CMPT 300
Operating System I

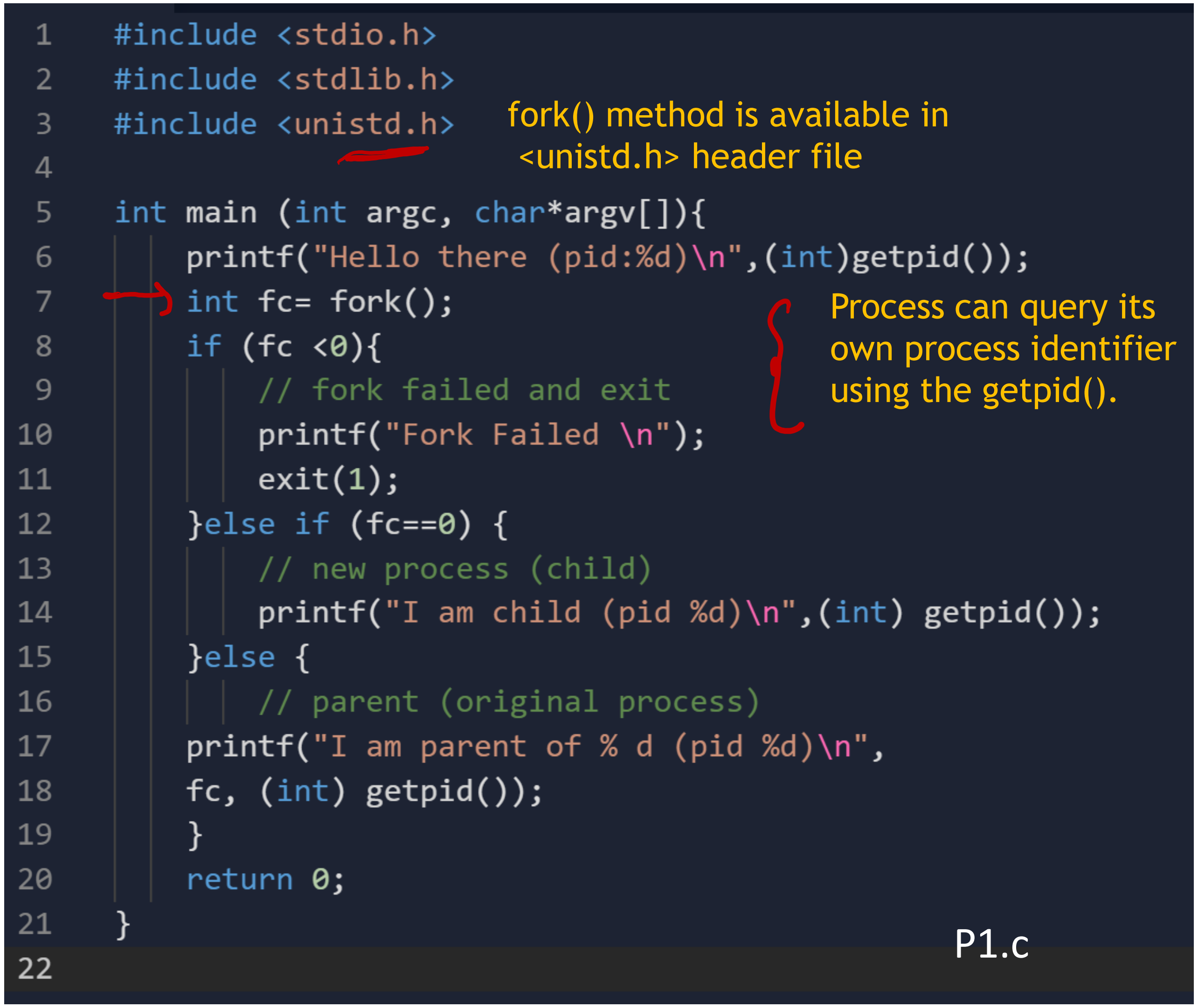
4.1 –Process 4
Chapter 3

Dr. Hazra Imran

Learning Objectives

- Understand fork() to create new processes
- Learn how exec*() execute another program within a process
- Understand how process coordinate

fork()



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char*argv[]){
    printf("Hello there (pid:%d)\n",(int)getpid());
    int fc= fork();
    if (fc <0){
        // fork failed and exit
        printf("Fork Failed \n");
        exit(1);
    }else if (fc==0) {
        // new process (child)
        printf("I am child (pid %d)\n",(int) getpid());
    }else {
        // parent (original process)
    printf("I am parent of % d (pid %d)\n",
    fc, (int) getpid());
    }
    return 0;
}
```

fork() method is available in <unistd.h> header file

Process can query its own process identifier using the getpid().

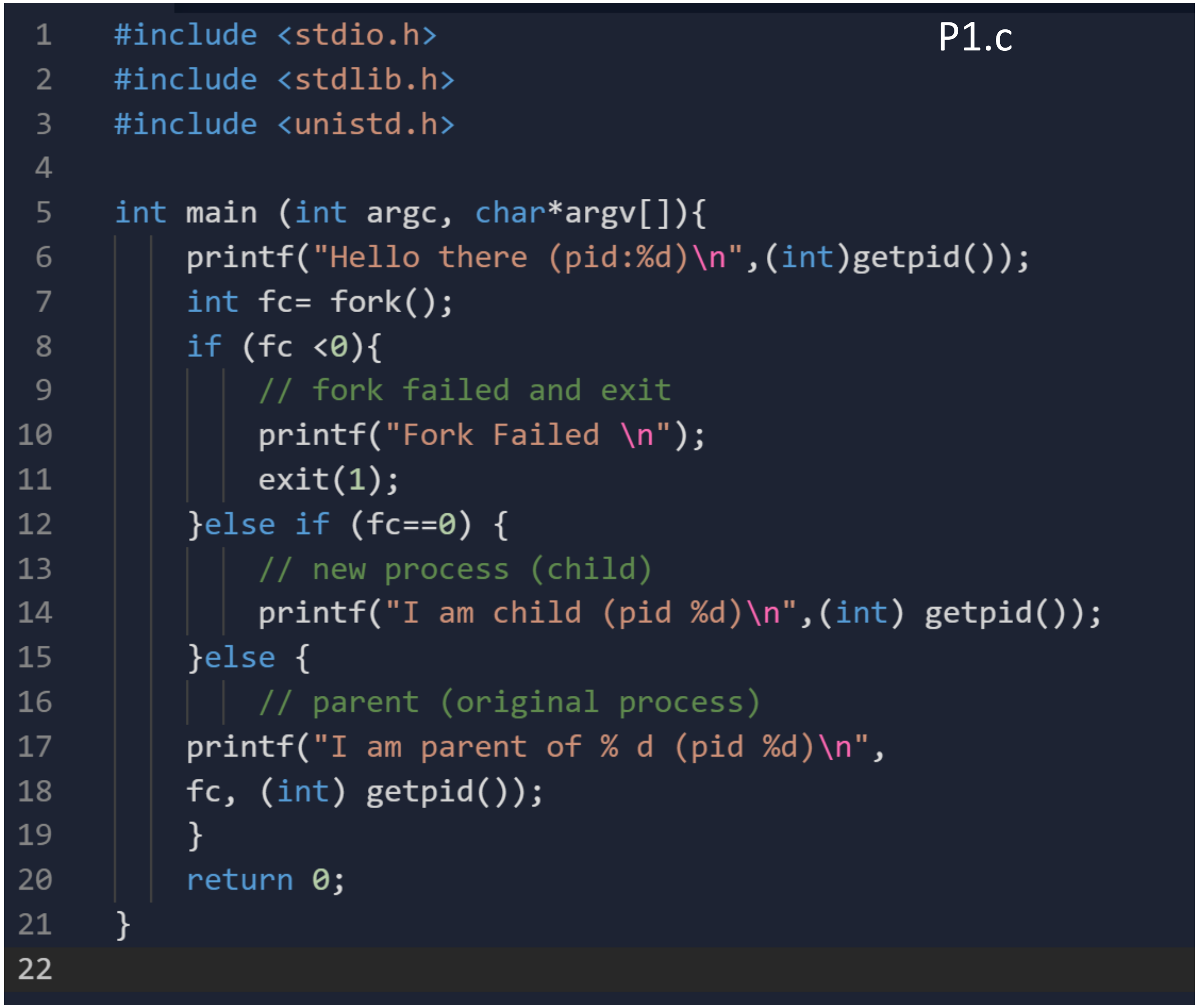
P1.c

fork successful

0 to child

PID to PP

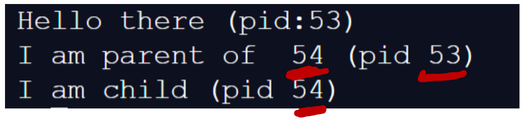
fork()



P1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char*argv[]){
    printf("Hello there (pid:%d)\n",(int)getpid());
    int fc= fork();
    if (fc <0){
        // fork failed and exit
        printf("Fork Failed \n");
        exit(1);
    }else if (fc==0) {
        // new process (child)
        printf("I am child (pid %d)\n",(int) getpid());
    }else {
        // parent (original process)
    printf("I am parent of % d (pid %d)\n",
    fc, (int) getpid());
    }
    return 0;
}

```

Output



```
Hello there (pid:53)
I am parent of  54 (pid 53)
I am child (pid 54)
```

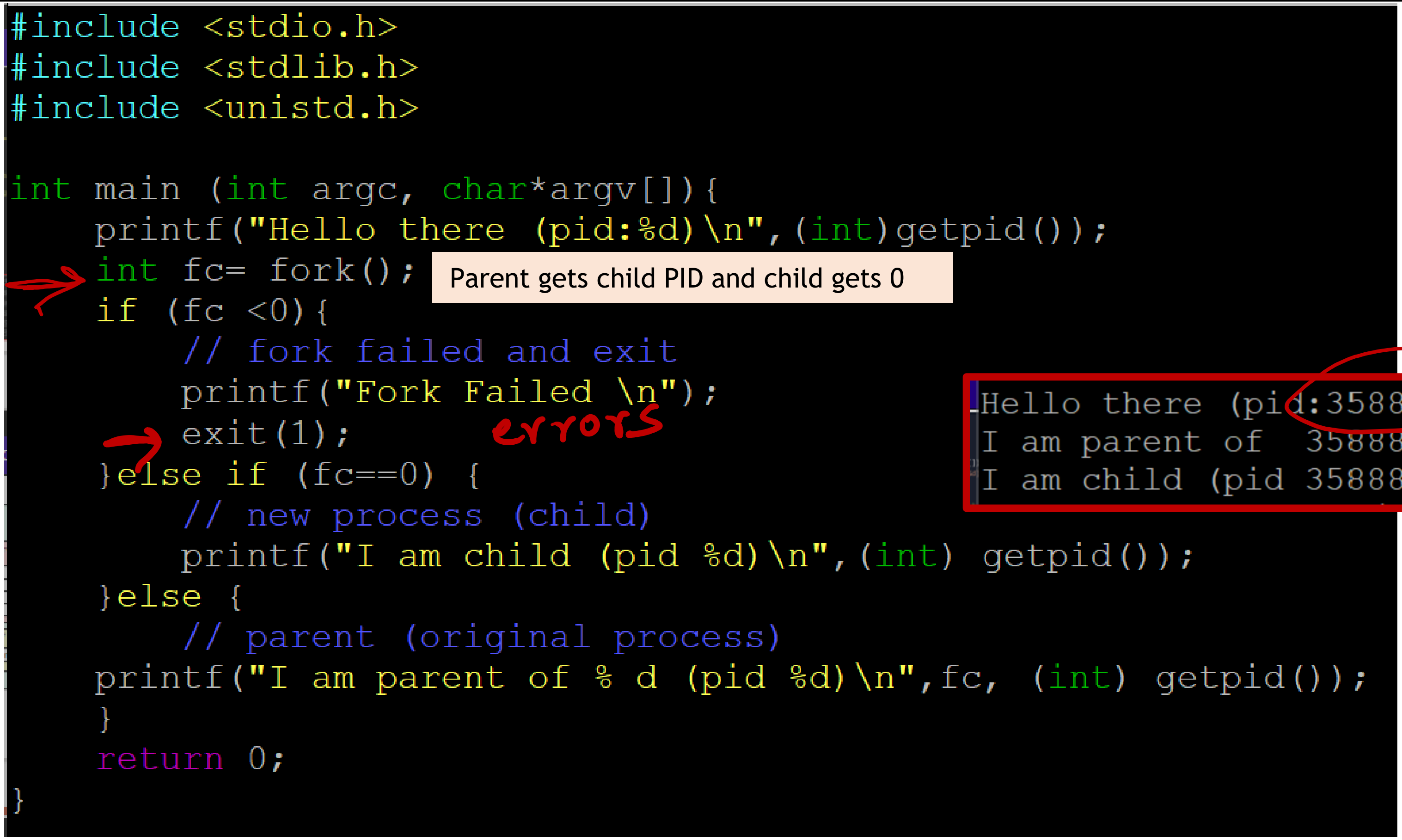
Why fork()

- Very useful when the child…
    - Relies upon the parent's data to accomplish its task
- Example: Web server

```
while (1) {
    int sock = accept();
    if (child_pid = fork()) == 0)
        Handle client request
    } else {
        close socket
    }
}
```

Example

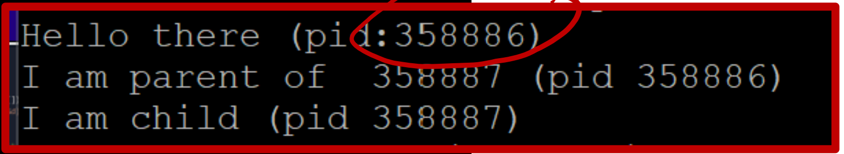
exit(0) – termination w/o msg



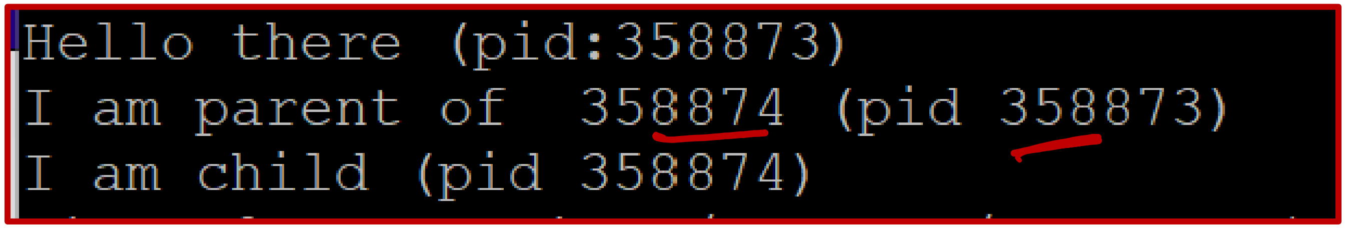
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char*argv[]){
    printf("Hello there (pid:%d)\n",(int)getpid());
    int fc= fork();
    if (fc <0){
        // fork failed and exit
        printf("Fork Failed \n");
        exit(1);
    }else if (fc==0) {
        // new process (child)
        printf("I am child (pid %d)\n",(int) getpid());
    }else {
        // parent (original process)
    printf("I am parent of % d (pid %d)\n",fc, (int) getpid());
    }
    return 0;
}
```

Parent gets child PID and child gets 0

errors



```
Hello there (pid:358886)
I am parent of  358887 (pid 358886)
I am child (pid 358887)
```

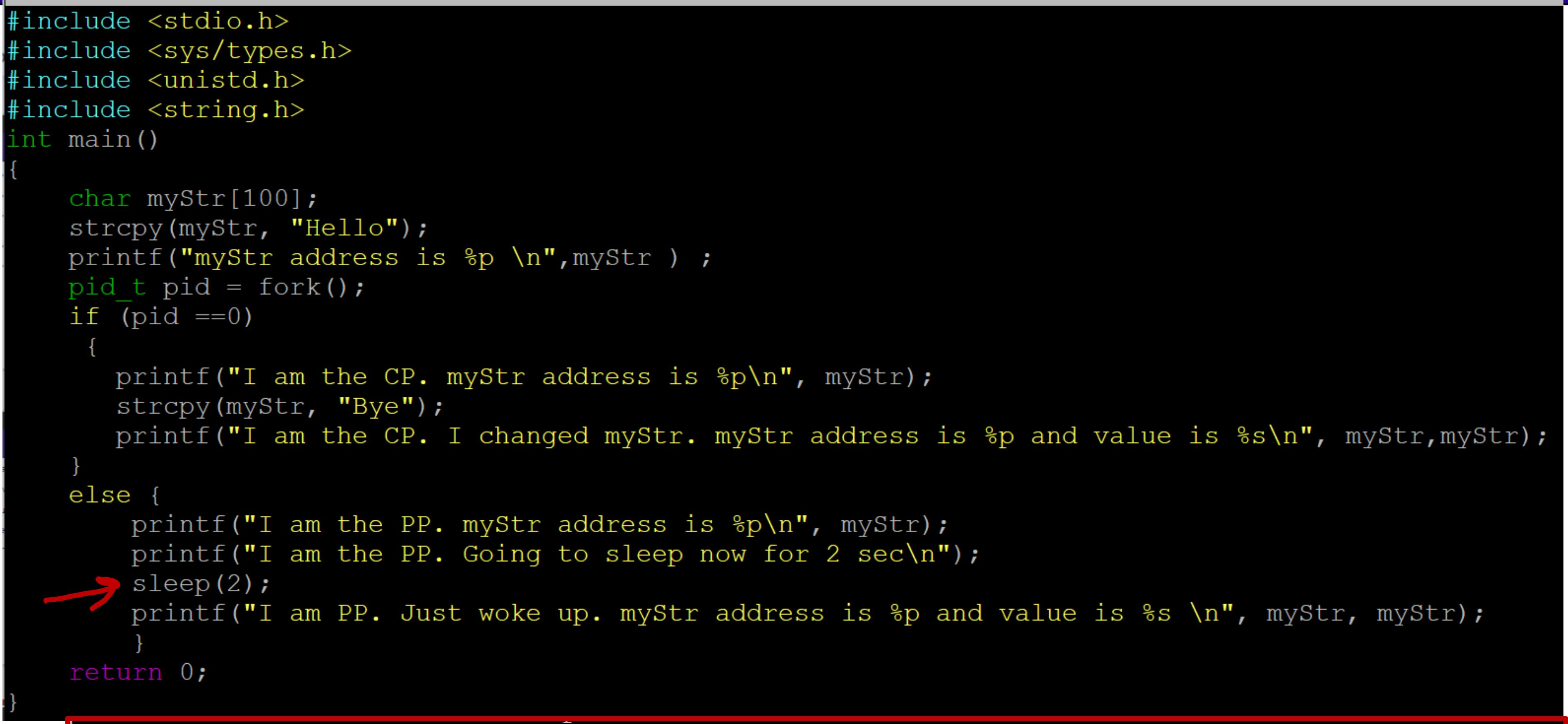


```
Hello there (pid:358873)
I am parent of  358874 (pid 358873)
I am child (pid 358874)
```

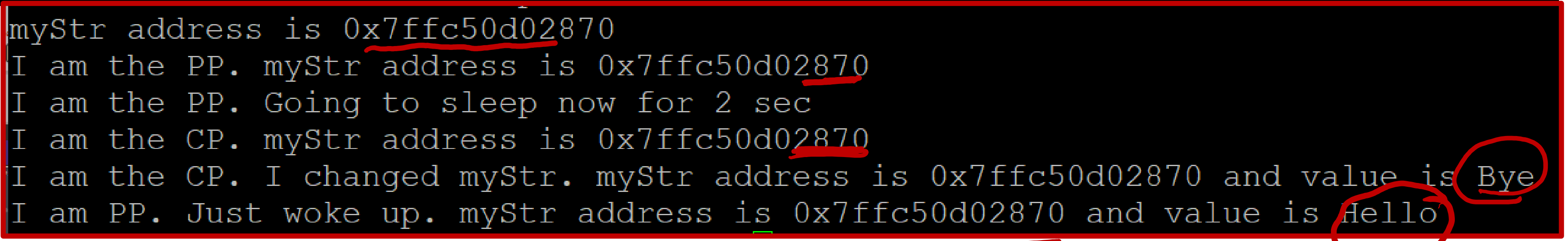
fork() – What happen to variables and addresses?

- Each program thinks they have all (virtual) memory addresses to use
- Actually, OS maps these virtual addresses ⟷ physical addresses
- After forks, the virtual address space stays the same
- OS map the child process virtual addresses to different physical addresses than for the parent

Example

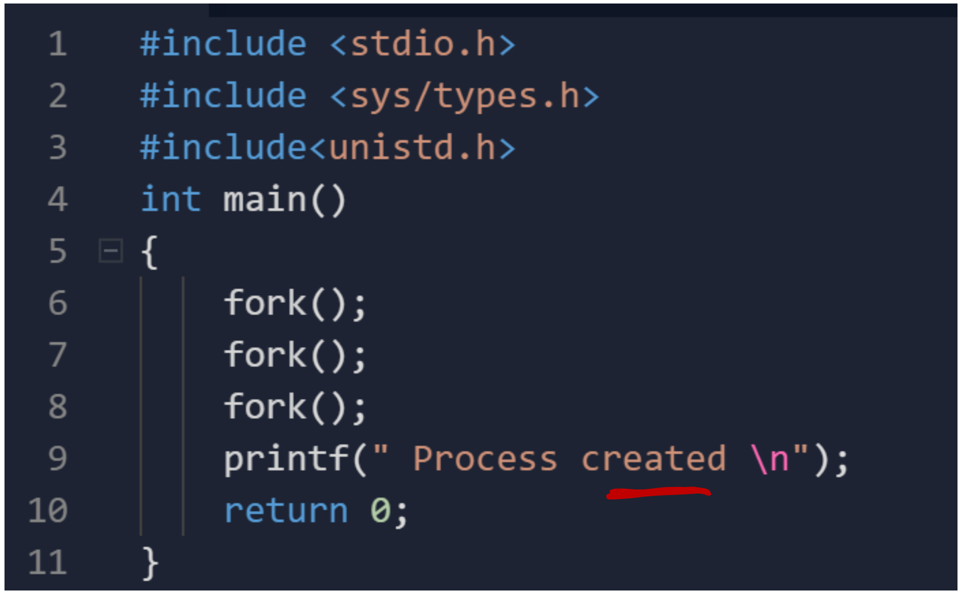


```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
int main()
{
    char myStr[100];
    strcpy(myStr, "Hello");
    printf("myStr address is %p \n",myStr ) ;
    pid_t pid = fork();
    if (pid ==0)
     {
       printf("I am the CP. myStr address is %p\n", myStr);
       strcpy(myStr, "Bye");
       printf("I am the CP. I changed myStr. myStr address is %p and value is %s\n", myStr,myStr);
    }
    else {
        printf("I am the PP. myStr address is %p\n", myStr);
        printf("I am the PP. Going to sleep now for 2 sec\n");
        sleep(2);
        printf("I am PP. Just woke up. myStr address is %p and value is %s \n", myStr, myStr);
        }
    return 0;
}
```



```
myStr address is 0x7ffc50d02870
I am the PP. myStr address is 0x7ffc50d02870
I am the PP. Going to sleep now for 2 sec
I am the CP. myStr address is 0x7ffc50d02870
I am the CP. I changed myStr. myStr address is 0x7ffc50d02870 and value is Bye
I am PP. Just woke up. myStr address is 0x7ffc50d02870 and value is Hello
```

Clicker

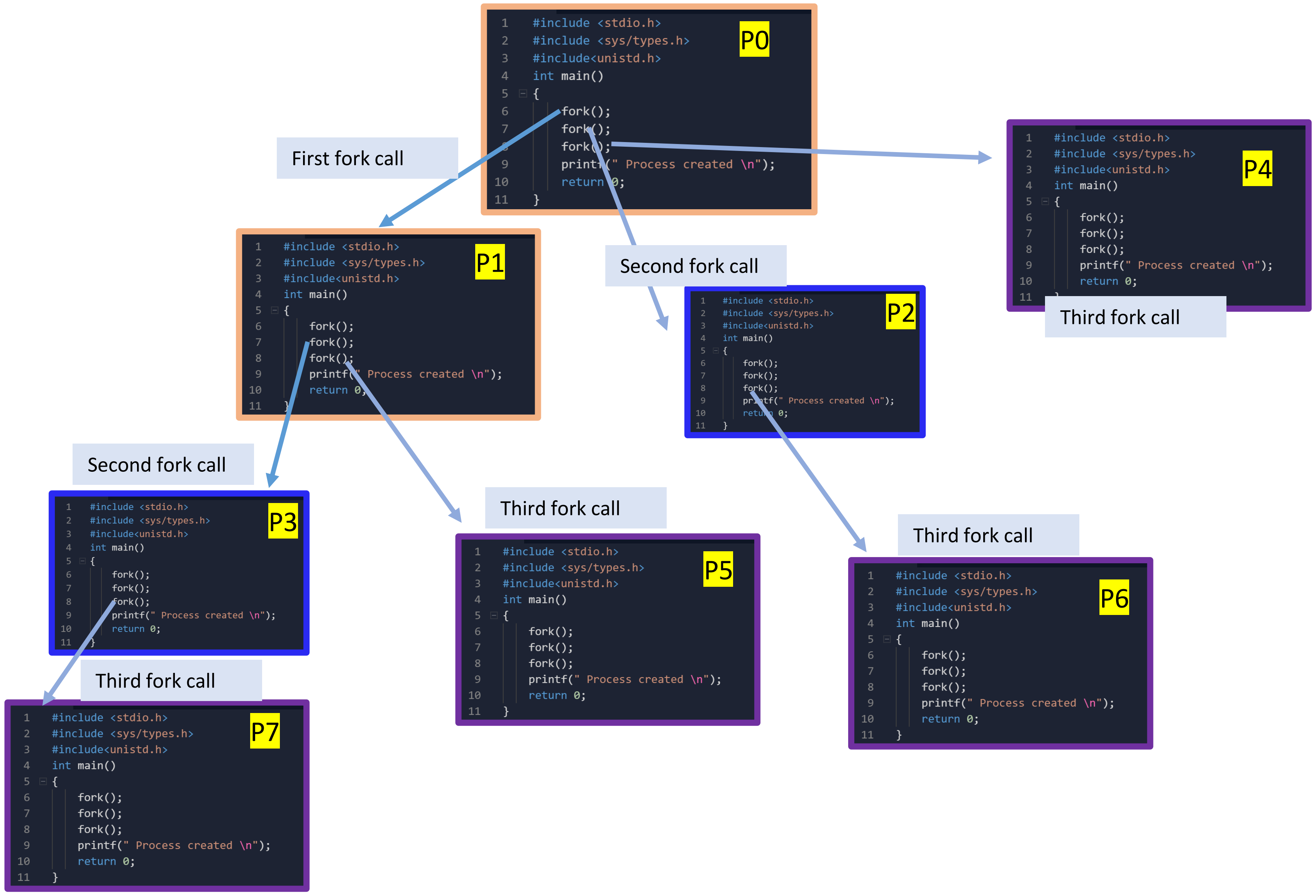


```
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
        fork();
        fork();
        fork();
        printf(" Process created \n");
        return 0;
}
```

How many times “Process created” will be printed?

A. 1 times
B. 4 times
C. 8 times
D. No idea

Clicker revisited



P0

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P1

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P2

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P3

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P4

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P5

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P6

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

P7

```c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf(" Process created \n");
    return 0;
}
```

File descriptors and fork()

- There are 3 major data structures that are relevant to file I/O.
  - file descriptors - process specific data structure
  - open file descriptions – kernel data structure
  - and the file buffers – kernel data structure

Ques. What happens to the file descriptors during a fork?
They are part of the process, so they are copied for the new process.

The open file descriptions and the file buffers - not duplicate by fork()

P_A C_B

close()
dup()

Waiting

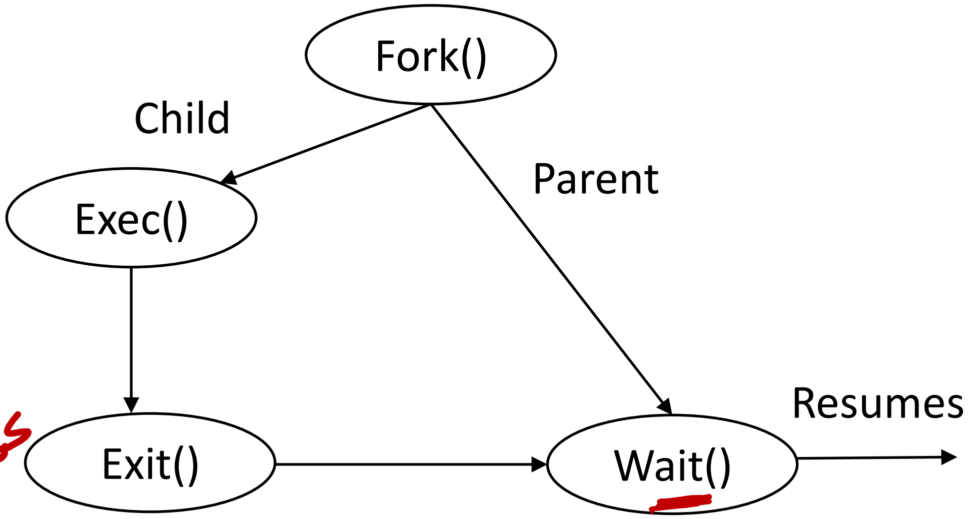
wait() blocks the parent process until one of its child processes exits.

After child process terminates, parent continues its execution after wait system call instruction.

Why does parent waits for a child process?

① parent assign task to child process

② Rid of zombie processes



Zombie Process

- Once the child terminates using exit(), all the resources associated with child are freed except for the process control block. Now, the child is in zombie state.

- Parent process can inquire about the status of the child using wait(). Then ask the kernel to free the PCB.

- In case parent doesn’t uses wait(), the child will remain in the zombie state.

Zombie Process

pwd & date
~~pwd&&~~ pwd&& date

```
ps aux | grep 'Z'
```

$\rightarrow$ list of all zombie processes

How to kill Zombie processes?

Send a SIGCHLD signal to the parent, using kill command.

```
kill -s SIGCHLD pid
```

$\downarrow$ process ID of the parent process.

signal inform PP to clean up zombie process

Waiting

Called by parent to `wait` (block) until child exits

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
```

/ returns process ID if OK, or -1 on error */*

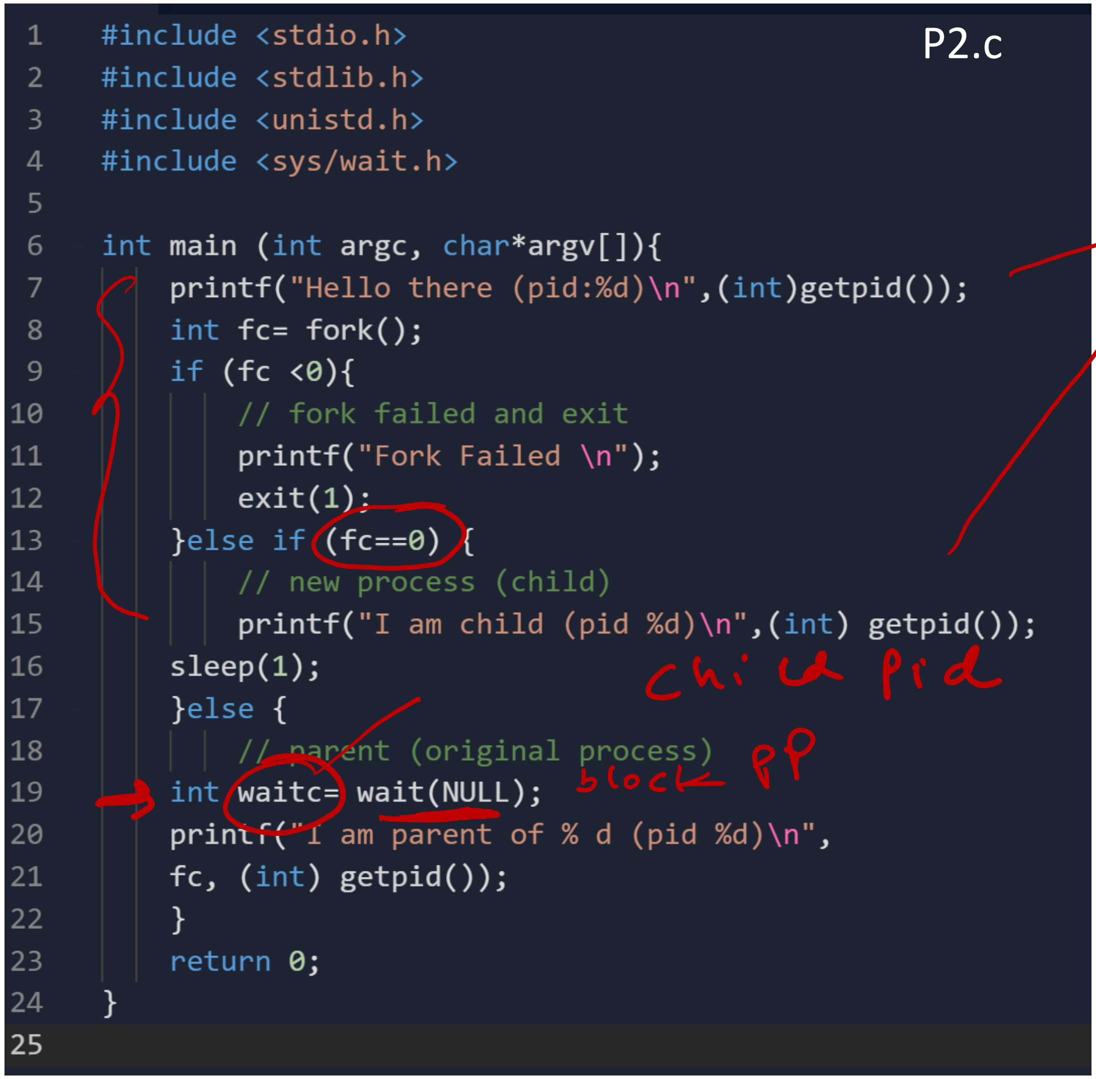
NULL, or
a pointer to a location where the function can store the terminating status of the child

wait() blocks the parent process until one of its child processes exits.

After child process terminates, parent continues its execution after wait system call instruction.

If you don't care about the child's exit status (which is quite often), you can just call wait (NULL)

wait()



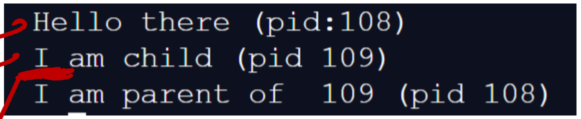
P2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char*argv[]){
    printf("Hello there (pid:%d)\n",(int)getpid());
    int fc= fork();
    if (fc <0){
        // fork failed and exit
        printf("Fork Failed \n");
        exit(1);
    }else if (fc==0) {
        // new process (child)
        printf("I am child (pid %d)\n",(int) getpid());
    sleep(1);
    }else {
        // parent (original process)
    int waitc= wait(NULL);
    printf("I am parent of % d (pid %d)\n",
    fc, (int) getpid());
    }
    return 0;
}
```

child pid

block pp

Output



```
Hello there (pid:108)
I am child (pid 109)
I am parent of  109 (pid 108)
```

delay

Waiting

→ man waitpid

If parent process has more than one child processes, then **waitpid()** can be used by the parent process to query the change state of a particular child.

```
pid_t waitpid (pid_t pid, int *stat_loc,int options);
```

where pid is the process of the child it should wait.

stat_loc is a pointer to the location where status information for the terminating process is to be stored

WCONTINUED — return the status for any child that was stopped and has been continued.

WEXITED — wait for the process(es) to exit,

WNOHANG — return immediately if there are no children to wait for.)

the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)

If the child process has already exited, this returns immediately - otherwise, it blocks

While waiting for children to terminate...

- Process termination scenarios
  - By calling exit() (exit is called automatically when end of main is reached)
  - OS terminates a misbehaving process

- Terminated process exists as a zombie
  - When a parent calls wait(), zombie child is cleaned up

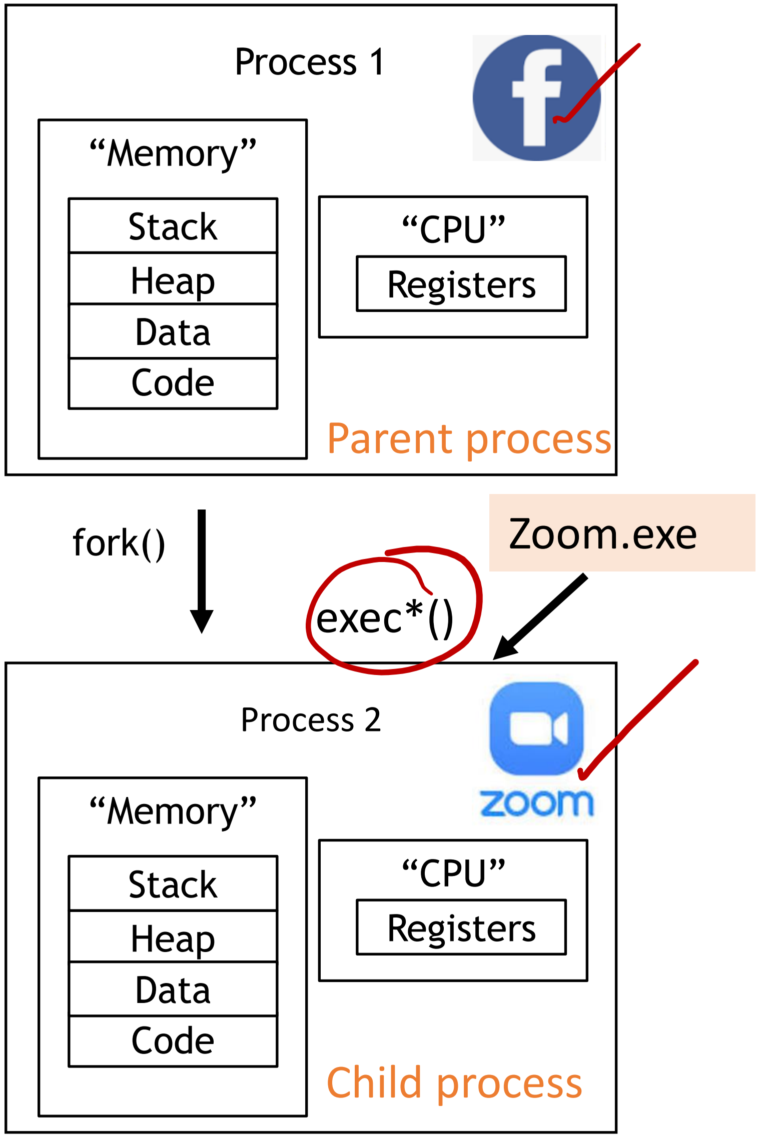
- wait() blocks in parent until child terminates

- What if parent terminates before child?

orphan

  - init process adopts orphans and acquire them

exec*()

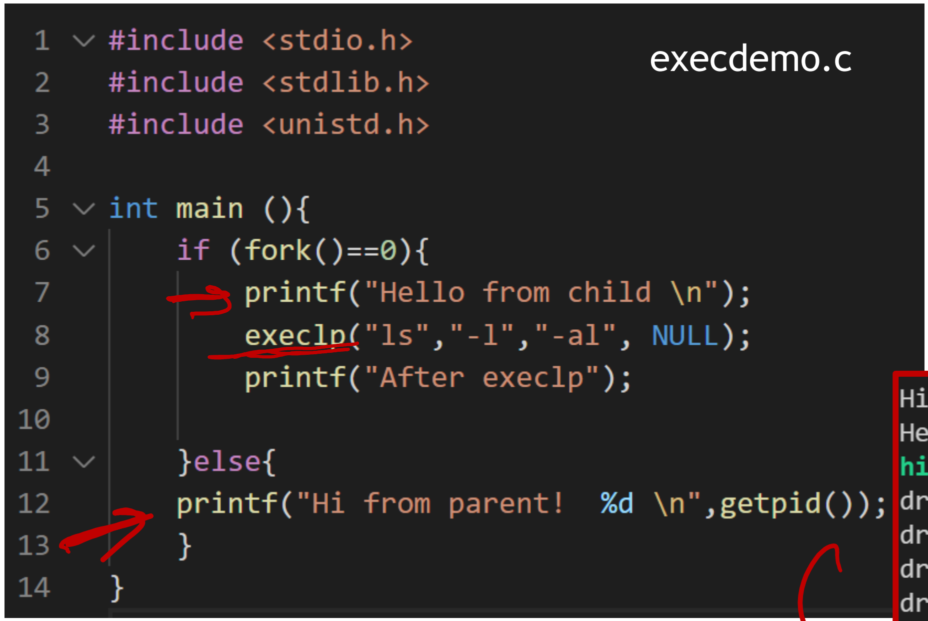


- The fork system call creates a new process.
- The exec() system call replaces the current process image (i.e., the process address space) by that of a specific program (stored on disk as an executable)

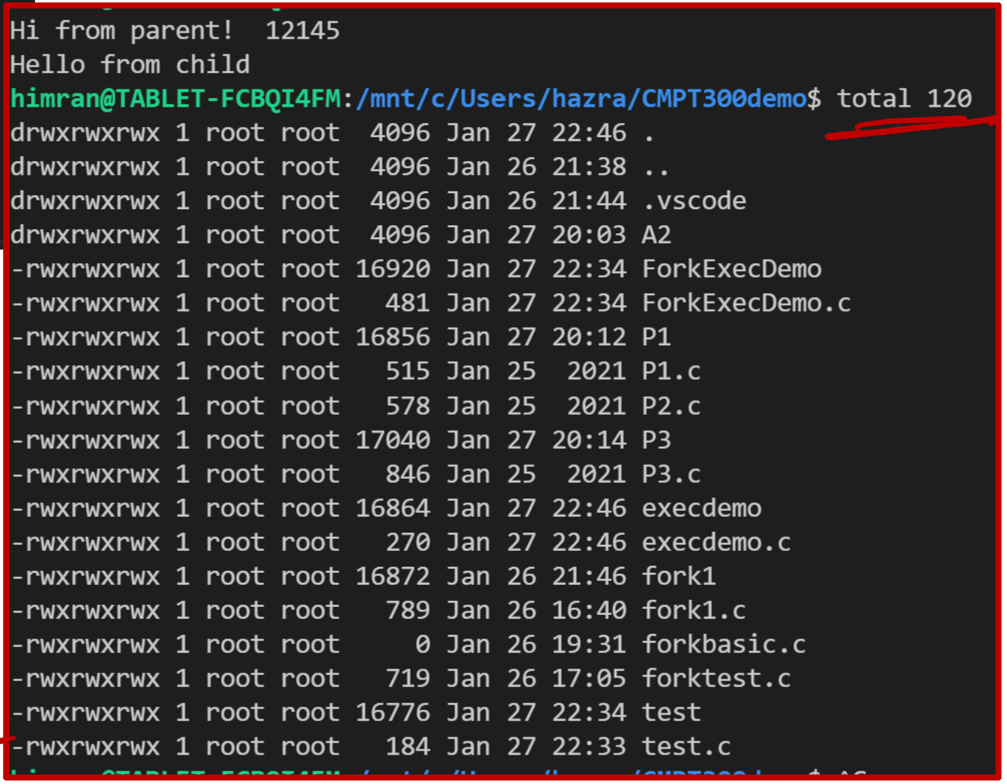
- We give exec
  - A path to an executable
  - Arguments to be passed to that executable
  - Set of environment variables

The call to exec() never returns – Unless there is an error

exec*()


execdemo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (){
    if (fork()==0){
        printf("Hello from child \n");
        execlp("ls","-l","-al", NULL);
        printf("After execlp");

    }else{
    printf("Hi from parent!  %d \n",getpid());
    }
}
```



```
Hi from parent!  12145
Hello from child
himran@TABLET-FCBQI4FM:/mnt/c/Users/hazra/CMPT300demo$ total 120
drwxrwxrwx 1 root root  4096 Jan 27 22:46 .
drwxrwxrwx 1 root root  4096 Jan 26 21:38 ..
drwxrwxrwx 1 root root  4096 Jan 26 21:44 .vscode
drwxrwxrwx 1 root root  4096 Jan 27 20:03 A2
-rwxrwxrwx 1 root root 16920 Jan 27 22:34 ForkExecDemo
-rwxrwxrwx 1 root root   481 Jan 27 22:34 ForkExecDemo.c
-rwxrwxrwx 1 root root 16856 Jan 27 20:12 P1
-rwxrwxrwx 1 root root   515 Jan 25  2021 P1.c
-rwxrwxrwx 1 root root   578 Jan 25  2021 P2.c
-rwxrwxrwx 1 root root 17040 Jan 27 20:14 P3
-rwxrwxrwx 1 root root   846 Jan 25  2021 P3.c
-rwxrwxrwx 1 root root 16864 Jan 27 22:46 execdemo
-rwxrwxrwx 1 root root   270 Jan 27 22:46 execdemo.c
-rwxrwxrwx 1 root root 16872 Jan 26 21:46 fork1
-rwxrwxrwx 1 root root   789 Jan 26 16:40 fork1.c
-rwxrwxrwx 1 root root     0 Jan 26 19:31 forkbasic.c
-rwxrwxrwx 1 root root   719 Jan 26 17:05 forktest.c
-rwxrwxrwx 1 root root 16776 Jan 27 22:34 test
-rwxrwxrwx 1 root root   184 Jan 27 22:33 test.c
```

What happens during exec?

- After fork, parent and child are running same code—-Not too useful!
- A process can run exec() to load another executable to its memory image—- So, a child can run a different program from parent
- Variants of exec(), e.g., to pass command line arguments to new executable

exec*()

man them all

execv
execvp
execve

Family 1

execl
execlp
execle

Family 2

argument list has to be NULL terminated

$$\begin{tikzcd} & \text{exec*()} \arrow[dl] \arrow[dr] & \\ \text{v (vector or basically as an array of pointers)} & & \text{l (list- part of argument list)} \end{tikzcd}$$

p (path searching)

e (to pass a different set of environment var to new program)

Process Termination

- Process executes last statement and asks OS to delete it (**exit**)
  - Process' resources are de-allocated by OS
  - Output data from child to parent (via **wait**)

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some OS do not allow children to continue
    - Children of the children are terminated (cascade termination)

Process Termination

- Should be orderly
  - Child terminates before parent and parent is waiting ✓

- Zombie process
  - If child exits when parent is not waiting
  - Child becomes a "zombie"
  - Return value held in memory
  - OS keeps it around long enough until parent does a wait to get exit status or until parent exits

- Orphan process
  - Process running but parent exited without calling wait()
  - Adoption (init process in Linux)

Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process

- ***Cooperating*** process can affect or be affected by the execution of another process

- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Finally… How does a shell work?

- In a basic OS, the init process is created after initialization of hardware
- The init process spawns a shell like bash
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like ls are all executables that are simply executed by the shell

Finally… How does a shell work?

- In a basic OS, the init process is created after initialization of hardware
- The init process spawns a shell like bash
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like ls are all executables that are simply executed by the shell

Next

- Thread (Chapter 4)

finish ch-3

Inclass activity