Lecture 1, Jan 9, 2024

Overview of Operating Systems

- 3 core concepts:
  - *Virtualization*: sharing one resource by mimicking many independent copies
  - *Concurrency*: handle multiple things happening at the same time
  - *Persistence*: retain data consistency even without power
- A *process* is an instance of a running program – our first layer of abstraction
  - Each process needs its own virtual registers, stack, and heap
- How do we run two processes at the same time and still keep them independent?
  - Each process has its own *virtual memory* – its own independent view of the memory; this includes the stack and the heap
    - The process thinks it has access to all the memory, and the OS maintains that illusion
    - In reality one process cannot access the memory of another process for reasons of security
    - The same memory address in virtual memory for two different processes is mapped to different locations in physical memory
  - For local variables, the OS allocates an independent stack in memory for each process
  - For global variables, the compiler just picks some address for each variable on compilation, and the OS ensures there are no conflicts

Lecture 2, Jan 10, 2024

Operating System Concepts

- *IPC* (inter-process communication) is how processes transfer data between each other
- *File descriptors* are a resource that users may read bytes from or write to, identified by an index stored in a process
  - 0 is standard input, 1 is standard output, 2 is standard error
- *System calls* (syscalls) make requests to the OS
  - The `write` syscall takes a file descriptor, a pointer to a buffer, and a number of bytes to write
    - `ssize_t write(int fd, const void *buf, size_t count);`
  - The `exit_group` syscall takes a status code and exits the current process with that code
    - `void exit_group(int status);`
  - Syscalls are traceable via the program `strace`
- Note: API: application programming interface; abstracts the details and describes the arguments and return value of a function; ABI: application binary interface: the actual details of the function, how to pass arguments and what the return value is, e.g. passing arguments using the stack
- System calls are not like function calls; instead we generate an interrupt for the OS using an `svc` instruction (aarch64)
  - Arguments are not passed on the stack, but through registers x0 to x5; register x8 stores the syscall number (the type of system call)
  - This has the disadvantage that the number of arguments and size of arguments is limited
  - In x86_64 the arguments are partially passed using the stack
- *ELF* (Executable and Linkable Format) is the format used to specify executables and libraries
  - The first 4 bytes are always 0x7f followed by "ELF" in ASCII; these are the *magic bytes* that indicate the file format
  - There is a 64 byte file header and 56 byte program header; these indicate endianness, ISA, ABI, etc as well as what to load into memory

The Kernel

- *Kernel mode* (aka S-mode) is a privilege level on the CPU that allows access to more instructions, allowing more direct access to hardware

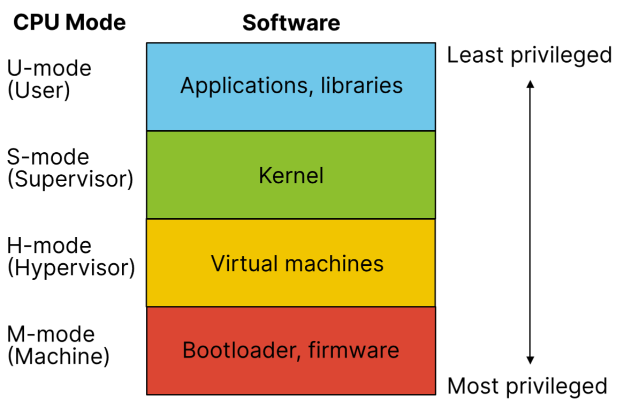


Figure 1: CPU privilege levels (RISC-V).

  - All user programs run in *user mode*
  - This is a security measure that only allows trusted software to access hardware, e.g. to manage virtual memory
- The *kernel* is simply software running in kernel mode
- Syscalls are the only way to transition between user and kernel mode; i.e. if a user program wants to access hardware, it has to do so via a syscall to the kernel
- The kernel can load *modules*, which allows loading code on-demand
  - The modules are executed in kernel mode so they allow access to hardware
- Kernel architecture is the way we decide whether to run services in user or kernel space
  - *Monolithic kernel* run all OS services in kernel mode, including file systems, drivers, etc
  - *Microkernels* run the minimum amount of services in kernel mode, including only services close to hardware such as virtual memory but not file systems or drivers
  - *Hybrid* kernels are between the two; e.g. on Windows emulation services run in user mode, on macOS device drivers run in user mode
  - *Nanokernels* and *picokernels* run even more services in user mode

Lecture 3, Jan 12, 2024

Libraries

- The definition of an operating system depends on the application; the OS allows you to run the application you want
  - An OS consists of a kernel and *libraries* required for your application
- During normal compilation, code is complied into object files, and then linked against libraries to produce an executable
- *Static libraries* (.a files) are included in the binary at link time
  - A bunch of object files are linked together to form an archive (a static library), and then when an executable is linked, this archive is copied into the executable
  - This means the static libraries have to be copied for each executable that needs to use them, so it is inefficient
  - Additionally, updates to the static library requires the executable to be recompiled
- *Dynamic libraries* (.so files) are included at runtime and not embedded in the executable binary
  - e.g. the C standard library (libc)
  - Multiple applications can use the same library; the OS only loads the library once and shares it with all applications that need it
    * This is more efficient and applications don't need to be recompiled if the library is updated
    * But if the dynamic library has a bug, all applications that use it will be affected

    - The `ldd` utility shows the dynamic libraries used by an executable
- Updating dynamic libraries may subtly change the ABI and cause incompatibilities and bugs
    - e.g. if the order of fields in a struct declaration changes, the ABI would change due to different memory layout, leading to subtle bugs
    - If a dynamic library exposes a struct, it should never be changed again!
- To avoid compatibility issues, use *semantic versioning*
- The `LD_LIBRARY_PATH` environment variable changes the path that dynamic libraries are searched for
- The `LD_PRELOAD` environment variable causes dynamic libraries in the specified path to be loaded first and overrides the usual dynamic libraries
    - This is how `valgrind` works; it overrides the standard `malloc()`/`free()` so that memory usage can be analyzed
- Another tool is clang's AddressSanitizer (ASan)
    - Add the `-Db_sanitize=address` flag to `meson setup build`
- Most system calls have corresponding function calls in C, but the wrappers may do additional tasks:
    - Set `errno` (error detection)
    - Buffer reads and writes to reduce the number of syscalls (since syscalls are slow)
    - Simplify interfaces (e.g. combining two syscalls into the same function)
    - Adding new features
    - e.g. the C `exit()` also runs functions registered with `atexit()` before an `exit_group` syscall; returning from `main()` is the same as calling `exit()` with the return value

Lecture 4, Jan 16, 2024

Process Creation

- A *Process Control Block* (PCB) contains all information about a process
    - This includes:
        * Process state
        * CPU registers
        * Scheduling information
        * Memory management information
        * I/O status
        * Anything else that the process needs
    - In Linux this is the `task_struct` struct

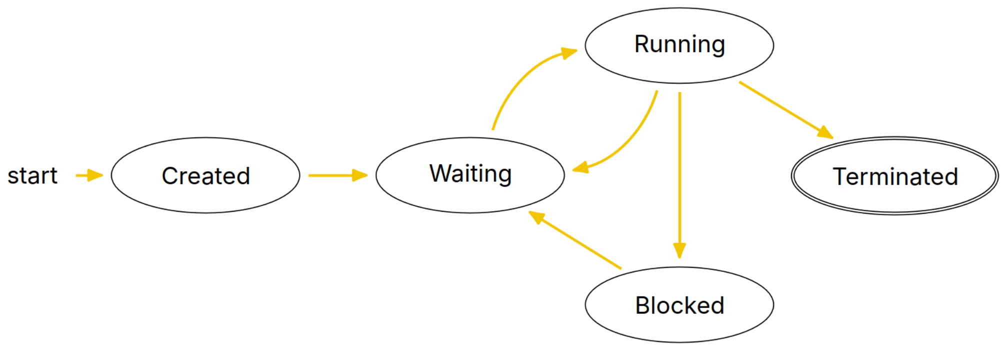


Figure 2: Process lifecycle diagram.

- Each process goes through a lifecycle as depicted above
    - The "waiting" state means a process is ready to run, but the CPU is not running it yet (due to scheduling)
    - The "blocked" state means a process is waiting for I/O and cannot be run
- In Linux, the `/proc` directory contains a special filesystem that present the kernel's state

    - Every directory that is a PID that represents a process
- Processes could be created from scratch; we can load the program into memory and create the PCB (which is what Windows does), but on Unix this works differently
- On Unix systems, instead of creating a new process, we can clone an existing process
    - This clones the entire PCB of the old process, so everything is copied, including variables
    - The two processes are distinguished using a parent-child relationship
    - We could then allow either process to load a new program and set up a new PCB
- To clone a process, use the `pid_t fork(void);` function (note `typedef int pid_t`)
    - The return value is the PID of the child process, or 0 if currently in the child process, or -1 on failure
    - Syscalls `pid_t getpid();` can be used to get the current process PID; `pid_t getppid();` gets the parent process PID
        * Note: `man <func>` can be used to view the manual pages (documentation) for syscalls and library functions
- `int execve(const char *pathname, char *const argv[], char *const envp[]);` replaces the current process with another program and resets
    - `pathname` is the full path of the program; `argv` are the program args; `envp` are the environment variables
        * Note: the first element of `argv` should still be set to the program name
    - Returns -1 on failure and sets `errno`
    - This allows a process to be replaced with another one; so to execute another program from a process, we can `fork()` and call `execve()` in the child process

Lecture 5, Jan 17, 2024

Process Management

- On Linux a process's state can be read through `/proc/<PID>/status`:
    - R: Running and runnable (running/waiting)
    - S: Interruptible sleep (blocked; can be resumed by the kernel if desired)
    - D: Uninterruptible sleep (blocked; cannot be resumed since it is waiting on I/O)
    - T: Stopped (can be continued explicitly by the user or another process)
    - Z: Zombie
- On Unix systems the kernel launches a single user process, `init`, which is the parent of all other processes
    - This is located at `/sbin/init`, and is usually `systemd`
    - This executes every other process on the machine and must always be active; if it exits the kernel will think you're shutting down
    - Some OSes will also create an idle process (e.g. Windows)

- Each process is assigned a process ID (PID) on creation, which does not change and is unique for every active process
    - On most Linux systems this goes up to 32768; 0 is reserved/invalid
    - The OS can recycle a PID after the process dies
    - Each process has its own *address space* (i.e. its own copy of virtual memory)

Zombies and Orphans

- The parent is responsible for the child and should acknowledge when the child terminates
- If the child exits first, it becomes a *zombie process* until the parent reads its exit status
    - The PCB cannot be removed by the OS until its exit status is read
    - Use the `pid_t wait(int *status);` syscall to check the child's status
        * Returns -1 on failure, 0 for nonblocking calls with no child changes, and the PID of the child on success

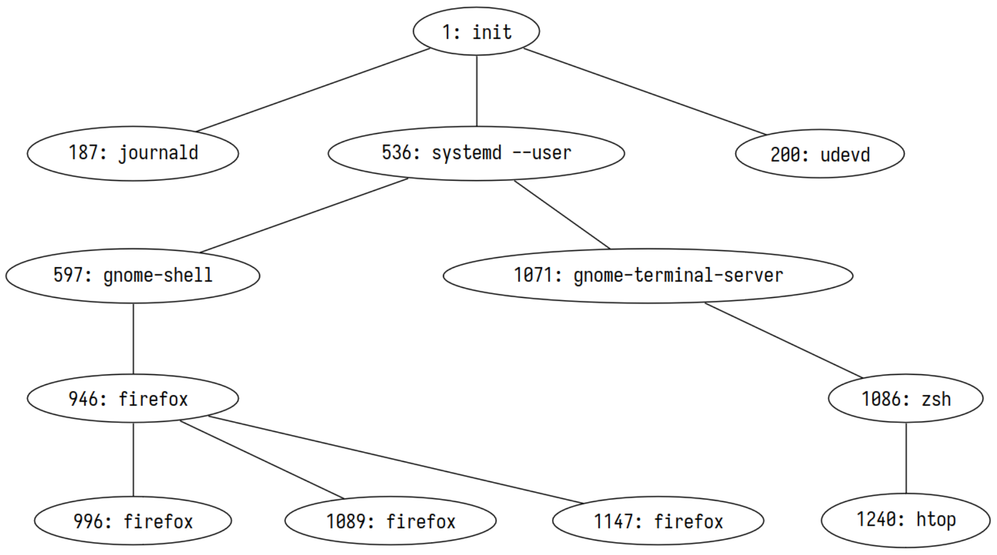


Figure 3: Example process tree.

        - If there are multiple children, it returns the PID of the first child to terminate
      * The child's status is written to the address `*status`, which is a bit mask
      * Use macros such as `WIFEXITED()`, `WEXITSTATUS()` etc to check specifics about the status
  - `wait()` is a *blocking* system call, i.e. it will not return until the child is terminated
  - The `waitpid()` syscall can be used to check on a child with a specific PID, and allows nonblocking calls
- When a child terminates the OS sends the parent a *signal* (`SIGCHLD`) to ask the parent to acknowledge the child
  - The parent is free to ignore this
  - If the parent always ignores it, the child will stay as a zombie until the parent dies, at which point it becomes an orphan and gets re-parented
- If the parent exits first, the child becomes an *orphan process*
  - Since some process still needs to acknowledge the child's exit, it needs a new parent
  - The OS will re-parent the child, usually to `init`
    * Note: A *subreaper* process (relatively new Linux feature) will take the place of `init` and adopt all orphans that are descendant from it

Lecture 6, Jan 19, 2024

Basic IPC

- IPC is any method to transfer bytes between two or more processes
- Reading and writing files is a form of IPC
- The `ssize_t read(int fd, void buf[.count], size_t count);` syscall reads from a file and returns number of bytes read
  - 0 is returned on end of file or a closed file descriptor
  - We should check for errors!
- The `ssize_t write(int fd, const void buf[.count], size_t count);` syscall writes to a file and returns number of bytes written
- Linux always uses the lowest available file descriptor for new opened files, so we could close file descriptor 0 and open a new file, and this will replace stdin
  - Similarly we can also replace stdout, stderr, etc

  - This can be done in a shell using a redirect, e.g. `./program < input_file > output_file`
  - Without changing the code the program can work with any type of input/output stream

Signals

- *Signals* are a form of IPC that interrupts the program
  - The kernel sends a number to the program indicating the type of signal
  - Note signals can interrupt syscalls like `read()` and `write()`, resulting in an `EINTR`
- Using Ctrl+C sends a `SIGINT` (interrupt) to the program, which is a signal
- We can write handlers to handle these signals
  - The default handler behaviour is to exit the program with an exit code of 128 + signal number
  - We can write handlers to ignore the signals so the program won't exit immediately
- Using the `sigaction()` syscall allows us to define our own signal handlers
  - Signal handlers return nothing and takes an int argument, which is the signal number
  - Some standard signal numbers:
    - 2: `SIGINT` (interrupt from keyboard)
    - 9: `SIGKILL` (terminate immediately)
    - 11: `SIGSEGV` (segmentation fault)
    - 15: `SIGTERM` (terminate)
      - Can be ignored
- Using the `int kill(pid_t pid, int sig);` syscall we can send signals manually
  - We can use the `kill` command to send signals (by default sends `SIGTERM`); use `pidof` to get a process's PID

Interrupts

- Most operations are non-blocking, i.e. returning immediately and we check later if something occurs
- `read()`, `write()`, `wait()` are blocking by default, but they have nonblocking variants/arguments
- To react to changes to a non-blocking call, we can use a *poll* or *interrupt*
- Polling continuously checks for changes
  - Very simple to setup and we don't have to worry about things getting interrupted
  - This is inefficient and if we don't poll fast enough our response can be delayed
- Using interrupts, we can register a signal handler to check for e.g. `SIGCHLD` when children exit, to get notified immediately

Lecture 7, Jan 23, 2024

Context Switching

- Older OSes are *uniprogramming* operating systems, which can only run one process at a time
- All modern operating systems are *multiprogramming* operating systems, which allow processes to run in parallel or concurrently
- The scheduler decides when to switch processes, which involves a process of:
  1. Pausing the current process
  2. Saving the current process state
  3. Getting the next process to run from the scheduler
  4. Load the next process' state and let it run
- We can have processes pause themselves voluntarily via a syscall (*cooperative multitasking*), or the OS will take control and choose when to pause processes (*true multitasking*)
  - Generally all general-purpose operating systems use true multitasking
  - For true multitasking the OS can give processes time slices and wakes up periodically using interrupts to do scheduling
- Swapping between processes is called *context switching*, which is overhead – this is wasted time that we want to minimize

Pipes

- `int pipe(int pipefd[2]);` forms a one-way communication channel using two file descriptors
  - The first file descriptor is the read end of the pipe; the second is the write end of the pipe
    - Note the `pipefd` is an output parameter
  - Any data written to `pipefd[1]` can be read from `pipefd[0]`
  - The pipe is managed by the kernel using an internal buffer
  - Returns 0 on success, -1 and sets `errno` on failure
- One usage of this is to create a pipe and then fork, which will also give the child access to the pipe, allowing IPC between the child and parent
- The pipe closes if no process has the read or write end of the pipe open
  - If both ends of the pipe are open by at least one process, the pipe won't close, even if both ends are opened by the same process
  - Best practice is to close the file descriptors when we no longer need them, otherwise the pipe won't close
  - e.g. if the parent needs to send data to the child, the parent should close the read end of the pipe and the child close the write end of the pipe

Lecture 8, Jan 24, 2024

Subprocesses

- `int execlp(const char *path, const char *arg, ...);` is a more convenient alternative to `execve()`
  - Instead of having to build an argument array we can use varargs to specify program arguments
  - Will also search `PATH` for the executable, so we don't have to specify the full path
- `int dup(int oldfd);` and `int dup2(int oldfd, int newfd);` duplicates a file descriptor and returns an independent file descriptor that refers to the same file
  - The `oldfd` and `newfd` file descriptors will refer to the same thing after the call
  - `dup()` will return the lowest file descriptor
  - `dup2()` will close the `newfd` file descriptor and then make that file descriptor refer to the same thing as `oldfd`
    - This can guarantee that we get the exact file descriptor number that we want
  - This is an atomic system call (can't be interrupted)
  - Note: Closing the original file descriptor does not close the new returned file descriptor! (i.e. we need to close the file descriptor returned by `dup()` separately)
- Our goal is to create a new process with a specified executable name, and be able to send to the process' `stdin` and receive any data it writes to `stdout`
  - We `fork()` and call `execlp()` in the child to start the process
  - To get the child's output, `pipe()` before forking, and then call `dup2()` to replace the child's `stdout` file descriptor with the write end of the pipe; now in the parent process we can read from the pipe to get the child's output
  - Similarly to send data to the child, replace the child's `stdin` with the read end of a pipe

Lecture 9, Jan 26, 2024

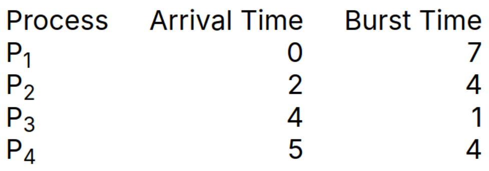
Scheduling

- A resource is *preemptible* if it can always be taken away and used for something else at any time
  - e.g. a CPU is preemptible since we can perform context switching to use it for another process at any time
- A *non-preemptible* resource cannot be taken away with acknowledgement
  - e.g. memory and disk space
  - In this case, it is shared through allocations and deallocations

  - Some systems may allow you to allocate CPUs
- Scheduling is done by two components: the *dispatcher* and the *scheduler*
  - The dispatcher is a low-level mechanism that does the actual work of context switching
  - The scheduler is a high-level policy that decides which processes to run and when
    * The scheduler runs whenever a process changes state
    * For non-preemptible processes, once started they have to run until completion
- Process scheduling involves balancing the following tradeoffs:
  - Minimize waiting time and response time
    * The waiting time of each process is the time that it exists minus the amount of time it is actually executing
    * The response time is the time it waited from arrival until its first time on the CPU
  - Maximize CPU utilization
  - Maximize throughput
    * By extension, we should minimize context switching whenever possible since it introduces overhead
  - Try to achieve fairness
- The *burst time* of a process is the amount of time a process runs

First Come First Served (FCFS) Scheduling

- The most basic form of scheduling, assumes no preemption
- The first process that arrives gets the CPU
- Processes are stored in a FIFO queue in arrival order



Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Figure 4: Example processes with arrival and burst times.

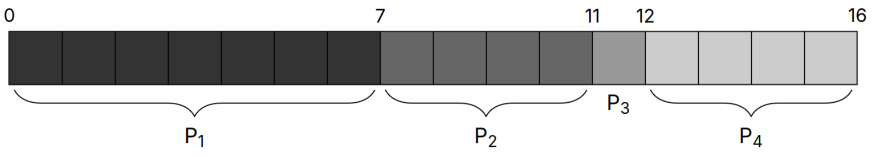


Figure 5: Scheduling for the example processes with FCFS.

Shortest Job First (SJF)

- A tweak to FCFS to schedule the job with the shortest burst time first (still assumes no preemption)
- Theoretically, compared to FCFS this could reduce the waiting time of processes since shorter jobs are run first
  - In fact, SJF is provably optimal at minimizing wait time with no preemption
- However, SJF is not practical since we don't actually know the burst times of each process
- Furthermore, SJF starves long-running processes (shorter processes will always get in front of longer processes, so the longer process can never execute)

Shortest Remaining Time First (SRTF)

- Adapts SJF to work for preemptions

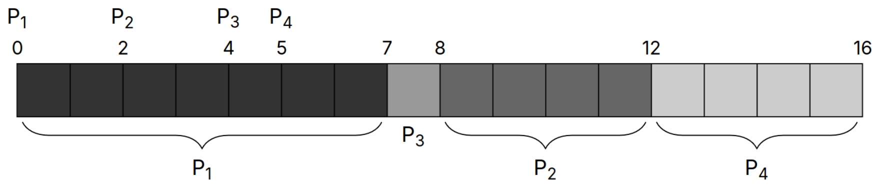


Figure 6: Scheduling for the example processes with SJF.

- Any time a new process arrives, the process with the least remaining runtime gets switched to and executed
- Further reduces waiting time compared to SJF but again impractical

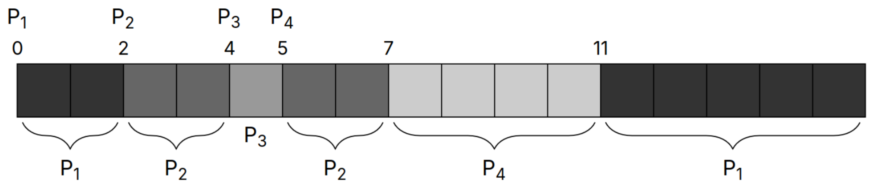


Figure 7: Scheduling for the example processes with SRTF.

Round-Robin (RR)

- Incorporates fairness unlike the previous algorithms, and actually used in practice
- Execution is divided into time slices (aka *quanta*) and uses a FIFO queue similar to FCFS; if a process is still running by the end of its time slice, we preempt it and add it to the back of the queue to ensure fairness
- On a tie (new process arrives when previous one is preempted), favour the new process first
- Generally RR performance depends a lot on the quantum length and job length
  - Typically it has the advantages of low response time and good interactivity, with fair allocation of the CPU and low average waiting time (when job lengths vary)
  - If the time slice is too big, this reduces to FCFS; if the time slices are too small, the processes are preempted all the time, so a lot of time is wasted doing context switching
    - Generally we want time spent context switching to be less than 1%
    - We can reduce the time slice size until context switching overhead reaches 1%
  - Average waiting time is poor when jobs have similar lengths

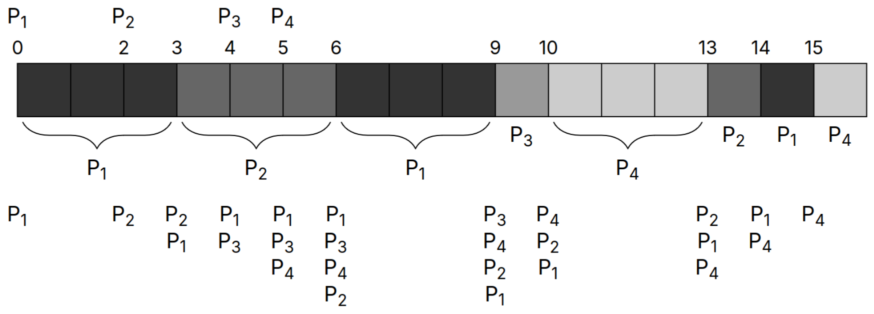


Figure 8: Scheduling for the example processes with RR and a time slice of 3 units. The queue is shown on the bottom, vertically.

Lecture 10, Jan 30, 2024

Advanced Scheduling

- Sometimes we want to favour some processes over others, so we can assign a priority to each process
  - Processes with higher priority will run first, and equal priority processes use round-robin
  - Can be preemptive or non-preemptive
  - e.g. on Linux the priority ranges from -20 (highest) to 19 (lowest)
- If there are lots of higher priority processes this can lead to starvation
  - We can have the OS dynamically change the priority
  - Increase the priority of processes that haven't been executed for a long time and then restore it after it runs
- In *priority inheritance* a process inherits the highest priority of the waiting processes, and is reverted back to the original priority after the dependency is resolved
- Processes can be *foreground* (receives input and interacting with the user) or *background*
  - Foreground processes need better response time since the user is interacting with it
  - Background processes would have a group ID that is different from its terminal group ID
  - This is harder to determine today as systems have gotten more complex
- To address this we can use different queues for foreground and background processes, e.g. RR for foreground and FCFS for background processes
  - To decide which queue runs, we can use a further layer of RR between the queues and have priorities for each queue
- In general scheduling involves a series of tradeoffs and heuristics instead of one right answer

Multiprocessor Scheduling

- Assume every core is a symmetric multiprocessing (SMP) system, i.e. all CPUs have the same physical memory but each have their own private cache
- We can use the same scheduling system (global scheduler) and just keep running processes as long as CPUs are available
  - This is not scalable since there is only a single scheduler and each CPU needs to wait for the same global scheduler
  - Also poor cache locality as processes are swapped between cores
  - Approach in Linux 2.4
- Each CPU can use its own scheduler; new processes are assigned to some CPU and after that each CPU manages its own scheduling
  - Can assign to the CPU with the lowest number of processes
  - This avoids the scalability issue (no blocking on resources) and cache locality issue
  - Can lead to load imbalance, as we don't know how long each task will run, so some CPUs may end up with fewer or less intensive processes
- We can use a compromise between the two approaches and use a global scheduler that can re-balance per-CPU queues
  - If a CPU is idle, we can take a process from another CPU; this is known as *work stealing*
  - Use *processor affinity* (preference of a process to stay on the same core) to decide which processes can switch CPUs
  - This is a simplified version of the $O(1)$ scheduler in Linux 2.6
- Sometimes we want to schedule multiple processes simultaneously as a group (*gang scheduling* or *coscheduling*)
  - The processes may have dependences; occurs mostly in high-performance computing
  - Each process should run on its own core all at the same time to maximize performance
  - This requires a global context-switch across all CPUs as each CPU can't be independent

Real-Time Scheduling

- In real-time systems, processes have time constraints for either deadlines or rates

    - e.g. audio output, autopilot
- *Hard real-time* systems guarantee that a task completes within a certain amount of time
    - Each instruction will be counted so we know exactly how long each process is running for
    - This is often the case on simple embedded systems
- *Soft real-time* systems just assign a higher priority to critical processes
    - The deadline is met in practice
    - Most general-purpose operating systems are soft real-time since we have little control over what the user does and modern systems are very complex
    - e.g. Linux

Scheduling on Linux

- Linux uses FCFS and RR scheduling
    - Processes with the same priority use a multilevel queue
    - For soft real-time processes, the highest priority process is always scheduled first
    - For normal processes it adjusts the priority based on aging and available CPU time
- Real-time processes are always prioritized in Linux
    - They will either be scheduled using FCFS (`SCHED_FIFO`) or RR (`SCHED_RR`)
    - There are 100 static priority levels (0 - 99)
- Normal processes use normal scheduling policies
    - Priority ranges from -20 to 19 with higher numbers being lower priority
    - By default the priority is 0
- Priories can be set with the `nice` and `sched_setscheduler` syscalls
    - The "nicer" a process is, the lower its priority and it'll use up less of the CPU
- Linux maps niceness and soft real-time priority to an internal priority where lower numbers are always higher priority as shown in the figure below

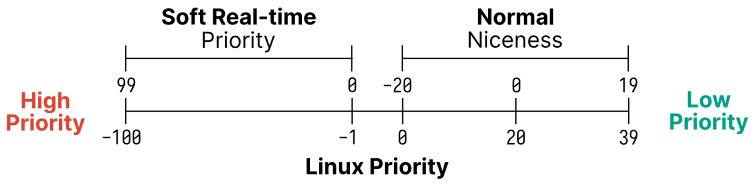


Figure 9: Linux process priorities.

Completely Fair Scheduler

- Modern versions of Linux (past 2.6.23) use the completely fair scheduler (CFS) instead of the $O(1)$ per-CPU scheduler
- The $O(1)$ scheduler had fairness issues for different priority processes
- If context switching had no cost, then we'd have an infinitely small timeslice and all processes would be running at the same time and get the same amount of CPU time

- In CFS, each runnable process has a "virtual runtime" in nanoseconds
- At each scheduling point where the process runs for time t, the virtual runtime of the process is increased by t multiplied by a weight, which is based on priority
    - Higher priority processes have lower weight, so their virtual runtime increase slowly and as a result they get scheduled more
    - Virtual runtime only increases
- The scheduler will always select the process with the lowest virtual runtime and computes its dynamic time slice based on the IFS
- CFS uses a red-black tree with virtual runtime as the key
- CFS favours I/O bound processes by default (processes that spend the most time waiting)

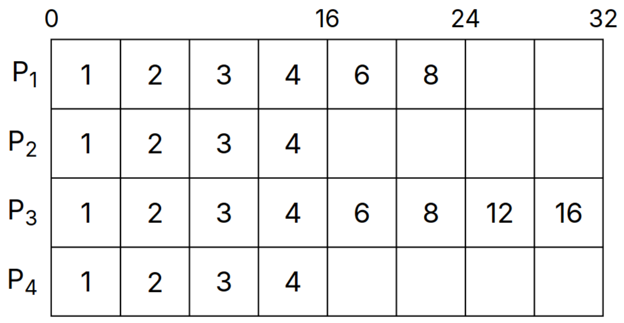


Figure 10: Ideal fair scheduling for 4 processes arriving at time 0, with burst times 8, 4, 16, 4.

Lecture 11, Jan 31, 2024

Virtual Memory

- We need virtual memory to satisfy the following goals:
  - Multiple processes must be able to co-exist – the same virtual address can map to different physical addresses
  - Processes should not be aware that they are sharing physical memory
  - Processes cannot access another process' memory (unless explicitly allowed)
  - Performance close to directly using physical memory
  - Limit fragmentation (wasted memory)
- The *memory management unit* (MMU) is the hardware responsible for memory mapping and permission checks
  - Memory is divided into fixed size *pages* (typically 4096 bytes)
  - Pages in virtual memory are pages, while pages in physical memory are called *frames*
  - A page is the smallest possible unit of memory that the kernel can allocate
- Virtual memory used to be implemented with *segmentation*, which is no longer used
  - Virtual address space is divided into segments for code, data, stack and heap which can all be resized
  - Segments are costly to relocate and leads to fragmentation
  - Each segment contains a base address, limit, and permissions
  - When accessing memory, the MMU checks that the offset is within the limit, and then checks for permissions before giving access
- Usually the more virtual memory we map, the more expensive it will be
  - For most systems we use a 39-bit virtual address space, which gives 512 GiB of addressable memory to each process
- Mapping is usually implemented using a page table (a lookup table), indexed by the virtual page number (VPN) and gives the physical page number (PPN)
  - The kernel sets up the page table and the MMU indexes it
  - The least significant 12 bits are the offset (4096 possible values to match our page size) and the other 27 are used to index the page table
  - The number of bits used for the PPN can be different than the VPN
  - Each entry in the page table has a structure shown in the figure below
- Example: given an 8-bit virtual address, 10-bit physical address, 64 byte pages:
  - How many virtual pages are there?
    * Each page is 64 bytes so offset is 6 bits
    * This leaves 2 bits for the VPN, which gives 4 virtual pages
  - How many physical pages are there?
    * $10 - 6 = 4$ bits for the PPN gives 16 physical pages
  - How many entries are in the page table?

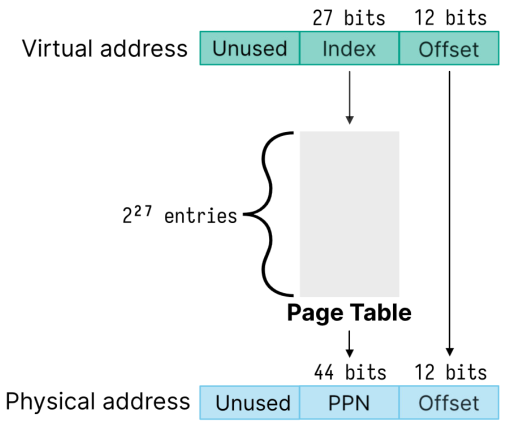


Figure 11: Illustration of the page table.

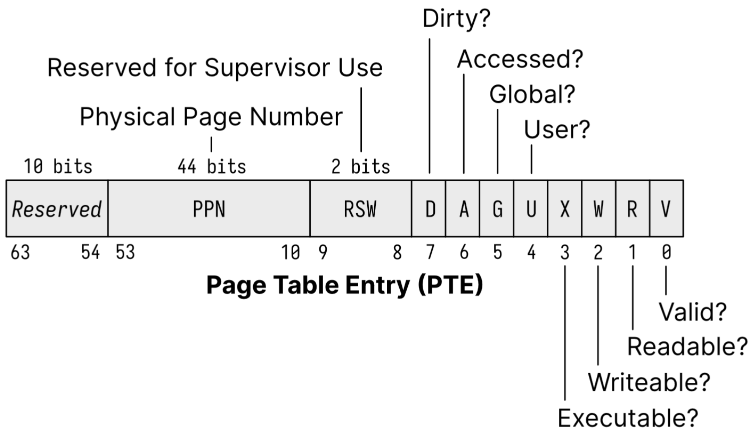


Figure 12: Structure of a page table entry (PTE).

    * 4 entries since there are 4 virtual pages
  - Given the page table [0x2, 0x5, 0x1, 0x8] what is the physical address of 0xF1?
    * 0xF1 = 0b1111'0001
    * Offset is 0b11'0001
    * VPN is 0b11 (page 3) so PPN is 0x8 = 0b1000
    * Final address is 0b10'0011'0001 = 0x231
- Each process has its own page table, which is managed in software
  - When a process is fork()ed, the page table is copied from the parent
  - The kernel implements copy-on-write for fork()ed programs – the memory is shared until a process tries to write to it, at which point it is copied
    * The write permission bit is turned off initially before memory is copied
  - We can use the vfork() syscall to fork but do not copy the page tables
    * If any memory is modified by the child, the behaviour is technically undefined since the memory is shared with the parent
    * Use only for performance sensitive programs or when we exec() immediately after fork()
- Next time on ECE353: The page table has 2^{27} bits which would take an entire gigabyte, so how do we give each process its own copy?

Lecture 12, Feb 2, 2024

Page Tables

- If each process' page table contained entries for the entire memory space, they would take up too much space
- Most processes don't use all the virtual memory space, so we can make the page tables smaller
- We will make the page table fit on a single page

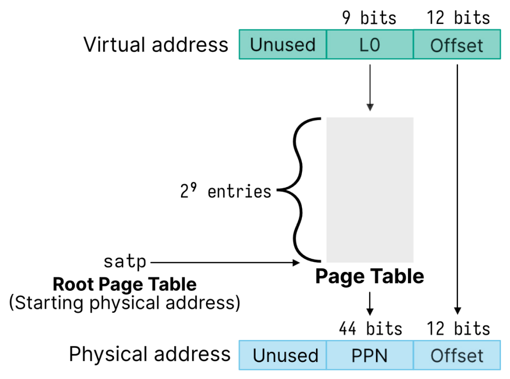


Figure 13: Page table existing in a single page.

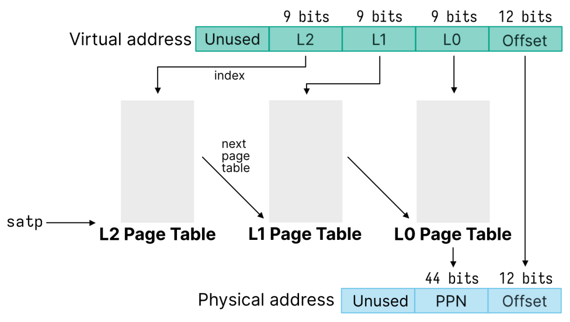


Figure 14: Indexing a multi-level page table.

- Multi-level page tables are how we save space for programs that don't use a lot of addresses
  - Virtual addresses are split into 9-bit sections, each corresponding to a different level (`L2`, `L1`, `L0`)
    - We still have the same 27-bit VPN but now it's split into 3 sections of 9 bits each
    - Each entry is 8 bytes (64 bits), which means we need 512 entries or 9 bits for the address in each level
  - Instead of giving the PPN of the physical memory, the higher level page tables instead give the PPN of the next lower level's page table, to look up the next address and so on, until `L0` stores the PPN of the actual memory that the virtual address maps to
  - e.g. first look up the PPN of the `L1` page table in the `L2` page table, then look up the PPN of the `L0` page table in the `L1` page table, and then look up the actual page number of the memory in the `L0` page table
  - The `satp` stores the *root page table* (i.e. the PPN of the `L2` page table)
- Each process will start with only 3 pages of page tables (1 for each `L2` to `L0`) which is enough for all the VPNs
  - If a process ends up using more memory, then the OS will allocate more pages to the page table
  - For smaller programs this saves a lot of space
  - If we use all of the virtual memory this would end up using more space, but not that much
- The kernel maintains a linked list of all the free physical pages (a free list)
  - When allocating memory the kernel simply removes it from the list and allocates it to the program
- Example: consider the virtual address `0x3FFFF008` with 30-bit virtual addresses and 4096 byte page sizes
  - In this case we have an 18-bit VPN so we only need `L1` and `L0` page tables
  - Split this into `0b111111111'111111111'000000001000`
  - First we would use the `satp` to look up the address of the `L1` table
  - The `L1` page table index would be 511, which stores, e.g. a PPN of `0x8`
  - Then we look up the `L0` page table, at address `0x8000`, and index it with 511, which stores, e.g. `0xCAFE`
  - The final physical address would be `0xCAFE008`
- Example: if we have a program that uses 512 pages, what is the minimum number of page tables we need and what is the maximum?
  - Minimum: 512 pages fits into a single `L0` page table, so we only need 3 tables (`L2`, `L1`, `L0`)
  - Maximum: each page has its own `L0` table, which has its own `L1` table, but with a shared `L2` (since `L2` is always unique); in this case we need 1025 tables
- Example: assume a 32-bit virtual address with a page size of 4096 bytes and a PTE size of 4 bytes
  - To have each page table fit into a single page, we need $4096/4 = 1024$ PTEs on a single page
  - To index each page table we need 10 bits
  - Each virtual address has 12 bits dedicated to the offset (for the page table size of 4096), so 20 bits are left for the VPN
  - Therefore we need 2 levels since each level gives 10 bits
  - The number of levels can be found by taking the virtual address bits, subtracting the offset bits and dividing by the number of bits to index each table (ceil if not an integer)
- Next time on ECE353: Now we have to do 3 memory accesses to look up an address, so this is 4 times slower than accessing physical memory directly; how do we make this faster?

Lecture 13, Feb 6, 2024

Page Table Implementations

- How do we make page tables faster?
  - We'll likely access the same page multiple times in succession
  - A process may only need a few page mappings at a time
  - To speed up memory accesses, we use *caching*
- The *translation look-aside buffer* (TLB) caches page table entries
  - When a process accesses memory, it would be first looked up in the TLB, and only translated in

  the case of a TLB miss
  - On a hit, the time taken to access memory is the TLB search time plus memory access time; on miss the time is the TLB search time, plus memory access time and PPN lookup time
  - We can calculate the *effective access time* (EAT, the expected value of access time) by taking a weighted average using α, the TLB hit rate

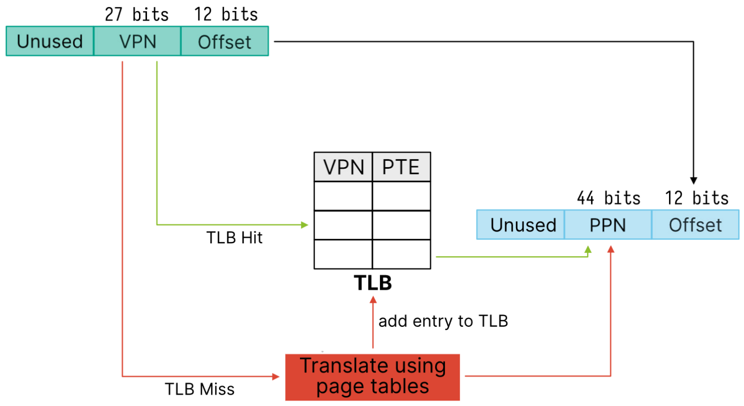


Figure 15: Translation look-aside buffer.

- Note since each process has its own virtual memory mapping, we need to handle the TLB when context switching
  - Most implementations flush the TLB on context switch
  - Some implementations attach process IDs to the TLB
- Because of the TLB, programs run faster if they use memory continuously and access the same pages most of the time
- The `sbrk` syscall is used for userspace page allocation
  - This grows or shrinks the heap
  - Memory allocators use `mmap` to bring in large blocks of memory

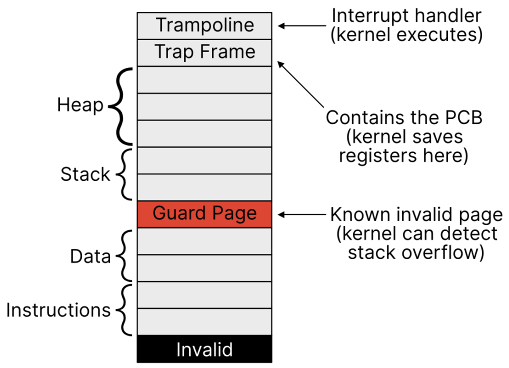

Trampoline
Trap Frame
Interrupt handler (kernel executes)
Heap
Contains the PCB (kernel saves registers here)
Stack
Guard Page
Known invalid page (kernel can detect stack overflow)
Data
Instructions
Invalid

Figure 16: Kernel allocation of the virtual address space.

- The kernel can also map its memory directly to processes' virtual memory as a fixed virtual address, so the process can make syscalls without actually doing a syscall
  - e.g. `clock_gettime()`

Lecture 14, Feb 7, 2024

Dynamic Priority Scheduling

- Feedback scheduling is a scheme where the scheduler itself manages the priories of the processes

  - Processes that don't use their time slice has their priority increased, while priorities that do use their full time slice has their priority decreased
  - Each process starts with some priority P_n
  - The scheduler always picks the lowest priority number to schedule
    * If the process yields (gives up execution, e.g. due to I/O), switch to another process
    * If another process with lower priority becomes ready, switch to that process and preempt the original one
  - The time each process executes for during the time slice is recorded as C_n
  - At the end of the time slice, the priority of the process is updated as $P_n \leftarrow \frac{P_n}{2} + C_n$, and then $C_n \leftarrow 0$
    * Note priorities are only *updated* at the end of time slices or priority intervals
    * When a process with lower priority is ready, we compare the current priorities without updating first
- Example below shows dynamic priority scheduling for processes X, Y, A, B arriving in order
  - X and Y are I/O bound processes that execute for 1 time unit and block for 5
  - Timer interrupts occurs every time unit, time slices are 10, priority interval is 10

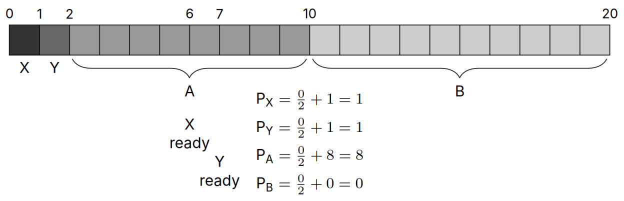

0
1
2
6
7
10
20
X
Y
A
B
X ready
Y ready
$P_X = \frac{0}{2} + 1 = 1$
$P_Y = \frac{0}{2} + 1 = 1$
$P_A = \frac{0}{2} + 8 = 8$
$P_B = \frac{0}{2} + 0 = 0$

Figure 17: Scheduling for the case above if all processes start with priority 0.

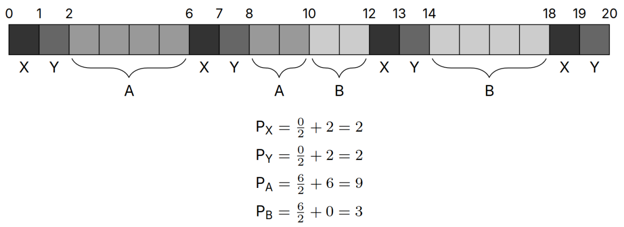

0
1
2
6
7
8
10
12
13
14
18
19
20
X
Y
A
X
Y
A
B
X
Y
B
X
Y
$P_X = \frac{0}{2} + 2 = 2$
$P_Y = \frac{0}{2} + 2 = 2$
$P_A = \frac{6}{2} + 6 = 9$
$P_B = \frac{6}{2} + 0 = 3$

Figure 18: Scheduling for the case above if A and B have priority 6, X and Y have 0.

Memory Mapping

- The `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);` syscall is used to map files to a process' virtual memory space
  - This allows us to map a file's contents to memory, so instead of reading from the file, we can read from the virtual addresses
    * Note writing back to the virtual address doesn't write back to the file
    * The contents of the file are copied into memory on-demand
  - Arguments:
    * `addr`: suggested virtual address to map to (`NULL` will let the kernel pick)
    * `length`: number of bytes to map
    * `prot`: protection (permission) flags (read/write/execute)
    * `flags`: mapping flags (shared/private/anonymous)
      - This specifies the behaviour when the process gets forked
      - Private means the child process can't access the same virtual addresses; shared means both processes will share the memory
      - Shared memory mapping is a way to implement IPC

                - Anonymous means there is no underlying file (use memory)
            * `fd`: file descriptor to map
                - If this is set to -1 (in conduction with using the anonymous flag) this will be mapped to memory instead of a file
            * `offset`: offset in the file to start the mapping at
                - This needs to be a multiple of the page size for alignment reasons
        - Use `int munmap(void *addr, size_t length);` to undo the mapping
- `mmap` calls are lazy and only sets up page tables
    - The newly created PTE is invalid, so on first access, the MMU triggers a page fault, the kernel detects this and copies over the page
    - Only the parts that get used get read
- `mmap` can be useful for random access of a file and to only read parts of the file that we actually need instead of reading in the entire file

Lecture 15, Feb 13, 2024

Threading

- *Concurrency*: switching between two or more tasks (interrupting the tasks to context switch)
    - The goal is to make progress on multiple things
- *Parallelism*: running two or more things independently at the same time
    - The goal is to run as fast as possible
- *Threads* are like processes, but the memory is shared
    - Registers, program counter, and stack are still independent
    - Address space is shared, so if one thread modifies memory, all other threads see it
    - To get memory specific to a thread, we need to specify *thread-local storage* (TLS)
    - One process can have multiple threads
- Due to fast context switching, threads can execute concurrently even with just a single CPU
- Threads are lighter than processes and faster to run because:
    - Code/data/heap is shared
    - Cheap creation (no need to copy resources like page tables)
    - Cheap context switching (no need to flush caches like the TLB)
- Threads live within an executing process (unlike processes which can execute independently); when a process dies, all its threads will die with it
    - When a thread dies, only its stack is removed from the process
    - So once the main thread dies, all other threads immediately stop executing
        * There is no such thing as orphan threads, but there can be zombie threads since their resources don't get released until you wait
    - There is no parent-child relationship for threads

```
int pthread_create(pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void*),
                   void* arg);
```

- To create a thread, use the `<pthread.h>` library
    - Arguments:
        * `thread`: output handle to a thread struct that will be populated
        * `attr`: thread attributes
        * `start_routine`: a function pointer to start execution at
        * `arg`: additional argument to pass to `start_routine`
    - Returns 0 on success or error number otherwise
    - Unlike `fork()`, the new thread starts executing at a specified different location instead of the current location

- `int pthread_join(pthread_t thread, void **retval);` will wait for the thread to terminate before returning
  - This is the equivalent of `wait()` for threads
  - The pointer `**retval` is set to the location of the `void*` returned by the thread function
  - Note calling this more than once on a thread leads to undefined behaviour!
- `void pthread_exit(void *retval);` will terminate a thread early with the specified return value
  - This is called implicitly when the function of a thread returns
- *Joinable* threads are the default kind, which wait for a thread to call `pthread_join()` before releasing its resources
  - We can *detach* threads via `int pthread_detach(pthread_t thread);`
    - These threads are non-joinable, so calling join on them is undefined behaviour
    - Calling detach on a detached thread is undefined behaviour
  - Calling `pthread_exit(NULL);` in the main thread will keep the process alive until all other threads have exited
    - This is useful for detached threads since we can't join them
- Thread attributes such as the stack size can be set explicitly using `pthread_attr` functions

Lecture 16, Feb 14, 2024

Threading Implementations

- Threading can be implemented either as user threads, or kernel threads
  - User threads run completely in the user-space; the kernel doesn't treat the process any differently
  - Kernel threads are managed by the kernel and gets treated specially
- In both models, threads require a thread table (just like a process' process table); this is either in user or kernel space depending on implementation
- Generally threading libraries can work in one of three ways:
  - Many-to-one: many user threads are mapped to a single kernel thread; threads are completely managed in user space
    - The kernel only sees a single process
    - Since everything is done in user space, thread creation/deletion is fast and no context switching is needed
    - However if one thread blocks, all other threads are blocked since the kernel can't distinguish
    - Does not allow parallelism since the kernel only sees a single process
  - One-to-one: each user thread maps to a single kernel thread
    - The kernel handles everything while the threading library is just a thin wrapper
    - Threads are slower, but can execute in parallel; one thread can't block everything
    - This is what `pthread` is
  - Many-to-many: many user threads map to many kernel threads
    - We have more user threads than kernel threads
    - Can set the number of kernel threads to the same as the number of CPU cores to fully allow parallelism
    - In theory gives the best of both worlds, but in reality can be very complicated and unpredictable
- What happens when we call `fork()` on a multithreaded program?
  - On Linux the new process has only one thread, corresponding to the one that called `fork()`
  - `pthread_atfork()` can be used to register functions to be run on fork, for advanced control of forking behaviour
- If a multithreaded process gets sent a signal, on Linux only one random thread will receive the signal
  - We have no control over the thread that gets interrupted
- Instead of many-to-many thread mappings, a *thread pool* is often used
  - A thread pool maintains a number of threads, usually corresponding to the number of CPUs in the system
  - When no tasks are given, the threads are sleeping; as tasks come in, the threads get waken up and starts executing the tasks

    - These threads are reused after tasks are done
    - This is often done when you have lots of very small tasks, so thread creation can introduce significant overhead

Lecture 17, Feb 16, 2024

Note: This was a midterm review lecture. See posted slides. No notes for this lecture since there was no new content.

Lecture 18, Feb 27, 2024

Sockets

- Sockets are another form of IPC that allows communication over a network (in addition to on the same machine)
    - All network connections have to go through sockets
- After sockets are set up, a file descriptor is returned that we can `read()` and `write()` to as usual and `close()` when done
- The server follows these steps:
    1. `int socket(int domain, int type, int protocol);`: Creates the socket
        - `domain` specifies the general protocol
            * `AF_UNIX`: local communication of the same machine
            * `AF_INET`: IPv4
            * `AF_INET6`: IPv6
        - `type` is either `SOCK_STREAM` or `SOCK_DGRAM` (TCP or UDP)
            * For stream connections, the data sent arrives in the same order; we have a persistent connection (we'll know when we lose it), which is reliable but slow
            * For datagram connections, there is no guarantee of arrival order and connection persistence, but is faster
        - `protocol` further specifies the protocol and is mostly unused
        - Returns a file descriptor (but for a server we shouldn't read/write to this)
    2. `int bind(int socket, const struct sockaddr *addr, socklen_t addr_len);`: Attach the socket to some location (a file, IP and port, etc)
        - 3 different types of `sockaddr` structures: `sockaddr_un` (UNIX socket, i.e. a path), `sockaddr_in` (IPv4 address), `sockaddr_in6` (IPv6 address)
        - `addr_len` is `sizeof(sockaddr)`
        - Set `sun_type` of the `sockaddr` struct to the same as the domain of the socket and `sun_path` to the path (note this is a `char[]`, not `char*`, so size is limited)
        - For UNIX sockets, we should use `int unlink(const char *pathname);` to clean up the socket path (after closing the socket); otherwise the file corresponding to the socket will remain
    3. `int listen(int socket, int backlog);`: Listen for connections on the socket and sets the queue limit
        - `backlog` is the limit of outstanding connections queue, managed by the kernel; passing 0 uses the default kernel queue size
            * If the queue is full, new connections will not be allowed
    4. `int accept(int socket, struct sockaddr *address, socklen_t *address_len);`: Accept an incoming connection
        - `address`, `address_len` is an optional return of the connecting address (`NULL` to ignore)
        - This returns a file descriptor we can read and write to, corresponding to the new client connection
        - Will block until a client connects
- The client follows these steps:
    1. `int socket(int domain, int type, int protocol);`: Creates the socket

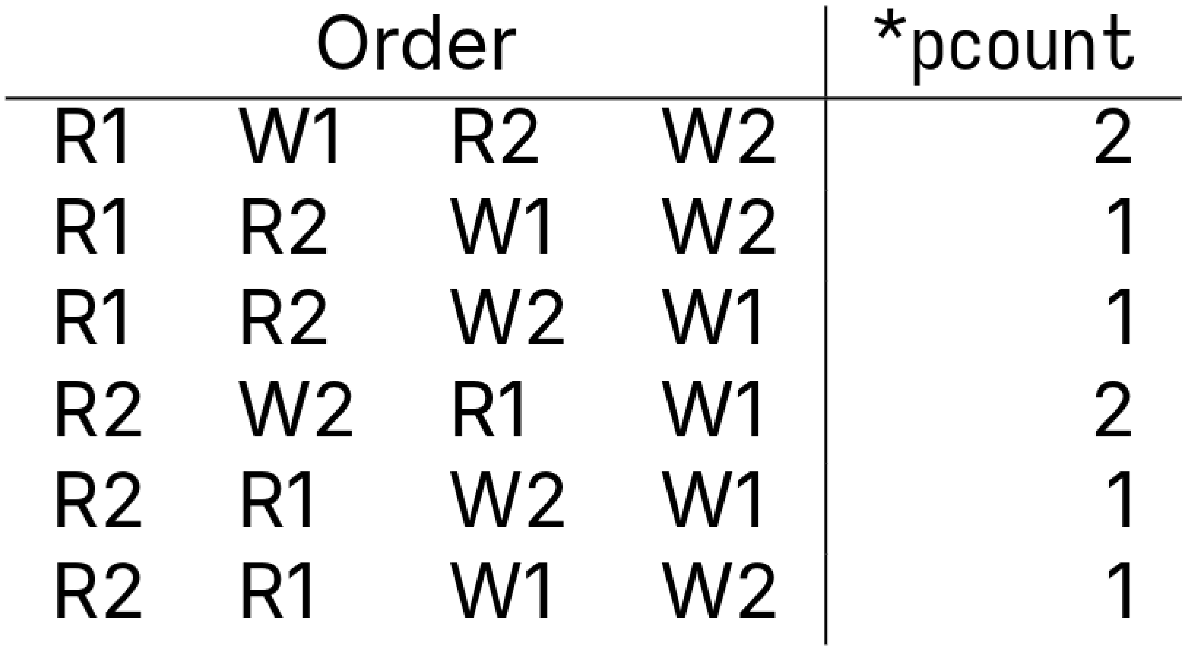
2. `int connect(int sockfd, const struct sockaddr *addr, socklen_t addr_len);`: Connects to some location, giving a file descriptor
    - This will use the same name as the `bind()` call of the server
    - On success, `sockfd` may be used as a normal file descriptor (the function returns 0 on success)

- Instead of read and write, we can use `send()` and `recv()` syscalls, which are similar but take additional flags
  - e.g. `MSG_OOB` (send/receive out-of-band data), `MSG_PEEK` (look at data without reading), `MSG_DONTROUTE` (send without routing packets)
  - `sendto()` and `recvfrom()` take an additional address
    - Ignored for stream sockets since there's a persistent connection

Lecture 19, Feb 28, 2024

Locks

- When two concurrent threads access the same variable and at least one of them writes to it, a *data race* can occur
  - When this happens, we can get an inconsistent view of memory
- An *atomic* operation is an indivisible operation that cannot be interrupted
  - The thread can only be preempted between two atomic operations but not during one
- Compilers use an intermediate representation called *three address code* (TAC)
  - Mostly used for analysis and optimization by compilers
  - We can use this to reason about data races since it's low level but easier to read than ASM
  - Consists of only individual (atomic) statements, each taking at most 2 operands
  - GCC's TAC is called GIMPLE; use `-fdump-tree-gimple`/`-fdump-tree-all` to see it
- Example: two concurrent threads incrementing a shared counter which starts at 0
  - Each increment consists of a read, increment, and then write back
  - If the reads and writes are interleaved, one thread may read the value of the counter before the other is done incrementing it, so they will overwrite each other's results
  - Depending on the specific ordering of reads and writes, the result may be different



Order				*pcount
R1	W1	R2	W2	2
R1	R2	W1	W2	1
R1	R2	W2	W1	1
R2	W2	R1	W1	2
R2	R1	W2	W1	1
R2	R1	W1	W2	1

Figure 19: All possible orderings and results of two concurrent threads incrementing a counter, starting at 0.

- To avoid data races, we need to prevent two threads from accessing the variable at the same time
- We can use a *mutex* (stands for Mutual Exclusion)
  - `pthread_mutex_t` can be used
  - Use `pthread_mutex_init()` or assign to `PTHREAD_MUTEX_INITIALIZER` to init the mutex
  - Use `pthread_mutex_destroy()` to destroy the mutex
  - Between a call to `pthread_mutex_lock()` and `pthread_mutex_unlock()`, we have a *critical section* (or *protected*), where only a single thread can execute at a time
    - A thread can only enter this section if it can acquire the lock
    - The lock can only be acquired by a single thread at a time
  - Use `pthread_mutex_trylock()` to attempt to acquire the lock in a non-blocking manner
- If we wrap the counter increment between a `pthread_mutex_lock()` and `pthread_mutex_unlock()`, we won't ever see a data race
- Critical sections should have the following properties:
  - Safety (aka mutual exclusion)
    - Only a single thread should be in the critical section at a time
  - Liveness (aka progress)
    - If multiple threads reach the critical section, only one can proceed
    - The critical section can't depend on other threads (which can lead to deadlock)
  - Bounded waiting (aka starvation-free)
    - A thread waiting to acquire a lock must eventually proceed
- The locking mechanism should have the following properties:
  - Efficiency: shouldn't consume resources when waiting
  - Fair: each thread should wait approximately the same amount of time
  - Simple: should be easy to use, hard to misuse
- Synchronization can happen on different levels
  - At the lowest level we have hardware atomic operations
  - Then we have high-level synchronization primitives like mutexes
  - Finally we have properly synchronized applications without data races
- For a single processor system, locks are very easy to implement – simply disable interrupts during the critical section

Lecture 20, Mar 5, 2024

Locks Implementation

- Locks can be implemented with minimal hardware, assuming atomic loads and stores and in-order execution
  - However these do not scale well and real processors often execute out-of-order
- Assume there exists an atomic hardware instruction "compare and swap" (e.g. `cmpxchg` on x86), which checks if a value is equal to something, and assigns something else to it if it is, while returning the old value, then the lock can be implemented with:

```c
void init(int *l) {
    *l = 0;
}
void lock(int *l) {
    while (compare_and_swap(l, 0, 1));
}
void unlock(int *l) {
    *l = 0;
}
```

- The above is a *spinlock*

- The first problem is the busy wait – if a thread cannot acquire the lock, it should yield and not keep taking CPU time
  - We can add a yield inside the `while` loop
- The second problem is the *thundering herd* problem – if multiple threads are waiting on the lock at the same time, we don't know which one will get the lock when it becomes available
  - We want threads to get the lock in FIFO order to ensure fairness
  - We can add a wait queue to the lock and wake up the next thread on unlock

```c
void lock(int *l) {
    while (compare_and_swap(l, 0, 1)) {
        // add myself to the lock wait queue
        thread_sleep();
    }
}
void unlock(int *l) {
    *l = 0;
    if (/* threads in wait queue */) {
        // wake up one thread
    }
}
```

- This has 2 issues: lost wakeup (a thread goes to sleep and never wakes up) and wrong thread getting the lock
  - If a context switch happens after a thread calls `compare_and_swap` to lock (and fails), but before it inserts itself to the wait queue, the thread calling unlock will not wake it up and this thread will sleep forever
  - If a context switch happens after a thread unlocks but before it wakes up the next thread in queue, and the thread switched to acquires the lock immediately, that thread will have the lock before the threads in queue

```c
typedef struct {
    int lock;
    int guard;
    queue_t *q;
} mutex_t;

void lock(mutex_t *m) {
    while (compare_and_swap(m->guard, 0, 1));
    if (m->lock == 0) {
        m->lock = 1; // acquire mutex
        m->guard = 0;
    } else {
        enqueue(m->q, self);
        m->guard = 0;
        thread_sleep();
        // wakeup transfers the lock here
    }
}

void unlock(mutex_t *m) {
    while (compare_and_swap(m->guard, 0, 1));
    if (queue_empty(m->q)) {
        // release lock, no one needs it
        m->lock = 0;
    }
    else {
        // direct transfer mutex
        // to next thread
        thread_wakeup(dequeue(m->q));
    }
    m->guard = 0;
}
```

- To fix this, we can combine both a spin lock and a wait queue, using a *lock* and *guard*
  - The guard is a spin lock for `lock()` and `unlock()` and ensures while one thread is trying to lock, other threads cannot try to unlock or vice versa
  - The lock itself still has a yield and wait queue to ensure efficiency and fairness
- One final data race remains – in `lock()`, we unlock the guard after putting the current thread in queue, but before going to sleep
  - If a context switch happens after guard unlock but before sleep, then the thread attempting to unlock will be trying to wake up a thread that never went to sleep
  - However, this is easily detected, and we can simply just try again (keep trying to wake up until the thread sleeps)
- Data races only occur if a write is happening while threads are trying to read, so if we have many threads reading at the same time, we won't get a data race
  - We can have different locking modes for read and write, so multiple reads can happen at the same time; these are known as *read-write locks*
  - Multiple threads can hold a read lock, but only one may hold a write lock at a time
    * Attempting to acquire a write lock will wait for all read locks to be unlocked first
    * Attempting to acquire a read lock while another thread has the write lock will wait for the write lock to be unlocked
  - `pthread_rwlock_rdlock` and `pthread_rwlock_wrlock` implements this functionality
  - Read-write locks can be implemented using two mutexes as shown below

```
typedef struct {
    int nreader;
    lock_t guard;
    lock_t lock;
} rwlock_t;

void write_lock(rwlock_t *l) (
    lock(&l->lock);
}
void write_unlock(rwlock_t *l) (
    unlock(&l->lock);
}

void read_lock(rwlock_t *l) (
    lock(&l->guard);
    ++nreader;
    if (nreader == 1) { // first reader
        lock(&l->lock);
    }
    unlock(&l->guard);
}
```

```c
void read_unlock(rwlock_t *l) (
    lock(&l->guard);
    --nreader;
    if (nreader == 0) { // last reader
        unlock(&l->lock);
    }
    unlock(&l->guard);
}
```

- Note the above simplified implementation allows for starvation of write, if reads keep coming in (i.e. it's not fair for write)

Lecture 21, Mar 6, 2024

Semaphores

- How can we ensure some fixed order of execution between threads?
- *Semaphores* are shared values between threads/processes that are used for signaling
  - They have a `value` that is an unsigned integer, which can be initialized to anything
    * Setting this to some initial number sets the number of `wait`s that can occur at a time without `post`
  - Two fundamental operations:
    * `wait`: decrement the value atomically; if the value is 0, it waits until a `post` increments the value again before decrementing it and returning
    * `post`: increment the value atomically
- Semaphores are offered in the `<semaphore.h>` library
- Use `int sem_init(sem_t *sem, int pshared, unsigned int value);` to initialize a semaphore
  - `pshared` specifies whether the semaphore should be shared between forked processes
  - Use `sem_destroy(sem_t *sem);` to destroy
- Use `int sem_wait(sem_t *sem);` and `int sem_trywait(sem_t *sem);` for wait and `int sem_post(sem_t *sem);` for post
- If we want to make one line always execute before another, we can use a semaphore initialized to 0
  - Call `wait` before the line that executes second, which cannot return until `post` is called by the other thread
  - Call `post` after the line that executes first, to indicate that the line has been run
  - If we want a third line to execute after the previous two, we cannot reuse the same semaphore because we can't control whether the second line or third line runs first (even if we `post` twice or initialize to 1)
    * This would require a second semaphore
  - If we initialized the semaphore to 1 instead, the first thread won't block when it `wait`s initially

```c
static sem_t sem; /* New */
void* print_first(void* arg) {
    printf("This is first\n");
    sem_post(&sem); /* New */
}
void* print_second(void* arg) {
    sem_wait(&sem); /* New */
    printf("I'm going second\n");
}
int main(int argc, char *argv[]) {
    sem_init(&sem, 0, 0); /* New */
    /* Initialize, create, and join threads */
}
```

- Semaphores can be used like mutexes; instead of lock we just use `wait` and instead of unlock we use `post`, and initialize the value to 1
  - This is often a bad idea since it depends on the initialization of the semaphore, which could be far from the code that actually uses it
- Example: suppose we have a circular buffer, which producers write to and consumers read from; all consumers share an index and all producers share an index
  - The producer shouldn't write to the buffer if it's full
  - The consumer shouldn't read from the buffer if it's empty
  - To ensure this, use a semaphore to track the number of empty slots (for the producers) and another to track the number of full slots (for the consumers)

```
void init_semaphores() {
    sem_init(&empty_slots, 0, buffer_size);
    sem_init(&filled_slots, 0, 0);
}
void producer() { while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
    sem_post(&filled_slots); /* New */
} }
void consumer() { while (/* ... */) {
    sem_wait(&filled_slots); /* New */
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
} }
```

Lecture 22, Mar 8, 2024

Advanced Locking

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

- *Condition variables* work like semaphores; they maintain queues of threads
  - `wait` adds the calling thread to the queue for the condition variable, unlocks the mutex, and blocks
    * Calls to `wait` must already have acquired the mutex
    * One mutex can be used to protect multiple condition variables
  - When another thread calls `signal` or `broadcast`, if the thread is selected, it will get unblocked, tries to lock the mutex, and returns from `wait` if the mutex is successfully locked
    * `signal` wakes up any thread waiting, `broadcast` wakes up all of them
  - The mutex is used to protect variables that are a part of some more complex condition
  - We usually use `while` to check the condition instead of just `if` to account for the possibility of the condition updating before wakeup/locking
- Semaphores are a special case of condition variables, protecting an integer, going to sleep when the value is 0 and waking up when the value is greater than 0
  - One can be implemented with the other but it can get complex
  - Condition variables are favoured for more complex conditions since it improves code readability

- Example: producer-consumer with condition variables

```
pthread_mutex_t mutex;
int nfilled;
pthread_cond_t has_filled;
pthread_cond_t has_empty;

void producer() {
    // produce data
    pthread_mutex_lock(&mutex);
    while (nfilled == N) {
        pthread_cond_wait(&has_empty, &mutex);
    }
    // fill a slot
    ++nfilled;
    pthread_cond_signal(&has_filled);
    pthread_mutex_unlock(&mutex);
}

void consumer() {
    pthread_mutex_lock(&mutex);
    while (nfilled == 0) {
        pthread_cond_wait(&has_filled, &mutex);
    }
    // empty a slot
    --nfilled;
    pthread_cond_signal(&has_empty);
    pthread_mutex_unlock(&mutex);
    // consume data
}
```

- *Granularity* is the size of our critical sections; do we lock large sections or divide it into multiple smaller locks on smaller sections?
  - Locking can have overhead (memory, init/destruction, acquire/release time), which increases with the number of locks
  - More granular locks can increase performance through more parallelization, but increases the potential for deadlocks
- More locks increases the possibility of *deadlocks*, when threads are waiting forever
  - Deadlock conditions:
    1. Mutual exclusion
    2. Hold and wait – after acquiring a lock, attempting to acquire another lock
    3. No preemption – can't take locks away
    4. Circular wait – waiting for a lock held by another process
  - Example: two threads both need 2 locks; thread 1 acquires lock 1 first and tries to acquire lock 2, in the meantime thread 2 acquires lock 2 first and tries to acquire lock 1; now both are waiting on each other
    * This example can be prevented by enforcing order of locking and unlocking
    * Use a function that always locks them in the same order, and unlocks them in the opposite order
    * Another fix is to use `trylock` for the second one; if it doesn't succeed, give up the first lock for some time and try again

Lecture 23, Mar 12, 2024

Lecture 24, Mar 13, 2024

SSDs and RAID

- SSDs use NAND flash, which can only be read in complete pages and write to freshly erased pages
- Erasing is done per-block (128 or 256 pages each)
  - Erasing a block can be a hundred times slower than reading a page or ten times slower than writing a page
- SSDs will garbage collect blocks
  - The disk controller does not know what blocks are still alive
  - The OS can use the TRIM command to inform the controller that a block is unused
- To store lots of data, we can use a Single Large Expensive Disk (SLED) or an Redundant Array of Independent Disks (RAID)
  - SLED uses a single large disk and presents a single point of failure
  - RAID has redundancy to prevent data loss and increase throughput
- RAID 0 (aka *striped volumes*) distribute data in block-level stripes (128 or 256 KiB) over multiple disks; used for performance at the cost of increased risk of failure
  - e.g. a file can be broken up into stripes, and odd stripes can go on one disk while even stripes can go on the other
  - RAID 0 increases performance since now we can read/write to multiple disks in parallel; read/write performance is increased by a factor of N
  - If any of the disks in the array fails, we will lose data, so the number of points of failure is increased
  - Typically only 2 is used for a balance between performance and risk of failure

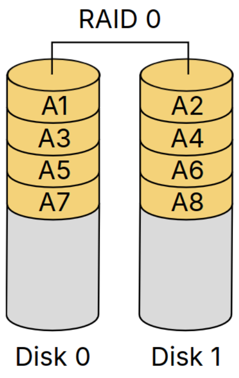


Figure 20: RAID 0.

- RAID 1 (aka *mirror*) simply duplicates data across all disks identically; used for redundancy, at the cost of being wasteful
  - Redundancy means as long as at least one disk remains, no data will be lost
  - Can still increase read performance by reading different parts from different disks, but writes are not any faster (since we need to write to all drives)
  - This is wasteful since the data we can store is the same as a single disk; we can only use $\frac{1}{N}$ of the total space
- RAID 10 (aka *stripe of mirrors*) is RAID 1 + 0 and uses both; RAID 1 is used at the top level, and RAID 0 is used at the bottom level
  - We have multiple identical copies of RAID 0 arrays
  - If N mirrors of M stripes are used, read performance is increased by a factor of NM and write performance by a factor of M
  - Whenever any matching pair of N disks storing the same stripes dies, we have data loss

    * In the best case we can lose up to half the drives without data loss
    * In the worst case we can only tolerate 1 failure
  - No parity involved so replacement of disks is faster than RAID 4+

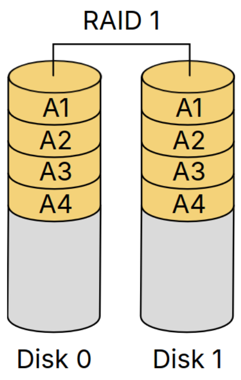


Figure 21: RAID 1.

- RAID 4 uses a dedicated disk for *parity*, which stores the XOR of the other copies; provides a balance between redundancy and performance
  - If any one of the disks fails, we can use the other disks to reconstruct the one that was lost
    * XORing gives whether there are an even or odd number of 1s, so by comparing the bits on the other disks we can deduce the missing bit
    * But if we lose more than one at a time, the missing disk cannot be reconstructed
  - Data can be distributed in block-level stripes across the other disks, so read (and theoretically write) performance is increased by a factor of $N - 1$
    * Note RAID 2 and 3 use bit-level and byte-level striping (instead of block-level) and are practically very bad ideas
  - Since every write must update the parity disk, the parity disk gets written to much more than the other disks, which can cause it to fail first
    * Write performance in practice can also suffer since everything must write to the parity disk
  - We can use $1 - \frac{1}{N}$ of the total available space
  - Note this needs at least 3 disks

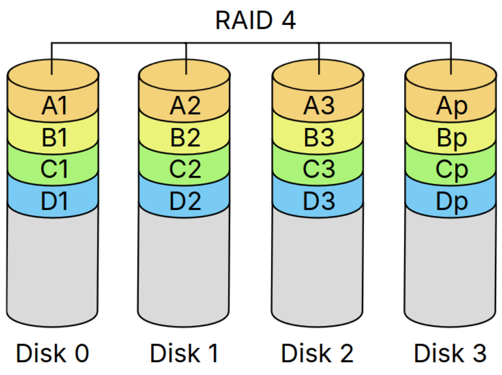


Figure 22: RAID 4.

- RAID 5 distributes the parity information across all disks instead of putting it on a single disk
  - Used a lot in practice
  - This is better for write performance and doesn't cause a single disk to fail first
  - Has the same benefits as RAID 4
- RAID 6 adds another checksum block compared to RAID 5

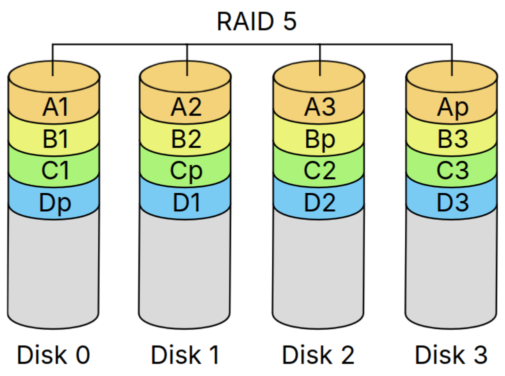


Figure 23: RAID 5.

- Performance is sped up by a factor of $N-2$ and we can use $1-\frac{2}{N}$ of the total space
- Redundancy is increased to tolerate at most two disks failing at the same time
- Note this requires at least 4 disks (in practice often 5 or more)
- Write performance is slightly worse than RAID 5 due to the increased checksum calculation

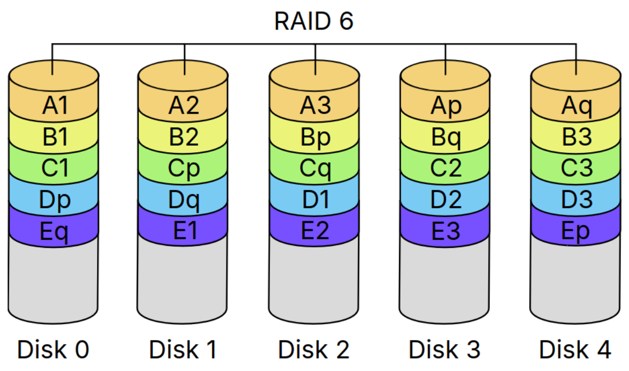


Figure 24: RAID 6.

Lecture 25, Mar 15, 2024

Filesystems

- File access can be sequential (within a file) or random
  - The `read()` and `write()` syscalls will perform sequential access
  - For random access use `off_t lseek(int fd, off_t offset, int whence);` to set the offset
    * `whence` is either `SEEK_SET` (absolute/relative to start of file), `SEEK_CUR` (relative to current offset) or `SEEK_END` (relative to end of file)
- Each process stores a local open file table, which is indexed by file descriptor number
  - Each item in the local file table points to a location in a system-wide *global open file table*
  - Each GOF location stores the position (offset), flags, and a virtual node pointer (data can be read from/written to it) for each opened file
    * VNodes can represent files, pipes, sockets, etc
- On fork, the PCB is copied, including the local open file table; however, they would still point to the same entries in the GOF
  - This means both processes share the same position; read, write, or seek by one process will affect all others

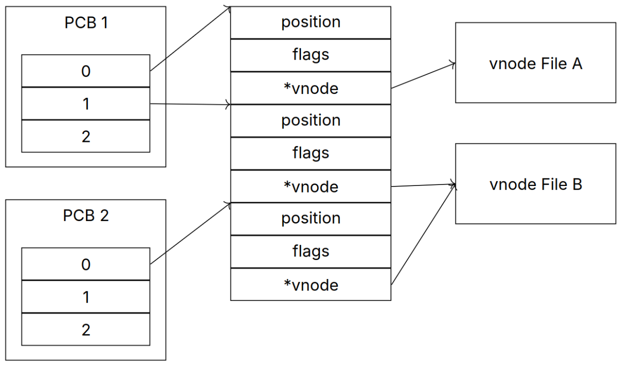


Figure 25: Structure of file tables.

  - To get an independent copy, we need another open

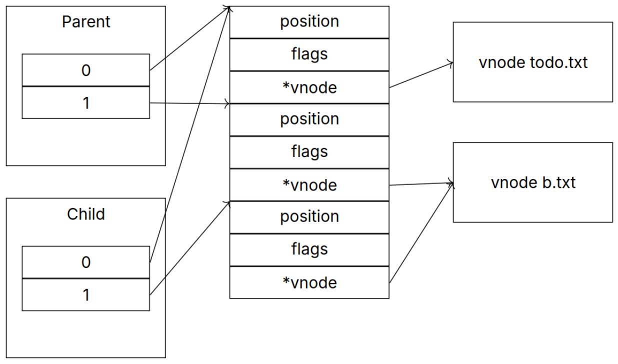


Figure 26: File table structure created by `open("todo.txt", O_RDONLY); fork(); open("b.txt", O_RDONLY);`.

File Allocation

- How do we store a file?
- We can use contiguous allocation and always use sequential blocks, like an array
  - Space efficient: only needs to store start block and number of blocks
  - Random access is fast since we can simply calculate the block number
  - Very slow if files need to expand, just like how we have to copy an array to expand it
  - Susceptible to *internal fragmentation* (not filling an entire block) and *external fragmentation* (wasted blocks between files)
  - Not used in practice
- Linked allocation uses a linked list-like structure where each block points to the next
  - Files can grow very easily and there is no external fragmentation (still internal fragmentation however)
  - Random access is very slow since we have to traverse all the blocks (and read them from disk) to find the block we want
  - Not used in practice
- File allocation tables (FAT) moves the linked list to a separate table; each entry in the table points towards a block used by a file
  - One of the most simple filesystems and very commonly used in low-end devices
  - Since the table is moved out, this is more performant than linked allocation
  - Can grow and shrink easily, no external fragmentation
  - Random access is much faster since the table can be held in memory/cache

  - Table size has to grow linearly with disk size so this can be very large

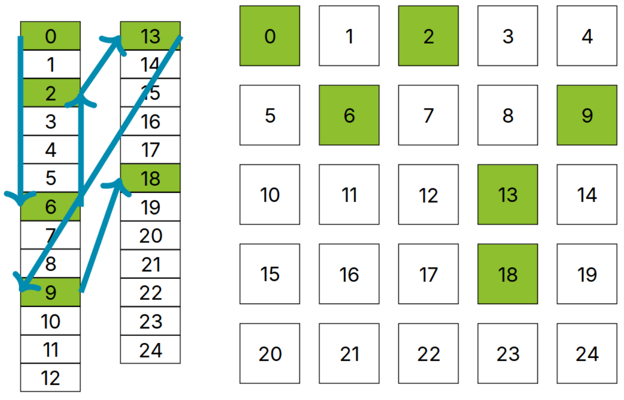


Figure 27: File allocation table (FAT) filesystem.

- Indexed allocation uses an array of pointers to blocks to keep track of which blocks are used by files
  - Provides even faster random access than FAT, and table size has no relation to disk size
  - Each new file will have a block that stores its table
    * This means the max size of files is limited by the number of pointers we can store in the block
  - e.g. if each block is 8 KiB and pointers are 4 bytes, files can only be 16 MiB large

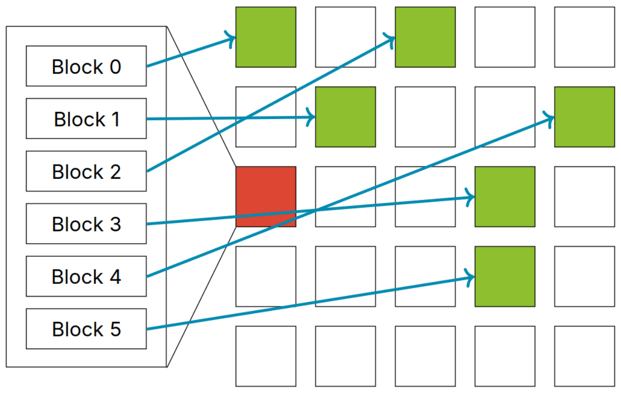


Figure 28: Indexed allocation.

Lecture 26, Mar 19, 2024

inodes

- *inodes* (*index nodes*) are how files are stored in a filesystem
  - Each inode stores the file size, type, number of hard links (to know when to erase the file), access rights, creation/modification timestamps, sometimes file contents, and an ordered list of data blocks
  - This is an alternative to other formats such as FAT; UNIX-style filesystems use them
- inodes store pointers to file blocks in addition to metadata
  - To be efficient for all file sizes, the file contents can have different levels of indirection
  - A typical Linux inode has 15 pointers in total, 12 are direct, 1 single, 1 double, and 1 triple indirect
    * The direct pointers directly point to blocks used for the file
    * The higher level indirection pointers point to blocks that store additional pointers, like with page tables

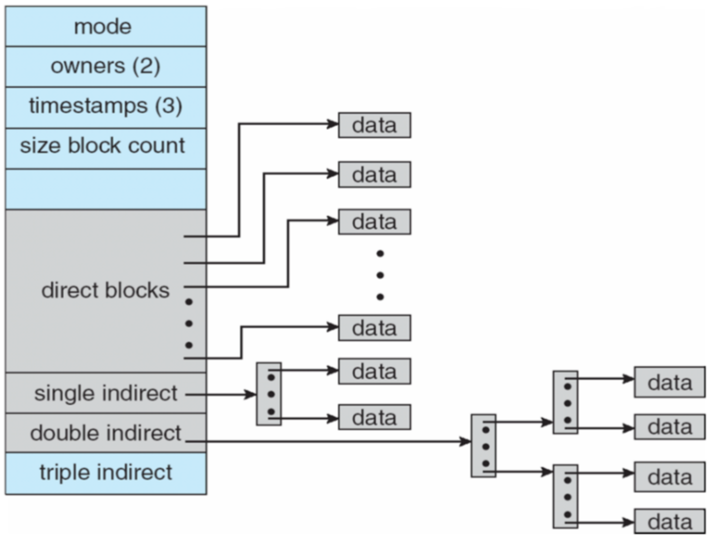


Figure 29: Structure of a Linux inode.

  - With this setup the maximum size of a file is approximately 64 TiB
  - Another optimization is to store files that are less than 15×4 bytes directly in the inode in place of the pointers, instead of pointing to data blocks
- A directory entry (aka filename) is called a *hard link*; each hard link points to one inode
  - Multiple hard links can point to the same inode; modifying the inode through any of the links will change the contents
    * Additional hard links can be created with `ln <src> <dest>`
  - `ls -li` will show the inode number linked to by each filename and the number of hard links to the inode
  - When we make a directory, its `.` and `..` will be hard links to inodes, increasing the link count
  - Hard links form a DAG (aside from the self-loops of `.`)
- Deleting a file only removes a hard link (hence the syscall for this is `unlink` and not delete)
  - When there are no more hard links to an inode, it can be recycled and its blocks reused
- A *soft link* (or *symbolic link*/*symlink*) is a pointer to another file on the system
  - These can be created with `ln -s <src> <dest>`
  - When attempting to access the symlink, the filesystem is redirected to that file
  - Soft link targets do not need to exist; they can be created with a nonexistent target, or the target can be deleted without notice of the soft link
  - If the target does not exist, attempting to resolve the link leads to an error (`ENOENT`/no such file or directory)
  - The size of a soft link is the size of the name it points to
- Soft links can point to each other, and create cycles
  - The kernel will detect this when attempting to resolve the link (`ELOOP`/too many levels of symbolic links)
- inodes can have different types to represent files, directories, block devices, etc
  - Directory inodes do not store pointers to data blocks, but tuples of names and pointers to inodes
- Caching is often done in filesystems to speed up writing to disk
  - File blocks are cached in main memory in the *filesystem cache*
  - The blocks have temporal locality (referenced blocks are likely to be referenced again) and spacial locality (nearby blocks are likely to be referenced)
  - A kernel daemon thread periodically writes the changes to disk
    * A `sync` syscall forces the write immediately
- Filesystems can also support copy-on-write and other advanced features (e.g. btrfs)

- Deleting a file involves removing its directory entry, releasing the inode, and freeing the disk blocks, and a crash can happen at any time, which leads to a storage leak
- *Journaling* filesystems (e.g. ext4) record current operations in progress in a journal
  - This makes the filesystem more resilient; if an unexpected crash occurs, we can recover using the journal

Lecture 27, Mar 27, 2024

Memory Replacement

- We want the speed of faster memory with the capacity of larger memory; this is accomplished with caching
- *Demand paging* is when memory is used as a cache for the filesystem
  - Memory pages are mapped to filesystem blocks
  - When a block is used for the first time, it is loaded through a page fault handler
  - Memory pages can be moved into swap space if we need more physical memory than we have
- The number of pages a process uses at a given time is called its *working set*
  - If the working set cannot fit into physical memory, it will *thrash*, i.e. constantly move entries in and out of cache
- Page replacement algorithms:
  - Optimal: replace the page that will be idle for the longest
    - Used as a benchmark
    - Can't be implemented in practice because we have no way of knowing which pages will be used in the future
    - Theoretically has the fewest number of page faults
  - Random: replace random pages
  - FIFO (first-in first-out): replace the oldest page first
  - Least recently used (LRU): replace the page that hasn't been used in the longest time
    - For parallel accesses, we can use FIFO tiebreaking
- When evaluating page replacement algorithms, we care mainly about the number of page faults, since each is an expensive page read from disk

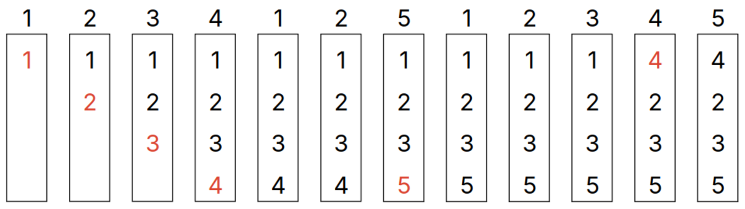


Figure 30: Optimal page replacement example. The number above each box is the page being accessed and the boxes are physical memory holding pages. Red denotes page faults. 6 total page faults.

- The examples below show FIFO replacement with a memory size of 4 pages compared to a size of 3
  - In both cases, the number of page faults is significantly more than optimal, as expected
  - Counter-intuitively, the example with a memory of 3 pages only has 9 faults while the larger memory had 10 faults
- The fact that having more page frames can cause more page faults is known as *Bélády's anomaly*
  - This is a problem with FIFO in particular and doesn't exist with LRU or "stack-based" algorithms
  - The anomaly is unbounded – you can construct a sequence to get any arbitrary page fault ratio
  - In general as the number of pages increases, the trend in page faults is random
  - For all other algorithms, having more page frames decreases the number of faults

- In practice random page replacement actually works better than FIFO since it avoids the worst case and has no anomaly

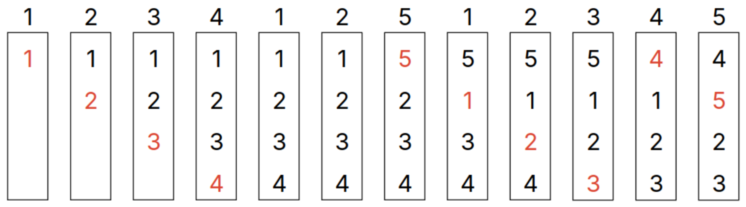


Figure 31: FIFO example. 10 total page faults.

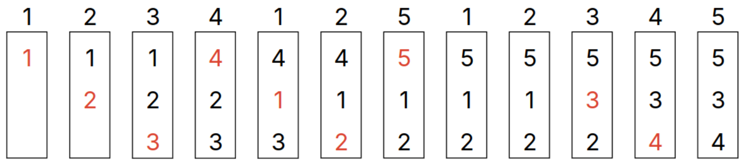


Figure 32: FIFO example with a smaller physical memory. 9 total page faults.

- LRU tries to approximate optimal replacement and is better than FIFO

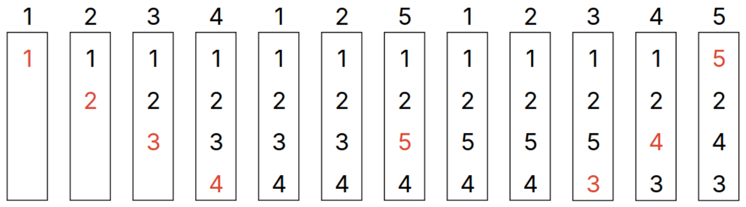


Figure 33: LRU example. 8 total page faults.

- Figuring out which page to replace for LRU can be expensive
  - Searching through all the pages to find the least recently used one is too expensive
  - We could use a doubly linked list that has all pages in least recently used order, and on each page access, we move the page to the front of the list, and always check the back of the list for page replacement
    - However this requires 6 pointer updates for each memory access, so we slow down memory access by a factor of 7
    - This can also bottleneck multiple processors since the list needs synchronization to avoid data races
  - Practically exact LRU is too slow to be implemented
- We can tweak the LRU algorithm to make it more efficient
  - We will look at the *clock algorithm* which approximates LRU
- Other algorithms include least frequently used (LFU), 2Q (hardware linked lists), adaptive replacement cache (ARC)

Lecture 28, Apr 2, 2024

Clock Page Replacement

- The clock replacement algorithm is an approximation of the LRU algorithm that is cheap to implement
  - Maintain a circular list of pages in memory, with each page having a reference bit, indicating whether it was recently accessed
    - The reference bit is usually stored in the PTE
  - An iterator or “hand” points to the next page to be replaced

- When inserting a new page, check the reference bit of the page under the hand; if it is zero, replace the page and advance the hand; if it is one, set it to zero, don't replace the page, advance the hand and check the next page
- When a page is accessed (that is already loaded), the reference bit is set to 1
  - This is normally done automatically by the MMU so we get it for free
  - The hand is not advanced

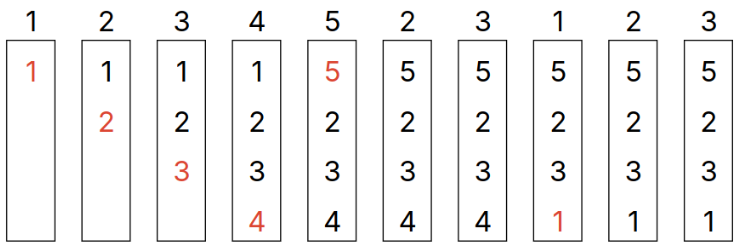


Figure 34: Clock replacement example. 6 total page faults.

- Initially we will load all pages into memory in order and fill the entire circular list
  - The first time we access an unloaded page, all loaded pages will have a reference bit of 1, so the hand has to go a full circle
  - After the hand goes a full circle and sets all reference bits to 0, we're back to the original page being pointed, which is the page being replaced
- For performance, we may choose to disable swapping altogether; sometimes it might make more sense to know that we've ran out of memory, rather than having things run slowly
  - Linux has an out of memory (OOM) killer which kills processes that use a lot of memory when the system runs out
- Increasing the page size allows for speedups; some systems use sizes such as 2 MiB (eliminate the lowest level of page table), or even 1 GiB (eliminate two levels of page tables)
  - 2 MiB is usually known as "huge pages" and 1 GiB is known as "gigapages"
  - Larger pages means we can cache more memory in the TLB
  - However this leads to more fragmentation (if we only use a small amount of memory, we still need to use an entire page)

Lecture 29, Apr 3, 2024

Memory Allocation

- When declaring normal variables on the stack, the compiler inserts `alloca()` calls, which allocate memory on the stack
  - The function that called `alloca()` frees the memory when it returns, so an explicit `free()` is not needed
- Dynamic allocation via `malloc()` can lead to *fragmentation*
  - A *fragment* is a small contiguous block of memory that is too small to be allocated; like a "hole" in memory which wastes space
    - Every allocation is permanent and contiguous, so between blocks of allocated memory there can be fragments
- Fragmentation requires the following:
  1. Different allocation lifetimes
     - e.g. stack allocation does not suffer from fragmentation since all variables live for the same time
  2. Different allocation sizes
     - e.g. page allocation does not have fragmentation because all pages are the same size, so any block of memory is the same

   3. Inability to relocate (defragment) previous allocations
      - e.g. Java does not have fragmentation since its garbage collector can move blocks of memory and update reference addresses
- *External fragmentation* occurs when allocating different sized blocks, and there is no more room for allocation between blocks
  - This is fragmentation between blocks
- *Internal fragmentation* occurs when allocating fixed blocks, and there is wasted space within a block
  - This is fragmentation within a block
  - e.g. if memory is only allocated in sizes of powers of 2
- To reduce fragmentation, we want to reduce the number of "holes" between blocks of memory
  - If we have holes, we want them to be as large as possible so we can fit future allocations in them
- Allocator implementations usually use a linked list of free blocks
  - When an allocation is needed, choose a free block large enough for the request, remove it for the free list and return it
    * Choosing which block to use requires a strategy
  - When freeing a block of memory, add it back to the free list
    * If it's adjacent to another free block, we can merge the two to get a larger chunk of free memory
- There are 3 general allocation strategies:
  1. Best fit: choose the smallest block that can satisfy the request
     - Requires searching through the entire list unless we come across an exact match
     - Too slow in practice
     - Tends to leave very large and very small holes; smaller holes may be useless
  2. Worst fit: choose the largest block
     - Also requires searching the entire list (can't stop early in this case)
     - Too slow in practice
     - Tends to be the worst in terms of storage utilization
  3. First fit: choose the fist block that can satisfy the request
     - Tends to leave "average" sized holes
     - Much faster and actually used in practice

Lecture 30, Apr 5, 2024

Buddy and Slab Allocators

- We will look at memory allocation schemes for the kernel
  - Algorithms are designed to be fast and less general than `malloc()`
- With allocators, we want fast searching of free blocks and merging of freed blocks
- *Buddy allocators* restrict all allocations to powers of 2, up to $2^k, 0 \leq k \leq N$; blocks are split into two recursively until the request can be handled
  - The implementation would use a free list for each block size, with $N + 1$ free lists in total
  - To allocate a block of 2^k, search the free list corresponding to k to find a block of that size
    * If there is no free block of that size, check the free list for $k + 1$ (i.e. double the size) and so on
    * After we find a block that is big enough, recursively break the block into two until we reach size 2^k as requested
  - Every time a block is broken into two, the two halves become *buddies*
    * The unused buddy is inserted back into the correct list as a free block
  - When a block is freed, if its buddy is also free, it is merged with its buddy to become a larger block
    * Blocks are recursively merged if needed
    * Due to the power-of-two setup, we can look at the memory addresses to easily check if two blocks are buddies by checking for equal bits in the addresses
- Linux uses buddy allocators
- Buddy allocators are fast and doesn't have external fragmentation (for the most part), but relies on the

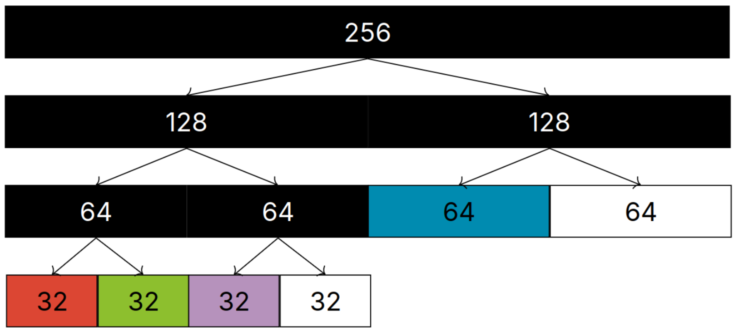


Figure 35: Buddy allocator structure. Black blocks are unavailable (broken), coloured blocks are used, white blocks are free. In this instance, the lists for 2^8 and 2^7 are both empty.

user to reduce internal fragmentation since allocated memory is always in powers of two

- *Slab allocators* allocate only fixed size chunks of memory; used for allocating objects of the same size from a pool
  - Each allocation size has a corresponding *slab* of slots; one slot holds one allocation
  - A bitmap is used to keep track of which blocks are in use
  - When allocating a block, set the corresponding bit and return the slot
  - When freeing a block, just clear the bit
  - This is how inodes and blocks are allocated in the ext filesystems
- Slab allocators completely prevent fragmentation, if we only need memory blocks of a fixed size
- We can mix the two allocators, e.g. allocating slabs with a buddy allocator or putting a slab allocator on top of a buddy allocator

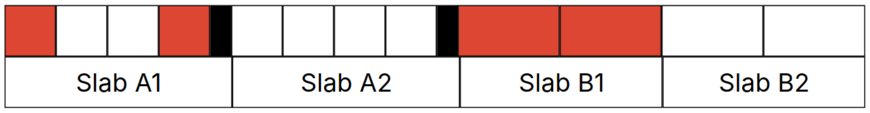


Figure 36: Slabs allocated with a buddy allocator.

Lecture 31, Apr 9, 2024

Virtual Machines

- The goal of a virtual machine is to be able to run multiple OSes on the same system; to each OS, it appears as if it is the only one running
  - The *host* is the machine that the OSes are running on, which will have its own OS
    - A *guest* OS sees its own virtual copy of the host
  - The VM is isolated from the host for security
- The *hypervisor* or *virtual machine manager* (VMM) controls virtual machines, including creation, management and isolation
  - Type 1: *bare metal hypervisor*, which runs directly on the host hardware, with special hardware support
    - Kernel will run in kernel mode, so the hypervisor has even higher privileges than kernel mode
  - Type 2: *hosted hypervisor*, which runs as a normal process on the host's OS and simulates a hardware hypervisor
    - Slower but no need for specialized hardware
- Note VMs are not the same as *emulation* (which typically translates one ISA to another); the guest executes instructions directly using the same ISA as the underlying hardware

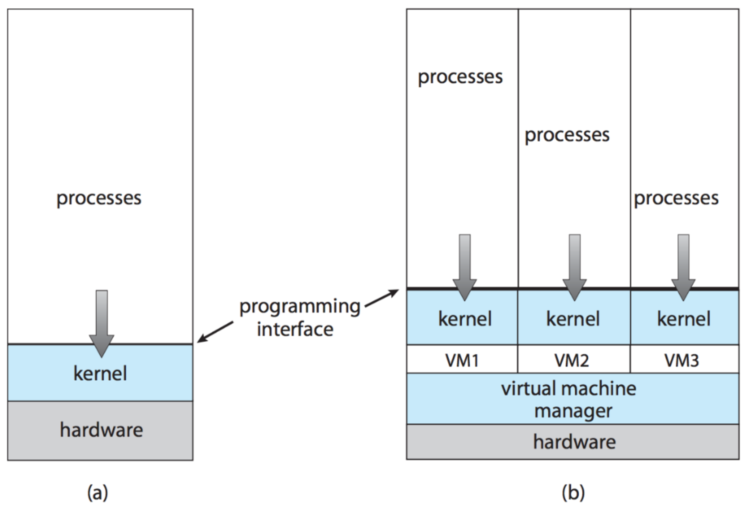


Figure 37: Structure of a machine: (a) typical machine, (b) one machine running 3 kernels through VMs.

  - A VM could use emulation, but this introduces heavy performance penalties
- VMs enable pause and play; just like the kernel can pause a process, a hypervisor can pause an entire OS
  - To enable this, the hypervisor does context switching between virtual machines
  - The guest can be moved between machines without its knowledge just like a process
  - This can be useful in e.g. cloud compute services
- Each guest is isolated from each other and the host
  - The hypervisor can set limits on CPU, memory, network bandwidth, disk space, etc
  - Guests can only access its own virtual hardware, which is useful for experimentation
- VMs can help consolidation
  - In a datacenter, there are often many servers that don't make use of all of their resources (e.g. one is using a lot of CPU and one is using a lot of memory)
  - Using VMs we can have different servers share the same hardware to be more efficient
- A *virtual CPU* (VCPU) saves all of the state of the entire CPU, allowing the hypervisor to pause and context switch guests
  - This is similar to a PCB, but a PCB only saves enough data for a user-mode process
  - When a VM is resumed, it loads the VCPU info and resumes the guest
- Each guest still uses user and kernel modes, without any change to their code
  - The kernel can still use privileged instructions
  - For type 1 hypervisors, the CPU's hypervisor mode is used to enter a privilege level higher than kernel mode
  - For type 2 hypervisors, the host/hypervisor needs to create a virtual kernel and user mode and emulate/simulate the hardware

- Type 2 hypervisors need extra strategies to simulate kernel mode instructions that the guest is attempting to run
  - *Trap-and-emulate* is the strategy of running the guest in user mode, and trapping any instructions that can only execute in kernel mode
    * The errors are caught and explicitly handled by the hypervisor to emulate/simulate the operation
    * The VCPU state is updated according to the instruction, and then we return to the guest
    * This significantly slows down these instructions
  - Trap-and-emulate doesn't always work; there are instructions that can be both kernel mode and user mode
    * e.g. on x86, the `popf` instruction loads flags from the stack, which is different if the instruction

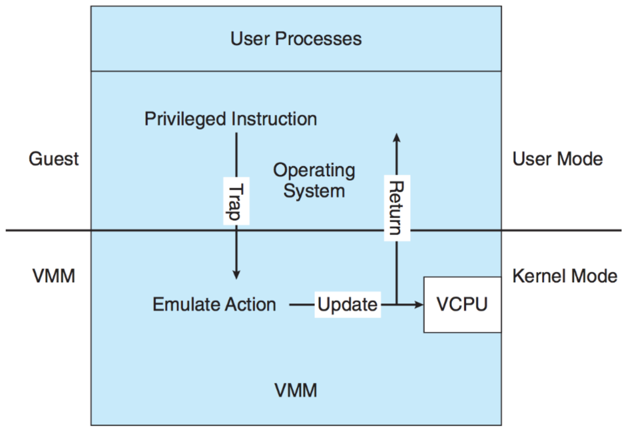


Figure 38: Illustration of trap-and-emulate.

is executing in kernel mode vs. user mode
    * Such instructions would not generate a trap and would just always behave as if it were in user mode
  - These special instructions need *binary translation*; when the VCPU is in kernel mode (according to the guest), then hypervisor inspects every instruction before execution, and handles any special instructions
    * We trap the user to kernel mode switch instruction
    * When the guest is in kernel mode, all instructions can be run natively as normal
    * Overall performance suffers a lot, but it works fine most of the time
    * This is how Valgrind works

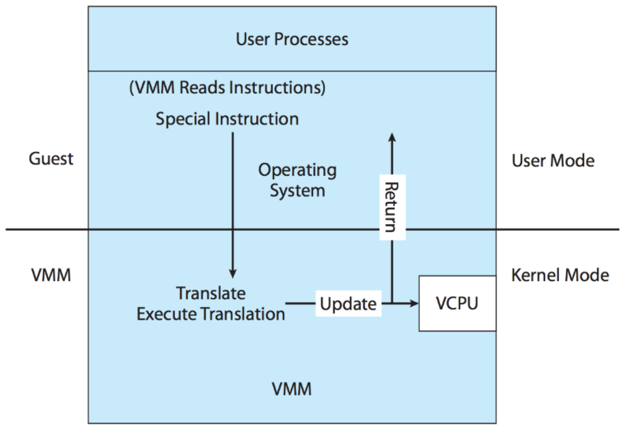


Figure 39: Illustration of binary translation.

- Intel and AMD both introduced virtualization as a standard in CPUs in the mid-2000s (VT-x/VMX and AMD-V/SVM)
  - This adds the concept of "ring -1", which is the hardware hypervisor mode
  - The host OS kernel claims the hypervisor, allowing it to manage the isolation for guests
- A hypervisor needs to perform scheduling between CPUs on the host machine
  - Virtual CPUs in the guest are mapped to physical CPUs; the number of CPUs may not be the same
  - The simplest approach is CPU assignment, which maps VCPUs to physical cores one-to-one, with the host using spare physical cores
    * This only works if there are more physical cores than VCPUs

  - If there are more VCPUs than host CPUs, we need to use a scheduling algorithm like for processes; this is called *overcommitting*
    * Overcommitting leads to very bad performance for soft real-time tasks on the guest
      - Processes may be context switched out even when the guest is not trying to do so
    * In this case virtualization has a different observable behaviour
- The hypervisor also needs to manage virtual memory between VMs
  - The hypervisor translates the guest's page tables to the real physical page table
    * Each guest kernel is still trying to do its own page management
    * This leads to a nested page table
    * If there is hardware support, the MMU can use the nested page tables and do both steps of the translation at once, avoiding the slowdown
  - Memory can also be overcommitted
    * The hypervisor can have a swap space, leading to double paging
    * However the hypervisor doesn't have a good idea of the guest's memory access patterns, so page replacement is usually left to the guest
- Guests could share pages if they are duplicates, like copy-on-write
  - The hypervisor detects duplicates by hashing the page contents; when hashes are the same, check each byte individually to confirm
- The hypervisor also provides virtualised I/O, e.g. network interfaces
  - One physical device can be multiplexed to many virtual devices
  - The hypervisor can also emulate nonexistent devices
  - Direct mapping from physical devices to virtual devices can also be used to give the VM exclusive access to the device
    * The hypervisor still does translation in the middle
    * IOMMU is a new hardware solution that removes the hypervisor in these cases to improve performance
- VMs boot from a virtualised disk, which is specified using a disk image
  - The images contain partitions and filesystems within the partitions similar to a physical disk
  - The disk image is often one big file; some formats allow for splitting
  - The disk image is easily moved to move the VM
- VMs can be used to isolate an application; the app is packaged with all its dependencies into the VM image, so ABI changes on the host won't affect it
  - However we pay the cost of VM overhead
  - For smaller apps, the kernel itself likely takes up a lot of the VM
  - *Containers* (e.g. Docker) are lighter-weight alternatives that use mechanisms on the host OS, e.g. control groups (`cgroups`) on Linux for process isolation
    * Unlike a VM, the contained application shares the same kernel as the host
    * Processes inside the container are still isolated, and its resources can still be limited