Introduction to Operating Systems

Concurrency

Contents

List of Figures

List of Tables

1 Acknowledgments

As with any Computer Science course, Introduction to Operating Systems is constantly evolving. This evolution does not happen in a vacuum. In addition to many outside resources being consulted, we have been very fortunate to have important contributions from our TAs. Ted Cooper and Jeremiah Peschka have been particularly valuable in improving these lecture notes, as editors and contributors. Ted and Matt Spohrer developed the quiz infrastructure and questions. The quiz questions form the basis of the study questions now included with each set of notes. Ted and Jeremiah have made significant contributions and improvements to the course projects which use the xv6 teaching operating system.

As always, any omissions and/or errors are mine, notifications of such are appreciated.

2 Learning Objectives

- Discuss and explain
  - ▷ Determinism and concurrency
  - ▷ Mutual exclusion
  - ▷ Critical section
  - ▷ Starvation
  - ▷ Deadlock
  - ▷ Livelock
- Compare and contrast three main approaches
  - ▷ Disable interrupts
  - ▷ Atomic instructions
  - ▷ Atomic transactions
- List and explain the conditions necessary and sufficient for deadlock
- List and explain certain approaches for deadlock detection and avoidance
  - ▷ Prevention
  - ▷ Avoidance
  - ▷ Detection
  - ▷ Resource allocation graphs
- Define, explain, and compare
  - ▷ Locks
    - – Spin locks
    - – Wait (blocking) locks
    - – Ordered acquisition
    - – Greedy acquisition
  - ▷ Semaphores
    - – Binary
    - – General (counting)
  - ▷ Condition variables
  - ▷ Monitors
- Describe the issues and explain solutions to deadlock and starvation for
  - ▷ The Dining Philosophers Problem

- ▷ The Producer-Consumer Problem
- ▷ The Readers-Writers Problem

- List and explain more advanced concurrency control approaches
  - ▷ Specialized data structures
  - ▷ Read-modify-update
  - ▷ Read-copy-update
  - ▷ Unix pipes
  - ▷ Messages / Mailboxes
  - ▷ Shared memory
  - ▷ Barriers

3 Key Concepts

3.1 Time Slice

The maximum amount of time a process is allowed to run in the CPU, before an involuntary context switch, is generally called the time slice or quantum. The scheduler is run (at least) once every time slice to choose the next process to run. The length of each time slice can be critical to balancing system performance vs process responsiveness – if the time slice is too short then the scheduler will consume too much processing time, but if the time slice is too long, processes will take longer to respond to input.

An interrupt is scheduled to allow the operating system kernel to switch between processes when their time slices expire, effectively allowing processor time to be shared between a number of tasks, giving the illusion that it is dealing with these tasks in parallel (simultaneously). The operating system which controls such a design is called a multi-tasking system[1].

3.2 Concurrent / Concurrency / Parallelism

For our purposes, the term **concurrent** means "to happen at the same time or appear to happen at the same time" ← remember this definition.

The concept of concurrent computing is frequently confused with the related but distinct concept of parallel computing, although both can be described as "multiple processes executing during the same period of time". In parallel computing, execution occurs at the same physical instant: for example, on separate processors of a multi-processor machine, with the goal of speeding up computations. Parallel computing is impossible on a (one-core) single processor, as only one computation can occur at any instant (during any single clock cycle). By contrast, concurrent computing consists of process lifetimes overlapping, but execution need not happen at the same instant. Structuring software systems as composed of multiple concurrent, communicating parts can be useful for tackling complexity, regardless of whether the parts can be executed in parallel[2].

For example, concurrent processes can be executed on one core by interleaving the execution steps of each process via time-sharing slices[3]: only one process runs at a time, and if it does not complete during its time slice, it is paused, another process begins or resumes, and then later the original process is resumed. In this way, multiple processes are part-way through execution at a single instant, but only one process is being executed at that instant.

[1] See Wikipedia.
[2] See Wikipedia.
[3] The phrase "time slices" may be more common

Concurrency means that two or more threads (lightweight processes or LWPs, or even traditional processes) can be in the middle of executing code at the same time; it could be the same code, it could be different code (see Figure 1). They may or may not be actually executing at the same time, but they are in the middle of it (i.e., one started executing, it was interrupted, and the other one started). Every multitasking operating system has always had numerous concurrent processes, even though only one could be on each CPU at any given time[4].

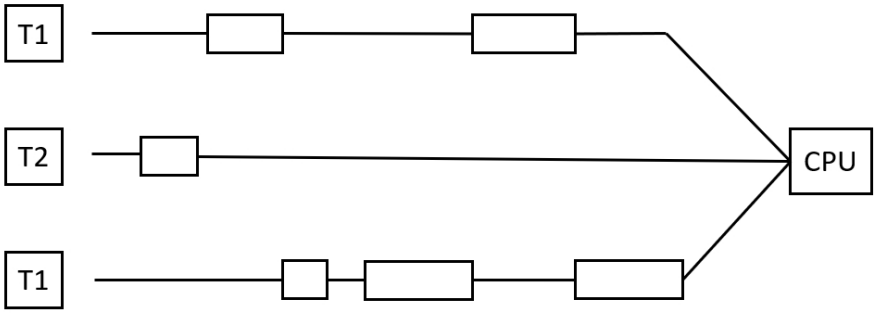

Figure 1: Three Threads Running Concurrently on One CPU

Parallelism means that two or more threads actually run at the same time on different CPUs (see Figure 2). On a multiprocessor machine, many different threads can run in parallel[5]. They are, of course, also running concurrently[6].

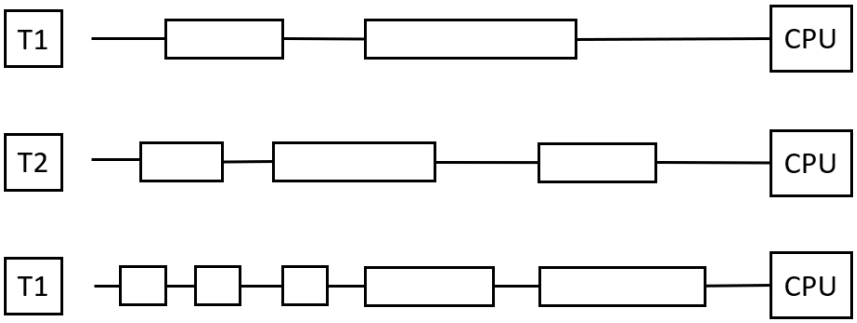

Figure 2: Three Threads Running in Parallel on Three CPUs

The vast majority of timing and synchronization issues in multi-threading (MT) are issues of concurrency, not parallelism. Indeed, the threads model was designed to avoid your ever having to be concerned with the details of parallelism. Running an MT program on a uniprocessor does not simplify your programming problems at all. Running on a multiprocessor doesn't complicate them. This is a good thing. Let us repeat this point. *If your program is correctly written on a uniprocessor, it will run correctly on a multiprocessor.* The probability of running into a race condition is the same on both a uniprocessor and a multiprocessor. If it deadlocks on one, it

[4]PThreads Primer: A Guide to Multithreaded Programming, Bil Lewis and Daniel J. Berg. SunSoft Press, 1996. ISBN 0-13-443698-9.

[5]The degree of parallelism is limited by the number of processors or cores.

[6]The degree of concurrency is *not* limited by the number of processors or cores.

will deadlock on the other. (There are lots of weird little exceptions to the probability part, but you'd have to try hard to make them appear.) A buggy program, however, may run as (naïvely) expected on a uniprocessor, and only show its problems on a multiprocessor[7].

3.3 Mutual Exclusion

Mutual exclusion means that only one thread of control can be in a critical section at a time. That is, the resource must not be accessed concurrently by more than one thread of control.

The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control access to a shared resource when access must be exclusive while the thread is doing its work? The mutual-exclusion solution to this makes the shared resource available only while the thread is in a specific code segment called the *critical section*. It controls access to the shared resource by controlling each mutual execution of all parts of its program where the resource would be used.

Hardware solutions

On uniprocessor systems, the simplest solution to achieve mutual exclusion is to disable interrupts during a critical section. This will prevent any interrupt service routines from running (effectively preventing a process from being preempted). Although this solution is effective, it leads to many problems. If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt is no longer serviced, so tracking time is impossible during the critical section. Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more elegant method for achieving mutual exclusion is the busy-wait.

Busy-waiting is effective for both uniprocessor and multiprocessor systems. The use of shared memory and an atomic test-and-set instruction provide the mutual exclusion. A process can test-and-set on a location in shared memory, and since the operation is atomic, only one process can set the flag at a time. Any process that is unsuccessful in setting the flag can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function?even if a process halts while holding the lock.

Several other atomic operations can be used to provide mutual exclusion of data structures; most notable of these is compare-and-swap (CAS). CAS can be used to achieve wait-free mutual exclusion for any shared data structure by creating a linked list where each node represents the desired operation to be performed. CAS is then used to change the pointers in the linked list during the insertion of a new node. Only one process can be successful in its CAS; all other processes attempting to add a node at the same time will have to try again. Each process can then keep a local copy of the data structure, and upon traversing the linked list, can perform each operation from the list on its local copy.

Software solutions

In addition to hardware-supported solutions, some software solutions exist that use busy waiting to achieve mutual exclusion. Examples[8] include:

[7] Ibid.

[8] These algorithms do not work if out-of-order execution is used on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread.

- Dekker's algorithm
- Peterson's algorithm
- Lamport's bakery algorithm
- Szymanski's algorithm
- Taubenfeld's black-white bakery algorithm
- Maekawa's algorithm

It is often preferable to use synchronization facilities provided by a multithreading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist. For example, when a lock library is used and a thread tries to acquire an already acquired lock, the system could suspend the thread using a context switch and swap it out with another thread that is ready to be run, or could put that processor into a low power state if there is no other thread that can be run. Therefore, most modern mutual exclusion methods attempt to reduce latency and busy-waits by using queuing and context switches. However, if the time that is spent suspending a thread and then restoring it can be proven to be always more than the time that must be waited for a thread to become ready to run after being blocked in a particular situation, then spinlocks are an acceptable solution (for that situation only)[9].

3.4 Race Condition

A race condition, data race, or just race, occurs when more than one thread of control tries to access a shared data item without using proper concurrency control. The code regions accessing the shared data item are collectively known as the critical section for that data item.

An example of a data race is how `i++` can be unsafe in a concurrent environment, see §4.1. Without proper concurrency control, there is potential for state corruption within a set of assembly instructions that we need to be executed as a single unit. This is often because of inconvenient interrupts – including the malicious scheduler 😈 – or the latency of the time of check to time of use (TOCTOU) in a critical section.

3.5 Atomic Operations

A race condition exists with `i++` because it is comprised of more than one assembly operation, one of which is a **modify operation**. Strangely enough, it can be comprised entirely of atomic operations but is not, itself, atomic.

Most processor architectures and compilers provide some support for atomicity. In general, the compiler will attempt to utilize the best approach for the code being compiled and may also offer features that extend *beyond* what is provided by the atomic operations of the target processor. Knowledge of atomic operations for different processors is important for understanding when you can rely on the atomic operations of your target processor. Be very careful to make no assumptions about the atomicity of operations for your target processor – RTRM (read the reference manual) as there are often subtle issues if the instruction is not used precisely as intended.

[9] See Wikipedia.

> [...] During an atomic operation, a processor can read and write a location during the same data transmission. In this way, another input/output mechanism or processor cannot perform memory reading or writing tasks until the atomic operation has finished.
>
> [...] The problem comes when two operations running in parallel (concurrent operations) utilise the same data and a disparity between the results of the operations occurs. Locking locks variable data and forces sequential operation of atomic processes that utilize the same data or affect it in some way[10].

3.6 Critical Section

A critical section for a shared variable is the set of all code that accesses that specific shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread. Even code that does not modify the shared variable are in the set that comprises the critical section.

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section or critical region. It cannot be executed by more than one process at a time. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses[11].

3.7 Deadlock

3.7.1 Four Conditions Necessary and Sufficient for Deadlock

1. Mutual exclusion. Only one thread of control can use the resource at any one time.
2. Hold and Wait. A thread of control currently holding one exclusive resource is allowed to request another exclusive resource.
3. No preemption. Once a thread of control holds a resource, it cannot be taken away. The held resource must be voluntarily given up by the thread of control.
4. Circular wait. A cycle in the resource allocation graph must exist. Each thread of control must be waiting for a resource which is being held by another thread of control, which in turn is waiting for the first thread of control to release the resource. See Fig. 3.

[10] See Technopedia.
[11] See Wikipedia.

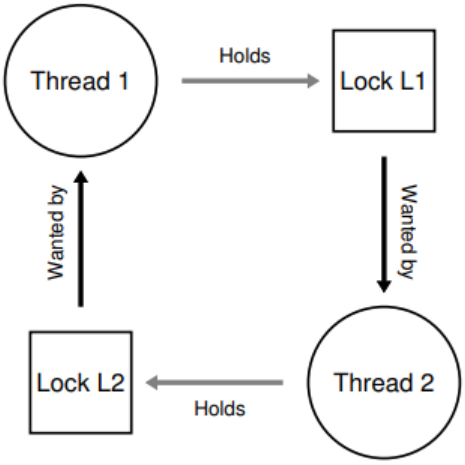

Figure 3: Resource Allocation Graph Cycle

A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the same set. The key point here is that processes are all in the blocked state, but only a process in that set can cause a process to move out of the blocked state. A process must be in the running state to be able to perform this action – something clearly impossible.

3.7.2 Deadlock Prevention

We can prevent Deadlock by eliminating, also termed "violating", any of the four conditions.

- Eliminate Mutual Exclusion. Rewriting the code to remove the need for mutual exclusion is often possible. Changing a data structure or algorithm may be required. Also, mutual exclusion is not always possible since some resources, such as a DVD or printer, are inherently non-shareable.
- Eliminate Hold and Wait. Allocate all required resources to the process before the start of its execution, the hold and wait condition is eliminated but it will lead to low device utilization. For example, if a process requires printer at a later time and we have allocated the printer before the start of the execution of the process, printer will remain unavailable to other processes until the current process has completed its execution. The process can make a new request for resources after releasing all of the currently held resources. This solution may lead to starvation.
- Eliminate No Preemption. Take resources away from the process when the resource are required by other, higher priority, processes. Requires the imposition of priorities, which, if not done carefully, can lead to *priority inversion*.
- Eliminate Circular Wait. Each resource can be assigned a number. A process can request the resources in a strict increasing/decreasing order of numbering. For example, in a system with strict ordering, it may be that only *higher numbered* resources can be requested. So if a process holds a resource set R = {R1, R2, R5, R12} and a request is made for resource R15, then the request can be satisfied. However, a request for resource R10 cannot be satisfied and must be denied.

3.7.3 Deadlock Avoidance

Deadlock avoidance[12] can be done with the banker's algorithm, as well as other approaches.

Banker's Algorithm The banker's algorithm[13] is a resource allocation and deadlock avoidance algorithm that test all the requests made by processes for resources. The algorithm checks for the safe state; if after granting request system remains in the safe state it allows the request and if there is no safe state it denies the request made by the process. This approach is termed "admission control".

Inputs to Banker's Algorithm:

1. Maximum need of resources by each process.
2. Currently allocated resources by each process.
3. Maximum free available resources in the system.

A request will only be granted under these conditions:

1. If the request made by the process is less than equal to the maximum needed to allow that process to run and eventually free all resources held.
2. If the request made by the process is less than equal to the freely available resource in the system.

[12] See GeeksForGeeks.
[13] See Wikipedia.

Total resources

A	B	C	D
6	5	7	6

Available resources

A	B	C	D
3	1	1	2

Currently allocated

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Maximum resources
(per process)

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Needed resources
(need = maximum – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Figure 4: Banker's Algorithm

1. Give an example of schedule that can be safely executed. This means that all processes can eventually make progress without resource preemption. No deadlock on the resources.

2. Give an example of a schedule that cannot be safely executed. This means that there is a path, which if executed, results in deadlock on the resources.

Lock Ordering One way to prevent deadlock[14] is to put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order.

For an example based on the Harry Potter books, let's assume that we want to track Wizard information. We might always acquire the locks on Wizard objects in alphabetical order by the wizard's name. Thread A and thread B are both going to need the locks for Harry and Snape, they both acquire them in that order: Harry's lock first, then Snape's. If thread A gets Harry's

[14]See 6.005 – Software Construction at MIT.

lock before B does, it will also get Snape's lock before B does, because B can't proceed until A releases Harry's lock again. The ordering on the locks forces an *ordering* on the threads acquiring them, so there's no way to produce a cycle in the resource allocation graph.

Here's what the code might look like:

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

Note that the decision to order the locks alphabetically by the person's name would work fine for this book, but it wouldn't work in a real life social network. Why not? What would be better to use for lock ordering than the name?

Although lock ordering is useful (particularly in code like operating system kernels), it has drawbacks.

- Not modular – the code has to know about all the locks in the system, or at least in its subsystem.
- It may be difficult or impossible for the code to know exactly which of those locks it will need before it even acquires the first one. It may need to do some computation to figure it out. Think about doing a depth-first search on the social network graph, for example – how would you know which nodes need to be locked, before you have even started looking for them?

3.7.4 Prevention v. Avoidance

Factors	Deadlock Prevention	Deadlock Avoidance
Concept	Blocks at least one of the conditions necessary for deadlock to occur.	Ensures that system does not go into an unsafe state.
Resource Request	All in advance (a priori).	Permit/deny based on an available safe path.
Information required	Does not requires information about existing resources, available resources and resource requests.	Requires information about existing resources, available resources and resource requests.
Procedure	Prevents deadlock by constraining resource request process and handling of resources.	Automatically considers requests and check whether it is safe for system or not. Think "graph traversal".
Preemption	Sometimes, preemption occurs more frequently.	No preemption.
Allocation strategy	Conservative.	Optimistic.
Future requests	No knowledge of future resource requests.	Requires knowledge of future resource requests.
Advantage	No overhead. Just make one of the conditions false. Potential for resource under-utilization.	Overhead. Dynamic algorithm must be executed at each allocation/deallocation. No resource under-utilization.
Disadvantage	Low device utilization.	Can block processes excessively.
Example	Spooling and non-blocking synchronization algorithms.	Banker's algorithm.

Table 1: Deadlock: Prevention vs Avoidance

3.7.5 Deadlock Detection and Recovery

Resource types

- Single instance resources. Check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is a sufficient condition for deadlock.
- Multiple instance resources. Detection of a cycle is necessary but not sufficient condition for deadlock detection. The system may or may not be in deadlock; varies according to different situations. Consumable and non-consumable becomes important.

Recovery can be tricky. The options are limited.

- Kill processes. Terminate (potentially) all processes involved in the deadlock. Terminate processes one by one. Order of termination can be determined algorithmically. After terminating each process, check for deadlock again and keep repeating until system recovers.
- Resource preemption. Resources are taken from one or more processes. Preempted resources are allocated to other processes in the hope that deadlock can be removed. Very tricky to get right.

3.7.6 Resource Allocation Graphs

A resource allocation graph is very powerful. Simple algorithms exist to determine if a graph is cyclic or acyclic. If the resource graph is acyclic, then there cannot be deadlock.

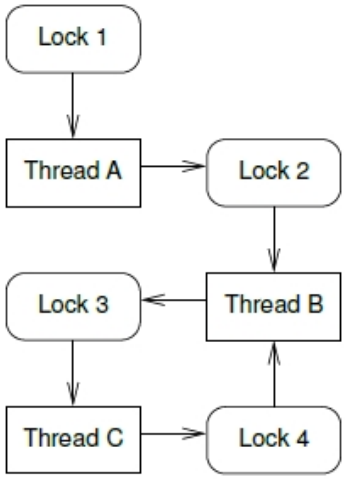

Figure 5: Resource Allocation Graph Showing Deadlock

3.8 Starvation

Starvation is a problem encountered in concurrent computing where a process is perpetually denied access to necessary resources. Starvation may be caused by errors in mutual exclusion algorithm, or the result of the scheduler, but can also be caused by resource leaks, and can be intentionally caused via a denial-of-service attack such as a fork bomb.

Starvation is normally caused by deadlock in that it causes a process to freeze. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set. On the other hand, a process is in starvation when it is waiting for a resource that is continuously denied. Starvation-freedom is a stronger guarantee than the absence of deadlock: a mutual exclusion algorithm that must choose to allow one of two processes into a critical section and picks one arbitrarily is deadlock-free, but not starvation-free.

A possible solution to starvation is to use a scheduling algorithm with priority queue that also uses the aging technique. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time[15]. The Multi-Level Feedback Queue scheduler (MLFQ) is one example of a scheduler that uses aging to prevent starvation. MLFQ will be studied later.

3.9 Livelock

Livelock occurs when two or more threads continually repeat the same interaction in response to changes in the other threads without doing any useful work. These threads are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all threads are in the waiting state.

Livelock is very difficult to detect, unlike deadlock. Based on state transitions, threads involved will appear to make progress. There can be almost no outward sign of livelock.

[15] See Wikipedia.

Perpetually spinning on a lock is an example of livelock.

Livelock Example

```
MutexLock Mutex1, Mutex2;

//thread 1
Mutex1.Lock();
while (Mutex2.isLocked()) {
    Mutex1.Unlock();
        // wait for a while
    Mutex1.Lock();
}
Mutex2.Lock(); // lock Mutex2. We have both locks now

// thread 2
Mutex2.Lock();
while (Mutex1.isLocked()) {
    Mutex2.Unlock();
    // wait for a while
    Mutex2.Lock();
}
Mutex1.Lock();
```

- Thread 1 locks Mutex1 first. If Mutex2 is not locked, thread 1 acquires it. Otherwise, thread 1 yields Mutex1, waits for a while (for thread 2 to take Mutex1 and finish its task), reacquires Mutex1, and checks again Mutex2 is open.
- Thread 2 does this sequence the same way with the role of Mutex1 and Mutex2 switched.

To avoid this type of livelock, order the locking sequence (**lock ordering**), hierarchically. Thus, only one thread can lock both locks successfully. This can be quite difficult in practice.

4 Determinism

A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently[16].

In a *deterministic algorithm*, for a given particular input, the computer will always produce the same output going through the same states but in case of *non-deterministic algorithm*, for the same input, the algorithm may produce different output in different runs. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it[17].

[16] See ComputerScienceWiki.
[17] See GeeksforGeeks.

4.1 A Simple Example

A very simple C expression is:

```
i++;
```

This is simply a single C Language expression that increments the value in the integer variable `i`. We need to know that, while this is a single C Language statement, it does not map to a single assembly language statement. Because is maps to more than one `fetch` → `decode` → `execute` → `check for interrupt` cycle, we have to worry about concurrency.

If the process is single-threaded, we have no concerns because the variable i is not shared. However, if we have more than one thread of control and the variable i is shared, then we have a problem. Let's see how this occurs.

The C language expression "i++" (or "++i") compiles to at least 3 assembly language instructions

```
load from <memory address of i> to <register>
increment <register>
store from <register> to <memory address of i>
```

In terms of how these instructions get executed, our model yields this result for i++

(fetch load instruction)
→ (decode load instruction)
→ (execute load instruction)
→ (**check for interrupt**)
→ (fetch increment instruction)
→ (decode increment instruction)
→ (execute increment instruction)
→ (**check for interrupt**)
→ (fetch store instruction)
→ (decode store instruction)
→ (execute store instruction)
→ (**check for interrupt**)
→ (continue *ad infinitum*)

So there are three places where this sequence of instructions can be interrupted, although only 2 impact our example. Let's see how this could all go wrong with two threads executing at the same time.

Let's begin with the variable i having the value "3".

Time	Thread 1	Thread 2
↓	load i into register	
	$R_1 = 3$	
	incr register value	
	$R_1 = 4$	
	interrupt!	
		load i into register
		$R_2 = 3$
		incr register value
		$R_2 = 4$
		store register value at variable i
		$i = 4$
	interrupt!	
	store register value at variable i	
	$i = 4$	

Table 2: With Interrupted Code Sequence

Incrementing variable i twice should result in the final value being $i = i + 2$ not $i = i + 1$! This execution is termed nondeterministic because the outcome of running the code cannot be determined from straightforward code inspection.

The number of possible orderings of the critical section is $\mathcal{O}(n^2)$, or N^2, where N is the number of threads. We will use two threads in our examples as they suffice to see the issue clearly.

Table 2 shows one possible ordering of this code. This ordering results in non-deterministic behavior.

The result that we would prefer is one where either the code section for one thread runs to completion first and then the other thread runs to completion.

Let's begin with the variable i having the value "3".

Time	Thread 1	Thread 2
↓	load i into register	
	$R_1 = 3$	
	incr register value	
	$R_1 = 4$	
	store register value at variable i	
	$i = 4$	
		load i into register
		$R_2 = 4$
		incr register value
		$R_2 = 5$
		store register value at variable i
		$i = 5$

Table 3: Without Interrupted Code Sequence

Now, the result of incrementing the variable i twice is now $i = i + 2$. This is *one possible* execution order for this code. This code gives us the answer we expect, but there is no *guarantee* that the code will be executed in this order every time.

4.2 Another Example

The OSTEP text has an extended example in Chapter 26, *Concurrency: An Introduction.* The example is long, but well worth looking at closely.

> What we have demonstrated here is called a race (or, more specifically, a **data race**): the results depend on the timing execution of the code. With some bad luck (i.e., a context switch that occurs at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice deterministic computation (which we are used to from computers), we call this result **indeterminate**[18], where it is not known what the output will be and it is indeed likely to be different across runs.
>
> Because multiple threads executing this code can result in a race condition, we call this code a critical section. What we really want for this code is mutual exclusion. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so[19].

> When we say that a lock protects data, we really mean that the lock protects one or more invariant that applies to the data.

[18] In CS333, we will use the term nondeterministic

[19] OSTEP, Ch 26, pp 9-10

5 Atomic Operations

5.1 Intel Atomic Operations

The 32-bit IA-32 processors support locked atomic operations[20] on locations in system memory. These operations are typically used to manage shared data structures in which two or more processors may try to modify concurrently.

Certain basic memory transactions (such as reading or writing a byte in system memory) are always guaranteed to be handled atomically. That is, once started, the processor guarantees that the operation will be completed before another processor or bus agent is allowed access to the same memory location. The processor also supports bus locking for performing selected memory operations (such as a read-modify-write operation in a shared area of memory) that typically need to be handled atomically, but are not automatically handled this way. Because frequently used memory locations are often cached in a processor's L1 or L2 caches, atomic operations can often be carried out inside a processor's caches without asserting the bus lock. Here the processor's cache coherency protocols ensure that other processors that are caching the same memory locations are managed properly while atomic operations are performed on cached memory locations.

Where there are contested lock accesses, software may need to implement algorithms that ensure fair access to resources in order to prevent lock starvation. The hardware provides no resource that guarantees fairness to participating agents. It is the responsibility of software to manage the fairness of semaphores and exclusive locking functions.

Intel Guaranteed Atomic Operations: The 486 and newer processors guarantee that the following basic memory operations will **always** be carried out atomically:

- Reading or writing a byte
- Reading or writing a word aligned on a 16-bit boundary
- Reading or writing a doubleword aligned on a 32-bit boundary

The Pentium processor (and newer processors since) guarantees that the following additional memory operations will always be carried out atomically:

- Reading or writing a quadword aligned on a 64-bit boundary
- 16-bit accesses to *uncached* memory locations that fit within a 32-bit data bus

The P6 and newer processors guarantee that unaligned 16-, 32-, and 64-bit accesses to cached memory that fit within a *single cache line* will be carried out atomically. Modern Intel systems all use a 64-byte cache line size for L1 data cache, the L2 and L3 unified caches; the L1 instruction cache uses a 32-byte cache line size.

5.2 GCC Atomics

GCC has implemented atomic operations designed to support the C++ memory model and can be used with standard C programs compiled with the gcc compiler. These functions are intended to replace the legacy '__sync' builtins. The memory modes supported can be found at the Wikipedia entry for Memory model synchronization modes.

[20] See volume 3, section 8.1.1, of the Intel 64 and IA-32 Architectures Software Developer Manuals.

5.3 LLVM Atomics

Some languages, such as the Rust programming language, use the LLVM compiler infrastructure. The LLVM Atomic Instructions and Concurrency Guide discusses instructions which are well-defined in the presence of threads and asynchronous signals. The Rust compiler will automatically make uses of these features or you can call out to a routine[21] if you have specific requirement not native to Rust. Not all features of the LLVM compiler infrastructure are utilized in the Rust programming language; however, searching crates.io may yield a crate that provides the necessary feature.

5.4 An xv6 Example

In `sysproc.c`, the implementation for `getpid()` takes advantage of an Intel guaranteed atomic operation:

```
int
sys_getpid(void)
{
    return myproc()->pid;
}
```

In this code, the routine `myproc()` will return a pointer to the `proc` structure for the currently executing process and then will *atomically read* the `pid` field, which is then returned. Note that the entire line `return myproc()->pid;` is not atomic. However, the read of the `pid` field is an atomic operation, so there is no need to hold the `ptable` lock.

The `myproc()` routine disables interrupts. Interrupts are not disabled here as a form of concurrency control. They are disabled in order to ensure that an interrupt does not remove the current process from the CPU before a pointer to the `proc` struct is obtained. There is a potential race[22] between the call to `mycpu()` and the assignment to variable `p` that disabling interrupts prevents. If interrupts were not disabled, the process *could* be removed from the current CPU and then allocated a different CPU while executing the routine. Consider the possible issue if interrupts were not disabled.

```
// Disable interrupts so that we are not rescheduled
// while reading proc from the cpu structure
struct proc*
myproc(void) {
    struct cpu *c;
    struct proc *p;
    pushcli();
    c = mycpu();
    p = c->proc;
    popcli();
    return p;
}
```

[21] Rust can call out to C or C++, as described in the Embedded Rust Book in the section titled, A little C with your Rust.

[22] Note that this is not a race involving shared data.

Similarly, the `ticks` global variable can be read atomically even though it may be modified (written to) in `trap.c`.

```
int
sys_uptime(void)
{
    uint xticks;

    xticks = ticks;  // Why use a local variable?
    return xticks;
}
```

Code in `trap()` makes use of an Intel atomic increment instruction (see `x86.h`) to ensure that writing to the `ticks` global variable is atomic.

```
    atom_inc((int *)&ticks);
```

Also, a gcc attribute is used

```
    uint ticks __attribute__ ((aligned (4)));
```

to ensure that the `ticks` global variable is properly aligned on a 32-bit boundary (see above).

The handling of the `ticks` variable demonstrates two different approaches, one for atomic read and one for atomic increment.

6 Atomic Transactions

An atomic[23] transaction is an *indivisible* and *irreducible* series of two or more operations such that either all occur as a single unit or no part of the transaction occurs. When a transaction *aborts* it is said to *roll back* to the state before the start of the transaction, making it appear that the transaction was not executed. Note that this is with respect to the execution of the transaction, not the overall state of the system, which may be concurrent.

A guarantee of atomicity prevents other threads of control from seeing updates occurring only partially. An atomic transaction cannot be observed to be in progress by another thread of control. At one moment in time, it has not yet happened, and at the next moment it has already occurred or nothing happened if the transaction was canceled (*rollback*). That is, to all other threads of control, the operations of the atomic transaction appear to be *instantaneous*.

Atomic transactions can still be interrupted by the scheduler or other portions of the system, but the transaction will still be atomic with respect to all other threads of control. The processor instruction set must provide atomic operations such as test-and-set, fetch-and-add, compare-and-swap, load-link, or store-conditional, perhaps coupled with memory barriers[24]. A memory barrier, also known as a membar, memory fence or fence instruction, is an instruction that causes the CPU, or compiler, to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier[25].

Projects 3 and 4 will require that students correctly implement atomic transactions.

[23] from the Greek $\alpha\tau o\mu o\nu$, atomon ("indivisible").

[24] See Wikipedia.

[25] Compare with *out-of-order execution*.

A Note on the Phrase "without interruption"

Many sources will state that an atomic operation or transaction occurs *without interruption*.

Be careful not to confuse *interruption* with *processor interrupt*. Processor interrupts are allowed to happen during atomic operations and instructions, but no thread of control outside of the one performing the action is allowed to "see" the intermediate state of the action. In particular, without interruption does not imply that a context switch cannot happen.

For this class, it is preferable to say that *an atomic transaction executes as a single unit with no intermediate state visible to any other thread of control*.

6.1 Abort / Rollback

A rollback is an operation, or set of operations, that returns the system to some known good previous state. For an atomic transaction, it is used to return the system to the state *before* the transaction was started. In this way, abort/rollback of a transaction results in a state where it appears that the transaction has *not to have been executed at all*.

6.2 Checkpointing

It is possible for a system to periodically save some or all of its state, periodically. This save point is called a *checkpoint*. A transaction could choose to checkpoint several times during its execution so that a partial rollback can be accomplished without aborting the entire transaction. There are many potential issues with a partial rollback, so the programmer must be very careful to avoid deadlock, livelock, and/or starvation.

7 Locks

The most basic concurrency control primitive is the mutex lock, usually just called lock or mutex. A lock is a common synchronization primitive used in concurrent programming.

There are two types of locks, spinlocks and wait locks. A wait lock is also called a *blocking lock* but is most often implemented as a binary semaphore which we will discuss later. A spinlock uses CPU time to check on the condition of the lock; continuing to check a "locked" until the lock becomes free. A wait lock uses a queue that causes the thread of control to wait (enter the blocked state) without using additional CPU time. Both lock types have strengths and weaknesses. More later.

Generally, locks are *advisory locks*, where each thread *agrees to voluntarily acquire* the lock before accessing the corresponding data. Some systems implement *mandatory locks*, where attempting to access a locked resource, without holding the corresponding lock, will force an exception in the thread of control attempting to make the access. In general, the term "lock" in this class will mean *advisory lock*.

At any one time, a lock can be in one and only one of two states.

- Locked. The lock is *exclusively held* by one thread of control and may not be acquired by another thread of control until the lock is released.
- Unlocked. The lock is not held by any thread of control and may be acquired.

We know that it takes a non-zero amount of time to change from one state to the other, so we know that there must be intermediate states (e.g., partially locked, nearly unlocked) but these intermediate states cannot be detected by any outside observer.

7.1 Hardware Support

We require hardware support to build a lock. The reason is the instruction cycle.

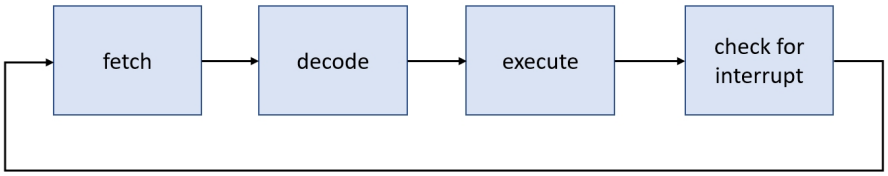

Figure 6: CPU Execution Cycle

We need the ability to acquire a lock without a CPU interrupt occurring since the interrupt may cause the scheduler to run. The only general way to accomplish this is with an atomic assembly instruction. The instruction set architects provide the assembly instruction set, so we need them to create instructions for this purpose. This is one example of the close cooperation between OS developers and processor architects. A software engineer cannot *implement* a working lock without at least one atomic instruction; although a software engineer can certainly *design* a lock.

Without an atomic instruction, it is impossible to acquire a lock without the possibility of an interrupt occurring (see figure 6).

There are several common types of atomic instructions that can be used to build a lock.

1. TSL – Test and Set Lock[26]. The test-and-set instruction writes 1 (set) to a memory location and returns its old value as a single atomic operation.

```
#define LOCKED 1

int TestAndSet(int* lockPtr) {
    int oldValue;

 / -- Start of atomic segment --
    // This should be interpreted as pseudocode for illustrative purposes only.
    // Traditional compilation of this code will not guarantee atomicity, the
    // use of shared memory (i.e., non-cached values), protection from compiler
    // optimizations, or other required properties.
```

[26]See Wikipedia

```
    oldValue = *lockPtr;
    *lockPtr = LOCKED;
// -- End of atomic segment --

    return oldValue;
}
```

Use as a spinlock:

```
volatile int lock = 0;

void Critical() {
    while (TestAndSet(&lock) == 1)
        ;
    critical section // only one process can be in this section at a time
     lock = 0 // release lock when finished with the critical section
}
```

2. XCHG – Exchange Register/Memory with Register

Use as a spinlock:

```
// The xchg is atomic.
while(xchg(&lk->locked, 1) != 0)
    ;
```

xv6 inline assembly using `xchg` atomic instruction[a].

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
    uint result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    return result;
}
```

[a] from `spinlock.c`

3. CAS – Compare and Swap[27]
   Pseudo code:

[27] See Wikipedia

```
int compare_and_swap(int* reg, int oldval, int newval)
{
    ATOMIC();
        int old_reg_val = *reg;
        if (old_reg_val == oldval)
        *reg = newval;
    END_ATOMIC();
    return old_reg_val;   // success
}
```

Example usage pseudo code:

```
int
function add(int *p, int a)
{
     done ← false
     while not done {
          value ← *p
          done ← cas(p, value, value + a)
     }
     return value + a
}
```

7.2 Spinning vs Blocking

Advantages:

- Spinning. A spinlock is conceptually simple and easy to implement correctly.
- Blocking. Usually has better resource utilization.

Disadvantages:

- Spinning. Potentially wastes resources such as CPU time.
- Blocking. Complexity of the implementation; managing the wait list.

When would you use spinning instead of blocking?

- Single CPU System: Never.

  This is because, on a single CPU system, the spinning thread will use its entire time slice each time that it is scheduled. When the spinning thread is running, the thread holding the lock is delayed unnecessarily. If it cannot run, it cannot complete its critical section and release the lock. This implies that spinning increases the delay before the spinning thread will acquire the lock. If the thread instead blocked, the thread holding the lock will be able to finish the critical section and release the lock in a much shorter period of time.

  On a single CPU system, spinning guarantees worst case behavior.

- Multi CPU System: When the anticipated time for spinning is less than the latency of waiting, use a spinlock.

  This is because the spinning thread can be running on one CPU while the thread holding the lock can be running on a separate CPU and therefore free the lock without being delayed by the spinning thread. If the time spent spinning will be less than the time for waiting, use a spinlock. Once the number of threads exceeds the number of CPUs, however, the decision is more difficult.

  Hard and fast rules are always tricky, but this is our advice.

7.3 Reader-Writer Locks

An reader-writer (RW) lock allows concurrent access for read-only operations, while write operations require exclusive access. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data. When a writer is writing the data, all other writers or readers will be blocked until the writer is finished writing. A common use might be to control access for a data structure in memory that cannot be updated atomically and is invalid (and should not be read by another thread) until the update is complete, e.g., implement as a transaction.

RW locks can be designed with different priority policies for reader vs. writer access. The lock can either be designed to always give priority to readers (read-preferring), to always give priority to writers (write-preferring) or be unspecified with regards to priority. These policies lead to different tradeoffs with regards to concurrency and starvation.

- *Read-preferring RW locks* allow for maximum concurrency, but **can lead to write starvation** if contention is high. This is because writer threads will not be able to acquire the lock as long as at least one reading thread holds it. Since multiple reader threads may hold the lock at once, this means that a writer thread may continue waiting for the lock while new reader threads are able to acquire the lock, even to the point where the writer may still be waiting after all of the readers which were holding the lock when it first attempted to acquire it have released the lock. Priority to readers may be weak, as just described, or strong, meaning that whenever a writer releases the lock, any blocking readers always acquire it next.

- *Write-preferring RW locks* **avoid the problem of writer starvation** by preventing any new readers from acquiring the lock if there is a writer queued and waiting for the lock; the writer will acquire the lock as soon as all readers which were already holding the lock have completed. The downside is that write-preferring locks allows for less concurrency in the presence of writer threads, compared to read-preferring RW locks. Also the lock is less performant because each operation, taking or releasing the lock for either read or write, is more complex, internally requiring taking and releasing two mutexes instead of one. This variation is sometimes also known as "write-biased" reader-writer lock.
- *Unspecified priority RW locks* do not provide any guarantees with regards read vs. write access. Unspecified priority can in some situations be preferable if it allows for a more efficient implementation.

The read-copy-update (RCU) algorithm, see §14.4, is one solution to the readers-writers problem. RCU is wait-free for readers. The Linux kernel implements a special solution for few writers called seqlock.

7.4 Lock Evaluation

1. Correctness. Basically, does the lock work, preventing multiple threads from entering a critical section?
2. Fairness. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: does any thread contending for the lock starve while doing so, thus never obtaining it?
3. Performance. What are the costs of using a spinlock? To analyze this more carefully, we suggest thinking about different cases. In the first, imagine threads competing for the lock on a single processor; in the second, consider threads spread out across many CPUs.

   For spinlocks, in the single CPU case, performance overheads can be quite painful; imagine the case where the thread holding the lock is preempted within a critical section. The scheduler might then run every other thread (imagine there are $N-1$ others), each of which tries to acquire the lock. In this case, each of those threads will spin for the duration of a time slice before giving up the CPU, a waste of CPU cycles.

   However, on multiple CPUs, spinlocks work reasonably well (if the number of threads roughly equals the number of CPUs). The thinking goes as follows: imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A (CPU 1) grabs the lock, and then Thread B tries to, B will spin (on CPU 2). However, presumably the critical section is short, and thus soon the lock becomes available, and is acquired by Thread B. Spinning to wait for a lock held on another processor doesn't waste many cycles in this case, and thus can be effective.

In this order. Why?

7.5 Evaluating Spinlocks

Example code:

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Evaluation:

- Correctness. The answer here is yes: the spinlock only allows a single thread to enter the critical section at a time. Thus, we have a correct lock.
- Fairness. How fair is a spinlock to a waiting thread? Can you guarantee that a waiting thread will ever enter the critical section? The answer here, unfortunately, is bad news: spinlocks don't provide any fairness guarantees. Indeed, a thread spinning may spin forever when under contention. Simple spinlocks are not fair and may lead to starvation. You should carefully consider why they are still in use when they are not fair.
- Performance. It makes no sense to evaluate performance until we get fairness.

7.6 xv6 Spin and Sleep Locks

7.6.1 xv6 holding() Helper Function

spinlock:

```
// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
    int r;
    pushcli();
    r = lock->locked && lock->cpu == mycpu();
```

```
    popcli();
    return r;
}
```

Why does this routine turn off interrupts? Hint: context switch.

sleep lock:

```
int
holdingsleep(struct sleeplock *lk)
{
    int r;

    acquire(&lk->lk);
    r = lk->locked;
    release(&lk->lk);
    return r;
}
```

7.6.2 xv6 acquire()

spinlock:

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
    panic("acquire");

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

A Note on `pushcli()` and `popcli()`

The xv6 `pushcli()` and `popcli()` routines are *wrappers*[a] for the Intel instructions `cli` (clear interrupt) and `sti` (set interrupt). These wrappers allow `pushcli()` invocations to be nested and interrupts will remain off (cleared) until the same number of `popcli()` invocations occur. That is, the actual Intel `sti` instruction will not be invoked until the push and pop instructions are balanced.

- It takes two `popcli()` to undo two `pushcli()`. They count.
- Interrupt state is preserved. For example, if interrupts are off, then a single `pushcli()` followed by a single `popcli()` will not reenable interrupts for the processor. The `sti` instruction will not be executed.

Compare this to the behavior of `sti` regardless of how many times `cli` have have been invoked. It is useful to consider what would happen if a programmer mixed this approach with direct calls to `cli` and `sti`.

[a] In computer science, a wrapper is any entity that encapsulates (wraps around) another item. Wrappers are used for two primary purposes: to convert data to a compatible format or to hide the complexity of the underlying entity using abstraction. Examples include object wrappers, function wrappers, and driver wrappers. See TechTerms. Many library functions, such as those in the C Standard Library, act as interfaces for abstraction of system calls. The fork and execve functions in glibc are examples of this. They call the lower-level `fork()` and `execve()` system calls, respectively. See Wikipedia.

sleep lock:

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    while (lk->locked) {
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}
```

7.6.3 xv6 release()

spinlock:

```
// Release the lock.
void
release(struct spinlock *lk)
{
    if(!holding(lk))
        panic("release");
```

```
    lk->pcs[0] = 0;
    lk->cpu = 0;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

sleep lock:

```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}
```

7.7 The Dining Philosophers Problem Using Locks

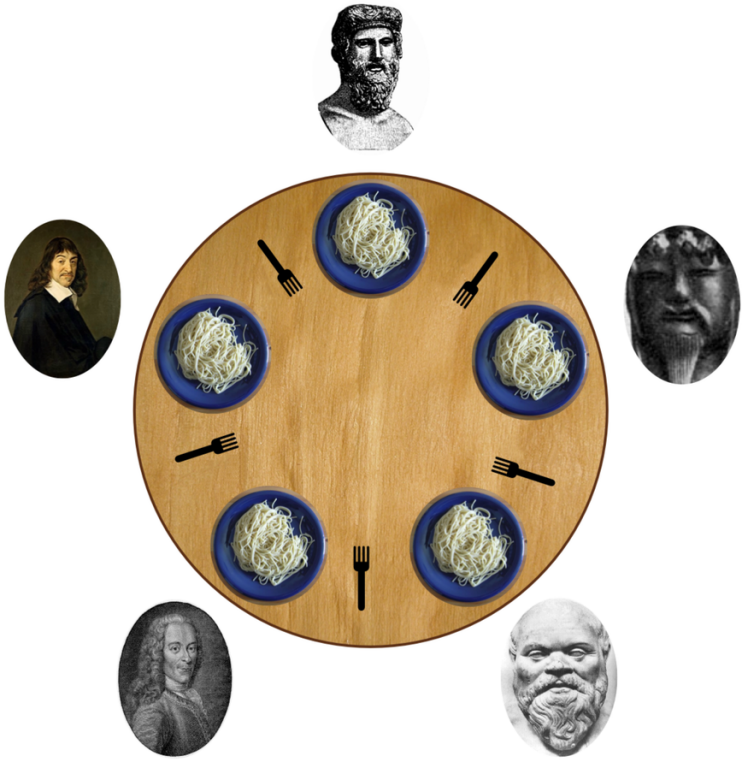

Figure 7: The Dining Philosophers Problem

Five silent philosophers sit at a round table with bowls of noodles. In Figure 7, forks are placed between each pair of adjacent philosophers. That is, a fork is a *shared resource*[28].

Each philosopher must alternately think and eat. However, a philosopher can only eat noodles when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of noodles or stomach space; an infinite supply and an infinite demand are assumed. Otherwise, there is no deadlock or starvation.

The problem is how to design a concurrent algorithm such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows[29].

1. think until the left fork is available; when it is, pick it up;

[28] See Wikipedia.

[29] Assume that each step is an atomic transaction.

2. think until the right fork is available; when it is, pick it up;
3. when both forks are held, eat for a fixed amount of time;
4. then, put the right fork down;
5. then, put the left fork down;
6. repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.

Starvation might also occur independently of deadlock if a particular philosopher is perpetually unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of live lock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again[30].

A lock-based solution

Let us suppose that the table has a lock. All philosophers are required to hold the lock before they can attempt a change in the state of their utensils. No deadlock can result, but individual philosophers can be starved. See section §8.

The left-handed philosopher

Assume all philosophers always attempt to pick up their utensils in this order: right utensil followed by left utensil (right-handed). This is essentially the original problem statement. Now, pick one philosopher at random to be *left-handed*, meaning that this philosopher will always attempt to pick up the left utensil before the right utensil. No deadlock can result, but starvation is still possible.

What about starvation?

To eliminate starvation, you must guarantee that no philosopher may be blocked in an unbounded manner. For example, suppose you maintain a queue of philosophers. When a philosopher is hungry, they get put onto the tail of the queue. A philosopher may eat only if they are at the head of the queue and the necessary forks are free. When both forks are not free the philosopher again waits by moving back onto the tail of the queue. This approach solves the deadlock and starvation problems, using ordering on lock acquisition. It isn't obvious that this is the case, so consider this idea carefully.

[30] See Wikipedia

7.8 Pthread Mutexes

- `int pthread_mutex_lock(pthread_mutex_t *mutex)` – acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` – attempt to lock a mutex or will return error code if busy. Useful for preventing deadlock conditions. The main questions is how long to wait before retrying. The use of a binary back-off[31] algorithm is common.
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)` – unlock a mutex variable. An error is returned if mutex is already unlocked or owned by another thread.

Example Solution Using A Mutex

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

pthread_t philosopher[5];
pthread_mutex_t utensil[5];

void *func(int n)
{
    printf ("Philosopher %d is thinking\n",n);
    //when philosopher 5 is eating he takes utensil 1 and fork 5
    pthread_mutex_lock(&utensil[n]);
    pthread_mutex_lock(&utensil[(n+1)%5]);
    printf ("Philosopher %d is eating\n",n);
    sleep(3);
    pthread_mutex_unlock(&utensil[n]);
    pthread_mutex_unlock(&utensil[(n+1)%5]);
    printf ("Philosopher %d finished eating\n",n);
    return(NULL);
}

int main()
{
    int i;

    for(i=0;i<5;i++)
        pthread_mutex_init(&utensil[i],NULL);
    for(i=0;i<5;i++)
        pthread_create(&philosopher[i],NULL,(void *)func,(void *)i);
    for(i=0;i<5;i++)
        pthread_join(philosopher[i],NULL);
```

[31] See Wikipedia

```
    for(i=0;i<5;i++)
         pthread_mutex_destroy(&utensil[i]);
    exit(EXIT_SUCCESS);
}
```

7.9 Problems with Locks

Lock-based resource protection and thread/process synchronization have many disadvantages[32]:

Problem	Description
Contention	some threads/processes have to wait until a lock (or a whole set of locks) is released. If one of the threads holding a lock dies, stalls, blocks, or enters an infinite loop, other threads waiting for the lock may wait forever.
Overhead	the use of locks adds overhead for each access to a resource, even when the chances for collision are very rare. (However, any chance for such collisions is a race condition.)
Debugging	bugs associated with locks are time dependent and can be very subtle and extremely hard to replicate, such as deadlocks.
Instability	the optimal balance between lock overhead and lock contention can be unique to the problem domain (application) and sensitive to design, implementation, and even low-level system architectural changes. These balances may change over the life cycle of an application and may entail tremendous changes to update (re-balance).
Composability	Composability is a system design principle that deals with the inter-relationships of components. A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements. Locks are only composable with relatively elaborate software support and perfect adherence by applications programming to rigorous conventions[33].
Priority inversion	a low-priority thread/process holding a common lock can prevent high-priority threads/processes from proceeding. Priority inheritance can be used to reduce priority-inversion duration. The priority ceiling protocol can be used on uniprocessor systems to minimize the worst-case priority-inversion duration, as well as prevent deadlock.
Convoying	all other threads have to wait if a thread holding a lock is descheduled due to a time-slice interrupt or page fault.

Table 4: Problems with Locks

Some concurrency control strategies avoid some or all of these problems. For example, a funnel

[32] See Wikipedia
[33] See Wikipedia.

or serializing tokens can avoid the biggest problem: deadlocks. Alternatives to locking include non-blocking synchronization methods, like lock-free programming techniques and transactional memory. However, such alternative methods often require that the actual lock mechanisms be implemented at a more fundamental level of the operating software. Therefore, they may only relieve the application level from the details of implementing locks, with the problems listed above still needing to be dealt with beneath the application.

In most cases, proper locking depends on the CPU providing a method of atomic instruction stream synchronization (for example, the addition or deletion of an item into a pipeline requires that all contemporaneous operations needing to add or delete other items in the pipe be suspended during the manipulation of the memory content required to add or delete the specific item). Therefore, an application can often be more robust when it recognizes the burdens it places upon an operating system and is capable of graciously recognizing the reporting of impossible demands.

One of lock-based programming's [sic] biggest problems is that *locks don't compose*: it is hard to combine small, correct lock-based modules into equally correct larger programs without modifying the modules or at least knowing about their internals.

8 Semaphores

A semaphore is comprised of an integer counting variable and a wait list. Unlike locks and condition variables, semaphores can be shared between processes. An interesting thread discussing some of the differences between locks and semaphores can be found at Stack Overflow.

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions[34].

A useful way to think of a semaphore as used in the real-world system is as a record of how many units of a particular resource are available, coupled with operations to initialize and adjust that record safely (i.e. to avoid race conditions) as units are required or become free, and, if necessary, wait until a unit of the resource becomes available.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement locks.

The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1962 or 1963, when Dijkstra and his team were developing an operating system for the Electrologica X8. That system eventually became known as THE multiprogramming system.

There are two types of semaphores:

- A *binary semaphore* can only take on one of two integer values, 0 and 1. It is used to implement locks.

[34] See Wikipedia.

- A *general semaphore* or counting semaphore can take on any integer value, positive, negative, or zero.

To avoid starvation, a semaphore has an associated queue of processes (usually with FIFO semantics). If a process performs a wait operation on a semaphore that has a zero or negative value, the process is added to the semaphore's wait queue and its execution is suspended. When another process increments the semaphore by performing a signal operation, if there are processes on the wait queue, one of them is removed from the queue and resumes execution. When processes have different priorities the queue may be ordered differently, so that the highest priority process is taken from the queue first. Priority opens the door to starvation, but if properly managed can be avoided.

8.1 Library Analogy

Suppose a library has 10 identical study rooms, to be used by one student at a time. Students must request a room from the front desk if they wish to use a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available, which they only know correctly if all of the students actually use their room while they've signed up for them and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the current value of the semaphore is 0, they are forced to wait until a room is freed (when the count is increased from 0)[35]. If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or flipping a coin). And of course, a student needs to inform the clerk about releasing their room only after really leaving it, otherwise, there can be an awkward situation when such student is in the process of leaving the room (they are packing their textbooks, etc.) and another student enters the room before they leave it.

8.2 Implementation

Binary Semaphore Example

```
Semaphore my_semaphore = 1; // Initialize to nonzero
int withdraw(account, amount) {
    wait(my_semaphore);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
```

[35]For implementations that allow the count to be less than zero, a negative value represents the number of threads/processes waiting on the semaphore.

```
    signal(my_semaphore);
    return balance;
}
```

General Semaphore Example

struct semaphore:

```
struct semaphore {
    int val;
    thread_list waiting; // List of threads waiting for semaphore
}
```

Wait until > 0 then decrement.

wait(semaphore Sem):

```
while (Sem.val <= 0) {
    add this thread to Sem.waiting;
    block(this thread);
}
Sem.val = Sem.val - 1;
return;
```

Increment value and wake up a waiting thread, if necessary.

signal(semaphore Sem):

```
Sem.val = Sem.val + 1;
if (Sem.waiting is nonempty) {
    remove some thread T from Sem.waiting;
    wakeup(T);
}
```

Why use a `while` loop in `wait()` instead of an `if` statement?

```
while (Sem.val <= 0) { . . .
```

Because another, non-blocked, thread could call `wait()` while this thread is waiting.

8.3 Ensuring Atomicity of the Semaphore

How do we ensure that the semaphore implementation is atomic?

1. One option: use a lock for `wait()` and `signal()`
    - Make sure that only one `wait()` or `signal()` can be executed at a time
    - Need to be careful to release lock before sleeping, acquire lock on waking up

2. Another option: Add more complex hardware support
    - Atomic instructions
    - Specialized memory

If the implementation does not ensure atomicity of the increment, decrement and comparison operations, then there is a risk of increments or decrements being lost or forgotten. Atomicity may be achieved by using a machine instruction that is able to read, modify and write the semaphore in a single operation. In the absence of such a hardware instruction, an atomic operation may be synthesized through the use of a software mutual exclusion algorithm, such as an atomic transaction.

On uniprocessor systems, atomic operations can be ensured by temporarily suspending preemption or disabling hardware interrupts. This approach does not work on multiprocessor systems where it is possible for two programs sharing a semaphore to run on different processors at the same time. To solve this problem in a multiprocessor system a locking variable can be used to control access to the semaphore. The locking variable is manipulated in the usual manner.

8.4 Problems with Semaphores

Much of the power of semaphores derives from calls to `wait()` and `signal()` that are unmatched.

- Unlike locks, where acquire and release are always paired.
- This means it is a lot easier to get into trouble with semaphores. Any thread or any process can potentially signal a semaphore.
- There is no concept of ownership. The thread acquiring the semaphore does not have to be the one signaling on it. This means that any thread of control can "unlock" it! This is a key difference between a binary semaphore and a spinlock.

8.5 Semaphores on Linux

The GNU C Library implements the semaphore APIs as defined in POSIX and System V. Semaphores can be used by multiple processes to coordinate shared resources. The following is a complete list of the semaphore functions provided by the GNU C Library.

8.5.1 System V Semaphores

- `int semctl (int semid, int semnum, int cmd)`
- `int semget (key_t key, int nsems, int semflg)`
- `int semop (int semid, struct sembuf *sops, size_t nsops)`
- `int semtimedop (int semid, struct sembuf *sops, size_t nsops, const struct timespec *timeout)`

8.5.2 POSIX Semaphores

- `int sem_init (sem_t *sem, int pshared, unsigned int value)`
  Initializes the semaphore object pointed to by `sem`. The count associated with the semaphore is set initially to `value`. The `pshared` argument indicates whether the semaphore is local to the current process (`pshared` is zero) or is to be shared between several processes (`pshared` is not zero).

- `int sem_destroy (sem_t * sem)`
  `sem_destroy()` destroys a semaphore object, freeing the resources it might hold. If any threads are waiting on the semaphore when `sem_destroy()` is called, it fails and sets errno to EBUSY.

- `int sem_wait (sem_t * sem)`
  Decrements (locks) the semaphore pointed to by `sem`. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

- `int sem_trywait (sem_t * sem)`
  A non-blocking variant of `sem_wait()`. If the semaphore pointed to by `sem` has non-zero count, the count is atomically decreased and `sem_trywait()` immediately returns 0. If the semaphore count is zero, `sem_trywait()` immediately returns -1 and sets errno to EAGAIN.

- int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
  The same as `sem_wait()`, except that `abs_timeout` specifies a limit on the amount of time that the call should block if the decrement cannot be immediately performed. The `abs_timeout` argument points to a structure that specifies an absolute timeout in seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). This structure is defined as follows:

  ```
  struct timespec {
      time_t tv_sec;      /* Seconds */
      long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
  };
  ```

  If the timeout has already expired by the time of the call, and the semaphore could not be locked immediately, then `sem_timedwait()` fails with a timeout error (errno set to ETIMEDOUT).

  If the operation can be performed immediately, then `sem_timedwait()` never fails with a timeout error, regardless of the value of `abs_timeout`. Furthermore, the validity of `abs_timeout` is not checked in this case.

- `int sem_post (sem_t * sem)`
  Atomically increases the count of the semaphore pointed to by `sem`. This function never blocks. On processors supporting atomic compare-and-swap (Intel 486, Pentium and later, Alpha, PowerPC, MIPS II, Motorola 68k, Ultrasparc), the `sem_post()` function is can safely be called from signal handlers. This is the only thread synchronization function provided

by POSIX threads that is *async-signal safe*. On the Intel 386 and earlier Sparc chips, the current Linux threads implementation of `sem_post()` is not async-signal safe, because the hardware does not support the required atomic operations.

`sem_post()` always succeeds and returns 0, unless the semaphore count would exceed `SEM_VALUE_MAX` after being incremented. In that case `sem_post()` returns -1 and sets errno to EINVAL. The semaphore count is left unchanged.

- `int sem_getvalue (sem_t * sem, int * sval)`
  Stores in the location pointed to by `sval` the current count of the semaphore `sem`. It always returns 0.

8.6 The Producer – Consumer Problem

In the producer – consumer problem, one or more processes generate data items and one or more processes receive and use the data. They communicate using a queue of maximum size N and are subject to the following conditions:

- The consumer must wait for the producer to produce something if the queue is empty;
- The producer must wait for the consumer to consume something if the queue is full.
- The semaphore solution to the producer – consumer problem tracks the state of the queue with two semaphores: emptyCount, the number of empty places in the queue, and fullCount, the number of elements in the queue. To maintain integrity, emptyCount may be lower (but never higher) than the actual number of empty places in the queue, and fullCount may be lower (but never higher) than the actual number of items in the queue. Empty places and items represent two kinds of resources, empty boxes and full boxes, and the semaphores emptyCount and fullCount maintain control over these resources.

The binary semaphore useQueue ensures that the integrity of the state for the queue itself is not compromised, for example by two producers attempting to add items to an empty queue simultaneously, thereby corrupting its internal state. Alternatively a mutex could be used in place of the binary semaphore.

The emptyCount is initially N, fullCount is initially 0, and useQueue is initially 1.

The producer does the following repeatedly:

```
produce:
    P(emptyCount)
    P(useQueue)
    putItemIntoQueue(item)
    V(useQueue)
    V(fullCount)
```

The consumer does the following repeatedly:

```
consume:
    P(fullCount)
    P(useQueue)
    item = getItemFromQueue()
```

```
V(useQueue)
V(emptyCount)
```

Note that while these appear to be paired, they are **not paired**. One signal can cause several waiting threads to wake up. However, implementing semaphores as paired is common and may require a little extra code in the user program to count the wait and signal operations[36]. See §8.

Example:
A single consumer enters its critical section. Since fullCount is 0, the consumer blocks. Several producers enter the producer critical section. No more than N producers may enter their critical section due to emptyCount constraining their entry. The producers, one at a time, gain access to the queue through useQueue and deposit items in the queue. Once the first producer exits its critical section, fullCount is incremented, allowing one consumer to enter its critical section.

Note that emptyCount may be much lower than the actual number of empty places in the queue, for example in the case where many producers have decremented it but are waiting their turn on useQueue before filling empty places. Note that emptyCount + fullCount $\leq N$ always holds, with equality if and only if no producers or consumers are executing their critical sections.

8.7 The Dining Philosopher Problem Using Semaphores

We have seen a lock-based solution to the Dining Philosophers Problem that removes deadlock but leaves starvation. We can solve both problems with semaphores.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes utensil %d and %d\n",
            phnum + 1, LEFT + 1, phnum + 1);
```

[36]Simple "wrapper functions", such as xv6's `pushcli()` and `popcli()`, can be used.

```
        printf("Philosopher %d is Eating\n", phnum + 1);

        // sem_post(&S[phnum]) has no effect
        // during take_utensil
        // used to wake up hungry philosophers
        // during put_utensil
        sem_post(&S[phnum]);
    }
}

// take up utensil
void take_utensil(int phnum)
{
    sem_wait(&mutex);
    // state that hungry
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}

// put down utensil
void put_utensil(int phnum)
{
    sem_wait(&mutex);
    // state that thinking
    state[phnum] = THINKING;
    printf("Philosopher %d putting utensil %d and %d down\n",
    phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void* philospher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_utensil(*i);
        sleep(0);
        put_utensil(*i);
```

```
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];
    // initialize the semaphores
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
    for (i = 0; i < N; i++) {
        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
        philospher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);
    exit(EXIT_SUCCESS);
}
```

9 Condition Variables

A condition variable (CV) compares against some *arbitrary condition*. The vast majority of the time, the condition check will resolve to FALSE (0) or TRUE (not 0) a binary condition asking if the condition has been met. A lock is always associated with a condition variable. Note that CVs do not have the concept of ownership. Condition Variables are used with Monitors (next section).

A condition variable can be used to implement a two-phase approach to lock acquisition.

A Naïve and Broken Version

```
volatile int i = 0;

static void *f1(void *p)
{
    while (i==0) {  } // busy wait
    printf("i's value has changed to");
    printf(" %d.\n", i);
    return NULL;
}

static void *f2(void *p)
{
    sleep(60);
    i = 99;   // wrong way to signal
    printf("t2 has changed the value of");
    printf(" i to %d.\n", i);
    return NULL;
}

int main()
{
    int rc;
    pthread_t t1, t2;

    rc = pthread_create(&t1, NULL, f1, NULL);
    if (rc != 0) {
        // print error and exit
     }

    rc = pthread_create(&t2, NULL, f2, NULL);
    if (rc != 0) {
        // print error and exit
    }

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    puts("All pthreads finished.");
```

```
    exit(EXIT_SUCCESS);
}
```

Why is this example naïve?

A condition variable represents a condition on which a thread can:

- **Initialize**
- **Wait** until the condition occurs; or
- **Signal**, or notify, other waiting threads that the condition has been met

Note that while these appear to be paired, they are **not paired**. One signal can cause several threads to wake up. However, implementing CVs as paired is common and may require a little extra code to "count" the wait and signal operations for each semaphore used.

This is a very useful primitive for asynchronous communication between threads.

There are three operations that can be performed on condition variables:

Problem	Description
`wait()`	Block until another thread calls `signal()` or `broadcast()` on the CV
`signal()`	Wake up one thread waiting on the CV
`broadcast()`	Wake up all threads waiting on the CV

Table 5: Operations on Condition Variables

9.1 POSIX Condition Variables

With POSIX pthreads, a condition variable is of type `pthread_cond_t`. Four operations are supported:

- `pthread_cond_init()` to initialize. Different ways exist, see the man page.
- `pthread_cond_wait(&theCV, &someLock)` to wait on the semaphore. Note that the lock is **required**.
- `pthread_cond_signal(&theCV)` to signal the semaphore.
- `pthread_cond_broadcast(&theCV)` to broadcast a signal to all threads waiting on the semaphore.

Example:

```
/* globals */
pthread_mutex_t myLock;
pthread_cond_t myCV;
int counter = 0;
```

```
/* Thread A */
pthread_mutex_lock(&myLock);
while (counter < 10) {
    pthread_cond_wait(&myCV, &myLock);
}
pthread_mutex_unlock(&myLock);

/* Thread B */
pthread_mutex_lock(&myLock);
counter++;
if (counter >= 10) {
    pthread_cond_signal(&myCV);
}
pthread_mutex_unlock(&myLock);
```

10 Monitors

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex (lock) object and condition variables. A condition variable is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

Another definition of monitor is a thread-safe class, object, or module that wraps around a mutex in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. By using one or more condition variables it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor").

Monitors were invented by Per Brinch Hansen and C. A. R. Hoare, and were first implemented in Brinch Hansen's Concurrent Pascal language.[37].

The proper basic usage of a monitor is:

```
acquire(m); // Acquire this monitor's lock.
while (!p) {
    wait(m, cv); .
}
// ... Critical section of code goes here ...
signal(cv2);
// -- OR -- notifyAll(cv2); // broadcast semantics
// cv2 might be the same as cv or different.
release(m);
```

The monitor is a common technique and appears to be built into POSIX condition variable. The mutex will be released while the thread is waiting but will be reacquired before the `wait()`

[37]See Wikipedia.

call returns. This approach violates the *hold and wait* condition for deadlock (see §3.7.1) and therefore guarantees that deadlock will not occur. Compare with semaphores.

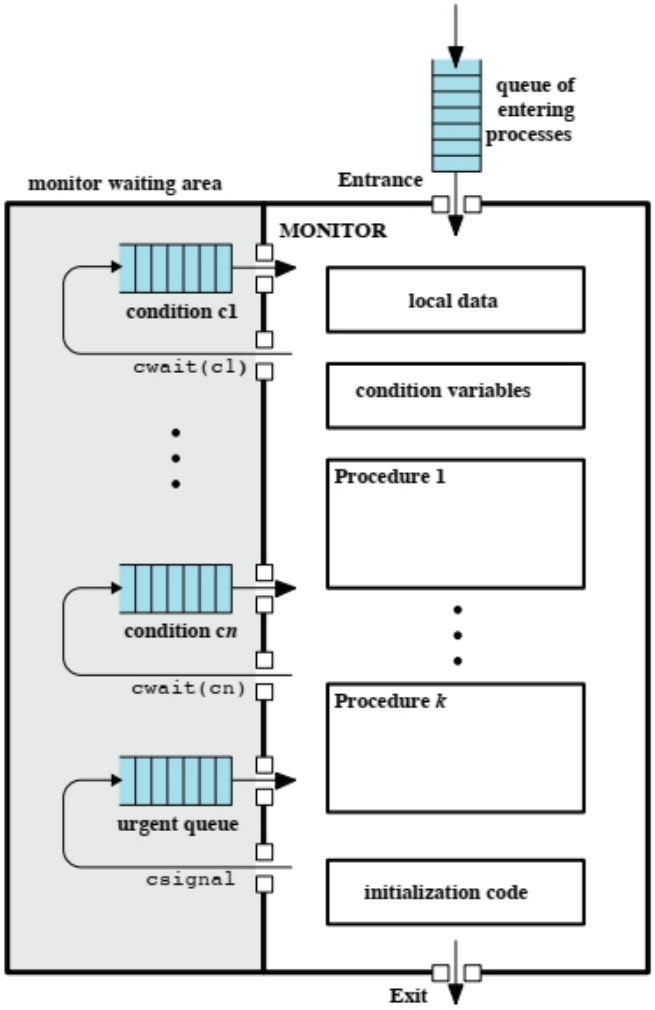

Figure 8: Monitor Structure

11 Shared Memory

Different processes, as well as thread, can use a shared memory model. This model of concurrency is quite tricky and harder to manage than a simple critical section, simply because *all of the shared memory is merely a collection of one or more critical sections.* A very careful approach is warranted. For this example, let's assume that a family of (at least) four shares the same ATM card. Alternately, you can assume identity theft, but that's another course.

Let's look at an example of a shared memory system. The point here is to show that concurrent programming is hard, because it can have subtle bugs.

A Shared Memory Model for Bank Accounts[38]

Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.

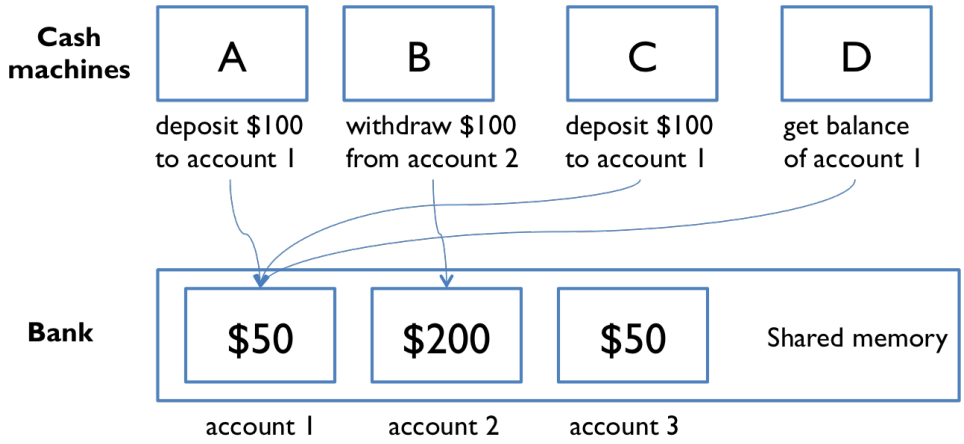

Figure 9: Shared Memory Example

To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

[38] From an MIT course example.

In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.

But if we run this code, we discover frequently that the balance at the end of the day is *not 0*. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then balance may not be zero at the end of the day. Why not? **Interleaving**

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the deposit() step typically breaks down into low-level processor instructions:

```
get balance (balance=0)
add 1
write back the result (balance=1)
```

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

```
A get balance (balance=0)
A add 1
A write back the result (balance=1)
                                    B get balance (balance=1)
                                    B add 1
                                    B write back the result (balance=2)
```

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

```
A get balance (balance=0)
                                    B get balance (balance=0)
A add 1
                                    B add 1
A write back the result (balance=1)
                                    B write back the result (balance=1)
```

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

This is an example of a *race condition*. A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say "A is in a race with B."

Some interleavings of events may be OK, in the sense that they are consistent with what a single, non-concurrent process would produce, but other interleavings produce wrong answers – violating postconditions or invariants.

You can't tell just from looking at the code how the processor is going to execute it. You can't tell what the indivisible operations – the atomic operations – will be. It isn't *atomic* just because it's one line of source code. It doesn't touch balance only once just because the balance identifier occurs only once in the line. The compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code.

The key lesson is that you can't tell by looking at an expression whether it will be safe from race conditions.

12 Message Passing

Message passing is a technique for invoking behavior in a separate thread of control or even a separate process on a remote computer. The invoking program sends a message to a second process and relies on that second process and its supporting infrastructure to then select and run some appropriate code. Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is key to some models of concurrency and object-oriented programming.

Message passing is used ubiquitously in modern computer software. It is used as a way for the objects that make up a program to work with each other and as a means for objects and systems running on different computers to interact. Message passing may be implemented by a wide variety mechanisms.

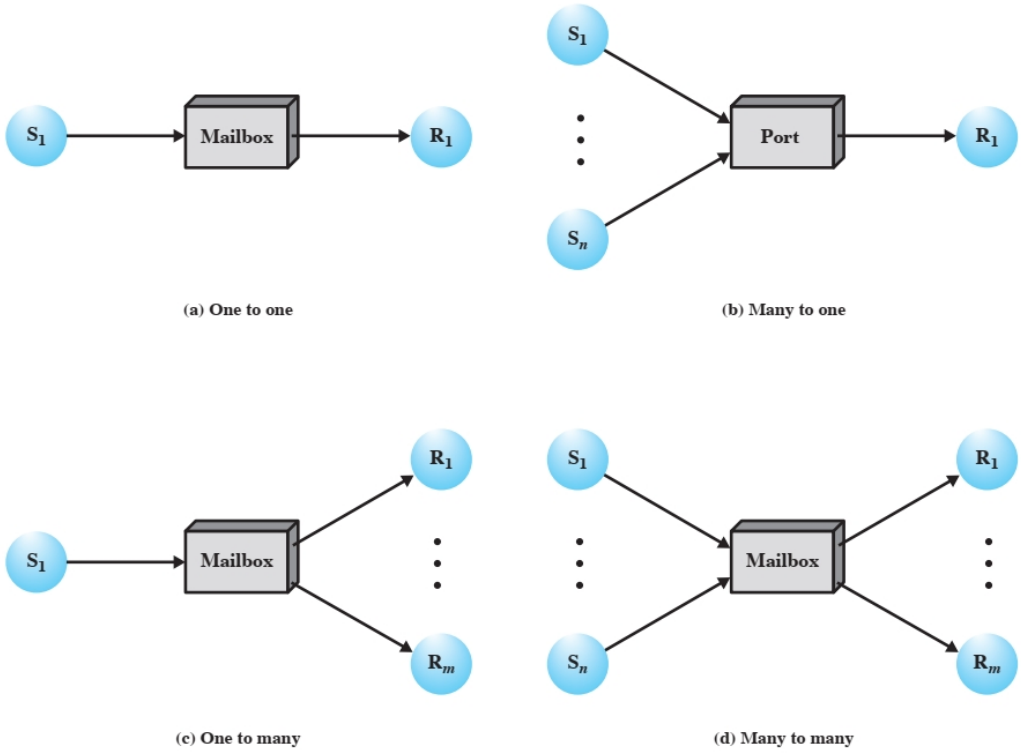

Figure 10: Approaches to Message Passing[39]

Approaches

- Figure 10(a) shows a simple message passing mechanism that associates on producer with one consumer. The key here is that the mailbox is typically managed by the operating system, which makes code development much easier. This is a *one-to-one* relationship.
- Figure 10(b) shows multiple (possibly remote) producers communicating with a single consumer. This is a common approach for when a problem is split among many threads of control and one thread collects the individual results for further use. This is a *many-to-one* relationship.
- Figure 10(c) shows a single producer communicating with many consumers. This can be seen as the reverse of (b), where a single thread of control sends out work to other threads of control, which may be remote. This is a *one-to-many* relationship.
- Figure 10(d) shows multiple producers communicating with multiple consumers. This *many-to-many* relationship is very common in a distributed system.

[39]From Stallings, *Operating Systems, Internals and Design Principles*, 9th Ed.

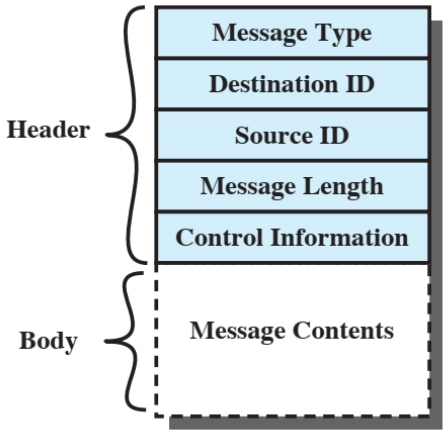

Figure 11: Generic Message Format[40]

A message must use an agreed upon format. Figure 11 is a simple example. This ensures that the consumer will be able to decipher how the message should be processed. The message format may be specified by a standard, such as the Internet Protocol Suite (TCP/IP). A good example is the UDP[41] protocol.

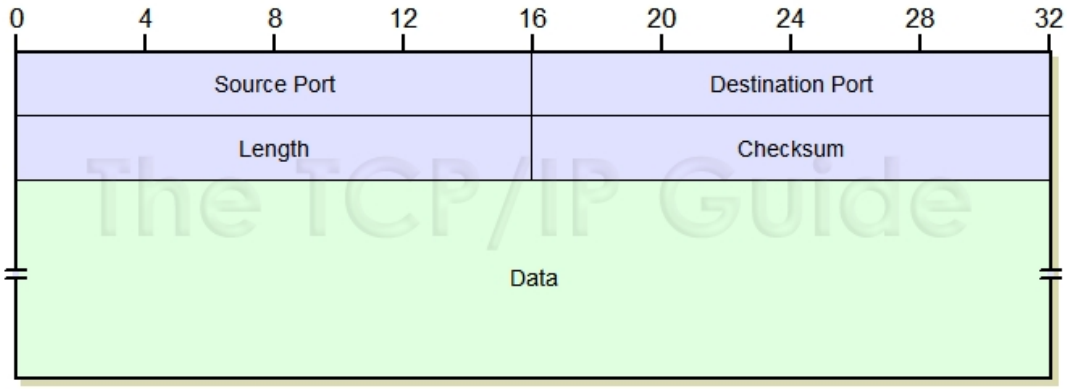

Figure 12: UDP Message Format[42]

A comparison can be made with the postal service. A letter constitutes a single message. Multiple messages can be en route between 2 communicants, with messages moving in both directions. However – and this is important – the postal service makes no guarantee as to the order of arrival or the amount of time each letter will be in transit. Many message passing services have this same "feature".

[40] ibid.
[41] See Wikipedia and RFC 8085.
[42] See The TCP/IP Guide.

13 Unix Pipes

A pipe is a form of redirection that is used in Linux and other Unix-like operating systems to send the output of one program to another program for further processing.

Redirection is the transferring of standard output to some other destination, such as another program, a file or a printer, instead of the display monitor (which is its default destination). Standard output, sometimes abbreviated stdout, is the destination of the output from command line (i.e., all-text mode) programs in Unix-like operating systems.

Pipes are used to create what can be visualized as a pipeline of commands, which is a temporary direct connection between two or more simple programs. This connection makes possible the performance of some highly specialized task that none of the constituent programs could perform by themselves. A command is merely an instruction provided by a user telling a computer to do something, such as launch a program. The command line programs that do the further processing are referred to as filters.

This direct connection between programs allows them to operate simultaneously and permits data to be transferred between them continuously, rather than having to pass it through temporary text files or through the display screen and having to wait for one program to be completed before the next program begins.

Pipes rank alongside the hierarchical file system and regular expressions as one of the most powerful yet elegant features of Unix-like operating systems. The hierarchical file system is the organization of directories in a tree-like structure which has a single root directory (i.e., a directory that contains all other directories). Regular expressions are a pattern matching system that uses strings (i.e., sequences of characters) constructed according to pre-defined syntax rules to find desired patterns in text.

Pipes were first suggested by M. Doug McIlroy, when he was a department head in the Computing Science Research Center at Bell Labs, the research arm of AT&T (American Telephone and Telegraph Company), the former U.S. telecommunications monopoly. McIlroy had been working on macros since the latter part of the 1950s, and he was a ceaseless advocate of linking macros together as a more efficient alternative to series of discrete commands. A macro is a series of commands (or keyboard and mouse actions) that is performed automatically when a certain command is entered or key(s) pressed.

McIlroy's persistence led Ken Thompson, who developed the original UNIX at Bell Labs in 1969, to rewrite portions of his operating system in 1973 to include pipes. This implementation of pipes was not only extremely useful in itself, but it also made possible a central part of the Unix philosophy, the most basic concept of which is modularity (i.e., a whole that is created from independent, replaceable parts that work together efficiently)[43].

[43] See LINFO, the Linux Information Project.

13.1 Regular Pipes

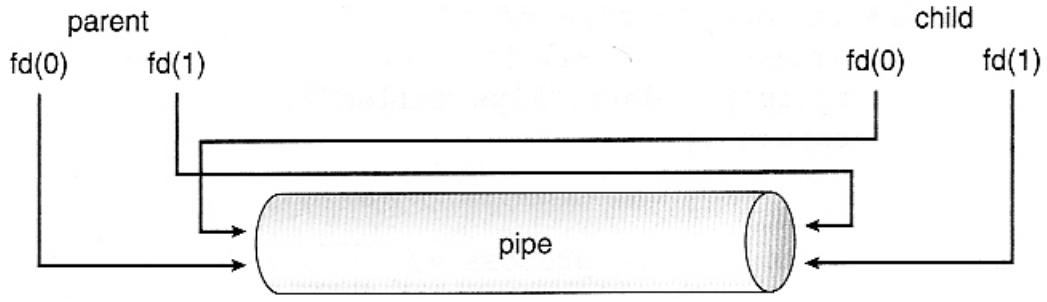

Figure 13: Ordinary UNIX Pipe[44]

DESCRIPTION

`pipe()` creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe. `pipefd[0]` refers to the read end of the pipe. `pipefd[1]` refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. For further details, see `pipe(7)`.

SYNOPSIS

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

EXAMPLE

The following program creates a pipe, and then fork(2)s to create a child process; the child inherits a duplicate set of file descriptors that refer to the same pipe. After the fork(2), each process closes the file descriptors that it doesn't need for the pipe (see pipe(7)). The parent then writes the string contained in the program's command-line argument to the pipe, and the child reads this string a byte at a time from the pipe and echoes it on standard output.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
```

[44] Dr. John T. Bell

```
  int pipefd[2];
  pid_t cpid;
  char buf;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <string>\n", argv[0]);
    exit(EXIT_FAILURE);
  }

  if (pipe(pipefd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
  }

  cpid = fork();
  if (cpid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }


  if (cpid == 0) {     /* Child reads from pipe */
    close(pipefd[1]);  /* Close unused write end */

    while (read(pipefd[0], &buf, 1) > 0)
    write(STDOUT_FILENO, &buf, 1);

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    exit(EXIT_SUCCESS);

  } else {                    /* Parent writes argv[1] to pipe */
    close(pipefd[0]);         /* Close unused read end */
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);         /* Reader will see EOF */
    wait(NULL);               /* Wait for child */
    exit(EXIT_SUCCESS);
  }
}
```

13.2 Named Pipes

Named pipes aren't used all that often, but they provide some interesting options for inter-process communications[45] and synchronization.

One of the key differences between regular pipes and named pipes is that named pipes have a presence in the file system. That is, they show up as files. But unlike most files, they never appear

[45] See Network World.

to have contents. Even if you write a lot of data to a named pipe, the file appears to be empty.

There are many ways to write to a file, read from a file, and empty a file, though named pipes have a certain efficiency going for them.

For one thing, named pipe content resides in memory rather than being written to disk. It is passed only when both ends of the pipe have been opened. And you can write to a pipe multiple times before it is opened at the other end and read. By using named pipes, you can establish a process in which one process writes to a pipe and another reads from a pipe without much concern about trying to time or carefully orchestrate their interaction.

You can set up a process that simply waits for data to appear at the output end of the pipe and then works with it when it does.

Under Linux, a named pipe is created with the `mkfifo` command.

NAME

```
mkfifo - make FIFOs (named pipes)
```

SYNOPSIS

```
mkfifo [OPTION]... NAME...
```

DESCRIPTION

```
Create named pipes (FIFOs) with the given NAMEs.
```

Any process with read and/or write permissions to the named pipe may use it as a rendezvous or synchronization mechanism.

Using a Pipe to Pass Data Between a Parent Process and a Child Process

The following pseudo code[46] demonstrates the use of a pipe to transfer data between parent and child processes. Error handling is excluded, but otherwise this code demonstrates good practice when using pipes: after the fork() the two processes close the unused ends of the pipe before they commence transferring data.

```
#include <stdlib.h>
#include <unistd.h>
...

int fildes[2];
const int BSIZE = 100;
char buf[BSIZE];
ssize_t nbytes;
int status;
```

[46] See The Open Group.

```
status = pipe(fildes);
if (status == -1 ) {
    /* an error occurred */
    ...
}

switch (fork()) {
    case -1: /* Handle error */
    break;

    case 0:  /* Child - reads from pipe */
    close(fildes[1]);                       /* Write end is unused */
    nbytes = read(fildes[0], buf, BSIZE);   /* Get data from pipe */
    /* At this point, a further read would see end-of-file ... */
    close(fildes[0]);                       /* Finished with pipe */
    exit(EXIT_SUCCESS);

    default:  /* Parent - writes to pipe */
    close(fildes[0]);                       /* Read end is unused */
    write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
    close(fildes[1]);                       /* Child will see EOF */
    exit(EXIT_SUCCESS);
}
```

14 Advanced Concurrency Control

14.1 Memory Barriers

A *memory barrier*, also known as a *membar*, *memory fence* or *fence instruction*, is an instruction that causes a processor or compiler to enforce an ordering constraint on memory operations issued before and after the instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier[47].

In C and C++, the `volatile` keyword was intended to allow C and C++ programs to directly access memory-mapped I/O. Memory-mapped I/O generally requires that the reads and writes specified in source code happen in the exact order specified with no omissions. Omissions or reorderings of reads and writes by the compiler would break the communication between the program and the device accessed by memory-mapped I/O. A C or C++ compiler may not omit reads from and writes to volatile memory locations, nor may it reorder read/writes relative to other such actions for the same volatile location (variable). The keyword `volatile` does not guarantee a memory barrier to enforce cache-consistency, however. Therefore, the use of `volatile` alone is not sufficient to use a variable for inter-thread communication on all systems and processors. Different architectures behave differently, so RTRM[48].

The C and C++ standards prior to C11 and C++11 do not address multiple threads (or multiple processors), and as such, the usefulness of `volatile` depends on the compiler and hardware.

[47] See Wikipedia.
[48] Read The Reference Manual.

Although `volatile` guarantees that the volatile reads and volatile writes will happen in the exact order specified in the source code, the compiler may generate code (or the processor may re-order execution) such that a volatile read or write is reordered with regard to non-volatile reads or writes, thus limiting its usefulness as an inter-thread flag or mutex. Preventing such is compiler specific, but some compilers, like gcc, will not reorder operations around in-line assembly code with volatile and memory tags, like in: `asm volatile ("" ::: "memory");` Moreover, it is not guaranteed that volatile reads and writes will be seen in the same order by other processors or cores due to caching, cache coherence protocol and relaxed memory ordering.

The gcc compiler has the `__sync_synchronize();` macro, which would simply translate into a hardware, processor-level, barrier, such as a fence operation on x86, or its equivalent in other architectures. The processor may also do various optimizations at runtime, the most important one is actually performing operations out-of-order. See the gcc documentation on volatile objects and the legacy __sync Built-in Functions.

Example: xv6 memory barrier use
To ensure correct memory ordering, xv6 uses the older __sync_synchronize() function, still available in gcc.

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    if(holding(lk))
    panic("acquire");

    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();

    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
```

14.2 Two-Phase Locking

In concurrency control of databases, transaction processing (transaction management), and various transactional applications (e.g., transactional memory and software transactional memory), both centralized and distributed, a transaction schedule is serializable if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transaction executions. It is considered the highest level of isolation between transactions, and plays an essen-

tial role in concurrency control. As such it is supported in all general purpose database systems. Strong strict two-phase locking (SS2PL) is a popular serializability mechanism utilized in most of the database systems (in various variants) since their early days in the 1970s[49].

14.3 Lock-less Data Structures

This section is from Preshing on Programming, An Introduction to Lock-Free Programming. This section is written from Preshing's perspective.

Lock-free programming is a challenge, not just because of the complexity of the task itself, but because of how difficult it can be to penetrate the subject in the first place.

I was fortunate in that my first introduction to lock-free (also known as *lockless*) programming was Bruce Dawson's excellent and comprehensive white paper, Lockless Programming Considerations. And like many, I've had the occasion to put Bruce's advice into practice developing and debugging lock-free code on platforms such as the Xbox 360.

Since then, a lot of good material has been written, ranging from abstract theory and proofs of correctness to practical examples and hardware details. I'll leave a list of references in the footnotes. At times, the information in one source may appear orthogonal to other sources: For instance, some material assumes sequential consistency, and thus sidesteps the memory ordering issues which typically plague lock-free C/C++ code. The new C++11 atomic library standard throws another wrench into the works, challenging the way many of us express lock-free algorithms.

In this post, I'd like to re-introduce lock-free programming, first by defining it, then by distilling most of the information down to a few key concepts. I'll show how those concepts relate to one another using flowcharts, then we'll dip our toes into the details a little bit. At a minimum, any programmer who dives into lock-free programming should already understand how to write correct multithreaded code using mutexes, and other high-level synchronization objects such as semaphores and events.

People often describe lock-free programming as programming without mutexes, which are also referred to as locks. That's true, but it's only part of the story. The generally accepted definition, based on academic literature, is a bit more broad. At its essence, lock-free is a property used to describe some code, without saying too much about how that code was actually written.

Basically, if some part of your program satisfies the following conditions, then that part can rightfully be considered lock-free. Conversely, if a given part of your code doesn't satisfy these conditions, then that part is not lock-free.

[49] See Wikipedia.

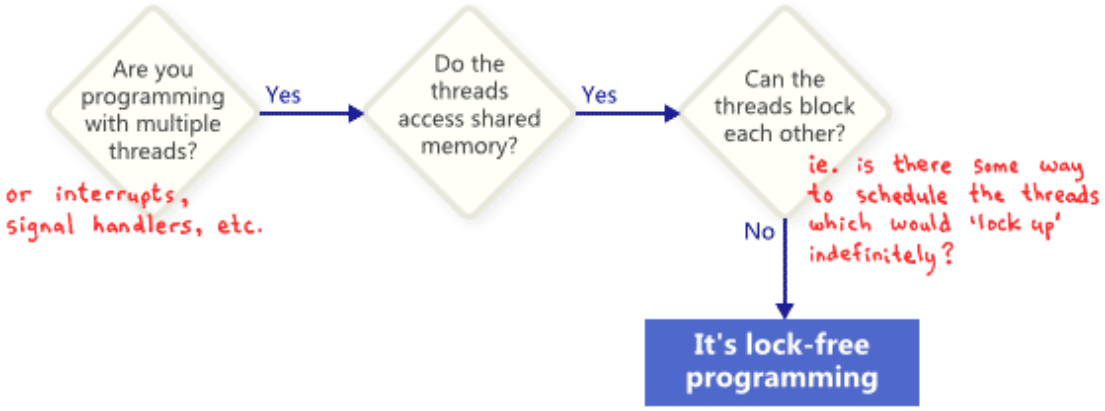

Figure 14: Lock-Free Flowchart

In this sense, the lock in lock-free does not refer directly to mutexes, but rather to the possibility of "locking up" the entire application in some way, whether it's deadlock, livelock – or even due to hypothetical thread scheduling decisions made by your worst enemy. That last point sounds funny, but it's key. Shared mutexes are ruled out trivially, because as soon as one thread obtains the mutex, your worst enemy could simply never schedule that thread again. Of course, real operating systems don't work that way – we're merely defining terms.

Here's a simple example of an operation which contains no mutexes, but is still not lock-free. Initially, `X = 0`. As an exercise for the reader, consider how two threads could be scheduled in a way such that neither thread exits the loop.

```
while (X == 0) {
    X = 1 - X;
}
```

Nobody expects a large application to be entirely lock-free. Typically, we identify a specific set of lock-free operations out of the whole code base. For example, in a lock-free queue, there might be a handful of lock-free operations such as `push`, `pop`, perhaps `isEmpty`, and so on.

Herlihy & Shavit, authors of The Art of Multiprocessor Programming, tend to express such operations as class methods, and offer the following succinct definition of lock-free: "In an infinite execution, infinitely often some method call finishes." In other words, as long as the program is able to keep calling those lock-free operations, the number of completed calls keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations.

One important consequence of lock-free programming is that if you suspend a single thread, it will never prevent other threads from making progress, as a group, through their own lock-free operations. This hints at the value of lock-free programming when writing interrupt handlers and real-time systems, where certain tasks must complete within a certain time limit, no matter what state the rest of the program is in.

A final precision: Operations that are designed to block do not disqualify the algorithm. For example, a queue's `pop` operation may intentionally block when the queue is empty. The remaining code paths can still be considered lock-free.

It turns out that when you attempt to satisfy the non-blocking condition of lock-free programming, a whole family of techniques fall out: atomic operations, memory barriers, avoiding the ABA problem, to name a few. This is where things quickly become diabolical.

So how do these techniques relate to one another? To illustrate, I've put together the following flowchart. I'll elaborate on each one below.

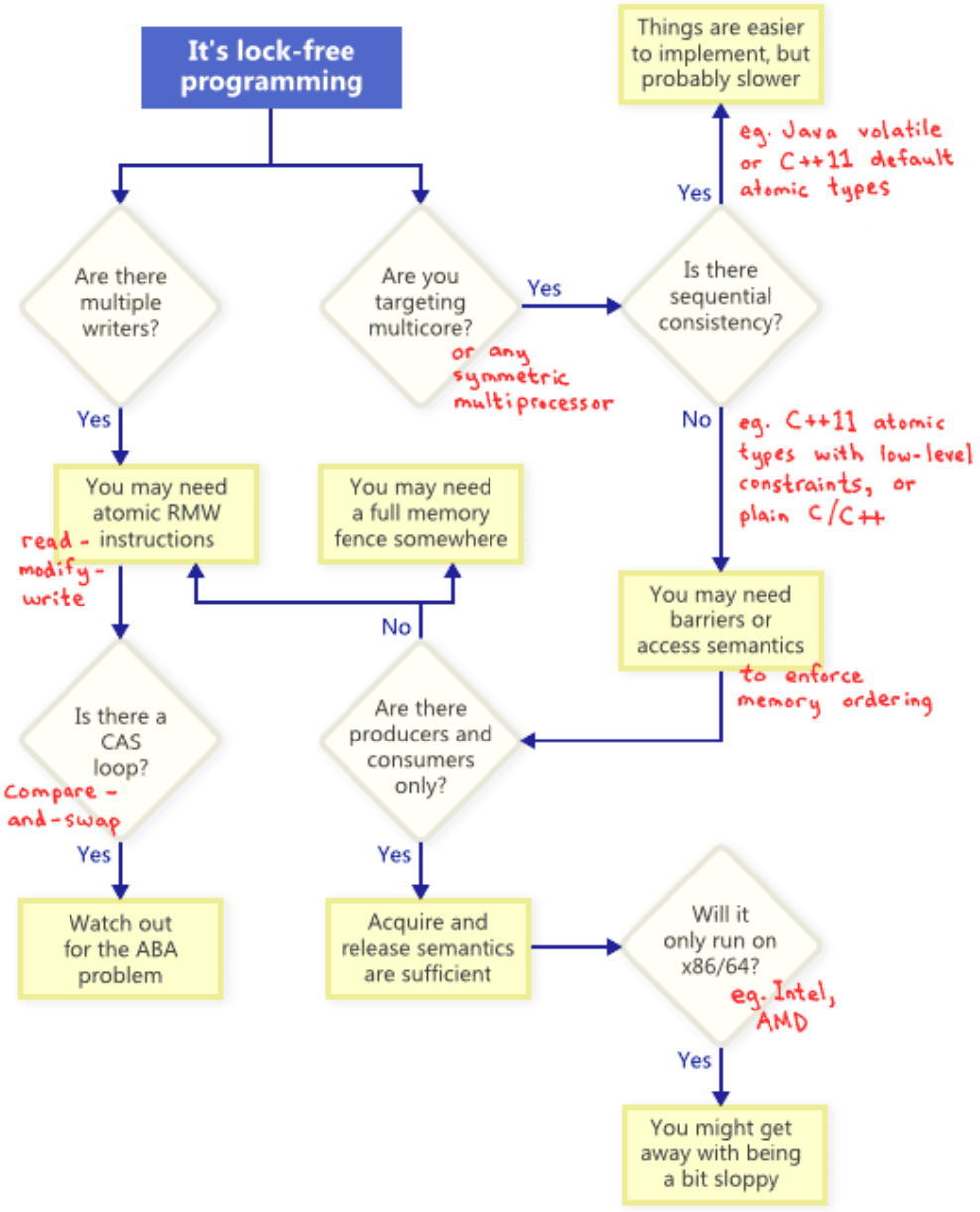

Figure 15: Lock-Free Techniques

14.3.1 Atomic Read-Modify-Write Operations

Atomic operations are ones which manipulate memory in a way that appears indivisible: No thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic.

Read-modify-write (RMW) operations go a step further, allowing you to perform more complex transactions atomically. They're especially useful when a lock-free algorithm must support multiple

writers, because when multiple threads attempt an RMW on the same address, they'll effectively line up in a row and execute those operations one-at-a-time. I've already touched upon RMW operations in this blog, such as when implementing a lightweight mutex, a recursive mutex and a lightweight logging system.

Examples of RMW operations include `_InterlockedIncrement` on Win32, `OSAtomicAdd32` on iOS, and `std::atomic<int>::fetch_add` in C++11. Be aware that the C++11 atomic standard does not guarantee that the implementation will be lock-free on every platform, so it's best to know the capabilities of your platform and tool chain. You can call `std::atomic<>::is_lock_free` to make sure.

Different CPU families support RMW in different ways. Processors such as PowerPC and ARM expose load-link/store-conditional instructions, which effectively allow you to implement your own RMW primitive at a low level, though this is not often done. The common RMW operations are usually sufficient.

As illustrated by the flowchart, atomic RMWs are a necessary part of lock-free programming even on single-processor systems. Without atomicity, a thread could be interrupted halfway through the transaction, possibly leading to an inconsistent state.

14.3.2 Compare-And-Swap Loops

Perhaps the most often-discussed RMW operation is compare-and-swap (CAS). On Win32, CAS is provided via a family of intrinsics such as `_InterlockedCompareExchange`. Often, programmers perform compare-and-swap in a loop to repeatedly attempt a transaction. This pattern typically involves copying a shared variable to a local variable, performing some speculative work, and attempting to publish the changes using CAS:

```
void LockFreeQueue::push(Node* newHead)
{
    for (;;) {
        // Copy a shared variable (m_Head) to a local.
        Node* oldHead = m_Head;

        // Do some speculative work, not yet visible to other threads.
        newHead->next = oldHead;

        // Next, attempt to publish our changes to the shared variable.
        // If the shared variable hasn't changed, the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
            return;
   }
}
```

Such loops still qualify as lock-free, because if the test fails for one thread, it means it must have succeeded for another – though some architectures offer a weaker variant of CAS where that's not necessarily true. Whenever implementing a CAS loop, special care must be taken to avoid the ABA problem.

14.3.3 Sequential Consistency

Sequential consistency means that all threads agree on the order in which memory operations occurred, and that order is consistent with the order of operations in the program source code. Under sequential consistency, it's impossible to experience memory reordering issues.

A simple (but obviously impractical) way to achieve sequential consistency is to disable compiler optimizations and force all your threads to run on a single processor. A processor never sees its own memory effects out of order, even when threads are preempted and scheduled at arbitrary times.

Some programming languages offer sequentially consistency even for optimized code running in a multiprocessor environment. In C++11, you can declare all shared variables as C++11 atomic types with default memory ordering constraints. In Java, you can mark all shared variables as `volatile`.

14.3.4 Memory Ordering

As the flowchart suggests, any time you do lock-free programming for multicore (or any symmetric multiprocessor), and your environment does not guarantee sequential consistency, you must consider how to prevent memory reordering.

On today's architectures, the tools to enforce correct memory ordering generally fall into three categories, which prevent both compiler reordering and processor reordering:

- A lightweight sync or fence instruction
- A full memory fence instruction
- Memory operations which provide acquire or release semantics

Acquire semantics prevent memory reordering of operations which follow it in program order, and release semantics prevent memory reordering of operations preceding it. These semantics are particularly suitable in cases when there's a producer/consumer relationship, where one thread publishes some information and the other reads it.

14.3.5 Different Processors Have Different Memory Models

Different CPU families have different habits when it comes to memory reordering. The rules are documented by each CPU vendor and followed strictly by the hardware. For instance, PowerPC and ARM processors can change the order of memory stores relative to the instructions themselves, but normally, the x86/64 family of processors from Intel and AMD do not. We say the former processors have a more *relaxed memory model*.

There's a temptation to abstract away such platform-specific details, especially with C++11 offering us a standard way to write portable lock-free code. But currently, I think most lock-free programmers have at least some appreciation of platform differences. If there's one key difference to remember, it's that at the x86/64 instruction level, every load from memory comes with acquire semantics, and every store to memory provides release semantics – at least for non-SSE instructions and non-write-combined memory. As a result, it's been common in the past to write lock-free code which works on x86/64, but fails on other processors.

14.4 Read-Copy-Update

Reader-writer locks do not scale well. The overhead to acquire the read lock can be a limiting factor. RCU is one approach to increasing scalability in a system with many readers and few writers, where read lock overhead is a limiting factor. One of leading names with regards to RCU is Paul McKinney[50] whose PhD topic was RCU. The Linux kernel makes heavy use of RCU, partially due to Paul during his time with a Linux research group at IBM in Beaverton, Oregon.

There are two important features that allow RCU to scale so well.

- A read lock doesn't lock. A reader identifier, such as a PID or ASID, is recorded so that RCU *knows* which processes are reading the data structure. The release on the lock *forgets* about the process. This is *very low overhead*.
- Updates by a single writer must be atomic. While this may seem a limiting constraint, in practice it is not.

The key differences between a regular reader-writer lock and RCU are:

1. Restricted update. The writer must *publish* its updates to the shared data structure with a single, atomic operation.
2. Multiple concurrent versions. RCU allows any number of readers to be in process *at the same time as the update*! This differs substantially from a reader-writer lock which can have no readers and only a single writer during an update.
3. Integration with the scheduler. The scheduler plays a key role. It tracks aspects of active readers of the shared data structure which is necessary for the *grace period* portion of RCU – this is the time after the publish until the last reader of the prior version of the data structure is *guaranteed* to have completed the critical section or moved to the new version of the shared data structure. It is critical that the old version not be removed until this guarantee is met. Think of the role that caches play here.

RCU Example

Let's walk through an example to see how RCU works. We will use the data structure shown in Figure 16.

[50] Paul receive his PhD in computer science from Oregon Graduate Institute, part of Oregon Health and Sciences University, in 2004 under Prof. Jon Walpole. He is the editor of Is Parallel Programming Hard, And, If So, What Can You Do About It?, an excellent e-book on concurrency. The current version of RCU is explained in §9.5 **Read-Copy Update (RCU)**. Paul is also an active contributor to the C++ standard.

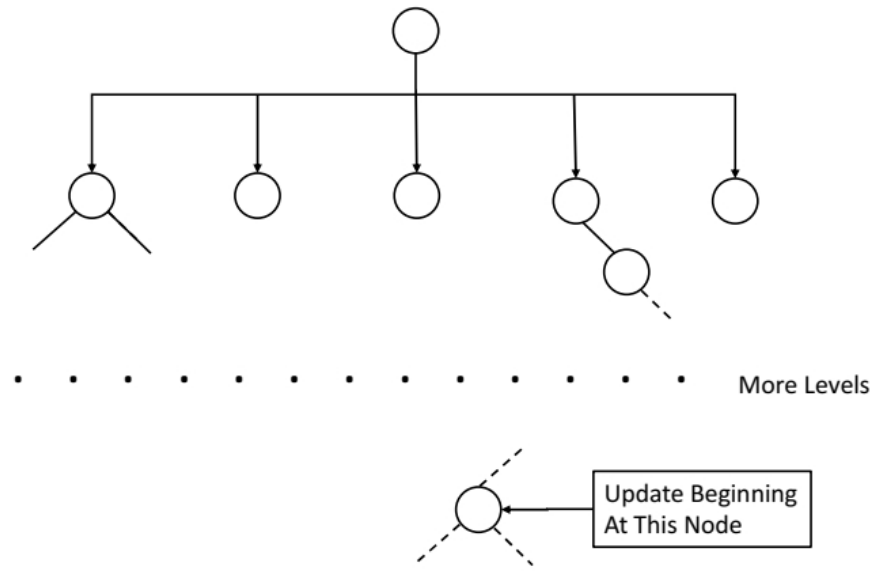

Figure 16: RCU Example Data Structure

A simple RCU implementation would work something like this:

1. There is a single reader lock and a single writer lock for the shared data structure in Figure 16.

2. The reader lock merely tracks which processes are current accessing the data structure. The set of *active readers* is denoted R. Without loss of generality, we will assume $R \neq \{\phi\}$. With a concurrent data structure, such as a lock-free list for tracking readers, this approach has very little overhead.

3. At some point, a process acquires a write lock on the data structure. This is a wait lock. Without loss of generality, we will assume that the lock is granted, exclusively, to some process P.

4. P modifies a *copy* of a node or subtree of the data structure. For example, the node of a the subtree pointed out in Figure 16. The operations performed by P for the update are not required to be atomic, or even transactional[51]. The only atomic operation is the publish.

5. Once P finishes with its update, P *publishes* the new node or subtree. The publish atomically updates the shared data structure at the designated node. It is important to note that there are now two versions of the subtree beginning at the designated node[52]. For convenience, we will call one $NODE_{n-1}$, the subtree before modification, and one $NODE_N$, the subtree after the publish, which is by definition after P completes all modifications for this write operation.

[51] Students should understand why transactional semantics are not required.
[52] Students should understand *why*. Hint: caches.

6. Any existing member of R will see the subtree as it existed in $NODE_{n-1}$ or $NODE_n$. Once a member of R sees $NODE_n$, it will no longer see $NODE_{n-1}$.

7. After some time, called the *grace period*, which requires coordination with the scheduler to guarantee, no $p \in R$ will see $NODE_{n-1}$ and so this subtree may be freed.

8. It is important to note that the set R can continue to have members added or removed, and may, in fact become $\{\phi\}$ during the grace period. However, it is known that no $p \in R$ sees $NODE_{n-1}$ once the grace period has expired. Note that the length of the grace period is not computable in advance.

Alternate Approach

Locking the entire data structure by the writer is not efficient. There are variations where the writer first obtains a reader lock to the data structure and, once the correct node is found, a writer lock is obtained for that subtree. This allows for multiple, concurrent writes to the shared data structure. Students should study this alternate and ask be prepared to answer questions about it.

15 Study Questions

1. Explain why concurrency is or is not a superset of parallelism. Give one example of parallelism and one of non-parallel concurrency. Details matter.
2. Explain why concurrency control in not necessary for read-only variables.
3. For the Dining Philosophers Problem, assume that we have a single lock and an ordered wait queue. When a thread gets to the head of the list, if both utensils are not available, then the thread goes to the end of the queue. Explain why this approach solves both deadlock and starvation.
4. A semaphore is more general than a lock. Can a semaphore, without a timeout, be used to implement a spinlock? Explain.
5. Explain how a semaphore with a timeout can be used to approximate a spin lock.
6. Explain the term "critical section".
7. Explain the term "race condition".
8. Explain how to protect against non-determinism in race conditions.
9. In general, can locks be implemented by simply reading and writing a binary variable in memory? Explain.
10. Does disabling interrupts work well for concurrency control on a uniprocessor system? Explain.
11. Does disabling interrupts for concurrency control work on a multiprocessor or multicore system? Explain.
12. Explain why hardware support necessary for mutual exclusion?
13. Explain why it is sometime better to block rather than spin on a uniprocessor.
14. Explain why it is sometime better to spin rather than block on a multiprocessor.
15. How do we evaluate lock implementations? Why in a specific order?
16. Is the critical section limited to where shared data is modified? Explain.
17. Explain why busy waiting can less efficient than a blocking wait.
18. Explain why busy waiting can be more efficient than a blocking wait.
19. Explain why deadlock prevention is "conservative".
20. Explain why deadlock avoidance is "optimistic".
21. Explain how deadlock detection and recovery works.
22. (T/F) Starvation cannot occur if the resource allocation graph is acyclic.

23. (T/F) A reader-writer lock that is *write-preferring* will allow multiple writers simultaneous access to the critical section and block any readers while a writer is active.

24. Explain the strengths and weaknesses of recovery from deadlock in a deadlock avoidance scheme.

25. Explain why a single lock can prevent deadlock for the Dining Philosophers problem, but it cannot prevent starvation.

26. For the Dining Philosophers Problem, explain how the introduction of a single left-handed philosopher among a group of right-handed philosophers solves deadlock.

27. Describe the four principle parts of a semaphore.

28. Explain how can atomicity be guaranteed in the internal implementation of a semaphore.

29. Can a process invoking `wait()` on a semaphore signal the same semaphore at a later time without any other process signaling the same semaphore first? Explain.

30. Provide correct pseudo-code for a semaphore solution to the Dining Philosophers problem.

31. Explain what a monitor is and how it is used.

32. Can message passing prevent deadlock? Explain.

33. Can message passing prevent starvation? Explain.

34. How does a shell (e.g., bash) implement I/O redirection? Be sure to cover both `stdin` and `stdout`.

35. Explain how mutual exclusion can be implemented for an arbitrary number of processes (not threads) that share the same memory region.

36. Explain why determinism is a concern for programmers in a multi-threaded environment.

37. Explain what is meant by the term "atomic transaction".

38. Reader-Writer locks can be "read preferring" or "write preferring". Explain the key differences between these two approaches.

39. Explain the purpose of the lock argument to the `wait()` routine for a condition variable.

40. Define "livelock" and give an example.

41. Explain how to detect that a process is in livelock.

42. In our lock ordering example there was this statement:

> Note that the decision to order the locks alphabetically by the person's name would work fine for this book, but it wouldn't work in a real life social network.

    Explain the problem and suggest a solution.

43. Discuss the difficulties in debugging a program that has one or more concurrency bugs.

44. In concurrent programming, why is it critical to develop a solid design before any programming?

45. There is an approach to software development called *iterative development*[53], which your instructor claims is the best approach for concurrent programming. Provide a reasoned argument either supporting or refuting this statement. Any refutation must include an alternate that you consider to be a better approach. Note that there is no one right or wrong answer to this question – however, you must *think* about the overall problem in some depth.

46. **Advanced Concurrency Control questions not yet available.**

Consider the following C program fragment. Threads T1 and T2 concurrently run in the same address space and complete the functions with corresponding names. Assume that lock is initially unlocked, and is an xv6 spinlock as described in the lecture notes. This question asks you about the state of the process after both threads have completed. Note that x and y are allocated in the static data segment and initially have the value 0; you may treat them as if they are on the heap for this question, and are word-sized and word-aligned.

```
int x = 0;
int y = 0;

void T1() {
    acquire(\&lock);
    x = x + 1;
    y = y + 1;
    release(\&lock);
}

void T2() {
    acquire(\&lock);
    x = x + 1;
    y = y + 1;
    release(\&lock);
}
```

47. What values could (x, y) have after both threads complete? Select all that apply:

    (a) (0, 0)
    (b) (0, 1)
    (c) (1, 0)
    (d) (1, 1)
    (e) (0, 2)
    (f) (2, 0)
    (g) (1, 2)

[53] See Wikipedia and Airbrake.

(h) (2, 1)

(i) (2, 2)

The following question asks you to consider a C program fragment. Threads T1 and T2 concurrently run in the same address space and complete the functions with corresponding names. This question asks you about the state of the process after both threads have completed. Note that x is allocated in the static data segment; you may treat it as if it is on the heap, and assume that it is an aligned 32-bit word.

```
int x = 0;

void T1() {
    x = x + 1;
}

void T2() {
    int r1 = x;
}
```

48. What value(s) could r1 have when T2() completes? Select any that apply:

(a) 0

(b) 1

(c) 2

The following question asks you to consider a C program fragment. Threads T1 and T2 concurrently run in the same address space. This question asks you about the state of the process after both threads have completed in two cases: when T1 runs T1A(), and when T1 runs T1B(). In both cases, T2 runs T2(). Assume that lock is initially unlocked, and is an xv6 spinlock as described in the lecture notes. Note that x is allocated in the static data segment; you may treat it as if it is on the heap, and assume that it is an aligned 32-bit word.

```
int x = 0;

void T1A() {
    x = x + 1;
}

void T1B() {
    acquire(\&lock);
    x = x + 1;
    release(\&lock);
}

void T2() {
```

```
    int r1 = x;
}
```

49. Consider the possible values that r1 could have when T2 completes, and mark the following statement as true or false: Whether T1 runs T1A() or T1B() does not change the possible values T2 can have in r1 when it completes.

The following question asks you to consider a C program fragment. Threads T1, T2, and T3 concurrently run in the same address space and complete the functions with corresponding names. This question asks you about the state of the process after all threads have completed. Note that x and y are allocated in the static data segment; you may treat them as if they are on the heap, and assume that they are aligned 32-bit words.

```
int x = 0;
int y = 0;

void T1() {
    x = x + 1;
}

void T2() {
    y = y + 1;
}

void T3() {
    int r1 = x;
    int r2 = y;
}
```

50. What value(s) could (r1, r2) have when T2() completes? Select any that apply:

(a) (0, 0)

(b) (0, 1)

(c) (1, 0)

(d) (1, 1)