




UNIVERSITY OF REGINA

CS330-001 INTRODUCTION TO OPERATING SYSTEMS

ANDRÉ E. **DOS SANTOS**

dossantos@cs.uregina.ca

andreeeds.github.io

An aerial photograph of a city, likely Regina, Saskatchewan, showing a large park with a lake in the foreground and a dense urban area with various buildings in the background. The image is in black and white, with a red watermark 'andreds.github.io' diagonally across the center.

CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

SYNCHRONIZATION TOOLS

ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreds.github.io

BACKGROUND

- Processes can execute concurrently
 - 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
- Illustration of the problem:
 - Suppose that we wanted to provide a solution to the **consumer-producer problem** that fills all the buffers
 - We can do so by having an integer counter that keeps track of the number of full buffers
 - Initially, counter is set to 0
 - It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer

BACKGROUND

Producer

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
    /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

○ S ₀ : producer execute	register1 = counter	{register1 = 5}
○ S ₁ : producer execute	register1 = register1 + 1	{register1 = 6}
○ S ₂ : consumer execute	register2 = counter	{register2 = 5}
○ S ₃ : consumer execute	register2 = register2 - 1	{register2 = 4}
○ S ₄ : producer execute	counter = register1	{counter = 6}
○ S ₅ : consumer execute	counter = register2	{counter = 4}

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with **count = 5** initially:

○ S ₀ : producer execute	register1 = counter	{register1 = 5}
○ S ₁ : producer execute	register1 = register1 + 1	{register1 = 6}
○ S ₂ : consumer execute	register2 = counter	{register2 = 5}
○ S ₃ : consumer execute	register2 = register2 - 1	{register2 = 4}
○ S ₄ : producer execute	counter = register1	{counter = 6}
○ S ₅ : consumer execute	counter = register2	{counter = 4}

It has arrived at the incorrect state
`count == 4`, indicating that four buffers are
full, when, in fact, five buffers are full.

CRITICAL SECTION PROBLEM

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be 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**, then **remainder section**

Critical Section

- General structure of process P_i

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


Algorithm for Process P_i

```
do {  
    while (turn == j);  
    critical section  
    turn = j;  
    remainder section  
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_1 is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist 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
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
 - **Preemptive** – allows preemption of process when running in kernel mode
 - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - Essentially free of race conditions in kernel mode

PETERSON'S SOLUTION

- Good algorithmic description of solving the problem
- **Two process solution**
- Assume that the **load** and **store** machine-language instructions are **atomic (cannot be interrupted)**
- The two processes share two variables:
 - **int turn;**
 - **Boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the **critical section**
- The **flag** array is used to indicate if a process is ready to enter the critical section
 - **flag[i] = true** implies that process P_i is ready

Algorithm for Process P_i

The two processes are numbered P_0 and P_1

when presenting P_i ,
 P_j to denote the other process:
 $j = 1 - i$

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

PETERSON'S SOLUTION

- Provable that the three CS requirement are met:
 1. **Mutual exclusion** is preserved
 - P_i enters CS only if:
 - either `flag[j] = false` or `turn = i`
 2. **Progress** requirement is satisfied
 3. **Bounded-waiting** requirement is met

See p. 263 for proof

Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies

SYNCHRONIZATION HARDWARE

- Many systems provide hardware support for implementing the critical section code
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - **Atomic** = non-interruptible
 - Either **test memory word and set value** (**test_and_set**)
 - Or **swap contents** of two memory words (**compare_and_swap**)

Solution to Critical-section Problem Using Locks

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

test_and_set Instruction

- Definition:

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

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**

Solution using `test_and_set`

- Shared Boolean variable `lock`, initialized to `false`

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

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;  
}
```

- Executed atomically
- Returns the original value of passed parameter **value**
- Set the variable **value** the value of the passed parameter **new_value**
 - but only if ***value == expected**
 - That is, the swap takes place only under this condition

Solution using `compare_and_swap`

- Shared integer variable `lock` initialized to 0

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

0 means lock is free

MUTEX LOCKS

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

A practice that allows a thread or process to use CPU time continuously while waiting for something

A locking mechanism that continuously uses the CPU while waiting for access to the lock

acquire() and release()

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

acquire() and release()

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - Originally called **P()** and **V()**

Edsger Dijkstra

wait and signal

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

SEMAPHORE USAGE

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between **0** and **1**
 - **Same as a mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
 - Create a semaphore **synch** initialized to **0**
 - **P1:**

 S_1 ;
 signal(synch) ;
 - **P2:**

 wait(synch) ;
 S_2 ;
- Can implement a counting semaphore S as a binary semaphore

SEMAPHORE IMPLEMENTATION

- Must guarantee that **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
 - Could now have **busy waiting in critical section implementation**
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

SEMAPHORE IMPLEMENTATION WITH NO BUSY WAITING

- With each semaphore there is **an associated waiting queue**
- Each entry in a waiting queue has two data items:
 - **value** (of type integer)
 - pointer to **next** record in the list
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- `typedef struct {`
 - `int value;`
 - `struct process *list;`
 - `} semaphore;`

wait and signal with no Busy waiting

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

we have only moved busy waiting from the entry section to the critical sections of application programs

LIVENESS

- **Liveness** refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle
 - A process waiting indefinitely under the circumstances just described is an example of a “liveness failure”

DEADLOCK

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to **1**

P0

```
wait(S);
```

```
wait(Q);
```

```
...
```

```
signal(S);
```

```
signal(Q);
```

P1

```
wait(Q);
```

```
wait(S);
```

```
...
```

```
signal(Q)
```

```
signal(S);
```

Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked

PRIORITY INVERSION

- **Starvation – indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**

A protocol for solving priority inversion in which all processes that are accessing resources needed by a higher-priority process inherit that higher priority until they are finished with the resources in question.

MONITORS

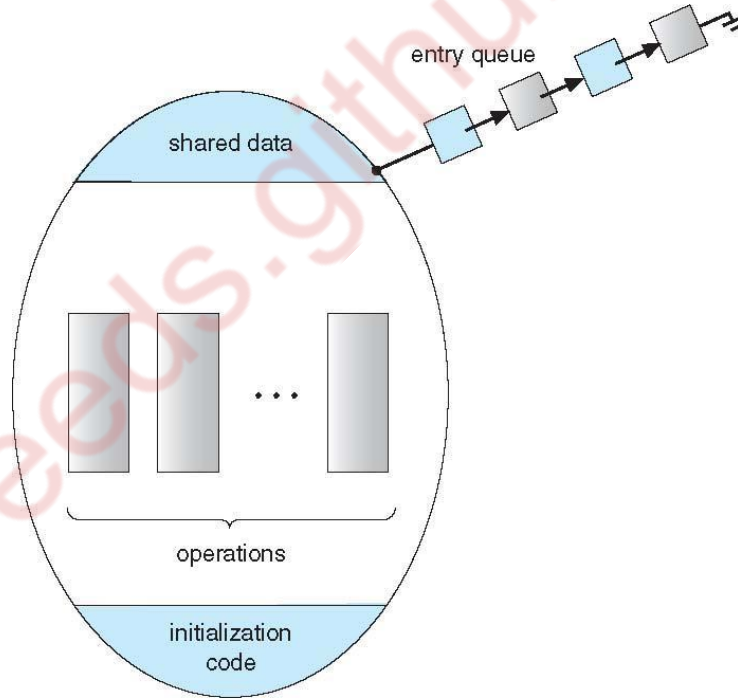
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- An **ADT** encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT
- **Only one process may be active within the monitor at a time**
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure Pn (...) { ... }

    Initialization code (...) { ... }
}
```

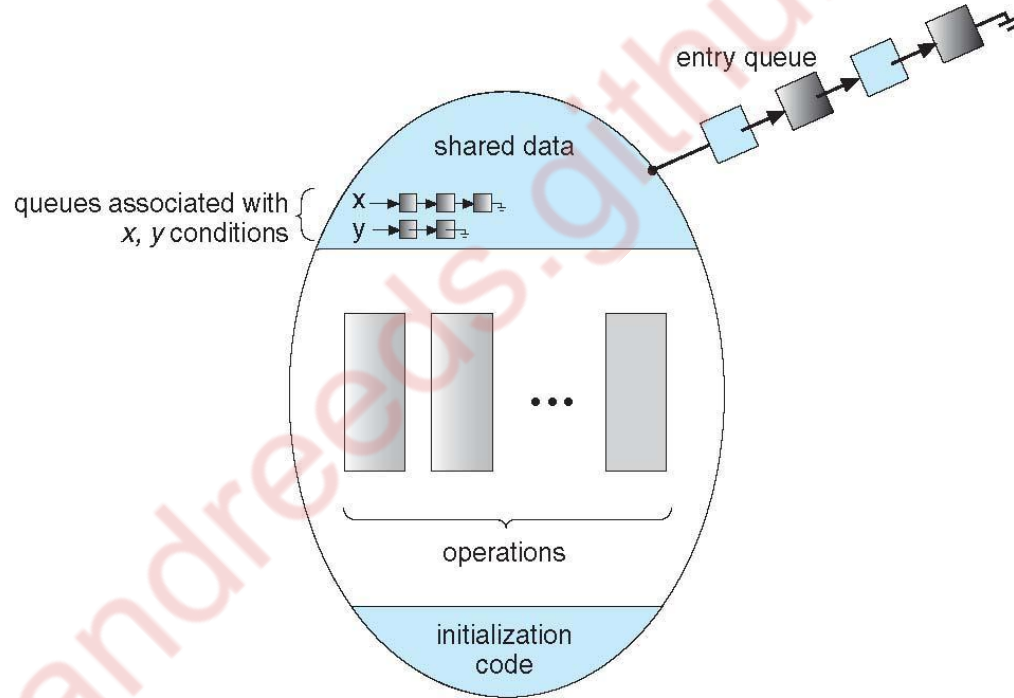
SCHEMATIC VIEW OF A MONITOR



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
 - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
 - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
 - If no `x.wait()` on the variable, then it has no effect on the variable

MONITOR WITH CONDITION VARIABLES



Condition Variables Choices

- If process **P** invokes `x.signal()`, and process **Q** is suspended in `x.wait()`, what should happen next?
 - Both **Q** and **P** **cannot** execute in parallel
 - If **Q** is resumed, then **P** must wait
- Options include
 - **Signal and wait** – **P** waits until **Q** either leaves the monitor or it waits for another condition
 - **Signal and continue** – **Q** waits until **P** either leaves the monitor or it waits for another condition

MONITOR IMPLEMENTATION USING SEMAPHORES

- Variables

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

- Each procedure **F** will be replaced by

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

- Mutual exclusion within a monitor is ensured

MONITOR IMPLEMENTATION – CONDITION VARIABLES

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

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

- The operation **x.wait** can be implemented as:

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

MONITOR IMPLEMENTATION – CONDITION VARIABLES

- The operation **x.signal** can be implemented as:

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


RESUMING PROCESSES WITHIN A MONITOR

- If several processes queued on condition **x**, and **x.signal()** executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form **x.wait(c)**
 - Where **c** is **priority number**
 - Process with lowest number (highest priority) is scheduled next



Differences between semaphore operations and monitor operations

Semaphore	Monitor
<ul style="list-style-type: none">1. wait may or may not block2. signal always has an effect3. signal unblocks an arbitrarily chosen blocked process (weak semaphore) or the process at the head of the queue (strong semaphore)4. A process unblocked by signal can resume execution immediately	<ul style="list-style-type: none">1. waitC always blocks2. signalC has no effect if the queue is empty3. signalC unblocks the process at the head of the queue4. A process unblocked by signalC must wait for the signaling process to leave the monitor

SIMULATING SEMAPHORES USING A MONITOR

```
monitor Sem
  integer s <- k
  condition notZero
  bool lock <- false

  operation wait
    lock <- true
    if s = 0
      waitC (notZero)
    s <- s - 1
    lock <- false

  operation signal
    lock <- true
    s <- s + 1
    if emptyC (notZero)
      lock <- false
    else
      signalC (notZero)
```

p

```
loop forever
  non-critical section
p1: Sem.wait
  critical section
p2: Sem.signal
```

q

```
loop forever
  non-critical section
q1: Sem.wait
  critical section
q2: Sem.signal
```

A SOLUTION TO THE PRODUCER-CONSUMER PROBLEM USING MONITORS

```
monitor PC
  bufferType buffer <- empty
  condition notEmpty
  condition notFull
  bool lock <- false

  operation Append (dataType v)
    lock <- true
    if full (buffer)
      waitC (notFull)
    append (v, buffer)
    if emptyC (notEmpty)
      lock <- false
    else
      signalC (notEmpty)

  operation Take ()
    dataType w
    lock <- true
    if empty (buffer)
      waitC (notEmpty)
    w <- head (buffer)
    if emptyC (notFull)
      lock <- false
    else
      signalC (notFull)
    return w
```

p

```
dataType d
loop forever
  d <- produce
p1:  PC.Append (d)
```


q

```
dataType d
loop forever
q1:  d <- PC.Take
     consume (d)
```



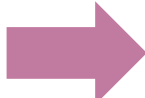
REVIEW QUESTIONS

SYNCHRONIZATION TOOLS




A race condition ____.

- A) results when several threads try to access the same data concurrently
- B) results when several threads try to access and modify the same data concurrently
- C) will result only if the outcome of execution does not depend on the order in which instructions are executed
- D) None of the above




A(n) _____ refers to where a process is accessing/updating shared data.

- A) critical section
- B) entry section
- C) mutex
- D) test-and-set




Instructions from different processes can be interleaved when interrupts are allowed. True or False?



A solution to the critical section problem does not have to satisfy which of the following requirements?

- A) mutual exclusion
- B) progress
- C) atomicity
- D) bounded waiting



A nonpreemptive kernel is safe from race conditions on kernel data structures. True or False?

REVIEW QUESTIONS

SYNCHRONIZATION TOOLS

➡ In Peterson's solution, the ____ variable indicates if a process is ready to enter its critical section.

- A) turn
- B) lock
- C) flag[i]
- D) turn[i]

➡ ____ is/are not a technique for managing critical sections in operating systems.

- A) Peterson's solution
- B) Preemptive kernel
- C) Nonpreemptive kernel
- D) Semaphores

➡ ____ is not a technique for handling critical sections in operating systems.

- A) Nonpreemptive kernels
- B) Preemptive kernels
- C) Spinlocks
- D) Peterson's solution

REVIEW QUESTIONS

SYNCHRONIZATION TOOLS

➡ *Race conditions are prevented by requiring that critical regions be protected by locks. True or False?*

➡ *Both the **test_and_set()** instruction and **compare_and_swap()** instruction are executed atomically. True or False?*

➡ *A mutex lock ____.*

- A) is exactly like a counting semaphore
- B) is essentially a boolean variable
- C) is not guaranteed to be atomic
- D) can be used to eliminate busy waiting

➡ *What is the correct order of operations for protecting a critical section using mutex locks?*


- A) **release()** followed by **acquire()**
- B) **acquire()** followed by **release()**
- C) **wait()** followed by **signal()**
- D) **signal()** followed by **wait()**

➡ *Busy waiting refers to the phenomenon that while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire the mutex lock. Yes or No?*




REVIEW QUESTIONS

SYNCHRONIZATION TOOLS



A counting semaphore ____.

- A) is essentially an integer variable
- B) is accessed through only one standard operation
- C) can be modified simultaneously by multiple threads
- D) cannot be used to control access to a thread's critical sections




_____ can be used to prevent busy waiting when implementing a semaphore.

- A) Spinlocks
- B) Waiting queues
- C) Mutex lock
- D) Allowing the wait() operation to succeed




Mutex locks and binary semaphores are essentially the same thing.
True or False?



Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

A deadlock-free solution eliminates the possibility of starvation.
True or False?




_____ occurs when a higher-priority process needs to access a data structure that is currently being accessed by a lower-priority process.

- A) Priority inversion
- B) Deadlock
- C) A race condition
- D) A critical section




REVIEW QUESTIONS

SYNCHRONIZATION TOOLS



A ___ type presents a set of programmer-defined operations that are provided mutual exclusion within it.

- A) transaction
- B) signal
- C) binary
- D) monitor



The local variables of a monitor can be accessed by only the local procedures. True or False?