Operating Systems

Associate Prof. Yongkun Li
中科大-计算机学院 副教授
http://staff.ustc.edu.cn/~ykli

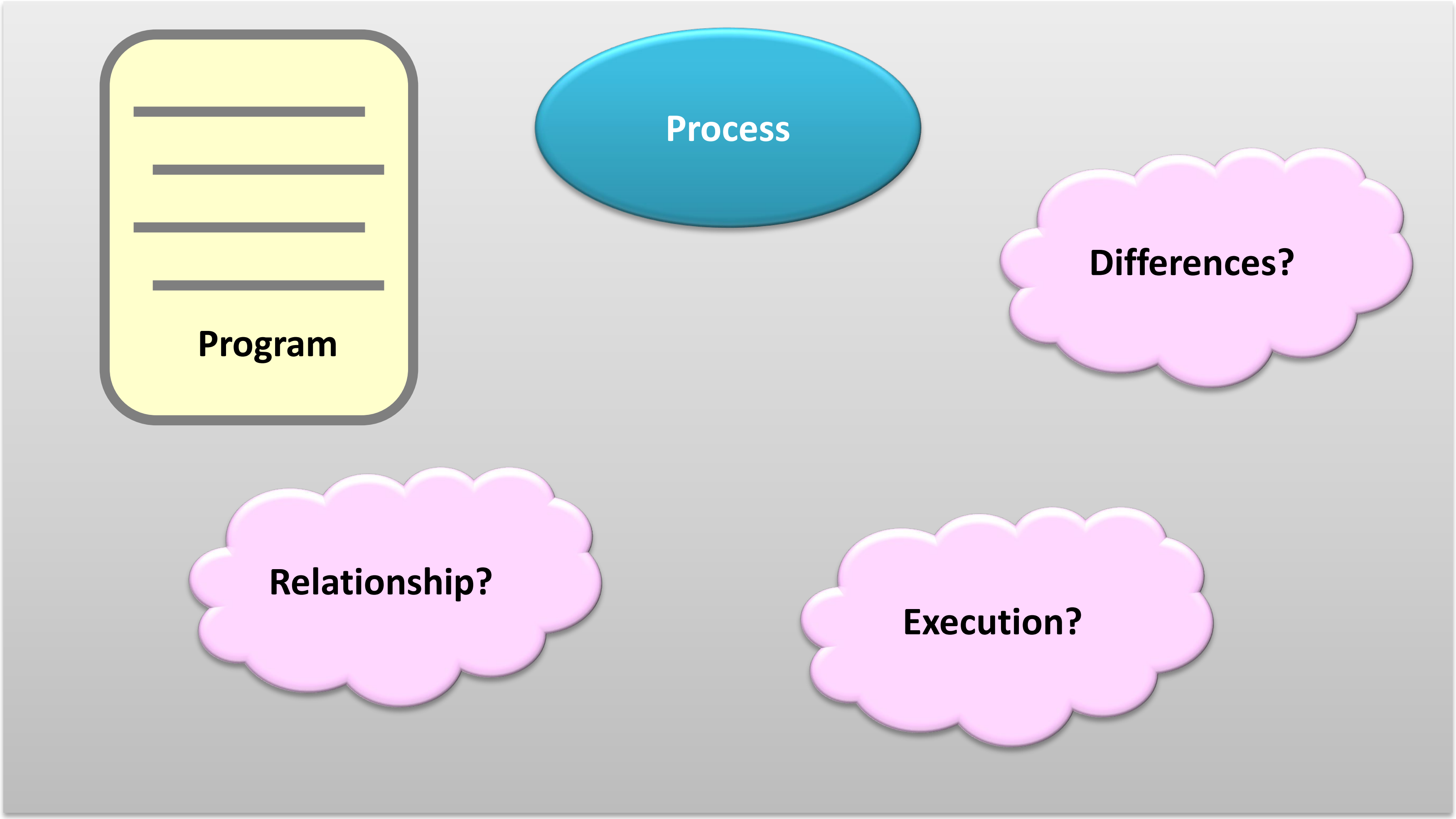
Chapter 3
Process Concepts & Operations

Outline

- Process Concept
  - Program vs process
  - Process in memory & PCB
  - Process state
- Processes Operations
  - Process creation, program execution, process termination
  - UNIX example: `fork(), exec*(), wait()`

What is a process?

Informally, a process is a program in execution.



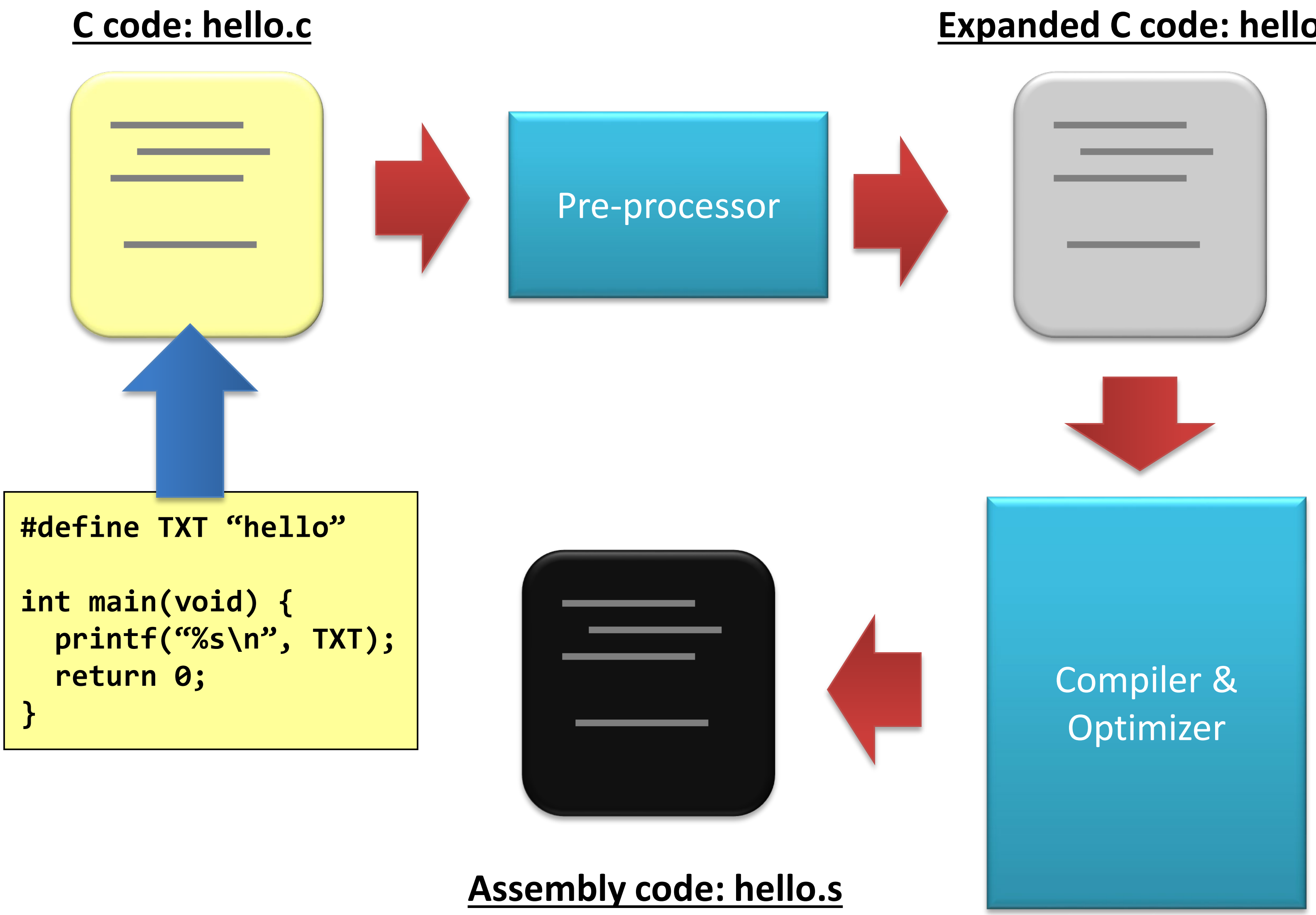
What is a program?

What is a program?

- What is a program?
  - A program is a just **a piece of code**.

- But, which code do you mean?
  - High-level language code: C or C++?
  - Low-level language code: assembly code?
  - Not-yet an executable: object code?
  - Executable: machine code?

Flow of building a program (1 of 2)




```
#define TXT "hello"

int main(void) {
  printf("%s\n", TXT);
  return 0;
}
```

(Still…1 of 2) Pre-processor

- The pre-processor expands:
  - **`#define`**, **`#include`**, **`#ifdef`**, **`#ifndef`**, **`#endif`**, etc.
  - Try: "**`gcc -E hello.c`**"

```
#define TXT “hello”

int main(void) {
  printf(“%s\n”, TXT);
  return 0;
}
```

Original code

Pre-processor

`gcc -E hello.c`

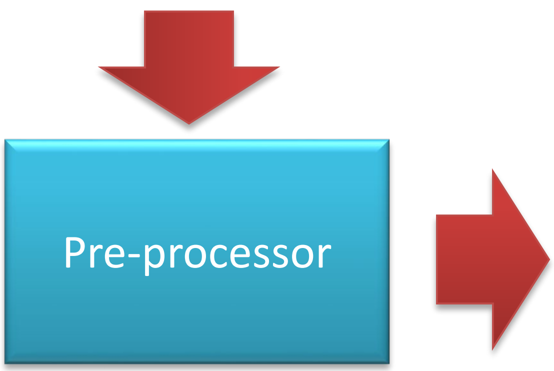
```
int main(void) {
  printf(“%s\n”, "hello");
  return 0;
}
```

Expanded code

Pre-processor

- Another example: **the macro**!

```
#define SWAP(a,b) { int c; c = a; a = b; b = c; }

int main(void) {
    int i = 10, j = 20;
    printf("before swap: i = %d, j = %d\n", i, j);
    SWAP(i, j);
    printf("after swap: i = %d, j = %d\n", i, j);
}
```




```
int main(void) {
    int i = 10, j = 20;
    printf("before swap: i = %d, j = %d\n", i, j);
    { int c; c = i; i = j; j = c; };
    printf("after swap: i = %d, j = %d\n", i, j);
}
```

(Still…1 of 2) Pre-processor

- How about: **`#include`**?

```c
#include "header.h"

int main(void) {
    add_fun(1,2);
    return 0;
}
```

Program: include.c

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

Program: header.h

Pre-processor

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

int main(void) {
    add_fun(1,2);
    return 0;
}
```

(Still…1 of 2) Compiler and Optimizer

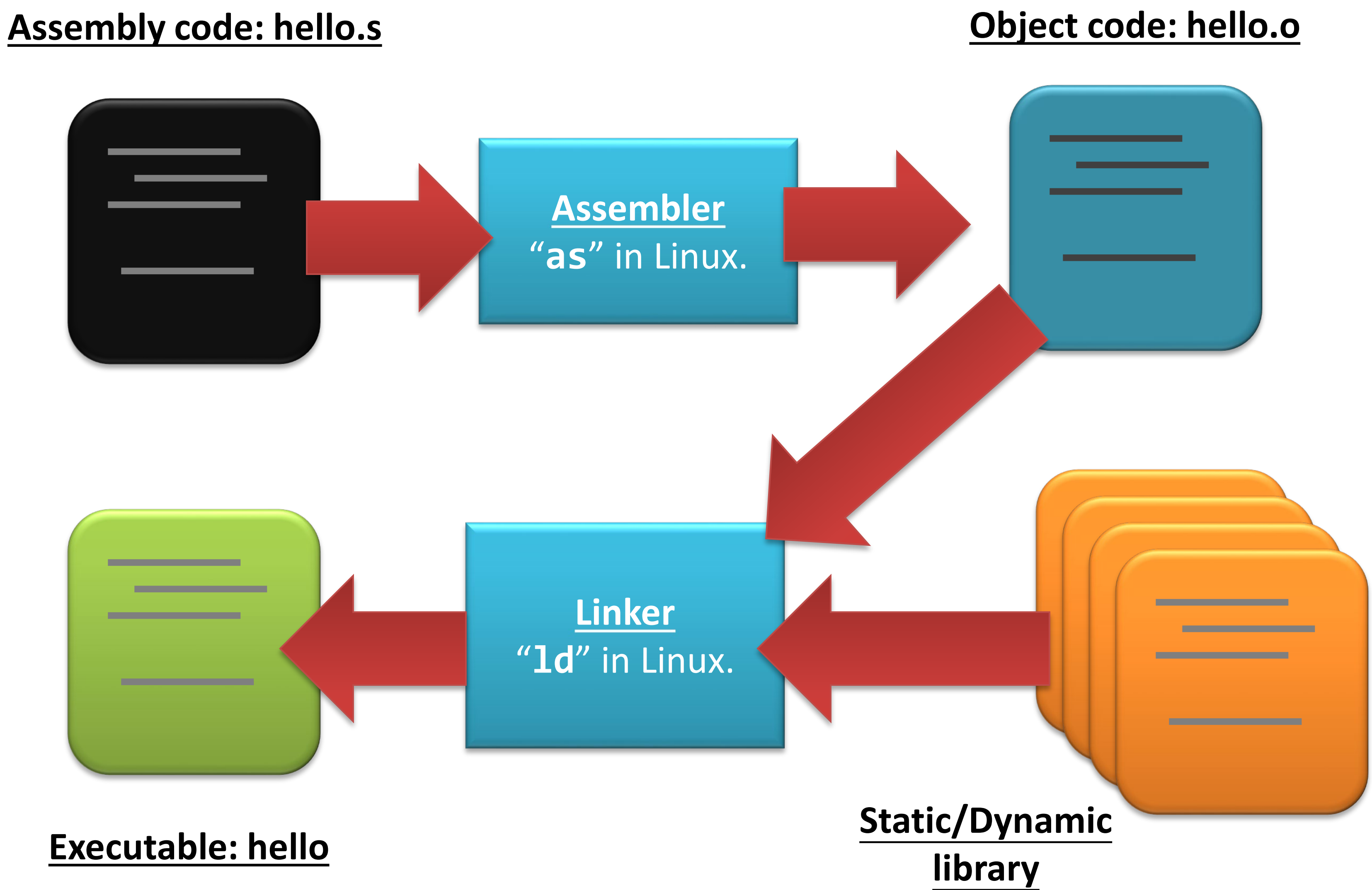
- The compiler performs:
  - Syntax checking and analyzing;
  - If there is no syntax error, construct intermediate codes, i.e., assembly codes;
- The optimizer optimizes codes
  - **It improves stupid codes!**
  - Check the parameter of gcc

"**-O**" means to optimize.

The number followed is the optimization level. Max is level 3, i.e., "**-O3**". Default is level is "**-O1**".

"**-O0**": means no optimization.

Flow of building a program (2 of 2)



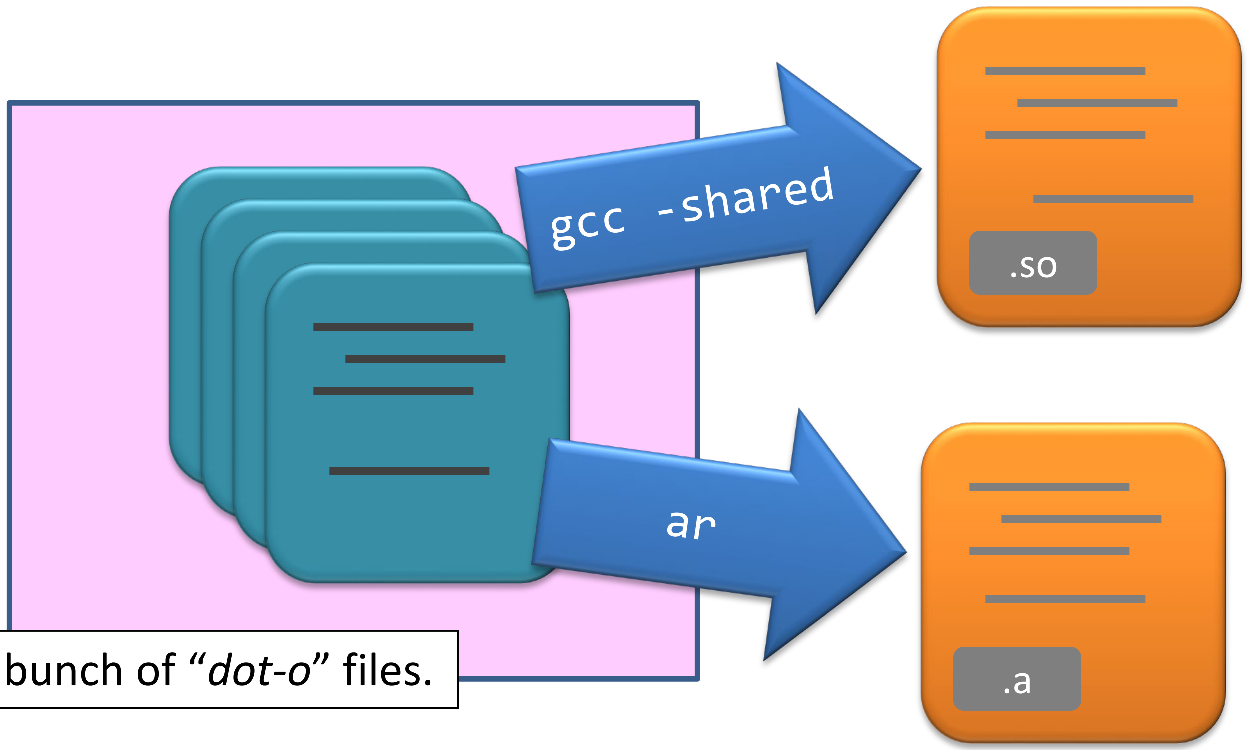
(Still…2 of 2) Assembler and Linker

- The assembler assembles “**`hello.s`**” and generates an object code “**`hello.o`**”
  - A step closer to machine code
  - Try: “**`as hello.s -o hello.o`**”

- The linker puts together all object files as well as the libraries
  - There are two kinds of libraries: statically-linked and dynamically-linked ones

Sidetrack: Library files

- A library file is…
  - just a bunch of function implementations.
  - for the linker to look for the function(s) that the target C program needs.

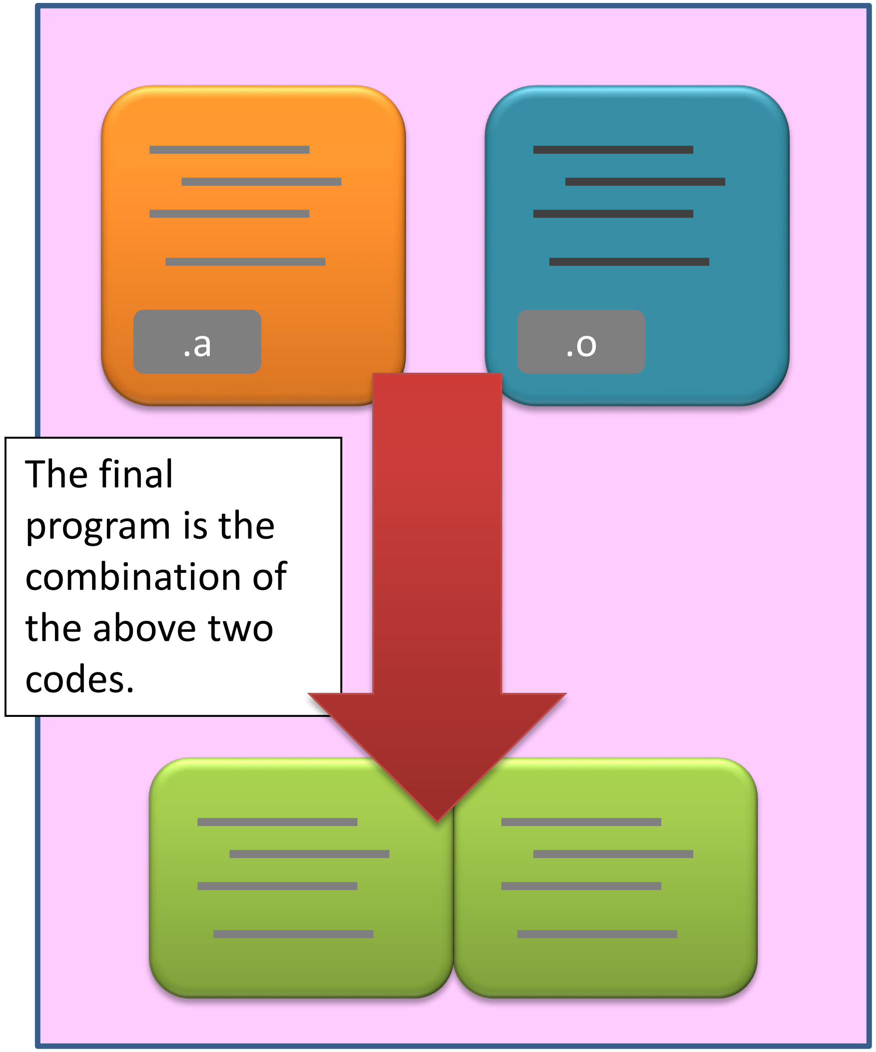


A bunch of "*dot-o*" files.

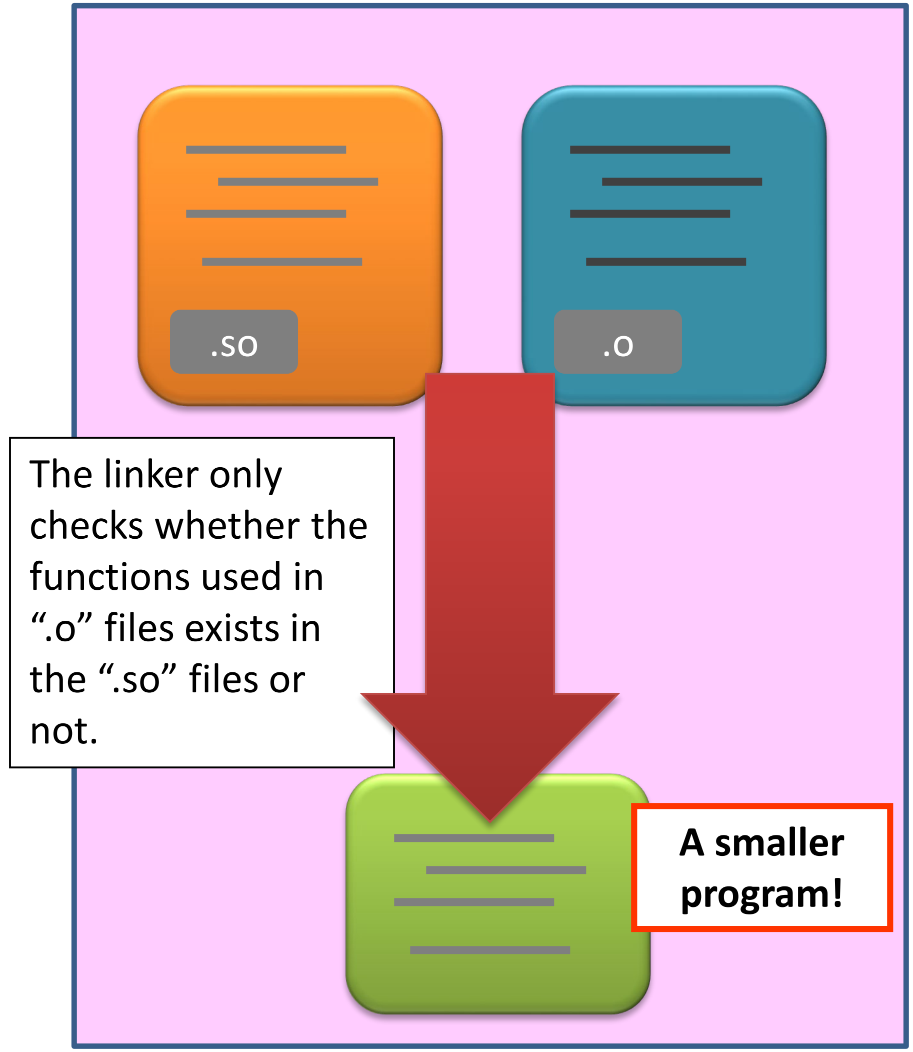
Shared library with "**.so**" file extension.

A static library with "**.a**" file extension. It is also called an archive.

Sidetrack: Library files



Linking with static library file.



Linking with dynamic library file.

How to compile multiple files?

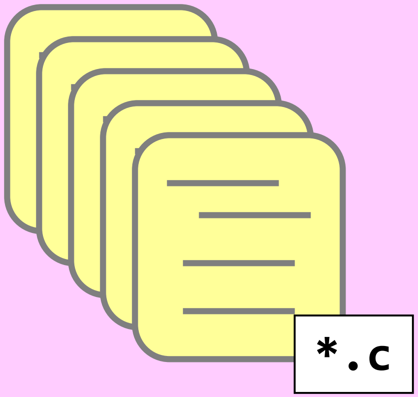
- **`gcc`** by default hides all the intermediate steps.
  - Executable: "**`gcc -o hello hello.c`**" generates "**`hello`**" directly.
  - Object code: "**`gcc -c hello.c`**" generates "**`hello.o`**" directly.

- How about working with multiple files?

How to compile multiple files?

Remember, below shows one of the solution.

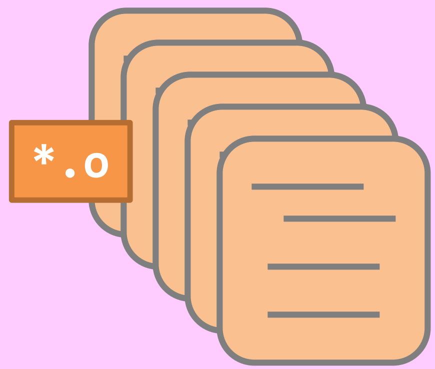
Step 1.



Prepare all the source files.
Important: there must be one and only one file containing the main function.

Step 2.

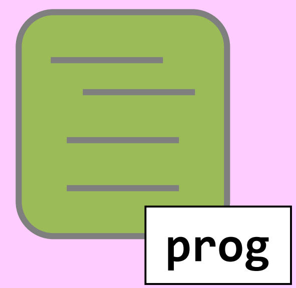
```
$ gcc -c code.c
......
```



Compile them into object codes one by one.

Step 3.

```
$ gcc -o prog *.o
```



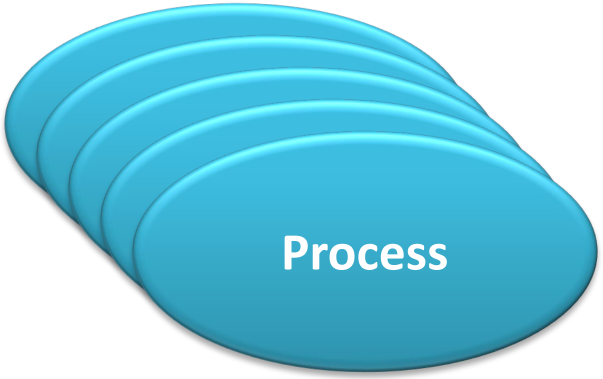
Construct the program together with all the object codes.

Conclusion on "*what is a program?*"

- A program is just an executable file!
  - It is static;
  - It may be associated with dynamically-linked files;
    - "<u>*.so</u>" in Linux and "<u>*.dll</u>" in Windows.

- It may be compiled from more than one file

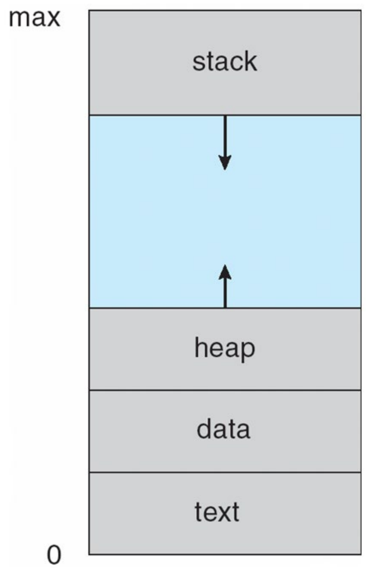
What is a process?



Process in Memory

- A process is a program in execution
  - A program (an executable file) becomes process when it is loaded into memory
  - Active
- Process in memory
  - Text section
  - Stack
  - Heap
  - Data section
  - Program counter
  - Contents of registers

Process in Memory



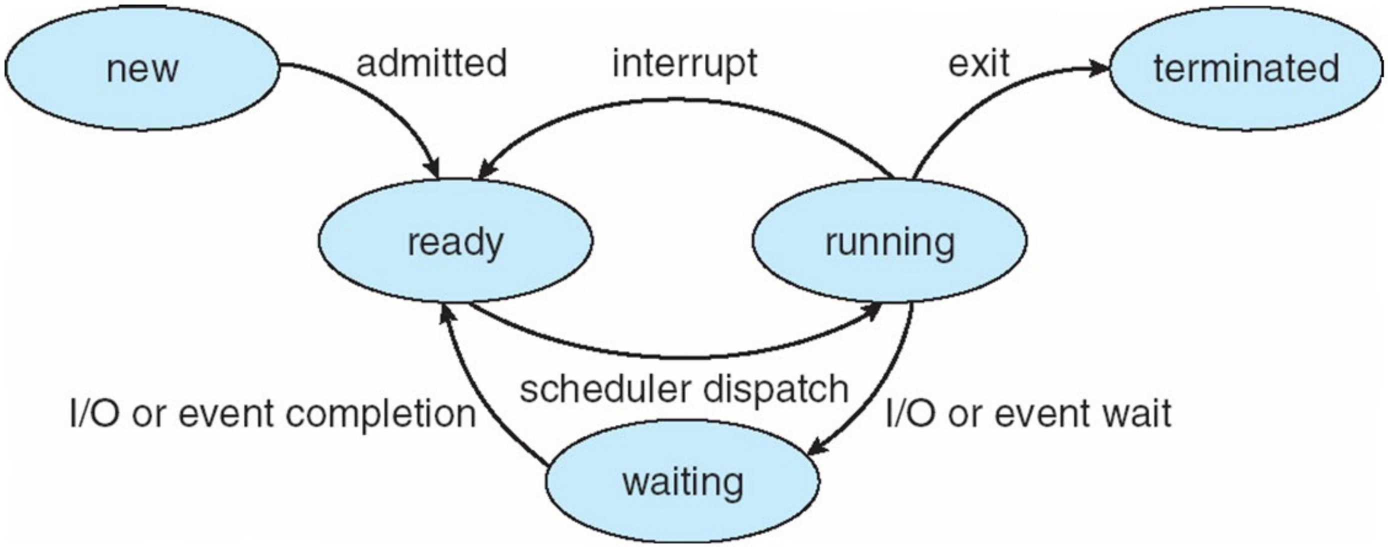
- Text section
  - Program code
- Data section
  - Global variables
- Stack
  - Temporary data (function parameters, return addresses, local variables)
- Heap
  - Dynamically allocated memory during process run time
- Program counter and contents of registers

Process State

- As a process executes, it changes state, which is defined in part by the current activity
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
    - I/O completion or reception of a signal
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

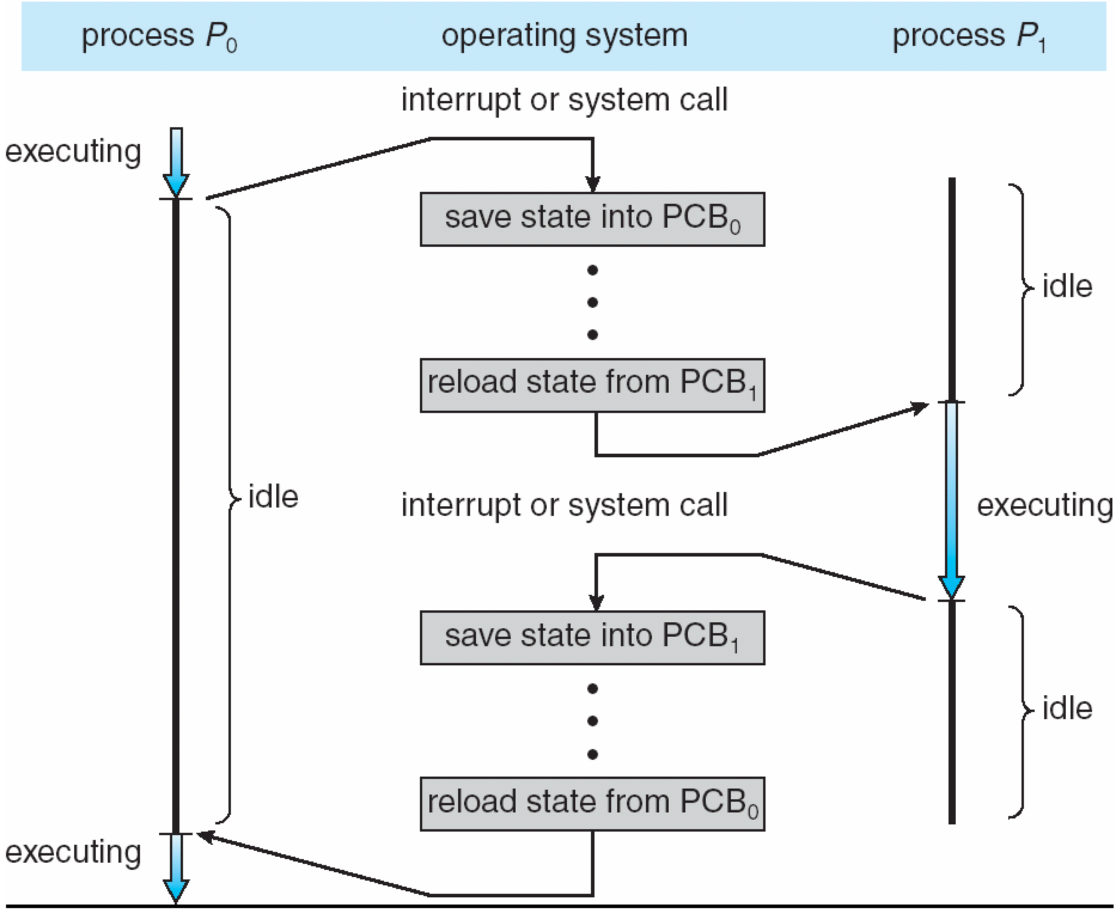
Diagram of Process State

- State diagram



- Only one process can be running on any processor at any instant
- Many processes may be ready and waiting

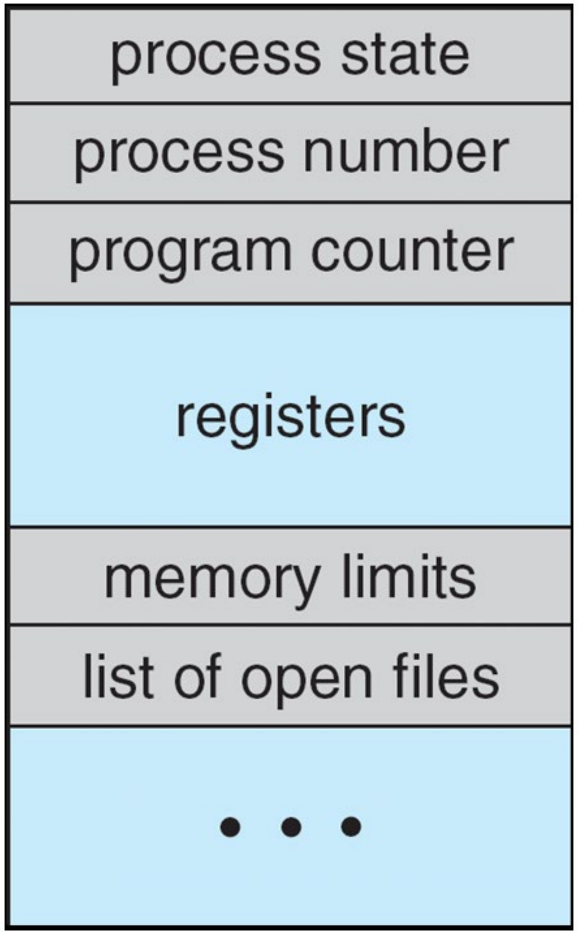
How to locate/represent a process?



Example: CPU switch from process to process

Process Presentation (Data Structure)

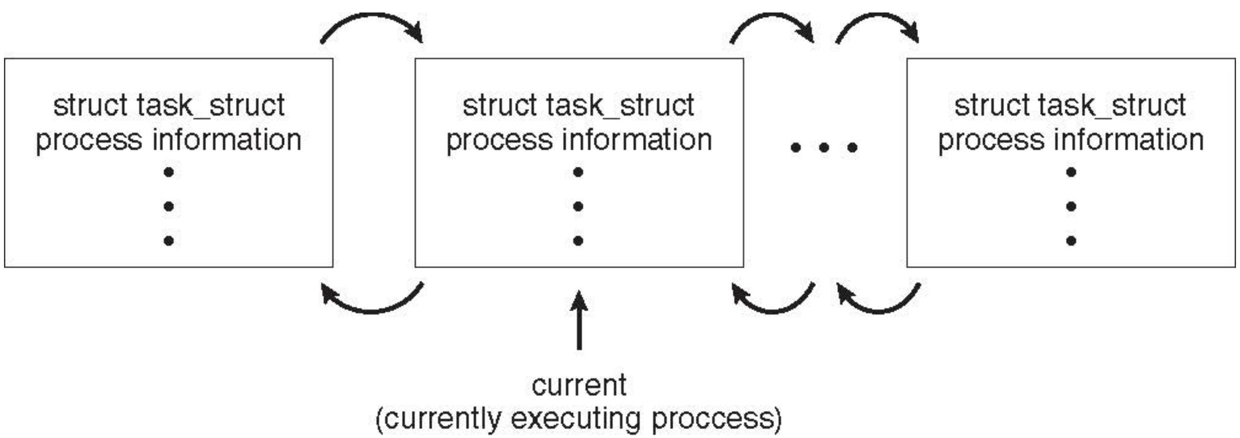
- Process control block (PCB) or task control block
  - Process state (running, waiting, etc)
  - Program counter
    - location of next instruction to execute
  - CPU registers
    - contents of all process-centric registers
  - CPU scheduling information
    - priorities, scheduling queue pointers
  - Memory-management information
    - memory allocated to the process
  - I/O status information
    - I/O devices allocated to process, list of open files
  - Accounting information
    - CPU used, clock time elapsed since start, time limits



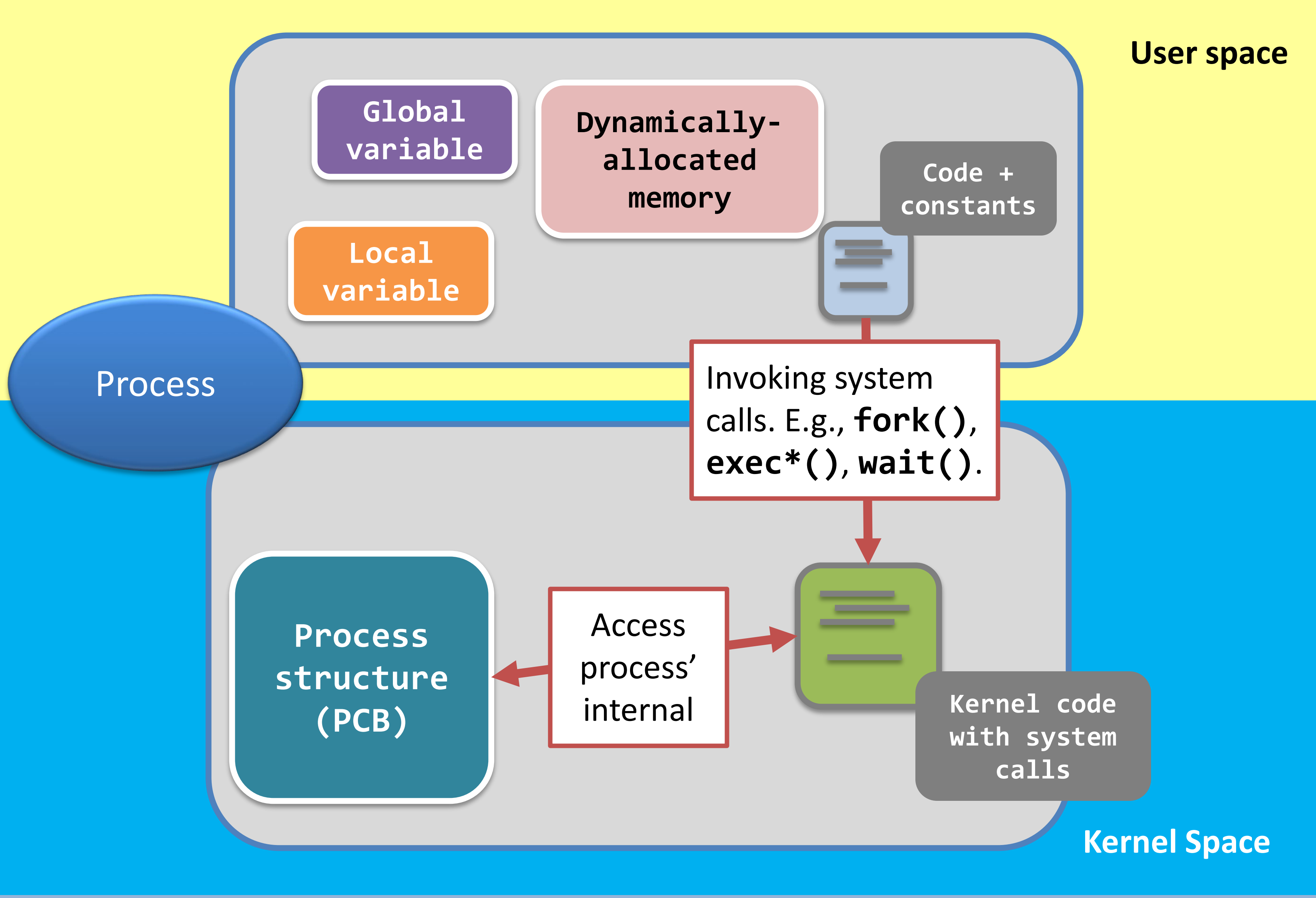
Process Data Structure in Linux

- Represented by C structure `task_struct`
  - `<linux/sched.h>`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Relationship between Process Data & PCB



Conclusion on "*what is a process?*"

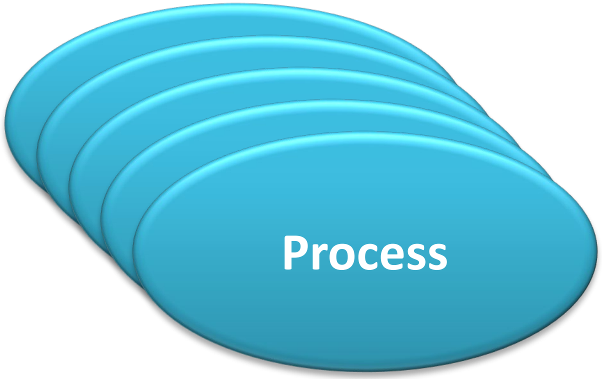
- A process is a program in execution
  - process (active entity) != program (static entity)
  - Why active?
    - A program counter specifying the next instruction to execute + a set of associated resources

- Only one process can be running on any processor at any instant

Conclusion on “*what is a process?*”

- Two processes maybe associated with the same program (Two users are running the same program)
  - Example
    - The same user invokes two copies of the web browser
  - Separate execution sequences
    - The text section may be equivalent
    - The data, heap, and stack sections vary
- A process can be an execution environment for other code
  - Java programming environment
  - `java Program` (`java` runs JVM as a process)

Process Operations



Process Operations

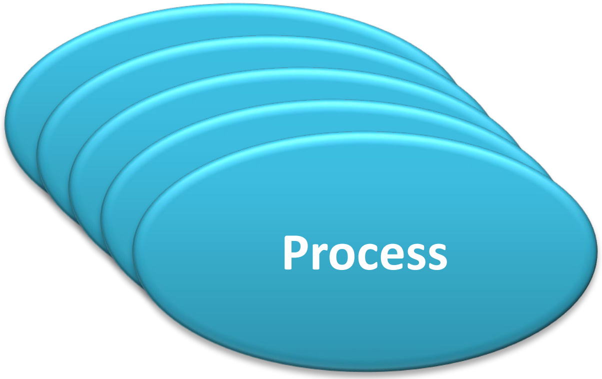
- Process
  - It associates with all the files opened by that process.
  - It attaches to all the memory that is allocated for it.
  - It contains every accounting information,
    - running time, current memory usage, who owns the process, etc.

- You couldn’t operate any things without processes.

Process Operations

- System must provide mechanisms for:
  - process identification
  - process creation
  - program execution
  - process termination
- Some <u>basic</u> and <u>important</u> system calls
  - **`getpid()`**
  - **`fork()`**
  - **`exec*()`**
  - **`wait()`**
  - **`exit()`**

Process Operations
- process identification



Process identification

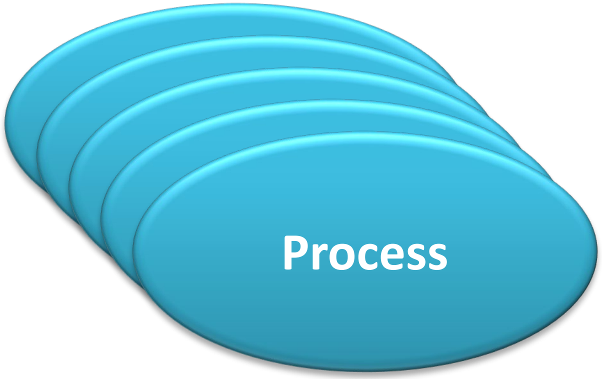
- How can we identify processes?
  - Each process is given an unique ID number, and is called the **process ID**, or the **PID**.
  - The system call, **`getpid()`**, prints the PID of the calling process.

```c
#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 Operations

- process identification
- process creation

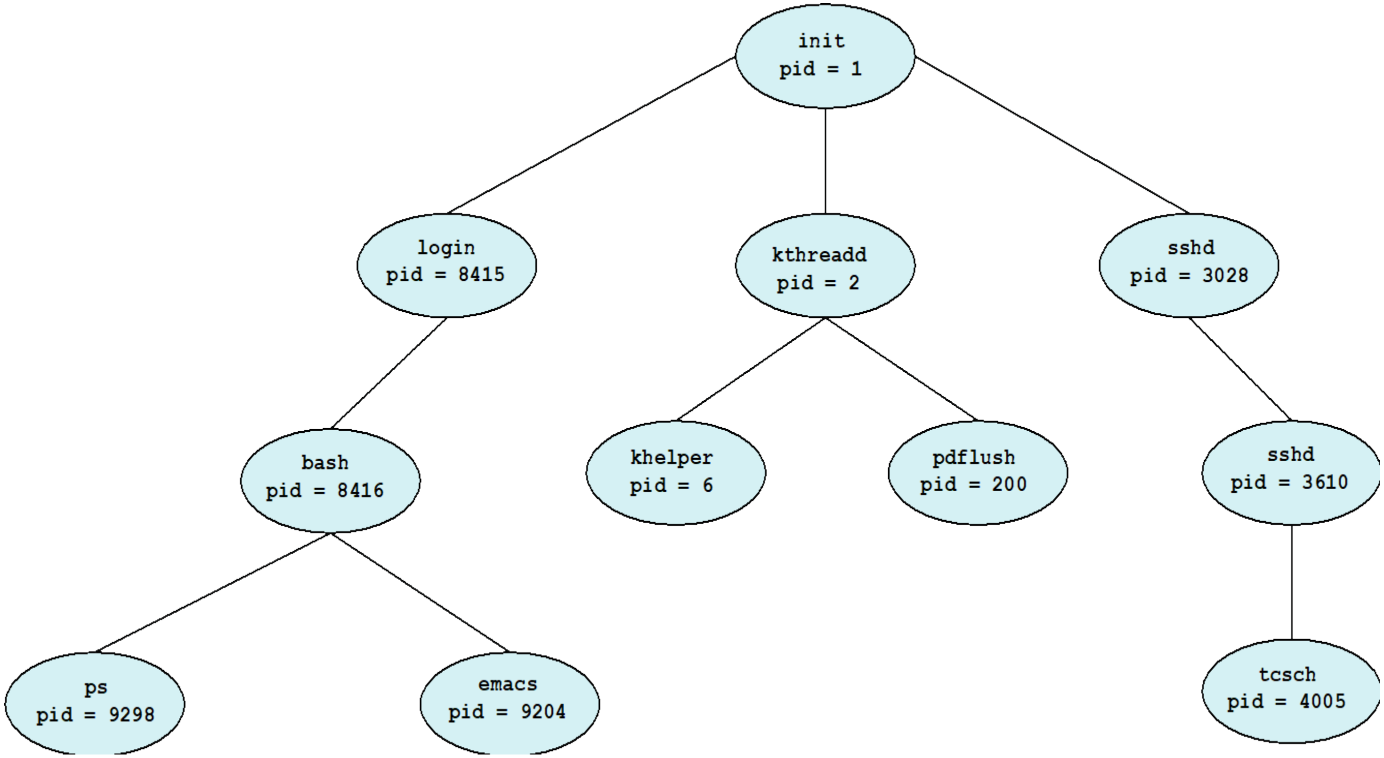


Process Creation

- A process may create several new processes
  - **Parent** process: the creating process
  - **Children** processes: the new processes
- The first process
  - The kernel, while it is booting up, creates the first process – **`init`**.
  - The "**`init`**" process:
    - has **PID = 1**, and
    - is running the program code "**`/sbin/init`**".
  - Its first task is to **create more processes**…

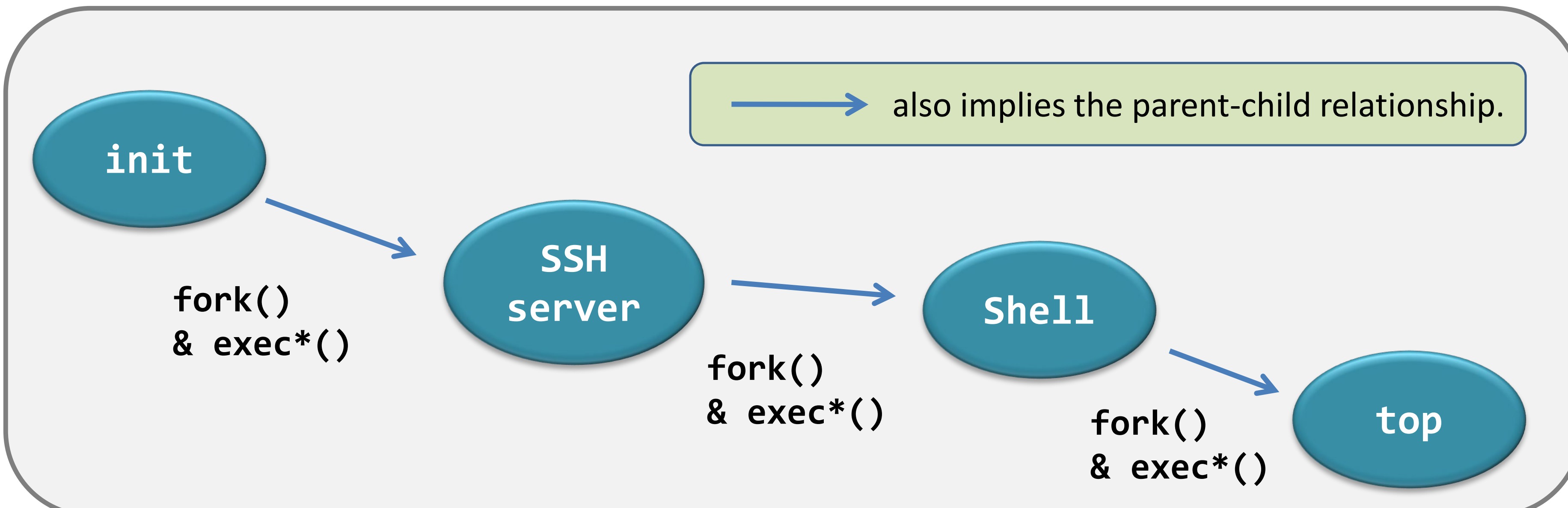
Process Creation

- Tree hierarchy
  - Each of the new process may in turn create other processes, and form a tree hierarchy



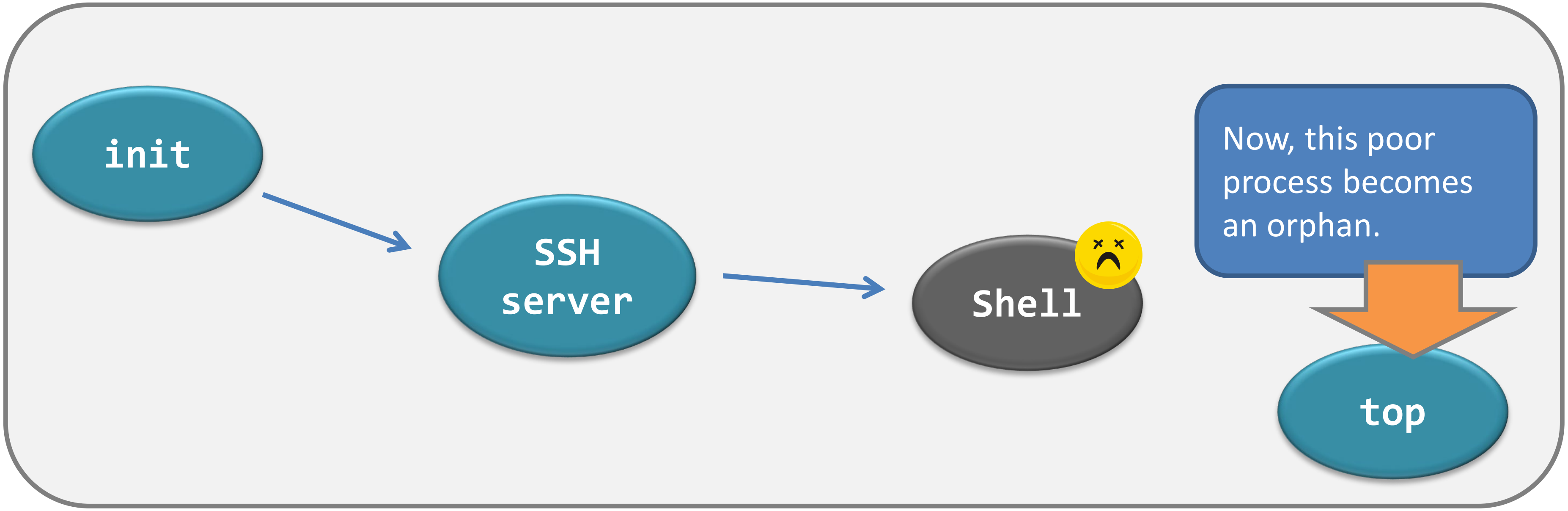
Process blossoming

- You can view the tree with the command:
  - “**`pstree`**”; or
  - “**`pstree -A`**” for ASCII-character-only display.



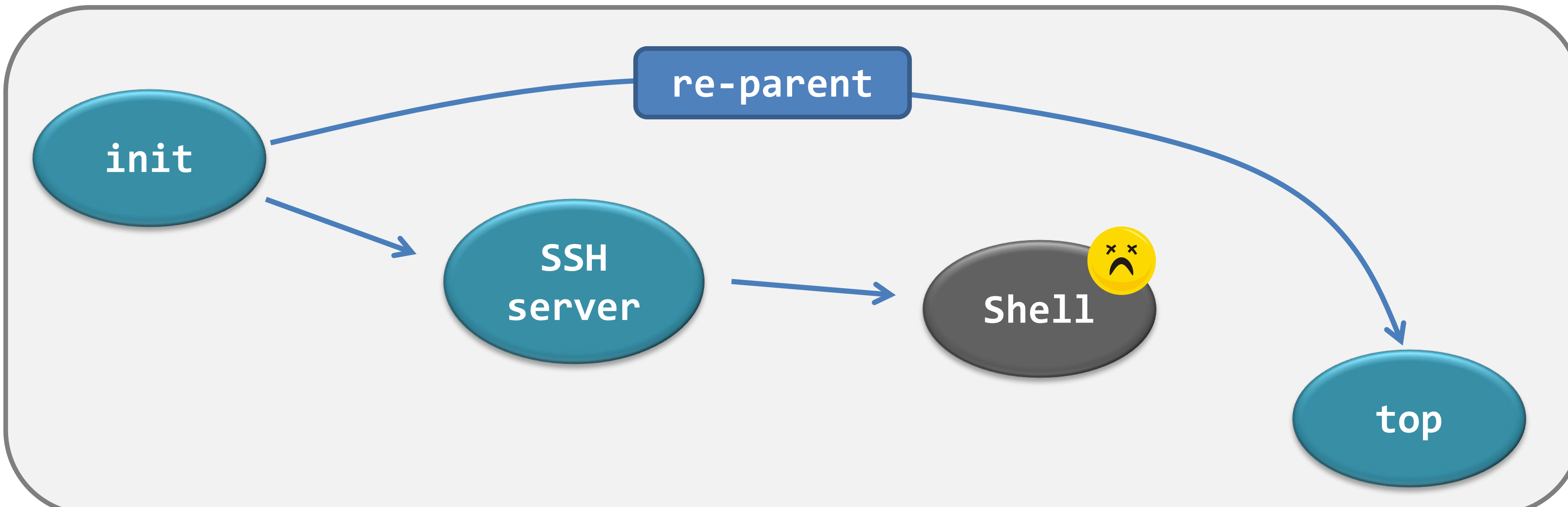
Process blossoming…with orphans?

- However, termination can happen, at any time and in any place…
  - All the resources are deallocated to OS when a process terminates
  - A process may become an orphan when its parent terminated
  - An orphan turns the hierarchy from a **tree** into a **forest**!
  - Plus, no one would know the termination of the orphan.



Process blossoming…with re-parent!

- In Linux…
  - We have the **re-parent operation**.
  - The "**`init`**" process will become the step-mother of all orphans.
- Well…Windows maintains a *forest-like* hierarchy.



A short summary

- **Observation 1**
  - The processes in Linux is always organized as a tree.
  - Because of the re-parent operation, there is always **only one process tree**.

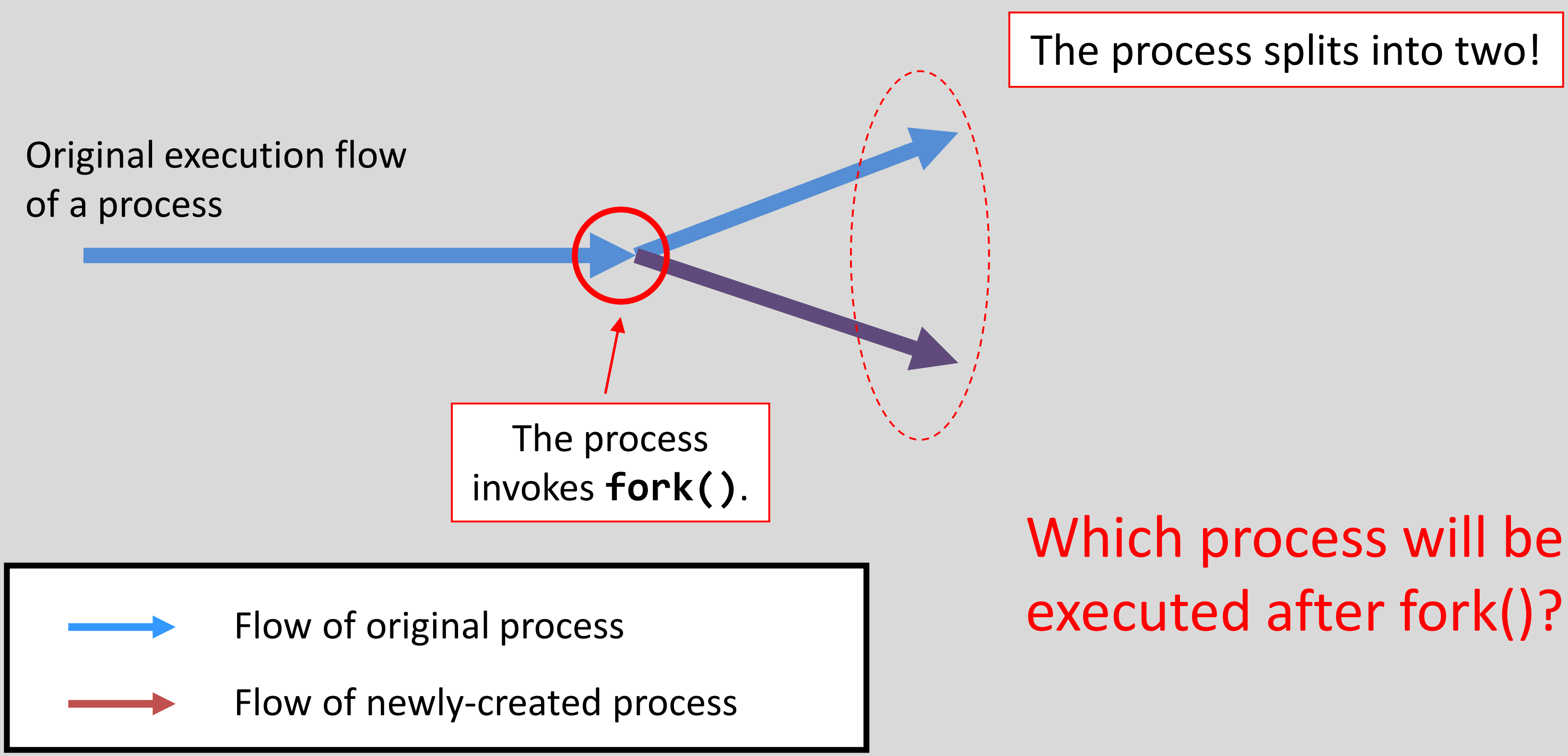
- **Observation 2**
  - The re-parent operation allows processes running **without the need of a parent terminal**.
  - Thus, the **background jobs** survive even though the hosting terminal is closed.

Relationship between Parent and Child

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space options
  - Child is a duplicate of parent
  - Child has a new program loaded into it
- We focus on UNIX examples to illustrate

Process creation

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



Which process will be executed after fork()?

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
$ _
```

PID 1234

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

PID 1235

```
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**.

Why is this line of code executed twice?

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

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

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

What do we know so far?

-Both the parent and the child execute **the same program before and after `fork()`**.
-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

One more example

```
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 ...
```

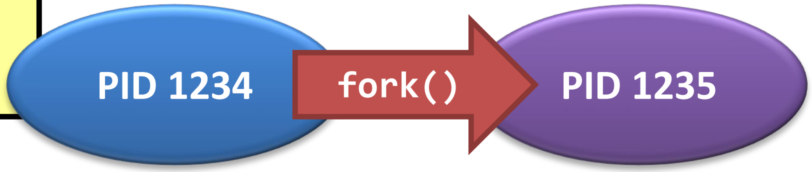
PID 1234

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

One more example

```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 ...
```



Process creation – `fork()` system call

Assumption

Let there be only **<u>ONE CPU</u>**. Then…
- <u>Only one process</u> is allowed to be executed at one time.
- However, we can't predict which process will be chosen by the OS.
- By the time, this mechanism is called **process scheduling**.

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

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
```

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

```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.
```

PID 1234
(dead)

PID 1235
(waiting)

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)

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
I'm the child.
My PID is 1235
program terminated.
$ _
```

PID 1234
(dead)

PID 1235
(dead)

Process creation – `fork()` system call

- **`fork()`** behaves like "*cell division*".
  - It creates the child process by **cloning** from the parent process, including…

Cloned items	Descriptions
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.
Program counter [CPU register]	**That's why they both execute from the same line of code after fork() returns.**

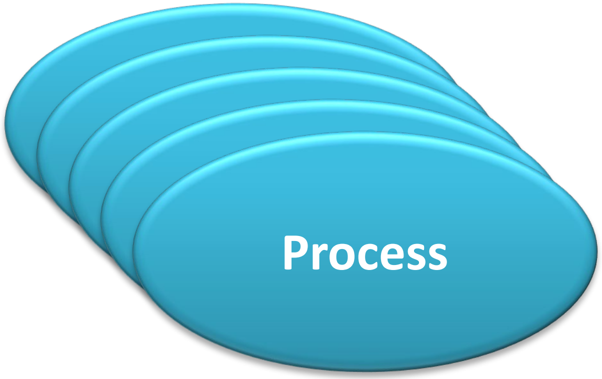
Process creation – `fork()` system call

- However...
  - `fork()` does not clone the following...
  - Note: they are all data inside the memory of kernel.

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.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.

Process Operations

- process identification
- process creation
- **program execution**



`fork()` can only duplicate…

- **`fork()`** is rather **boring**…
  - If a process can only duplicate itself and always runs the same program, then…
  - how can we execute other programs?

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

Program execution

- **`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 ...
```

<u>Arguments of the `execl()` call</u>

1st argument: the program name, “**`/bin/ls`**” in the example.
2nd argument: 1st argument to the program.
3rd argument: indicate the end of the list of arguments.

Program execution

- **`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 ...
```

<u>What is the output?</u>

The same as **<u>the output of running “`ls`” in the shell.</u>**

Program execution

- **`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
```

Program execution

- 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 **first program argument**.
3	`NULL`	This states the end of the program argument list.

Program execution

- 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 **first program argument**.
3	**"-l"**	When the process switches to **"/bin/ls"**, this string is the **second program argument**.
4	**NULL**	This states the end of the program argument list.

Program execution

- **`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
```

GUESS:
What happens next?

Program execution

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

WHAT?!
The shell prompt appears!

```c
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?!

Program execution

- 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**”.

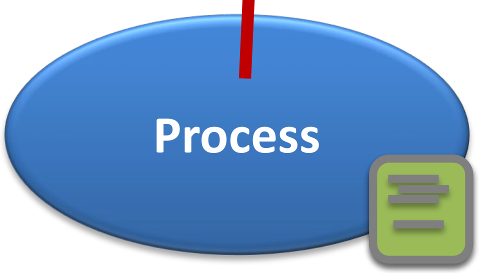
Program execution

- 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;
}
```

The execl() call changes the execution from "**exec_example**" to "**/bin/ls**"

```
/* The program "ls" */

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



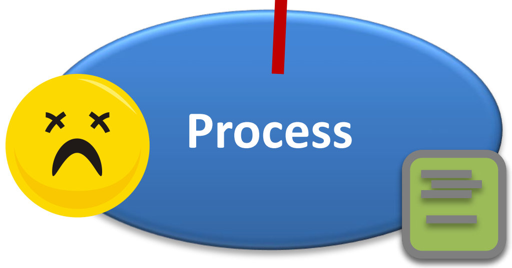
Program execution

- 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);
}
```



Program execution - observation

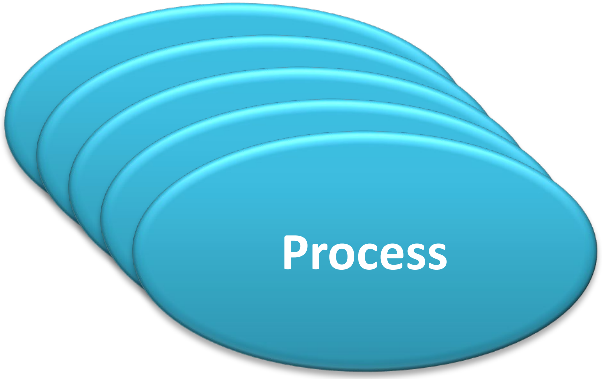
- The process is changing the code that is executing and **never returns to the original code**.
  - The last two lines of codes are therefore not executed.

- The process that calls any one of the member of the exec system call family will **throw away** many things, e.g.,
  - Memory: local variables, global variables, and dynamically allocated memory;
  - Register value: e.g., the program counter;

- But, the process will **preserve** something, including:
  - PID;
  - Process relationship;
  - Running time, etc.

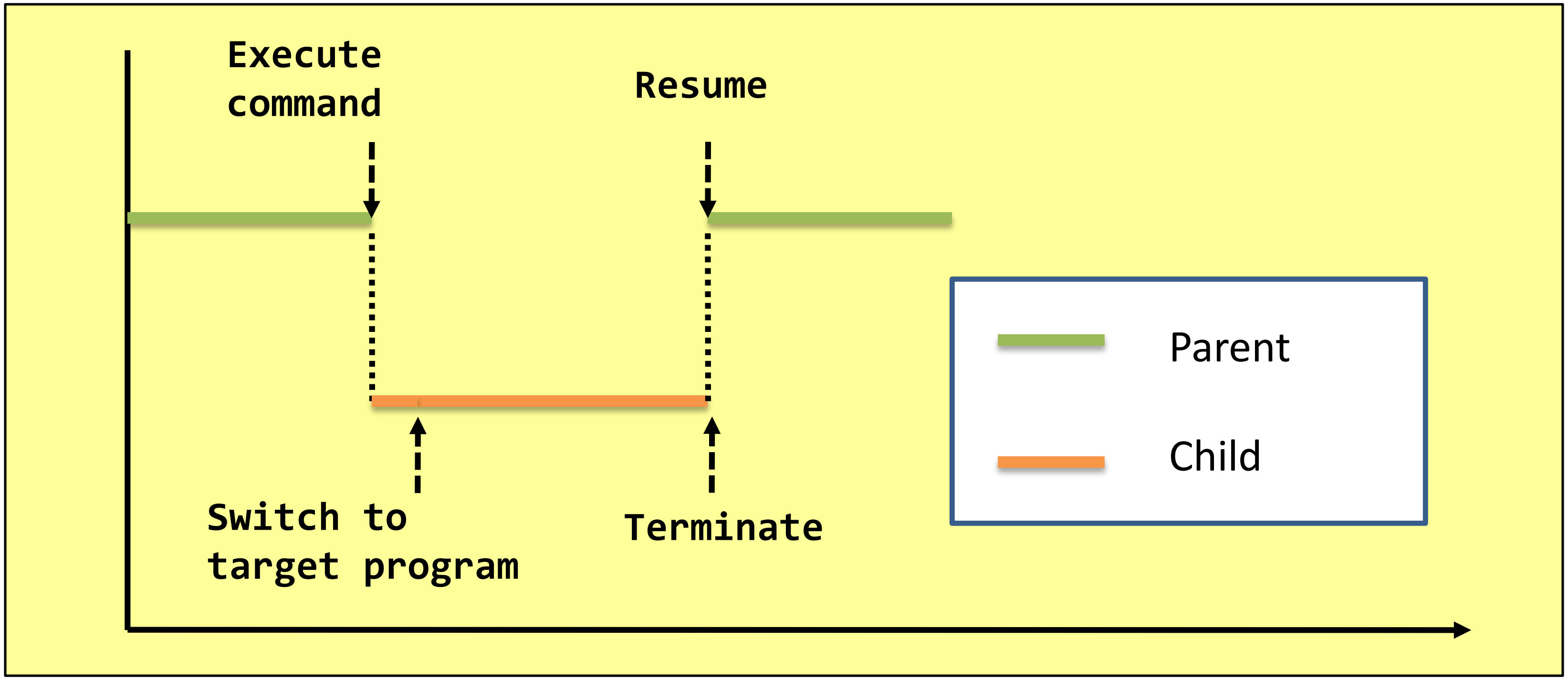
Process Operations

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



When **fork()** meets **exec*()**…

- The mix can become:
  - A shell,
  - The **system()** library call, etc…



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

```c
int system_test(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_test("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

Is this the only result?

```
$ ./system_implement_1
before...

system_implement_1
system_implement_1.c

after...
$ _
```

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

```c
int system_test(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_test("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

Some strange cases happened when the program is **executed repeatedly**!! Why?

```
$ ./system_implement_1
before...

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

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

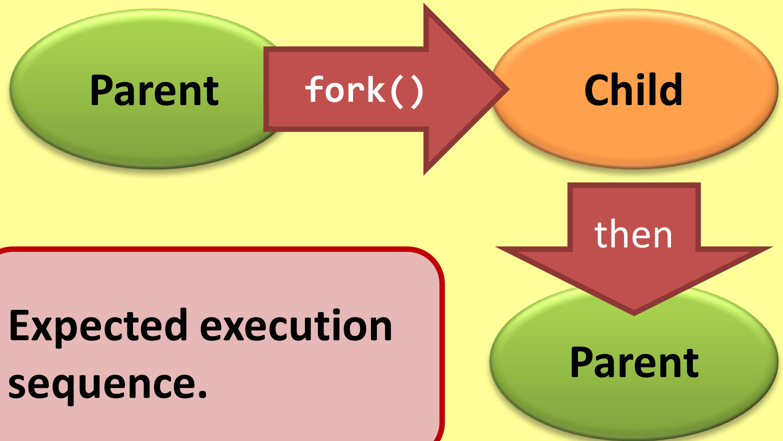
```c
int system_test(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl(cmd_str, cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    return 0;
}

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

Let's re-color the program!

- Parent process
- Child process
- Both processes

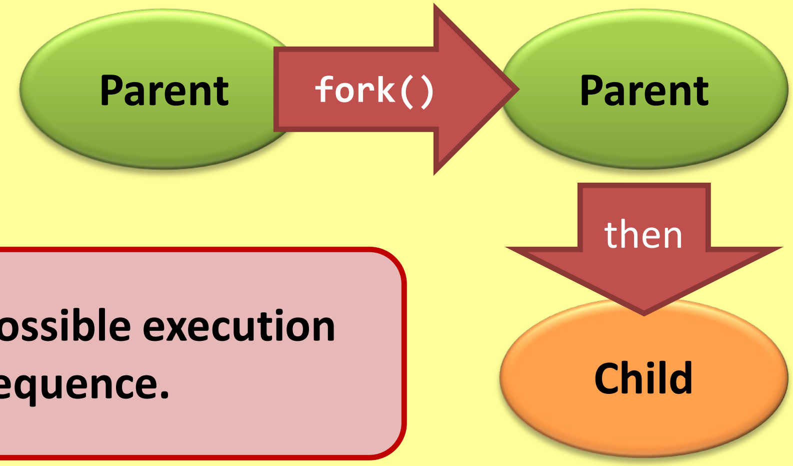
```
$ ./system_implement_1
before...

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

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*()

Is it enough?

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

- Don’t forget that we’re trying to implement a **system()**-compatible function…
  - It is very weird to allow different execution orders.

- How to let the child to execute first?
  - But…we can’t control the **process scheduling** of the OS to this extent.

- 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_test(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_test("/bin/ls");
    printf("\nafter...\n");
    return 0;
}
```

What is the output now?

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

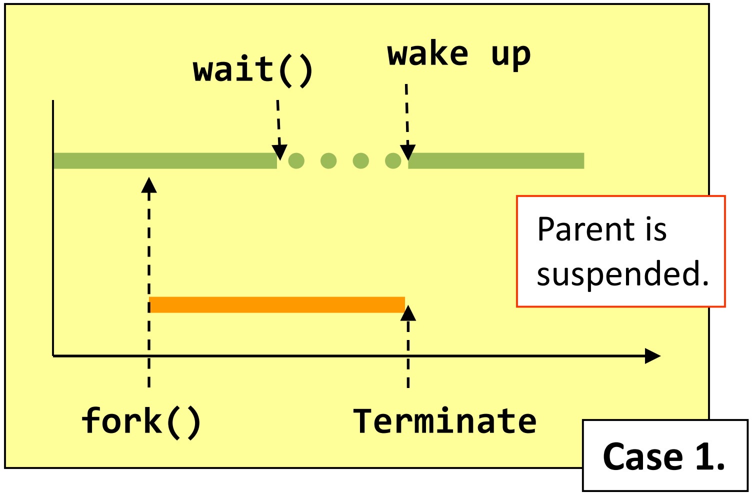
```c
int system_test(const char *cmd_str) {
    if(cmd_str == -1)
        return -1;
    if(fork() == 0) {
        execl("/bin/sh", "/bin/sh",
              "-c", cmd_str, NULL);
        fprintf(stderr,
           "%s: command not found\n", cmd_str);
        exit(-1);
    }
    wait(NULL);
    return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

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

The parent is suspended until the child terminates

```
$ ./system_implement_2
before...

system_implement_2
System_implement_2.c

after...
$ _
```

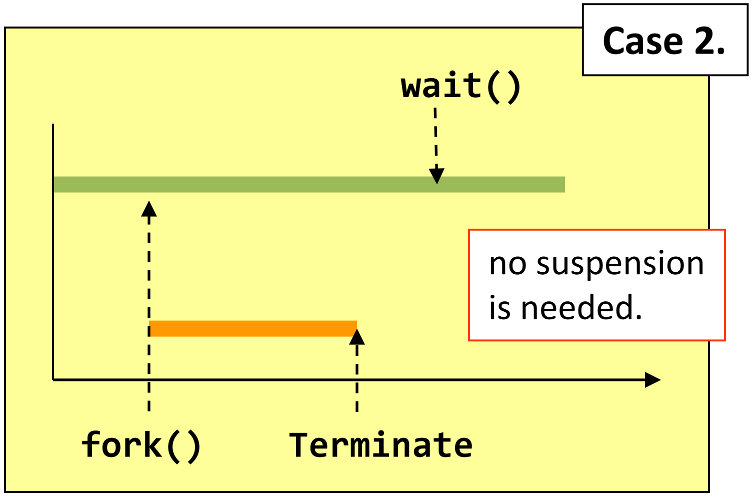
`wait()` – properties explained

- The **`wait()`** system call **suspend** the calling parent process (Case 1).
- When to wake up?
  - **`wait()`** returns and wakes up the calling process when the one of its child processes changes from **running to terminated**.



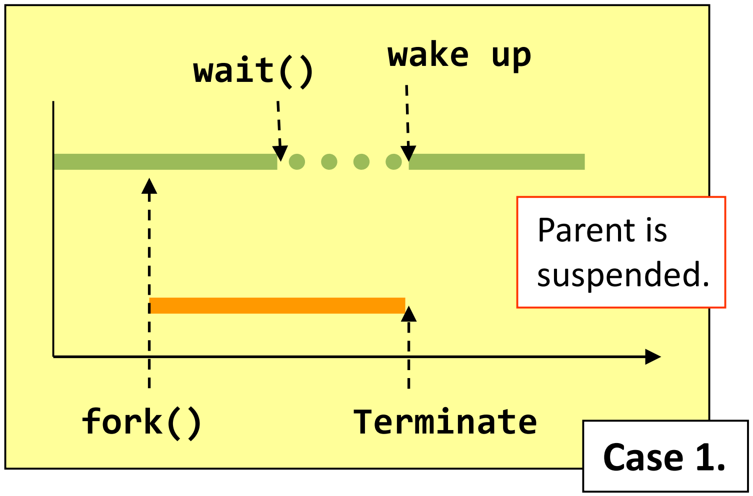
`wait()` – properties explained

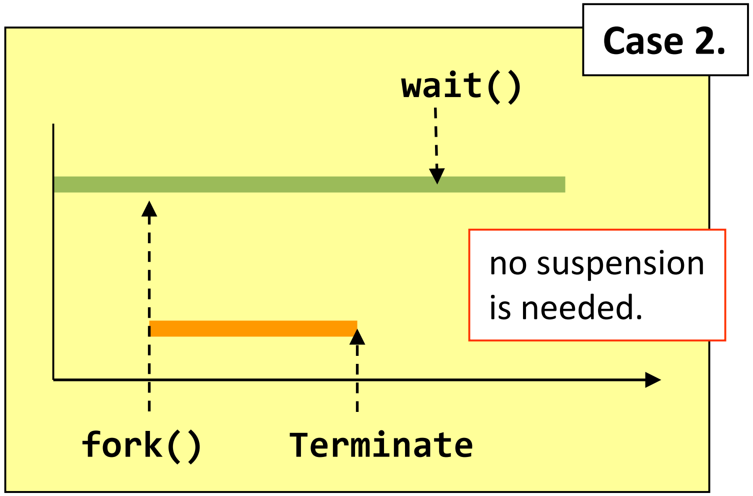
- What happens if
  - There were no running children;
  - There were no children;
- **`wait()`** does not suspend the calling process (Case 2)



`wait()` – summary

- The **`wait()`** system call **suspend** the calling parent process (Case 1).
- **`wait()`** returns and wakes up the calling process when the one of its child processes changes from **running to terminated**.
- **`wait()`** does not suspend the calling process (Case 2) if
  - There were no running children;
  - There were no children;





More powerful `wait()`?

- Limitation of **`wait()?`**
  - waits for any one of the children
  - Detect child termination only
- How to wait for a particular process?
  - **`waitpid()`**

wait() VS waitpid()

`wait()`	`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 child's status changing**: -from running to suspended, and -from suspended to running.

For more details, you **must read** the man pages of **`wait()`** and **`waitpid()`**.

Summary of Process Operations

- A process is created by cloning
  - **`fork()`** is the system call that clones processes
  - Cloning is copying
    - What are inherited?
    - What are not?
    - Metaphor of father-son relationship
  - **`wait()`** can be used to suspend the parent process, so as to guarantee the expected execution sequence
- Program execution is fundamental, but not trivial
  - A process is the place that hosts a program and run it
  - **`exec()`** system call family changes the program that a process is running.
  - A process can run more than one program…
    - as long as there is a set of programs that keeps on calling the **exec** system call family.

Summary of Ch3

- Concepts
  - Process data in memory
  - PCB
- Operations
  - `fork(), exec*(), wait()`
  - Just introduced how they could be used to create processes and execute programs
  - How about the internal working of these system calls?
    - How does the kernel behaves when calling these system calls?

End of Chapter 3