Operations on Processes

Courtesy: Slides are adapted from the textbook "Operating System Concepts" by Silberschatz et al

Operations on Processes

- During the course of execution, a process may create several new processes
  - The creating process is called a parent process, and
  - the new processes are called the children of that process

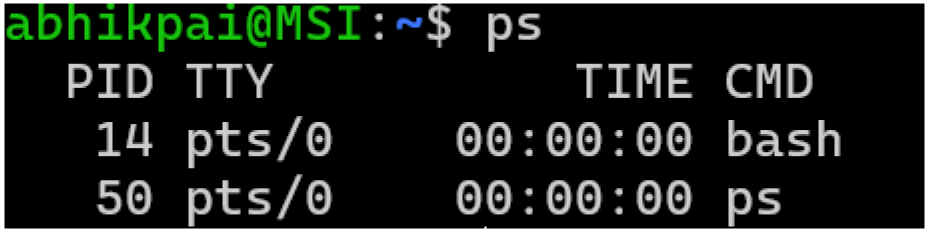
- System must provide mechanisms for:
  - process creation,
  - process termination,

Process Creation

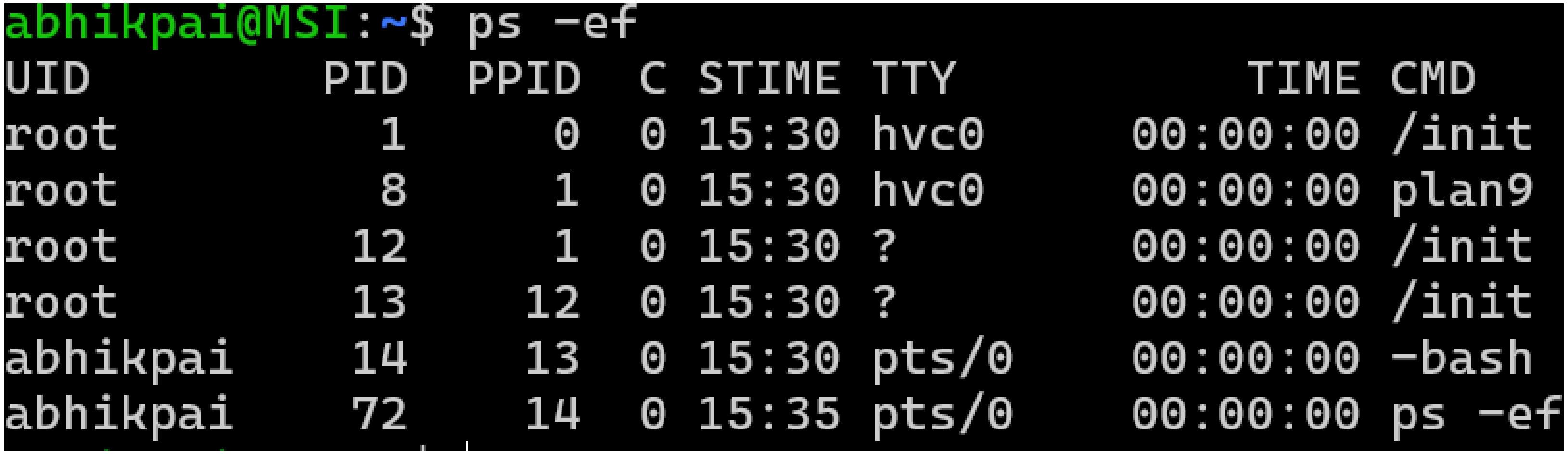
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Process identified and managed via a **process identifier** (**pid**)
  - PID is an integer number
  - The PID provides a unique value for each process in the system, and
  - it can be used as an index to access various attributes of a process within the kernel.

Demo (Linux)

- **ps** (process status) command:
  displays information about a selection of the active processes



```
abhikpai@MSI:~$ ps
  PID TTY          TIME CMD
   14 pts/0    00:00:00 bash
   50 pts/0    00:00:00 ps
```



```
abhikpai@MSI:~$ ps -ef
UID          PID   PPID  C STIME TTY          TIME CMD
root           1      0  0 15:30 hvc0     00:00:00 /init
root           8      1  0 15:30 hvc0     00:00:00 plan9
root          12      1  0 15:30 ?        00:00:00 /init
root          13     12  0 15:30 ?        00:00:00 /init
abhikpai      14     13  0 15:30 pts/0    00:00:00 -bash
abhikpai      72     14  0 15:35 pts/0    00:00:00 ps -ef
```

Demo: Getting Process Info.

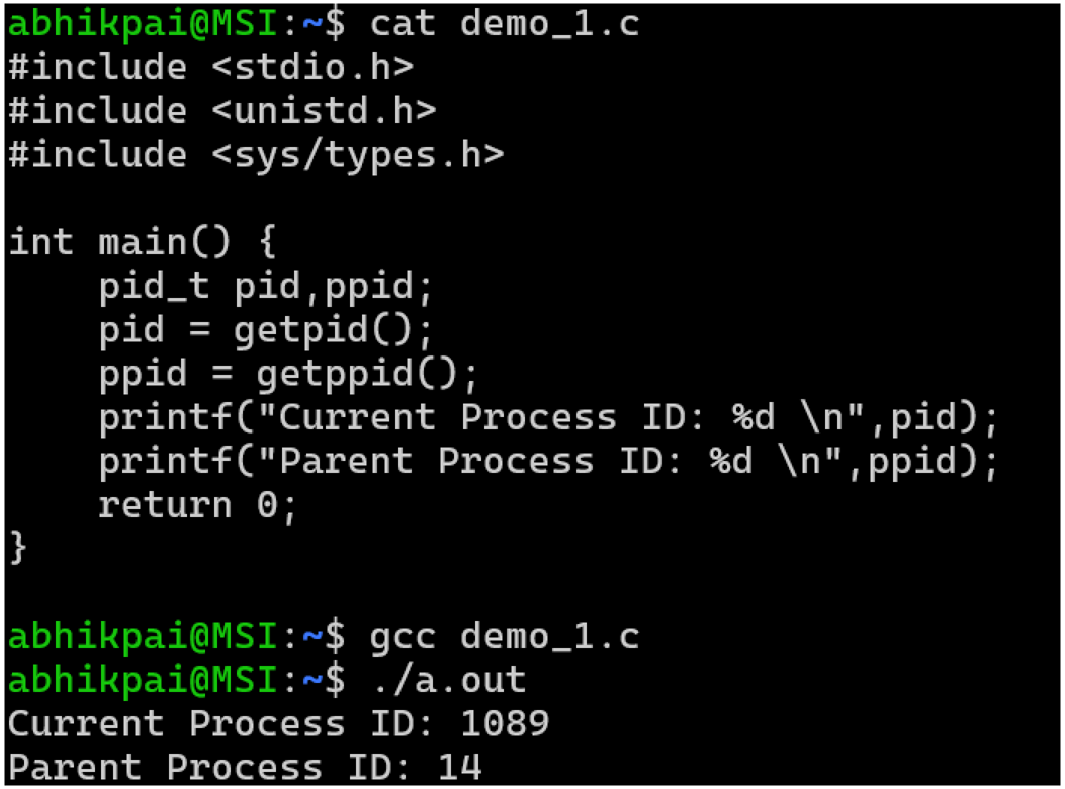
We can programmatically print process IDs using: **getpid (), getppid ()**

This function returns the process identifiers of the calling process.

```
#include <sys/types.h>
#include <unistd.h>
```

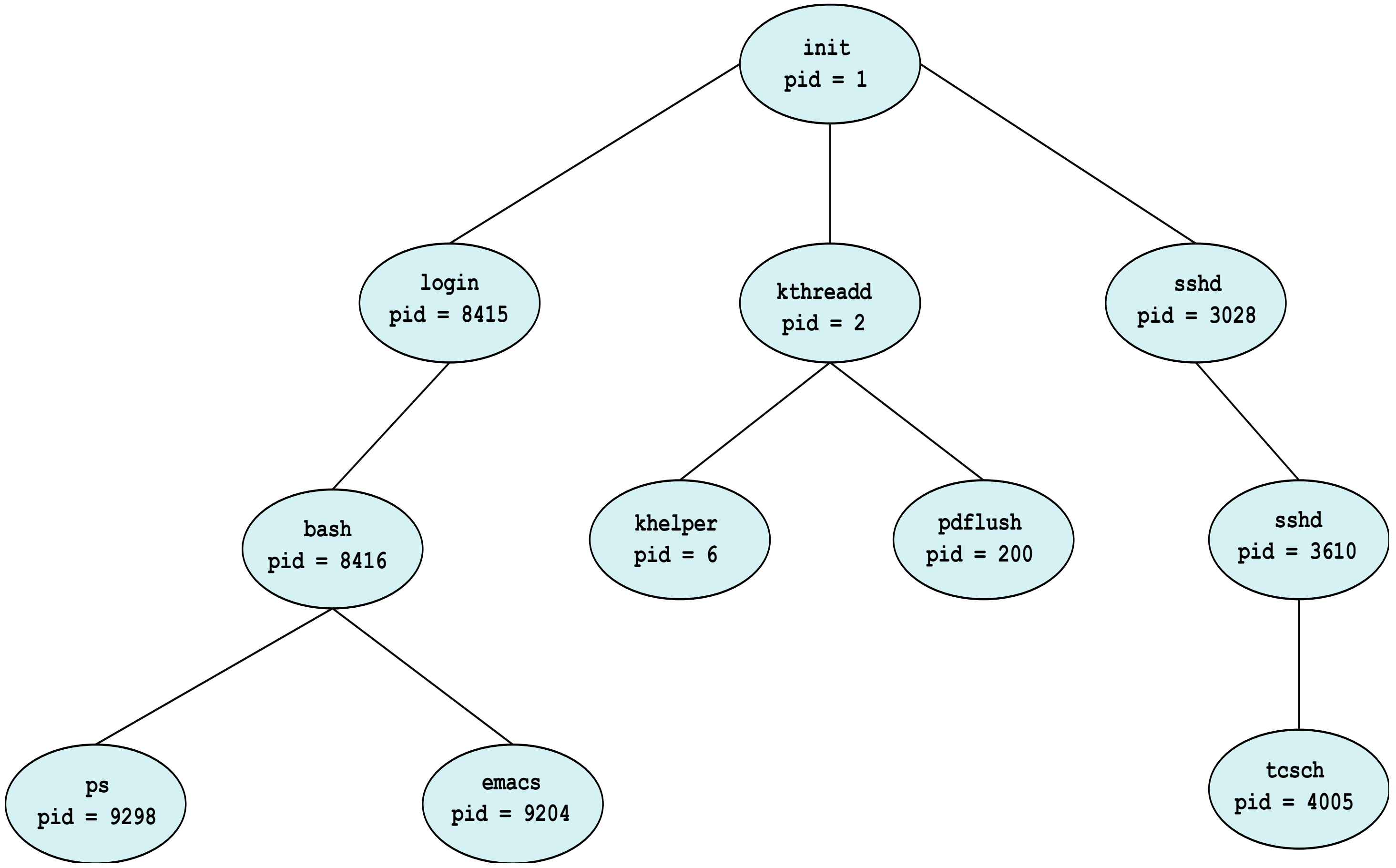
pid_t getpid(void); // this function returns the process identifier (PID)

pid_t getppid(void); // this function returns the parent process identifier (PPID)



```
abhikpai@MSI:~$ cat demo_1.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid,ppid;
    pid = getpid();
    ppid = getppid();
    printf("Current Process ID: %d \n",pid);
    printf("Parent Process ID: %d \n",ppid);
    return 0;
}

abhikpai@MSI:~$ gcc demo_1.c
abhikpai@MSI:~$ ./a.out
Current Process ID: 1089
Parent Process ID: 14
```

A Tree of Processes in Linux



A Tree of Processes in Linux

- The **init process** (which always has a pid of 1) serves as the root parent process for all user processes.
- The **kthreadd process** is responsible for creating additional processes that perform tasks on behalf of the kernel
- The **sshd process** is responsible for managing clients that connect to the system by using ssh (Secure Shell)
- The **login process** is responsible for managing clients that directly log onto the system.
- In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
- Using the bash command-line interface, this user has created the process ps as well as the emacs editor.
- Parent waits until children terminate
- **Pdflush: a set of kernel threads which are responsible for writing the dirty pages to disk**

Process Creation

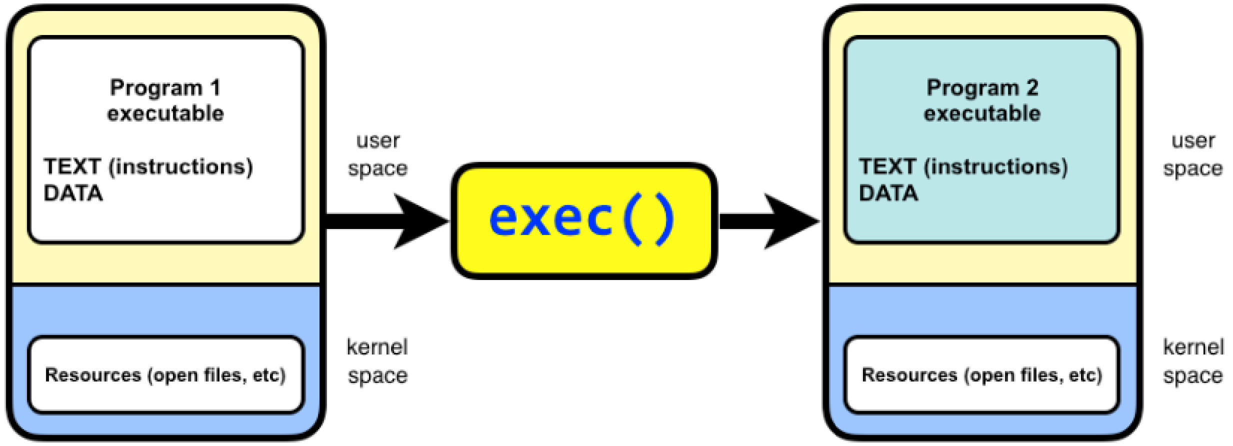
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
    - ***Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes***

Process Creation: fork()

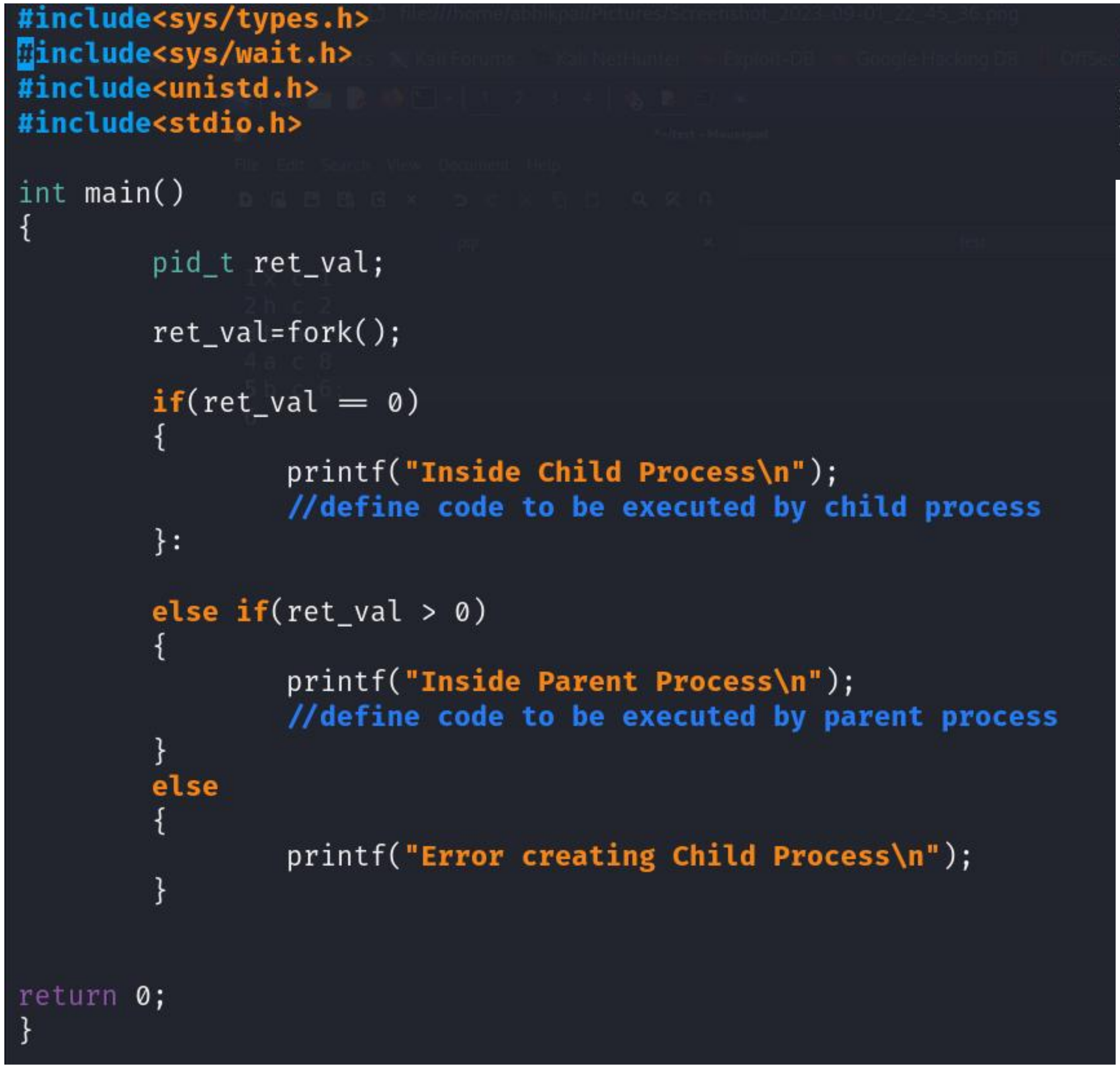
- A new process is created by the **fork()** system call
  - A new process is created by calling fork.
  - This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process.
  - The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors.
  - After a new child process is created, both processes will execute the **next instruction following the fork() system call.**

Process Creation: fork()

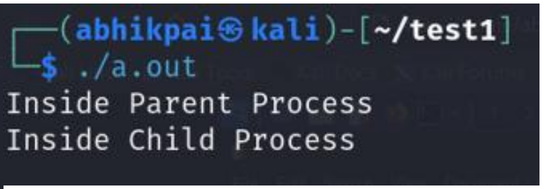
- Different values returned by fork():
  - ***Negative Value***: creation of a child process was unsuccessful.
  - ***Zero***: Returned to the newly created child process.
  - ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program
- The exec() system call loads a binary file into memory and starts its execution
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child



Demo: fork()



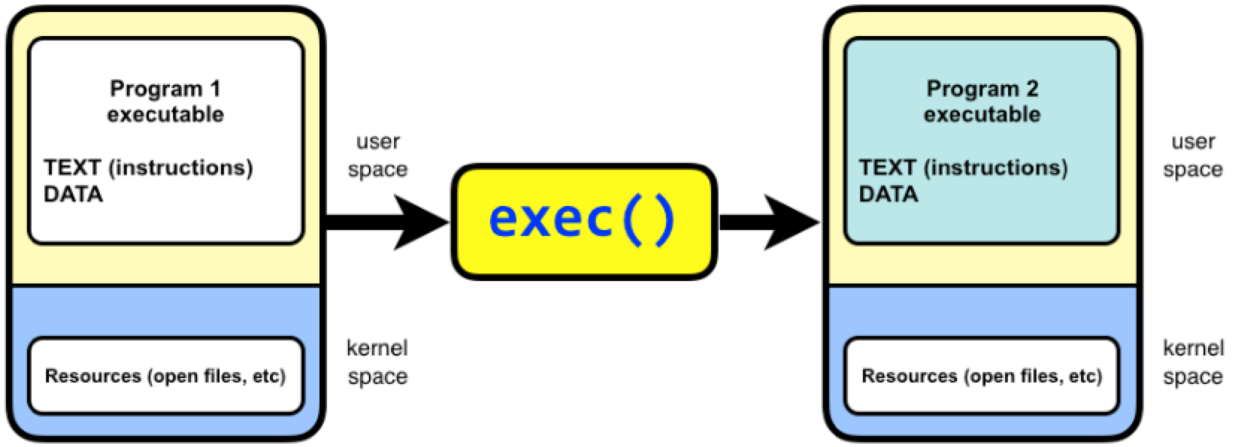
```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
        pid_t ret_val;

        ret_val=fork();

        if(ret_val == 0)
        {
                printf("Inside Child Process\n");
                //define code to be executed by child process
        }:

        else if(ret_val > 0)
        {
                printf("Inside Parent Process\n");
                //define code to be executed by parent process
        }
        else
        {
                printf("Error creating Child Process\n");
        }


return 0;
}
```



```
┌──(abhikpai㉿kali)-[~/test1]
└─$ ./a.out
Inside Parent Process
Inside Child Process
```

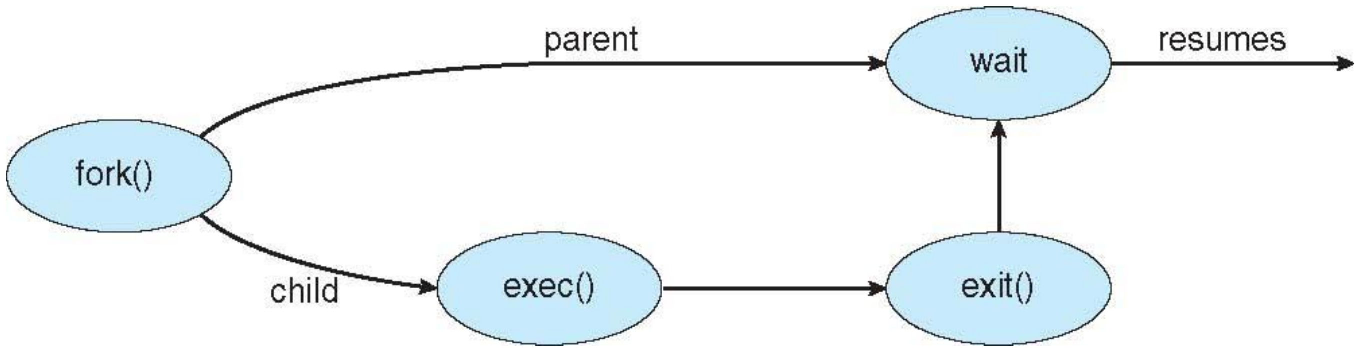
Process Creation

- When a process creates a new process, two possibilities for execution exist:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- There are also two address-space possibilities for the new process:
  - **1.** The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - **2.** The child process has a new program loaded into it.

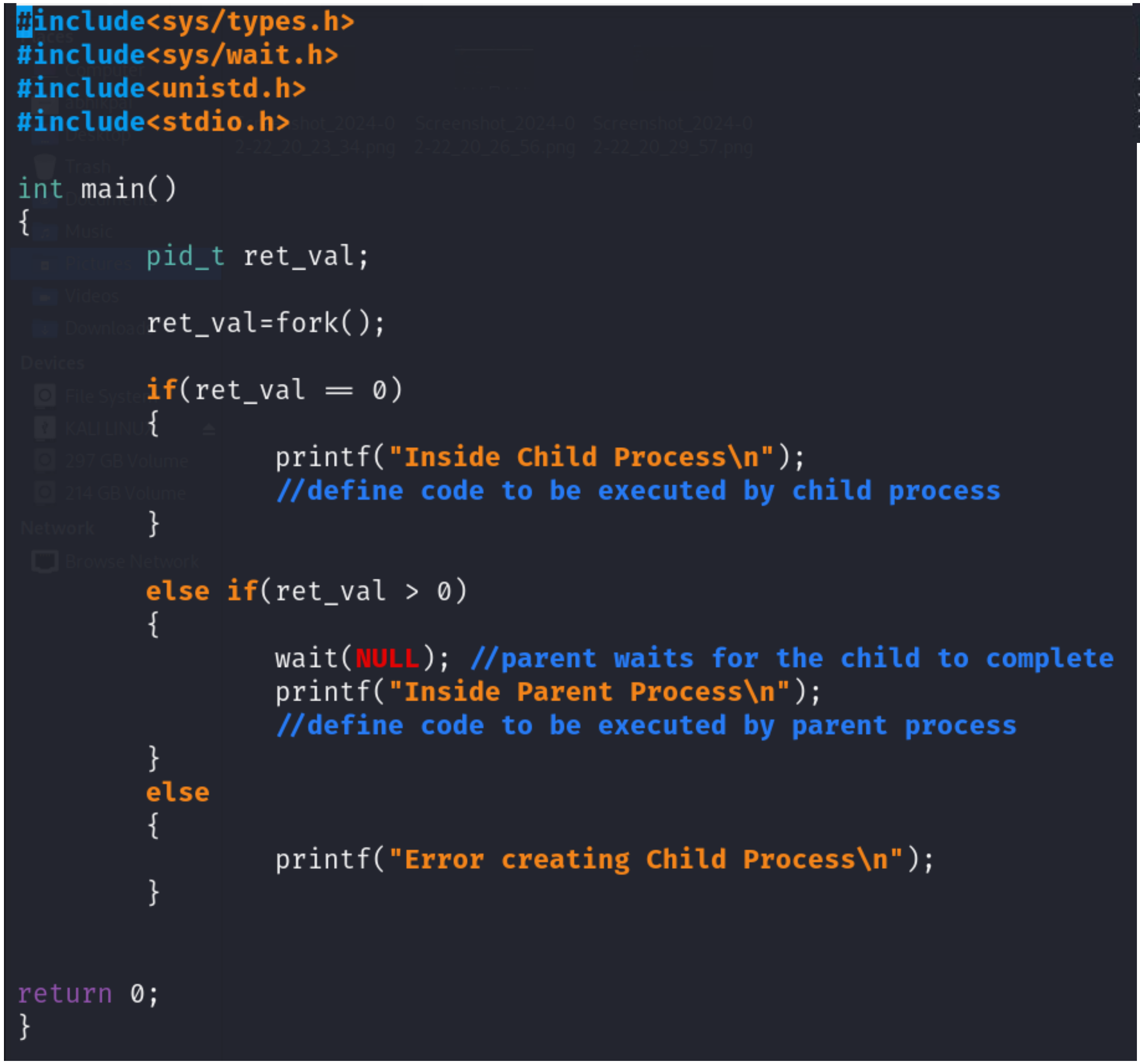


Process Creation (Cont.)

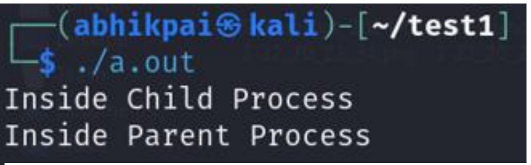
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Demo: wait()



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

int main()
{
        pid_t ret_val;

        ret_val=fork();

        if(ret_val == 0)
        {
                printf("Inside Child Process\n");
                //define code to be executed by child process
        }

        else if(ret_val > 0)
        {
                wait(NULL); //parent waits for the child to complete
                printf("Inside Parent Process\n");
                //define code to be executed by parent process
        }
        else
        {
                printf("Error creating Child Process\n");
        }


return 0;
}
```



```
┌──(abhikpai㉿kali)-[~/test1]
└─$ ./a.out
Inside Child Process
Inside Parent Process
```

Program for Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Termination

1. Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
   - Returns status data from child to parent (via **`wait()`**)
   - Process' resources are deallocated by operating system
2. A process can cause the termination of another process via an appropriate system call
   - Usually, such a system call can be invoked only by the parent of the process.
   - Otherwise, users could arbitrarily kill each other's jobs.
   - Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

Process Termination

- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

1. Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
   - **cascading termination.** All children, grandchildren, etc. are terminated.
   - The termination is initiated by the operating system.

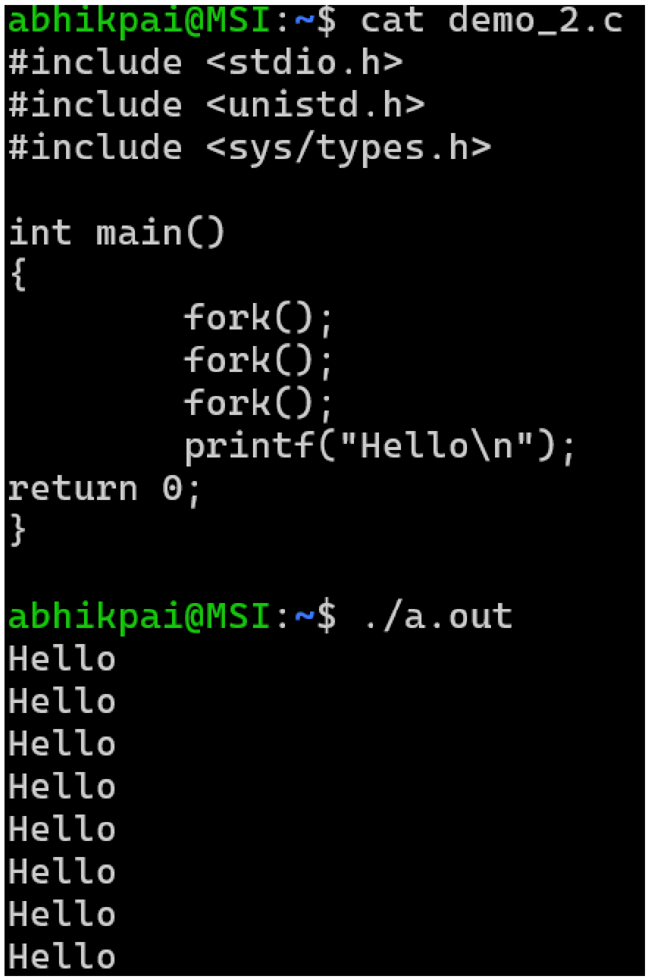
Process Termination

- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

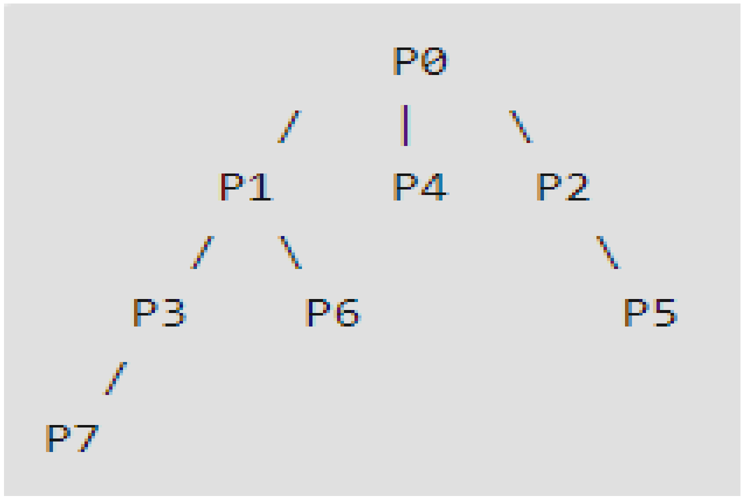
- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`** , process is an **orphan**
- Linux and UNIX address this scenario by assigning the **init process** as the new parent to orphan processes

Demo: fork()



```
abhikpai@MSI:~$ cat demo_2.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
        fork();
        fork();
        fork();
        printf("Hello\n");
return 0;
}

abhikpai@MSI:~$ ./a.out
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

Suppose, the main process is : P0
Processes created by the 1st fork: P1
Processes created by the 2nd fork: P2, P3
Processes created by the 3rd fork: P4, P5, P6, P7



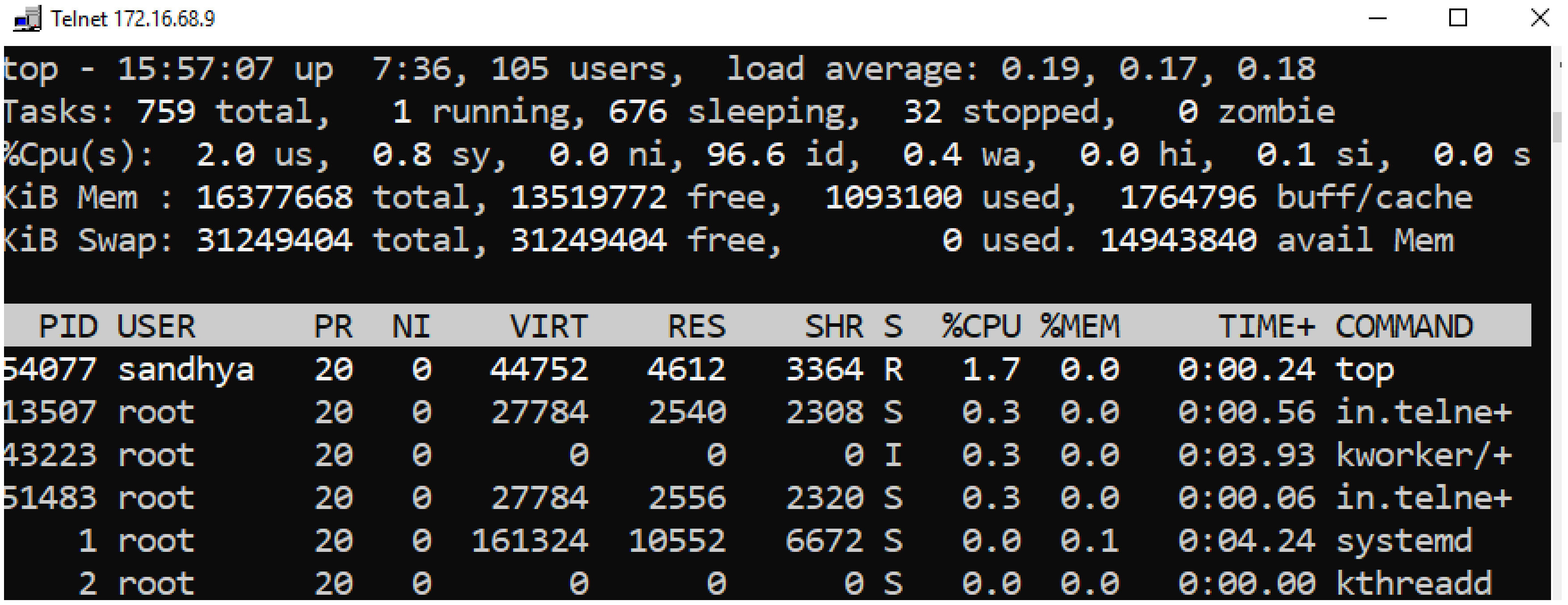
```
                P0
           /    |    \
        P1      P4    P2
       /  \             \
     P3    P6            P5
    /
  P7
```

In general, if there are n fork system calls, the total number of child processes created= 2^n-1.

Try these commands in Linux!

top

This utility tells the user about all the running processes on the Linux machine.

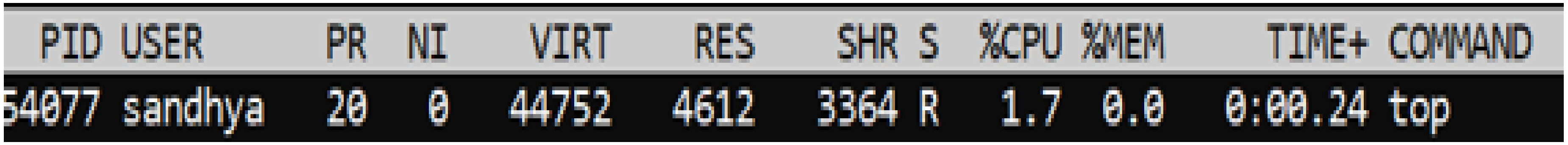


Telnet 172.16.68.9

```
top - 15:57:07 up  7:36, 105 users,  load average: 0.19, 0.17, 0.18
Tasks: 759 total,   1 running, 676 sleeping,  32 stopped,   0 zombie
%Cpu(s):  2.0 us,  0.8 sy,  0.0 ni, 96.6 id,  0.4 wa,  0.0 hi,  0.1 si,  0.0 s
KiB Mem : 16377668 total, 13519772 free,  1093100 used,  1764796 buff/cache
KiB Swap: 31249404 total, 31249404 free,        0 used. 14943840 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
54077 sandhya   20   0   44752   4612   3364 R   1.7  0.0   0:00.24 top
13507 root      20   0   27784   2540   2308 S   0.3  0.0   0:00.56 in.telne+
43223 root      20   0       0      0      0 I   0.3  0.0   0:03.93 kworker/+
51483 root      20   0   27784   2556   2320 S   0.3  0.0   0:00.06 in.telne+
    1 root      20   0  161324  10552   6672 S   0.0  0.1   0:04.24 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
```

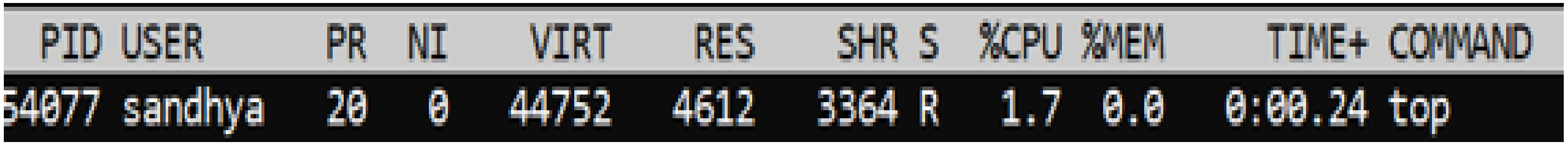
Press 'q' on the keyboard to move out of the process display

Field	Description	Example 1	Example 2
PID	The process ID of each task	1525	961
User	The username of task owner	Home	Root
PR	Priority Can be 20(highest) or -20(lowest)	20	20
NI	The nice value of a task	0	0
VIRT	Virtual memory used (kb)	1775	75972
RES	Physical memory used (kb)	100	51
SHR	Shared memory used (kb)	28	7952



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
54077 sandhya   20   0   44752   4612   3364 R   1.7  0.0   0:00.24 top
```

S	Status There are five types: 'D' = uninterruptible sleep 'R' = running 'S' = sleeping 'T' = traced or stopped 'Z' = zombie	S	R
%CPU	% of CPU time	1.7	1.0
%MEM	Physical memory used	10	5.1
TIME+	Total CPU time	5:05.34	2:23.42
Command	Command name	Photoshop.exe	Xorg



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
54077 sandhya   20   0   44752   4612   3364 R   1.7  0.0   0:00.24 top
```

Try these commands in Linux!

kill

- **terminates running processes** on a Linux machine.

  kill PID
- To find the PID of a process

  pidof Process name

- When running in foreground, hitting Ctrl + c (interrupt character) will exit the command

- If a process ignores a regular kill command, you can use kill -9 followed by the process ID

Try these commands in Linux!

Command	Description
bg	To send a process to the background
fg	To run a stopped process in the foreground
top	Details on all Active Processes
ps	Give the status of processes running for a user
ps PID	Gives the status of a particular process
pidof	Gives the Process ID (PID) of a process
kill PID	Kills a process
nice	Starts a process with a given priority
renice	Changes priority of an already running process
df	Gives free hard disk space on your system
free	Gives free RAM on your system

Practice Questions

- Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

```
#include <stdio.h>
#include <unistd.h>
int main()
{ /* fork a child process */
            fork();
  /* fork another child process */
            fork();
 /* and fork another */
            fork();
            return 0;
}
```

Figure 3.31 How many processes are created?

Practice Questions

- When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?

  a. Stack

  b. Heap

  c. Shared memory segments

- Describe the differences among short-term, medium-term, and long term scheduling.

Practice Questions

- Using the program in Figure 3.34, identify the values of pid at lines A, B,C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     pid1 = getpid();
     printf("child: pid = %d",pid); /* A */
     printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
     pid1 = getpid();
     printf("parent: pid = %d",pid); /* C */
     printf("parent: pid1 = %d",pid1); /* D */
     wait(NULL);
   }

   return 0;
}
```

Figure 3.34 What are the pid values?