Operating Systems

Muchang Bahng

January 2024

Contents

Up until now, we've seen the dynamics of how one program works in a computer system. The code, which first resides in the disk, is fetched (through blocks) into memory, and after compiling (precomiling, compiling, assembling, linking), we have a binary. The binary is then loaded into memory in the stack frame, and the CPU executes the instructions. The CPU also has a cache, which stores the most frequently accessed data during the process, taking advantage of locality for efficiency.

Our computer obvious does not just run one program. It runs several, and to run several, we need some control mechanism to manage how these programs interact with the CPU, memory, and disk. For example, one problem is that if we download application A and application B and run their binaries, how do we know whether they share memory addresses and consequently overwrite each other's data?[1] The operating system takes care of these, which manages *processes* that each have their own *virtual memory space*.

Furthermore, consider some of the components of the computer: the RAM, disk, and IO devices like your keyboard and monitor. For security reasons, it is not wise to let the user applications (e.g. Chrome or Slack) control these devices completely. Their power must be restricted in some way.

1. When you have a Chrome window and resize it, Chrome should not be able to modify the pixels outside that window.

2. When you want to print some statement using `printf`,

3. When you're editing a code file with VSCode, you want to limit the application to save to certain parts of the disk.

4. When you are running Chrome and Slack together, you don't want them to read each other's data directly.

This is also for convenience. Say that if you are creating an application that has the option to save files to disk, you don't want to write the hardware backend to write to the disk. You want to just call a function that writes to the disk, and the OS will take care of the rest.

Definition 0.1 (Operating System)

A common confusion is that people think that the **operating system** describes the computer itself, but it is really just another piece of software. What makes this piece of software so special is that it manages every other software in the computer. It provides generally three services:

1. It **multiplexes** the hardware resources. Since there are many applications/programs with finite CPU resources (number of cores) and shared access to storage devices, the OS schedules some sharing mechanism for execution time on CPU cores and manages access to storage devices.
2. It **abstracts** the hardware platform. Since each CPU core simply executes a sequence of instructions, the OS introduces processes and thread abstractions. Furthermore, it introduces *filesystems* (file/directories) on top of raw storage devices.
3. It **protects** software principals from each other. Since many applications from various users are using the CPU, the OS provides isolation between them. It enforces user access permission (read/write) for files.

The OS is booted by the system firmware (BIOS or UEFI), which lives in ROM (sRAM and therefore non-volatile) and copies the OS from a fixed part of the disk, called the **bootloader**, into the RAM, which itself then loads the OS into memory. Once the OS starts running, it loads the rest of itself from disk, discovers and initializes hardware resources, and initializes its data structures and abstractions to make the system ready for users.

[1] This is different from linking, where we have relocation tables to ensure that *object files* do not conflict with each other.

Definition 0.2 (Kernel)

The **kernel** is the actual binary that is loaded into RAM that runs the OS. The kernel code and data resides in a fixed and protected range of addresses, called the **kernel space**, and user programs cannot access kernel space.

1 Process Management

1.1 Control Flow

When we worked with jumps (conditional and unconditional), calls, and returns in assmebly, all of these operations were with respect to the **program state**, which is the isolated environment that the program is in. One program doesn't have any clue of what is going on anywhere else, such as other programs or input/output signals. This means that given what we have learned,

1. programs cannot to write files to the disk (since that is outside the program).
2. programs cannot be terminated by pressing `CTRL + C` on the keyboard.
3. programs cannot receive data that arrives from the disk.
4. programs cannot send data to the monitor to display.
5. programs cannot react accordingly when there is an instruction to divide by 0.[2]

To do all these things, we need to have access to the global **system state**, which the OS has access to.

It turns out that it is impossible for jumps and procedure calls to achieve this, and rather the system needs mechanisms for **exceptional control flow** (i.e. control flow that is not within the regular program state), or commonly referred to as **exceptions**. This requires the CPU to enter into a more powerful state than its current place in the program state, called the kernel state. The actual thing that triggers this is called an **interrupt**, which can come from both the hardware and software. In the kernel state, the CPU can access the hardware and perform operations that the program state cannot to handle these exceptions.

Example 1.1 (Interrupts)

Some examples of how the OS can be interrupted is:

1. when one's WiFi card detects a signal.
2. a hard disk drive may interrupt the OS if a read fails due to a bad sector.
3. an application may request a system call to open a file.
4. If you have 10 applications running on 1 CPU core, you may want the CPU core to run to the next application every 10 milliseconds. So, there may be a system call every 10 milliseconds in each program to the OS to switch to the next application.

Definition 1.1 (Execution Modes)

The CPU helps with this by providing two execution modes, which is determined by a special bit in the CPU called the **mode bit**.

1. In **user mode**, the CPU executes only user-level instructions and accesses only the memory locations that the OS makes available to it. It also restricts which hardware components the CPU can directly access.
2. In **kernel mode**, the CPU executes any instructions and accesses any memory location (including those that store OS instructions and data). It can also directly access hardware components and execute special instructions.

[2] I guess you can use a conditional jump to check if the divisor is 0 and then jump to a different part of the code.

Note that the execution mode is *property of the CPU*!

Example 1.2 (Monitor)

A monitor is really just some device that scans a certain portion of memory at a certain frequency that is higher than the human eye can detect. In user mode, if you try to access this memory buffer, you get an exception. No user mode can access this memory buffer.

Example 1.3 (Amazon.com)

When you are on Amazon to search up some product, you want to type in some keyword in the search bar. The web browser, say Chrome, that you are running it on, runs in user mode. When you type in the keyword, Chrome sends a system call to the OS, triggering the kernel mode which retrieves the keys that you pressed, and redirects it to Chrome. The same goes with the location of your mouse. When you move and click on a product, Chrome sends a system call to the OS, which then receives the mouse location and sends it back to Chrome. The application has no way to directly access the hardware.

Now specifically, how does one enter in this kernel mode? We've already hinted at it before, but to elaborate, there are 4 types of exceptions.

Definition 1.2 (Types of Exceptions/Interrupts)

As we have mentioned, we go into kernel mode through exceptional control flows. To go back from the kernel mode to the user mode after the exception handling is done, the kernel must explicitly give back the control to the user program, which is done with a special instruction, which changes the CPU to the user mode again. At this point, it can return back to user mode at the current instruction, next instruction, or abort it.

1. These control flows can either by **synchronous** (caused by an instruction) or **asynchronous** (caused by some other event external to the processor). Asynchronous interrupts are indicated by setting the processor's interrupt pins.
2. Furthermore, **intentional** exceptions transfer control to the OS to perform some function, and **unintentional** exceptions happen when there is a bug.

This gives us 4 categories of exceptions.

1. Intentional synchronous exceptions are **system calls**, aka **traps** (e.g. `printf`, `open`, `close`, `write`, breakpoint traps, special instructions). It returns control to the next instruction.
2. Unintentional synchronous exceptions are **faults** (possibly recoverable) or **aborts** (unrecoverable) (e.g. invalid or protected address or opcode, page fault, overflow, divide by zero). This automatically triggers the kernel mode which then uses an exception handler to kill the process.
3. Intentional asynchronous exceptions are **software interrupts**, which is when software requests an interrupt to be delivered at a later time (e.g. there's some task you want the kernel to do later).
4. Unintentional asynchronous exceptions are **hardware interrupts** caused by an external event (e.g. IO such as `CTRL + C`, op completed, timers which may switch to another application every 10ms, power fail, keyboard, mouse click, disk, receiving a network packet). Unlike system calls, which come from executing program instructions, hardware interrupts are delivered to the CPU on an **interrupt bus**.

Once a system call or hardware interrupt is finished, the program continues to resume back in user mode.

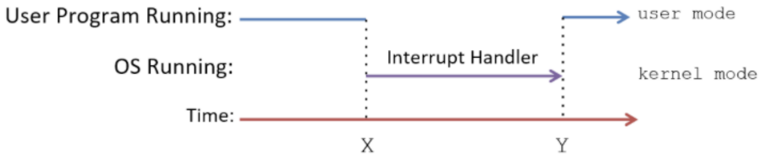


Figure 1: The CPU and interrupts. User code running on the CPU is interrupted (at time X on the time line), and OS interrupt handler code runs. After the OS is done handling the interrupt, user code execution is resumed (at time Y on the time line).

Now the question arises: how does the CPU know where to go when an system call or interrupt occurs? These are done through tables that map some unique ID number to some functionality. These tables are stored in a protected memory space reserved by the kernel.

Definition 1.3 (System Call Table)

This is done through the **system call table**, which is a table of addresses in memory that the CPU can jump to when a system call occurs. Each system call has a unique number k, and the handler function k is called each time system call k occurs.

Example 1.4 (Common System Calls)

Some common system calls, or **syscalls**, are shown below with their unique ID number (in Linux x64).

Table 1: System Call Functions

Number	Name	Description
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

Example 1.5 (Syscalls of Open)

Look at the following objdump file below. The corresponding C code just calls `open(filename, options)` and the corresponding syscall ID is `0x2`. We are simply loading the syscall ID into the `%eax` register (only needs last 32 bits since the syscall IDs are quite small), which is then executed by the `syscall` instruction to go into the kernel mode.

```
00000000000e5d70 <__open>:
...
e5d79:  b8 02 00 00 00       mov    $0x2,%eax     # 2 is the open syscall number
e5d7e:  0f 05                syscall              # return value in %rax
e5d80:  48 3d 01 f0 ff ff   cmp    $0xfffffffffffff001,%rax
...
```

A negative number in `%eax` gives an error corresponding to negative `errorno`. It is also worth mentioning that `%eax` is used rather than `%rdi` or `%rsi` because we need these two parameter registers as arguments for the `open` function itself.

Note that whether we are in the program stack or the kernel stack, we always have stack pointers and other registers to navigate them. In fact, for every CPU core, it has its own set of registers and its own kernel stack.

Example 1.6 (Syscall of Read)

If we have read syscall, then

1. We use the syscall table to go to the trap handler for the read syscall.
2. The handler identifies the block and allocates a buffer.
3. Then it reads the block from the disk, which may take a while (in CPU time) since it is extremely slow for all IO tasks. The CPU, while waiting, can be put to sleep for other processes to run on the CPU. When the disk is done reading, it (the hardware) can send a hardware interrupt to the CPU, telling it that it is done.
4. Then it copies the block to the user buffer and returns from the syscall back into the user mode in the program state.

Definition 1.4 (Exception Table)

This is done through the **exception table**, which is a table of addresses in memory that the CPU can jump to when an exception occurs. Each type of event has a unique exception number k, and the handler function k is called each time exception k occurs.[a]

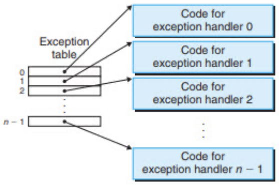

Exception table
0
1
2
n − 1
Code for exception handler 0
Code for exception handler 1
Code for exception handler 2
Code for exception handler n − 1

Figure 2: System call table is stored in a protected memory space reserved by the kernel.

[a] This is similar to a hardware implementation of a switch statement in C.

Example 1.7 (Common Exception Numbers)

Some common exception numbers are listed below.

Table 2: Exception Summary

Exception Number	Description	Exception Class
0	Divide Error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

From the application's point of view, even if an interrupt happens, it just thinks it is running line by line.

Definition 1.5 (Process Address Space)

Interrupts can happen at any time, and one way to efficiently support this execution context switch from user mode to kernel mode is to do the following. At boot time, the OS loads its kernel code at a fixed location in RAM. Every time you create a new program state, the OS initializes a CPU register with the starting address of the OS handler function. On an interrupt, the CPU switches to kernel mode and executes OS interrupt handler code instructions that are accessible at the top addresses in every process's address space. Because every process has the OS mapped to the same location at the top of its address space, the OS interrupt handler code is able to execute quickly in the context of any process that is running on the CPU when an interrupt occurs. This OS code can be accessed only in kernel mode, protecting the OS from user-mode accesses; during regular execution a process runs in user mode and cannot read or write to the OS addresses mapped into the top of its address space.[a]

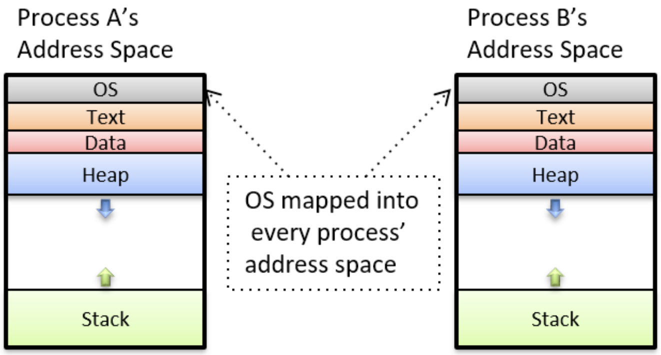

Process A's Address Space
OS
Text
Data
Heap
Stack
Process B's Address Space
OS
Text
Data
Heap
Stack
OS mapped into every process' address space

Figure 3: Process address space: the OS kernel is mapped into the top of every process's address space.

[a] However, due to security reasons where the user space can read kernel space data, this is obsolete.

In summary, a good visual is that each program runs as independent processes, with its own virtual address space (elaborated next) and the OS mediates access to shared resources.

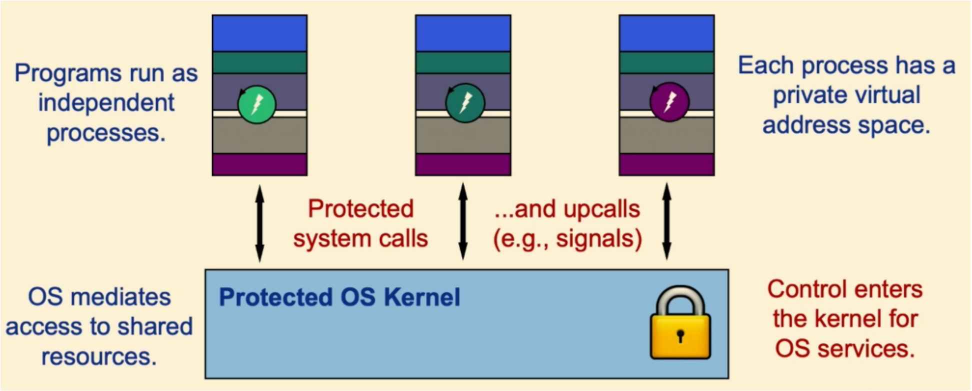

Programs run as independent processes.
Each process has a private virtual address space.
Protected system calls
...and upcalls (e.g., signals)
OS mediates access to shared resources.
Protected OS Kernel
Control enters the kernel for OS services.

Figure 4: Multiple programs running and controlled by an operating system.

Each process can be in one of three states. It can either be currently running on the state, ready to run, or if there is a long IO operation, it can be blocked, which is then unblocked with a hardware interrupt. Usually anything that involves IO puts the state to blocked (e.g. reading data from disk, the keyboard, or the internet). The pool of processes that are concurrently running is the running and ready states.

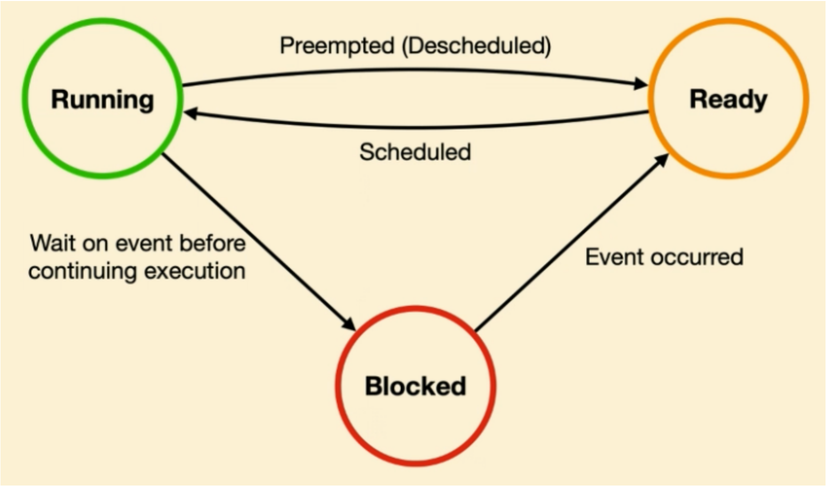


Figure 5: Three states that a single process can be in. The pool of processes that are concurrently running is the running and ready states. The blocked state is waiting to be put back into this pool by a hardware interrupt.

Example 1.8 (Running a Binary)

Therefore, to run a binary file `a.out`,

1. The kernel first loads the binary file from disk into RAM.
2. Then the OS kernel creates a new process with its own virtual memory stack and its global variables, etc.
3. Then the CPU's `%rip` register point to the address of the `main` function.
4. The kernel's virtual memory space is mapped to the top of the process's virtual memory space, where it is not visible to the user mode.

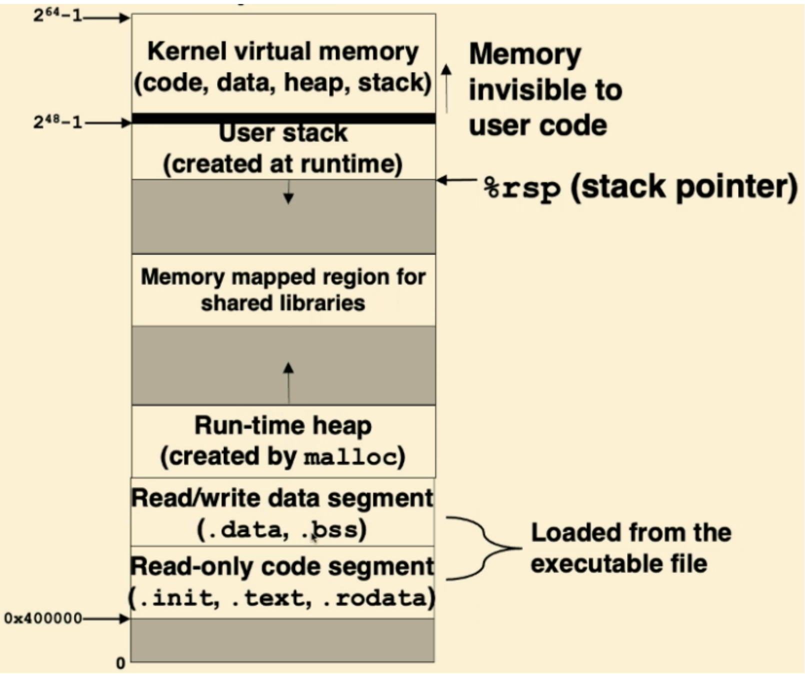

$2^{64}-1$
$2^{48}-1$
Kernel virtual memory (code, data, heap, stack)
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory mapped region for shared libraries
Run-time heap (created by malloc)
Read/write data segment (.data, .bss)
Read-only code segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
0

Figure 6: The kernel's virtual memory space is mapped to the top of the process's virtual memory space.

2 Thread Management

3 Concurrency and Synchronization

So far, we've talked about everything as a sequential process of instructions. In practicality, we have improved this from the memory perspective by implementing caches, virtual memory, and swapping, but in the CPU perspective. In CPUs, we can't just simply increase the clock frequency indefinitely since there are physical limitations.[3] The current trend is to increase parallelism to compute faster, which is implemented with cores and threads. Let's clear some of these definitions up.

Definition 3.1 (Processors, Cores)

In almost every consumer computer, there exists one **processor** (CPU) in it. The CPU can have multiple **cores**. Each core has its own set of registers, L1/L2 cache, and possibly even a shared L3 cache.

Now these cores must run a certain program. Let's define what this means exactly.

Definition 3.2 (Program, Process)

A **program** can be thought of as a binary executable produced after linking. A **process** is a running instance of some program.

1. It is identified by a **process ID (PID)** number.
2. It is run on a CPU core, with its own registers.
3. It has its own virtual address space, containing the code (instructions), heap, and pagetable that maps it to physical memory.

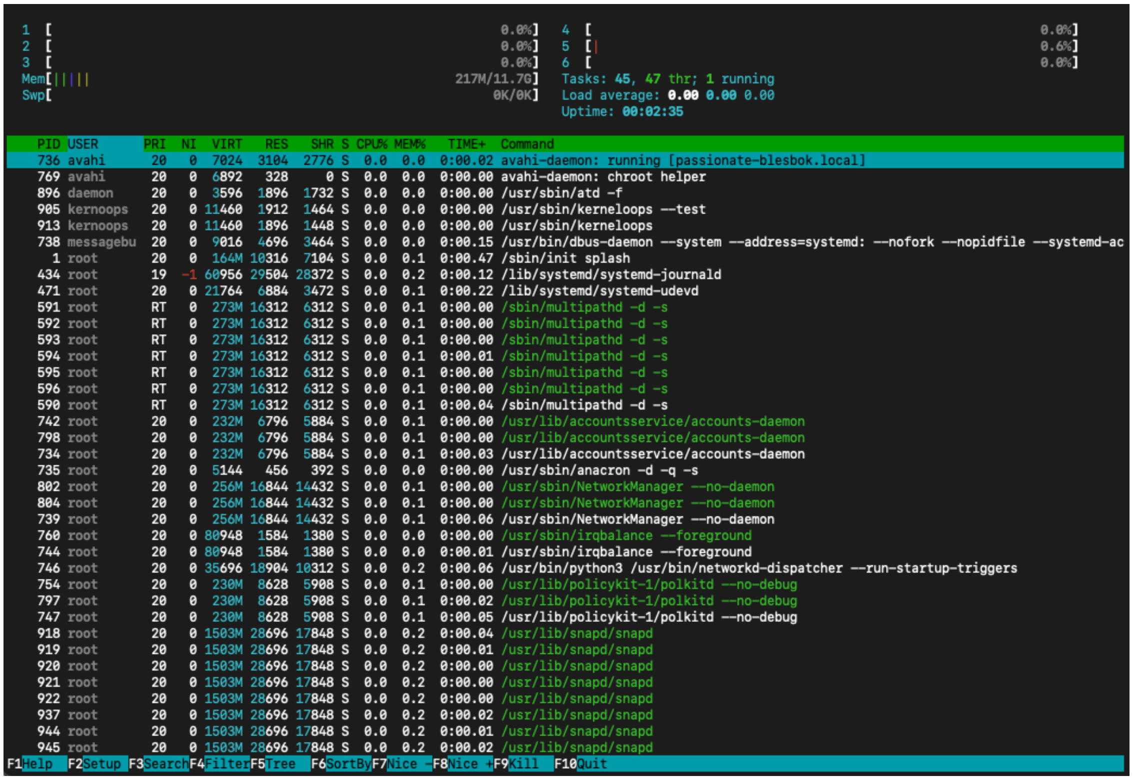
Example 3.1 (Where to look for PIDs)

We can see the PIDs either by using `htop` (for UNIX systems) or by looking at the `/proc` directory in Linux systems. Each directory name represents the PID of the process.

[3] It turns out that power consumption increases faster than clock frequency, so it scales badly.

```
ubuntu@passionate-blesbok:/proc$ ls
1      118    1368  26    44   590  762        cpuinfo      modules
10     119    1369  27    448  599  763        crypto       mounts
101    12     1370  28    45   6    764        devices      net
1012   120    1371  29    46   605  765        diskstats    pagetypeinfo
102    1232   138   3     467  608  769        driver       partitions
103    1234   139   30    468  611  796        execdomains  pressure
1031   1261   14    309   47   612  8          fb           sched_debug
1037   129    15    31    471  613  801        filesystems  schedstat
1038   13     16    32    473  629  810        fs           scsi
104    132    17    33    474  638  822        interrupts   self
1043   1342   18    34    475  651  836        iomem        slabinfo
105    1353   180   35    48   666  850        ioports      softirqs
106    1354   19    356   49   696  872        irq          stat
```

(a) You can see the PIDs of the process by looking at the `/proc` directory. This changes quite often as processes are destroyed and created often, so to maybe track this in real time you might want to run `watch -n 0.1 'ls'`.



```
  1  [                                            0.0%]   4  [                                            0.0%]
  2  [                                            0.0%]   5  [|                                           0.6%]
  3  [                                            0.0%]   6  [                                            0.0%]
  Mem[|||||                                 217M/11.7G]   Tasks: 45, 47 thr; 1 running
  Swp[                                           0K/0K]   Load average: 0.00 0.00 0.00
                                                          Uptime: 00:02:35

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  736 avahi      20   0  7024  3104  2776 S  0.0  0.0  0:00.02 avahi-daemon: running [passionate-blesbok.local]
  769 avahi      20   0  6892   328     0 S  0.0  0.0  0:00.00 avahi-daemon: chroot helper
  896 daemon     20   0  3596  1896  1732 S  0.0  0.0  0:00.00 /usr/sbin/atd -f
  905 kernoops   20   0 11460  1912  1464 S  0.0  0.0  0:00.00 /usr/sbin/kerneloops --test
  913 kernoops   20   0 11460  1896  1448 S  0.0  0.0  0:00.00 /usr/sbin/kerneloops
  738 messagebu  20   0  9016  4696  3464 S  0.0  0.0  0:00.15 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-ac
    1 root       20   0  164M 10316  7104 S  0.0  0.1  0:00.47 /sbin/init splash
  434 root       19  -1 60956 29504 28372 S  0.0  0.2  0:00.12 /lib/systemd/systemd-journald
  471 root       20   0 21764  6884  3472 S  0.0  0.1  0:00.22 /lib/systemd/systemd-udevd
  591 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.00 /sbin/multipathd -d -s
  592 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.00 /sbin/multipathd -d -s
  593 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.00 /sbin/multipathd -d -s
  594 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.01 /sbin/multipathd -d -s
  595 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.00 /sbin/multipathd -d -s
  596 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.00 /sbin/multipathd -d -s
  590 root       RT   0  273M 16312  6312 S  0.0  0.1  0:00.04 /sbin/multipathd -d -s
  742 root       20   0  232M  6796  5884 S  0.0  0.1  0:00.00 /usr/lib/accountsservice/accounts-daemon
  798 root       20   0  232M  6796  5884 S  0.0  0.1  0:00.00 /usr/lib/accountsservice/accounts-daemon
  734 root       20   0  232M  6796  5884 S  0.0  0.1  0:00.03 /usr/lib/accountsservice/accounts-daemon
  735 root       20   0  5144   456   392 S  0.0  0.0  0:00.00 /usr/sbin/anacron -d -q -s
  802 root       20   0  256M 16844 14432 S  0.0  0.1  0:00.00 /usr/sbin/NetworkManager --no-daemon
  804 root       20   0  256M 16844 14432 S  0.0  0.1  0:00.00 /usr/sbin/NetworkManager --no-daemon
  739 root       20   0  256M 16844 14432 S  0.0  0.1  0:00.06 /usr/sbin/NetworkManager --no-daemon
  760 root       20   0 80948  1584  1380 S  0.0  0.0  0:00.00 /usr/sbin/irqbalance --foreground
  744 root       20   0 80948  1584  1380 S  0.0  0.0  0:00.01 /usr/sbin/irqbalance --foreground
  746 root       20   0 35696 18904 10312 S  0.0  0.2  0:00.06 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
  754 root       20   0  230M  8628  5908 S  0.0  0.1  0:00.00 /usr/lib/policykit-1/polkitd --no-debug
  797 root       20   0  230M  8628  5908 S  0.0  0.1  0:00.02 /usr/lib/policykit-1/polkitd --no-debug
  747 root       20   0  230M  8628  5908 S  0.0  0.1  0:00.05 /usr/lib/policykit-1/polkitd --no-debug
  918 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.04 /usr/lib/snapd/snapd
  919 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.01 /usr/lib/snapd/snapd
  920 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.00 /usr/lib/snapd/snapd
  921 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.00 /usr/lib/snapd/snapd
  922 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.00 /usr/lib/snapd/snapd
  937 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.02 /usr/lib/snapd/snapd
  944 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.01 /usr/lib/snapd/snapd
  945 root       20   0 1503M 28696 17848 S  0.0  0.2  0:00.02 /usr/lib/snapd/snapd
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

(b) You can see the PID number of each process (binary) running on the left column when running `htop` on UNIX systems.

Figure 7: Two different ways to see the PIDs of all current processes.

There is a specific numbering to each process.

1. The process with PID 1 is always the kernel process.
2. The smaller PIDs (perhaps less than 300) are also reserved for the kernel, so don't kill it.

If you go into each process, you can see a few things.

```
ubuntu@passionate-blesbok:/proc/750$ sudo ls
attr       comm       fd         map_files   net         pagemap      sessionid     statm     uid_map
autogroup   coredump_filter  fdinfo    maps      ns       personality  setgroups     status    wchan
auxv       cpuset       gid_map   mem       numa_maps     projid_map   smaps      syscall
cgroup       cwd          io        mountinfo   oom_adj    root          smaps_rollup    task
clear_refs  environ       limits    mounts      oom_score     sched        stack        timers
cmdline     exe           loginuid  mountstats  oom_score_adj  schedstat    stat       timerslack_ns
```

Figure 8: There are many files in each PID folder that tells you about the process.

1. To get information about the status of this process, you can `cat status`.
2. The virtual address space is stored in `pagemap`. If you're on an 64-bit machine, this file will be extremely big, so just `cat pagemap` won't work. Therefore you should try `cat maps`, which shows you something like the following.

```
ubuntu@passionate-blesbok:/proc/750$ sudo cat maps
aaaac673c000-aaaac67e3000 r-xp 00000000 08:01 2576     /usr/sbin/rsyslogd
aaaac67f3000-aaaac67f6000 r--p 000a7000 08:01 2576     /usr/sbin/rsyslogd
aaaac67f6000-aaaac67fd000 rw-p 000aa000 08:01 2576     /usr/sbin/rsyslogd
aaaac67fd000-aaaac67fe000 rw-p 00000000 00:00 0
aaaadfe94000-aaaadfed7000 rw-p 00000000 00:00 0                          [heap]
```

In here, you can see that the lefthand column represents the range of virtual memory address. The next column gives us the permissions (read, write, executable, shared/private).

3.1 Process Level Concurrency

Definition 3.3 (Context Switch)

Let us first start off with a single core system. At this point, everything is sequential, and to run all these processes at once[a] we want to use system calls to transition between these processes. This is called a **context switch**.

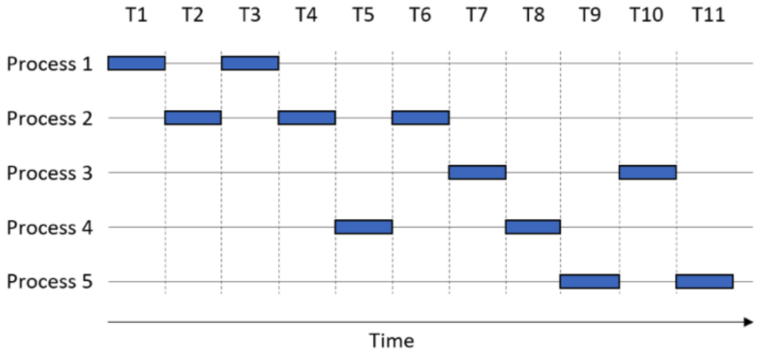


Figure 9: 5 processes may be executed as such on a single core.

Note that due to context switches, the **CPU time**, which is the time is takes to run a process on a CPU, is much shorter than the **wall-clock time**, which is the time a human perceives a process takes to complete.

[a] not programs, since there can be multiple instances of one program, like two Chrome instances. In fact, Chrome produces multiple processes to help run each part of the browser, so one program may translate to multiple processes.

Note that context switches are expensive. To do one, you must essentially replace two things.

1. First, you need to clear out all the register values. This can be done by storing them in the current stack at the VAS, which then gets mapped through the page table into the physical address space.
2. Now the register values (like the instruction and stack pointers) are stored safely in the stack in the VAS, the actual page table must be swapped out too since each process must have its own virtual address space.

Since it is quite expensive to context switch all the time, the simplest thing to do is add more cores, which gives us the double benefit of distributing the process workload *and* having to do less context switches. This is called **physical concurrency**, and given the same workload, it speeds up our computation absolutely. However, this can physically take us so far due to the limited number of cores, and we must go further and use **logical concurrency**.

3.2 Thread Level Concurrency

It turns out that it is much more expensive to reload the page table of a new process rather than clearing out the register values. So, perhaps maybe we can try to implement multiple related "processes" that *share* the same VAS, but have their own execution stream (i.e. own stack and registers). This is precisely the concept of a *thread*.

Definition 3.4 (Threads)

Threads are multiple execution streams within a single process. To summarize them, a thread is an execution context within a process that has a...

1. thread ID
2. its own stack frame
3. its own register context[a]

This is all that is really needed to execute some computation. Now, given that there are some number of threads in process K, they *share* the same virtual address space (VAS), are all under the same PID, share the same code, static data, heap, and file table. The individual stacks living within the VAS are protected from each other to avoid stack overflow.

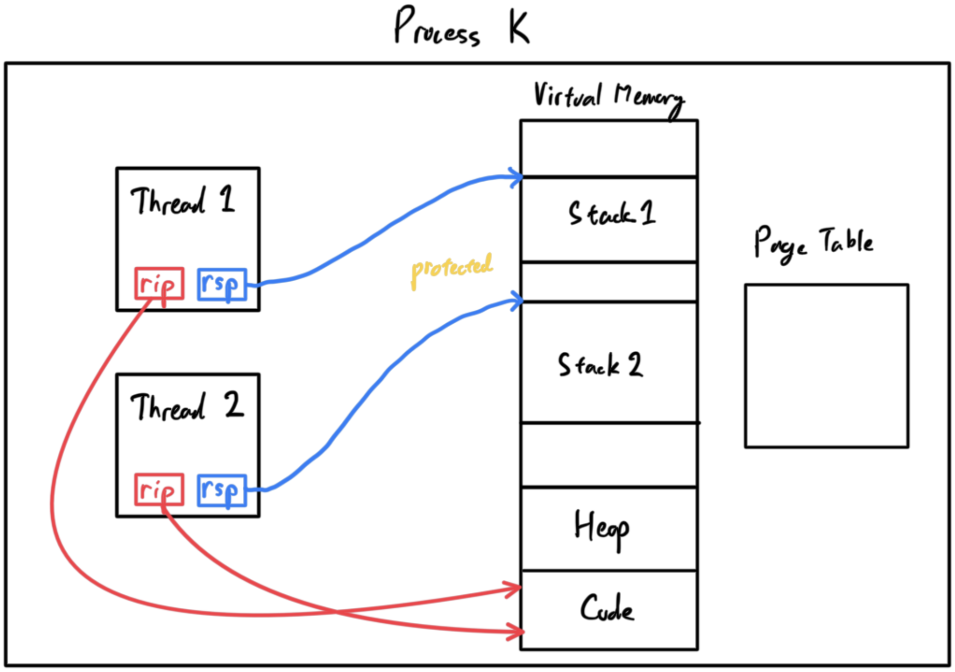


Figure 10: When there are two threads of a single process, the threads share the same virtual memory space. However, they each have their own set of registers. For example, they each have their own instruction pointer that points to the next line of code, along with their own stack pointer. Furthermore, to prevent stack overflow, there are protection mechanisms that prevent one stack from growing past a certain limit into another stack owned by a different thread.

Therefore, we can speed up our program in two ways.

1. If we have one core, we can do context switching faster between each thread (since we only have to load the register values).
2. If we have multiple cores, we can take thread 1 and have it run on one core while taking thread 2 and running it on another core. This is really analogous to having two separate processes on two cores, but these two processes simply share the same VAS, with the same code, data, and heap.

[a] This does not mean that each process has its own physical registers. It has its own *value* that is loaded into the registers. The physical number of registers is determined by the number of CPU cores.

Threads are advantageous for multiple reasons. First, by utilizing multiple cores we can speed up our program to reduce our *CPU time*. However, if we are sharing threads between one core, we're not actually speeding up anything at all but rather reducing our *wall-clock time*. The main speedup that we will feel is that latency heavy tasks will get offloaded to other threads, while more relevant programs can be run on the main thread. This is explained more in the following example.

Example 3.2 (Mobile Application)

If we have a single threaded messaging mobile app, then this is painfully slow since if we want to scroll down our messages while also sending and receiving messages, then we would have to wait for the message to receive from the server, into our disk, and into our memory, before the app responds when scrolling.

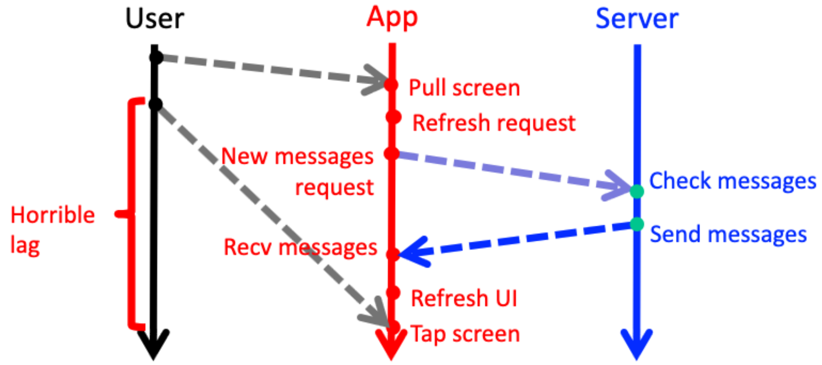


Figure 11: Single threaded app.

However, if we have a multithreaded app with one thread for the app UI and the other one for the server through a background thread, then we can have good UI response time.

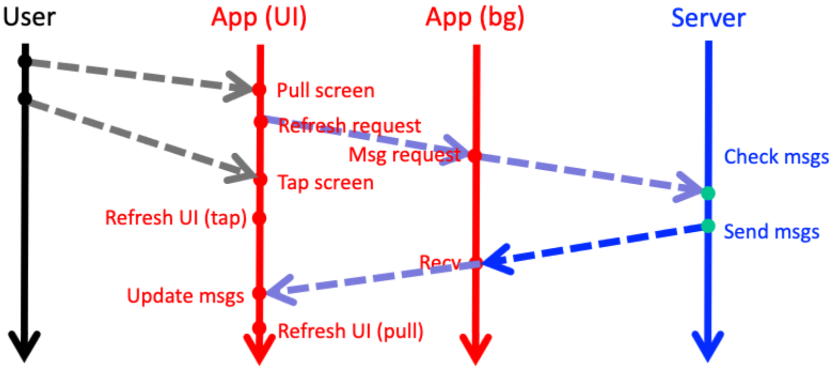


Figure 12: Multithreaded app. Methods on the UI thread must be fast to ensure user satisfaction while anything slow can run on a background thread.

The following law gives us a certain bound on how much parallelization can help us. Note that this does not talk about the responsiveness of an application due to clever thread sharing. It just says given a certain amount of computational task, how much can we reduce the CPU time with parallelization?

Theorem 3.1 (Amdahl's Law)

Say that we have code that runs in 1 second. Given that proportion f of our code can be parallelized, and the speedup for that portion is N, then the new time that our program will take is

$$T_{\text{new}} = (1 - f) + f/N \tag{1}$$

since the sequential part $1 - f$ cannot be sped up, and the remaining parallel part f can be sped up by distributing over N cores. Therefore, defining the speedup as $T_{\text{new}}/T_{\text{old}}$, we get our total parallelized speedup is

$$\frac{1}{(1 - f) + f/N} \tag{2}$$

Note that it is bounded by the sequential portion as $N \to \infty$.

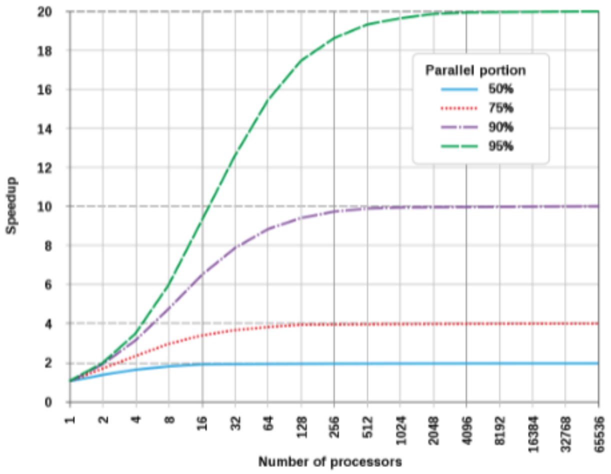


Figure 13: Amdahl's law for different f's over multiple number of processors N.

This is implemented in C with the `pthread.h` library, which is included in the standard library directory and follows the POSIX (Portable Operating System Interface) standard. Essentially, we want to do the following:

1. Define a function that will be called for each thread. It must return a void pointer `void *` and its arguments must also be a void pointer `void *`. Think of this as our new main function for each stack that will be created from each thread.[4] Since we are only restricted to a function taking in one void pointer argument, it is common to define a new struct like `arg_t` that contains all the parameters you need to run each thread. The void pointer can be typecast into the struct pointer at the beginning of each thread function.
2. We create `pthread_t` objects, which are the thread objects.
3. We call the `pthread_create` function that takes in the pointer of the thread object, some settings, the function to be called, and its arguments. At this point, the operating system will determine how these threads will be run, so you can't make any sequential assumptions about them.
4. Then we join them using `pthread_join`, which basically waits until all the threads are complete before `main` continues.

We will show two examples that go over this process. But more importantly, the concurrency of these two examples will show unpredictable behavior.

Example 3.3 (Simply Print out Thread Number)

We can make threads to print out the number. But these aren't really in the same order.

[4] Called a function pointer?

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* thread(void* args) {
  printf("Thread %d\n", *(int*)args);
  return NULL;
}

int main(int argc, char *argv[]) {
  int size = 10;
  pthread_t threads[size];
  int rc, i;

  // thread creation
  for (i = 0; i < size; i++) {
    rc = pthread_create(&threads[i], NULL, thread, &i);
  }

  // join waits for the threads to finish
  for (i = 0; i < size; i++) {
    rc = pthread_join(threads[i], NULL);
  }
return 0;
}
```

```
Thread 1
Thread 2
Thread 6
Thread 3
Thread 8
Thread 5
Thread 2
Thread 4
Thread 3
Thread 3
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
```

Figure 14: Threads output shows that the order in which the functions are called cannot be predicted.

3.3 Atomicity Violation Bugs and Mutex Locks

We see that there are some parts of the code that are not meant to be parallelized.

Definition 3.5 (Atomicity-Violation Bugs)

This bug happens when the desired **atomicity** (indivisibility) among multiple memory accesses is violated.

Example 3.4 (Atomicity-Violation in SQL)

For example, if we have two threads doing the following (in MySQL):

```
Thread1::
if (thd-> proc_info) {
  ...
  fputs(thd->proc_info);
  ...
}

Thread2::
thd->proc_info = NULL;
```

If we pass the if statement but within it, `thd->proc_info` becomes `NULL`, then this would be very bad. Therefore, we should put locks around.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

  Thread1::
  pthread_mutex_lock(&lock);
  if (thd-> proc_info) {
    ...
    fputs(thd->proc_info);
    ...
  }
  pthread_mutex_unlock(&lock);

  Thread2::
  pthread_mutex_lock(&lock);
  thd->proc_info = NULL;
  pthread_mutex_unlock(&lock);
```

Example 3.5 (Incrementing Shared Counter between Two Threads)

The volatile keyword for `counter` means that it can be changed by all threads.

```c
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <stdlib.h>
4  
5  static volatile int counter = 0;
6  
7  void* thread(void* args) {
8    printf("%s : Start \n", (char*)args);
9  
10   for (int i = 0; i < 10 * 1000 * 1000; i++) {
11     counter += 1;
12   }
13 
14   printf("%s : End \n", (char*)args);
15   return NULL;
16 }
17 
18 int main(int argc, char *argv[]) {
19   pthread_t thread1, thread2;
20 
21   int rc;
22   rc = pthread_create(&thread1, NULL, thread, "A");
23   rc = pthread_create(&thread2, NULL, thread, "B");
24 
25   rc = pthread_join(thread1, NULL);
26   rc = pthread_join(thread2, NULL);
27 
28   printf("Counter : %d\n", counter);
29   return 0;
30 }
```

```
A : Start
B : Start
A : End
B : End
Counter : 10229646
.
.
A : Start
B : Start
B : End
A : End
Counter : 14965289
.
.
A : Start
B : Start
A : End
B : End
Counter : 10086690
.
.
.
.
.
.
.
.
.
.
.
```

Figure 15: From looking at the behavior, we can see that the start and end times of thread A and thread B is complete unpredictable. It is only within the sequential nature within each thread (i.e. `X : start` must come before `X : End` that is predictable. More so, after all the increments, the total sum collected by counter isn't even 20 million!). The actual speed and ordering of this is determined at runtime.

To really see what's going on here, we must look into the assmebly code behind this. If we focus on line 11 when the counter is incremented and look at its assembly, we see

```
1  7a7: 8b 05 0a 0b 20 00       mov    0x200b0a(%rip),%eax        # 200c1c <counter>
2  7ad: 83 c0 01                add    $0x1,%eax
3  7b0: 89 05 01 0b 20 00       mov    %eax,0x200b01(%rip)        # 200c1c <counter>
```

What this means is that every thread consists of loading the `counter` value from the instruction pointer plus an offset into `%eax`, then adding 1 to it, and then storing it back to the same memory location (the rip changes, so the memory address between the 1st and 3rd line will be slightly different, but they are the same address). If we have threads 1 and 2, we can have the following possible interweavings:

1. (1 loads, 1 adds, 1 stores, 2 loads, 2 adds, 2 stores). This would result in a +2.
2. (1 loads, 1 adds, 2 loads, 2 adds, 2 stores, 1 stores). This would result in a +1 since thread 2 loads before 1 could store the incremented version.
3. (1 loads, 2 loads, 1 adds, 2 adds, 1 stores, 2 stores). This would also result in a +1 for the same reasons as before.

There is a big problem here: it seems that these things overwrite each other.

Definition 3.6 (Data Race)

We have just seen an example of a **data race**, which occurs if two or more threads concurrently accesses the same memory location with at least one write. The section of code where a data race can occur is called the **critical section**.

So how do we address this challenge of concurrency? This is where locks and mutexes come in.

Definition 3.7 (Lock)

A **lock** is a construct to enforce mutual exclusion in conflicting code sections (critical sections). It is implemented as a special data object in memory. We can use the API methods

1. `acquire()` or `lock()` is called when going into a critical section.
2. and `release()` or `unlock()` is called when going out of a critical section.

If the lock is already acquired by a thread and not released yet, then other threads will not be able to acquire the lock and execute the next instructions. There are two ways this is implemented. First, an incoming thread can wait if another thread holds the lock, called a **spinlock**,[a] or it can be blocked, called a **mutex** (mutual exclusion).[b][c]

[a] This is mainly implemented through kernel and used almost exclusively for OS development, not application development.

[b] This is mainly implemented in user space.

[c] This has a FIFO queue.

To implement this in C, there's a few things that we have to do.

1. First, make a `pthread_mutex_t` global variable.
2. Then, make initialize the mutex before you create the threads and destroy the mutex after you join the threads in the `main` function.
3. Finally, put the specific locks and unlocks in the locations of the functions that the thread calls.

There are few strategies to correct the counter example above, which all produce the correct output `Counter: 20000000`.

1. We can put the lock around the entire for loop of the `thread` function. However, this essentially takes us back to the sequential regime.

```c
static volatile int counter = 0;
pthread_mutex_t mutex; // global declaration of mutex

void* thread(void* args) {
  pthread_mutex_lock(&mutex); //acquire the mutex lock
  for (int i = 0; i < 10 * 1000 * 1000; i++) {
    counter += 1;
  }
  pthread_mutex_unlock(&mutex); //release the mutex lock
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t thread1, thread2;
  int rc;
  rc = pthread_mutex_init(&mutex, NULL); //initialize the mutex
  rc = pthread_create(&thread1, NULL, thread, "A");
  rc = pthread_create(&thread2, NULL, thread, "B");
  rc = pthread_join(thread1, NULL);
  rc = pthread_join(thread2, NULL);
  pthread_mutex_destroy(&mutex); //destroy (free) the mutex
  printf("Counter : %d\n", counter);
  return 0;
}
```

Figure 16: Putting the mutex locks around the entire for loop.

2. A better idea to actually implement parallelization is to put the locks around the line that says `counter += 1;`. This is the critical code that loads the counter value from the stack into the register, increments it by 1, and sends it back to the stack. We should isolate this so that no other threads can execute during these 3 assembly lines.

```c
static volatile int counter = 0;
pthread_mutex_t mutex; // global declaration of mutex

void* thread(void* args) {
  for (int i = 0; i < 10 * 1000 * 1000; i++) {
    pthread_mutex_lock(&mutex); //acquire the mutex lock
    counter += 1;
    pthread_mutex_unlock(&mutex); //release the mutex lock
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t thread1, thread2;
  int rc;
  rc = pthread_mutex_init(&mutex, NULL); //initialize the mutex
  rc = pthread_create(&thread1, NULL, thread, "A");
  rc = pthread_create(&thread2, NULL, thread, "B");
  rc = pthread_join(thread1, NULL);
  rc = pthread_join(thread2, NULL);
  pthread_mutex_destroy(&mutex); //destroy (free) the mutex
  printf("Counter : %d\n", counter);
  return 0;
}
```

Figure 17: Now we put the locks within the counter. However, this is much slower since locking and unlocking are relatively expensive. It runs in 0.274s.

3. The first two tries are not ideal, but what we can do is have each thread store its local work all within each of its stack, and then when it communicates with the shared memory on `counter`, this is where the locks should come in place. This has the double benefit of locking/unlocking very few times, along with protecting the critical section of the code.

```c
static volatile int counter = 0;
pthread_mutex_t mutex; // global declaration of mutex

void* thread(void* args) {
  int my_counter = 0;
  for (int i = 0; i < 10 * 1000 * 1000; i++) {
    my_counter += 1;
  }
  pthread_mutex_lock(&mutex); //acquire the mutex lock
  counter += my_counter;
  pthread_mutex_unlock(&mutex); //release the mutex lock
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t thread1, thread2;
  int rc;
  rc = pthread_mutex_init(&mutex, NULL); //initialize the mutex
  rc = pthread_create(&thread1, NULL, thread, "A");
  rc = pthread_create(&thread2, NULL, thread, "B");
  rc = pthread_join(thread1, NULL);
  rc = pthread_join(thread2, NULL);
  pthread_mutex_destroy(&mutex); //destroy (free) the mutex
  printf("Counter : %d\n", counter);
  return 0;
}
```

Figure 18: In here, we have each thread increment its own local version of the counter and store it in `my_counter`. Then, we increment the global `counter` by the total `my_counter`, which will require one mutex lock. It runs in 0.049s.

Example 3.6 (Inserting into Linked List)

Inserting into a linked list is a sequential process, but what if we want to parallelize it with multiple threads? Consider the following code.

```c
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
} list_t;

void List_Init(list_t *L) {
  L -> head = NULL;
}

void List_Insert(list_t *L, int key) {
  // insert a new node with the key value at the beginning of the list
  node_t *new = malloc(sizeof(node_t));
  assert(new);
  new -> key = key;
  new -> next = L -> head;
  L -> head = new;
}
```

```c

int main(void) {
  list_t L;
  list_t *Lp = &L;
  List_Init(Lp);
  for (int i = 0; i < 10; i++) {
    List_Insert(Lp, i);
  }
  return 0;
}
```

Say that we want to parallelize the creation of the length 10 linked list. Simply initializing some threads and calling `List_Insert` 10 times won't properly create this linked list. This is because two threads may overwrite where `L -> head` points to. The most naive thing to do is to put locks around the whole function, but now we are back to the sequential regime! If we think about it, the `malloc` calls, asserting that there is viable memory, and assigning the input key value to `new -> key` does not overwrite anything else. It is only when we assign `new -> next` to the head value of `L` that things may get overwritten, so we put the locks shown below, while slightly modifying the `list` struct to have the `pthread_mutex_t` attribute.

```c
typedef struct __list_t {
  node_t *head;
  pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
  L -> head = NULL;
  pthread_mutex_init(&L -> lock, NULL);
}

void List_Insert(list_t *L, int key) {
  // insert a new node with the key value at the beginning of the list
  node_t *new = malloc(sizeof(node_t));
  assert(new);
  new -> key = key;
  pthread_mutex_lock(&L->lock)
  new -> next = L -> head;
  L -> head = new;
  pthrea_mutex_unlock(&L->lock)
}
```

3.4 Deadlock Bugs

Locks are extremely useful to segment out a portion of code that should be uninterrupted. However, there are many consequences of misusing it.

Definition 3.8 (Deadlock Bugs)

A **deadlock bug** happens when you make locks such that the program cannot run anymore. These aren't specific to threads, but also processes as well. They require the four preconditions:

1. Mutual exclusion: you must have a lock to begin with to have a deadlock, so this is trivial.
2. Hold and Wait: Threads must have the ability to hold resources (e.g. the philosophers must hold forks which prevent others from taking them) .
3. No Preemption: Preemption refers to the ability to take the fork out of someone else's hand. So

if you have a thread, you can first take lock A, then look at whether lock B is available. If not, then unlock A and try again. However, this can lead to a bunch of threads just simply picking up and putting down forks, causing a *livelock*.

4. Circular Wait: That is, there exists a circular chain of threads such that each thread holds a resource needed by the next thread. A strategy is too define a fixed acquisition order for locks (e.g. lock A always before lock B).

You shouldn't try to hold multiple locks at once, but if you must, you should have a strategy to avoid deadlock. Choosing a lock order is the recommended way.

Example 3.7 (Bank Accounts)

Let's go through an example. Suppose we had the following code below.

```c
struct account {
  pthread_mutex_t lock;
  int balance;
};

struct arg_t {
  struct account fromAcct;
  struct account toAcct;
  int amt;
};

pthread_mutex_t mutex; // global declaration of mutex

void* Transfer(void* args) {

  struct arg_t* data = (struct arg_t*)args;

  struct account* fromAcct = &(data -> fromAcct);
  struct account* toAcct = &(data -> toAcct);
  int amt = data -> amt;


  pthread_mutex_lock(&fromAcct->lock);
  pthread_mutex_lock(&toAcct->lock);

  fromAcct->balance -= amt;
  toAcct->balance += amt;

  pthread_mutex_unlock(&fromAcct->lock);
  pthread_mutex_unlock(&toAcct->lock);

  return NULL;
}
```

Suppose that Threads 0 and 1 are executing concurrently and represent users A and B, respectively. Now consider the situation in which A and B want to transfer money to each other: A wants to transfer 20 dollars to B, while B wants to transfer 40 to A.

Both threads concurrently execute the `Transfer` function. Thread 0 acquires the lock of acctA while Thread 1 acquires the lock of acctB. Now consider what happens. To continue executing, Thread 0 needs to acquire the lock on acctB, which Thread 1 holds. Likewise, Thread 1 needs to acquire the lock on acctA to continue executing, which Thread 0 holds. Since both threads are blocked on each other, they are in deadlock.

This can be simply fixed by rearranging the locks so that each lock/unlock pair surrounds only the balance update statement associated with it.

```
void *Transfer(void *args){
  //argument passing removed to increase readability
  //...

  pthread_mutex_lock(&fromAcct->lock);
  fromAcct->balance -= amt;
  pthread_mutex_unlock(&fromAcct->lock);

  pthread_mutex_lock(&toAcct->lock);
  toAcct->balance += amt;
  pthread_mutex_unlock(&toAcct->lock);

  return NULL;
}
```

Example 3.8 (Dining Philosopher's Problem)

There are 5 philosophers at a roundtable with 5 plates of food with 5 forks. Each philosopher needs two forks (both on their left and on their right) to start eating their plate. However, there can be a deadlock if every philosopher takes the fork on their right and is always waiting for the left fork (this happens due to a circular dependency).

1. One way to resolve this issue is to set an ID to every fork and have the philosophers always take the lower ID fork first before trying to take the higher ID. Then this resolves and at least one philosopher can eat.

Example 3.9 (Set-Intersection Problem)

Again, like the bank account section, we have the following code that can create deadlocks if there is a code that attempts to compute $A \cap B$ and $B \cap A$ at similar times.

```
set_t *set_intersection(set_t *s1, set_t *s2) {
  set_t *result = set_create();
  pthread_mutex_lock(&s1->lock);
  pthread_mutex_lock(&s2->lock);

  for (int i = 0; i < s1->size; i++) {
    if (set_contains(s2, s1->data[i])) {
      set_add(result, s1->data[i]);
    }
  }

  pthread_mutex_unlock(&s2->lock);
  pthread_mutex_unlock(&s1->lock);
  return result;
}
```

Like using the IDs and ordering, we can resolve this by grabbing locks in a defined order, say from high to low. Therefore, we replace the locks as such.

```
if (&m1 > &m2) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
} else {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
}
```

However, this still creates a deadlock if we compute $A \cap A$, so we should place a deadlock.

It turns out that we can make logically equivalent code without locks! By using atomic primitives that have implementations in C, we can create wait-free algorithms. For example, consider the `int CompAndSwap(int* addr, int expected, int new)` function. If `*addr == expected`, then we set `*addr` to `new` and return 1. Otherwise, we do nothing and return 0. Then, the two pieces of code are equivalent, but the advantage of the RHS is that there is never a deadlock.

1. In the left, we pass by pointer to increment a variable.
2. In the right, we first set `old` to the value of `val`, and if the value of `val` is equal to `old` (which may not always be true if some other thread modifies this value), then we set `old = *val` again and increment it by `amt`.

```
void add(int *val, int amt) {
  pthread_mutex_lock(&m);
  *val += amt;
  pthread_mutex_unlock(&m);
}
.
```

```
void add(int *val, int amt) {
  do {
    int old = *val;
  }
  while(!CompAndSwap(val, old, old+amt));
}
```

However, the left hand, despite the risk of deadlocks, is still recommended. The mutex API allows for better distribution of CPU.

3.5 Order Violation Bugs and Condition Variables

Definition 3.9 (Order Violation Bugs)

An **order violation bug** happens when the desired order between two memory accesses is flipped, i.e. A should always be executed before B, but the order is not enforced during execution.

1. For example, one thread that acts on a linked list may already assume that the linked list is initialized when it is not. Therefore, we should have these threads wait until initialization.
2. Another example is if there is a set of producers and consumers that manage the flow of items through a buffer. If there is no good, then consumers cannot consume and they must wait. If the supply is full then the producers must wait. This also requires some process of waiting.

We can use a regular conditional to check, but this may not be efficient due to the following example.

Example 3.10 (Condition Checks with Vanilla While Loops)

Say that you want to call the following function on a linked list, but you must wait until the list is not `NULL`.

```
int getItem(list_t *list) {
  pthread_mutex_lock(&list -> m);
  // TODO: wait until list is non-empty
  int item = list -> head -> item;
```

```c
  list -> head = list -> head -> next;
  pthread_mutex_unlock(&list -> m);
  return item;
}
```

To check whether `list` is not a null pointer, we can simply use a while loop to check.

```c
int tryGetItem(list_t *list, int *out) {
  int success = 0;
  ptherad_mutex_lock(&list -> m);
  if (list -> head) {
    success = 1;
    *out = list -> head -> item;
    list -> head = list -> head -> next;
  }
  pthread_mutex_unlock(&list -> m);
  return success;
}
.
.
```

```c
int getItem( list_t *list) {
  pthread_mutex_lock(&list -> m);
  // TODO: wait until list is non-empty
  while (list -> head == NULL) {
    pthread_mutex_unlock(&list -> m);
    yield();          // optional
    pthread_mutex_lock(&list -> m);
  }
  int item = list -> head -> item;
  list -> head = list -> head -> next;
  pthread_mutex_unlock(&list -> m);
  return item;
}
```

Figure 19: On the LHS, this returns 0 if the retrieval is not successful, and so you must wrap this function within some while loop to check if it is actually successful. On the RHS, we can use the `yield()` function which gives control to the OS to schedule another thread.

This constant checking through a while loop leads to a waste of CPU resources. Therefore, we want to put the thread to sleep while there is no element in `list` so other processes can use the CPU core. This is the motivation behind *conditional variables*.

Definition 3.10 (Condition Variables)

Condition Variables force a thread to be blocked until a particular condition is reached. This construct is useful for scenarios in which a condition must be met before the thread does some work.

1. Every CV is bound to exactly one mutex. This is because the state of a condition, even if true on one thread, can be changed immediately by another thread, so some sort of locking is needed.
2. Condition variables have the type `pthread_cond_t`.
3. To initialize a condition variable, use the `pthread_cond_init` function.
4. To destroy it, use `pthread_cond_destroy`.
5. `pthread_cond_wait(&cond, &mutex)` takes the address of a condition variable `cond` and a mutex `mutex` as its arguments. It causes the calling thread to block on the condition variable cond until another thread signals it (or "wakes" it up).
6. The `pthread_cond_signal(&cond)` function causes the calling thread to unblock (or signal) another thread that is waiting on the condition variable `cond` (based on scheduling priority). If no threads are currently blocked on the condition, then the function has no effect. Unlike `pthread_cond_wait`, the `pthread_cond_signal` function can be called by a thread regardless of whether or not it owns the mutex in which `pthread_cond_wait` is called.

It is important to know about the states that a thread can be in. It can either be currently running/active, ready to run (perhaps through a syscall), or blocked.

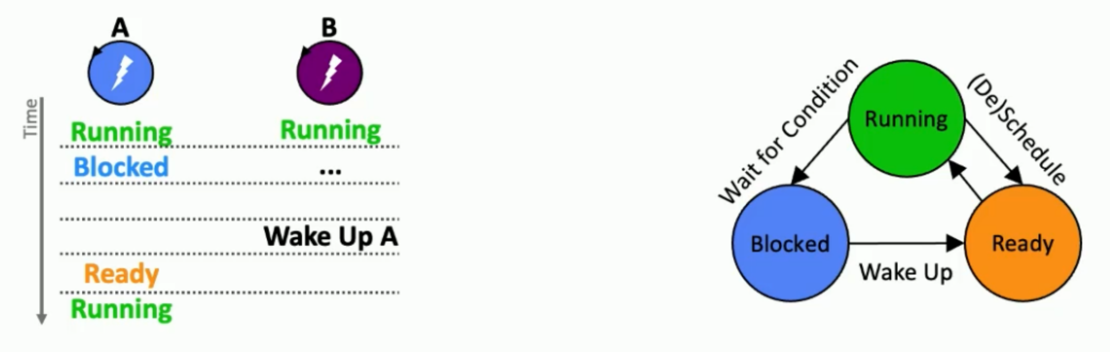


Figure 20: Note that the only way a thread can be blocked is if it is waiting for a condition to happen. If that condition happened and a signal arrives at the thread, then it "wakes up." Condition variables allow these thread to go in and out of the blocked state.

With this, the general design pattern is as such.

Theorem 3.2 (Condition Variable Design Pattern)

To implement this effectively, we must first identify a state that will be accessed by 2+ threads concurrently and add locks to protect the shared state. If we need to wait on some condition, we use condition variables.

```
methodThatWaits() {
  pthread_mutex_lock(&m);

  // Read/write shared state

  while (!checkSharedState()) {
    pthread_cond_wait(&cv, &m);
  }

  // Read/write shared state

  pthread_mutex_unlock(&m);
}
```

```
methodThatSignals() {
  ptherad_mutex_lock(&m);

  // Read/write shared state

  // If checkSharedState() is now true
  pthread_cond_signal(&cv);

  // Read/write shared state

  pthread_mutex_unlock(&m);
}
.
```

The implementation is quite complex at first, so let's go through an example.

Example 3.11 (Soda Machine)

Say that we want to model a soda machine, with consumers taking soda from the machine and producers filling the machine up.

1. We'd like to create some variables that encode the state of the soda machine. This is the **shared state**.

```
static volatile int numSodas;
#define MaxSodas 100;
```

2. We also want to implement *one* lock to protect all shared states, say

```
pthread_mutex_t sodaLock;
```

This allows us to implement mutual exclusion so that only one thread can manipulate the machine (state) at a time.

3. The **ordering constraints** are that the consumer must wait if the machine is empty (CV `hasSoda`), and the producer must wait if the machine is full (CV `hasRoom`).
4. The first thing we must do is make sure that the consumer and producer function has a lock and unlock over its body since both functions modify the vending machine.

```
consumer() {
  pthread_mutex_lock(&sodaLock);

  // take a soda from machine

  pthread_mutex_unlock(&sodaLock);
}
```

```
producer() {
  pthread_mutex_lock(&sodaLock);

  // add a soda to machine

  pthread_mutex_unlock(&sodaLock);
}
```

5. Moreover, the consumer and producer's actions of taking or adding soda is dependent on the state already. For the consumer, it should **wait** if the machine is empty for a signal that notifiies that it is not empty. Once the consumer receives the signal, it takes a soda and can send a signal that the machine is not full to the producer function.

```
consumer() {
  pthread_mutex_lock(&sodaLock);
  // wait if empty
  // take a soda from machine
  // notify that it is not full
  pthread_mutex_unlock(&sodaLock);
}
```

```
producer() {
  pthread_mutex_lock(&sodaLock);
  // wait if full
  // add a soda to machine
  // notify that it is not empty
  pthread_mutex_unlock(&sodaLock);
}
```

6. To put this into code, we finally have

```
consumer() {
  pthread_mutex_lock(&sodaLock);

  while (numSodas == 0) {
    // while empty
    wait(sodaLock, hasSoda);
  }
  numSodas -= 1;    // take soda
  signal(hasRoom);
  pthread_mutex_unlock(&sodaLock);
}
```

```
producer() {
  pthread_mutex_lock(&sodaLock);

  while (numSodas == MaxSodas) {
    wait(sodaLock, hasRoom);
  }

  numSodas += 1;    // add soda
  signal(hasSoda);
  pthread_mutex_unlock(&sodaLock);
}
```

Let's go through this. From the consumer function, we have:

1. The consumer function can start by first acquiring a lock on the `sodaLock` mutex using `pthread_mutex_lock()`. This ensures that only one consumer thread can access the shared resources (e.g., `numSodas`) at a time.
2. It then enters a while loop[a] that checks if `numSodas` is zero. If `numSodas` is zero, it means there are no sodas available for consumption. In this case, the consumer thread calls the `wait()` function, which unlocks the `sodaLock` mutex and waits for a signal on the `hasSoda` condition variable. This allows other threads to acquire the lock and proceed.
3. When the consumer thread receives a signal indicating that sodas are available (`hasSoda` condition variable), it wakes up and reacquires the lock on `sodaLock`.
4. The consumer thread then decrements `numSodas` by 1, indicating the consumption of a soda.
5. After consuming a soda, the consumer thread signals the `hasRoom` condition variable using the `signal()` function. This notifies any waiting producer threads that there is now room available in the soda buffer.
6. Finally, the consumer thread unlocks the `sodaLock` mutex using `pthread_mutex_unlock()`, al-

lowing other threads to access the shared resources.

From the producer function, we have:

1. The producer function can start by first acquiring a lock on the `sodaLock` mutex using `pthread_mutex_lock()`. This ensures that only one producer thread can access the shared resources at a time.
2. It then enters a while loop that checks if `numSodas` is equal to `MaxSodas`. If `numSodas` is equal to `MaxSodas`, it means the soda buffer is full, and the producer cannot add more sodas. In this case, the producer thread calls the `wait()` function, which unlocks the `sodaLock` mutex and waits for a signal on the `hasRoom` condition variable. This allows other threads to acquire the lock and proceed.
3. When the producer thread receives a signal indicating that there is room available in the soda buffer (`hasRoom` condition variable), it wakes up and reacquires the lock on `sodaLock`.
4. The producer thread then increments `numSodas` by 1, indicating the production of a new soda.
5. After producing a soda, the producer thread signals the `hasSoda` condition variable using the `signal()` function. This notifies any waiting consumer threads that a soda is now available for consumption.
6. Finally, the producer thread unlocks the `sodaLock` mutex using `pthread_mutex_unlock()`, allowing other threads to access the shared resources.

Note that we must have a while loop since if the producer ended up broadcasting the `hasSoda` condition to say 10 threads when there are 5 sodas, then 5 of those threads will get a soda while 5 may not, and this possibility should be detected by the while loop.

[a] We want this to be a while loop since we want to reevaluate the condition even after reacquiring the lock.

4 Memory Management

4.1 Virtual Memory

We have mentioned that there is a problem where two different application developers, who have linked their own C files to create binaries, can be installed on one computer and run at the same time. However, the linking has already been finished and the memory addresses of the symbols in each executable are fixed. This can be a problem if there are overlaps in the memory addresses.

Definition 4.1 (Virtual Memory)

The actual main memory of our system is referred to as the **physical memory**. To prevent such overlaps, the kernel and each user process has its own **virtual memory**. That is, there exists a **memory management unit (MMU)** in the CPU that translates virtual addresses to physical addresses through a hashmap.

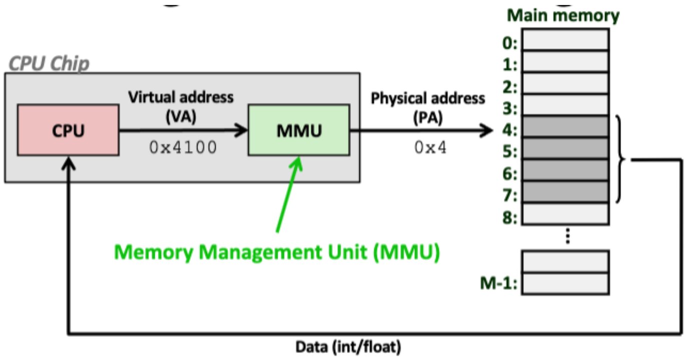


Figure 21: Memory management unit maps each virtual address to a physical address.

This allows the kernel to map the virtual memory of each process to the physical memory.

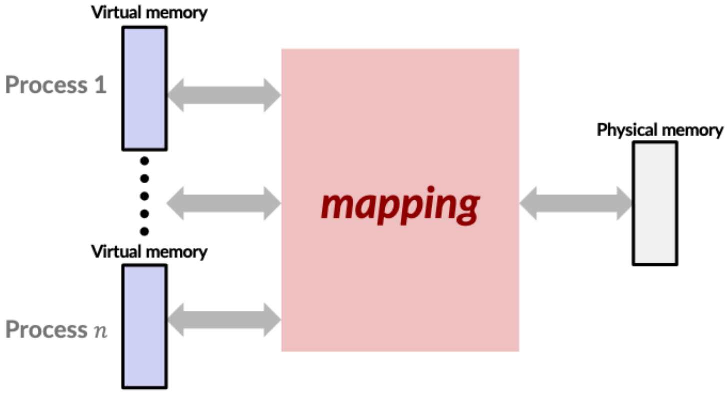


Figure 22: Each process has its own virtual memory space, which is mapped by the MMU to the physical memory space.

Example 4.1 (Virtual and Physical Memory Size)

Given a n-bit machine with 2^m-bytes of memory, $n > m$ and so there are more virtual addresses than physical addresses. If we have a 64-bit machine with 16GB of memory, then there are 2^{64} virtual addresses and $2^3 \cdot 2^{34} = 2^{37}$ bits of physical memory. If there are 8 processes running then there are $8 \cdot 2^{64} = 2^{67}$ bits of virtual memory.

There are many properties of virtual memory that solves a lot of problems and makes things more convenient. The main property is called **indirection** which means that the virtual memory is not the actual physical memory.

1. The first problem is that there are much more virtual addresses than physical addresses. Even storing a table for one process would take up more than all of your RAM. Therefore, for every byte in main memory, there exists one physical address (PA) and zero, one, or more virtual addresses (VA). We will elaborate on the specifics of this implementation later.
2. We also need to have memory management. Every process has its own stack, heap, `.text`, and `.data` sections. We must be able to allocate and deallocate memory and fit this accordingly.
3. We also need to have protection. We need to ensure that one process cannot read or write to another process's memory.
4. While we want isolation, we also want sharing between processes if needed (e.g. signing into Slack using Google on a browser). Furthermore, if there are multiple calls of the `printf` function, we can just have a single copy of the `printf` function in memory rather than having multiple copies for each process. This can be done through the concept of permissions.

Let's talk about how we should actually map these addresses. One property of this mapping is that we want contiguous addresses both in the virtual and the physical level so that we can store arrays, exploit locality, etc. Therefore, we can use larger blocks known as *pages*. Just like how we have divided memory addresses into sections that can be used to map to caches, we can divide the memory addresses into sections that can be used to map to the physical memory. Note that this also takes care of the first problem partially since now we can fit this table in the memory.

Definition 4.2 (Page)

Both in virtual and physical memory, an n-bit address can be divided into a **page number** and an **offset**. The page number is $n - 12$ and the offset is 12 bits. The page number is used to index into a **page table** that maps the page number to a physical address.

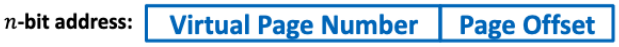


Figure 23: A page is a contiguous block of memory addresses.

While the entire page table is stored in memory (at memory stored by a protected CPU register), a portion of the page table is stored in the CPU cache.

1. The virtual page number (VPN) is equivalent to the block number.
2. The page offset is equivalent to the block offset.

Example 4.2 (Page Number)

In a 64-bit machine with 16GB of RAM, you have $2^{64}/2^{12} = 2^{52}$ virtual pages and $2^{37}/2^{12} = 2^{25}$ physical pages.

Therefore, our translation table is really a map from a virtual page number to a physical page number, rather than a virtual address to a physical address. This is created at runtime. Therefore,

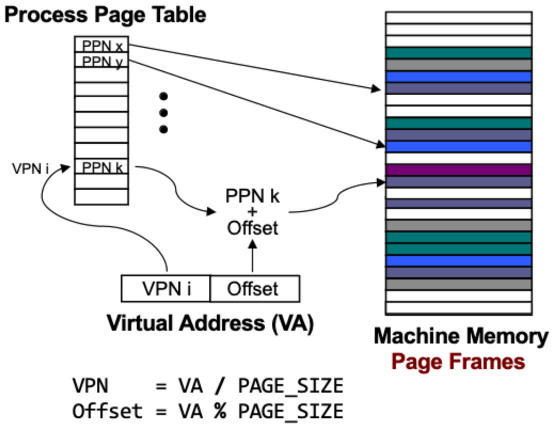
1. The virtual page number VP is mapped through some map M to get the physical page number PP.
2. The virtual offset is the same as the physical offset.

Definition 4.3 (Page Table)

The **page table** is a hashmap that maps the virtual page number to the physical page number defined as the mapping

$$H : \underbrace{(VP, m)}_{\text{64 bits}} \longrightarrow PP \tag{3}$$

Each input-output pair is called a **page table entry (PTE)**, and virtual memory is **fully associative**, meaning that any virtual page can be placed in any physical page, though it requires a large mapping function (the PT), which is different from CPU caches.

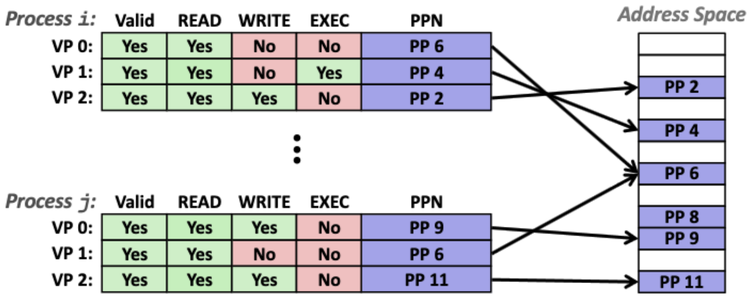


```
VPN    = VA / PAGE_SIZE
Offset = VA % PAGE_SIZE
```

Figure 24: The page table only needs the virtual page number plus the metadata to map to the physical page number. The offset is provided by the virtual memory address itself.

Note that while we want to store the 52-bit VP in the page table, the actual input is still 64-bits, with 12 bits of metadata m. This metadata contains some information about the following

1. A bit that indicates whether the page is a read, write, or executable piece of code (3 bits).
2. A bit that indicates whether the page is valid or not.



Process i:	Valid	READ	WRITE	EXEC	PPN
VP 0:	Yes	Yes	No	No	PP 6
VP 1:	Yes	Yes	No	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

Process j:	Valid	READ	WRITE	EXEC	PPN
VP 0:	Yes	Yes	Yes	No	PP 9
VP 1:	Yes	Yes	No	No	PP 6
VP 2:	Yes	Yes	Yes	No	PP 11

Figure 25: The page table entry contains the physical page number and some metadata.

Therefore, if you malloc, you are really just allocating some virtual memory addresses, which then get mapped to physical memory addresses in one or more pages.

Definition 4.4 (Page Fault)

It is clear that not every virtual page number can be mapped to a physical page number. If it turns out that a **page fault** happens if

1. the virtual page number maps to no physical page (i.e. is not in the page table) in the RAM
2. if some user program tries to access a physical page owned by the kernel
3. if the page number maps to some place in the disk (but it is not in physical RAM)

Page faults can be used in a lot of creative ways, but to reduce the risk of a page fault, e.g. when running out of physical memory, we can move some physical pages into disk and allocate memory by creating a new entry in our page table that maps this application's virtual page into the now empty physical page.

Note that by this construction, instructions that are contiguous in virtual memory may not be contiguous in physical memory. This may seem like it defeats the purpose of locality, but for most purposes, the 4KB page size will be enough to exploit it. We also see that malloced addresses in the heap (while we have learned that they were higher on the stack on higher addresses), are not necessarily in higher addresses in physical memory. Therefore, physical memory is scattered, and this is good since you don't need a giant contiguous block of memory to run large programs; you can divide it up into multiple physical pages.

Definition 4.5 (Swap Space)

Sometimes, the memory might not be in physical memory. Since memory is constrained (e.g. only 16GB), if we initialize a large array in the stack or global data, we may run out of memory. Therefore, the OS can flush out some physical pages in memory to disk, which is called **swapping**. The portion of the disk space that can be used in swapping is called the **swap space**.

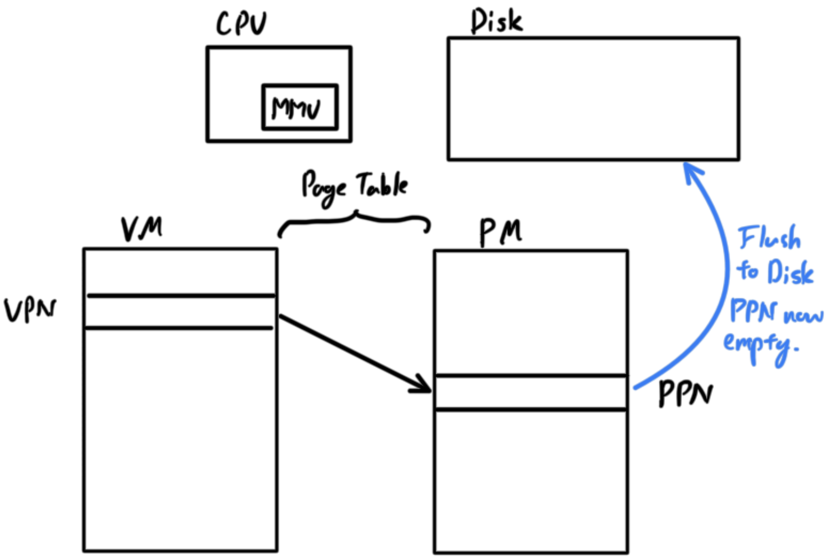


Figure 26: Swapping out physical pages to disk.

This allows us to abstract software into having almost infinite memory. Another important property is that swapping is **write-back** rather than write-through. We really don't want to write to disk every time we modify memory, so some thing may never end up on the disk (e.g. stack for short-lived processes). This is why when we open a file in C or Python, you may have to call `close()` since that will flush the memory to disk.

Example 4.3 (Page Fault)

When we swap out a physical page to disk, the physical page is now empty and accessing the virtual memory at this page table will cause a page fault. Say, when we want to write to a memory address that is swapped into the disk. The following will happen.

1. You execute code normally in user mode.
2. Then you try to write to a memory address that is swapped out, say through a `mov` operation. Say it is the following assembly code.

   ```
   80483b7: c7 05 10 9d 04 08 0d          movl $0x0,0x8049d10
   ```

   This raises a page fault, an exception, and so the OS goes into kernel mode.
3. The kernel then finds the location of this physical page in the disk. The implementation is OS-specific (e.g. you can store some metadata).

4. Then it must copy the page back from disk into memory, and it may also have to swap out some other physical page to disk to make space if needed.
5. Then the OS goes back into user mode, which now has access to the relevant memory in disk.

Ultimately, the moving operation is called twice. The first time it fails in user mode, and the second time (after the kernel mode, but now back to user mode) it succeeds. Note that this is different from a system call, which returns back to the *next* instruction. This call returns to the current instruction.

Definition 4.6 (Page Sharing)

This also makes protection and sharing to be quite nice. Given two virtual pages VP_1 and VP_2, owned by two different processes, we can have them share information by mapping to the same physical page PP.

Section	**Read**	**Write**	**Execute**
Stack	1	1	0
Heap	1	1	0
Static Data	1	1	0
Literals/const	1	0	0
Instructions	1	0	1

Table 3: Permissions for different sections of virtual memory.

Example 4.4 (Page Sharing Between Two Applications)

Furthermore, we can set process 1 to have only read permissions and process 2 to have read/write permissions. Therefore, say Google Chrome (process 2) can write your password into some memory, and then Slack (process 1) can read it, copy it into the CPU, and do stuff with it.

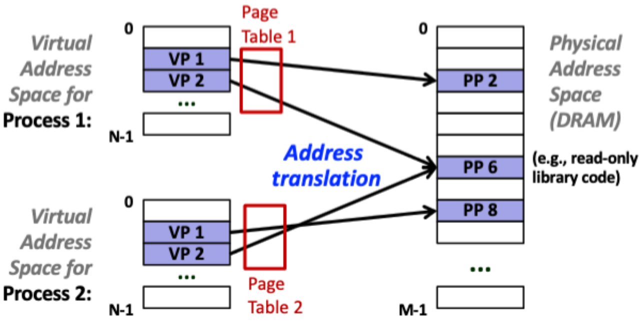

Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Page Table 1
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Address translation
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1
Page Table 2

Figure 27: Sharing of data between two processes.

Now that we see how memory is swapped in the backend, we can see why larger memory can sometimes mean faster programs and why thrashing occurs.

Definition 4.7 (Thrashing)

The set of virtual pages that a program is "actively" accessing at any point in time is called its **working set**.

1. If the working set of one process is less than physical memory, then there is good performance for one process.
2. If the working set of all processes is greater than physical memory, then we have **thrashing**, which

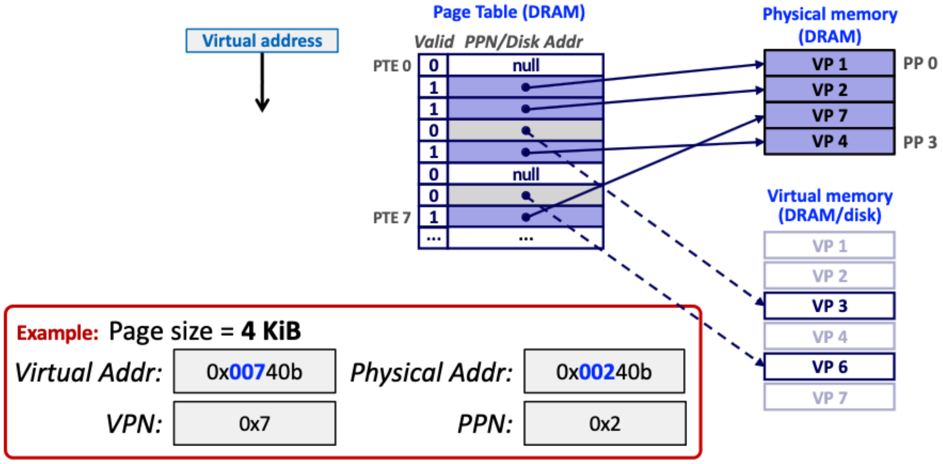
is a performance meltdown where pages are swapped between memory and disk continuously, and the CPU is always waiting or paging.

Example 4.5 (Computation Exercise)

Suppose that you have 16 KiB pages, 48-bit virtual addresses, and 16 GiB physical memory. How many bits wide are the following fields?

1. Virtual page number : $48 - 14 = 34$ bits.
2. Virtual page offset : 16 KiB is 2^{14} bytes, so we need 14 bits.
3. Physical page number : 16 GiB is 2^{34} bytes, so we need $34 - 14 = 20$ bits.
4. Physical page offset : 16 KiB is 2^{14} bytes, so we need 14 bits.

Furthermore, we have



Example: Page size = 4 KiB

Virtual Addr: 0x00740b Physical Addr: 0x00240b

VPN: 0x7 PPN: 0x2

Figure 28: Given the virtual address, we can figure out the physical address, VPN, and PPN easily.

5 Filesystems

Before we get into anything, even the loading of the firmware or the operating system kernel, we must talk about the hardware and how a computer stores data. Data, whether it is in memory or some disk, is just a bunch of sequences of bits. A **drive** is a physical device that can store data. A **partition** is a logical division of a drive, and a **filesystem** is a way to organize data on a drive. For example, if I have a 1TB SSD, I can run it as a single partition, or I can divide it into two partitions, one for a Windows operating system and another for a Linux operating system. A filesystem is a bit more confusing, so here are some examples.

Example 5.1 (Linux Filesystems)

Listed.

1. **ext4**: The most common filesystem for Linux.
2. **XFS**: Designed for high performance and scalability, often used in enterprise environments for large-scale storage.
3. **btrfs**: A modern filesystem that offers advanced features like snapshots, dynamic inode allocation, and integrated device management for better data reliability and performance.
4. **zfs**: Originally developed by Sun Microsystems for Solaris, ZFS is known for its data integrity, support for enormous storage capacities, and features like snapshots, copy-on-write, and built-in data compression.

Example 5.2 (Windows Filesystems)

Listed.

1. **NTFS (New Technology File System)**: The standard filesystem for Windows operating systems, supporting file permissions, encryption, and large file sizes.
2. **FAT32 (File Allocation Table 32)**: An older filesystem with wide compatibility across different operating systems, including Windows, macOS, and various Linux distributions, though it has limitations on file and partition sizes.
3. **exFAT (Extended File Allocation Table)**: Designed to be a lightweight filesystem similar to FAT32 but without its limitations, exFAT is used for flash drives and external hard drives due to its support for larger files and compatibility.

Example 5.3 (MacOS Filesystems)

Listed.

1. **APFS (Apple File System)**: The default filesystem for macOS, iOS, and other Apple operating systems since 2017, designed for SSDs and featuring strong encryption, space sharing, and fast directory sizing.
2. **HFS+ (Hierarchical File System Plus)**: Also known as Mac OS Extended, it was the primary filesystem for Mac computers before APFS, supporting journaling for data integrity.

When your computer boots up, it needs to know where to find the operating system kernel. This is done by mounting the filesystems. The **mount point** is the directory where the filesystem is attached to the system. The **root filesystem** is the filesystem that contains the operating system kernel.

Depending on your hardware specs, you may have multiple drives. To list all drives and their partitions, run **lsblk**. The type determines whether it is a disk or a partitions, and the mountpoints determine where the partitions are mounted. Furthermore, the `RO` indicates whether this is a HDD (1) or SSD (0).

```
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
zram0       254:0    0     4G  0 disk [SWAP]
nvme0n1     259:0    0 953.9G  0 disk
  nvme0n1p1 259:1    0   240M  0 part
  nvme0n1p2 259:2    0   128M  0 part
  nvme0n1p3 259:3    0 309.4G  0 part
  nvme0n1p4 259:4    0   990M  0 part
  nvme0n1p5 259:5    0  16.7G  0 part
  nvme0n1p6 259:6    0   1.4G  0 part
  nvme0n1p7 259:7    0   500M  0 part /boot
  nvme0n1p8 259:8    0   4.7G  0 part [SWAP]
  nvme0n1p9 259:9    0 619.9G  0 part /
```

Figure 29: This is the following output on my personal computer.

The **swap** partition is a special type of partition that is used as a temporary storage area for the operating system. It is used when the system runs out of RAM.

For a more detailed view on what the partitions consist of, you can run **fdisk -l**.

```
Disk /dev/nvme0n1: 953.87 GiB, 1024209543168 bytes, 2000409264 sectors
Disk model: PM9A1 NVMe Samsung 1024GB
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

```
Disklabel type: gpt
Disk identifier: 26D88CE9-B388-4CF1-856C-14D5EEB0C143

Device              Start        End    Sectors   Size Type
/dev/nvme0n1p1       2048     493567     491520   240M EFI System
/dev/nvme0n1p2     493568     755711     262144   128M Microsoft reserved
/dev/nvme0n1p3     755712  649658367  648902656 309.4G Microsoft basic data
/dev/nvme0n1p4 1960380416 1962407935    2027520   990M Windows recovery environment
/dev/nvme0n1p5 1962407936 1997441023   35033088  16.7G Windows recovery environment
/dev/nvme0n1p6 1997443072 2000377855    2934784   1.4G Windows recovery environment
/dev/nvme0n1p7  649658368  650682367    1024000   500M EFI System
/dev/nvme0n1p8  650682368  660447231    9764864   4.7G Linux swap
/dev/nvme0n1p9  660447232 1960380415 1299933184 619.9G Linux filesystem
```

As you can see here, my single disk has 9 partitions.

1. The first EFI system (1) or the Microsoft reserved (2) partition contains the Windows operating system kernel.
2. The Microsoft basic data (3) partition contains the Windows files.
3. The Windows recovery environment (4, 5, 6) is a partition that contains the Windows recovery environment, which are partitions set aside by the manufacturer to hold an image of your system before it was shipped from the factory.
4. The EFI system (7) partition contains the Linux operating system kernel, which is required to load the operating system.
5. The Linux swap (8) partition is a partition that contains the Linux swap.
6. The Linux filesystem (9) is a partition that contains the actual Linux operating system itself, along with all your files.

5.1 Mounting

You can further go into the `/dev` directory to see the devices that are mounted, e.g. the `/dev/nvme0n1p9` is the device that is mounted on the root directory, and most of these files are either device files (which are special files that provide an interface to hardware devices, allowing software and users to interact with them as if they were normal files) or symlinks.

The **mount** command is used to attach a filesystem to the system's directory tree. The **umount** command is used to detach a filesystem from the system's directory tree.

1. **Mounting a filesystem**: The general syntax is `mount -t type device dir`. For example, to mount the `/dev/nvme0n1p9` to the root directory, you can run `mount -t ext4 /dev/nvme0n1p9 /mnt`.
2. **Unmounting a filesystem**: The general syntax is `umount dir`. For example, to unmount the root directory, you can run `umount /mnt`.

When the computer boots up, it must automatically mount the specific filesystems. This is configured in the **fstab** file.

Definition 5.1 (fstab)

The **fstab** file is a system configuration file that contains information about filesystems. It is located at `/etc/fstab`. It is used to define how disk partitions, various other block devices, or remote filesystems should be mounted into the filesystem. Each line in the file contains six fields, separated by whitespace. The fields include:

1. **Filesystem**: The block device or remote filesystem to be mounted. This can be the UUID

(Universally Unique Identifier), the label, or the traditional device name (like /dev/sda1) that specifies which device or partition is being referred to.

2. **Mount Point**: The directory where the filesystem should be mounted.
3. **Type**: The type of the filesystem, e.g. ext4, vfat, swap, etc.
4. **Options**: Mount options for the filesystem, e.g. `rw` for read-write, `ro` for read-only, `noexec` to prevent execution of binaries, etc.
5. **Dump**: A number used by the **dump** command to determine whether the filesystem should be backed up. It is often set to 0 to disable backups.
6. **Pass**: A number used by the **fsck** command to determine the order in which filesystems should be checked. Root filesystems should have this set to 1, and other filesystems should either be 2 (to check after the root) or 0 (to disable checking).

```
# Static information about the filesystems.
# See fstab(5) for details.

# <file system> <dir> <type> <options> <dump> <pass>
# /dev/nvme0n1p9
UUID=abcfef03-bfae-4d1f-b463-fd6538f18a41 / ext4 rw,relatime 0 1
# /dev/nvme0n1p7
UUID=150D-7A67 /boot vfat rw,relatime,fmask=0077,dmask=0077,codepage=437,
    iocharset=ascii,shortname=mixed,utf8,errors=remount-ro  0 2
# /dev/nvme0n1p8
UUID=5c191f65-b016-475d-b04a-5b7c89bda31d none swap defaults 0 0
```

Figure 30: My personal fstab file.

5.1.1 Mounting a Remote Disk

It is actually possible to mount a folder on a server into your local machine. To do this, you use **sshfs** to mount a remote directory over SSH. The general syntax is `sshfs user@host:/remote/dir /local/dir` to mount and `fusermount -u /local/dir` to unmount.

5.2 Maintence

5.2.1 SSD

As soon as your write or delete bits from the SSD (e.g. when you're deleting a file), it degrades the speed of the read/write. To alleviate the effects, you can use TRIM, which is a command that allows the operating system to inform the SSD which blocks of data are no longer considered in use and can be wiped internally. It can be downloaded as a part of the `util-linux` package, which provides the systemd services **fstrim.timer** and **fstrim.service**. It is recommended to use weekly trims rather than continuous trims.

5.2.2 Filesystem

Occasionally, you may have a corrupt partitions, whether it is your boot or root directory. In this case, you should use the **fsck** command to check and repair a filesystem. The general steps are:

1. unmount the specific partition you want (identified with **lsblk**) using `sudo umount /dev/partition`.

2. run `sudo fsck -t type device` (or for specific filesystem types like vfat you can be a bit more specific by running `sudo fsck.vfat /dev/partition`) to check the filesystem and fix any changes.

3. mount the specific partition back using `sudo mount /dev/partition`.

5.3 Modifying Partitions

Modifying partitions require specialized software. Partitioning can be done using two main partitioning schemes **GPT** (the modern one) and **MBR** (legacy). The **parted** utility gives detailed info on your partitions. To see which scheme you have, just run `sudo parted -l`, where the output can be shown in Figure 31.

```
Model: PM9A1 NVMe Samsung 1024GB (nvme)
Disk /dev/nvme0n1: 1024GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system     Name                          Flags
 1      1049kB  253MB   252MB   fat32           EFI system partition          boot, esp
 2      253MB   387MB   134MB                   Microsoft reserved partition  msftres
 3      387MB   333GB   332GB   ntfs            Basic data partition          msftdata
 7      333GB   333GB   524MB   fat32                                         boot, esp
 8      333GB   338GB   5000MB  linux-swap(v1)                                swap
 9      338GB   1004GB  666GB   ext4
 4      1004GB  1005GB  1038MB  ntfs                                          hidden, diag
 5      1005GB  1023GB  17.9GB  ntfs                                          hidden, diag
 6      1023GB  1024GB  1503MB  ntfs                                          hidden, diag


Model: Unknown (unknown)
Disk /dev/zram0: 4295MB
Sector size (logical/physical): 4096B/4096B
Partition Table: loop
Disk Flags:

Number  Start  End     Size    File system     Flags
 1      0.00B  4295MB  4295MB  linux-swap(v1)
```

Figure 31: Output of `sudo parted -l` on my own machine.

It is important to know which partition scheme you should use.

1. To dual-boot with Windows (both 32-bit and 64-bit) using Legacy BIOS, the MBR scheme is required.
2. To dual-boot Windows 64-bit using UEFI mode instead of BIOS, the GPT scheme is required.
3. If you are installing on older hardware, especially on old laptops, consider choosing MBR because its BIOS might not support GPT.
4. If you are partitioning a disk that is larger than 2TB, you need to use GPT.
5. It is recommended to always use GPT for UEFI boot, as some UEFI implementations do not support booting to the MBR while in UEFI mode.

6 I/O Systems

7 Storage Management

8 Virtualization

9 Firmware

Let us go through the steps of a booting (bootstrapping) process. Administrators have little direct, interactive control over most of the steps required to boot a system, but they can modify bootstrap configurations by editing config files or system startup scripts.

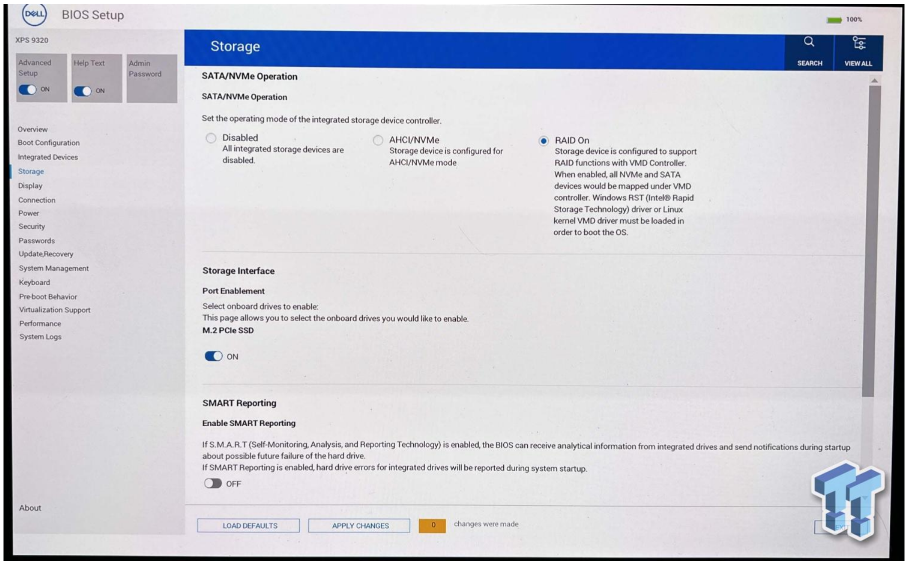
1. **Power On**: You power on the machine.
2. **Load firmware from NVRAM**: You want to be able to identify the specific piece of hardware to load your operating system in. The firmware is a permanent piece of software that does this.
3. **Probe for hardware**: We look for hardware that is on the computer.
4. **Select boot device (disk, network, etc.)**: We select the storage device that we want to load the operating system on.
5. **Identify EFI system partition**:
6. **Load boot loader (e.g. GRUB)**: A software that allows you to identify and load the proper OS kernel is provided.
7. **Determine which kernel to boot**: You choose which kernel you want to load.
8. **Load kernel**: The OS kernel is identified and loaded into the boot device.
9. **Instantiate kernel data structure**:
10. **Start init/systemd as PID 1**:
11. **Exectute startup scripts**:
12. **Running system**: You now have a running system!

Right above the hardware, the **system firmware**, is a piece of software that is executed whenever the computer boots up.

1. **Power Supply Activation**: Once the computer is turned on, the power supply begins to provide electricity to the system's components. One of the first signals generated is the "Power Good" signal, indicating that the power supply is stable and at the correct voltages.
2. **CPU Reset**: Upon receiving the "Power Good" signal, the CPU resets and starts its operations. The CPU is designed to start executing instructions from a predefined memory address, which is hardwired into the CPU. This address, stored in ROM, contains the starting point of the firmware.**Read Only Memory** is simply another type of computer memory that stores permanent data and instructions for the device to start up.
3. **Predefined Memory Address**: For BIOS systems, the CPU begins executing code at the firmware entry point located in the system's ROM (Read-Only Memory). In UEFI systems, the process is similar, but the UEFI firmware provides more functionalities and a more flexible pre-boot environment.
4. **POST (Power on Self Test)**: The firmware conducts a series of diagnostic tests to ensure that essential hardware components like RAM, storage devices, and input/output systems are functioning correctly. This stage is critical for verifying system integrity before loading the operating system.

To be honest, there is not a lot that the user can control here with just software. The firmware is a permanent piece of software that is executed whenever the computer boots up, which makes it relatively safe from tampering. If your computer fails to boot up, the most fundamental reason may be a firmware problem. However, we're not screwed yet.

Most firmware offers a user interface which can be accessed by pressing the F2, F11, F12, or some combination of magic keys at the instant the system first powers on. Depending on what computer model you have, you may have some control of basic functionalities.



SATA/NVMe Operation

SATA/NVMe Operation

Set the operating mode of the integrated storage device controller.

○ Disabled
All integrated storage devices are disabled.

○ AHCI/NVMe
Storage device is configured for AHCI/NVMe mode

● RAID On
Storage device is configured to support RAID functions with VMD Controller. When enabled, all NVMe and SATA devices would be mapped under VMD controller. Windows RST (Intel® Rapid Storage Technology) driver or Linux kernel VMD driver must be loaded in order to boot the OS.

Storage Interface

Port Enablement

Select onboard drives to enable:
This page allows you to select the onboard drives you would like to enable.
M.2 PCIe SSD

ON

SMART Reporting

Enable SMART Reporting

If S.M.A.R.T (Self-Monitoring, Analysis, and Reporting Technology) is enabled, the BIOS can receive analytical information from integrated drives and send notifications during startup about possible future failure of the hard drive.
If SMART Reporting is enabled, hard drive errors for integrated drives will be reported during system startup.

OFF

Figure 32: Firmware of Dell XPS 13 9320

Some important functionalities you can do with the firmware are:

1. Determine the boot order of the devices, usually by prioritizing a list of available options (e.g. try to boot from a DVD drive, then a USB, then the hard disk).
2.

The **BIOS**, which stands for **Basic Input/Output System**, has been used traditionally. It is mainly responsible for loading the bootloader. When the computer starts, it runs a **Power on Self Test (POST)** to make sure that core hardware such as the memory and hard disk is working properly. Afterward, the BIOS will check the primary hard drives' **Master Boot Record (MBR)**, which is a section on your hard drive where the bootloader is located.

A more formalized and modern standard called **EFI** (**Extensible Firmware Interface**) has replaced it, and it has been revised to the **UEFI** (**Unified Extensible Firmware Interface**) standard, but we can treat EFI and UEFI as equivalent in most cases. Fortunately, most UEFI systems can fall back to a legacy BIOS impelmentation if the operating system they're booting doesn't support UEFI. Since we're likely to encounter boot firmware systems, it's worthwhile to go into both of them.

9.1 Updating Firmware

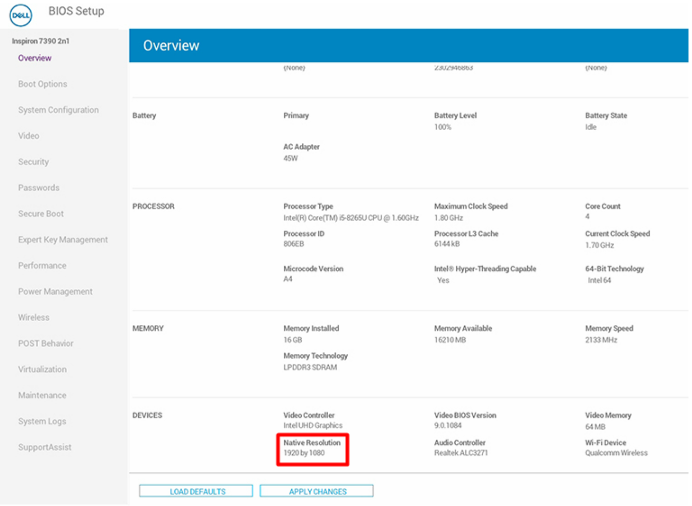
The first thing you should do when you're having trouble with firmware is use **fwupd**, which is a daemon that handles firmware updates. It is a simple daemon to allow session software to update device firmware on your local machine. Upon installation, it creates a systemd agent on `/lib/systemd/system/fwupd.service`. It does not start automatically. I have used this to update my firmware, which saved a lot of booting errors, with instructions accessed in this link.

9.2 Modifying UEFI Variables

You can directly examine and modify UEFI variables on a running system with the `efibootmgr` command. You get a following summary of the configuration:

```
BootCurrent: 0005
Timeout: 0 seconds
BootOrder: 0005,0001,0002,0000,0003,0004
Boot0000* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1   ...
Boot0001* ubuntu  HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd6026c1,0x800,0x780...
Boot0002* Windows Boot Manager   HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd60...
Boot0003* Linux Firmware Updater HD(1,GPT,ede98b7e-75ad-452e-ab47-...
Boot0004* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1 2 PciRoot(0x0)/...
Boot0005* Linux Boot Manager  HD(7,GPT,2d28b70f-725b-4ca3-98d4-25f5c83fc00e...
```

It shows you which disk you are currently booted into, the boot order that is currently configured, and information about each of the disks. You can use a GUI to do this as well. You can press a certain key when booting (F2 on my Dell XPS15 9500) to enter the **BIOS setup**.



(None) 2302946863 (None)

Battery
Primary
Battery Level: 100%
Battery State: Idle
AC Adapter: 45W

PROCESSOR
Processor Type: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
Maximum Clock Speed: 1.80 GHz
Core Count: 4
Processor ID: 806EB
Processor L3 Cache: 6144 kB
Current Clock Speed: 1.70 GHz
Microcode Version: A4
Intel® Hyper-Threading Capable: Yes
64-Bit Technology: Intel 64

MEMORY
Memory Installed: 16 GB
Memory Available: 16210 MB
Memory Speed: 2133 MHz
Memory Technology: LPDDR3 SDRAM

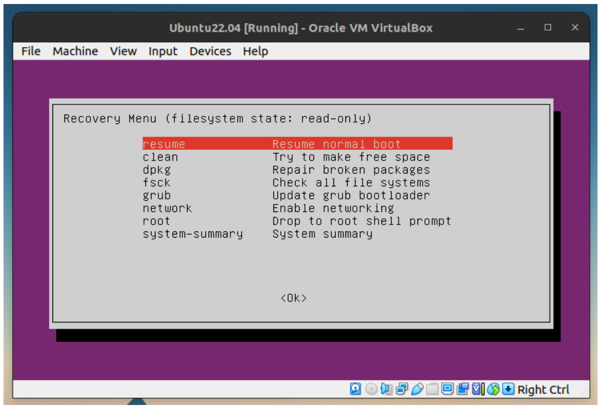
DEVICES
Video Controller: Intel UHD Graphics
Video BIOS Version: 9.0.1084
Video Memory: 64 MB
Native Resolution: 1920 by 1080
Audio Controller: Realtek ALC3271
Wi-Fi Device: Qualcomm Wireless

Figure 33: The BIOS setup can look very different depending on the computer but looks like this for me.

From here, we can edit different settings like boot options (priority of booting OS), certain video settings, etc.

9.3 Recovery Mode

Occasionally, you may run into problems with booting up the system. You can go into **recovery mode** by looking at the advanced options in the GRUB menu and selecting the option that literally says recovery mode.


Ubuntu22.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Recovery Menu (filesystem state: read-only)

        resume          Resume normal boot
        clean           Try to make free space
        dpkg            Repair broken packages
        fsck            Check all file systems
        grub            Update grub bootloader
        network         Enable networking
        root            Drop to root shell prompt
        system-summary  System summary



                          <Ok>
```

This gives us a list of options that we can take to fix the system. Every setting except root is automatically done. The root command gives us root privileges (no sudo is needed). This also means we have full access to all files, and we may cause irreversible damage to our system if we made a mistake. If we had not enabled read/write access with "Enable networking" the filesystem will be mounted read only, and we are unable to edit files. In case we don't have access to a network, or this was not desired, we can remount our filesystem(s) giving write access with the following command:

```
mount -o rw,remount /
```

With editing privileges, we can hopefully better diagnose or undo our problems. Finally, from the root shell type `exit` to go back to the menu.

10 Bootloaders

Once the firmware is loaded, which probes the system to find the hardware, it must load the operating system kernel. This is the job of the boot loader.

Definition 10.1 (Boot Loader, Boot Manager)

The **bootloader** is another critical piece of software that allows you to identify and load the proper operating system kernel. If it also provides an interactive menu with multiple boot choices, then it is often called a **boot manager**.

In modern systems which support UEFI (not the legacy BIOS), you must configure your partitions so that there exists an EFI partition (at `/boot`) that contains this bootloader.

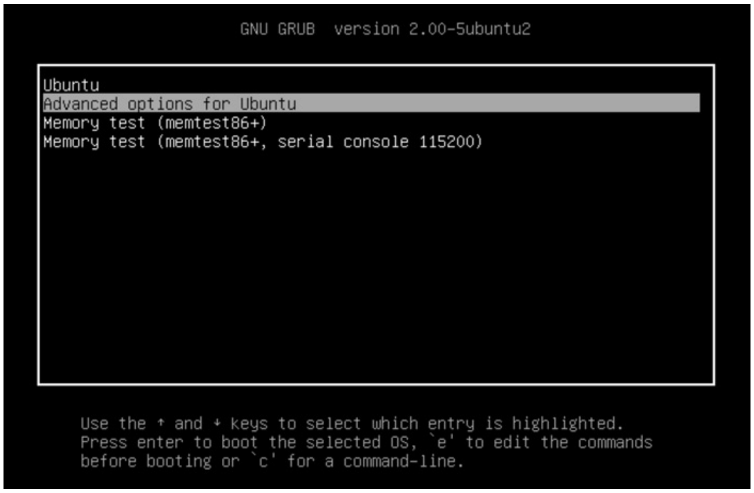
EFI bootloaders usually have a `.efi` extension, and it is crucial that you know where the bootloaders are in your system in case they go missing or are corrupt. To see the configuration, you can run **efibootmgr** (with verbose), which gives you information on several things:

1. It scans the entire system for EFI bootloaders and lists them.
2. It lists the locations of the EFI bootloaders. It starts off which what partition they are in, and then lists the directory where the bootloader is located. `BootX64.efi` is the Windows bootloader and `grubx64.efi` is the GRUB bootloader. For example, you may have a bootloader at `(partition 7)/boot/efi/EFI/Boot/bootx64.efi`.
3. It lists the boot order, which is the order in which the bootloaders are loaded. In case a boot loader fails to load, the next one is loaded. Therefore, if you have an arch linux bootloader that is corrupt, and the next in line is the Windows bootloader, you will automatically boot into Windows. You can also set the boot order in the BIOS.

In case you can't boot in, you can always get an Arch ISO burned in on a thumb drive, boot into it, mount the relevant partitions containing the Arch bootloader and the root directory, and then chroot into the root directory to modify files.

10.1 GRUB

The way that these kernels can be loaded can be configured through the bootloader, and the most popular boot manager is **GRUB**, the **Grand Unified Bootloader**. GRUB, developed by the GNU project, is the default loader on most Linux distributions. There is an old version called GRUB legacy and the more modern GRUB 2. Most people refer to GRUB 2 and simply GRUB. FreeBSD, which is another complete (non-Linux) OS, have their own boot loader, but GRUB is compatible with it. Therefore, for dual-boot or triple-boot systems that have multiple kernels, GRUB is the go-to bootloader for loading any of them.


GNU GRUB version 2.00-5ubuntu2
Ubuntu
Advanced options for Ubuntu
Memory test (memtest86+)
Memory test (memtest86+, serial console 115200)

```
Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands
before booting or `c' for a command-line.
```

Figure 34: GRUB menu on my screen. Ubuntu does not display the GRUB menu by default. To see GRUB during boot you need to press the right-hand SHIFT key during boot.

As a critical piece of software, we would expect its configuration files to be in the NVRAM, but GRUB understands most of the filesystems in common use and can find its way into the root filesystem on its own. Therefore, we can read its configuration from a regular text file, kept in `/boot/grub/grub.cfg`. Changing the boot configuration is as simple as updating the `grub.cfg` file, but it is not advised to edit it directly. Rather, we can edit the `/etc/default/grub` file and run `sudo update-grub` to that the changes are written to `grub.cfg` automatically.