Operating Systems (02340123)
Summary - Spring 2025

Razi & Yara

July 22, 2025

Contents

Part I

Lectures & Tutorials

Topic 1: Introduction

Operating System (OS) An Operating System's job is:

- Coordinate the execution of all SW, mainly user apps.
- Provide various common services needed by users & apps.
- An OS of a physical server controls its physical devices, e.g. CPU, memory, disks, etc.
- An OS of a virtual server only *believes* it does. There's another OS underneath, called **hypervisor** which fakes it.

Using an OS allows us to take advantage of ***"virtualization"***:

- **Server Consolidation:** Run multiple servers on one physical server. This is allows for better resource utilization, smaller spaces, and less power consumption.
- **Disentangling SW from HW:** allows for backing up/restoring, live migration, and HW upgrade. This give us the advantage of easier provisioning of new (virtual) servers = "virtual machines", and easier OS-level development and testing.

Most importantly, an OS is **reactive**, "event-driven" system, which means it waits for events to happen and then reacts to them. This is in contrast to typical programs which run from start to end without waiting for external events to occur to invoke them.

	Typical Programs	**OS**
What does it typically do?	Get some input, do some processing, produce output, terminate	Waits & reacts to "events"
Structure	Has a `main` function, which is (more or less) the entry point	No `main`; multiple entry points, one per event
Termination	End of `main`	Power shutdown
Typical goal	$\sim$ Finish as soon as possible	Handle events as quickly as possible $\Rightarrow$ more time for apps to run

Event Synchronousation OS events can be classified into two:

- **Asynchronous interrupts:** keyboard, mouse, network, disk, etc. These are events that can happen at any time and the OS must be ready to handle them.
- **Synchronous:** system calls, divide by zero, page faults, etc. These are events that happen as a result of the program's execution and the OS must handle them immediately.

Multiplexing Multiplexing is the ability of an OS to share a single resource (e.g. CPU, memory, disk) among multiple processes or threads. This allows for better utilization of resources and enables multiple applications to run concurrently. *Multiprogramming means multiplexing the CPU recourse.

Notable services provided by an OS:

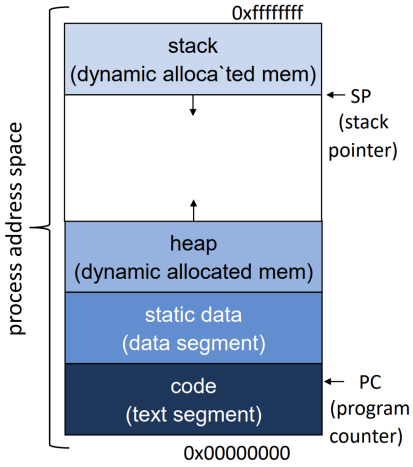
1. **Isolation:** Allow multiple processes to coexist using the same resources without stepping on each other's toes. Usually achieved by multiplexing the CPU, memory, and other resource done by the OS. However, some physical resources know how to multiplex themselves, e.g. network cards, sometimes called *"self-virtualizing devices"*.
2. **Abstraction:** Provides convenience & portability by:
   - offering more meaningful, higher-level interfaces
   - hiding HW details, making interaction with HW easier.

CPL (Current Privilege Level) The CPU defines two levels: (1) User mode, CPL=3, (2) Kernel mode, CPL=0. Privileged instructions require kernel mode to execute them, such as disk access. Syscalls (system calls) our only gateway as users to execute commands in kernel mode, where a syscall raises the CPL to 0 and executes a specific request.

Topic 2: Processes & Signals

Processes

Each process is an instance of a program in execution, which includes:



- **Program code:** The actual code of the program.
- **Process state:** The current state of the process, including the program counter, registers, and memory management information.
- **Process control block (PCB):** A data structure used by the OS to manage the process, containing information such as process ID, process state, CPU registers, memory management information, and I/O status information.

  A process doesn't have direct access to its PCB, it is managed by the OS (kernel space), which is `task_struct current`.

  Each PCB contains: `real_parent`, `parent`, `children`, `siblings`,...
  Each process has a `task_struct current` pointer to its PCB.
- **Process ID (PID):** A unique 32-bit identifier assigned to each process by the OS. Typically only the 15 LSB bits are used. The OS keeps a hash table of PID → PCB.

Process States A process can be in one of the following states:

- **Running:** The process is currently being executed by the CPU.
- **Ready:** The process is ready to be executed but is waiting for the CPU to become available.

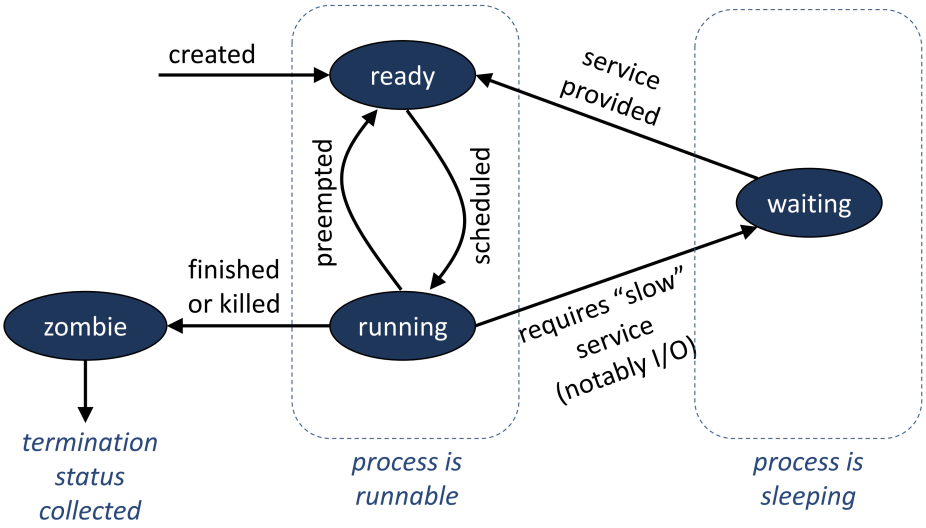


Figure 1: Process States

- **Waiting:** The process is waiting for an event to occur, such as I/O completion or a signal.
- **Zombie:** The process has terminated but its PCB is still in the system, waiting for the parent process to read its exit status. In this state, the process has released almost all of its resources, but the PCB is still in the system.

`state` Field in PCB Each PCB keeps a 32bit `state` variable where each bit indicates one of the following states:

1. `TASK_RUNNING` : The task is either running or ready to run
2. `TASK_ZOMBIE` : The task has terminated but is still in the system
3. `TASK_INTERRUPTIBLE` : The task is waiting for an event to occur, but it can be interrupted by signals and come back to TASK_RUNNING state.
4. `TASK_UNINTERRUPTIBLE` : The task is waiting for an event to occur, but it cannot be interrupted by signals. e.g. page fault, you can't run anything (or handle any other signals) until the page fault is handled to an end.
5. `TASK_STOPPED` : The task has been stopped in a controlled manner by a different task, e.g. by a debugger or by the user pressing Ctrl+Z in the terminal.

As we saw in "ATAM", each process can only access a certain set of utilities and functions, those who require privilege level 3 (user mode). So to access the OS services, a process must use **system calls** which are functions provided by the OS that allow processes to request services from the OS. System calls are typically implemented in the OS kernel and provide a controlled interface for processes to interact with the OS.

Each *syscall*, in case of an error, will change the `errno` variable to indicate the error type. The `errno` variable is a global variable that is set by system calls and some library functions in the event of an error to indicate what went wrong. It is defined in the header file `errno.h`. **Note:** `errno` is not reset to 0 after a successful syscall, so it must be checked immediately after the syscall, and be reset before usage if need be (if there is not any other way to make sure there is an error indeed).
i.e. `errno = <Number of Last Syscall Error>;`

As noted above, each process must be `wait()`ed for by its parent process to be able to release its PCB and resources. This is done by the `wait()` syscall, which suspends the calling process until one of its children terminates. In case the parent process terminates before the child, the child process becomes an orphan process and is adopted by the init process (PID 1), which will then wait for it to terminate and release its resources.

Process Management The OS offers various system calls to manage processes, including: (More details in the functions reference)

- `fork()`: Creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process. For the child, the return value is `0`, and for the parent its the pid of the child, (on failure -1 for parent).
- `exec()`: Replaces the current process image with a new process image, effectively running a different program in the same process.
- `wait()`: Suspends the calling process until one of its children terminates.
- `exit()`: Terminates the calling process and releases its resources.
- `getpid()`: Returns the process ID of the calling process.
- `getppid()`: Returns the process ID of the parent process.
- `kill()`: Sends a signal to a process, which can be used to terminate or suspend the process.

Parent Vs. Real Parent Process The real parent process is the one that created the current process using `fork()`, or the one that adopted it in case the real parent terminated before the child.

The parent process is the one *tracing* the current process, e.g. using `ptrace()`. The parent process is the one that will receive signals from the current process, e.g. `SIGCHLD` when the current process terminates.

In most cases, the parent process is the real parent process, but it can be different in some cases, e.g. when a process is being traced by a debugger.

Daemon Processes A daemon process is a background process not controlled by the user. To run a process as a daemon use `nohup <command> &`.

Daemon names usually end with the letter "d", e.g. `sshd` (SSH daemon), `httpd` (HTTP daemon), etc.

Idle & Init The `idle` process is the first one that the system loads, with `pid=0`, and it utilized `hlt` when no process is ready to run. The next one is `init` which is created by idle, with `pid=1`, and it created all other processes. Every orphan process becomes the child of `init`.

Process List The OS maintains a cyclic double linked list of all processes, allowing for efficient process management and scheduling. The list starts with `idle` then `init`, where `idle` is the head of the list pointed to by `init_task`.

Signals

Signals Signals are "notifications" sent to a process to asynchronously notify it that some event has occurred.

* Receiving a signal only occurs when returning from kernel mode, which in turn invokes the corresponding signal handler, i.e. signal handling is in user mode.

** Default signal handling actions: Either die or ignore

*** In case of several signals from different types, they will be handled by the order of their definition in the signals register.

Each signal has a name, a number (1-31), and a default action. All but 3 signals can be blocked, i.e. ignored until the process is ready to handle them. The 3 signals that cannot be blocked are:

- `SIGKILL`: Used to forcefully terminate a process. (Process becomes a zombie)
- `SIGSTOP`: Used to suspend the receiving process. (Make it sleep) The signals is sent when the user presses Ctrl+Z in the terminal. Note: In truth Ctrl+Z sends the `SIGTSTP` signal, however, we don't learn about the differences between the two signals in this course.
- `SIGCONT`: Used to resume a suspended process, usually sent after a `SIGSTOP` signal. The handler for this signal can be customized but it will always resume the process.

`SIGSTOP` and `SIGCONT` are useful for debugging purposes, allowing the user to pause and resume the execution of a process.

Signal Handling A process can define a custom signal handler for a specific signal using the `signal()` or `sigaction()` preferred system calls. To ignore a signal, the process can set its handler to `SIG_IGN`. To restore the default action for a signal, the process can set its handler to `SIG_DFLT`. codesignal_struct is the kernel structure in PCB that holds the signal handlers for the process.

Signal Masking A process can block signals using the `sigprocmask()` system call, which allows the process to specify a set of signals to block. This is allows the process to overcome *Race Conditions* resulted from the asynchronous nature of signals.

This is achieved by maintaining a set of currently blocked signals & a set of masked signals which is saved in the PCB.
`Blocked Signals` = mask array.
`Pending Signals` = signals that were sent to the process while it was blocked, and will be handled when the process unblocks them.

When handling a signal, the handler masks signals of the same type (of the one being handled), to prevent problems of reentrancy.

Common Signals the following are some of the most common signals:

1. `SIGSEGV, SIGBUS, SIGILL, SIGFPE`: These are driven by the associated (HW) interrupts - The OS gets the associated interrupt, then the OS interrupt handler sees to it that the misbehaving process gets the associated signal, lastly the signal handler is invoked.

   - `SIGSEGV`: Segmentation violation (illegal memory reference, e.g., outside an array).
   - `SIGBUS`: Dereference invalid address (null/misaligned, assume it's like SEGV).
   - `SIGILL`: Illegal instruction (trying to invoke privileged instruction).
   - `SIGFPE`: Floating-point exception (despite the name, *all* arithmetic errors, not just floating point. e.g., division by zero).

2. `SIGCHLD`: Parent (and then real parent) get it whenever `fork()`ed child terminates or is `SIGSTOP`-ed.

3. `SIGALRM`: Get a signal after some specified time, can be set using the `alarm()` & `setitimer()` system calls.

4. `SIGTRAP`: When debugging/single-stepping a process, the debugger can set a breakpoint in the code, which will cause the process to receive a `SIGTRAP` signal when it reaches that point.

5. `SIGUSR1, SIGUSR2`: User-defined signals, user can decide the meaning of these signals and their handlers.

6. `SIGPIPE`: Write to pipe with no readers.

7. `SIGINT`: Sent when the user presses Ctrl+C in the terminal. The default action is to terminate the process, but it can be customized.

8. `SIGXCPU`: Delivered when a process used up more CPU than its soft-limit allows: soft-/hard limits are set using the `setrlimit()` system call. Soft-limits warn the process its about to exceed the hard-limit, Exceeding the hard-limit will cause `SIGKILL` to be sent to the process.

9. `SIGIO`: Can configure file descriptors such that a signal will be delivered whenever some I/O is ready.

   Typically makes sense when also configuring the file descriptor to be *non-blocking*, e.g., when `read()`ing from a non-blocking file descriptor, the system call immediately returns to user if there's currently nothing to read. In this case, `errno` will be set to `EAGAIN=EWOULDBLOCK`.

10. **Signal 0:** The null signal, which is used to test for the existence of a process.

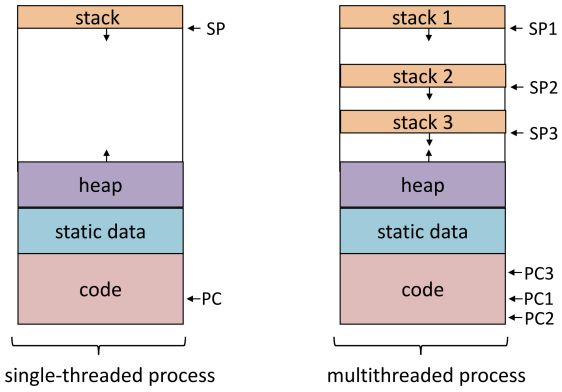
Signals Vs. Interrupts

	interrupts	**signals**
Who triggers them? Who defines their meaning?	Hardware: CPU cores (sync) & other devices (async)	Software (OS), HW is unaware
Who handles them? Who (un)blocks them?	OS	processes
When do they occur?	Both synchronously & asynchronously	Likewise, but, technically, invoked when returning from kernel to user

Topic 3: IPC - Inter-Process Communication

Threads & IPC

Multiprocessing Using several CPU cores (=processors) for running a single job to solve a single "problem".



Threads A process can have multiple threads, which share (nearly) everything, including the process's memory space, file descriptors, heap, static data, code segment and more.

However, each thread has its own stack, registers, and program counter. But they can still access each other's stack since they share the same memory space.

Note that *Global/Static Variables & Dynamically Allocated Memory* are shared between threads since they are stored in the Static Data Segment and the Heap, respectively, which are shared between all threads of a process. However, *Local Variables* are not shared between threads since they are stored in the stack, which is unique to each thread.

	Unique to Process	Unique to pthread
Registers (notably PC)	Y	Y
Execution stack	Y	Y
Memory address space	Y	N
Open files	Y	N
Per-open-file position (=offset)	Y	N
Working directory	Y	N
User/group credentials	Y	N
Signal handling	Y	N

OpenMP Open Multi-Processing (OpenMP) consists of a set of compiler *'pragma'* directives that allows the compiler to generate multi-threaded code for parallel execution.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
        arr[i] = 2*i;
}
```

Pthreads POSIX threads (pthreads) is a standard for multi-threading in C/C++. It provides a set of functions to create and manage threads, as well as to synchronize them.

Common Functions

The following are some of the most commonly used pthread functions:

- `pthread_create(...)`: Creates a new thread that starts executing a given function. The thread ID is stored in a pointer.
- `pthread_self()`: Returns the thread ID of the calling thread. This ID is internal to the pthread library.
- `pthread_exit(...)`: Terminates the calling thread and makes its exit status available to any thread that joins it.
- `pthread_cancel(...)`: Sends a request to terminate a specified thread. The thread's exit status is set to `PTHREAD_CANCELED`.
- `pthread_join(...)`: Blocks the calling thread until a specified thread terminates, then retrieves its exit status and frees its resources.

Thread Termination

Reason for Termination	Thread Terminates?	Process Terminates?
Calling `pthread_exit()`	Yes	Not necessarily
Calling `pthread_cancel()`	Yes	Not necessarily
Return from `start_routine`	Yes	Not necessarily
Calling `exit()`	Yes	Yes
Illegal operation	Yes	Yes
Return from `main`	Yes	Yes

Threads in Linux In Linux, threads are implemented as *processes* with a shared memory space. Each thread has its own PCB, but they share the same memory address space, file descriptors, and other resources. Threads do not have family ties between one another.

Thread Groups A thread group is a collection of threads that share the same process ID (PID) and are managed by the same process. In Linux, each thread group has a unique group ID (TGID), which is the PID of the first thread in the group. The TGID is used to manage signals and other resources shared by the threads in the group.

Since each thread has his own PCB and PID, to send a signal to all threads in a thread group, the signal must be sent to the TGID (the PID of the first thread in the group). Each PCB contains a field `thread_group` which points to the head of the linked list of all threads in the group.

The syscall `getpid()` returns the `TGID` of the calling thread. The personal "PID" of each thread is called `TID` (Thread ID), which is the same as the personal "PID" of the thread's PCB. The syscall `gettid()` returns the TID of the calling thread.

`clone()` The syscall: `clone(int (*fn)(void *), void *child_stack, int flags, void *arg);` is used to create a child process that shares resources with the parent process. The parameters are:

- `fn`: A pointer to the function that will be the main function of the child process.
- `child_stack`: A pointer to the user stack that will be used by the child process. The highest address of the stack is passed, i.e. the top of the stack.
- `arg`: A pointer to the argument that will be passed to the function `fn`.
- `flags`: A bitmask that specifies which resources will be shared between the parent and child processes. The flags can include:
  - `CLONE_VM`: Share the same memory address space.
  - `CLONE_FS`: Share the same filesystem information (e.g., current working directory).
  - `CLONE_FILES`: Share the same file descriptor table.
  - `CLONE_PARENT`: Share the same parent process as the parent of the calling thread.
  - `CLONE_THREAD`: Create a new thread in the same thread group.

The return value is the TID of the child.

File Descriptors A non-negative integer representing an I/O "channel" on some device. File descriptors are saved in the process's `PCB` inside the `FDT` (File Descriptor Table), which is an array of pointers to *file objects*, each representing an open file or device. So in fact, an FD is an index to a kernel array of channels.

Processes don't share `PCB` nor `FDT` but they can share file descriptors, i.e. two processes can have the same file descriptor pointing to the same file object, which allows them to share the same open file or device. Upon forking, the child process inherits a copy of the parent's `FDT`. Note: Threads do share the same `FDT`.

Pipes A pipe is a unidirectional communication channel between two processes, allowing one process to send data to another. Pipes are a pair of two file descriptors `int pipe_fd[2]`. Each integer is a handle to a kernel communication object ("file"), pipes reside only on DRAM (memory) and never on the disk, and they are not in the filesystem. (pipes are anonymous shared pages)

- `pipe_fd[0]` = read side of the communication channel.

- `pipe_fd[1]` = write side of the communication channel.
- Everything written via `pipe_fd[1]` can be read via `pipe_fd[0]`.
- Blocking: If the read side is empty, the read operation will block until data is written to the write side. If the write side doesn't have enough space for all the data, the write operation will block until space is available.
- EOF: If the write side is closed and the pipe is empty, the read operation will return 0, indicating EOF (End of File).
- `SIGPIPE`: Writing to a pipe whose read end is `close()`d will result in a `SIGPIPE` signal.

FIFO (Named Pipe) A FIFO is a named pipe that allows for communication between unrelated processes (no family connection), i.e. it's a public "file". FIFO is a file even though it is NOT on the disk, though it can be swapped there.

To make a FIFO, use the `mkfifo(const char *pathname, mode_t mode)` system call, which creates a special file in the filesystem with path `pathname` and permissions `mode`. The FIFO can then be opened by any process using the `open()` system call, just like a regular file. The FIFO is bidirectional, thus having one FD.

A process that opens FIFO for read only, will block until another process opens it for write, and vice versa. If one opens it as both read & write (O_RDWR) then it will not be blocking. *The read write rules are similar to the one of pipe.

FIFOs are not removed from the system automtically (unlike pipes whom do), so they have to be *explicitly* removed by `rmunlink` for example. Meaning a closed FIFO can be reopened until removed completely.

Multi-tasking	**Multi-programming**	**Multi-processing**
Having multiple processes *time slice* on the same CPU core.	Having *multiple jobs* in the system (either on the same core or on different cores). i.e. the existence of multiple processes in the system, regardless of whether they are running on the same core or not.	Using *multiple processors (CPU cores) for the same job* in parallel. i.e. creating multiple threads to run on different cores for the same process.

Intro. Context Switching & Caching

Context Switching is the process of saving the state of a currently running process and restoring the state of another process to allow it to run. This is done by the OS kernel and is necessary for multitasking, allowing multiple processes to share the CPU.

- **Context** = the state of a process, including its registers, program counter, stack pointer, and memory management information.
- **Context Switch** = the process of saving the context of the currently running process and restoring the context of another process.

Context Switch Overhead consists of two components:

- **Direct Overhead:** The measurable time it takes to perform the context switch, which includes saving the current process's state and restoring the next process's state.
- **Indirect Overhead:** The time it takes for the CPU to cache the new process's data, which can be significant if the new process's data is not already in the CPU cache.

Cashing The CPU cache is a small, fast memory that stores frequently accessed data to speed up access times. The Cache allows the CPU access to a fast memory that is closer to the CPU than the main memory (DRAM), and a larger one than the CPU registers. The cache works because of the *Principle of Locality*:

- **Temporal Locality:** If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again soon.
- **Spatial Locality:** If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced soon.

Copy-on-Write (COW) The `fork()` system call creates a copy of the address space of the parent, but:

- It only creates a *logical* copy.
- There is no physical duplication of the memory pages, until we really need to write to them (from either process). And even then, only the page that is being written to is duplicated. (More details in the virutal memory lectures)

User Level Threads (ULTs) are threads that are managed by the user-level library, rather than the OS kernel. ULTs are not visible to the OS, which means that the OS does not know about them and does not schedule them. This allows for faster context switching between ULTs, but it also means that the OS cannot take advantage of multiple CPU cores to run ULTs in parallel.

The usage of ULTs is mainly for concurrent programming, where we want multiple multiple tasks to progress in parallel without the overhead of kernel-level threads.

Topic 4: Boot & Modules

Booting the OS

Operating System at Power ON When powering on a PC, the OS has yet to load into the memory, as it resides on the disk. So we need to load the OS from the disk to the memory, but we need the OS to do that! So the solution is hierarchal booting:

1. The BIOS code is loaded into the memory (DRAM) and the CPU starts executing it.
2. The BIOS loads the first sector of the disk (MBR) to the memory
3. The MBR code loads the boat loader
4. The boot loader loads the kernel of Linux
5. The kernel loads the init process (`/init`) and runs it

BIOS (Basic Input/Output System) The BIOS is located in the motherboard, loaded to the same memory location always, and is the first code that runs when the computer is powered on. It performs the following tasks:

- Identify the hardware components of the system that are connected.
- Check if basic hardware components are working properly (screen, keyboard, etc)
- Search for the device (from a pre-defined list) that enables booting (bootable device)
- If the device is not found, the BIOS shows an error message and stops the booting process. Otherwise, it loads the first sector of the device to a constant place in memory.

MBR (Master Boot Record) The bootable device holds in the first sector (512 bytes), the MBR, which is a very basic assembly program that is used to load the boot loader, since it requires more space than the MBR can provide.

Boot Loader (Grub) GRUB (GRand Unified Bootloader) is a boot loader package from the GNU Project. It allows users to choose between multiple operating systems at boot time and can load a variety of operating systems, including Linux, Windows, and others.

Loading the Kernel The boot loader loads the kernel in the following steps:

1. Load the compressed kernel image from the disk to the memory.
2. Load a basic initial filesystem (initramfs or initrd)
   - The initial filesystem includes the necessary modules for the linux kernel to find and load the real filesystem.
3. The kernel runs the script `/init` (typically a shell script) from the initial filesystem.
4. The `/init` script loads the necessary drivers and mounts the real filesystem instead of the initial one.
5. After the real filesystem is mounted, the linux kernel runs the `/sbin/init` program.

Modules

Kernel Modules are pieces of code that can be loaded into the kernel at runtime, without need to recompile the kernel. New modules can be loaded only in kernel mode (CPL=0). The primary use for modules is to implement drivers for hardware devices, but they can also be used to implement new system calls, filesystems, and other kernel features. Beware Modules run in kernel mode so they have access to everything.

The advantages of using kernel modules are: (1) Allowing to add functionality without rebooting, (2) Allowing for faster compile time of the kernel, (3) The kernel will use less space, (4) Allowing to add support for new hardware.

To implement a module we use (Beware to use kernel functions and not user functions, e.g. "kprintf" instead of "printf"):

- `int init_module(void)`: The function that is called when the module is loaded into the kernel.
- `void cleanup_module(void)`: The function that is called when the module is unloaded from the kernel.
- `module_param(name, type, perm)`: A macro to declare a module parameter. each parameter must have a default value.

Devices & Drivers Devices are represented by special files in the filesystem, which are called **device files**, which are located in `/dev`. To communicate with a device, we use the standard file operations (open, read, write, close) on the device file. The kernel uses **device drivers** to handle the communication with the device. Device drivers are kernel modules that implement the necessary functions to communicate with the device.

Block vs Character Devices Character devices are ones that can be accessed as a stream of bytes (e.g. keyboard, mouse, etc.), typically for transferring data and in a sequential matter. Block devices are ones that can be accessed only in multiples of blocks (e.g. hard drives), typically for storing data in random-access data.

Character Devices Each character device is represented by a **major number (1-512)** and a **minor number (0-255)**. The major number identifies the driver associated with the device, while the minor number identifies the specific device. Each major number can have multiple minor numbers, and vice versa. Both numbers are found in the `inode` of the device file.

Registering a new device is done by: (default permissions are write for owner, read for others)

```
mknod <Name> <Type (c for character, b for block)> <Major Number> <Minor Number>
```

Pseudo-Devices Linux provides pseudo-character devices that are not connected to any physical hardware. All of these devices share a major number of 1.

write()	**read()**	**device file**
Succeeds and does nothing (data is discarded).	Immediately returns EOF (end of file).	`/dev/null`
	Returns a sequence of null bytes (`\0`) of the requested length.	`/dev/zero`
Immediately returns ENOSPC (no space left on device).		`/dev/full`

Contributes to the kernel's entropy pool from "noise".	Returns a stream of random bytes generated at runtime.	`/dev/random`
		`/dev/urandom`
		`/dev/arandom`

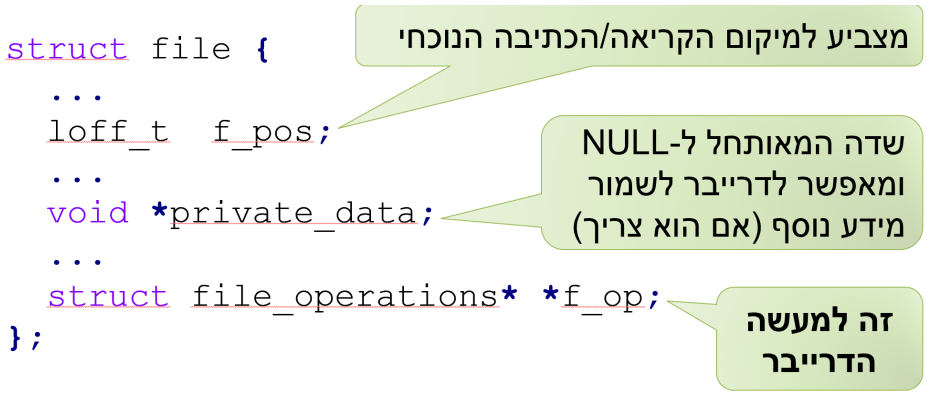
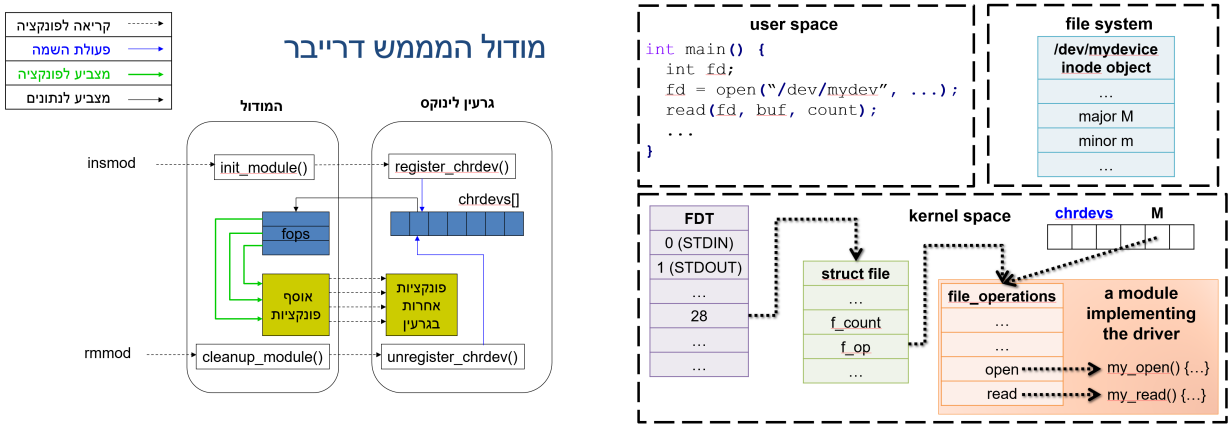


Figure 2: File struct and file operations

file operations Each file struct has a `file_ops` field, which is a pointer to a struct that contains pointers to functions that implement the file operations for that file. These functions include:

- `int (*open) (struct inode *, struct file *)`: Called when the file is opened. If open is null, it will succeed.
- `int (*release) (struct inode *, struct file *)`: Called when the last instance of the file is closed. If release is null, it will succeed.
- `int (*flush) (struct file *)`: Called when the file is closed (every time an FD is closed, not just the last one). If flush is null, it will succeed.
- `ssize_t (*read) (struct file *, char *, size_t, loff_t *)`: Called when the file is read. If read is null, it will return (-EINVAL).
- `ssize_t (*write) (struct file *, const char *, size_t, loff_t *)`: Called when the file is written to. If write is null, it will return (-EINVAL).
- `loff_t (*llseek) (struct file *, loff_t, int)`: Called when the file's offset is to be changed.
- `int (*ioctl) (struct inode *, struct file *, unsigned int cmd_id, unsigned long arg)`: Allows adding additional functionality to the device. Arg is an optional parameter.

Drivers as Modules Drivers can be implemented as kernel modules, where `init_module()` will register the driver using `register_chrdev()`. Note that registering a driver only connects it with a major number, and not to a device. Unregistering a driver is done through `cleanup_module()` that uses `unregister_chardev()`.



(a) Driver Module Flow (b) Module File Flow

Figure 3: Driver and Module File Flows

Registering a Driver To register a driver, we use the `register_chrdev()` function, which adds the driver to the kernel's array of character drivers, `chrdevs[]`, where the index is the major number. It also adds the driver to `/proc/devices` file, which lists all registered drivers and their major numbers (both character & block drivers), character & block devices use different arrays thus multiples of the same number can exist. `chrdevs[]` is an array of driver names + pointers to their `file_ops` structs.

To allocate a major number, Linux allows 512 number options, and each driver can register itself with a major number of its choice (could cause conflicts), or could use 0 which then returns the first available number from 512 to 1.

Unregistering a driver using `unregister_chrdev()` will remove the driver from `chrdevs[]`, but will not remove the device from `/dev`.

Topic 5: Scheduling

Batch (Non-Preemptive) Scheduling

Supercomputers Supercomputers are compromised of multiple nodes, each with multiple CPU cores, and are used for running large-scale computations. Users submit **batch jobs** to the supercomputer with a specified time limit and size. Terminology:

- **Size** = the number of CPU cores to use for the job. Jobs are said to be *wide/big* or *narrow/small*.
- **Runtime** = the time limit for the job to run. Jobs are said to be *short* or *long*.

Metrics for Performance Evaluation When evaluating the performance of a batch scheduling algorithm, we use the following metrics:

- **Average Wait Time:** The wait time of a job is the interval between the time the job is submitted to the time the job starts to run. i.e.

$$\text{waitTime} = \text{startTime} - \text{submitTime}$$

- **Average Response Time:** The response time of a job is the interval between the time the job is submitted to the time the job is terminated. i.e.

$$\text{responseTime} = \text{terminateTime} - \text{submitTime}$$

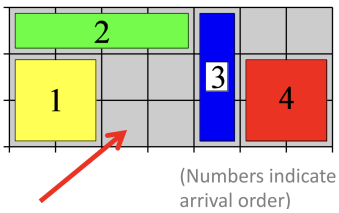
  Note: Average wait time and response time differ only by a constant factor, which is the job's average runtime.

- **The Slowdown / Expansion Factor:** The ratio between a job's response time and its runtime. i.e.

$$\begin{aligned}\text{slowdown} &= \frac{\text{responseTime}}{\text{runtime}} \\ &= \frac{(\text{waitTime} + \text{runtime})}{\text{runtime}} \\ &= 1 + \frac{\text{waitTime}}{\text{runtime}}\end{aligned}$$

- **Utilization:** The percentage of time the resource (CPU) is busy.
- **Throughput:** How much work is done in one time unit.

Batch Scheduling Algorithms



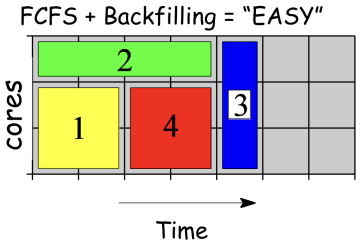
First-Come, First-Served (FCFS) Jobs are scheduled by their arrival time. If there are enough free cores, a newly arriving job starts to run immediately, otherwise it wait, sorted by arrival time, until enough cores are available.

- **Pros:** Simple to implement (FIFO Queue), fair to all jobs.
- **Cons:** Creates *fragmenation*, small/short jobs might wait a long time.

EASY Scheduling (= FCFS + backfilling) Back-filling optimization: A short waiting job can jump over the head of the wait queue (i.e. start at an earlier time) provided that it doesn't delay the job head of the FCFS queue.

The algorithm: whenever a job arrives or terminates, try to start the job head of the FCFS wait queue. Then, iterate over the rest of teh waiting jobs (in FCFS order) and try to backfill them.

- **Pros:** Better utilization (less fragmentation), short jobs have a better chance of running sooner.
- **Cons:** Must know runtimes in advance.



Shortest Job First (SJF) Instead of ordering jobs by their arrival time, we order them by their (typically estimated) runtimes.

- **Pros:** Optimal in terms of performance (min. avg. wait time).
- **Cons:** Unfair as it may cause **starvation** of long jobs. starvation = can theorically wait forever

---

Convoy Effect Slowing down all (possibly short) processes due to currently servicing a very long process. *FCFS* suffers from this effect, as does *EASY* scheduling but to a lesser

extent. *SJF* without assumptions also suffers from this effect but to an even lesser extent.

Optimality of SJF As mentioned above, SJF is optimal in terms of performance, in particular, it minimizes the average wait time. **Claim:**
Given:

1. A 1-core system where all jobs are serial.
2. All process arrive together.
3. Their runtimes are known in advance.

Then: *The average wait time of SJF is equal to or less than the average wait time of any other batch scheduling order.*

Fairer Varients of SJF Motivation: disallow job starvation.

- **Shortest-Job Backfilled First (SJBF):** Exactly like EASY in terms of servicing the head of the wait queue in FCFS order (and not allowing anyone to delay it), but the *backfilling* traversal is done in SJF order, i.e. the next job to be backfilled is the one with the shortest estimated runtime.
- **Largest eXpansion Factor (LXF):** LXF is similar to EASY, but instead of ordering the wait queue in FCFS, it orders jobs based on their current slowdown (greater slowdown = higher priority).

  On every job arrival or termination, the expansion factors are recalculated and the wait queue is resorted, so that the job with the largest expansion factor is always at the head of the queue. Both the head of the queue and the backfilled jobs are serviced in LXF order.

Preemptive Schedulers

Preemption *is the act of suspending one job (process) in favor of another even though it is not finished yet.*

Why do we need preemption? Preemption is necessary for **responsiveness** and when the runtime of jobs vary or unknown in advance.

Quantum *is the maximum amount of time a process is allowed to run before it is preempted.* Quantum is typically milliseconds to 10s of milliseconds, and is *often set per-process.* Usually a CPU-bound process gets long quanta while an I/O-bound process gets short quanta with higher priority.

Performance Metrics for Preemptive Schedulers we use the following:

- **Average Wait Time:** As before, the wait time of a job is the interval between the time the job **is submitted** to the time the job **starts to run**. *Note* that the wait time does not include the preemption wait times (i.e. the time the job is waiting for its turn to run again after being preempted).
- **Response Time (=Turnaround Time):** Like before, the response time is the time from process submission to process completion. *Note* that with preemption we get:

  $$\text{responseTime} \geq \text{waitTime} + \text{runtime}$$

  due to the preemption wait times and **Context Switches** cost.
- **Overhead:** How long a context switch takes, and how often context switches happen. (Should be minimized)
- **Utilization & Throughput:** Same as before, but we may want to account for context switch overhead.
- **Makespan Time:** The time it takes for the system to finish all jobs in the system.

Round Robin (RR) Scheduling *Processes are arranged in a cyclic read-queue.* The algorithm works in the following steps:

1. The head process runs until its quantum is exhausted.
2. The head process is then preempted (suspended) and moved to the tail of the queue.
3. The scheduler resumes the next process in the circular list.
4. When we've cycled through all processes in the run-list (and we reach the head process again), we say that the current **"epoch"** is over, and the next epoch begins.

Note: RR Requires a timer interrupt; Typically it's a periodic interrupt (fires every few milliseconds) and upon receiving the interrupt, the OS checks if its time to preempt. **Note 2:** For small enough quantum, it's like everyone of the N processes advances in $1/N$ of the speed of the core called sometimes virtual time. With huge quantum infinity, RR becomes FCFS.

Gang Scheduling Think of it as RR for *parallel* systems. It works as follows:

- Time is divided to slots (seconds or minutes).
- Every job has a **"native"** time slot.
- Algorithm attempts to fill holes in time slots by assigning to them jobs from other native slots (called **"alternative"** slots).
- Algorithm attempts to minimize slot number using **slot unification** when possible.
- Rarely used in practice, if lots of memory is swapped out upon context switch, context switch overhead is too high.

Detailed explanation about alternative slots and slot unification:

Alternative Slots To further enhance resource utilization, the gang scheduling algorithm attempts to fill holes in time slots. A job has a "native" time slot where it is primarily scheduled to run. However, if there are idle processors in other time slots (termed "alternative" slots), the scheduler can assign jobs from their native slots to run in these otherwise unused resources. This allows jobs to get more processing time without interfering with the primary scheduling of other jobs.

Slot Unification This technique aims to minimize the total number of active time slots. If, after some jobs have completed, the remaining jobs in two different time slots can fit into a single slot without conflict, the scheduler can merge them. By unifying slots, the overall cycle time of the matrix is reduced, which can lead to shorter job turnaround times and improved system throughput.

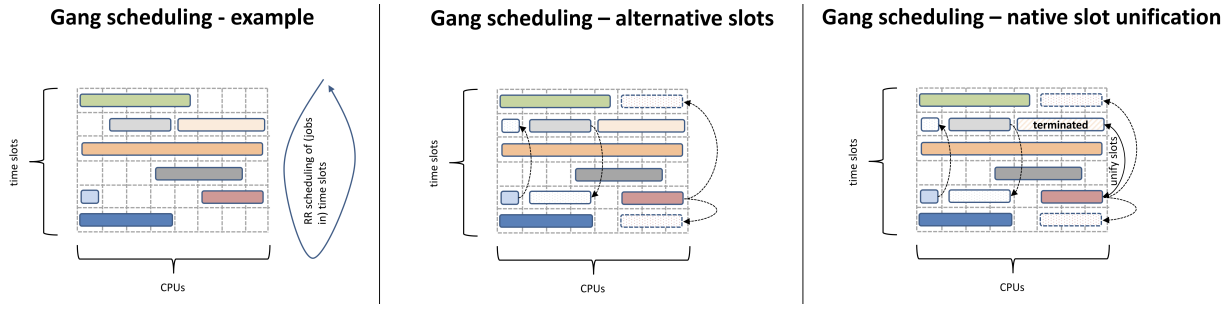


Figure 4: Gang Scheduling

Batching vs. Preemption **Assume:** (1) A single core system (2) All jobs arrive together (3) Context switch price = 0. **Then:** There exists a non-preemptive algorithm such that:

$$\text{avgResponseTime(non-preemptive)} \leq \text{avgResponseTime(preemptive)}$$

As a result, SJF is also optimal relative to preemptive scheduling (if our assumptions hold).

$$\text{avgResponseTime(SJF)} \leq \text{avgResponseTime(batch or preemptive scheduler)}$$

Connection Between RR & SJF **Assume:** (1) A single core system (2) All jobs arrive together (and only use CPU, no I/O) (3) Quantum is uniform + no context switch overhead. **Then:**

$$\text{avgResponse(RR)} \leq 2 \cdot \text{avgResponse(SJF)}$$

i.e. RR at best is as good as SJF, and at worst is $2X$ slower.

Shortest Remaining Time First (SRTF) If we remove the assumption that all jobs arrive together, then SJF is not optimal anymore, as it may cause a convoy effect. **SRTF** is a preemptive version of SJF, where the scheduler always runs the job with the shortest remaining time. i.e. when a new job arrives or an old one finishes, the scheduler runs the job with the shortest remaining time, even if it means preempting the currently running job. This is optimal in terms of average response time, but it can cause starvation for long jobs.

Selfish RR Selfish RR is a variant of RR where only "old enough" processes are running in the run-queue. i.e.

- New processes wait in a FIFO queue, not yet scheduled.
- Older processes scheduled using RR.
- New processes are scheduled when either of the following happens:
  1. The run-queue is empty (i.e. no ready-to-run "old" processes exist).
  2. "Aging" is being applied to new processes (a per-process counter increases over time); When the counter passes a certain threshold, the "new" process becomes "old" and is transferred to the RR queue.
- **Fast Aging** = Algorithm resembles RR, **Slow Aging** = Algorithm resembles FCFS.

General Purpose Schedulers (Priority-Based & Preemptive)

Scheduling using priority Every process is assigned a priority. The priority reflects how "important" it is in that time instance, and it changes dynamically over time. Process with higher priority are favored, as they are scheduled before lower priority processes. The concept of priority can be also applied in batch scheduling. e.g. SJF:priority = runtime (smaller runtime = higher priority), FCFS:priority = arrival time (earlier arrival = higher priority), etc.

Negative Feedback Principle The schedulers of all general-purpose OSes employ a negative feedback policy: Running reduces priority to run more & Not running increases priority to run. As a result, I/O-bound processes (that seldom use the CPU) get higher priority than CPU-bound processes (that use the CPU a lot). This ensures that I/O-bound processes are responsive.

Multi-Level Priority Queue A multi-level priority queue consists of several RR queues, each associated with a priority. processes migrate between the queues so they have a dynamic priority, "important" processes move up (e.g. I/O-bound) and "unimportant" processes move down (e.g. CPU-bound). Priority is greatly affected by the **negative feedback principle**, i.e. by CPU consumption:

- I/O-bound $\iff$ move up
- CPU-bound $\iff$ move down

Some schedulers allocate short quanta to higher priority queues, and some don't or even do the opposite.

Linux $\leq$ 2.4 Scheduler

All of the definitions below are relative only to the Linux $\leq$ 2.4 scheduler, which is a preemptive priority-based scheduler that uses a multi-level priority queue. **Note:** A task is either a process or a thread.

Standard POSIX Scheduling Policies POSIX dictates that each task is associated with one of three scheduling policies:

- **"Realtime"** Policies:
  1. `SCHED_RR` (round-robin): The time slice is calculated as $(100 * HZ)/1000$ HZ = number of times interrupts per second.
  2. `SCHED_FIFO` (first-in, first-out): doesn't give up the CPU unless: (1) voluntarily using `sched_yeild()`, (2) I/O operation, (3) preempted by a higher priority task.
- The default policy:
  3. `SCHED_OTHER` (the default time-sharing policy)

POSIX defines the meaning of `SCHED_OTHER` (aka `SCHED_NORMAL` in Linux) is decided by the OS. Typically employs some multi-level priority queue with the negative feedback loop.

Realtime tasks are always favored by the scheduler. Only an admin can set a task to run with a *realtime* policy, the users can set a policy using the `sched_setscheduler` syscall. The multi-level priority queue is leveled from 0-99: In kernel - 0 is the highest, In user - 99 is the highest (e.g. syscalls), as kernel calculates (99-input).

Epoch As mentioned before,every runnable task gets allocated a quantum, which is the CPU time the task is allowed to consume before is it's stopped by the OS. Whenever all quanta of all *runnable* tasks become zero a **new epoch** begins. In this case we allocate additional running time to *all tasks* (runnable or not).

Definitions

- **Task's Priority:** Every task is associated with an integer, the higher the value the higher the priority to run. Every task has a *static* priority and a *dynamic* priority.

- **Static Priority:** A fixed value that indirectly determines the maximal quantum for the task. Fixed unless the user invokes `nice()` or `sched_setscheduler` syscalls.

- **Dynamic Priority:** Is the both the remaining time for the task and its current priority. The dynamic priority decreases over time (while running), if it reaches zero the task is preempted until the next epoch. The dynamic priority is *reset* to the static priority at the beginning of each epoch.

- **HZ:** Linux gets a timer interrupt *HZ* times per second, i.e. it gets a timer interrupt every $\frac{1}{HZ}$ seconds. The default value of *HZ* is 100 for x86/Linux2.4.

- **Tick:** A tick can mean either of the following:

  - The time that elapses between two consecutive timer interrupts, i.e. $\frac{1}{HZ}$ seconds.
  - The timer interrupt itself that fires every $\frac{1}{HZ}$ seconds.

  Ticks are used to determine the scheduler timing resolution, the OS measures the passage of time in ticks. The units of the *dynamic priority* are ticks.

- **task_struct:** Every task is represented by a *task_struct* object, which contains (among other things):

  1. nice static priority
  2. counter dynamic priority
  3. processor the last CPU core the task ran on
  4. need_resched boolean
  5. mm task's memory address space

- **task's nice** There are two types of nice, user nice and kernel nice. The **kernel's nice** is the *static priority* of the task, which is between 1...40 (higher is better) and the default is 20. The **user's nice** is the parameter passed to the `nice()` syscall, which is between -20...19 (lower is better). Values below 0 require superuser privileges.

$$\text{kernel\_nice} = 20 - \text{user\_nice}$$

- **task's counter** The *dynamic priority* of the task. It is calculated as follows:
  - Upon task creation:

$$\begin{aligned} \text{child.counter} &= \text{parent.counter}/2; \quad \text{(round down)} \\ \text{parent.counter} &-= \text{child.counter}; \quad \text{(round up)} \end{aligned}$$

  - Upon a new epoch:

$$\begin{aligned} \text{task.counter} &= \text{task.counter}/2 + \text{NICE\_TO\_TICKS(task.nice)} \\ &= \text{half of prev dynamic + convert\_to\_ticks(static)} \end{aligned}$$

  - When running: decrement each tick by 1 (task.counter–) until it reaches 0.

  The **NICE_TO_TICKS** function scales 20 (=DEF_PRIORITY) to number of tick compromising **50+ ms**. By default, scales 20 to 5+ ticks:
  `#define NICE_TO_TICKS(kern_nice) ((kern_nice) / 4 + 1)`
  So the quantum range is therefore: (recall that 1 tick = 10 ms)
  - (1/4 + 1=) 1 tick = 10 ms (min.)
  - (20/4 + 1=) 6 ticks = 60 ms (default)
  - (40/4 + 1=) 11 ticks = 110 ms (max.)
- **task's processor** Logical ID of CPU core upon which task has executed most recently, if task is currently running, then this is the core it is running on.
- **task's need_resched** A boolean flag that is checked by kernel just before switching back to user-mode. If set, check if there's a "better" task than the one currently running, and if so, switch to it. Can be thought of as a per-core rather than per-task flag as it is checked only for the currently running task.
- **task's mm** A pointer to the task's memory address space. (More details in the virtual memory lectures)

The scheduler is implemented in the `kernel/sched.c` file, and the task structure is defined in `include/linux/sched.h`. The scheduler is compromised of **4 main functions**:

1. `goodness(task, cpu)` - Given a task and a CPU core, returns how "desirable" it is for that CPU. We compare tasks by this value to determine which task to run next.
2. `schedule()` - Actual implementation of the scheduler. Uses `goodness` to decide which task to run on a given core.
3. `__wake_up_common(wait_queue q)` - Wakes up task(s) when waited-for event has happened. e.g. completion of I/O operation, or a signal being sent to the task.
4. `reschedule_idle(task t)` - Given a task, check whether it can be scheduled on some core. Preferably on an idle core, but if not then by preempting a less "desirable" task on a busy core. Note: used by the `schedule()` & `__wake_up_common()` functions.

Counter Convergence **Claim:** The counter value of an I/O-bound task will quickly converge to 2α geometric series. **Corollary:** By default, an I/O-bound task will have a counter of 12 ticks (=120 ms) as long as it remains I/O-bound and consumes negligble CPU time.

The Drawback The scheduler is a linear scheduler, i.e. it runs in $O(n)$ time. It was replaced by the "$O(1)$" scheduler which then was replaced by the "*Completely Fair Scheduler*" (CFS) which is $O(\log n)$.

CFS (Completely Fair Scheduler)

CFS The CFS is a preemptive, priority-based scheduler that aims to provide a fair distribution of CPU time among all tasks while maintaining efficiency and scalability ($O(\log N)$).

`vruntime` - Virtual Run Time Every running task obtains virtual run-time. Ideally all tasks maintain an equal amount of virtual run time, and that's the goal. The algorithm stores the minimum & maximum virtual run times at all times. The minimal virtual runtime task is always the next task to run.

`sched_latency` The CFS uses a *sched_latency* parameter to determine the length of an epoch (48 ms). Each task gets a quantum of $Q_i = \text{sched\_latency}/N$, where N is the number of tasks.

The algorithm also defines a minimum quantum length, **`sched_min_granularity`**, which is the smallest time slice a task can receive (6 ms). So in practice, tasks can have a longer epoch than defined by **`sched_latency`**.

CFS Algorithm The CFS uses a red-black tree (similar to AVL tree) to maintain the tasks in the system, ordered by their **`vruntime`**. New tasks are inserted into the tree, with their **`vruntime`** set to the `max_vruntime`. Tasks that come back from I/O operations or are woken up have their **`vruntime`** set to the `min_vruntime`.

CFS Priorities & Weights The CFS uses a priority system based on the *nice* value of the task. The **`vruntime`** is adjusted based on the task's priority, with higher priority tasks gaining lower **`vruntime`**. The lower the nice the higher the priority, and for each nice value, there is a corresponding weight (high weight for high priority)

Let W_0 be the weight of nice=0, then:

$$VR_i = \frac{W_0}{W_i} \cdot \Delta T$$

$$Q_i = \frac{W_i}{\sum_k W_k} \cdot \text{sched\_latency}$$

Topic 6: Context Switch

Caller Rules	Callee Rules
Before the function call: • Save caller-saved registers on the stack (`rax, rcx, rdx, rdi, rsi, r8{r11`). • Pass the first 6 arguments in registers (from left to right: `rdi, rsi, rdx, rcx, r8, r9`). • Pass the rest of the arguments (beyond the first six) on the stack. • Use the `call` machine instruction. **Upon return from the function:** • Restore the saved registers.	**On function entry:** • Create a new stack frame (save the old `rbp` and point to the new top of the stack). • Allocate space for local variables. • Save callee-saved registers on the stack (`rbx, r12{r15`). **On function exit:** • Place the return value in the `rax` register. • Restore the saved registers. • Deallocate local variables. • Return to the old stack frame (restore the previously saved `rbp`). • Use the `ret` machine instruction.

Two types of context switches:

Forced Context Switch (Preemption)	Initiated Context Switch
The kernel preempts (i.e., forcibly takes) the CPU from the process, for example, following: • A timer interrupt (`scheduler_tick`) which reveals that the time slice allocated to the current process has expired. • An asynchronous event that wakes up a process with a higher priority than the currently running process. For example: a disk interrupt or the release of a lock for which a process was waiting.	The process voluntarily gives up the CPU, for example, by means of: • A blocking system call (such as `wait()`, `read()`, ...), which puts the process into a waiting state. • The `exit()` system call, which terminates the process. • The `sched_yield()` system call – a dedicated system call to yield the CPU.

Preemption in Kernel The (new) linux kernel is preemptive, meaning that a process running in the kernel can be preempted!

When switching context we switch the kernel stacks of the tasks (each task has a corresponding kernel stack in the kernel space).

The only gateway in linux for context switches is the function `schedule()`, which can not be interrupted.

Context Switch Saving when a context switch occurs we save:

- The stack, heap, code (memory regions) are in the memory space of the process, so they do not need to be backed up.
- open files and other data are already residing in the PCB, which is in the memory.
- Registers & flags (the CPU state) are backed up to the kernel stack of the thread on a field called thread_struct.
- The base of the kernel stack base is pointed to by TSS.sp0 (the top of the kernel stack) so it needs to be backed up.

TSS (Task State Segment) is a unique data structure for each CPU core that contains the state of the currently running task. It is pointed to by TR, and it contains the TSS.sp0 field which points to the top of the kernel stack of the currently running task.

Each PCB contains a `thread_struct` field which contains the state of the task, including the CPU registers, flags, and most importantly, the head of the kernel stack.

To do a context switch we do the following steps:

1. `context_switch()`: This function calls for switching the memory space, and calls the next function -
2. `__switch_to_asm()`: The macro that saves the current context and loads the next one. More specifically, it backs the registers & the rsp of the kernel and loads the new one, pops the backed registers (of the new one) and then calls the next function -
3. **`__switch_to()`**: This function updates the TSS.sp0 field to point to the new kernel stack, and returns to return address of `__switch_to_asm()` or `ret_from_fork()`.

`do_fork()` is the kernel function used by `fork()` and `clone()` to create the new context of the new thread:

1. It allocates a new PCB for the new thread, and a new kernel stack
2. Calls for `copy_thread()`, which fills the kernel stack of the child process to be as it was the one who called `fork()` and then `__switch_to_asm()`
3. Copies to the child process most of the PCB fields and data from the father.
4. Connects the child process to its family connections.
5. Adds the child process to the global process list. and to the pid hash table
6. Transfers the child task to `TASK_RUNNING` state and adds it to the `runqueue`.
7. The functions returns the pid of the child, and this value is returned from the father too.

`copy_thread()` copies the registers from the parent to the child, sets the `rax` register of the child to 0, and updates the return address that is saved on the stack to the `ret_from_fork()` function, and finally sets the task kernel stack head (rsp) to the new kernel stack of the child process.

`ret_from_fork()` is the function that pops the registers from the kernel stackand returns to the user code (using `sysret`)

`do_exit()` is the function that is called when a process exits (either by calling `exit()` or by signals/interrupts). It does the following:

1. Frees the resources of the task, closes all open files, frees memory, etc.
2. Sets the exit value to the `exit_code` field of the PCB.
3. Updates family ties: all children become orphans (children of init), the parent keeps the pointer to the exiting child, all siblings lose their pointer to the exiting process.
4. Updates the task state to `TASK_ZOMBIE`
5. Calls the function `schedule()` to run a different task. The task runtime is finished by `__switch_to_asm()`.

Topic 7: Synchoronization (& Deadlocks)

Race Condition We say that the program suffers from a *race condition* if the *outcome is nondeterministic and depends on teh timing of uncontrollable, un-synchronized events*

Memory (Cache) Coherency is the mechanism that ensures that any change to a shared piece of data in one cache is eventually propagated to all other caches, preventing processors from working with outdated information.

Cache Consistency it dictates how memory operations (reads and writes) from one processor can be observed by other processors in the system. If store operations are immediately seen by other cores and they can't be reordered with load operations, then consistency holds. (but it's not a necessary condition)

Cache **coherency** relates to a **single** memory location, while cache **consistency** relates to **multiple** memory locations. coherency ensures cores see writes in some order that makes sense, and is uniform across all cores. Consistency is about the order of different read and write operations across multiple memory locations, and how they are observed by different cores.

To enforce ordering for memory consistency, explicit **memory fence** (memory barrier) must be used. The fence makes all store-s that happened before the fence visible to all load-s after the fence, notably, flushes the store buffer to teh memory system.

Atomicy In programming, atomicity means an operation, or a series of operations, appears to happen all at once, without any interruption from other processes. This ensures that either the entire operation completes successfully, or it has no effect at all, preventing partial or incomplete results.

In single core CPUs each instruction is atomic, i.e. it can't be interrupted by another instruction. In multi-core CPUs, atomicity is only achieved by using special instructions or locks that ensure no other thread can access the data while the operation is being performed. (e.g. `lock; instruction`)

Critical Section A group of operations we need to atomically execute is called a **critical section**. Atomicity will make sure other threads don't see partial results! The critical section can be a uniform code across all threads, or it can be a different.

Mutual Exclusion (mutex) is a property of a critical section that ensures that only one thread can execute it at a time and never simultaneously. Thus a critical section is a "serilaization point".

Locks

Lock A lock is an abstraction that supports two operations: `acquire(lock)` and `release(lock)`. Semantically: It's a memory fence, only one thread at a time can acquire the lock and other simultaneous attempts to acquire are postponed until the lock released. Implementing a lock is nowadays done using hardware support with special instructions that ensure mutual exclusion.

Spinlock A spinlock is a lock that uses busy-waiting to acquire the lock. i.e. it repeatedly checks if the lock is available and only then acquires it. Spinlocks are typically used in low-level code where the overhead of sleeping and waking up is too high, or when the critical section is very short. If interrupt or signal handlers access the same data as the critical section, then the we must disable them, or make sure they acquire the lock too!

```
struct spinlock {
    uint locked; // 0 = unlocked, 1 = locked
};

// The volatile keyword is crucial here. It tells the compiler that the value at addr
    can be changed by external factors (like another CPU core or hardware) at any time.
    This prevents the compiler from making optimizations like caching the value in a
    register, forcing it to read directly from memory every time.
inline uint xchg(volatile uint *addr, uint newval) {
    // atomic [oldval = *addr, swap(*addr, newval)]
    // xchg is atomic
    // lock adds a memory fence
    uint oldval;
    asm volatile("lock; xchg %0, %1"
                 : "+m" (*addr), "=r"(oldval):
                 "1"(newval):
                 "cc");
    return oldval;
}

void acquire(struct spinlock *lock) {
    disable_interrupts();
    while (xchg(&lock->locked, 1) != 0) {
        // busy-wait until the lock is available
        enable_interrupts();  // if we want to enable
        disable_interrupts(); // interrupts while spinning
    }
}

void release(struct spinlock *lock) {
    xchg(&lock->locked, 0); // set the lock to unlocked
    enable_interrupts();
```

```
30 }
```

Listing 1: Spinlock Implementation

Other Atomic Operations We know of **3** atomic operations that are used to implement spinlocks:

1. **xchg(addr, newval)** - Atomically exchanges the value at `addr` with `newval` and returns the old value.
2. **test_and_set(bool *b)** - Atomically sets `*b` to *true* and returns the old value of `*b`.
3. **cas(int *p, int old_val, int new_val)** - (Compare-and-Swap) Atomically compares the value at `*p` with `old_val`, and if they are equal, sets `*p` to `new_val` and returns true. Otherwise, it returns false and does not change `*p`.

Mutex Lock

Spinning or Blocking A lock can be implemented using either **spinning** or **blocking**:

- **Spin (busy wait)** - The thread repeatedly checks if the lock is available, consuming CPU cycles while waiting. This is suitable for short critical sections where the wait time is expected to be very short. Note that the user can not block context switches.
- **Block (go to wait/sleep)** - The thread goes to sleep and is woken up when the lock is available. This is suitable for longer critical sections where the wait time can be significant.

Rule of thumb: If the critical section is shorter than the time a context switch takes, then use spinning, otherwise use blocking.
Unlike spinning which can be done by the user, going to sleeping is done by a request to the OS (since it involves changing process states).

Mutex interface the following are the common ops:

- `mutex_init(mutex_t *mutex, const mutex_attr_t *att)`: Initializes the mutex object. The `att` parameter can be used to set attributes.
- `mutex_lock(mutex_t *mutex)`: Acquires the mutex lock. If the lock is already held by another thread, the calling thread will block until the lock is available.
- `mutex_trylock(mutex_t *mutex)`: Attempts to acquire the mutex lock without blocking. If the lock is available, it acquires it and succeeds. Otherwise if the lock is already held by another thread, it fails and continues.
- `mutex_unlock(mutex_t *mutex)`: Releases the mutex lock. If there are threads waiting for the lock, one of them will be woken up.
- `mutex_destroy(mutex_t *mutex)`: Destroys the mutex object and releases any resources associated with it. The mutex must not be locked when this function is called, otherwise it fails.

Undefined Actions In a correct implementation of a mutex, only the thread that holds the lock can call `mutex_unlock()`. The following are undefined behaviors:

- re-locking the mutex by the same thread that holds it.
- unlocking the mutex by a thread that does not hold it.
- unlocking a mutex that is not locked.

Condition Variables A condition variable is a synchronization primitive that allows threads to wait for certain conditions to be met, while inside a critical code section without the need to block the CPU! A condition variable is always associated with a mutex, and it allows threads to wait for a condition to be signaled by another thread. The typical usage pattern is as follows:

- A thread acquires the mutex lock.
- The thread checks the condition. If the condition is not met, it calls `cond_wait(cond_var, mutex)` to release the mutex (it must be held by the thread beforehand) and block until another thread signals the condition variable.
- When the condition is met, another thread can call `cond_signal(cond_var)` to wake up one waiting thread or `cond_broadcast(cond_var)` to wake up all waiting threads.
- The waiting thread must re-acquire the mutex lock before continuing execution. (part of the `cond_wait()` operation)
- The thread should then check the condition again and proceed if it is met.

Semaphores

The Basics A semaphore is usually implemented as a counter and a queue of waiting threads. A task that announces it's waiting for a resource will get the resource if it is available or will go to sleep otherwise. In case it goes to sleep it will be awakened when the resource becomes available to it.

The fields of the semaphore are as follows:

- Value (integer) (the counter):

  If value is Non-negative, it indicates the number of available resources. If value is negative, it indicates the number of tasks waiting for the resource.

- A queue of waiting task:

  Every task that is waiting for the resource to become available is stored in this queue. When the value is negative, |value| = queue.lenght|/

The functions used to manipulate the semaphore are:

- `wait(sem)`: Decrements the value of the semaphore. If the value after decrementing is negative, the task goes to sleep and is added to the queue of waiting tasks. Otherwise, it continues execution.

- `signal(sem)`: Increments the value of the semaphore. If the value was negative before incrementing, it wakes up one task from the queue of waiting tasks.

Semaphore Implementation The semaphore is implemented in the kernel and not part of the course material. However, the following is a demonstration of the problem known as **lost wakeups**.

```
struct semaphore_t {
    int value;
    wait_queue_t wait_queue;
    lock_t lock;
};

void wait(semaphore_t *sem) {
    lock(&sem->lock);
    sem->value--;
    if (sem->value < 0) {
        enque(self, &sem->wait_queue);
        unlock(&sem->lock);             // This is the problem!
        block(self);
    } else
        unlock(&sem->lock);
}

void signal(semaphore_t *sem) {
    lock(&sem->lock);
    sem->value++;
    if (sem->value <= 0) {
        p = dequeue(&sem->wait_queue); // Part of the problem too!
        wakeup(p);
    }
}
```

Listing 2: Lost Wakeups Problem

The problem with the above implementation is that we release the the lock before going to sleep but after joining the queue, so if another task calls `signal()` redbefore we go to sleep, it will wake us up (without being asleep) and we will exit the wait queue so *no one will be*

able to wake us up again. This is known as the **lost wakeups** problem.

Spinlock vs. Semaphore

	spinlock	**semaphore**
wait by	spinning	sleeping
granularity	fine	coarse: might wait for a long time
complexity	lower	greater: typically uses lock
expressiveness	lower	greater: equivalent to lock if maximal value=1 (called "binary semaphore"); otherwise, counting how many resources are available
interface ordering	strict: must acquire before release, must release after acquire	relaxed: may signal(s) without wait(s), may wait(s) without signal(s)
interface dependence	strict: release only invoked by locker	relaxed: signal may be invoked by threads that didn't previously wait and vice versa

Amdahl's Law Amdahl's Law is a formula that gives the maximum improvement to an overall system when only part of the system is improved.

So what is the maximal expected speed when parallelizing?

- Let n be the number of threads.
- Let s be the fraction of the program that is strictly serial ($0 \leq s \leq 1$).
- Let T_n be the time it takes to run the algorithm with n threads.
- Then, optimally:
$$T_n = T_1 \times \left(s + \frac{1-s}{n} \right) \geq T_1 \times sem$$
- The speedup is defined as:
$$speedup = \frac{T_1}{T_n} = \frac{1}{s + \frac{1-s}{n}} \leq \frac{1}{s}$$

Deadlocks & Livelocks

Progress At any given time, at least one thread is able to do the critical section, i.e. No "deadlock" or "livelock" occur.

- **Deadlock** - A situation in which two or more competing actions are each waiting for the other to finish, and thus no one ever finishes. e.g. two threads waiting for each other to release a lock.
- **Livelock** - A situation in which two or more competing actions are each trying to finish, but they keep interfering with each other, and thus no one ever finishes. e.g. two threads trying to acquire a lock, but they keep releasing it and trying again.
- **Starvation** - A situation in which a thread is unable to finish its critical section because it is always preempted by other threads. e.g. a low-priority thread that is never scheduled because high-priority threads keep running.

Deadlock State We say that a system is in a deadlock state, and not a given process. **The popular definition:** A set of processes is deadlocked if each process in the set is waiting for an event that only a process in the set can cause. **Our definition:** A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set (The set can be a subset of the set of all process). **Note:** Deadlocks are tough to deal with since the system is not guaranteed to enter a deadlock if it is prone to deadlocks.

Resource types Resource types can be either physical (e.g. Disk/Disk Block, NIC (network controller), Keyboard, etc.) or logical (e.g. Process Table / node in a tree, etc.).

Some resources only have one instance (e.g. keyboard) and others have multiple instances (e.g. producer-consumer ring).

Therefore, **Locks** are a synchronization construct that reflect a single instance, while **Semaphores** are a synchronization construct that reflect multiple instances of a resource type. **Note:** A binary semaphore is equivalent to a mutex lock.

Resource Allocation Graph When considering resource management, it is convenient to represent the system state with a directed graph. We can represent the system in one of two ways:

- **Multi-Resource Allocation Graph:** A directed graph where: (1) "round" nodes are processes (2) "square" nodes are resources with the dots in them representing the number of instances (3) edges from process to resource represent request (4) edges from resource to process represent allocation.
- **Single Instance Resources (Simplified):** We can simplify the system with a directed graph where: (1) nodes are processes (2) edges mark dependencies between processes, i.e. an edge from process A to process B means that A is waiting for a resource held by B.

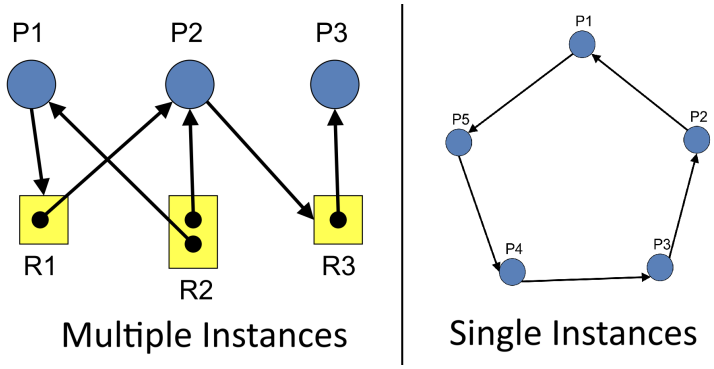


Figure 5: Resource Allocation Graph

Necessary Conditions for Deadlocks All of the following must hold for a deadlock to occur:

1. **Mutual Exclusion:**

   At least one resource must be held in a non-shareable mode, i.e. only one process at a time can use the resource. if every resource is shareable or used by a single process, then no deadlock can occur.

2. **Hold & Wait:**

   Processes holding resources are allowed to request additional resources without releasing the resources they already hold. If processes must release all resources before requesting new ones or atommically request all resources, then no deadlock can occur.

3. **Circular Wait:**

   There must be a circular chain of two or more processes, each waiting for a resource held by the next process in the chain. i.e. $P(i)$ wait for resource held by $P((i+1)\%n)$. Otherwise there exists at least one process that doesn't need to wait

4. **No Resource Preemption:**

   Resources cannot be forcibly taken from a process holding them, they must be voluntarily released. If resources held can be released after some period of time, then no deadlock can occur.

Dealing with Deadlocks There are **3** main strategies to deal with deadlocks:

- **Deadlock Prevention:** Violate one of the 4 conditions necessary for deadlock to occur.
- **Deadlock Detection & Recovery:** Allow the system to enter deadlock state, but put in place mechanisms that can detect, and then recover from this situation. Typically refers to killing one or more processes in the deadlock state, using the algorithmic resource-graph problem.

- **Deadlock Avoidance:** Use a resource allocation graph to detect potential deadlocks and avoid them by not allocating resources that would lead to a deadlock. (e.g. Banker's Algorithm). Requires full knowledge of the system's resource requirements and current allocation.

Deadlock Prevention There are 4 conditions for deadlock to occur, and we can prevent deadlocks by violating one of them:

1. **No Mutual Exclusion:**

   Making all resources sharable, by implementing some data structure using only hardware-supported atomic operations. These algorithms are called "lock free".

2. **No Hold & Wait:**

   Each process must request all resources it needs at once, or release all resources before requesting new ones. The system either grants all resources or none and block the process. **Con:** Processes will hold on to their resources for more time than they actually need them, which limits concurrency and hence limits performance.

3. **No Circular Wait:** the most usable flexible method

   Impose a strict ordering on resources, i.e. number all resources in one sequence. Processes can only request resources in increasing order, a process that wants to acquire a resource that has a lower order must first release all resources it holds that have a higher order.

4. **No "No Resource Preemption":**

   When needed, choose a victim process and release its resources, allowing other processes to continue.

   e.g. if there isn't enough memory, we can backup the victim's state to disk and release all its memory, or just kill the process.

Deadlock Detection & Recovery If there is only one instance of each resource type, then a deadlock can be detected by checking for cycles in the **simplified resource allocation graph**. In the general case (multiple instances of each resource type), Necessary conditions for deadlock are not sufficient, nevertheless, we can still detect deadlocks but it is *outside the scope of this course.*

After a deadlock is detected, we can recover from it by: **(1)** Killing one or more processes in the deadlock state until the deadlock is resolved, or **(2)** Forcibly preempting resources from one or more processes in the deadlock state. Finding the minimal set of processes to kill/preempt is a hard problem.

In literature, Detection and Recovery refer to the algorithmic problem of finding the minimal set of processes to kill/preempt in the resource graph.

Deadlock Avoidance The system can avoid deadlocks by not allocating resources that would lead to a deadlock. This is achieved by the **Banker's Algorithm**, which is a resource allocation and deadlock avoidance algorithm that tests for safety before allocating resources. The algorithm maintains a *safe state* and only allocates resources if it can guarantee that the system will remain in a safe state after the allocation. The algorithm requires knowledge of the maximum resource requirements of each process and the current allocation of resources.

Safe State: A system state where we are sure that all processes can be executed **in some order**, one after the other, such that each one will obtain all the resources it needs to finish. We assume once a process receives all the resources it needs, it will finish and release all its resources.

Rules of the Banker's Algorithm The Banker's Algorithm works as follows:

- There are n processes and k resource types (each resource can have multiple instances).
- Upon initialization, each process p declares the maximal number of resource-instances it will need for each type, $\max[p]$.
- While running, OS maintains how many resources are currently allocated to each process, $cur[p]$.
- And how many resource instances are currently available (free), $avail$.
- Upon a request for resources by a process p, R, the OS allocates iff the system will stay in a safe-state after the allocation. If not, the process is blocked until the resources are available.

Note: The Banker's Algorithm is conservative, i.e. it will not allocate resources even if there is possibility to allocate them without deadlocking the system. The OS doesn't take chances! With Time Complexity $O(n^2)$.

```
// *Assume* that request R was granted to process q
cur[q] += R;                              // vector addition
avail -= R;
// Check if the system is in a safe state (i.e. all processes can finish in some order)
P = ... // All non-terminated processes (processes that are not finished yet)
while(P is *not* empty){
    found = false                         // true = found a process that can finish
    for each p in P {                     // find a process that can finish
        if (max[p] - cur[p] <= avail) { // can finish? (by maxing its resources)
            avail += cur[p]               // release resources (pretend p finishes)
            P.remove(p)
            found = true
        }
    }
    if (!found) return FAILURE
}
return SUCCESS
```

Listing 3: Banker's Algorithm

Topic 8: Networking

Protocol A protocol is a set of rules defining the *format & order* of messages sent & received, and what actions are taken upon message transmission or receipt.

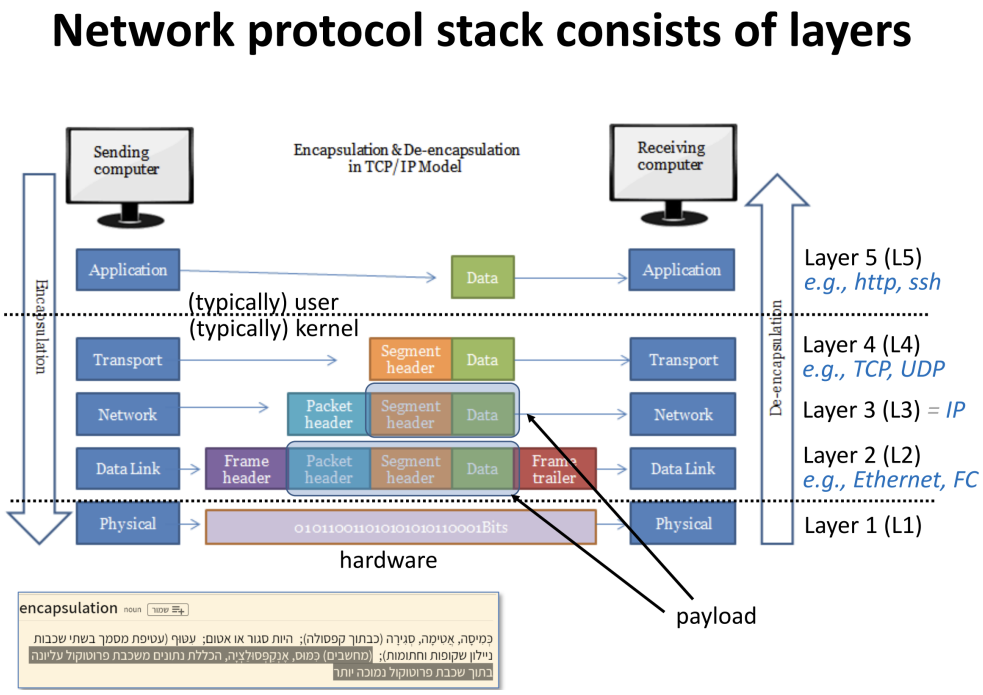


Figure 6: Network Layers Overview

In a nutshell, networks can be divided into 5 layers as follows:

- **Application Layer - L5:** The top layer, where applications communicate with each other. Examples include HTTP, FTP, SMTP, etc.
- **Transport Layer - L4:** Responsible for end-to-end communication between applications. It provides reliable or unreliable delivery of data. Examples include TCP (reliable) and UDP (unreliable).
- **Network Layer - L3:** Responsible for routing data between devices on different networks. It provides logical addressing and routing. Examples include IP (Internet Protocol).

- **Data Link Layer - L2:** Responsible for node-to-node communication within the same network. It provides physical addressing and error detection. Examples include Ethernet, Wi-Fi, etc.
- **Physical Layer - L1:** The lowest layer, responsible for the physical transmission of data over a medium. It includes the hardware components such as cables, switches, and network interface cards (NICs).

Application Layer (L5)

The Application Layer is the topmost layer in the network architecture. It provides network services directly to the user's applications. This layer is responsible for facilitating communication between software applications and lower layers of the network stack.

The Application Layer interacts with the Transport Layer (L4) to establish connections and manage data flow between applications on different devices. One can easily invent a new protocol as needed.

There are **Standard** protocols (determined by multiple organizations): HTTP, HTTPS, NFS, SSH, Bitcoin... And **Proprietary** protocols (determined by a single organization): Microsoft Exchange, Skype, Zoom, Whatsapp...

Problems In Networking - There are 2 main problems to consider when discussing networks:

- **Lossiness:** Networks are inherently lossy, meaning that data sent won't always arrive at the destination. This can be due to various reasons such as hardware failures... But more commonly, it is due to running out of *memory buffer space* in network elements.
- **Reordering:** Networks may mess up the order of the messages sent.

Transport Layer (L4)

Protocols in Layer 4 provide host-to-host communication services *for processes*, they are implemented (typically) by the OS. Two most common protocols are: **TCP** (Transmission Control Protocol) and **UDP** (User Datagram Protocol).

TCP "The protocol that cares", i.e. it cares about data loss & reordering. It ensures that data is delivered reliably and in the correct order. Provides **stream** of bytes between two processes. Said to be **connection-oriented** because it establishes a connection before transmitting data. Simplified view of TCP:

- **Sender**

1. Establish a connection with the receiver by performing a *3-way handshake*.
2. Split bytes into contiguous chunks called *segments*.
3. Assign a sequence number to each segment, and keep them around until they are acknowledged by the receiver.
4. Resend segments that are not acknowledged for some time, or receiver says they are missing.
5. Slow down the sending rate if segments sent are not reaching the receiver.
6. Implement **Congestion Control** = changing the rate in response to drops.

- **Receiver**

1. Send ack messages (acknowledgments) upon receiving chunks.
2. Ask sender to resend chunks that appear to be missing for some time.
3. "Advertise" free buffer space to the sender, so it knows how much data it can send without overflowing the receiver's buffer.
4. Implement **Flow Control** (advertising) = Receiver controls sender transmission rate, thereby preventing its own buffer overflow.
5. Deliver the bytes in-order to receiving user-level app.

UDP "The protocol that doesn't care", i.e. it does not guarantee reliable delivery nor in-order delivery of data. Provides **datagrams** (as opposed to a stream) of bytes between two processes. Chunks might get lost or be delivered out of order, but per-chunk bytes integrity is supported. Said to be **connectionless** because it does not establish a connection before transmitting data. Compared to TCP, UDP is simpler and has less overhead, lower latency, and no congestion control.

Sockets Apps communicate with the Transport Layer through **socket file descriptors**, created via the `socket()` system call, instead of `open()`. A `sockfd` constitutes a communication endpoint.

Note: It is recommended to understand what IPs are before continuing this section (read next section - IP)

Ports Ports exist on the same host, and there can be a multiple of them. Each communicating process utilizes `sockfd`s to identify its own communication channel, each with its own associated **port**. As IPs are not sufficient, we use a unique port for each socket, which is an unsigned 16-bit integer that identifies it on the host. Each chunk transmitted is associated with IP-address + port.

Ports can either be **Ephemeral** (temporary, dynamically assigned by the kernel), or **Well-Known** (static, determined by standards/known values). Ports ≤ 1023 are reserved for well-known protocols, such as HTTP (80), HTTPS (443), etc.

Each socket is identified by a **5-tuple** consisting of:

1. Source IP address

2. Source port number
3. Destination IP address
4. Destination port number
5. Transport protocol (e.g. TCP, UDP)

Client-Server Model (TCP)

In the client-server model, a server listens for incoming connections from clients, and clients initiate connections to the server. The **server** passively waits for client requests and then reacts, "request-response" paradigm. The server has a well-known domain name (and/or IP address) & port. The **client** asynchronously/intermittently connects to the server, sends requests, and waits for responses. The client may have dynamic IP, and may use ephemeral port.

Sockets are initialized using values from `addrinfo` structure, which contains the following info: (1) Protocol family (e.g. IPv4, IPv6), (2) Protocol type (TCP), (3) Socket type (stream), (4) server/client.

Creating and Using TCP Sockets

server	message	client
`srvfd = socket(...)`		`sockfd = socket(...)`
`bind( srvfd , port )`		
`listen( srvfd )`		
loop:		
`clifd = accept( srvfd )`	request service (transport) ←	`connect( sockfd , host+port )`
`read(clifd )`	request (application) ←	`write( sockfd )`
`write( clifd )`	response (application) →	`read( sockfd )`

Network Layer (L3) - IP

The Network Layer is responsible for routing data between devices on different networks. It provides logical addressing and routing. The most common protocol in this layer is the Internet Protocol (IP).

Domain & Host Names Computers are associated with hierarchical human-readable "domain" names, sometimes referred to as host names, e.g. *www.google.com*. Each domain names can be backed by one or more host machines, but we will use the term host name assuming it corresponds to a single host machine.

IP Address Domain names are only for humans to read, but computers need to know the **IP address** of the host they want to communicate with. An IP address is a unique identifier assigned to each device connected to a network. Each host is mapped to a 32-bit IP address obtained via a domain name resolution protocol called **DNS** (Domain Name System). "IP" for us stands for IPv4 (Internet Protocol version 4), but there is also IPv6 which is a 128-bit address space which we will ignore.

Each IP address corresponds to a single host, unlike domain names. Domain names can point to multiple IP addresses (e.g. load balancing), while each IP address is unique to a specific host. The opposite is also not true, i.e. one host can have multiple IP addresses. For simplicity we will assume there's a 1-to-1 host-to-IP mapping.

Note: IP is used to identify host machines, while ports are used to identify individual communication channels of processes on the host.

For example, the IP of `csa.cs.technion.ac.il` is

Notation	**IP address**
Decimal	2219057153 (not terribly convenient)
Binary (8x4 = 32 bits)	$\overset{132}{\texttt{10000100}}$ | $\overset{68}{\texttt{01000100}}$ | $\overset{32}{\texttt{00100000}}$ | $\overset{1}{\texttt{00000001}}$
Dot-decimal	132.68.32.1 (a bit more convenient)

Physical-Layer (L1) & Link Layer (L2)

LAN (Local Area Network) A LAN is a network that connects devices within a limited geographical area, such as a home, office, or campus. It typically uses Ethernet or Wi-Fi technologies to connect devices. Each LAN device is equipped with **Network Interface Card (NIC)**, which is a hardware component that allows the device to connect to the network.

We will focus on Ethernet, which is the most common LAN technology.

Ethernet Ethernet is a protocol that combines hardware, firmware, and software (OS). Ethernet is both a link-layer protocol (L2) as it allows nodes to communicate within the LAN by sending **frames** (byte chunks in layer L2), and a physical-layer protocol (L1) as it defines raw bits are transmitted over the physical medium.

MAC (Media Access Control) Address Each device on a LAN has a unique identifier called a **MAC address**. It is a 48-bit address that is assigned to the NIC by the manufacturer. The MAC address is used to identify devices on the same LAN and is used by Ethernet to deliver frames to the correct destination. The 48-bits are usually represented as 6 hexadecimal numbers separated by colons, e.g. `00:1A:2B:3C:4D:5E`, where the first 3 bytes are the device's manufacturer ID and the last 3 bytes are the device's unique identifier.

Ethernet Frame Structure An Ethernet frame is the basic unit of data transmission in Ethernet networks. It consists of several fields that provide information about the frame and its contents. The structure of an Ethernet frame is as follows:

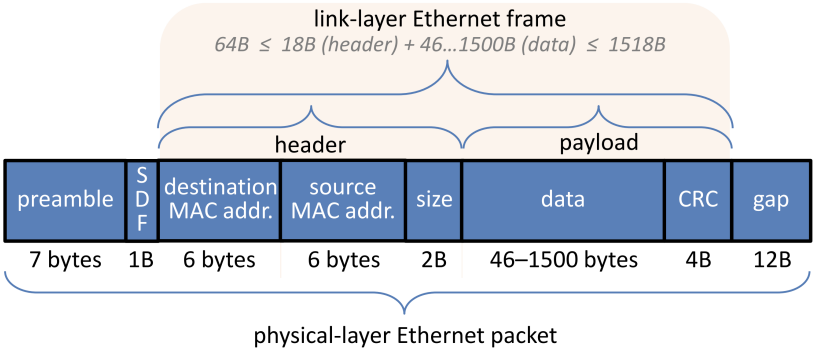
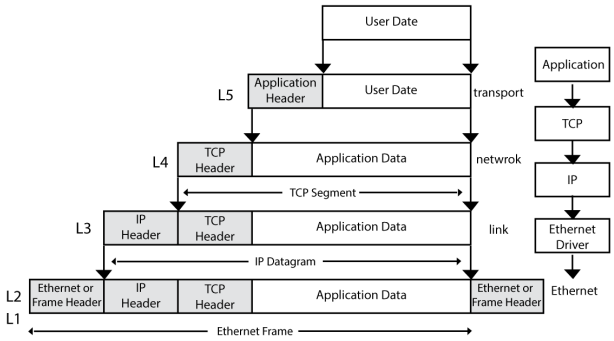


Figure 7: Diagram of an Ethernet Frame Structure.

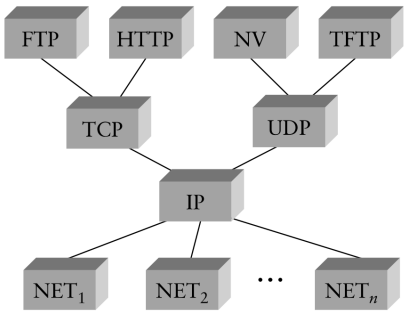
1. **Preamble:** Its purpose is purely for timing and synchronization.
2. **Start Frame Delimiter (SFD):** The actual frame starts with the very next bit.
3. **Destination/Source MAC Address:** This is the MAC of the NIC that the frame is *intended for* (Destination) and the MAC of the NIC that *sent* the frame (Source).
4. **Size:** Represents the length of the frame.
5. **data:** Contains the actual data from the upper layers, such as an IP packet from the Network Layer (L3).
6. **Cyclic Redundancy Check (CRC):** It is an error-detection code. i.e. the frame is dropped in case of a CRC error.
7. **Gap:** After a frame is sent, devices on the network must wait for a brief period before any device is allowed to send the next frame. This gives device interfaces time to process the frame they just received and prepare for the next one.

The Internet

IP & Routers The problem for now is LAN hardware communicate through L2, meaning two different LANs cannot communicate with each other. The solution is to use **routers**, which are devices that connect different networks and **forward** packets between them. Thus the internet is a inter-network of networks, where each inter-network is connected by routers. This is made possible using the IP (L3) protocol, which provides a way to route packets between different networks. The routers "peel off" the L2 headers of the source LAN and re-encapsulate the data within L2-headers of the destination LAN.



(a) The Internet Layers and Encapsulation.



(b) Hourglass Model of the Internet.

Hourglass Model The hourglass model is a conceptual framework that illustrates how the Internet is structured. It shows that the Internet has a narrow waist, which is the IP protocol, and that all other protocols (e.g. TCP, UDP, HTTP) are built on top of it. The model emphasizes the importance of IP as the foundation of the Internet, allowing different protocols to coexist and communicate with each other.

IP address structure An IP consists of two parts: **Subnet Part** (high order bits) and **Host Part** (low order bits). The subnet part identifies the network, while the host part identifies the specific device within that network. The subnet part is determined by the **subnet mask**, which is a 32-bit number that specifies how many bits are used for the subnet part and how many bits are used for the host part. An IP address is always coupled with a subnet mask; e.g. `192.168.1.1/24` means that the first 24 bits are the subnet part and the last 8 bits are the host part. **Meaning of subnet** is that all devices in the same subnet can communicate with each other directly, without going through a router. i.e. it defines the boundaries of the Ethernet LAN.

ARP (Address Resolution Protocol): IP $\Longrightarrow$ MAC
ARP is a protocol used to map IP addresses to MAC addresses. When a device wants to communicate with another device on the same local network, it needs to know the MAC address of the destination device. If it only knows the IP address, it broadcasts an ARP query packet (containing the destination IP and -1 as the MAC) to the local network, asking "Who has IP address X.X.X.X? Tell me your MAC address." The device with the matching IP address responds with its MAC address, allowing the sender to encapsulate the data in a frame and send it to the correct destination. If the destination is not on the same local network, the sender will get the MAC to **its default gateway (router)**, its first-hop router, which will then forward the packet to the correct destination network.

NAT (Network Address Translation): private IPs
NAT is a technique used to allow multiple devices on a local network to share a single public IP address. NAT defines new private IP addresses for each device on the local network, allowing them to communicate with each other using these private addresses. When a device wants to communicate with the outside world, NAT translates its private IP address

to the public IP address, allowing the device to access the internet. This helps conserve the limited pool of public IP addresses and provides an additional layer of security by hiding the internal network structure from external entities. To do so, NAT maintains a mapping between pairs of (private IP addresses + port) and (public IP address + port).

DHCP (Dynamic Host Configuration Protocol): get an IP
DHCP is an L5 protocol used to dynamically assign IP addresses to devices on a network. The goal of DHCP server is to allow client's host to dynamically get IP address when joining LAN, which can provide: (1) subnet mask, (2) Default gateway (first hop router), (3) Name + IP of DNS server for client. DHCP uses **UDP** to communicate allowing the ability to broadcast to all hosts in LAN. When a device connects to a network, it sends a DHCP request to the DHCP server, which responds with an available IP address and other configuration information.

Topic 9: Virtual Memory

NOTE: ALWAYS say if you are talking about **physical memory** (RAM) or **virtual memory** in your answers.

Virtual Memory Concepts

Virtual Memory Virtual memory is the abstraction of physical memory, allowing processes to "think" they have access to the entire memory space, even if the physical memory is limited. The motivation for virtual memory (vmem):

- **Per-Program Contiguous Memory Illusion:** Each process has its own virtual address space, which is contiguous simplifying memory management and allowing processes to use the same address space without conflicts.
- **Isolation:** Virtual memory provides isolation between processes, ensuring that one process cannot access the memory space of another process.
- **Dynamic Growth:** Virtual memory allows processes to grow their heap/stack (and `mmap` regions) as needed.
- **Illusion of Large Memory:** Virtual memory allows processes to use more memory than is physically available. Allowing for memory overcommitment.
- **Access Control** Deciding if individual memory chunks (pages) can be read/written/executed.

Terminology **Virtual Address (VA)** is the address used by a process (& programmer) to access its own memory space, the CPU processes VAs. **Physical Address (PA)** is the actual address in RAM where data is stored. **Page** = chunk of contiguous data (in virutal or physical space), **Frame** = physical memory exactly big enough to hold one page. $|\text{Page}| = |\text{Frame}| = 2^k$, typically 4KB (k=12).

The basic idea The OS maps virtual addresses to physical addresses using a **page table**, such that VA spaces are contiguous. Pages can be mapped to: (1) memory, (2) disk, (3) unallocated yet (allocated on-demand).all programs are written using VAs, OS sets the VA$\Longrightarrow$PA mapping (governs control plane), HW does on-the-fly translation from VA to PA (governs data plane).

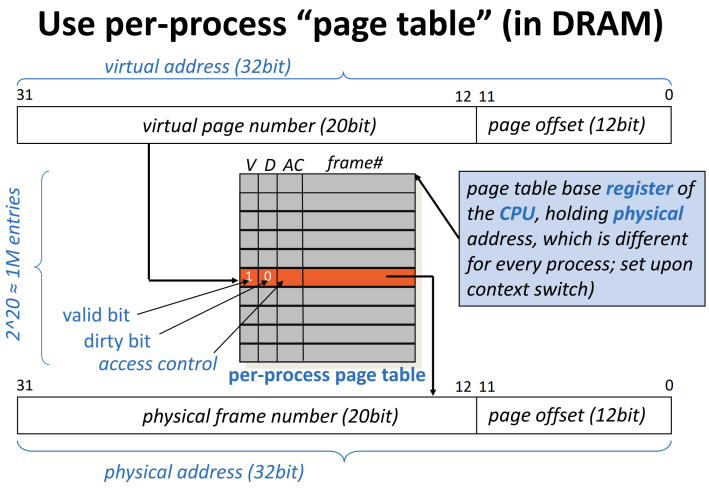


Figure 9: Virtual Memory Page Table

The basic premise of **page tables** is that each process has its own page table, which maps virtual pages to physical frames. Using the virtual page number (VPN) we check the page table entry (PTE) associated with it to find the physical frame number (PFN) it is mapped to, in addition to other flags. One important flag is the **valid bit**, which indicates whether the page is currently mapped to a physical frame or not. If the valid bit is set, the page is mapped to a physical frame, and the OS can access it. If the valid bit is not set, then if all other bits are zero it means the page has not been allocated yet. Otherwise it means it resides on the disk in the written address.

Memory Access steps the following is the basis of how the CPU accesses memory:

```
if (valid == 1)                          // Otherwise HW raises a page fault interrupt
    page is in main memory @ PA stored in table \(\Longrightarrow\) data can be used
else if (page resides on disk)           // "Major" page fault
    OS suspends process
    Fetches page from disk and adds VA\(\Longrightarrow\)PA mapping to page table
    Resumes process, which will re-execute faulting instruction
else if (should be but still is not allocated ||
         in page cache but not in page table) // "Minor" page fault
    OS allocates page if needed & adds VA\(\Longrightarrow\)PA mapping to page table
else if (some other legal situations for which OS is responsible)
    Resolve (example: CoW; see later)
else                                 // process performed an illegal memory ref
    OS (which handles the page fault interrupts) delivers a signal
    (SIGSEGV/SIGBUS) to the offending process
    The matching signal handler terminates process by default
```

Listing 4: Memory Access Steps in Virtual Memory

If access type is incompatible with specified access rights, then a protection violation fault happens raising an interrupt, which the OS resolves if it is its fault, or delivers a signal to the offending process.

Major Page Fault Flow Major = need to fetch page from disk.

1. **CPU detects** the situation (valid bit = 0)
2. **CPU generates interrupt** and transfers control to the OS
3. **OS regains control**, realizes page is on drive, initiates I/O read ops
4. **OS suspends the process** & context switches to another process
5. **Upon read completion**, OS makes the process runnable again
6. When the **process is scheduled** again, it re-executes the faulting instruction (which will succeed this time)

Minor Page Fault Minor = NO need to go to drive to resolve. Examples: (1) **CoW (Copy on Write)** which is used to implement fork for example, (2) **Demand-based (Lazy) memory allocation**, (3) reading a file content that already is found in the **page cache**, possibly by other processes.

Page in & out a chunk of data: **Paging** is the process of swapping pages in and out of physical memory to disk storage.

$$\text{Page in} \iff \text{Copy page from disk to DRAM (read)}$$

$$\text{Page out} \iff \text{Copy page from DRAM to disk (write)}$$

mmap syscall The `mmap` syscall is used in one of two ways: (1) map a given file to a given virtual address range in the process's memory space. (2) allocate a new annonymous memory region in the process's memory space.

1. **Named Pages:** When mapping a file, the OS maps the file's as if it was an "array of bytes" (backed by the disk) in the VA space of the process. Reads/writes to the mapped region will read/write the file's content, and the OS will update the file accordingly every few seconds, in batches. To force the OS to write the changes to disk, the process can use `fsync(fd)` syscall. This helps making programming easier, and saves on the overhead of syscalls to `read`/`write` (as reads/writes to the `mmap` are equivalent to memory reading/writing), which in turn saves on I/O copying.
2. **Anonymous Pages:** are heap/stack pages, or pages of `mmap` with `fd=-1`, `flag=MAP_ANONYMOUS` which are not backed by any file/disk. i.e. we can implement `malloc` using `mmap` (for heap we use sbrk syscall).

We also divide the `mmap` pages into two types: (1) **MAP_SHARED** = changes affect the actual file (& visible to other processes), (2) **MAP_PRIVATE** = changes don't affect the actual fiile; rather, there will be a CoW for the process (other processes will not see changes)

Swapping **Swap Space** is a disk area where anonymous pages are written, if the OS decided they have no room in DRAM (memory is full). Pages are said to be "swapped out" when this occurs (and "swapped in" on return). Nowadays we don't distinguish between the terms swapping and paging.

On-demand Paging A memory management technique where pages are loaded into memory only when they are needed, rather than preloading all pages at once. i.e. pages are allocated/brought into DRAM only when the process attempts to access it for the first time (resulting in a page fault)

Readahead Prefetching When `read()` identifies sequential accesses to file, the `read()` reads ahead pages not yet requested, prefetching them, so they would be ready, in an attempt to minimize page faults. (same with the on-demand paging page fault handler)

TLB (Translation Lookaside Buffer) The TLB is a cache used by the CPU to reduce the time taken to access memory locations. It stores recent translations of virtual memory addresses to physical memory addresses, allowing the CPU to quickly find the physical address corresponding to a virtual address without having to go through the entire page table. When a virtual address is accessed, the CPU first checks the TLB; if the translation is found (a TLB hit), the physical address is retrieved quickly. If not (a TLB miss), the CPU must consult the page table, which is slower. TLBs are typically small and fast, and they use various replacement policies to manage the limited space.

Typically, HW fills TLB automatically by reading the page table on its own (without SW help), and OS invalidates TLB entries (upon context switches, for non-global entries).

Page Replacement Algorithms using an intelligent page replacement policy helps with virtual memory performance:

- **Belady - Optimal** - (theoretical) Replace the page that will not be used for the longest period of time in the future.
- **LRU (least recently used)** - (impractical) Replace the page that has not been used for the longest period of time in the past.
- **Clock** - (practical LRU) A circular list of pages in memory, where each page has a reference bit. When a page is accessed, its reference bit is set to 1. When a page needs to be replaced, the algorithm checks the pages in a circular manner, clearing the reference bit of each page until it finds a page with a reference bit of 0, which is then replaced.
- **NRU (not recently used)** - (practical more sophisticated LRU) Keep to bits for each page, dirty & referenced, and periodically turn off referenced (as in clock). Replace victim page with the lowest class from the following: (class 0) not referenced, not dirty, (class 1) not referenced, dirty, (class 2) referenced, not dirty, (class 4) referenced, dirty. Can enhance further by keeping a counter (`active`) and incrementing periodically for

pages that are referenced, and decrementing periodically. Instead of referenced we will split to active and not active (two lists) by a threshold.

Virtual Memory Implementation

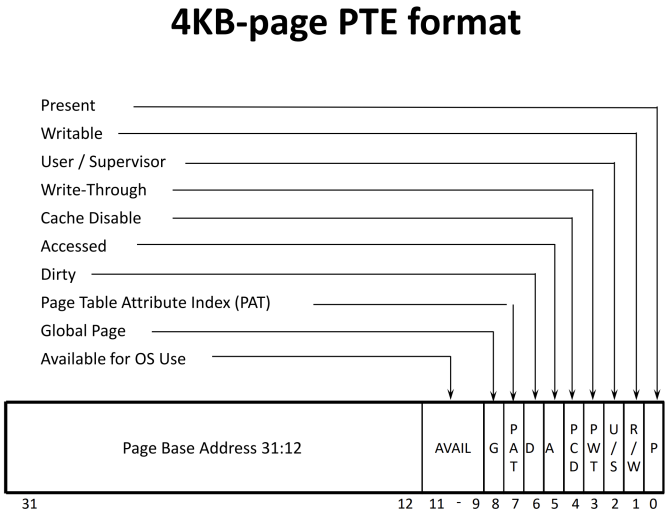
Intel x86 Architecture

32-bit x86 address translations 32-bit address means 2^{32} virtual addresses, which is 4GB of virtual memory space. The virtual address space is divided into pages, typically $4KB$ in size. The page table maps these virtual pages to physical frames in memory.

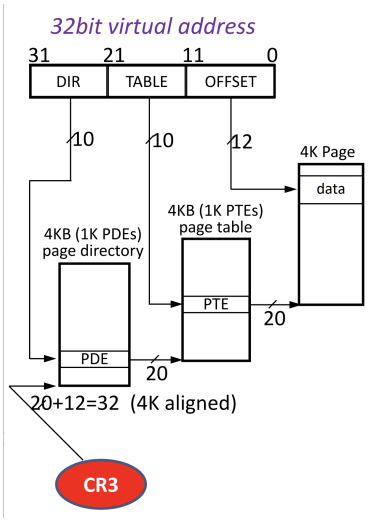
The job of the x86 virtual memory subsystem is to translate 20-bits (=virtual page number - VPN) to 20-bits (= physical frame - PFN). Every process holds a **Page Directory** (= 4KB page) holding 1024 PDEs (page directory entries) each containing a pointer to a **Page Table** (= 4KB page) holding 1024 PTEs (page table entries) each containing a point to a **Page Frame** (= 4KB page). The offset is the last 12 bits of the virtual address, which is used (as is) to access the data in the page frame (same offset).

Each PDE/PTE holds additional flags, such as **P**resent, **A**ccessed, **D**irty, **U**ser, **W**riteable, Cache **D**isabled, etc.

CR3 Register The CR3 register holds the physical address of the page directory for the currently running process. When a context switch occurs, the operating system updates the CR3 register to point to the new process's page directory. (Or more generally, the CR3 register holds the physical address of the first layer of page tables).



(a) Page Table Entry Format for x86.



(b) 32-bit Page Tables

Global + User/Supervisor bits The "user/supervisor" bit is used to indicate that only the OS (ring 0) can access the page, while user processes (ring 3) cannot. The "global" bit

is used to indicate that the page is shared across all processes and should not be flushed from the TLB on context switches. This is used to implement shared kernel space across all memory spaces.

HW/OS Cooperation HW defines the data structures (hierarchy, PTE bits, etc.). While the OS determines most of the content of page directories & page tables (setting the values & mappings). The HW sets the bits of dirty and accessed, and does the table walk automatically:

1. If VA$\Longrightarrow$PA translation not found in TLB, "TLB miss"
2. HW knows where to find the root page directory (CR3 register)
3. HW walks the tables, hierarchially, until it reaches the data page
4. It inserts the VA$\Longrightarrow$PA mapping into the TLB

The HW populates the TLB, and the OS invalidates TLB entries when needed (e.g. on context switches, or when a page is evicted from memory). The OS is also responsible for Synchronizing between TLBs of different cores (TLB shootdown).

64-bit x86 address translations

64-bit x86 address translations 64-bit address means 2^{64} physical addresses, which is 16 Exabytes of physical memory space. The virtual address is only 48-bits, which is 2^{48} virtual addresses, which is 256TB of virtual memory space. The page size is still $4KB$, but the PTE is now 8 bytes (64 bits) instead of 4 bytes (32 bits). The page table structure is similar to the 32-bit x86 architecture, but with additional levels of indirection.

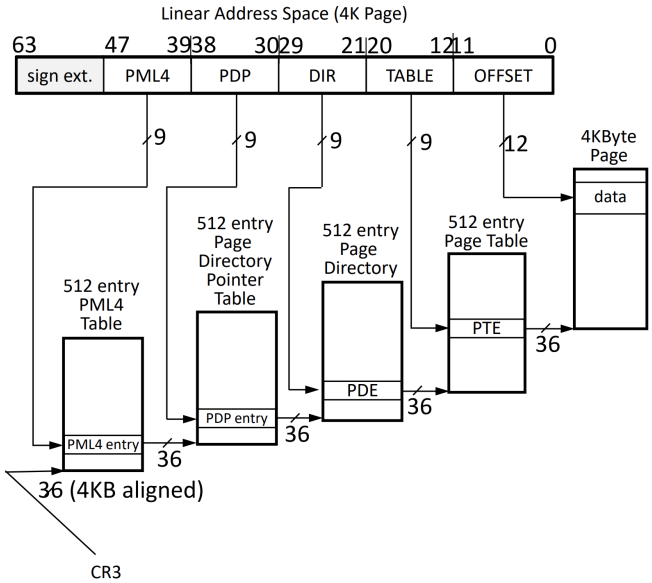


Figure 11: 64-bit Page Tables

Increased TLB convergence To make the TLB more useful we use two concepts:

- TLB hierarchy: We use multiple TLB caches (like the L1, L2, ...) each bigger but slower than the last. Usually we use only two levels.
- **Superpages/Hugepages:** Different sizes are supported by different architectures;
  - 32-bit x86: 4KB, 4MB pages
  - 64-bit x86: 4KB, 2MB, 1GB pages

  We merge the last N levels of page table bits in the VA into one offset of the superpage, allowing us to have fewer page table levels but larger pages.

64-BIT Power-PC (PPC) Virtual Memory

A completely different architecture, but similar concepts, used in IBM PowerPC processors.

3 Address Types effective *Longrightarrow* virutal *Longrightarrow* real

- **Effective:** Each process uses 64-bit effective addresses, which are not unique across processes. The equivalent of virtual addresses in x86.
- **Virtual:** A huge 80-bit address space, which is unique and shared by all processes. If two processes have the same effective address, they will can different virtual addresses, but if they have the same virtual address, they will have the same real address.
- **Real:** The actual physical address in memory, which is 62-bit addresses.

PPC Segments Effective & virtual spaces are partitioned into **segments**, which are 256MB in size. Each segment is contiguous in the per-process effective address space, and each segment is contiguous in the virtual address space.

PPC SLB (Segment Lookaside Buffer) The SLB is a cache used to speed up the translation of effective addresses to virtual addresses. OS manages the SLB, and HW searches the SLB for the effective address. Upon a miss, the OS is notified, and it will insert right segment table entry (STE) to SLB. Upon context switch, the OS invalidates the SLB entries.

PPC TLB (Translation Lookaside Buffer) The TLB is a cache used to speed up the translation of virtual addresses to real addresses. The TLB is managed by both OS & HW. The virtual addresses are shared so no need to invalidate the TLB upon context switch.

PPC ERAT (Effective-to-Real Address Translation) The ERAT is a cache used to speed up the translation of effective addresses to real addresses. Analogous to the TLB (in x86).

PPC HTAB (Hash Table) The HTAB is a hash table used to store the mappings between virtual addresses and physical addresses. Each PTE contains both PPN & VPN, and the OS uses a hash function to map the VPN to a PPN. PTEs are divided into **PTEG** (Page Table Entry Groups), each containing 16 PTEs. Each PTE can either reside in a "primary" or a "secondary" PTEG. If a PTE is missing, a page-fault interrupt is raised, and the OS will place the PTE in either the primary or secondary PTEG.

Topic 10: Interrupts

Interrupts & Exceptions Our intterupts are Intel's interrupts and exceptions, such that: exceptions = internal (synchronous) implicit interrupts, and interrupts = all the rest. There are two sources for intterupts:

1. **External / Asynchronous** - Triggered by devices external to the core's compute circuitry. Assynchronous because they can occur at any time.
2. **Internal / Synchrounous** - Triggered by the core's compute ciruitry (CPU intructions). Sometimes called "Software interrupts" as they are triggered by the CPU executing an instruction.

Internal Interrupts Internal interrupts can be further divided into:

- **Explicit** - Triggered by the CPU executing an instruction that deliberately causes an interrupt, such as `int` instruction, or `syscall` instruction, or `Ptrace`-ing.
- **Implicit** - Triggered by the CPU executing an instruction that causes an error or exception, such as division by zero, invalid memory access, or illegal instruction. i.e. the CPU can't complete the instruction so it transfers control to the OS. Two types of exceptions:
  - **Error condition** tells the OS that the app did something illegal.
  - **Temporary problem** such as a page fault, which can be resolved by the OS.

Interrupt Handling Each interrupt is associated with a Number (called interrupt vector), which is used to identify the interrupt handler. Handlers are pointed to by and `array[256]` of pointers-to-functions. The interrupt handler saves the state of the current running procesess and then executes the handler.

Context Type If the interrupt is asynchronous then the kernel is said to be in interrupt context, meaning it doesn't server the process that has just been stopped. If the interrupt is synchronous then the kernel is in process context as it needs to provide some service to the process that invoked it.

NOTE: interrupt handlers are not schedulable entities, they run immediately (without invoking the scheduler), unless the interrupt is masked. We can mask using a bitmap, incase of maksing an interrupt the HW will remember at most 2 instances of each interrupt that was blocked by the mask (the rest get lost, if they exist).

Top & Bottom Halves Interrupt handling is often divided into two parts: the **top half** and the **bottom half**. The top half is the part of the interrupt handler that runs in response to the interrupt and does the minimal amount of work necessary to handle the interrupt. The bottom half is the part of the interrupt handler that can be deferred and run later, allowing the top half to return quickly and minimize the impact on the system's responsiveness.

Resuming an interrupted process We split to multiple cases:

- **Asynchronous (external) interrupts:** Resume in the next process instruction, right after the last instruction that was exectued. Note that asynchronous interrupts are caught and handled between instructions.
- **Synchronous (=internal) + explicit:** Likewise, resume in the next instruction.
- **Synchronous (=internal) + implicit:** Resume in the same instruction that caused the HW failure, called restartable instruction. For temporal problems the OS will fix the problem, but for error conditions it doesn't make sense so a signal will also be sent to the process.

Interrupt handling in multicores Internal (synchronous) interrupts are always handled by the core that executed the instruction that caused the interrupt. External (asynchronous) interrupts can be handled by any core, so we can do any kind of partition between the cores (static/dynamic/...).

Polling Polling is a technique used to check the status of a device or resource at regular intervals. In the context of interrupts, polling can be used as an alternative to interrupts for handling events. Instead of relying on the hardware to generate interrupts, the CPU actively checks the status of devices and resources to determine if they require attention. This can be useful in situations where interrupts are cause an interrupt storm (too many interrupts in a short time), leading to performance degradation.

Interrupt Coalescing The hardware tries to not create an "interrupt storm" by coalescing multiple events into a single interrupt. And the Software also temporarily switches to polling mode when the flow of incoming intterupts exceeds a certain thershold.

DMA & interrupts DMA = Direct memory access, which is when an I/O device can directly read/write to memory without involving the CPU. Conceptually devices fire an interrupt when a DMA operation is finished, not only when reciving (reading) data, but also when sending (writing) data. This allows to notify OS when the send buffer has become free.

Note: All interrupts are handled in kernel mode, CPL=0, including the timer interrupt.

Topic 11: Storage

Files & Directories

File A file is a logical unit of information which is an ADT (abstract data type), with a name, content, metadata, and can be manipulated via operations (read, rename,...)

File Metadata Metadata is information about the file and its content, such as: (1) file name, (2) file size, (3) owner, (4) permissions, (5) timestamps (creation, modification, access), (6) physical location, (7) file type.

File Descriptors A file descriptor (FD) is a nonnegative integer that uniquely identifies an open file within a process. An FD is an index into a per-process file descriptor table (FDT), which maps the FD to a file object in the kernel. Threads share the FDT of their process. Some POSIX syscalls operate on FDs and others operate on file names (chmod, unlink, ...).

File Operations The following are the most common file operations:

- **Creation** (syscalls: `creat`, `open`; C: `fopen`)
  - Associate with a name; allocate physical space (at least for metadata)
- **Open** (`open`; C: `fopen`, `fdopen`)
  - Load required metadata to allow process to access file
- **Deletion** (`unlink`, `rmdir`; C: `remove`)
  - Remove name/file association & (possibly) release physical content
- **Close** (`close`; C: `fclose`)
  - Mark end of access; release associated process resources
- **Rename** (`rename`; -)
  - Change associated name
- **Stat** (`stat`, `lstat`, `fstat`; -)

  - Get the file's metadata (timestamps, owner, etc.)
- **Chmod** (`chmod`, `fchmod`; -)
  - Change readable, writable, executable properties
- **Chown** (`chown`, `fchown`; -)
  - Change ownership (user, group)
- **Seek** (`lseek`; C: `fseek`, `rewind`)
  - Each file is typically associated with a "current offset"
  - Pointing to where the next read or write would occur
  - `lseek` allows users to change that offset
- **Read** (`read`, `pread`, `readv`; C: `fscanf`, `fread`, `fgets`)
  - Reads from "current offset"; `pread` gets offset from caller
  - Need to provide buffer & size
  - "v" = vector of buffer+size; this version is called "scatter-gather" (why?)
- **Write** (`write`, `pwrite`, `writev`; C: `fprintf`, `fwrite`, `fputs`)
  - Change content of file; `pwrite` gets the offset explicitly; v=vector
  - Likewise, need to provide buffer & size
  - If the current offset (or the given offset in the case of `pwrite`) points to end of file, then file grows
- **Sync** (`sync<fs>`, `fsync<fd>`; $\neq$ C: `fflush`)
  - Recall that
    - All disk I/O goes through OS "page cache", which caches the disk
    - OS sync-s dirty pages to disk periodically (every few seconds)
  - Use this operation if we want the sync now
  - 'sync' is for all the filesystem, and 'fsync' is just for a given FD
  - Sync $\neq$ fflush; the latter flushes user-space (`fprintf`, `cout`) buffers to the kernel
- **Lock** (`flock`, `fcntl`; C: `flockfile`)
  - "Advisory" lock (= processes can ignore it, if they wish)
  - There exists mandatory locking support (in Linux and other OSes)
    - E.g., every open implicitly locks; and can't open more than once
    - But that's not POSIX

File Types In UNIX (POSIX) we classify files into:

- **Regular files** - Contain user data, such as text files, binary files, etc. They are the most common type of file.
- **Directories** - Special files that contain a list of files and
- **Symbolic links (soft links)** - Special files that point to another file or directory, allowing for easy access to files in different locations.
- **FIFOs (named pipes)** - Special files that allow for inter-process communication (IPC) by providing a way for processes to communicate with each other through a named pipe.
- **Sockets** - Special files that allow for communication between processes over a network. They are used for network communication and can be either stream-oriented (TCP) or datagram-oriented (UDP).
- **Device files** - Special files that represent hardware devices, such as disks, printers, and terminals. They allow processes to interact with hardware devices as if they were regular files. Can be either character devices (e.g., terminals) or block devices (e.g., disks).

Magic Numbers Magic numbers are a semi-standard Unix way to tell the type of a file by looking at the first few bytes of the file. Each file type has a unique magic number that identifies it. For example, a PDF file starts with the ASCII string: `%PDF`. Not all files have magic numbers, but many do, since nowadays it is a more complex scheme.

File Protection (Ownership & Mode) For each file, POSIX divides users into 3 classes: "User" (the owner of the file), "group", "all" (all other users). Each class has 3 permissions: "read", "write", "execute". The file mode is a 9-bit number, where each bit represents a permission for a class. The first 3 bits are for the user, the next 3 bits are for the group, and the last 3 bits are for all users. Note: Users can belong to several groups, but not files.

Other operating systems (NOT POSIX) use Access Control Lists (ACLs), where it allows a finer more detailed control of what each user/group can do (delete, rename,...).

Directories A filesystem has a hierarchial (tree) structre of directories, where each directory can contain files and subdirectories. Each directory is a special file that contains a list of files and subdirectories, along with their metadata.

Absolute & Relative File Paths File name = File path = path. Every process has its own "working directory" (WD). A path is absolute if it starts with "/" (the root directory), e.g. "/users/admin/jon". A path is relative (to the WD) otherwise, e.g. if WD="/home" then "admin/jon" is the relative to "/home/admin/jon".

Directory Operations Each directory contains 2 special subdirectories, "."=Current directory, ".."=Parent directory. To create a directory we use `mkdir`, and to remove an empty one `rmdir`.

Hard Links File ≠ file name. They are not the same thing, in fact, the name is not even part of the file's metadata! A file can have multiple names, and therefore appearing in multiple unrelated places. Every file has a "reference count" to keep track of how many hard links it has, once it reaches zero, *if no process is using it through FDs*, then and only then we can remove the file's content.

Hard links to directories are usually disallowed, since it causes infinite circular looping and nesting. However, remember that each directory contains "." & ".." hence each directory has a minimum of 2 hard links, and a maximum depending on the number of subdirectories it contains.

Note: Hard links can only exist on the same filesystem, as they use the same `file_object` or as seen later, same `inode`.

Symbolic Links A symbolic link (or soft link) is a special file that points to another file or directory. It is a way to create a shortcut to a file or directory, allowing you to access it from a different location. Symbolic links are not hard links, and they can point to files or directories that do not exist. **Soft links contain target file path, and not a pointer to a file object like hard links**. They are no counted in the target file's ref count, they can refer to files, directories, and even files outside the filesystem. (since they only save a path)

When applying a syscall to a symlink, the syscall is applied to the target file. Except, (1) unlink, which removes the soft link. (2) symlink-specific syscall, such as symlink (create a symlink) and readlink (read the content of the symlink, i.e. the path of the target file).

inode An inode (index node) is a data structure used to represent a file in a filesystem. Each file has its own (single) inode, which can be thought of as the "file". File names (hard links) point to inodes, and the inode contains all metadata of the file.

Note: file objects (which represent open files) are not the same as inodes, as they contain additional information such as the current offset, number of file descriptors pointing to it, etc. However, file objects point to the inodes, as they are the ones holding to the actual data.

inode `*stat` syscall POSIX gurantees that at leas the following fields will be found in the stat structure and have meaningful values:

1. `st_dev` ID of device containing file
2. `st_ino` inode number, unique for st_dev
3. `st_mode` specifies file type & permissions (S_ISREG(m), ...
4. `st_nlink` number of hard links to the file
5. `st_size` file size in bytes
6. `st_uid` user ID of owner
7. `st_gid` group ID of owner
8. `st_ctime` last time metadata (=inode) or data **changed**
9. `st_mtime` last time data **changed**
10. `st_atime` last time metadata (=inode) or data **accessed**
11. `st_blksize` block size of this file object (preferred size for I/O ops)
12. `st_blocks` number of sectors allocated for this file object

`lstat` is like `stat` but it applies to the symbolic links when used on one, i.e. it returns the metadata of the symlink, not the target file. The only guaranteed metadata fields to be found are: `st_mode` (which will specify it is a symlink) and `st_size` (which will be the length of the symlink's target path).

It is not specified in POSIX whether a symlink gets an inode or not (could be either depending on implementaion).

Dirents (Directory Entries) A directory file is compromised of directory entries (dirents), which point to the inodes of the files and subdirectories it contains. Each dirent contains the following fields: (1) inode number, (2) file type, (3) file name, (4) offset to the next dirent, (5) length of the dirent. In **FAT** (more on that later) dirents are the actual files as there are no inodes, preventing multiple hard links.

Path Resolution Resolving a path means finding the inode of the file or directory that the path points to. The algorithm that syscalls use to resolve paths is called **"namei"**. Each path is compromised of "atoms", where an atom is a single file or directory name in the path. The algorithm works as follows:

```
int my_open(char *fname) {
  if (fname is absolute){
    // Start from the root directory
    chdir("/");
    make fname relative;
  }
  foreach atom in fname do // e.g. atoms of "x/y" are "x" and "y"
    if (is symlink) SYS (fd = my_open(atom symlink target))
    else            SYS (fd = open(atom)) // this checks permissions too

    if (!terminal){
      // If not the last atom, change the current directory to the opened one
      chdir(fd directory);
      close(fd);
    }
    else
      // If terminal, return the file descriptor of the last opened file
      break;
```

```
19   return fd;
20 }
```

Listing 5: Path Resolution Algorithm (namei)

For each non-symlink atom in the path, a user must be able to search all directories leading to the file. This means that the user must have the appropriate permissions to access each directory in the path. x bit = **search** allows the user to search for files in the directory. r bit = allows the user to list the contents of the directory (with `ls` for example). w bit = allows the user to create or delete files in the directory.

FDT & GFDT Each process has a **File Descriptor Table (FDT)**, which is an array of file descriptors that the process has opened. The FDT is indexed by file descriptor numbers, and each entry in the FDT points to a **File Object** in the kernel, which contains information about the open file, such as the current offset, the inode, and the file operations that can be performed on it.

All opened files for every process in the system are saved in one global table, GFDT.

By default, every process has the following three standard FDs: (`fd=0`) STDIN, the standard input channel, (`fd=1`) STDOUT, the standard output channel, (`fd=2`) STDERR, the standard error channel.

file object A file object is a kernel data structure that contains all the information about an open file, including its current offset, the inode it refers to, and the file operations that can be performed on it, the number of file descriptors pointing to it, the mode it was opened with, etc.

File Systems

Note: The CPU can't directly access/process data on the disk, it must reside on the DRAM.

HDD Latency HDD latency is **seek time** + **rotational latency**. Seek time is the time it takes for the read/write head to move to the correct track on the disk. Rotational latency is the time it takes for the disk to rotate to the correct sector.

Sectors & Blocks A disk is divided into **sectors**, which are the smallest addressable units on the disk. Each sector is typically 512 bytesor 4096 bytes in size. Sectors are grouped into **blocks**, which are the equivalent of pages in memory, thus each block is 4KB in size, and 8 sectors.

Partitions & Mounts A disk can be divided into **partitions** using `fdisk`, which are contiguous disjoint part of the disk that can host a filesystem. Disks are typically names "/dev/sda","/dev/sdb", while partitions are named "/dev/sda1", "/dev/sda2", etc.

To create an empty filesystem on a partition we use `mkfs`, which then after it need to be `mount`ed to make it accessible. Mounitng is done using `mount -t <filesystem> <partition> <path>` which will then make the filesystem reachable from `<path>`. If any

files existed at `<path>` before mounting, they will become unavailable until we unmount the filesystem.

VSFS (Very Simple File System) Blocks are divided into 5 categories:

- **Superblock (S)** - Contains metadata about the filesystem, such as the magic number of the filesystem, the inode of the root directory "/", amount of inodes and data blocks, start of inode table, etc. It is used when mounting the filesystem.
- **Inode Table (I)** - Contains the inodes of all files in the filesystem.
- **Data Blocks (D)** - Contains the actual data of the files in the filesystem.
- **bitmap of allocated inodes (i)** - A bitmap that indicates which inodes are allocated and which are free. Each bit corresponds to an inode.
- **bitmap of allocated data blocks (d)** - A bitmap that indicates which data blocks are allocated and which are free. Each bit corresponds to a data block.

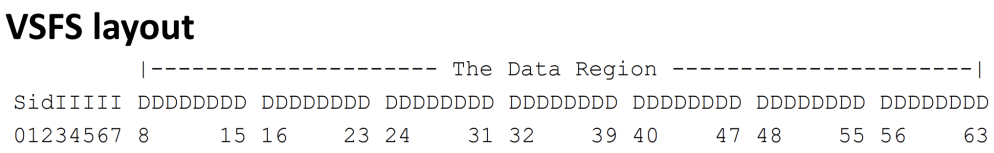


Figure 12: VSFS File System Layout

VSFS - Finding an inode to find the address of an inode, we can simply do

$$\text{inodeAddr} = \text{inodeStartAddr} + \text{inodeSize} \times \text{inodeIndex}$$

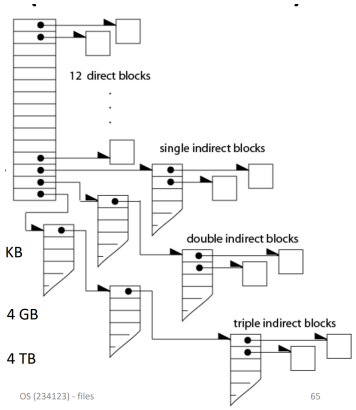
and to find the block:

$$\text{blockAddr} = \text{inodeAddr}/\text{blockSize}$$

Pointers to data in VSFS could be:

- **Direct:** An array of pointers pointing directly to all data blocks (limited amount $\Longrightarrow$ limits file size)
- **Indirect:** Multi-level array, like in virutal memory page tables.
- **Linked List:** Linked list of pointers from one data block to the next

Multi-Level Index: Each inode will have an array of 15 pointers such that: (0-11) The first twelve are direct pointers to data blocks, (12) a single-indirect pointer that points to a block completely comprised of pointers to data blocks, (13) a double-indirect pointer that points to a block that contains pointers to single-indirect blocks, and (14) a triple-indirect pointer that points to a block that contains pointers to double-indirect blocks. Note: each indirect access requires an additional disk access, so it is slower than direct access.



Extents An extent is a contiguous block of data blocks that are allocated together. Instead of having a separate pointer for each data block, an extent is a single pointer that points to the first block of the extent and contains the length of the extent. This allows for more efficient allocation and reduces fragmentation. Extents are used in modern filesystems such as ext4.

- ext2 = the "extended file system", does not support extents.
- ext3 = ext2 + journaling. journaling = keeping track of uncommitted changes.
- ext4 = ext3 + extents. (so "ext" name unrelated to "extent")

FAT (File Allocation Table) Keeping track of a linked list using the data blocks is inefficient. So to solve that, FAT keeps a file allocation table that for each data block, it registers that pointer to the next data block from the same file. The starting block for the file is pointed to by the dirent, which holds all of the metadata too, so no inodes are needed! (but no multipicity of hard links is allowed)

The advantage for FAT is that it will be copied to (cached in) the memory, which allows fast lookup in the table. The disadvantage is that it can become huge.

RAID

RAID (Redundant Array of Independent Disks) RAID is a technology that allows us to combine multiple physical disks into a single logical disk. It is based on striping, which is the process of dividing data into blocks and distributing them across multiple disks. And redundancy, which is the process of storing multiple copies of data to protect against disk failures.

- **RAID 0** - Striping without redundancy.
  - Files are stripped in **block** resolution.
  - Pros: Increased performance, as multiple disks can be accessed in parallel.

  - Cons: No redundancy, if one disk fails, all data is lost. and increased chance of failure.

- **RAID 1** - Mirroring.

  - Files are striped across half the disks, and each block is mirrored on the other half.
  - Pros: Increased reliability, as data is stored on multiple disks. Read performance is as good as RAID 0.
  - Cons: Write performance is slower than RAID 0, as data must be written to multiple disks.

- **RAID 4** - Dedicated parity disk.

  - Data blocks are striped across multiple disks, and a dedicated parity disk is used to store parity information. RAID 2 is bit resolution, RAID 3 is byte resolution.
  - Pros: Improved read performance, as data can be read from multiple disks in parallel. Fault tolerance, as data can be reconstructed from parity information.
  - Cons: **"small write problem"**, as writing to a single block requires reading the old data + the parity block, computing the new parity, and writing both the data and the parity block. However, if we write to the entire stripe (whole block on each disk) then we don't have to read the parity to update it, hence "small write problem".

- **RAID 5** - Distributed parity.

  - Data is striped across multiple disks, and parity block is distributed each time on a different disk.
  - Pros: Like RAID 4 + No hotspot disk.

- **RAID 6** - Double distributed parity.

  - Data is striped across multiple disks, and two independent parity blocks are distributed across the disks.
  - Pros: Fault tolerance for up to two disk failures, as data can be reconstructed from the remaining disks.
  - Cons: wastes $2/N$ instead of $1/N$ of the disk space for parity (compared to RAID 5).

- **RAID n+k** - n data blocks and k parity blocks (in each stripe).

Erasure (n+k) vs. Replication (r) Erasure is the "k parity" blocks, which allows us to reconstruct the data from the remaining blocks. Replication is duplicating the data r times. Erasure is less wasteful but slower, while replication is faster but more wasteful.

Part II

Overall Summary

Part III

Highlights and Notes