Synchronization and its pitfalls

Jeff Chase
Duke University
A thread

When a thread is blocked its TCB is placed on a **sleep queue** of threads waiting for a specific wakeup event.

This slide applies to the process abstraction too, or, more precisely, to the main thread of a process.

**Thread Control Block**

- **TCB**
- **ucontext_t**
- **user stack**

Storage for context (register values) when thread is not running.

**Wait**

- **active**
- **ready or running**

- **sleep**
- **wait**

- **wakeup**
- **signal**

When a thread is **blocked** its TCB is placed on a **sleep queue** of threads waiting for a specific wakeup event.
Threads are orthogonal to address spaces
Thread states and transitions

If a thread is in the **ready** state thread, then the system may choose to run it “at any time”. When a thread is running, the system may choose to preempt it at any time. From the point of view of the program, dispatch and preemption are **nondeterministic**: we can’t know the **schedule** in advance.

These **preempt** and **dispatch** transitions are controlled by the kernel scheduler.

**Sleep** and **wakeup** transitions are initiated by calls to internal sleep/wakeup APIs by a running thread.
An Introduction to Programming with C# Threads

Andrew D. Birrell

This paper provides an introduction to writing concurrent programs with "threads". A threads facility allows you to write programs with multiple simultaneous points of execution, synchronizing through shared memory. The paper describes the basic thread and synchronization primitives, then for each primitive provides a tutorial on how to use

```csharp
    Thread t = new Thread(new ThreadStart(foo.A));
    t.Start();
    foo.B();
    t.Join();
```
C# lock (mutex)

class KV {
    string k, v;
    public void SetKV(string nk, string nv) {
        lock (this) {
            this.k = nk; this.v = nv;
        }
    }
    ...
}

An Introduction to Programming with C# Threads
C# monitors

```csharp
public sealed class Monitor {
    public static bool Wait(Object obj) { ... }
    public static void Pulse(Object obj) { ... }
    public static void PulseAll(Object obj) { ... }
}
```

Monitors are a common underlying mechanism for programming synchronization objects.

A thread that calls “Wait” must already hold the object’s lock (otherwise, the call of “Wait” will throw an exception). The “Wait” operation atomically unlocks the object and blocks the thread*. A thread that is blocked in this way is said to be “waiting on the object”. The “Pulse” method does nothing unless there is at least one thread waiting on the object, in which case it awakens at least one such waiting thread (but possibly more than one). The “PulseAll” method is like “Pulse”, except that it awakens all the threads currently waiting on the object. When a thread is awoken inside “Wait” after blocking, it re-locks the object, then returns.

An Introduction to Programming with with C# Threads
For example, we could add the following “GetFromList” method to the class “KV”. This method waits until the linked list is non-empty, and then removes the top item from the list.

```csharp
public static KV GetFromList() {
    KV res;
    lock (typeof(KV)) {
        while (head == null) Monitor.Wait(typeof(KV));
        res = head; head = res.next;
        res.next = null; // for cleanliness
    }
    return res;
}
```

And the following revised code for the “AddToList” method could be used by a thread to add an object onto “head” and wake up a thread that was waiting for it.

```csharp
public void AddToList() {
    lock (typeof(KV)) {
        /* We're assuming this.next == null */
        this.next = head; head = this;
        Monitor.Pulse(typeof(KV));
    }
}
```

*An Introduction to Programming with C# Threads*
Commandment 9. *Thou shalt cover thy naked waits.*

Commandment 10. *Thou shalt guard your wait predicates in a while loop. Thou shalt never guard a wait statement with an if statement.*
The "missed wakeup problem" occurs when a thread calls an internal sleep() primitive to block, and another thread calls wakeup() to awaken the sleeping thread in an unsafe fashion. For example, consider the following pseudocode snippets for two threads:

**Sleeper thread**

```
S1 { Thread sleeper = self();
    listMx.lock();
    list.put(sleeper);
    listMx.unlock();
    sleeper.sleep();
}
S2
```

**Waker thread**

```
W1 { listMx.lock();
    Thread sleeper = list.get();
    listMx.unlock();
    sleeper.wakeup();
}
W2
```

(a) What could go wrong? Outline how this code is vulnerable to the missed wakeup problem, and illustrate with an example schedule.

One possible schedule is [S1, S2, W1, W2]. This is the intended behavior: the sleeper puts itself (a reference to its Thread object) on a list and sleeps, and the waker retrieves the sleeping thread from the list and then wakes that sleeper up.

These snippets could also execute in some schedule with W1 < S1 (W1 happens before S1) for the given sleeper. In this case, the waker does not retrieve the sleeper from the list, so it does not try to wake it up. It wakes up some other sleeping thread, or the list is empty, or whatever.

The schedule of interest is [S1, W1, W2, S2]. In this case, the sleeper is on the list, and the waker retrieves that sleeper from the list and issues a wakeup call on that sleeper, as in the first schedule. But the sleeper is not asleep, and so the wakeup call may be lost or it may execute incorrectly. This is the missed wakeup problem.

Note that these raw sleep/wakeup primitives, as defined, are inherently unsafe and vulnerable to the missed wakeup problem. That is why we have discussed them only as "internal" primitives to illustrate blocking behavior: we have not studied them as part of any useful concurrency API. The point of the question is that monitors and semaphores are designed to wrap sleep/wakeup in safe higher-level abstractions that allow threads to sleep for events and wake other threads when those events occur. Both abstractions address the missed wakeup problem, but they resolve the problem in different ways.
What could go wrong?

Consider schedule [S1, W1, W2, S2]. In this case, the sleeper is on the list, and the waker retrieves that sleeper from the list and issues a wakeup call on that sleeper. But the sleeper is not asleep, and so the wakeup call may be lost or it may execute incorrectly. This is the missed wakeup problem. Condition variables are designed to solve it.
(b) How does blocking with monitors (condition variables) avoid the missed wakeup problem? Illustrate how the code snippets in (a) might be implemented using monitors, and outline why it works.

Monitors (condition variables) provide a higher-level abstraction: instead of using raw sleep and wakeup, we use wait() and signal/notify(). These primitives serve the desired purpose, but the wait() primitive is integrated with the locking, so that the sleeper may hold the mutex until the sleep is complete. The implementation of wait() takes care of releasing the mutex atomically with the sleep. For example:

```c
listMx.lock();
sleeper++;
listCv.wait();
sleeper--;
listMx.unlock();
```

In these snippets we presume that the condition variable listCv is bound to the mutex listMx. Various languages show this with various syntax. I didn't require it for full credit.

In this example, the sleeper's snippet may execute before or after the waker, but it is not possible for the waker to see a sleeper's count (sleeper > 0) and then fail to wake a/the sleeper up. The missed wakeup problem cannot occur.
SharedLock: Reader/Writer Lock

A reader/write lock or SharedLock is a new kind of “lock” that is similar to our old definition:

- supports Acquire and Release primitives
- assures mutual exclusion for writes to shared state

But: a SharedLock provides better concurrency for readers when no writer is present.

class SharedLock {
    AcquireRead(); /* shared mode */
    AcquireWrite(); /* exclusive mode */
    ReleaseRead();
    ReleaseWrite();
}

Multiple readers may hold the lock concurrently in **shared** mode.

If each thread acquires the lock in **exclusive** (*write*) mode, *SharedLock* functions exactly as an ordinary mutex.

Writers always hold the lock in **exclusive** mode, and must wait for all readers or writer to exit.

<table>
<thead>
<tr>
<th>mode</th>
<th>read</th>
<th>write</th>
<th>max allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>yes</td>
<td>no</td>
<td>many</td>
</tr>
<tr>
<td>exclusive</td>
<td>yes</td>
<td>yes</td>
<td>one</td>
</tr>
<tr>
<td>not holder</td>
<td>no</td>
<td>no</td>
<td>many</td>
</tr>
</tbody>
</table>
Reader/Writer Lock: outline

```c
int i; /* # active readers, or -1 if writer */

void AcquireWrite() {
    while (i != 0)
        sleep....;
    i = -1;
}

void AcquireRead() {
    while (i < 0)
        sleep....;
    i += 1;
}

void ReleaseWrite() {
    i = 0;
    wakeup....;
}

void ReleaseRead() {
    i -= 1;
    if (i == 0)
        wakeup....;
}
```
Reader/Writer Lock: adding a little mutex

```c
int i;       /* # active readers, or -1 if writer */
Lock rwMx;

AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        sleep…;
    i = -1;
    rwMx.Release();
}

AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        sleep…;
    i += 1;
    rwMx.Release();
}

ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    wakeup…;
    rwMx.Release();
}

ReleaseRead() {
    rwMx.Acquire();
    i -= 1;
    if (i == 0)
        wakeup…;
    rwMx.Release();
}
```
Reader/Writer Lock: cleaner syntax

int i; /* # active readers, or -1 if writer */

synchronized AcquireWrite() {
    while (i != 0)
        wait();
    i = -1;
}

synchronized AcquireRead() {
    while (i < 0)
        wait();
    i += 1;
}

synchronized ReleaseWrite() {
    i = 0;
    notifyAll();
}

synchronized ReleaseRead() {
    i -= 1;
    if (i == 0)
        notify();
}

We can use Java syntax for convenience. That's the beauty of pseudocode. We use any convenient syntax. These syntactic variants have the same meaning.
The Little Mutex Inside SharedLock
Limitations of the SharedLock Implementation

This implementation has weaknesses; see [Birrell89/03].

• **spurious lock conflicts** (on a multiprocessor): multiple waiters contend for the mutex after a signal or broadcast.
  
  *Solution*: drop the mutex before signaling (if permitted).

• **spurious wakeups**

  *ReleaseWrite* awakens writers as well as readers.
  
  *Solution*: add a separate condition variable for writers.

• **starvation**

  How can we be sure that a writer can ever acquire if faced with a continuous stream of arriving readers?
A note on “thundering herd”

Thundering herd: an event occurs (e.g., a CV broadcast) and many threads wake up as a result, where only one of them can actually consume/handle the event. The others are forced to go back to sleep, and their work is wasted. Thus the system is briefly under heavy load while a "herd of threads thunders through", like a herd of cattle stampeding.

SharedLock (reader/writer lock) illustrates the problem when you use the same CV for readers and writers. ReleaseWrite broadcasts on the CV because it wants to wake all the waiting readers. But there might also be one or more waiting writers. If a writer gets the lock first, then all the others must wait again. The "second try" fixes this.

Birrell discusses the reader/writer example in detail. Thundering herd is related to Birrell’s "spurious wake ups".
Reader/Writer Lock: Second Try

SharedLock::AcquireWrite() {
    rwMx.Acquire();
    while (i != 0)
        wCv.Wait(&rwMx);
    i = -1;
    rwMx.Release();
}

SharedLock::AcquireRead() {
    rwMx.Acquire();
    while (i < 0)
        ...rCv.Wait(&rwMx);...
    i += 1;
    rwMx.Release();
}

SharedLock::ReleaseWrite() {
    rwMx.Acquire();
    i = 0;
    if (readersWaiting)
        rCv.Broadcast();
    else
        wCv.Signal();
    rwMx.Release();
}

SharedLock::ReleaseRead() {
    rwMx.Acquire();
    i -= 1;
    if (i == 0)
        wCv.Signal();
    rwMx.Release();
}

Use two condition variables protected by the same mutex.
We can’t do this in Java, but we can still use Java syntax in our pseudocode. Be sure to declare the binding of CVs to mutexes!
Reader/Writer Lock: Second Try

synchronized \textbf{AcquireWrite}() {
    while (i != 0)
        wCv.Wait();
    i = -1;
}

synchronized \textbf{AcquireRead}() {
    while (i < 0) {
        readersWaiting+=1;
        rCv.Wait();
        readersWaiting-=1;
    }
    i += 1;
}

synchronized \textbf{ReleaseWrite}() {
    i = 0;
    if (readersWaiting)
        rCv.Broadcast();
    else
        wCv.Signal();
}

synchronized \textbf{ReleaseRead}() {
    i -= 1;
    if (i == 0)
        wCv.Signal();
}

wCv and rCv are protected by the monitor mutex.
Starvation

• The reader/writer lock example illustrates starvation: under load, a writer might be stalled forever by a stream of readers.

• Example: a one-lane bridge or tunnel.
  – Wait for oncoming car to exit the bridge before entering.
  – Repeat as necessary…

• Solution: some reader must politely stop before entering, even though it is not forced to wait by oncoming traffic.
  – More code…
  – More complexity…
Sellshort (writer priority)

synchronized EnterC() {
    while (i != 0) {
        cw++;
        wCv.Wait();
        cw--;
    }
    i -= 1;  /* -1 */
}

synchronized EnterE() {
    while (i < 0 || cw) {
        rCv.Wait();
    }
    i += 1;
}

synchronized ExitC() {
    i += 1;  /* 0 */
    if (cw)
        wCv.Signal();
    else
        rCv.Broadcast();
}

synchronized ExitE() {
    i -= 1;
    if (i == 0 && cw)
        wCv.Signal();
}
A note on “loop before you leap”

**Loop before you leap** is my slogan for the looping requirement of CVs with Mesa semantics: when a thread wait()s for a condition to become true, it can't be sure that the condition is in fact true when it returns from wait(). This it might have to loop and wait again.

SharedLock and soda machine illustrate “loop before you leap”: every wait() is in a while loop that rechecks the condition each time it wakes up, before continuing ("leaping") into the rest of the program. Be sure that you understand why this is necessary (see the slide/s on Mesa semantics).

Note also that if your CV programs loop before leaping, that also protects you against “sloppy” use of broadcast/notifyAll and of condition variables that represent multiple conditions. This can simplify programs considerably.

In particular, it is correct to use notifyAll() if all waiters "loop before leaping". If the wakeup was spurious, they will just go back into the wait(). Similarly, if multiple threads wait on the same condition variable for different reasons, or if other threads signal/notify the condition variable early, then some threads may wake up before the condition they are waiting for is true. That is OK as long as they loop before leaping.

In fact, a good way to write correct code quickly is to always use notifyAll/broadcast and always loop before leaping. It always works, although it might be slow (due to thundering herds). But that’s OK: correctness first, then performance.
Dining Philosophers

• $N$ processes share $N$ resources
• resource requests occur in pairs w/ random think times
• hungry philosopher grabs fork
• ...and doesn’t let go
• ...until the other fork is free
• ...and the linguine is eaten

```java
while(true) {
    Think();
    AcquireForks();
    Eat();
    ReleaseForks();
}
```
Resource Graph or Wait-for Graph

- A vertex for each process and each resource
- If process $A$ holds resource $R$, add an arc from $R$ to $A$. 

A grabs fork 1

B grabs fork 2
Resource Graph or Wait-for Graph

- A vertex for each process and each resource
- If process $A$ holds resource $R$, add an arc from $R$ to $A$.
- If process $A$ is waiting for $R$, add an arc from $A$ to $R$.

A grabs fork 1 and waits for fork 2.

B grabs fork 2 and waits for fork 1.
Resource Graph or Wait-for Graph

- A vertex for each process and each resource
- If process $A$ holds resource $R$, add an arc from $R$ to $A$.
- If process $A$ is waiting for $R$, add an arc from $A$ to $R$.

The system is deadlocked iff the wait-for graph has at least one cycle.

A grabs fork 1 and waits for fork 2.

B grabs fork 2 and waits for fork 1.
Deadlock vs. starvation

- A **deadlock** is a situation in which a set of threads are all waiting for another thread to move.
- But none of the threads can move because they are all waiting for another thread to do it.
- Deadlocked threads sleep “forever”: the software “freezes”. It stops executing, stops taking input, stops generating output. **There is no way out.**
- **Starvation** (also called **livelock**) is different: some schedule exists that can exit the livelock state, and the scheduler may select it, even if the probability is low.
There are really only 9 states we care about: the key transitions are acquire and release events.
Two Philosophers Living Dangerously
The Inevitable Result

This is a deadlock state: There are no legal transitions out of it.
Four Conditions for Deadlock

Four conditions must be present for deadlock to occur:

1. **Non-preemption of ownership.** Resources are never taken away from the holder.

2. **Exclusion.** A resource has at most one holder.

3. **Hold-and-wait.** Holder blocks to wait for another resource to become available.

4. **Circular waiting.** Threads acquire resources in different orders.
Not All Schedules Lead to Collisions

• The scheduler+machine choose a schedule, i.e., a trajectory or path through the graph.
  – Synchronization constrains the schedule to avoid illegal states.
  – Some paths “just happen” to dodge dangerous states as well.

• What is the probability of deadlock?
  – How does the probability change as:
    • think times increase?
    • number of philosophers increases?
Dealing with Deadlock

1. **Ignore it.** Do you feel lucky?
2. **Detect and recover.** Check for cycles and break them by restarting activities (e.g., killing threads).
3. **Prevent it.** Break any precondition.
   - Keep it simple. Avoid blocking with any lock held.
   - Acquire nested locks in some predetermined order.
   - Acquire resources in advance of need; release all to retry.
   - Avoid “surprise blocking” at lower layers of your program.
4. **Avoid it.**
   - Deadlock can occur by allocating variable-size resource chunks from bounded pools: google “Banker’s algorithm”.

[Google Banker’s algorithm](https://en.wikipedia.org/wiki/银行家算法)
File abstraction

Program A
- open "/a/b"
- write ("abc")
- read

Library

Program B
- open "/a/b"
- read
- write ("def")

system call trap/return

OS kernel
Unix Pipes

Example: cat | cat

cat pseudocode (user mode)
while (until EOF) {
    read (0, buf, count);
    compute/transform data in buf;
    write (1, buf, count);
}

Kernel pseudocode for pipes:
Producer/consumer bounded buffer

Pipe write: copy in bytes from user buffer to in-kernel pipe buffer, blocking if k-buffer is full.

Pipe read: copy bytes from pipe's k-buffer out to u-buffer. Block while k-buffer is empty, or return EOF if empty and pipe has no writer.

Example: cat | cat
Kernel-space pseudocode
System call internals to read/write N bytes for buffer size B.

read(buf, N)
{
    for (i = 0; i++ < N) {
        move one byte into buf[i];
    }
}
Pipes

read(buf, N)
{
    pipeMx.lock();
    for (i = 0; i++ < N) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
}

Read N bytes from the pipe into the user buffer named by buf. Think of this code as deep inside the implementation of the read system call on a pipe. The write implementation is similar.
Pipes

```c
read(buf, N)
{
    readerMx.lock();
    pipeMx.lock();
    for (i = 0; i++; i<N) {
        while (no bytes in pipe)
            dataCv.wait();
        move one byte from pipe into buf[i];
        spaceCV.signal();
    }
    pipeMx.unlock();
    readerMx.unlock();
}
```

In Unix, the **read/write** system calls are “atomic” in the following sense: no **read** sees interleaved data from multiple **writes**. The extra lock here ensures that all read operations occur in a serial order, even if any given operation blocks/waits while in progress.
Why exactly does Pipe (bounded buffer) require a nested lock?

**First:** remember that this is the exception that proves the rule. Nested locks are generally not *necessary*, although they may be useful for performance. Correctness first: always start with a single lock.

**Second:** the nested lock is not necessary even for Pipe if there is at most one reader and at most one writer, as would be the case for your typical garden-variety Unix pipe.

The issue is what happens if there are multiple readers and/or multiple writers. The nested lock is needed to meet a requirement that read/write calls are *atomic*. Understanding this requirement is half the battle.

Consider an example. Suppose three different writers \{A, B, C\} write 10 bytes each, each with a single write operation, and a reader reads 30 bytes with a single read operation. The read returns the 30 bytes, so the read will "see" data from multiple writes. That's OK. The atomicity requirement is that the reader does not observe bytes from different writes that are *interleaved* (mixed together).

A necessary condition for atomicity is that the writes are *serialized*: the system chooses some order for the writes by A, B, and C, even if they request their writes "at the same time". The data returned by the read reflects this ordering. Under no circumstances does a read see an interleaving, e.g.: 5 bytes from A, then 5 bytes from B, then 5 more bytes from A,… (Note: if you think about it, you can see that a correct implementation must also serialize the reads.)

This atomicity requirement exists because applications may depend on it: e.g., if the writers are writing records to the pipe, then a violation of atomicity would cause a record to be "split".

This is particularly important when the size of a read or write (N) exceeds the size of the bounded buffer (B), i.e., N>B. A read or write with N>B is legal. But such an operation can’t be satisfied with a single buffer’s worth of data, so it can’t be satisfied without alternating execution of a reader and a writer ("ping-pong style"). On a single core, the reader or writer is always forced to block at least once to wait for its counterparty to place more bytes in the buffer (if the operation is a read) or to drain more bytes out of the buffer (if the operation is a write). In this case, it is crucial to block any other readers or writers from starting a competing operation. Otherwise, atomicity is violated and at least one of the readers will observe an interleaving of data.

The nested lock ensures that at most one reader and at most one writer are moving data in the “inner loop” at any given time.