Operating Systems
[6. Synchronization Tools]

Chung-Wei Lin
cwlin@csie.ntu.edu.tw
CSIE Department
National Taiwan University

From Operating System Concepts (10th Edition)

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

Outline

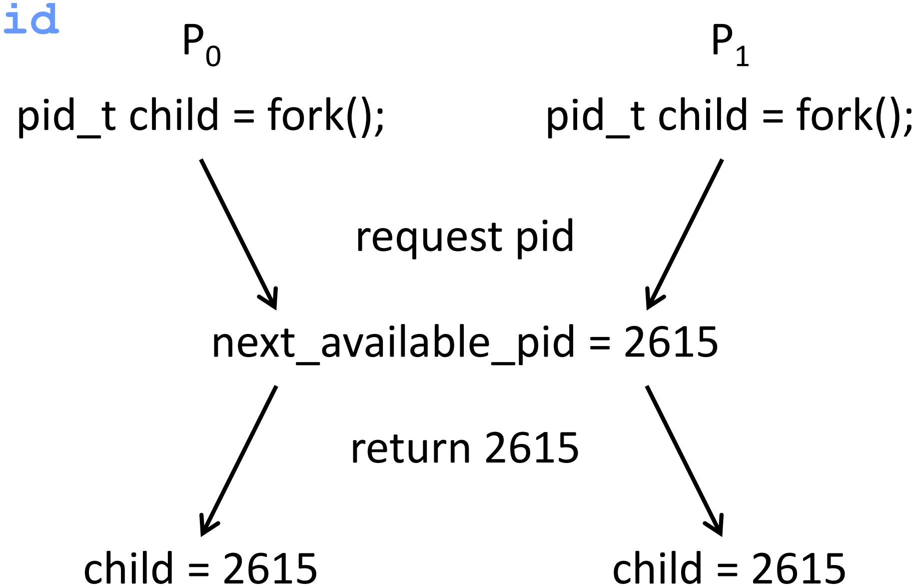
- <u>**Background**</u>
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Background

- Processes can execute concurrently
  - They may be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- The "bounded buffer" problem
  - Use of a counter that is updated concurrently by the producer and consumer leads to race condition
  - Process 1: `count++`
    - `reg1 = count; reg1 = reg1 + 1; count = reg1;`
  - Process 2: `count--`
    - `reg2 = count; reg2 = reg2 - 1; count = reg2;`
  - What if the lower-level statements are interleaved in some order?

Race Condition

- Processes access and manipulate the same data concurrently and the outcome depends on the accessing order
  - Processes P_0 and P_1 are creating child processes using **`fork()`**
  - Race condition on kernel variable **`next_available_pid`** which represents the next available process identifier **`pid`**
  - The same **`pid`** could be assigned to two different processes
    - Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable **`next_available_pid`**



Outline

- Background
- **The Critical-Section Problem**
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Critical-Section Problem

- Consider a system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has **critical section** segment of code
  - The process may be accessing and updating data that is shared with at least one other process
  - When one process is executing in its critical section, no other process is allowed to execute in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process
  - Must ask permission to enter critical section in **entry section**
  - May follow critical section with **exit section** and then **remainder section**

General Structure of Process

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

Solution Requirements (1/2)

- Requirement 1: Mutual Exclusion
  - If a process is executing in its critical section
  - Then no other processes can be executing in their critical sections
- Requirement 2: Progress
  - If
    - No process is executing in its critical section, and
    - Some processes wish to enter their critical sections
  - Then
    - Only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and
    - This selection cannot be postponed indefinitely

Solution Requirements (2/2)

- Requirement 3: Bounded Waiting
  - There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections
    - After a process has made a request to enter its critical section, and
    - Before that request is granted
- Assumption
  - We assume that each process is executing at a nonzero speed
  - However, we can make no assumption concerning the relative speed of the n processes

Interrupt-Based Solution

- Entry section
  - Disable interrupts
- Exit section
  - Enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve (never enter their critical section)?
  - What if there are two CPUs?

Software Solution (1/2)

- Two-process solution
  - Assume that the **`load`** and **`store`** machine-language instructions are atomic (cannot be interrupted)
  - The two processes share **`turn`** to indicate whose turn
- Algorithm for process P_i

```
while (true){
    turn = i;
    while (turn == j)
        ;
    /* critical section */
    turn = j;
    /* remainder section */
}
```

Software Solution (2/2)

```
while (true){

    while (turn == j)
        ;
    /* critical section */
    turn = j;
    /* remainder section */
}
```

- Mutual exclusion is preserved?
  - **`turn`** cannot be both 0 and 1 at the same time, but ...
- What about the progress requirement?
- What about the bounded-waiting requirement?
- What if we remove "**`turn = i`**"?

Outline

- Background
- The Critical-Section Problem
- **Peterson's Solution**
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Peterson's Solution (1/2)

- Two-process solution
  - Assume that the **`load`** and **`store`** machine-language instructions are atomic (cannot be interrupted)
  - The two processes share: **`int turn`** and **`boolean flag[2]`**
    - **`turn`** indicates whose turn it is to enter the critical section
    - **`flag`** array indicates if a process is ready to enter the critical section
    - **`flag[i] = true`** implies that process P_i is ready

Peterson's Solution (2/2)

- Algorithm for process P_i

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

For Comparison

```
while (true){
    turn = i;
    while (turn == j)
        ;
    /* critical section */
    turn = j;
    /* remainder section */
}
```

Peterson's Solution: Correctness

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

- Provable that the three requirements are met
  - Mutual exclusion is preserved
  - The progress requirement is satisfied
  - The bounded-waiting requirement is met

Modern Architecture

- Peterson's solution is not guaranteed to work on modern computer architectures
  - To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies
    - For a single-threaded application, the reordering is fine as the final values are consistent with what is expected
    - For a multi-threaded application with shared data, the reordering may render inconsistent or unexpected results

Modern Architecture: Example (1/2)

- Two threads share the data

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

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

- Thread 2 performs

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

- What is the expected output?

```
100
```

Modern Architecture: Example (2/2)

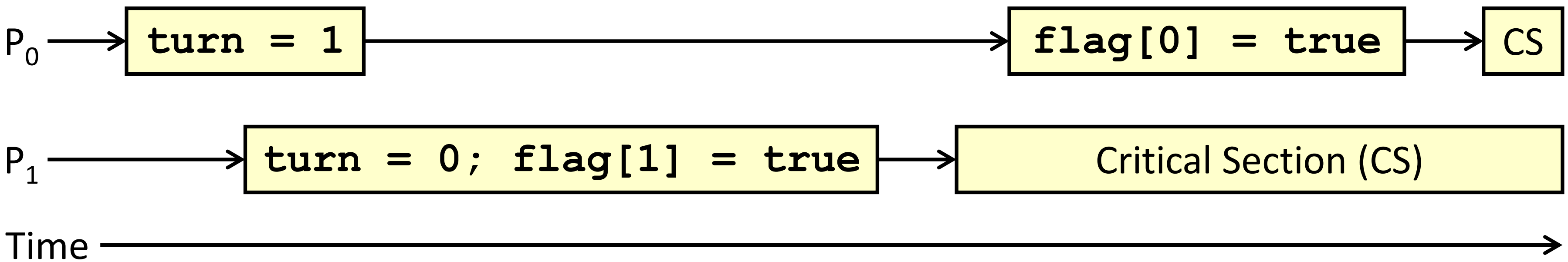
- However, since the variables **`flag`** and **`x`** are independent of each other, the following instructions for Thread 2 may be reordered

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

- If this occurs, the output may be 0

Peterson's Solution Revisited

- Instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- **Hardware Support for Synchronization**
  - **Memory Barriers**
  - Hardware Instructions
  - Atomic Variables
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Memory Barriers (1/2)

- **Memory model**
  - How a computer architecture determines what memory guarantees it will provide to an application program
- Memory models may be either
  - **Strongly ordered**
    - A memory modification on one processor is immediately visible to all other processors
  - **Weakly ordered**
    - Modifications to memory on one processor may not be immediately visible to other processors
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors

Memory Barriers (2/2)

- When a memory barrier instruction is performed
  - The system ensures that all loads and stores are completed before any subsequent load or store operations are performed
- Therefore, even with instructions reordering, the memory barrier ensures that
  - The store operations are completed in memory and visible to other processors before future load or store operations are performed
- Note that
  - Memory barriers are considered very low-level operations
    - They are typically only used by kernel developers when writing specialized code that ensures mutual exclusion

Memory Barriers: Example

- Add memory barriers to the previous example
- Thread 1 performs

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

- Thread 2 performs

```
x = 100;
memory_barrier();
flag = true
```

- We are guaranteed that
  - For Thread 1, the value of **`flag`** is loaded before the value of **`x`**
  - For Thread 2, the assignment to **`x`** occurs before the assignment to **`flag`**

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- **Hardware Support for Synchronization**
  - Memory Barriers
  - **Hardware Instructions**
  - Atomic Variables
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Hardware Instructions

- Many modern computer systems provide special hardware instructions that allow us either
  - To test and modify the content of a word, or
    - `test_and_set`
  - To swap the contents of two words atomically (uninterruptedly)
    - `compare_and_swap` (CAS)

`test_and_set` Instruction

- Definition

```
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Properties
  - Be executed atomically
  - Return the original value of the passed parameter
  - Set the value of the passed parameter to `true`

Solution Using `test_and_set`

- A shared boolean variable **`lock`**, initialized to **`false`**
- Solution

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

- Does it solve the critical-section problem?

`compare_and_swap` Instruction

- **Definition**

```
    int compare_and_swap(int *value, int expected,
int new_value) {
        int temp = *value;
        if (*value == expected)
            *value = new_value;
        return temp;
    }
```

- **Properties**
  - Be executed atomically
  - Return the original value of the passed parameter **`value`**
  - If **`*value == expected`**, set the value of the passed variable **`value`** to the passed parameter **`new_value`**
    - "Swap" takes place only under this condition

Solution Using `compare_and_swap`

- A shared integer **`lock`**, initialized to 0
- Solution

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-Waiting with CAS

```
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 */
}
```

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- **Hardware Support for Synchronization**
  - Memory Barriers
  - Hardware Instructions
  - **Atomic Variables**
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Atomic Variables

- Typically, instructions such as **`compare-and-swap`** are used as building blocks for other synchronization tools
  - Not used directly to provide mutual exclusion
- One such tool is an **atomic variable**
  - Provide atomic (uninterruptible) operations on basic data types such as integers and booleans

Atomic Variables: Example

- Example
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be an operation on **`sequence`**
  - The command **`increment(&sequence)`** ensures **`sequence`** is incremented without interruption
- **`increment()`** function

```
void increment(atomic_int *v) {
    int temp;
    do {
        temp = *v;
    }
    while(temp!=(compare_and_swap(v,temp,temp+1));
}
```

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- **<u>Mutex Locks</u>**
- Semaphores
- Monitors
- Liveness
- Evaluation

Mutex (Mutual Exclusion) Locks (1/2)

- Previous solutions are complicated and generally inaccessible to application programmers
  - OS designers build software tools to solve the critical section problem
- A **mutex lock** protects critical sections and thus prevent race conditions
  - First `acquire()` a lock and then `release()` the lock
    - Calls to `acquire()` and `release()` must be **atomic**; usually implemented via hardware atomic instructions such as `compare_and_swap`
  - A boolean variable `available` indicating if the lock is available or not
- This solution requires **busy waiting**
  - While a process is in its critical section, another process trying to enter its critical section must loop continuously in the call to `acquire()`
    - This lock therefore called a **spinlock**

Mutex Locks (2/2)

```
while (true){
   acquire lock
   /* critical section */
   release lock
   /* remainder section */
}
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
release() {
   available = true;
}
```

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- **Semaphores**
  - Semaphore Usage
  - Semaphore Implementation
- Monitors
- Liveness
- Evaluation

Semaphore

- A more robust tool that can
  - Behave similarly to a mutex lock
  - Provide more sophisticated ways for processes to synchronize activities
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two atomic operations
- **`wait()`** operation (originally called **`P()`**)

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

- **`signal()`** operation (originally called **`V()`**)

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

Outline

Semaphore Usage

- **Counting semaphore**
  - Range over an unrestricted domain
- **Binary semaphore**
  - Range only between 0 and 1
    - Same as a mutex lock
  - Can implement a counting semaphore S as a binary semaphore
- We can solve various synchronization problems with semaphores

Semaphore Usage: Examples

- Create a semaphore "mutex" initialized to 1

```
wait(mutex);
/* critical section */
signal(mutex);
```

- Consider P_1 and P_2 with two statements S_1 and S_2, respectively, and require that S_1 happens before S_2
  - Create a semaphore "synch" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- **Semaphores**
  - Semaphore Usage
  - **Semaphore Implementation**
- Monitors
- Liveness
- Evaluation

Semaphore Implementation

- No two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
  - Thus, the implementation becomes the critical section problem where the **`wait()`** and **`signal()`** code are placed in the critical section
- The definitions of the **`wait()`** and **`signal()`** semaphore operations have the busy-waiting problem
  - Applications may spend lots of time in critical sections

Semaphore without Busy Waiting (1/4)

- Modify **`wait()`** and **`signal()`** operations
  - When a process executes the **`wait()`** operation and finds that the semaphore value is not positive, it must wait
    - The process can suspend itself, rather than engaging in busy waiting
    - The suspend operation places a process into a waiting queue associated with the semaphore
    - The state of the process is switched to the waiting state
  - Control is transferred to the CPU scheduler (selecting another process)
  - A process that is suspended, waiting on a semaphore S, should be restarted when some other process executes a **`signal()`** operation
    - The process is restarted by a **`wakeup()`** operation, which changes the process from the waiting state to the ready state
    - The process is then placed in the ready queue

Semaphore without Busy Waiting (2/4)

- Each semaphore has an integer **`value`** and a list of processes **`list`**

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

Semaphore without Busy Waiting (3/4)

- When a process must wait on a semaphore, it is added to the list of processes
- **`sleep()`** suspends the process that invokes it

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

Semaphore without Busy Waiting (4/4)

- **`signal()`** removes one process from the list of waiting processes and awakens that process
- **`wakeup(P)`** resumes the execution of a suspended process P

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

Semaphore Discussion

- Semaphore operations must be executed atomically
- We can use a FIFO queue to ensure bounded waiting
  - However, in general, the list can use any queuing strategy
- We have not completely eliminated busy waiting with this definition of the **`wait()`** and **`signal()`** operations
  - Rather, we have moved busy waiting from the entry section to the critical sections of application programs
  - We have limited busy waiting to the critical sections of the **`wait()`** and **`signal()`** operations
    - These sections are usually short

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- **Monitors**
  - Monitor Usage
  - Implementing a Monitor Using Semaphores
  - Resuming Processes within a Monitor
- Liveness
- Evaluation

Monitors

- Unfortunately, timing errors can still occur when either mutex locks or semaphores are used
- Examples
  - `signal(mutex);` … critical section … `wait(mutex);`
    - Several processes may be executing in their critical sections simultaneously
  - `wait(mutex);` … critical section … `wait(mutex);`
    - The process will permanently block on the second call to **`wait()`**
  - Omit the **`wait(mutex)`**, or the **`signal(mutex)`**, or both
    - Either mutual exclusion is violated or the process will permanently block
  - They can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem
    - Using them incorrectly can result in timing errors that are difficult to detect

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- **Monitors**
  - **Monitor Usage**
  - Implementing a Monitor Using Semaphores
  - Resuming Processes within a Monitor
- Liveness
- Evaluation

Monitor Usage

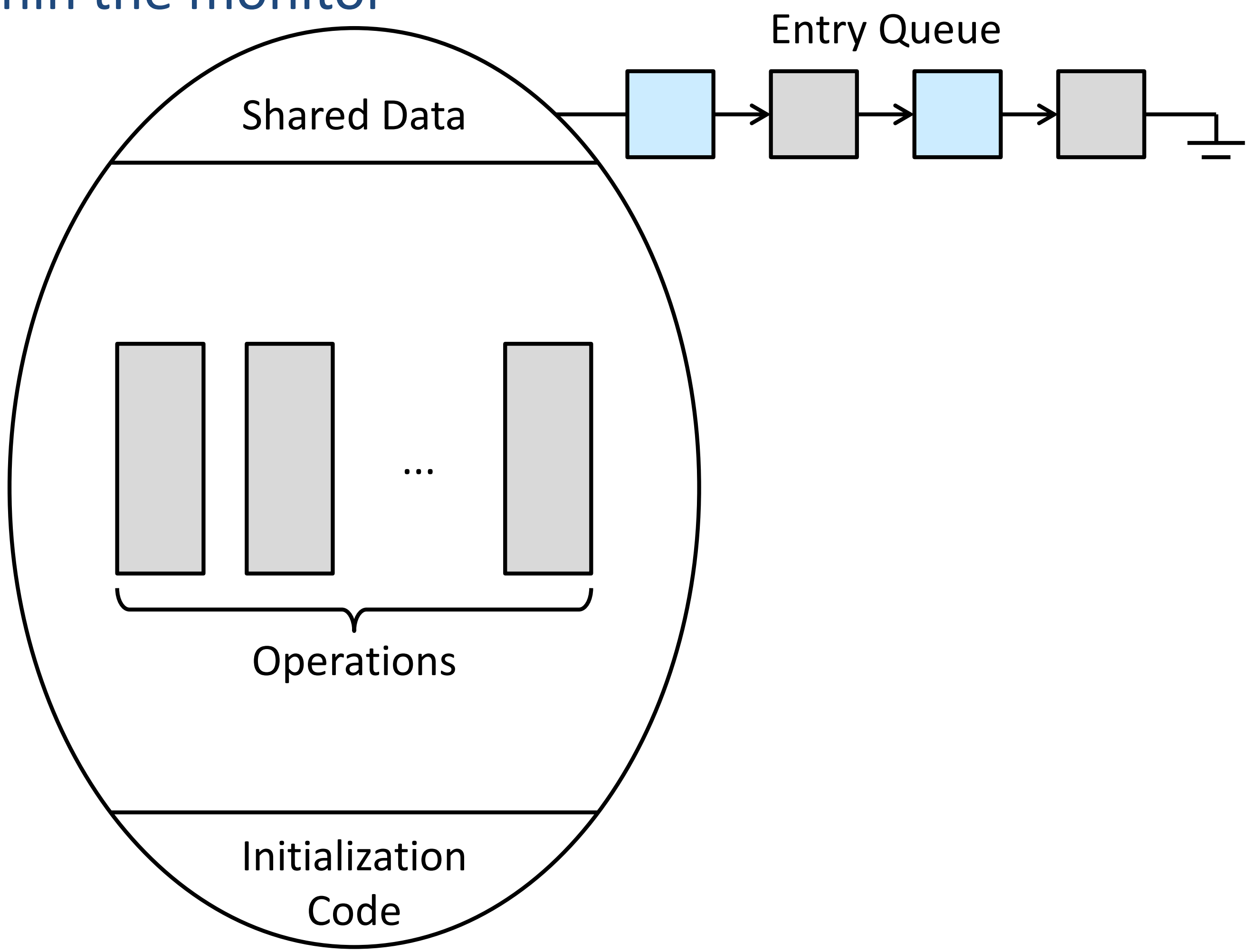
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- **<u>Abstract data type (ADT)</u>**
  - Encapsulate data with a set of functions to operate on that data that are independent of any specific implementation of the ADT
- Monitor type
  - An ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor
    - A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters
    - The local variables of a monitor can be accessed by only the local functions

Pseudocode Syntax of a Monitor

```
monitor monitor-name {
   /* shared variable declarations */
   procedure P1 (...) {...}
   procedure P2 (...) {...}
   procedure Pn (...) {...}
   initialization_code (...) {...}
}
```

Schematic View of a Monitor

- The monitor construct ensures that only one process at a time is active within the monitor

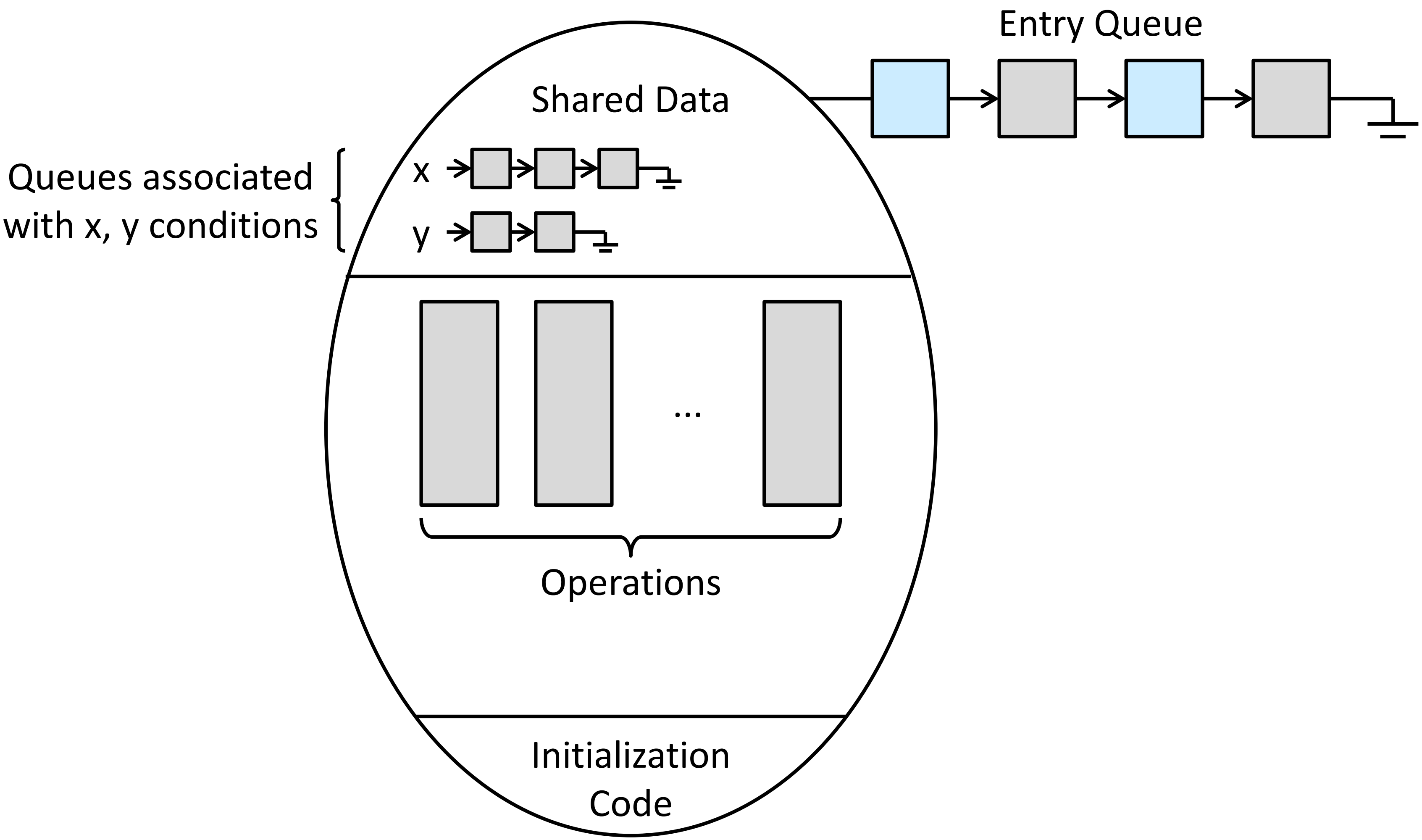


Condition Variables (1/4)

- **`condition x, y;`**
- The only operations that can be invoked on a condition variable are **`wait()`** and **`signal()`**
  - **`x.wait()`**
    - The process invoking this operation is suspended until another process invokes **`x.signal()`**
  - **`x.signal()`**
    - Resume exactly one suspended process (if any)
    - If no process is suspended, then it has no effect

Condition Variables (2/4)

- Monitor with condition variables



Condition Variables (3/4)

- Suppose that when the **`x.signal()`** operation is invoked by a process P, there exists a suspended process Q associated with condition **`x`**
  - Conceptually both processes can continue with their execution
  - However, both P and Q should not be active simultaneously within the monitor
- Two possibilities
  - Signal and wait
    - P either waits until Q leaves the monitor or waits for another condition
  - Signal and continue
    - Q either waits until P leaves the monitor or waits for another condition

Condition Variables (4/4)

- Reasonable arguments in favor of adopting either option
  - Since P was already executing in the monitor, the signal-and-continue method seems more reasonable
  - If we allow P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold
- A compromise between the two options
  - When thread P executes the **`signal`** operation, it immediately leaves the monitor
  - Hence, Q is immediately resumed

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- **Monitors**
  - Monitor Usage
  - **Implementing a Monitor Using Semaphores**
  - Resuming Processes within a Monitor
- Liveness
- Evaluation

Monitor Using Semaphores (1/3)

- A binary semaphore **`mutex`** is provided to ensure mutual exclusion

```
semaphore mutex;  // (initially = 1)
```

- A process must
  - Execute **`wait(mutex)`** before entering the monitor
  - Execute **`signal(mutex)`** after leaving the monitor

Monitor Using Semaphores (2/3)

- Use the signal-and-wait scheme
  - An additional binary semaphore **`next`**
    - The signaling processes can use **`next`** to suspend themselves
  - An integer variable **`next_count`** counts the number of processes suspended on **`next`**

```
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next_count = 0;
```

- Each external function F is replaced by

```
wait(mutex);
... body of F ...
if (next_count > 0)   signal(next);
else                  signal(mutex);
```

Monitor Using Semaphores (3/3)

- For each condition variable **x**, we have

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- **x.wait()** operation

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

- **x.signal()** operation

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

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- **Monitors**
  - Monitor Usage
  - Implementing a Monitor Using Semaphores
  - **Resuming Processes within a Monitor**
- Liveness
- Evaluation

Resuming Processes within a Monitor

- Which process should be resumed?
  - If several processes are suspended on condition **`x`**, and
  - An **`x.signal()`** operation is executed by some process
- One simple solution: first-come, first-served (FCFS) ordering
  - Not adequate in many circumstances
- The **conditional-wait** construct **`x.wait(c)`** can be used
  - **`c`** is an integer expression (called a priority number) evaluated when the **`wait()`** operation is executed
  - The value of **`c`** is then stored with the name of the process that is suspended
  - When **`x.signal()`** is executed, the process with the smallest priority number is resumed next

Single Resource Allocation (1/3)

- Each process, when requesting an allocation of a resource, specifies the maximum time **`t`** it plans to use the resource
- The monitor allocates the resource to the process that has the shortest time-allocation
- **`R`** is an instance of type **`ResourceAllocator`**

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

Single Resource Allocation (2/3)

```
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;
    }
}
```

Single Resource Allocation (3/3)

- **Problems**
  - A process might access a resource without first gaining access permission to the resource
  - A process might never release a resource once it has been granted access to the resource
  - A process might attempt to release a resource that it never requested
  - A process might request the same resource twice (without first releasing the resource)
- **Incorrect use of monitor operations**
  - `release() ... acquire()`
  - `acquire() ... acquire()`
  - Omit `acquire()` and/or `release()`

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- **Liveness**
  - Deadlock
  - Priority Inversion
- Evaluation

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore
  - Waiting indefinitely violates the progress and bounded-waiting requirements
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress
  - Indefinite waiting is an example of a liveness failure
- **Starvation** (indefinite blocking)
  - A process may never be removed from the semaphore queue in which it is suspended

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- **Liveness**
  - **Deadlock**
  - Priority Inversion
- Evaluation

Deadlock

- Every process in the set is waiting for an event that can be caused only by another process in the set
- Example
  - Let S and Q be two semaphores initialized to 1
  - P_0: `wait(S); wait(Q); ... signal(S); signal(Q);`
  - P_1: `wait(Q); wait(S); ... signal(Q); signal(S);`
  - P_0 executes `wait(S)` and P_1 executes `wait(Q)`
  - When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`
  - However, P_1 is waiting until P_0 executes `signal(S)`
  - Since these `signal()` operations will never be executed, P_0 and P_1 are deadlocked
- Chapter 8

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- **Liveness**
  - Deadlock
  - **Priority Inversion**
- Evaluation

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Example
    - There are three processes L, M, and H with priorities L < M < H
    - H requires a semaphore S which is currently being accessed by L
    - H waits for L to finish using resource S
    - M becomes runnable and preempts L
- **Priority-inheritance protocol**
  - All processes accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- **Evaluation**

Evaluation

- Trying to identify when to use which tool can be a daunting challenge
  - Low overhead
  - Ability to scale
  - Ease to develop and test
  - Simplicity and ease of use
  - Optimistic approach vs. pessimistic approach
    - Uncontended
    - Moderate contention
    - High contention

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

Q&A