Lecture 3: Process I

Yinqian Zhang @ 2021, Spring

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?

```
$ ls | cat | cat
[Ctrl + C]
$
```

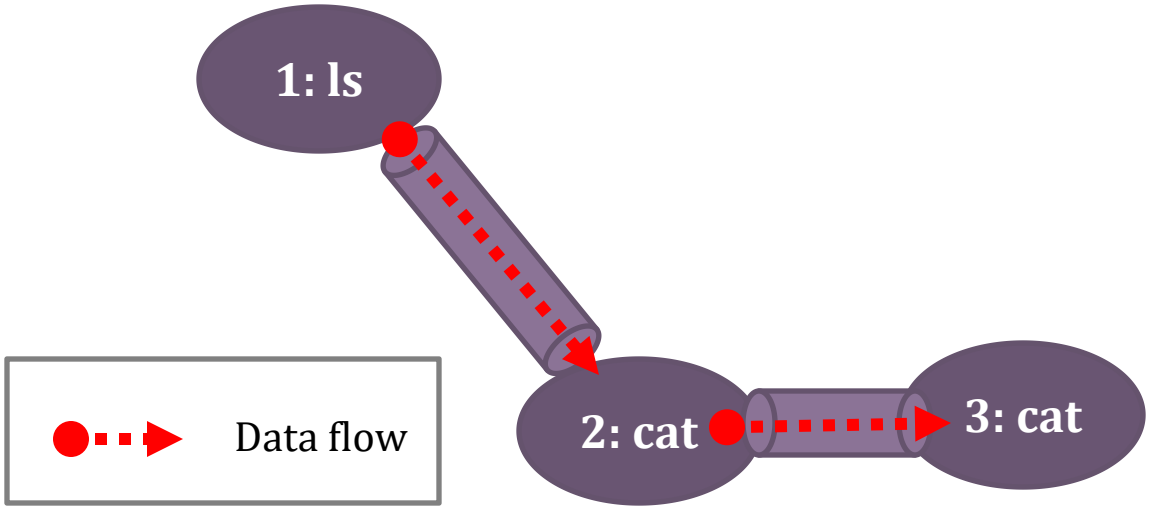
- The command involves **three processes**.
- It will stop early if I send a **signal** to interrupt it.
- Its progress is determined by the **scheduler**.
- The three processes **cooperate** to give useful output.

*For illustration purpose only, this command line doesn't need two cats indeed

What is a process?

- What are those two “cats”?
  - 2 different processes using the same code “**`/bin/cat`**”.

```
$ ls | cat | cat
[Ctrl + C]
$
```



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);
  }
}
```

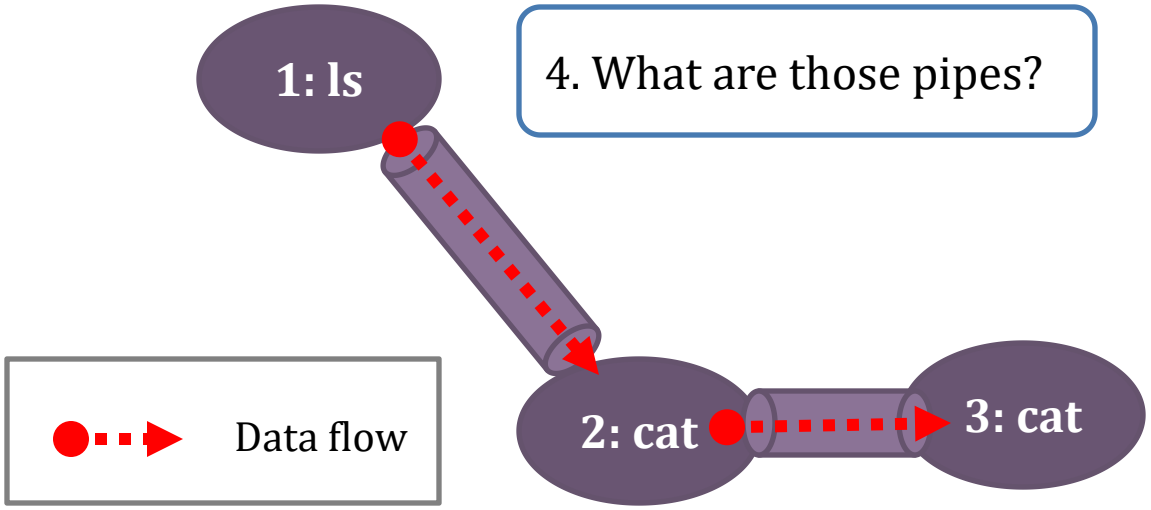
Our Roadmap

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.

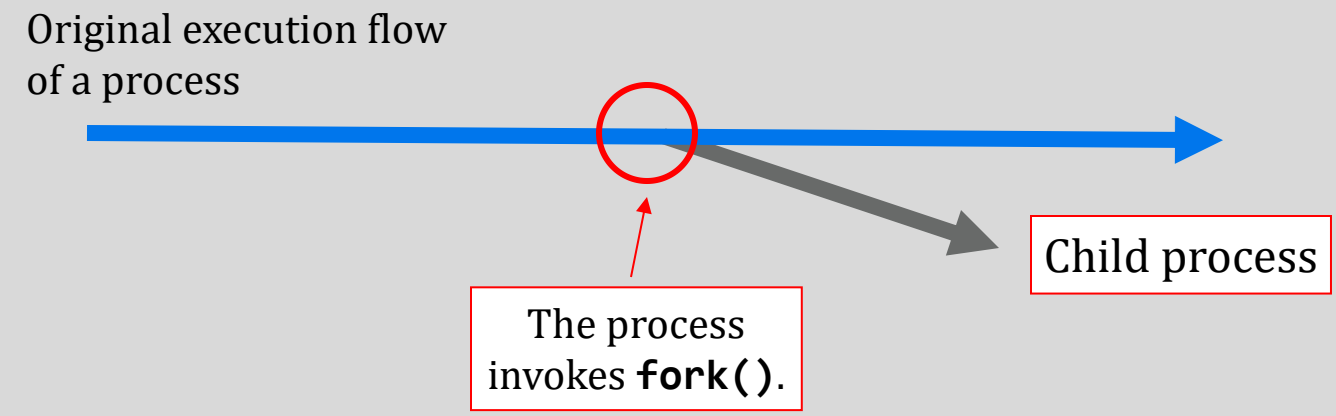
```
#include <stdio.h>   // printf()
#include <unistd.h>  // getpid()

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
```

Process creation

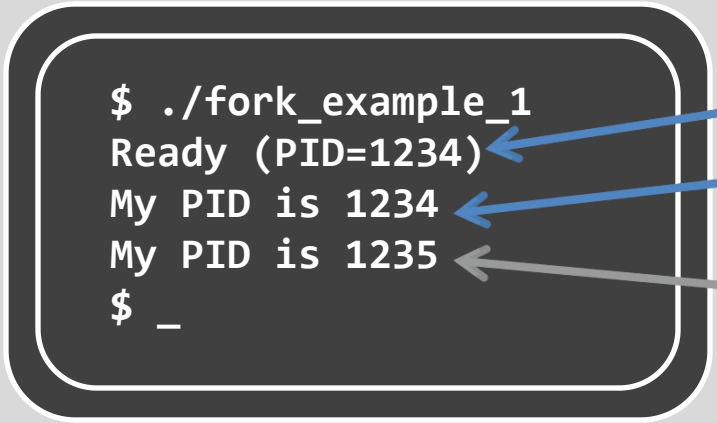
- To create a process, we use the system call **`fork()`**.

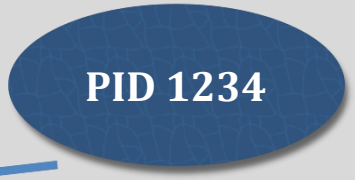
Not this

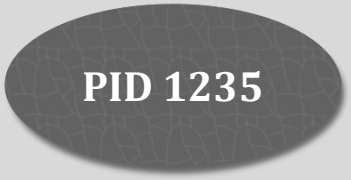


Process creation – `fork()` system call

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







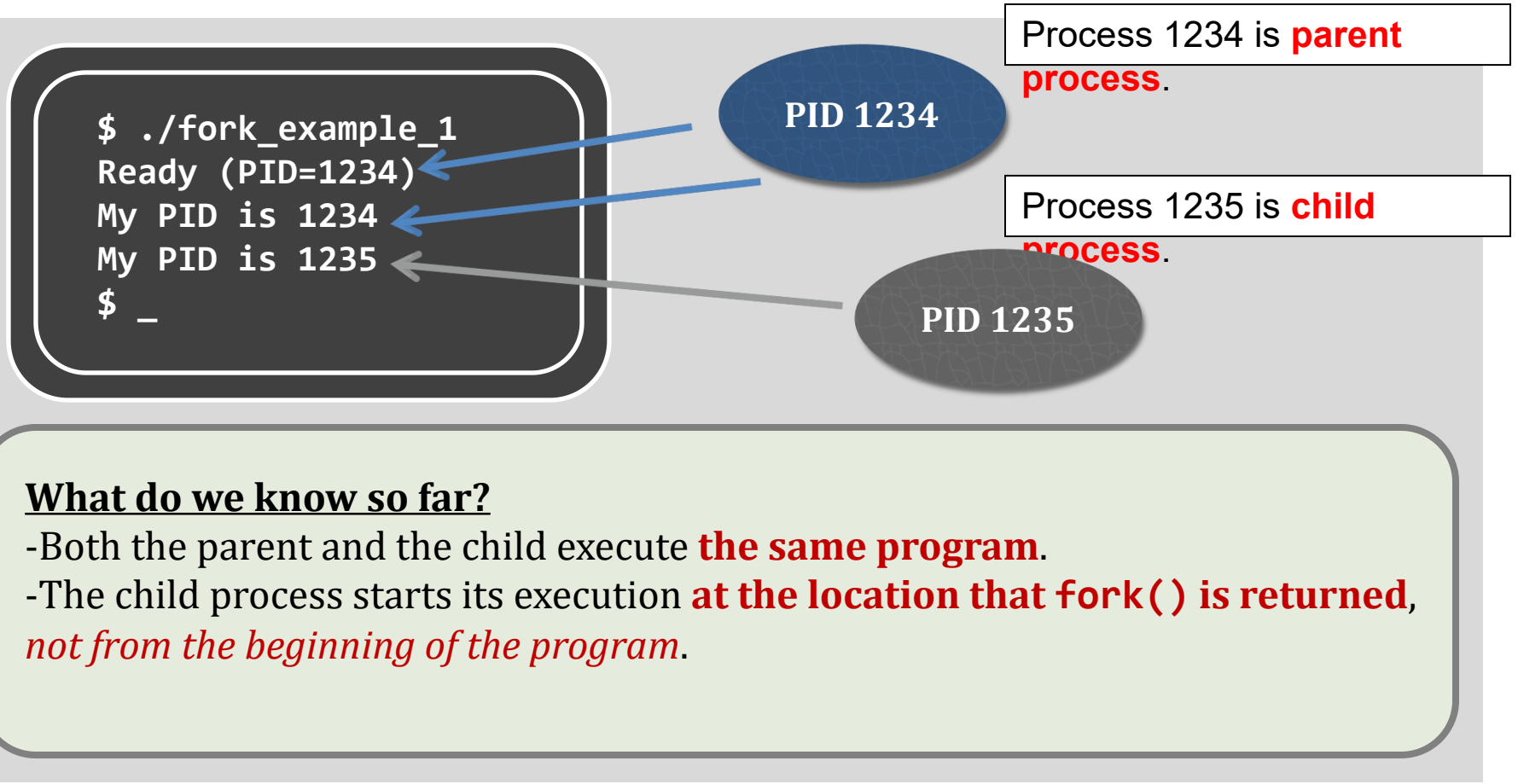
Process 1234 is the original process, and we call it the **parent process**.

```
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**.

Process creation – `fork()` system call

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



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*.

Process creation – **`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  }
```

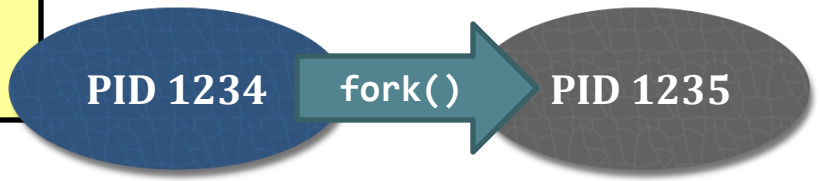
```
$ ./fork_example_2
before fork ...
```

PID 1234

Process creation – **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 ...
```



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.

Process creation – `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
```

Important

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

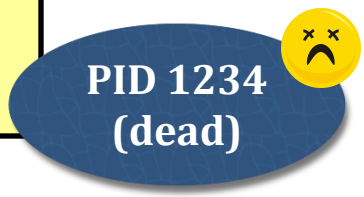
PID 1234 (running)

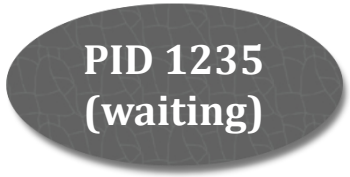
PID 1235 (waiting)

Process creation – **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.
```





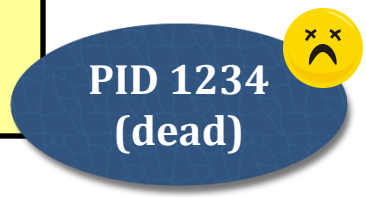
Process creation – `fork()` system call

```
 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

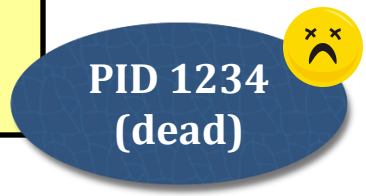
For child, the return value of `fork()` is **0**.

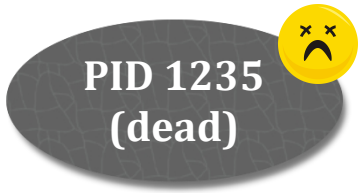


Process creation – **fork()** system call

```
 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.
$ _
```





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, 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.

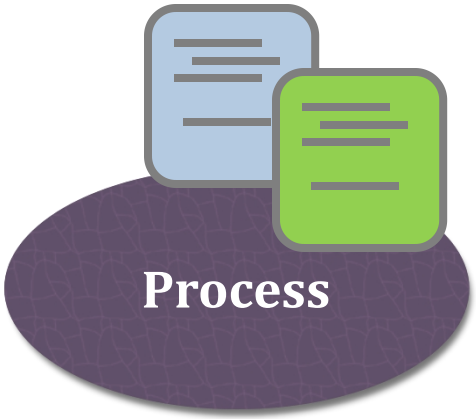
Process creation – `fork()` system call

- However…
  - **`fork()`** does not clone the following…
  - Note: they are 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.

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

- **`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;
}
```

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.

```
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**".

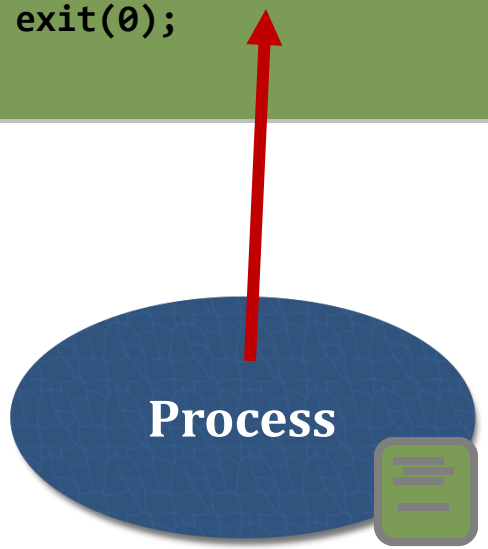
exec

- 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);
}
```



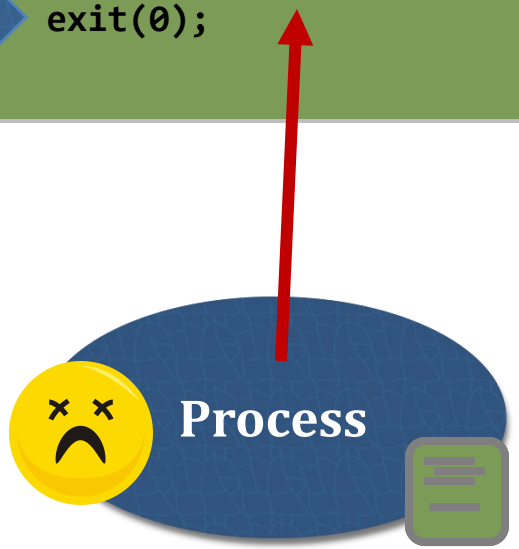
exec

- 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);
}
```



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 is preserved, including:
  - PID;
  - Process relationship;
  - etc.

~ reverse takeover in stock market

exec*() – arguments explained

- Environment variables
  - A set 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

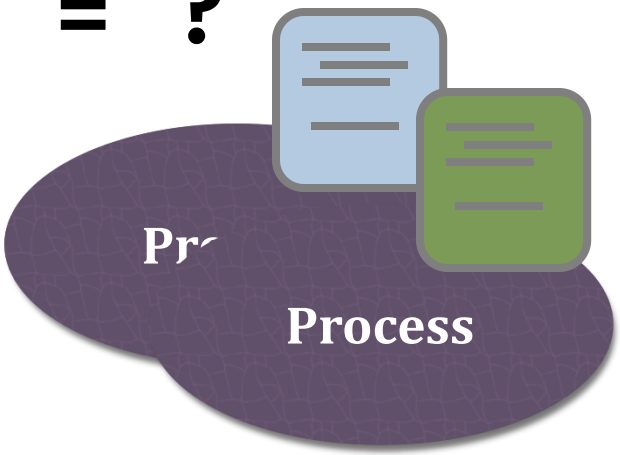
exec*() – arguments explained

- Environment variables
  - A set of strings maintained by the shell.
  - 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.

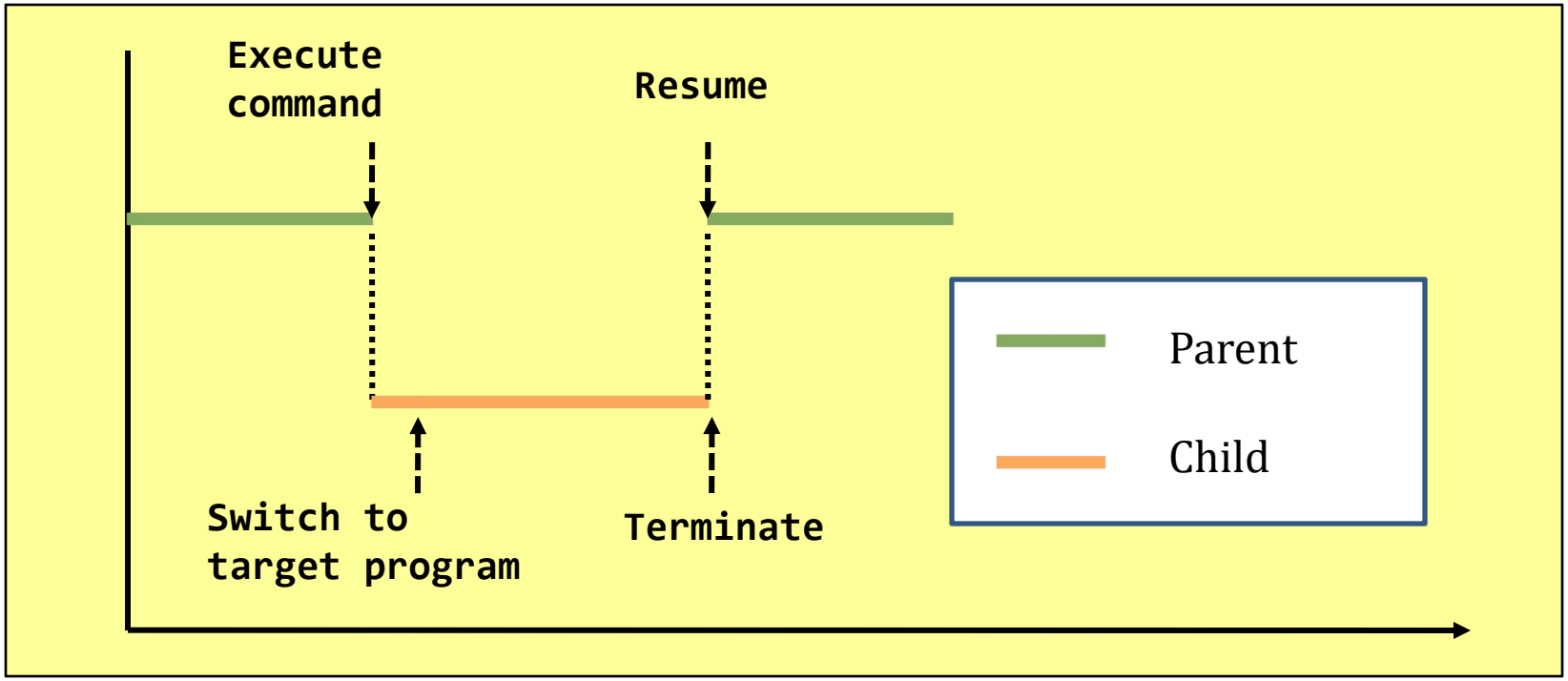
What is a process?

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



When **`fork()`** meets **`exec*()`**…

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



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

```
 1  int system_ver_CS302(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_CS302("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

```
$ ./system_implement_1
before...
Makefile
system_implement_1
system_implement_1.c
…
after...
$ _
```

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

```
 1  int system_ver_CS302(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_CS302("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

Some strange cases may happen some times

```
$ ./system_implement_1
before...

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

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

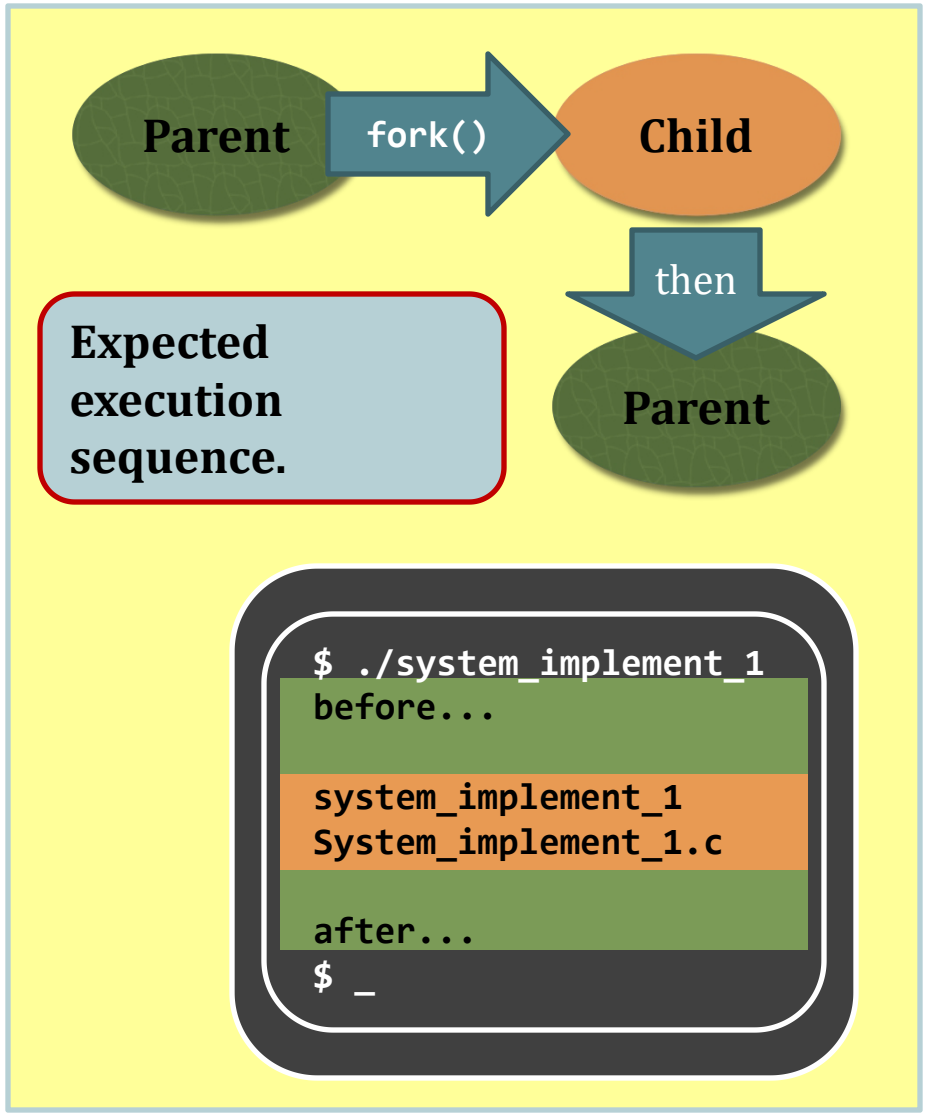
```
 1  int system_ver_CS302(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl(cmd_str, cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      return 0;
10  }
11
12  int main(void) {
13      printf("before...\n\n");
14      system_ver_CS302("/bin/ls");
15      printf("\nafter...\n");
16      return 0;
17  }
```

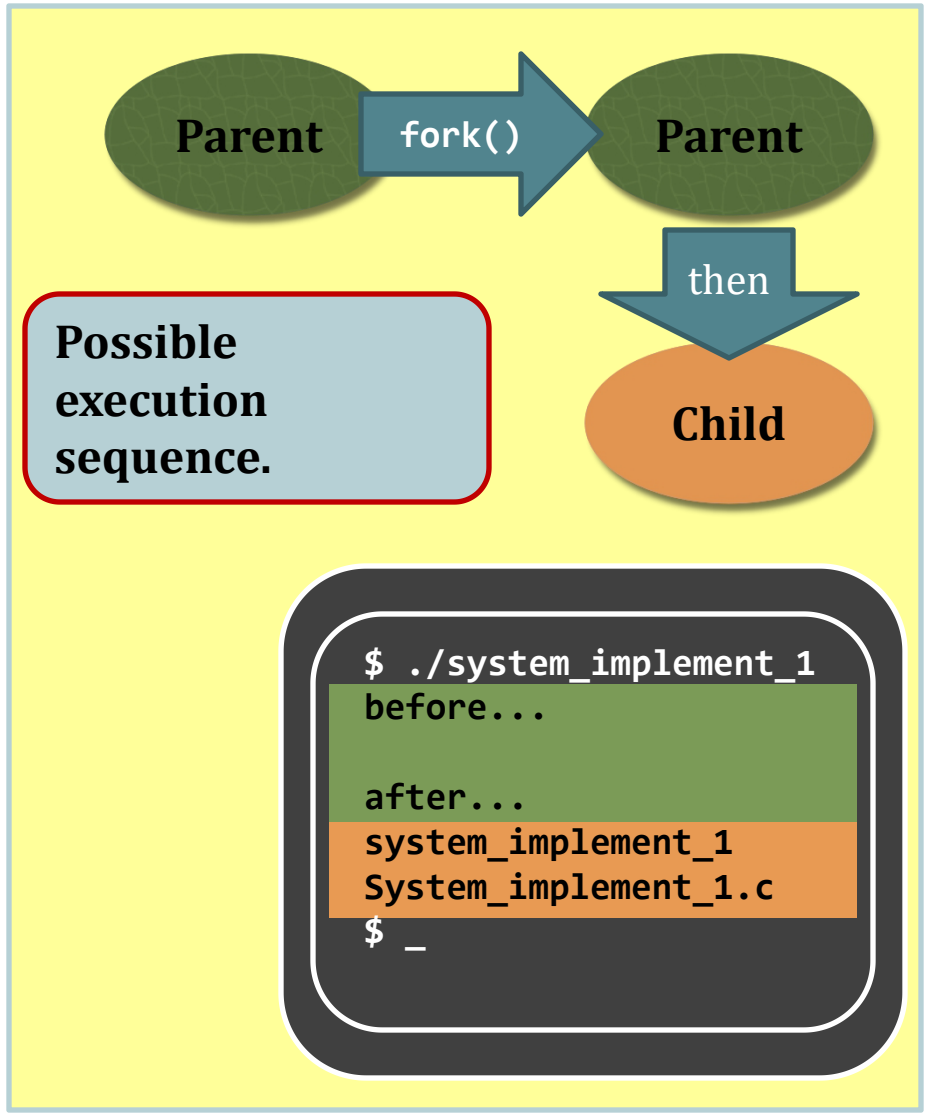
Let's re-color the program!

- Parent process
- Child process
- Both processes

```
$ ./system_implement_1
before...

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

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





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

- It is very weird to allow different execution orders.

- How to let the child 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?

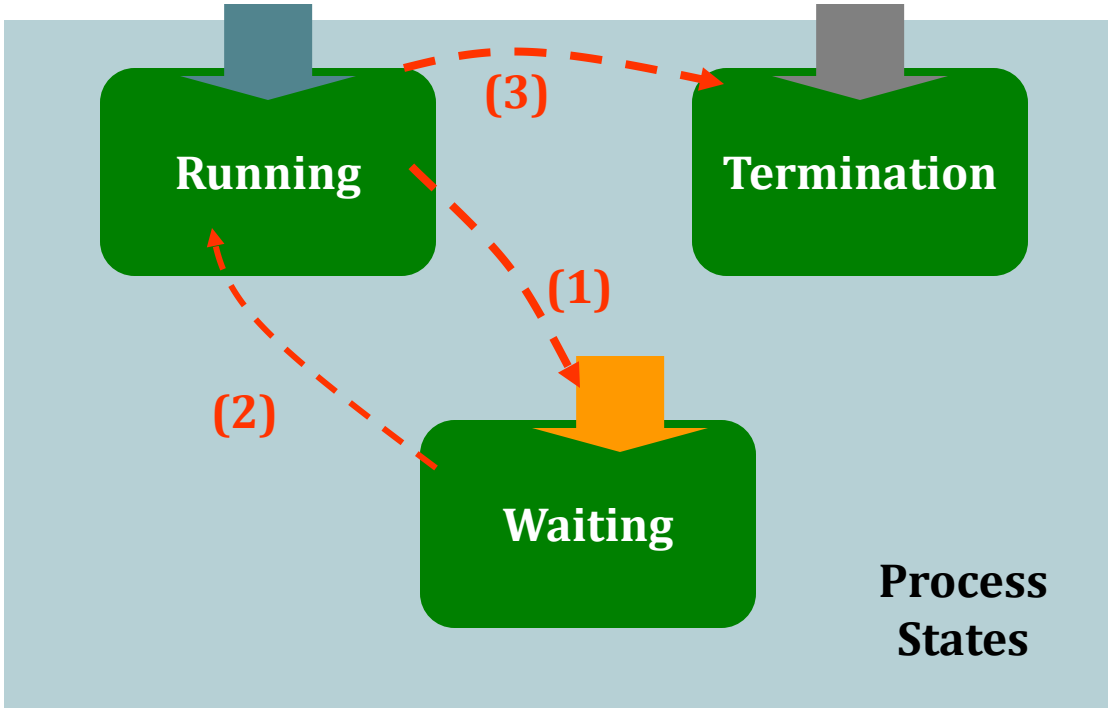
fork()+ exec*() + wait() = system()

```
 1  int system_ver_CS302(const char *cmd_str) {
 2      if(cmd_str == -1)
 3          return -1;
 4      if(fork() == 0) {
 5          execl("/bin/sh", "/bin/sh",
                    "-c", cmd_str, NULL);
 6          fprintf(stderr,
                "%s: command not found\n", cmd_str);
 7          exit(-1);
 8      }
 9      wait(NULL);
10      return 0;
11  }
12
13  int main(void) {
14      printf("before...\n\n");
15      system_ver_CS302("/bin/ls");
16      printf("\nafter...\n");
17      return 0;
18  }
```

```
$ ./system_implement_2
before...

system_implement_2
System_implement_2.c

after...
$ _
```

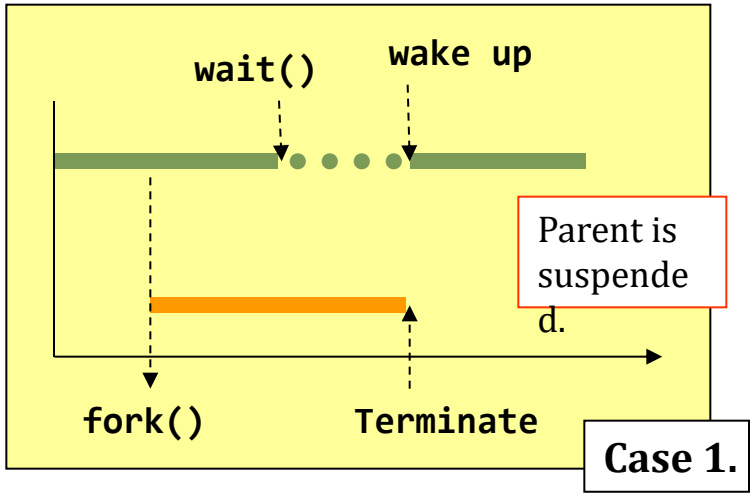
Process Life Cycle (user-space)

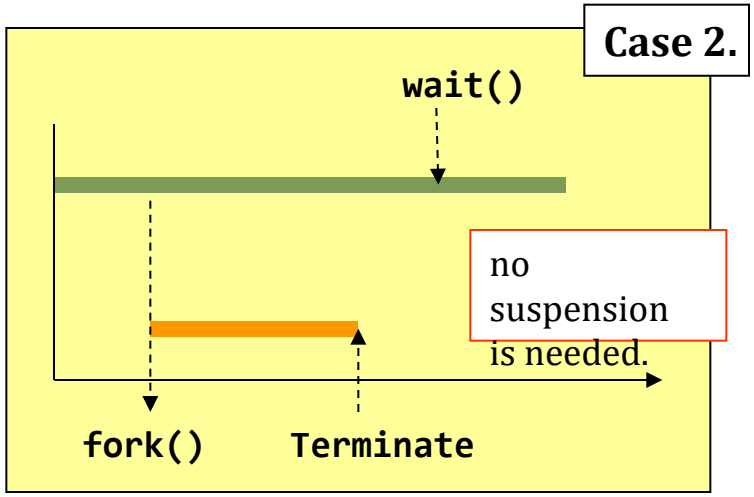
```
int main(void) {
    int x = 1;
    getchar();
    return x;
}
```



`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()	vs	waitpid()
Wait for any one of the children.		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?
    - Yes, keeps on calling the **`exec`** system call family
- You now know how **`system()`** C library call is implemented by syscalls **`fork()`** , **`exec()`** , and **`wait()`**

Thank You!