Reading Reference:
Textbook 1 Chapter 3
Molay Reference Text: Chapter 8

PROCESS PROGRAMMING INTERFACE

Tanzir Ahmed
CSCE 313 Spring 2020

Theme of Today’s Lecture

- Talk a bit about Unix Shell
- Introduce some key process control concepts
  - *Executing a program from within another program*
  - *Creating a new process*
  - *Introducing Wait dependencies between parent and child processes*

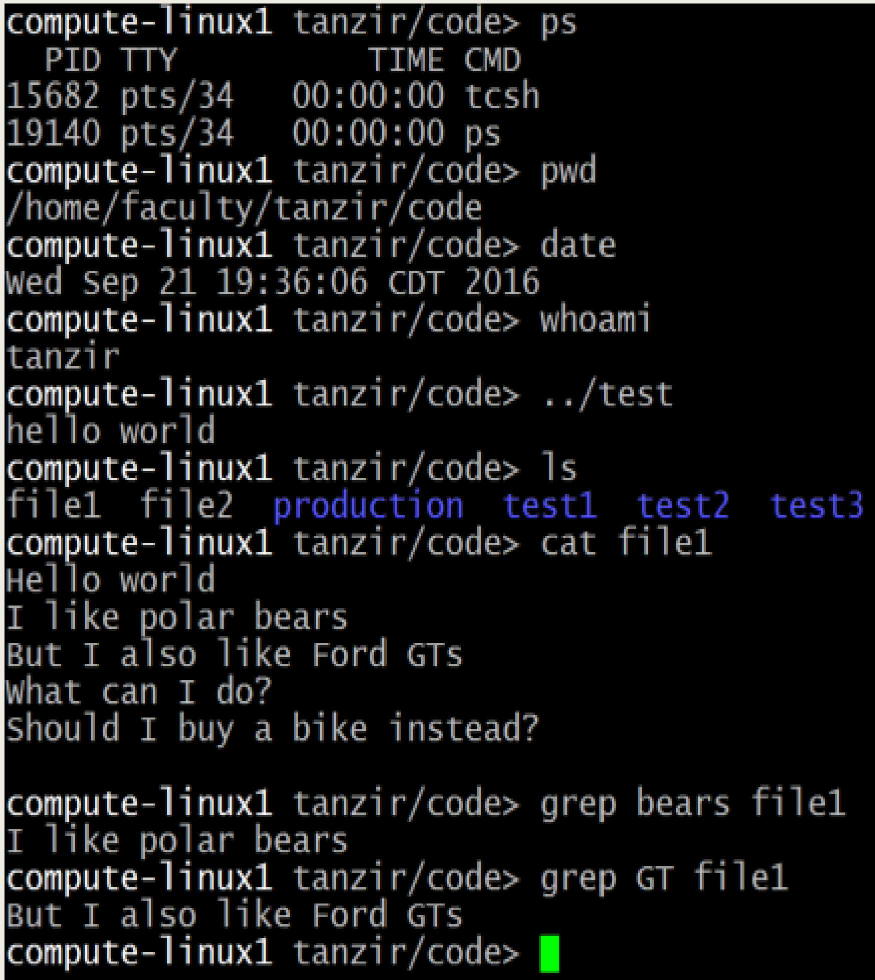
What is a Shell?

- Shell is a program which
  - *Runs programs*
  - *Manages inputs and outputs*
  - *Can be programmed*
- How about **Windows Explorer** in Windows or Ubuntu?
  - *Effectively provides similar environments (i.e., can double click to launch a program)*
  - *Less programmable to some extent*
- Are these user apps or part of kernel?
  - *User apps, because you can shut them down*
  - *Can crash, but the OS is OK*

Shell – Running Programs

- The commands **`ps, ls, grep, date,`** etc. are regular programs

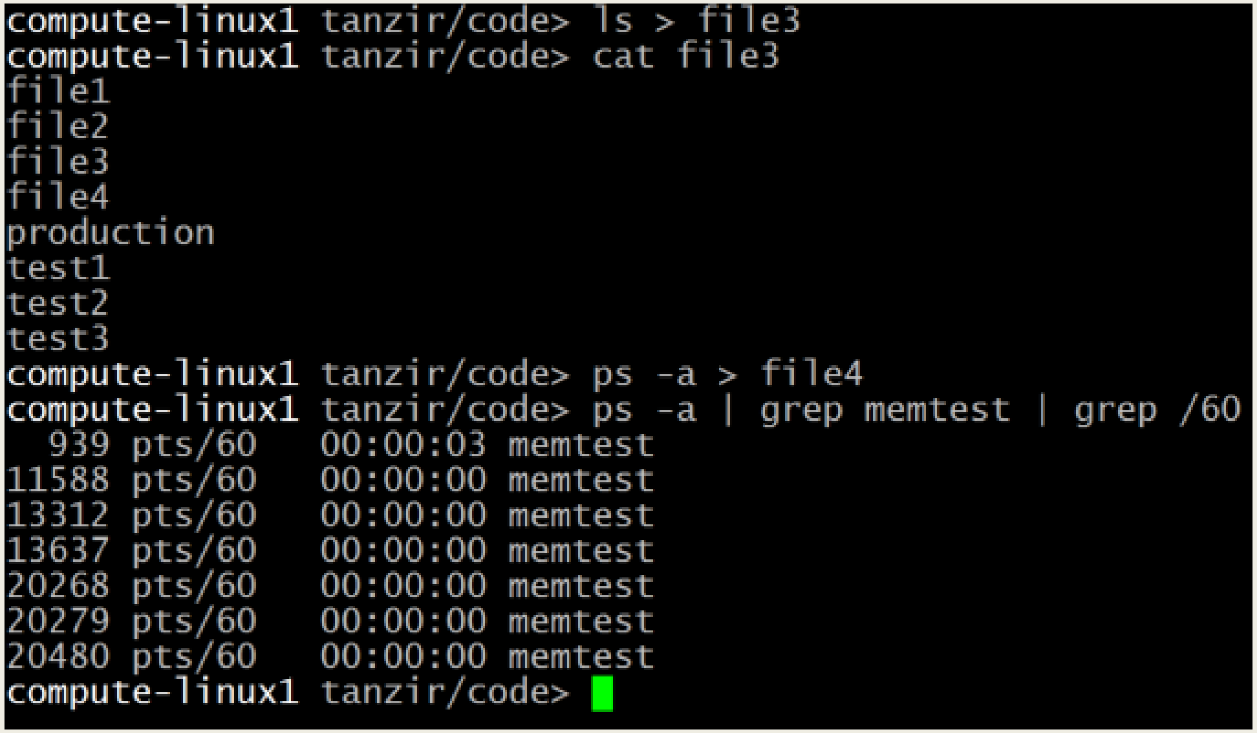
- The shell is just another user app (not kernel)
  - *loads these programs into memory and runs them, with the help of* ***System Calls***



```
compute-linux1 tanzir/code> ps
  PID TTY          TIME CMD
15682 pts/34   00:00:00 tcsh
19140 pts/34   00:00:00 ps
compute-linux1 tanzir/code> pwd
/home/faculty/tanzir/code
compute-linux1 tanzir/code> date
Wed Sep 21 19:36:06 CDT 2016
compute-linux1 tanzir/code> whoami
tanzir
compute-linux1 tanzir/code> ../test
hello world
compute-linux1 tanzir/code> ls
file1  file2  production  test1  test2  test3
compute-linux1 tanzir/code> cat file1
Hello world
I like polar bears
But I also like Ford GTs
What can I do?
Should I buy a bike instead?

compute-linux1 tanzir/code> grep bears file1
I like polar bears
compute-linux1 tanzir/code> grep GT file1
But I also like Ford GTs
compute-linux1 tanzir/code> 
```

Shell – Managing I/O

- Using ‘**>**’ (output redirect), ‘**<**’ (input redirect), ‘**|**’ (pipeline) etc. the output/input can be sent/recvd to/from a **file** , or to another **process** or even **shell variables.**
  - *the default is* ***standard output***



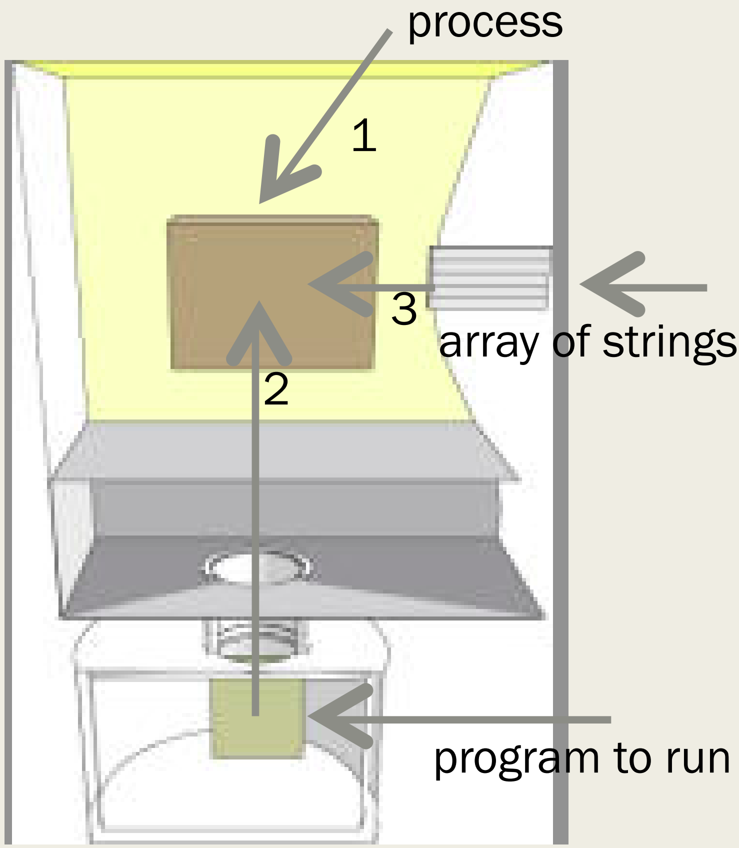
```
compute-linux1 tanzir/code> ls > file3
compute-linux1 tanzir/code> cat file3
file1
file2
file3
file4
production
test1
test2
test3
compute-linux1 tanzir/code> ps -a > file4
compute-linux1 tanzir/code> ps -a | grep memtest | grep /60
  939 pts/60   00:00:03 memtest
11588 pts/60   00:00:00 memtest
13312 pts/60   00:00:00 memtest
13637 pts/60   00:00:00 memtest
20268 pts/60   00:00:00 memtest
20279 pts/60   00:00:00 memtest
20480 pts/60   00:00:00 memtest
compute-linux1 tanzir/code> 
```

If I ask you to write a Shell!!

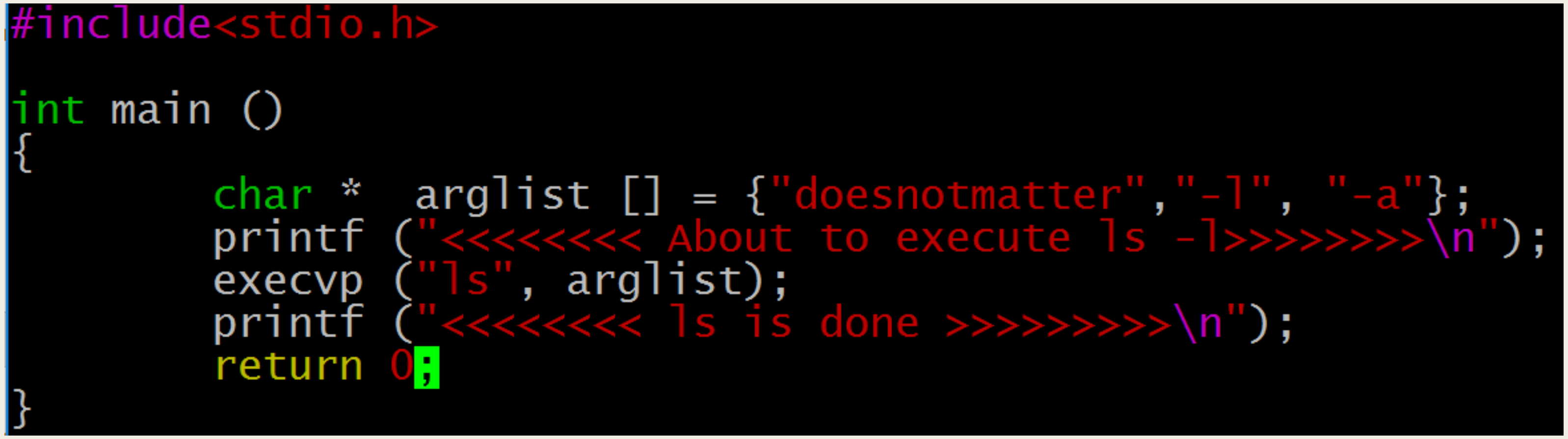
- I will do that for Programming Assignment 3
- First thing to know:
  - *How to run 1 program from another*

One Program Running Another

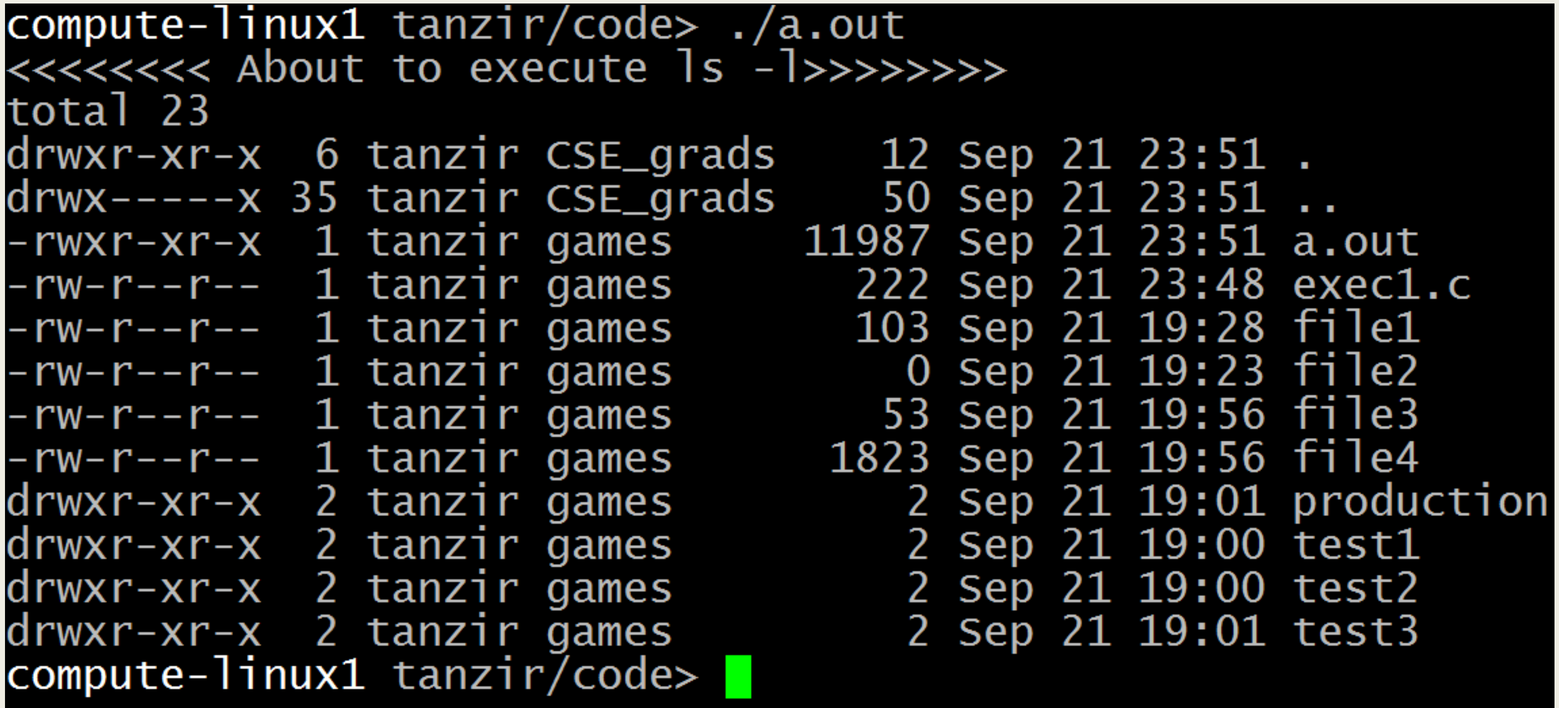
- Say, the other program's name is **name**
- The current program makes a system call
  `exec("name", arglist)`
- Kernel loads the "**name**" executable program from disk into the process
- Kernel copies **arglist** into the process
- Kernel calls **main(arglist)** of the **name** program



Example: One Program Running Another



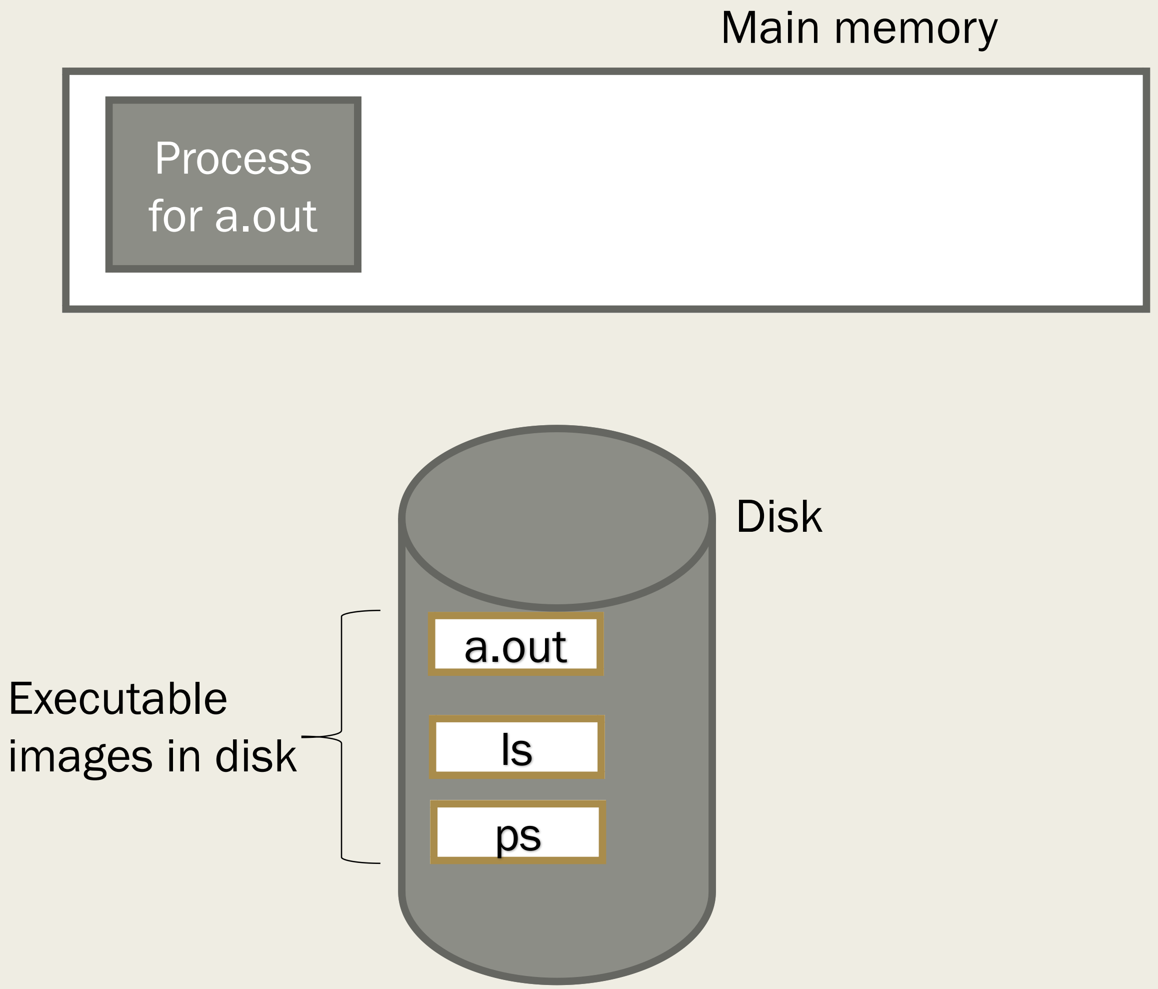
```c
#include<stdio.h>

int main ()
{
        char *  arglist [] = {"doesnotmatter","-l", "-a"};
        printf ("<<<<<<<< About to execute ls -l>>>>>>>>\n");
        execvp ("ls", arglist);
        printf ("<<<<<<<< ls is done >>>>>>>>\n");
        return 0;
}
```



```
compute-linux1 tanzir/code> ./a.out
<<<<<<<< About to execute ls -l>>>>>>>>
total 23
drwxr-xr-x  6 tanzir CSE_grads    12 Sep 21 23:51 .
drwx-----x 35 tanzir CSE_grads    50 Sep 21 23:51 ..
-rwxr-xr-x  1 tanzir games     11987 Sep 21 23:51 a.out
-rw-r--r--  1 tanzir games       222 Sep 21 23:48 exec1.c
-rw-r--r--  1 tanzir games       103 Sep 21 19:28 file1
-rw-r--r--  1 tanzir games         0 Sep 21 19:23 file2
-rw-r--r--  1 tanzir games        53 Sep 21 19:56 file3
-rw-r--r--  1 tanzir games      1823 Sep 21 19:56 file4
drwxr-xr-x  2 tanzir games         2 Sep 21 19:01 production
drwxr-xr-x  2 tanzir games         2 Sep 21 19:00 test1
drwxr-xr-x  2 tanzir games         2 Sep 21 19:00 test2
drwxr-xr-x  2 tanzir games         2 Sep 21 19:01 test3
compute-linux1 tanzir/code>
```

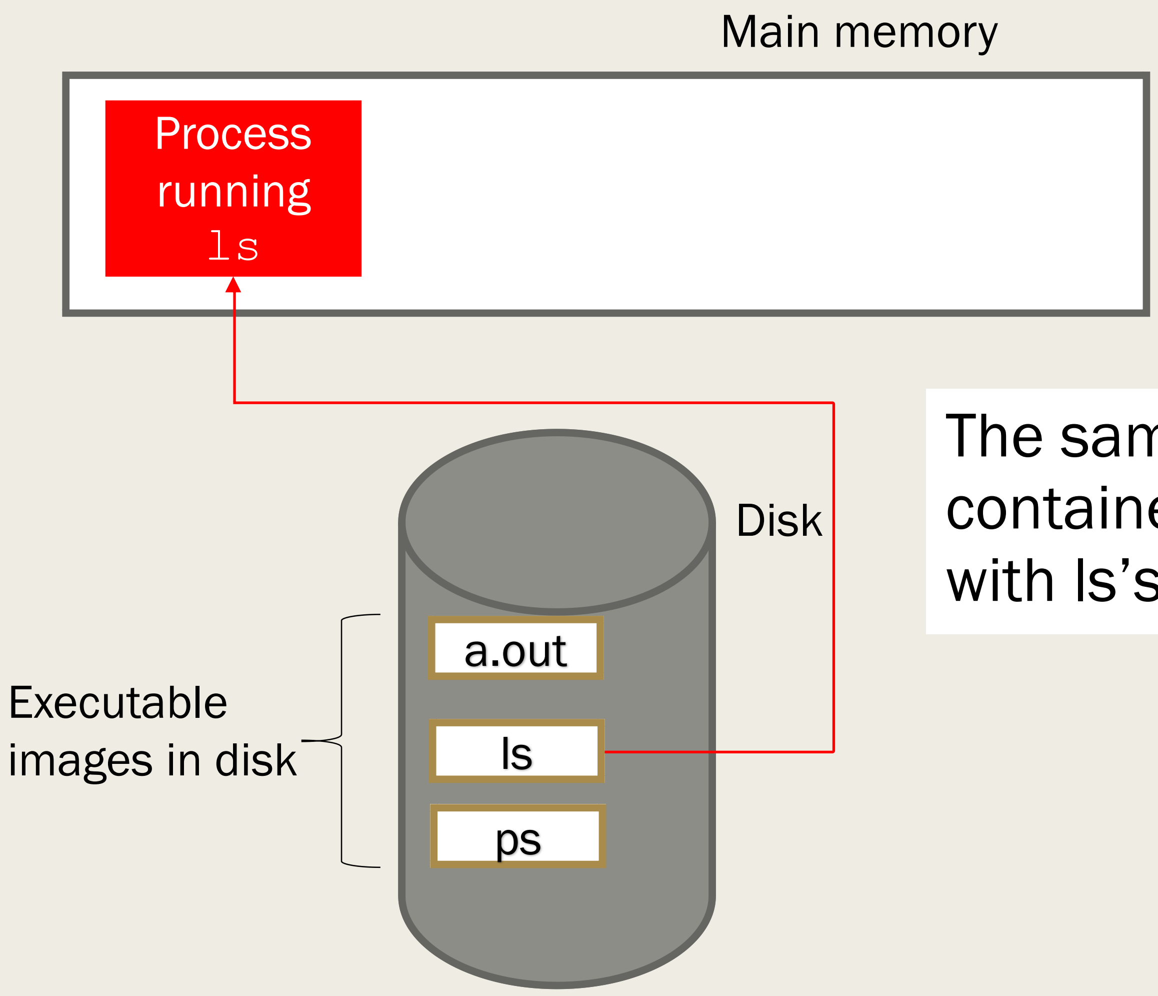
What happens in exec()

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



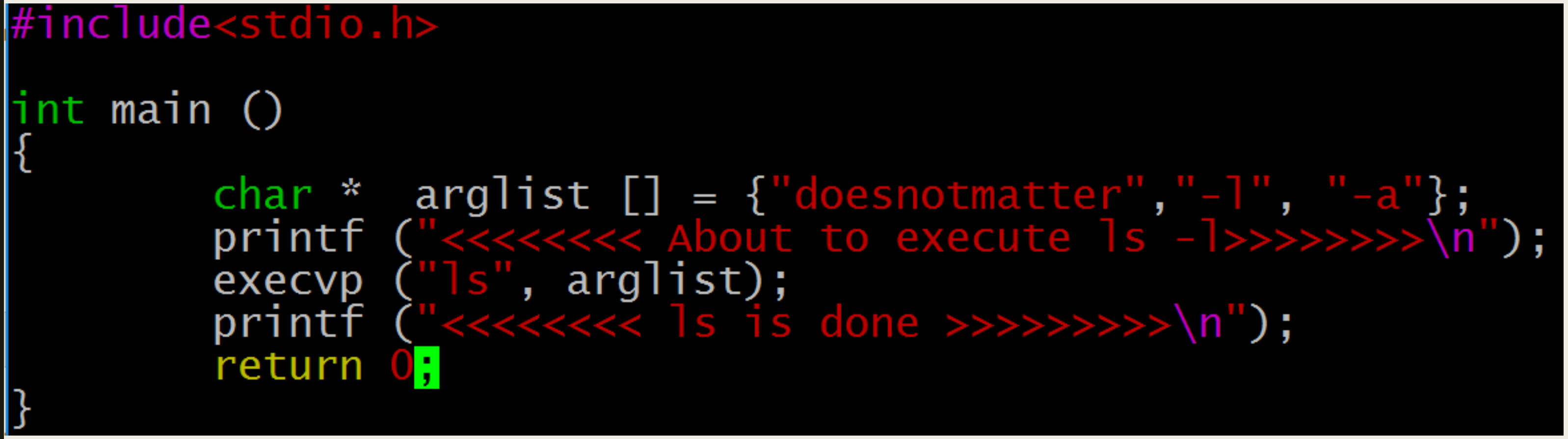
What happens in exec() contd

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



The same process container is overwritten with ls's image

Example: contd.



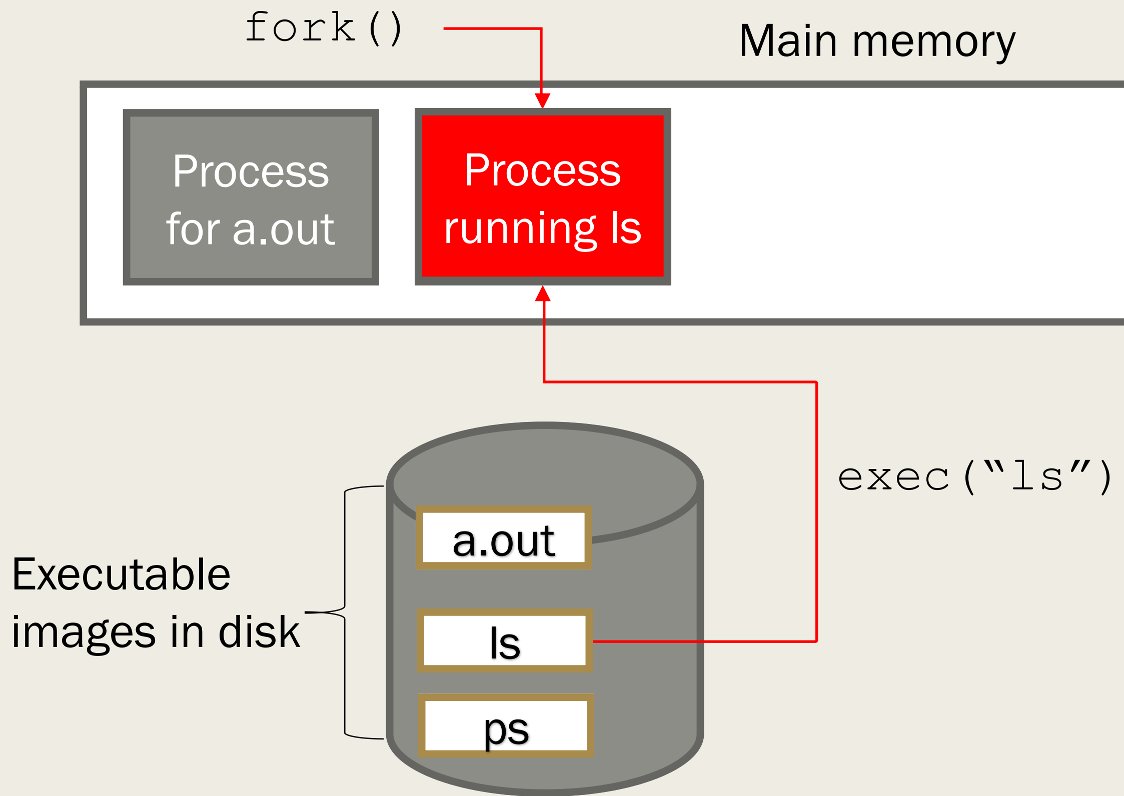
```
#include<stdio.h>

int main ()
{
        char *  arglist [] = {"doesnotmatter","-l", "-a"};
        printf ("<<<<<<<< About to execute ls -l>>>>>>>>\n");
        execvp ("ls", arglist);
        printf ("<<<<<<<< ls is done >>>>>>>>>\n");
        return 0;
}
```

- 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

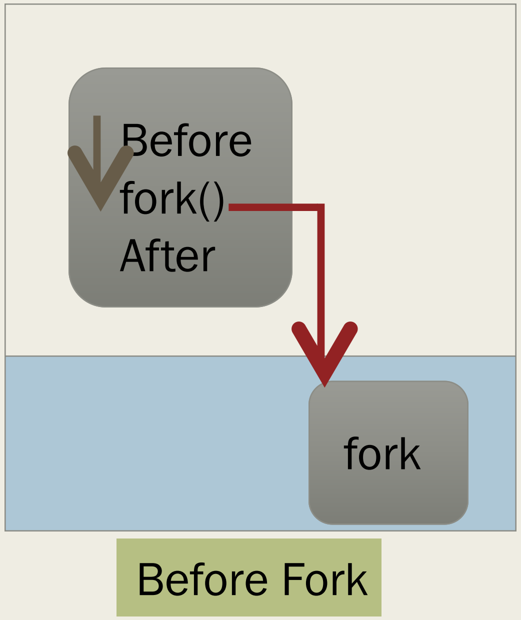
To Avoid Image Overwrite

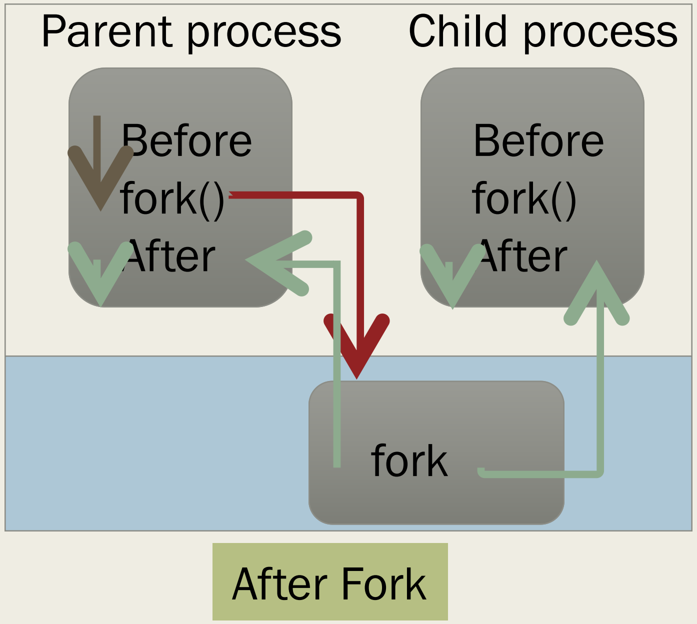
- First, if we are running shell, we need to continue having the shell image intact
  - *Otherwise it is out after exec()ing the first process*
- We need another function to create a separate Process container first using another function `fork()`, and then call `exec("ls")`



Creating a New Process

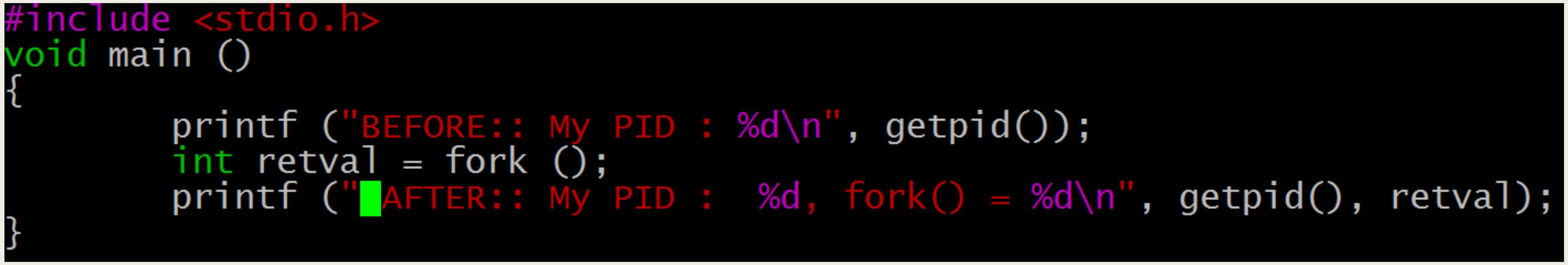
- Calling **`fork()`** function



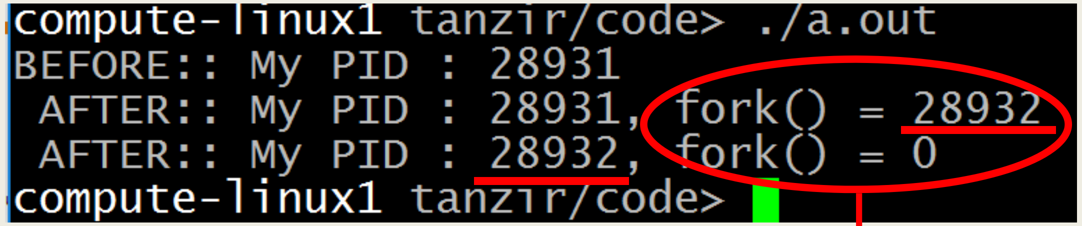


- After a process invokes fork(), control passes to the Kernel, which does the following:
  - Allocates address space and data structures
  - Copies the original process into the new process (everything including PC)
  - Adds the new process to the set of ready-to-run processes
  - Returns control back to both processes

Example: Fork



```
#include <stdio.h>
void main ()
{
        printf ("BEFORE:: My PID : %d\n", getpid());
        int retval = fork ();
        printf ("AFTER:: My PID :  %d, fork() = %d\n", getpid(), retval);
}
```

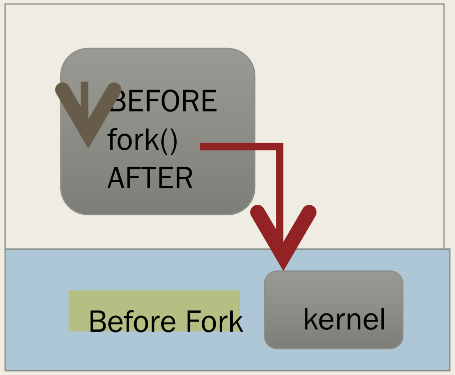


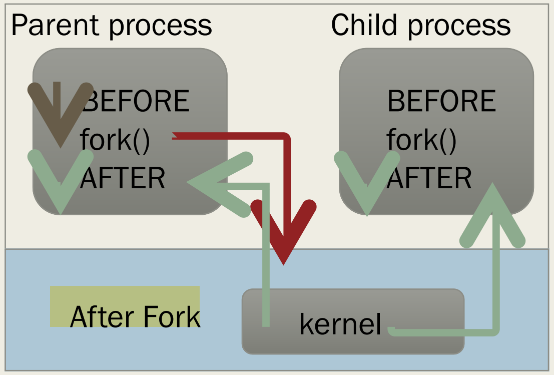
```
compute-linux1 tanzir/code> ./a.out
BEFORE:: My PID : 28931
 AFTER:: My PID : 28931, fork() = 28932
 AFTER:: My PID : 28932, fork() = 0
compute-linux1 tanzir/code>
```

Fork() returns the child's ID to the parent side

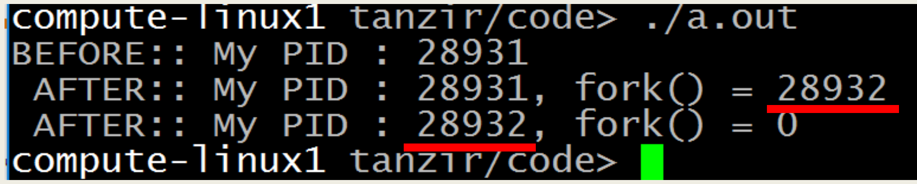
BEFORE printed once, AFTER printed 2 times

Fork() returns different things (0 on one side, nonzero on the other)





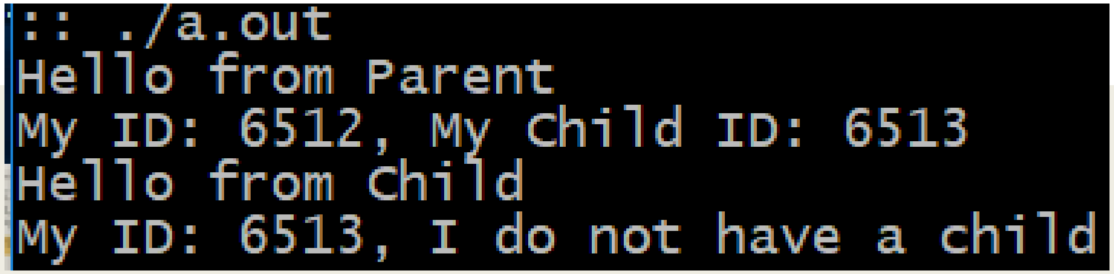
Fork() function



```
compute-linux1 tanzir/code> ./a.out
BEFORE:: My PID : 28931
 AFTER:: My PID : 28931, fork() = 28932
 AFTER:: My PID : 28932, fork() = 0
compute-linux1 tanzir/code>
```

- What this function returns depends on to which process
  - *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());
   }
}
```



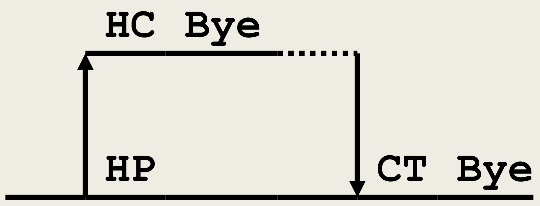
```
:: ./a.out
Hello from Parent
My ID: 6512, My Child ID: 6513
Hello from Child
My ID: 6513, I do not have a child
```

`wait`: Synchronizing with Children

- **`int wait(int *child_status)`**
  - *Notice that the argument is not an input argument, rather a place for wait() function to put return value*
    - If you set the `child_status != NULL` (i.e., a valid address), then integer it points to will be set to indicate why child terminated (-1 usually means error, 0 success)
    - You will see this when a C-style function wants to return >1 values
- Suspends current process until one of its children terminates
- Return value is `pid` of child process that terminated
  - *Of course, you call the argument a return value too that tells you the exit status*

`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");
}
```



Fork() Example 1

- 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");
}
```

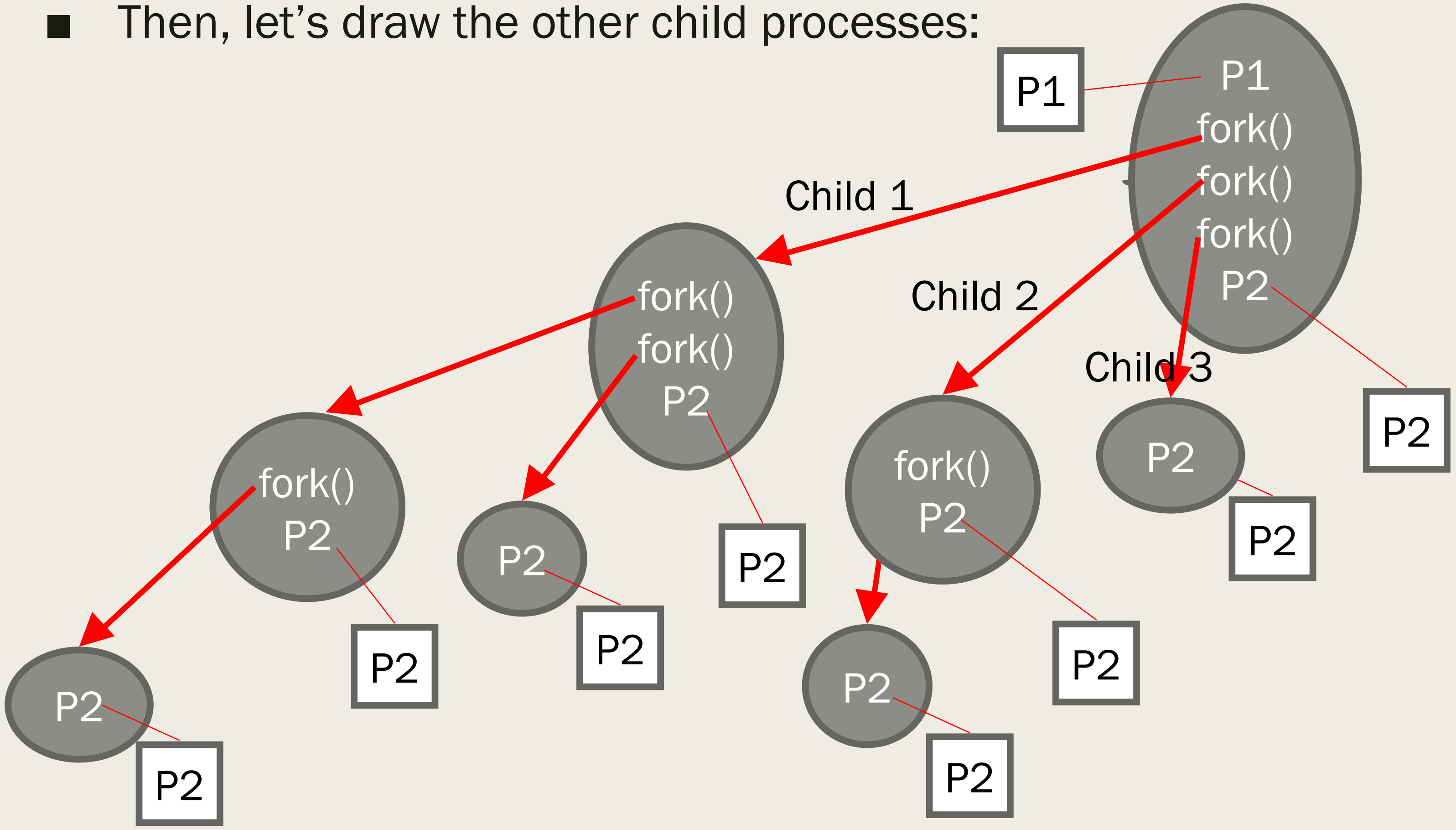
- First, we have to unroll the loop:

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

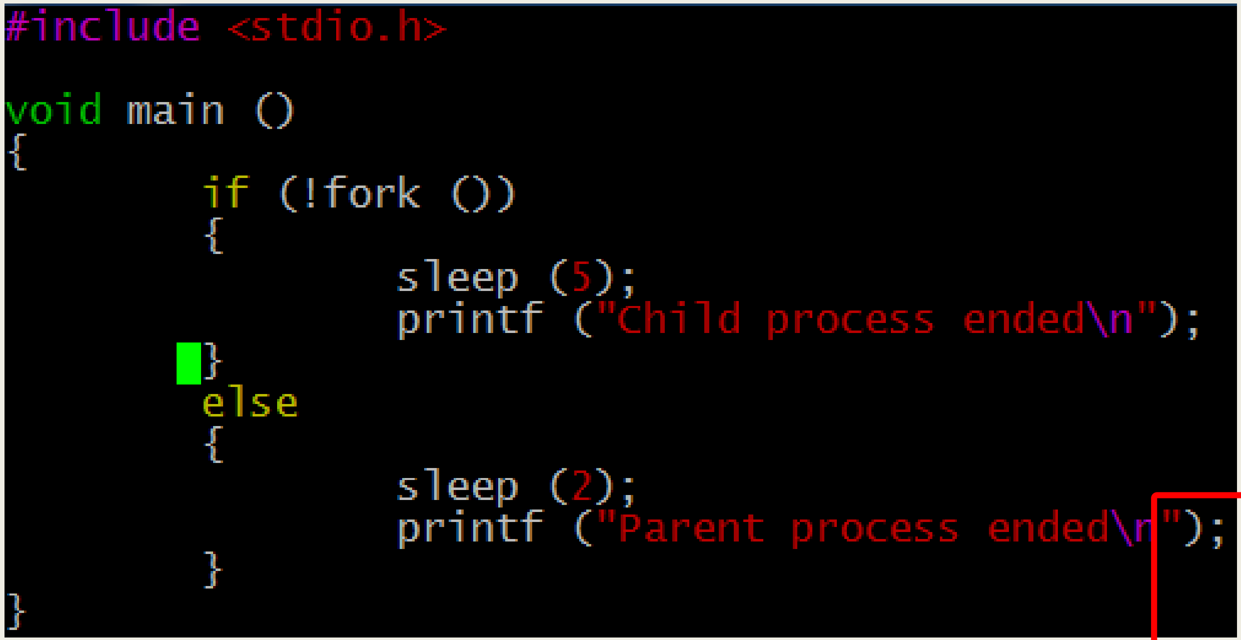
Fork Example 1

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

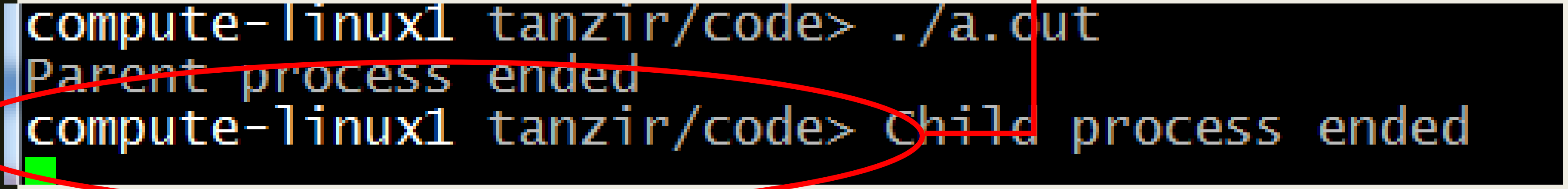
- Let's redraw the process for convenience:
- Then, let's draw the other child processes:



Example. 2



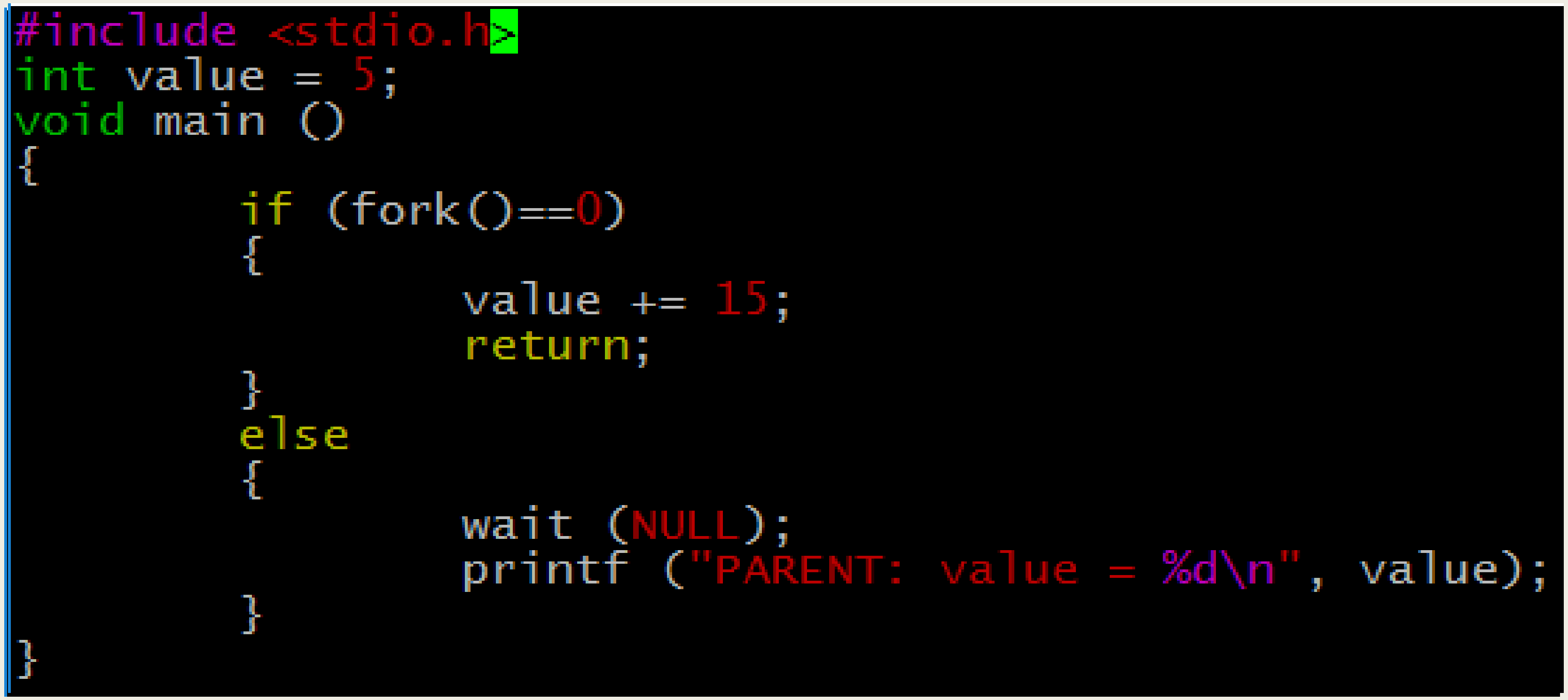
```
#include <stdio.h>

void main ()
{
        if (!fork ())
        {
                sleep (5);
                printf ("Child process ended\n");
        }
        else
        {
                sleep (2);
                printf ("Parent process ended\n");
        }
}
```



```
compute-linux1 tanzir/code> ./a.out
Parent process ended
compute-linux1 tanzir/code> Child process ended
```

1. Command prompt shows up
2. Shell does not wait for “grand children”

Example 3



```
#include <stdio.h>
int value = 5;
void main ()
{
        if (fork()==0)
        {
                value += 15;
                return;
        }
        else
        {
                wait (NULL);
                printf ("PARENT: value = %d\n", value);
        }
}
```

```
PARENT: value = 5
```

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`*