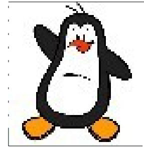


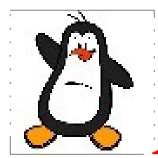
Video Lecture # 17
Process Management
Part - I

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

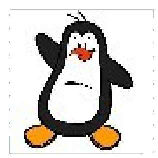
Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: **https://bitbucket.org/arifpucit/spvl-repo/src**
Lecture Slides available at: **http://arifbutt.me**

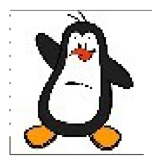


Agenda

- Overview of Processes in Linux
- The `task_struct` Structure
- The `/proc` Directory
- Accessing Process Identifications
- Modifying Process Identifications
- Process Creation using `fork()`
- Process Trees
- Process Chains
- Process Fans



Processes vs Files



Kernel Data Structures

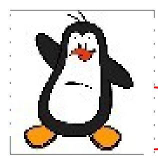
Executable Object File

Executable Object File
ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)

Process Address Space

Address	Region	Note
0xffffffff / 0xc0000000	Kernel virtual memory	Memory invisible to user code
0xbfffffff	User stack (created at runtime)	%esp (stack pointer)
	↓ ↑ (unallocated)	
0x40000000	Memory-mapped region for shared libraries	
	↑ (unallocated)	brk
end	Run-time heap (created by `malloc`)	
edata	Read/write segment (.data, .bss)	Loaded from the executable file
etext / 0x08048000	Read-only segment (.init, .text, .rodata)	Loaded from the executable file
0x00000000	Unused	

0 (offset of ELF header)



Process Control Block (task_struct)

A **Process** is an entity that can be assigned to and executed by a **Processor**

1. Process identification:

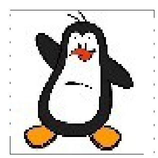
- PID & PPID
- UID & GID
- EUID & EGID
- Saved SUID& SGID
- File System UID & GID
- Supplementary GIDs

2. Process State Information:

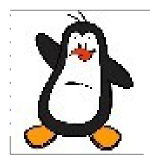
- User Visible Registers
- Control and Status Registers (flags)

3.Process Control Information:

- Scheduling info
- Privileges info
- Memory management info
- Resource ownership and utilization
- IPC



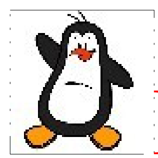
Accessing Process IDs



Process ID & Parent Process ID

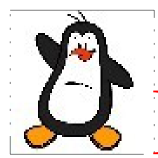
```
pid_t getpid();
pid_t getppid();
```

- Above calls returns PID and PPID of the current process. Never fails
- Each process has a Process ID (PID), a positive integer that uniquely identifies a process on the system
- Linux kernel limits PIDs to being less than or equal to 32767. Once it has reached 32767, the PID counter is reset to 300, rather than 1
- On Linux 2.4 & earlier, PID limit of 32767 is defined by kernel constant PID_MAX
- On Linux 2.6, the default upper limit for PIDs remain 32767 & is adjustable via **`/proc/sys/kernel/pid_max`** file
- On 32 bits platforms, the maximum value for this file is 32768 but on 64-bit platforms, it can be adjusted to any value up to 2^{22}(approx 4 million)



Process ID & Parent Process ID (cont...)

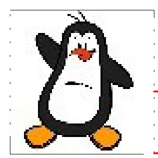
- On a shell you can get the PID of the shell in the environment variable in $$ and the parent ID in environment variable **PPID**
- The parent of any process can be found by looking at the 4th field of `/proc/PID/stat` file. Also see the 3rd field which shows the state of the process (RSDZTX). (See man page of `proc` for details)
- The **swapper** or scheduler is a system process having a PID of 0. It manages memory allocation for processes, swaps processes from run state to Ready Queue or other and may be to disk. No program file for swapper in `/proc/` directory
- The `init`, now `systemd` is a user process having a PID of 1. It is invoked by the kernel at the end of the booting process
- **Page daemon** now `kthreadd` is a system process having a PID of 2. It support the paging of virtual memory system



Real User ID & Real Group ID

```
uid_t getuid();
gid_t getgid();
```

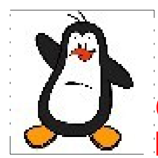
- The real user ID & real group ID identify the user and group to which the process belongs. (Defines ownership of the process)
- As part of the login process, a login shell gets it real UID and real GID from the 3rd & 4th field of the user's password record in **/etc/passwd** file
- When a new process is created (when a shell **exec** a program), it inherits these IDs from its parent
- Above two calls returns the real UID and real GID of the current process. Never fails



Effective User ID & Effective Group ID

```
uid_t geteuid();
gid_t getegid();
```

- On most Unix implementations, the effective IDs are used to determine a process's permission when accessing resources such as files, system V IPC objects, which themselves have associated user and group IDs determining to which they belong
- Normally the effective IDs have the same values as the real IDs but there are two ways using which the effective IDs can take different values:
  - By execution of programs having their SUID & SGID bit set
  - Use of system calls (**`setuid(), setgid()`**.....)



Saved Set-User-ID and Saved Set-Group-ID

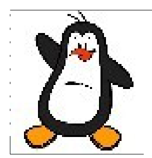
- These two IDs are designed for use with executable programs having their SUID/SGID bit set
- When your shell executes a program with its SUID bit set then the effective IDs of the process are made the same as the owner of the executable. If the SUID bit of the program is not set, then no change is made to the effective IDs of the process, i.e, they remain the same as real IDs of the process. Finally (after both cases) the EUID and EGID are copied to the saved SUID & saved SGID respectively
- Example: Suppose a process whose real, effective & saved set-user-ID are all 1000, **`execs`** a program with its SUID bit set & owned by root. After the **`exec`**, the user IDS of the process will be changed as follows:

  real=1000 effective=0 saved=0

Real Effective and Saved User IDs

```
int getresuid(uid_t *ruid, uid_t*euid, uid_t*suid);
int getresgid(gid_t *rgid, gid_t*egid, gid_t*sgid);
```

- **`getresuid()`** & **`getresgid()`** returns the current values of the calling process real, effective and saved user/group IDs in the locations pointed by these arguments
- On success 0 is returned, on error -1 is returned and `errno` is set appropriately
- Only error is EFAULT, i.e., one of the arguments specified an address outside the calling process's address space



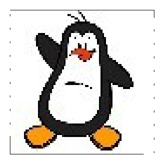
Other IDs

File System User ID & File System Group ID

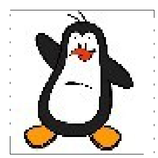
- On Linux it is the File-System user and group IDs rather than Effective user & group IDs that are used to determine permission when performing file system operations such as opening file, changing file ownership, modifying file permission

Supplementary Group IDs

- The supplementary group IDs are a set of additional groups (secondary defined in **`/etc/group`**) to which a process belongs
- A login shell obtain these from the **`/etc/group`** file while a new process inherit these IDs from its parent



Accessing PIDs
Proof of concepts



Modifying Process IDs

Modifying Effective IDS

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Changes the effective IDs and possibly the real and the saved-set IDs of the calling process to the value given by the argument. On success returns 0, and -1 on error
- Following rules govern the changes that a process can make to its IDs using **`setuid()`** and **`setgid()`**
  - When an unprivileged process calls **`setuid()`** only the effective user ID of the process is changed. Furthermore, it can be changed only to the same values as either the real user ID or saved set user-ID
  - When a privileged process calls **`setuid()`** with a non zero argument then real, effective & saved set user ID are all set to the value specified in the `uid/gid` argument. *This is a one way trip*, i.e. subsequently the process cannot use **`setuid()`** to reset the identifiers back to 0

Modifying Effective IDS (cont...)

```
int seteuid(uid_t euid);
int setegid(gid_t egid);
```

- Changes the effective IDs to the value given by the argument
- Following rules govern the changes that a process can make to its effective IDs using **`seteuid()`** and **`setegid()`**
  - When an unprivileged process calls **`seteuid()`** only the effective user ID of the process is changed. Furthermore, it can be changed only to the same values as either the real user ID or saved set user-ID
  - When a privileged process calls **`seteuid()`** with a non zero argument then only effective ID is set to the value specified in the **`euid/egid`** argument. However, *this is **NOT** a one way trip*, i.e. if a privileged process uses **`seteuid()`** to change its effective user ID to a nonzero value, it ceases to be privileged but later it can again use **`seteuid()`** to change its effective ID to 0

Modifying Effective IDS (cont...)

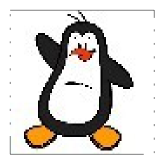
```
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

- Allows the calling process to independently change to values of its real and effective user IDs. If you want to change only one of the identifiers, then we can specify -1 for the other argument.
- Following rules govern the changes that a process can make to its IDs using **`setreuid()`** and **`setregid()`**
  - An unprivileged process can set the real user ID only to the current value of the real or effective user ID. Effective user ID can be set only to the current value of the real, effective or saved set user ID
  - A privileged process can independently change real, and effective IDs to any values. (Changing the effective ID implicitly changes the Saved ID as well)

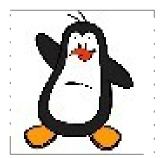
Modifying Real, Effective & saved set IDS

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, uid_t sgid);
```

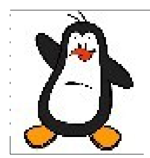
- Allows the calling process to independently change the values of all three IDs. Specifying -1 leaves the corresponding ID unchanged
- Following rules govern the changes that a process can make to its IDs using **`setresuid()`** and **`setresgid()`**
  - An unprivileged process can set the real, effective & saved set user ID to any of the values currently in its current real, effective or saved set user IDs
  - A privileged process can independently change its real, effective & saved set user ID to any values



Modifying PIDs
Proof of concepts



Process Creation



Process Creation using `fork()`

```
pid_t fork();
```

- The **`fork()`** system call allows one process, *the parent*, to create a new process, *the child*
- It is a system call which is called once but return twice, once in the parent and once in the child. To the parent process it returns PID of child process and to the child process it returns zero. After the call returns both parent and child processes continues their execution concurrently from the next line of code
- The child process is a clone of the parent and obtains copies of the parent's **`stack, data, heap,and text segments`**
- PIDs are allocated sequentially to the new child processes, so effectively unique (but do wrap up after a very long time)

- **On success**, the return value to the child process is 0 and the return value to the parent process is PID of the child
- **On failure**, a -1 will be returned in the parent context, no child process created, and `errno` will be set appropriately

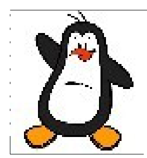
Process Creation using `fork()` (...)

Three main reasons of failure:

- **EAGAIN:** A system-imposed limit on the number of processes was encountered. There are number of factors that can trigger this, e.g.,
  - Number of processes under one user has reached
  - Kernel limit on total number of processes has reached
- **ENOMEM:** Failed to allocate the necessary kernel structures because memory is tight
- **ERESTARTNOINTR:** System call interrupted by a signal and will be restarted

Why `fork()` is called once but return twice?

- **Return value to the child process is 0**, because 0 is the ID of swapper process only and a child process can always call **`getppid()`** to obtain its parent's ID
- **Return value to the parent process is PID of child**, because a process can have more than one child and there is no function that allows a process to obtain PIDs of its children



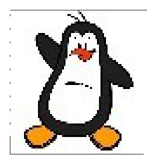
Process Creation using `fork()` (...)

pid = 597

code

data

system data
e.g. file descriptors



Process Creation using `fork()` (...)

597

code

data

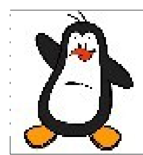
system data

632

code

data

system data



Process Creation using `fork()` (...)

- **Parent forks**

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.       int i = 54, cpid = -1;
5. →     cpid = fork();
6.       if (cpid == -1)
7.       {
8.              printf ("\nFork failed\n");
9.              exit (1);
10. }
11. if (cpid == 0)       //child code
12.     printf ("\n Hello I am child \n");
13. else                 //parent code
14.     printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54
cpid = -1

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.   int i = 54, cpid = -1;
5.   cpid = fork();
6.   if (cpid == -1)
7.   {
8.          printf ("\nFork failed\n");
9.          exit (1);
10. }
11. if (cpid == 0)    //child code
12.         printf ("\n Hello I am child \n");
13. else                 //parent code
14.         printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54
cpid = 632

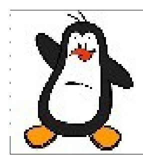
PID: 632 **Child**

```
1. //fork1.c
2. int main()
3. {
4.   int i = 54, cpid = -1;
5.   cpid = fork();
6.   if (cpid == -1)
7.   {
8.          printf ("\nFork failed\n");
9.          exit (1);
10. }
11. if (cpid == 0)    //child code
12.         printf ("\n Hello I am child \n");
13. else                 //parent code
14.         printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54
cpid = 0

- After the **`fork()`** system call parent and child are identical except for the return value of fork (and of course their PIDs)
- They are free to execute on their own from now onwards, i.e., after a successful or unsuccessful **`fork()`** system call both will start their execution from line#6



Process Creation using `fork()` (...)

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.  if (cpid == -1)
7.  {
8.          printf ("\nFork failed\n");
9.          exit (1);
10. }
11. if (cpid == 0)    //child code
12.         printf ("\n Hello I am child \n");
13. else                     //parent code
14. ──>     printf ("\n Hello I am parent \n");
15. }
```

DATA

i – 54
cpid = 632

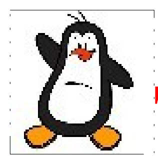
PID: 632 **Child**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.  if (cpid == -1)
7.  {
8.          printf ("\nFork failed\n");
9.          exit (1);
10. }
11. if (cpid == 0)    //child code
12. ──>     printf ("\n Hello I am child \n");
13. else                     //parent code
14.         printf ("\n Hello I am parent \n");
15. }
```

DATA

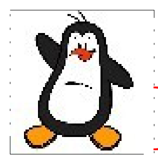
i – 54
cpid = 0

- When both will execute line#11, parent will now execute line#12, while child will execute line#14



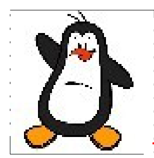
Two main uses of `fork()`

- When a process wants to duplicate itself, so that parent & child each can execute different sections of code concurrently. Example: Consider a network server; parent waits for a service request from a client. When the request arrives, parent calls `fork` & let the child handle the request. Parent goes back to listen for the next request
- When a process wants to execute a different program. This is common for command shells where the child does an **`exec()`** right after it returns from the `fork`



Race Condition after a `fork()`

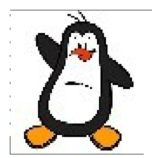
- After a `fork()` it is indeterminate which process – the parent or child will execute. On a multiprocessor system they may both get simultaneous access to the CPU. In most of the cases, the parent will execute first. The two options are:
  - Parent first after `fork()`
  - Child first after `fork()`
- In Linux 2.6.32, since parent's state is already active in the CPU and its memory management information is already cached in the h/w TLB, therefore, running the parent first should result in better performance



Attributes Inherited after `fork()`

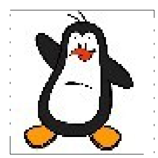
- Real, effective and saved UIDs / GIDs
- Open file descriptors (PPFDT)
- Environment variables
- Present working directory
- Nice value
- File mode creation mask (umask)
- Signal mask and signal disposition
- Attached shared memory segments

Note: Since the child and parent processes have copies of the same PPFDT, so any read/write operations on the files that the parent process has opened before creating the child process will be visible to both processes, because the entry in the SWFT is shared for those files



Difference Between Parent & Child After `fork()`

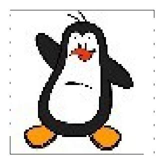
- Child has different PID and PPID
- Return value from fork
- Child't times for CPU usage are reset to 0
- File locks held by the parent (using `fcntl()`) are not inherited by the child
- Set of pending Alarms in the parent are cleared in the child
- Set of pending Signals in the parent are cleared in the child



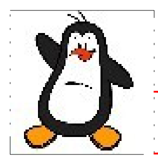
Process Creation
Proof of concept

`fork1.c to fork9.c`

`forkfile.c`



Process Tree, Chain, Fan

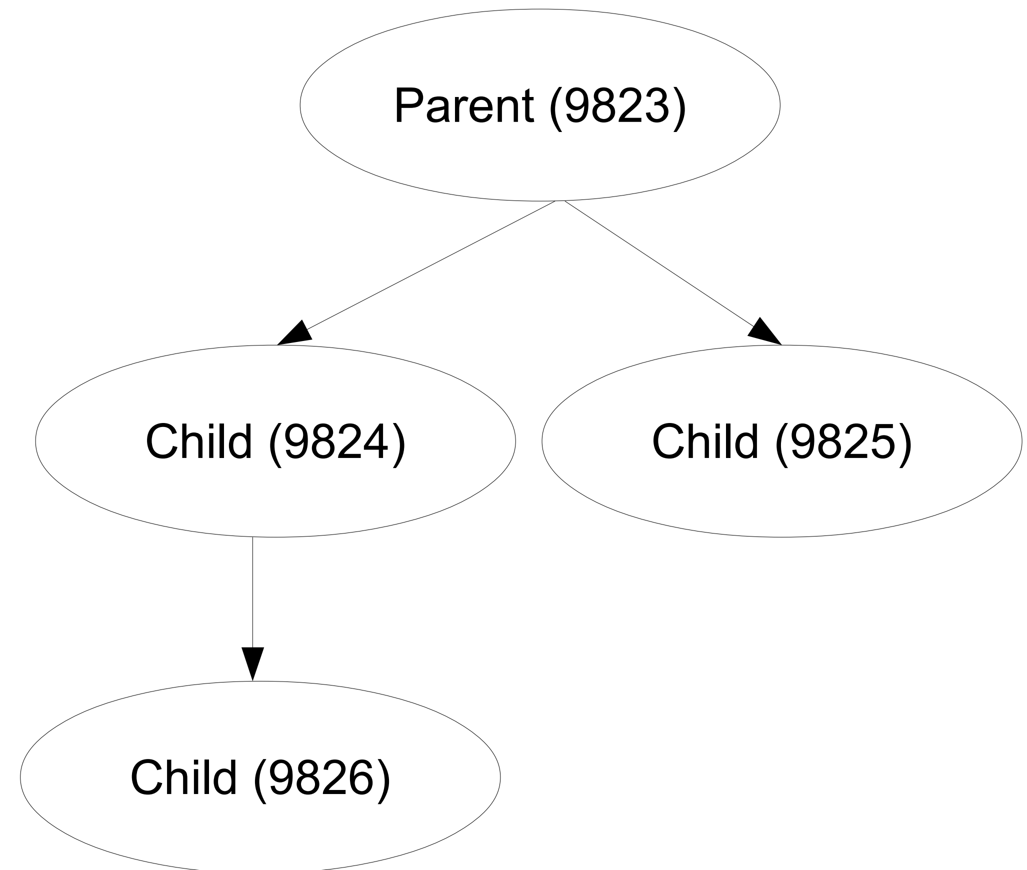


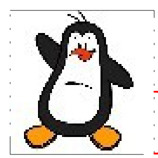
Example: Process Tree

Parent forks once and then child call next fork and so on

A sample o/p of code is:

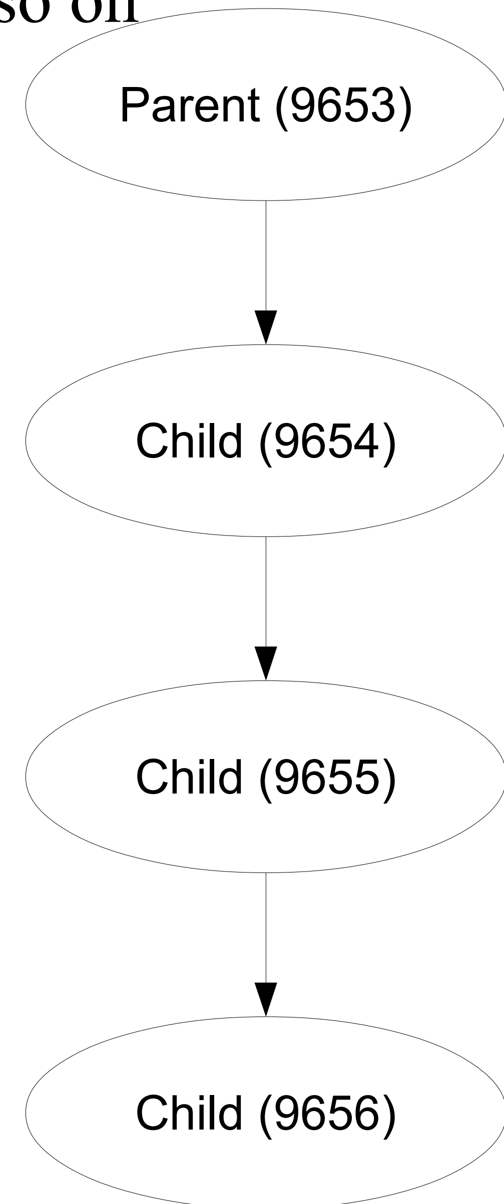
```
arif@kali:~$ echo $$
4920
arif@kali:~$ ./a.out 2
PID=9823, PPID=4920
PID=9825, PPID=9823
PID=9824, PPID=9823
PID=9826, PPID=9824
```





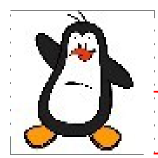
Example: Process Chain

Parent forks once and then child call next fork and so on



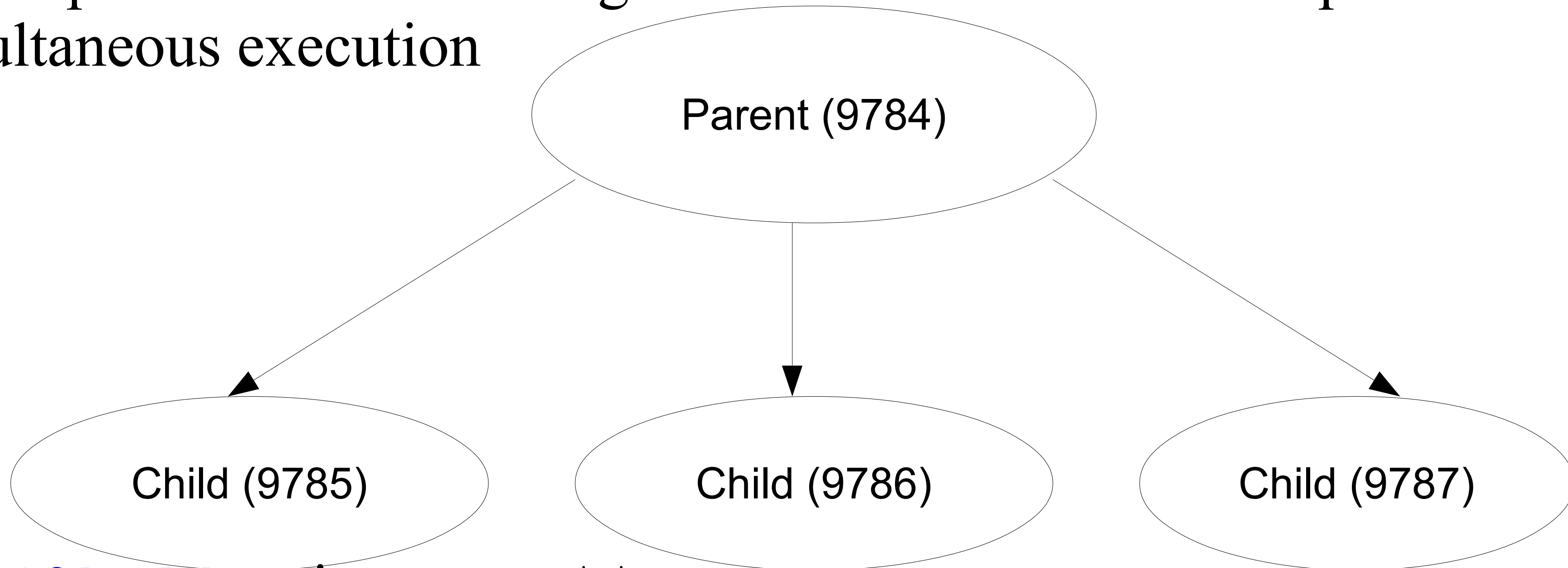
A sample o/p of code is:

```
arif@kali:~$ echo $$
4920
arif@kali:~$ ./a.out 3
PID=9653, PPID=4920
PID=9654, PPID=9653
PID=9655, PPID=9654
PID=9656, PPID=9655
```

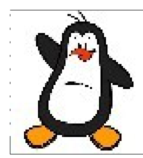


Example: Process Fan

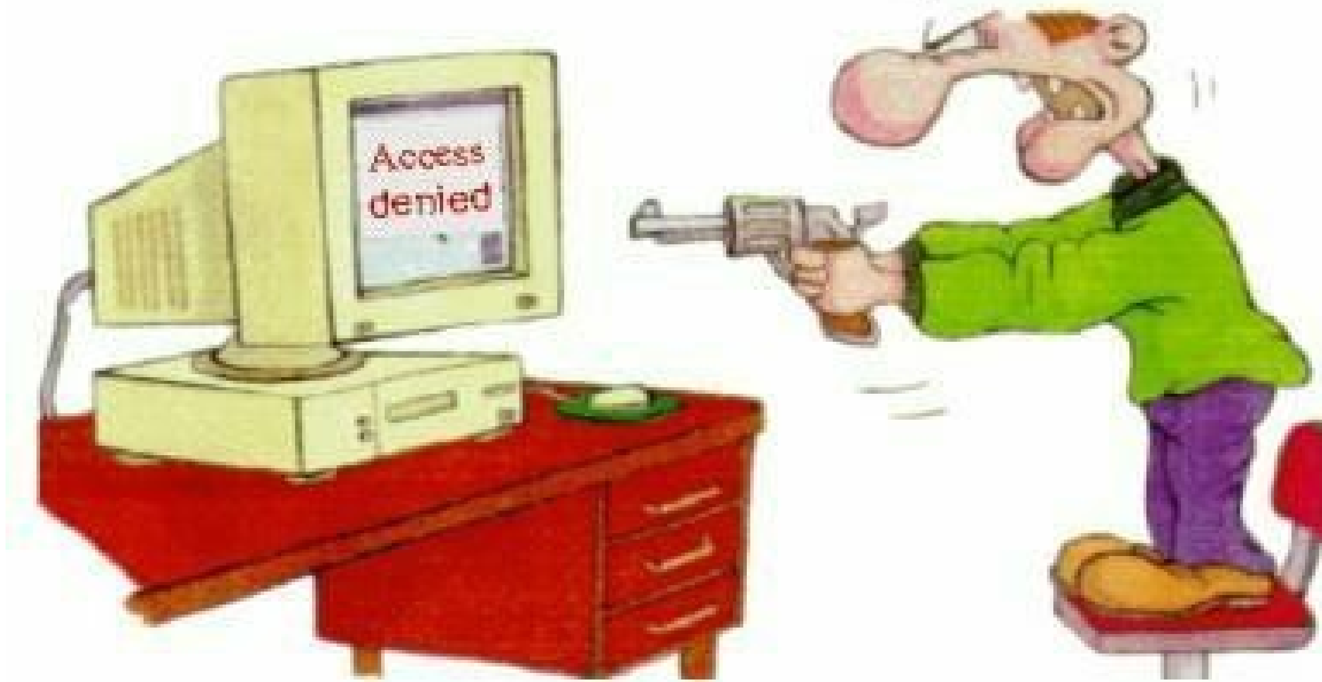
Parent is responsible for every fork. Child processes will break, while parent process will iterate again to create another child process. Used for simultaneous execution




```
arif@kali:~$ echo $$
4920
arif@kali:~$ ./a.out 3
PID=9784, PPID=4920
PID=9787, PPID=9784
PID=9786, PPID=9784
PID=9785, PPID=9784
```



Things To Do



If you have problems visit me in counseling hours. . . .