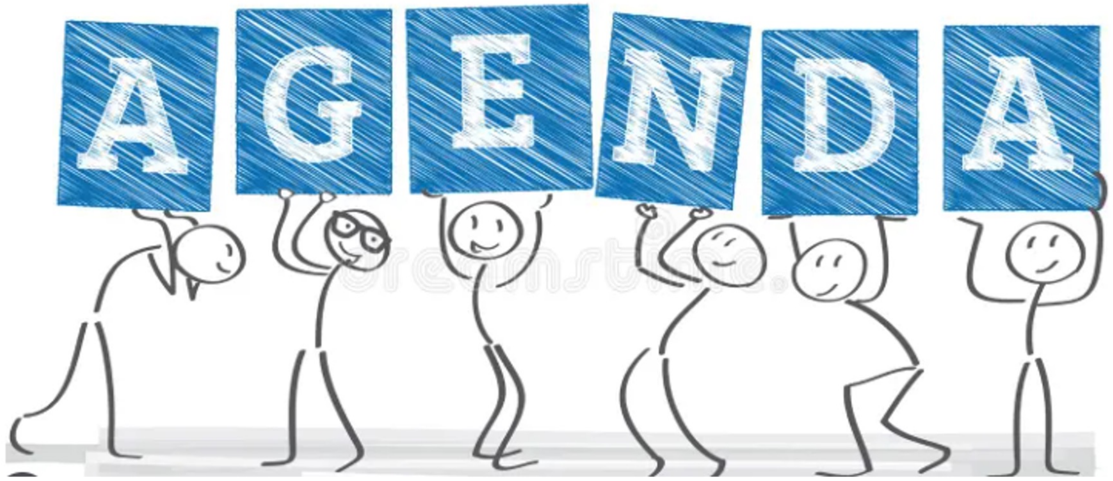
Operating Systems

Lecture 3.1

Process Management (Part-I)

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- Process Management on Shell
- Managing Foreground/Background Tasks
- How a Process starts and terminates?
- Retrieving Process Identifications
- Modifying User IDs
- Process creation using `fork` and `vfork`
- Process trees, chains and fans
- Orphan and Zombie processes

Process Management on Terminal

Process Monitoring Commands

Commands	Description
`ps`	Shows the information about the programs that are currently running on your computer
`top/htop`	Display interactive real-time view of running processes and resource usage
`atop`	Advanced performance monitor that logs long-term resource usage including disk and network
`pstree`	Display the system processes into a tree like structures
`pgrep`	used to search process by name and return their Process ID (PID)
`pidof`	Returns the PID of a named program
`pidstat`	Displays CPU, memory, and I/O usage statistics per process over time
`ulimit`	Sets or displays limits on user-level resources (e.g., max file size, number of open files)
`prlimit`	Sets or gets resource limits of a running process (PID-based)
`uptime`	Shows how long the system has been running, number of users, and load average

The ps vs ls Command

- We know that the **`ls -l`** command presents metadata and permission details associated with files stored on disk. Similarly ,the **`ps -l`** command provides detailed information regarding the attributes and states of processes currently residing in system memory.

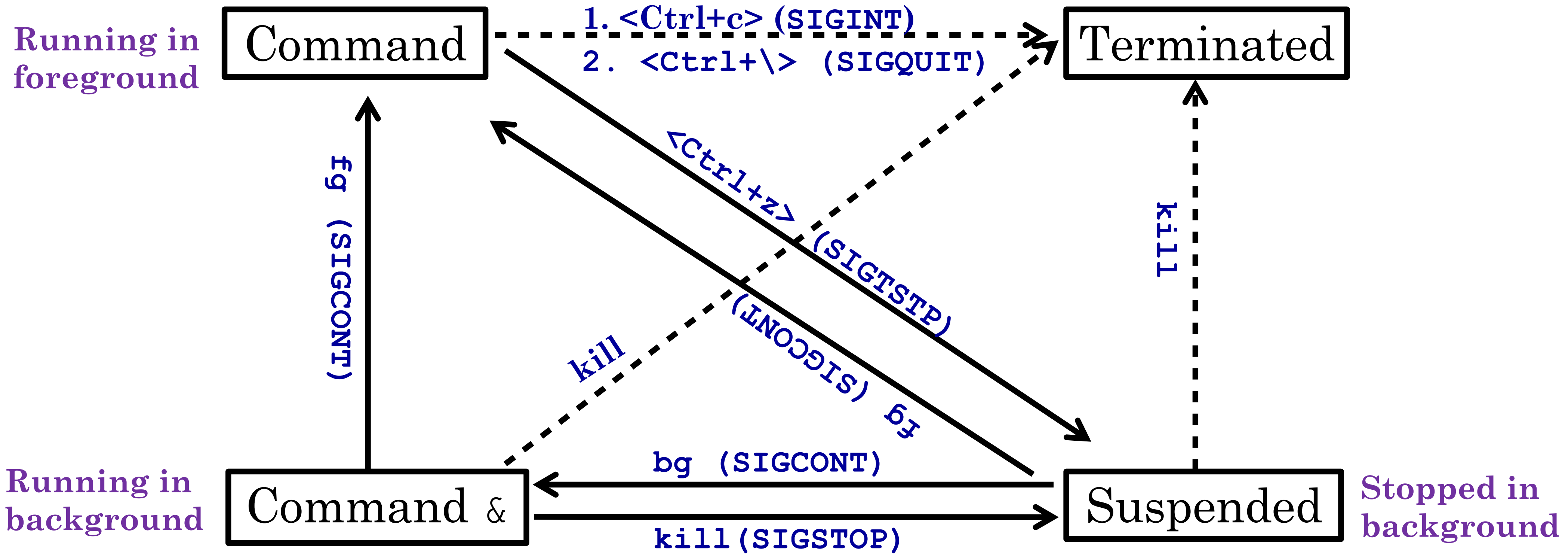
```
F S   UID    PID   PPID  C  PRI  NI  ADDR    SZ   WCHAN    TTY         TIME      CMD
4 S  1000   2245   2243  0   80   0   -     1785   wait   pts/0     00:00:00    bash
4 R  1000   2287   2245  0   80   0   -     1352   -      pts/0     00:00:00     ps
```

- **`F`**: Process flags (e.g., 2=PF_STARTING, 32=PF_SIGNALED, 4=PF_FORKNOEXEC)
- **`S`**: Process state (R=running, S=sleeping, D=uninterruptible sleep, Z=zombie, T=stopped, X=dead)
- **`UID`**: User ID of the process owner
- **`PID`**: Process ID
- **`PPID`**: Parent process ID
- **`C`**: CPU usage in percentage (recent usage)
- **`PRI`**: 20 (base user priority) + nice value + 60
- **`NI`**: Nice value (range: -20 [high priority] to 19 [low priority], default is zero)
- **`ADDR`**: Memory address of the process (often shown as "-" for user processes)
- **`SZ`**: Size in memory pages (virtual memory size)
- **`WCHAN`**: Kernel function in which the process is sleeping (if applicable)
- **`TTY`**: Controlling terminal of the process (e.g., pts/0, tty1)
- **`TIME`**: Total accumulated CPU time
- **`CMD`**: Command that started the process

Foreground vs Background Processes

- In a CLI, processes can exist in one of two states: foreground or background:
    - **Foreground Process:** This is the single process that is actively holding the terminal. It receives all keyboard input and its output is displayed directly on the screen. A program like `vim` or `less` is a classic example of a foreground process. You must wait for a foreground process to finish or be stopped before you can execute another command.
    - **Background Process:** These are processes that run without a direct connection to the terminal. They don't typically require user input and their output is not shown on the screen, allowing the user to continue using the terminal for other commands. Common examples include an audio player or a file search utility like `find`.
- Unlike a Graphical User Interface (GUI) where you can simply click to minimize or switch applications, a CLI requires specific commands and signals to move a process between these two states. For example, pressing Ctrl+Z sends a SIGTSTP signal to a foreground process, moving it to a stopped state in the background. From there, you can use the bg command to resume it in the background or fg to bring it back to the foreground

Job Control States



Instructor: Muhammad Arif Butt, PhD

Managing Background & Foreground Tasks

- The **`jobs`** command is used to track and manage background or suspended tasks within the current shell session. It works alongside **`fg`**, **`bg`**, and **`kill`** to control process states interactively.
- This is especially useful in scripting, multitasking in terminal sessions, or when dealing with long-running foreground commands.
- Every command run in the shell can be:
  - **Foreground**: Takes control of the terminal.
  - **Background**: Runs while the terminal remains usable.
- **Background execution:** Use & at the end of a command → **`sleep 60 &`**
- To view background/suspended jobs → **`jobs`**

Commands	**Description**
`jobs`	Lists jobs with status (Running / Stopped)
`fg %n`	Brings job **n** to the foreground
`bg %n`	Resumes stopped job **n** in the background
`kill %n`	Sends terminate signal to job **n**

```
$ sleep 100 &
[1] 15018
$  jobs
[1]  + running    sleep 100
$ fg %1
[1]  + 15018 running    sleep 100
^Z
[1]  + 15018 suspended  sleep 100
$ bg %1
[1]  + 15018 continued  sleep 100
$ kill %1
[1]  + 15018 terminated  sleep 100
```

Viewing contents of running process

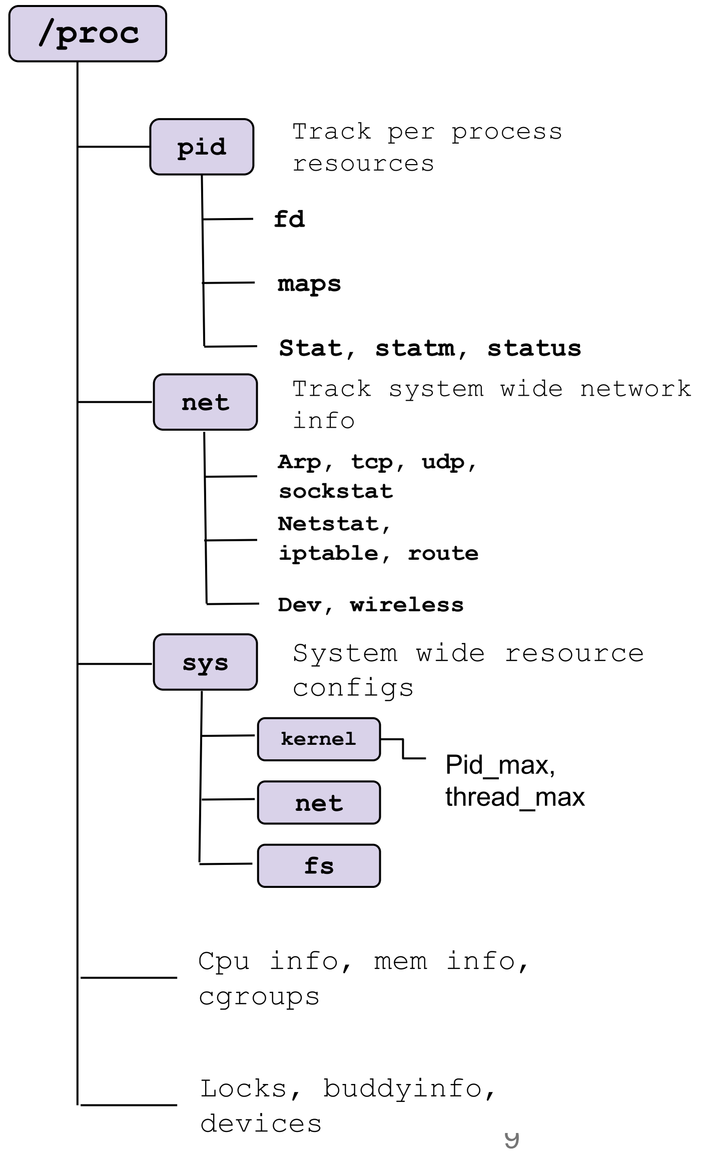
Linux provides a virtual filesystem called `/proc` to access information about running processes. Each running process gets a directory under `/proc`, named after its Process ID (PID), containing files that expose real-time information about the process. View contents of current shell process in `/proc/$$` directory

Common Files in `/proc/<PID>/`

- `/proc/<PID>/cmdline`: Command that was used to start the process (null-separated)
- `/proc/<PID>/environ`: Environment variables for the process (null-separated)
- `/proc/<PID>/limits`: Contain soft and hard resource limits for the process
- `/proc/<PID>/stat`: Low level process status and runtime information (machine readable)
- `/proc/<PID>/status`: Human readable process status and metadata
- `/proc/<PID>/stack`: Kernel stack trace of the process
- `/proc/<PID>/sched`: Detailed scheduling statistics and parameters
- `/proc/<PID>/exe`: A symbolic link to the actual executable file
- `/proc/<PID>/cwd`: A symbolic link to current working directory of the process
- `/proc/<PID>/root`: A symbolic link to process's root directory of the process

Common Directories in `/proc/<PID>/`

- `/proc/<PID>/fd/`: Contains symbolic links to all open file descriptors for the process
- `/proc/<PID>/task/`: Contains one subdirectory per thread in the process each with its own `/proc` style files


/proc
pid
Track per process resources
fd
maps
Stat, statm, status
net
Track system wide network info
Arp, tcp, udp, sockstat
Netstat, iptable, route
Dev, wireless
sys
System wide resource configs
kernel
Pid_max, thread_max
net
fs
Cpu info, mem info, cgroups
Locks, buddyinfo, devices

How a Process Starts and Terminates

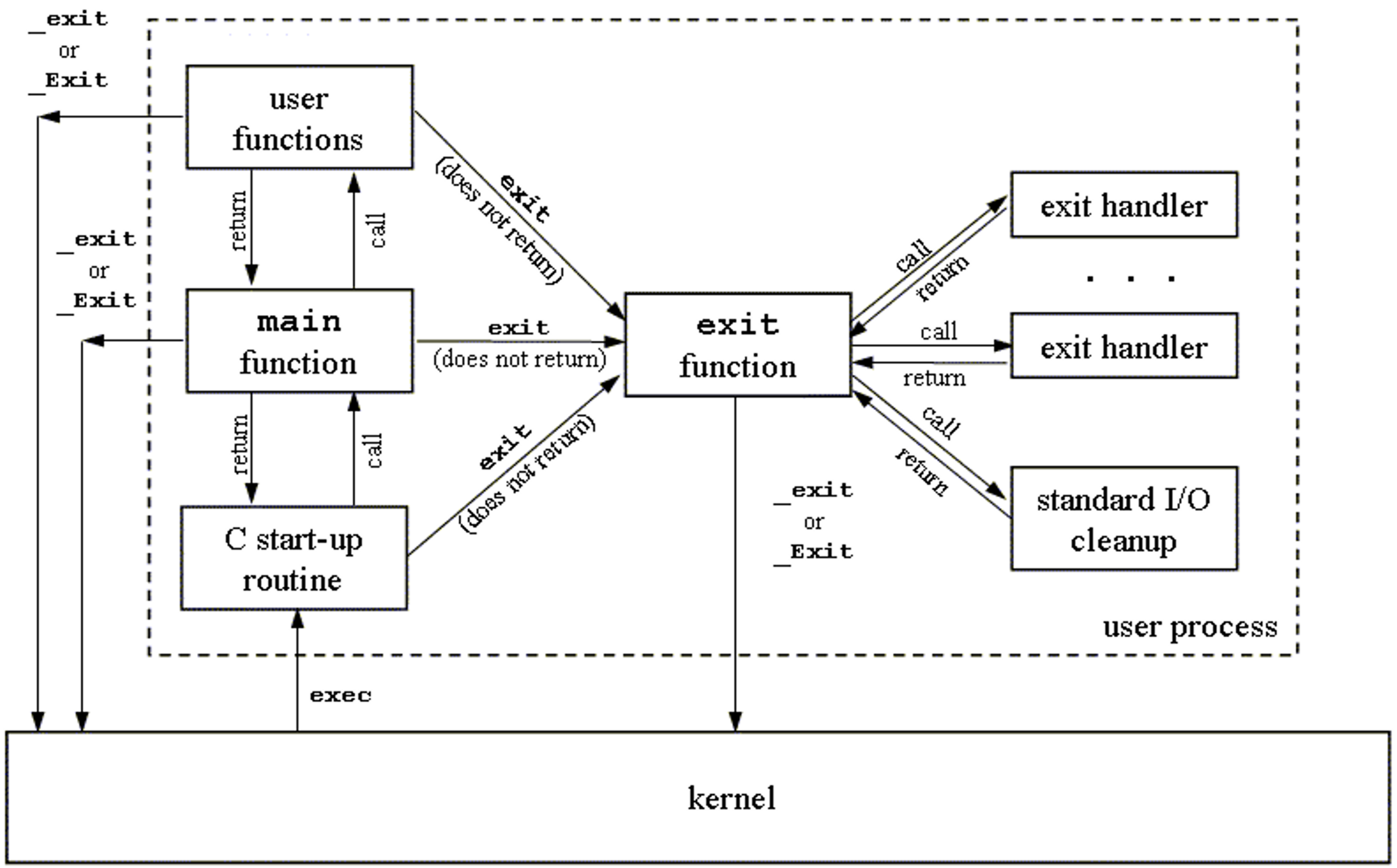
How a Process Starts and Terminates

Normal termination:

- The main function's **return** statement
- Any function calling **exit()** library call

Abnormal termination:

- Any function calling **_exit()** system call
- Calling **abort()** function
- Terminated by a signal



Exit Handlers

```
int atexit(void (*func)(void));

int on_exit(void (*func)(int, void *), void *arg);
```

- Exit handlers in Linux C programming provide a way to register functions that are automatically called when a program terminates normally.
- Exit handlers are called on normal program termination via `exit()` or `return` from main. They are not called on abnormal termination, i.e., when program terminates via `_exit()`, `abort()`, or receiving a signal
- The handlers are called in reverse order of registration (LIFO), allowing for proper dependency management in cleanup
- To register an exit handler we can use either `atexit()` or `on_exit()` function. The on_exit() is GNU extension, so it may not be available on all systems.
- **Limitations of exit handler registered via `atexit()`**
  - An exit handler doesn't know what exit status was passed to exit(); which may be useful. e.g., we may like to perform different actions depending on whether the process is exiting successfully or unsuccessfully
  - We can't specify an argument to exit handler when called; which may be useful to define an exit handler that perform different actions depending on its argument
- The `on_exit()` is more powerful than `atexit()`. It accepts two arguments, a function pointer and a void pointer. The `func` is a function pointer that is passed two arguments (an integer and a void*). The first argument to `func` is the integer value passed to `exit()`, and the second argument is the second argument to `on_exit()`.

Demonstration

Exit Handlers

```
Lec3.1/exit/atexit_ex1.c
Lec3.1/exit/atexit_ex2.c
Lec3.1/exit/onexit_ex.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Process Identification

Program Control Block

A Process Control Block (PCB) is a data structure used by the operating system to manage information about a process. In Linux, PCB is implemented as the `task_struct` structure, which resides in kernel space and encapsulates all critical information required to manage a process. This structure includes the process ID, current state, scheduling information, memory management details, open file descriptors, and CPU register context for context switching. By maintaining this centralized data structure for every process, the kernel efficiently supports scheduling, resource management, inter-process communication, and process lifecycle control.

Process Identification

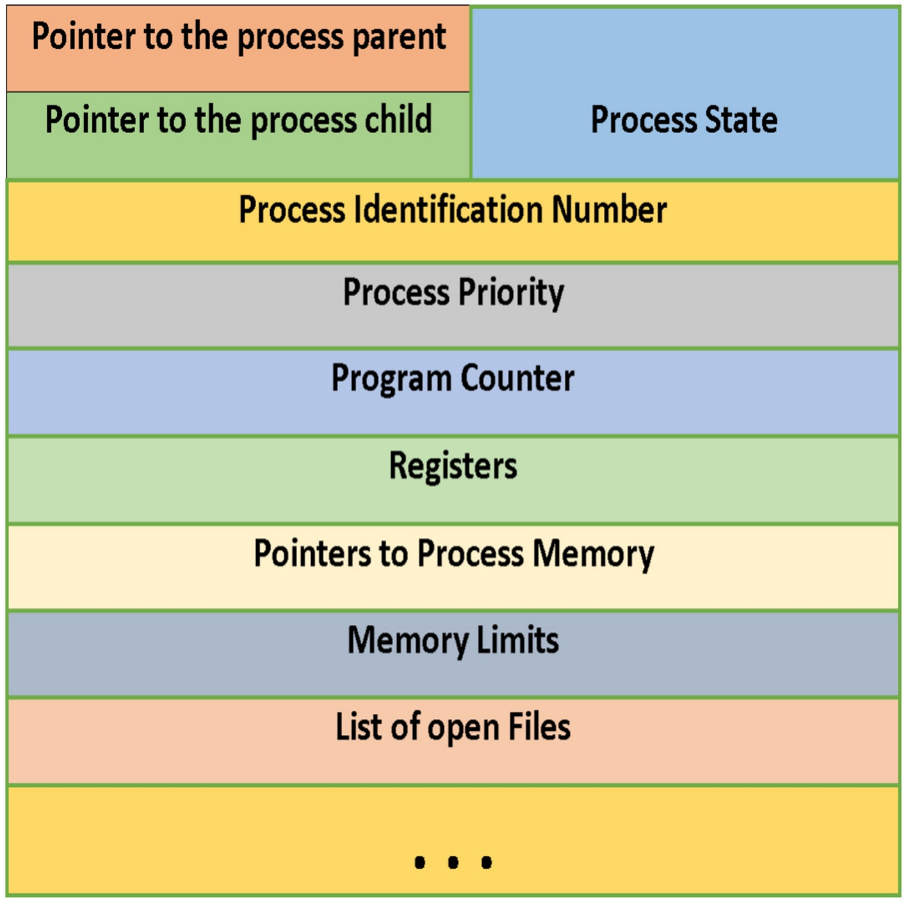
- PID & PPID
- UID & GID
- Saved SUID & SGID
- File System UID & GID

Process state information

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

Process control information

- Scheduling Info
- Privileges Info
- Memory Management Info
- Resource Ownership and Utilization


Pointer to the process parent
Pointer to the process child
Process State
Process Identification Number
Process Priority
Program Counter
Registers
Pointers to Process Memory
Memory Limits
List of open Files
. . .

Process IDs in Linux

- **Basic PID Properties:** Each process is assigned a process ID (PID), a positive integer that uniquely identifies a process on the system. The PID of 1 is reserved for the `init / system`. The PID of 0 is reserved for the `swapper/idle` process that runs when CPU has no other work to do. There is no program file for swapper process in the /proc directory. The PID of 2 is reserved for `kthreadd` (kernel thread daemon), that is mostly responsible for creating other kernel threads. You won't find `/proc/0/` but you can see `/proc/1/` and `/proc/2/`.
- **PID Limits and Recycling:** On 32-bit platforms running Linux kernel, the maximum value for PID is defined in `/proc/sys/kernel/pid_max` file, which is 32768. On 64-bit platforms, it can be adjusted up to 2^{22} (approx. 4 million). When the PID counter reaches the maximum value, it wraps around to **300** (not 1) to avoid reserved low-numbered PIDs. PIDs 1-299 are typically reserved for kernel and system processes.
- **Accessing PID Information:** Current shell PID is available in `$$` and parent PID in `$PPID` environment variable. Moreover, parent PID can be found in the 4th field of `/proc/PID/stat` file, while the 3rd field shows process state (R=Running, S=Sleeping, D=Disk sleep, Z=Zombie, T=Traced/Stopped, X=Dead)

Accessing PID and PPID

```
pid_t getpid();

pid_t getppid();
```

- `getpid()` and `getppid()` returns PID and PPID of the current process respectively and never fails.
- 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. You won't find it's directory in `/proc/` because it doesn't have a user-space representation.
- The `systemd` (or older `init`) is a user process having a PID of `1`. It is invoked by the kernel at the end of the booting process. The `pgrep systemd` command displays multiple PIDs, because system isn't just one process. It is actually a collection of related processes running different parts of the `systemd` system (`system`, `system-journald`, `system-networkd`, `system-resolvd`, `system-logind`, `system-udevd`)
- Page daemon now `kthreadd` is a system process having a PID of `2`. The `kthreadd` is responsible for creating and managing all other kernel threads

Accessing Real User ID & Real Group ID

```
uid_t getuid();

gid_t getgid();
```

- `getuid()` and `getgid()` calls returns the real UID and real GID of the current process and never fails (Defines the ownership of the process).
- The real user ID & real group ID identify the user and group to which the process belongs. Real user and group IDs defines ownership of the process.
- Used for determining default access rights and ownership.
- As part of the login process, a login shell gets it real UID and real GID from the 3rd & 4th field of the `/etc/passwd` file.
- When a new process is created (when a shell `exec` a program), it inherits these IDs from its parent.

Accessing Effective User ID & Effective Group ID

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

- `geteuid()` and `getegid()` calls returns the effective UID and effective GID of the current process and never fails. (Determines a process permission when accessing resources)
- Effective UID and effective GID determine a process's permissions for 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.
  - Change via system calls like `setuid()`, `setgid()`, `seteuid()`, `setegid()`

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

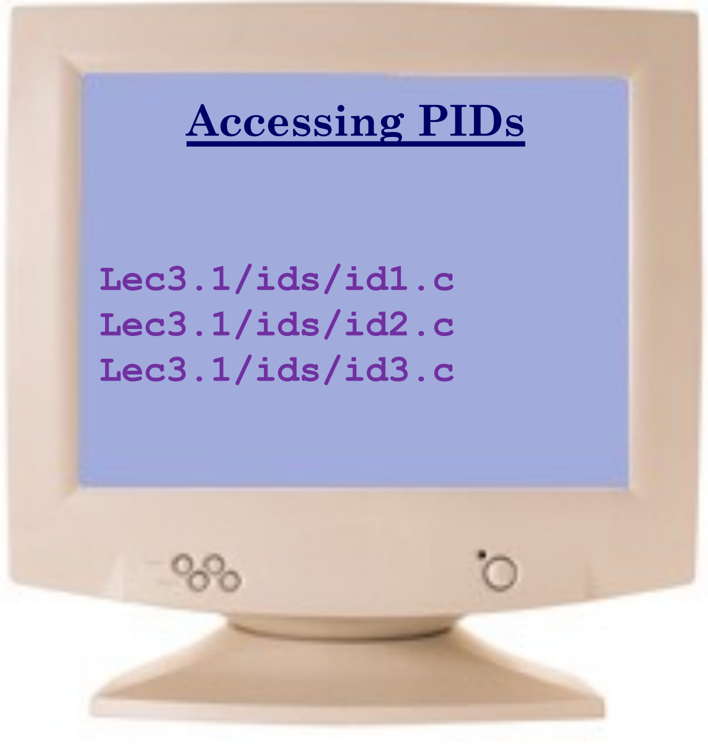
- On the shell, we have already seen and understood the purpose of Saved SUID and Saved SGID bits, which are designed for executable programs that need elevated privileges. They allow programs to run with the privileges of the file owner/group rather than the executing user
- How SUID/SGID Works:
  - SUID bit set: Process's effective UID becomes the UID of the executable file's owner.
  - SGID bit set: Process's effective GID becomes the GID of the executable file's group.
  - No SUID bit: Effective UID remains the same as the real UID.
  - No SGID bit: Effective GID remains the same as the real GID.
- Practical Application:
  - The `/usr/bin/passwd` executable has its SUID bit set and is owned by root.
  - When any user executes `passwd`, the process runs with root's effective UID (EUID=0)
  - This elevated privilege allows the program to modify the `/etc/shadow` file, which has restrictive permissions (readable/writable by root only)
- A process with real, effective, and saved UID = 1000, `execs` a program with its SUID bit set and owned by root, after execution: `real-uid = 1000, effective-uid = 0, saved-uid = 0`

Accessing 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.

Demonstration



Accessing PIDs

```
Lec3.1/ids/id1.c
Lec3.1/ids/id2.c
Lec3.1/ids/id3.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Modifying Process IDs

Modifying User IDs

```
int setuid(uid_t uid);
int seteuid(uid_t euid);
int setreuid(uid_t ruid, uid_t euid);
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
```

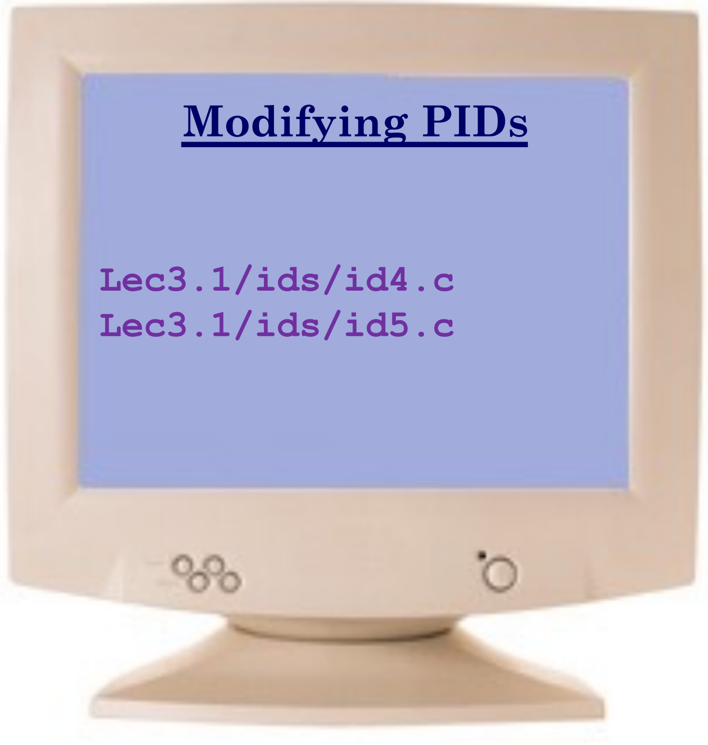
- **`setuid()`:**
  - Unprivileged process can only set the EUID to match its current real UID or saved SUID.
  - Privileged process (EUID = 0) can set real UID, EUID, and saved SUID all to `uid` argument.
  - Once UID is changed from 0, root privileges are permanently lost.
- **`seteuid()`:**
  - Unprivileged process can only set the EUID to match its current real UID or saved SUID.
  - Privileged process can set EUID to any UID, allowing temporary privilege drop and regain (as long as SUID = 0).
- **`setreuid()`:**
  - Unprivileged process can set real UID and/or EUID, but only to values equal to current real UID, EUID, or saved SUID.
  - Privileged process can set both real UID and EUID freely.
  - SUID is updated to new EUID; setting EUID away from 0 may permanently drop privileges.
- **`setresuid()`:**
  - Unprivileged process can change real UID, EUID, and saved SUID, but only to values equal to current real UID, EUID, or saved SUID.
  - Privileged process can set all three independently to any UID.
  - Root privileges are lost permanently if all three are changed from 0.

Modifying Group IDs

```
int setgid(gid_t gid);
int setegid(gid_t egid);
int setregid(gid_t rgid, gid_t egid);
int setresgid(gid_t ruid, gid_t egid, gid_t sgid);
```

- **`setgid()`:**
  - Unprivileged process can only set the EGID to match its current real GID or saved SGID.
  - Privileged process (EUID = 0) can set real GID, EGID, and saved SGID all to `gid` argument.
  - Once GID is changed from 0, group related root privileges are permanently lost.
- **`setegid()`:**
  - Unprivileged process can only set the EGID to match its current real GID or saved SGID.
  - Privileged process can set EGID to any GID, allowing temporary privilege drop and regain (as long as SGID = 0).
- **`setregid()`:**
  - Unprivileged process can set real GID and/or EGID, but only to values equal to current real GID, EGID, or saved SGID.
  - Privileged process can set both real GID and EGID freely.
  - SGID is updated to new EGID; setting EGID away from 0 may result in permanent loss of group privileges.
- **`setresgid()`:**
  - Unprivileged process can change real GID, EGID, and saved SGID, but only to values equal to current real GID, EGID, or saved SGID.
  - Privileged process can set all three independently to any GID.
  - Group privileges are lost permanently if all three are changed from 0.

Demonstration



Modifying PIDs

```
Lec3.1/ids/id4.c
Lec3.1/ids/id5.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Process Creation

Process Creation

- When Linux system starts up, it runs in kernel mode. The first user space process that kernel creates is the `init/systemd` process, which is responsible for starting all other user-space services and background daemons. It launches login prompts, system services, and essential background processes.
- Process creation and termination are the only mechanisms used by the UNIX systems to execute external commands.
- So each time a user runs a command, a new process is created.
- The parent process create children processes, which, in turn create other processes, forming a tree of processes
- Once the OS decides to create a process it proceeds as follows:
  - Assigns a unique PID to the new process
  - Allocate space
  - Initialize the PCB for that process
  - Set appropriate linkages
  - Create or expand other data structures

Process Creation (cont…)

- **Resource sharing**
  - Parent and children share all resources
  - Children share a subset of parent’s resources (UNIX)
  - Parent and child share no resources
- **Execution**
  - Parent and children execute concurrently (UNIX)
  - Parent waits until children terminate
- **Address Space**
  - Child duplicate of parent process (UNIX)
  - Child has a program loaded onto it

Process Creation (cont…)

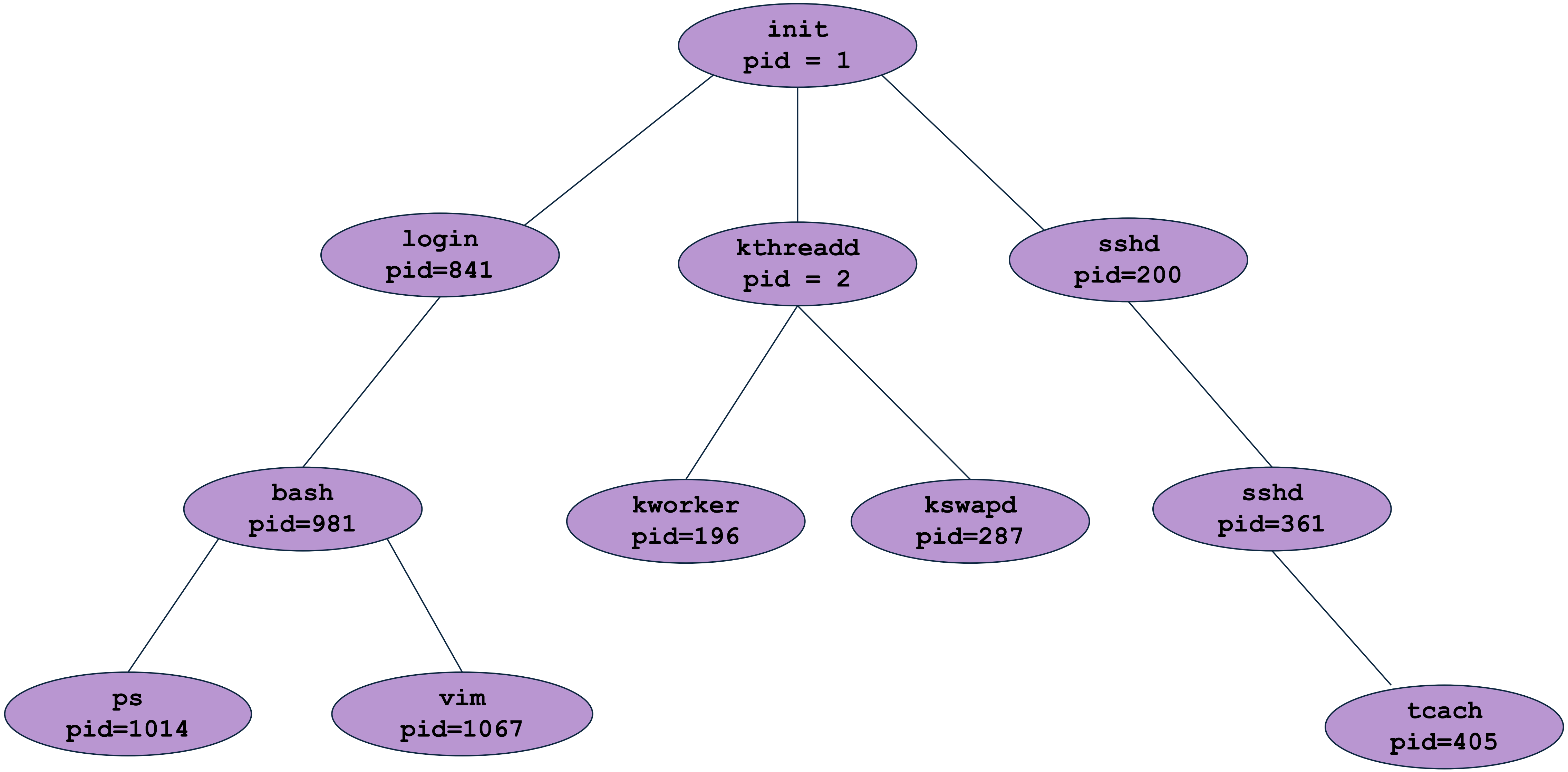
UNIX Examples

- The `fork()` system call is used to create a new process.
- The `exec()` system call is used after a fork to replace the child process memory image with a new executable.
- The `wait()` system call is used by the parent process to wait for the termination of its child.

Reasons for Process Creation

- **Start-up Daemons / Services:** OS creates background processes to perform system functions at start-up or later on behalf of user programs.
- **User Login:** A process is created when a user logs into an interactive system.
- **Process Spawning:** An existing process explicitly requests creation of a new child process
- **Application Launch:** A process is created when a user or system starts an application
- **Job Submission:** A process is created when a job is submitted in batch processing systems

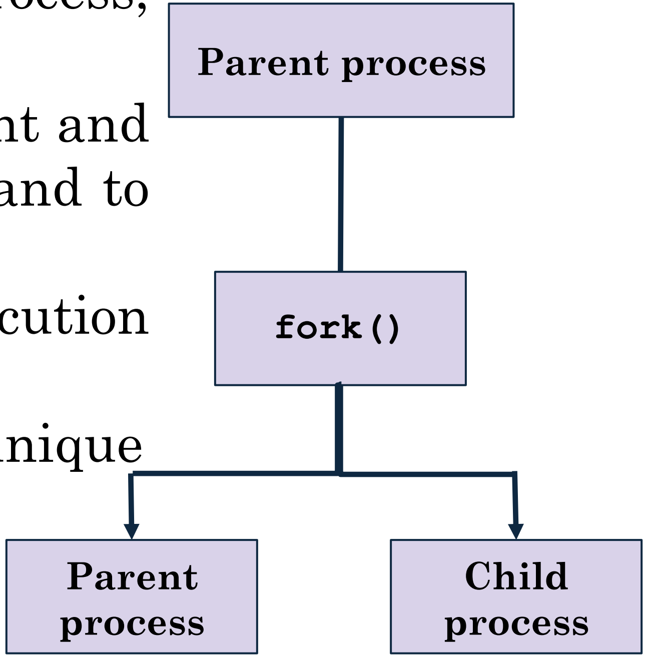
UNIX Process Tree



Process Creation using `fork()` system call

```
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.
- PIDs are allocated sequentially to the new child processes, so effectively unique
- **On success, a child process is created:**
  - The return value to the child process is 0
  - The return value to the parent process is PID of the child



- **Reasons of failure:**
  - Maximum number of processes allowed to execute under one user has reached.
  - Kernel limit on total number of processes has reached.
  - Failed to allocate the necessary kernel structures because memory is tight.
- The child process is a clone of parent and obtains copies of the parent's stack, data, heap, and text.

Uses of `fork()` system call

Command Execution by the Shell:

- When a user executes a command (`a.out` or `cat` or `less`) on a terminal, the shell program calls **`fork()`** to create a nearly exact copy of the shell process.
- The program binary is loaded inside the child process usually using the **`exec`** family of system calls.
- The binary reads its input from the outside world, from command-line arguments and environment variables
- The binary code executes and does what the developer has coded it for.
- While the child process executes, the parent process either waits for the termination of the child process, or may executes concurrently along with the child process. More on this later ☺

Concurrent C Programming:

- When a process wants to duplicate itself, so that parent & child each can execute different sections of code concurrently. For example, consider a network server, which wants to serve multiple clients. The parent process waits for a service request from a client, and when a request arrives, parent calls fork() & let the child handle the request. Parent goes back to listen for the next request.

Illustration of `fork()` system call

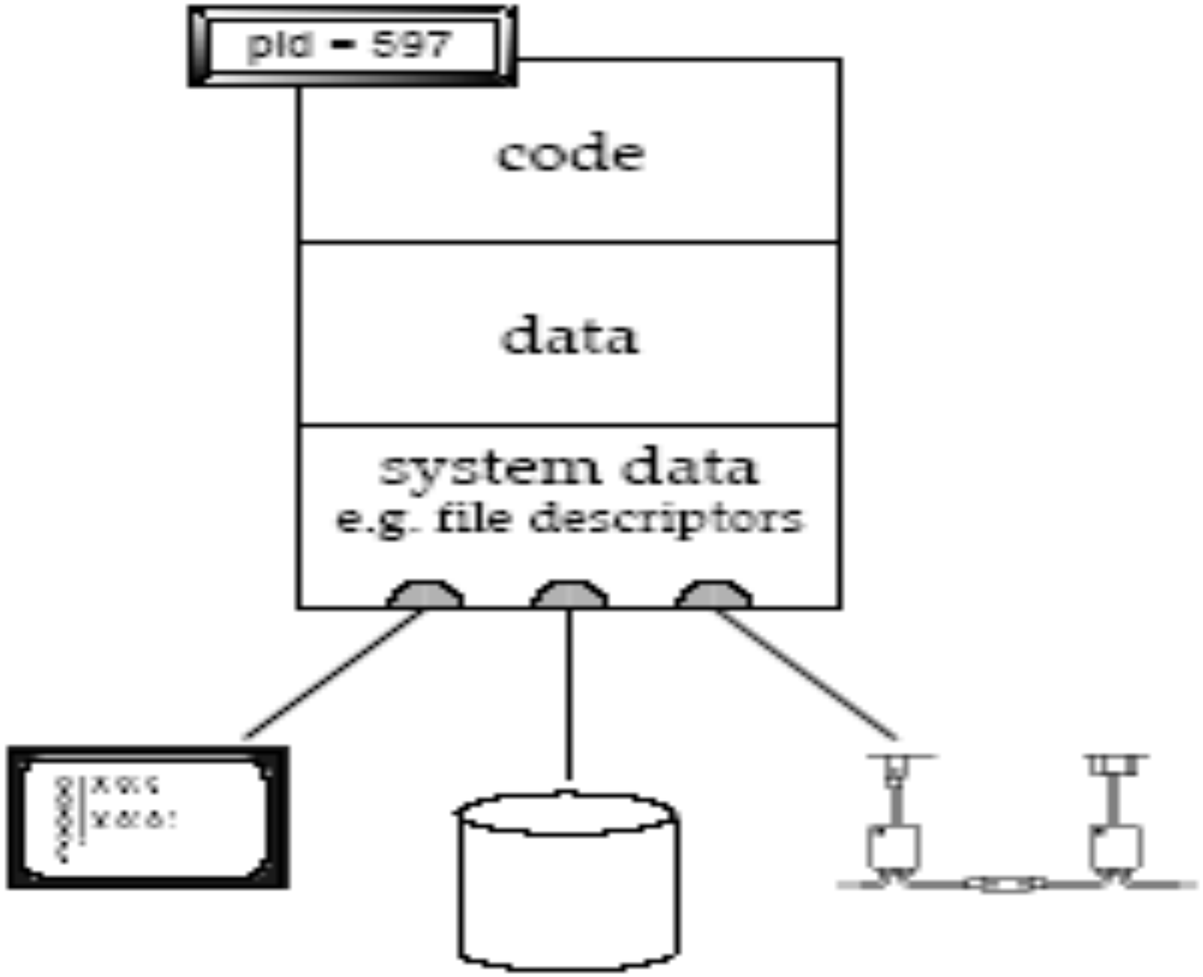


Illustration of `fork()` system call (cont…)

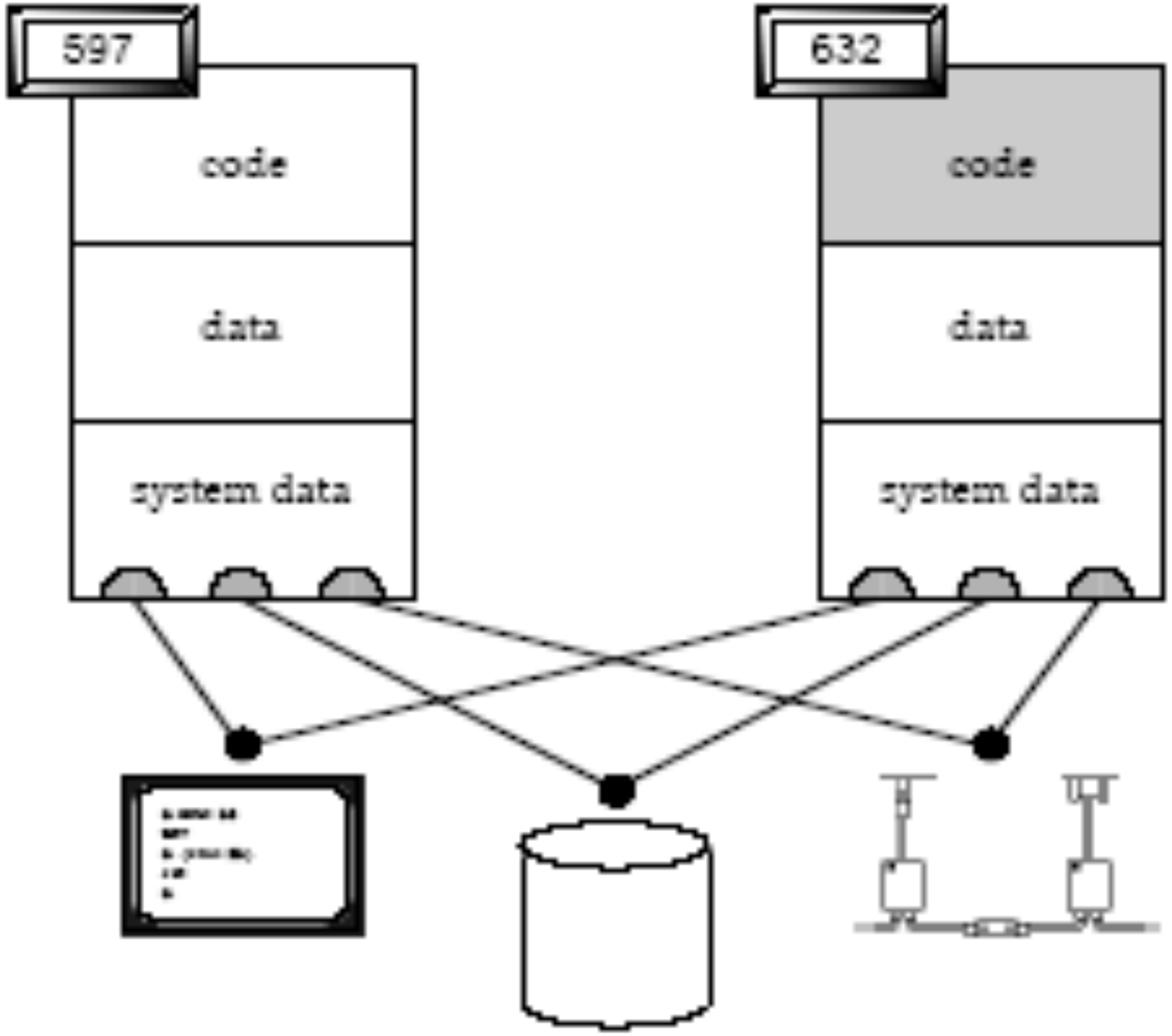


Illustration of `fork()` system call (cont…)

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.

PID: 597

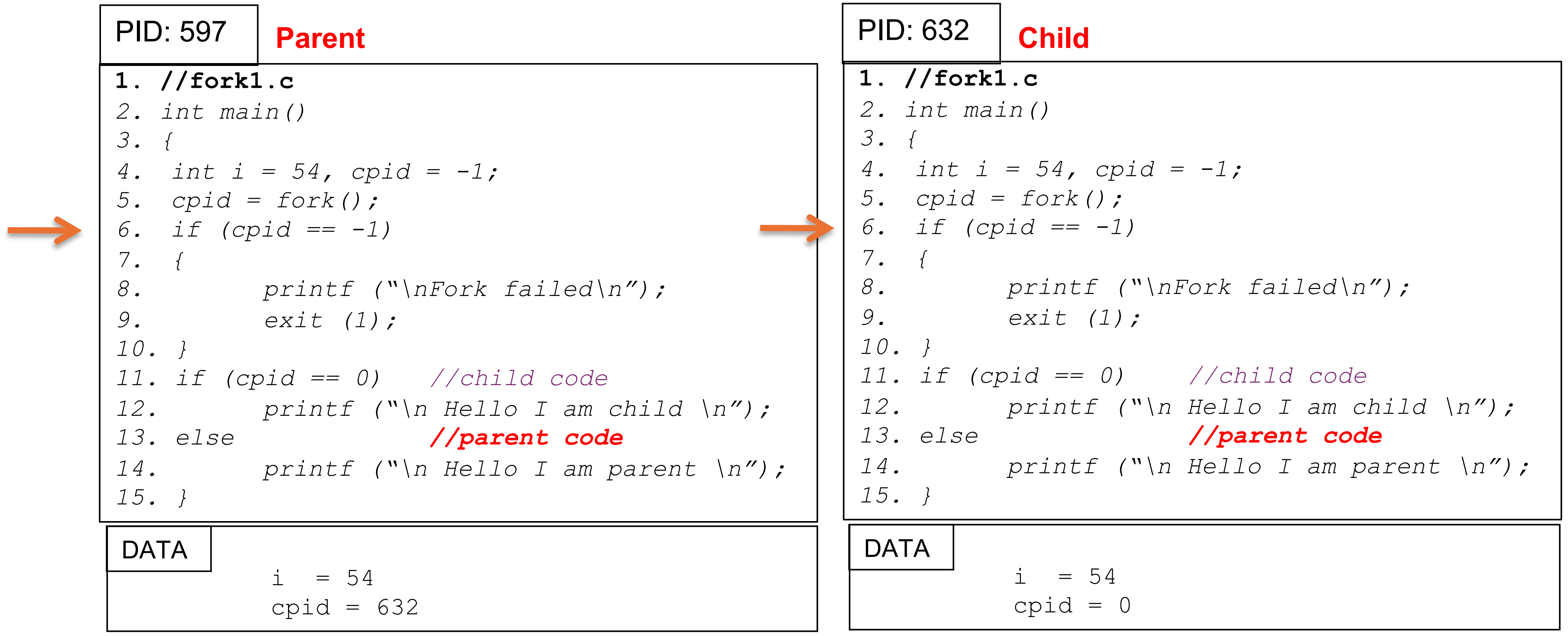
Parent Process forks

```
//fork1.c
int main()
{
int i = 54, cpid = -1;
cpid = fork();
if (cpid == -1)
{
        printf ("\nFork failed\n");
        exit (1);
}
if (cpid == 0)    //child code
        printf ("\n Hello I am child \n");
else              //parent code
        printf ("\n Hello I am parent \n");
}
```

DATA

```
i    = 54
cpid = -1
```

Illustration of `fork()` system call (cont…)



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 fork parent and child are identical, except for the return value of `fork` and of course their `PID`.
- 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`**.

Illustration of `fork()` system call (cont…)

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**

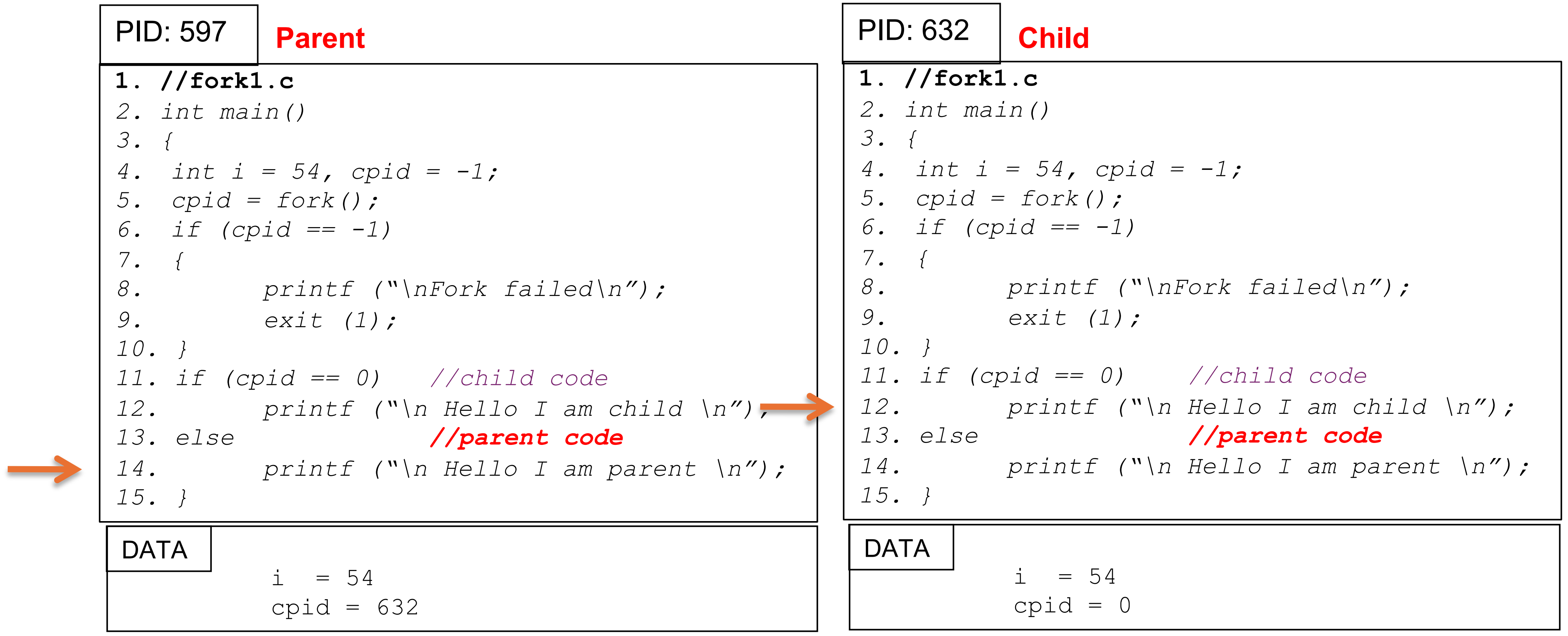
```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.  if (cpid == -1)
7.  {
8.        printf ("\nFork failed\n");
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       printf ("\n Hello I am child \n");
13. else              //parent code
14.       printf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

DATA

```
i    = 54
cpid = 0
```

- Since, the `fork` is successful and `cpid` in both cases do not contain -1, therefore, the control of execution in both processes moves to **`line#11`**.

Illustration of `fork()` system call (cont...)


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
```

- **In parent process,** since `cpid` contains a value other than zero (child PID), therefore, the control of execution is transferred to **`line#14`**.
- **In child process,** since `cpid` contains a value of zero, therefore, the control of execution is transferred to **`line#12`**.

Race Condition after a `fork()`

- After `fork()`, execution order between parent and child is indeterminate and depends on scheduler decisions.
- On multiprocessor systems, both processes may execute simultaneously on different cores.
- Historical kernels sometimes showed slight parent preference due to the parent already being in running state.
- Modern Linux kernels use the Completely Fair Scheduler (CFS) which schedules both processes based on fairness and load balancing without inherent parent/child priority.
- No assumptions should be made about execution order, so developer should use explicit synchronization when ordering is critical.

Attributes Inherited after `fork()`

Process Identity and Permissions:

- Real, effective, and saved user IDs (UIDs) and group IDs (GIDs)
- Process group ID and session ID
- Controlling terminal

File System and I/O:

- Per process file descriptor table (PPFDT)
- Current working directory and root directory
- File mode creation mask (`umask`)

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

Process Environment:

- Environment variables (all variables, not just exported ones)
- Command-line arguments
- Nice value (process priority)

Signal Handling:

- Signal mask (blocked signals)
- Signal dispositions (signal handlers)
- Pending signals are **not** inherited

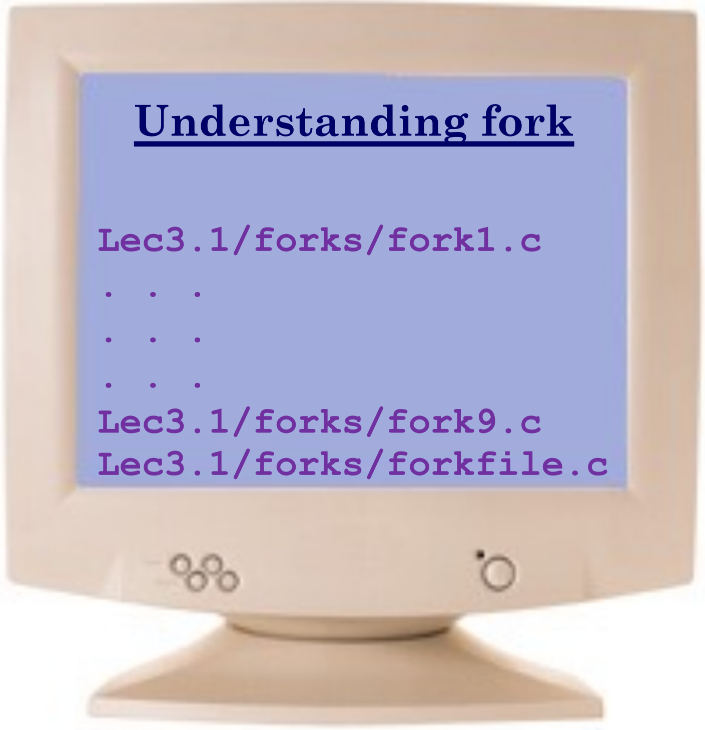
Memory and Resources:

- Both processes initially share the same memory pages (copy-on-write), which are copied only when either process attempts to modify them.
- Attached shared memory segments and memory mappings

Difference between Parent and Child after `fork()`

- Child has different PID and PPID.
- Return value from `fork()`.
- The child's CPU usage counters (user and system time) are reset to zero.
- File locks held by the parent (using `fcntl()`) are not inherited by the child.
- Pending signals in the parent are not inherited by the child.
- Set of pending alarms/timers in the parent are cleared in the child.

Demonstration



Understanding fork

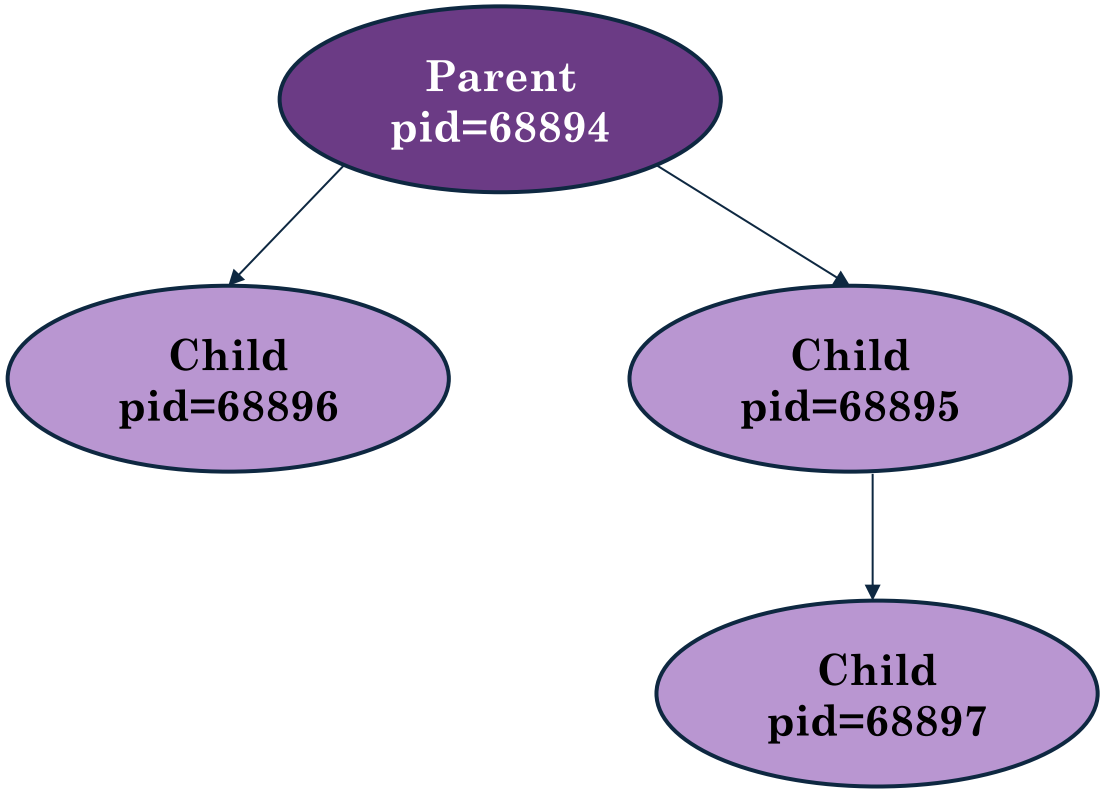
```
Lec3.1/forks/fork1.c
. . .
. . .
. . .
Lec3.1/forks/fork9.c
Lec3.1/forks/forkfile.c
```

GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

Process Tree, Chain, Fan

Process Tree

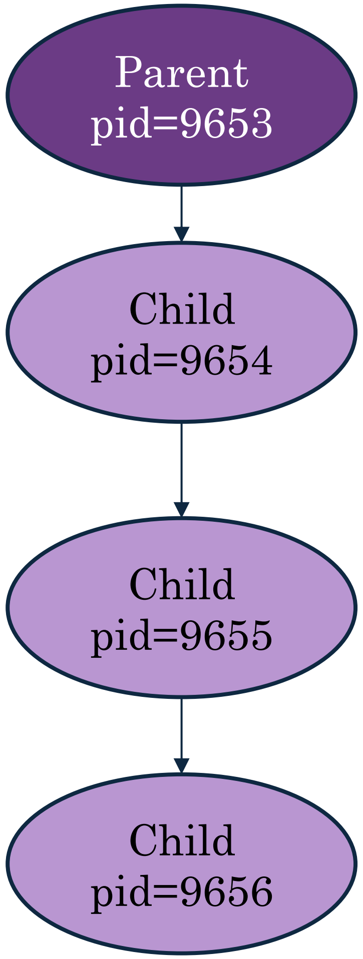
Parent forks and then child call next fork and so on



```
$ echo $$

66591

$ ./processtree 2

PID=68894, PPID=66591
PID=68896, PPID=68894
PID=68895, PPID=68894
PID=68897, PPID=68895
```

Process Chain

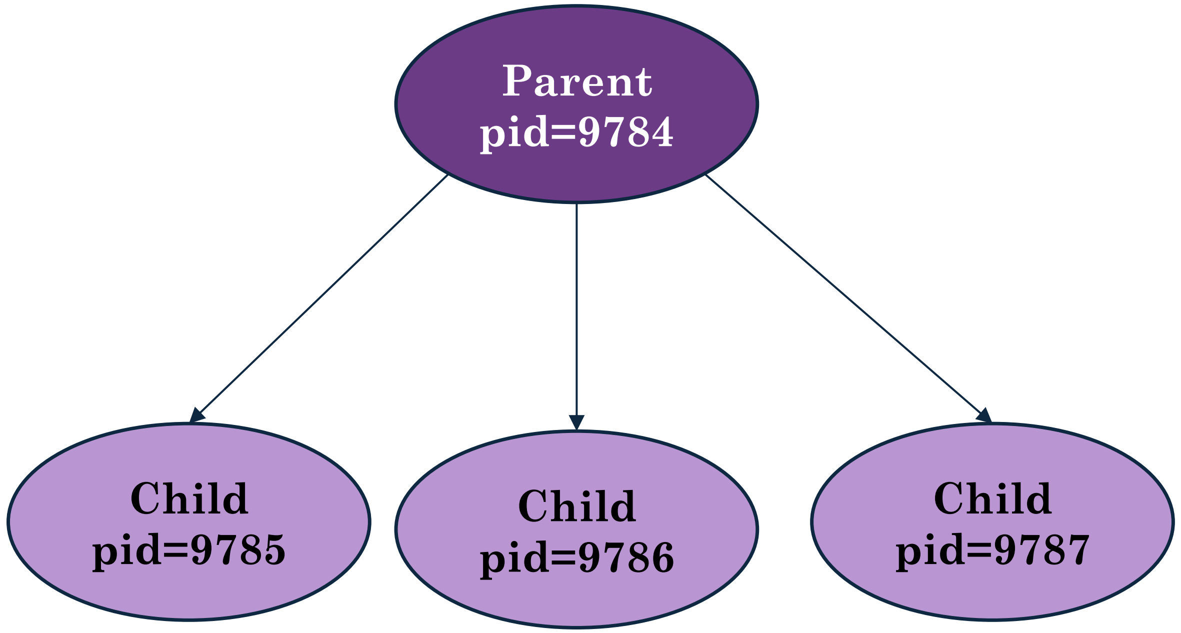
Parent forks once and then child call next fork and so on



```
$ echo $$

4920

$ ./processchain 3

PID=9653, PPID=4920
PID=9654, PPID=9653
PID=9655, PPID=9654
PID=9656, PPID=9655
```

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



```
$ echo $$

4920

$ ./processfan 3

PID=9784, PPID=4920
PID=9785, PPID=9784
PID=9786, PPID=9784
PID=9787, PPID=9784
```

Demonstration

Process tree, chain, fan

```
Lec3.1/forks/processtree.c
Lec3.1/forks/processchain.c
Lec3.1/forks/processfan.c
```

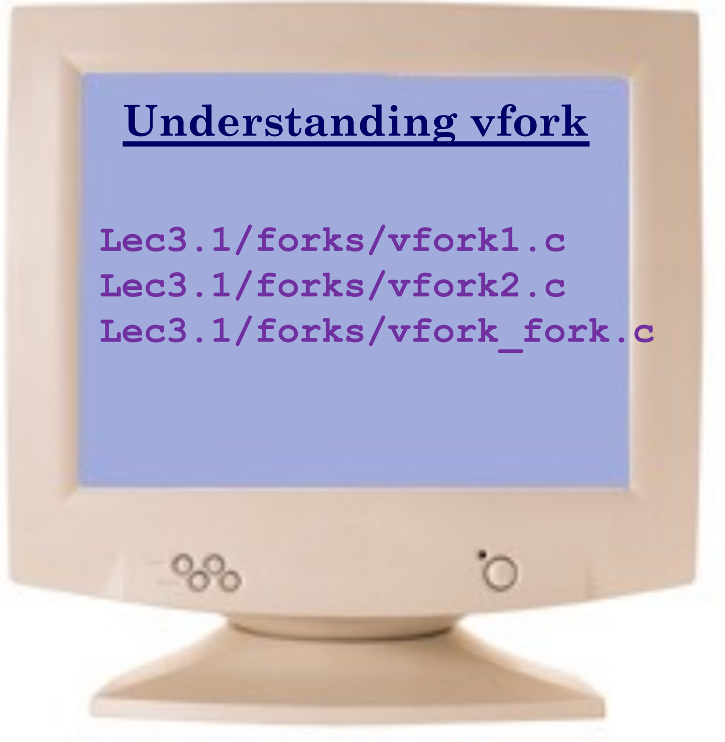
GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

Process Creation using `vfork()`

```
pid_t vfork();
```

- In bad old days a fork() would require making a complete copy of the parent data space. This was an overhead because, since immediately after a fork the child calls `exec()` most of the times. So for greater efficiency BSD introduced `vfork()` system call.
- `vfork()` is intended to create a new process when the purpose of the new process is to `exec` a new program, and it do so without fully copying the parent address space into the child.
- Child process suspends execution of parent process until child process completes its execution as both processes share the same address space.
- Features that make `vfork()` more efficient than `fork()` are:
  - No duplication of virtual memory pages is done for child process. Child shares the parent's address space until it either performs `exec()` or call `exit()`.
  - Execution of parent process is suspended until the child has performed an `exec()` or an `exit()`.

Demonstration



Understanding vfork

```
Lec3.1/forks/vfork1.c
Lec3.1/forks/vfork2.c
Lec3.1/forks/vfork_fork.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Orphan and Zombie processes

Orphan Process

- If a parent has terminated before reaping its child, and the child process is still running, then that child is called orphan.
- In UNIX, the orphan process is adopted by `init` or `systemd`, which do the reaping to avoid resource leakage.

```
int main(){
   int cpid = fork();
   if (cpid == 0){
        printf("Running child:PID=%ld PPID=%ld\n",(long)getpid(),(long)getppid());
       while(1);
   }else{
        printf("Terminating parent: PID=%ld PPID=%ld\n",(long)getpid(),(long)getppid());
        exit(0);
    }
   return 0;
}
```

Zombie Process

- Process that has terminated but their parent have not collected their exit status and has not reaped them are called zombies or defunct. So a parent must reap its children.
- When a process terminates but still is holding system resources like PCB and various tables maintained by OS. It is half-alive & half-dead because it is holding resources like memory but it is never scheduled on the CPU.
- Zombies can't be killed by a signal, not even with the silver bullet (SIGKILL). The only way to remove them from the system is to kill their parent, at which time they become orphan and adopted by `systemd`.

```
int main(){
   int cpid = fork();
   if (cpid == 0){
        printf("Terminating child with PID = %ld\n", (long)getpid());
        exit (0);
   }else{
        printf("Running parent, PID=%ld\n",(long)getpid());
        while(1);
   }
   return 0;
}
```

Demonstration

Orphan & Zombie Process

```
Lec3.1/forks/orphan.c
Lec3.1/forks/zombie.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Watch SP video on Process Management-I:
  https://youtu.be/R_01xGLp0ZQ?si=NQfovVEPiM0t1PMh

- Watch SP video on Process Management-II:
  https://youtu.be/91qzstPN1p8?si=4VnUXtlGw9hJadyV

Coming to office hours does NOT mean that you are academically weak!