Operating Systems Notes - Chapter 6

June 9, 2025

Contents

1 Process Synchronization

1.1 Background

- Systems have many concurrent/parallel threads, often sharing user data.
- **Race condition**: Occurs when shared data access is uncontrolled, leading to potential data corruption.
- **Process synchronization**: Tools to control shared data access, preventing race conditions.
- **Caution**: Incorrect use of synchronization tools can cause poor system performance, including deadlock.

1.2 Introduction

- A **cooperating process** can affect or be affected by other processes.
- Cooperating processes share a logical address space (code and data) or data via shared memory/message passing.
- Concurrent access to shared data can lead to data inconsistency.
- This chapter focuses on mechanisms to ensure orderly execution of cooperating processes sharing a logical address space to maintain data consistency.

1.3 Chapter objectives

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

1.4 Background (continued)

- Processes can execute concurrently (CPU scheduler switches rapidly) or in parallel (simultaneous execution on separate cores).
- A process can be interrupted at any point, and the core reassigned to another process.
- Concurrent/parallel execution can lead to data integrity issues with shared data.
- **Example: Bounded Buffer with ‘count‘ variable**
  - Original bounded buffer allowed ‘BUFFER_SIZE - 1‘ items.
  - To remedy, added ‘count‘ variable, initialized to 0.
  - ‘count‘ increments when item added, decrements when item removed.
- **Producer Process Code Modification:**

```
while (true) {
     /* produce an item in next_produced */

     while (count ==  BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

- **Consumer Process Code Modification:**

```
while (true) {
     while (count == 0)
        ; /* do nothing */

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

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

- **Issue with Concurrent Execution:**
  - If ‘count‘ is 5, and producer (‘count++‘) and consumer (‘count–‘) execute concurrently.
  - Expected result: ‘count‘ == 5.
  - Possible incorrect results: ‘count‘ == 4, 5, or 6 due to interleaving.
- **Machine Language Implementation of ‘count++‘:**

```
register1 = count
register1 = register1 + 1
count = register1
```

- **Machine Language Implementation of ‘count–‘:**

```
register2 = count
register2 = register2 - 1
count = register2
```

- **Race Condition Interleaving Example (Incorrect 'count' == 4):**

```
T0: producer execute register1 = count         {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = count         {register2 = 5}
T3: consumer execute register2 = register2 - 1  {register2 = 4}
T4: producer execute count = register1         {count = 6}
T5: consumer execute count = register2         {count = 4}
```

- **Race Condition Definition:**
  - Several processes access and manipulate the same data concurrently.
  - Outcome depends on the specific order of access.
- **Solution Requirement:** To prevent race conditions, only one process at a time should manipulate shared variables (e.g., 'count'). This requires process synchronization.
- **Importance:** Race conditions are frequent in OS (resource manipulation) and multithreaded applications (shared data on multicore systems). Process synchronization and coordination are crucial to prevent interference.

1.5 Section glossary

Term	Definition
cooperating process	A process that can affect or be affected by other processes executing in the system.
race condition	A situation in which two threads are concurrently trying to change the value of a variable.
process synchronization	Coordination of access to data by two or more threads or processes.
coordination	Ordering of the access to data by multiple threads or processes.

2 The critical-section problem

- **Critical-section problem**: Designing a protocol for n processes $(P_0, \ldots, P_{n-1})$ to cooperatively share data and synchronize activity.
- Each process has a **critical section**:
  - Code segment where shared data is accessed and updated.
  - **Key rule**: Only one process allowed in its critical section at any given time (mutual exclusion).
- **Protocol components for each process:**
  - **Entry section**: Code to request permission to enter the critical section.
  - **Critical section**: The shared data access/update code.
  - **Exit section**: Code to cleanly exit the critical section.
  - **Remainder section**: All other code.
- **General process structure:**

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

- **Three requirements for a critical-section problem solution:**
  1. **Mutual exclusion**: If P_i is in its critical section, no other process can be.
  2. **Progress**:
     - If no process is in its critical section, and some wish to enter.
     - Only processes not in their remainder sections can participate in deciding who enters next.
     - Selection cannot be postponed indefinitely.
  3. **Bounded waiting**:
     - A limit exists on how many times other processes can enter their critical sections.
     - This limit applies after a process requests entry and before its request is granted.
- **Assumptions**:
  - Each process executes at a nonzero speed.
  - No assumptions about the relative speeds of processes.
- **Kernel Code and Race Conditions**:
  - Many kernel-mode processes are active in the OS.
  - **Kernel code** is susceptible to race conditions.
  - **Example 1**: Kernel data structure for open files.
    - Modified when files are opened/closed (add/remove from list).
    - Simultaneous file opening by two processes could cause a race condition on this list.
  - **Example 2**: Two processes using 'fork()' system call.
    - Race condition on 'next_available_pid' kernel variable.
    - Without mutual exclusion, same PID could be assigned to two processes.
  - **Other prone kernel data structures**: Memory allocation, process lists, interrupt handling.
  - Kernel developers must ensure OS is free from such race conditions.
- **Single-core vs. Multiprocessor Environments**:
  - **Single-core**: Critical-section problem could be solved by preventing interrupts during shared variable modification.
    - Ensures current instruction sequence executes without preemption.
    - No unexpected modifications to shared variable.
  - **Multiprocessor**: This solution is less feasible.
    - Disabling interrupts on multiprocessor is time-consuming (message to all processors).
    - Delays critical section entry, decreases system efficiency.
    - Affects system clock if updated by interrupts.
- **Approaches to handle critical sections in OS**:
  1. **Preemptive kernels**:

- Allows a process to be preempted while running in kernel mode.
- Must be carefully designed for shared kernel data to be race-condition free.
- Especially difficult for SMP architectures (two kernel-mode processes can run simultaneously).
- **Advantages**: More responsive (less risk of long kernel-mode runs), more suitable for real-time programming (real-time process can preempt kernel process).

2. **Nonpreemptive kernels**:
   - Does not allow a process in kernel mode to be preempted.
   - Kernel-mode process runs until it exits kernel mode, blocks, or voluntarily yields CPU.
   - Essentially free from race conditions on kernel data structures (only one process active in kernel at a time).

2.1 Section glossary

Term	Definition
critical section	A section of code responsible for changing data that must only be executed by one thread or process at a time to avoid a race condition.
entry section	The section of code within a process that requests permission to enter its critical section.
exit section	The section of code within a process that cleanly exits the critical section.
remainder section	Whatever code remains to be processed after the critical and exit sections.
preemptive kernel	A type of kernel that allows a process to be preempted while it is running in kernel mode.
nonpreemptive kernels	A type of kernel that does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

3 Peterson's solution

- **Peterson's solution**: A classic software-based solution to the critical-section problem.
- **Modern Architecture Limitations**:
  - Not guaranteed to work correctly on modern architectures due to reordering of 'load' and 'store' instructions.
  - Presented for its algorithmic description and illustration of complexities in designing software for mutual exclusion, progress, and bounded waiting.
- **Scope**: Restricted to two processes (P_0, P_1) that alternate execution between critical and remainder sections.
- **Notation**: When discussing P_i, P_j denotes the other process (i.e., $j = 1 - i$).
- **Shared Data Items**:

```
int turn;
boolean flag[2];
```

  - 'turn': Indicates whose turn it is to enter the critical section ('turn == i' means P_i can enter).
  - 'flag' array: Indicates if a process is ready to enter its critical section ('flag[i] == true' means P_i is ready).
- **Peterson's Algorithm (for process P_i):**

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

- **Entry Mechanism**:
  - P_i sets 'flag[i] = true'.
  - P_i sets 'turn = j', signaling that P_j can enter if it wishes.
  - If both processes try to enter concurrently, 'turn' is set to both 'i' and 'j' almost simultaneously.
  - Only one 'turn' assignment will persist; the final value determines which process enters first.
- **Proof of Correctness (Requirements):**
  1. **Mutual exclusion is preserved**:
     - P_i enters critical section only if 'flag[j] == false' OR 'turn == i'.
     - If both P_0 and P_1 are in critical sections, then 'flag[0] == true' and 'flag[1] == true'.
     - This implies they couldn't have both passed their 'while' statements simultaneously, as 'turn' can only be 0 or 1.
     - One process (e.g., P_j) must have successfully executed its 'while' statement.
     - At that point, 'flag[j] == true' and 'turn == j', which persists while P_j is in its critical section, thus preserving mutual exclusion.
  2. **Progress requirement is satisfied**:
     - P_i is prevented from entering only if stuck in 'while (flag[j] && turn == j)'.
     - If P_j is not ready ('flag[j] == false'), P_i enters.
     - If P_j is ready ('flag[j] == true') and in its 'while' loop:
       - If 'turn == i', P_i enters.
       - If 'turn == j', P_j enters.
     - Once P_j exits its critical section, it sets 'flag[j] = false', allowing P_i to enter.
     - If P_j re-enters, it sets 'flag[j] = true' and 'turn = i'.
     - Since P_i doesn't change 'turn' in its 'while' loop, P_i will eventually enter.
  3. **Bounded-waiting requirement is met**:
     - As shown above, P_i will enter after at most one entry by P_j.
- **Why Peterson's Solution Fails on Modern Architectures**:
  - Processors/compilers may reorder read/write operations without data dependencies for performance.
  - For single-threaded apps, reordering is fine (final values consistent).
  - For multithreaded apps with shared data, reordering can lead to inconsistent/unexpected results.
- **Reordering Example**:
  - Shared data: 'boolean flag = false; int x = 0;'

- Thread 1:

```
while (!flag)
   ;
print x;
```

- Thread 2:

```
x = 100;
flag = true;
```

- **Expected**: Thread 1 prints 100.
- **Possible Reordering (Thread 2)**: ‘flag = true; x = 100;‘
  - Thread 1 could print 0 if ‘flag‘ is set before ‘x‘ is updated.
- **Possible Reordering (Thread 1)**: Processor loads ‘x‘ before ‘flag‘.
  - Thread 1 could print 0 even if Thread 2’s instructions are not reordered.
- **Impact on Peterson’s Solution**:
  - If the first two statements in Peterson’s entry section (‘flag[i] = true; turn = j;‘) are reordered.
  - It’s possible for both threads to be in their critical sections simultaneously.
- **Conclusion**: Proper synchronization tools are necessary to preserve mutual exclusion.
- **Next Steps**: Discussion will cover hardware support and abstract software APIs for synchronization.

3.1 Section glossary

Term	Definition
Peterson’s solution	A historically interesting algorithm for implementing critical sections.

4 Hardware support for synchronization

- Software-based solutions (like Peterson's) are not guaranteed on modern architectures.
- This section introduces three hardware instructions for critical-section problem support.
- These primitives can be used directly or as foundations for more abstract synchronization.

4.1 Memory barriers

- **Problem**: Systems may reorder instructions, leading to unreliable data states.
- **Memory model**: How a computer architecture guarantees memory visibility to applications.
- **Categories of Memory Models**:
  1. **Strongly ordered**: Memory modification on one processor is immediately visible to all others.
  2. **Weakly ordered**: Memory modifications on one processor may not be immediately visible to others.
- **Issue**: Kernel developers cannot assume memory modification visibility on shared-memory multiprocessors due to varying memory models.
- **Solution**: **Memory barriers** (or **memory fences**) are instructions that force memory changes to propagate to all other processors.
- **Functionality**: When a memory barrier is performed, all preceding loads and stores are completed before any subsequent load or store operations.
- **Benefit**: Even if instructions are reordered, memory barriers ensure store operations are completed and visible before future operations.
- **Example (Thread 1 with memory barrier):**

```
while (!flag)
  memory\_barrier();
print x;
```

  * Guarantees 'flag' is loaded before 'x'.
- **Example (Thread 2 with memory barrier):**

```
x = 100;
memory\_barrier();
flag = true;
```

  * Ensures assignment to 'x' occurs before assignment to 'flag'.
- **Application to Peterson's Solution**: A memory barrier could be placed between the first two assignment statements in the entry section to prevent reordering.
- **Note**: Memory barriers are low-level operations, typically used by kernel developers for specialized mutual exclusion code.

4.2 Hardware instructions

- Modern systems provide special hardware instructions for **atomically** (as one uninterruptible unit) testing/modifying a word or swapping two words.
- These instructions simplify critical-section problem solving.
- **Abstracted Instructions**: 'test_and_set()' and 'compare_and_swap()' (CAS).
- **'test_and_set()' instruction**:
  * **Definition**:

```
boolean test\_and\_set(boolean *target) {
  boolean rv = *target;
  *target = true;

  return rv;
}
```

  * **Characteristic**: Executed atomically.
  * **Behavior**: If two 'test_and_set()' instructions run simultaneously (on different cores), they execute sequentially in an arbitrary order.
  * **Mutual Exclusion Implementation**:
    · Declare 'boolean lock', initialized to 'false'.
    · **Process P_i structure**:

```
do {
  while (test\_and\_set(&lock))
     ; /* do nothing */

     /* critical section */

  lock = false;

     /* remainder section */
} while (true);
```

– **‘compare_and_swap()‘ (CAS) instruction**:
  * Operates on three operands atomically.
  * **Definition**:

```
int compare\_and\_swap(int *value, int expected, int new\_value) {
  int temp = *value;

  if (*value == expected)
    *value = new\_value;

  return temp;
}
```

  * **Functionality**: ‘value‘ is set to ‘new_value‘ ONLY if ‘(*value == expected)‘ is true.
  * **Return Value**: Always returns the original value of ‘value‘.
  * **Characteristic**: Executed atomically.
  * **Behavior**: If two CAS instructions run simultaneously, they execute sequentially.
  * **Mutual Exclusion Implementation (Basic)**:
    · Global variable ‘lock‘, initialized to 0.
    · First process calling ‘compare_and_swap(&lock, 0, 1)‘ sets ‘lock‘ to 1 and enters critical section (original ‘lock‘ was 0).
    · Subsequent calls fail (current ‘lock‘ is not 0).
    · Process exits critical section: sets ‘lock‘ back to 0, allowing another process.
    · **Process P_i structure**:

```
while (true) {
  while (compare\_and\_swap(&lock, 0, 1) != 0)
     ; /* do nothing */

     /* critical section */

     lock = 0;

     /* remainder section */
}
```

    · **Limitation**: This basic algorithm satisfies mutual exclusion but NOT bounded waiting.
  * **Mutual Exclusion with Bounded Waiting using CAS**:
    · **Common data structures**:

```
boolean waiting[n];
int lock;
```

    · ‘waiting‘ array elements initialized to ‘false‘, ‘lock‘ initialized to 0.
    · **Mutual Exclusion Proof**:
    · P_i enters the critical section only if `waiting[i] == false || key == 0`.
    · ‘key‘ becomes 0 only if ‘compare_and_swap()‘ is executed.
    · First process executing CAS finds ‘key == 0‘; others wait.
    · ‘waiting[i]‘ becomes ‘false‘ only when another process leaves critical section; only one ‘waiting[i]‘ is set to ‘false‘, maintaining mutual exclusion.
    · **Process P_i structure (with bounded waiting):**

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare\_and\_swap(&lock,0,1);
  waiting[i] = false;

    /* critical section */

  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = 0;
  else
     waiting[j] = false;

     /* remainder section */
}
```

    · **Progress Proof**: A process exiting critical section sets ‘lock‘ to 0 or ‘waiting[j]‘ to ‘false‘, allowing a waiting process to proceed.

· **Bounded-Waiting Proof**:
· When a process leaves its critical section, it cyclically scans ‘waiting‘ array $(i+1, \ldots, n-1, 0, \ldots, i-1)$.
· It designates the first process in the entry section (‘waiting[j] == true‘) as the next to enter.
· Any waiting process will enter its critical section within $n-1$ turns.
∗ **Intel x86 Architecture**: ‘cmpxchg‘ assembly instruction implements ‘compare_and_swap()‘.
· ‘lock‘ prefix used to enforce atomic execution by locking the bus during destination operand update.
· **General form**: ‘lock cmpxchg ¡destination operand¿, ¡source operand¿.‘

4.3 Atomic variables

– ‘compare_and_swap()‘ is often a building block for other synchronization tools, not used directly for mutual exclusion.
– **Atomic variable**: A programming construct providing atomic operations on basic data types (integers, booleans).
– **Purpose**: Ensures mutual exclusion for single variable updates (e.g., counter increments) where data races might occur.
– **Implementation**: Often use ‘compare_and_swap()‘ operations.
– **Example: Incrementing atomic integer ‘sequence‘**:

```
increment(&sequence);
```

– **‘increment()‘ function using CAS**:

```
void increment(atomic\_int *v)
{
  int temp;

  do {
    temp = *v;
  }
  while (temp != compare\_and\_swap(v, temp, temp+1));
}
```

– **Limitation**: Atomic variables provide atomic updates but don’t solve all race conditions.
– **Bounded-Buffer Problem Example (with atomic ‘count‘):**
∗ If buffer is empty and two consumers wait for ‘count ¿ 0‘.
∗ Producer adds one item, ‘count‘ becomes 1 (atomically).
∗ Both consumers could exit their ‘while‘ loops and proceed to consume, even though ‘count‘ is only 1.
– **Usage**: Commonly used in OS and concurrent applications, but often limited to single updates of shared data (counters, sequence generators).
– **Next**: Explore more robust tools for generalized race conditions.

4.4 Section glossary

Term	Definition
memory model	Computer architecture memory guarantee, usually either strongly ordered or weakly ordered.
memory barriers	Computer instructions that force any changes in memory to be propagated to all other processors in the system.
memory fences	Computer instructions that force any changes in memory to be propagated to all other processors in the system.
atomically	A computer activity (such as a CPU instruction) that operates as one uninterruptable unit.
atomic variable	A programming language construct that provides atomic operations on basic data types such as integers and booleans.

5 Mutex locks

- Hardware-based solutions (Section **??**) are complex and generally inaccessible to application programmers.
- Operating system designers build higher-level software tools.
- The simplest tool is the **mutex lock** (short for **mutual exclusion**).
- **Purpose**: Protect critical sections and prevent race conditions.
- **Usage**: A process must ‘acquire()‘ the lock before entering a critical section and ‘release()‘ it upon exiting.
- **General structure**:

```
while (true) {

   acquire lock

       critical section

   release lock

       remainder section

}
```

- **Mutex Lock Properties**:
  - Has a boolean variable ‘available‘.
  - ‘available = true‘: Lock is available.
  - ‘acquire()‘ succeeds if ‘available‘ is true, then sets ‘available = false‘.
  - Process attempting to acquire an unavailable lock is blocked until released.
- **‘acquire()‘ definition**:

```
acquire() {
   while (!available)
     ; /* busy wait */
   available = false;
}
```

- **‘release()‘ definition**:

```
release() {
  available = true;
}
```

- **Atomicity**: Calls to ‘acquire()‘ and ‘release()‘ must be atomic.
  - Can be implemented using the CAS operation (Section **??**).
- **Lock Contention**:
  - **Contended lock**: A thread blocks while trying to acquire the lock.
  - **Uncontended lock**: Lock is available when a thread attempts to acquire it.
  - **High contention**: Many threads attempt to acquire the lock.
  - **Low contention**: Few threads attempt to acquire the lock.
  - Highly contended locks decrease overall performance of concurrent applications.
- **Spinlocks**:
  - The type of mutex lock described is also called a **spinlock**.
  - Process "spins" (loops continuously) while waiting for the lock.
  - **Disadvantage**: Requires **busy waiting**.
    - Wastes CPU cycles, especially problematic in single-CPU core multiprogramming systems.
    - (Section **??** discusses strategies to avoid busy waiting by putting processes to sleep.)
  - **Advantage**: No context switch required when waiting on a lock.
    - Context switches can be time-consuming.
    - Preferable on multicore systems for short-duration locks.
    - One thread can "spin" on one core while another performs its critical section on another core.
    - Widely used in modern operating systems on multicore systems.
  - **Rule of thumb**: Use a spinlock if the lock will be held for less than two context switches (as waiting involves two context switches).
- **Further Discussion**: Chapter Synchronization Examples will cover mutex locks in classical synchronization problems and their use in various operating systems and Pthreads.

Section glossary

Term	Definition
mutex lock	A mutual exclusion lock; the simplest software tool for assuring mutual exclusion.
contended	A term describing the condition of a lock when a thread blocks while trying to acquire it.
uncontended	A term describing a lock that is available when a thread attempts to acquire it.
busy waiting	A practice that allows a thread or process to use CPU time continuously while waiting for something. An I/O loop in which an I/O thread continuously reads status information while waiting for I/O to complete.
spinlock	A locking mechanism that continuously uses the CPU while waiting for access to the lock.

6 Semaphores

- Mutex locks are simple synchronization tools.
- **Semaphore** S: A more robust integer variable, accessed only through two standard atomic operations: ‘wait()‘ and ‘signal()‘.
- Introduced by Edsger Dijkstra.
- **Original terms**: ‘wait()‘ was ‘P‘ (proberen, ”to test”); ‘signal()‘ was ‘V‘ (verhogen, ”to increment”).
- **‘wait(S)‘ definition (classical, with busy waiting)**:

```
wait(S) {
   while (S <= 0)
     ; /* busy wait */
     S--;
}
```

- **‘signal(S)‘ definition (classical)**:

```
signal(S) {
   S++;
}
```

- **Atomicity Requirement**:
  - All modifications to semaphore value in ‘wait()‘ and ‘signal()‘ must be atomic.
  - No two processes can simultaneously modify the same semaphore value.
  - For ‘wait(S)‘, testing ‘S ¡= 0‘ and decrementing ‘S–‘ must be uninterruptible.

6.1 Semaphore usage

- Operating systems distinguish between two types of semaphores:
  1. **Counting semaphore**:
     - Value can range over an unrestricted domain.
     - Used to control access to a resource with a finite number of instances.
     - Initialized to the number of available resources.
     - ‘wait()‘: Decrements count when a process wishes to use a resource.
     - ‘signal()‘: Increments count when a process releases a resource.
     - If count goes to 0, all resources are in use; processes block until count ¿ 0.
  2. **Binary semaphore**:
     - Value can only be 0 or 1.
     - Behaves similarly to mutex locks.
     - Can be used for mutual exclusion on systems without mutex locks.
- **Solving Synchronization Problems (Example)**:
  - Two concurrent processes: P_1 with statement S_1, P_2 with statement S_2.
  - Requirement: S_2 executes only after S_1 completes.
  - **Implementation**:
    - Share a common semaphore ‘synch‘, initialized to 0.
    - **In process** P_1:

      ```
      S1;
      signal(synch);
      ```

    - **In process** P_2:

      ```
      wait(synch);
      S2;
      ```

    - Result: P_2 executes S_2 only after P_1 calls ‘signal(synch)‘ (i.e., after S_1 completes).

6.2 Semaphore implementation

- The classical ‘wait()‘ and ‘signal()‘ definitions suffer from busy waiting (like mutex locks in Section **??**).
- **To overcome busy waiting**:
  - When ‘wait()‘ finds semaphore value not positive, process suspends itself instead of busy waiting.
  - **Suspend operation**: Places process into a waiting queue associated with the semaphore; process state switches to waiting.
  - Control transfers to CPU scheduler, which selects another process.
  - **Restarting a suspended process**: When another process executes ‘signal()‘, the suspended process is restarted by a ‘wakeup()‘ operation.

    - **‘wakeup()‘ operation**: Changes process from waiting to ready state, places it in the ready queue.
    - CPU scheduling algorithm determines if CPU switches to newly ready process.
- **Semaphore definition for this implementation**:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

    - ‘value‘: Integer value of the semaphore.
    - ‘list‘: List of processes waiting on this semaphore.
- **‘wait(semaphore *S)‘ definition (without busy waiting)**:

```
wait(semaphore *S) {
     S-}value--;
     if (S-}value < 0) {
            add this process to S-}list;
            sleep();
     }
}
```

- **‘signal(semaphore *S)‘ definition (without busy waiting)**:

```
signal(semaphore *S) {
     S-}value++;
     if (S-}value <= 0) {
            remove a process P from S-}list;
            wakeup(P);
     }
}
```

- **‘sleep()‘ operation**: Suspends the invoking process (system call).
- **‘wakeup(P)‘ operation**: Resumes execution of suspended process ‘P‘ (system call).
- **Negative Semaphore Values**:
    - In this implementation, semaphore values can be negative.
    - Magnitude of a negative value = number of processes waiting on that semaphore.
    - This results from decrementing ‘S-¿value‘ before testing it in ‘wait()‘.
- **Waiting Process List Implementation**:
    - Can be a link field in each Process Control Block (PCB).
    - Semaphore contains integer value and pointer to a list of PCBs.
    - FIFO queue (with head/tail pointers) ensures bounded waiting.
    - Any queuing strategy can be used; correct semaphore usage doesn’t depend on it.
- **Atomicity of Semaphore Operations (Crucial)**:
    - No two processes can execute ‘wait()‘ and ‘signal()‘ on the same semaphore simultaneously.
    - **Single-processor environment**: Solved by inhibiting interrupts during ‘wait()‘ and ‘signal()‘ execution.
        - Ensures current process executes without interleaving until interrupts reenabled.
    - **Multicore environment**:
        - Interrupts must be disabled on *every* processing core (difficult, diminishes performance).
        - SMP systems use alternative techniques like ‘compare_and_swap()‘ or spinlocks to ensure atomicity of ‘wait()‘ and ‘signal()‘.
- **Busy Waiting Re-evaluation**:
    - Busy waiting is not entirely eliminated; it’s moved from application critical sections to the critical sections of ‘wait()‘ and ‘signal()‘ operations themselves.
    - These critical sections are very short (e.g., 10 instructions).
    - Busy waiting occurs rarely and for a short time in this context.
    - In contrast, application critical sections can be long (minutes/hours) or always occupied, making busy waiting highly inefficient there.

Section glossary

Term	Definition
semaphore	An integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
counting semaphore	A semaphore that has a value between 0 and N, to control access to a resource with N instances.
binary semaphore	A semaphore of values 0 and 1 that limits access to one resource (acting similarly to a mutex lock).

7 Monitors

- Semaphores are convenient but incorrect use can lead to hard-to-detect timing errors.
- These errors occur only with specific execution sequences and may not be reproducible.
- **Example**: Producer-consumer problem with ‘count‘ (Section 1) showed ‘count‘ off by 1, an unacceptable solution.
- Timing errors can still occur with mutex locks or semaphores if not used correctly.
- **Review of Semaphore Critical-Section Solution**:
  - Processes share binary semaphore ‘mutex‘, initialized to 1.
  - Each process must execute ‘wait(mutex)‘ before critical section and ‘signal(mutex)‘ afterward.
  - If this sequence is not observed, mutual exclusion can be violated.
  - Difficulties arise even if a *single* process is ill-behaved (due to programming error or uncooperative programmer).
- **Examples of Incorrect Semaphore Usage**:
  - **Interchanged ‘wait()‘ and ‘signal()‘**:

```
signal(mutex);
   ...
  critical section
   ...
wait(mutex);
```

    - Result: Multiple processes may enter critical section simultaneously, violating mutual exclusion.
    - Error may not always be reproducible.
  - **Replacing ‘signal(mutex)‘ with ‘wait(mutex)‘**:

```
wait(mutex);
   ...
  critical section
   ...
wait(mutex);
```

    - Result: Process permanently blocks on the second ‘wait()‘ call (semaphore unavailable).
  - **Omitting ‘wait(mutex)‘ or ‘signal(mutex)‘ (or both)**:
    - Result: Mutual exclusion violated or process permanently blocks.
- These examples show how easily errors can be generated with incorrect semaphore/mutex lock usage.
- **Solution Strategy**: Incorporate simple synchronization tools as high-level language constructs.
- This section describes the **monitor type**, a fundamental high-level synchronization construct.

7.1 Monitor usage

- **Abstract Data Type (ADT)**: Encapsulates data with a set of functions, independent of implementation.
- **Monitor type**: An ADT that includes programmer-defined operations with mutual exclusion *within* the monitor.
- Declares variables defining its state and bodies of functions operating on those variables.
- **Syntax of a monitor type**:

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
    . . .
  }

  function P2 ( . . . ) {
    . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
    . . .
  }

  initialization_code ( . . . ) {
    . . .
  }
}
```

- **Access Rules**:
  - Monitor type representation cannot be used directly by processes.
  - Functions within a monitor can only access locally declared variables and formal parameters.
  - Local variables of a monitor can only be accessed by local functions.

- **Mutual Exclusion Guarantee**: The monitor construct ensures only one process is active within the monitor at a time.
  - Programmer does not need to explicitly code this synchronization constraint.
- **Limitation**: Monitor construct alone is not powerful enough for all synchronization schemes.
- **Additional Mechanism**: The 'condition' construct.
  - Programmers define one or more variables of type 'condition':

    ```
    condition x, y;
    ```
  - **Operations on condition variables**: Only 'wait()' and 'signal()'.
  - **'x.wait()' operation**:

    ```
    x.wait();
    ```
    - Process invoking it is suspended until another process invokes 'x.signal()'.
  - **'x.signal()' operation**:

    ```
    x.signal();
    ```
    - Resumes exactly one suspended process.
    - If no process is suspended, 'signal()' has no effect (state of 'x' unchanged).
    - **Contrast with semaphore 'signal()'**: Semaphore 'signal()' always affects its state.
- **'x.signal()' and Concurrent Processes**:
  - If process P invokes 'x.signal()' and process Q is suspended on 'x'.
  - If Q resumes, P must wait (otherwise both P and Q would be active in monitor simultaneously).
  - **Two possibilities for P and Q continuation**:
    1. **Signal and wait**: P waits until Q leaves the monitor or waits for another condition.
    2. **Signal and continue**: Q waits until P leaves the monitor or waits for another condition.
  - **Arguments**:
    - Signal-and-continue seems more reasonable as P was already executing in the monitor.
    - However, if P continues, the logical condition Q was waiting for might no longer hold when Q resumes.
  - **Compromise**: When P executes 'signal()', it immediately leaves the monitor, and Q is immediately resumed.
- **Language Support**: Many languages (Java, C#) incorporate monitors; others (Erlang) provide similar concurrency support.

7.2 Implementing a monitor using semaphores

- **Mutual Exclusion**:
  - For each monitor, a binary semaphore 'mutex' (initialized to 1) ensures mutual exclusion.
  - Process executes 'wait(mutex)' before entering monitor, 'signal(mutex)' after leaving.
- **Signal-and-Wait Scheme Implementation**:
  - An additional binary semaphore 'next' (initialized to 0) is introduced.
  - Signaling processes use 'next' to suspend themselves.
  - Integer variable 'next_count' counts processes suspended on 'next'.
  - **Each external function 'F' is replaced by**:

    ```
    wait(mutex);
        ...
      body of F
        ...
    if (next_count > 0)
      signal(next);
    else
      signal(mutex);
    ```
  - Mutual exclusion within the monitor is ensured.
- **Condition Variable Implementation (using semaphores)**:
  - For each condition 'x', introduce binary semaphore 'x_sem' and integer 'x_count' (both initialized to 0).
  - **'x.wait()' implementation**:

    ```
    x_count++;
    if (next_count > 0)
      signal(next);
    else
      signal(mutex);
    wait(x_sem);
    x_count--;
    ```

* **'x.signal()' implementation**:

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

* This implementation applies to Hoare and Brinch-Hansen monitor definitions.
* Efficiency improvements are possible in some cases where generality is not needed.

7.3 Resuming processes within a monitor

- **Problem**: If multiple processes are suspended on condition 'x' and 'x.signal()' is executed, which process resumes?
- **Simple Solution**: First-Come, First-Served (FCFS) ordering (longest waiting process resumes first).
- **Limitation of FCFS**: Not adequate in many circumstances.
- **Alternative**: **Conditional-wait** construct.
  * **Form**:

    ```
    x.wait(c);
    ```

  * 'c': Integer expression evaluated when 'wait()' is executed.
  * 'c' is a **priority number**, stored with the suspended process.
  * When 'x.signal()' is executed, the process with the smallest priority number resumes.
- **Example: 'ResourceAllocator' monitor**:
  * Controls allocation of a single resource among competing processes.
  * Process specifies maximum resource usage time when requesting.
  * Monitor allocates resource to process with shortest time-allocation request.
  * **Required access sequence for a process**:

    ```
    R.acquire(t);
        ...
     access the resource;
        ...
    R.release();
    ```

  * 'R' is an instance of 'ResourceAllocator'.
  * **'ResourceAllocator' monitor structure**:

    ```
    monitor ResourceAllocator
    {
      boolean busy;
      condition x;

      void acquire(int time) {
        if (busy)
          x.wait(time);
        busy = true;
      }

      void release() {
        busy = false;
        x.signal();
      }
      initialization_code() {
        busy = false;
      }
    }
    ```

- **Limitations of Monitor Concept (regarding access sequence)**:
  * Cannot guarantee the required access sequence will be observed.
  * **Possible problems**:
    · Process accesses resource without permission.
    · Process never releases a granted resource.
    · Process attempts to release a resource it never requested.
    · Process requests same resource twice without releasing.
  * These are similar to semaphore misuse issues, but now with higher-level operations where the compiler cannot assist.
- **Possible Solution (and its drawback)**: Include resource-access operations within 'ResourceAllocator' monitor.
  * Drawback: Scheduling would follow built-in monitor-scheduling algorithm, not the custom one.
- **Ensuring Correct Sequence and Preventing Direct Access**:
  * Requires inspecting all programs using 'ResourceAllocator' monitor and its resource.

    - **Two conditions to check for correctness**:
        1. User processes always make calls on the monitor in a correct sequence.
        2. Uncooperative processes do not ignore mutual-exclusion gateway and access shared resource directly.
    - Only if both conditions are ensured can time-dependent errors be prevented and scheduling algorithm not be defeated.
- **Scalability Issue**: Inspection is feasible for small, static systems but not large or dynamic ones.
- **Ultimate Solution**: This access-control problem requires additional mechanisms described in the chapter Protection.

Section glossary

Term	Definition
monitor	A high-level language synchronization construct that protects variables from race conditions.
abstract data type (ADT)	A programming construct that encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.
conditional-wait	A component of the monitor construct that allows for waiting on a variable with a priority number to indicate which process should get the lock next.
priority number	A number indicating the position of a process in a conditional-wait queue in a monitor construct.

8 Liveness

- The critical-section problem is a type of **liveness** problem.
- **Liveness**: Properties a system must satisfy to ensure processes make progress.
- This section discusses two common liveness problems: deadlock and starvation.

8.1 Deadlock

- In multiprogramming, processes compete for finite resources.
- **Normal process operation sequence**:
  1. **Request**: Process requests resource. If unavailable, process waits.
  2. **Use**: Process operates on the resource.
  3. **Release**: Process releases the resource.
- **Deadlock**: A situation where two or more processes wait indefinitely for an event that can only be caused by one of the waiting processes.
- **Consequences**: Deadlocked processes never complete, and their resources are never released.
- **Example (Printer and DVD drive)**:
  - P_1 holds DVD drive, P_2 holds printer.
  - P_1 requests printer, P_2 requests DVD drive.
  - Result: Deadlock. P_1 waits for P_2's printer, P_2 waits for P_1's DVD drive. Neither can proceed or release resources.
- **Deadlock Scenarios**:
  - **Single-core system**: Process waits for an event that never occurs (e.g., message from a terminated process).
  - **Multicore system**: Two or more processes wait for each other to release resources.
- **Scope**: Deadlock is not limited to process synchronization; it can occur in memory management, file systems, etc.
- **Semaphores and Deadlock**: Semaphores can lead to deadlock.
- **Example (Semaphores S and Q, both initialized to 1)**:
  - **Process** P_0:

```
wait(S);
wait(Q);
...
signal(S);
signal(Q);
```

  - **Process** P_1:

```
wait(Q);
wait(S);
...
signal(Q);
signal(S);
```

  - **Scenario**: P_0 executes 'wait(S)', then P_1 executes 'wait(Q)'.
  - **Result**:
    - P_0 waits for 'wait(Q)' (needs P_1 to 'signal(Q)').
    - P_1 waits for 'wait(S)' (needs P_0 to 'signal(S)').
    - Both 'signal()' operations never execute, leading to deadlock.
- Deadlock can involve multiple processes and resources.
- **Further Discussion**: Chapter Deadlocks covers prevention, avoidance, detection, and recovery methods.

8.2 Starvation

- **Starvation**: A process is ready to run but cannot obtain a required resource (CPU core, mutex lock) for an indefinite period.
- **Distinction from Deadlock**: In starvation, resources *do* become available, but the process is repeatedly denied access. In deadlock, the event never occurs.
- **Example (Priority-based CPU scheduling)**:
  - Low-priority process is continuously preempted by high-priority processes.
  - May never get a chance to execute.
  - This is a form of starvation.
- **Example (Mutex locks)**:
  - One process repeatedly acquires and releases a mutex lock.
  - Other waiting processes may never acquire the lock if the scheduling algorithm continuously favors the same processes.

- **Impact**: Serious problem in real-time systems; a starving process may miss its deadline, causing system failure.
- **Techniques for Preventing Starvation**:
  1. **Aging**: Gradually increases the priority of processes waiting in the system for a long time.
     * Example: If priorities are 0-127, increment priority by 1 every 15 minutes.
     * Eventually, the process reaches highest priority and executes.
  2. **Fair scheduling algorithm**: Ensures every process eventually gets access to needed resources.

Section glossary

Term	Definition
liveness	A set of properties that a system must satisfy to ensure that processes make progress.
deadlock	A situation in which two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
starvation	A situation in which a process is ready to run but is unable to obtain a required resource for an indefinite period.
aging	A technique of gradually increasing the priority of processes that wait in the system for a long time.

9 Evaluation

- This section evaluates different synchronization tools for solving the critical-section problem.
- Given correct implementation and usage, these tools ensure mutual exclusion and address liveness issues.
- With the rise of concurrent programs on multicore systems, performance of synchronization tools is crucial.
- Choosing the right tool can be challenging. This section provides strategies for tool selection.
- **Hardware Solutions (Section ??)**:
  - Considered very low-level.
  - Typically used as foundations for constructing other synchronization tools (e.g., mutex locks).
  - Recent focus on using CAS instruction for **lock-free** algorithms.
    - **Lock-free algorithms**: Provide race condition protection without locking overhead.
    - **Advantages**: Low overhead, ability to scale.
    - **Disadvantage**: Often difficult to develop and test.
- **Priority Inversion**:
  - More than a scheduling inconvenience, especially in real-time systems.
  - Can cause a process to take longer than expected, leading to cascading failures and system failure.
  - **Mars Pathfinder Example (1997)**:
    - Sojourner rover experienced frequent computer resets.
    - **Cause**: High-priority task ("bc_dist") was delayed waiting for a shared resource held by a lower-priority task ("ASI/MET").
    - The lower-priority task was preempted by multiple medium-priority tasks.
    - "bc_dist" stalled, "bc_sched" detected the problem and initiated a reset.
    - This was a typical case of priority inversion.
    - **Solution**: VxWorks real-time OS (on Sojourner) had a global variable to enable priority inheritance on all semaphores. Setting this variable solved the problem.
- **CAS-based Synchronization vs. Traditional Locking**:
  - **CAS-based (Optimistic approach)**:
    - Optimistically update a variable first.
    - Use collision detection to see if another thread updated concurrently.
    - If conflict, retry operation until successful without conflict.
  - **Mutual-exclusion locking (Pessimistic strategy)**:
    - Assume another thread is concurrently updating the variable.
    - Pessimistically acquire the lock before making any updates.
- **Performance Guidelines (CAS vs. Traditional Synchronization)**:
  - **Uncontended loads**: Both are generally fast; CAS protection is somewhat faster.
  - **Moderate contention**: CAS protection is faster (possibly much faster).
    - CAS operation succeeds most of the time.
    - If it fails, it iterates only a few times before succeeding.
    - Traditional locking: Any contended lock acquisition involves a more complex, time-intensive code path (suspends thread, places on wait queue, context switch).
  - **High contention**: Traditional synchronization is ultimately faster than CAS-based synchronization.
- **Choosing Synchronization Mechanisms and Performance Impact**:
  - Atomic integers are much lighter-weight than traditional locks.
  - More appropriate than mutex locks or semaphores for single updates to shared variables (e.g., counters).
  - Spinlocks are used on multiprocessor systems when locks are held for short durations.
  - Mutex locks are simpler and have less overhead than semaphores.
  - Mutex locks are preferable to binary semaphores for protecting critical section access.
  - Counting semaphores are more appropriate than mutex locks for controlling access to a finite number of resources.
  - Reader-writer locks may be preferred over mutex locks for higher concurrency (multiple readers allowed).
- **Higher-Level Tools (Monitors, Condition Variables)**:
  - **Appeal**: Simplicity and ease of use.
  - **Drawbacks**: Significant overhead; may scale less effectively in highly contended situations depending on implementation.
- **Ongoing Research**: Much research is focused on developing scalable, efficient tools for concurrent programming.

    * Designing compilers for more efficient code.
    * Developing languages with built-in concurrent programming support.
    * Improving performance of existing libraries and APIs.
- **Next Chapter**: Examines how various operating systems and APIs implement the synchronization tools discussed in this chapter.

Section glossary

Term	Definition
lock-free	An algorithmic strategy that provides protection from race conditions without requiring the overhead of locking.

10 Summary

- **Race Condition**:
  - Occurs when processes concurrently access shared data.
  - Final result depends on the specific order of concurrent accesses.
  - Can lead to corrupted values of shared data.
- **Critical Section**:
  - A code segment where shared data may be manipulated.
  - A possible race condition may occur here.
  - **Critical-section problem**: Design a protocol for processes to synchronize activity and cooperatively share data.
- **Requirements for a Critical-Section Solution**:
  1. **Mutual exclusion**: Only one process active in its critical section at a time.
  2. **Progress**: Processes cooperatively determine which process enters its critical section next.
  3. **Bounded waiting**: Limits the time a program waits before entering its critical section.
- **Software Solutions**:
  - Peterson's solution is an example.
  - Do not work well on modern computer architectures due to instruction reordering.
- **Hardware Support for Critical Section**:
  - Memory barriers.
  - Hardware instructions (e.g., 'compare-and-swap' instruction).
  - Atomic variables.
- **Mutex Locks**:
  - Provide mutual exclusion.
  - Process must acquire a lock before entering a critical section.
  - Process must release the lock on exiting the critical section.
- **Semaphores**:
  - Can provide mutual exclusion, similar to mutex locks.
  - Unlike mutex locks (binary value), semaphores have an integer value.
  - Can solve a wider variety of synchronization problems.
- **Monitors**:
  - An abstract data type (ADT).
  - Provide a high-level form of process synchronization.
  - Use condition variables:
    - Allow processes to wait for specific conditions to become true.
    - Allow processes to signal one another when conditions are met.
- **Liveness Problems**:
  - Solutions to the critical-section problem may suffer from liveness issues.
  - Includes problems like deadlock and starvation.
- **Tool Evaluation**:
  - Various synchronization tools can be evaluated under different contention levels.
  - Some tools perform better than others under specific contention loads (e.g., uncontended, moderate, high contention).