
Concurrency and synchronization

CSE513 · Week 3

Prof. Dr. Mehmet Göktürk

Operating System Concepts · Chapters 6–7

The two halves of this lecture

First half: trace conflicting operations, specify invariants, and protect critical sections with atomic operations and mutexes.

Second half: coordinate waiting with semaphores and condition variables; apply them to bounded buffers and shared resources.

By the end, explain why an execution is correct, rather than relying on one successful run.

Week 2 asked which ready thread runs next. Week 3 asks which interleavings are safe when threads share data.

On one CPU, preemption can interrupt an update. On several CPUs, updates may overlap in real time.

A scheduler does not know the application invariant. A time slice is not a critical section.

Threads share their process's address space, including globals and heap objects. Each thread has its own execution state and stack.

A local pointer can refer to shared storage. Where a pointer is declared does not determine who can access its target.

Identify the shared object, the operations on it, and the rule that must remain true before choosing a lock.

A shared counter

Two workers each increment a counter 100,000 times. After both workers finish, the intended total is 200,000.

Conceptually, increment means read the old value, add one, and write the new value. These steps need not form one indivisible operation.

We first trace an abstract machine with atomic reads/writes and sequential consistency; this is not a promise about racy C.

A lost update

Initial counter = 0. Both threads perform one increment.

A reads 0; B reads 0; A computes 1; B computes 1; A writes 1; B writes 1.

Final counter = 1 although two increments were requested. The later store overwrites an update based on the same old value.

The defect is the unprotected read–modify–write sequence, not an arithmetic error.

A serialized update

A reads 0, computes 1, and writes 1 before B reads the counter. B then reads 1, computes 2, and writes 2.

Final counter = 2. This schedule succeeds even when the algorithm permits the failing schedule.

One correct execution does not prove correctness for every allowed interleaving.

Predict an interleaving

Model each increment as two atomic steps: read the counter into a private register; write register + 1.

Two threads perform one increment each. Preserve each thread's read-before-write order.

List the six possible interleavings. Which finish with 1, and which finish with 2?

Interleaving answer

AABB and BBAA serialize the increments and finish with 2.

ABAB, ABBA, BAAB and BABA let both threads read 0 before either completes; they finish with 1.

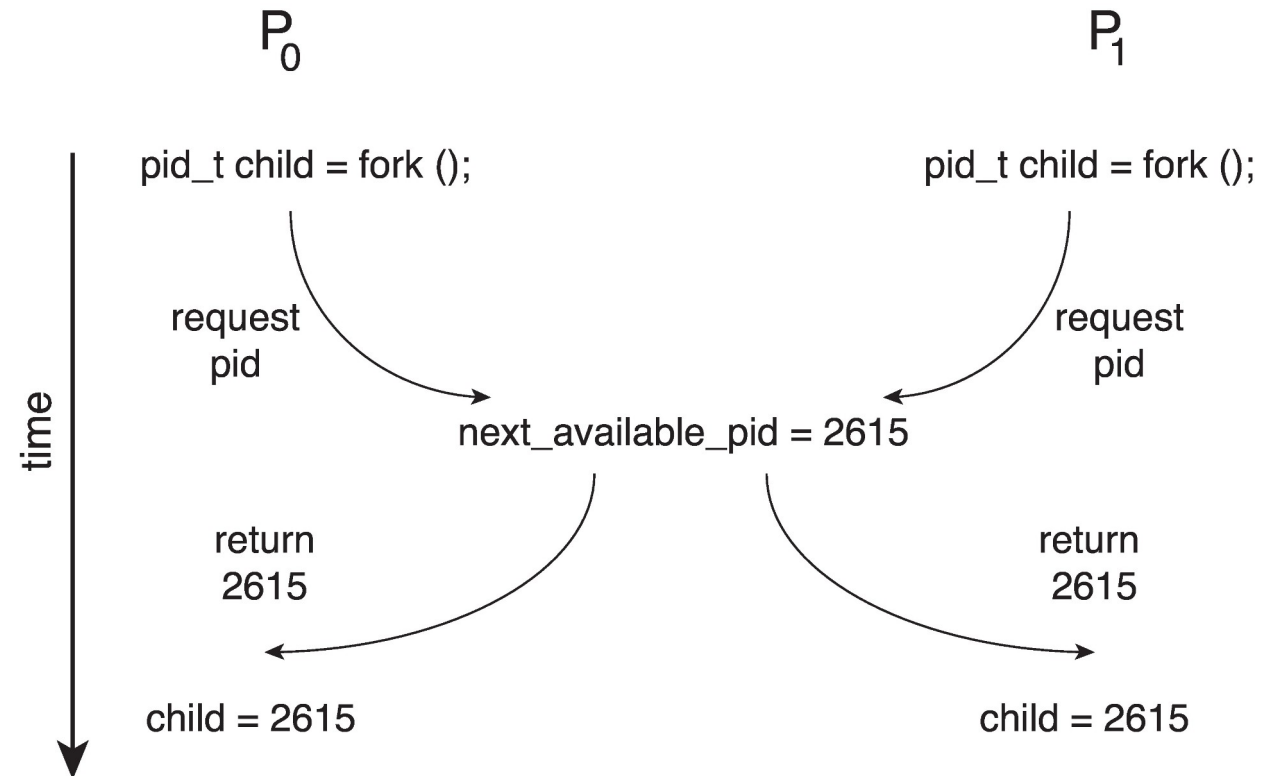
These are six abstract schedules, not six equally likely executions. Do not infer a failure probability from this enumeration.

A resource-allocation race

The textbook illustration shows two requests observing the same next-available identifier.

The check and reservation must be coordinated. Merely reading the identifier atomically does not reserve it.

This is a conceptual failure scenario, not evidence that current Linux fork() allocates duplicate live PIDs.



Race condition and C data race

A race condition is a correctness failure that depends on timing or ordering. It can occur even when individual accesses are atomic.

In C, conflicting accesses in different threads, at least one non-atomic, without the required ordering constitute a data race: behaviour is undefined.

Use abstract traces to teach lost updates. Use a mutex or C atomics in executable examples; volatile is not synchronization.

The invariant

An invariant states what valid shared state means: a queue has $0 \leq \text{count} \leq \text{capacity}$; a transfer preserves the total balance.

Choose the whole operation that restores the invariant as the critical section. Locking only the final write is insufficient.

All conflicting access paths must follow the same protocol, including readers that need a consistent snapshot.

Critical-section structure

Entry requests permission. The critical section operates on protected state. Exit releases permission.

The remainder does independent work without retaining the lock.

The protected resource determines which critical sections conflict; unrelated resources can use different locks.

```
while (true) {  
    entry section  
  
    critical section  
  
    exit section  
  
    remainder section  
}
```

Three correctness requirements

Mutual exclusion: no two participants execute conflicting critical sections together.

Progress: when the section is free and requests exist, choosing a next entrant cannot be postponed indefinitely.

Bounded waiting: after a request, there is a bound on how many others can enter first. This is stronger than exclusion alone.

Safety: something bad never happens, such as two owners changing a protected queue together.

Liveness: useful progress eventually happens. A system can protect data perfectly by blocking every thread forever, yet be useless.

When evaluating a solution, state scheduler and termination assumptions. Mutual exclusion alone does not establish fairness.

Strict alternation

Let turn be 0 or 1. Thread i enters only when $\text{turn} = i$ and hands the turn to its peer on exit.

Mutual exclusion holds in the abstract model, but progress fails: thread 0 may wait for thread 1 even when thread 1 does not want to enter.

A thread doing unrelated work must not prevent the only interested thread from entering.

Peterson's two-thread protocol

Each thread announces interest, then gives its peer priority in a tie.

Abstract assumptions: atomic reads/writes and sequential consistency; two threads; critical sections eventually finish.

This is reasoning pseudocode, not a safe copy-and-paste C implementation.

```
flag[i] = true;  
turn = j;  
while (flag[j] && turn == j) { }  
/* critical section */  
flag[i] = false;
```

Peterson's tie-breaking argument

If both flags are true, the last write to turn selects the thread that waits. The other can enter.

Both cannot simultaneously satisfy the entry condition when both flags remain true, because turn has only one value.

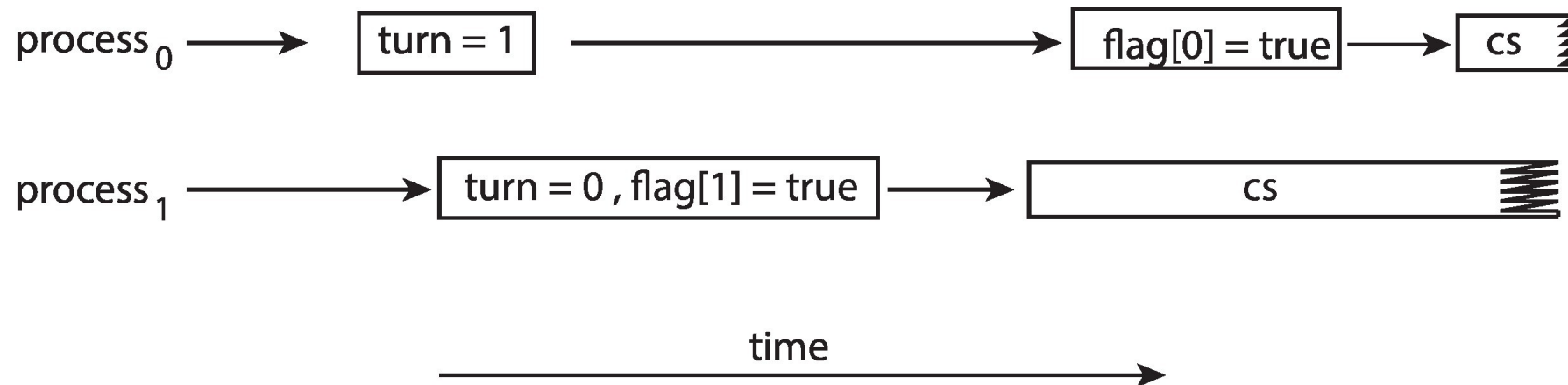
When the owner leaves, it clears its flag. The waiting peer can proceed under the stated execution assumptions.

Ordering assumptions matter

The original diagram illustrates how changing the order of interest and turn operations can defeat the proof.

A compiler or processor need not implement naïve shared-variable pseudocode as the sequentially consistent machine assumed by the proof.

Use synchronization APIs or a correctly specified atomic algorithm; adding volatile does not repair this.



Silberschatz, Galvin & Gagne · OSC 10e · Ch. 6

Atomic read–modify–write

Test-and-set atomically returns an old value and sets a flag. Compare-and-swap changes a value only if it matches an expected value.

Two contenders cannot both successfully change one lock from free to held in the same logical instant.

Atomicity of the lock operation and appropriate memory ordering are both needed to protect other shared data.

Compare-and-swap trace

Initial lock = 0. A and B both attempt CAS(lock, 0, 1).

If A succeeds first, lock becomes 1. B observes a mismatch and does not acquire ownership.

When A releases, a contender can retry. A simple retry loop does not itself guarantee bounded waiting.

The pseudocode CAS here returns the old value; library interfaces may report success differently.

Atomic operations and compound logic

An atomic fetch-add updates one counter without a lost update.

An atomic load followed by a separate atomic store is still two operations: both threads may load the same value and overwrite each other.

“If capacity remains, reserve a slot” is a compound protocol. Atomic variables alone do not make the whole decision atomic.

The gas-station bathroom key



One key = one owner. Take the key before entering; keep it until you leave; then return it.

What the key analogy assumes

Taking the key must be indivisible: checking that it is on the hook and claiming it cannot be separate competing steps.

Keep the key for the entire visit. Returning it early exposes an occupied resource; forgetting it prevents later access.

The long stick discourages losing the key. It does not guarantee fair admission, prevent a duplicate key, or enforce the protocol.

Mutex ownership

Acquire the mutex before touching the protected state; release it after restoring the invariant.

An ordinary mutex has ownership: the acquiring thread is responsible for unlocking it. A semaphore is not an ownership substitute.

Every participant must agree on which mutex protects which state.

```
/* acquire the mutex lock */  
pthread_mutex_lock(&mutex);  
  
/* critical section */  
  
/* release the mutex lock */  
pthread_mutex_unlock(&mutex);
```

Initializing a POSIX mutex

Initialize once before worker threads access it. A static initializer is also available for default attributes.

The original snippet shows initialization; full examples must check return codes.

Destroy the mutex only after all users have finished. Never copy an initialized mutex as though it were ordinary data.

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

The protected counter update

The mutex encloses the entire increment, so each worker sees the result of prior completed increments.

Join both workers before reading the final total. This separates completion from merely having started work.

The provided counter.c checks pthread return codes and offers mutex, atomic and split-operation modes.

```
pthread_mutex_lock(&m);  
counter++;  
pthread_mutex_unlock(&m);
```

```
/* main: join all workers, then inspect */
```

A transaction across two variables

Suppose $A = 100$ and $B = 50$. Transferring 10 must leave $A = 90$, $B = 60$ and $A + B = 150$.

Lock both the debit and credit as one transaction if readers require a consistent combined view.

Two independent atomic stores do not prevent a reader from seeing a partially completed transfer.
Define the snapshot protocol too.

A coarse lock is easier to reason about but serializes more work. Fine-grained locks permit more overlap and require more coordination.

Perform independent computation outside the lock. Keep the state check and dependent update together inside it.

Avoid long I/O or waiting for another resource while holding a lock unless the design explicitly requires and analyzes it.

A spinlock repeatedly tests availability while consuming CPU time. A blocking mutex can suspend a waiter.

Spinning may suit a very short wait when the owner can run on another CPU. Blocking avoids wasting a CPU during a longer wait.

Implementations may combine strategies. A POSIX mutex is not defined to always spin, and lock names do not prove implementation details.

Interrupt disabling and kernel locks

Disabling interrupts on one CPU does not stop another CPU accessing the same data. Applications cannot use this as a general mutex.

Kernel locking rules depend on execution context and configuration. PREEMPT_RT changes the semantics of some Linux lock types.

This lecture uses application-level pthreads; it does not prescribe kernel locking code.

Counter experiment: predictions

mutex mode: two workers $\times$ 100,000 increments should produce 200,000.

atomic mode: `atomic_fetch_add` provides the same final total for this isolated counter.

split mode: coordinated atomic load/store steps force both workers to read 0 and then store 1. Final value is 1 without a C data race.

Compare the protocol, not only the type of the shared variable.

Counter experiment: evidence

Build: `cc -std=c11 -O2 -Wall -Wextra -pthread counter.c -o counter`

Run: `./counter mutex` `./counter atomic` `./counter split`

Record compiler, OS, command and output. Explain the interleaving that split mode deliberately enforces.

Repeated correct totals support the observation; the ordering argument explains correctness beyond the sampled runs.

Why can a single CPU still have a race? Preemption interleaves steps of a shared update.

Why is volatile insufficient? It does not provide mutual exclusion or the synchronization required between threads.

Why can atomic accesses still produce a wrong result? The intended compound operation may span several atomic accesses.

What does a mutex protect? A specified invariant, through a protocol followed by all relevant access paths.

Second half: a thread may need to wait for work, space, or permission even when no one owns a lock.

A mutex answers who may access shared state. A predicate answers whether an operation can proceed.

We will connect counts, queues and predicates in producer–consumer and resource-sharing examples.

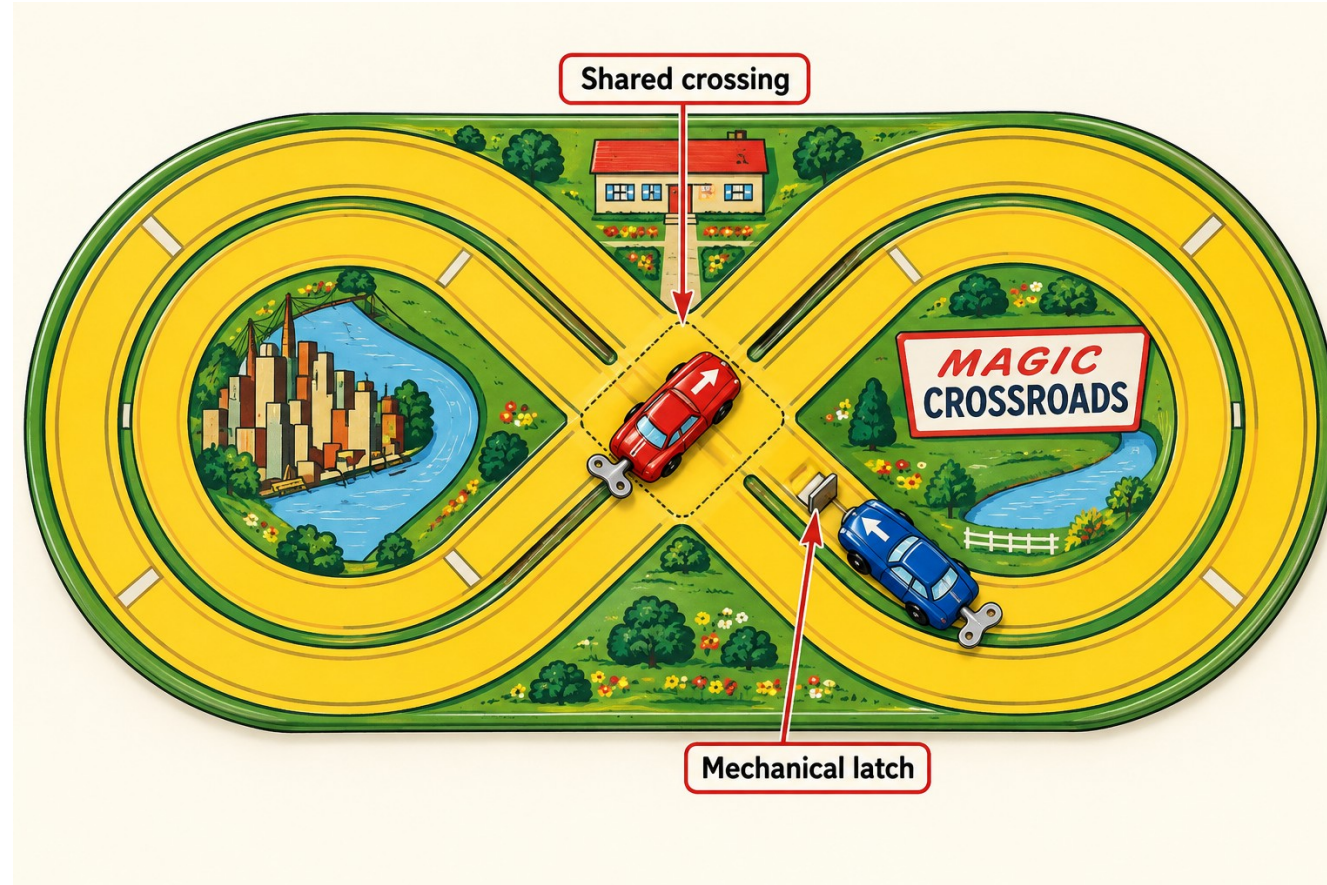
Counting semaphores

A semaphore represents permits. Successful wait consumes one permit; post adds one. A wait blocks when no permit is available.

For a resource pool of size 3, initialize the semaphore to 3. Each successful acquisition must have a matching release.

The waiting order and fairness policy are separate from the permit count.

Magic Crossroads and the mechanical latch



Crossing = critical section. A small latch holds the waiting car; conflicting cars must not enter together.

A semaphore trace

Start $S = 2$. A waits successfully: $S = 1$. B waits successfully: $S = 0$. C must wait.

When A posts, a permit becomes available and a waiting contender can acquire it. Do not assume the posting thread chooses the next owner.

$S = 0$ means no available permit. Textbooks may use an internal negative count for waiters; do not confuse it with this application model.

Exclusion and event ordering

A semaphore initialized to 1 can guard a region when the program uses it correctly. Unlike a mutex, it does not inherently identify an owner.

For “initialize before use,” initialize ready to 0. The initializer posts after initialization; the user waits before use.

A semaphore remembers posted permits even when nobody is waiting. A condition-variable notification does not store a permit.

POSIX semaphore operations

The source snippet shows wait and post on an unnamed semaphore.

Check errors: `sem_wait` can be interrupted by a signal; retry `EINTR` when appropriate for the application.

Do not post after an unsuccessful wait as if acquisition had succeeded. That would invent an extra permit.

```
/* acquire the semaphore */  
sem_wait(&sem);  
  
/* critical section */  
  
/* release the semaphore */  
sem_post(&sem);
```

Where a semaphore lives

For threads in one process, `sem_init` with `pshared = 0` is the usual unnamed-semaphore setup.

A process-shared unnamed semaphore requires `pshared  $\neq$  0` and storage in memory shared by the processes.

Named semaphores offer a name-based interface. “Unnamed” does not mean “impossible between unrelated processes.”

```
#include <semaphore.h>
sem_t sem;

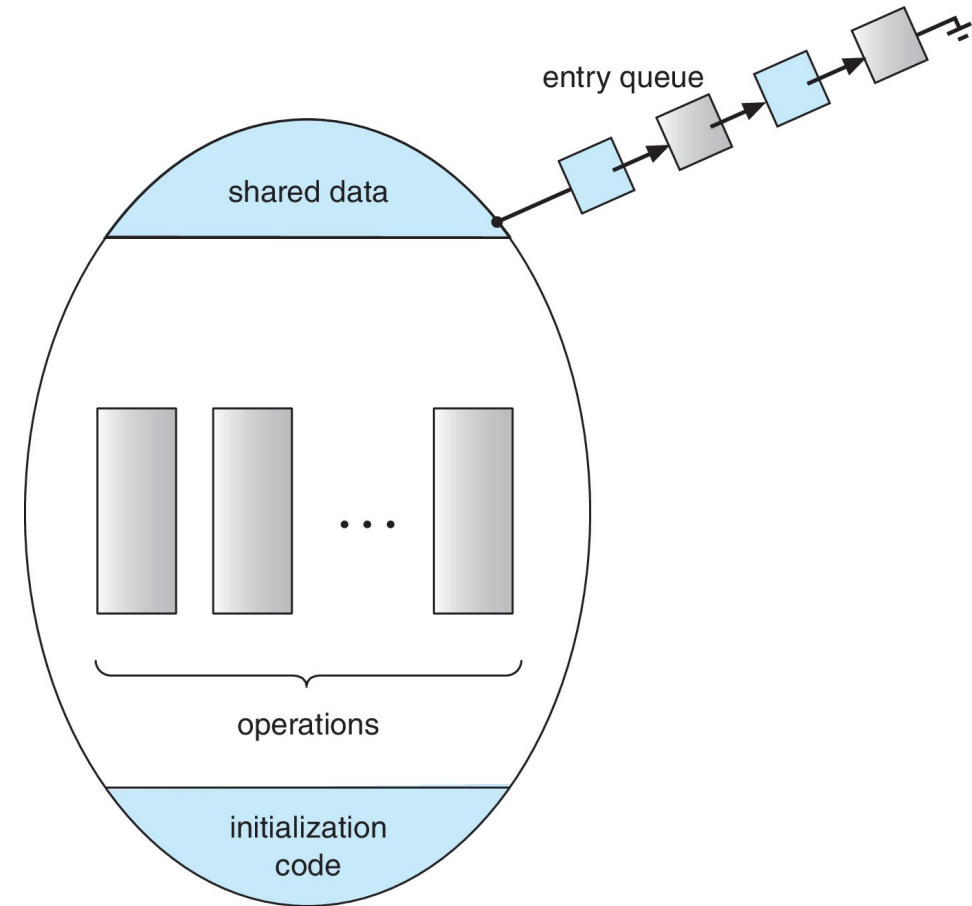
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

Monitors

A monitor combines protected state and operations under a mutual-exclusion discipline.

The entry queue contains threads waiting to enter. The operation executes with exclusive access to the monitor state.

A useful interface hides the state so clients cannot accidentally bypass its protection.



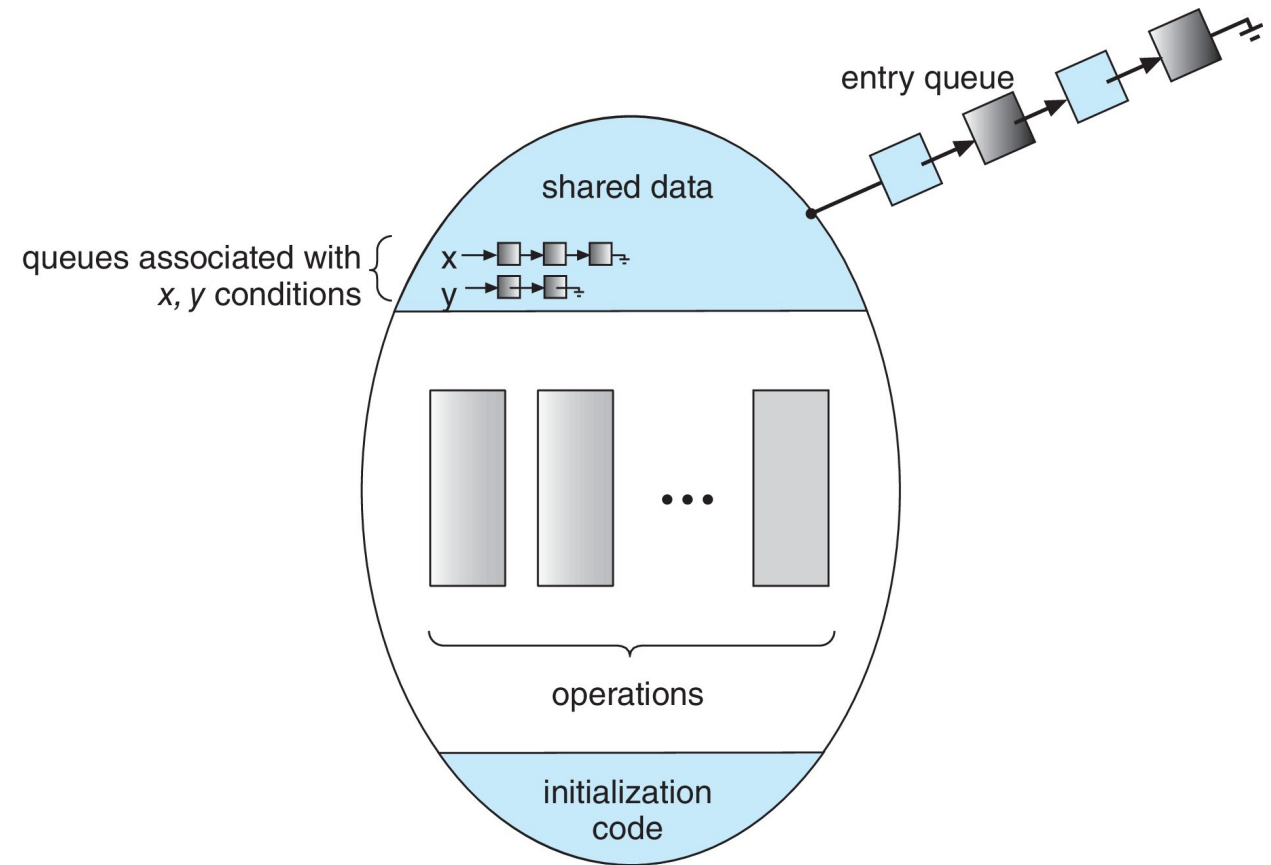
Silberschatz, Galvin & Gagne · OSC 10e · Ch. 6

Condition variables and predicates

A condition variable lets a thread wait for a change that may make a predicate true. The predicate lives in shared application state.

Example: $\text{count} > 0$ means an item can be removed. The mutex protects count and the queue.

Waiting must release the mutex so another thread can change the predicate, then reacquire it before checking again.



The condition-wait protocol

Lock the mutex; check the predicate; wait while false; perform the operation; unlock.

`pthread_cond_wait` atomically releases the mutex and starts waiting relative to the required synchronization protocol; it returns with the mutex reacquired.

Always recheck: a wakeup is not a reservation, and spurious wakeups are permitted.

```
pthread_mutex_lock(&mutex);  
while (a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

Changing state and notifying

The producer changes the predicate while holding the same mutex and notifies a suitable waiter.

The awakened thread must still reacquire the mutex. Another thread may change the state before it does.

This POSIX/Mesa-style discipline differs from a monitor model that immediately hands control to the signaled thread.

```
pthread_mutex_lock(&mutex);  
a = b;  
pthread_cond_signal(&cond_var);  
pthread_mutex_unlock(&mutex);
```

Why while is necessary

Queue empty: C1 and C2 wait. A producer inserts one item and wakes C1.

Before C1 reacquires the mutex, another consumer may remove the item. C1 wakes to an empty queue.

With if, C1 might remove nonexistent data. With while, it checks count again and waits safely.

A spurious wakeup gives another independent reason to use the loop.

The lost-wakeup mistake

Incorrect: check empty without the mutex; a producer inserts and signals; only then does the consumer start waiting.

The signal was not a stored permit. The consumer may sleep despite available work.

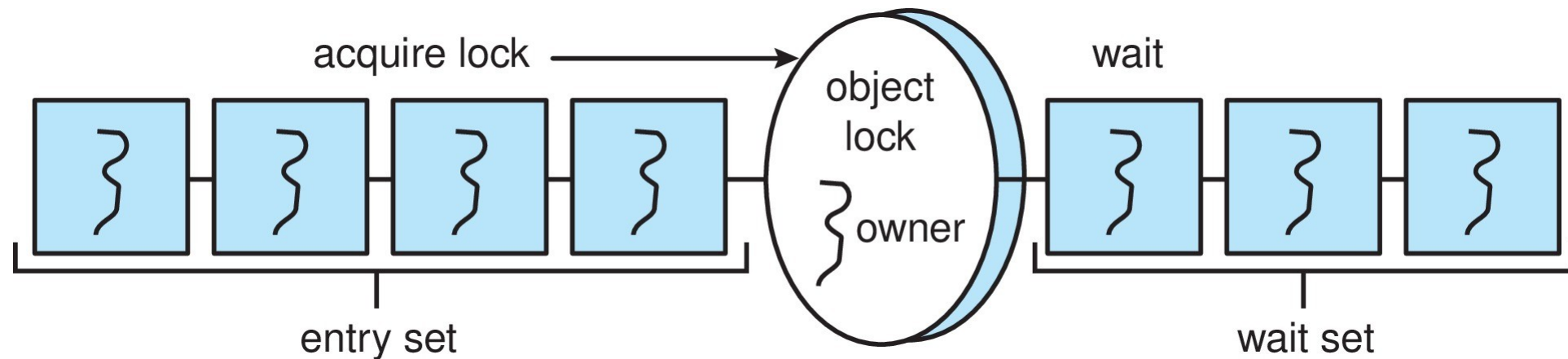
Correct: test the predicate under the same mutex and use condition wait's release-and-wait operation. The state, not a remembered signal, decides whether to sleep.

Entry and wait queues

The original Java diagram makes two different reasons for waiting visible.

Entry set: waiting to acquire the object lock. Wait set: released that lock while waiting for a condition.

A notification permits competition for the lock; it does not mean the predicate is now guaranteed true.



Silberschatz, Galvin & Gagne · OSC 10e · Ch. 7

Melons from truck to conveyor



Loader = producer. Conveyor = finite buffer. Unloader = consumer. Wait when full or empty.

The bounded buffer

A producer creates items; a consumer removes them through a queue of capacity N .

Maintain $0 \leq \text{count} \leq N$. Each inserted item must be removed exactly once and in queue order for this FIFO example.

Use in and out indices that wrap modulo N . The count distinguishes empty from full when the indices coincide.

Semaphores for a bounded buffer

Initialize empty = N, full = 0, mutex = 1.

Producer: reserve empty space, lock, insert, unlock, publish one full slot.

Consumer: reserve a full slot, lock, remove, unlock, publish one empty slot.

The count of available permits can differ transiently from physical occupancy because a worker may hold a reservation.

Producer and consumer ordering

Reserve the needed permit before taking the buffer mutex.

Producer: wait(empty); wait(mutex); insert; post(mutex); post(full).

Consumer: wait(full); wait(mutex); remove; post(mutex); post(empty).

If a consumer holds mutex while waiting for full, the producer cannot lock the buffer to create the missing item.

A capacity-two trace

Initially empty = 2, full = 0, queue = [].

After producing A: empty = 1, full = 1, queue = [A]. After B: empty = 0, full = 2, queue = [A, B].

A third producer waits. Removing A leaves [B] and releases one space; C can then be inserted to give [B, C].

These counts describe completed operations with no outstanding reservation.

A bounded buffer with conditions

Use one mutex and predicates $\text{count} < N$ for producers, $\text{count} > 0$ for consumers.

The producer waits on `not_full` while full, inserts under the mutex, then signals `not_empty`.

The consumer waits on `not_empty` while empty, removes under the mutex, then signals `not_full`.

Keep both the state check and the state change inside the same locked operation.

Closing the queue

A finite program must define how consumers stop. Set closed after the producer has finished, under the queue mutex.

Consumers wait while count == 0 and not closed. If closed and empty, they exit; otherwise they drain remaining items.

Broadcast on closure so every sleeping consumer can recheck the termination predicate.

The supplied example uses one producer and one consumer; extending it requires revisiting completion accounting.

Several readers may coexist if they do not modify protected data. A writer requires exclusive access.

A reader-preference policy can postpone a writer indefinitely while new readers keep arriving. Writer preference can disadvantage readers.

Choose and state the admission policy. Read-mostly does not automatically make a reader–writer lock faster.

Five philosophers, five shared utensils



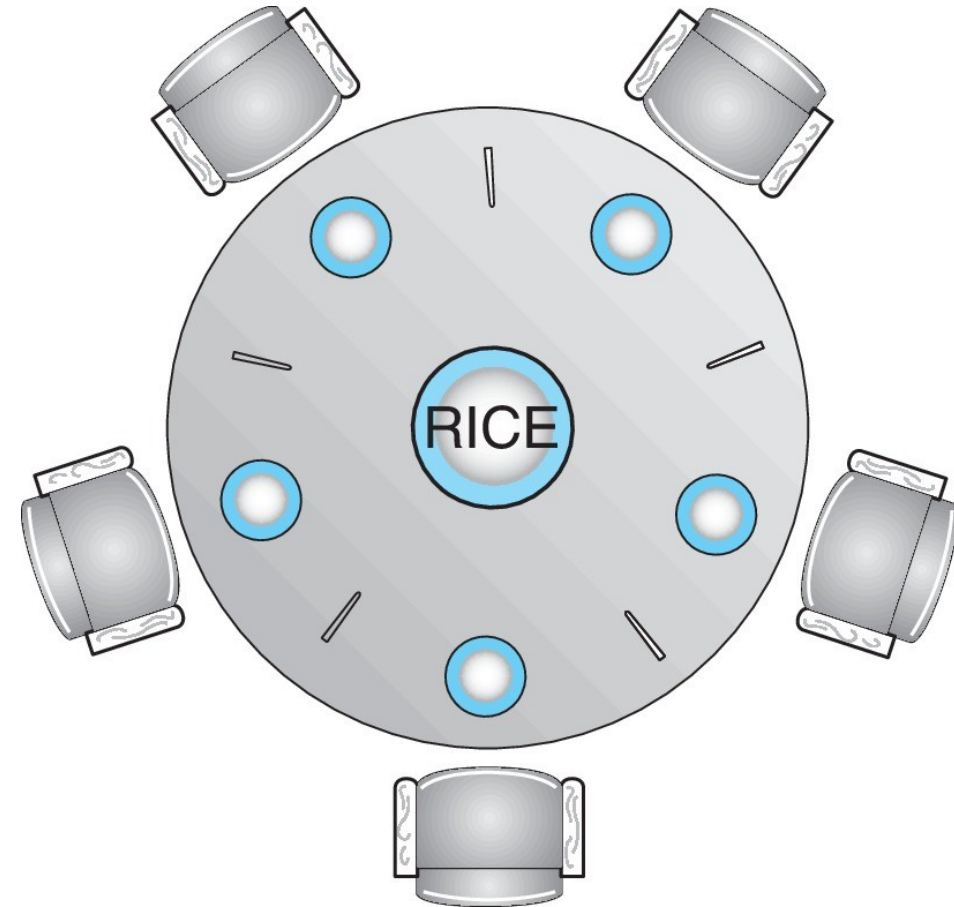
Each philosopher needs both neighboring utensils. One owner per utensil does not ensure global progress.

Dining philosophers

Five philosophers alternate thinking and eating.
Each needs both adjacent chopsticks to eat.

Each chopstick is exclusive, but protecting individual chopsticks does not guarantee global progress.

The shared-resource pattern applies to threads needing several locks, not only to the table story.



Silberschatz, Galvin & Gagne · OSC 10e · Ch. 7

How all five can block

Each philosopher takes the left chopstick. All five left chopsticks are now held.

Each then waits for the right chopstick held by the next philosopher. No one reaches the release step.

Mutual exclusion is working: no chopstick has two owners. The failure is a circular dependency in progress.

Week 4 will formalize deadlock conditions and resource graphs.

Breaking the circular acquisition pattern

Number chopsticks 0–4. Every philosopher acquires the lower-numbered required chopstick before the higher-numbered one.

A cycle of waiting would require strictly increasing resource numbers all the way around and back to the start, which is impossible.

This removes this lock-order deadlock under the model. It does not by itself prove bounded waiting or fair eating opportunities.

A monitor solution for philosophers

Keep THINKING, HUNGRY and EATING states under one monitor lock. Grant eating only if neither neighbor is eating.

A waiting philosopher releases the monitor lock; eating occurs after permission has been recorded. On return, update state and reconsider neighbors.

Neighbor exclusion can prevent deadlock in the textbook protocol, yet fairness still needs a separate argument.

Deadlock, starvation and livelock

Deadlock: participants wait for events that the blocked participants must cause.

Starvation: one participant is perpetually denied progress while others continue.

Livelock: participants keep reacting or retrying without completing useful work.

A successful exclusion test does not rule out any of these progress failures.

A high-priority thread waits for a lock held by a low-priority thread. Medium-priority work can delay the owner and therefore the high-priority thread.

Priority inheritance can temporarily raise the owner's effective priority so it can finish and release.

This is a specific protocol, not a cure for every deadlock or an assumed property of default mutexes.

Choosing the synchronization tool

One shared invariant across several operations: a mutex is often the clearest starting point.

A limited number of permits: a counting semaphore. A wait for shared state to satisfy a predicate: a condition variable plus mutex.

One independent numeric update: an atomic operation may suffice.

Minimize shared mutable state where possible, then specify the remaining protocol.

Optional Ubuntu queue practice

Build queue.c with: `cc -std=c11 -O2 -Wall -Wextra -pthread queue.c -o queue`

The producer sends integers 1 through 10,000. The consumer checks FIFO order; main checks count and sum after joining.

Expected consumed count = 10,000; expected sum = 50,005,000.

Change capacity from 4 to 1 and repeat. Explain why correctness should remain the same despite a different blocking pattern.

Observe, explain, challenge

Before running, identify each shared field, its mutex, each wait predicate, and which operation can make it true.

Save output and environment details. Explain why the consumer drains queued work after closure.

AI assistance is permitted: ask for an adverse interleaving, then check it against the actual code. Do not accept “thread-safe” without a protocol.

This is optional practice, not a released graded assignment or submission deadline.

Review: diagnose the protocol

A counter is atomic but updated by load then store. Is the increment indivisible? No: a read–modify–write operation is needed.

A consumer waits while holding a mutex without releasing it. Can the producer satisfy the predicate? Not if it needs that mutex.

A philosopher protocol avoids deadlock. Is starvation impossible? No: fairness is a separate property.

Review: justify a correct queue

Safety: all queue fields use the same mutex; count stays within $0 \dots N$; each dequeue follows a valid enqueue.

Waiting: condition waits release the mutex; while loops recheck predicates after reacquisition.

Completion: the producer closes only after inserting every item; consumers drain, then stop on closed and empty.

Testing samples executions. Invariants and ordering explain why the permitted executions preserve the intended result.

Adapted from Silberschatz, Galvin and Gagne, Operating System Concepts, 10th edition, Chapters 6–7. Original figures retain their source credits.

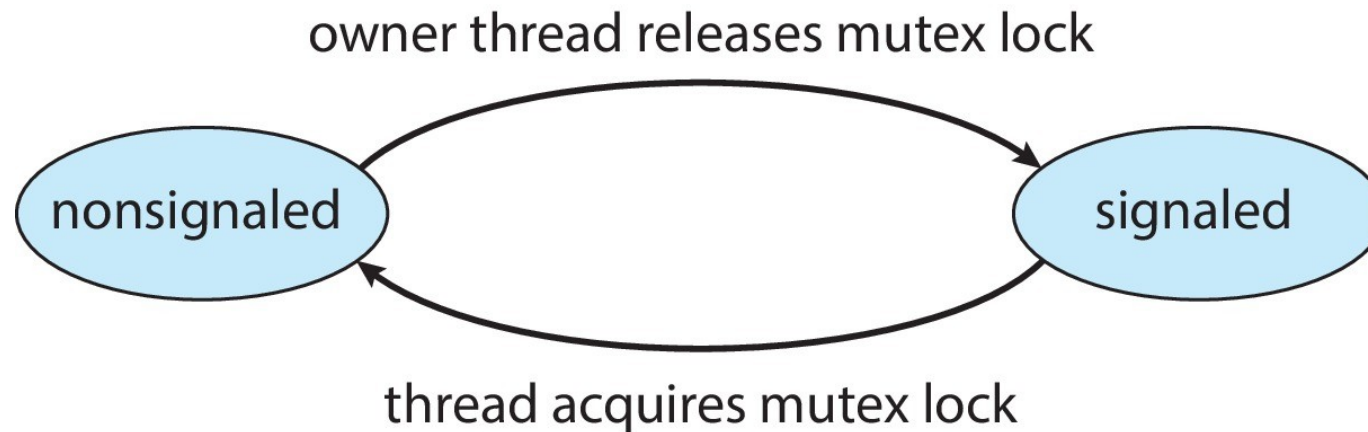
Companion references: OSTEP concurrency, locks, condition variables and semaphores; POSIX manuals linked in the reading.

Next: deadlocks and resource allocation. Bring the philosopher wait cycle and the difference between exclusion and progress.

The illustrated reading includes complete practice programs, worked answers and source links.

Textbook supplement · The signaled state means the mutex is available; acquisition makes it nonsignaled. The owner releases it to make it available again.

Read the arrows as transitions, not a FIFO queue guarantee. This is the textbook Windows example, not a Linux mutex diagram.



Silberschatz, Galvin & Gagne · OSC 10e · Ch. 7

Textbook supplement · Trace the arithmetic: set 5, add 10, subtract 4, increment once, then read 12, assuming no other updates. Each operation acts on the atomic object.

This is the kernel `atomic_t` interface. Our user-space C exercise uses C11 atomics. Atomicity of one counter does not protect a multi-variable invariant.

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter, 5);</code>	<code>counter = 5</code>
<code>atomic_add(10, &counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4, &counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

Creating a named semaphore

Textbook supplement · A name lets separate processes open the same semaphore. The initial value applies when a new semaphore is created, not every time an existing one is opened.

For portable practice use a leading-slash name such as /cse513_demo. Check SEM_FAILED; choose permissions deliberately and plan sem_close/sem_unlink cleanup.

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Using a named semaphore

Textbook supplement · A successful `sem_wait` consumes a permit; `sem_post` supplies one. Here a single permit guards the critical section.

This screenshot omits error handling. Handle an interrupted or failed wait before entering, and do not post as though a failed acquisition succeeded.

```
/* acquire the semaphore */  
sem_wait(sem);
```

```
/* critical section */
```

```
/* release the semaphore */  
sem_post(sem);
```

Initializing a condition variable

Textbook supplement · The mutex protects the shared predicate. The condition variable lets threads wait for changes that may make that predicate true.

Initialize both before concurrent use; check return codes. Destroy them only after all users and waiters have finished. A condition variable does not store permits.

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

Textbook supplement · The buffer holds at most five items. count tracks occupancy; in and out wrap around the array. Synchronized methods use the same object monitor.

Connect this state to the melon conveyor: count == 0 means empty and count == BUFFER_SIZE means full. The next producer/consumer figures provide the method bodies.

```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

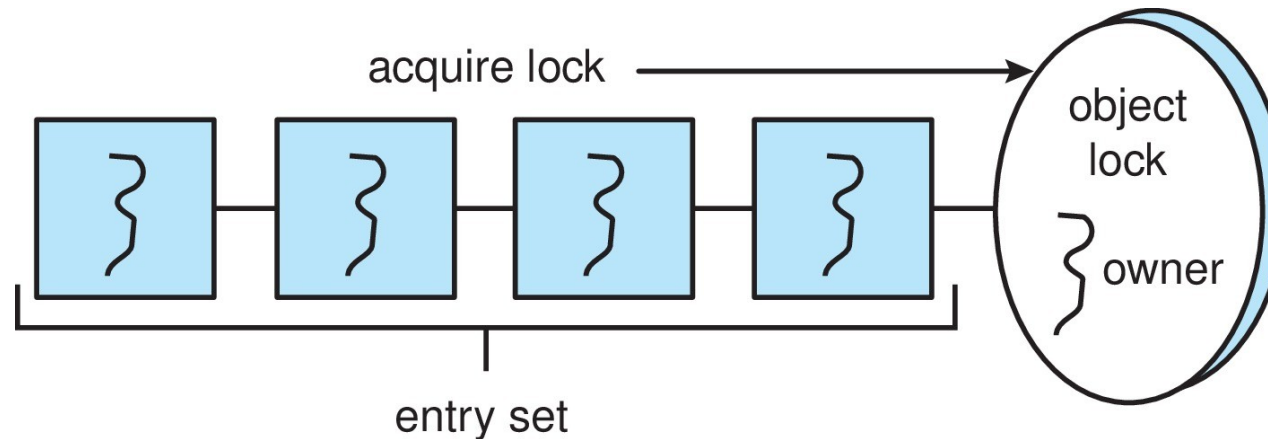
    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```

Textbook supplement · A thread seeking an occupied object monitor cannot enter its synchronized region. The entry set represents contenders for that monitor.

Compare with the wait-set figure in the main lecture: waiting for the lock and waiting for a predicate are different. The drawing does not promise FIFO admission.



Silberschatz, Galvin & Gagne · OSC 10e · Ch. 7

Java bounded-buffer producer

Textbook supplement · While the buffer is full, wait releases the monitor. On return, the thread reacquires it and checks count again before inserting and advancing in.

The source uses notify and ignores interruption. For a general multi-producer/multi-consumer version with one wait set, use notifyAll and an explicit interruption policy.

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```

Textbook supplement · While empty, wait releases the monitor. After reacquisition and rechecking, remove the item, advance out and decrement count.

Compare each transition with our C queue. The source has no close/drain protocol and discards interruption; it is a teaching fragment, not our runnable practice program.

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    notify();

    return item;
}
```

Java explicit lock and cleanup

Textbook supplement · lock obtains the ReentrantLock. The try/finally structure releases it when the protected body completes or throws an exception.

Place acquisition before the try whose finally unlocks. Release only a lock actually acquired, and keep every access to the invariant under the same discipline.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```

Textbook supplement · The image sketches the constructor parameter: its value establishes the initial permit count. One permit can model exclusive admission; more permit bounded concurrency.

This line is an API sketch, not a complete declaration to paste into a class. The ordinary one-argument Java Semaphore constructor does not guarantee FIFO ordering.

```
Semaphore(int value);
```

Java semaphore: find the permit bug

Textbook supplement · Inspect this original snippet critically: acquire can throw InterruptedException before obtaining a permit, but the finally block still releases one.

That path creates an extra permit. Fix by acquiring before entering the try/finally release region, or track successful acquisition. Never silently discard the interruption policy.

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```

Textbook supplement · `newCondition` creates a condition associated with this `ReentrantLock`. `await` releases that lock while waiting and reacquires it before returning normally.

Hold the lock when checking the predicate, waiting or signaling. Wait in a while loop because notification alone does not establish the predicate.

```
Lock key = new ReentrantLock();  
Condition condVar = key.newCondition();
```

One condition per turn

Textbook supplement · Five conditions share one lock. A separate turn variable identifies which thread may work; each condition provides a waiting location.

The condition objects do not store whose turn it is. Initialize turn and protect it with the lock. A stalled designated thread can stop this strict-turn protocol.

```
Lock lock = new ReentrantLock();  
Condition[] condVars = new Condition[5];  
  
for (int i = 0; i < 5; i++)  
    condVars[i] = lock.newCondition();
```

Silberschatz, Galvin & Gagne · OSC 10e · Ch. 7

Turn-taking: recheck after waking

Textbook supplement · The figure tests
threadNumber != turn with if before await.
General Java condition code must use while so
a spurious wakeup cannot authorize the wrong
thread.

The empty catch also hides interruption. Define
cancellation behavior and keep turn
updates/signaling under the lock; the next turn
must correspond to a participating thread.

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```

An update protected by a lock

Textbook supplement · The acquire/update/release sequence makes cooperating callers execute the protected update one at a time. Identify exactly which shared state belongs to the invariant.

All conflicting access must follow the same lock. Real code needs a defined cleanup path so an error or exception cannot leave the lock held.

```
void update ()  
{  
    acquire();  
  
    /* modify shared data */  
  
    release();  
}
```

Transactional-memory notation

Textbook supplement · The atomic block expresses a desired transaction over the enclosed reads and writes: commit as a unit, or handle a conflict according to the supporting system.

This is conceptual textbook notation, not standard C or Java syntax. Do not confuse a multi-operation transaction with a single atomic integer instruction.

```
void update ()  
{  
    atomic {  
        /* modify shared data */  
    }  
}
```