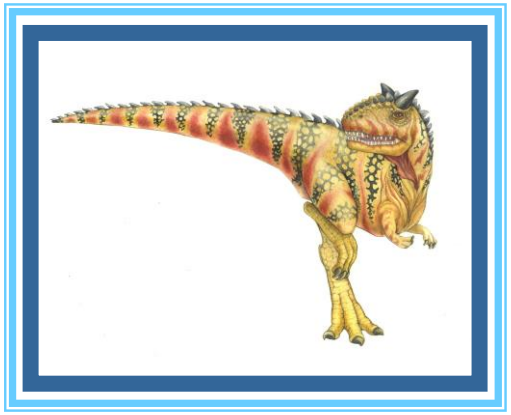
Chapter 6: Synchronization Tools

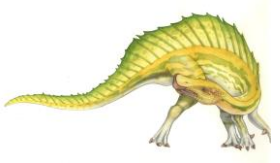


Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors

Objectives

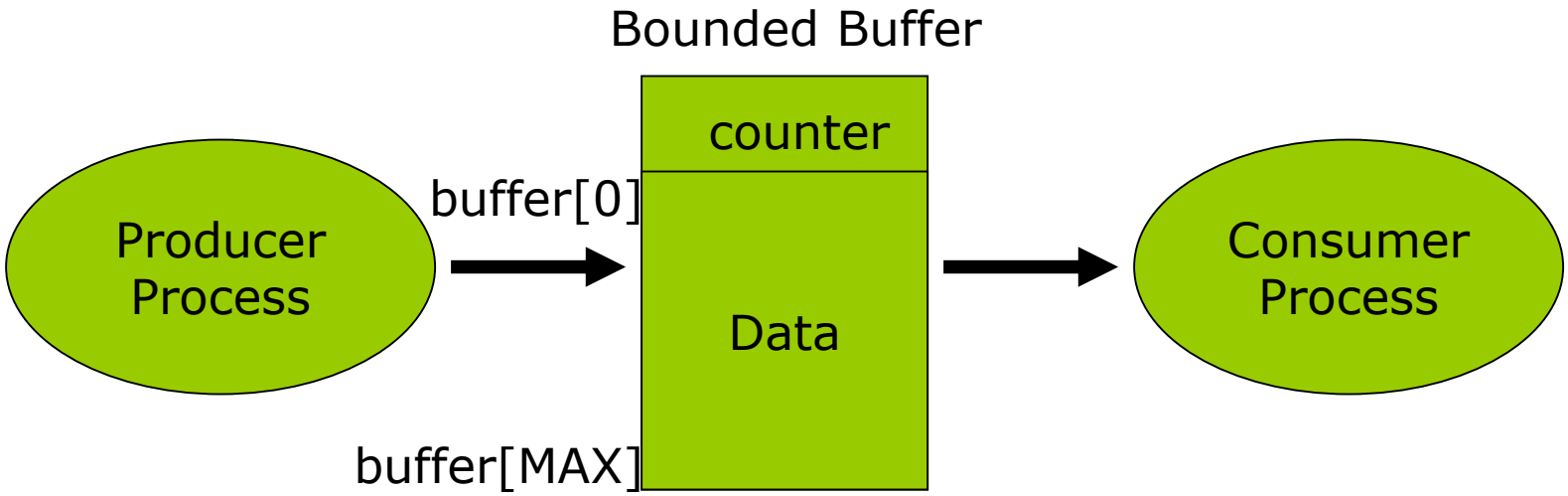
- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems



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.

Synchronization




```
while(1) {
    while(counter==MAX);
    buffer[in] = nextdata;
    in = (in+1) % MAX;
    counter++;
}
```

```
while(1) {
    while(counter==0);
    getdata = buffer[out];
    out = (out+1) % MAX;
    counter--;
}
```

counter updates can conflict!

Producer writes new data into buffer and increments counter

Consumer reads new data from buffer and decrements counter

Synchronization

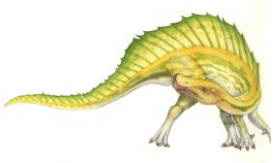
counter++; can translate into several machine language instructions, e.g.

```
reg1 = counter;
reg1 = reg1 + 1;
counter = reg1;
```

counter--; can translate into several machine language instructions, e.g.

```
reg2 = counter;
reg2 = reg2 - 1;
counter = reg2;
```

If these low-level instructions are interleaved, e.g. the Producer process is context-switched out, and the Consumer process is context-switched in, and vice versa, then the results of counter's value can be unpredictable



Race Condition

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

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

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

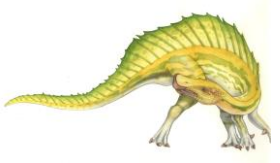
```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - the “loser” of the race is the process that updates last and will determine the final value of the variable



```python
import threading

# Shared bank account balance
balance = 500

# Function to withdraw money
# from the bank account (without a lock)
def withdraw(amount):
    global balance
    if balance >= amount:
        initial_balance = balance
        balance -= amount
        print(f"Withdrawn {amount}. Initial Balance:\
              {initial_balance}, New Balance: {balance}")
    else:
        print(f"Insufficient funds to withdraw {amount}.\
                Current Balance: {balance}")

# Create two threads simulating two withdrawals
thread_1 = threading.Thread(target=withdraw, args=(300,))
thread_2 = threading.Thread(target=withdraw, args=(100,))

# Start both threads
thread_1.start()
thread_2.start()

# Wait for both threads to finish
thread_1.join()
thread_2.join()

print(f"Final account balance: {balance}")
```

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- 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);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

   A **mutex** is a locking mechanism ensures only one thread can access a resource at a time.

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

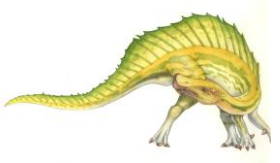
   - Assume that each process executes at a nonzero speed
     - Every process keeps making progress.
     - Synchronization algorithms must account for all processes, ensuring none are ignored.
   - No assumption concerning **relative speed** of the ***n*** processes
     - Processes can execute at any speed relative to each other.
     - Synchronization mechanisms must be independent of timing or speed assumptions

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

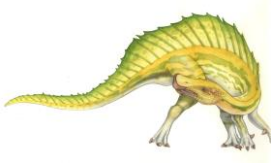
- **Preemptive Kernel:** Allows a process running in kernel mode to be preempted (interrupted) to switch to another process.
  - Race conditions can occur in kernel mode due to concurrent access.
  - Requires synchronization mechanisms to protect shared resources.

- **Non-preemptive** – Kernel: Once a process enters kernel mode, it runs to completion (exits kernel mode), blocks, or voluntarily yields the CPU.
  - Essentially free of race conditions in kernel mode
  - processes execute kernel code without interruption.
  - Simplifies kernel code as less synchronization is needed.



Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, 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 $\mathbf{P_i}$ is ready!



Peterson's Solution (Cont.)

- 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

Peterson's Solution – Sequential (Mutual Exclusion)

process Pi

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

process Pj

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

Announce finished

Flag

0	1
F	F

turn
0

i	j
0	1

```
falg[0]: True,  turn =1.

Process 0 is entering the critical section.
Process 0: Shared data = 1
Process 0 is leaving the critical section.

falg[1]: True,  turn =0.
Process 0 is in the remainder section.


Process 1 is entering the critical section.
Process 1: Shared data = 2
Process 1 is leaving the critical section.

Process 1 is in the remainder section.

Both processes have finished execution.
```

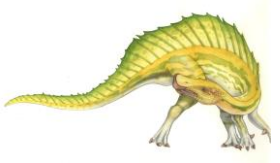
Synchronization Hardware

- 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
      - - operations that are indivisible and uninterruptible.
  - Either test memory word and set value (Test-and-Set)
  - Or swap contents of two memory words (Swap)

Solution to Critical-section Problem Using Locks

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

- **acquire & release lock** are indeed implemented in hardware.
- They are provided as atomic machine instructions by modern processors.
- These hardware-implemented instructions are crucial for building efficient and reliable synchronization mechanisms in operating systems and concurrent applications.



test_and_set Instruction

Definition:

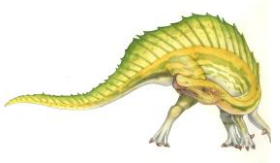
```
boolean test_and_set (boolean *target)
   {
        boolean rv = *target;
        *target = TRUE;
        return rv:
   }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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



Test and Set – Sequential (Mutual Exclusion)

process Pi

```
do{
   while(test_and_set(&lock));
     /* critical section*/
   lock = false;
     /* remainder section*/
} while (true);
```

process Pj

```
do{
   while(test_and_set(&lock));
     /* critical section*/
   lock = false;
     /* remainder section*/
} while (true);
```

Busy-Waiting: thread consumes cycles while waiting

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

lock
F

target	rv
F	F

target	rv
F	F

```
Process 0 is entering the critical section.

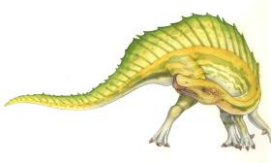
Process 0: Shared data = 1
Process 0 is leaving the critical section.
Process 0 is in the remainder section.
Process 1 is entering the critical section.


Process 1: Shared data = 2
Process 1 is leaving the critical section.
Process 1 is in the remainder section.

Both processes have finished execution.
```

Test & Set

- **Advantages**
  - Simplicity: Easy to implement.
  - Hardware Support: Supported by many processor architectures.
  - Atomicity: Guarantees atomic operations without additional synchronization.
- **Disadvantages**
  - Busy Waiting: Can lead to performance issues due to CPU cycle consumption.
  - Starvation: No guarantee of fairness; some processes may wait indefinitely.
  - Not Scalable: Inefficient in systems with many processors or long critical sections.



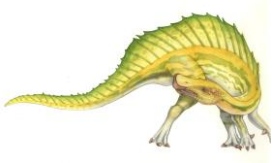
compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

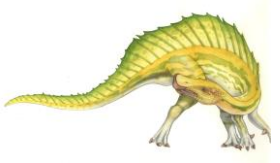
1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.



Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

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



Swap Instruction – Concurrent (Mutual Exclusion)

process Pi

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

process Pj

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

Swap Instruction – Concurrent (Mutual Exclusion)

process Pi

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

process Pj

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp:
```

lock	key	key
T	F	T

Swap Instruction – Concurrent (Mutual Exclusion)

process Pi

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

process Pj

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

lock	key	key
T	F	T

Swap Instruction – Concurrent (Bounded Wait)

process Pi

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

process Pj

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
    <critical section>
    lock = FALSE;
    <remainder section>
} while (TRUE);
```

Bounded Wait holds

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

lock	key	key
T	F	F

Swap

- **Advantages**
  - Atomic Exchange: Provides a simple way to implement locks.
  - Hardware Support: Available in some processor architectures.
- **Disadvantages**
  - Busy Waiting: Similar to Test-and-Set, can cause high CPU usage.
  - Limited Availability: Not all architectures support an atomic Swap instruction.
  - Fairness Issues: Does not prevent starvation or guarantee fairness.

Bounded-waiting Mutual Exclusion with test_and_set

n-processes Solution – Mutual Exclusion

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
P1 P3 P4 key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

n = 6

j

lock
F

P	W	K
0	F	F
1	T	T
2	F	F
3	T	T
4	T	T
5	F	F

n-processes Solution – Mutual Exclusion

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
P1 P3 P4 key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

n = 6

j

lock
T

P	W	K
0	F	F
1	T	T
2	F	F
3	T	T
4	T	F
5	F	F

n-processes Solution – Bounded Wait

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
P1 P3   key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
P4  j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

n = 6

j

lock
T

P	W	K
0	F	F
1	T	T
2	F	F
3	T	T
4	F	F
5	F	F

n-processes Solution – Bounded Wait

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
P1 P3   key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
P4      waiting[j] = false;
    /* remainder section */
} while (true);
```

n = 6

j
1

lock
T

P	W	K
0	F	F
1	T	T
2	F	F
3	T	T
4	F	F
5	F	F

n-processes Solution – Progress

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
P3    key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
P1  j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

n = 6

j
1

lock
T

P	W	K
0	F	F
1	F	T
2	F	F
3	T	T
4	F	F
5	F	F

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**

acquire() and release()

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

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

- Definition of the **`signal()`** operation

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

Semaphore Usage

Two main types of semaphores:

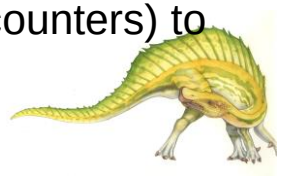
- **Counting semaphore** – integer value can range over an unrestricted domain, Allows access to a resource for up to a specified number of threads.
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**

- 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

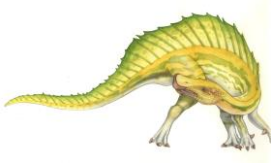
```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore
  - Use multiple binary semaphores along with additional variables (like counters) to simulate the effect of a counting 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



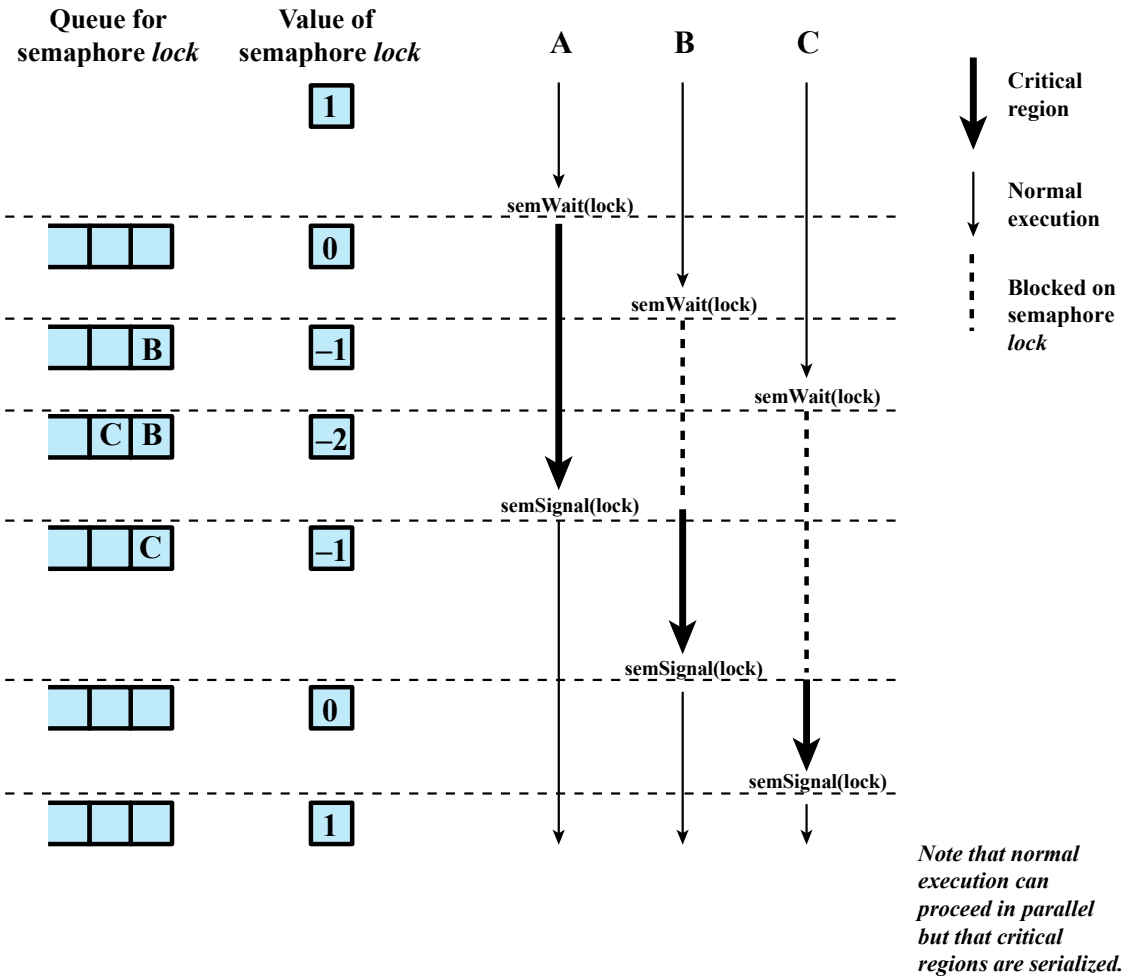
Semaphore Implementation with no Busy waiting

- 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;
```

Implementation with no Busy waiting (Cont.)

```
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);
    }
}
```



Processes Accessing Shared Data Protected by a Semaphore

Semaphore Implementation in Python

```
import threading
# Custom semaphore class
class SimpleSemaphore:
    def __init__(self, initial_count):
        # The number of available resources
        self.count = initial_count
        # A lock to protect the critical section
        self.lock = threading.Lock()
        # Condition variable to signal waiting threads
        self.condition = threading.Condition(self.lock)

    # Acquire a resource
    def acquire(self):
        # Enter the critical section
        with self.lock:
            while self.count == 0:
                # If no resources are available, wait until another thread releases one
                self.condition.wait()
            # Decrement the semaphore count as a resource is acquired
            self.count -= 1
            print(f"Resource acquired, remaining resources: {self.count}")

    # Release a resource
    def release(self):
        with self.lock:  # Enter the critical section
            # Increment the semaphore count as a resource is released
            self.count += 1
            print(f"Resource released, remaining resources: {self.count}")
            # Notify one waiting thread that a resource has been released
            self.condition.notify()
```

Semaphore Implementation in Python

```
# Example usage
def thread_task(semaphore, thread_id):
    print(f"Thread {thread_id} is trying to acquire a resource...")
    semaphore.acquire()
    print(f"Thread {thread_id} has acquired a resource!")
    threading.Event().wait(2)  # Simulate some work by the thread
    print(f"Thread {thread_id} is releasing a resource...")
    semaphore.release()

# Initialize the custom semaphore with 2 resources
semaphore = SimpleSemaphore(2)

# Create multiple threads simulating resource usage
threads = []
for i in range(5):
    thread = threading.Thread(target=thread_task, args=(semaphore, i))
    threads.append(thread)
    thread.start()

# Wait for all threads to complete
for thread in threads:
    thread.join()

print("All threads have finished.")
```

Using Python's built-in threading.Semaphore

```
import threading
import time

# Create a semaphore allowing up to 2 threads to access the critical secti
semaphore = threading.Semaphore(2)

# Function to simulate some work with a shared resource
def access_shared_resource(thread_id):
    print(f"Thread {thread_id} is waiting to access the resource.")
    # Acquire the semaphore (decrement its counter)
    with semaphore:
        print(f"Thread {thread_id} has accessed the resource.")
        time.sleep(2)  # Simulate the thread doing some work
        print(f"Thread {thread_id} is releasing the resource.")

# Create multiple threads that will attempt to access the shared resource
threads = []
for i in range(5):
    thread = threading.Thread(target=access_shared_resource, args=(i,))
    threads.append(thread)
    thread.start()

# Wait for all threads to complete
for thread in threads:
    thread.join()

print("All threads have finished.")
```

Deadlock and Starvation

- **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
  - Incorrect use of semaphore operations

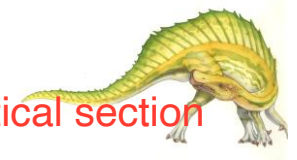
P_0		P_1	
`wait(S);`	Decreases S from 1 to 0. It now holds semaphore S.	`wait(Q);`	Decreases Q from 1 to 0. It now holds semaphore Q.
`wait(Q);`	Q is already 0 (held by P1), so P0 waits for Q to become 1.	`wait(S);`	S is already 0 (held by P0), so P1 waits for S to become 1
`...`		`...`	
`signal(S);`		`signal(Q);`	
`signal(Q);`		`signal(S);`	

No Mutual Exclusion: allowing multiple processes to enter the critical section simultaneously.

Race Conditions: leading to inconsistent data

- **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**

child process takes priority of parent process if another process wants to enter critical section



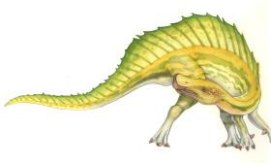
Monitors

- Monitors are high-level synchronization constructs that provide mutual exclusion and condition synchronization.
- Encapsulate shared variables, procedures, and synchronization mechanisms
- Ensuring that only one process can be active within the monitor at any given time.

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

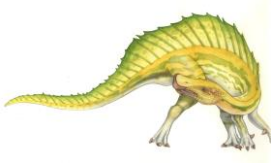
    procedure Pn (…) {……}

     Initialization code (…) { … }
}
```

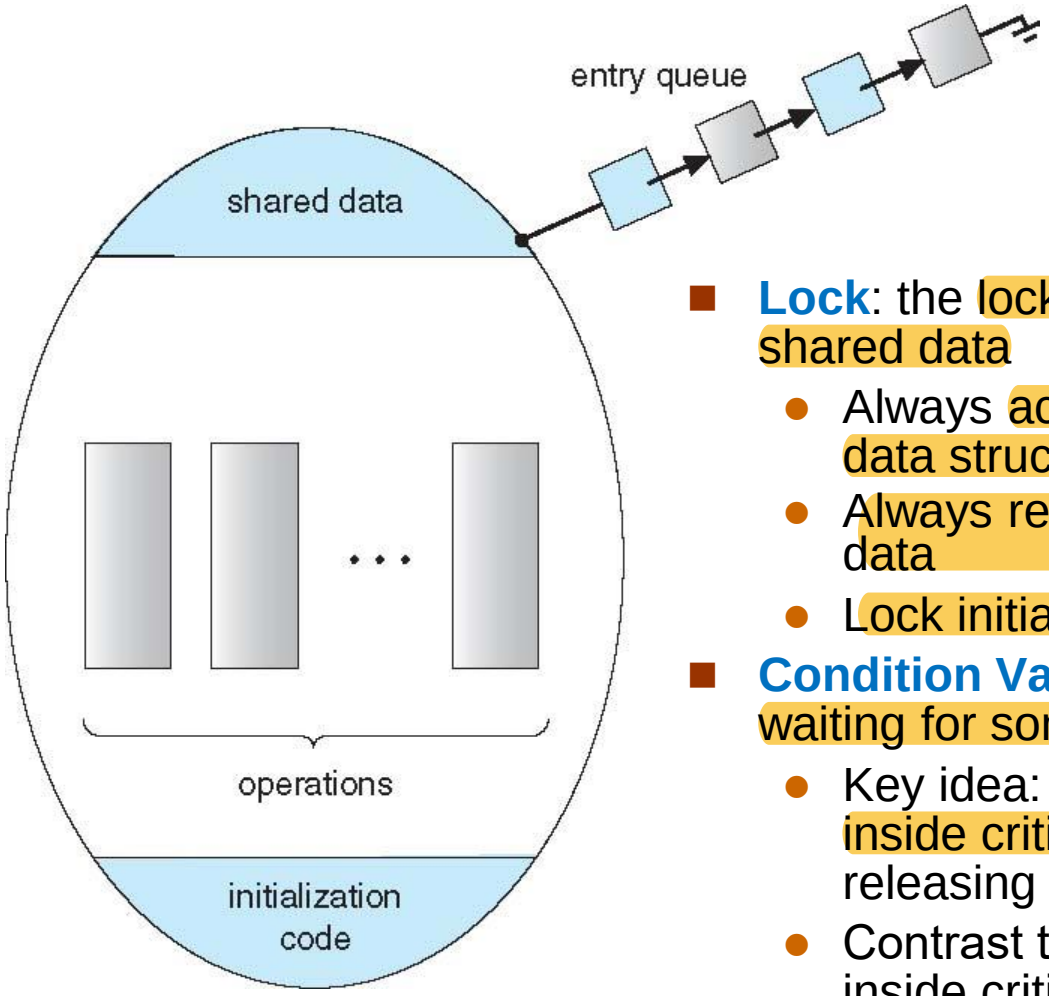


Condition Variables

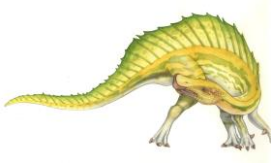
- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then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



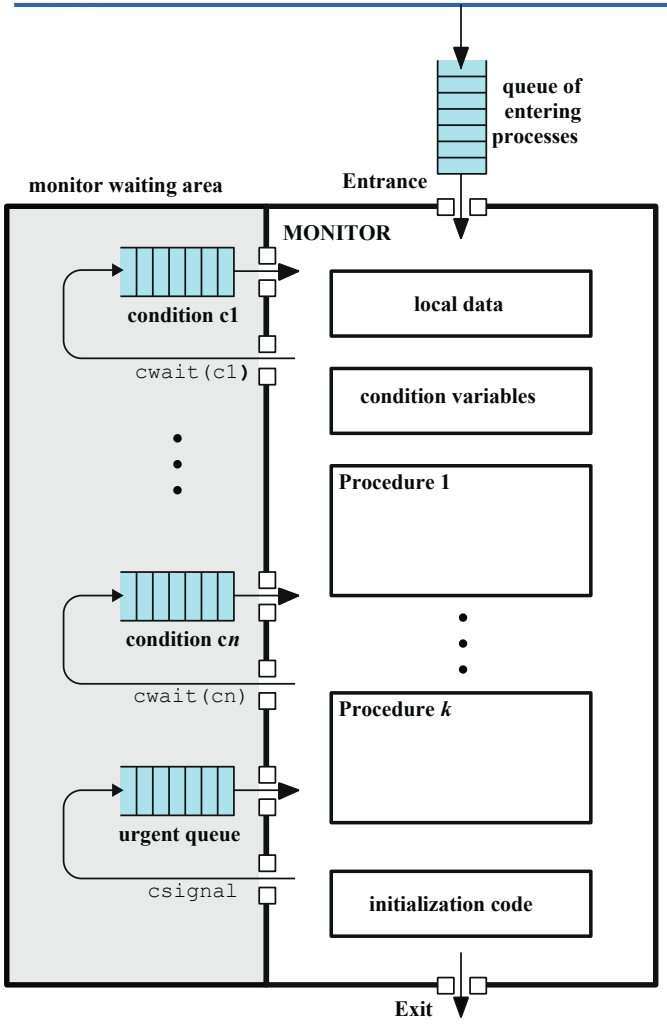
Schematic view of a Monitor



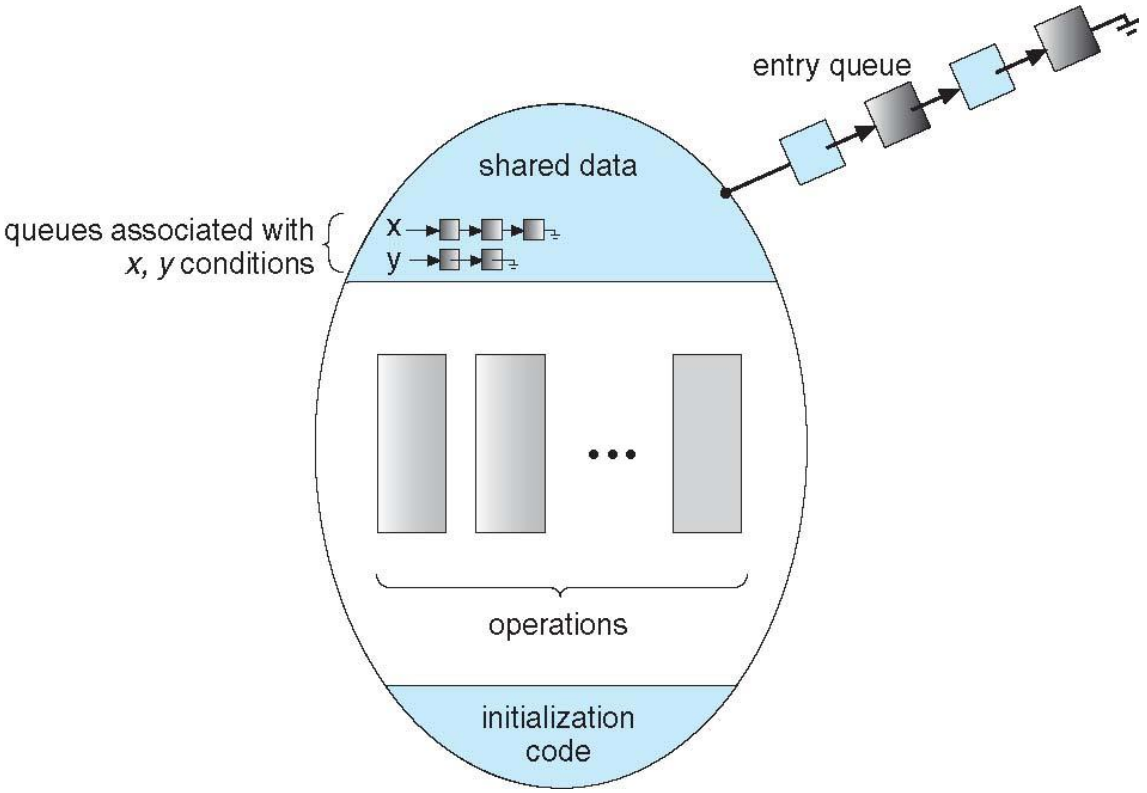
- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

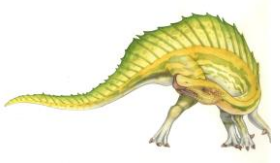


Monitor with Condition Variables



Structure of a Monitor





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 paralel. 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
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including C#, Java

```
monitor ExampleMonitor {
    // Shared variable
    int resource;

    // Condition variable
    condition x;

    // Procedure that uses the shared resource
    procedure useResource() {
        // Some condition to wait on
        if (/* resource not available */) {
            x.wait();
        }
        // Critical section: use the resource
        // ...

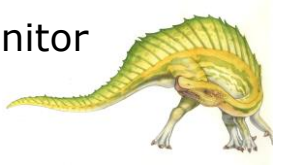
        // Optionally signal another process
        x.signal();
    }
}
```

Signal and wait

P: [useResource()] -- signal(x) -- (waits) <------ resumes after Q exits
|
Q: (waiting) <-- resumes -- uses resource -- exits monitor

Signal and continue

P: [useResource()] -- signal(x) -- continues -- exits monitor
|
Q: (waiting) ---------- resumes -- uses resource -- exits monitor



Pros and Cons of Each Option

Signal and Wait

- **Pros**:
  - Ensures that the signaled process (Q) proceeds as soon as possible.
  - Reduces waiting time for `Q`, which can be important if Q has higher priority.
- **Cons**:
  - The signaling process (P) is delayed, which may not be optimal if P has more work to do.
  - Can lead to inefficiencies if P frequently signals and waits.

Signal and Continue

- **Pros**:
  - The signaling process (P) is not interrupted and can complete its execution efficiently.
  - May improve throughput in systems where P has minimal remaining work.
- **Cons**:
  - The waiting process (Q) may experience increased latency.
  - Potential for starvation if P (or other processes) continue to hold the monitor for extended periods.

Language Implementations and Compromises

Language Implementer's Decision

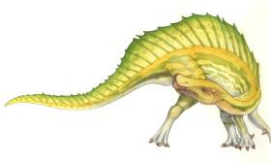
- Different programming languages and monitor implementations choose between these options based on:
  - **Performance considerations**
  - **Fairness requirements**
  - **Ease of reasoning for programmers**
- There isn't a one-size-fits-all solution; each option has trade-offs.

Compromise in Concurrent Pascal

- **Behavior**: When `P` executes `x.signal()`, it **immediately exits the monitor**, The waiting process `Q` is then resumed and enters the monitor.
- **Advantages**:
  - `P` doesn't continue executing inside the monitor, so `Q` doesn't wait unnecessarily.
  - `P` doesn't wait inside the monitor; it exits, allowing others to proceed, can continue its execution outside the monitor or attempt to re-enter if needed..

Implementation in Other Languages

- **C#, Java**: Generally adopt the **signal and continue** semantics.
- Prioritizes throughput and avoids unnecessary context switches.
- Responsibility on the programmer to handle potential delays for waiting threads.



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

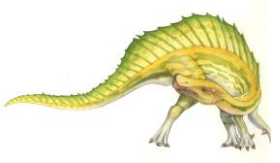
Monitor Implementation – Condition Variables

- 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--;
```



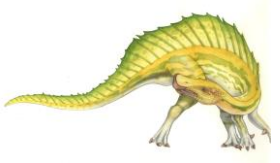
Monitor Implementation (Cont.)

- 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



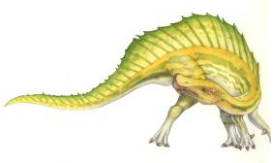
Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

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

R.release;
```

- Where R is an instance of type **ResourceAllocator**



A Monitor to Allocate Single Resource

```
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;
   }
}
```

End of Chapter 6