CSCI 3120

Introduction

McAllister

Alex Safatli

Tuesday, May 7, 2013

Contents

Introduction

Administration

A textbook is available for this course; roughly the structure of the chapters of it will be followed and questions may be reproduced from the textbook and placed into assignments. A copy of the latest edition, the **Ninth Edition** of *Operating System Concepts*, will be available in the library.

Five assignments will be given in this course; not all assignments are 100% programming. Four of them will include programming and written components, and the one around midterms will be only written. The concepts of programming for this course are not trivial.

Programming will be done in C in this course in order to increase our level of comfort. Get used to debuggers again (`gdb`). Code will be handed in through SVN (see syllabus).

Resource Management

This course focuses on the concept of **resource management**: using resources provided efficiently and with convenience. Operating systems are common examples of resource managers were other examples include web servers, web browsers with sandboxes (e.g., Chrome), database management systems, and virtual machine hypervisors.

The course will introduce material with a single CPU (single core) in mind. Multi-processor and/or multi-core issues can be raised after the basics are clear.

Operating systems are software that acts as an intermediary between the user (or user programs) and the hardware; the **kernel** is the part of the OS that remains loaded in the computer – it is a subset of the operating system that does the work that should always remain in memory all the time.

What do we expect from an OS? Multitasking, reliability, speed/efficiency, hardware component access, portability, good user interface (easy and user-friendly), a file system, security, memory management, being correct and predictable, fair, extensible, stable, well-documented, protection, transparency (not to know it is there at all; should not see it and knows what it is doing or can find out), and flexible. In order to please the user, tailor towards many groups of users, and manage items efficiently, a trade-off has to occur. More or less emphasis could be put on different characteristics.

The OS provides several roles:

- **Provide an environment in which we can run programs**: I/O operations, file system access, accounting, shared libraries, and communication between processes.
- **Allocate and manage scarce resources**: memory, CPU, printing, and network activity.
- **Control processes**: process creation, execution, protection, error/conflict detection and response, and security.

If an infinite loop is compiled and ran in a program intentionally, the system should still remain usable and has to deal with trying to manage resources — this may involve **scheduling** and the key mechanism for this are (1) **interrupts** combined with **timers**. Before giving the CPU to the process, a timer is started that generates an interrupt to the OS when it expires. This ensures a regular guaranteed return to the OS.

NOTE `printf` is a system call provided by the OS; calling it implicitly returns control to the OS.

Resource Management continued on next page...

If the programmer could explicitly stop the timer or change the interrupt code, we would have another problem. Therefore, this is intentionally considered forbidden. The hardware could provide (2) **memory protection** so the timer or interrupt vector can only be changed under specific conditions – by the operating system.

A separate user and (3) **kernel/supervisor/monitor mode** (**privileged instructions and execution modes**) is provided where certain instructions are only available in kernel mode. The basic assumption is that if you can get into kernel mode, you are the operating system. The hardware is the mechanism that sets the bit for kernel mode – an implicit trust here is that the operating system will exit the mode.

The hardware places the CPU in kernel mode as it processes an OS-level interrupt. System calls can invoke a trap to invoke an OS-level interrupt where an index is provided to a short, protected table of operating system code. The OS code exits kernel mode before returning control to the user process. Entering kernel mode *cannot be a user-level command*.

NOTE Hardware can only depend on itself; for the hardware to remain in control of itself, it must be isolated and must set up a failsafe – for instance, safety-critical systems will have a deadman switch: a special timer that, if ever goes off, will reset the entire hardware. The entire purpose of the software is to reset the timer.

Interrupts and Events

When an **interrupt** is triggered, the user/kernel bit is flipped and an entry in a list of addresses in memory is referred to by an interrupt vector and pushed onto the program counter after the reference environment is saved (along with the mode — this is important because of interrupts interrupting interrupts; we do not want to make the assumption that we always flip the bit when coming out of an interrupt). This table can only be modified while in kernel mode (it is protected). **Each interrupt has an entry in an array of address pointers** (the **interrupt vector**).

NOTE Some hardware have a second stack: a **kernel stack**. This can only be used in kernel mode. If an entire user stack is filled, this may interfere with the kernel in hardware that does not make this differentiation.

NOTE Programmers are who store registers on the stack; the hardware only stores the **program counter** — the interrupt handler does the former.

Are all interrupts equal? No. We can differentiate between software and hardware events. We can define priority of interrupts, and their **maskability** (some can be delayed; this is defined by an **interrupt mask** – `0` for block). We also might want the user to be allowed to install their own interrupt handlers (in some cases) – this is done with the `signal()` system call and goes through the OS; these are what to do when you try to abort an application normally. Other interrupts must only be for OS code (what to do on a memory fault).

Entries in an interrupt vector that a user can change becomes specific to each program that is running; so, whenever a given program is run, a copy of memory is made. Anytime the OS changes between one program and another, this is changed as well. This is built into the idea of **program state**: not just registers, etc. In many operating systems, once

Interrupts and Events continued on next page...

NOTE Daemons are long-running programs. They sometimes have configuration files. As a system administrator, having to restart the entire system after changing it may be very frustrating; the same is true for rerunning the daemon. The `SIGHUP` is used here: a hang-up of the terminal and can be used to prompt to re-read config files.

NOTE `SIGUSR` user-defined interrupts allow us to lay traps in order to try and prompt the system into kernel mode. Part of the way through a signal call, a trap may be invoked to go into kernel mode. System calls may or may not require kernel mode to execute. Common API types to interact with the OS include Win32, POSIX (portable between many systems), and Java API.

System calls have many categories: process control, file management, device management (often abstracted to look like file management), information maintenance, and communications (network).

This leads us into **event management**. These could be in parallel with or not parallel to execution. There are two types of **events**:

- **Synchronous**: the executing program needs to be aware of the events as it happens; e.g. asking user for input.
- **Asynchronous**: the event can happen independently of the executing program (program can deal with it later); e.g. pre-filling a buffer with data from a disk.

This leads to two different behaviours to handle event: **event detection**. We can do this through **polling** or **interrupts**. Polling is when the program continually asks the device or system about whether or not a particular event has happened. This is typically integrated into the normal flow of the program. Interrupts are when an entity causing the event notifies the program (via the OS) when an event happens; program flow is diverted to deal with this event as it happens and is modelled as an exception to the normal flow of the program.

NOTE Sample code is available on the website. No marks are associated with creating a complicated data structure or a linked list; this is part of the process, though. The marking scheme sets aside marks for documentation, coding style, and testing. Helps to set up test cases before coding; **black box testing**.

Operating System Architectures

How will an operating system look like if it has to do all this? This is typically coming down to a design architecture. There are several definitions to what this is (see slide). It all has to do with having an arrangement of different parts in order to build a whole.

Standalone OS architectures include:

- simple (monolithic),
- layered,
- microkernel,
- modular, and
- virtual machine

The architecture is one of the first things to discuss when doing external documentation. It is central to how a system works. A **simple architecture** is typically for small or preliminary systems; it is almost not

Operating System Architectures continued on next page...

an architecture in itself. All of the OS runs in *kernel mode* (no well-defined structure, everything is small enough that we can still find pieces) and, if unchecked, it can lead to **monolithic systems** (all components are intertwined or interdependent — one big piece). Breaking these apart is typically known as **refactoring**. It ma be ideal to start here, but not to end here.

A **layered architecture** is a place one can typically go. It divides the functionality of the OS into different layers; different areas of responsibility. The layers are organized hierarchically: lowest layer deals with hardware, the next layer provides basic functionality (memory management, scheduling), and the next layer adds *more abstraction*. **Each layer only uses the services of the layer directly below it and only provides services to the layer immediately above it** — all interaction is done via APIs. If two layers are going to be straddled by a single feature, something should be rethought.

The advantages for this include that each layer is isolated from the others (can swap in or out new implementations of layers without affecting the whole system), and it is easier to construct, debug, and maintain because of the isolation. Disadvantages include the fact you need to carefully plan the functionality of each layer, some features conceptually might want us to form cycles in the layer structure, and it is not as efficient since one task may need multiple functional calls to get through all of the layers.

What happens most of the time here is that the entire set of layers lays in kernel mode: everyone has been given access to kernel mode in one shape or form. Think: device driver. This is the inspiration for the following. A **microkernel architecture** isolates the smallest amount of functionality into the **kernel**: process management, minimal memory management, and interprocess communication (remember: the kernel is the part of the OS that should be running all of the time; this is why it is as small as possible). All other functionality is run as separate user-space which goes through the microkernel when communication is needed between other spaces.

The advantages of this architecture include minimal functionality running in kernel mode, it is quick to swap in or out parts of the overall OS, and it is easy to add to the OS. The disadvantages include efficiency being sometimes a concern: must still go through APIs and possible leading to a monolithic system if kept unchecked (adding more into the microkernel). Furthermore, going into and out of kernel mode initiates more and more interrupts.

> **NOTE** Previous advancements in processing involved computation pipelining: these are thrown out during interrupts, typically.

Modular architecture allows OS functionality to be integrated into the OS on demand by using dynamically loaded modules (at boot time or at run time). Functionality is therefore divided into modules (like a layered architecture) but a module can call any other module; this is akin to Object-Oriented Programming.

> **NOTE** Realize that the compilation process for many files is reminiscent of this. After pre-processing, compilation, assembly, and linking, when a program is run, further modules can be loaded when needed: shared objects (`so`) at runtime or libraries such as `dll` files. This latter step is done by the OS. Solaris is run on a modular-type system.

NOTE Recall the principle of a **symbol table**: it ties in semantics to tokens and symbols (e.g. a function called `main` is a function that is global and has an associated address). For instance, when a `printf` function is called, the OS dynamically loads a pointer to OS code. The code is comprised of finding a library, loading it, finding a function, getting the address, updating the symbol table, and restarting the function call — the function has been stored in the program's memory space. *Providing a flag to the linker means it will link in all dynamic libraries, as well, so a completely stand-alone executable can be made.*

NOTE Unix is typically modular. Windows is typically hybrid: part of it works one way, and another one works a different way. Mac OS X and iOS is also like this.

Hybrid architectures are not uncommon, combining some of these architectures. A coarse set of layers are defined for the system, and a microkernel is used to instantiate the lowest layer. Mach is an operating system that was sold off as a microkernel in the mid-90s and has evolved over time. One of the modules that are connected to it provides access to the upper layers. The others do not touch the other layers. See slide.

A **virtual machine architecture** is almost a *meta-architecture*. Layering is taken to an extreme here. The OS provides an abstraction of a hardware machine to each process, and each process then installs its own kernel for its operation (known as the **guest OS**); each process can choose to use a different kernel or different style of kernel. This is advantageous because virtual registers, etc. can be mapped to real hardware and swapped in and out when necessary, meaning a variety of real hardware can be supported.

Therefore, **virtualized hardware** can be mapped directly to real hardware or be truly simulated by the VM. This could be done by **direct access** (efficient execution within the virtual machine; can create some risks of security compromise) and must not let the virtual machine entire kernel mode: all kernel mode operations are trapped to the host OS. An example of this is VMWare.

Simulated hardware provides better security to the host OS, allows for more flexible design of virtual hardware, allows for common virtual hardware across different real hardware, and is often seen as slower since assembly-level instructions are interpreted (e.g. early Java VM; see diagram on slide before).

Finally, a middleground uses simulated hardware with **Just-in-Time Compilation** (JIT). This has some trade-offs: certain sets of instructions are allowed to be interacted with, and dangerous code will remain in bytecode.

Broader operating system architectures also exist: distributed operating systems (clusters), network operating systems, parallel architectures (multi-core, multi-processor, clusters, parallel machines, etc.), high performance computing, grid computing, and cloud computing.

NOTE To test a linked list, one would typically test adding new elements (at the start, at the end, or in the middle) to an empty list, a list of one element, and a "full" list. Or they could test adding an item already in the list.

Processes and Scheduling

Process Management

A **process** is oftentimes used synonymously with the word **program**. In the operating system world, they are different. A program is a *static* series of machine instructions. On the other hand, a process is a program with its execution state (which is *dynamic*); you can have many processes all executing the same program

Process Management continued on next page...

with different state. A program, compiled into an executable comprised of program code, static info, and a symbol table, would have an appropriate stack and heap in a process. There will also be OS *context* about this process and a hardware state for process execution. **Processes are interdependent of one another**.

Other definitions are important here. **Multitasking** is the manner at which two or more programs are active with interleaved execution. On the other hand, **multiprocessing** involves a computing environment with *multiple* processors.

NOTE At one point in time, it was cool to create **self-modifying** programs. This is often discouraged; they are typically known as viruses or worms.

In more advanced operating systems, more than one process may share static elements of processes (like code and DLLs) — programs are typically assumed to be static. Typically, this is helpful for memory use and conservation.

Information contained about processes include:

- **execution state**: register values, instruction register, stack pointer, heap pointer, interrupt vector, interrupt mask, status register, and memory tables.
- **OS state**: scheduling information (priority, time on CPU, ...), security information (groups, permissions, ...), file table, child and parent processes (processes can start other processes; without this, we would only have one process on startup — shells are parent processes to processes they run), execution state, dedicated resources (locks), timers, and process ID.

All process information is stored in a **process control block** (PCB); the OS tracks or identifies processes by their PCB. In the Linux kernel, these are typically known as a `task_struct`.

The lifecycle of a process is typically as follows:

- it is started,
- a process is cycling between CPU bursts and I/O bursts; CPU bursts are periods of CPU-intensive computation and I/O bursts are periods where the progress of the process is gated by lots of I/O operations, and
- a process ends.

In modern-day operating systems, when a process asks for the I/O, we usually let another process us the CPU — it is, abstractly, in a state of waiting for some event. When the I/O is done, the waiting process does not necessarily get the CPU back. If a process control block was created or resources were allocated, these still have to be managed when a process ends: cleanup has to occur.

Process states are involved in a 5-state model: they can be...

1. **New** (*get ready to run*): the OS is creating the PCB and allocating initial resources to the process,
2. **Ready** (*wait to do stuff*): the process is waiting for its turn to use the CPU,
3. **Running** (*do stuff*): the process is running on the CPU,
4. **Blocked** (*wait for some event*): the process is waiting for some event to happen (I/O, timer, child to end, interrupt), and
5. **Exiting** (*clean up when done*): the OS is reclaiming the process' resource and notifying its parents.

Every device will typically have a **wait queue**. An OS will have a **ready queue**, **new queue**, **blocked queue**, and **exit queue** — these can be, to save time, done in batches; this is overhead for an operating system.

Transitions

Running to Ready Transition. There are two kinds of **scheduling**: **nonpreemptive** and **preemptive**. Nonpreemptive scheduling is when the process itself decides when its computations are in a stable state to give the CPU up (e.g., a system call); this is uncommon today. Preemptive scheduling is when the OS decides when a process has had enough CPU time; packages up the state of the process and moves it to the ready queue. The transition here is *undetectable* to the process. This is *very important* and key to preemptive scheduling.

A **context switch** is the event of saving the state of a process in the running state to its PCB and of restoring the state of another process from its PCB to the hardware. A full context switch impacts a process' performance. The implications of this event are:

- emptying of hardware caches,
- flushing of instruction pipelines, and
- stopping all look-ahead processing.

Common schedulers include the **short term scheduler** and **long term scheduler**. The short term scheduler determines who will execute next: orders the ready queue and is executed frequently. On the other hand, a long term scheduler determines when a process can transition from new to ready: tries to ensure a good mix if CPU-bound and I/O-bound processes.

Less common schedulers include the **dispatcher** and **medium term scheduler**. A dispatcher is the code that actually does context switches: if no dispatcher was present, then the short term scheduler does the context switch. A medium term scheduler monitors the performance of the entire system as a whole and can temporarily remove processes from the ready queue and/or adjust resources seen by the long term scheduler to ensure good system performance. These less common schedulers are not necessary for functionality but are merely a convenience (however, without a dispatcher's functionality, multitasking does not occur at all).

NOTE `main` is not the first thing that is run in C; `__main` is run first which sets up the stack, heap, and then calls the `main()` function the programmer has specified. When `main()` returns, it returns to this former function which tears down the heap and makes a system call to `exit()` which is never meant to come back – this is when we return to the parent process.

Threading

Types of processes include **heavyweight processes**, the standard notion of a process when you start a program, and **threads**, the notion of **concurrency** within a single "program" – multitasking within the notion of one program. The lack of parallelization in the name is because of the history of the term *lightweight process*.

Heavyweight processes are processes that are independent of one another; they possess a private address space, private OS resources, a private process control block, and is scheduled on its own by the operating system. Therefore, it is managed *directly* by the operating system. On the other hand, threads typically exist as a set of related ones and bound up inside a single heavyweight process.

Threading continued on next page...

NOTE In reality, these boundaries may be blurred in practice on modern-day operating systems.

Within one of these processes, the threads share address space, the PCB, resources (like the open file table), are scheduled together along with the heavyweight process, share the interrupt vector, and only have the *basic hardware state* as **private components** which they do NOT share (instruction register, registers, condition code register, and the stack) — they are designed to be unaware of other threads. Each thread will have its own, smaller, process control block stored within the heavyweight process. Therefore, they are often managed in **user-space** (context switches are faster). These are often, therefore, managed by **user libraries**. The operating system does not concern itself with the existence of threads.

This begs the question: *why have multiple types of processes*? Realize that heavyweight processes are considerably slower (there are more things to switch back-and-forth); there are even possible assembly-level instructions that can move registers and restore them (so these can be pretty quick for a thread). The cache does not have to be flushed between threads, either, since that is shared. The same address space is being used. The amount of work being done between context switches between heavyweight processes and threads is significantly different — this could mean more time handling this overhead than running actual code. *Why not run the entire system on threads*? This would mean we would run on the same code base, we are all sharing the same data, and all sharing the same files.

NOTE Since the only thing an operating system knows about are *heavyweight processes*, and an entire process moves into the blocked state, all threads will effectively stop. This is not necessarily the best behaviour and has been fixed.

Therefore, overall, multiple types of processes results in different context switch times, a difference in availability of resources, a dedication of OS resources, two different scheduling systems that have to maintained, possible differences in I/O blocking behaviour, differences in types of communication (communication is really a way of sharing information; the more state that is shared, the easier to communicate — threads are therefore easier at communication with other threads), and protection of the processes.

Typically, the most common way to create **heavyweight processes** involves the system command `fork()`. It creates an identical but *independent* copy of the process – it is the mental equivalent of `clone()`. The only difference is the return value of the `fork()` command; the parent gets the process ID of the child and the child gets a return value of 0.

NOTE Fork-bombing in its naive form (with a simple loop) has been prevented in modern-day processes by setting limits to how many processes a parent and its descendants can create.

What happens after this is that the changing of executable code must take place (using a variant of `exec()`). It allows the child to execute *different* code than the parent. A child's end can be detected with `wait()` or `waitpid()` if the parent process wants to know what happens to it. This is what underlies **shells** and **system start-up**.

Threads are managed and created in user-space with libraries like pthreads (UNIX-based); few, if any, system calls to the OS to manage the threads. All creation, deletion, and waiting is done with function calls, and all threads run from the same executable – cannot swap out the code to run for just one of the threads (no `exec` calls are made here).

Process Communication

What can separate processes do? They can ignore one another: **act independently**. They can also do **serial multitasking**: hand off control between two processes in a pre-determined way (something that looks parallel and made linear — think of the conch from *Lord of the Flies*). Finally, they can **cooperate**: one process can *affect* the other (process execution is not pre-determined). This requires communication and synchronization (IPC). This is useful when certain processes are *specialized* at doing something and can do it well.

Why have cooperating processes? Responsiveness, resource sharing, economy, utilization of **multiprocessor architectures** (two different heavyweight processes can be placed onto different processors or two different threads can be placed onto different cores), and clearer programming models.

Interprocess communication (IPC) is done using different methods: **files**, **shared memory and data structures**, and **named systems** (direct, indirect communication). Files are not the way to deal with real-time process communication, but can be useful for longer-term communication between different processes that are not running synchronously. If two processes share some piece of memory, one process can write data in one part of memory and the other can read from it.

Communication via shared memory raises some new problems: **mutual exclusion** (having only one person accessing the data bus to memory at one time), **atomicity** (the idea of having a set of instructions or actions occurring indivisibly — all at the same time or not at all, e.g. transferring money between accounts), and **critical sections**: a section of non-atomic code that must be executed as if it were atomic so as to avoid confusion. This sort of communication is generally for threads, is efficient (as fast as memory access) and is easily bi-directional.

NOTE The linked list and threading problem shown in class creates a **race condition**: both threads are communicating with the linked list and are both trying to make sense of what they are doing. They are both looking for the end of the list when an element must be added but might be stopped when they find the "end of the list", and when returned to they will no longer be at the end but still consider it to be. Ultimately, this is all a result of uncontrolled **context switches**; this can be solved with serial multitasking.

Communication via named systems is done typically using heavyweight processes. A channel is created to manage communication through **direct communication**. We have a specific way to identify a peer process (addressing, process ID, OS construct; can be brittle if identity changes). Realize that there is an intermediary here: the operating system. Data is sent to a specific destination peer process. This can be symmetric or asymmetric (sender and receiver name each other or only the sender names the receiver — unidirectional). Examples of this are **pipes** and **named pipes** (modelled as stdin and stdout).

NOTE stdin and stdout are modelled as files and processes that access them have a **file table**. When piping, file table entries are rewired together and bridged by a buffer (pipe). When a `pipe()` system call is made, two unidirectional pipes are created (they come in pairs) each pointing a different direction. popen() creates one direct line of communication.

Indirect communication involves the idea of a mailbox and message passing paradigm: the OS keeps a repository of messages to be delivered and anyone can give a message to the mailbox. The mailbox can be shared with other processes — more than two processes can participate (can have many processes send to one mailbox or more than one process read from a single mailbox). Mailboxes generally have a *maximum capacity*; when the mailbox is full, either the sender blocks, overwrites or the message is lost. Different

Process Communication continued on next page...

mailbox systems will have to make a different decision here (this depends on functionality and purpose). This introduces the idea of **message passing paradigms**.

There are two different behaviours that can occur on both sides (the sender and receiver) depending on if the mailbox is empty or full. A mix of non-blocking and blocking implies one-way synchronization, both sides being non-blocking implies asynchronous communication (e.g., networks), and both sides being blocking creates a rendezvous situation (synchronization; useful in time-sensitive situations). A special version of a rendezvous is a **send-receive-reply scenario**: two processes are communicating but are meant to look like a **function call** – what occurs is a *double rendezvous*.

Problems can occur here with message passing. A peer could disappear, messages could be lost, messages could be scrambled, and messages could arrive out of order. This is a matter of **risk control**. With shared memory, there is a low probability of these issues (but scrambling could be a concern). On a shared computer, peers may disappear but the rest are improbable. Finally, across a network, anything can happen (and the risk is very high here; the network has to be designed to deal with this).

When a message is being sent, we can also consider the matters of **buffering** (none, bounded, unbounded), **method of transmission** (pass by copy, pass by reference), and **message size** (fixed size, variable size).

NOTE Assignment 1, *Question 4*: The purpose of this question is to reflect on what we learned about processes and applying it somewhere else. If a process is an aspect of work for an OS, if this is brought to either of the given servers, what is this basic unit of work? Registers are really what differentiate one process from another. *Question 5*: Documentation could include data structures, coordination pieces, and the bigger picture.

Multithreading Models

This involves the mapping of threads to kernel threads. A **kernel thread** is the basic unit of scheduling and process management seen by the kernel. Traditionally, one heavyweight process matches one kernel thread. This can be many-to-one, one-to-one, or many-to-many.

MANY-TO-ONE was the model we have had so far. All threads in one heavyweight process map to a single kernel level thread. All threads share the CPU time allocated to the single kernel level thread. If the heavyweight process, all of its threads block. This can be managed completely in *user space* and is the traditional or original view of threads. This is also fairly easy to implement.

A ONE-TO-ONE model will make the kernel aware of all threads (each thread has its own kernel thread). This breaks the traditional model and allows for *more concurrency*. If one thread blocks, all other remaining threads can continue. This also requires the intervention of the kernel to create threads or do context switches. Furthermore, there is typically a number of threads that can be created. Examples of this sort of system are found in Windows XP and Linux (no distinguishing between heavyweight threads and heavyweight processes).

A MANY-TO-MANY model maps a set of threads to a set of kernel threads (a set of threads shares a set of kernel threads). Can have up to the number of kernel threads block before the whole heavyweight process blocks. This allows one heavyweight process to acess multiple processors at once and we can create as many threads as we want; they just share CPU time. A hybrid model lets you bind a few threads directly to kernel thread while the other share a pool of threads.

In these models, we have to consider the functionality of `fork()`. There are two options here: duplicate *all* threads or only duplicate the thread that called it; this latter functionality works best in a system like Linux. `exec()` changes the executing code (so all threads are affected) and will generally stop *all* threads in

Multithreading Models continued on next page...

the heavyweight process. Now, how about interrupt handling? Which thread handles the interrupt? This can include:

- the thread to which the interrupt applies (like I/O done – easy and typically done),
- all threads,
- threads that asked to see the interrupt (threads can register an interrupt), and
- one specifically designated thread.

This all depends on the environment being intended for the user and the system being aimed for.

Note that, in earlier days, **lightweight processes** were synonymous with threads. In some OSs today, a lightweight process is an *abstraction layer* between a thread and its kernel thread.

Scheduling

The goals of scheduling, inherited from the past, is to **maximize CPU utilization**. Having the CPU idle was a waste: it is expensive. Today, this is not so much an issue — CPU utilization is not always the best thing and an idle CPU is okay. When a certain threshold of CPU power is reached, the luxury to say using the hardware most efficiently is not so much a concern for a general user (the hardware *can* keep up with us now). What has been traded off today is *convenience for the user* (GUIs, networks — these are less expensive than before).

NOTE If a CPU is **idle**, there is nothing typically in the **ready queue**. There is always a background process being run — idle processes exist until a ready queue process is present. The other option is halting the CPU but what can start it back up? In multiprocessor systems, halting other CPUs might occur just for power saving.

NOTE For a monolithic system, priority is not such a concern.

Once this has been maximized, secondary goals exist and are numerous:

- fairness (same CPU cycles),
- throughput,
- predictability (consistency in running times),
- low overhead,
- priorities,
- efficient utilization of other resources held by processes,
- quick response times (for GUIs, especially – want this in interactive situations; might actually choose to have polling interrupts expecting little information each time), and
- graceful degradation.

Note that we cannot satisfy all goals at once. See table on slides.

As spoken about before, the opportunities for context switches are when a process blocks for I/O (preemptive, nonpreemptive), when a process ends (preemptive, nonpreemptive), when a process switches from blocked to ready (preemptive), and when a process' turn on the CPU ends and it returns to the ready state (preemptive).

Scheduling continued on next page...

Performance measures are identified with the performance-based goals. Assuming that a process P_i arrives at time a_i and runs on the CPU for time t_i and enters the exit state at time e_i: we can calculate CPU utilization (time when CPU in use), throughput (number of processes completed per unit of time), turnaround time ($e_i - a_i$), normalized turnaround time (ideally 1), waiting time ($e_i - a_i - t_i$) and response time (time between time slices for the process).

The average value is usually considered. We could consider the maximum measure (how bad could things get), the average (what can we normally expect), and the variance (how predictable is the system).

Assumptions for scheduling for now will include: scheduling for a single CPU, programs cycle between CPU bursts and I/O bursts (view each process as a single CPU burst to schedule); we know the length of the CPU burst for each process for algorithms that rely on the service time. We will track the previous behaviour of the executable and this process' CPU bursts; therefore, we can estimate the next CPU burst size. The latter is a possibly unreasonable assumption but we can make estimations using an exponentially weighted average (the alpha value balances between trusting the past or the present more – a low value of alpha means to consider the past more).

Basic scheduling algorithms include **fixed schedule**, **first come first served** (FCFS), **round robin**, **shorted job first** (SJF), **shortest remaining time first** (SRTF), **priority queue**. At the core of more complex scheduling algorithms will use one or more of these. On the fringe are meta-algorithms or ways to organize multiple algorithms together: multi-level queues and multi-level feedback queue.

When the processes being run are well-known in advance (such as the processor in a car — embedded systems), we can use **fixed schedule**. Dynamic process creation is not occurring very often and therefore a hard-coded schedule can be made. A scheduler would not even be necessary; only a dispatcher or nothing at all. The "scheduler" could be as simple as an infinite loop of function calls.

More often than not, we can consider a **first come first served** scheduler; this is the simplest algorithm and is inherently nonpreemptive (can only be nonpreemptive). A FIFO queue is used for the ready queue. This is both a good and bad scheduler. This can be subjected to **queueing theory** (under some assumptions; predicts the ready queue grows infinitely big as CPU load nears 100%, *asymptotic*) — this impacts throughput and waiting time. Furthermore, this method suffers from the **convoy effect**: one big process would force all of the other ones to pile up behind it (even if it got taken off the CPU, it still had arrived *first*).

Ultimately, this method is good for fairness and has low overhead. However, it is bad for throughput, wait time, predictability, priorities, resource use, turnaround time, and has possible degradation.

Round robin handles processes in their order of arrival into the ready queue (but only allow a limited time on the CPU). The time allowed on the CPU is called the time slice or the time quantum. The choice of this slice size is the tricky part. If it was large, this would be FCFS; a small time slice means that with n processes, each process thinks that it has its own CPU running at $\frac{1}{n}$ of the regular speed. Tiny means more time is being spent doing context switches and the ideal scenario is having each CPU burst fit into just one time slice.

Context switches come into play into calculations in **turnaround time** and **wait time**: $e_i - a_i$ and $e_i - a_i - t_i$ respectively where the increase in the latter increases the normalized turnaround time. Context switch times are folded into the wait time (and is hidden there).

Variants in round robin systems include allowing each process to have a different size of time slice (basing size on priorities) or one can allow a process to appear more than once in the round robin cycle (allows for quicker response time for the repeated processes).

Round robin is inherently *fair*, predictable, has very low overhead, and degrades well (adding in processes merely grows the list by one; $\frac{1}{n} \longrightarrow \frac{1}{n+1}$). However, disadvantages include that it does not handle priorities

Scheduling continued on next page...

well and does not balance resource usage. Asking if this is better than the other one is unfair (preemptive vs. nonpreemptive scheduler).

Shortest job first (SJF) is a *nonpreemptive scheduler*; each process knows the size of its CPU burst and the CPU is given to the process with the shortest "next burst". Of our schedulers, this ordering (along with the other ones in this section) is typically implemented by the short-term scheduler.

This implementation *provably* minimizes average waiting time across *all* nonpreemptive schedules. It is implemented through the use of a priority queue, sorted list, or heap. It suffers from **process starvation**: starvation happens when a process is blocked indefinitely from the CPU and the blocking is out of the process' control. Since a process could come in with a burst size that is large and many small ones are coming in every few time units; that first process is never assigned the CPU.

NOTE The comparison proving how the average waiting time is minimized involves comparing this schedule to any other schedule of the same processes (of minimal total wait time). The first process different in sequence can be compared: the time for it in SJF is $\leq$ the time for the other. If that process in SJF is swapped in in the other schedule ahead, it has a greater total wait time (and contradicts the statement that it has a minimal total wait time).

The advantages for SJF is that there is great throughput, low overhead, minimum wait time, and minimum turnaround time. The disadvantages include process starvation, priorities that have to be managed, predictability is not good, degradation is not graceful (one process can affect the entire flow), and resources prioritized.

Shortest remaining time first (SRTF) is a *preemptive version* of SJF; anytime a process is added to the ready queue, the one with the smallest remaining time first to run is chosen first. The biggest effect here is when a process arrives from the new or blocked state: process blocked on I/O might get access to the CPU immediately after the I/O is done if the predicted CPU burst is small.

Priority scheduling assigns a priority number to each process. High priority corresponds to a high number or a low number (this varies by system). A CPU chooses the process with the highest priority; this can be implemented nonpreemptively or preemptively. Low numbers are often chosen because of integer wraparounds. A priority scheduler is extremely expressive: it can emulate shortest job first, etc. but is more complicated to implement and perform (this has overhead).

NOTE `nice` levels allow programs to offset priorities.

This method also begs questions: on what do we base priority? Who sets the priority? When is the priority set? Can the priority change? What range of numbers are valid and can they grow unboundedly? See slides. Notice that a low priority process can be starved by one or more high priority processes; this can be solved by the notion of **aging**: as it is around longer, it begins to get higher priority. Long-running processes could receive a penalty to their priority or processes waiting in the ready queue have their priorities raised over time; eventually, that process will get run.

When are processes ordered? One notion is that of an **active scheduler**: sorting the processes as each process arrives (can use incremental algorithms since the rest of the list is already sorted). This requires an **immediate cost** of CPU time (but the ready queue is always sorted). The other notion is that of **lazy schedulers**: wait for a few processes to arrive before sorting all of them in one batch. This allows us to *amortize* some of the sorting cost across several entries and in the intermediate time you have one of two conditions: the dispatcher may not get the highest priority process or the dispatcher must look through them (this small wait list) for the highest priority one.

Scheduling continued on next page...

The idea of a **multilevel feedback queue** is the same as that of a **multilevel queue**: classifying processes into different queues and allowing different scheduling algorithms for each queue — with multilevel feedback queue, it can move around between queues on set conditions.

Multiprocessor scheduling comes about in one of two styles: **asymmetric multiprocessing** (AMP) where CPUs are differentiated (often with one "master" assigning work to other CPUs); this can apply to CPUs or specialized hardware like graphics cards, onboard DSP, ... **Symmetric multiprocessing** (SMP) considers all CPUs identical and self-scheduling. The overall goal is **load sharing**: the processes are divided among the processors so that no CPU is unnecessarily idle.

There is also a notion of **processor affinity** in SMP; the inclination of a process to be better run on *one processor* than another. An example is whether a processor has been run on a processor before (so the cache is already loaded). Or, the process could also be specifically assigned to a process.

Another notion is **load balancing**: where an architecture decision chooses whether the SMP share a common ready queue or if each processor has its own ready queue. It is easier to manage with individual ready queues, but with individual queues we must resolve whether to push or pull processes between processors when one processor goes idle. Like in networking, we can talk about *push* and *pull*. When processor A notices an imbalance in the ready queue sizes of the processors, A could move processes from overloaded processors to less busy ones (*push*). This could overload a processor, possibly. Or, when process B becomes idle, it extracts a process from the ready queue of another processor (*pull*).

How do we analyze schedulers? We could consider a deterministic evaluation of sample cases, use queueing theory (statistical analysis of properties), simulation (random process generation, replay of known trace), and implementation (deploy and see how people adapt). This are in ascending order of complexity. The latter simulation is not necessarily ideal because an OS is not an open-loop system where input goes to a control to create a process, is measured, and then outputs something; in a closed-loop system, the measure returns to the control to provide *feedback* (from the user). In a simulation of a trace, this feedback would be present and the same as the time it was previously done. Finally, one could look for proofs of properties (which is generally academic).

Sample Linux Scheduler

In the open source Linux operating system, the ready queue has two **sub-queues**: an active and expired queue. This gives a better notion of fairness between processes: each process has a notion of time slice, priority, or credits available to run. The dispatcher only draws on processes from the active queue and once a process spends all of its credits or time slice, it is moved to the expired queue. Only after the active queue is empty do we replenish the active queue from the expired queue (giving them all "credits" again).

On older systems, when redistributing credits, new credits are calculated on the basis of $0.5\times$ old credits + baseline credits. Processes in the blocked state could accumulate credits while blocked so they got the CPU faster once unblocked.

> **NOTE** An O(1) scheduler came out after the above because of the fact so many processes can run on a single processor. However, it got very messy.

The current common system is known as a **Completely Fair Scheduler** (CFS). The process in the ready queue that has run the least of its time slice gets priority: processes are maintained in a balanced binary tree (rebalancing could be local or done later). The dispatcher therefore takes O(log n) time with n processes in the ready queue. The constant-time factor was not necessary, it was found.

Critical Sections

Synchronization

Synchronization is a critical issue in cooperating sequential processes. A **race condition** can occur when two or more processes access and manipulate the same data concurrently and the outcome depends on the order of access. **Race conditions** are not desirable; proper synchronization asks us to eliminate race conditions: ensuring that only one process at a time manipulates shared data.

Bernstein's Conditions states that the result of two statements S1 and S2 (either single lines or blocks of code) that execute in parallel are independent of the order of their execution if (1) neither of them use a variable whose value is set by the other, and (2) both do not set a common variable. The section of code where it is important if it has to happen in a complete manner (cannot be "half-done") is known as a **critical section**: section of non-atomic code that must be executed indivisibly (as if it were atomic). An atomic action is one that occurs indivisibly.

The anatomy of a critical section is: **entry code** (code run before the critical section to help ensure atomicity), the **critical section** (the code that must be run indivisibly), **exit code** (after the section to allow other threads access to this section) and **remainder code** (code that can be preempted anywher in its execution without affecting the computed outcome).

The objectives of a critical section are: **mutual exclusion**, **progress**, and **bounded waiting time** (*bathroom analogy*). Mutual exclusion is the binary idea of there being at most one process executing in the critical section at any given time. Progress relates to the situation that when there is no process in the critical section and some processes are wishing to enter, then only the processes waiting around decide on who will execute next (cannot be delayed indefinitely). Finally, bounded waiting time is the idea that there is a bound on the number of other processes allowed to enter after a given process has requested entry.

The above assumes that processes run at *relative arbitrary speeds* in arbitrary order and make progress. In particular, if a process enters a critical section it will eventually leave it.

Our objective is to devise a solution to this problem (write entry and exit code to guarantee the objectives) and focus on synchronizing *two processes* (synchronizing more will follow).

```
int who = -1;

// entry code
while (who != -1) do nothing;
who = me;

// critical section
...

// exit code
who = -1;
```

This is formalized as **Peterson's algorithm**; it synchronizes two processes 0 and 1; both processes run identical entry and exit code and knows its process ID and the other process' ID. The presence of a *flag* allows for execution of remainder code gracefully and to have a tiebreaker (the nested while loop) which allows for alternation.

```
int turn; /* Global variable */
Boolean flag[2], both initialized to False; /* Global variable */
```

Synchronization continued on next page...

```
do {
flag[me] = True;
turn = you;
while (flag[you] && turn == you) ;
/* do the critical section here */
flag[me] = False;
/* remainder code goes here */ } while (True);
```

NOTE It is important to realize *context switches* can occur at **any time**.

This satisfies mutual exclusion, progress, and bounded waiting time.

NOTE Assignment 2 has a system which shuttles text (messages) from a ticket granting service to other services. The ticket granter knows what services can or cannot be used by a specific user. Another service will possess passwords (but not have all control) and therefore will act as an authentication thread, letting the ticket granter know a caller is who they say they are.

Can **Peterson's Algorithm** extend to n processes? No; there is too much to test the condition of the while loop; two processes could alternate between themselves and starve a third process from the critical section. To handle multiple processes, we use the **Bakery Algorithm** (e.g., taking out the number from the ticker system). The steps for a given process i:

```
Boolean choosing[n] start at False /* Global variable */ int number[n] start at 0 /* Global variable */
choosing[i] = True;
number[i] = max{number} + 1; choosing[i] = False
for j=1 to n {
   while (choosing[j]) ; /* wait while they are choosing a number */
   /* breaks tie unequivocally by comparing process IDs in terms of Cartesian space */
   while (number[j] != 0 && (number[j], j) < (number[i], i)) ;
}
/* Do the critical section */ number[i] = 0;
```

If the critical section is expensive, there is the possibility of overflow of numbers (e.g., there is always a queue). This algorithm and Peterson's Algorithm are both examples of **spinlocks**; they are solutions to the problem where the process consumes CPU cycles repeatedly polling for entry (also known as busy waiting); note this is present in the OS for small critical sections. This is generally a waste of single processor systems but useful in multiprocessor systems when they are small.

How can this be made more efficient? Turning to the hardware, we can use **interrupt masks**; interrupts could be *disabled* in a **critical section** but this is not a general solution unless it is inside the OS inside (e.g., it trusts its other parts; example: dispatcher). Problems, however, are that this allows **starvation**, control is removed from the OS, is a poor use of CPU if doing I/O in this section, and only works for one processor. This allows for mutual exclusion, progress; bounded waiting time is not provided by this solution and is dependent on the scheduler.

Extra hardware support are specialized assembly level instructions (atomic hardware instruction; **test-and-set**) to simultaneously test a bit/bye for zero status and set the bit/byte to 1. For example:

```
int lock = 0; /* global variable */
while (test_and_set(lock)) ;
```

```
// critical section
lock = 0;
```

Is there room in the hardware for this? This satisfies the first two objectives but not necessarily the last one (bounded waiting time); allows for starvation.

A possibly more viable solution is an **atomic swap**. An atomic instruction that exchanges the value at two memory locations: `tmp = a; a = b; b = tmp;`. This, however, still allows for starvation. Code would then be:

```
int lock = 0; /* global variable */
int newval; /* local variable */
newval = 1;
do { atomic_swap(lock,newval); } while (newval==1);
// critical section
lock = 0;
```

Hardware solutions do not typically avoid starvation; the hardware does not know about processes so it cannot distinguish; this is typically only used by the OS alone to provide a software solution for user processes.

This brings us to **semaphores**. They are an abstract data type that provides a *locking service* for critical sections that could be conceptually viewed as an integer value with two functions: `wait()` and `signal()`. The former requests entry into the critical section and the latter notifies processes that the critical section is being left (some authors use `p()` and `v()`). If the integer is restricted to 0 or 1, this is a **binary semaphore** (mutex, for mutual exclusion). If it is unrestricted, we have a **counting semaphore** and is used for managing resources (interprocess communication) with a finite number of instances.

NOTE Critical sections do not all have to be a single block (all in one place). Given a data structure with some functions `add()`, `delete()` and code in-between them and after that is needed to be atomic — together, all of this code is a critical section without the two functions. This can be achieved with a single semaphore (wait and signal) in both of these two atomic sections.

For critical sections, it is recommended to **minimize the time you spend in a critical section**: other processes are waiting for you to finish. Think about how important concurrency is to your task (see slide). Common problems include forgetting both `wait()` and `signal()`, forgetting the former only (breaks mutual exclusion), forgetting the latter (breaks progress), waiting more than once before signalling (breaks progress), signalling more than once as you leave (breaks mutual exclusion), or reversing their order (breaks mutual exclusion).

There are three ways to generally manage resources in computer science: **prevention**, **avoidance**, and **detection/recovery**: prevent means including constructs to handle critical sections (drawbridges for an intersection), avoidance means setting up rules on usage (like we have done; traffic lights), and detection/recovery is nasty for critical sections; instead, we take advantage of the possible problem: adapt the misuse for synchronization (insurance).

Critical Sections continued on next page...

Granularity

NOTE *For Assignment 3.* Linked lists are useful because they are easy to implement; however, they have to be completely traversed for searching. A **skip list** tries to solve this problem by providing more than one copies of the list allowing for skipping through the list to milestone points (less nodes than each preceding list). In a manner, it is a cheap way of doing fast searches with a linked list by emulating a binary tree. Complexity here is logarithmic.

Another data structure that is interesting in the context of synchronization are **threaded binary trees** where there is an addition node that allows for navigation through the tree as a list (e.g., can iterate through elements as a linked list – can use best of both worlds).

NOTE Fixing a node with a semaphore lock (`granular_list`) will provide a significantly greater amount of concurrency to the lists *but* the locking and unlocking code, which is operating system-dependent (and going into kernel mode, blocking its own critical sections, etc.) adds a greater deal of overhead and will overwhelm the use of all of these semaphores. The code will take a great deal longer. See **granularity**.

Granularity refers to the idea of how a system is broken down into small parts. Sandpaper grading uses a similar concept of coarse-grained (big pieces on their own) *v.* fine-grained (small pieces on their own). **Coarse-grained locking** locks an entire data structure (or big pieces); simple to implement and keeps consistent but decreases concurrency. In databases, this could mean locking entire tables or all of them at once. **Fine-grained locking** creates locks for small pieces of the data structure (designed for high concurrent access) but provides more room to forget a lock or implement incorrectly.

Higher levels of concurrency do not mean high throughput: locking overhead can overwhelm your processing time. Given a process time quantum (time your process will use no matter what as dictated by the operating system), the more semaphore operations performed decreases the amount of user work done in a quantum of time. *Can you always get the pieces you need*? This becomes a problem with finer granularity.

With coarse-grade locking, there will be a lot longer time waiting around for locks to be released; with finer-grain locking, waiting becomes less (this may be an exponential decrease, its shape and position depending on process environment). The best place where time is used most effectively and the time waiting is minimized is somewhere at the minimum of a combined function

Synchronization Problems

Counting semaphores can be used to manage a number of instances of a resource; classic synchronization problems include the **producer / consumer problem**, **bounded buffer problem**, and **reader / writer problem**.

The first problem involves having a separate producer and consumer of items. The producer creates items and places them sequentially in a FIFO queue; the consumer removes items sequentially from the FIFO queue and does something with the items. Problems that can arrive is how efficient the two are relative to each other (their speed), and the shifting of positions of the queue as items are added and removed.

A typical solution would be to track the number of elements in the array, requiring a critical section around the "full" variable, but this is not safe if we have multiple producers or consumers. A better solution is to use a counting semaphore to track the availability of items; a global semaphore "item_ready" is initialized

Synchronization Problems continued on next page...

to 0; this solution does not handle multiple producers or consumers, but it keeps the consumer "behind" the producer without a busy wait for items.

The *bounded buffer problem* is a consideration of the producer and consumer problem where the FIFO queue is now *a bounded buffer*; we do not want the producer to overwrite existing entries. One solution is to go back to tracking the number of full items (add a busy wait int he producer when full is equal to n and where n is the size of the bounded buffer); how can we do better? Pretend we have two problems: producers add and consumers remove; consumers add empty space and producers remove. Use the solution twice!

NOTE A **rendezvous** is the idea that two processes can coordinate and know where each other is, relatively, in their execution sequence.

If we add multiple producers and consumers, access to the front of the queue is now a critical section *among producers* and access to the end of queue is a critical section *among consumers*; there is therefore a worry about starving producers or consumers. FCFS would mean there would be no starvation; shortest job first would allow for starvation. How about a priority system? Priority would be difficult to manage here.

Priority inversion happens when a low priority process holds a semaphore and a high priority process is waiting for that same semaphore: the low priority process cannot get to the CPU to complete the critical section and the high priority process cannot proceed without the semaphore. To fix this, increasing the priority of this low priority means that it would finish what it would do in order to allow the high priority process in.

NOTE A new counting semaphore semantics allows for us to save on CPU cycles (waiting around does nothing).

The *reader / writer problem* is where, given a data structure, one would want to allow concurrent readers of the data structure and ensure that a writer has sole access to the data structure (recall the Condition seen above). Therefore, this can be viewed as a *scheduling problem* between readers and writers; different solutions might allow preference to readers (writers might starve), preference to writers (readers might starve), or a bounded waiting time for all (impacting performance, possibly). Some systems define separate read and write semaphores.

NOTE A solution to the above is discussed in Assignment 3.

NOTE Another question: is it possible to create a system such that neither starve but multiple readers can come in at once. One possible solution is to have two queues (a reader and writer queue) where there is alternation between the two — all readers would be able to come in at once. Or: always let in as many readers as possible if a reader is in *and* only let in one writer at a time.

Monitors

NOTE *Assignment # 4* will have a smaller programming problem; in particular, it will exercise a bit more design. A puzzle will be given to solve (sort of like a jig-saw or Sudoku puzzle); a number of threads will be utilized to try and solve this and must be managed correctly. If a piece is being "looked at" it must be taken into the hand of a thread (by one a single thread) and must be safe.

Monitors continued on next page...

NOTE The midterm will cover to the end of **Semaphores** (not incl. semaphore problems).

In summary, if we want it to be such the case that a critical section become a fundamental part of a programming language such that the language constructs take care of having basic principles of a critical section satisfied: we consider the notion of **monitors**. It can be simplified to the idea of a **module** or **class**: as a language construct, the monitor, which *encapsulates a critical section*, will allow the compiler to automatically acquire and release locks to ensure mutual exclusion. Two things must and will be ensured: (1) that there are no lost locks, and (2) the monitor knows whether or not we are already within a critical section to decide whether or not it is safe to wait for a lock.

Monitors ensure that only *one* thread can be in any part of the monitor at one time. Some languages restrict what you can do inside a monitor; e.g., only monitor variables, local variables, and parameters. Monitors came around before object-oriented programming but it can be treated as a module of sort comprised of critical section(s). If this was done with **semaphores**, a series of `wait()` and `post()` calls would act as boundaries to all critical section(s) comprising the module. With this module, the compiler would put this code in automatically.

NOTE With the example of a `binarytree` problem where an `add()` function could call a `search()` function which has already locked; an example solution to this could involve non-locking and locking search functions. This sort of system is ideal for monitors where a single thread is restricted to the code in the monitor module.

To give us the flexibility of waiting for some event to happen, we add in something called a **condition variable**, which is very similar to a semaphore. This data type has a `wait()` and `notify()` function; wait blocks the current process until the next call to notify (blocking process must release the monitor lock) and notify moves one or more processes currently waiting to the ready state. Notice the large difference here: this is not protecting a critical section. The semantics are different — wait always blocks no matter how many previous notify commands have been issued (and looks only to the future). The semantics of notify could different between languages; one *or* more processes currently waiting could be put back in the ready state. In a semaphore, `signal()` always increments the semaphore by one and a `wait()` checks to see if it positive (which means unlocked).

NOTIFY POLICY. Which process receives it? Which process executes next? The process that did the notify, the notified process (middle ground between the two), or both could be executed next (competition between all other processes; not important that it be coded by the notifier immediately), depending on the policy. The process that runs in the monitor next could be any waiting process or processes waiting on conditions have priority.

NOTE In order to show that semaphores and monitors have the same power, a monitor implementation and a semaphore implementation for an identical problem could be instated.

In Java, monitors are implemented using the `synchronized` keyword; this can be applied to methods or a sequence of statements; applied to a method, the monitor covers all synchronized methods in the same class for the object. For a sequence of statements, the name of the object whose monitor you will use for synchronization must be named. Condition variables and semaphores are found in `java.util.concurrent`; not common to have both conditions and semaphores. Theory budded heads against practice here: original versions of Java only had `synchronized` (monitors). Practice won out: coders wanted semaphores.

Critical Sections continued on next page...

Databases & Recovery

In database terminology, critical sections are generally called **transactions**. These are typically sets of SQL statements that must be done atomically. One of two actions are intended for a transaction: (1) *committed* (instructions done as atomic) or (2) *aborted* not done at all and the system will not try again; any partially-done results are *rolled back*.

The location of a result, in terms of a notion of how committed it can be, can occur in **volatile storage** (does not survive a system crash; e.g., RAM), **nonvolatile storage** (survives a system crash; e.g., disk drive), or **stable storage** (never lost; duplication often involved). Committed transactions are located *out of volatile storage*. The notion of these first two can also occur in *buffers* on a computer: printing to a screen, for instance.

Recovery mechanisms to correct errors, etc. include three different techniques: **log-based recovery**, **checkpoints**, and **serializability**. **Log-based recovery** refers to the notion of, before making any changes, changes are recorded *in a log*; it can be played backward to undo and/or replay any changes. An example of this can occur in the installation of an *operating system*: customization usually occurs after the base installation has occurred. These are all recorded and kept track of.

Checkpoints are the idea of periodically taking *stable snapshots* of the system; in the case of a problem, a previous checkpoint state can be restored (and actions can be re-done from there, presumably in a different order). Think of this as an SVN repository's current state.

Finally, **serializability** ensures *actions are ordered*; this is so their effects are done as if *one transaction* was completed before another one has started. A small proof is maintained the database such that this is ensured; everything is forced and reordered to be linear. From there, the serial nature can be returned to in order to reverse changes.

How does this translate to an operating system? In an operating system, *log-based recovery* could be analogous to **journaling**: any changes made are kept in a log file. This can be seen in the initialization of a drive or file system (a recent advantage are **journaled file systems** in order to track smaller changes which can processed when idle). In theory, this can also be done with actual process work; after each instruction or function call, changes can be recorded — this can build up a great deal of overhead. Modern-day operating systems, however, can handle this in certain situations, such as with `ptrace` in Linux — the foundation of a debugger. Or, it can be used to *fingerprint* processes in order to track possible viruses.

Checkpoints could be analogous to a *system restore* in Windows; checkpoints can be intentionally made and restored to. It is feasible to do this with programs; long-running simulations, which run for a couple of days, could crash. A common thing to do, as programmers, can be to take snapshots of the data structures and throw them into a file. Operating systems could do the same for processes when they go from state to state.

Finally, **serializability** could be analogous to **locking protocols**. Structures could be provided to how locks are acquired and released; *two-phased locking* can then ensure; in the first phase, all locks are acquired (growing phase) and in the second phase, changes are effected and locks are released (shrinking phase). This is especially important for synchronization over a network. Another analogy can be **timestamps**: maintain order of execution of instructions and use timestamps to detect non-serializable instances. This can be useful in distributed systems or across a network (which is difficult; no concept of "same time").