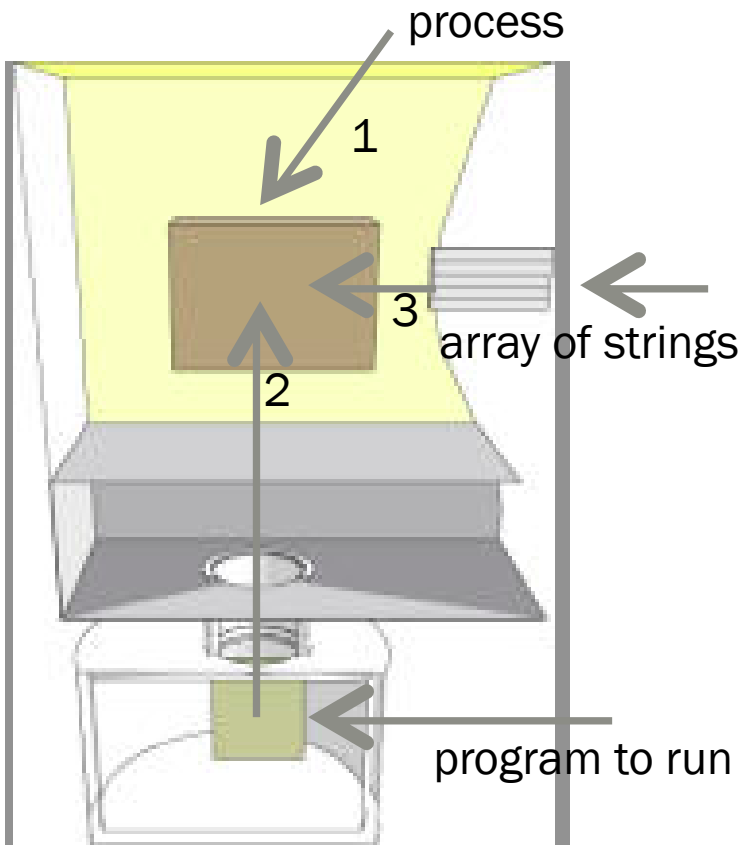
Reading Reference:
Textbook 1 Chapter 8

PROCESS API

Tanzir Ahmed
CSCE 313 Spring 2021

Running Another Program from a C/C++ program

- Say, the other program's name is **name**
- The current program makes a system call
  `exec("name", arglist)`
- Kernel loads the "**name**" executable program from disk into the process
- Kernel copies **arglist** into the process
- Kernel calls **main(arglist)** of the **name** program



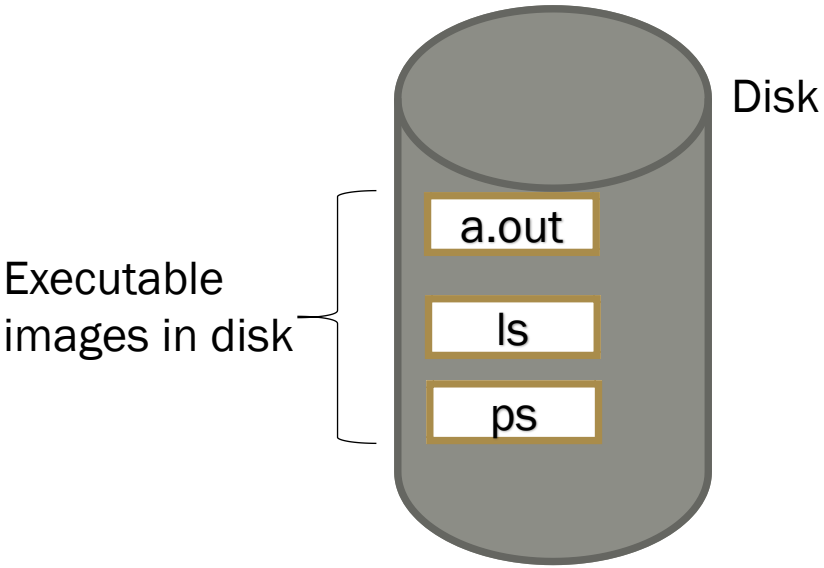
Example: One Program Running Another

```
int main() {
        char* args [] = {"ls", "-l", "-a", NULL};
        cout << "=====BEFORE========"<<endl;
        execvp (args[0], args);
        cout << "======AFTER========"<<endl;
}
```

```
prompt> ./a.out
=====BEFORE========
total 40
drwxr-xr-x  2 osboxes osboxes  4096 Sep  2 14:22 .
drwxr-xr-x 21 osboxes osboxes  4096 Sep  2 10:27 ..
-rwxrwxr-x  1 osboxes osboxes 17408 Sep  2 14:22 a.out
-rw-rw-r--  1 osboxes osboxes   256 Sep  2 14:21 exec1.cpp
-rw-rw-r--  1 osboxes osboxes   288 Sep  1 23:08 fork1.cpp
-rw-rw-r--  1 osboxes osboxes   496 Sep  2 14:01 fork2.cpp
prompt>
```

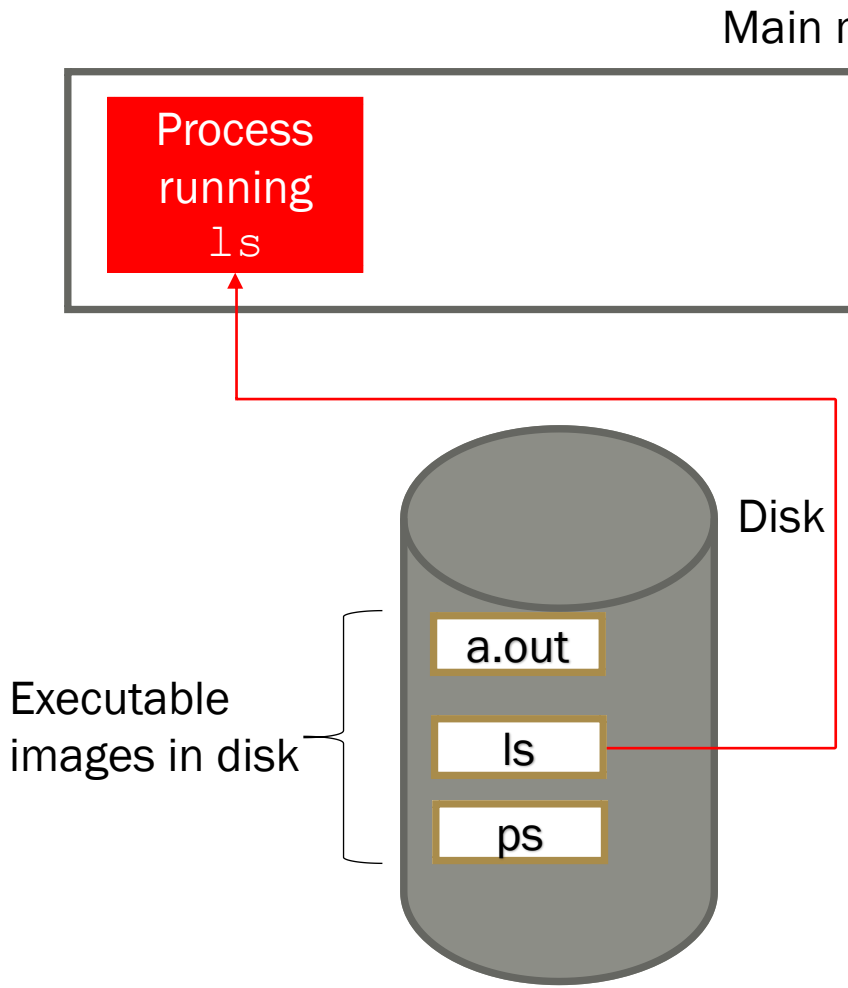
What happens in exec()

- Initial state of memory before hitting line: `exec("ls")`



What happens in exec() contd

- Initial state of memory before after line: `exec("ls")`



The same process container is overwritten with ls's image

Example: contd.

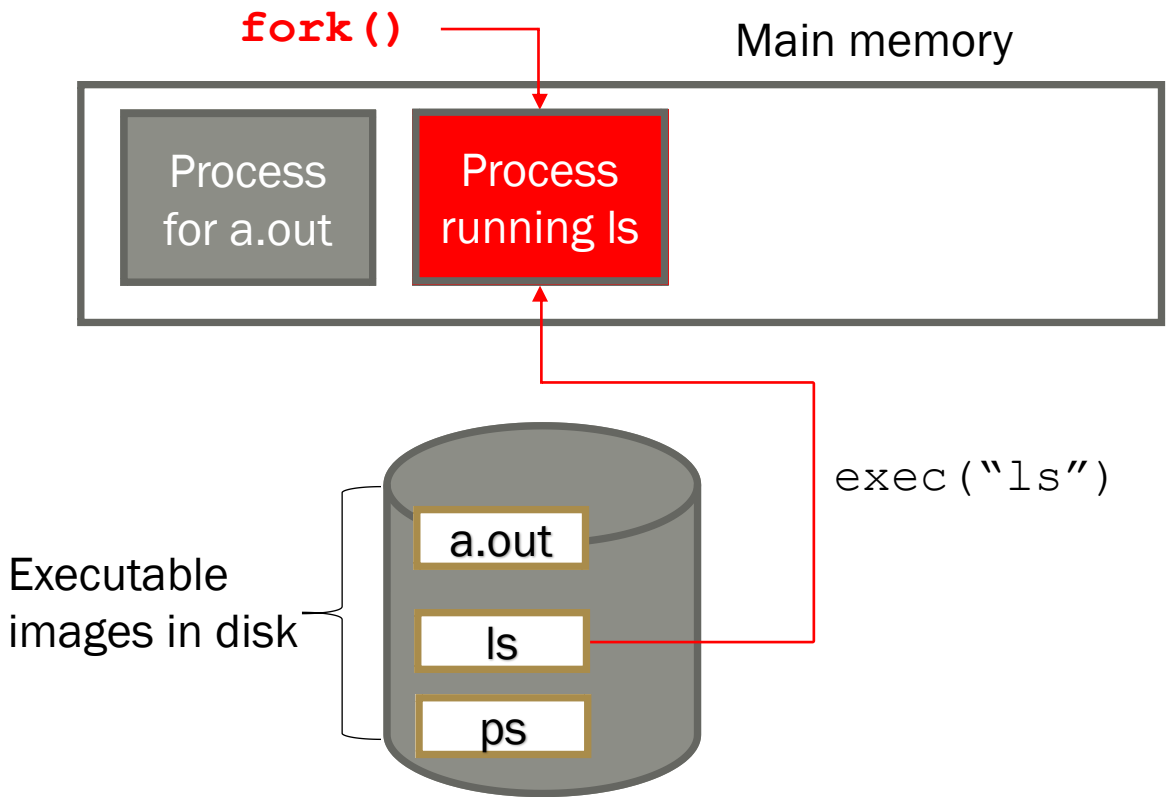
```
int main() {
        char* args [] = {"ls", "-l", "-a", NULL};
        cout << "=====BEFORE========"<<endl;
        execvp (args[0], args);
        cout << "======AFTER========"<<endl;
}
```

```
prompt> ./a.out
=====BEFORE========
total 40
drwxr-xr-x  2 osboxes osboxes  4096 Sep  2 14:22 .
drwxr-xr-x 21 osboxes osboxes  4096 Sep  2 10:27 ..
-rwxrwxr-x  1 osboxes osboxes 17408 Sep  2 14:22 a.out
-rw-rw-r--  1 osboxes osboxes   256 Sep  2 14:21 exec1.cpp
-rw-rw-r--  1 osboxes osboxes   288 Sep  1 23:08 fork1.cpp
-rw-rw-r--  1 osboxes osboxes   496 Sep  2 14:01 fork2.cpp
prompt>
```

- Where is the second message?
  - *The exec system call clears out the machine language code of the current program from the current process and then in the now empty process puts the code of the program named in the exec call and then runs the new program*
- execvp does not return if it succeeds
- exec is like a brain transplant

To Avoid Image Overwrite

- First, if we are running shell, we need to continue having the shell image intact
  - *Otherwise it is out after exec()ing the first process*
- We need another function to create a separate Process container first using another function `fork()`, and then call `exec("ls")`



Creating New Processes

- The following program creates a new process by invoking system call **`fork():`**
  - *It also uses system calls* **`getpid()`** *to get the calling process's ID and* **`getppid()`** *for the parent's ID*

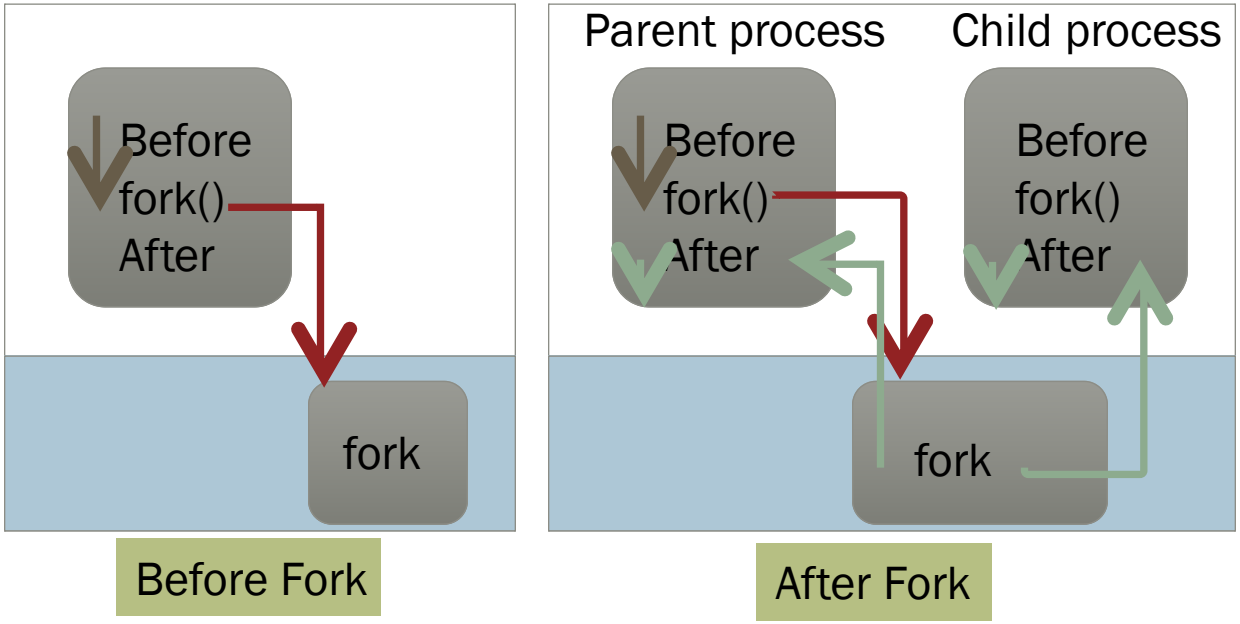
```
int main() {
        cout <<"Hello!! My ID="<<getpid()<<", my parent ID="<<getppid()<< endl;
        pid_t pid = fork();
        cout << "Bye!! My ID="<<getpid()<<", my parent ID="<<getppid()<< endl;
        return 0;
}
```

  - *The following is the output when run twice:*

```
prompt> ./a.out
Hello!! My ID= 3108, my parent ID=3101
Bye!! My ID= 3108, my parent ID=3101
Bye!! My ID= 3109, my parent ID=3108
prompt> ./a.out
Hello!! My ID= 3110, my parent ID=3101
Bye!! My ID= 3110, my parent ID=3101
Bye!! My ID= 3111, my parent ID=3110
```

Creating a New Process

- Calling **`fork()`** function



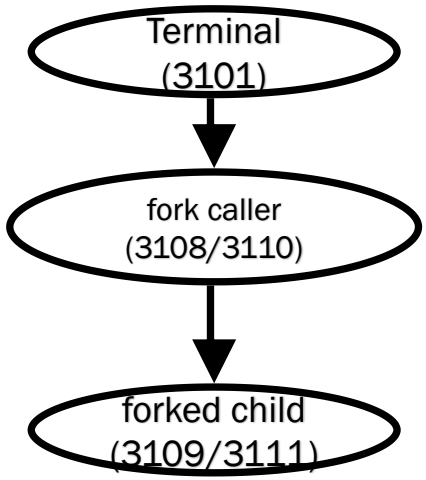
- After a process invokes fork(), control passes to the Kernel, which does the following:
  - Allocates address space and data structures
  - Clones original process into the new process (everything in the PCB, e.g., PC, SP, EFLAGS, regs, file descriptors)
  - Adds the new process to the set of ready-to-run processes
  - Returns control back to both processes

Example: Fork

```
int main() {
        cout <<"Hello!! My ID="<<getpid()<<", my parent
        ID="<<getppid()<< endl;
        pid_t pid = fork();
        cout << "Bye!! My ID="<<getpid()<<", my parent
        ID="<<getppid()<< endl;
        return 0;
}
```

- Because of such cloning, everything after the fork() is executed twice – once from the parent and once from the child
  - *Hence, we see "Hello" once and "Bye" twice*
- The terminal (ID=3101) is parent during both runs
- Process IDs are usually assigned sequentially and recycled after process termination by garbage collector
- After the child is created, the schedule of the parent and children are independent (i.e., either the parent or the child might be scheduled first)
  - *However, we can synchronize/order them*

```
prompt> ./a.out
Hello!! My ID= 3108, my parent ID=3101
Bye!! My ID= 3108, my parent ID=3101
Bye!! My ID= 3109, my parent ID=3108
prompt> ./a.out
Hello!! My ID= 3110, my parent ID=3101
Bye!! My ID= 3110, my parent ID=3101
Bye!! My ID= 3111, my parent ID=3110
```



Controlling Fork()-ed Processes

- Return value depends on the side (i.e., parent or child)
  - *To the parent it returns the child PID*
  - *To the child, it returns 0*
- Can we use this fact to make the child behave differently from the parent? Of course.

```c
#include <stdio.h>
int main (){
   int ret = fork();
   if (ret){
         printf ("Hello from Parent\n");
         printf ("My ID: %d, My Child ID: %d\n", getpid(), ret);
   }else{
         printf ("Hello from the Child\n");
         printf ("My ID: %d, I do not have a child\n", getpid());
   }
}
```

```
:: ./a.out
Hello from Parent
My ID: 6512, My Child ID: 6513
Hello from Child
My ID: 6513, I do not have a child
```

Example

- Parent and child processes have independent address spaces

```
int main(int argc, char *argv[]) {
        int value = 5;
        bool isChild = fork() == 0;
        if (isChild){
                value += 5;
                cout<< "Child has value="<<value<<endl;
        }else{
                value += 10;
                cout<< "Parent has value="<<value<<endl;
        }
}
```

```
prompt> ./a.out
Parent has value=15
Child has value=10
```

Example – with Parent Delayed

- Parent and child processes have independent address spaces

```
int main(int argc, char *argv[]) {
        int value = 5;
        bool isChild = fork() == 0;
        if (isChild){
                value += 5;
                cout<< "Child has value="<<value<<endl;
        }else{
                sleep (1); // forces delay in parent
                value += 10;
                cout<< "Parent has value="<<value<<endl;
        }
}
```

```
prompt> ./a.out
Child has value=10
Parent has value=15
```

Coordinating Processes

- The previous example showed how we were trying to control the schedule by artificial sleep, which clearly is not ideal
- How about a proper synchronization, i.e., one process waiting for another process to finish?
- The following function makes one process wait for another:
  - *Also reaps the child process when the parent does the waiting*

```
pid_t waitpid (pid_t pid, int *status, int options)
```

- The 3 arguments are the following:
  1. *the target process's ID*
  2. *the address of an integer where termination information (i.e., exit code) can be placed. You can pass NULL if we don't care for that.*
  3. *A collection of bitwise-or'ed flags we'll study later. For now, you can just use 0 to block until the target's termination*
- A simpler wait for any child process (not targeted!!) is the following:

```
pid_t wait (int *status)
```

  - *This function waits until **any** of the child processes finish*
  - *Excellent when there are many children and you want to wait for them in the order they finish*

Synchronizing Processes - Example

```
int value = 5;
int pid = fork();
if (!pid){
        waitpid (getppid(), 0, 0); // wait for the parent
        value += 5;
        cout<< "Child has value="<<value<<endl;
}else{
        value += 10;
        cout<< "Parent has value="<<value<<endl;
}
```

```
prompt> ./a.out
Parent has value=15
Child has value=10
```

```
int value = 5, status;
int pid = fork();
if (!pid){
        value += 5;
        cout<< "Child has value="<<value<<endl;
        exit(100);
}else{
        waitpid (pid, &status, 0); // wait for the child
        value += 10;
        cout<<"Child terminated,
        status="<<WEXITSTATUS(status)<<endl;
        cout<< "Parent has value="<<value<<endl;
}
```

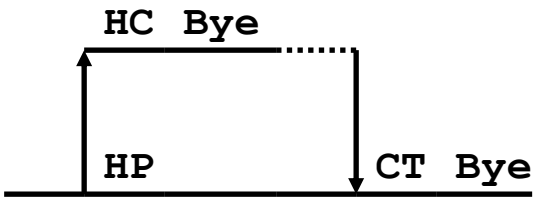
Trying to tell something to the parent

Parent listening

```
prompt> ./a.out
Child has value=10
Child terminated, status=100
Parent has value=15
```

`wait`: Synchronizing With Children

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void wait_demo() {
   int child_status;
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

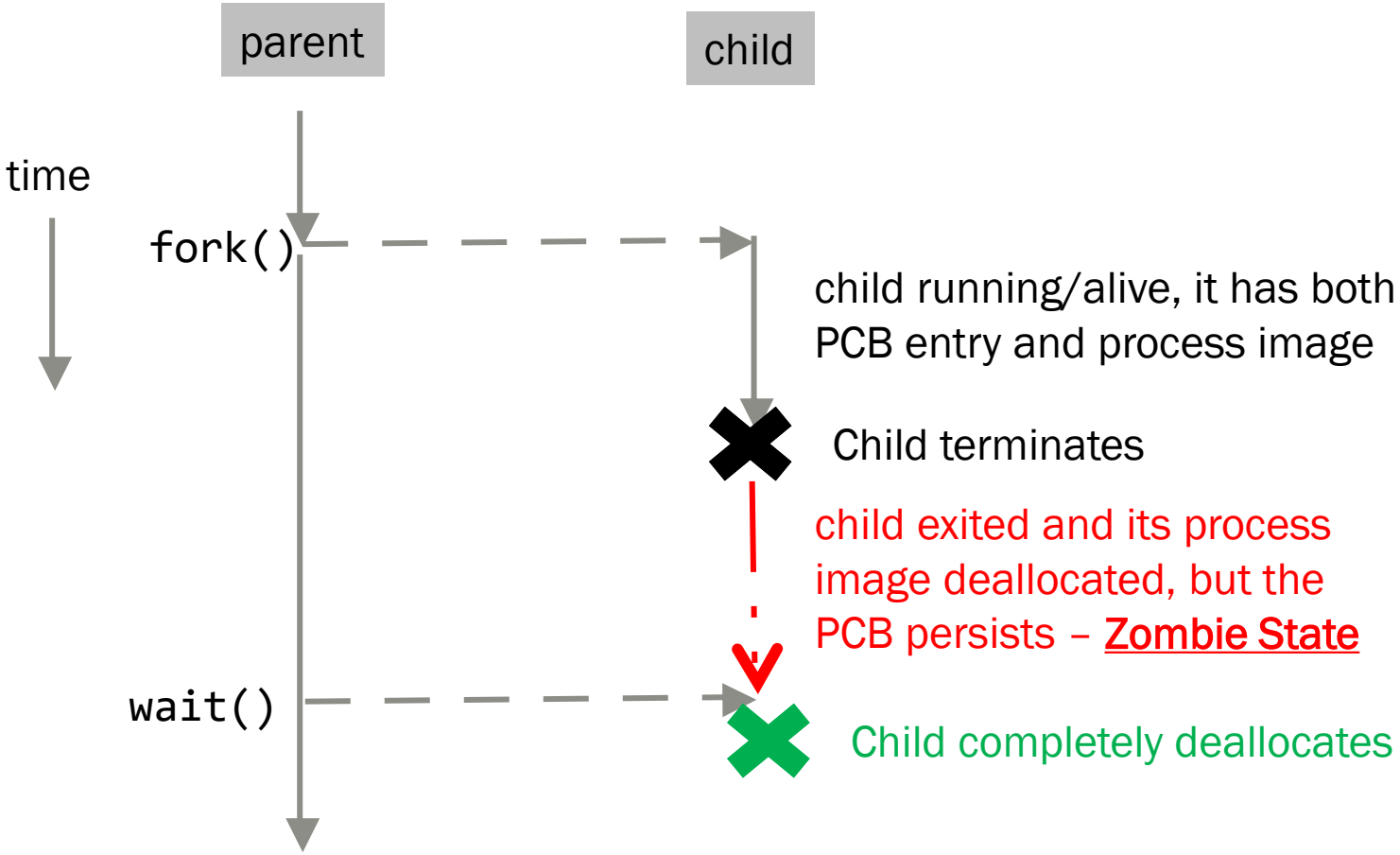


Reaping Processes using `wait()/waitpid()`

- Another important use case of `wait()` function is to **reap child processes** by parents
- After `fork()` the child process goes its own way independent of the parent
- However, if the child process terminates before the parent does, the kernel keeps must keep the PCB (i.e., an entry in the process table) around
  - *Because the parent may want to know its exit status*
- This leads to **Zombie processes** (undead child process)
  - *This is a memory leak as well, because the PCB takes memory*
- This entry is deallocated when:
  - *The parent process calls `wait()/waitpid()`, or*
  - *The parent process itself terminates*

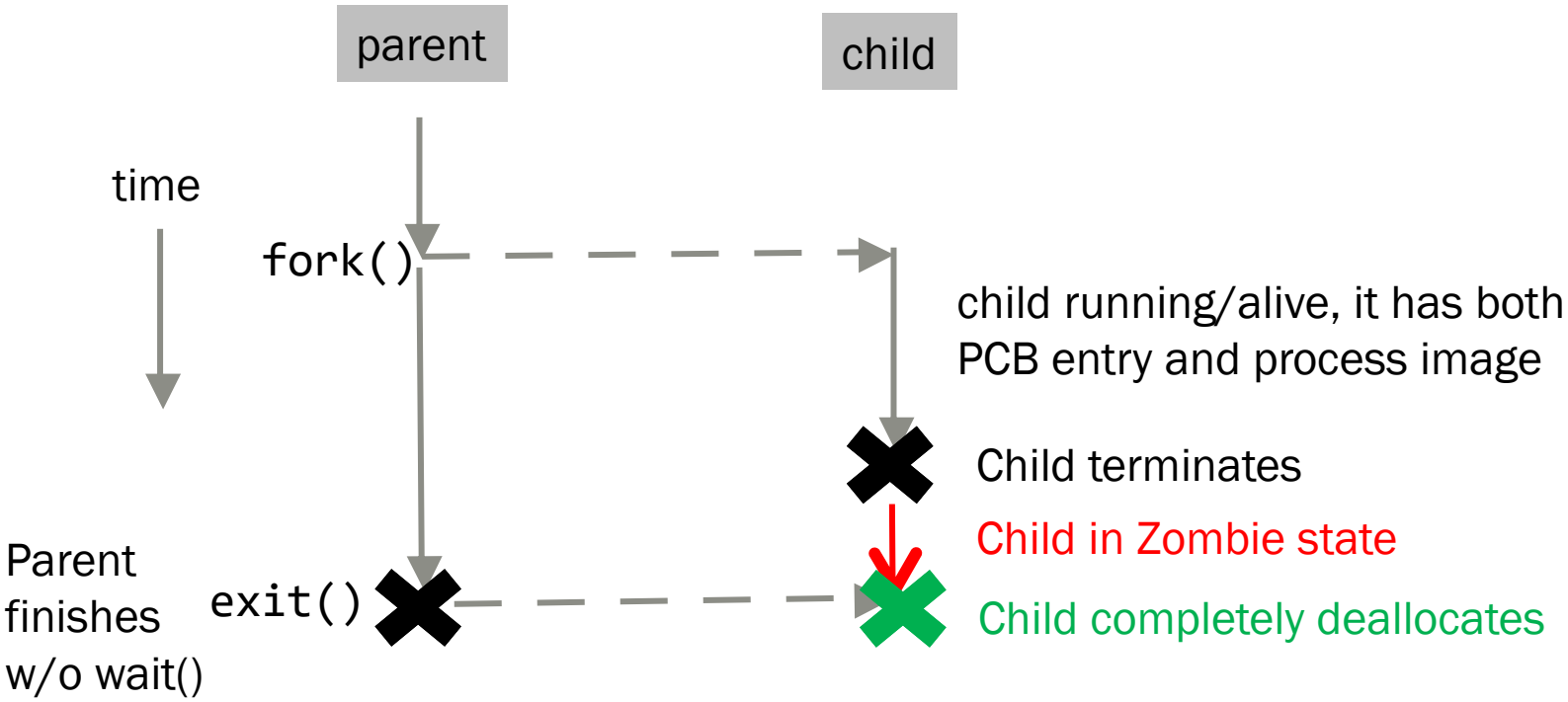
Reaping Processes using `wait()`

- The following is a timeline illustration of such scenario:



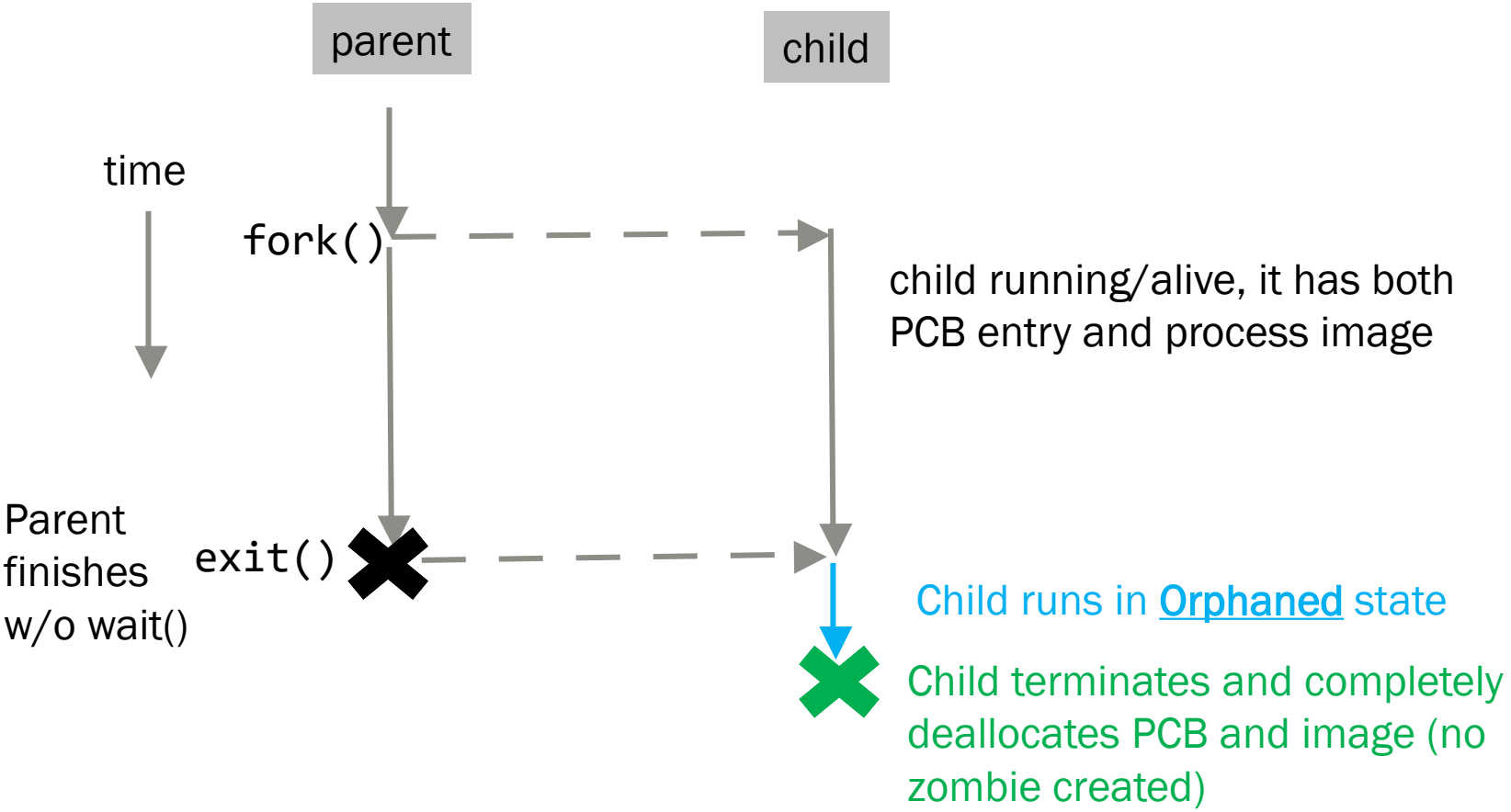
Automatic Reap w/o `wait()`

- This happens when the parent itself terminates w/o wait()
  - *The kernel is no longer obligated to keep PCB of the dead child*



Automatic Reap w/o `wait()` – Another Scenario

- This happens when the parent itself terminates w/o wait()
  - *The kernel is no longer obligated to keep PCB of the dead child*



Zombie State

- The state of a terminated child when the parent is still running w/o calling wait()

```
int main (){
    if (fork()){ // parent
        while (true){//infinite loop
            sleep (1);
        }
    }else{// child
        cout<<"Child about to exit"<<endl;
    }
}
```

```
prompt> ./a.out
Child about to exit
^Z
[1]+  Stopped                 ./a.out
prompt> ps
    PID TTY          TIME CMD
   3260 pts/0    00:00:00 bash
   3290 pts/0    00:00:00 a.out
   3291 pts/0    00:00:00 a.out <defunct>
   3292 pts/0    00:00:00 ps
prompt> kill -9 3291
prompt> ps
    PID TTY          TIME CMD
   3260 pts/0    00:00:00 bash
   3290 pts/0    00:00:00 a.out
   3291 pts/0    00:00:00 a.out <defunct>
   3293 pts/0    00:00:00 ps
prompt> kill -9 3290
prompt> ps
    PID TTY          TIME CMD
   3260 pts/0    00:00:00 bash
   3294 pts/0    00:00:00 ps
[1]+  Killed                  ./a.out
prompt>
```

Pressed Ctrl+Z to suspend the parent

List current processes

Showing zombie as defunct process

Cannot kill zombie

Killing the parent causes the zombie to be killed as wee

Multiple Fork()s - Example

- How many lines will be printed?

```
#include <stdio.h>
int main (){
        printf ("PID: %d\n", getpid());
        for (int i=0; i<3; i++){
                fork();
        }
        printf ("Done");
}
```

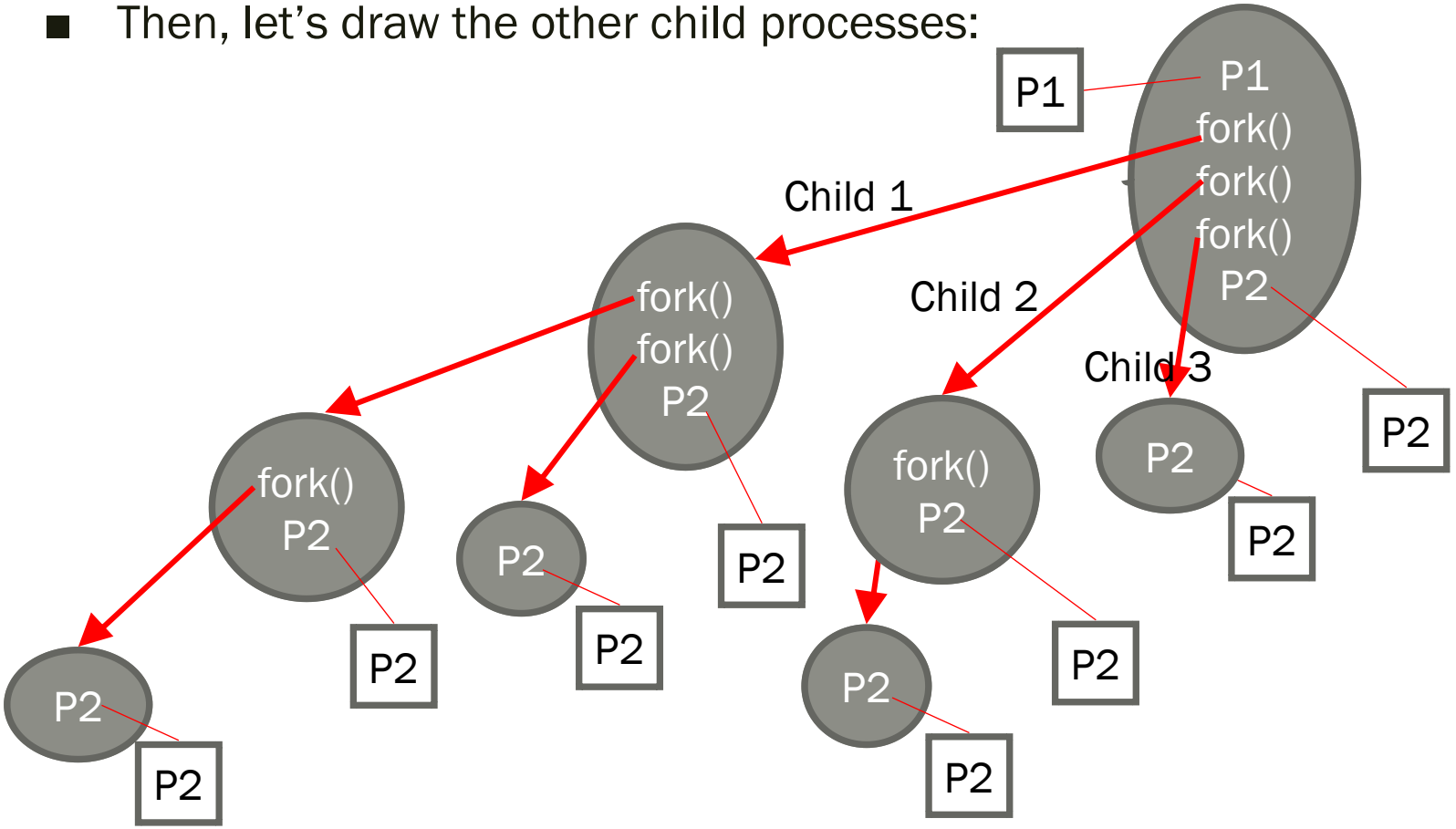
- First, we mustunroll the loop:

```
int main (){
        printf ("PID: %d\n", getpid());
        fork();
        fork();
        fork();
        printf ("Done");
}
```

Fork Example 1

```c
#include <stdio.h>
int main (){
   printf("PID:%d\n",getpid());//print1:P1
   fork();
   fork();
   fork();
   printf ("Done");   //print2:P2
}
```

- Let's redraw the process for convenience:
- Then, let's draw the other child processes:



Key Learnings Today

- Shell Basics
- Replacing Program Executed by Process
  - *Call `execv` (or variant)*
    - One call, (normally) no return
- Spawning Processes
  - *Call to `fork()`*
    - One call, two returns
- Reaping Processes
  - *Call `wait()`*