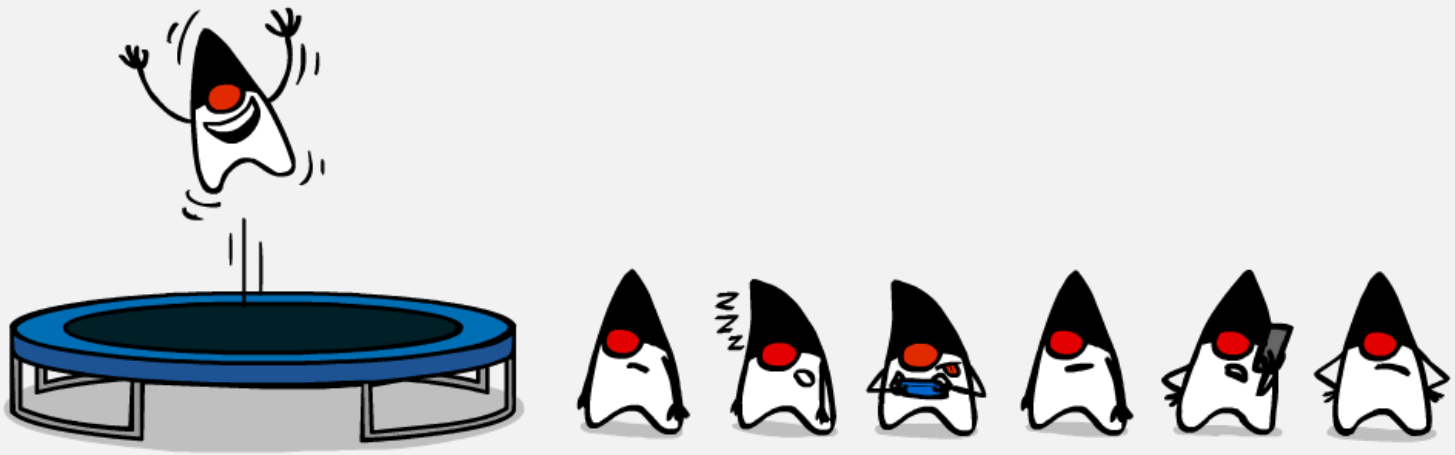
Synchronization (Part 2)



Learning Objectives

- Understand and apply the monitor concept to solve synchronization problems in concurrent programs
- Understand and apply the concept of condition variables in monitors
- Understand and use the readers-writers concurrency design pattern
- Use synchronization constructs from Pthreads library and Java packages

Monitors

A monitor is designed to allow concurrency while retaining the advantage of a structured construct.

- Each monitor is entrusted with a specific task and in turn it will have its own privileged data and instructions.
- Entry to the monitor by one process excludes entry by any other process.
- Since all processing is encapsulated inside the monitor, the programmer cannot accidentally change the monitor variables.
- A system would be more robust and efficient if we have a monitor for each specific task that involves multiple processes.

Condition Variables

Monitors include condition variables to allow processes to signal each other. A condition variable is a global structure inside the monitor that supports three operations.

- `wait()`: Suspends the invoking process until it receives another signal from another process.
- `signal()`: Resumes exactly one process if any processes are waiting on the condition variable. If no process is waiting, then the signal has no effect.
- `queue()`: Returns `TRUE` is there is at least one process suspended on the condition variable, and `FALSE` otherwise.

Queue Management in the Monitor

Signal takes effect immediately (Tony Hoare Semantics).

- The signal statement should always be the last statement before a process leaves the monitor. This allows the signaled process to be woken up immediately. So a process that gets past the `wait()` can be guaranteed that the condition is true.
- In case we have processes waiting to enter the monitor as well as processes waiting on a condition variable, then the next process to be allowed in the monitor is taken from the queue of processes waiting on the condition variable.
- The processes waiting in the queue on a condition variable can be woken by the scheduler in either arbitrary, first-come-first-served, or priority order. In the priority scheduling case we will modify the `wait()` to have an argument `wait(priority)`, where `priority` is an integer with smaller integers implying higher priority.

Signal does not have effect immediately (Per Brinch Hansen Semantics).

The signal is recorded and delivered after the signaling process leaves the monitor. In this case a process on waking up has to test the condition again before proceeding. This would lead to fewer context switches.

Bank Account Monitor (pseudocode)

```
monitor sharedBalance
{
    int balance;
    public:
        credit(int amount) { balance = balance + amount;}
        debit(int amount) { balance = balance - amount;}
}
```

Simulation of a Binary Semaphore by a Monitor


```
monitor SemaphoreSimulation {
boolean busy = FALSE;
condition notbusy;
    public:
        down() {
            if (busy) notbusy.wait();
            busy = TRUE;
        }

        up() {
            busy = FALSE;
            notbusy.signal();
        }
}
```

Readers and Writers

Suppose a resource is to be shared among a community of processes of two types: *readers* and *writers*.

- A *reader* process can share the resource with any other reader process, but not with any writer process.
- A *writer* process requires exclusive access to the resource whenever it acquires any access to the resource.
- Several different policies can be implemented for managing the shared resource.
  - *Policy 1: Readers preference*. As long as a reader holds the resource and there are new readers arriving, any writer must wait for the resource to become available. Writer can starve.
  - *Policy 2: Writers preference*. When a writer process requests access to the shared resource, any subsequent reader process must wait for the writer to gain access to the shared resource and then release it. Reader can starve.
  - *Policy 3: Give neither preference*. Neither readers nor writers can starve.

Readers and Writers Monitor: Policy 1


```
monitor ReadersWriters {
    int numberOfReaders = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;
public:
    void startRead() {
        if (busy) okToRead.wait();
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal();
    }
    void finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders == 0) okToWrite.signal();
    }
    void startWrite() {
        if (numberOfReaders > 0 || busy) okToWrite.wait();
        busy = TRUE;
    }
    void finishWrite() {
        busy = FALSE;
        if (okToRead.queue())
            okToRead.signal();
        else
            okToWrite.signal();
    }
}
```

Readers and Writers Monitor: Policy 2


```
monitor ReadersWriters {
    int numberOfReaders = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;
public:
    void startRead() {
        if (busy || okToWrite.queue()) okToRead.wait();
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal();
    }
    void finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders == 0) okToWrite.signal();
    }
    void startWrite() {
        if (numberOfReaders > 0 || busy) okToWrite.wait();
        busy = TRUE;
    }
    void finishWrite() {
        busy = FALSE;
        if (okToWrite.queue())
            okToWrite.signal();
        else
            okToRead.signal();
    }
}
```

A Resource Allocation Monitor

```
monitor ResourceAllocator {
    boolean busy = FALSE;
    condition resource;

    public:
        acquire(time: integer) {
        // time = How long is the resource needed for
            if (busy)  resource.wait(time);
            busy = TRUE;
        }

        release() {
            busy = FALSE;
            resource.signal();
        }
}
```

-We assume priority scheduling for the wait queue.

Examples of using Monitors

Other monitor examples:

- solve the producer consumer problem using monitors.
- an alarm clock monitor.
- a monitor that allows access to a file by no more than n processes.
- a monitor that allows access to a group of processes as long as the sum total of their process ids doesn't exceed a certain integer, say n.

The idea is to use the weakest scheduling that we can get away with.
(arbitrary $<$ FIFO $<$ Priority)

An Alarm Monitor

Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock, which invokes a procedure *tick* in your monitor at regular intervals.

```
monitor AlarmClock {
    int now = 0;
    condition  TimeToWakeup;
public:
    void SetAlarm (int period) {
      (* fill in the details here ... *)
    }
    void tick() {
        now++;
        TimeToWakeup.signal();
    }
}
```

An Alarm Monitor (solution)


```
monitor AlarmClock{
  int now = 0;
  condition TimeToWakeup;
  public:
  void  SetAlarm(int period){
    period = period + now; //at what time to wake up
    while (now < period) TimeToWakeup.wait(period);
    TimeToWakeup.signal();//Wake up other processes due now
                          //A cascaded wake up like the one
            //we used in the Readers and Writers problem
  }
  void tick(void) {
    now = now + 1;
    TimeToWakeup.signal();
  }
}
```

Note: The variable `now` and `period` may overflow (if your operating system is not rebooted for many decades!).

See synchronization-part2/monitors/java/alarm.monitor for a Java solution.

Writing Monitors with Pthreads library

- The POSIX threads (Pthreads) library provides locks, semaphores and condition variables. Combining these concepts makes it possible to write monitors in C/C++.
- The idea is to have a monitor class (or class like file in C). For example, we may have monitor.c and monitor.h. Associate a mutex lock with it. In each method of the class, we acquire the lock upon entering the method. We unlock the mutex before returning from each method.
- The condition variables in Pthreads library are the same as the condition variables used in the monitor concept.
- How to protect a class where public functions can call each other? Use a wrapper function for each public function. The wrapper function locks the mutex, calls the real function and then unlocks the mutex.

Account: A Monitor Example

- Account.h
- Account.c
- TestAccount.c

Monitors Support in POSIX threads

Mutexes and Semaphores have already been discussed previously. The following are the functions provided in Pthreads library for supporting condition variables.

```
pthread_cond_t condvar;
int pthread_cond_init(pthread_cond_t *restrict cond,
                const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(...);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

See man pages for more information.

Notes on `pthread_cond_wait` and `pthread_cond_timedwait`: These functions always have an associated mutex (the one that is being used to enforce mutual exclusion in the monitor). These functions atomically release the mutex and cause the calling thread to block on the condition variable. Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

Alarm Monitor in Pthreads

- AlarmClock.h
- AlarmClock.c
- TestAlarm.c

Producers Consumers in Pthreads

- A simple procuder consumer example that uses an array of buffers and conditional variables: producers-consumers.c
- How would you modify the producer-consumer example so that it stops cleanly on receiving SIGTERM, SIGINT or SIGHUP signals?
- SIGINT is generated by typing Ctrl-c on the keyboard. SIGTERM or SIGHUP can generated via the kill command.

```
kill -s TERM pid
kill -s HUP pid
```

where the process can be found with the ps augx command.

Creating Threads in Java (1)

Threads are part of the core Java language. There are two ways to create a new thread of execution.

- One is to the extend the class `java.lang.Thread`. This subclass should override the `run` method of class Thread. An instance of the subclass can then be allocated and started.

```
class Student extends Thread {
    public void run() {
        //listen, talk, daydream, fidget
        ...
    }
}
...
Student myStudent = new Student();
myStudent.start();
...
```

Creating Threads in Java (2)

- The second way to create a thread is to declare a class that implements the `Runnable` interface. That class then implements the `run` method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started.

```
class Student implements Runnable{
    public void run() {
        //listen, talk, daydream, fidget
        ...
    }
}
...
Student myStudent = new Student();
Thread myThread = new Thread(myStudent).start();
...
```

Java Synchronization (1)

- Java threads are preemptible and native. The programmer should not make any timing assumptions.
- Threads have priorities that can be changed (increased or decreased).
- This implies that multiple threads will have race conditions (read/write conflicts based on time of access) when they run. The programmer has to resolve these conflicts.
- Example of a race condition. See Account.java and TestAccount.java files.

Java Synchronization (2)

- Java has `synchronized` keyword for guaranteeing mutually exclusive access to a method or a block of code. Only one thread can be active among all synchronized methods and synchronized blocks of code in a class.

```
// Only one thread can execute the update method at
// a time in the class.
synchronized void update() { //... }
```

```
// Access to individual datum can also be synchronized.
// The object buffer can be used in several classes,
// implying synchronization across classes.

synchronized(buffer) {
    this.value = buffer.getValue();
    this.count = buffer.length();
}
```

- Every Java object has an implicit monitor associated with it to implement the synchronized keyword. Inner class has a separate monitor than the containing outer class.
- Java allows Rentrant Synchronization, that is, a thread can reacquire a lock it already owns. For example, a synchronized method can call another synchronized method.

Java Synchronization (3)

- The `wait()` and `notify()` methods (of the `Object` class) allow a thread to give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing.
- Another thread must call `notify()` for the waiting thread to wakeup. If there are other threads around, then there is no guarantee that the waiting thread gets the lock next. Starvation is a possibility. We can use an overloaded version of `wait()` that has a timeout.
- The method `notifyAll()` wakes up all waiting threads instead of just one waiting thread.

Example with `wait()/notify()`


```
class MyThing {
  synchronized void waiterMethod() {
    // do something and then wait
    // This gives up the lock and puts the calling
    // thread to sleep
    wait();
    // continue where we left off
  }

  synchronized void notifierMethod() {
    // do something
    // notifier the waiter that we've done it
    notify();
    //do more things
  }

  synchronized void relatedMethod() {
    // do some related stuff
  }
}
```

Writing Monitors with Java

- Java directly implements the notion of monitors with the synchronized keyword. Any Java class can be made into a monitor by following two rules.
  - Add the `synchronized` keyword in front of each method declaration.
  - Make sure that there are no directly accessible class variables. For example, make all class variables be private (which is the recommended Object-Oriented practice anyways).
- Every java object has a `wait()`, `notify()`, `notifyAll()` methods that correspond to monitor concepts.

Example 1: Bank Account

See examples in the folder: monitors/java/account

- Race conditions: Account.java, `TestAccount.java`
- Thread safe version using `synchronized` keyword: ThreadsafeAccount.java

Example 2: Producer/Consumer Problem

- A *producer* thread creates messages and places them into a queue, while a *consumer* thread reads them out and displays them.
- The queue has a maximum depth.
- The producer and consumer don't operate at the same speed. In the example, the producer creates messages every second but the consumer reads and displays only every two seconds.
  How long will it take for the queue to fill up? What will happen when it does?

Example 2 (contd.)

See examples in the folder:
`monitors/java/producers-consumers`

- Producer and Consumer sharing a synchronized queue.
- See examples: SharedQueue.java, Producer.java, Consumer.java, PC.java

Example 3: Alarm Monitor

- See example in the folder monitors/java/alarm.monitor

Condition variables support in MS Windows API

Condition variables were introduced in Vista and Windows Server 2008. Condition variables aren't supported in Windows Server 2003 and Windows XP/2000.

`CONDITION_VARIABLE condvar;`

`InitializeConditionVariable`: Initializes a condition variable.

`SleepConditionVariableCS`: Sleeps on the specified condition variable and releases the specified critical section as an atomic operation.

`SleepConditionVariableSRW`: Sleeps on the specified condition variable and releases the specified SRW lock as an atomic operation.

`WakeAllConditionVariable`: Wakes all threads waiting on the specified condition variable.

`WakeConditionVariable`: Wakes a single thread waiting on the specified condition variable.

CS stands for critical section. SRW stands for Slim Read Write locks.

`From the MSDN Library documentation.`

Condition variables example in MS Windows API


```
CRITICAL_SECTION CritSection;
CONDITION_VARIABLE ConditionVar;
void PerformOperationOnSharedData()
{
  EnterCriticalSection(&CritSection);
  // Wait until the predicate is TRUE
  while( TestPredicate() == FALSE ) {
    SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE)
  }
  // The data can be changed safely because we own the
  // critical section
  ChangeSharedData();
  LeaveCriticalSection(&CritSection);
  // If necessary, signal the condition variable by calling
  // WakeConditionVariable or WakeAllConditionVariable
  // so other threads can wake
}
```

From the MSDN Library documentation.

Producers Consumers in MS Windows API

- Code example: producers-consumers.c

Inter-Process Communication (IPC)

- Semaphores and Signals are useful for synchronization but not for sending information among processes.
- Monitors allow communication using *shared memory*.
- Message passing allows communication without using shared memory. The message passing mechanism copies the information from the address space of one process to the address space of another process.
- What's a message? How do we send a message? Why is the operating system needed as an intermediary to send messages? (for independent processes, since for threads we can always use shared memory).

Message Passing Using Mailboxes

- Mailboxes in user space versus in system space.
- Message protocols. A message may contain an instance of a C structure, may be a string of ASCII characters etc. The format is between the sender and receiver to decide a priori or negotiate.
- Message headers may contain information such as the sending process id, the receiver process id, number of bytes in the message, message type etc. Messages may not even have a header depending on how they are implemented.
- Performance issues. Copy-on-write mechanism can be used to improve the performance.

Message Passing Primitives

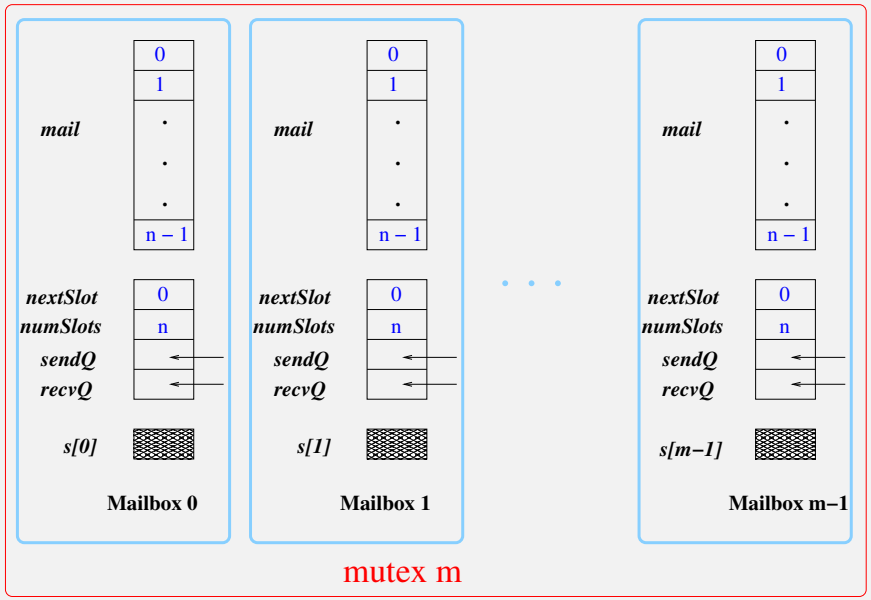
- `send()`.
  - *synchronous send*. Blocks the sending process until the message is received by the destination process. This synchronization is an example of a producer-consumer problem.
  - *asynchronous send*. The message is delivered to the receiver's mailbox and then the sender is allowed to continue without waiting for the receiver to read the message. The receiver doesn't even have to ever retrieve the message.
- `receive()`.
  - *blocking receive*. If there is no message in the mailbox, the process suspends until a message in placed in the mailbox. A receive operation is analogous to a resource request.
  - *non-blocking receive*. The process is allowed to query the mailbox and then control is returned immediately either with a message if there is one in the mailbox or with a status that no message is available.

Message passing and Semaphores

- How to simulate a semaphore using message passing?
- How to simulate a monitor using message passing?
- How to implement message passing using mailboxes (with all processes having access to shared memory) and semaphores?
- How to implement message passing using mailboxes (with all processes having access to shared memory) and monitors?

Message Passing using Semaphores

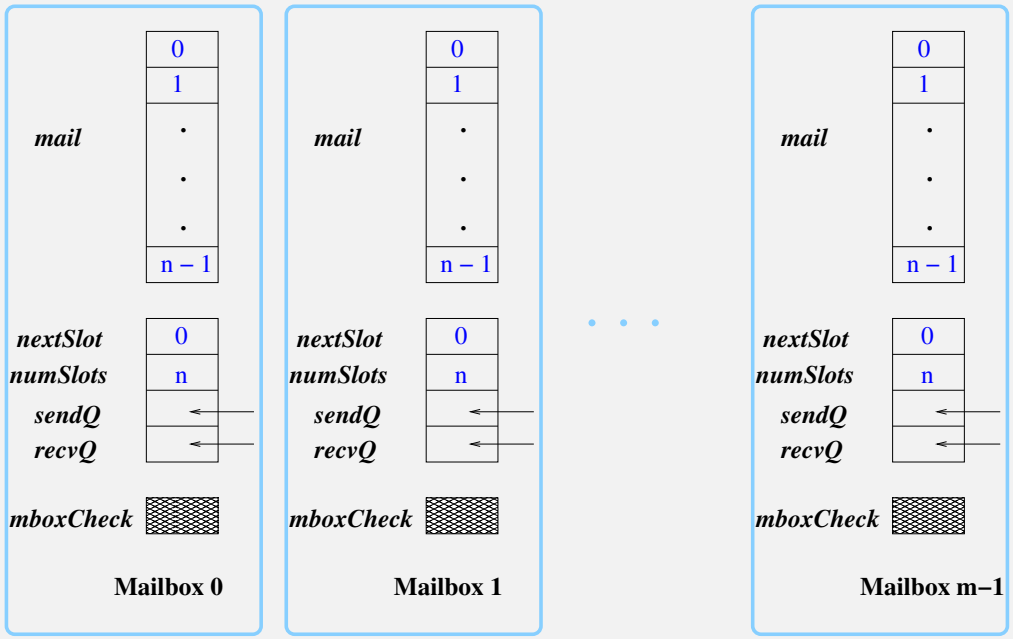
- A mailbox area in shared memory will be used to hold the mailboxes. Each mailbox contains an array of message slots, named *mail*. Each mailbox has variables *numSlots* and *nextSlot*, to keep track of the number of slots and the next available slot. It also has two queues for waiting threads.
- Each process has an associated semaphore s_i (initially zero) on which it will block when a send/receive must block.
- A global *mutex* is used to ensure mutual exclusion in accessing the mailbox area in shared memory to prevent race conditions.



Based on the above, now sketch out the pesudo-code for send/receive.
How can we improve the performance of the above implementation?

Message Passing using Monitors

- Each *mailbox* is managed by a monitor. Each mailbox contains an array of message slots, named *mail*. Each mailbox has variables *numSlots* and *nextSlot*, to keep track of the number of slots and the next available slot. It also has two queues for waiting threads.
- Each process has a condition variable (say *mboxCheck*) on which it will block when a send/receive must block.



Based on the above, sketch out the pesudo-code for send/receive methods.

Synchronization in the Linux kernel

- Atomic operations on bits, 32-bit and 64-bit integers. See include/asm-generic/atomic.h and arch/x86/include/asm/atomic.h for architecture specific implementation. See arch/x86/include/asm/bitops/atomic.h for test-and-set related bit operations.
- Spinlocks. Threads do not sleep if waiting for a lock. Suitable only for short duration locks.
- Reader-Writer spinlocks. Gives preference to readers over writers. Multiple readers can hold the lock but only one writer..
- Semaphores. See include/linux/semaphore.h and kernel/locking/semaphore.c. Uses wait queues and sleep.
- Reader-Writer Semaphores.
- Mutexes. Similar to a binary semaphore but with a simpler interface, more efficient performance, and additional constraints on its use.
- Completion variables. Similar to condition variables.
- Sequential locks. Reader and writers with preference given to writers.
- Read-Write barriers and ordering.
- BKL: Big Kernel Lock. Removed in kernel version 2.6.39.
- Preemption disabling. Bad thing :-(