CS6200 Final Exam Review

George Kudrayvtsev

Fall 2018

A Word of Warning to Ye Who Enter
I'm just a former student. I *think* most of the content in this guide is reasonably correct, but please don't treat these review answers as gospel. And, if you ask me something about it, I probably won't even be able to answer you because I've forgotten 80% of this stuff.

Enter at your own peril! *(though I would appreciate corrections)*

Part 3 – Lesson 1: Scheduling

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3399758762/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1CR-2icdEH4rNG5xRIdHzqBx7QQ9T4oHQlxKG-ZRjWXQ

Metric Mania

Warning: I'm pretty sure this whole box confused everyone that read it but me. The purpose of the formula "derivations" was to try to get a better understanding of each metric by generalizing it. You won't need the formulas, but you should definitely understand what each of the metrics entails!

When discussing scheduling, there are a number of important metrics to keep in mind to effectively compare scheduling algorithms. For all of them, we define T_i as being the completion time of task i, where the is are ascending values based on their *queueing* order (not *execution* order, as determined by the scheduler). This value is either known prior to execution (in an ideal world) or guesstimated based on some heuristics (in the real world). We also define n to be the number of tasks used for the metric.

The first three metrics assume in their formulas that all tasks are submitted at the same time. Such a simplistic scenario is unlikely to appear on the exam, and so it is likely better to understand how these formulas were *derived* rather than to simply memorize them. Understanding their derivation will synthesize the information better for a final-worthy question that may involve more complex scenarios involving priorities, preemption, and varied task arrival times. Such scenarios are hard to generalize, and are thus much better done by hand.

- **Throughput** (τ) – We define this as the number of tasks that can be completed during a selected period of time. To keep things simple, the "selected period of time" should just be the total time to execute all of the tasks in a given scenario.

  For both FCFS and SJF policies,

$$\tau = \frac{n}{\sum_{i=1}^{n} T_i}$$

- **Avg. Job Completion Time** (c_{avg}) – This is defined as the average time it takes for a task to go from "queued" to "completed."

  For a FCFS policy,

  $$c_{\text{avg}} = \frac{nT_i + (n-1)T_{i+1} + \ldots + T_n}{n} = \frac{1}{n}\sum_{i=1}^{n}(n-i+1)T_i$$

  For a SJF policy, we first define C_j as being the *sorted* order of the tasks based on their expected completion time (ascending). So C_1 is the shortest task, and so on. Then, the formula is identical to FCFS, except with C_j replacing T_i.

- **Avg. Job Wait Time** (w_{avg}) – This is defined as the average time it takes from a task to move from a "queued" state to an "executing" state.

  For a FCFS policy,

  $$w_{\text{avg}} = \frac{(n-1)T_1 + (n-2)T_2 + \ldots + T_{n-1}}{n} = \frac{1}{n}\sum_{i=1}^{n-1}(n-i)T_i$$

  For a SJF policy, we again define C_j as outlined for c_{avg} and the formula again mirrors FCFS but with C_j instead of T_i.

- **CPU Utilization** (μ) – This is defined as the percentage of time that the CPU is using to do "useful work" (i.e. not context switches). In general,

  $$\mu = \frac{t_{\text{useful}}}{t_{\text{useful}} + t_{\text{overhead}}} \cdot 100$$

  The times should be calculated over a consistent, recurring interval. It's useful to use an interval that is related to the given timeslice, then figure out how much overhead occurs during that timeslice.

How does scheduling work? What are the basic steps and data structures involved in scheduling a thread on the CPU?

I think of scheduling as "the thing that happens between the other things." Existing as a kernel-level task, the scheduling algorithm executes periodically and is triggered for a number of reasons, including:

- **Idling** — when the CPU becomes idle, the scheduler runs to determine the next task to execute, in order to avoid wasting resources.
- **New Tasks** — when a new task arrives on the ready queue, the scheduler may need to determine what to do with the task. If the scheduler respects task priority, for example, it may choose to interrupt the current task (known as **preemption**) and prioritize the new one.
- **Timeslice Expiration** — to ensure fairness, tasks are usually assigned a specific time limit in which they are allowed exclusive access on the CPU. Once the timeslice expires, the scheduler needs to determine who is allotted the next timeslice.

The decisions made in each of the above scenarios depends on the **scheduling policy**; different policies enable different types of decisions and require different types of implementations. In the abstract, the data structure used for scheduling is known as the **runqueue**, and it's tightly coupled to a policy to enable specific decision-making.

What are the overheads associated with scheduling? Do you understand the tradeoffs associated with the frequency of preemption and scheduling/what types of workloads benefit from frequent vs. infrequent intervention of the scheduler (i.e. short vs. long time-slices)?

One of the obvious overheads of scheduling is the execution of the scheduling algorithm itself. Just like it's only beneficial to context switch between tasks when:

$$2t_{\text{context switch}} < t_{\text{blocking task}}$$

Similarly, it's only beneficial to run the scheduler if it takes less time than the amount of time it would take to finish the current task. To discretize the overhead further, there's the overhead of the interruption, the overhead of the algorithm, and the overhead of the resulting context switch. It's even possible for the scheduler to take control even when there's nothing more to do – for example, a task's timeslice expired, but it's the only task.

Longer timeslices are better for CPU-bound tasks. By their nature, metrics like average wait time are irrelevant because the task's goal is simply to do useful work. CPU-bound tasks don't require user interactivity and can always make the most of their time on the CPU.

Shorter timeslices are better for I/O-bound tasks. I/O-heavy tasks will implicitly create their own schedule by yielding during I/O operations. Additionally, shorter timeslices allow them opportunity to interact with the user, respond to incoming requests, etc. which are essential for perceiving "interactivity." The sooner an I/O-bound task can issue an I/O request, the better; it's less idle time for that task and better for the other tasks, who also want CPU time. Shorter timeslices allow for a better distribution of utilization – both devices and the CPU can stay highly in-use.

Can you compute the various metrics described at the top of this section given a scenario describing some workload (i.e. given a few threads with their compute and I/O phases) and its scheduling policy?

Consider a scenario with the following workload:

- 3 tasks – T_1 is I/O-bound but needs 3ms of CPU time; T_2 and T_3 are CPU-bound and need 4ms and 5ms of CPU time, respectively.
- T_1 issues an I/O request after 1ms of CPU time.
- I/O requests take 5ms to complete.
- The scheduling algorithm runs in 0.2ms, and a context switch costs 0.3ms.
- The scheduling policy is round robin with 2ms timeslices.
- Tasks arrive in the order 1, 2, 3.

The resulting timeslice graph looks like the following:

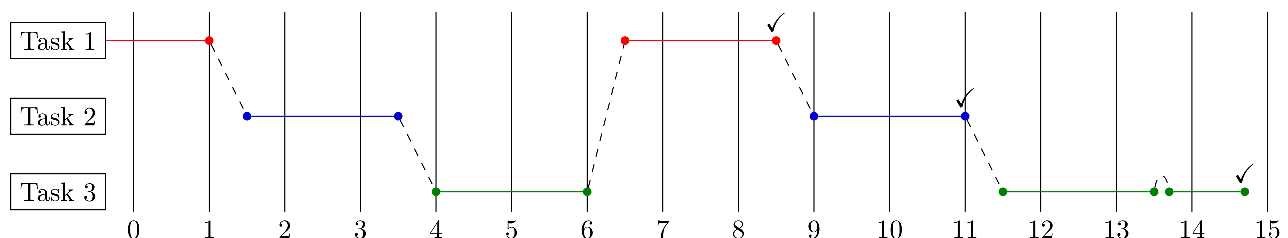


Figure 1: The timeslice graph for tasks T_1, T_2, and T_3. Note the assumption that the scheduler is still invoked if there are no other tasks. This may not be true – I'm planning on asking a question about this on Piazza/Slack.

Throughput 3 tasks completed in 14.7 seconds means $\tau = 3/14.7 \approx 0.204$ tasks/s.

Avg. Completion The completion times are $(8.5, 11, 14.7)$ for the tasks, in seconds. Thus, the average completion time is:

$$c_{\text{avg}} = \frac{8.5 + 11 + 14.7}{3} = \frac{34.2}{3} = 11.4\,\text{s}$$

Avg. Wait The wait times are $(0, 1.5, 4)$ for the tasks, in seconds. Thus, the average wait time is simply:

$$w_{\text{avg}} = \frac{1.5 + 4}{3} = 1.8\overline{33}\,\text{s}$$

CPU Utilization We first choose the interval of, well, the entire running time of the tasks. It'll give us the most accurate utilization. Over the course of 15 seconds, we had 5 context switches and 6 algorithm invocations. That means,

$$\mu = \frac{14.7 - (5 \times 0.5 + 0.2)}{14.7} \approx 0.816$$

Thus, we had about 82% CPU utilization for these tasks. Not ideal, but the overhead is an absurd amount of time, so... *shrugs*.

> Do you understand the motivation behind the multi-level feedback queue? Specifically, why different queues have different timeslices and how threads move between these queues? Can you contrast this with the $O(1)$ scheduler? Do you understand what were the problems with the $O(1)$ scheduler which led to the CFS?

The **multi-level feedback queue** (or MLFQ), was designed to dynamically balance a task's timeslice during its runtime based on the history of the task and other heuristics. Each level of the MLFQ contained increasingly longer timeslice values. If a task yields voluntarily during its given timeslice, it belongs at that timeslice level. However, if it uses the entire timeslice, it is considered to be more CPU-bound than I/O-bound. Hence, it's moved into a lower level of the queue. Likewise, if the scheduler notices that a task is yielding often for its timeslice – an indication of an I/O-bound task – it can push the task back up the queue into a shorter timeslice level.

> **Theory in Action**
>
> The Solaris scheduler uses a ***60-level*** MLFQ, along with some fancy rules calculating the feedback that moves tasks between levels.

Linux $O(1)$ Scheduler This scheduler takes constant time to perform both task selection and insertion. It borrows concepts from the MLFQ but has a different approach to incorporating feedback. The priority of a task – ranging from 0 (high) to 139 (low), where priorities $\in [0, 99]$ are "real-time" and kernel-level tasks – is proportional to its timeslice (priority $\uparrow \Rightarrow$ timeslice $\uparrow$), and the priority is adjusted to be inversely proportional to the task's idle time. In other words, more idle time indicates more I/O operations, and so the task's priority is raised. Similarly, less idle time indicates more CPU usage, and so priority is lowered (by 5, in both cases). Please note the word "priority" in this paragraph means the *logical* priority (as in, "this is a high-priority task"), not the priority *value*. A high-priority task would have a low priority value, which can be very confusing (especially on the sample final lmao).

Notice that this is the *opposite* of what we determined was the "ideal" scheduling for both I/O- and CPU-bound tasks. This is intentional to enforce fairness: an I/O-bound task will not use its entire timeslice and will yield naturally, and a CPU-bound task should be interrupted frequently to ensure it doesn't hog resources (since it gladly will).

Implementation The runqueue for this scheduler used two parallel arrays of lists of tasks (i.e. something like `struct task_list[140]`). One of these was the **active array** whereas the other was the **expired array**; in both, the array index corresponded to the priority.

Tasks remain in the active array until their timeslice expires; voluntarily yielding means the task stays "active." On expiration, the task is moved to the expired array. When there are no more tasks in the active array, it's swapped with the expired array. Note again that higher priority tasks (which are often I/O-bound) have higher timeslices; this means that they will keep getting scheduled if they yield appropriately.

Achieving $O(1)$

Insertion is trivially $O(1)$: it requires indexing into an array and making the "end" pointer of the list point to the new task. **Selection** is achieved by using a hardware instruction to find the first set bit in a bitmap corresponding to the array in which a set bit indicates a non-empty index.

Problems Workloads changed to become more interactive; things like video chat, gaming, etc. meant that the jitter caused by long time slices became unacceptable. Even though I/O-bound tasks would naturally get a higher priority, moving to the expired array meant it may be a while before a task ever got another chance to run. These significant delays grew too noticeable for users. The $O(1)$ scheduler didn't make any guarantees or claims for fairness: task execution time was not really proportional to priority.

Linux CFS Scheduler This scheduler – the "completely fair scheduler" – replaced $O(1)$ as workloads changed. Its runqueue uses a **red-black tree**, which is self-balancing and tries to ensure that the tree's depth is similar across all branches.[more] The tree is ordered by "virtual runtime," or **vruntime**. This is proportional to the time spent by a task on the CPU (with nanosecond granularity) *and* its priority. In other words, vruntime is adjusted based on the actual time spent on the CPU, then *scaled* by the task's priority and "niceness" value.[1]

Implementation In any given branch, the left node has had less vruntime than the right node. Hence, the left-most node in the tree is always picked, as it has had the least amount of time on the CPU. Vruntime increases faster for low-priority tasks, moving them to the right faster, and increases slower for high-priority tasks, allowing them more time on the left side of a branch. The same tree is used for *all* priorities; the priority is implicitly included in the balancing of the tree via vruntime.

Algorithmic Complexity Selecting a task is still $O(1)$, but insertion is now $O(\log_2 N)$, where N is the total number of tasks in the runqueue.

> In Fedorova's paper on scheduling for chip multiprocessors, what's the goal of the scheduler she's arguing for? What are some performance counters that can be useful in identifying the workload properties (compute vs. memory bound) and the ability of the scheduler to maximize the system throughput?

Fedorova argues that the ideal scheduler would use information about its pending workloads to make scheduling decisions. She identifies the theoretical metric **CPI**, or **c**ycles **p**er **i**nstruction, as a way to measure the "average instruction delay." A device-heavy workload would spend a lot of ticks waiting on an instruction to complete, whereas a CPU-heavy workload could complete faster.

Unfortunately, CPI is not only a theoretical, but unreliable metric for most applications. Applications are often not heavily skewed towards being either CPU-bound or I/O-bound, meaning CPI does not vary enough to make reliable scheduling decisions.

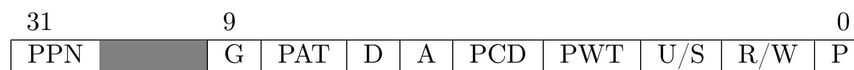
[1] Recall: A task's "nice" value adjusts its priority from the default.

Part 3 – Lesson 2: Memory Management

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3399758763/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1utxw0Ou8o7iRqwexJRlGm9P1t60Eqc4alHX8iHuyZTA

Intel PTE

The following is a page table entry on the MMU in modern Intel processors.



The labels correspond to the following meanings:

- **D** — The dirty bit, indicating whether or not the page has been written to since it was loaded. This is useful for optimizing whether or not the page needs to be flushed to disk when it's replaced.
- **A** — The accessed bit, indicating whether or not the page table has been accessed during some period of time, typically used in page replacement algorithms.
- **PCD** — Indicates whether or not **p**age **c**aching is **d**isabled.
- **PWT** — Toggles **p**age **w**rite-**t**hrough.
- **U/S** – Indicates the privilege level of the page, either **u**ser-mode or **s**ystem-mode.
- **R/W** – Controls the permissions on the page: 0 indicates it's a read-only page, whereas 1 indicates both read and write permissions.
- **P** — The present bit, akin to a "valid" bit.

How does the OS map the memory allocated to a process to the underlying physical memory? What happens when a process tries to access a page not present in physical memory? What happens when a process tries to access a page that hasn't been allocated to it? What happens when a process tries to modify a page that's write protected? How does COW work?

In order to create an illusion of isolation for processes, in which they perceive that they have access to all (or more!) of the system's physical memory, computers use **page tables**. Processes are assigned a **virtual address space**; this is associated with physical memory as it's referenced. Page tables are the data structure that performs the association and translation between virtual and physical addresses (or between nested levels of page tables, known as **multi-level page tables**). They require *hardware support* and are outside of the scope of the kernel – modern architectures have a **memory management unit** (or MMU) responsible for these tasks.

When the process accesses memory, the virtual address is translated to a physical memory location by the MMU. There are a variety of things that can occur as a result of an *unsuccessful* translation:

Not Present If the translation for requested address is present in the page table but does not translate to an existing physical memory address, that data (the entire "page") has been moved elsewhere (like to

the disk). Thus, the *memory* read results in a *disk* read, placing the page back into physical memory and updating the page table accordingly.

Not Allocated There are two forms of unallocated address accesses: intended and unintended.

An intended unallocated access would be something like the process stack growing beyond a page boundary. The stack needs another physical page to keep growing. This kind of access would likely be interpreted as "valid" by the kernel, allowing it to create a new page table entry and associate the new page with physical memory, replacing some outdated page if necessary. This is known in lecture as "allocation on first touch."

An *un*intended unallocated access would be something like dereferencing an invalid pointer. This is another type of page fault that would instead result in a segmentation fault (`SIGSEGV`) from the user process' perspective.

Not Allowed If a particular type of access is forbidden – such as a write instruction to a read-only page (like the static, immutable `.data` section of a program) – the MMU will issue a page fault to the operating system. This type of protection is critical for certain exploitation prevention techniques such as W⊕X (write-xor-execute), which mark memory as *either* writable *or* executable, preventing things like buffer overflows that use the stack to store executable code.

Copy-on-Write Page tables allow us to include an optimization when a process is cloned. Since the clone will contain much of the same data as its parent, there's no need to copy it to a different location in physical memory. In other words, the two different virtual address spaces can refer to the same physical pages. That is, until the clone writes some new data to the page. Then, and *only* then, do we copy the physical page to make sure the parent's memory isn't modified (achieved by marking the page as write-protected for the clone). Other pages remain uncopied, such as those containing the process' instructions. This is known as **copy-on-write**, or COW.

> How do we deal with the fact that processes address more memory than physically available? What's demand paging? How does page replacement work?

Because addresses are virtualized, we can address an arbitrary amount of memory. Any pages that don't fit into physical memory can still be stored on disk and are "paged in" or "paged out" as needed. A page table exists *per process* (this is an important part of a context switch) and has enough entries to map to all *possible* memory. Need for entries beyond physical memory is managed by the operating system: it tracks extra memory being stored on disk, etc. When a page that's not in physical memory is requested, the MMU will trap into the kernel since an invalid page has been requested. The kernel will then look up and internally manage where it stored the page on disk, placing it into physical memory and updating the page table accordingly. This process of swapping out pages to disk to simulate a larger address space than is physically available is known as **demand paging**.

Decisions about which page to replace depend on the page replacement policy. One such policy is the **least-recently used** policy (or LRU); this policy will evict the oldest unaccessed page in the page table (via the **access bit**) to make room for a new entry from disk. Replacements can also use the **dirty bit**: if a page hasn't been modified, it doesn't need to be written out somewhere to be replaced.

> How does address translation work? What's the role of the TLB?

Address translation works as follows:

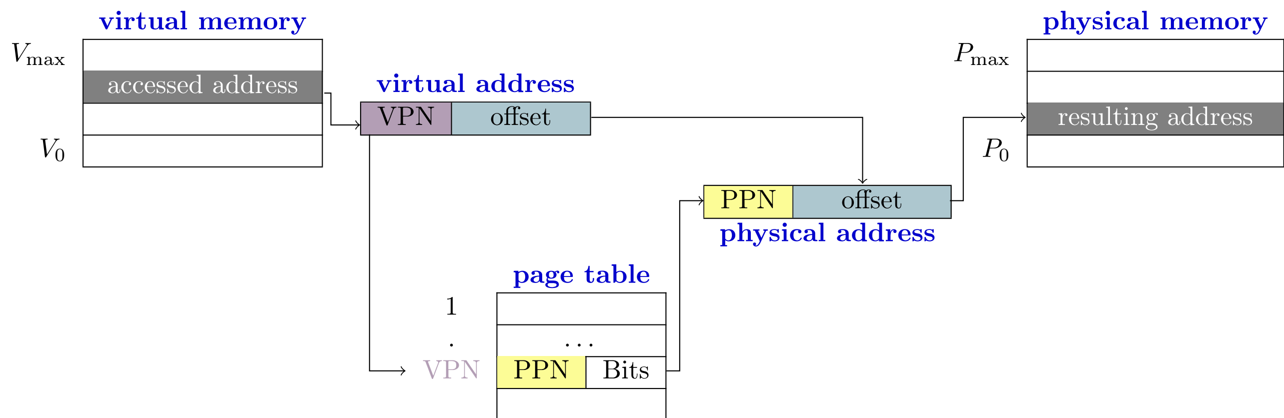


Figure 2: A demonstration of translation from a virtual address to a physical memory location. Keep in mind that this diagram doesn't include TLB lookups, use hierarchical page tables, or handle what happens on a missed entry.

To explain further, a single virtual address is discretized into one or more virtual page numbers (VPNs) that are used as the *index* into a page table. The resulting page table entry at that index correlates the VPN to a physical page number (PPN); this is appended to the rest of the virtual address (called the offset) as-is. The combined result is the physical memory location.

TLB The **translation lookaside buffer** is essentially a cache of recently-used page table entries. It allows translations to occur even faster: since the page tables themselves must be stored in memory, they take more time to access relative to an on-chip, domain-specific cache.

> Do you understand the relationships between the size of an address, the size of the address space, the size of a page, the size of the page table...

Let's consider a sample scenario of a 24-bit architecture that has 1MiB of physical memory. This corresponds to the following page table values:

- **Address Size** — The address size – both virtual and physical – simply follows the bus width specified by the architecture. Theoretically, we should be able to convert *any* sized of virtual address to a physical address, but because it's a hardware-provided interface, we must rely on how things work in practice. Thus, the address size is 24 bits long.
- **Address Space** — The address space is directly correlated to the hardware bus width. Despite only having 1MiB of physical memory, the address space is still contains 2^{24} addresses. Anything beyond 1MiB will be marked as being paged out to disk, but must still have an entry in the page table! In other words, any description about the amount of physical memory available is a red herring when it comes to page table calculations.
- **Page Size** — This is a system-specific, often-customizable value. For the purposes of demonstration, we will assume say these are 4KiB ($4 \cdot 2^{10} = 2^{12}$).

With all of these in mind, let's calculate the size of a single page table. Once again, we need to be able to address 2^{24} addresses, discretized into 2^{12}-sized chunks. This means:

$$\frac{2^{24}}{2^{12}} = 2^{12} = 4096 \text{ page table entries}$$

This can also be calculated by looking at the remaining bits. A 4KiB page means 12 bits are used for the page, leaving $24 - 12 = 12$ for the index, so 2^{12} values.

Do you understand the benefits of hierarchical page tables? For a given address format, can you workout the sizes of the page table structures in different layers?

Page tables are a fixed size; they contain enough entries to map to the entire address space, regardless of whether or not there's enough physical memory. Recall, also, that page tables a per-*process* data structure. A process likely will not use much of its available virtual address space; this means there are many empty entries in the page table. By splitting a virtual address to index into multiple levels of page tables, we can reduce the size of these gaps and wasted PTEs.

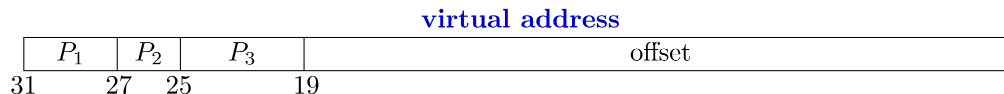
This optimization becomes essential when dealing with larger address spaces. A 64-bit architecture has too many addressable memory locations to store all of PTEs. Hence it becomes necessary to divide it into levels.

Theory in Action

Modern Intel systems have a five-level page table scheme, capable of addressing 128PiB (peta!) of memory, given 4KiB pages. It allows us to (theoretically) have 57 addressable bits.

Sample Calculation

Suppose we have a hierarchical page table for a 32-bit architecture with 3 levels: level 1 is 4 bits, level 2 is 2 bits, and level 3 is 6 bits. The first two levels represents indices into the subsequent, deeper level, whereas the last level behaves like an index into a page table as before. This results in a virtual address chunking like so:



Thus, P_1 indexes into the first table, which has $2^4 = 16$ entries. P_2 indexes into the second table, which has $2^2 = 4$ entries. Finally, P_3 indexes into the third table and combines with the offset just like in a single-level table (see Figure 2), having $2^6 = 64$ entries.

How does this impact the address space? Well, a 20-bit offset means 1MiB pages. Thus, a single inner-most page table can address 26 bits of memory ($2^6 \cdot 2^{20} = 64$MiB). The second layer contains 4 entries, and so can address 28 bits of memory (256MiB). In other words, a *single entry* in the third layer addresses 28 bits of memory. There are 16 such layers, and thus the entire schema can address the full 32 bits of memory.

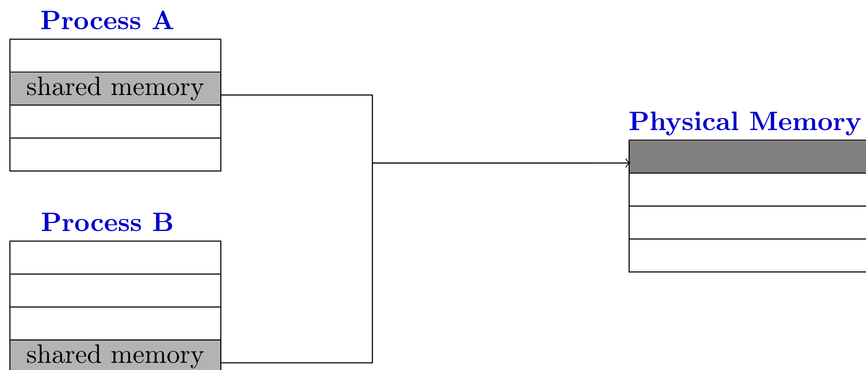
What's the benefit over a single page table? Suppose a program needs 2MiB of memory, and hence two page table entries. With a single page table, there are $2^{12} = 4096$ entries created immediately, with just two slots occupied. With the hierarchical page table, we create the top-level table (16 entries), a single second-level table (4 entries), and the full third-level table (64 entries), for a total of 84 entries. Even if the two pages are not in the same level, that's still only $84 \cdot 2 = 168$ entries. This is a huge savings in per-process memory requirements, especially with low memory usage.

Part 3 – Lesson 3: Interprocess Communication

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3397338682/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1KfdgAPTA6yumWB-OJQ-fikVu_wbDQbDKw10lSd9x67k

For processes to share memory, what does the OS need to do? Do they use the same virtual addresses to access the same memory?

For processes to share memory, they need to ask the kernel to create a shared memory segment. This is a special piece of memory in which both processes map their unique virtual address space to the *same* physical memory location:



This association requires some expensive initial setup on behalf of the kernel: the OS must create (and track) the virtual address mapping and allow it to be used by multiple processes. After that, though, the sharing happens exclusively in userspace. This is a big advantage over other IPC methods, such as message queues or sockets, which rely on the kernel to deliver every message.

For processes to communicate using a shared memory-based communication channel, do they still have to copy data from one location to another? What are the costs associated with copying vs. (re)mapping? What are the tradeoffs between message-based vs. shared-memory-based communication?

When using shared memory, changes to that memory in one process are reflected in the other instantly and transparently. Of course, that means that from the perspective of one process, its view of the shared memory could change at any time. Hence, if a process needs to perform some calculations based on data in shared memory that are at a risk of being overwritten, it must copy them to its local memory. So there *may* still be copying involved, but this isn't universally true; with a good design and proper synchronization, the amount of copying from unshared to shared memory can be greatly reduced relative to the other, kernel-dependent IPC mechanisms.

Trade-Offs Message-based IPC has the advantage of using a standard, kernel-provided API that can make guarantees about thread-safety, message arrival, etc. All of these have to be re-implemented by the programmer if desired with shared memory IPC. On the downside, though, such convenience comes with the cost of performance. As mentioned previously, message-based IPC requires a lot of expensive hops from user mode to kernel mode. Similarly, though, the initial setup of a shared memory region requires some expensive time in the kernel. If the cost of such setup can be amortized over a long period of *usage* of the shared memory (where it shines), it can outperform message-based APIs, at the cost of a more complex (and situational) implementation.

What are different ways you can implement synchronization between different processes?

There are a number of different IPC mechanisms, each of which are conducive to different synchronization mechanisms. Let's talk about the different IPC mechanisms briefly, aside from the already-discussed shared memory and message queue mechanisms.

Using **pipes** connects processes as outlined in Figure 3.[2] Using **sockets** allows you to get intermachine communication as well for free, provided you serialize things properly if you are anticipating communication between different architectures. Both of these mechanisms rely on the kernel to transfer data between processes, but are different than message queues in that they are more "stream-based" rather than "message-based."

Figure 3: A succint explanation of pipes.

For synchronization between processes, the kernel-based methods lend themselves to using the **named semaphores**, which are essentially semaphores represented as files. By using the same name in both processes, they can synchronize actions as if they were synchronizing via semaphores in their own memory. Shared memory is more diverse: though named semaphores are perfectly viable, both **unnamed semaphores** and **inter-process `pthread` primitives** can be used instead. These are placed in the shared memory with a special flag and initialized by one end. Then, they can be used exactly the same way as if they were in regular memory within a single process. When synchronizing across machines in the socket-based approach, it likely makes sense to use some form of an event-driven architecture like we discussed in previous lessons, though thankfully we didn't need to explore this much for the project.

Part 3 – Lesson 4: Synchronization Constructs

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3373318994/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1SavGKDc7qVYDFEy1wpjqyWCJ8uDf4AVzZ4TzQD3y_gc

To implement a synchronization mechanism, at the lowest level you need to rely on a hardware atomic instruction. Why? What are some examples?

Synchronizing involves two operations: a read of some shared state, and a write to that shared state by one (and only one!) of the processes. Without these operations *both* occurring at the same time, it's possible that the write in one interleaves with the read in another, and thus more than one thread will think it has exclusive access to the critical section. Some examples of atomic instructions include:

- **Test & Set** — This instruction is equivalent to the following code, executed atomically:

```
int test_and_set(int* value) {
  int old = *value;
  *value = BUSY;
  return old;
}
```

[2] Disclaimer: I didn't use or even research pipes for the project. All of my knowledge comes via passive absorption of Ioan's comments on Slack.

- **Read & Increment** — This instruction is equivalent to the following code, executed atomically:

```
int read_and_inc(int* value) {
  int old = *value;
  ++(*value);
  return old;
}
```

  This construct is essential for the queueing lock described in the last question of this section.

Different hardware implements different atomic instructions; certain synchronization constructs that rely on particular atomic instructions are not necessarily platform-independent.

> Why are spinlocks useful? Would you use a spinlock in every place where you're currently using a mutex?

Spinlocks are useful for short critical sections: it's more efficient to simply wait for the critical section to be available than yield and perform a context switch to another thread. In other words, it's useful in situations in which our fateful equation:

$$2t_{\text{context switch}} < t_{\text{blocking task}}$$

does not hold. It's also useful when there are no other tasks, so yielding (and then immediately being run again) is essentially a costlier version than simply looping. For these specific scenarios, spinlocks are useful. They generally should not replace existing approaches to synchronization, such as those using mutexes.

> Do you understand why is it useful to have more powerful synchronization constructs, like reader-writer locks or monitors? What about them makes them more powerful than using spinlocks, or mutexes and condition variables?

Advanced synchronization constructs are beneficial because they assume some sort of usage pattern. The more generic the pattern, the less assumptions can be made about the way it will be used. Reader/writer locks, for example, even if they use semaphores under the hood, can choose better internal design decisions because they describe more specific usage semantics.

From the perspective of the programmer, higher-level constructs are easier to use and harder to configure incorrectly. When low-level requirements like locking and unlocking are implicit, it's harder to make these simple mistakes.

> Can you work through the evolution of the spinlock implementations described in the Anderson paper, from basic test-and-set to the queuing lock? Do you understand what issue with an earlier implementation is addressed with a subsequent spinlock implementation?

First, there was test-and-set, just simple spinning on the atomic instruction. Then, there was test-and-test-and-set, which made better use of the cached value, since test-and-set invalidates the cache even if the value doesn't change. Then, there were sleeps (both static and dynamic) introduced both before the memory reference and before the lock check (to allow contending threads to quiesce without invalidation). Finally, there was a queueing lock in which threads occupied a position in a line and each release of the lock would trigger the next thread to go; this method required a more complex atomic instruction, though (or the emulation of one).

Performance

Different types of spinlocks were better suited for different levels of contention and processor count. In general, this can be divided into two categories: high load (many contending processors) and low load (a handful of contending processors). The latter is a better model for modern x86 systems, which typically have 2-4 cores.

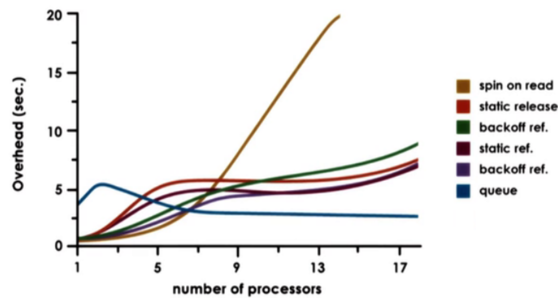


Figure 4: Performance comparison of Anderson's spinlock implementations.

High Loads Under high loads, the queueing lock was the best implementation: it scaled the best as processor count increased. Test-and-test-and-set performed the worst because it introduced the highest level of contention. Static sleeps performed better than dynamic sleeps in their respective categories, and sleeping after a memory reference out-performed sleeping on the lock's release since it avoided extra invalidations.

Low Loads Under low loads, the basic spin-on-read (a.k.a. test-and-test-and-set) lock performed the best, likely because it was simplest and the contention wasn't bad enough to expose its underlying limitations. It had the lowest latency. For this same reason, dynamic backoff out-performed static backoff, since the delay was lower between seeing the lock and acquiring it than it would've been in the static case. The queueing lock actually performed the worse because of its higher initial cost: the `read-and-increment` atomic function is pricier to execute and results in a high latency.

Part 3 – Lesson 5: I/O Management

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3450238824/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1kVn9PSmevfh5seCyp-Ix0CcRfGyB4PlVG5rhCW-r6jw

> What are the steps in sending a command to a device (say packet, or file block)? What are the steps in receiving something from a device? What are the basic differences in using programmed I/O vs. DMA support?

The interface for interacting with a class of hardware devices is typically defined by the operating system; this is the abstraction layer we discussed at the beginning of the class. The bridge between what the operating system expects and how the device actually provides that functionality is implemented by the **device driver**. Even still, there are differences in how a CPU processes events from devices.

Processing Device Events

Broadly, devices can be split into two categories: **polling-based devices** and **interrupt-based devices**. The communication mechanism we use should be decided based on the device properties and the goals;

things such as update frequency (how fast does a file get read vs. how often a user moves the mouse), performance goals (latency vs. throughput, etc.), data rate (i.e. input- or output-only) or and other device-specific considerations.

Polling This type of communication between the device and the CPU is best-used for devices that don't need to be instant. Polling occurs at the CPU's convenience, and thus incurs the cost of processing overhead and a delay between a device action and the processing of the action. Such a model is useful when the delay doesn't pose a problem to the overall responsiveness of a device. For example, screens often refresh at 60Hz (about every 13ms), and thus a mouse that incurs a tiny, microsecond delay between updates due to polling would be largely imperceptible.

Interrupt This type of communication is best when time is of the essence. Interrupts allow the CPU to handle the I/O event as soon as it occurs, since the event is generated as soon as possible. The obvious downside is it incurs the overhead of the interrupt handler, though this sort of delay is largely unavoidable.

Transferring Data to/from Devices

Interfaces to a devices are often implemented with a series of registers that are used for different actions. Typically, there are three types: status registers, command registers, and data registers. These simply appear as memory locations to the CPU, so writing to a command register (a memory store) would trigger a device action. There are two means of transferring the *data* to and from the device: **programmed I/O** or **direct memory access**. These essentially diverge based on whether the CPU accesses device memory or main memory.

PIO The benefit of programmed I/O, or PIO, is that it requires no additional hardware support: the CPU simply treats reads and writes to special registers like it would any other memory load/store operations. Using their own registers often comes with a hefty performance cost, though. For example, sending a 1500-byte packet on a NIC with a single 8-byte register would require 189 load/store instructions!

DMA Allowing the device to access main memory directly is a faster, albeit more complex, alternative. Direct memory access, or DMA, requires hardware support in the form of a "DMA controller." In this model, rather than accessing the device's data registers directly, the data transfer is managed by the controller. By using main memory directly, we avoid a series of fragmented transfers: the CPU simply tells the device where to find the data in memory. Continuing with the network packet example, the transmission request (which is still written to the command register by the CPU) includes information about where to find the packet in main memory. From the CPU's perspective, we now only need a single store instruction (for the command) and a single DMA configuration.

An important thing to keep in mind for DMA is that the main memory in question can no longer be paged out; it must persist as long as the device needs it to perform a data transfer.

> For block storage devices, do you understand the basic virtual filesystem stack, the purpose of the different entities? Do you understand the relationship between the various data structures (block sizes, addressing scheme, etc.) and the total size of the files or the filesystem that can be supported on a system?

A block storage device is one that operates on discrete blocks rather than characters, packets, etc. and is usable by a user via the operating system's **filesystem** abstraction layer. A "file" is an abstract OS concept that uses a filesystem to define a series of blocks on the actual device that make up the file's contents.

Operating systems introduce another layer between the typical user → kernel → driver → device flow: the **generic block layer** exists between the kernel and the device driver to create a uniform, OS-standardized "block interface" for the kernel to interact with devices.

The virtual filesystem, which defines the way that abstract "files" are represented on block devices, introduces concepts such as **file descriptors**, which are the OS representation of a file, **inodes**, which are persistent representation of a file, and **dentry**s (or **d**irectory **entry**) which temporarily represent components of a file path. A **directory** has no special meaning within a virtual filesystem; it is represented as yet another file, but it's contents are details about other files. This is in contrast with dentries, which are *not* persistent, but rather generated on-the-fly (and maintained in the **dentry cache**) as files are accessed on the system. There is also a **superblock**, which contains filesystem-specific information regarding its layout, such as free blocks, data blocks, etc.

mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect

Figure 5: inode structure

To avoid filesize limitations introduced by simple "list of indices" inodes, we introduce a more-complex inode structure that allows near-arbitrary filesizes via **indirect pointers**. As seen in Figure 5, the inode now contains additional metadata about the file, as well as a tiered structure of block pointers. The direct blocks, as in the simple model, are simply a list of blocks indices. The single-indirect blocks are blocks that lead to *another* inode that contains a list of block indices. This pattern repeats for the double-indirect and triple-indirect blocks, allowing for a *very* large maximum file size. Of course, as the level of indirection increases, performance degrades, since the device has to spend more time seeking across the disk to retrieve data.

Let's consider the math for a particular filesystem configuration. Suppose we have 1KiB blocks, a block index is 4 bytes long, and an inode stores 32 bytes of inode metadata. In the inode, we will track two single-indirect blocks and one double-indirect block. What is the maximum file size for such a filesystem? With 32 bytes of metadata and 12 bytes of indirect blocks, we have enough space in our inode block for:

$$\frac{2^{10} - (32 + 12)}{4} = 245$$

direct blocks, which can track $245 \cdot 2^{10} = 250,880$ bytes of data. Let's assume for simplicity that only the root inode needs to store metadata. Then, each single-indirect block leads to 256 direct blocks, allowing $2 \cdot (2^8 \cdot 2^{10}) = 2^{19} = 0.5$MiB more data. Finally, the double-indirect block contains 2^8 single-indirect blocks, each of which, as we previously calculated, allow indexing $2^{18} = 256$KiB bytes of data. Thus, in total, we can have files up to

$$250,880 + 2^{19} + \left(2^8 \cdot 2^{18}\right) = 67,884,032$$

bytes, or approximately **67.9 GiB**.

For the virtual file system stack, we mention several optimizations that can reduce the overheads associated with accessing the physical device. Do you understand how each of these optimizations changes how often we need to access the device?

Disk access is the slowest type of data access; we employ several techniques that optimize both the software- and hardware-specific aspects of disk access to avoid such significant latency:

Caching The operating system maintains a special **file buffer cache** in main memory. This cache stores parts of the file and maintains changes to those parts. The changes are periodically flushed via `fsync()`. This is why it can be dangerous to unplug a USB drive without "ejecting" it first; the data might still be in the cache! This cache is per-*file*, so all viewers of a file see changes simultaneously and instantaneously; these are known as **Unix semantics** when discussing update synchronization.

I/O Scheduling Because files are essentially scattered across a disk's blocks, performance can improve if we schedule reads such that the disk head moves across blocks in a more sequential order.

Prefetching When reading a block from disk, it's possible that the nearby blocks will also be useful! If we prefetch these blocks in advance just in case, we won't need to do another read later if they're needed.

Journaling Instead of writing out the data to the proper disk location, which will require a lot of random access, we write updates in a log called the **journal**. The log contains a description of the write that's supposed to take place, specifying the block, the offset, and the value, essentially describing an individual write. The journal is periodically flushed to disk and must be the most reliable part of the system in order not to lose data.

Latency Analogy

Suppose you're studying for the GIOS final and you have some flashcards laid out on your desk. If accessing data stored in a CPU register is akin to recalling a flashcard from memory, and accessing the cache is like grabbing a flashcard off of the desk, then accessing data stored on a traditional hard drive would be like *traveling to Mars* to get a flash card![source]

Part 3 – Lesson 6: Virtualization

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3614709322/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1wo-M2ehLTDCF63Mefxc8T0F39WJ91oe88MyAyrYh2pc

What is virtualization? What's the history behind it?

The meaning of **virtualization** has changed as technology has evolved. These days, virtualization is basically when one or more systems run in isolated environments on a single machine and all have access to the hardware, typically managed by some virtualization layer. Initially, though, there was a very strict definition that differentiated virtualization from things like the Java Virtual Machine.

The virtual machine must be an *efficient, isolated duplicate* of the host machine. The "duplicate" aspect is the key differentiator: running a Windows VM on a Linux host, for example, would not qualify as virtualization under this definition. Further still, running a Linux VM without networking on a Linux host *with* networking would be a violation of this requirement. The "efficient" aspect requires only a minor decrease in speed for the virtualized environment; this isn't *emulation*. Finally, the "isolated" aspect means that there's a management layer that has the "true" control over system resources and enforces isolation. All of these requirements are delivered and enforced by a **virtual machine monitor**, which is also called a **hypervisor**.

What's hosted vs. bare-metal virtualization? What's paravirtualization, why is it useful?

There are two primary forms of virtualization: **hosted** and **bare-metal** virtualization. The main differentiator between these is the location of the VMM in the stack. With hosted virtualization, the machine runs an actual operating system, with its own applications, kernel, etc. In this case, the VMM is actually an application running on said OS! This is in stark contrast with bare-metal virtualization, in which the VMM is the *only* thing running on the machine, aside from the virtualized environments it's supporting.

Both of these solutions assume, to an extent, that the guests are unmodified: the VMM takes care of any translations, permissions, architecture-specific quirks, etc. that may arise. The alternative approach is **paravirtualization**, in which the guest OS is *aware* that it's being virtualized. In this case, the OS makes

hypercalls – somewhat analogous to system calls under a non-virtualized environment – which are handled by the hypervisor to perform various actions.

> What were the problems with virtualizing x86? How did protection of x86 used to work, and how does it work now? How were the virtualization problems on x86 fixed?

An important requirement in virtualization is that the hypervisor needs to be aware of, and handle correctly, *all* of the privileged system calls. This relies on the fact that execution of a privileged instruction by a guest would trap into the VMM. A big problem when virtualizing x86 systems was that, well, this didn't always happen (for 17 instructions, specifically). For example, enabling and disabling interrupts would silently fail rather than trap into the kernel, meaning guests couldn't do this, and VMMs weren't even aware that they were trying!

Though these inconsistencies were eventually fixed in the architecture, the solution at the time was to actually translate such problematic instructions into ones that *would* trap into the VMM! This technique is known as **binary translation**.

From the hardware side of things, x86 – which is now obviously the most dominant architecture – added a lot of features to improve virtualization support:

- **Better Privileges** — Instead of the 4-ring approach that didn't really define clear purposes for each ring, there is now simply a "root," and "non-root" mode for instructions.
- **VM Control Structure** — Instructions that come from VMs are handled better; the hardware can actually inspect instructions and determine whether or not it should expect a trap. Furthermore, these "VCPUs" track metrics and are accessible by the hypervisor.
- **Page Tables** — Page tables are extended to perform better under virtualized environments. Furthermore, the TLB in the MMU now tags entries with a "virtual machine identifier," which allows it to cache translations better.
- **Devices** — There was support introduced for devices to be more aware of virtualized environments. For example, **multiqueue devices** have *logical* interfaces that can each interact with a VM separately. Interrupts are routed better, as well.

> In device virtualization, what is the passthrough vs. split-device model?

The massive variety in I/O devices means that we can't use a more uniform, standard approach to virtualization as we can with CPUs or memory. There are many models for achieving device virtualization.

Passthrough Model In the **passthrough** model for device virtualization, the VMM configures the device to allow exclusive and direct access to a particular guest. This is great from the guest's perspective, as it's the most performant and "greedy" approach. The obvious downside is that sharing is difficult. Furthermore, since this model relies on the guest to have the appropriate driver for the device, those have to match. Of course, by the formal, strict definition of virtualization we gave, this must be the case anyway. Another important downside that heavily impacts the usefulness of VMs is that **migration** is difficult: device state and in-progress requests must be carefully managed before a VM can move to another machine.

Hypervisor-Direct Model In this model, the guests don't interact with the hardware devices directly at all. The VMM intercepts all requests and emulates the requested operation. It converts the guest's request into a hardware-specific request that's then serviced by the VMM-resident device driver. As you can imagine, this adds significant overhead and latency to guest device operations, and heavily complicates

the VMM itself. It now needs this "emulation" layer and needs this driver ecosystem to interact with the hardware, rather than relying on the guest OS to know how to interact with it. The obvious benefit is decoupling between the guest and the hardware, allowing better management and migration of guests.

Split-Device Model This model is radically different than the previous two approaches. Device requests are split across two components, one of which is a new **service VM** that is separate from the VMM. The guest has a **front-end device driver**, which is a driver that is unaware of any real hardware and simply performs a wrapping or translation of requests into a format consumable by the **back-end device driver**. The back-end driver is similar to the normal device driver that would exist on a non-virtualized OS. This technique implies an important caveat: **it's limited to paravirtualized guests**. Though we've eliminated the "device emulation" of the previous model, and have a management layer (the service VM), we lose "pure" virtualization and complicate the requirements for guests.

Part 4 – Lesson 1: Remote Procedure Calls

Lecture video: `https://classroom.udacity.com/courses/ud923/lessons/3450238825/concepts/last-viewed`
Transcript: `https://docs.google.com/document/d/1LVTiSBgtUfeo8KlbqnwCXkTTe_JYbsbacF1JhfpNPU8`

> What's the motivation for RPC? What are the various design points that have to be sorted out in implementing an RPC runtime (e.g., binding process, failure semantics, interface specification. . .)? What are some of the options and associated tradeoffs?
>
> In the Sun RPC model, what is specifically done to address these design points?

The motivation behind RPC is to remove boilerplate, hide cross-machine interaction complexities, and decouple the communication protocol from the application logic; it defines a high-level interface for data movement and communication.

Interface Definition An **interface definition language** is an independent approach to defining the type of calls available on a server. It can be a language-agnostic definition file or specific to the implementation.

Registry & Binding A **registry** is a database of available services: it's used by clients to find servers that support the procedures they're looking for. It can be distributed (as in, "give me all RPC servers that provides an `add(int, int)` procedure"), or machine-specific; for the latter, clients need to know the specific server's address in advance (and it must define some port number).

Failures Because of the complexity of the system (cross-machine, network-dependent, etc.), there are plenty of situations in which a call might fail. It's sometimes impossible to determine the cause of such failures, and even a timeout/retry approach may never succeed. Because of this, an RPC implementation will define an error specification, defining all of the ways in which an RPC call can fail. Such a case results in an "error notification."

Sun RPC Design Choices

Sun's RPC implementation makes the following choices: a language-agnostic IDL (called **XDR** that uses `rpcgen` to convert `.x` files to language-specific implementations), a per-machine registry daemon that specifies

the service details (called **portmap**), and a retry-on-failure mechanism that makes a best-effort attempt to include as many details as possible.

> What's (un)marshalling? How does an RPC runtime serialize and deserialize complex variable size data structures? What's specifically done in Sun RPC/XDR?

To translate a local representation of a structure into a machine-independent one that can be transmitted over a network, a structure undergoes **marshalling**. This encodes a remote procedure call into a contiguous message buffer than contains all of its details (procedure name, its arguments, etc.) and prepares it to be transmitted. **Unmarshalling** is, obviously, the inverse of this process. This encoding process is typically automatically generated for a particular structure by the RPC toolchain from an IDL specification.

Part 4 – Lesson 2: Distributed File Systems

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3391619282/concepts/33765997180923
Transcript: https://docs.google.com/document/d/1ttWP1Z3yNtCxhGAV9GkBldcRB8xjxeg4zMiOIbXULlw

> What are some of the design options in implementing a distributed service? What are the tradeoffs associated with a stateless vs. stateful design? What are the tradeoffs associated with optimization techniques – such as caching, replication, and partitioning – in the implementation of a distributed service.

One of the most important design decisions in a distributed service is regarding **redundancy**: to what extent is state duplicated across machines? Does every (or every other, or two, or . . .) machine store all state? Though most designs incorporate both, there are two primary approaches: a **replicated** system, in which every server stores all data, and a **partitioned** system, in which each server stores part of the data.

> **Latency Analogy**
>
> If accessing the hard drive is like traveling to Mars for a flash card, then communicating with a remote server (without even considering what type of access *they'll* have to do!) is almost like going to ***Pluto*** in a modern space ship.[source 1],[source 2] In other words, optimizations are really important.

Tracking State We must distinguish between the implications of a **stateful** vs. **stateless** model. A stateless model requires requests to be self-contained; this is highly limiting as it prevents us from supporting caching and consistency management. It also results in larger requests, but it does make server-side processing much simpler and cheaper. The biggest benefit is resilience towards failure: just restart. Stateful systems offer better performance at the cost of complexity in check-pointing and recovery.

Caching This is an important optimization to minimize network latency. Clients maintain some file state and operate on it locally. This introduces the problem of consistency and coherence across clients, which is much like the consistency problem between caches in multi-processors. There, we used write-invalidate and write-update techniques to enforce coherence; here, we can have both client- and server-driven approaches that are dependent on the specific file sharing semantics of the DFS.

> The Sprite caching paper motivates its design based on empirical data about how users access and share files. Do you understand how the empirical data translated in specific design decisions? Do

you understand what type of data structures were needed at the server- and client-side to support the operation of the Sprite system (i.e. what kind of information did they need to keep track of, what kids of fields did they need to include for their per-file/per-client/per-server data structures).

The empirical data, though likely to be largely bogus and completely inapplicable when considering modern-day file access patterns (most files are open for 0.5 seconds? really? okay dude; that's not even enough time to edit it), affects the design of the Sprite NFS in significant ways.

Write-Back Delays Sprite adopted an approach in which every 30 seconds, dirty blocks that had been since unmodified during that period (i.e. in the last 30 seconds) are written out of a client's cache to the server's cache. After another 30 seconds under the same conditions, it would be written to the server's disk. This design choice was made under the observation that "20-30% of new data was deleted within 30 seconds, and 50% was deleted within 5 minutes;" as such, it made sense to delay disk writes in hopes that they wouldn't be necessary in the first place, since the original write would be deleted.

Rejecting Write-on-Close Part of the rationale to implementing delayed write-back was rejecting the write-on-close alternative. The BSD study indicated to the authors that "75 percent of files are open less than 0.5 seconds and 90 percent are open less than 10 seconds," which indicated that on-close writes would occur very frequently, and thus still incur a high cost overall.

Benchmark Estimates The authors used the BSD study to approximate how many simultaneous clients a machine running Sprite could maintain. They used the "average file I/O rates per active user" to approximate load, and extrapolated this to the maximum total of concurrent users under the various networked filesystem implementations.

In the Sprite NFS, various parts of the system track different metadata points. For example, to prevent the use of out-dated caches, both the clients and the servers store version information about their files; the clients compare their local versions to that of the server when they open them, in order to see whether or not their version is out-dated. Both endpoints track per-file timers for the delayed write-back, as well as "cacheability" flags that kick in when there's a writer to a file. The server also tracks both the current and the last writer of a file; the last writer is referenced when the server needs its dirty blocks (since these are sometimes fetched on-demand instead of pushed).

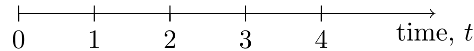
Part 4 – Lesson 3: Distributed Shared Memory

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3391619283/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1haUkgr-HHHtPivy7aNglOaENRua4i1O_cQGFsvM_fA0

Consistency Model Notation & Summary

This is the notation for consistency models outlined in lecture. The notation $R_{m_1}(x)$ means that the memory location x was read from main memory location m_1. Similarly, the notation $W_{m_1}(y)$ means that the memory location y was written to main memory location m_1.

Furthermore, we use a timeline like the following to demonstrate an absolute view of the system, in which each tick in time and operation is viewed perfectly by an omniscient outside observer.[a]



Consistency models have a lot of small variations. The following is a quick enumeration of their guarantees without the lengthy examples and explanations that follow in this section, for reference.

Strict All nodes see all updates immediately.

Sequential Updates from multiple nodes can arrive in *any* order but must arrive in the *same* order to all nodes. Updates from the same node must arrive in the order they occurred.

Causal Updates from multiple nodes can arrive in *any* order at *any* node. Updates from the same node must arrive in the order they occurred. When a write occurs after a read, the update of both the variable read and the new write must occur in that order.

Weak No guarantees outside of `sync` boundaries. On a `sync`, the node is guaranteed to see all updates that occurred from other nodes up to their latest `sync`.

It's important to differentiate an *update* vs. an *operation*. This distinction is discussed further in the sequential consistency section.

[a] Though they're not necessarily an omnipresent, omnipotent, and perfectly-good observer as in the Christian view of God, we *can* guarantee that said observer won't lie to us about events occurring on the systems.

When sharing state, what are the tradeoffs associated with the sharing granularity?

The choice of granularity is correlated to the layer in the software stack in which the shared memory implementation resides. For example, sharing by the page is typically tightly-involved with the operating system, since the operating system also manages memory by the page. Other approaches, like per-object sharing, are application- or runtime-specific. One thing to be careful of as granularity increases is **false sharing**, which is the phenomenon that occurs when two applications operate on different portions of the same shared memory region. For example, given some 1KiB page, application instance A might need the first 100B and instance B might need the last 100B. Neither of these technically care that the other is operating in the same page, but since granularity is page-based, there will be a lot of (pointless) synchronization overhead between them. Similarly, with object-based sharing granularity, one might need *only* `Object::member_a` while another needs only `Object::member_b`.

For distributed state management systems (think distributed shared memory) what are the basic mechanisms needed to maintain consistency – e.g. do you understand why it is useful to use "home nodes," and why we differentiate between a global index structure to find the home nodes and a local index structures used by the home nodes to track information about the portion of the state they are responsible for?

When distributing state it helps to permanently associate a particular piece of state with a "home." In the context of distributed shared memory, this is called a **home node**, and it's responsible for maintaining metadata about the state. This includes tracking access and modifications to its memory segments, cache permissions (i.e. whether or not a particular memory location can / should be cached), locking for exclusive access, etc. The home node is different than the **state owner**, which is the (temporary) owner and operator on a particular memory segment. Of course, the home node does *track* the current owner; all of this metadata, indexed by the object identifier itself, is maintained locally in a local metadata map.

Since the home node is a static property of a memory segment, the mapping between memory and its respective home (called the **global map**) is distributed and replicated across every node in the system. The prefix of a shared memory address identifies its home node, whereas the suffix identities the resulting frame number. In a way, the address behaves akin to how a virtual address behaves under virtual address translation: the first part is an index into a global mapping to nodes (like the index into the page table), and the second part is a frame number (like the offset into physical memory).

Move, Home Node, Get Out the Way

By adding a level of indirection between a shared memory address and its home, we can allow the home to move. Instead of treating the address prefix as a node ID, it's translated to an index into a table that maps to a dynamic home node.

If we introduce explicit replicas to the system – existing alongside the on-demand replicas that are already temporarily present in local caches – we can implement features like load balancing, hotspot avoidance, and redundancy. Management of such replicas can either be done by the home node or a higher-level **management node**, which effectively acts as a home node for one or more backing nodes.

Do you have some ideas for how you would go about implementing a distributed shared memory system?

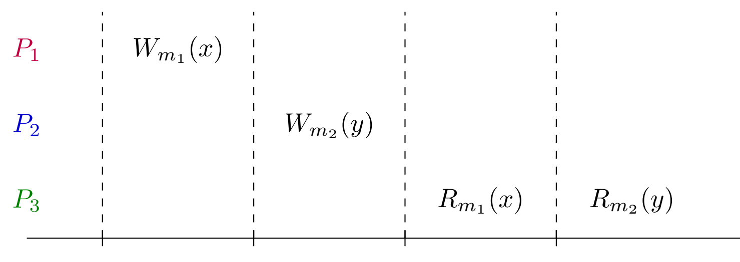
Yeah: no caching, every page fault results in a fetch from its corresponding node. Nodes get exclusive access to pages. Boom, ezpz. Perfect consistency. *(jk, I'm just lazy)*

What's a consistency model? What are the different guarantees that change in the different models we mentioned (strict, sequential, causal, and weak)? Can you work through a hypothetical execution example and determine whether the behavior is consistent with respect to a particular consistency model?

A consistency model is a convention defined by the DSM implementation; it makes certain guarantees about the availability, validity, and "current-ness" of data to higher-level applications *as long as the applications follow the rules*. There are different consistency models that guarantee different things with varying levels of confidence, but the most important part is that the applications need to *follow* the model for those guarantees to hold.

The two main operations for managing consistency are **push invalidations** and **pull modifications**. The former is a pessimistic approach that assumes any changes will need to be visible elsewhere immediately; it proactively pushes invalidations to other systems as soon as they occur. The latter is a lazier approach with a more optimistic outlook; as memory is needed, modification info is reactively pulled from other nodes on demand.

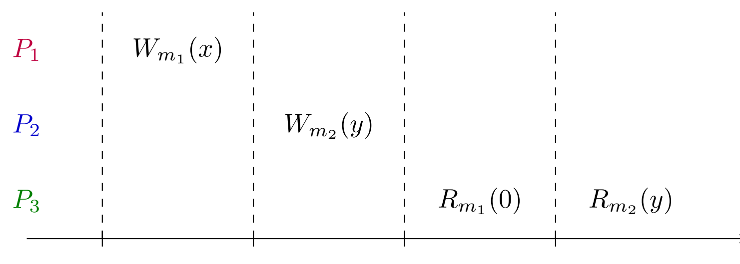
Strict Consistency This is the simplest model: every update is seen everywhere, immediately, and in the exact order in which it occurred. Consider the following timeline:



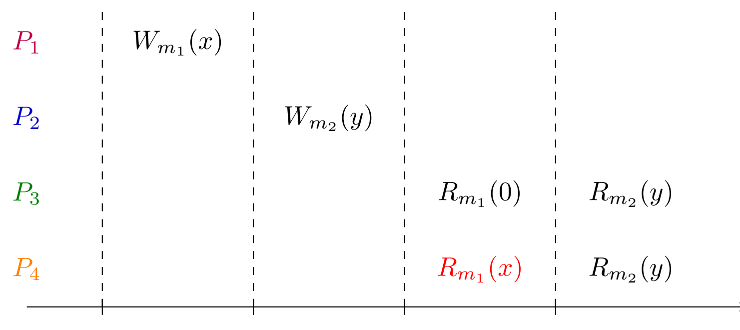
With a strict consistency model, both of the write events are seen by P_3 "immediately," and in the order they occurred, regardless of any latency variation due to spatial locality. For example, if the time deltas were small and P_2 was physically near P_1, and its write event was received by P_1 before P_3's, it still wouldn't be reflected in P_1 until after P_3's was received.

This model is purely theoretical: in practice, even with complex locking and synchronization, it is still impossible to guarantee (or sustain) this level of consistency because of physical complexities introduced by packet loss and network latency.

Sequential Consistency This alternative model is the "next best thing" regarding an actually-achievable consistency. Here, we only guarantee that any *possible* ordering of the operations – in other words, any ordering that would be possible in execution on a regular, *local* shared memory system (think data-races between threads). In other words, the following timeline is a valid alternative to the one guaranteed by the strict consistency model:

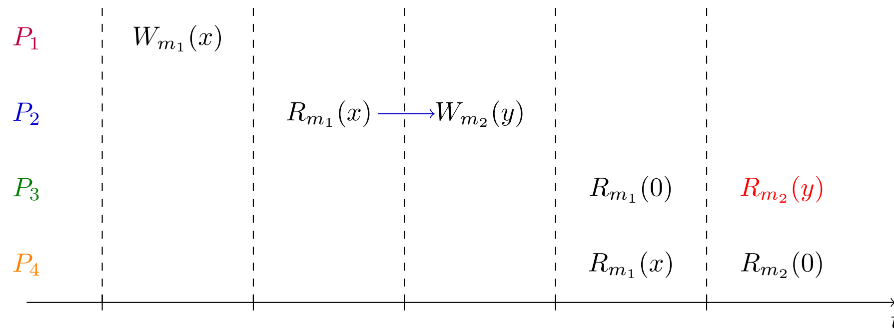


In summary, memory updates from different processes may be arbitrarily interleaved. There is a catch, though: this model must guarantee that the interleaving is the ***same*** across all processes. In other words, another process P_4 must see the same order as P_3. The following timeline would ***not*** be allowed, as it violates this principle:



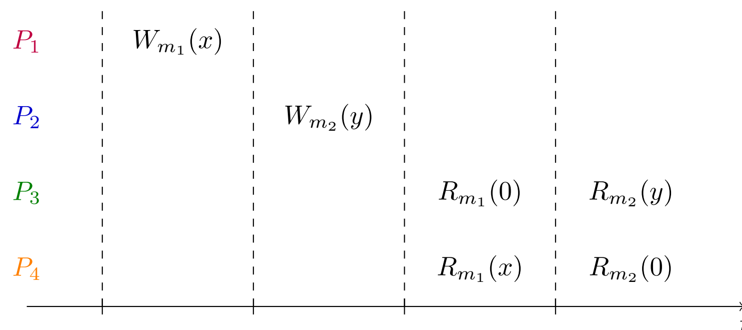
There is a final requirement to this model: operations from the same process always appear in the same order that they were issued. It's important to distinguish the time of an *update* from an *operation*. For example, in the above graph, P_4 seeing $R_{m_2}(0)$ would still be valid; it's possible that it hasn't received the update with y yet!

Causal Consistency The causal consistency model enforces consistency across relationships between memory locations. For example, consider the following timeline:



The arrow in P_2's timeline indicates a dependency relationship between y and x. For example, maybe $y = x^2$. Thus, it's non-sensical reading the value of y without knowing its *cause* (e.g. x in m_1). The causal consistency model is aware of such dependencies; thus, under this model, the m_2 read in P_3 would be inconsistent. P_3's first read should instead be $R_{m_1}(x)$.

As in sequential consistency, writes occurring on the same processor must be seen in the same order that they occur. The causal consistency models make no guarantees about **concurrent writes** (i.e. writes that are not causally related), though. Like in the sequential consistency model, reads occur in any *possible* ordering of the writes, but ***unlike*** in the sequential consistency model, these interleaves do not need to be consistent across processes. In other words, the following is valid under causal consistency, but violates sequential consistency, because sequential consistency mandates that if P_4 sees the x write first, then all other processes must also see the x write first:



How does the model define a causal relationship? If lecture inference and Slack discussion can be trusted, a causal relationship is *implicitly* defined: if a memory location is written to after another has been read from, the system **assumes** a causal relationship between them. It may ***not*** *actually be causal*; in the above example, it's possible that $y = 10$ rather than x^2, but the system would still enforce a causal relationship because of the memory access pattern. This lack of a programmer-enforced relationship leads to the following model, which tries to enable such enforcement and not make inferences.

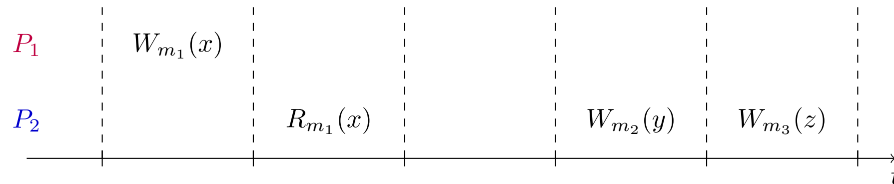
Helpful Hint: Consistency Consistence

It's important to note that causal consistency is **strictly weaker** than sequential consistency. In other words, sequential consistency implies causal consistency. This fact is super useful when answering questions regarding whether or not a system is causally and/or sequentially consistent. If it's not causally consistent, it's *definitely* not sequentially consistent, so you don't even need to consider it.

Weak Consistency This model enables the programmer to define explicit **synchronization points** that will *create* consistency with the system. A synchronization point does the following:

- It enforces that all *internal* updates can now be seen by other processors in the system.
- It enforces that all *external* updates prior to a synchronization point will eventually be visible to the system performing the synchronization.

Consider the following timeline, without synchronization points:



Suppose, then, that P_1 performs a `sync` after its write. This does **not** guarantee that P_2 will $R_{m_1}(x)$ as it does in the timeline; in fact, it may $R_{m_1}(0)$ and that would be perfectly valid. Until P_2 does *its own* `sync` call, its view of the system may be outdated. Of course, when it *does* make the call, the consistency model guarantees that it will see all updates from other processors that have `sync`'d their changes.

A weak consistency model allows the programmer to define explicit synchronization points that are important to their application, essentially managing causal relationships themselves. The model makes *no* guarantees about what happens between `sync` operations. Updates can (or can not) arrive in any order for any variable. Variations on the model include a global `sync` operation that synchronizes the entirety of the shared memory, or more granular synchronization that operates on a per-object or per-page basis.

Part 4 – Lesson 4: Datacenter Technologies

Lecture video: https://classroom.udacity.com/courses/ud923/lessons/3653269007/concepts/last-viewed
Transcript: https://docs.google.com/document/d/1Fe9EqPEbIZmfEPLN6S2WGmvHLizeLKvIbHKsyW-TR7Q

> When managing large-scale distributed systems and services, what are the pros and cons with adopting a homogeneous vs. a heterogeneous design?

A **functionally homogeneous** approach to distributed systems means every system is equal: it executes every possible step necessary for the processing of a request. This is much like the standard "boss-worker" model, in which workers are generic threads that process all of the steps in a certain task. Alternatively, a **functionally heterogeneous** approach isolates certain steps to certain machines, or for certain request types. This is somewhat akin to the "pipeline" model, in which individual steps of a task are isolated and can be scaled according to the time it takes to complete them.

The tradeoffs between each approach are similar to the tradeoffs we discussed in Part I of this course regarding boss-worker and pipeline models, with some additional tradeoffs that are a part of the "distributed" aspect of things. A homogeneous model is simpler; the front-end is more straightforward and just behaves as a simple dispatcher. Scaling is simpler, as it just requires adding more nodes to the system. Of course, the simplicity and generality comes at a performance cost, since specific things can't be optimized. Caching is difficult, since a simple front-end isn't aware of which node processed which task, and thus can't make any inferences about existing workload state. With a heterogeneous design, both the front-end and the overall management is more complex, but now the dispatcher (and workers themselves) can make meaningful inferences about data locality and caching in its workers.

> Do you understand the history and motivation behind cloud computing, and basic models of cloud offerings? Do you understand some of the enabling technologies that make cloud offerings broadly useful?

Cloud computing is beneficial due to its elasticity and abstraction. There's no need to manage infrastructure; the scale of your application is directly correlated with its demand, saving money and resources. Management of your application is standardized and easily-accessible via a web API provided by the cloud computing service. Billing is based on usage, typically by discretized "tiers" that correspond to the power of a rented system.

The cloud operates on a massive scale and requires technologies that support such a scale. Aside from the obvious need for virtualization to isolate tenants, clouds also need **resource provisioning** technologies to fairly allocate, track, and manage resources for their tenants. Furthermore, **big data** technologies are essential in storing and processing the insane amounts of data that their customers (and the providers themselves) produce or consume. The necessity of flexible "virtual infrastructure" by customers also means **software-defined solutions** for networking, storage, etc. are essential in providing these definitions. Finally, **monitoring** at scale requires complex real-time log processing solutions to catch and analyze failures.

> Do you understand what about the cloud scales make it practical? Do you understand what about the cloud scales make failures unavoidable?

The **Law of Large Numbers** makes cloud services viable. Different clients, of which there are many, have different usage patterns and different peak times. Averaging these factors leads to a relatively steady pattern of resource usage for the provider.

Unfortunately, this same law leads to unavoidable failures, too. The scale at which cloud services operate means massive amounts of hardware. Even if the failure rate of a particular component – say, a hard drive – is 0.01% a month (that's 1 in 10,000), a sufficiently-large data center might have 100,000 hard drives. That means about 10 hard drives will fail a month, every month! Of course, that's just hard drives; there are hundreds, if not thousands, of individual components in a particular system. Thus, individual failures happen with a fairly high frequency.

Additional Resources

Though some say that,[3]

> *George's alternative is the more simple and concise one. What a time to be alive.*

there are other guides available on this topic, some crowd-sourced, others not. Please check them out and thank them if you see them!

- David: https://docs.google.com/document/d/1DwVQL8jKrCsZyTtc5C4GgVhe8tdBb1t7RR7nWnX9x4E
- Piazza: https://docs.google.com/document/d/1XIbwSYZ7krXvw16zhkt3h88ZvMaurTrHo5em0eFdhSg
- Kevin: https://docs.google.com/document/d/1xA4v5jmv6Z6aOarR0iUypwOafW2WzKawG-9TQl66aEw
- Aja: https://docs.google.com/document/d/1-tHwY3BArTVzijbbalNM-V9KFFT40614o21OS4rl5OA

[3] Thanks, Joves: https://omscs6200.slack.com/archives/CE9LY0M5M/p1543874516119400