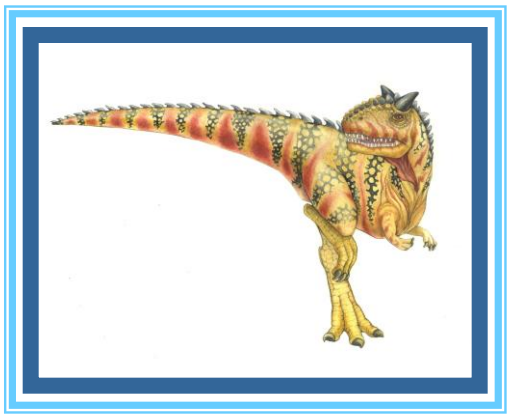
Chapter 6: Synchronization Tools

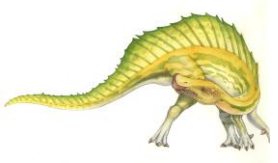


Silberschatz, Galvin and Gagne ©2018



Outline

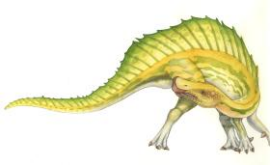
- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation





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 synchronization tools under low, moderate, and high contention

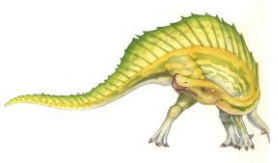


Troy University Resources



Background

- Concurrent execution can interleave operations, even on a single CPU.
- Shared mutable data needs a synchronization protocol to preserve its invariants.
- A read, an arithmetic operation, and a write do not together form one atomic operation.
- The bounded-buffer counter below illustrates a lost update. Chapter 7 gives synchronized solutions.





Bounded Buffer: Unsynchronized Counter

Illustrative pseudocode. Both processes share counter, initially 0. This version is unsafe.

```
/* producer */
while (true) {
  item = produce();
  while (counter == N) { }
  buffer[in] = item;
  in = (in + 1) % N;
  counter++;
}
```

```
/* consumer */
while (true) {
  while (counter == 0) { }
  item = buffer[out];
  out = (out + 1) % N;
  counter--;
  consume(item);
}
```

The two updates can overwrite one another. Ordinary C/C++ variables also create data races.



Troy University Resources

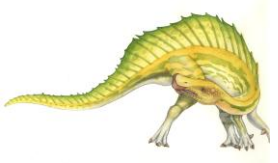


Race Condition: A Lost Update

Initial counter = 5. One produced item and one consumed item should leave counter = 5.

Step	Producer	Consumer	counter
0	read 5		5
1	compute 6		5
2		read 5	5
3		compute 4	5
4	write 6		6
5		write 4	4

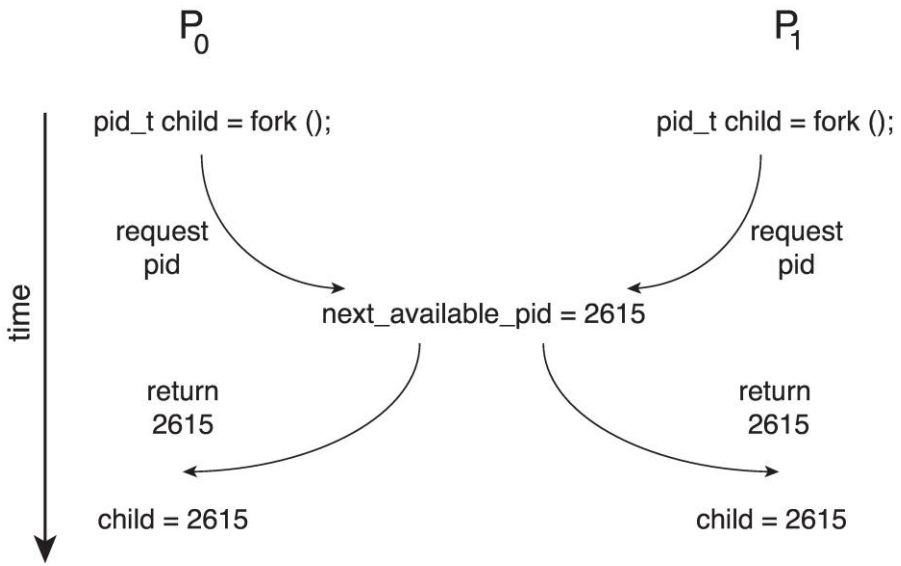
The final value is 4. Reversing the last two writes can leave 6.





Race Condition

- Concurrent PID allocation needs mutual exclusion.
- Without protection, two requests could observe the same available PID.



The figure illustrates a missing synchronization protocol. Actual kernels protect PID allocation.

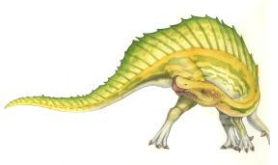


Troy University Resources



Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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**

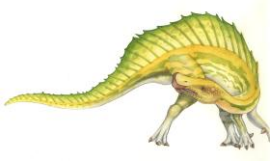




Critical Section

- General structure of process P_i

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



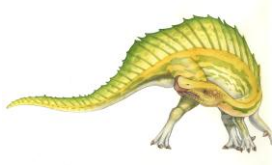
Troy University Resources



Critical-Section Problem (Cont.)

Requirements for 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
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 process 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





Interrupt-Based Synchronization

- Privileged kernel code can mask local interrupts for a short critical section.
- This can protect against relevant interrupt handlers on that CPU. It does not stop another CPU.
- Multiprocessor access to shared data therefore still needs an appropriate lock or atomic protocol.
- Keep interrupt-disabled regions short. Normal user processes cannot use this as a general locking API.

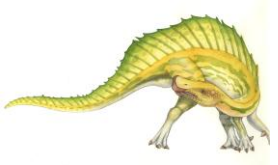


Troy University Resources



Critical Sections in the Kernel

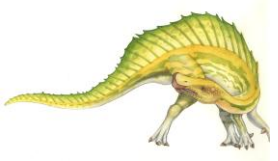
- A preemptive kernel can switch tasks while they execute in kernel mode, subject to its locking rules.
- In a nonpreemptive kernel, a task normally runs until it blocks, yields, or returns to user mode.
- Nonpreemption simplifies task-to-task synchronization on one CPU. Interrupts and other CPUs can still race.
- Disabling preemption and disabling interrupts protect different execution contexts.





Strict Alternation

- Two processes share turn, initially 0 or 1.
- The idealized model assumes atomic reads and writes with sequentially consistent ordering.
- Process i enters only when turn == i and gives the turn to j on exit.
- This model explains the algorithm. A real C/C++ implementation requires atomic objects or locks.



Troy University Resources



Algorithm for Process P_i

```
while (true){

        while (turn == j);

        /* critical section */

        turn = j;

        /* remainder section */

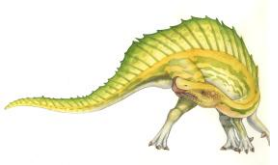
}
```





Strict Alternation: Correctness

- Mutual exclusion holds: turn cannot equal 0 and 1 at the same time.
- Progress fails: if turn == j and process j stays in its remainder section, process i cannot enter an empty critical section.
- A solution must allow a process to enter when no competing process wants the resource.
- Alternation alone therefore does not solve the critical-section problem.

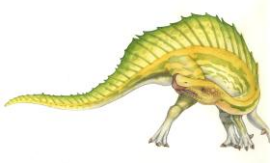


Troy University Resources



Peterson's Solution

- Two processes share flag[2], initially false, and turn.
- flag[i] records that process i wants to enter. turn resolves simultaneous requests.
- The classic proof assumes atomic accesses and sequentially consistent ordering.
- Plain bool and int variables in C/C++ do not provide those guarantees. A correct implementation can use sequentially consistent atomics.





Peterson's Algorithm

Pseudocode under sequential consistency. Initialize both flags to false. j = 1 - i.

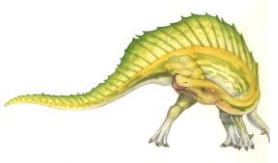
```
while (true) {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j) { }
     /* critical section */
     flag[i] = false;
     /* remainder section */
}
```





Peterson's Solution: Correctness

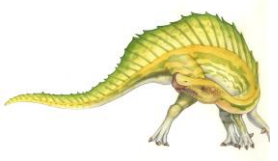
- Mutual exclusion: if both flags are true, the shared turn value lets at most one process pass its entry test.
- Progress: a process with no interested competitor can enter without waiting for that competitor to take a turn.
- Bounded waiting: after a process requests entry, the other can enter at most once before yielding to it.
- These arguments assume sequential consistency and continued execution of the participating processes.





Synchronization and Memory Ordering

- Compilers and CPUs may reorder memory operations while preserving permitted single-thread behavior.
- A concurrent algorithm must use the ordering and visibility guarantees of its programming language and CPU.
- In C/C++, conflicting accesses without a happens-before relation form a data race if at least one access is non-atomic.
- A C/C++ data race has undefined behavior. Adding volatile or an arbitrary fence does not make it safe.



Troy University Resources



Message Passing Without Ordering

Atomicity alone is insufficient to publish related data. result can be 0 or 100.

```
/* Shared atomics, initialized before threads start. */
atomic_bool ready = false;
atomic_int x = 0;

/* writer */
atomic_store_explicit(&x, 100, memory_order_relaxed);
atomic_store_explicit(&ready, true, memory_order_relaxed);

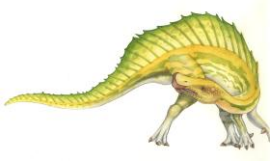
/* reader */
while (!atomic_load_explicit(&ready,
                             memory_order_relaxed)) { }
int result = atomic_load_explicit(&x,
                                  memory_order_relaxed);
```





Why the Message Can Be Stale

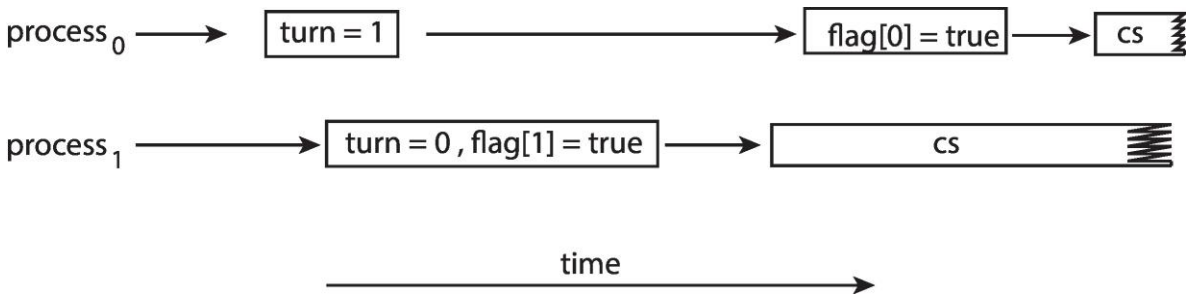
- Relaxed atomic operations prevent data races on each atomic object.
- They do not establish a synchronizes-with edge between the writer and reader.
- Seeing ready == true therefore does not require the reader to observe the earlier store of 100 to x.
- If x and ready were ordinary shared C variables instead, the unsynchronized program would have undefined behavior.



Troy University Resources



Peterson's Solution and Reordering



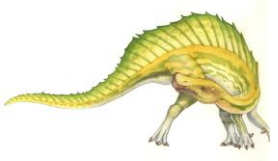
- Reordering can invalidate the reasoning used for Peterson's algorithm.
- The textbook proof assumes sequentially consistent reads and writes.
- In C/C++, ordinary shared variables cause a data race. A standalone memory barrier does not make those accesses valid.
- A C11 implementation can use sequentially consistent atomic variables.





Memory Models and Barriers

- A memory model defines which observations of concurrent memory operations a program may make.
- Stronger models restrict more reorderings. They do not promise instantaneous visibility of every store.
- A barrier or fence constrains specified classes of operations. Its exact guarantees depend on the architecture and instruction.
- A synchronization protocol needs both atomic accesses and suitable ordering.

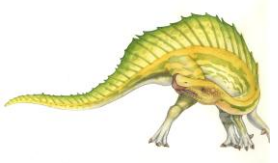


Troy University Resources



Ordering in C11

- A release store can publish earlier writes to another thread.
- An acquire load that reads from that release establishes synchronization, making the earlier writes available to the reader.
- A sequentially consistent atomic operation also participates in the global order of sequentially consistent operations.
- Fences have specific rules involving atomic operations. A fence alone does not repair ordinary-variable data races.





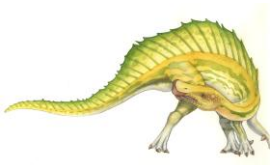
Release and Acquire Publication

Once the reader observes true, it can safely read x and print 100. No further writes to x occur.

```
#include <stdatomic.h>
#include <stdbool.h>
#include <stdio.h>
atomic_bool ready = false;
int x = 0;

/* writer: runs once */
x = 100;
atomic_store_explicit(&ready, true,
                      memory_order_release);

/* reader */
while (!atomic_load_explicit(&ready,
                             memory_order_acquire)) { }
printf("%d\n", x);
```

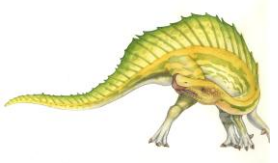


Troy University Resources



Hardware Support for Synchronization

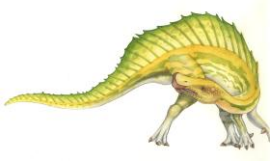
- Atomic read-modify-write instructions provide indivisible updates to a shared location.
- Examples include test-and-set and compare-and-swap (CAS).
- Memory-ordering instructions and the language memory model govern how surrounding accesses become ordered.
- These mechanisms support locks, atomic counters, and more complex synchronization tools.





Atomic Read-Modify-Write Instructions

- Test-and-set reads the previous value and stores true as one atomic operation.
- Compare-and-swap reads a location and replaces it only if it matches an expected value.
- Atomic means other participants cannot observe an intermediate state of that operation.
- The next definitions describe hardware semantics. Ordinary C functions with these statements would not themselves be atomic.



Troy University Resources

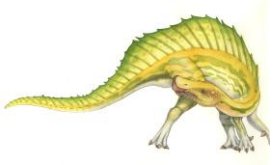


The test_and_set Instruction

Hardware pseudocode: the entire read-modify-write executes atomically.

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

Returns the previous value and sets the location to true.





Lock Using test_and_set

Pseudocode with suitable acquire/release ordering. The simple lock provides no bounded-waiting guarantee.

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



Troy University Resources



The compare_and_swap Instruction

Hardware pseudocode: comparison and conditional update form one atomic operation.

```
int compare_and_swap(int *value,
                     int expected, int desired) {
    int old = *value;
    if (old == expected)
        *value = desired;
    return old;
}
```

This textbook CAS returns the old value. C11 compare-exchange instead returns success and updates expected on failure.

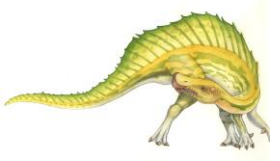




Lock Using compare_and_swap

Pseudocode with suitable memory ordering. Repeated CAS attempts can starve a competitor.

```
/* Shared lock = 0. */
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0) { }
    /* critical section */
    lock = 0;
    /* remainder section */
}
```





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 */
}
```

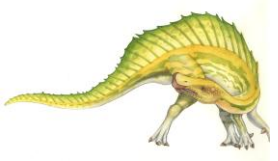
Pseudocode under sequential consistency. Shared waiting[0..n-1] = false and lock = 0.





Atomic Variables

- Atomic objects support operations that avoid conflicting partial updates on that object.
- An atomic read followed by an atomic write still consists of two operations. A read-modify-write increment avoids a lost update.
- Atomicity of one counter does not protect a compound invariant involving several variables.
- Choose the memory ordering required by the algorithm. Use a mutex when it is the clearer correct design.



Troy University Resources



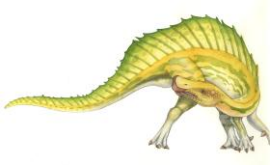
Atomic Increment in C11

The default C11 atomic operations below use sequentially consistent ordering.

```
#include <stdatomic.h>

void increment(atomic_int *v) {
    atomic_fetch_add(v, 1);
}

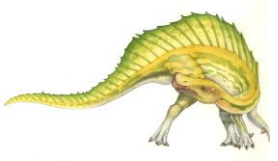
/* Equivalent CAS loop, assuming the result fits in int */
void increment_cas(atomic_int *v) {
    int expected = atomic_load(v);
    while (!atomic_compare_exchange_weak(v, &expected,
            expected + 1)) { }
}
```





Mutex Locks and Spinlocks

- A mutex gives one owner exclusive access to protected state. The owner releases the lock.
- A mutex implementation may block a waiting thread, spin briefly, or combine both approaches.
- A spinlock waits by repeatedly checking the lock. It consumes CPU time while waiting.
- Spinning can suit a very short wait when the owner is running on another CPU. Blocking suits longer waits.



Troy University Resources



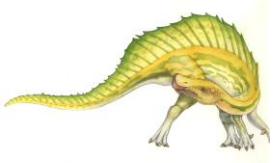
Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

            critical section

        release lock

remainder section
}
```





Semaphore Operations

Abstract atomic operations. A spin-based implementation may busy-wait instead of blocking.

```
wait(S):
  atomically:
    if permit exists:
      consume one
    else:
      enqueue and block
```

```
signal(S):
  atomically:
    add a permit or
      wake a waiter
```

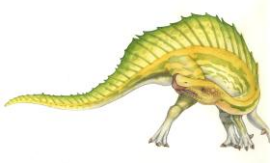
The test and decrement must form one protected operation. An ordinary while loop followed by S-- is unsafe.





Counting and Binary Semaphores

- A counting semaphore represents available permits. Public APIs use a nonnegative count with an implementation-defined maximum.
- A binary semaphore restricts the logical permit count to 0 or 1.
- A semaphore can coordinate events between different threads. It does not impose the ownership rule of a mutex.
- A binary semaphore initialized to 1 can implement mutual exclusion when the program uses wait and signal correctly.





Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

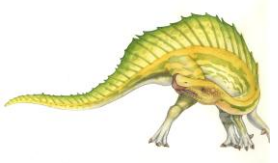
```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```





Implementing a Semaphore

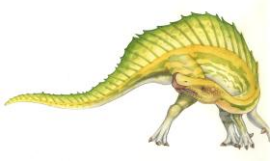
- Updates to a semaphore count and its wait queue must be synchronized.
- Blocking must be coordinated with queue insertion so a wakeup cannot be lost.
- An implementation can use a short internal lock plus scheduler support to park and wake threads.
- Blocking the application while a permit is unavailable avoids spinning for the entire application critical section.





A Semaphore with a Wait Queue

- The semaphore holds a count and a queue of waiting processes or threads.
- A queue entry identifies a waiter and links it to the remaining waiters.
- block() suspends the caller after safely enqueuing it.
- wakeup() makes a selected waiter ready to run. It does not guarantee immediate execution.
- This textbook implementation may use a negative internal count to record the number waiting.

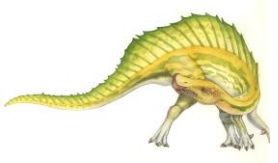




Implementation with no Busy waiting (Cont.)

- Waiting queue

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





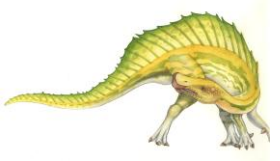
Semaphore Implementation

Kernel pseudocode. Count and queue changes are protected; parking must coordinate atomically with wakeup.

```
wait(S):
  S.value--
  if S.value < 0:
    enqueue(current, S.queue)
    park_queued_thread()
```

```
signal(S):
  S.value++
  if S.value <= 0:
    P = dequeue(S.queue)
    make_ready(P)
```

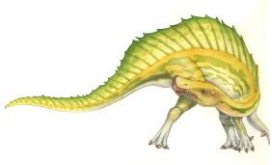
A negative internal count records waiters. Waking a thread makes it ready; it does not immediately run.





Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal(mutex)`** …. **`wait(mutex)`**
  - **`wait(mutex)`** … **`wait(mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.





Monitors

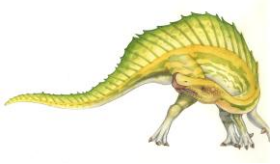
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

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

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

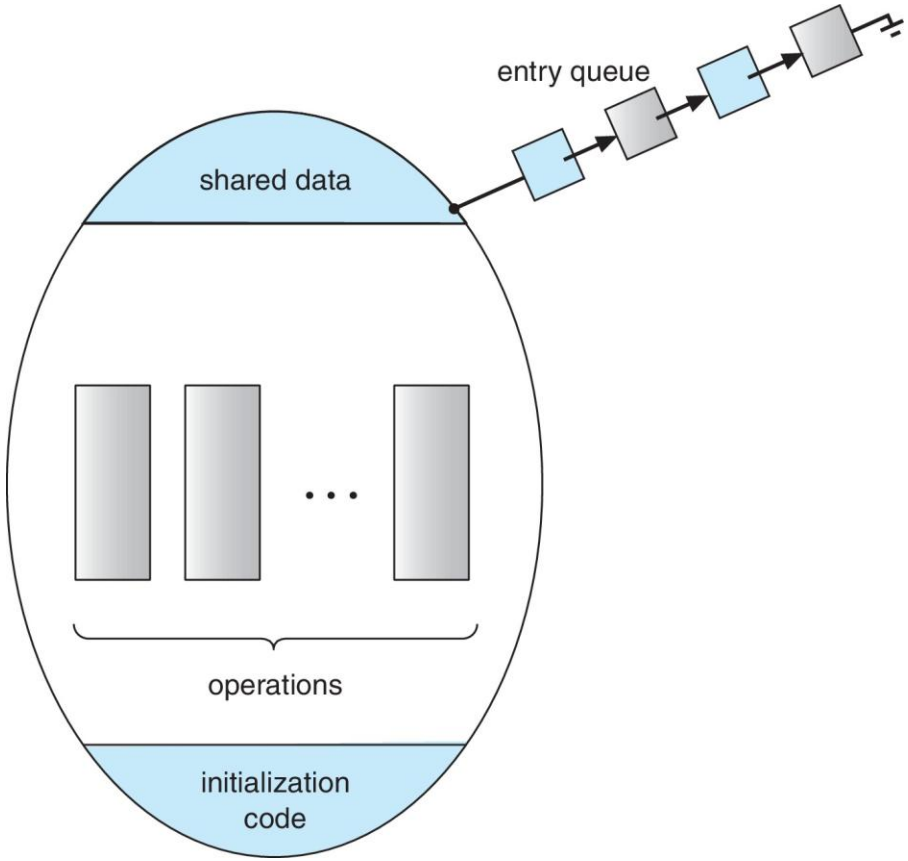
  initialization code (…) { … }
}
```

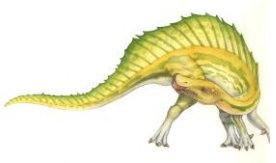


Troy University Resources



Schematic view of a Monitor







Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

```
wait(mutex);
    …
    body of P;
    …
signal(mutex);
```

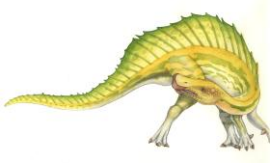
- Mutual exclusion within a monitor is ensured





Condition Variables

- A condition variable is a wait queue associated with a monitor or mutex-protected predicate.
- wait() atomically releases the monitor lock and suspends the caller. It reacquires the lock before returning.
- signal() wakes an eligible waiter, if one exists. It stores no permit when the queue is empty.
- With signal-and-continue semantics, always recheck the predicate in a while loop after waking.





Condition-Variable Signal Semantics

A monitor allows only one executing thread inside at a time.

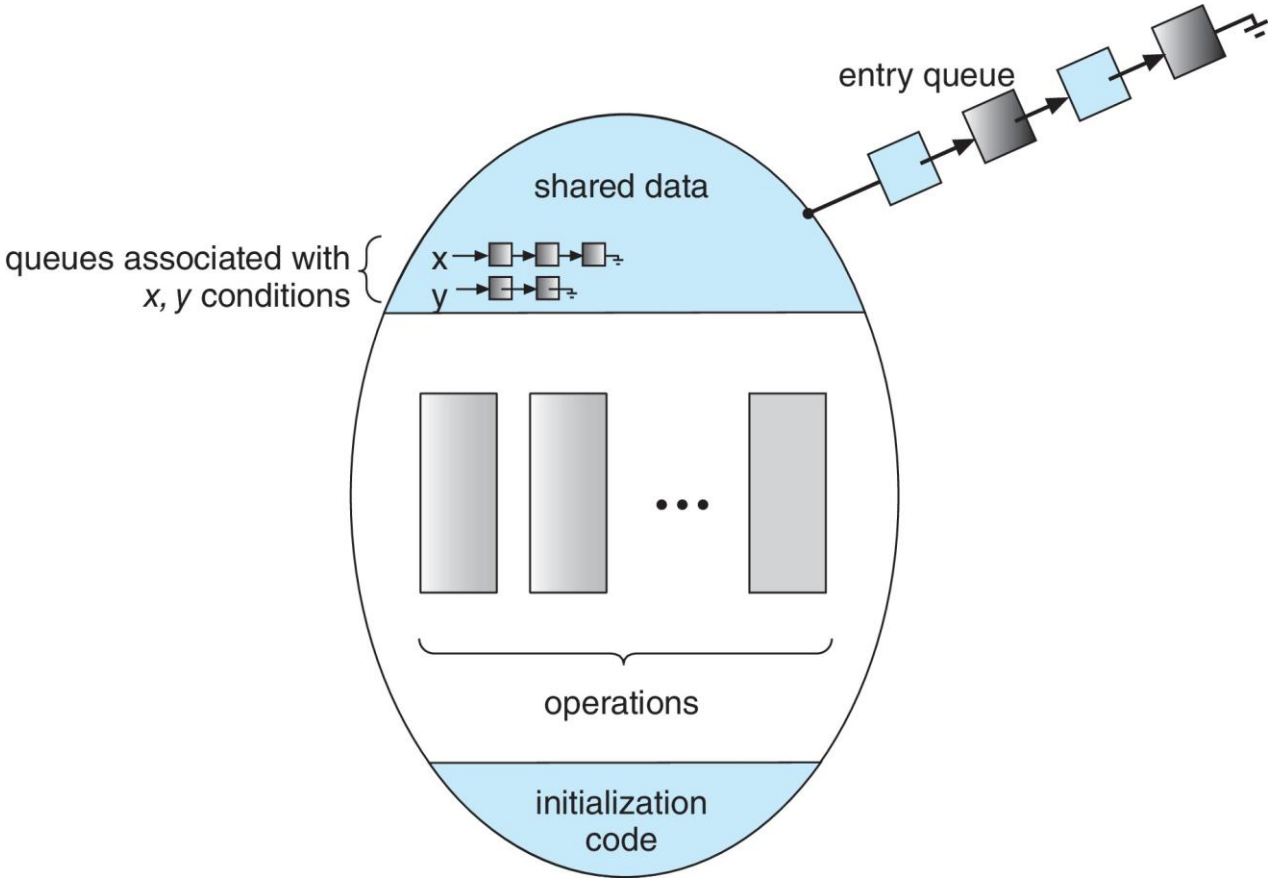
Model	After P signals Q	Predicate on return
Hoare: signal and wait	Q runs in the monitor immediately; P waits.	Handoff preserves the signaled condition under the model.
Mesa: signal and continue	P continues; Q later reacquires the lock.	Q must test again in a while loop.

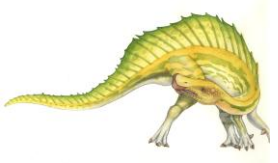
Java and POSIX use signal-and-continue behavior. Spurious wakeups are also possible.





Monitor with Condition Variables







Condition Variable: S1 Before S2

Both procedures execute under the same monitor lock. done records the event even if no one is waiting.

```
/* Shared monitor state */
boolean done = false;
condition x;

procedure F1() {
    S1;
    done = true;
    x.signal();
}

procedure F2() {
    while (!done)
        x.wait();
    S2;
}
```



Troy University Resources



Hoare Monitor Implementation

Signal-and-wait model: next holds signallers until the awakened thread leaves the monitor or waits again.

```
semaphore mutex = 1;
semaphore next = 0;
int next_count = 0;

procedure P(...) {
    wait(mutex);
    /* monitor body */
    if (next_count > 0)
        signal(next);
    else
        signal(mutex);
}
```





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--;
```

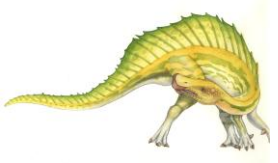




Implementation (Cont.)

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

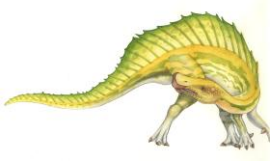
```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```





Priority-Ordered Condition Waiting

- A monitor can offer an abstract wait(priority) operation to choose among queued waiters.
- In this example, a smaller priority number selects an earlier waiter.
- This is a textbook scheduling extension. POSIX pthread_cond_wait and Java Object.wait do not take such a priority argument.
- Priority order alone does not guarantee bounded waiting for low-priority requests.



Troy University Resources

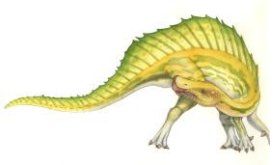


Single-Resource Allocation

t is the declared maximum usage time. A priority queue selects the waiting request with the smallest t.

```
R.acquire(t);
/* use the resource */
R.release();
```

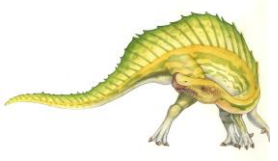
The caller must release the resource after use. The allocator does not automatically stop it when t expires.





Resource-Allocator Policy

- The allocator serializes access to one resource.
- Among queued requests, it prefers the smallest declared usage time.
- The monitor implementation assumes immediate handoff to the selected waiter (Hoare semantics).
- New shorter requests can repeatedly bypass a long request. Fairness needs an additional policy.
- Production code also needs an exception-safe release path.



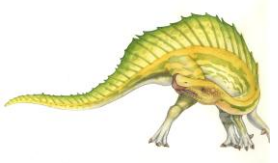
Troy University Resources



Resource-Allocator Monitor

```
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;
   }
}
```

Textbook Hoare monitor with priority wait(time). The if test relies on immediate handoff.





Single Resource Monitor (Cont.)

- Usage:

```
acquire
  ...
release
```

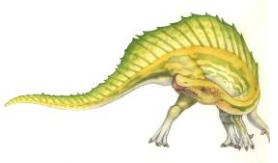
- Incorrect use of monitor operations
  - `release()` … `acquire()`
  - `acquire()` … `acquire())`
  - Omitting of `acquire()` and/or `release()`





Liveness

- Safety concerns what must never happen, such as simultaneous access to a protected critical section.
- Liveness concerns eventual progress, under stated scheduling assumptions.
- Deadlock prevents a set of participants from making progress. Starvation can delay one participant indefinitely while others continue.
- Bounded waiting is stronger than eventual entry: it limits how often other requests can overtake a waiting request.



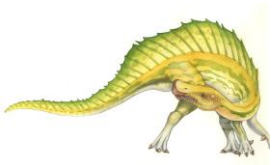


Liveness

- **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

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

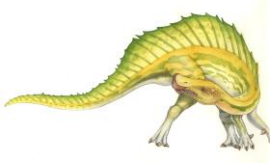
- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.



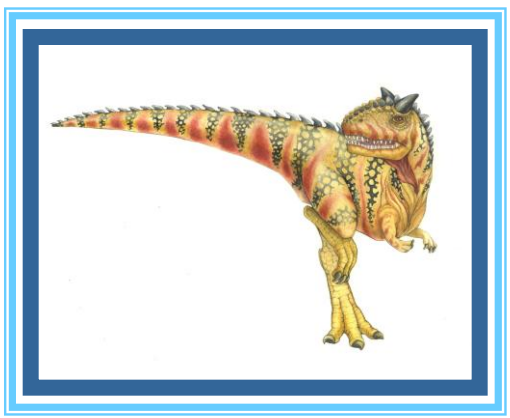


Starvation and Priority Inversion

- Starvation: a participant waits indefinitely while other participants continue to make progress.
- Priority inversion: a high-priority thread waits for a lock held by a lower-priority thread.
- An intermediate-priority thread can delay that lock owner and extend the high-priority thread's wait.
- Priority inheritance temporarily boosts the lock owner's priority. It helps control inversion, but does not solve arbitrary deadlocks.



End of Chapter 6



Silberschatz, Galvin and Gagne ©2018