CS6200 Midterm 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)*

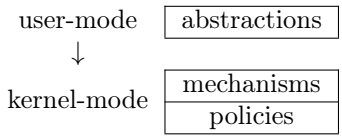
Questions: Part 1

What are the key roles of an operating system?

An operating system is the key **mediator** between applications and the system hardware resources. It provides **common abstractions** to differing hardware interfaces, such as "file" objects for both hard disks and solid state drives. It provides **security** via process isolation through virtual memory, as well as by restricting access to hardware resources. It **enforces fairness** among multiple, potentially disjoint, users and their processes by task scheduling algorithms.

Can you make distinction between OS abstractions, mechanisms, policies?

Abstractions are the user-facing virtual representation of access to hardware resources, like sockets for network hardware, as well as virtual concepts like the idea of "users" themselves to create a layer above the hardware that allows complex interactions. **Mechanisms** are the back-end that supports the way that abstractions are implemented. For example, "users" are a virtual concept that is implemented through the mechanism of profiles, access control lists, etc. Finally, **policies** are the varying mechanisms that can exist for a particular abstraction. For example, user threads (an abstraction), which are scheduled on the CPU by their respective kernel thread (mechanism), follow a particular scheduling policy, which can be something like a "round-robin" policy, or "shortest task first" scheduling.



What does the principle of "separation of mechanism and policy" mean?

In essence, this is just further abstraction but on a lower level in the system: for kernel developers instead of users.

This ties into the idea of "separation of concerns." A particular mechanism, such as scheduling, shouldn't actually care what the scheduling policy is. It should delegate that responsibility to the policy itself, which should be swappable / configurable. By providing a common interface for policies, it makes it easier for

people to define their own without actually being concerned with how its implemented; they can trust the mechanism to do the right thing based on their policy.

What does the principle "optimize for the common case" mean?

It's pretty self-explanatory: the design should be optimized for the way it's most likely to be used. For example, the "interrupts as threads" design is beneficial to overall system performance because despite the fact that there is a cost incurred for interrupts, there are savings on *every* mutex lock/unlock operation. Because the latter occur much more frequently than the former (common case), it's an overall performance gain.

What happens during a user-kernel mode crossing, and what are some of the reasons why user-kernel mode crossing happens?

This "crossing" is known as a **system call**. These special calls are the only means through which users can interact with the system in a privileged way. Here's what happens, to some granularity of detail:

- A permission flag is set in hardware that indicates "kernel mode" privileges.
- The program jumps to the kernel **interrupt handler**. Here (based on the system call parameters) it will jump to the system call requested.
- The system call begins executing in the context of the kernel. This involves things like checking parameters, (safely) copying data from userspace, etc.
- The requested privileged action is taken (such as telling a disk driver to read some bytes and place them in a particular memory location).

Primarily, these crossings involve interaction with privileged resources and hardware. For example, applications reading from a file (`read()`), requesting more memory (`mmap()`), creating network resources (`socket()`), etc. are all examples of actions that require crossing into kernel mode. Crossings can also occur if users violate rules. For example, if a user tries to access a privileged memory address (recall that part of a process' address space is reserved for the OS), the CPU will trigger a trap[1] and the kernel will take control to investigate.

Similarly, the boundary can be crossed in the "opposite direction" if a **signal** occurs[2] If the program has the appropriate **signal mask** bit enabled, the kernel will transfer control to its **signal handler**.

What is a kernel trap? Why does it happen? What are the steps that take place during a kernel trap?

Though the terms trap and interrupt are sometimes used interchangeably, for the sake of differentiating this from the later "what's an interrupt?" question, we will say that a **kernel trap** as a kernel interruption that occurs in "exceptional" conditions. Exceptional conditions are cases in which errors are encountered. This could be something like a disk read failing or a user process attempting to access protected memory.

During a kernel trap, control is given to the kernel's interrupt handler with certain parameters (the type of trap, arguments, etc.), and the handler will inspect the cause of the trap. If it's caused by the user, it will take action accordingly; this could mean terminating the process, returning an error code to the trap-causing function, etc.

[1] You just activated my trap card 😎
[2] For example, pressing Ctrl+C in your program will trigger `SIGINT`.

Contrast the design decisions and performance tradeoffs among monolithic, modular and microkernel-based OS designs.

Each of these designs has many trade-offs, from efficiency to maintainability.

Monolithic OS: These include FreeBSD, OpenBSD, Solaris, and MS-DOS.

This design contains every service/abstraction at the kernel level. It is first and historical OS design: every service an application could need is part of the OS. This includes memory managers, device drivers, file management, processes/threads, scheduling, file systems for random *and* sequential access, etc.

Benefits: Everything is included in the OS. The abstractions, all the services and everything is packaged at the same time. This creates the possibility for **compile time optimizations**.

Disadvantages: Too much state, too much code that's hard to maintain, debugging and upgrading. The size also poses large memory requirements that can noticeably affect performance.

Modular OS: This includes most modern operating systems, including WinNT, macOS, and Linux.

This design contains basic services and APIs at OS level; everything else can be added as a module. This delivers significant architectural improvements over monolithic OS's due to its simplicity and "plug-and-play" approach. With this design, you can easily customize which file system or scheduler the OS uses. The **OS specifies interfaces** that any module must implement to be part of the OS, making it possible to install modules that are applied for different workloads, instead of having the OS contain all possible modules by default.

Benefits: These OSs are easier to maintain (patching, new modules, etc.) and upgrade. Smaller code bases and less resource-intensive code means more memory is available to applications; this can lead to better performance.

Disadvantages: Because of the levels of abstraction and interfaces defining the OS-module layer, there is a performance impact due to the bigger call stack and more general-purpose code required. We also miss out on inlining and compile-time optimizations on the system as a whole (because there is no "whole" anymore). These aren't very significant on modern-day hardware, though. Furthermore, the modular architecture means that code can come from many different places; this makes tracking down the cause of an error more complex, and development spans many groups and codebases.

Microkernel OS: This is used more historically, or in embedded systems. It includes kernels like MINIX 3, Mach, and QNX.

These operating systems only include the most basic primitives at the OS level; this is just the necessities for executing applications (address spaces, threads). Everything else, including file systems, device drivers, etc. are outside of the kernel at user level. This model requires a lot of interprocess interaction, which the microkernel must support (well!) as another one of its core abstractions.

Benefits: These kernels are small: less overhead and better performance for kernel code. Testing is another benefit; it's easier to verify that behaves as intended. As such, microkernels are valuable in cases where this is a critical aspect of the system more so than performance (typically embedded devices and control systems).

Disadvantages: Portability is questionable; the kernel is typically optimized for particular hardware and can be highly-specialized for the expected use case(s). Thus, it's harder to find common components for software, leading to complexity. Finally, as discussed previously, the lack of kernel support for common features leads to a lot of IPC and frequent user-kernel crossings, which are costly.

Questions: Part 2

Process vs. thread, describe the distinctions. What happens on a process vs. thread context switch.

Both **processes** and **threads** are units of execution. A process represents the current state of a program's execution; every new launch of the program results in a new process. Their distinction is primarily in the way their details are stored in the kernel. A process is the "parent" of its child thread(s). Its info is stored in the **process control block** (or PCB) and contains details like the **virtual address mapping**, the **signal handler**(s), any open **file descriptors**, etc. Its threads, on the other hand, store things like **register state**, their respective **stack pointers**, **thread-local storage** that isn't visible across threads, the individual **signal masks**.

When context switching a process, the entirety of the current PCB is swapped out for another process' PCB. This means that the new process now has access to the CPU and the old process is no longer executing. A context switch empties the cache (**cold cache**), so the new process has to "start from scratch" in terms of the memory it needs to access, which is a performance impact.

When there is a thread context switch, only the thread items are swapped out. Shared data being accessed could remain resident in the cache. The address space mapping also stays the same, since we're still within the same PCB. There is still quite a lot of stuff to swap out still, though.

Describe the states in a lifetime of a process.

Processes are in two "primary" states: **running** or **idle**. Various actions move the process between these states and to the other secondary states.

As you can see in Figure 1, there are many stages that a process goes through. When a process is initially created, it's in the **new** state. It's then placed on the **ready** queue. From there, it moves into the **running** state at the kernel scheduler's leisure. If it performs a "blocking" operation like a disk read that would cause the process to pause for an extended period of time, it's moved to the **waiting** queue until that operation completes (typically via a hardware interrupt). Then, it's ready again. Even if is doesn't do any blocking I/O, it will still end up on the ready queue when interrupted by the scheduler, so it doesn't hog processing time indefinitely. When it's done, it enters the **terminated** state to eventually be cleaned up by the kernel.

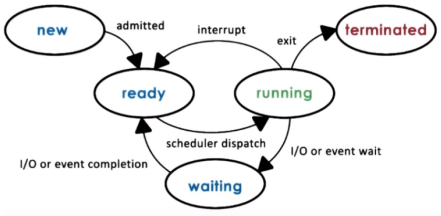


Figure 1: The Solaris process life cycle.

Describe the lifetime of a thread.

The lifetime of a thread is quite akin to that of a process; in fact, Linux doesn't really even track the concept of a "process," everything is treated like a "task" or thread.

There are some nuances, though, as seen in Figure 2. If a thread blocks on a synchronization variable, it enters the `SLEEPING` state, which is somewhat akin to the process `WAITING` state. There is also the concept of **thread parking**, which is an optimization to reduce context switches. A thread being "parked" means it's "set" on a semaphore; when that semaphore is "awoken," it can immediately resume the thread instead of context switches, checking queues, etc. Parking occurs when a bound thread blocks (since its LWP can no longer do anything), or when an unbound thread blocks and there are no more `RUNNABLE` threads (since there's nothing to switch to).

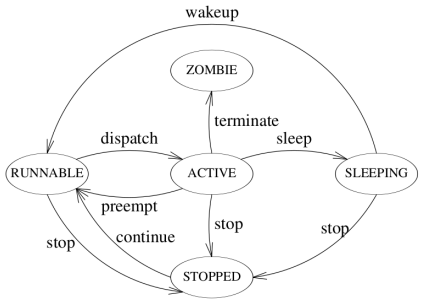


Figure 2: The Solaris user thread library life cycle.

An important thing to mention is the concept of the ZOMBIE state and **thread reaping**. A thread in the ZOMBIE state is considered to be dead, but its resources haven't been recovered. When a thread has completed its task, it enters the ZOMBIE state, but there are two variants as to what happens next:

- When a **detached** thread exits, it's put on the "deathrow" list, indicating it should eventually be cleaned up by the **reaper thread**.
- When an **undetached** thread exits, nothing happens to it until another thread `join()`s it, which causes it to be reaped.

What's the purpose of a reaper thread and what is "reaping"? This is an optimization to speed up both thread destruction and thread creation. By placing a dying thread on a queue, non-critical cleanup is delayed until there is idle time. After a thread is reaped, its allocated stack is placed in a list of free stacks; when a new thread starts, it can reuse one of these stacks instead of the costly operation of requesting a new one.

Describe all the steps which take place for a process to transition from a waiting (blocked) state to a running (executing on the CPU) state.

Let's assume that a thread was placed on the waiting queue following some blocking I/O operation, and now that operation has complete. A hardware interrupt will trigger a handler in the device driver, eventually somehow indicating to the kernel that the thread is now runnable. The kernel will place it on the ready queue and move on. Eventually, the scheduler will decide that this thread can begin executing. The kernel will restore process state from the PCB; this is things such as CPU register contents, local stacks, signal handlers, and masks. Finally, it will clear the kernel-mode permission bit (important! without this the user code is executing with kernel permissions) on the CPU and set the instruction pointer (`$PC`) to point to the last-executed instruction of the process; this gives controls over to the process.

What are the pros-and-cons of message-based vs. shared-memory-based IPC.

These are similar in terms of how they function. In **message-based IPC**s, there is a buffer in kernel memory that is accessible by both processes, whereas in **shared-memory IPC**s, the "shared buffer" is within the address space of both processes, outside of the kernel.

In the former case, the processes use kernel-provided APIs to communicate across the buffer (some form of `read()` and `write()`). This has the benefit of having a unified, standardized communication model, but suffers from the necessity of constant user-kernel crossings.

In the latter case, the processes gain the performance benefit of eliminating the OS from management, but have to suffer the downside of defining their own means through which to use the shared buffer to communicate. Eliminating this boundary is a *huge* performance gain, but setting up this shared buffer has a higher initial cost. Thus, shared memory IPC is only better if the higher initial cost can be amortized over the lifetime that the buffer is used.

What are benefits of multithreading? When is it useful to add more threads, when does adding threads lead to pure overhead? What are the possible sources of overhead associated with multithreading?

Multi-threading allows a system to always stay busy, or "hide idle time." For example, if an operation requires a disk read, it will need to wait for a long time. Allowing another thread to do something in the meantime is an efficient use of time and resources.

Threads are only beneficial in certain situations. If you need to process something in order, for example, threads would have no benefit. You'd end up with a clusterfuck of mutexes effectively making your threads

operate synchronously, with the additional overhead of constantly having to switch between them. If an operation is highly parallelizable (i.e. separable into disjoint tasks), though, threads can have a huge performance benefit. The general rule of thumb is that threads are beneficial if

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

This is just a rough guideline and doesn't take into account things like the performance hit of evicting data from the cache. Too many thread context switches can throttle and harm performance. This is further limited by things like the amount of CPUs available, the threading model used by the system (1-to-many, $m \times n$, etc.)

Describe the boss-worker multithreading pattern. If you need to improve a performance metric like throughput or response time, what could you do in a boss-worker model? What are the limiting factors in improving performance with this pattern?

The boss-worker multi-threading pattern involves using a **task queue**, managed by the boss and referenced by the workers, to distribute a single workload among multiple threads. The boss adds tasks to the queue and signals to workers that there is work available. Workers then remove the tasks from the queue and process them.

The task queue is uniform; that is, every task is the same. For example, a task may be responding to an incoming request on a web server. Each worker will take approximately the same amount of time as all of the others to complete a task (obviously, specific tasks may take longer, but in the average...). Furthermore, every worker operates independently. This means that you can only scale "horizontally," i.e. by adding more threads.

Improving throughput. The definition of **throughput** is the amount of tasks that can be completed in a given amount of time. Because each worker thread processes a discrete task, the throughput of a boss-worker pattern is limited by the number of workers; improving throughput means increasing the number of simultaneous workers. Of course, even a heavily parallelizable process is still limited, since thread context switching has a non-negligible cost.

Improving response time. Doing something like optimizing the boss thread's processing of incoming requests will mean that a worker will start on it sooner, which means it will get processed faster. Unfortunately, though, the time the boss spends on the task is typically much lower than the overall time the task takes. Thus, without optimizing the *worker itself* (i.e. the way a task is processed), response time can't really be lowered.

As such, the limiting factor in boss-worker performance is the worker. This is contrasted heavily with the pipeline model (discussed below), where individual portions of the task can be optimized, and different stages can have additional resources. An important calculation is the time it takes to finish n orders:

$$T = t_{\text{avg}} \left\lceil \frac{n}{w} \right\rceil$$

where t_{avg} is the average time it takes to finish a task and w is the number of *worker* threads (that is, the total number of threads sans the boss).

Describe the pipelined multithreading pattern. If you need to improve a performance metric like

throughput or response time, what could you do in a pipelined model? What are the limiting factors in improving performance with this pattern?

The pipelined multithreading pattern divides a task into disjoint stages and assigns workers to each of these stages. Each worker has a queue of pending inputs, which are the outputs of the previous stage.

Suppose a 2-stage pipeline has a fast first stage (10ms) but a slow second stage (100ms). This means that the bottleneck is the second stage. If two tasks are submitted simultaneously the first task (T_1) takes 110ms, but the second task (T_2) takes 210ms to complete:

Time (ms)	Stage 1	Stage 2	
t_0	T_1		Stage 1 pending: $[T_2, \ldots]$
t_{10}	T_2	T_1	
t_{20}		T_1	Stage 2 pending: $[T_2, \ldots]$
t_{110}		T_2	$T_1 \rightarrow$
t_{210}			$T_2 \rightarrow$

To generalize this, the total time (T) it takes for n tasks to complete an x-stage pipeline in the naïve model of a single thread per stage can be calculated as follows:

$$T = T_1 + ((n-1)t_{max})$$

Where T_1 is the time it takes to complete the first task (i.e. the duration of the entire pipeline) and t_{max} is the time it takes to complete the *longest* stage.

Improving throughput. Notice how the pending queue builds up for the longest possible stage, and this build-up accumulates a per-task delay for the rest of the pipeline and severely hinders throughput. To alleviate this delay, longer tasks can have additional worker threads! In this way, each pipeline stage takes on a boss-worker model. The longer the stage, the more threads should be available to keep the queue from building up. For example, for a 3-stage pipeline with stages taking 10, 30 and 20ms respectively, the second stage could have 3 workers and the third stage have 2 workers. This brings the time per *every* task to 60ms instead of just the first task. In this case (i.e. threaded stages such that each stage can keep up with the output of the previous stage), the formula hinges on whatever the fastest stage is:

$$T = T_1 + ((n-1)t_{min})$$

Improving response time. What's interesting about this model is that improving throughput also improves response time! By allowing each stage to process more requests simultaneously, the time it takes to process an entire request decreases. Again, though, the longest stage causes the longest build-up, so improving an individual task's response time can be achieved by optimizing the slowest stage.

What's a simple way to prevent deadlocks? Why?

The most-accepted solution for preventing deadlocks is to **maintain a lock order**. This means that all locks are taken in the same order, regardless of the unit of execution that requires them. It ensures that no threads are ever conflicting in access to separate locks, since they will need to wait on the previous lock first.

This works because it prevents a cycle in the "wait graph." Cycles cause deadlocks, and the absence of a cycle means a deadlock cannot occur, because there is always a resolution order.

Can you explain the relationship among kernel vs. user-level threads? Think though a general $m \times n$ scenario (as described in the Solaris papers), and in the current Linux model. What happens during scheduling, synchronization and signaling in these cases?

Whew, this question is deep. In general, user-level threads (ULTs) differ from kernel-level threads (KTs) in the amount of knowledge available about their friends on the opposite side of the privilege boundary.

The current Linux model, thanks to the low cost of memory and high speed of current hardware, maintains a kernel thread for every user thread. This is known as the **one-to-one model**.

Solaris Threading Model. It makes sense to discuss this model first because it's a generalized version of the Linux model. We can see the disadvantages of such an approach and how they're overcome later.

Scheduling. The Sun paper on Solaris approaches the generalized $m \times n$ scenario by introducing **light-weight processes** (or LWPs). LWPs have a 1:1 relationship with kernel threads, and they act like "virtual CPUs" from the ULT perspective. The ULT is assigned to the LWP, and the LWP is scheduled on the CPU by the KT. Thus there are two levels of scheduling: the scheduling occurring in the ULT library onto the LWP and the scheduling occurring by the kernel of KTs on various CPUs.

Synchronization. Synchronization is done by the ULT library. This comes with a big disadvantage, which is that the kernel has no knowledge of the sync actions by the ULTs. Consider the following case: a ULT is assigned to an LWP and blocks. The associated KT doesn't know that it can schedule a *different* ULT on the LWP since the current one is blocking. Thus, the user process thinks its hanging, despite the fact that there are threads waiting for the opportunity to work. This can be further generalized regardless of how many LWPs a user process has; if all ULTs block and there are others waiting, the kernel can't control scheduling. Another paper discussing a lightweight user-level threading library discusses introducing a new signal to the kernel for this, `SIGWAITING`, which would be triggered when the kernel detects that all of its executing threads are blocking. This would allow the LWP to perform scheduling.

Signaling. Recall that signal handlers are associated with a process and signal masks are associated with threads. There is a difficulty with managing signals because different threads may have different masks, yet we don't want to miss out on signals. If a thread is executing with a signal disabled, yet there is another thread that can execute to process that signal, we need to ensure that occurs. The threading paper proposes solving this by having a **global signal handler** that "pre-processes" all signals. Its mask is set to be equal or less restrictive than the intersection of the signal masks across all threads.

Linux Threading Model This model implicitly solves many of the workarounds in Solaris' model by simplifying the user-kernel interaction. It uses a one-to-one model, so the kernel is always aware of the user threads, as well as their requirements and actions.

Scheduling. The kernel sees all threads and their priorities can be refined to allow the scheduler to optimize things appropriately. There is no longer any need for user-level scheduling.

Synchronization. Primitives are provided by the kernel, and can thus be heavily optimized. User threads can be woken up immediately once a mutex is freed, rather than needing to traverse through a user-level threading library that relies on global synchronization primitives. Furthermore, blocking operations (like waiting on a lock) are trivial to recognize, and deadlocks can be avoided because the kernel sees all primitives.

Signaling. With 1:1 threads, there is no more management of signal masks. The kernel thread will deliver signals to its user thread if the mask is enabled, period.

> Can you explain why some of the mechanisms described in the Solaris papers (for configuring the degree concurrency, for signaling, the use of LWPs...) are not used or necessary in the current threads model in Linux?

This is touched on implicitly by the simplicity of the "Linux Threading Model" breakdown of the previous question, but we can dive into further detail here. **Degree concurrency** is used by the $m \times n$ model to control the amount of multiplexing from ULTs to LWPs (and thus KTs). More degrees means less dedication of a KT to a particular ULT. With a 1:1 model, each ULT gets a dedicated KT; thus, it doesn't need to worry about the degree of concurrency. **Signaling** is simpler because the KT signal mask always reflects the ULT mask; there is no ambiguity or need to have a global handler and check thread masks. Finally the **light-weight processes** aren't necessary because all of the data for a thread is stored within the thread itself; any shared data across threads is stored in their PCB.

> What's an interrupt? What's a signal? What happens during interrupt or signal handling? How does the OS know what to execute in response to a interrupt or signal? Can each process configure their own signal handler? Can each thread have their own signal handler?

Interrupts are hardware-triggered events that indicate to the kernel that something needs to be handled. Similarly, **signals** are software-triggered events that indicate to a process that something needs to be handled (or is handled for them, for certain signals like `SIGKILL`). For interrupts, the kernel has an **interrupt table** which jumps to a particular subroutine depending on the interrupt type. For signals, the process likewise has a **signal handler** which selectively enables certain signals using a thread-specific **signal mask**. The kernel calls the handler if the mask allows (i.e. has the bit *set*) the signal.

> What's the potential issue if a interrupt or signal handler needs to lock a mutex? What's the workaround described in the Solaris papers?

If code is executing and acquires a lock, then an asynchronous event occurs that triggers a handler (interrupting the aforementioned execution) and that handler tries to acquire the same lock, a deadlock will occur. For interrupts, this can be mitigated by spawning a thread that then executes on its own time after the lock is released. For signals, this is also viable, but more expensive. Instead, its suggested to clear the signal mask so that the offending signal will be ignored (or marked as pending) during the lock critical section and reset it afterward.

> Contrast the pros-and-cons of a multithreaded (MT) and multiprocess (MP) implementation of a web-server, as described in the Flash paper.

Honestly, an MP server doesn't really have any pros over an MT server on modern-day systems. It used to because it was more portable: kernel-level multi-threading support wasn't as ubiquitous as it is now. However, MT spanks MP in performance because synchronization across threads is much cheaper than the high cost of IPC on an MP architecture.

> What are the benefits of the event-based model described in the Flash paper over MT and MP? What are the limitations? Would you convert the AMPED model into a AMTED (async multi-threaded event-driven)? How do you think an AMTED version of Flash would compare to the AMPED version

of Flash?

An event-based model has many benefits: there are lower memory requirements, no context switches, lower "communication" costs for consolidation of data (like request logging or tracking statistics). Its limitations arise when processing disk-heavy requests; a single blocking operation halts the entirety of the server. In years past, there was also the issue of support for asynchronous I/O; not every OS allowed it, and even then, they may not do it right or reliably. A limitation that was lightly touched-on in lecture is that there is some overhead to an event-based approach when events are handled across multiple CPUs; there are complexities in routing and dispatching.

These limitations were the entire purpose of the AMPED model introduced for the Flash web server: it combined the efficiency of event-based request processing with the non-blocking benefit of MP file I/O handling. Just like MT leaves MP in the dust, I imaging that AMTED would likewise leave AMPED in the dust. The paper itself notes that AMPED is preferred as an architecture for portability. This is no longer an issue so AMTED is likely far superior.

There are several sets of experimental results from the Flash paper discussed in the lesson. Do you understand the purpose of each set of experiments (what was the question they wanted to answer)? Do you understand why the experiment was structured in a particular why (why they chose the variables to be varied, the workload parameters, the measured metric...).

The comparison graphs are a little hard to keep track of, so the terminology / abbreviations they use are defined here. The base model is the **AMPED** model (there's also AMTED which uses threads but this isn't compared) which is an event-driven architecture with helper processes for blocking operations. It's compared against its variants that replace the main with other architectures while maintaining the same performance optimizations they outline. Specifically, **MP** and **MT** variants refer to the multi-process and multi-threaded models that they developed specifically for a performance comparison. MP should be thought of as similar to **Apache**, whereas **SPED**, which is AMPED sans the helpers (i.e. purely single-process), is similar in design to **Zeus**.

With that out of the way, let's get into the summary of every performance comparison. Something is universally true across every scenario: **Apache sucks** relative to the others, which they try to gently attribute to the fact that it probably doesn't employ the same "aggressive optimizations" that they use. Some Flash variant (usually AMPED) also almost always out-performs Zeus, except in one special case outlined below. Furthermore, **MT outperforms MP** because of the expensive interprocess communication of the MP architecture, *but* MT isn't always available.

Synthetic workload: Clients requested the **same file multiple times**, with varying file sizes per series.

The curves are identical with minor increases in performance. SPED and AMPED were neck and neck (recall that these are both Flash variants), with SPED edging out because AMPED had to do some extra checks that provide better performance in later workloads.

This heavily tests performance on a cached workload, since the (identical) response is in-memory after the first request.

Trace-based workload: These use workloads based on a trace of two different systems: the **CS trace** and the **Owlnet trace**, with the latter dataset being much smaller and cache-friendly.

AMPED has the best throughput across the board; it's followed by **SPED for Owlnet**, which is cache-friendly, and **MT for the CS traces** which are disk-heavy.

Real-world workload: Logs from a real-world server are mimicked and truncated to a particular size to test different data- and working-set sizes.

Until the cache is exceeded, SPED beats AMPED beats Zeus. Then, there's a short range (approx. 100-110MB) in which Zeus beats AMPED beats SPED. This is attributed to Zeus prioritizing requests for small documents. After that, AMPED beats Zeus. The conclusion is that **AMPED is the best general-purpose architecture** which was their goal.

Flash optimization comparison: Different optimizations were tested against identical repeated requests.

The optimizations that they compared were **memory-mapped files** (essentially caching file contents in memory), **response header caching** (responding with a pre-made header for identical requests), and **caching pathname translation** (avoiding the lookup and parsing by storing the mapping in memory). The latter of these was actually the most impactful on its own. Without *any* optimizations, Flash's performance halves relative to having all of them enabled.

WAN comparison: Longer-lived connections stress-test increases in simultaneous requests.

In this, AMPED, SPED, and MT are all very similar. MT starts off ahead until about 200 clients, after which SPED pulls ahead. Again, AMPED would probably out-perform SPED once caches were insufficient.

If you ran your server from the class project for two different traces: (i) many requests for a single file, and (ii) many random requests across a very large pool of very large files, what do you think would happen as you add more threads to your server? Can you sketch a hypothetical graph?

Requests for the same file will benefit greatly from caching, whereas requests for a variety of large files will trash the disk very early on. This is results in a trend similar to Figure 3.

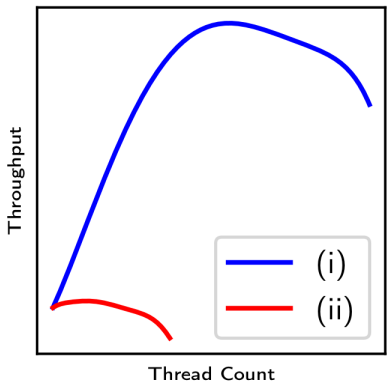


Figure 3: Theoretical performance of the GETFILE library under two different traces. (lmao I know this graph sucks but I got tired of trying to make `matplotlib` show what I wanted it to show...)

For (i), even if the file is larger than the available cache, requests will still hit a hot cache very often. Thus, the throughput is nearly linear as the number of threads increases. Eventually, though, thread context switches for every request will be costlier than the benefit of quickly serving a cached file. For the (ii) case, despite a slight increase in performance as multiple requests can be handled simultaneously, performance drops like a rock as threads spend increasing amounts of time waiting on I/O (longer than they would alone). This is called **disk trashing**, and it occurs very early on. Caches are constantly invalidated and performance suffers quickly as more threads are added, despite improving a bit at first.