Systems Advanced II
Linux

GNU/Linux kernel

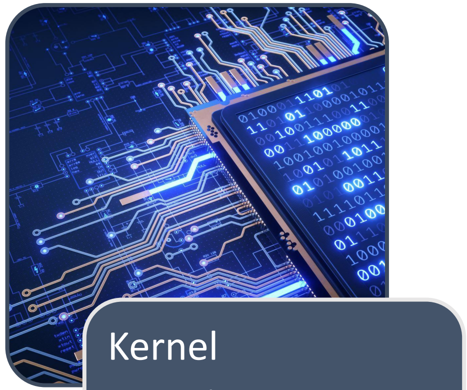
DE HOGESCHOOL
MET HET NETWERK

Elfde-Liniestraat 24, 3500 Hasselt, www.pxl.be

Doelstellingen

- De student:
  - De student kan Netwerk-services installeren, configureren en onderhouden.
  - De student kan microservices-infrastructuur opzetten en beheren.
  - De student kan een (eigen) cloud systeem opzetten a.d.h.v. opgelegde voorwaarden.
  - De student kan een systeem beveiligen.

Systems Advanced II - Linux: leerinhouden

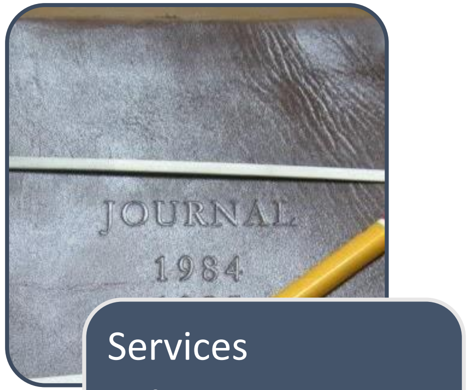


Kernel

- virtual memory
- user space and kernel space
- processes and threads
- sockets & pipes
- systemd
- Docker

Networking

- routing
- firewall
- eBPF
- certificates
- ssh
- latency
- bandwidth



Services

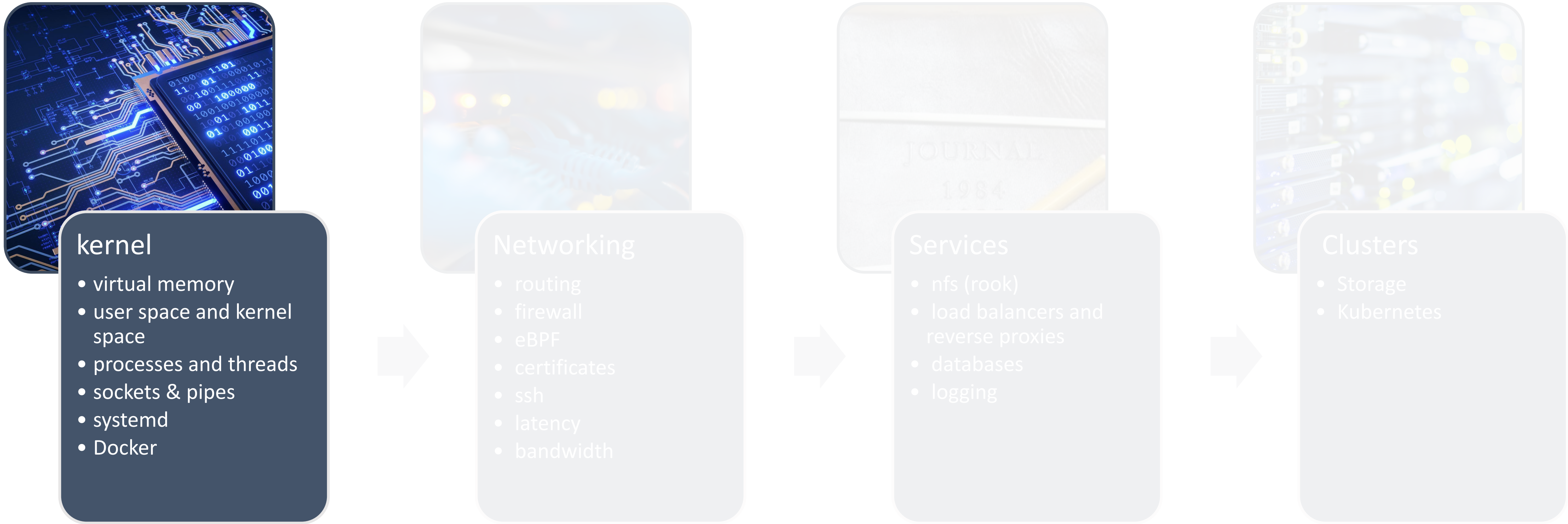
- nfs
- load balancers and reverse proxies
- databases
- logging

Clusters

- Storage
- Kubernetes

Licensing & open-source software

Systems Advanced II - Linux



kernel

- virtual memory
- user space and kernel space
- processes and threads
- sockets & pipes
- systemd
- Docker

Networking

- routing
- firewall
- eBPF
- certificates
- ssh
- latency
- bandwidth

Services

- nfs (rook)
- load balancers and reverse proxies
- databases
- logging

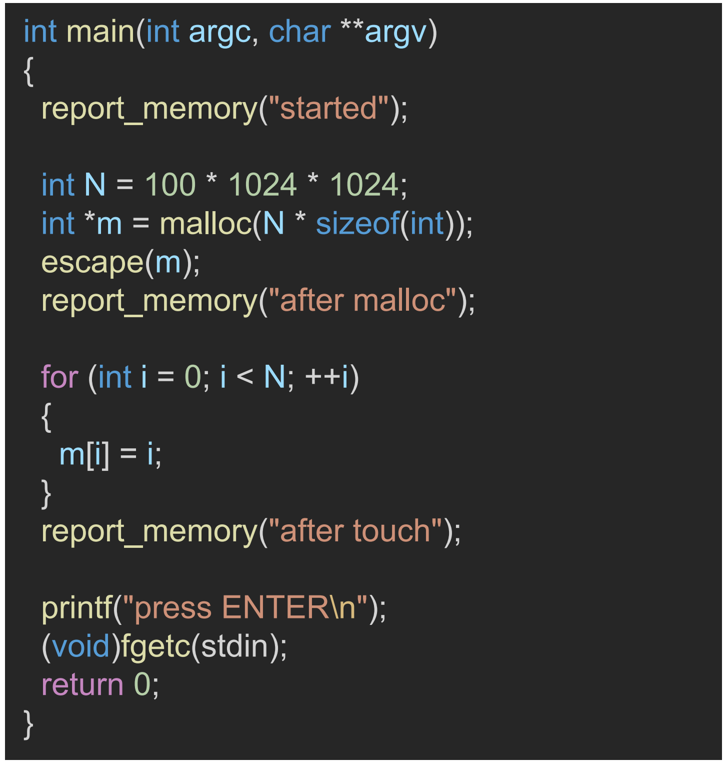
Clusters

- Storage
- Kubernetes

Linux virtual memory

- When a Linux program allocates memory, this memory initially doesn't really exist - it's just an entry in a table of the OS. It is virtual.
- Only when the program actually accesses the memory is the RAM for it found and used.
- "memory usage" of a process can mean two things
  - how much virtual memory it uses overall
  - how much actual or "resident" memory it uses, limited to the system's resident RAM capacity (+ swapping to disk).

- Experiment
  - https://github.com/eliben/code-for-blog/blob/master/2018/threadoverhead/malloc-memusage.c



```c
int main(int argc, char **argv)
{
  report_memory("started");

  int N = 100 * 1024 * 1024;
  int *m = malloc(N * sizeof(int));
  escape(m);
  report_memory("after malloc");

  for (int i = 0; i < N; ++i)
  {
    m[i] = i;
  }
  report_memory("after touch");

  printf("press ENTER\n");
  (void)fgetc(stdin);
  return 0;
}
```

malloc-memusage.c

This program starts by allocating 400 MiB of memory (assuming an int size of 4) with malloc, and later "touches" this memory by writing a number into every element of the allocated array. It reports its own memory usage at every step.

Linux Kernel

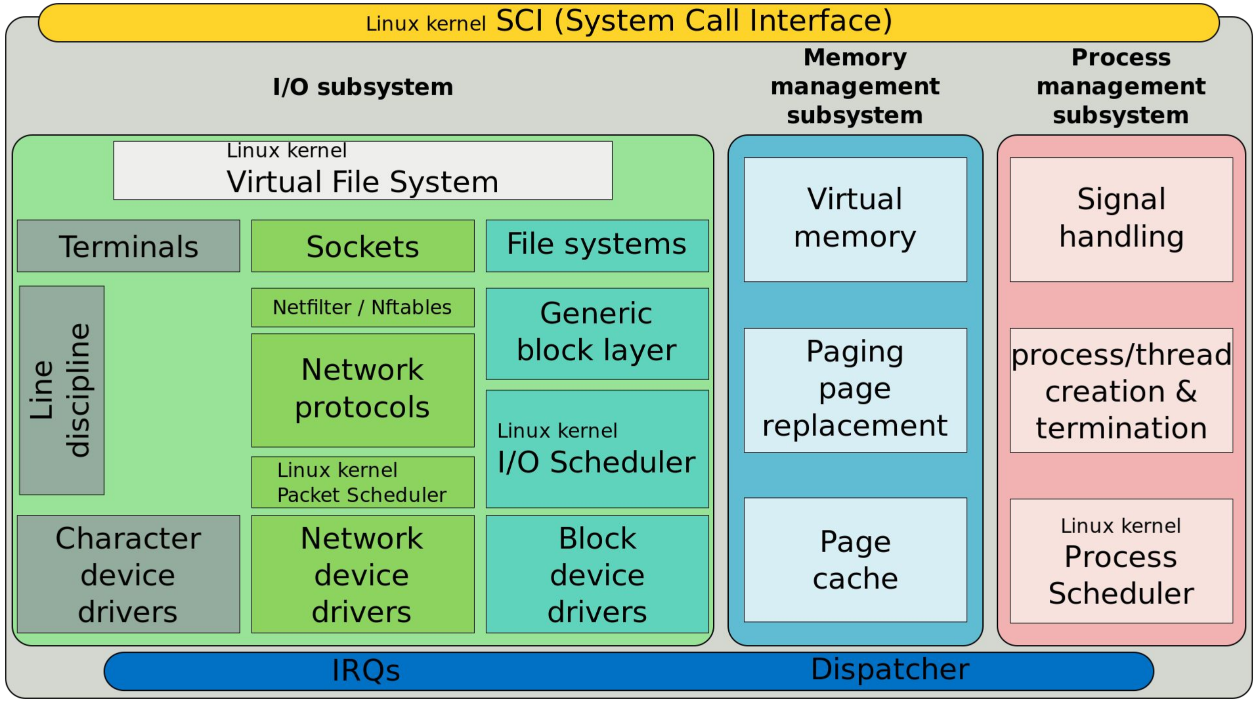
- Core component of the Linux operating system.
- Responsible for managing system resources, providing low-level services to other parts of the operating system, and controlling hardware devices.

Linux Kernel: Monolithic and Modular

- Typically described as monolithic
  - All system-level services are contained within a single executable file
  - Benefits of monolithic design: improved performance and simplified system management
- The kernel also includes a modular design
  - Certain features can be compiled as loadable kernel modules
  - Modules can be dynamically loaded or unloaded at runtime
  - Provides greater flexibility and allows for customization
  - Example: adding support for a particular hardware device or file system
    - Developers can experiment with new features without committing to the main kernel source code
    - If a new feature proves useful and stable, it can be integrated into the main kernel codebase
  - Modular design allows for greater customization and flexibility.

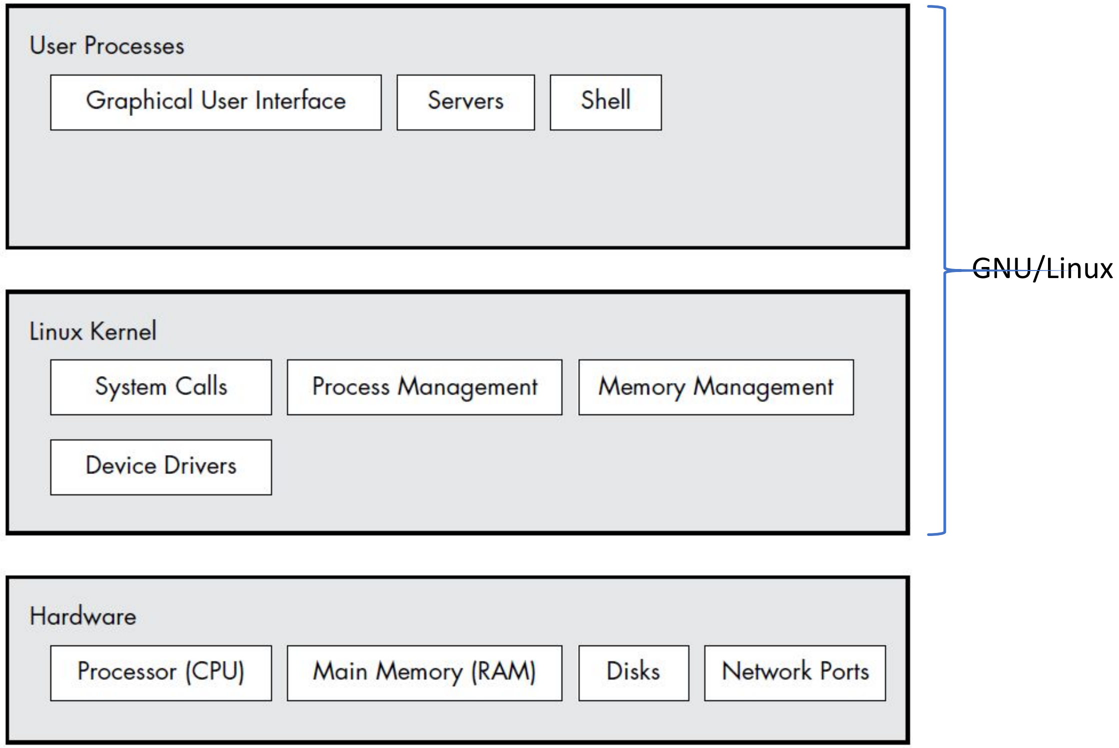
Features and Services of the Linux Kernel

- **Process management**: The kernel manages all running processes on the system, allocating resources and scheduling CPU time.
- **Memory management**: The kernel is responsible for managing the allocation and deallocation of system memory, as well as implementing virtual memory.
- **File system management**: The kernel provides the file system interface for managing multiple file systems.
- **Device management**: The kernel controls access to hardware devices, such as disk drives, network adapters, and input/output devices.
- **Network management**: The kernel provides the networking stack.



Linux Kernel Organization

- software residing in memory that tells the CPU where to look for its next task.
- Acting as a mediator, the kernel manages the hardware (especially main memory) and is the primary interface between the hardware and any running program.
- (User) processes, managed by the kernel, make up user space.



User Processes

Graphical User Interface

Servers

Shell

Linux Kernel

System Calls

Process Management

Memory Management

Device Drivers

Hardware

Processor (CPU)

Main Memory (RAM)

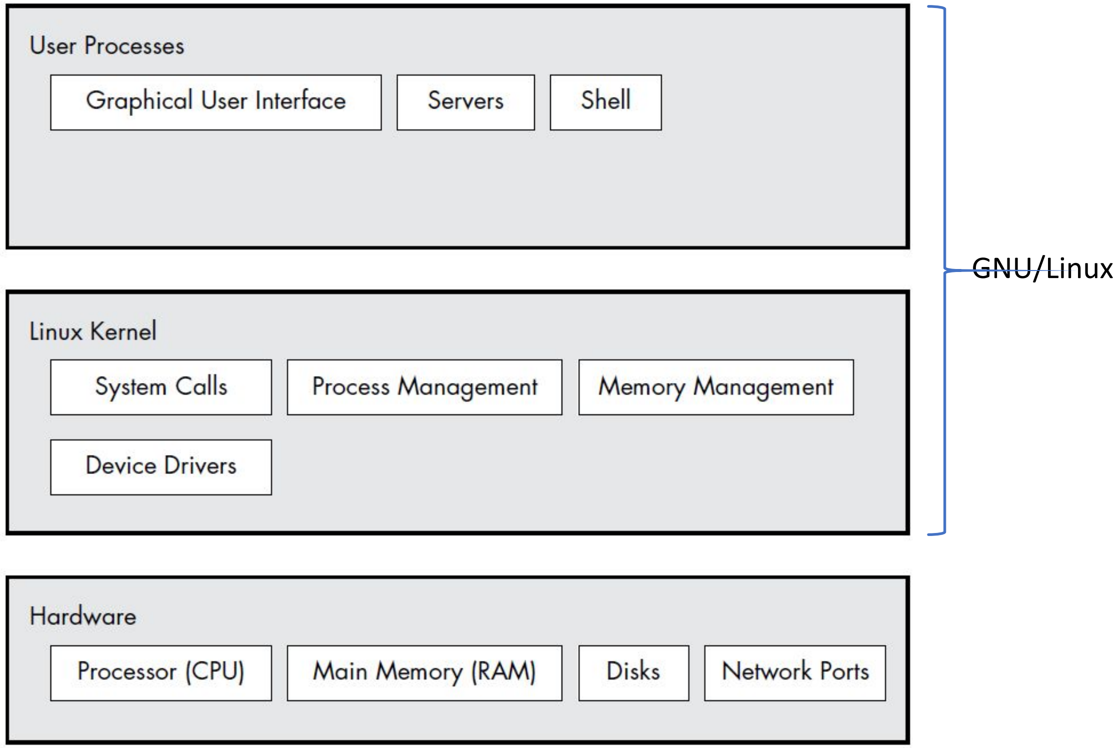
Disks

Network Ports

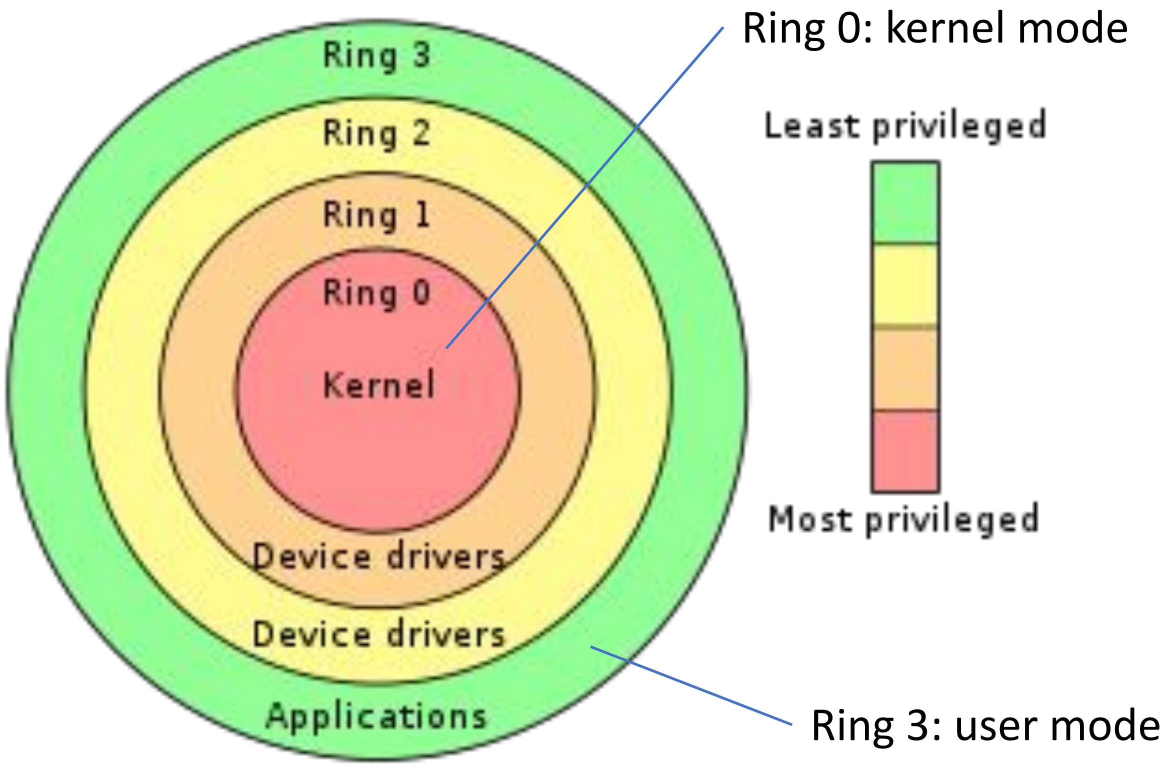
GNU/Linux

User Space and User Processes

- The kernel runs in **kernel mode**, and the user processes run in **user mode**.
- **User space**
  - User mode **restricts access** to a small subset of memory and safe CPU operations.
  - User space refers to the parts of main **memory** that the user processes can access. If a process makes a mistake and crashes, the consequences are limited and can be cleaned up by the kernel.
- **Kernel space**
  - Code running in kernel mode has **unrestricted access** to the processor and main memory.
  - This is a powerful but dangerous **privilege** that allows the kernel to easily corrupt and crash the entire system.
  - The **memory** area that only the kernel can access is called kernel space.



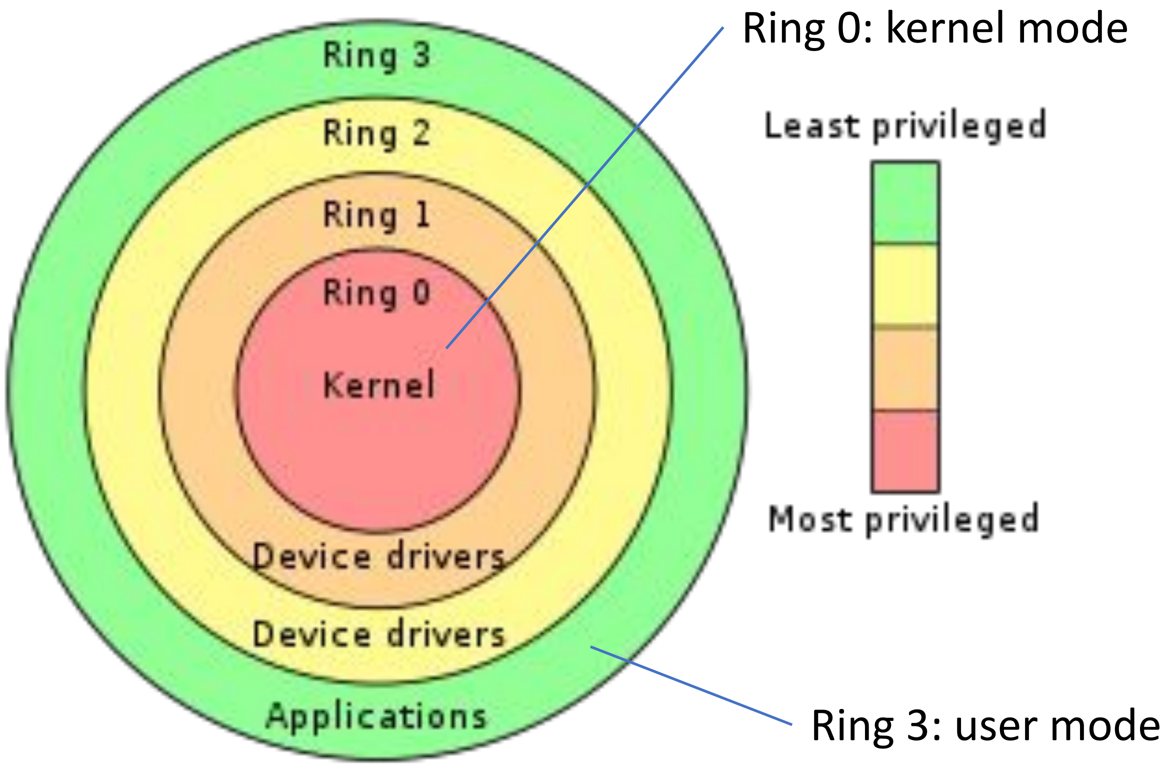
Linux kernel on CPU architectures



CPU Privilege rings for the Intel x86

- Many modern CPU architectures (including the popular ARM and Intel x86 architectures) include some form of ring protection
- The kernel space (Ring 0) has the highest authority and can directly access all resources
- User space (Ring 3) can only access restricted resources and cannot directly access hardware devices such as memory. It must be trapped in the kernel through system calls to access these privileged resources.

Linux kernel on CPU architectures

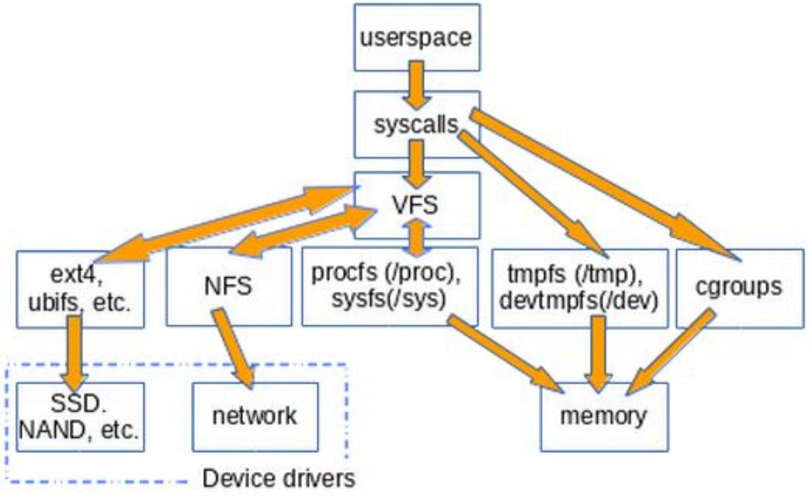


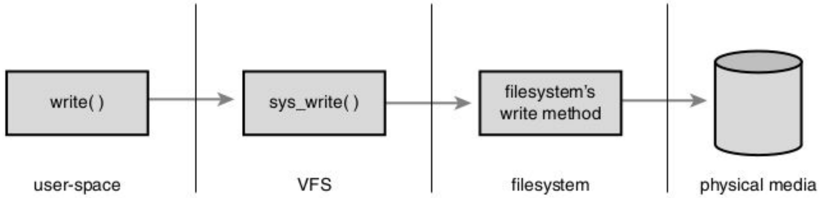
CPU Privilege rings for the Intel x86

- The privilege rings provide a way for the CPU to switch between different levels of privilege during system operation.
- When a user-level process makes a system call or requests a privileged operation, such as accessing a protected system resource or modifying system settings, the CPU switches to ring 0 to perform the requested operation. Once the operation is complete, the CPU switches back to ring 3 to resume normal user-level operation.

Linux Virtual File System

- VFS provides unified view of file system to apps & kernel.
- Abstracts underlying file system details for consistent interface.
- File systems mounted at specific mount points in file system hierarchy.
- VFS caches recently accessed files/directories in memory for performance improvement.
- Supports various file system types: local, network, special (e.g., procfs, sysfs).
    - **procfs**: virtual view of running system, allows access/modification of system info & config parameters.
    - **sysfs**: virtual view of system's hardware devices & drivers, allows view/modification of device attributes & settings.
- Provides flexible & extensible framework for managing files/directories in various file system types & locations.

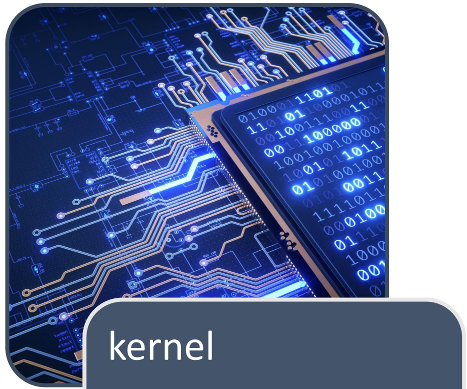




ret = write (fd, buf, len);

The write() call attempts to transfers the data in buf into the location indicated by fd (location of receiving file system) by invoking sys_write() in the VFS which then triggers the appropriate file system with the help of fd (file descriptor).

Systems Advanced II - Linux

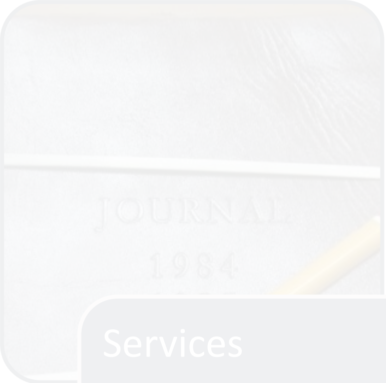


kernel

- virtual memory
- user space and kernel space
- processes and threads
- sockets & pipes
- systemd
- Docker

Networking

- routing
- firewall
- eBPF
- certificates
- ssh
- latency
- bandwidth



Services

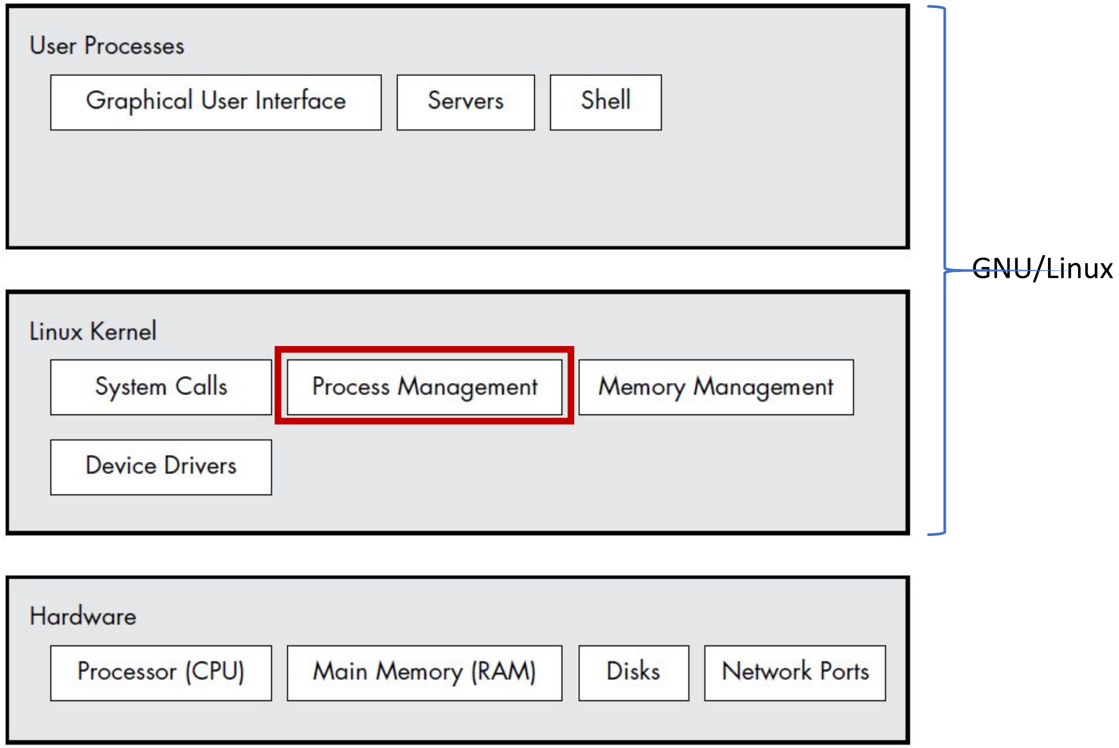
- nfs (rook)
- load balancers and reverse proxies
- databases
- logging

Clusters

- Storage
- Kubernetes

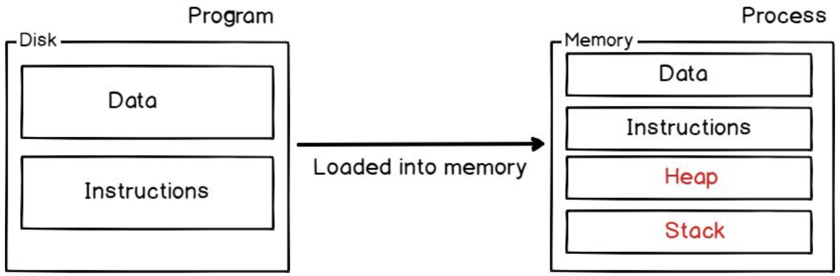
Kernel: Process Management

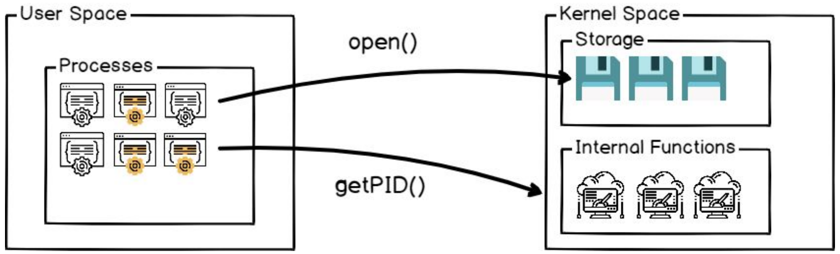
- Starting, pausing, resuming, scheduling, and terminating of processes.
- Each process uses the CPU for a small fraction of a second, then pauses; then another process uses the CPU for another small fraction of a second; then another process takes a turn, and so on.
- **Context switch**: The act of one process giving up control of the CPU to another process.
- **Time slice**: each piece of time gives a process enough time for significant computation.
- The Linux kernel also provides **system calls** for managing processes, such as creating new processes, changing process priority, and terminating processes.



Linux processes

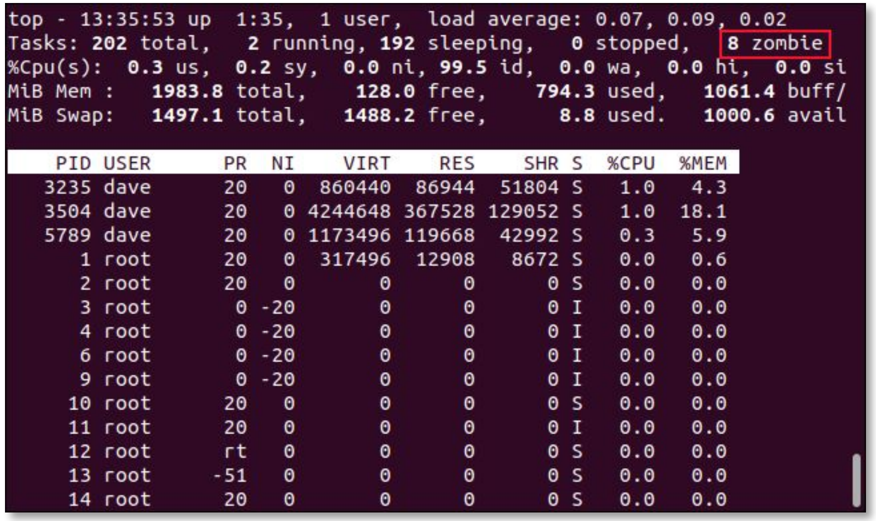
- A process is an instance of a program that is currently executing.
- Each process has its own memory space, program code, and execution context, including the program counter, stack, and other registers
- Processes can communicate with each other using various interprocess communication (IPC) mechanisms, such as pipes, sockets, and shared memory segments
- Processes are created using the fork system call, which creates a new process by duplicating the current process
  - The new process is called the child process, and it has its own unique process ID (PID)
  - The fork system call creates a copy of the parent process, including its memory and state, and then sets the PID of the child process to a new value
  - The child process can then execute a new program or perform other tasks independently of the parent process
- Processes can also be created using other system calls, such as exec, which replaces the current process with a new program, or clone, which creates a new process with shared memory and other resources



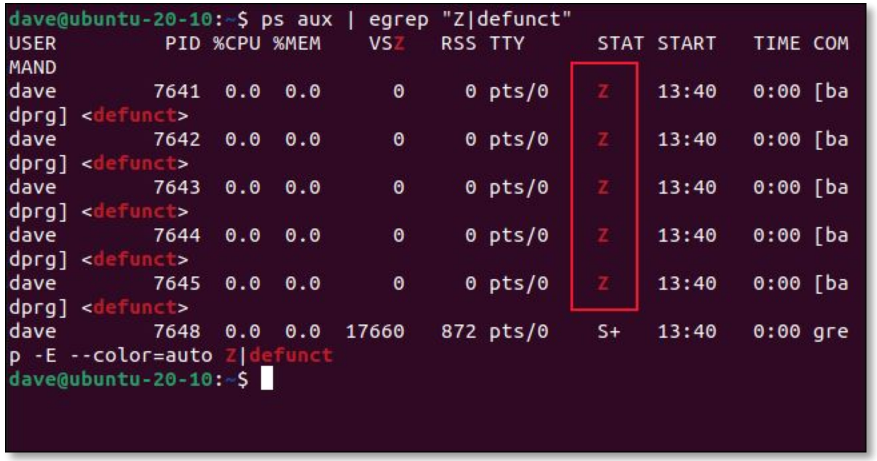


Types of Linux Processes

- **Init Process**: The "init" process is the first process that gets started when the Linux operating system boots up, and it becomes the parent process for all other processes.
- **Parent and Child Processes**: Each user process has a parent process in the system, with most commands having a shell as their parent.
- **Orphan Processes**: When a child process is killed or terminated, the parent process is updated about it through the SIGCHLD signal. But, when the parent process is killed before the termination of the child process, the child process becomes an orphan process with "init" process as its new PID.
- **Zombie Processes**: A process which is killed but still shows its entry in the process status or the process table is called a zombie process, they are dead and are not used.
- **Daemon Processes**: Daemon processes are system-related background processes that run with root permissions and wait for requests from other processes. They often run in the background and can work with other processes. Examples include print daemon.



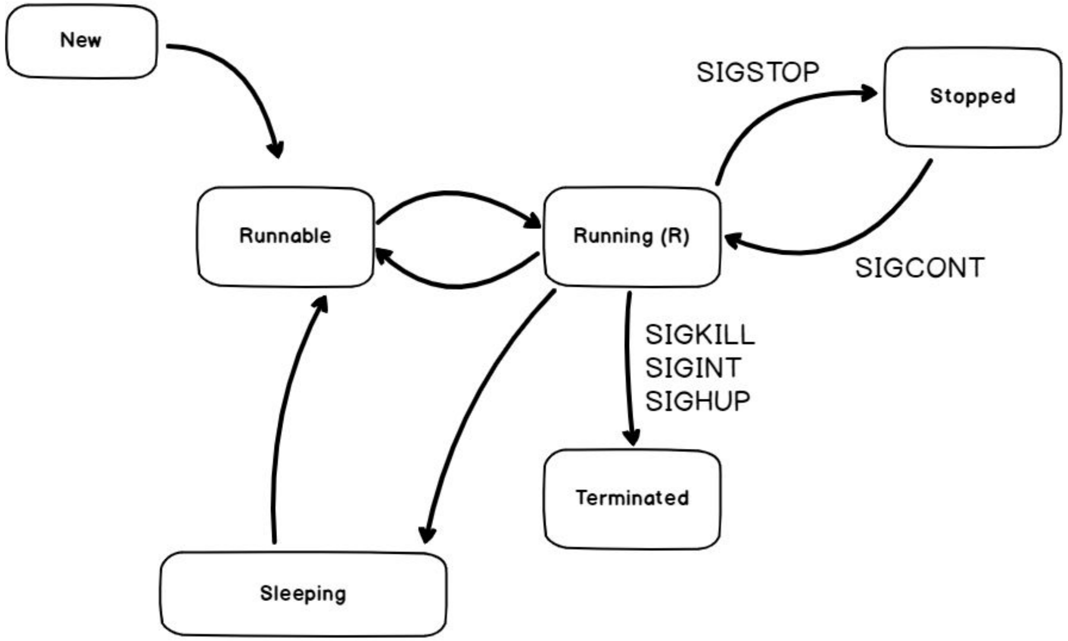
```
top - 13:35:53 up  1:35,  1 user,  load average: 0.07, 0.09, 0.02
Tasks: 202 total,   2 running, 192 sleeping,   0 stopped,   8 zombie
%Cpu(s):  0.3 us,  0.2 sy,  0.0 ni, 99.5 id,  0.0 wa,  0.0 hi,  0.0 si
MiB Mem :   1983.8 total,    128.0 free,    794.3 used,   1061.4 buff/
MiB Swap:   1497.1 total,   1488.2 free,      8.8 used.   1000.6 avail

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
   3235 dave      20   0  860440  86944  51804 S   1.0   4.3
   3504 dave      20   0 4244648 367528 129052 S   1.0  18.1
   5789 dave      20   0 1173496 119668  42992 S   0.3   5.9
      1 root      20   0  317496  12908   8672 S   0.0   0.6
      2 root      20   0       0      0      0 S   0.0   0.0
      3 root       0 -20       0      0      0 I   0.0   0.0
      4 root       0 -20       0      0      0 I   0.0   0.0
      6 root       0 -20       0      0      0 I   0.0   0.0
      9 root       0 -20       0      0      0 I   0.0   0.0
     10 root      20   0       0      0      0 S   0.0   0.0
     11 root      20   0       0      0      0 I   0.0   0.0
     12 root      rt   0       0      0      0 S   0.0   0.0
     13 root     -51   0       0      0      0 S   0.0   0.0
     14 root      20   0       0      0      0 S   0.0   0.0
```



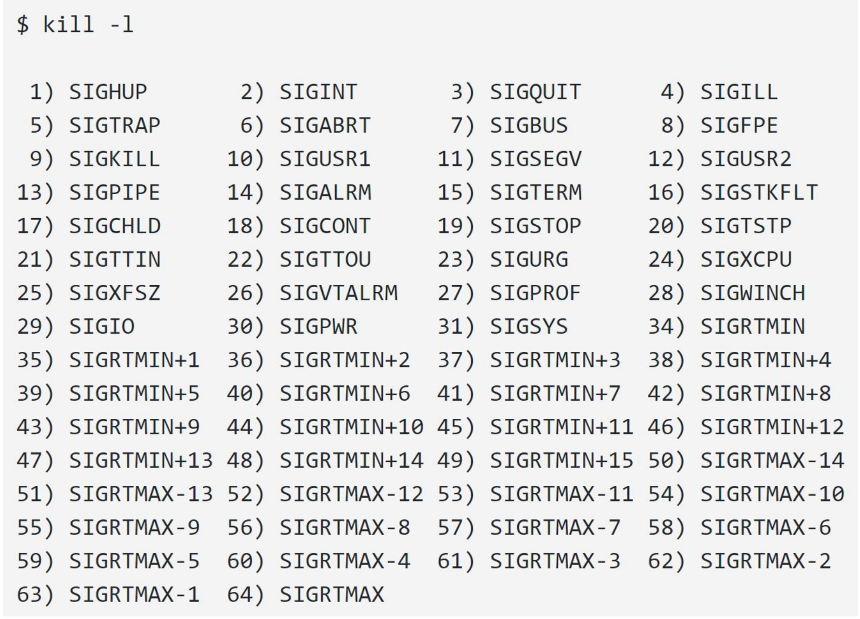
```
dave@ubuntu-20-10:~$ ps aux | egrep "Z|defunct"
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COM
MAND
dave        7641  0.0  0.0      0     0 pts/0    Z    13:40   0:00 [ba
dprg] <defunct>
dave        7642  0.0  0.0      0     0 pts/0    Z    13:40   0:00 [ba
dprg] <defunct>
dave        7643  0.0  0.0      0     0 pts/0    Z    13:40   0:00 [ba
dprg] <defunct>
dave        7644  0.0  0.0      0     0 pts/0    Z    13:40   0:00 [ba
dprg] <defunct>
dave        7645  0.0  0.0      0     0 pts/0    Z    13:40   0:00 [ba
dprg] <defunct>
dave        7648  0.0  0.0  17660   872 pts/0    S+   13:40   0:00 gre
p -E --color=auto Z|defunct
dave@ubuntu-20-10:~$
```

Process States & Signals

- Linux signals are a form of interprocess communication (IPC) used to notify a process of an event or condition.
- Signals can be sent to a process by another process, by the kernel, or by the process itself.
- Each signal has a unique number that identifies it, such as SIGTERM for termination or SIGKILL for immediate termination.
- Signals can be sent using the kill command, the kill system call, or other IPC mechanisms such as pipes or sockets.
- When a process receives a signal, it can either ignore the signal, handle the signal, or perform the default action associated with the signal.
- The signal system call allows a process to specify a custom signal handler function to be executed when a particular signal is received.
- Some common use cases for signals include graceful termination of a process, handling user interrupts, and sending status updates between processes.



Sleep can be interrupted or not

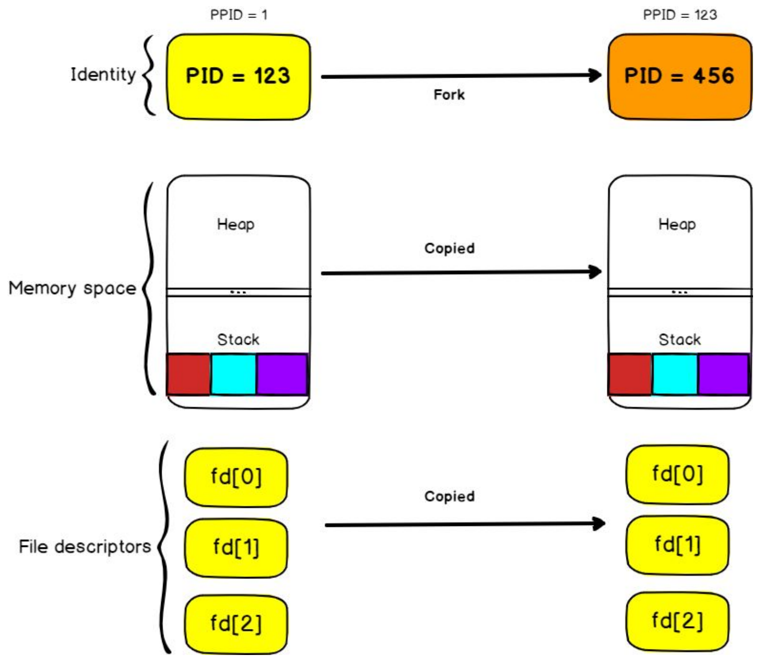


```
$ kill -l

 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

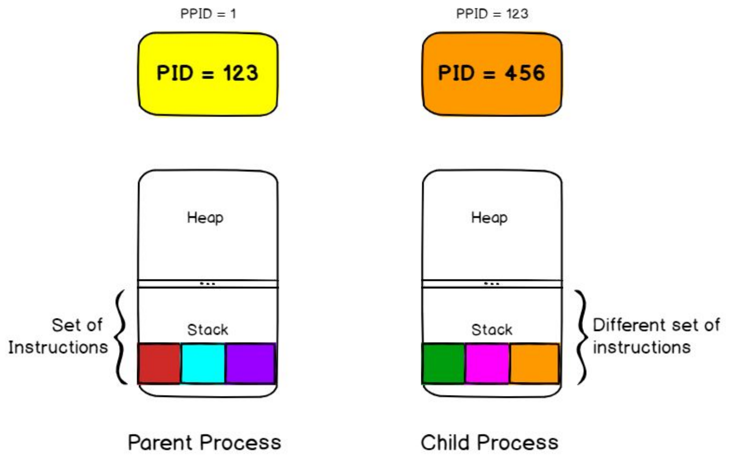
fork() and exec()

- fork()
  - Fork is a clone operation, it takes the current process, also called the parent process, and it clones it in a new process with a brand new process ID.
  - When forking, everything is copied from the parent process : the stack, the heap, but also the file descriptors meaning the standard input, the standard output and the standard error.
  - It means that if my parent process was writing to the current shell console, the child process will also write to the shell console.
- exec()
  - The execute operation is used on Linux to replace the current process image with the image from another process.

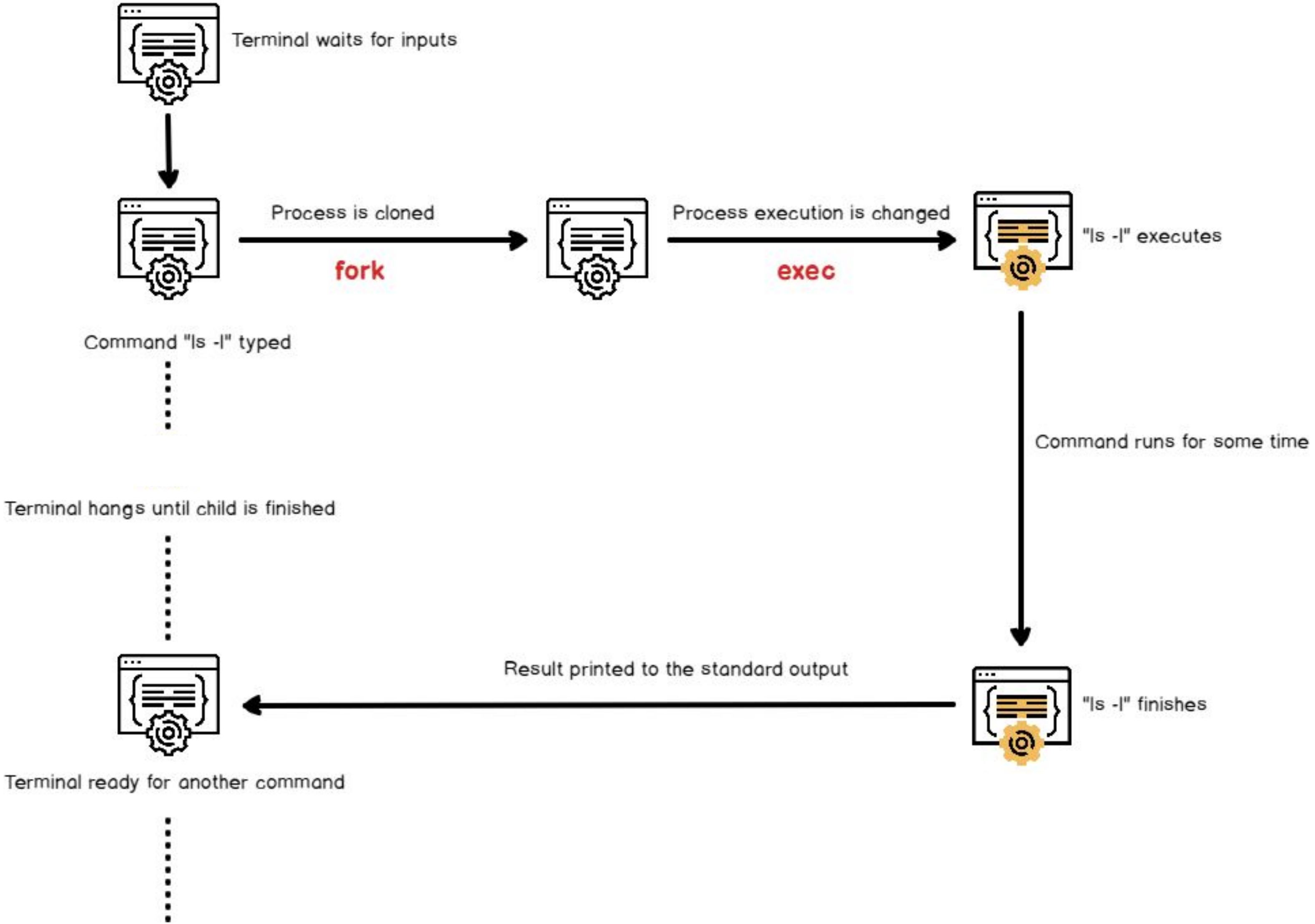
Fork Operation



Exec Operation



Shell Command Execution


Terminal waits for inputs
Process is cloned
fork
Process execution is changed
exec
"ls -l" executes
Command "ls -l" typed
Command runs for some time
Terminal hangs until child is finished
Result printed to the standard output
"ls -l" finishes
Terminal ready for another command

- A shell console is a process that waits for input from the user.
- It also launches a bash interpreter when you hit Enter and it provides an environment for your commands to run.

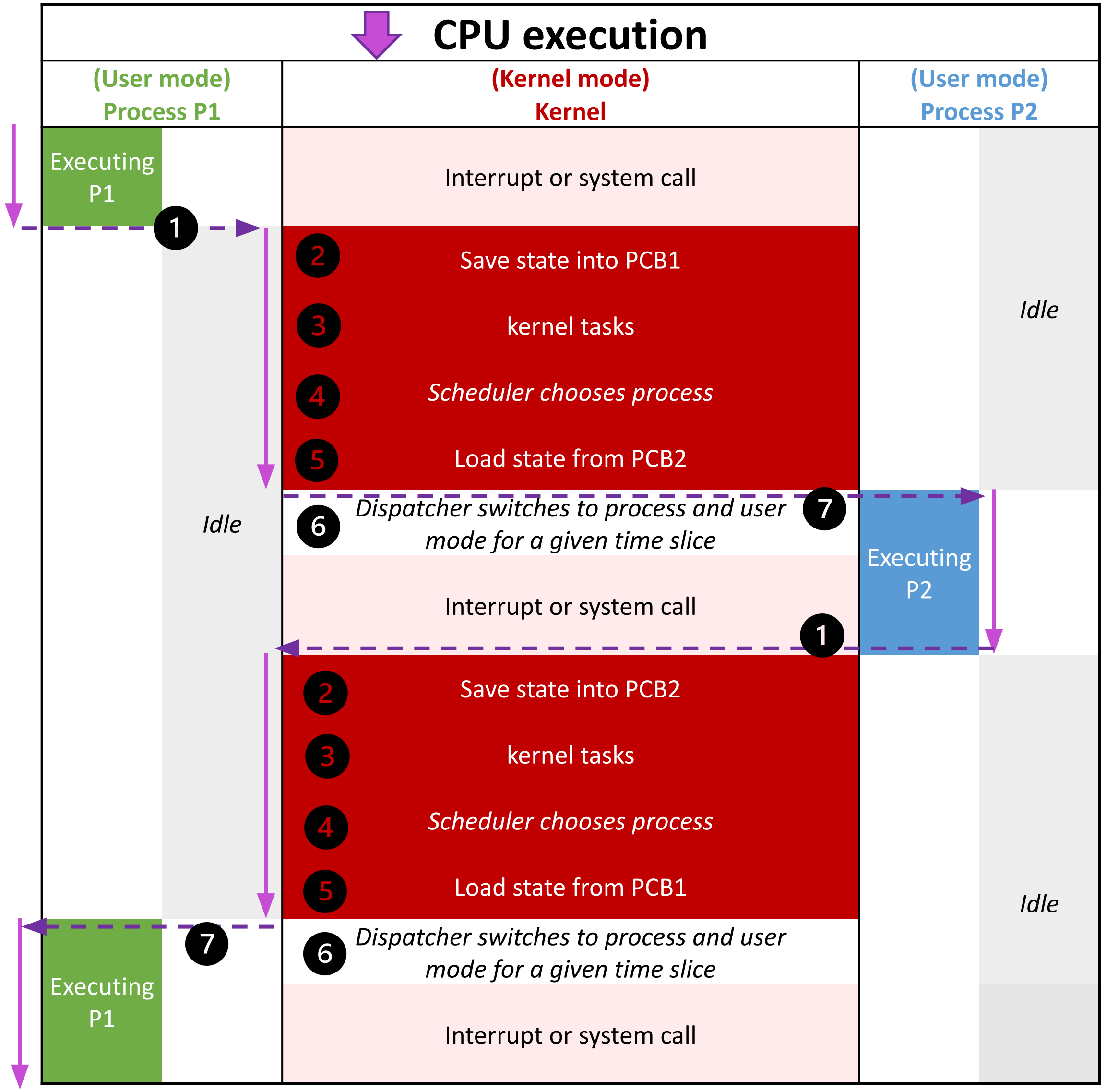
- When you type a command and hit enter, **the shell is forked to a child process** that will be responsible for running your command. The shell will wait patiently until the execution of the child process finishes.
- On the other hand, **the child process is linked to the same file descriptors** and it may share variables that were declared on a global scope.
- The child process executes the "**exec**" command in order to replace the current process image (which is the shell process image) in the process image of the command you are trying to run.
- The child process will eventually finish and it will print its result to the standard output it inherited from the parent process, in this case the shell console itself.

Kernel: Process Management - context switching

PCB: Process Control Block: state of a process with program counter (PC), stack pointer (SP), status of opened files

1. The CPU (the actual hardware) interrupts the current process based on an internal timer, switches into kernel mode, and hands control back to the kernel.
2. The kernel records the current state of the CPU and memory, which will be essential to resuming the process that was just interrupted.
3. The kernel performs any tasks that might have come up during the preceding time slice (such as collecting data from input and output, or I/O, operations).
4. The kernel is now ready to let another process run. The kernel analyzes the list of processes that are ready to run and chooses one.
5. The kernel prepares the memory for this new process and then prepares the CPU.
6. The kernel tells the CPU how long the time slice for the new process will last.
7. The kernel switches the CPU into user mode and hands control of the CPU to the process.

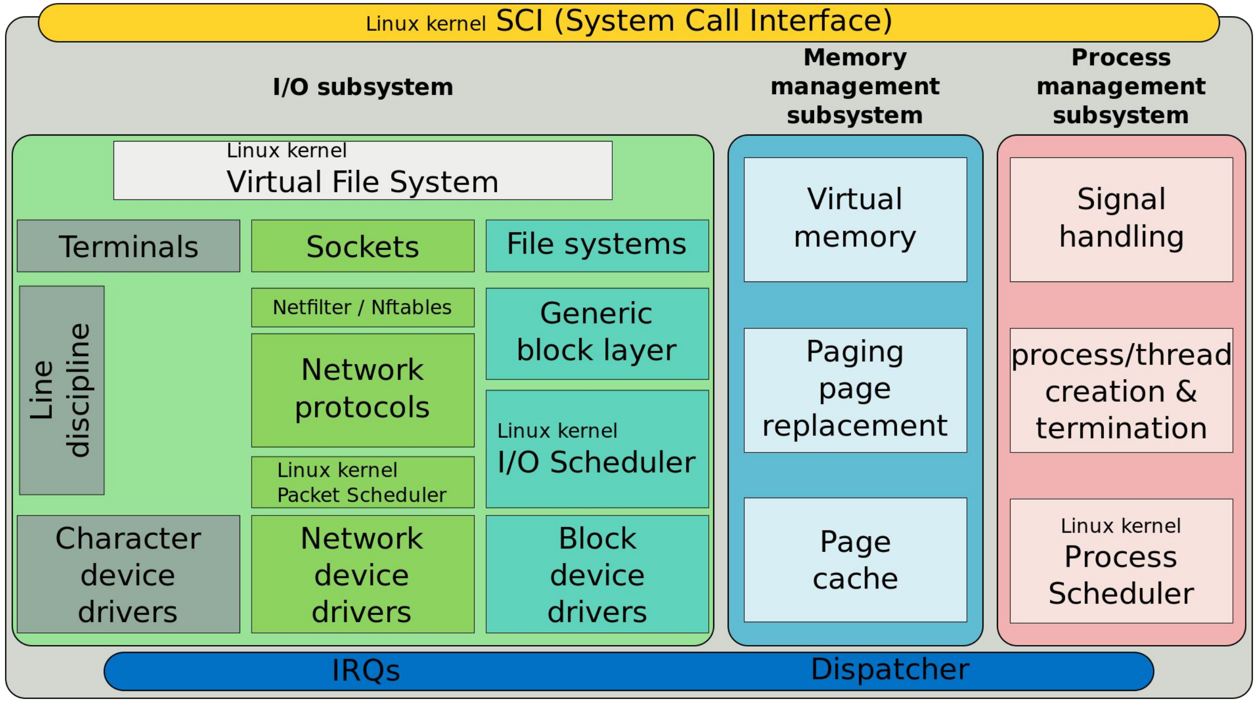
The context switch answers the important question of *when* the kernel runs. The answer is that it runs *between* process time slices during a context switch.

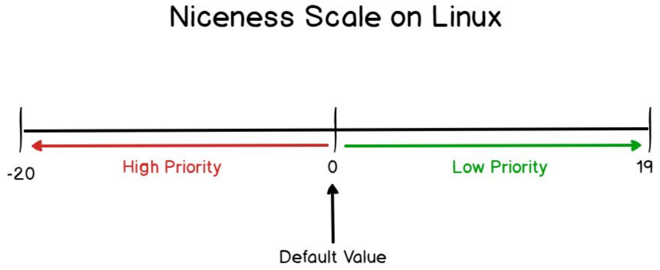


extra info: https://www.youtube.com/watch?v=lS1GOdXFLJo

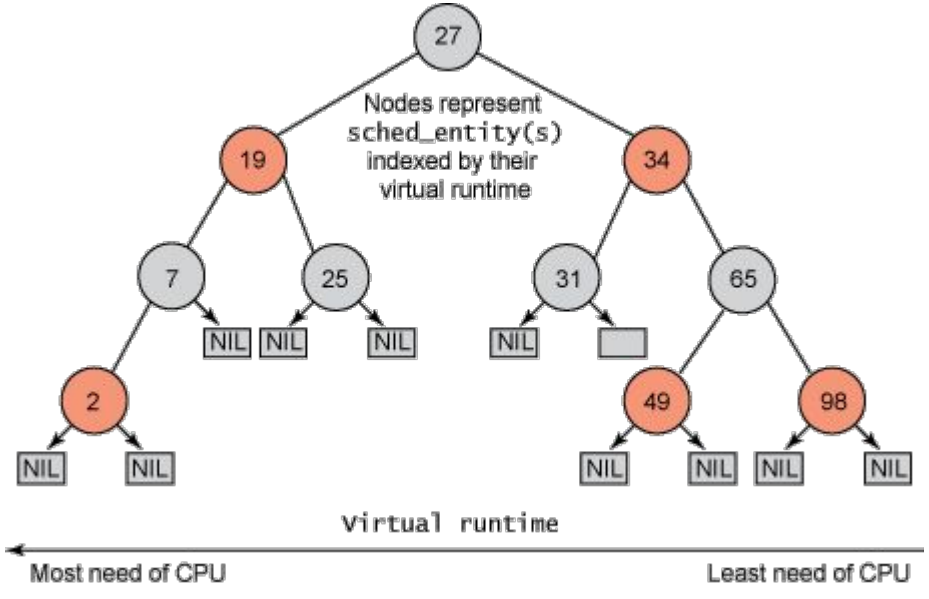
The Linux Scheduler

- Responsible for deciding which process to run next.
- The scheduler uses a variety of factors to make informed decisions, including process priorities, CPU utilization, process states, and scheduling classes.
- Each process is assigned a priority value, which ranges from -20 (highest priority) to 19 (lowest priority).
- The scheduler uses a **priority-based scheduling algorithm**, but also takes into account other factors to ensure that system resources are used efficiently.
- The scheduler can dynamically adjust priorities based on system load and other factors, e.g. process affinity and process real-time constraints.
- Linux supports multiple scheduling classes, each with its own scheduling algorithm, to optimize performance for different types of workloads.
- The **Completely Fair Scheduler** (CFS) is the default scheduler in most Linux distributions, which uses a fairness algorithm to ensure that all processes receive an equal share of the CPU time.





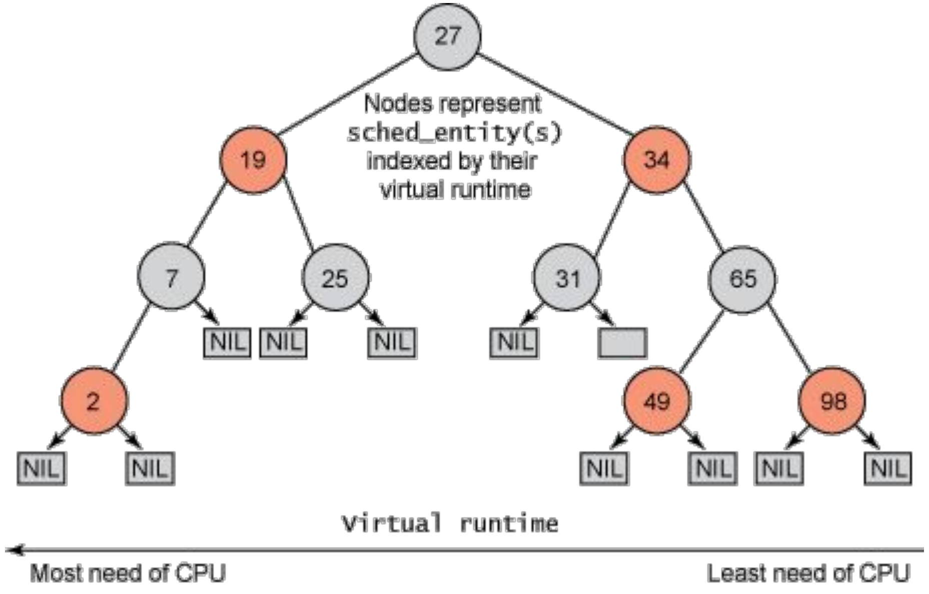
Completely Fair Scheduler



- Linux scheduler manages groups of threads, multi-threaded processes, and all the processes of a given user.
- Tasks are grouped and managed by scheduler as schedulable entities.
- Each per-CPU run-queue of processes sorts schedulable entities structures in a time-ordered fashion into a red-black tree.
- The leftmost node in the tree is occupied by the entity that has received the least slice of execution time, indexed by processor "execution time" in nanoseconds.

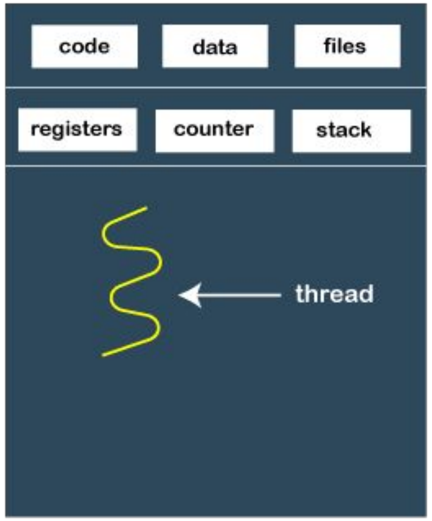
Completely Fair Scheduler

- When the scheduler is invoked to run a new process, it selects the leftmost node in the scheduling tree to execute.
- Each process has a "maximum execution time" that represents the time the process would have expected to run on an "ideal processor"
- When a process is stopped or reaches its maximum execution time, it is reinserted into the scheduling tree based on its newly spent execution time
- The new leftmost node is selected from the tree for execution
- If a process spends a lot of its time sleeping, its spent time value is low, and it automatically gets the priority boost when it finally needs it

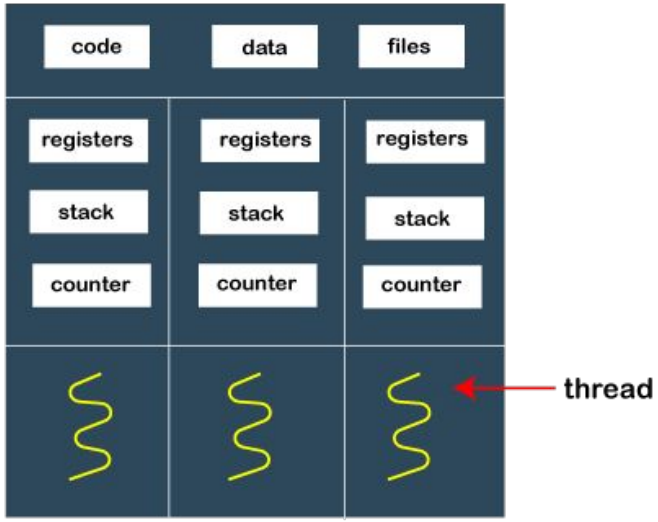


Linux threads

- Threads are lightweight execution contexts that share memory and code within a parent process.
- For example, in a browser, multiple tabs can be different threads.
- Each thread has its own stack and program counter (PC), but all threads within a process share the same heap and global variables.
- Threads allow a program to perform multiple tasks simultaneously.
- Threads within a process can communicate with each other using shared memory or other IPC mechanisms.
- Threads can synchronize their actions using mutexes, semaphores, and other synchronization primitives.



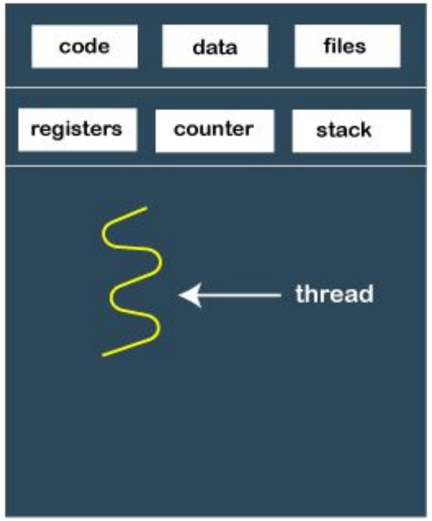
Single-threaded process



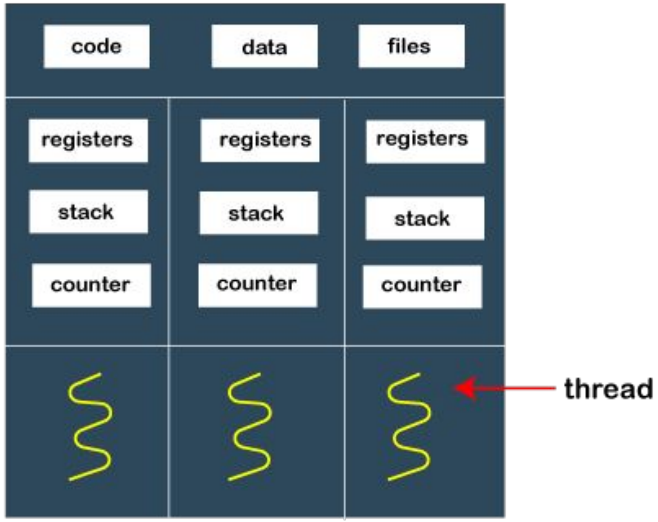
Multi-threaded process

Advantages of threads over processes

- **Responsiveness**: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Faster context switch**: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- **Effective utilization of multiprocessor system**: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors.
- **Resource sharing**: Resources like code, data, and files can be shared among all threads within a process.
- **Communication**: Communication between multiple threads is easier, as the threads shares common address space. Processes require Inter-Process Comununication (IPC) techniques.



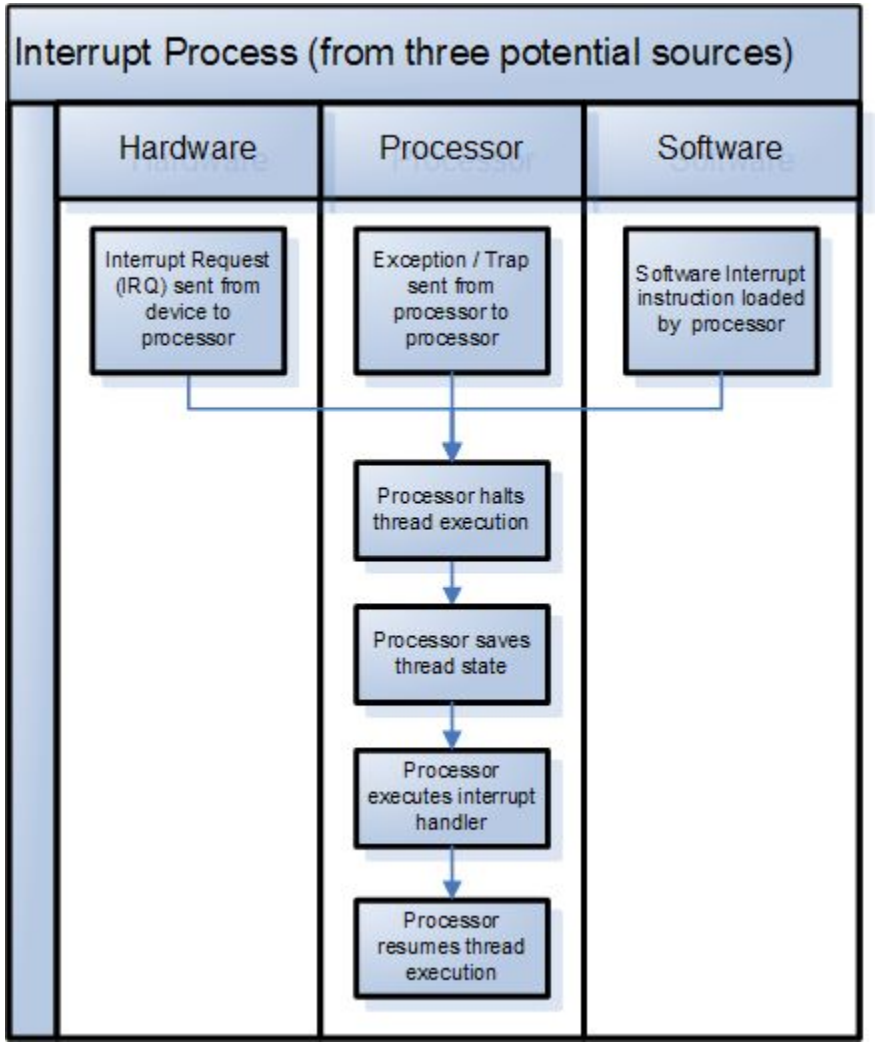
Single-threaded process



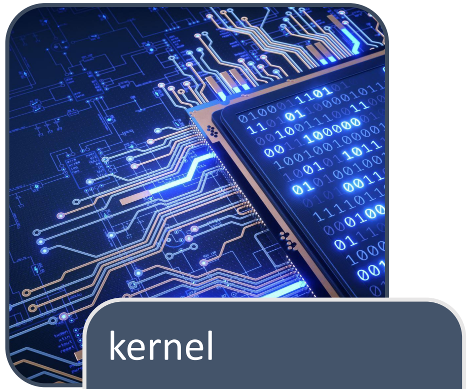
Multi-threaded process

Interrupts in Linux

- Signal emitted by hardware or software when a process or an event needs immediate attention.
- **Interrupts can be generated by 3 sources: Hardware, processor and software (see figure).**
- Interrupts cause the CPU to switch to calling the interrupt handler.
- The **interrupt handler** is a software routine that performs a specific task, such as reading data from a hardware device or notifying the kernel of a user input event.
- Interrupts in Linux are managed by the Interrupt Service Routine (ISR) and the **Interrupt Request (IRQ)** subsystem.
- The ISR is invoked to handle the interrupt, and the corresponding IRQ number is passed to the ISR.
- The Linux kernel supports **interrupt coalescing**, combining many interrupts into one, which reduces the overall overhead of handling interrupts.


Interrupt Process (from three potential sources)
Hardware
Processor
Software
Interrupt Request (IRQ) sent from device to processor
Exception / Trap sent from processor to processor
Software Interrupt instruction loaded by processor
Processor halts thread execution
Processor saves thread state
Processor executes interrupt handler
Processor resumes thread execution

Systems Advanced II - Linux



kernel

- virtual memory
- user space and kernel space
- processes and threads
- sockets & pipes
- systemd
- Docker

Networking

- routing
- firewall
- eBPF
- certificates
- ssh
- latency
- bandwidth

Services

- nfs (rook)
- load balancers and reverse proxies
- databases
- logging

Clusters

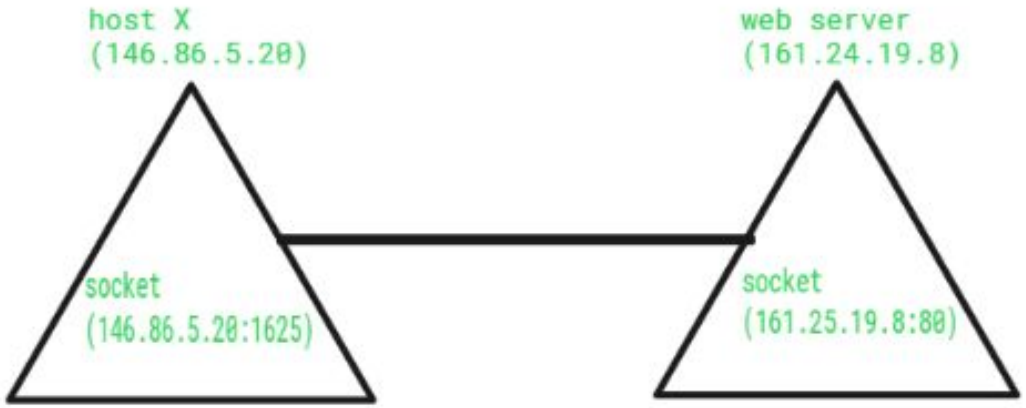
- Storage
- Kubernetes

What are sockets?

- Linux sockets are a type of file descriptor that allow processes to communicate with each other over a network or between processes on the same system.
- The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place and provide a standardized way to send and receive data between processes or between a process and a network service.
- Sockets can be created and managed using various system calls and APIs, such as socket(), bind(), listen(), accept(), connect(), send(), and recv().
- The socket API is implemented in the Linux kernel and is used by many network services and applications.
- Sockets are widely used for various network protocols, such as TCP/IP, UDP, and Unix domain sockets.

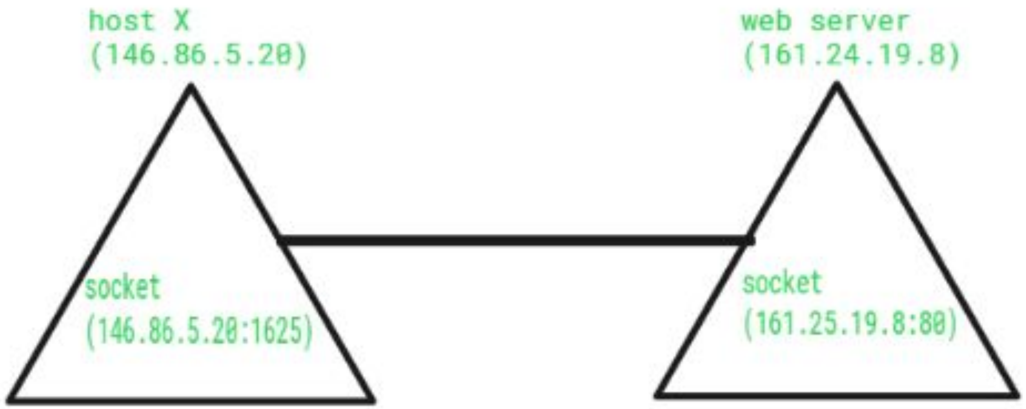
What are sockets used for?

- Socket are generally employed in client-server applications.
  - The server creates a socket, attaches it to a network port addresses then waits for the client to contact it.
  - The client creates a socket and then attempts to connect to the server socket.
  - When the connection is established, transfer of data takes place.



Sockets - practically

- A socket connecting to the network is created at each end of the communication.
- Each socket has a specific address, composed of an IP address and a port number.



Lab: linux sockets in bash

- built-in feature of the bash shell: open TCP/UDP sockets via the `/dev/tcp` (and `/dev/udp`) device file.
- `echo "text!" > /dev/$PROTO/$HOST/$PORT`
- `exec {file-descriptor}<>/dev/{protocol}/{host}/{port}`
    - File descriptors 0, 1 and 2 are reserved for stdin, stdout and stderr, respectively. Thus you must specify 3 or higher
    - <> implies that the socket is open for both reading and writing. Depending on your need, you can open a socket for read-only (<) or write-only (>).
    - The protocol field can be either tcp or udp.

```
#!/bin/bash

HOST=www.mit.edu
PORT=80

(echo >/dev/tcp/${HOST}/${PORT}) &>/dev/null
if [ $? -eq 0 ]; then
    echo "Connection successful"
else
    echo "Connection unsuccessful"
fi
```

```
#!/bin/bash
exec 3<>/dev/tcp/"$1"/80
echo -e "GET / HTTP/1.1\n" >&3
cat <&3
```

extra info: https://www.xmodulo.com/tcp-udp-socket-bash-shell.html

End