

Operating Systems Notes - Chapter 7

June 9, 2025

Contents

1	Classic problems of synchronization	2
1.1	Introduction	2
1.2	Chapter objectives	2
1.3	Classic problems of synchronization	2
1.4	The bounded-buffer problem	2
1.5	The readers-writers problem	3
1.6	The dining-philosophers problem	4
1.6.1	Semaphore solution	4
1.6.2	Monitor solution	4
1.7	Section glossary	5
2	Synchronization within the kernel	6
2.1	Synchronization in Windows	6
2.2	Synchronization in Linux	6
2.3	Section glossary	7
3	POSIX synchronization	8
3.1	POSIX mutex locks	8
3.2	POSIX semaphores	8
3.2.1	POSIX named semaphores	8
3.2.2	POSIX unnamed semaphores	9
3.3	POSIX condition variables	9
3.4	Section glossary	10
4	Synchronization in Java	11
4.1	Java monitors	11
4.2	Block synchronization	11
4.3	Reentrant locks	13
4.4	Semaphores	13
4.5	Condition variables	13
4.6	Section glossary	15
5	Alternative approaches	16
5.1	Transactional memory	16
5.2	OpenMP	17
5.3	Functional programming languages	17
5.4	Section glossary	18
6	Summary	19

1 Classic problems of synchronization

1.1 Introduction

- Previous chapter: Synchronization Tools.
- Focused on critical-section problem and race conditions with shared data.
- Examined tools to prevent race conditions:
 - Low-level hardware: memory barriers, compare-and-swap.
 - Higher-level: mutex locks, semaphores, monitors.
- Discussed challenges: liveness hazards like deadlocks.
- This chapter:
 - Applies synchronization tools to classic problems.
 - Explores synchronization mechanisms in Linux, UNIX, Windows.
 - Describes API details for Java and POSIX systems.

1.2 Chapter objectives

- Explain: bounded-buffer, readers-writers, dining-philosophers synchronization problems.
- Describe: specific tools used by Linux and Windows for process synchronization.
- Illustrate: how POSIX and Java solve process synchronization problems.
- Design and develop: solutions using POSIX and Java APIs.

1.3 Classic problems of synchronization

- Examples of concurrency-control problems.
- Used for testing new synchronization schemes.
- Solutions traditionally use semaphores; mutex locks can be used for binary semaphores in actual implementations.

1.4 The bounded-buffer problem

- Introduced in Section 6.1.
- Illustrates power of synchronization primitives.
- General structure presented; related programming project in exercises.
- Producer and consumer processes share data structures:

```
int N;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0
```
- Pool: N buffers, each holding one item.
- `mutex`: binary semaphore, mutual exclusion for buffer pool access, initialized to 1.
- `empty`: counts empty buffers, initialized to N.
- `full`: counts full buffers, initialized to 0.
- Producer process (Figure ??):

```
while (true) {
    . . .
    /* produce an item in next_produced */
    . . .
    wait(empty);
    wait(mutex);

    . . .
    /* add next_produced to the buffer */
    . . .
    signal(mutex);
    signal(full);
}
```

- Consumer process (Figure ??):

```
while (true) {
    wait(full);
    wait(mutex);

    . . .
    /* remove an item from buffer to next_consumed */
    . . .
    signal(mutex);
    signal(empty);

    . . .
    /* consume the item in next_consumed */
    . . .
}
```

```
}
```

- Symmetry between producer and consumer.
- Interpretation: producer produces full buffers for consumer, or consumer produces empty buffers for producer.

1.5 The readers-writers problem

- Shared database accessed by concurrent processes.
- **Readers**: only read database.
- **Writers**: update (read and write) database.
- Problem: If writer and another process (reader/writer) access simultaneously, chaos may ensue.
- Requirement: Writers must have exclusive access while writing.
- *Readers-writers problem*: Synchronization problem to ensure this.
- Variations involve priorities:
 - **First** readers-writers problem: No reader waits unless a writer already has permission. Readers don't wait for other readers if a writer is waiting.
 - **Second** readers-writers problem: Writer performs write ASAP once ready. If writer is waiting, no new readers may start.
- Starvation: Solutions may lead to starvation (writers in first case, readers in second).
- Solution to first readers-writers problem:
 - Shared data structures for reader processes:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```
 - `mutex` and `rw_mutex`: binary semaphores, initialized to 1.
 - `read_count`: integer, number of active readers, initialized to 0.
 - `rw_mutex`: common to reader and writer processes, acts as mutual exclusion for writers, used by first/last reader entering/exiting critical section.
 - `mutex`: ensures mutual exclusion when `read_count` is updated.
 - `read_count`: tracks current readers.
 - Writer process (Figure ??):

```
while (true) {
    wait(rw_mutex);
    .
    .
    .
    /* writing is performed */
    signal(rw_mutex);
}
```
 - Reader process (Figure ??):

```
while (true) {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    .
    .
    .
    /* reading is performed */
    .
    .
    .
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}
```
 - If writer in critical section and n readers waiting: 1 reader queued on `rw_mutex`, $n - 1$ readers queued on `mutex`.
 - When writer executes `signal(rw_mutex)`, scheduler selects waiting readers or a single waiting writer.
- *Reader-writer locks*: generalization of problem/solutions.
 - Acquire lock by specifying mode: *read* or *write*.
 - Read mode: multiple processes concurrently.
 - Write mode: only one process (exclusive access).
 - Most useful when:
 - * Easy to identify read-only vs. read-write processes.

- * More readers than writers (increased concurrency compensates for overhead).

1.6 The dining-philosophers problem

- Five philosophers, circular table, five chairs, bowl of rice, five single chopsticks.
- Philosopher thinks, then gets hungry.
- Tries to pick up two closest chopsticks (left and right neighbors).
- Picks up one chopstick at a time. Cannot pick up if neighbor holds it.
- Eats with both chopsticks, then puts them down and thinks.
- Classic synchronization problem: example of allocating several resources among several processes.
- Goal: deadlock-free and starvation-free allocation.

1.6.1 Semaphore solution

- Each chopstick represented by a semaphore.
- Grab chopstick: `wait()` operation on semaphore.
- Release chopstick: `signal()` operation on semaphore.
- Shared data:


```
semaphore chopstick[5];
```
- All `chopstick` elements initialized to 1.
- Philosopher i structure (Figure ??):


```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    .
    .
    .
    /* eat for a while */
    .
    .
    .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    .
    .
    .
    /* think for awhile */
    .
    .
    .
}
```
- Guarantees no two neighbors eat simultaneously.
- **Problem:** Could create deadlock.
 - * Example: All five philosophers hungry, each grabs left chopstick. All `chopstick` elements become 0.
 - * Each tries to grab right chopstick, delayed forever.
- Remedies to deadlock:
 - * Allow at most four philosophers at table simultaneously.
 - * Philosopher picks up both chopsticks only if both available (in critical section).
 - * Asymmetric solution: odd-numbered philosopher picks left then right; even-numbered picks right then left.
- Section 6.7 presents a deadlock-free solution.
- Satisfactory solution must guard against starvation (deadlock-free \neq starvation-free).

1.6.2 Monitor solution

- Deadlock-free solution using monitors.
- Restriction: Philosopher picks up chopsticks only if both available.
- Three states for a philosopher:


```
enum {THINKING, HUNGRY, EATING} state[5];
```
- `state[i] = EATING` only if neighbors not eating: `(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING)`.
- Condition variable:


```
condition self[5];
```
- Allows philosopher i to delay if hungry but cannot get chopsticks.
- Chopstick distribution controlled by monitor `DiningPhilosophers` (Figure ??).
- Philosopher i sequence:


```
DiningPhilosophers.pickup(i);
    .
    .
    .
    eat
    .
    .
    .
DiningPhilosophers.putdown(i);
```

- Monitor definition (Figure ??):

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

- Ensures no two neighbors eat simultaneously and no deadlocks.
- Possible for a philosopher to starve (solution not presented, left as exercise).

1.7 Section glossary

- *readers-writers problem*: Synchronization problem where processes/threads either read or read/write shared data.
- *reader-writer lock*: Lock for item access by read-only and read-write accessors.
- *dining-philosophers problem*: Classic synchronization problem where multiple operators (philosophers) access multiple items (chopsticks) simultaneously.

2 Synchronization within the kernel

2.1 Synchronization in Windows

- Windows OS: multithreaded kernel, supports real-time applications and multiple processors.
- Single-processor systems:
 - Kernel accesses global resource: temporarily masks interrupts for all interrupt handlers that may access the resource.
- Multiprocessor systems:
 - Kernel protects global resource access using spinlocks.
 - Spinlocks used only for short code segments.
 - Kernel ensures thread never preempted while holding a spinlock (for efficiency).
- Thread synchronization outside kernel: *dispatcher objects*.
 - Threads synchronize using: mutex locks, semaphores, events, timers.
 - Mutex locks: protect shared data; thread gains ownership to access, releases when finished.
 - Semaphores: behave as described in Section 6.6.
 - *Events*: similar to condition variables; notify waiting thread when condition occurs.
 - Timers: notify one (or more) threads when specified time expires.
- Dispatcher objects states:
 - *Signaled state*: object available, thread will not block.
 - *Nonsignaled state*: object not available, thread will block.
 - State transitions of mutex lock dispatcher object illustrated in Figure ??.
- Relationship between dispatcher object state and thread state:
 - Thread blocks on nonsignaled object: state changes from ready to waiting, placed in waiting queue.
 - Object moves to signaled state: kernel checks waiting threads.
 - Kernel moves one (or more) threads from waiting to ready state.
 - Number of threads selected depends on dispatcher object type:
 - * Mutex: only one thread (mutex owned by single thread).
 - * Event: all waiting threads.
- Mutex lock illustration:
 - Thread tries to acquire nonsignaled mutex: suspended, placed in waiting queue.
 - Mutex moves to signaled state (released by another thread): thread at front of queue moves from waiting to ready, acquires mutex.
- *Critical-section object*:
 - User-mode mutex, often acquired/released without kernel intervention.
 - Multiprocessor system: first uses spinlock while waiting.
 - If spins too long: acquiring thread allocates kernel mutex and yields CPU.
 - Efficient: kernel mutex allocated only when contention exists (rare in practice, significant savings).
- Programming project at end of chapter uses mutex locks and semaphores in Windows API.

2.2 Synchronization in Linux

- Prior to Version 2.6: nonpreemptive kernel (process in kernel mode could not be preempted).
- Now: Linux kernel is fully preemptive (task can be preempted while running in kernel).
- Mechanisms for synchronization in kernel:
 - **Atomic integers**:
 - * Simplest synchronization technique.
 - * Opaque data type: `atomic_t`.
 - * All math operations are atomic (performed without interruption).
 - * Example:

```
atomic_t counter;
int value;
```
 - * Atomic operations:
 - `atomic_set(&counter,5);` `counter = 5`
 - `atomic_add(10,&counter);` `counter = counter + 10`
 - `atomic_sub(4,&counter);` `counter = counter - 4`

- `atomic_inc(&counter);` `counter = counter + 1`
- `value = atomic_read(&counter);` `value = 12` (example result)
- * Efficient for updating integer variables (e.g., counters); no locking overhead.
- * Limited use: only for single integer variables. More sophisticated tools needed for multiple variables in race conditions.
- **Mutex locks:**
 - * Protect critical sections within kernel.
 - * Task invokes `mutex_lock()` before critical section, `mutex_unlock()` after.
 - * If unavailable: task calling `mutex_lock()` sleeps, awakened when owner invokes `mutex_unlock()`.
- **Spinlocks and Semaphores:**
 - * Linux also provides these (and reader-writer versions).
 - * SMP machines: spinlock is fundamental locking mechanism, held for short durations.
 - * Single-processor machines (e.g., embedded systems): spinlocks inappropriate. Replaced by enabling/disabling kernel preemption.
 - * Summary for single-processor:
 - Instead of holding spinlock: kernel disables kernel preemption.
 - Instead of releasing spinlock: kernel enables kernel preemption.
- **Nonrecursive locks:**
 - Both spinlocks and mutex locks in Linux kernel are nonrecursive.
 - If thread acquires lock, cannot acquire same lock again without releasing it first.
 - Second attempt to acquire will block.
- Disabling/Enabling kernel preemption:
 - Linux approach: `preempt_disable()` and `preempt_enable()` system calls.
 - Kernel not preemptible if task running in kernel holds a lock.
 - Enforcement: each task has `thread-info` structure with `preempt_count` counter.
 - `preempt_count`: indicates number of locks held by task.
 - Lock acquired: `preempt_count` incremented.
 - Lock released: `preempt_count` decremented.
 - If `preempt_count > 0`: not safe to preempt kernel (task holds lock).
 - If `preempt_count = 0`: kernel can be safely interrupted (assuming no outstanding `preempt_disable()` calls).
- When to use which lock:
 - Spinlocks (and kernel preemption disable/enable): only when lock held for short duration.
 - Semaphores or mutex locks: when lock must be held for longer period.

2.3 Section glossary

- *dispatcher objects*: Windows scheduler feature controlling dispatching and synchronization. Threads synchronize via mutex locks, semaphores, events, and timers.
- *event*: Windows OS scheduling feature, similar to a condition variable.
- *critical-section object*: User-mode mutex object in Windows OS, often acquired/released without kernel intervention.

3 POSIX synchronization

- Synchronization methods in previous section: kernel-level, for kernel developers.
- POSIX API: available for user-level programmers, not part of specific OS kernel.
- Implemented using host OS tools.
- This section covers: mutex locks, semaphores, condition variables in Pthreads and POSIX APIs.
- Widely used for thread creation and synchronization on UNIX, Linux, macOS.

3.1 POSIX mutex locks

- Fundamental synchronization technique with Pthreads.
- Purpose: protect critical sections of code.
- Thread acquires lock before entering, releases upon exiting.
- Data type: `pthread_mutex_t`.
- Creation: `pthread_mutex_init()` function.
 - First parameter: pointer to mutex.
 - Second parameter: NULL for default attributes.
- Example:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```
- Acquisition and Release: `pthread_mutex_lock()` and `pthread_mutex_unlock()`.
 - If `pthread_mutex_lock()` invoked and mutex unavailable: calling thread blocks until owner invokes `pthread_mutex_unlock()`.
- Protecting critical section example:

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```
- Return values: 0 for correct operation, nonzero for error.

3.2 POSIX semaphores

- Many systems implementing Pthreads provide semaphores.
- Not part of POSIX standard; belong to POSIX SEM extension.
- Two types: *named* and *unnamed*.
- Differences: how they are created and shared between processes.
- Linux systems (Version 2.6+ kernel) support both.

3.2.1 POSIX named semaphores

- Creation and opening: `sem_open()` function.
- Example:

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

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```
- "SEM": semaphore name.
- `O_CREAT` flag: semaphore created if it doesn't exist.
- `0666`: read and write access for other processes.
- Initialized to 1.
- Advantage: multiple unrelated processes can easily use common semaphore by name.
- Subsequent `sem_open()` calls (with same parameters) by other processes return descriptor to existing semaphore.
- Operations:
 - * `wait()` → `sem_wait()`


```
* signal() → sem_post()
```

- Protecting critical section example:

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

/* critical section */

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

- Supported by Linux and macOS.

3.2.2 POSIX unnamed semaphores

- Creation and initialization: `sem_init()` function.

- Parameters:

1. Pointer to the semaphore.
2. Flag indicating level of sharing.
3. Semaphore's initial value.

- Example:

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

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

- Flag 0: semaphore shared only by threads in creating process.
- Nonzero flag: allows sharing between separate processes (by placing in shared memory).
- Initialized to 1.
- Operations: uses same `sem_wait()` and `sem_post()` as named semaphores.
- Protecting critical section example:

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

/* critical section */

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

- Return values: 0 for success, nonzero for error.

3.3 POSIX condition variables

- Pthreads condition variables similar to those in Section 6.7.
- Difference: Section 6.7 uses monitors for locking; C (used with Pthreads) lacks monitors.
- Locking accomplished by associating condition variable with a mutex lock.
- Data type: `pthread_cond_t`.
- Initialization: `pthread_cond_init()` function.
- Example of creating and initializing condition variable and associated mutex:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

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

- Waiting on condition variable: `pthread_cond_wait()` function.

- Example of waiting for `a == b`:

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

- Mutex lock must be acquired before `pthread_cond_wait()` call.
- Mutex protects data in conditional clause from race conditions.
- If condition not true: `pthread_cond_wait()` invoked, passing mutex and condition variable.
- `pthread_cond_wait()` releases mutex lock, allowing other threads to access/update shared data.

- Conditional clause within a loop: important to recheck condition after being signaled (protects against program errors).
- Signaling a condition variable: `pthread_cond_signal()` function.
- Thread modifying shared data invokes `pthread_cond_signal()` to signal one waiting thread.
- Example of signaling:


```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```
- `pthread_cond_signal()` does NOT release mutex lock.
- Subsequent `pthread_mutex_unlock()` releases mutex.
- Once mutex released, signaled thread becomes owner of mutex and returns from `pthread_cond_wait()`.
- Programming problems/projects at end of chapter use Pthreads mutex locks, condition variables, and POSIX semaphores.

3.4 Section glossary

- *named semaphore*: POSIX scheduling construct, exists in file system, shareable by unrelated processes.
- *unnamed semaphore*: POSIX scheduling construct, usable only by threads in the same process.

4 Synchronization in Java

- Java language and API: rich support for thread synchronization since its origins.
- This section covers:
 - Java monitors (original mechanism).
 - Reentrant locks, semaphores, condition variables (introduced in Release 1.5).
- Focus on common locking and synchronization mechanisms.
- Java API has more features not covered (e.g., atomic variables, CAS instruction).

4.1 Java monitors

- Java provides a monitor-like concurrency mechanism.
- Illustrated with `BoundedBuffer` class (Figure ??).
- Producer/consumer invoke `insert()` and `remove()` methods.
- `BoundedBuffer` class structure:

```
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 7.4.3 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.4.3 */
    }
}
```

- Every Java object has a single associated lock.
- `synchronized` method: entering requires owning the object's lock.
- Declared by placing `synchronized` keyword in method definition (e.g., `insert()`, `remove()`).
- Entering `synchronized` method:
 - Requires owning lock on `BoundedBuffer` object instance.
 - If lock owned by another thread: calling thread blocks, placed in object's *entry set*.
 - *Entry set*: set of threads waiting for lock to become available.
 - If lock available: calling thread becomes owner, enters method.
 - Lock released when thread exits method.
 - If entry set not empty on lock release: JVM arbitrarily selects thread from set to own lock (often FIFO in practice).
- Entry set operation illustrated in Figure ??.
- Every object also has a *wait set* (initially empty).
- When thread enters `synchronized` method (owns lock):
 - May be unable to continue if condition not met (e.g., producer calls `insert()` and buffer is full).
 - Thread releases lock and waits until condition is met.

4.2 Block synchronization

- *Scope* of lock: time between acquisition and release.
- `synchronized` method: large scope if only small part manipulates shared data.
- Better: synchronize only the block of code manipulating shared data (smaller lock scope).
- Java allows block synchronization:

```

public void someMethod() {
    /* non-critical section */

    synchronized(this) {
        /* critical section */
    }

    /* remainder section */
}

```

- Only critical-section access requires ownership of `this` object lock.

- When thread calls `wait()` method:

1. Releases lock for the object.
2. Thread state set to blocked.
3. Thread placed in wait set for the object.

- Example (Figure ??):

- Producer calls `insert()`, buffer full, calls `wait()`.
- `wait()` releases lock, blocks producer, puts producer in wait set.
- Consumer enters `remove()` (lock now available), frees buffer space.

- Entry and wait sets illustrated in Figure ??.

- `insert()` and `remove()` methods using `wait()` and `notify()` (Figure ??):

```

/* 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();
}

/* 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;
}

```

- `notify()` method:

- Picks arbitrary thread `T` from wait set.
- Moves `T` from wait set to entry set.
- Sets state of `T` from blocked to runnable.

- `T` now eligible to compete for lock.

- Once `T` regains lock, returns from `wait()`, rechecks `count`.

- `notify()` ignored if no thread in wait set.

- Sequence of events with `wait()` and `notify()`:

- Buffer full, lock available.
- Producer calls `insert()`, enters, finds buffer full, calls `wait()`.
- `wait()` releases lock, blocks producer, puts producer in wait set.
- Consumer calls `remove()`, enters (lock available), removes item, calls `notify()`. Consumer still owns lock.

- `notify()` moves producer from wait set to entry set, sets state to runnable.
- Consumer exits `remove()`, releases lock.
- Producer reacquires lock, resumes from `wait()`.
- Producer tests `while` loop, finds room, proceeds with `insert()`.
- Producer exits `insert()`, releases lock.
- `synchronized`, `wait()`, `notify()` are original Java mechanisms.
- Later Java API revisions introduced more flexible/robust locking.

4.3 Reentrant locks

- Simplest locking mechanism in API: `ReentrantLock`.
- Similar to `synchronized` statement: owned by single thread, provides mutual exclusive access to shared resource.
- Additional features: e.g., *fairness* parameter (favors longest-waiting thread).
- Acquisition: invoke `lock()` method.
 - If lock available OR invoking thread already owns it (*reentrant*): `lock()` assigns ownership, returns control.
 - If lock unavailable: invoking thread blocks until owner invokes `unlock()`.
- Implements `Lock` interface.
- Usage example:


```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```
- `try` and `finally` idiom:
 - Ensures lock is released (via `unlock()`) after critical section completes or if exception occurs in `try` block.
 - `lock()` not placed in `try` clause because it doesn't throw checked exceptions.
 - Avoids `IllegalMonitorStateException` if unchecked exception occurs during `lock()` invocation (e.g., `OutOfMemoryError`), which would obscure original failure reason.
- `ReentrantLock` provides mutual exclusion.
- `ReentrantReadWriteLock`: Java API also provides this for scenarios with more readers than writers.
 - Allows multiple concurrent readers but only one writer.

4.4 Semaphores

- Java API provides counting semaphore (as in Section 6.6).
- Constructor: `Semaphore(int value)`.
- `value`: initial value of semaphore (negative allowed).
- `acquire()` method: throws `InterruptedException` if acquiring thread interrupted.
- Example using semaphore for mutual exclusion:


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

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```
- `release()` placed in `finally` clause to ensure semaphore is released.

4.5 Condition variables

- Java API utility: condition variable.
- Functionality similar to `wait()` and `notify()` methods.
- Must be associated with a reentrant lock for mutual exclusion.
- Creation:
 1. Create a `ReentrantLock`.

2. Invoke its `newCondition()` method.

- Returns a `Condition` object (representing condition variable for associated `ReentrantLock`).

- Example:

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

- Operations: `await()` and `signal()` methods.

- Function same as `wait()` and `signal()` in Section 6.7.

- Named vs. unnamed condition variables:

- Monitors (Section 6.7): `wait()` and `signal()` apply to *named* condition variables.
- Java (language level): does not provide named condition variables.
- Each Java monitor: associated with one unnamed condition variable.
- `wait()` and `notify()` (Section Java monitors): apply only to this single unnamed condition variable.
- When Java thread awakened via `notify()`: receives no info on why; reactivated thread must check condition itself.
- Condition variables (this section): remedy this by allowing specific thread to be notified.

- Example: Five threads (0-4), shared variable `turn`.

- `doWork(int threadNumber)` method (Figure ??):

- Only thread with `threadNumber` matching `turn` proceeds; others wait.

- `doWork()` method:

```
/* 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();
    }
}
```

- Requires `ReentrantLock` and five condition variables (`condVars`).

- Initialization:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];
```

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

- When thread enters `doWork()`:

- If `threadNumber != turn`: invokes `await()` on its associated condition variable.
- Resumes only when signaled by another thread.
- After work: signals condition variable for next thread's turn.

- `doWork()` does not need to be `synchronized`.

- `ReentrantLock` provides mutual exclusion.
- `await()` on condition variable releases associated `ReentrantLock`.
- `signal()` only signals condition variable; lock released by `unlock()`.

- Programming problems/projects at end of chapter use Pthreads mutex locks, condition variables, and POSIX semaphores.

4.6 Section glossary

- *entry set*: In Java, the set of threads waiting to enter a monitor.
- *wait set*: In Java, a set of threads, each waiting for a condition that will allow it to continue.
- *scope*: The time between when a lock is acquired and when it is released.

5 Alternative approaches

- Emergence of multicore systems: increased pressure to develop concurrent applications.
- Concurrent applications: increased risk of race conditions and liveness hazards (e.g., deadlock).
- Traditionally: mutex locks, semaphores, monitors used to address these.
- Challenge: as processing cores increase, designing multithreaded applications free from race conditions and deadlock becomes harder.
- This section explores: features in programming languages and hardware supporting thread-safe concurrent applications.

5.1 Transactional memory

- Idea originated in database theory, now used for process synchronization.
- *Memory transaction*: sequence of memory read-write operations that are atomic.
- If all operations complete: transaction committed.
- Otherwise: operations aborted and rolled back.
- Benefits obtained through language features.
- Example: `update()` function modifying shared data.
 - Traditional approach (with mutex locks/semaphores):

```
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```
 - Problems with traditional locking: deadlock, poor scalability with increasing threads (high contention for lock ownership).
 - Alternative: new language features using transactional memory.
 - Construct `atomic{S}`: ensures operations in `S` execute as a transaction.
 - Rewritten `update()` function:

```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```
 - Advantages of transactional memory:
 - * Transactional memory system (not developer) guarantees atomicity.
 - * No locks involved → deadlock not possible.
 - * System can identify concurrent execution of statements in atomic blocks (e.g., concurrent read access).
 - * Programmer identifying these situations (e.g., for reader-writer locks) becomes difficult as thread count grows.
- Implementation:
 - *Software transactional memory (STM)*:
 - * Implemented exclusively in software; no special hardware needed.
 - * Works by inserting instrumentation code inside transaction blocks (by compiler).
 - * Manages transactions by examining concurrency and low-level locking needs.
 - *Hardware transactional memory (HTM)*:
 - * Uses hardware cache hierarchies and cache coherency protocols.
 - * Manages and resolves conflicts for shared data in separate processors' caches.
 - * Requires no special code instrumentation (less overhead than STM).
 - * Requires modification of existing cache hierarchies and cache coherency protocols.
- Status: existed for years without widespread implementation.
- Current trend: growth of multicore systems and emphasis on concurrent/parallel programming has prompted significant research.

5.2 OpenMP

- Overview in Section Fork join: supports parallel programming in shared-memory environment.
- Includes: set of compiler directives and an API.
- `#pragma omp parallel`: compiler directive.
 - Code following this is a parallel region.
 - Performed by number of threads equal to processing cores.
- Advantage: OpenMP library handles thread creation and management (not application developers' responsibility).
- `#pragma omp critical`: compiler directive.
 - Specifies code region as a critical section.
 - Only one thread active at a time.
 - Ensures threads do not generate race conditions.
- Example: `update()` function modifying shared variable `counter`.

```
void update(int value)
{
    counter += value;
}
```

- If `update()` is part of/invoked from parallel region, race condition possible on `counter`.
- Remedy using critical-section compiler directive:

```
void update(int value)
{
    \#pragma omp critical
    {
        counter += value;
    }
}
```

- Behavior of critical-section directive:
 - Much like binary semaphore or mutex lock.
 - Ensures only one thread active in critical section at a time.
 - If thread tries to enter when another is active (owns section): calling thread blocks until owner exits.
 - Multiple critical sections: each can be named; rule specifies only one thread active in critical section of same name simultaneously.
- Advantages of OpenMP critical-section directive:
 - Generally considered easier to use than standard mutex locks.
- Disadvantages:
 - Developers must still identify possible race conditions.
 - Must adequately protect shared data using directive.
 - Deadlock still possible if two or more critical sections are identified (behaves like mutex lock).

5.3 Functional programming languages

- Most well-known languages (C, C++, Java, C#): *imperative* (or *procedural*) languages.
- Imperative languages:
 - Implement state-based algorithms.
 - Flow of algorithm crucial for correct operation.
 - State represented with variables and data structures.
 - Program state is mutable (variables can change values).
- Current emphasis on concurrent/parallel programming for multicore systems: greater focus on *functional* programming languages.
- Functional languages:
 - Different programming paradigm from imperative.
 - Fundamental difference: do not maintain state.
 - Once variable defined and assigned value, its value is immutable (cannot change).
 - Because mutable state disallowed: no concern with race conditions and deadlocks.
 - Most problems addressed in this chapter are nonexistent.
- Examples of functional languages:
 - Erlang: gained attention for concurrency support and ease of developing parallel applications.

- Scala: functional and object-oriented; syntax similar to Java and C#.

5.4 Section glossary

- *transactional memory*: Type of memory supporting memory transactions.
- *memory transaction*: Sequence of memory read-write operations that are atomic.
- *software transactional memory (STM)*: Transactional memory implemented exclusively in software; no special hardware needed.
- *hardware transactional memory (HTM)*: Transactional memory using hardware cache hierarchies and cache coherency protocols to manage/resolve conflicts for shared data in separate processors' caches.
- *imperative language*: Language for implementing state-based algorithms (e.g., C, C++, Java, C#).
- *procedural language*: A language that implements state-based algorithms (e.g., C, C++, Java, C#).
- *functional language*: Programming language that does not require states to be managed by programs written in it (e.g., Erlang, Scala).

6 Summary

- Classic process synchronization problems:
 - Bounded-buffer problem.
 - Readers-writers problem.
 - Dining-philosophers problem.
- Solutions use tools from "Synchronization Tools" chapter:
 - Mutex locks.
 - Semaphores.
 - Monitors.
 - Condition variables.
- **Windows synchronization:**
 - Uses dispatcher objects.
 - Uses events to implement synchronization tools.
- **Linux synchronization:**
 - Uses various approaches to protect against race conditions.
 - Includes atomic variables.
 - Includes spinlocks.
 - Includes mutex locks.
- **POSIX API synchronization:**
 - Provides mutex locks.
 - Provides semaphores.
 - Provides condition variables.
 - Two forms of semaphores:
 - * Named semaphores: easily accessed by unrelated processes by name.
 - * Unnamed semaphores: cannot be shared as easily; require placement in shared memory.
- **Java synchronization:**
 - Rich library and API for synchronization.
 - Available tools:
 - * Monitors (provided at language level).
 - * Reentrant locks (supported by API).
 - * Semaphores (supported by API).
 - * Condition variables (supported by API).
- **Alternative approaches to critical-section problem:**
 - Transactional memory.
 - OpenMP.
 - Functional languages.
- **Functional languages:**
 - Intriguing alternative programming paradigm to procedural languages.
 - Unlike procedural languages, do not maintain state.
 - Generally immune from race conditions and critical sections.