```

wait(): Sync Parent with Child

```
 1  int main(void) {
 2    int result;
 3    printf("before fork ...\n");
 4    result = fork();
 5    printf("result = %d.\n", result);
 6
 7    if(result == 0) {
 8      printf("I'm the child.\n");
 9      printf("My PID is %d\n", getpid());
10    }
11   else {
12       printf("I'm the parent.\n");
13      wait(NULL);
14      printf("My PID is %d\n", getpid());
15    }
16
17    printf("program terminated.\n");
18  }
```

Parent return from fork() first

```
$ ./fork_example_2
before fork ...
result = 1235
I'm the parent.
result = 0
I'm the child.
My PID is 1235
program terminated.
My PID is 1234
program terminated.
$ _
```

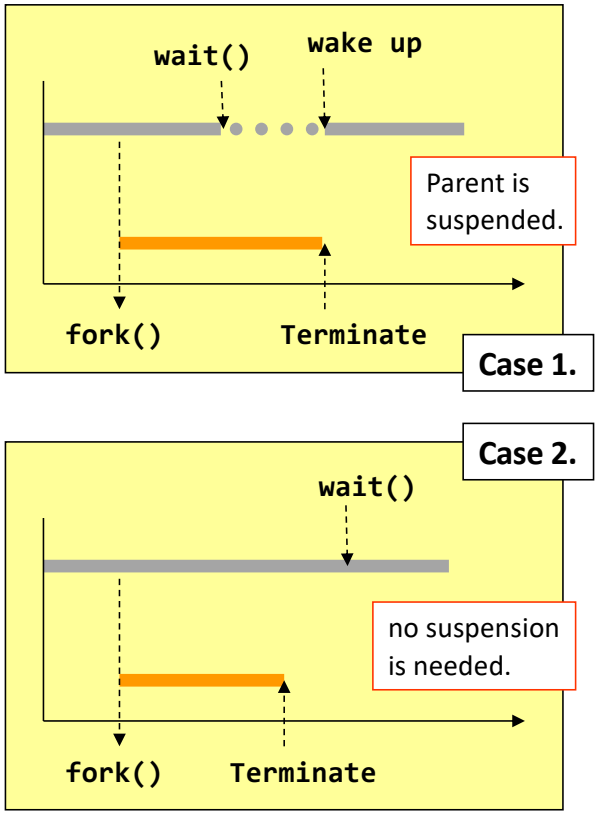
Child return from fork() first

```
$ ./fork_example_2
before fork ...
result = 0
I'm the child.
My PID is 1235
result = 1235
program terminated.
I'm the parent.
My PID is 1234
program terminated.
$ _
```

wait()

- wait() suspends the calling process to waiting
- wait() returns when
  - one of its child processes changes from running to terminated.

- Return immediately (i.e., does nothing) if
  - It has no children
  - Or a child terminates before the parent calls wait for

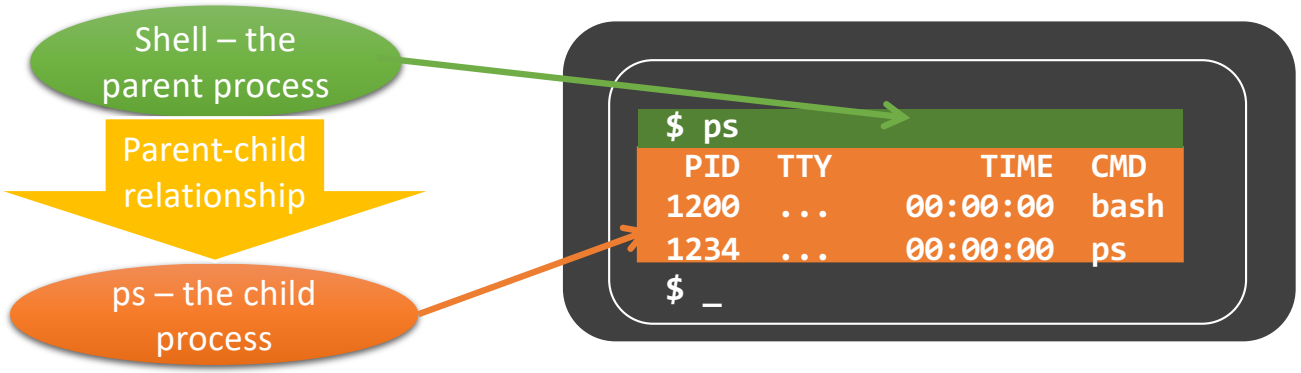


wait() v.s. waitpid()

- wait()
  - Wait for any one of the child processes
  - Detect child termination only
- waitpid()
  - Depending on the parameters, waitpid() will wait for a particular child only
  - Depending on the parameters, waitpid() can detect different status changes of the child (resume/stop by a signal)

Implement Shell with fork(), exec(), and wait()

- A shell is a CLI
  - Bash in linux
  - invokes a function fork() to create a new process
  - Ask the the child process to exec() the target program
  - Use wait() to wait until the child process terminates



Thank you!