Operating Systems

Eric Lo

3 Process (User Space)

System Programming

What is a process?

- Process is a program in **execution**.
  - It contains every accounting information of that running program, e.g.,
    - Current program counter
    - Accumulated running time
    - The list of files that are currently opened by that program
    - The page table
    - …

https://en.wikipedia.org/wiki/Process_control_block

What is a process?

- It will stop early if I send a **signal** to interrupt it.
- Its progress is determined by the **scheduler**.
- Multiple processes can work together to do more complicated tasks

```
$ ls -R /
[Ctrl + C]
$ ls -R / | cat
$ ls -R / | less
$ ls | cat | cat
```

If you don’t know what a **cat** is.

```
#include <stdio.h>

int main(void) {
  int c;
  while ( 1 ) {
    c = getchar();
    if( c == EOF )
      break;
    putchar(c);
  }
}
```

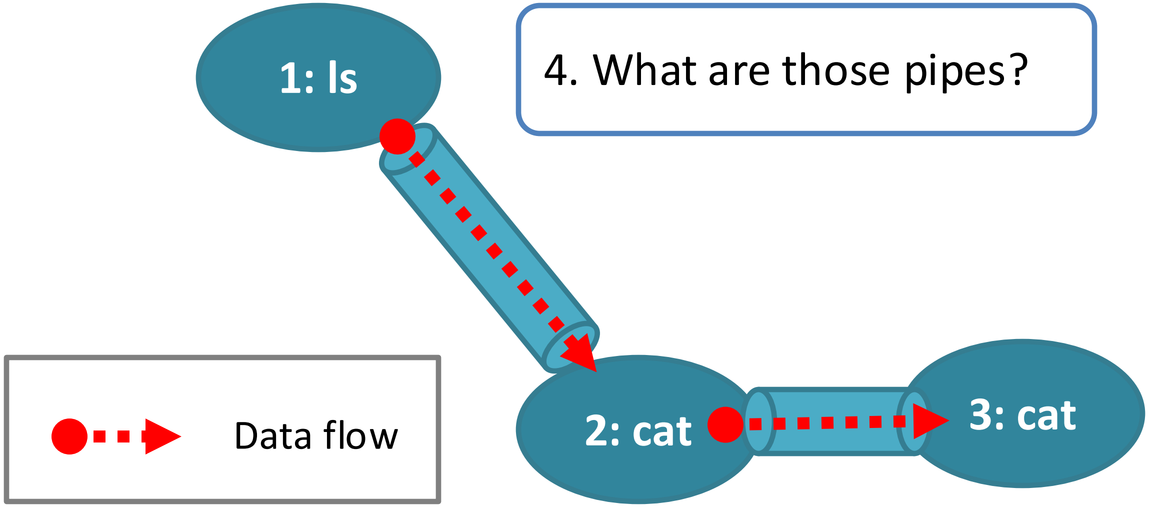
Topics about process

1. How to distinguish the two cats?

2. Who (and how to) create the processes?

3. Which should run first?

4. What are those pipes?



5. What if “**`ls`**” is feeding data too fast? Will the “**`cat`**” feels ***full and dies***?!

Process identification

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

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

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

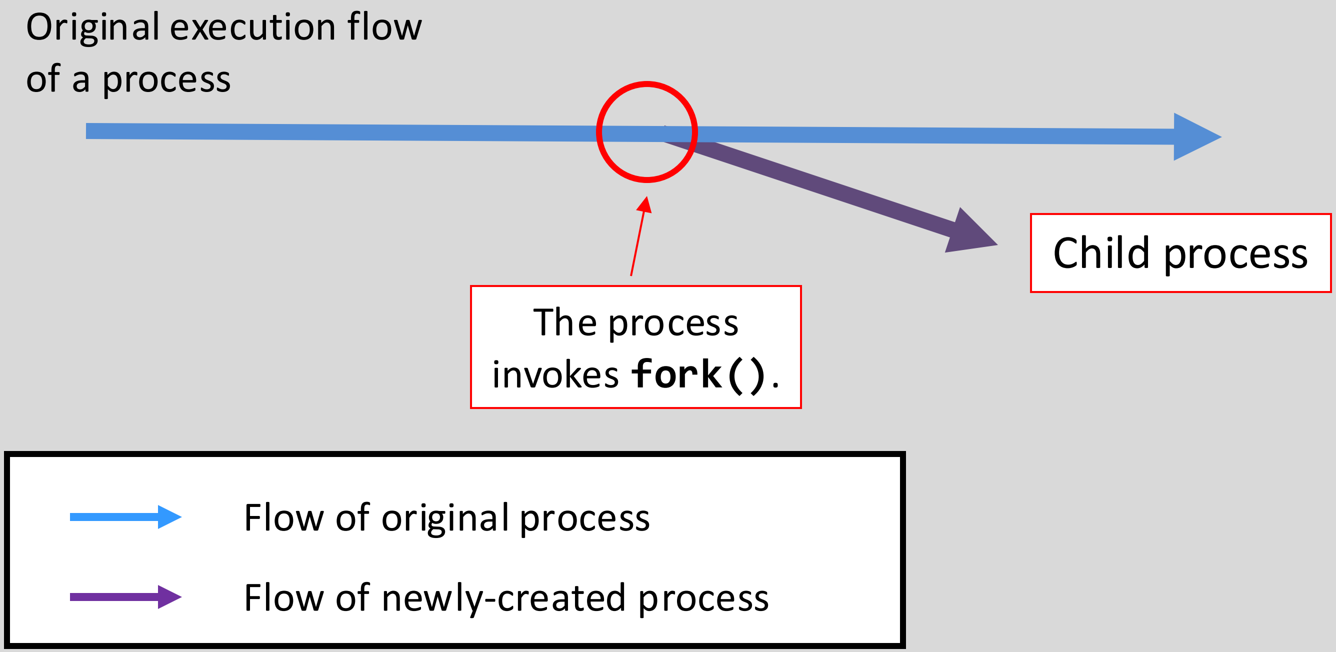
```
$ ./getpid
My PID is 1234
$ ./getpid
My PID is 1235
$ ./getpid
My PID is 1237
```

```
[example@3150]$ cat ProcessUser/getpid.c
```

Process creation

- To create a process, we use the system call **fork()**.

Not this



Original execution flow of a process

The process invokes **fork()**.

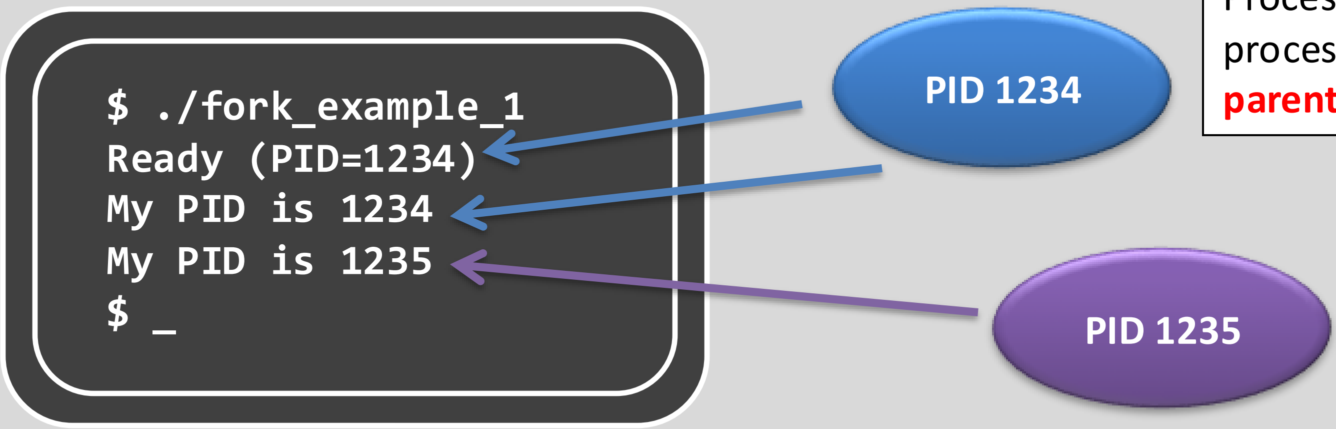
Child process

Flow of original process

Flow of newly-created process

Process creation – **fork()** system call

- So, how do **fork()** and the processes behave?



```
$ ./fork_example_1
Ready (PID=1234)
My PID is 1234
My PID is 1235
$ _
```

Process 1234 is the original process, and we call it the **parent process**.

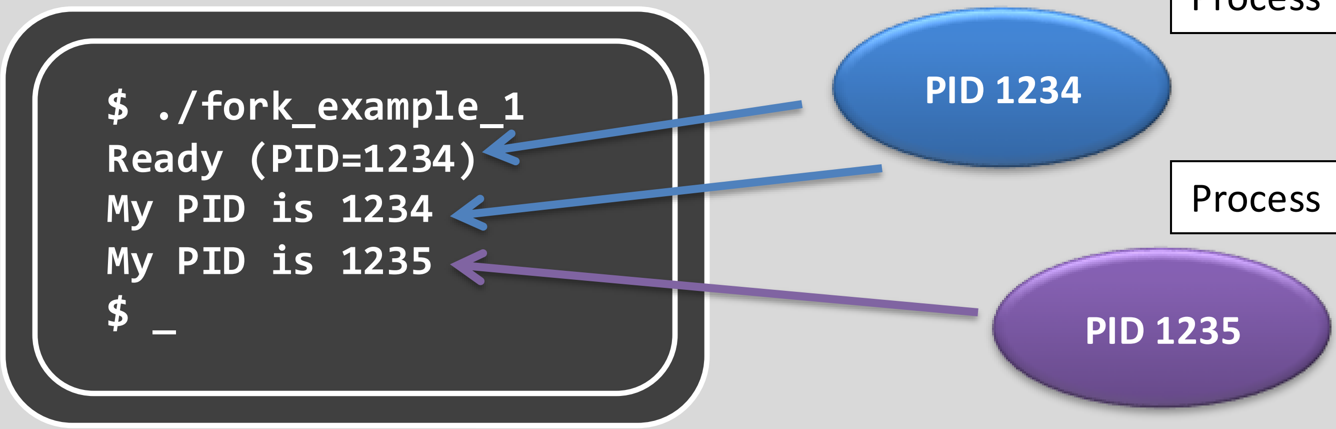
```
int main(void) {
    printf("Ready (PID = %d)\n", getpid());
    fork();
    printf("My PID is %d\n", getpid() );
    return 0;
}
```

Process 1235 is created by the **fork()** system call, and we call it the **child process**.

```
[example@3150]$ cat ProcessUser/fork_example_1.c
```

Process creation – **fork()** system call

- So, how do **fork()** and the processes behave?



```
$ ./fork_example_1
Ready (PID=1234)
My PID is 1234
My PID is 1235
$ _
```

Process 1234 is **parent process**.

Process 1235 is **child process**.

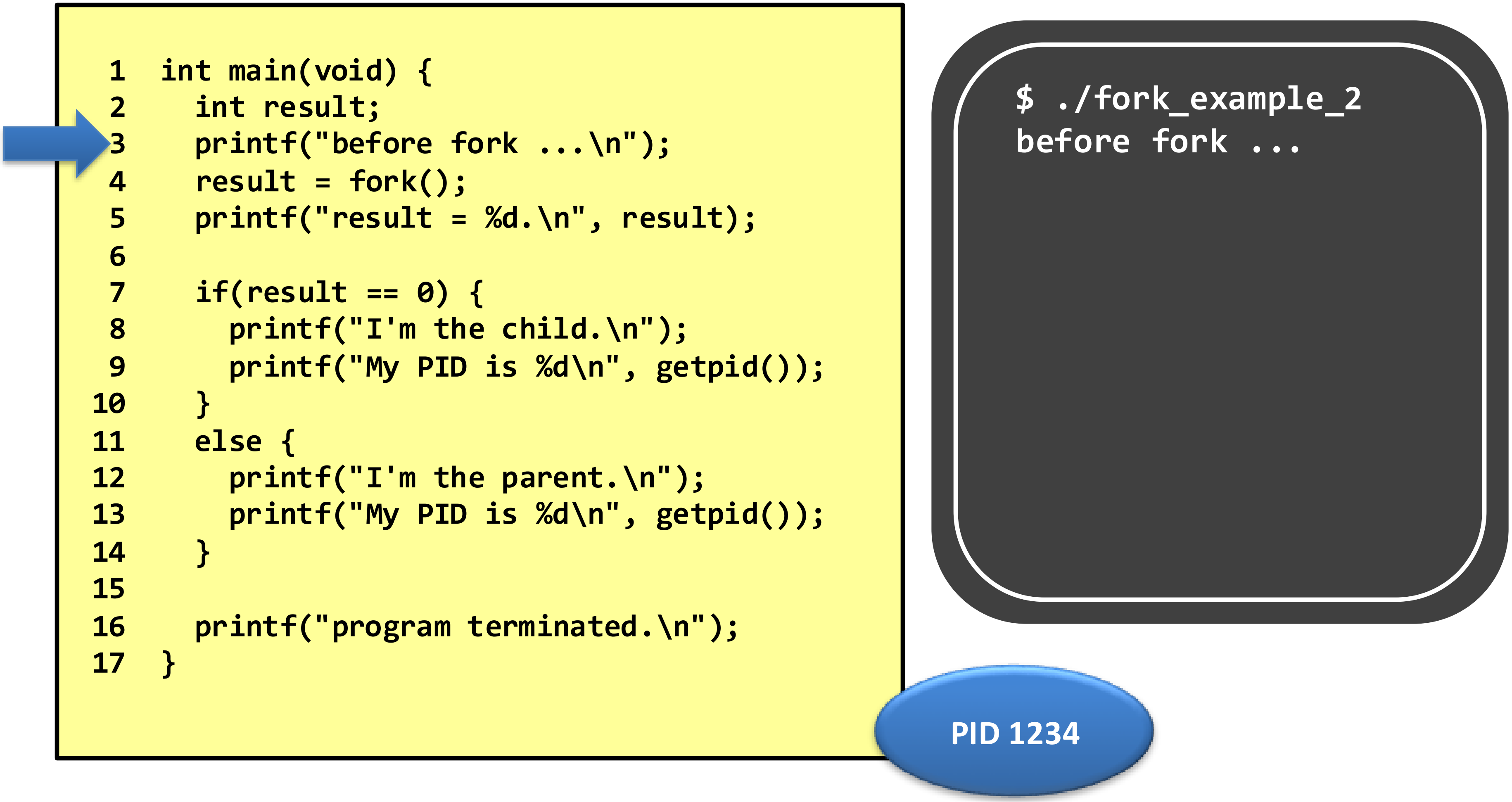
<u>What do we know so far?</u>

-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*.

```
[example@3150]$ cat ProcessUser/fork_example_1.c
```

Process creation – **fork()** system call



```c
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```

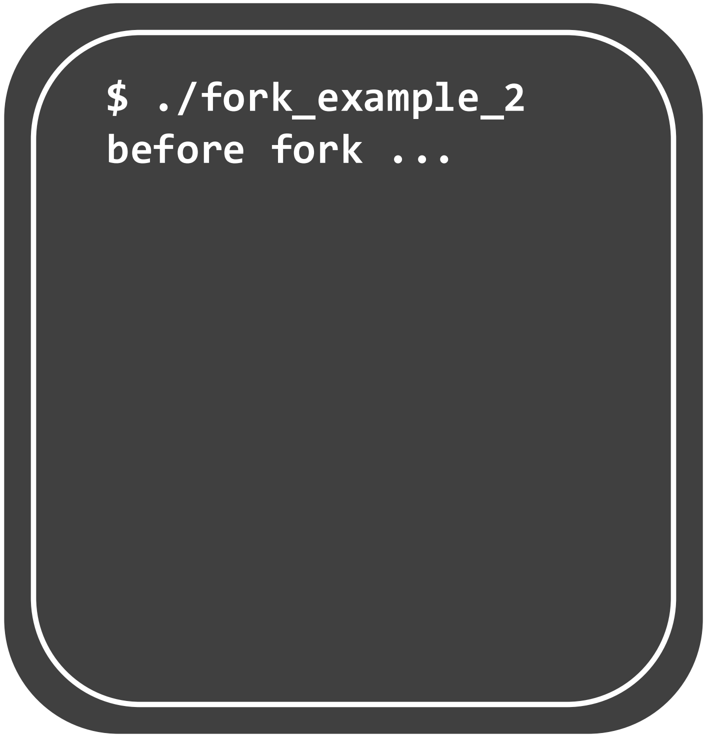
```
$ ./fork_example_2
before fork ...
```

PID 1234

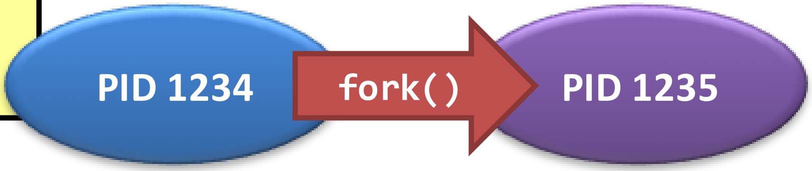
```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – **fork()** system call

```
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```



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



```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – **`fork()`** system call

Let there be only **ONE CPU**. Then…

- Only one process is allowed to be executed at one time.
- However, we can't predict which process will be chosen by the OS.
- That is controlled by the OS's **scheduler**.

NOTE THIS

In this example, we assume that the parent, PID 1234, runs first, after the **`fork()`** call.

```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

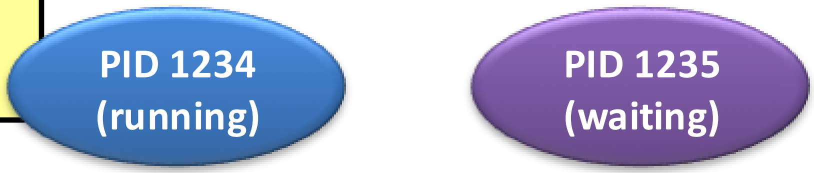
Process creation – **fork()** system call

```c
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```

```
$ ./fork_example_2
before fork ...
result = 1235
```

Important

For parent, the return value of **fork()** is the PID of the created child.

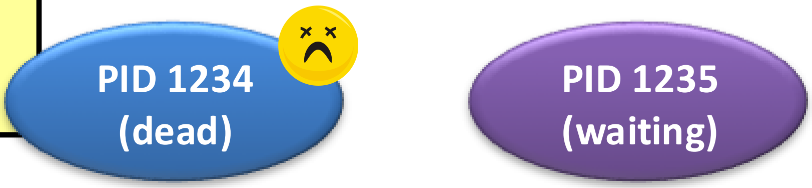


```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – **fork()** system call

```
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```

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



PID 1234
(dead)

PID 1235
(waiting)

```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – **fork()** system call

```c
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```

```
$ ./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 (dead)

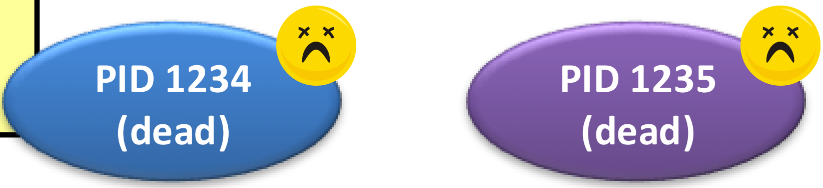
PID 1235 (running)

```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – **fork()** system call

```
int main(void) {
  int result;
  printf("before fork ...\n");
  result = fork();
  printf("result = %d.\n", result);

  if(result == 0) {
    printf("I'm the child.\n");
    printf("My PID is %d\n", getpid());
  }
  else {
    printf("I'm the parent.\n");
    printf("My PID is %d\n", getpid());
  }

  printf("program terminated.\n");
}
```

```
$ ./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.
$ _
```



```
[example@3150]$ cat ProcessUser/fork_example_2.c
```

Process creation – `fork()` system call

- `fork()` behaves like "*cell division*".
  - It creates the child process by **cloning** from the parent process, including all user-space data and some kernel-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 "A", then the child will also have file "A" opened automatically.

Process creation – `fork()` system call

- However…
  - `fork()` does not clone the following PCB data in the kernel space.

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.

What is a process?

- process creation.

- program execution.

`fork()` can only duplicate…

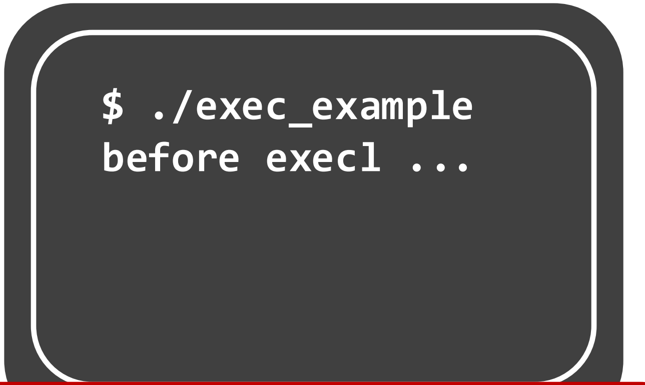
- If a process can only duplicate itself and always runs the same program, it’s not quite meaningful
  - how can we execute other programs?

- We want **CHANGE!**
  - Meet the **`exec*()`** system call family.

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 ...
```

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.

```
[example@3150]$ cat ProcessUser/exec_example.c
```

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.**

```
[example@3150]$ cat ProcessUser/exec_example.c
```

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

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

WHAT?!
The shell prompt appears!

```
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
$ _
```

The output says:

(1) The gray code block **is not reached**!

(2) The process is **terminated**!

WHY IS THAT?!

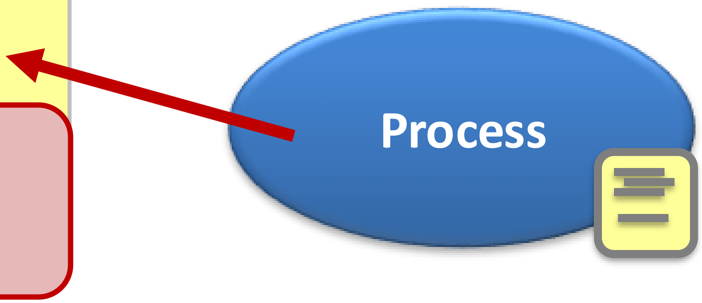
```
[example@3150]$ cat ProcessUser/exec_example.c
```

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;
}
```

Originally, the process is executing the program “**exec_example**”.



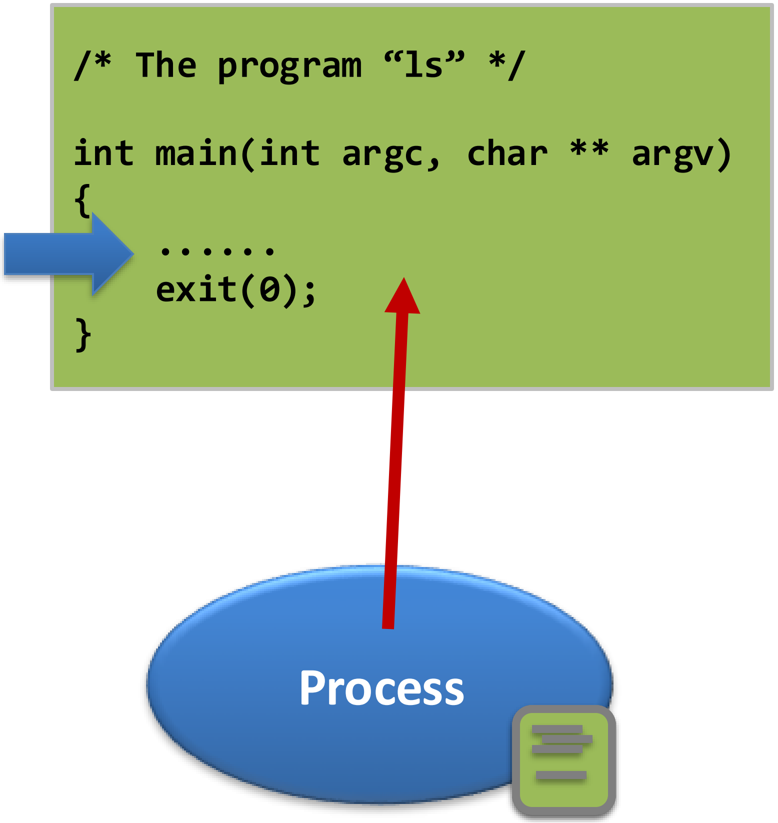
```
[example@3150]$ cat ProcessUser/exec_example.c
```

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;
}
```

execl() changes the execution from “**exec_example**” to “**/bin/ls**”



```
/* The program “ls” */

int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```

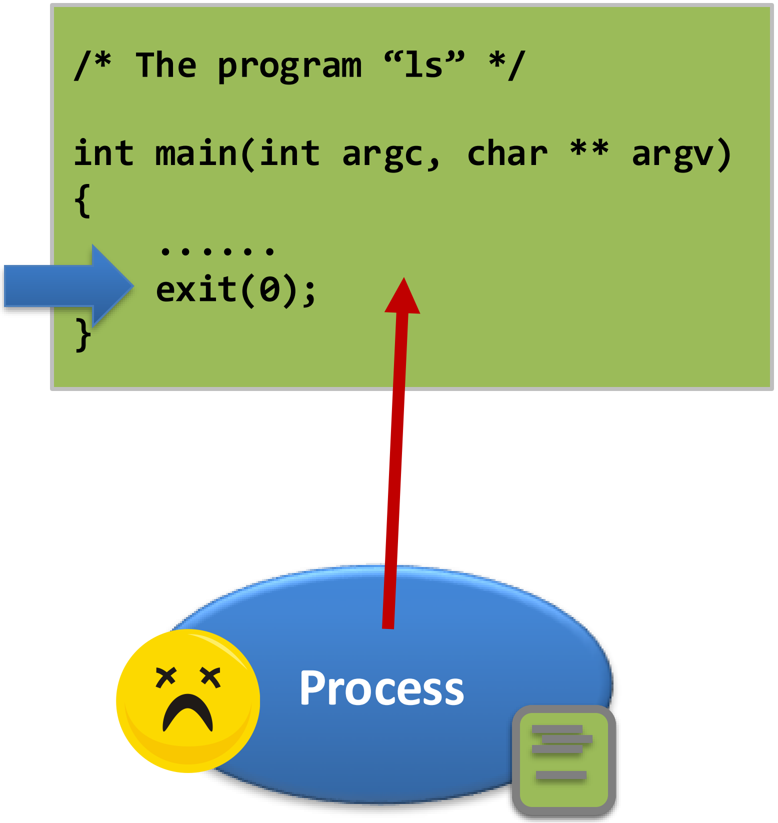
```
[example@3150]$ cat ProcessUser/exec_example.c
```

exec

- The **exec** system call family is not simply a function that “invokes” a command.

The “**return**” or the “**exit()**” statement in “**/bin/ls**” will terminate the process…

Therefore, it is certain that the process cannot go back to the old program!



```
/* The program “ls” */

int main(int argc, char ** argv)
{
    ......
    exit(0);
}
```

```
[example@3150]$ cat ProcessUser/exec_example.c
```

exec

- 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, i.e., PCB, **is preserved**, including:
  - PID;
  - Process relationship;
  - etc.

~ reverse takeover in stock market

exec*() – arguments explained

- Environment variables
  - An array of strings maintained by the shell.

```
int main(int argc, char **argv, char **envp) {
    int i;
    for(i = 0; envp[i]; i++)
        printf("%s\n", envp[i]);
    return 0;
}
```

```
$ ./envp
SHELL=/bin/bash
PATH=......
......
$ _
```

The “****envp**” variable is an array of string
A string is an array of characters

```
[example@3150]$ cat ProcessUser/envp.c
```

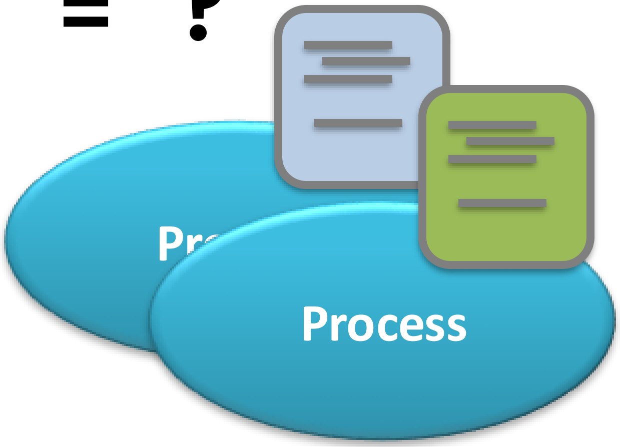
exec*() – arguments explained

- Environment variables
  - Quite a number of programs will read and make use of the environment variable.

Variable name	Description
`SHELL`	The path to the shell that you’re using.
`PWD`	The full path to the directory that you’re currently on.
`HOME`	The full path to your home directory.
`USER`	Your login name.
`EDITOR`	Your default text editor.
`PRINTER`	Your default printer.

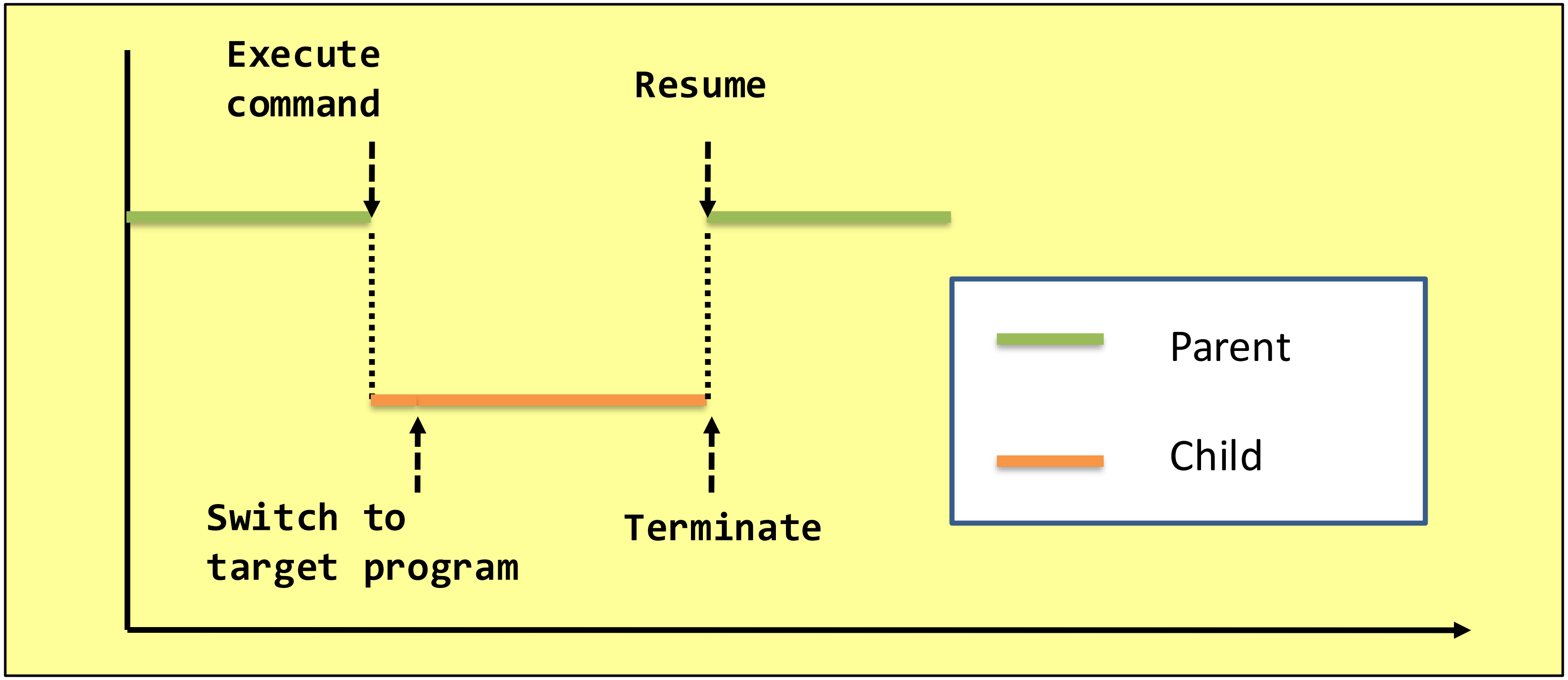
What is a process?

- process creation.
- program execution.
- `fork() + exec*() = ?`



When **fork()** meets **exec*()**…

- Can implement the core part of a shell,
- Can implement the C library call **system()**
- …



```
[examples@3150] cat ProcessUser/system_example.c
```

fork() + exec*() = system()?!

```c
int system_ver_3150(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl(cmd_str, cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    return 0;
}

int main(void) {
    printf("before...\n\n");
    system_ver_3150("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

Some strange cases may happen some times

```
$ ./system_implement_1
before...

after...
Makefile
system_implement_1
system_implement_1.c
$ _
```

```
[example@3150]$ cat ProcessUser/system_implement_1.c
```

fork() + exec*() = system()...

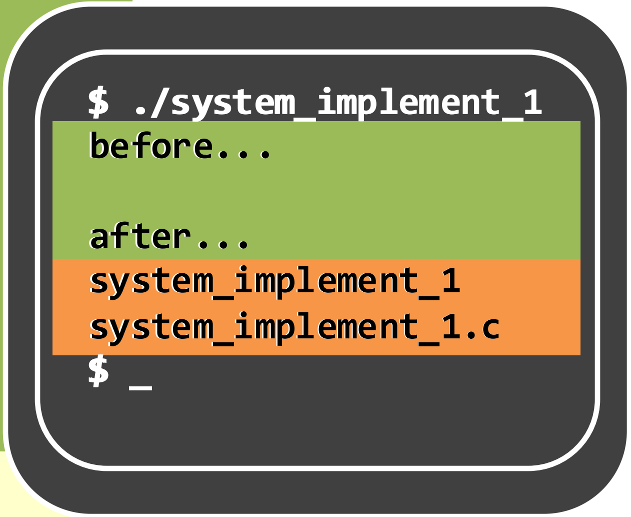
```c
int system_ver_3150(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl(cmd_str, cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    return 0;
}

int main(void) {
    printf("before...\n\n");
    system_ver_3150("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

Let's re-color the program!

🟩 Parent process

🟧 Child process

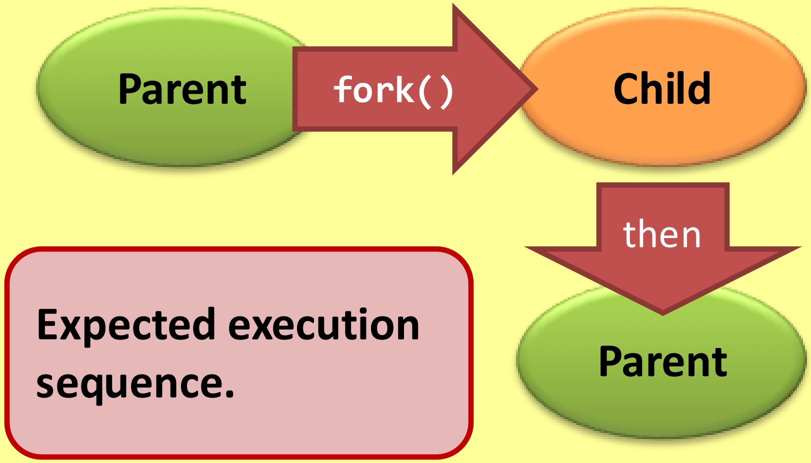
🟦 Both processes



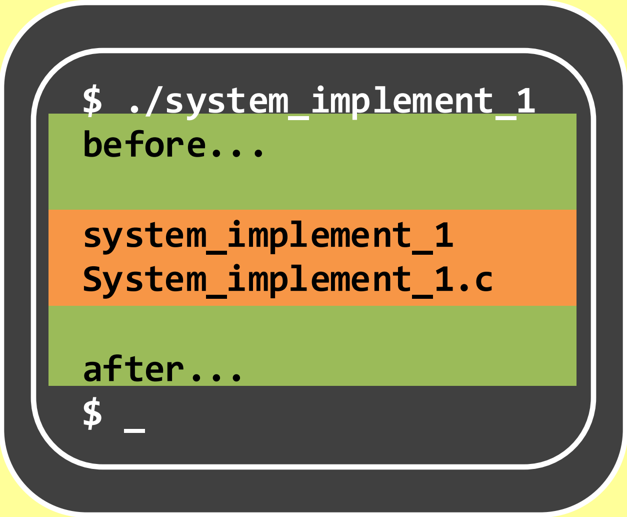
```
$ ./system_implement_1
before...

after...
system_implement_1
system_implement_1.c
$ _
```

```
[example@3150]$ cat ProcessUser/system_implement_1.c
```

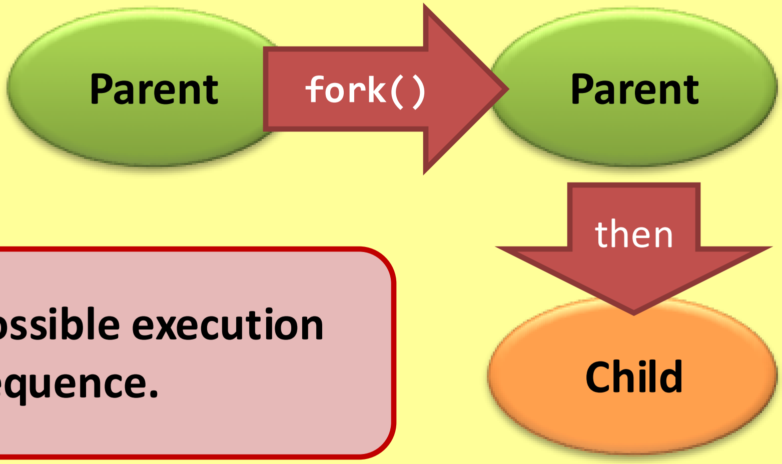
fork() + exec*() = system()...



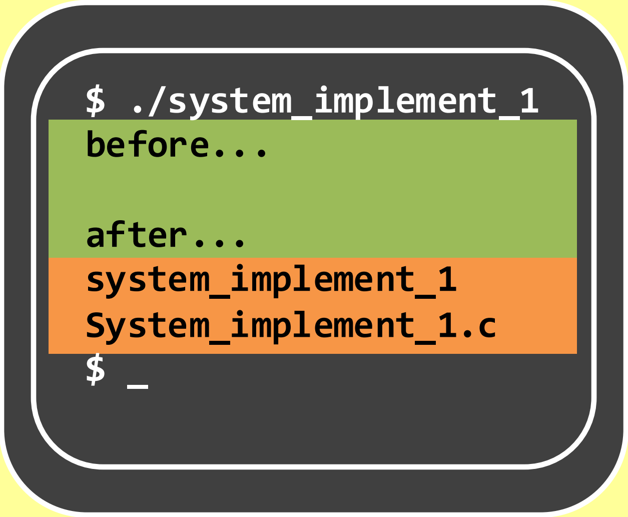
Expected execution sequence.



```
$ ./system_implement_1
before...

system_implement_1
System_implement_1.c

after...
$ _
```



Possible execution sequence.



```
$ ./system_implement_1
before...

after...
system_implement_1
System_implement_1.c
$ _
```

fork() + exec*() = system()...

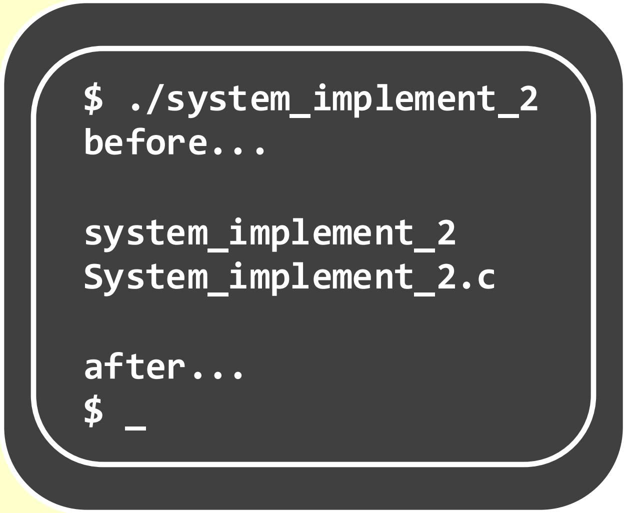
- It is very weird to allow different execution orders.

- How to let the child to execute first?
  - But…we can’t control the **OS scheduler**

- Then, our problem becomes…
  - How to **suspend** the execution of the parent process?
  - How to wake the parent up after the child is terminated?

fork()+ exec*() + wait() = system()

```
int system_ver_3150(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl("/bin/sh", "/bin/sh",
              "-c", cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    wait(NULL);
    return 0;
}

int main(void) {
    printf("before...\n\n");
    system_ver_3150("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

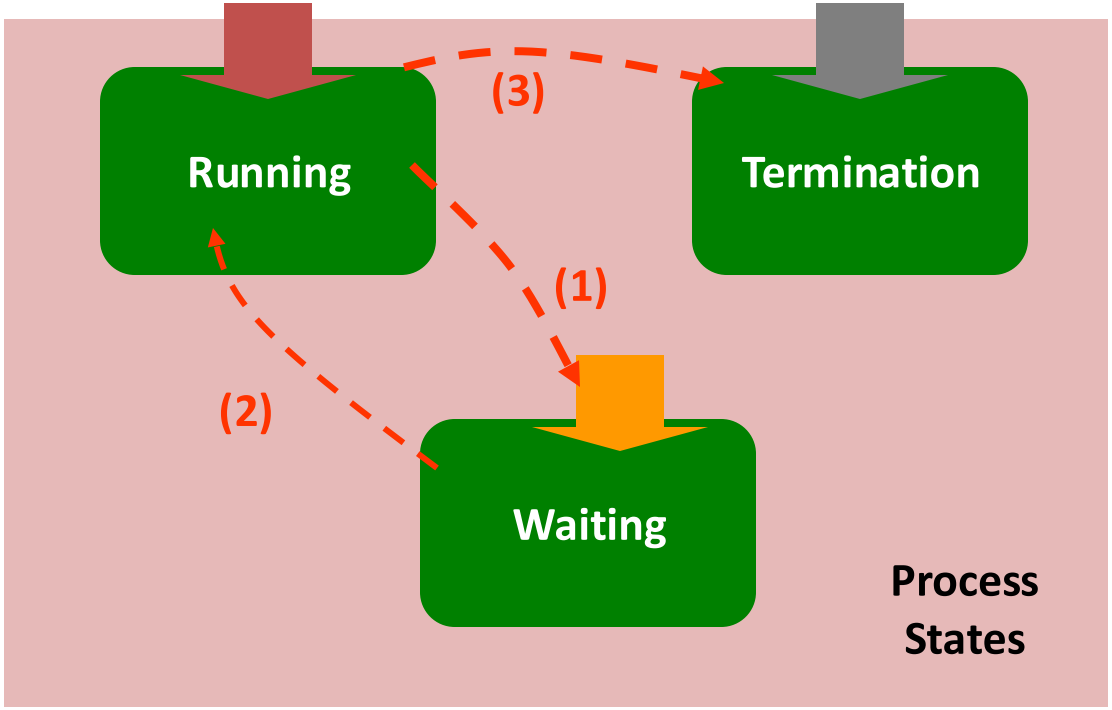


```
$ ./system_implement_2
before...

system_implement_2
System_implement_2.c

after...
$ _
```

```
[example@3150]$ cat ProcessUser/system_implement_2.c
```

Process Life Cycle (user-space)

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



Running

Termination

Waiting

(1)

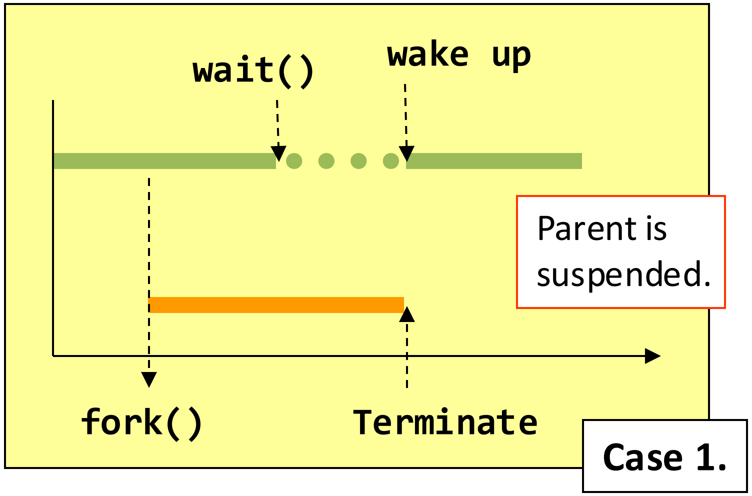
(2)

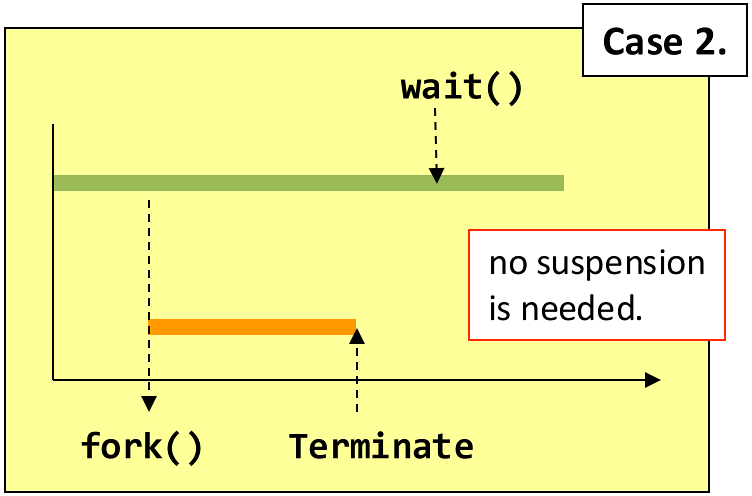
(3)

Process States

`wait()` – user-space

- `wait()` system call
  - **suspend** the calling process to waiting state and **return** (wakes up) when
    - one of its child processes changes from
      - **running to terminated**.
    - Or a signal is received (will cover)
  - **return immediately** (i.e., does nothing) if
    - It has no children
      - Or a child terminates before the parent calls wait for





wait() VS waitpid()

wait()		**waitpid()**
Wait for any one of the children.	vs	Depending on the parameters, **waitpid()** will wait for a particular child only.
Detect child **termination** only.		Depending on the parameters, **waitpid()** **can detect multiple child's status change**

wait() also has a very important hidden task (will cover next)

Summary

- A new process is created by **fork()**
  - Who is the first process?
- A process is a program being brought by **exec** to the memory
  - has state (initial state= ready)
  - waiting for the OS to schedule the CPU to run it
- Can a process execute more than one program?
- You now know how **system()** C library call is implemented by syscalls **fork()** , **exec()** , and **wait()**

Summary

- After going through “Process” Lab
  - You should be able to write your own shell now
    - while (user-command = scanf()   != “exit”)

```
If (fork()==0)
        exec(user-command);
wait();
```

➔ your assignment