Lecture 2 Process Management

2022年9月14日 22:40

Process ≠ program

A program is just a piece of code.
- High-level language (C, C++, Java, …)
- Intermediate language (Java bytecode, LLVM IR, .NET CIL, …)
- Low-level language (assembly)
- Machine code

Life of a C program

Preprocessor (cpp)

The preprocessor expands **directives** such as *#include*, *#define*, *#ifdef*, … in the C source code.
gcc -E main.c shows the **expanded code**.

Compiler (gcc)

The compiler takes the expanded C code, checks the syntax, and generates assembly code (gcc) or LLVM IR (clang).
In the meantime, it also **optimizes** the code.
gcc -S stupid.c shows the assembly code without optimization.
gcc -S -O2 stupid.c shows the assembly code with optimization. 2 specifies the level of optimization. Optimization is by default turned off, because it makes debugging the source code difficult.

Assembler (as)

The assembler converts the generated assembly code to an **object file**. The object file contains machine code, but is not yet executable.
as hello.s -o hello.o shows the contents in the object file.

Linker (ld)

The linker combines object files and libraries to produce the **executable file**. A library is just a collection of functions and variables.
- Static linking: the final program embeds all library function used.
- Dynamic linking: the final program does not embed the library functions. The OS will load dynamic libraries when the program runs.

Processes

The **process** is the most central concept in an operating system.
- It is an abstraction of a running program.
- It attaches to all the memory that is allocated for the process.
- It associates with all the files opened by the process.
- It contains accounting information such as its owner, running time, memory usage…

Process identification

The OS gives each process a unique identification number, the **Process ID (PID)**.
- *getpid()* returns the PID of the calling process.
- *getppid()* returns the PID of the parent of the calling process.

Remark: if we run program in the shell multiple times, PID changes but PPID does not change because the parent is the shell.

Process creation

Before you call *fork()*, you have only one parent process. After calling *fork()*, the parent process forks into itself and a child process.

Example:

```c
int main() {
  printf("Before fork, my PID is %d\n", getpid());
  fork();
  printf("After fork, my PID is %d\n", getpid());
}
```

fork.c

The second line will be printed twice, because the child process does not run from the beginning, and after forking both parent and child execute the same code after it.

The *fork()* function actually returns an integer value, 0 in child process and the pid of child in parent process (nonzero).

The child will inherit the program code (same code), the program counter (execute from same location), memory (variables and allocated memory), and opened files. However, the following are different: return value of *fork()*, pid, parent, running time (reset to 0), and file lock (do not inherit).

Example of special case:

```c
int main() {
  printf("Hello ");
  fork();
  printf("CS202\n");
}
```

fork_buffer.c

printf() actually has a buffer in the file structure to reduce system calls, and this is inherited by the child. STDERR is unbuffered (*write()* immediately), but STDIN and STDOUT are line-buffered (*write()* at newline character). Hence, the output of this program will in fact be two lines of "Hello CS202". *setvbuf()* changes the buffering strategy, and *fflush()* can force a write. For example, adding *fflush(stdout)* before *fork()* will result in second line containing only "CS202".

Process execution

execve() and the *exec*()* family of functions: replaces the current process image with a new program. When called, the code is replaced and it will never return to the original code. Variables, allocated memory, and registers (such as program counter) are discarded, but pid, process relationship, and running time are preserved.

- Path name or file name
  - Default: path name
  - Functions with *p*: file name (*execlp()*, *execvp()*)
- Argument list or array
  - Functions with *l*: list (*execl()*, *execlp()*, *execle()*), in fact variadic function ending with NULL argument
  - Functions with *v*: array (*execv()*, *execvp()*, *execve()*)
- Environment variables
  - Default: inherit the current environment
  - Functions with *e*: specify a new environment array

Note: environment variables are a set of strings maintained by the shell, contained in *char **envp*. For Lab 2, we are only interested in the PATH environment variable.

Process creation and execution

system() is implemented using *fork()* and the *exec*()* family. We can fork a child process and let it run. However, we cannot control the OS's scheduling so that the child runs first. However, we can let the parent pause while child is running.

The *wait()* system call suspends the calling process until one of its children terminates. The *waitpid()* function waits for a particular child or a stopped/resumed child.

```
01  int my_system(const char *command) {
02    if (fork() == 0) {
03      execl("/bin/sh", "/bin/sh", "-c", command, NULL);
04      exit(-1);
05    }
06    wait(NULL);
06    return 0;
07  }
08
09  int main() {
10    printf("Before system\n");
11    my_system("ls");
12    printf("After system\n");
13  }
```

Can we remove this exit()?

my_system.c

Note that we cannot remove the *exit()*, since *execl()* may fail, and we must imitate its behavior to exit the current code.

Processes in the kernel

Kernel-space memory

The memory is divided into user-space memory and kernel-space memory.
- User-space memory stores program code, process's memory, etc.
- Kernel-space memory stores kernel code, kernel data structures, loaded device drivers, etc.

In the kernel-space memory, the **process control block** contains everything related to a process.
- Doubly-linked list to store processes using *struct task_struct*
- Each process information contains pid, running time, file descriptors, etc.
- File descriptors is an array of opened files (0: STDIN, 1: STDOUT, 2: STDERR, 3+: yours)

Redirection: use *dup2()* function to duplicate the file descriptor, then use *close()* to close the unused file descriptor.

```
int main() {
  printf("Hello CS202\n");

  int fd = open("output.txt",
                O_CREAT|O_WRONLY|O_TRUNC,
                S_IRUSR|S_IWUSR);
  dup2(fd, 1);  // duplicate the file descriptor
  close(fd);    // close the unused file descriptor

  printf("Hello CS202 again\n");
}
```

redirect.c

Process execution

A process switches its execution from user mode to kernel mode by invoking system calls. When the system call finishes, the execution switches from kernel mode back to user mode.

Handling system calls

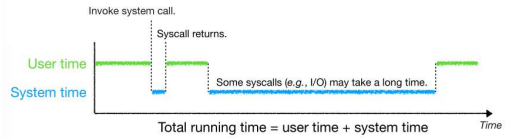
The CPU is running a process in user mode. At some time it wants to invoke the *getpid()* system call. Each system call has a unique **syscall number**, so the process puts the syscall number of *getpid()* in a specific CPU register (*e.g.*, *%rax*), and then it executes ***TRAP*** instrution to switch from user mode to kernel mode.

The kernel starts execution at the **syscall dispatcher.** It examines the syscall number, looks up the syscall table (mapping syscall number to syscall handler), and invokes the corresponding **syscall handler**.

The *sys_getpid()* handler reads the process ID of the calling process from **task_struct**, then it executes a ***RETURN-FROM-TRAP*** instruction to switch from kernel mode back to user mode.

Process time

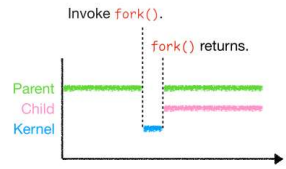
The execution of a process is divided into two parts: user time and system time.



The *time ./program* command can display the real, user, and sys time. Note that system calls can play a major role in performance. Some system calls can even stop your process until the data is available. Hence, it is important to reduce the number of system calls in a program for better performance.

Process-management syscalls in the kernel

fork()

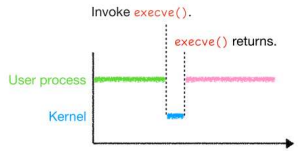


In kernel space memory:
- Create new process (new PID, running time reset to 0, clone file descriptors, etc.)
- Parent's list of children adds this new process, and this new process sets its parent.

In user space memory:
- Copy-on-write: if nothing writes to the new process, then user memory has a view of the new process but do not actually clone it. Clone only happens on writing.

If two processes sharing the same file write to that file together (e.g., maybe one has opened a file and forked before closing it), then the writing order is random. One needs to synchronize writing himself.

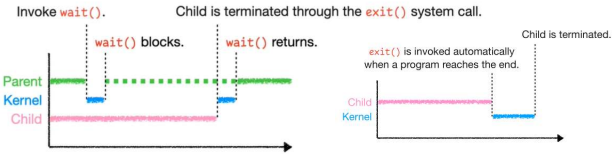
execve()



In kernel space memory:
- Everything of the process is preserved, and no processes are created.

In user space memory:
- The kernel loads the code of the new program and replaces the text.
- The kernel resets the memory (heap, stack, global variables, etc.)
- CPU registers (e.g. program counter) will be reset.

wait() and *waitpid()*



When calling *exit()*, the kernel frees all allocated memory and closes all opened files in kernel space memory. It also removes everything about the process from the user space memory.
The child's storage in the system is now minimal. However, it still owns the PID and *task_struct*, and the child now enters the **zombie state**.
Finally, the kernel sends the parent of the calling process a SIGCHLD signal.

Now, the parent is running concurrently with the child, and the SIGCHLD signal will be sent to parent. By default, a process **ignores the SIGCHLD signal** unless it has called *wait()* or *waitpid()*.

In detail, when *wait()* is called, the kernel sets a **signal handler** to the process, which will be executed when SIGCHLD arrives (note that this is a function pointer). Then, the kernel sets the process to the **blocked** (a.k.a. **sleeping**) state. Then, when SIGCHLD arrives, the parent's signal handler (not the original program code) is invoked. The signal handler lets the parent process **remove the signal** and **destroy the corresponding child process**. Finally, the signal handler is removed, so the process is once again ignoring SIGCHLD. It then returns to the previously-executing original program code **in the user space**. Therefore, it looks as if *wait()* is returned from its invocation.

Zombies

The zombie process lives up to the moment the parent process calls *wait()*. One should never leave any zombies in the system, and *wait()* and *waitpid()* are to reap zombie children. Zombies take up PIDs, so it is something regarding system resource management.

Signals

Signal is a form of **inter-process communication (IPC)**. A process can send signal to another process. When a signal arrives, the OS interrupts the process's normal control flow and executes the **signal handler**.

How are signals generated?

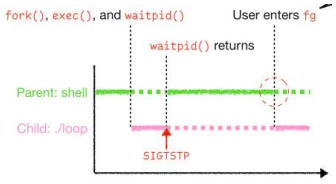
From the user space
- Keyboard: Ctrl-C sends SIGINT, Ctrl-Z sends SIGTSTP, Ctrl-\ sends SIGQUIT, etc.
- Commands: *kill*, *top*, etc.
- Using the *kill()* system call.

From the kernel or hardware
- SIGCHLD (child stopped or terminated) comes from the kernel.
- SIGFPE (floating point exception, e.g., division by zero) comes from the CPU.
- SIGSEGV (invalid memory reference, a.k.a. segmentation fault) comes from the CPU to the kernel and then to the process.

The *kill* command

The *kill* command sends SIGTERM to the target process by default. The default signal handler of SIGTERM is to terminate the process. For example, *kill 2250* terminates the process with PID 2250.

Actually, the *kill* command can send any signal. For example, *kill -TSTP 2250* suspends the process with PID 2250. The Linux shell will report something like *[1]+ Stopped ./program*. Also, *kill -CONT 2250* resumes this program.



Note that if simply a SIGCONT signal is sent to the child process instead of entering *fg* command, the child and the parent processes will in fact run concurrently, and the child process is running as a background job.

Just like the *kill* command, the *kill()* syscall **sends a signal** to a process in the format *int kill(pid_t pid, int sig)*. Also, the *raise()* library function sends a signal to yourself.

Signal	Description	Default handler
SIGINT	Interrupt from keyboard (Ctrl-C).	Terminate the process.
SIGTERM	Termination (the default signal sent by the kill command).	Terminate the process.
SIGKILL	Kill, no matter what.	Terminate the process (cannot be overridden).
SIGTSTP	Stop typed at terminal (Ctrl-Z).	Stop the process.
SIGCONT	Continue if stopped.	Continue the process.
SIGCHLD	Child stopped or terminated.	Ignore the signal.
SIGFPE	Floating point exception.	Terminate the process and dump core.

SIGTSTP	Stop typed at terminal (Ctrl-Z).	Stop the process.
SIGCONT	Continue if stopped.	Continue the process.
SIGCHLD	Child stopped or terminated.	Ignore the signal.
SIGFPE	Floating point exception.	Terminate the process and dump core.
SIGSEGV	Invalid memory reference (segmentation fault).	Terminate the process and dump core.

Changing signal handlers

*void (*signal(int sig, void (*handler)(int)))(int);*
Here, *type_a(*something)(type_b)* means a function pointer that takes arguments of *type_b* and returns value of *type_a*.
Hence, this can be simplified to *funcptr_signal(int sig, funcptr_handler)*, where both function pointers takes a single integer argument and returns void.
The returned function pointer is the old signal handler, and the one in the argument is the new signal handler.

Example: change the behavior of Ctrl-C

```
void handler(int sig) {
     // Do something  // The behavior of the new signal handler
}

int main() {
     signal(SIGINT, handler);  // register a new signal handler meeting Ctrl-C
     // Do something
}
```

The *pause()* system call puts the process to **block (sleep)** until the delivery of a signal handled by the process or a signal terminating the process. This is a special case for the *alarm()* system call, which sets up a **one-time asynchronous timer** for the process. A SIGALARM signal will be set to the process at timeout, default to terminate the process but can be overridden. On the other hand, the *setitimer()* system call sets up an **interval timer**. These functions can be used to implement *sleep()*.

Summary

Signal is a kind of **software interrupt**, which has many quirky behaviors. *kill()* is **not intended to kill anybody**, but to send signals. When in doubt, refer to the Linux manual pages.

Process organization

Booting the computer

BIOS locates the boot device. The boot code on the boot device is then executed, and the OS is started.

BIOS (Basic input/output system) is **firmware** (i.e., software providing low-level control for hardware). It is stored in a ROM chip or flash memory* on the motherboard.
*Flash memory: most of the time unchanged, only changed occasionally when for example firmware update.

The BIOS starts when the computer boots up. It checks all devices attached to the computer and locates the **boot device**. (Recent systems have replaced BIOS with UEFI, Unified Extensible Firmware Interface, which is a complete boot manager.)

Boot device

The boot code in the boot device's first sector is executed. The boot code reads the **partition table** and determine the **bootable partition**. The **boot loader** from the bootable partition is executed.

The boot loader locates and boots the **kernel image**. The kernel image is just a file (Linux: /boot/vmlinuz-*; macOS: /System/Library/Kernels/kernel; Windows: C:\windows\system32\ntoskrnl.exe). The kernel initializes the memory layout, device drivers, etc., and creates the **first** process.

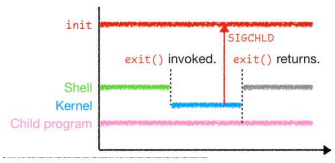
During booting, the kernel creates the first process (PID=1): /sbin/init.
- Recent Linux systems have replaced *init* with *systemd*.

init creates more processes using *fork()* and *execve()*.
- It reads configurations from /etc/rc.d about which programs to run.
- The entire **process tree** can be viewed using the *pstree* command.
- *init* continues running in the background **until the system is shut down**.

Orphans

A process becomes an **orphan** when its parent process terminates. The *init* process automatically adopts all orphaned processes (this is called **reparenting**).

In the kernel, the killed parent process (shell) becomes a zombie, and the child is reparented to *init* process.




```
void handler(int sig) {
  waitpid(…);
}

int main() {
  …
  signal(SIGCHLD, handler);
  …
  // doing other work
  …
}
```
init.c

Observations

Processes in Linux are organized as a **single tree**.
- Windows does maintain a forest-like process hierarchy, but we will focus on Unix/Linux.

Reparenting allows processes to run **without a parent shell**.
- Therefore, background jobs can survive even after the shell exits.

Process scheduling

Computers often do several things concurrently, even if it has only one CPU.

- It's called **multiprogramming**.

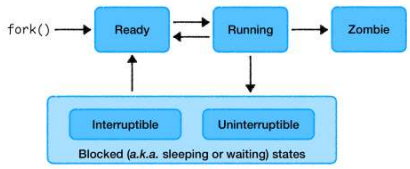
The CPU switches from process to process quickly, running each for a few milliseconds.

- It's called **multitasking**.

The OS needs to choose which process to run next.

- It's called **scheduling**.
- The part of the OS that makes the choice is called the **scheduler**.

Process states



Ready means **runnable but temporarily stopped** to let another process run.

- It is just created by *fork()*;
- It has run on the CPU for long enough, and the scheduler picks another process to run;
- It returns from a blocked state.

Running means the process is actually **using the CPU** at that instant.

- The scheduler picks this process to run.

Blocked means **unable to run** until some **external event** happens.

- It invokes the *pause()* system call;
- It reads from an I/O device;
- Etc.

Example: reading a file on the disk. The process has to wait for the response from the disk drive, so it is **blocked**. However, this blocked state is **interruptible**. For example, pressing Ctrl-C can get the process out of the blocked state.

Example: a blocked state may also be **uninterruptible**. For example, a process may need to wait for a resource, but it does not want to be disturbed while it is waiting. This scenario often happens within the kernel.

Zombie means the process **exits** and its **parent has not yet waited** for it. It terminates when the parent finally waits for it.

Context switching

The **scheduler** in the kernel will choose the next process to run. Before the scheduler can take up the CPU, it has to **back up register values**. The backup should be stored in the kernel space memory since the kernel space scheduler handles that. The register values are saved using *task_struct*. The **context** of a process consists of its user-space memory and its register values.

Then, the scheduler decides which process to run next. If the next process is different from the current one, the OS performs a **context switch**, including:

- Saving and restoring registers;
- Switching memory maps;
- Flushing and reloading the cache;
- Etc.

Context switching may be expensive (and the system is not doing any useful work).

Process scheduling

Most processes' execution **alternates between CPU execution and I/O wait**.

- **CPU-bound processes** spend most of their running time on the CPU. Examples: compiling a program, rendering a video, mining Bitcoin, scientific programming, etc. *time* command will show user time > sys time.
- **I/O-bound processes** spend most of their running time on I/O. Examples: /bin/ls (mainly disk I/O), downloading a file from the Internet, printing an image, etc. *time* command will show sys time > user time.

When to schedule

- A new process is created. It's up to the scheduler to decide whether to run the parent or the child.
- An existing process is terminated. The scheduler should choose another process to run. If no process is ready, and "idle" process is run.
- A process starts waiting for I/O (or something else). The process is **blocked** and thus the process should choose another process to run.
- A process finishes waiting for I/O. The process becomes **ready** to run again. It's up to the scheduler to decide whether to run it.
- Just periodic scheduling…

Nonpreemtive vs. preemptive scheduling

Nonpreemptive scheduling: when a process is scheduled, it keeps running until it starts **waiting** for the I/O (or something else), or it **voluntarily** relinquishes the CPU (*man 2 sched_yield*: *sched_yield()* causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.)

Preemptive scheduling:

- A process can run for a particular period of time.
- When the time is up or some particular events occurs (e.g., I/O completion), the process is **suspended** and another process may run.
- The **hardware clock interrupt** gives control of the CPU back to the scheduler.

General goals

Fairness: each process should have a fair share of the CPU.
Policy enforcement: the system's **policies** should be carried out. For example, the admin may state that certain processes have a higher priority.
Balance: all parts of the system should be kept busy all the time. For example, CPU-bound and I/O-bound processes should be mixed together.

Batch systems

A **batch system** collects a set of jobs and then process them **in batches**. Examples: bank systems, data analytics, etc. (or even your laundry basket, where you "submit" you dirty clothes at the end of each week).

Batch jobs can run without user interaction, so nonpreemptive scheduling or preemptive scheduling with large time slices are acceptable.

Scheduling goals for batch systems:

- High throughput: maximize the number of jobs per hour.
- Short turnaround time: minimize time between submission and completion.
- High CPU utilization: keep the CPU busy all the time.

Interactive systems

An **interactive system** may be a computer with an interactive user, or a server that serves multiple interactive users.

Preemption is essential to keep one process from denying service to others.

Scheduling goals for interactive systems:
- Short response time: minimize the time between request and response. Interactive jobs may take precedence over background jobs.
- Proportionality: meet users' expectations. You think it's OK to load a video game for several minutes, but not OK if it reacts a second after you press 'A'.

Real-time systems

Real-time systems must guarantee response within specified time constraints. Example: a robot welding cars moving down an assembly line.

Scheduling goals for real-time systems:
- Meeting deadlines: avoid losing data. If a device produces data at a regular rate, you must run the data-collection process on time.
- Predictability: avoid quality degradation (in multimedia systems). If the audio process runs too erratically (not regularly), the sound quality will deteriorate rapidly.

Scheduling algorithms *(... continued process scheduling)*

- First-come, first-served: it's nonpreemptive.
- Shortest job first: it can be nonpreemtive or preemptive.
- Round-robin: it's preemptive.
- Priority scheduling: it can be nonpreemptive or preemptive.

Metrics

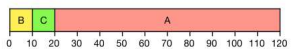
Wait time: the duration that the job is in the system but **not running**.
Turnaround time: the duration from when the job arrives in the system to the time it completes.

First-come, first-served (FCFS)

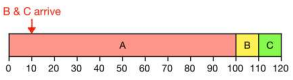
First-come, first-served (FCFS), a.k.a **first-in, first-out (FIFO)**, is a nonpreemptive scheduling algorithm in batch systems.

Nonpreemptive shortest job first (SJF)

Job	Arrival time	CPU requirement
A	0	100
B	0	10
C	0	10



Job	Arrival time	CPU requirement
A	0	100
B	10	10
C	10	10



The first case is good, but the second case is not good. Even though B and C arrived shortly after A, they have to wait until A completes. This is called the **convoy effect**, where a number of relatively short potential consumers of a resource get queued behind a heavyweight resource consumer.

Preemptive shortest job first (PSJF)

To address the previous concern, whenever a new job arrives, the **PSJF** scheduler determines which job has the **shortest remaining time**, and schedules that one.

Round-robin (RR)

PSJF looks great for batch systems. However, in interactive systems, users would demand a short response time, which is defined as the duration from when the job arrives in the system to **the first time it is scheduled**.

Round-robin (RR) is a preemptive scheduling algorithm in interactive systems. Each job is assigned a **time slice** (a.k.a **quantum**). The time slice is the amount of time the job is allowed to run. At the end of the time slice, the CPU is preempted and given to the next job. For simplicity, assume all jobs have the same time slice. Jobs are running one by one in a queue, which is called the **run queue**.

There is an inherent trade-off between performance and fairness. A fair scheduler (e.g., RR) evenly divides the CPU among active jobs on a small time scale, at the cost of turnaround time. To be more concise, typically RR has **worse CPU efficiency** than SJF. There are more context switching, and the average wait time and turnaround time are generally longer. However, jobs on a RR scheduler are **more responsive**, so you will not feel that a job freezes because it's on the CPU from time to time.

Most ordinary users run a lot of interactive jobs on modern operating systems. They value **responsiveness** more than CPU efficiency.

Priority scheduling

Each job is assigned a **priority**. The scheduler always chooses the job with the highest priority to run.

Priorities can be static or dynamic.
- **Static priority** means that a job is assigned a fixed priority when it is submitted to the system. Example: a background email process should get a lower priority than a real-time video game process.
- **Dynamic priority** means that a job's priority may be changing throughout its lifetime in the system.

Static priority scheduling

Jobs with the highest priority are scheduled first. They should be **short-lived** to prevent **starvation** of lower-priority jobs. When there is no job with the highest priority left, jobs with the second highest priority will be scheduled, and so on. Note that when a higher-priority job appears, the running job may or may not be preempted.

Limitations:
- High-priority jobs may run for a prolonged period, or even indefinitely.
- Low-priority jobs may starve to death.
- It does not differentiate between CPU-bound and I/O-bound jobs. More specifically, I/O-bound jobs spend most of their time waiting for I/O to complete. When such a job wants the CPU, we had better schedule it immediately to let it start its next I/O request. In that way, I/O requests can proceed **in parallel** with another process actually computing.

Dynamic priority scheduling

There is no standard way to assign priorities dynamically. An example policy can be:
- All jobs start running at Pr. 3 with time slice 30 ms.
- A job is preempted if its time slice is used up or it starts waiting for I/O.
- When a job is preempted, its priority is changed to **the ceiling of its time slice left / 10 ms**.

We are in fact using Round-Robin for each queue. If a process uses up its time slice, then it's priority will change to zero. If a process is blocked to wait for I/O, then its priority will be reassigned based on the time it has already used.

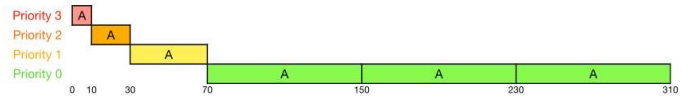
In this way, I/O-bound jobs will have a higher priority to access the CPU.

Multilevel feedback queue (MLFQ)

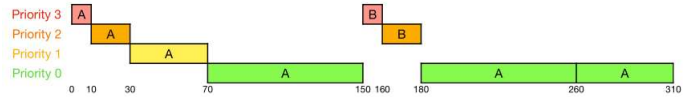
Ideally, we want to:
- Optimize **turnaround** time: run shorter jobs first, and give CPU-bound jobs a larger time slice to reduce context switching.
- Make the system feel **responsive** to interactive users: cannot give all jobs a large time slice, and need to minimize response time.

MLFQ is a kind of dynamic priority scheduling, but **each priority has its own policy**:
- All jobs start running at the highest priority.
- When a job uses up its time slice, its priority is reduced by 1.
- If a jobs gives up the CPU before the time slice is up, it stays at the same priority.

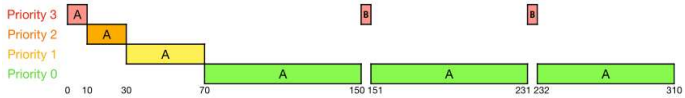
Also, for higher priority, the time slice of round-robin is shorter, and for lower priority, the time slice for round-robin is longer.



Case 1: long-running jobs get a large time slice to reduce context-switching.



Case 2: short jobs run first.



Case 3: I/O-bound jobs have a higher priority to access CPU.

Limitations:
- Long-running jobs may **starve** if there are too many interactive jobs.
- A job may need to run for a long time when it first starts, and becomes interactive after that. Such a job will be punished forever.

Hence, we introduce a new rule: **the priority boost**, which means to move all jobs to the highest priority after some time period.

MLFQ observes how jobs behave over time, and prioritize them accordingly.
- It can deliver excellent overall performance (similar to SJF/PSJF) for short-running interactive jobs.
- It is fair and makes progress for long-running CPU-intensive jobs.

Therefore, many modern operating systems use a form of MLFQ as their base scheduler.

Summary

There is no best or standard algorithm, partly because we cannot predict the CPU requirement of a process, and online scheduling is an NP-hard problem.

Linux employs the **Complete Fair Scheduler (CFS)** since kernel 2.6.23. It is a **dynamic priority scheduling** algorithm based on red-black trees.

Interprocess communication

What is IPC

Processes often need to **communicate** with one another. This is called **interprocess communication (IPC)**. Methods of IPC include but are not limited to: signals; pipelines; file, socket, shared memory; etc.

Why do we need IPC

- To share information.
- To reuse software.
- To speed up computation.
  - Example: MapReduce.
  - You can divide a job into tasks, run them as various processes in parallel, and combine the results.

Case study: piping

Example: *ls | less*
The *pipe()* system call returns two file descriptors:
- *pipefd[0]* refers to the read end of the pipe.
- *pipefd[1]* refer to the write end of the pipe.

来自 <https://man7.org/linux/man-pages/man2/pipe.2.html>
In this case, *ls* produces data into *pipefd[1]* and *less* consumes data from *pipefd[0]*. This is called a **producer-consumer model**.

From the kernel's perspective, if the pipe is full, then *write()* will block, while if the pipe is empty, then *read()* will block. This feature is called **process synchronization**.

Shared memory

Shared memory is a region of memory created by the kernel. It is visible to all processes in the system (by contrast, a pipe is only visible to the two processes at its two ends). Shared memory is also accessible by all processes in the system. However, there are syscalls to change the ownership and permissions of the shared memory.

In the case of a pipe, the kernel provides a form of synchronization. However, for shared memory, it is up to the processes to coordinate.

For example, a **race condition** (or, more specifically, a **data race**) may happen, where the results depend on the **timing** of execution, *i.e.*, the particular **order** in which the **shared resource** is accessed. Race conditions are always bad…
- Worse yet, compiler optimizations may generate crazy output if your code has data races. The behaviors are undefined.

Because the computation is **nondeterministic**, debugging is no fun at all. Bugs may disappear or change behavior during debugging.

Mutual exclusion

To avoid race conditions, we need **mutual exclusion**:

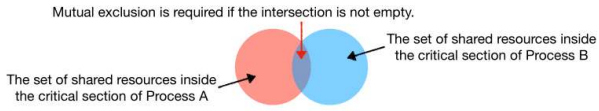
- If one process is accessing a shared resource, the other process must be **excluded** from accessing that same thing.
- Race condition is a problem, and mutual exclusion is a requirement to avoid such a problem. However, mutual exclusion may hinder the performance of parallel computations.

Critical sections

A **critical section** (*a.k.a.* **critical region**) is a piece of code that accesses a shared resource. We declare the entry of the critical region, in which lies the piece of code that accesses the shared resource. After that, we declare the exit of the critical section.

When a process enters its critical section, it **locks** the door. When it exits its critical section, it **unlocks** the door. Different processes may have **different critical sections**.

- A critical section is a piece of code, not a shared resource.
- A critical section should be as tight as possible.
- A critical section may access **multiple shared resources**.

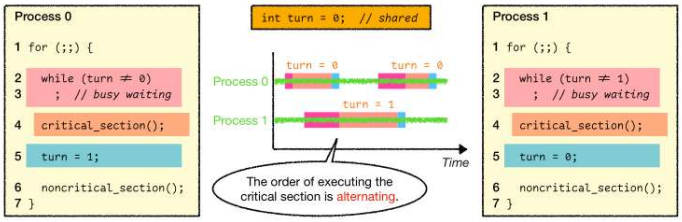


Requirements:
- No two processes may be simultaneously inside their critical sections. This is the mutual exclusion requirement: when one process is inside its critical section, any attempt to go inside the critical sections by other processes are **not allowed**.
- No assumptions may be made about the speeds or the number of CPUs. The solution cannot depend on the time spent inside the critical section or assume the number of CPUs in the system.
- No process running outside its critical section may block other processes. This ensures all processes can make progress. Otherwise, it may end up with a scenario where all processes are blocked but no process is inside its critical section.
- No process should have to wait forever to enter its critical section. This guarantees **bounded waiting**, *i.e.*, no process will starve to death.

Note: It is always the **section entry** that gets blocked, not the critical section itself!

Attempt 1: When one process is in its critical section, it disables interrupts, so that the CPU will not be switched to another process. On single processor systems, this may be correct, but on multiprocessor systems, other CPUs can still access the shared resource, so this is of no use. Also, even on a single processor system, it is a **terrible** idea to allow user processes to enable/disable interrupts at will. However, within the OS kernel itself, it is often convenient to disable interrupts for a few instructions.

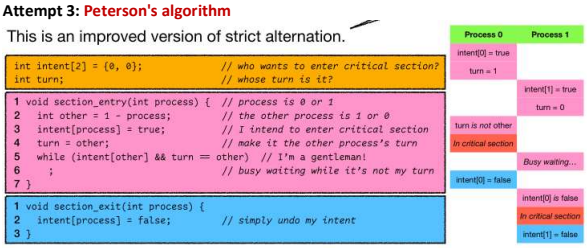
Attempt 2: Strict alternation



This indeed avoids all races. However, what if Process 0 wants to enter its critical section twice in a row? Hence, this is violating the requirement that **no process running outside its critical region may block other processes**.

Attempt 3: Peterson's algorithm

This is an improved version of strict alternation.



Peterson's algorithm is a combination of using **lock variables** and **taking turns**. It is indeed correct on early hardware, where *intent[]* and *turn* propagate immediately and atomically. But it is **not anymore correct** on modern hardware due to relaxed memory consistency models. Also we note that with a little **hardware support**, the solution could be much easier.

Attempt 4: spinlocks

Modern CPUs all have instructions that guarantee **atomic operation**. The simplest one is a **test-and-set** instruction. Conceptually, it behaves as if this code snippet is executed without interruption:

```
int test_and_set(int *ptr, int new) {
    // this code executes atomically
    int old = *ptr;
    *ptr = new;
    return old;
}
```

It returns the old value pointed to by *ptr* and simultaneously updates to *new*.

```
int lock = 0;  // 0: available, 1: held
```

```
void section_entry(int *lock) {

  while (test_and_set(lock, 1) == 1)
    ; // busy waiting

}
```

```
void section_entry(int *lock) {

  *lock = 0;

}
```

In this code, *test_and_set(lock, 1) == 1* means setting the pointer which is originally 1 to 1 again. If this is true, then another process must be in the critical section. As long as it exits the critical section, setting the value that *lock* points to as 0, immediately some process can set *lock* as 0, escape the while loop for waiting, and enter the critical section.

Spinlocks are built upon CPU instructions that guarantee **atomic operation** (e.g., test-and-set, compare-and-swap, fetch-and-add, load-linked/store-conditional, etc.) It is indeed correct on a **preemptive** scheduler, and beware if the **priority inversion problem**, where a high priority process waits for a low priority process to relinquish resource.

It is indeed a good solution. However, fairness is not guaranteed, since a process may spin forever under contention, leading to starvation. As for the performance, it is bad on single-processor systems, since with *N* processes contending for a lock, *(N-1)/N* of CPU time will be wasted. However, it is effective on multiprocessor systems if the number of processes is approximately the number of CPUs, and the **critical section is short**.

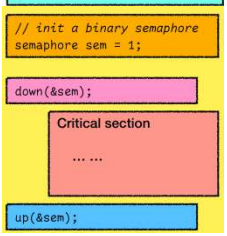
Notice that there may be too much spinning using spinlocks, so we may use *sched_yield()* to give up CPU in the busy waiting while loop. However, while better than spinning, the **run-and-yield** approach is still costly. Before the process holding the lock gets to run again, every other process still needs to run and yield, and note that context switching is expensive.

Attempt 6: semaphores

A **semaphore** is an object with a **non-negative** integer value, which must be initialized before being used. There are two operations: *down()* and *up* (see *man 7 sem_overview* for more information).

```
void down(semaphore *sem) {
  // atomic operation
  if (*sem > 0) {
    *sem = *sem - 1;
  } else {
    block on sem;
  }
}
```

```
void up(semaphore *sem) {
  // atomic operation
  if (some process is blocked on sem) {
    let one such process proceed
  } else {
    *sem = *sem + 1;
  }
}
```



In this implementation, a process tries to *down(&sem)*. If *sem* has value 1 then it sets it to 0 and enters the critical section. Otherwise, it is **blocked** on *sem*. Now when a process exits the critical section, it either choose a process blocked on *sem* to proceed into the critical section, or when there is nothing blocked on *sem*, it sets the value to 1.

We just used a **binary semaphore** to implement a **mutex** (mutual exclusion). It is indeed correct, and it is indeed a good solution. Actually, semaphores are more powerful than guaranteeing mutual exclusion. They can be used to realize **process synchronization**, *i.e.*, to coordinate the set of processes so they can produce meaningful output. Note that if the value of a semaphore is greater than 1, we call it a **counting semaphore**.

Classical IPC problems

The producer-consumer problem
- It models access to a bounded buffer.

The dining philosopher problem
- It models processes competing for exclusive access to a limited number of resources (*e.g.*, I/O devices).

The readers and writers problem
- It models access to a database.

The sleeping barber problem
- It models a queueing situation.

The producer-consumer problem

(*a.k.a.* the bounded-buffer problem)

Two processes **share a fixed-size buffer**. The **producer** inserts data to the tail of the buffer. The **consumer** removes data from the head of the buffer. (Yes, it is a pipe.)

Synchronization requirement 1: when the producer wants to insert an item into the buffer, but the buffer is **already full**, what should the producer and the consumer do?
- The producer should **block**.
- The consumer should **wake up** the producer after it has consumed an item.

Synchronization requirement 2: when the consumer wants to remove an item from the buffer, but the buffer is **empty**, what should the producer and the consumer do?
- The consumer should block.
- The producer should **wake up** the consumer after it has produced an item.

Of course, we also need **mutual exclusion**, meaning that no two process can access the shared buffer at the same time.

```
semaphore mutex = 1;          // controls access to critical section
semaphore empty = BUFFER_SIZE; // counts empty buffer slots
semaphore filled = 0;         // counts filled buffer slots
```

Why do we need three semaphores?

```
1  void producer() {
2    int item;
3
4    for (;;) {
5      item = produce_item();
6      down(&empty);  // decrement empty count
7      down(&mutex);  // enter critical section
8      insert(item);  // put new item in buffer
9      up(&mutex);    // exit critical section
10     up(&filled);   // increment filled count
11   }
12 }
```

```
1  void consumer() {
2    int item;
3
4    for (;;) {
5      down(&filled);    // decrement filled count
6      down(&mutex);     // enter critical section
7      item = remove();  // take item from buffer
8      up(&mutex);       // exit critical section
9      up(&empty);       // increment empty count
10     consume_item(item);
11   }
12 }
```

In the *producer()* function, it first produces an item, and it decrements the number of empty slots if the number of empty slots is not 0 (otherwise it is blocked on *empty*). Next it enters the critical section if there is no process currently in the critical section (otherwise it is blocked on *mutex*). In the critical section, it inserts the item in the shared buffer and then exits the critical section, choosing another process blocked on *mutex* to proceed if exists. Then it increments the number of filled slots if there is no processes blocked on *filled* (otherwise it lets a consumer blocked on *filled* to proceed and do not modify the count. Note that consumers may be blocked on *filled* because the shared memory is empty).

In the *consumer()* function, it decrements the number of filled slots if the number of filled slots is not 0 (otherwise it is blocked on *filled*). Next it enters the critical section if there is no process currently in the critical section (otherwise it is blocked on *mutex*). In the critical section, it removes an item from the shared buffer and then exits the critical section, choosing another process blocked on *mutex* to proceed if exists. Then it increments the number of empty slots if there is no process blocked on *empty* (otherwise it lets a producer blocked on *empty* to proceed and do not modify the count. Note that producers may be blocked on *empty* because the shared memory is full). Finally it consumes the item that it removes from the buffer.

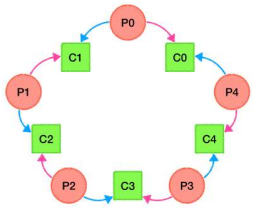
Important remarks:
- *mutex* guarantees **mutual exclusion**.
- *empty* and *filled* are for **synchronization**.
  - *empty* represents the number of **empty slots**. The producer blocks when this value is 0, *i.e.*, there is no empty slot, or in other words, the buffer is already full.
  - *filled* represents the number of **filled slots**. The consumer blocks then this value is 0, *i.e.*, there is no filled slot, or in other words, the buffer is empty.
- **Invariant**: *empty* + *filled* = buffer size.

If we change exchange line 6 and 7 in the *producer()* function, however, it will not work. Say the producer first holds the mutex lock, and then is blocked on empty. Then when the consumer wants to enter the critical section, it will be blocked on mutex since the corresponding lock is held by the producer. Then it will neither be able to increment the empty count, so that the producer will be blocked forever,

and the consumer as well. This scenario is called a **deadlock**. It happens when there is a **circular wait**.

The dining philosopher problem

5 philosophers around a table, 5 instant ramen noodles, and 5 chopsticks in between. A philosopher does only two things in his entire life: **eating** and **thinking**. In order to eat, a philosopher needs exactly two chopsticks.



For each philosopher P[i], his left chopstick is C[i] and his right chopstick is C[(i+1)%N]. His left neighbor is C[(i+N-1)%N] and his right neighbor is C[(i+1)%N]. Each philosopher has three states:

- Thinking
- Hungry (trying to get chopsticks)
- Eating (within critical section)

The random backoff approach

```
1  void philosopher(int i) {
2    for (;;) {
3      think();
4      take_chopstick(i);
5      while (try_to_take_chopstick((i + 1) % N) == FAIL) {
             /* it either succeeds and takes the chopstick,
                or fails and does not take the chopstick. */
6        put_chopstick(i);
7        sleep(random());    // wait a random time
8        take_chopstick(i);
9      }
10     eat();               // critical section
11     put_chopstick((i + 1) % N);
12     put_chopstick(i);
13   }
14 }
```

In this approach, we need a semaphore for each chopstick. The functions *take_chopstick(i)* is just to down the corresponding semaphore, and *put_chopstick(i)* is just to up the corresponding semaphore. For each philosopher, after trying to take up one of the chopsticks, it cannot directly try to take up the other one, since if he is not able to take up the other chopstick, he will keep locking the first chopstick he take, causing a **deadlock**. Therefore, he should try to take up the other chopstick, and if he is not able to do so, he puts the first chopstick back, waits for some time, tries to take the first chopstick back, and iterate this process. After eating, the philosopher will just simply release the locks of both chopsticks.

Notice that we need to wait for a **random** amount of time. Otherwise, if there are only two philosophers, and say they will wait for a fixed amount of time 1s if they fail to take the second chopstick. Then they will keep trying and waiting, trying and waiting, in a loop without making any progress. This scenario is called a **livelock**, in which no process is blocked, but they do not make any progress either.

This random backoff approach is good in most applications. In fact, many **network protocols** are designed in this way.

The non-randomized approach

```
int state[N];          // THINKING, HUNGRY, or EATING
semaphore mutex = 1;
semaphore sem[N];      // what are the initial values?
```

```
void take_chopsticks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);         // try to take both chopsticks
  up(&mutex);
  down(&sem[i]);  // block if cannot take chopsticks
}
```

```
void put_chopsticks(int i) {
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);   // see if left neighbor can now eat
  test(RIGHT);  // see if right neighbor can now eat
  up(&mutex);
}
```

```
void test(i) {
  if (state[i] == HUNGRY &&
      state[LEFT] ≠ EATING &&
      state[RIGHT] ≠ EATING) {
    state[i] = EATING;
    up(&sem[i]);  // take both chopsticks
  }
}
```

```
void philosopher(int i) {
  for (;;) {
    think();
    take_chopsticks(i);  // section entry
      // block until I take both chopsticks
    eat();               // critical section
    put_chopsticks(i);   // section exit
  }
}
```

In this approach, we assign a semaphore to each philosopher instead of each chopstick. When a philosopher finishes thinking, he will lock the mutex and mark himself as hungry. Then he will try to take both chopsticks. He will succeed if and only if his left and right philosopher are both not eating, and in this case he marks himself as eating and **ups** his semaphore. Otherwise he fails and does nothing. After such trial, he releases the mutex and **downs** his semaphore. Here if he succeeded taking the chopsticks he should have already upped it, and he will just proceed, but if he did not, he will not be able to down the semaphore and will get blocked.

We first see the case when he succeeded taking his chopsticks. Then he just simply eats in the critical section. After that, he will lock the mutex and mark himself back as thinking. Then he will try to test if his neighbors can eat. If they can, he will help them **up** their semaphore, so that they can return from block and eat. After helping his neighbors he will release the mutex, and proceed to the next **thinking-hungry-eating** loop.

In the case that he failed, as is just discussed, he will be blocked on his own semaphore. This will happen if and only if at least one of his neighbors are eating. After his neighbors finishes eating, they will help him return from block, and so that he can proceed to eat.

This non-randomized approach is **deadlock-free**, **starvation-free**, and allows the **maximum parallelism** for any number of processes. Just to summarize, this problem is again about mutual exclusion and synchronization. **Multiple** process compete for **multiple** resources. Each process needs **two distinct** resources to enter the critical section.

The readers and writers problem

Multiple processes are allowed to **read** a database at the same time, but if some process is **writing** to the database, then no other process can have access to it.

Synchronization

- When a reader is reading, other readers are allowed to read the database.
- When a reader is reading, no writers are allowed to write to the database.
- When a writer is writing, no readers or other writers are allowed to access the database.

Concurrency

- Concurrent access from multiple readers should be allowed.

The sleeping barber problem

There is one barber, one barber chair, and *n* chairs for waiting customers. If there are no customers, the barber falls asleep. When a customer arrive, he wakes up the barber. If more customers arrive while the barber is busy, they will wait as long as there are still empty chairs and leave otherwise.

This problem is similar to various **queueing** situations in a **client-server model**. There is one long-running server process (the barber) and many transient client processes (the customers).

```
semaphore barbers = 0;    // # of barbers ready to cut hair (why 0 instead of 1?)
semaphore customers = 0;  // # of customers waiting for service (not being cut)
int waiting = 0;          // same as above (because there's no way to read a semaphore's value)
semaphore mutex = 1;      // controls access to "waiting"
```

```
void barber() {
  for (;;) {
    down(&customers);  // sleep if no customer
    down(&mutex);
    —waiting;
    up(&barbers);  // a barber is ready to serve
    up(&mutex);
    cut_hair();    // outside critical section
  }
}
```

```
void customer() {
  down(&mutex);
  if (waiting < NUM_CHAIRS) {
    ++waiting;
    up(&customers);  // wake up barber if needed
    up(&mutex);
    down(&barbers);  // wait if no barber is free
    get_haircut();   // outside critical section
  } else {
    up(&mutex);      // leave if the shop is full
  }
}
```

It's data-race-free and deadlock-free. Any starvation?

For the barber, if there is no customer he just waits for customers to come, and otherwise he accepts one customer and decrements the waiting count by 1 (this is necessary and should be in the critical section, since the consumers also need to know the waiting count, and there is no way to read the value of a semaphore). Then he **ups** himself, meaning that he is busy, and cuts the hair.

For each customer, he will first access the waiting count (in the critical section). If he finds that there are empty customer chairs, then he increments the customer number (which can wake up the barber from blocked if needed) and also increments the waiting count by 1. Then he exits the critical section and tries to access the barber. If the barber is free the barber will be set busy by **downing**, and otherwise he will be blocked until the barber is free. He gets the haircut then. Otherwise there are no empty customer chairs, then he will just disappointedly exit the critical section, do nothing and leave.

Summary for IPC problems

IPC problems involve:
- One or more **shared** sources;
- **Multiple processes** that must be synchronized.

A good solution need to:
- Guarantee **mutual exclusion**;
- Guarantee proper **synchronization** among processes;
- Be **deadlock-free** and **starvation-free**.

Deadlocks

Conditions for resource deadlocks

- Mutual exclusion: each resource is either available or currently assigned to exactly one process.
- Hold and wait: processes currently holding resources that were granted earlier can request new resources.
- No preemption: resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- Circular wait: a **cycle** of 2+ processes, each waiting for a resource held by the next member of the cycle.

If we can attack any one of the above conditions, the deadlocks will be structurally impossible.

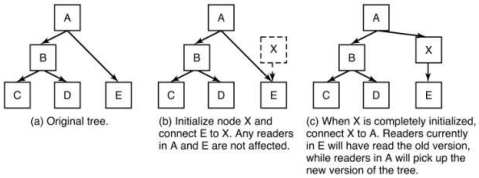
Attack #1: no mutual exclusion

Method 1: make data read only, so processes can use the data concurrently.
Method 2: implement **lock-free** data structures.
- Use powerful hardware instructions to build data structures in a manner that does not require locking.
- We discussed the *test-and-set* instruction when we introduced spinlocks, and another powerful instruction is *compare-and-swap* (CAS):

```
bool compare_and_swap(int *ptr, int expected, int new) {
    // this code executes atomically
    if (*ptr != expected)
        return false;
    *ptr = new;
    return true;
}
```

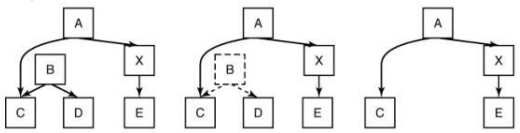
  Using this atomic instruction, we can make sure that it is the current thread/process that modifies the value. Examples include:
  - Atomically increment a value: *do { old = *ptr; } while (!compare_and_swap(ptr, old, old + increment));*
  - Insert a node at the head of a linked list: *do { n -> next = head; } while (!compare_and_swap(&head, n -> next, n));*

Method 3: **read-copy-update (RCU)**.
- We can allow a writer to update the data structure while other processes are still using it.
- A reader would see and traverse **either the old version or the new version**, but not some mash-up.
  - Example: insert a node X in a tree:



(a) Original tree. (b) Initialize node X and connect E to X. Any readers in A and E are not affected. (c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

  - Example: remove a node B from a tree:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version. (e) Wait until we are sure that all readers have left B and D. These nodes cannot be accessed any more. (f) Now we can safely remove B and D

    RCU decouples the **removal** and **reclamation** phases of the update.

Attack #2: no hold-and-wait

Method 1: request all resources at once before starting execution.
- However, a process may **not** know what resources they will need until they have started running.
- Resources are held for longer than needed, thereby **decreasing concurrency**.
- Nevertheless, some mainframe **batch systems** use this approach.

Method 2: release all resources before requesting a new one.

Attack #3: no no-preemption

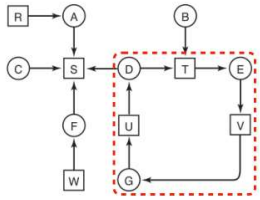
Technically, we cannot forcibly take away a resource that a process already holds. However, in practice, a process can **preempt** their own ownership in a graceful way. We can use **trylock**, as discussed in the dining philosopher problem.

Attack #4: no circular wait

It's probably the most practical and frequently used approach. All the resources are given a **total order** (*i.e.*, global numbering). A process can request only resources **higher** than what it's already holding.

Detecting deadlocks

A deadlock detector runs periodically and build the **resource allocation graph**.



Then DFS is applied to find whether there are cycles in the directed graph.

Threads: lightweight processes

A thread is an **execution entity** within a process. So far, we have only discussed **single-threaded** processes. A **multithreaded process** can have more than one execution in it. For instance, a word processor may have a thread interacting within the user, a thread formatting text in the background, and a thread handling automatic backups.

What is a thread?

- All threads share the same code. A new thread starts a specific **thread function** which can invoke other functions and system calls. However, the thread function does **not** return to its caller.
- All threads **share the same global variables and dynamically allocated memory**. Therefore, a multithreaded process can have **race conditions**.
- Each thread has its own stack of local variables. However, you can still access another thread's stack if you know the memory address.

Why threads?

Threads allow **multitasking** within a process, which can lead to better **performance** and **responsiveness**. Threads are also easier to create an destroy than processes (10--100 times faster). Since threads share the same address space, so sharing data is much easier.

Thread models

- Many-to-one model: implements threads in **user space**.
- One-to-one model: implements threads in the **kernel**.
- Many-to-many model: **hybrid** implementations.

Many-to-one model

The threads library is entirely in user space. The kernel knows that there is a user space, but knows nothing about the threads in there. Such implementation is used in earlier operating systems.

The advantage is that it does not need OS support, and it is fast since it involves no traps, context switches, cache flushing, etc. However, disadvantages of such an implementation include:
- When a **blocking system call** is invoked, all threads will be blocked.
- **Page faults** (discussed later) in a thread will block the entire process.
- **No preemption** of thread due to the absence of clock interrupts.

One-to-one model

The kernel manages all the threads. Each thread is mapped to a **kernel thread**. Such implementation is used in most operating systems.

The advantage is that, when a thread blocks, the kernel can **schedule** another thread (from either the same process or a different process). However, disadvantages of such an implementation include:
- Creating and destroying threads are **more expensive**.
- What if a multithreaded process forks? When a signal comes in, which thread should handle it?

Many-to-many model

The threads library **multiplexes** (多路复用) user-level threads onto kernel threads. It is a hybrid, **two-level** model, and implemented in many language runtimes (Erlang, Go, Haskell, JVM, ...)

This implementation is the best of both worlds (more flexible), and there will be no restrictions on the number of threads. It is just very complex to implement.

POSIX threads *(man 7 pthreads)*

Description	Process	Thread
Process/thread identification.	`pid_t`	`pthread_t`
Get process/thread ID.	`getpid()`	`pthread_self()`
Create a new process/thread.	`fork()`	`pthread_create()`
Terminate the calling process/thread.	`exit()`	`pthread_exit()`
Wait for a specific process/thread to exit.	`waitpid()`	`pthread_join()`
Send a signal to a process/thread.	`kill()`	`pthread_kill()`
Release the CPU to let another process/thread run.	`sched_yield()`	`pthread_yield()`

Note: need to link the *pthread* library, for instance, *gcc -pthread thread.c -o thread*

```c
void *greet(void *name) {
  printf("Hello, %s!\n", (char *)name);
  pthread_exit(NULL);
}

int main() {
  pthread_t tid;
  pthread_create(&tid, NULL, greet, "world");
  printf("I am still running...\n");
  pthread_join(tid, NULL);
}
```
thread.c

Mutual exclusion and condition variables in thread programming

A **mutex object** of *type pthread_mutex_t* is similar to a binary semaphore, but it cannot be used as counting semaphores. Locking is equivalent to downing, and unlocking is equivalent to upping.

Condition variables of type *pthread_cond_t* are used instead of counting semaphores for synchronization. They allow a thread to wait for a condition to become true.

- The function *pthread_cond_wait(&cond, &mutex)* **waits on** a condition. The calling thread **must have locked** the mutex. When the thread is about to block, mutex will be unlocked automatically and atomically. When the thread is unblocked, mutex will be blocked again, automatically and atomically.
- The function *pthread_cond_signal(&cond)* **signals** a condition. If some threads are blocked on the condition, at least one thread will then be unblocked. Otherwise the signal will just be lost.

The producer-consumer problem with *pthread* library

```c
pthread_mutex_t mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  not_full  = PTHREAD_COND_INITIALIZER;
pthread_cond_t  not_empty = PTHREAD_COND_INITIALIZER;
```
Bounded buffer

```c
void producer() {
  int item;
  for (;;) {
    item = produce_item();
    pthread_mutex_lock(&mutex);
    while (count == BUFFER_SIZE)
      pthread_cond_wait(&not_full, &mutex);
    insert(item);  // put new item in buffer
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
  }
}
```

```c
void consumer() {
  int item;
  for (;;) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
      pthread_cond_wait(&not_empty, &mutex);
    item = remove();  // take item from buffer
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);
    consume_item(item);
  }
}
```

Implementing semaphores using mutexes and condition values

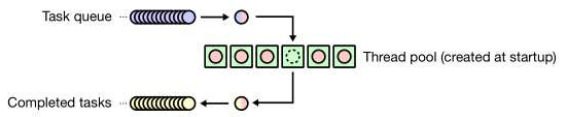
```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
int value;
```

```c
void down() {
  pthread_mutex_lock(&mutex);
  while (value == 0)
    pthread_cond_wait(&cond, &mutex);
  --value;
  pthread_mutex_unlock(&mutex);
}
```

```c
void up() {
  pthread_mutex_lock(&mutex);
  ++value;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

Thread pool

A **thread pool** is a design pattern for achieving concurrency of execution. It maintains **a pool of worker threads** waiting for tasks to be dispatched. If you have a lot of short-lived tasks, frequent creation and destruction of threads are time-consuming. Using a thread pool increases performance and avoids latency in execution.



Thread safety

Some data structures and functions are designed for single-threaded execution. For example, *strtok()* uses a static buffer while parsing a string, thus **not** thread safe. One way to achieve thread safety is to make the function **reentrant**, so that multiple invocations can safely run concurrently. The function may be interrupted in the middle of its execution and then safely be invoked again before its previous invocations complete. This requires saving state information in variables local to each execution. For example, *strtok_r()* is a reentrant version of *strtok()*, which takes an extra argument that is used internally by *strtok_r()* to maintain context between successive calls that parse the same string.

Lecture 3 File Systems

2022年11月10日 6:11

Introduction

Every FS has a unique **layout** of the storage device, which defines what the things stored on the device are and where they are.

The set of FS **operations** defines how the OS should work with the FS layout. In other words, **the OS knows the FS layout** and works with that layout.

The process invokes **system calls**, which in turn invoke the FS operations, to access the storage device.

FS, OS, and storage device

An OS supports one or more FS. An FS can be accessed by more than one OS.

An FS must be stored on a device, but a device may or may not contain any FS.

A storage device is just a dummy container. It does not know what or how many file systems are stored on it. The OS instructs the storage device how to store the data.

What are we going to learn

- **Two basic things**: **files** and **directories**.
- The **layout** of the file systems. The layout may affect the speed of operations on the file systems, the reliability of using the file systems, and the allocation and deallocation of disk spaces.
- **FAT32** (Microsoft) and **ext2/ext3/ext4** (Linux) file systems.
- Interactions between the OS kernel and the processes. Specifically, why Linux can support so many file systems.

Files

Why we need files

Storing information in memory is good because **memory is fast**. However, memory **vanishes** after reboot. Files provide **long-term** information storage (they are persistent). Files can also be **shared objects** for processes to access concurrently.

Filenames

When a file is created, a **name** must be given (extremely important to users). There are naming conventions:

- Length of the name
  - DOS: 8 characters of name + 3 characters of extension.
  - Windows NT, Linux, and macOS: 255 characters of name and extension combined.
- Case sensitivity
  - Windows and newer macOS: case insensitive.
  - Linux and older macOS: case sensitive.
- File extension
  - Application dependent.
  - For *gcc*, the extension is important.
  - For a text editor, the extension is not important.
  - Windows takes the file extension very seriously.

Pathname vs. Filename

- The **pathname** is **unique** across the entire file system.
- The filename is not unique across the file system.
- The filename is unique within the directory that it resides in.

The OS kernel translates the pathname into **a set of data addresses** on the device. That means the pathname is the **key**! If the pathname were not unique, the OS would not be able to locate the data correctly.

What are going to be stored

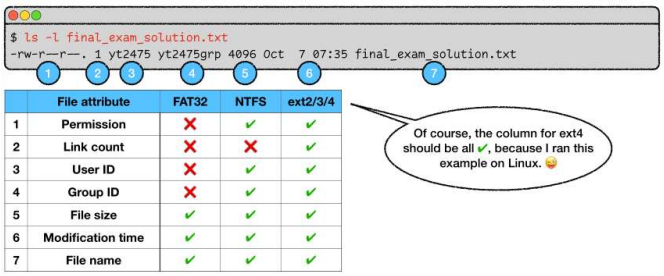
- Filename.
- File content.
- File size: attribute of the file. Note that the file can contain null bytes, so that string operations cannot be used. Therefore, storing the file size is necessary for us to know where the file content terminates.

File attributes

File attributes are important to the file system. A file can have empty content, but not an empty set of attributes.

- Important: size, owner, permission.
- Less important: creation time, modification time, last accessed time.
- Dynamic attributes (appear when the kernel is manipulating the files): file pointer, open file count, …

File attributes are **FS-dependent**, not OS-dependent.



	File attribute	FAT32	NTFS	ext2/3/4
1	Permission	✘	✔	✔
2	Link count	✘	✘	✔
3	User ID	✘	✔	✔
4	Group ID	✘	✔	✔
5	File size	✔	✔	✔
6	Modification time	✔	✔	✔
7	File name	✔	✔	✔

Using the command *stat*, it calls the *stat()* system call, which can obtain a *struct stat* (note that it contains an *off_t st_size* which gives the total size of a file in bytes).

Two examples using *stat()* system call:

- Check whether a file **exists** or not without opening that file:

```
int main(int argc, char **argv) {
  struct stat file_stat;
  if (stat(argv[1], &file_stat) == -1)
    exit(1);

  printf("file size of argv[1] = %d\n", file_stat.st_size);
}
```

- Check whether a filename points to a file or a directory:

```
int main(int argc, char **argv) {
  struct stat file_stat;
  if (stat(argv[1], &file_stat) == -1)
    exit(1);

  if (S_ISDIR(file_stat.st_mode))
    printf("%s is a directory\n", argv[1]);
  if (S_ISREG(file_stat.st_mode))
    printf("%s is a regular file\n", argv[1]);
}
```

Summary

Every file has its **unique pathname**. Its pathname leads you to its **attributes** and the **file content**. These are the two important components of a file, and they are usually stored in separate locations.

File types

- Regular file
  - Text files
  - Binary files
- Directory file
  - Will be discussed in the Directory section.
- Link file
  - A **link** is an **alias** of a file. In Windows, we have the **shortcut**. In Unix/Linux and macOS, we have the **hard link** and **symbolic link**. Both the shortcut (just an ordinary file for Windows) and the symbolic link (a special type of file for Linux) share the same concept. A link file is created and stores the **pathname** of another file. To create a symbolic link, we use *ln -s filename linkname*.
  - For example, in the Lab 3 autograder, *7.in -> 6.in* is a link file, only 4B, which is created using *ln -s 6.in 7.in*
- Device file
  - In Unix/Linux, most devices are represented as files. For example, */dev/had* is the first IDE hard disk drive, */dev/sda* is the first SATA/SCSI hard disk drive, */dev/fd0* is the first floppy drive, etc. Don't try anything like *cat something > /dev/sda*, since you will overwrite and erase all the files on your disk!
  - Unix/Linux creates some **pseudo-devices** for convenience.
    - */dev/zero*: outputs an constant (endless) stream of zeros.
    - */dev/urandom*: outputs a constant (endless) stream of random bytes.
    - */dev/null*: a black hole, so whatever data written to this device will be discarded.

Regular files

Text files: contain text only. They are human-readable, and can be inspected using *cat*. Examples include C programs, HTML documents, JSON files, etc.
Binary files: contain any data. They are not human-readable, and can be inspected using *xxd*. Examples include executable files, images, videos, encoded data, etc.

The command *file* can distinguish the type of a file. When distinguishing binary files, *file* command checks a special pattern called the **magic number**, which is stored at the beginning of most binary files. For example, GIF files always start with the string "GIF87a" or "GIF89a", PDF files always start with "%PDF-", etc.

The command *file* knows all the magic numbers of all file types. They are listed in the file */usr/share/file/magic*. However, *file* command does **not** check the extension of the filename. This means, if we manually modify the first few characters of a file to be a different magic number, *file* command will misread the file type regardless of the real file extension.

The anatomy of opening a file

Step 1: The process sends a pathname to the OS.
Step 2: The OS looks for the **file attributes** of the target file on the disk.
Step 3: The disk returns the file attributes.
Step 4: The OS then associates the attributes to a number, which is called the **file descriptor**.
Step 5: The OS returns the file descriptor to the process.

Opening a file only involves the **pathname** and the **attributes of the file**, instead of the file content!

The anatomy of reading a file

Step 1: The process sends a file descriptor to the OS.
Step 2: The OS reads the file attributes and uses the stored attributes (data location, current-file-position pointer, etc.) to **locate the required data**.
Step 3: The disk returns the required data.
Step 4: The OS keeps the data in a **buffer cache** and copies the data to the process. (The first time we read a file, we need to fetch the file from the disk, but subsequent read operations can just serve the file contents from the buffer cache.)

Sequential access vs. random access

Sequential access: only found in old systems. Bytes or records must be read in order, starting from the beginning. There is no *seek* operation (though *rewind* 倒带 is usually allowed). It is just like a cassette tape.
Random access: applications can read bytes at **any position**, in **any order**. *Seek* operation is supported.

Some system calls

System call	Description
open	Open a file. Create a set of dynamic attributes inside the kernel. This system call may create a new file, and it involves the update in the directory entries.
close	Flush all the outstanding data to the storage media. Clear the dynamic attributes and have the static file attributes updated.
read	Read data from the opened file. The file pointer will be updated after a successful read.
write	Write data to the opened file. The file pointer will be updated after a successful write.
lseek	Change the position of the file pointer.
link	Create a hard link.
symlink	Create a symbolic link.
stat	Read the attributes of a file.
rename	Change the pathname of a file.
unlink	Remove an existing file.

Directories

What is a directory

A directory is also a **file**. Hence, a directory has file content, and for most FS, it also has file attributes. A directory is **an array of directory entries**, recording all the files and directories that belongs to it. In a directory entry, there are names of the stored files, as well as their corresponding attributes (or the locations of their corresponding attributes).

Locating a file using the pathname

Step 1: The process sends the unique pathname "/bin/ls" to the OS.
Step 2: The OS retrieves the directory file of the root directory "/" from the disk.
Step 3: The disk returns the directory file of "/".
Step 4: The OS looks for the name "bin" in the directory file.
Step 5: If found, then the OS retrieves the directory file of "/bin" **using the information of the file attributes of "/bin"** from the disk.
Step 6: The disk returns the directory file of "/bin".
Step 7: The OS looks for the name "ls" in the directory file of "/bin".
Step 8: If found, then the OS knows that the file "/bin/ls" is found, and it starts the previously-discussed procedure to open the file "/bin/ls".

How to read a directory

It is very similar to the open-read-close pattern. Along with the *stat()* system call, you can implement a complete */bin/ls* program.

```
DIR *dir;
struct dirent *entry;
dir = opendir("/");  // open a directory
while ((entry = readdir(dir)) != NULL) {  // read a directory entry until there is no further entries
     printf("%s\n", entry -> d_name);
}
closedir(dir);  // close the opened directory
```

A **directory traversal process** is invoked whenever you open a file, execute a program, or list the content of a directory (etc.)

File creation

File creation is just **updating the directory file**!

Some system calls

System call	Description
mkdir	Create a new, empty directory.
rmdir	Remove an empty directory.
open	Actually, the open() system call is also used to open a directory. Library function: opendir().
close	Actually, the close() system call is also used to close a directory. Library function: closedir().
getdents	Read the directory entry. Library function: readdir().

File system design

What is a file system

A **file system** is about how the OS stores and locates a **file**, and how the OS stores and locates a **directory**. Also, it is about how the OS manages the storage in an **efficient** (about the speed and the storage utilization) and **reliable** (about whether we can get back what have been saved) way.

- Application level (user)
  - What are the **entities** and the **services** that the FS is providing to the applications.
- Logical representation level (user, kernel)
  - How does the FS **represent the entities in a logical way**.
- File organization level (kernel)
  - How does the FS **organize the entities on the storage device**.
- Device control level (kernel)
  - How does the FS **control the devices to read and write the entities**.

Logical representation level

In a file system, the logical representation goes **partially** to the users and **mostly** to the kernel.

- The users have to know what entities and services the FS provides, logically.
- The users do not need to know how the logical entities are implemented.

This is an FS-specific issue, which will be discussed later in detail through the FS of Linux.

File organization

Attempt #1: easy peasy lemon squeezy: just write files sequentially (scan from byte 0, and start writing a new file whenever we find the next empty space on the disk).

- It is **extremely slow** in retrieving data and writing data, since we will need to search through the whole disk space.
- When deleting files, there are holes left on the disk. Then writing a file in that hole may **overwrite** other files.

Attempt #2: keep a table of the starting and ending addresses of all files. Let's call it the file system table.

- I can quickly search any files.
- I can quickly find the next empty space.
- The file system table shows the holes on the disk, so the file overwriting problem can be avoided.
- The free space of the disk is **not used wisely**. A large file may not find a suitable hole, even if there is actually enough space. We call this issue the **external fragmentation** problem. The allocated space is fragmented into chunks, and there are many holes in between.

Attempt #2': align all files in a sequential way again. This means, the allocated space stays at the front of the disk, and the empty space stays at the end of the disk. After such operations, we also correspondingly rewrite the file system table.

- We call this procedure **defragmentation**.
- Defragmentation is very expensive. This procedure involve all the files on the disk, thus very slow. Moreover, the disk becomes **unavailable** during the defragmentation.
- If a file wants to **grow**, but there is no empty space adjacent to it, then the solution to this problem is very expensive.

This file system implementation is called **contiguous** 连续的 **allocation**. It is the best choice when no files will be deleted from the disk, and all file sizes are fixed. Real-life example include: ISO 9660, Joliet (file systems for CD/DVDs). CD/DVDs are read only and will not be modified.

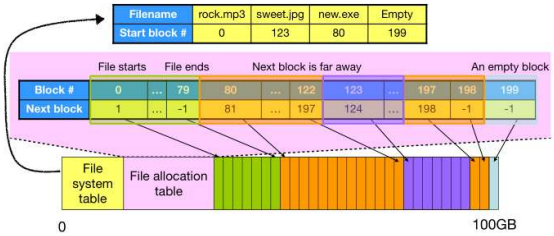
Attempt #3: fille the new file in the empty space **block by block**. That is, divide the disk space into small and equal-sized blocks, and divide the file in the same manner as well. In this way, the new file can be filled into the available empty spaces effectively. But now each file will have multiple starting and ending addresses.

Attempt #3.1: all files are organized as a **linked list** of data blocks. The **free space** is also organized as a linked list.

- It is easy to implement. The pointer value is just the ID (or the address) of the next block.
- The file system table is **smaller** than contiguous allocation.
- No external fragmentation problem.
- Files are allowed to **grow** and **shrink**.
- Free space is well-organized.
- **Random access problem**: it is expensive to access a (random) specific block. In the worst case, if you want to read the last block of your file, you have to traverse from the first block on the disk all the way until you reach the last block. This involves a large number of disk accesses.
- **Internal fragmentation problem**: a file is not always a multiple of the block size. Hence, the last block **may not be fully filled**. This remaining space will be wasted because no other files can be allowed to fill such space.
- **Trade-offs** in choosing the block size:.
  - If the block size is small, the total number of blocks is large. The number of **disk accesses** increases as each file contains more blocks. The list representation itself will also take more space, so there is **less space for real content**.
  - If the block size is large, the internal fragmentation problem will be **worsened**.

Attempt #3.2: store the linked list as a table. We call this approach the **file allocation table (FAT)** approach.

- All the information about the next block are **centralized** as a table.
- The entries in the table are stored **contiguously as an array**.
- Each entry corresponds to **one file allocation block** in the storage device.



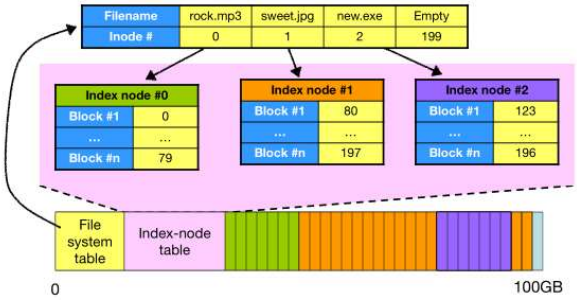
- The size of the FAT depends on the size of the storage device and the block size (their quotient is the number of the entry, then multiply the number of bytes for each entry).
- The file system table and the file allocation table can be **cached in memory** inside the kernel for fast block lookup.
- **No random access problem** because the block lookup is done in memory.
- The **file allocation table** takes a lot of space (but the benefit outweighs this drawback).
- FAT12, FAT16, FAT32 (Microsoft), USB drives, SD cards. NTFS does a similar thing, but more complex (widely used by newer Windows system).

If FAT, a block is called a **cluster**. Different versions of FAT support different number of clusters. For instance, FAT16 means 16 bits for the cluster address length, so there can be 2^16 clusters (note that FAT32 uses 28 bits but not 32 bits). Different versions of FAT also support **different maximum file sizes**. Let *L* be the length of the cluster's address, and let *S* be the size of a cluster, then the maximum file size is *2^L * S*.

See also FAT32 in action.

The FS has to load the entire file allocation table into memory in order to know the locations of allocated data blocks. The file allocation table itself may contain thousands of blocks, which takes a lot of memory. However, we may not need to store the entire set of block location information. If we can break the table into smaller pieces, then the FS only needs to read a few blocks in order to know the locations of all allocated data blocks.

Attempt #4: index node (inode) approach. We can divide the file allocation table based on files, so as to form an index-node table. It contains various index nodes (each file is assigned to an index node), and each index node contains the block locations for that specific file.



- The file system table maintains the mapping from the filenames to the index nodes (inodes).
- An inode stores the addresses of all the data blocks.
- All the inodes are stored in the inode table.
- The size of the inode table determines the number of files that the FS can store.
- A read operation on the **file system table** and a read operation on the **inode table** together locate all the data blocks of a file, and that **consumes less memory** than FAT.
- The size of an inode has to be **variable**, since the length of the list of block addresses is variable. Thus is becomes **hard for the FS to read the inode table efficiently**.

Attempt #4.1: a new idea to index a file's block allocation. We introduce indirect blocks. While direct blocks store data, indirect data blocks store addresses of other blocks. There can be multiple layer of indirect blocks, meaning that indirect blocks can also store addresses of other indirect blocks.

- Each inode has a **fixed size**, while it increases the **supported file size** significantly (exponential to the number of layers).
- At the beginning, the addresses of the data block are stored in the **direct block addresses** of the inode. After the direct block addresses have been used up, the **indirect block addresses** will be used.
- For instance, if there are 12 direct blocks, 1 indirect block, 1 double indirect block, and 1 triple indirect block. Suppose the block size is 2^x bytes and the address length is 4 bytes. Then max file size would be (12+2^x/4+(2^x/4)^2+(2^x/4)^3)2^x=12 2^x+2^(2x-2)+2^(3x-4)+2^(4x-6). That gives, if a block is 4KB, then the maximum file size supported would be 4TB, which is good enough.

See also Ext2/3/4 in action.

Free-space management

We would need to handle the free space on a disk when we delete a file, create a file, or initialize a file system. The goal is to keep track of **every free block** in the file system, and it should be as **FS-independent** as possible.

Attempt #1: allocate data blocks to store the list of addresses of the free blocks. The FS needs to remember the starting block of the list of free blocks (could be hard-coded).

- **Too much space** to maintain such a list of free blocks. For instance on a 128GB hard disk drive, assume that each block has 1KB, then there would be 2^27 blocks, and if a block address has 4 bytes, the list would be 512MB (far too large).

Attempt #2: bitmap. What we need to represent is a series of boolean values (a bit of 0 represents and unallocated block, while a bit of 1 represents an allocated block). Thus, we can construct a **bitmap of block allocation**.

- ext2/3/4 uses this implementation.
- Same assumption as above, there would be 2^27 blocks, and since each block only takes 1 bit, the space needed to maintain a bitmap of allocations is just 16MB.

FAT: In FAT16, there is almost no free space management. The FAT only indicates which clusters are free. In FAT32, there are fields that manage the address of the first free cluster, and the total number of free clusters.

- This needs only little space.
- However, it is slow to calculate the total number of free space (in FAT16).
- Also, it is slow to allocate free clusters for new data. A linear search must be performed from the start of the FAT (in FAT16) or from the start of the first free block (in FAT32).

Supporting containers (*i.e.*, directories)

A **directory** is stored as a set of structured data. From an FS's point of view, a directory is just **a list of directory entries**. Therefore, a directory is considered as a special type of file.

- A directory usually takes only one data block. Therefore, there is a limit on the number of directory entries in each directory.
- A directory entry must at least contain the filename and the information about the data blocks of the file.

The file system should know where the **root directory** is in order to know how to traverse the directory tree.

Storing file attributes

This is FS-dependent. In FAT16 and FAT32, file attributes are stored inside a directory entry. In ext2, ext3, and ext4, file attributes are stored inside an inode.

File system information

We cannot hardcode information such as the size of a block, the number of allocated blocks, the number of free blocks, the location of the root directory, etc. This is because different storage devices have different needs. Therefore, we have to read these information from somewhere in the disk.

FAT, NTFS: the **boot sector** is the **first 512 bytes** of the FS to store all the FS-specific data. For instance, bytes per sector, sectors per cluster, cluster number of the root directory, number of FATs, number of sectors of the reserved area, etc.

ext2/3/4: the **superblock** is a 1024-byte region to store all the FS-specific data. For instance, number of inodes, number of blocks, number of unallocated inodes, number of unallocated blocks, block size, size of each inode structure, etc. It also stores the current mount count and the maximum mount count (a file system check *fsck* will be performed if the maximum mount count is reached).

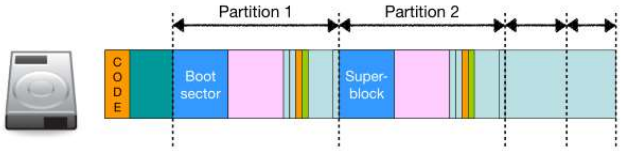
Disk partitions

A **disk partition** is a logical space to host a file system. There may be several file systems contained in the disk, so there may be several disk partitions.

Before the disk partitions, there is an **MBR (Master Boot Record) code** in the very beginning, which specifies which partition to boot, an following the MBR code is a **partition table** storing the first sector and the length and type of a partition.

Why do we need to partition a disk?
- A smaller file system is **more efficient**.
- **Multi-booting**: You can have Linux + macOS + Windows installed on a single disk.
- You can have one logical drive to store the OS-related files, one logical drive to store personal documents, and another logical drive to store movies. Therefore, there would be better data organization.

Formatting a disk means creating and initializing a file system.
- DOS/Windows: format.exe
- Linux: mkfs.fat, mkfs.ext2, mkfs.ext3, mkfs.ext4, …
- macOS: diskutil



FAT32 in action

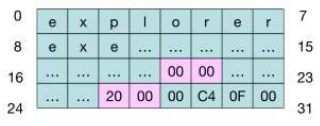
- Cluster size: 512B, 1KB, …, 128KB, 256KB.
- Cluster address length: 28 bits.
- Maximum file size: (4GB - 1) because the file size filed is 4-byte long (0xFFFFFFFF = 16^8).
- Maximum partition size: cluster size * 2^10 * 2^28 bytes. (For example, 32 KB cluster size has maximum partition size 8TB.)

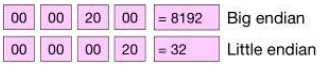
Boot sector
- Beginning of the file system, contain all information about the FS.
- Use *mkfs.fat* to format a disk, and use *fsck.fat* to verify that disk.
- How many bytes per logical sector, how many bytes per cluster -> the size of one block
- How many reserved sectors -> the number of sectors reserved at the very beginning of the FS
- Which byte/sector that the first FAT start at (there may be multiple FATs to survive possible damage of some FAT)
- **Root directory start at cluster 2 (arbitrary size)** -> this is just Microsoft convention, there are no cluster 0 or 1
- Which byte/sector that the data area starts at

Reading a directory
- Step 1: locate the **root directory**. Read the boot sector.
- Step 2: find the **directory entry**. Single dot is the current directory, double dot is the parent directory (parent of root is still cluster 2). Each directory entry has filename, attributes, and a cluster number (which is the cluster number of the first cluster of that file).
- Step 3: follow some cluster, read and return all directory entries.
- Step 4: read and return the entire directory entry structure (a 32-byte structure, Bill Gates did not consider permission and user information, so there are just those simple attributes).

Bytes	Value
0-0	1st character of the filename (0x00 or 0xe5 means unallocated).
1-10	7+3 characters of filename + extension.
11-11	File attributes (e.g., read only, hidden).
12-12	Reserved.
13-19	Creation and access time information.
20-21	High 2 bytes of the first cluster address (0 for FAT16 and FAT12).
22-25	Written time information.
26-27	Low 2 bytes of first cluster address.
28-31	File size.

  - Note that the cluster address are stored in two separate locations, since back in FAT16 and FAT32, they needed only 2 bytes to store the first cluster address. For compatibility, since the last 4 bytes are reserved for the file size and the 4 bytes after the high 2 bytes of the cluster address are reserved for written time information, so the low 2 bytes had to be stored separate from the high 2 bytes.
  - **Big endian** vs **little endian**: the former stores small value in larger addresses (everyday life, SPARC, network protocols, etc.), and the latter stores small value in smaller addresses (Intel, AMD, etc.) **Endianness** is about byte ordering, which is a computer architecture issue. Therefore, suppose we have 20-21 bytes as **00 00**, and 26-27 bytes as **20 00**, in big endian, it is organized as 00 00 20 00 = 8192, but on Intel machines (Bill Gates uses back them), in little endian, they should be instead organized as 00 00 00 20 = 32, as illustrated below:





  - **LFN (long filename)** directory entries: in the old days, Bill Gates set the rule that every file should follow the **8.3 naming scheme**. Yet he removed such a constraint in FAT32 by a backward-compatible but duper ugly design.

Bytes	Value (LFN directory entry)
0-0	Sequence Number.
1-10	File name characters (5 characters in Unicode).
11-11	File attributes — always 0x0F for LFN.
12-12	Reserved.
13-13	Checksum.
14-25	File name characters (6 characters in Unicode).
26-27	Reserved.
28-31	File name characters (2 characters in Unicode).

Reading a file

- Step 1: read the first cluster (data block) from the directory entry.
- Step 2: look up the next cluster in the FAT and read that cluster.
- Step 3: repeat the process until an EOF entry is found in the FAT. Note that EOF > 0x0ffffff8.

Note that the way to read a file may not be sequential. The kernel usually needs to know all the cluster addresses before the actual reading starts. Besides EOF, unallocated (0), and allocated states, a cluster can also be **damaged**, *i.e.*, having bad sectors.

Writing a file

- Step 1: for appending, locate the cluster of the end of the file (just start from the first cluster and follow the FAT until you reach some cluster whose next is EOF).
- Step 2: look up the FSINFO structure to find the next free cluster. It is stored in the reserved space in the beginning of the disk, in the boot sector. It contains the number of free clusters, and the index of the next free cluster.
- Step 3: allocate new cluster by changing the FATs and FSINFO. Note that all/both FATs need to be updated if there are multiple ones of them.

Note: FSINFO structure is useful for finding the next one structure, but if more than one extra cluster is needed, a circular, next-available linear search is performed to find the next unallocated cluster. Nevertheless, after the writing finishes, the kernel still needs to search for one more unallocated cluster, since we need to update FSINFO. Therefore, in modern FS, no one really cares about FSINFO structure any longer.

Deleting a file

- Step 1: find the directory entry and the locations of the clusters in the FATs.
- Step 2: set all the next address fields in the FATs of that file to 0, which means unallocated.
- Step 3: update the FSINFO structure. (As previously mentioned, this is now rarely used. It's just something Bill Gates liked.)
- Step 4: change the first character of the filename to **0xE5**.

However, notice that the file is **not really deleted!** If we perform a search in all the free space, then, we will find all deleted file contents. Those data will persist until the deallocated clusters are **reused** (overwritten). This is a trade-off between performance (during deletion) and security (since back then, Bill Gates did not care about permission and security issues).

How to delete a file securely? See Remembrance of Data Passed: A Study of Disk Sanitization Practices.
How to recover a deleted file? Suppose that the target clusters have not been overwritten. If the file size is no more than one cluster, then the recovery can be easily done, since the first cluster address is still reachable. Note that files with the same suffix may also be found (for example, Yang and Tang after deletion are the same). If the file size is larger than one cluster, other clusters' addresses are all gone. However, because of the next-available search, clusters of a file are **likely to be contiguously allocated**. If not, you had better have the **exact file size** the and the **checksum** of the deleted file beforehand so that you can use a **brute-force** method to recover the file.

Summary

FAT is a simple yet pretty good file system.
- Space efficiency: 4 bytes overhead (FAT entry) per data cluster.

Security is not a consideration in the design.
- It does not maintain owners or permissions in file attributes.
- It employs **lazy deletion**.

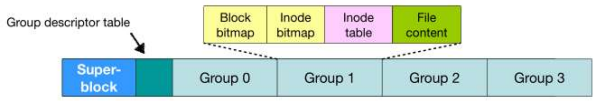
It was widely deployed, and it still is.
- FAT is still dominant in SD cards and USB flash drives due to its high compatibility.

Ext2/3/4 in action

- The primary file system on Linux is ext4 (fourth extended file system).
- The default **block size** is 4KB (configurable from 1KB to 64KB).
- The default **block address** is 4 bytes, but can be configured to be 8 bytes.

Block groups

- The file system is partitioned into **block groups**. Each block group has the same internal structure. The **group descriptor table (GDT)** stores important information including:
  - The start of the block bitmap
  - The start of the inode bitmap
  - The start of the inode table



- The block groups are used for performance and reliability. First, it keeps the metadata and file contents **close together**. Thus, the disk head does not need to travel a long distance. Second, the metadata is scattered, so there is **no single point of failure**.

Superblock

- File system information (inode count, block count, first block, etc.)
- The layout of the FS (blocks per group, inodes per group, etc.)
- The inode usage information (first inode, inode size, etc.)

The metadata structure

- **Block bitmap** stores the block allocation status of the same block group.
- **Inode bitmap** stores the inode allocation status of the same block group.
- **Inode table** stores the contents of all the inodes of the same block group. Each inode has a fixed size specified in the superblock, and the inodes are stored sequentially (as an array).

Inode structure

Bytes	Value
0-1	File type and permission
2-3	User ID
4-7	Lower 32 bits of file sizes in bytes
8-23	Time information
24-25	Group ID
26-27	Link count
...	...
40-87	12 direct data block pointers
88-91	Single indirect block pointer
92-95	Double indirect block pointer
96-99	Triple Indirect block pointer
...	...
108-111	Upper 32 bits of file sizes in bytes

- Note that in the inode table, the index starts from 1 (not 0, and not 2 in FAT). There is no Inode #0.
- The first 10 inodes are **reserved** for special purposes.
    - Inode #1 always points to the bad blocks, Inode #2 always points to the root directory, and Inode #8 usually points to the journal data blocks.

Directory entry

- A directory is stored in the data blocks. A directory is made up of at least two directory entries: . (dot) and .. (dotdot).
    - Inode number (4 bytes)
    - Entry length (2 bytes) (this is 4 + 2 + 2 + filename length including the `\0` character, and remember the byte alignment that rounds this size to some multiple of 4)
    - Name length (2 bytes)
    - File name (up to 255 bytes)
- Note that the directory entries in ext2/3/4 have variable length, depending on the filename lengths.

Link file

- A **hard link** is a directory entry pointing to an existing file, and **no** new file content is created. The command is *ln*.
    - Conceptually, this creates a file with **two filenames**. Deleting only one of the directory entries will not delete the file content.
    - The **link count** field in the inode keeps track of how many directory entries are pointing to that file. When the link count becomes 0, the file content is removed, including the data blocks and the inode.
        - There is not delete system call in Linux, but we use the *unlink()* system call to decrement the link count.
    - Special hard links
        - The directory . (dot) is a hard link to itself.
        - The directory .. (dotdot) is a hard link to its parent directory.
    - When a regular file is created, the link count is always 1. When a directory is created, the link count is always 2 (. of itself and and an entry in its parent directory).
- A **symbolic link** is a new file. Unlike a hard link, a **new inode** is created for a symbolic link. The command is *ln -s*.
    - Where is the **target path** stored?
        - If the path is fewer than 60 characters, it is stored in the 12 direct block and the 3 indirect block pointers, since (12+3)*4=60. There is **no** need to allocate a new data block in this case.
        - If the path is more than 60 characters, one data block is allocated to store the path.

Writing a file

- Step 1: use a linear search in the inode bitmap to find an unallocated inode for the new file.
- Step 2: use a linear search in the block bitmap to find unallocated data blocks for the new file.
- Step 3: read the inode of the root directory, *i.e.*, Inode #2.
- Step 4: following the data block pointers, read the directory entry structure we want (repeat this step until we are going to construct the file).
- Step 5: read the inode for the directory that the file to be created is in, and the corresponding data blocks. Construct the directory entries.
- Step 6: Add a new directory entry with a the first free inode to that directory.

Deleting a file

- Step 1: read the inode and the data blocks following the path, and locate the inode of the file to be deleted.
- Step 2: read that inode and decrement the link count of the file to be deleted. Assume that now the link count has become zero.
- Step 3: deallocate the data block and the inode by setting the corresponding bits in the block bitmap and the inode bitmap.
- Step 4: change the entry length of the **previous** directory entry of the deleted file to the **sum** of its original length and the length of the deleted file. By doing this, when we traverse through the parent directory, we will technically skip the deleted file.

The change in the entry length aims to **skip the deleted entry**. When a new file is created in that directory, the FS should check if there is a hole that is big enough to store the new directory entry. And again, as in FAT32, the file is **not** really deleted. In ext2, the data block pointers will not be removed, but in ext3/4, the data block pointers will be removed.

Linux file system internals

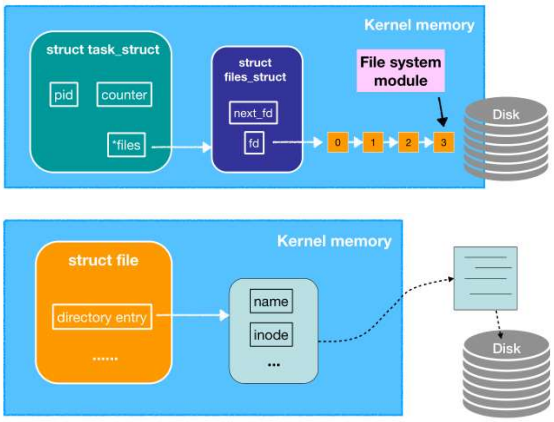
Kernel-process relationship

There is a strong relationship between a process and the opened files. The user program is given a **file descriptor** returned from the *open()* system call as an **abstract representation** of an opened file - it's just a number.

A process has its own set of opened files. It is managed inside the task structure *task_struct*. All the opened files are stored inside a structure of type *struct files_struct*.

By default, evert process initially has three opened files (0 - stdin, 1 - stdout, 2 - stderr). When the process calls the *open()* system call to open a file, the **file system module** reads the **metadata** of the file from the disk. The opened files will be represented as a structure and added to the process. A new file descriptor will be allocated for the new file.

The structure created contains the information about the **directory entry**. In turn, the directory entry contains the **name** and **inode** of that file. Note that the scope of a file descriptor is restricted inside the process. Different processes may use different file descriptors to the same file.



File system module

The **file system module** is the core of Linux file system support. It implements how Linux deals with the disk, and it implements the **virtual file system (VFS)**.

Virtual file systems

For each opened file, a set of **file system functions** is associated with it. They are just function pointers and are called **VFS functions**. The VFS functions are invoked by the FS-related system calls, and points to FS-specific functions (and that's why Linux can support many kinds of file systems, since file pointers can point to different functions specific to different file systems). The value of the function pointers are initialized when the file is opened, and determined by the location of the opened file (for instance, if the opened file is on an ext4 partition, the functions are mapped to the set of ext4 functions).

In short, same set of interface (VFS functions *e.g. open()*), but can invoke different sets of functions (which are FS specific).

- VFS allows the OS to support **multiple file systems** at the same time.
- It is easy to support a new file system in the future.
- However, VFS cannot utilize **special functions** of only some specific file systems, since otherwise the interface may fail for other file systems.

File system performance issues

Data fragmentation

The data fragmentation problem is about the **allocated blocks** of a file. A problem arises when they are **randomly scattered** on a disk.

Some basics about the hard disk drive:
- A hard disk drive has a movable **disk arm**, which has a **disk head** for reading and writing data. Data are stored sector-by-sector on the **platters** using magnetism. The platters are **spinning**, and therefore a spinning of the platters can retrieve a set of **continuous sectors**.
- If the data blocks are packed together, the number of disk head movements will be small because the data is placed in neighboring sectors. However, if the data blocks are **scattered**, the number of disk head movements will be large. Note that the number of disk head movements determines the time in accessing the file!

The cause is the **external fragmentation**, when deleting old data and as a result leaving unallocated spaces between allocated spaces. To deal with data fragmentation better, we need **defragmentation**. In other words, we need to pack the data blocks of a file together.

Windows provides the Microsoft Drive Optimizer. Linux provides *e2defrag* for ext2/3 and *e4defrag* for ext4 file system.

Cache

A **cache** is a temporary storage designed to save time and cash. In the computing world, the **faster** the device is, the **more cache** it costs. Disks are cheaper than the RAM (memory), and the RAM is cheaper than the CPU registers.

For instance, you expect a particular file to be frequently accessed (*e.g.*, the root directory file). To save disk access time, you want to keep it in the memory at all times. That piece of memory is called a **cache**.

However, you do not have a lot of cash to buy a large RAM. Sometimes, when the cache space is running out, the root directory file has to be written back to the disk. The **cache replacement algorithm** determines this, and is controlled by the kernel.

Buffer cache

The **buffer cache** (block cache) includes all the things related to the file system that can be read from the disk, for instance, inode entries, (partial) FAT table, directory entries, and data blocks.
- The data stays in the buffer cache even after the corresponding file is closed! For example, in Lab 3, we test our encoding algorithm using very large input files, and we may want to do it multiple times. If we do not clear it from the buffer cache immediately, then we can read from the memory except for the first time, which is much faster than reading from the disk every single time.

When a disk access is finished (**primary copy**), that data is kept in the **kernel-space** memory for caching purpose (**secondary copy**). The data is kept in kernel space since no matter what, we need to invoke system calls to do those I/O, which happen in kernel mode. When someone wants to read the same piece of data, the secondary copy is returned without any disk access.

When someone wants to write cached data, two approach can be considered.
- **Write-through**: both the primary and the secondary copies are updated together. The OS has to wait for the disk to finish updating. (Windows uses this approach for reliability reasons.)
- **Write-back**: the secondary copy is updated first. The update of the primary copy is **delayed**. (Linux uses this approach for performance issues.)

Cache replacement algorithm

Least recently used (LRU)

Every cached item is associated with an **age**. After a cache access has happened, the age of the accessed item is set to 0, and the age of other items are incremented by 1. For performance reasons, the cached items are sorted according to the ages.

When a read request cannot find the requested item in the cache (**cache miss**), data is read from the disk. If there are enough cache space, the new item is directed written to the cache space. Otherwise, the **least recently used** (oldest) **item** will be written back to the disk.

File system consistency and recovery

When everything is normal, to create a file, the kernel does the following:

1. Read free space management data from the disk.
2. Update free space management data in the disk.
3. Read file allocation information from the disk.
4. Update file allocation information and write it to the disk.
5. Read directory entry from the disk.
6. Update directory entry in the disk.
7. Write data to allocated blocks.

If a crash happens somewhere:

- After Step 1: no problem.
- After Step 2: free space is allocated to nothing -> the FS loses some free space.
- After Step 3: same as the previous case.
- After Step 4: allocated spaces are chained together now, but the space is not associated to any files -> the FS again loses some free space.
- After Step 5: same as the previous case.
- After Step 6: the directory is pointing to uninitialized data, so the content of the file is wrong.

We need to detect those inconsistencies after the OS comes back, and recover from them. There are some file system checking and recovery **tools**: *fsck* (Linux/macOS), *chkdsk* and *scandisk* (Windows).

Error if crashing after Step 2/3

Checking: Check the free space management data against the entire set of allocated blocks.
fsck can construct a **bitmap** of allocated blocks using the inode table. Then it compares this newly-generated bitmap with the bitmap stored on the disk.

Recovering:
fsck can discover missing blocks and then reset the **free space bitmap**, and reset the fields in the superblock (if necessary).

Error if crashing after Step 4/5

Checking: Check the entire directory (entry) tree, and locate the allocated inodes by referring to the inode number of every directory entry. Then, construct the **block allocation bitmap** based on the discovered inodes, and compare with the **free space bitmap** as well as the **inode allocation tables** on the disk. *fsck* can discover missing blocks and **dangling inodes**, *i.e.*, the inodes created without any directory entries referring to.

Recovering:
fsck can reset the **free space bitmap**, reset the fields in the superblock (if necessary), and delete the **dangling inodes** (similar to the garbage collection process).

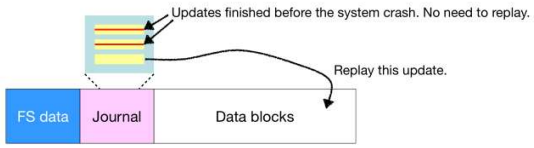
Error if crashing after 6/7

Checking: Journaling file systems. A **journal** is a log book for the file system, which is kept **inside** the file system, *i.e.*, on the disk. In order to make use of the journal, a set of file system becomes an **atomic operation** (either all operations are completed successfully or no operation is completed). A **transaction** marks all the changes that will be done to the file system, and every transaction is written to the journal.

All the changes to the file system are written to the journal first, including changes in the **metadata** and the **file content**. Then, the FS-related system call returns to the user process. Meanwhile, the entries in the journal are **replayed** and the changes are reflected to the actual file system.

The journal stores the update actions. The kernel updates the actual file system according to the actions written in the journal. When a change is **committed** to the disk, they will be marked as done. When all changes are committed, the transaction in the journal will be removed.

Recovering:



Issue: what if a crash happens while the system is writing the journal?

During recovery, we will find that some action in a transaction is incomplete, leading to an incomplete operation. An incomplete operation implies an incomplete transaction. Due to the **atomocity property**, the entire transaction has to be rolled back, *i.e.*, the changes that have been performed according to the transaction should also be undo.

Speed vs reliability

Data is written twice: once in the journal and once in the actual file system. Here are some different **journaling modes** in ext4:

- **Journal mode**: slowest, but most reliable. We write metadata and the file content to the journal first, and then to the file system.
- **Ordered mode**: default, pretty good speed and reliability. The file content is forced to be written to the file system first, then we write the metadata to the journal.
- **Write-back mode**: fastest, but incorrect data may appear. Only metadata is written to the journal.

RAID

- We know that disks are super slow compared with CPU, and disk speed improves slowly compared with CPU.
- Also, in single-disk systems, any disk failure leads to data loss.
- Moreover, a single fast, reliable disk (*e.g.*, for a data center) is extremely expensive.

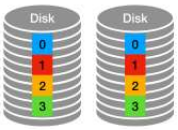
RAID (Redundant Array of Inexpensive/Independent Disks) uses redundancy to improve both performance and reliability.
- Use **redundant array of inexpensive disks** as one storage unit.
- Fast: **simultaneously** read and write disks in the array.
- Reliable: use **parity** (*i.e.*, XOR) to detect and correct errors.

RAID has seven different RAID levels (0--6), achieving different performance and reliability.

RAID 0: non-redundant striping

Data are striped across all disks in an array. There is no parity.
- Good performance: with N disks, we can achieve roughly N times speedup.
- Poor reliability: any disk failure leads to data loss.

RAID 1: mirroring



Keep a **mirrored** (shadow) copy of data.
- Good read performance: roughly 2 times speedup.
- Good reliability: it can recover from one disk failure.
- But high cost: each data disk requires a parity disk.

RAID 4: striping with parity

A set of data blocks (**parity group**) is striped across data disks. The parity disk is the XOR of all other disks. For a single bit, if there is an odd number of 1's, then the parity disk will show 1, and otherwise the parity disk will show 0. So there is an invariant: for each row (the corresponding bit of all disks including the parity disk), there is an even number of 1's.
- Good read performance: roughly N-1 times speedup (but not for random writes)
- Good reliability: it can recover from one disk failure using the XOR.
- But random write performance is poor: every write needs to write data to the parity disk, so heavy load will be posed on the parity disk.

RAID 5: striping with distributed parity

Parity blocks are **distributed** across all disks in a round-robin fashion.
- No too heavy load: there is no single parity disk
- Good performance: even for random writes
- Good reliability: it can recover from one disk failure using the XOR.
- But vulnerable during recovery: we need to read from all other disks during recovery, so they will have much load, and if a second failure happens during this process, this will lead to a total loss.

RAID 6: striping with double distributed parity

Similar to RAID 5, except that there are **two parity** blocks per parity group. Note that we use some parity method (independent of XOR) to compute Q.
- Same advantage as RAID 5.
- Better reliability: it can recover from two disk failures.

Summary

We can build **larger, faster, and more reliable** storage out of inexpensive disks.

Lecture 4 Memory Management

2022年12月14日 8:58

Introduction

We will focus on Intel's memory management (x86) when we talk about practical things

- Process level (logical level)
- Kernel level (mapping from logical level to physical level)
- Device level (physical level)

MM from a process' perspective

- Local variables: stack
- Dynamically allocated memory: heap
- Global variables: data segment
- Program code and constants: code/text segment

- Constants: fixed total space, and read-only.
- Global variables: fixed total space, but not read-only.
- Local variables: unfixed total space, and not read-only.
- Allocated memory: unfixed total space, and not read-only.
- Program code: unfixed total space, and not read-only (there is something called the self-modifying code).

Program code

A process is running on some **program code**. The process can change this relationship using the *exec*()* family of system calls. The running program code is required to be **in the memory**, since the CPU needs to fetch the instructions from the memory for execution.

Global variables and constants

Global variables and constants are **fixed at compile time**. When a new program code starts running (started by an *exec*()* system call), the global variables and the constants are created in the memory.

Global variables and constants are actually managed differently. We can modify the values of the global variables (size unchanged), but constants are **read-only** and should be protected. For instance, assigning a char pointer to "Hello world" and trying to modify the pointer[0] will give a **segmentation fault**.

Local variables

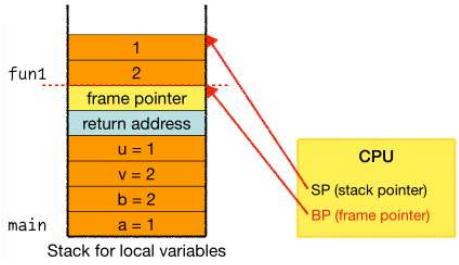
Local variables are **bound to a function only**. When a function is invoked, the local variables are created, and when the functions returns, those local variables are abandoned. Note that the kernel is not involved during the function calls, and the **compiled code** contains all the memory management.

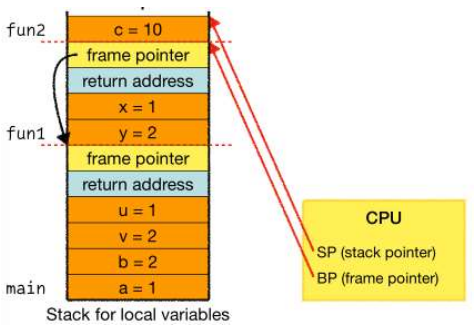
- Local variables are organized as a **stack**. When the *main()* function is invoked, its local variables are pushed onto the stack.
- Function parameter passing is also done through the stack: before the program counter jumps to some function, its **parameters** and the **return address** are pushed onto the stack*. The function refers to the parameters as if they are local variables since they are all in the stack. It also pushes the previous **frame pointer**. If there are new local variables in the called function, these new local variables will also be pushed onto the stack.
- When a function returns, all the local variables and functions parameters are **popped out** (logically, meaning that they are not zeroed out in the memory).
- The return value is stored in the **register EAX****, and the value of EAX will be copied to the return address upon return.
- When the main function finishes, the *exit()* system call is invoked, and all the process' memory is destroyed.

* On x86_64, the first six parameters are passed using registers, and the remaining parameters are pushed onto the stack.
** On x86_64, the return value is stored in the register RAX.

The implementation of the stack

- The CPU has a register called the **stack pointer (SP)**, which always points to the **top** of the stack.
- The stack pointer moves up and down to define the stack space. A function can ask the CPU to read/write anywhere in the stack, not just the top of the stack and not necessarily within the stack frame of the current function.
- The **frame pointer** (a.k.a **based pointer (BP)**) register points to the stack top as it was when the function was just invoked.





- The total size of the local variables changes **dynamically**. The compiler cannot estimate the size of the stack at compile time, since the number of function calls depends on many factors. What the kernel can do is just to reserve a large enough space for the stack.
- If there is no more space left in the stack, this will lead to a **stack overflow**.
- If a process needs more memory space, it will use the dynamically allocated memory, as we will discuss next.

Dynamically allocated memory

Dynamically allocated memory is **bound to the entire process**. The space is often called the **heap** (not organized as a binary heap structure). Once allocated, the memory can be accessed by **every** function of the program. Unlike local variables, the memory allocation is implemented by the **kernel**.

The kernel controls **all** the memory in the system. When you call *malloc()*, the kernel tries to find some memory for you.

- Note that *malloc()* is not a system call, but a function in the C standard library.
- Allocating memory is done by the *brk()* or *mmap()* system call.
  - The *brk()* system call can grow or shrink the allocated area.
  - The *mmap()* system call can map files or devices into memory.
- *malloc()* only manages the free space returned by the kernel.

When you ask *malloc()* for some bytes, it actually asks for more bytes using the *brk()* or *mmap()* system call. Note that it internally needs memory to manage the returned free space (thus some **overhead** for each call). For instance, *malloc*ing for each byte of a large file versus *malloc*ing the whole file once, the first version may run out of memory since there is some overhead for each call.

Segmentation

Segmentation is to view the memory of a process as a set of segments instead of an array of bytes. The access to different segments is controlled by the CPU. Note that if you access a piece of memory that is not allowed to be accessed, the OS will raise a **segmentation fault**! For instance,

- Modifying read-only segments
- Accessing unallocated segments
- Stack overflow (because you are trying to access unallocated heap space)
- Dereferencing a *NULL* pointer (because you are not allowed to visit the memory address 0x00000000)
- etc.

MM from the kernel's perspective

The kernel knows how many processes there are in the system, how much space each process needs, and how much memory there is in the system. But what if the space needed by all processes exceeds the total memory in the system?

Physical memory layout

Recall that the main memory is divided into the kernel space memory (the running kernel and its data) and the user space memory (the processes).

If there are too many processes in the system that causes a main memory overflow, we can use hard disk's space to store the processes. The portion of the hard disk will be reserved to serve as the memory, which is called the **swap**.

Swapping

When a process is scheduled, it needs memory space to run. But the scenario is, the physical memory is fully occupied. Then, one of the idle (not running) processes will be **swapped out** of the main memory, and the swapped-out process will be stored inside the **swap**. That process will be **swapped in** from the disk if it is scheduled again.

Other than supporting more processes in the system, swapping is actually a bad thing: it increases the time in context switching, and it causes the fragmentation problem (in particular, **external fragmentation**, where there are holes but not big enough).

We could use **defragmentation** (a.k.a. **memory compaction**) to pack the processes. However, the memory compaction process takes a lot of time, since the CPU needs to move the memory in a unit-by-unit manner. This further delays context switching.

Address space

When the CPU wants to read/write a piece of memory, a **memory address** is needed. In x86 (32-bit), addresses are 32-bit wide, so the addressable memory space is 2^32=4GB*.

*In x86_64 (64-bit), addresses are 64-bit wide (but only the lower 48 bits are used currently), so the addressable memory space is 2^48=256TB.

In the swapping example, the physical memory location of Process A may be changed when it is rescheduled, which implies that the same variable may have **different addresses** at different times. After a program has been compiled, the variables names are translated into memory addresses. However, due to swapping the physical addresses of the variables may be changed, and there is no way for a program to run in this case.

The fact is: **the compiled code is not using physical addresses, but using logical addressing**. A logical address can be translated to a physical address by the OS kernel or the CPU (there is a chip called the **memory management unit (MMU)** inside the CPU).

A **logical address** is an **offset** inside the memory of a process. Every process has a **base address**, which is a physical address, as well as the lowest address value of the process (the base address may be changed after swapping since it is physical). The kernel maintains **a table of base addresses**. With this approach, the compiled program only needs to maintain the offsets (*i.e.*, the logical addresses) instead of hardcoding the physical addresses, which solves the problem caused by swapping.

What if … a process' allocated memory grow? And recall that we also want to handle external fragmentation problem without memory compaction since it is time-inefficient.

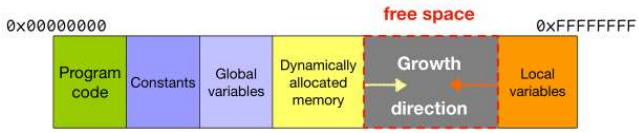
Virtual memory

Virtual memory is a memory management technique that solves all the problems previously mentioned:
- External fragmentation problem
- Memory growth problem
- Memory size limit problem

Virtual memory allows:
- The physical address space of processes can be **noncontiguous**.
- Processes can run even when they are only **partially** in the memory.

Virtual memory layout from the process' perspective

Every process has almost the **entire** address space. The heap and the stack grow inside the process' memory, one from the beginning and one from the end, growing in opposite directions.

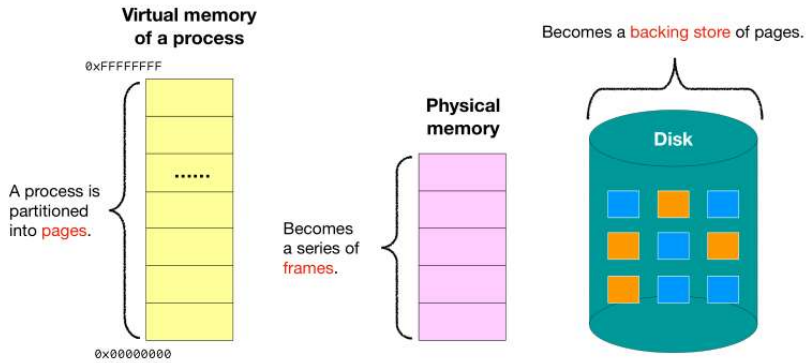


Virtual memory layout from the kernel's perspective

The physical memory is partitioned into fixed-size (*e.g.*, 4KB) blocks called **frames** (a.k.a. **page frames**). A process' logical memory is also partitioned into block of the same size called **pages**. When a process requires memory, the memory is allocated in terms of pages. Note that this may lead to internal fragmentation.

When a process is loaded into the memory, the memory is anchored to the frames. In order to support a process memory larger than the physical memory size, it is **not necessary** that all the pages of a process are loaded into the physical memory all at once. The missing pages are on the disk, which will be loaded into the frame on demand. Therefore, the virtual memory allows a process' memory to **grow**: the kernel does not allocate **any** page to the free space until *brk()* or *mmap()* is invoked. That is to say, *malloc()* is actually finding you **free pages** for the "dynamically allocated memory" to grow.

Virtual memory internals



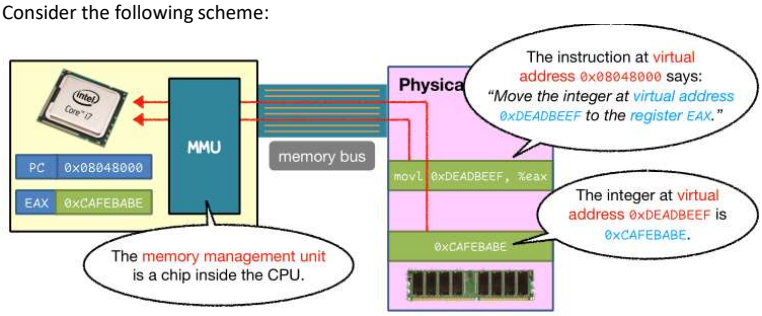
Some of the yellow blocks are pointing to the pink blocks, and some are pointing to the orange blocks. Therefore, the memory of a process (yellow) can **grow** beyond the size of the physical memory (pink).

The CPU runs in a **fetch-decode-execute cycle**.
1. Fetch instruction from the memory.
2. Decode it.
3. Execute it.

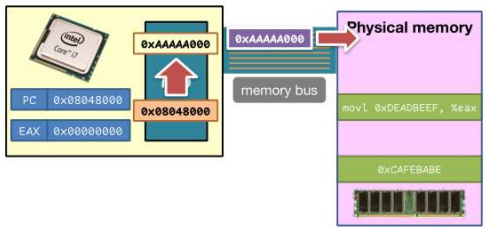
We need to translate virtual addresses into physical addresses, since CPU only works closely with the hardware but it does not know how to read virtual addresses.

Virtual memory support in modern CPUs

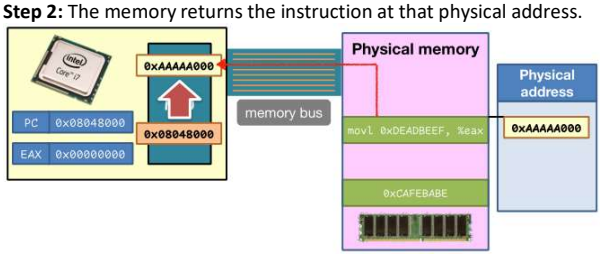
Consider the following scheme:



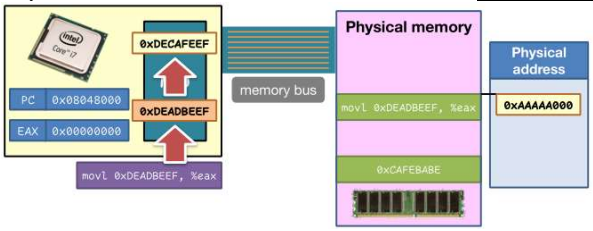
- **Step 1:** To fetch an instruction, the CPU sends the virtual address to the MMU, and the MMU translates it into the physical address



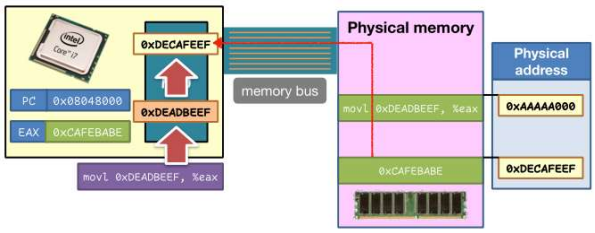
- **Step 2:** The memory returns the instruction at that physical address.



- **Step 3:** The CPU decodes the instruction. An instruction always uses virtual addresses, not physical addresses!

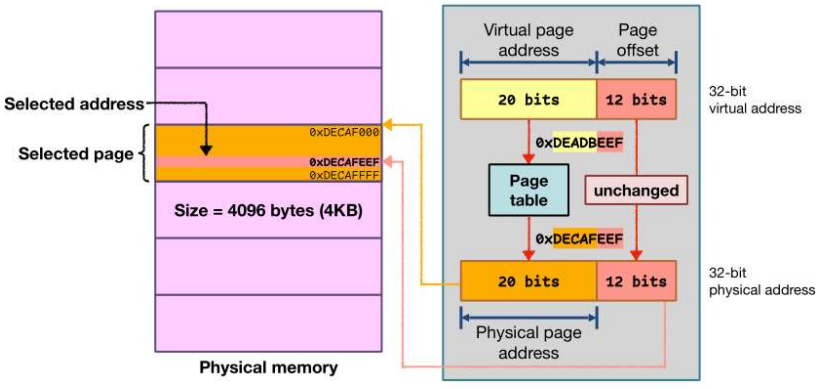


- **Step 4:** With the help of the MMU again, the target memory is retrieved.



Page table

A **page table** stores the **memory mapping**. It tells which pages are in the physical memory, and is stored in the memory and used by the MMU. The content of the page table depends on the paging mechanism.



Note that the 32-bit virtual address is split into the 20-bit virtual page address and the 12-bit page offset (page size is 4KB, so 12 bits). The page table deals only with the **high 20 bits** and translate it into the physical page address.

To summarize, the process only uses the virtual address, in which there are text, initialized data, uninitialized data, allocated heap, and allocated stack. The unallocated zone is between the allocated heap and the allocated stack, and can be occupied by both (from different directions). Allocated memory are broken into pages, while unallocated zone does not occupy any pages. Memory pages are then distributed on the physical memory or the disk.

Demand paging

Demand paging loads a page from the disk to the physical memory when that page is demanded. It is similar to swapping, but the system will not load the entire process into memory, but only the pages required. This allows even **more processes** to be hosted in the system.

The system can support more processes because only part of the process is needed to be in the memory. The kernel can allow **different page frames** to hold **different processes**. However, this may increase the number of disk accesses, since a process may need several pages during the execution.

The **page table** contains the entire view of the physical memory for each process. The column "Page #" denotes the page number, the column "Frame #" denotes the corresponding frame number, and the column "Valid bit" tells if a page is in the memory ('v' for valid and 'i' for invalid (maybe on the disk)). Each row (entry) is a page.

Issue #1

Do we need to access the memory twice for every memory access? Currently, for each access of the memory, CPU talks to the MMU, the MMU looks at the page table, the MMU gets the return of the page table, and finally the MMU fetches the instruction/data from the real page.

Solution: Translation lookaside buffer (TLB), which is an address-translation **cache** inside the MMU. It stores the **recent translations** of virtual memory to physical memory.

- When a TLB hits, MMU does not need to look at and get return from the page table, but refers to the TLB.
- When a TLB misses, we still need to use the original method. But the TLB updates itself, so next time we may have better performance.

Issue #2

Unlike virtual pages, the page table must be contiguous. What if the virtual address space is too large? Consider a system with a 32-bit logical address space: a page table consists of 2^20=1 million entries. Assuming each entry consists of 4 bytes, then each process needs 4MB of contiguous physical address space just for the page table alone!

Solution: Multilevel page tables, which is a page table of page tables. With that, we do not need to keep all page table entries in memory at all times. For instance, instead of having a 20-bit physical page address, we can split it into a 10-bit outer page address and a 10-bit inner page address. If an outer page table entry is allocated, then it points to an inner page table (otherwise we do not even need to create an inner page table).

Back to our story

Our scheme is: some time in a process, the CPU does not know where to fetch the instruction because the instruction is not in any frame. But the page table know about this, since it will show the **valid bit** as **i** (invalid). Then, the kernel steps in, locating the required memory page on the backing storage (*e.g.*, disk). Then, the kernel copies the memory page into a free frame, and update the page table (and **i** to **v**). After all these, the kernel will let the interrupted process resume execution, and this time the instruction can be fetched correctly.

The sum up our previous argument, the swapping decision and operation are handled as if an **exception** has happened. When a page is not found in the main memory, an exception called **page fault** is caught. The **page fault handler** inside the kernel then

- Locates the memory page on the disk
- Chooses the right frame to store the required page in the physical memory
- Updates the page table
- Resumes the program execution

Pure demand paging

The rule is: never bring a page into memory until it is required. Therefore, when a new process is forked, none of the pages of the new process is in the memory. Then, when the child process is executed, a page fault must be raised.

Note that however, this is an extreme solution. The new process produced by the *fork()* system call has the same code/text, stack, and heap as its parent process. Also, it is usually followed by an *exec()* system call right away. While talking about process management, we know that it is stupid to first copy all the memory pages of the parent for the new process (*fork()*) and then throw away all the memory pages of the new process (*exec()*). There is a better solution which we will discuss next.

Copy-on-write

The **copy-on-write** technique allows the parent and the child processes to **share pages** (which means that different virtual memory addresses of different processes may point to the same physical address) after the *fork()* system call. A separate page will be copied and modified only when one of the process wants to write to a shared page.

Page replacement

We have been assuming that there are free frames in the physical memory, but what if all the frames are already occupied? We may need page replacement.

Page replacement happens when both the following two conditions hold:

- The process requests a page that **does not exist** in the physical memory
  - For instance, the CPU is fetching an instruction, and the next instruction may be on the disk.
  - For instance, an instruction accesses a piece of memory. The instruction loads a piece of memory to the register, but the target page may be on the disk.
- There are **no free frames** available in the physical memory
  - The frames can be easily filled up by launching more and more processes, or by allocating more and more memory (note we have to access the page after allocation since otherwise it will not be loaded into the physical memory because it is swapped on demand).

Step 1: decide the victim page in the physical memory. The kernel maintains some data in order to make this decision.
Step 2: swap out the victim and write it back to the disk. The page in the physical memory **overwrites** the page in the swap.
Step 3: swap in the desired frame. The desired page on the disk is written into the victim page.

Page replacement algorithm

- **First-in first-out (FIFO) page replacement:** the oldest page will be swapped out of the frames. The age of a page is counted by the time it is stored in the memory.
  - Too many page faults invoked (whenever the desired page is not in the frame).
- **Optimal page replacement:** replace the page that will not be used for the longest period of time.
  - However, we do not know about the future.
- **LRU page replacement:** every page is assigned an age. Whenever a page is used, its age is reset to 0 and the ages of all other pages are incremented by 1. Replace the page that was the least recently used (*i.e.*, the oldest frame by definition).

Performance issues

Increasing the number of page frames (so increasing the amount of physical memory) may intuitively help increase the performance, but the reality is: **not always**. This is called **Belady's anomaly**, where the number of page frames but the number of page faults also increase, leading to poorer performance.

This is due to the **principle of locality**, which means that a program tends to access the same set of memory locations **repetitively** over a short period of time (*e.g.*, loops). If there are more processes, then more pages will be allocated, then there will be fewer or no free frames, then the number of page faults will increase, and the principle of locality **further** increases the number of page faults.

Therefore, a lot of CPU cycles are wasted in replacing the pages but not doing useful jobs. The more processes in the system, the more likely this will happen. This is called **thrashing** 颠簸.