NTNU – Trondheim
Norwegian University of
Science and Technology

TTK4145 – Real-time Programming

Lecture 13 – Synchronization Mechanisms

Example exam questions

- From 2010:

  Java's wait and notify (and POSIX condition variables) works in many ways similar to `suspend()` and `resume()`. How can these be seen as better or more high-level synchronization mechanisms?

- From 2011:

  We have a resource in our system that is used by many threads, creating the need for synchronizing access. When there are more waiting threads the last request should be given priority. Write pseudo code for the `allocate()` and `free()` functions that achieves this. Use the synchronization mechanisms in C/POSIX, Ada or Java as you prefer.

Learning goals:
Shared Variable Synchronization

- Ability to create (error free) multi thread programs with shared variable synchronization.

- Thorough understanding of pitfalls, patterns, and standard applications of shared variable synchronization.

- Understanding of synchronization mechanisms in the context of the kernel/HW.

- Ability to correctly use the synchronization mechanisms in POSIX, ADA (incl. knowledge of requeue and entry families) and Java.

Standard problems in shared variable synchronization

- Deadlock
  - The system is locked in a circular wait
  - Example: waiting for mutual resources, or waiting for lost message

- Livelock
  - The system is locked in a subset of states
  - More general form of deadlock, but in a sense the opposite
  - Harder to diagnose than deadlock since the program is doing work
  - Example: sending messages requiring acks,

- Starvation
  - A thread does “by accident” not get necessary resources
  - A resource could be a semaphore, or more abstract such as CPU time
  - Leads to the thread getting to run less than it needs to
  - Could be caused by bad luck, unfair scheduling, or timing cycles where the same thread ends up last in the priority queue every time
  - Example: multiple threads requiring dynamic memory from the heap. One thread uses a lot of memory over a short time, multiple other threads require small amounts of memory all the time. Might never be enough available memory for the first thread to run

- Race conditions
  - A bug that surface by unfortunate timing or order of events

Find the race condition

```
 1 void allocate(int priority){
 2   Wait(M);
 3   if(busy){
 4       Signal(M);
 5       Wait(PS[priority]);
 6   }
 7   busy=true;
 8   Signal(M);
 9 }
10
11 void deallocate(){
12   Wait(M);
13   busy=false;
14   waiting=GetValue(PS[1]);
15   if(waiting>0) Signal(PS[1]);
16   else{
17       waiting=GetValue(PS[0]);
18       if(waiting>0) Signal(PS[0]);
19       else{
20           Signal(M);
21       }
22   }
23 }
```

- The code (adapted from Burns & Wellings page 154) implements a resource allocation scheme where multiple threads wish to access a resource with priority 0 or 1
- If the resource is busy, the thread will be blocked at a semaphore in the array `PS` according to its priority
- The race condition occurs if the second to last user of the resource calls `deallocate()` and preempts the last user trying to allocate() right after `Signal(M)` in line 4
- The last user will then be blocked until another user calls `allocate()` and then `deallocate()`

Semaphores

- The most basic synchronization primitive that works
  - Avoids most preemption problems associated with `if` statements, spin locks, and `suspend()` and `resume()`

- Solves the mutual exclusion problems such as
  - Conditional synchronization
    - Last lecture implemented a Barrier with semaphores as turnstiles
    - `suspend()` and `resume()` could not solve this, lead to race conditions

  - Basic resource allocation
    - Easy to lock resources to prevent simultaneous access to shared memory between threads
    - Can also solve this with spin locks, but difficult to avoid race conditions
    - Open problem if this can be solved with `suspend()` and `resume()`

`signal()` – which thread will be awakened?

- Which strategy is the kernel using for deciding which thread to awaken?
  - *First In First Out* (FIFO), the one that has waited the longest gets woken up
  - Why not most important thread or highest priority thread get the semaphore?
  - FIFO strategy avoids starvation, and one of the kernels main tasks is to help the programmer minimize starvation

- From a Code Quality perspective, don't take advantage of knowing the schedulers strategy while coding
  - If you do, someone can't easily read your code and verify correctness without knowing the details of the underlying operating system
  - This is the reason that the semaphore documentation specifies that which thread being awakened on `signal()` is arbitrary
  - You shouldn't be depended on scheduling strategy, solve the problem yourself

Bounded buffer

- Also known as the *producer & consumer* problem

- A shared buffer is being acted upon by two or more threads that either adds (*produces*) elements or removes (*consumes*) elements from the buffer
  - There needs to be at least 1 *producer* and 1 *consumer*

- The bounded buffer has the following interface
  - `put(e)`**:** adds the element `e` to the buffer, if the buffer is full the calling thread (*producer*) should be suspended until a *consumer* has removed an element from the buffer
  - `e get()`**:** returns an element `e` from the buffer to the calling thread, if the buffer is empty the calling thread (*consumer*) should be suspended until another element has been added by a *producer* to the buffer

- Challenge to the class
  - How would you implement a bounded buffer using semaphores?
  - Hint: Use one semaphore for each reason for blocking a thread

Bounded buffer with semaphores

```
SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);

put(e){
    wait(NFree);
    wait(Mutex);
        // enter into buffer
    signal(Mutex);
    signal(NInBuffer);
}

get(e){
    wait(NInBuffer);
    wait(Mutex);
        //get e from buffer
    signal(Mutex);
    signal(NFree);
}
```

Drawbacks of semaphores

- Breaks module boundaries
  - The semaphore is an object in the operating system, a kind of global type object
  - Need to examine all parts of our program that might use our semaphore to ensure correctness

```
t1(){                    t2(){                                                   t3(){
      wait(A)                  wait(A)                                                 wait(A)
      i = i + 1                i = i - 1                                               //doWork
      signal(A)}               signal(A)}                                        }
```

  - Good programmers will reduce the scope of a semaphore, still no guarantee for no misuse

- Semaphore solutions invites deadlocks
  - Can get deadlocks with only 2 semaphores

```
t1(){                    t2(){
      wait(A)                  wait(B)
      wait(B)                  wait(A)
      //doWork                 //doWork
      signal(B)                signal(A)
      signal(A)}               signal(B)}
```

  - Need to examine all states the program can be in to ensure correctness
  - Deadlock analysis is a global analysis, lacks scalability

- Cannot release locks “temporarily”
  - Might want the possibility to “lend” other threads our locked resources while we’re blocked waiting for other resources
  - Difficult to solve with only semaphores, recall the *Allocating A, B or both* example from last lecture

Difficult synchronization challenges

- Most of the examples from *The Little Book of Semaphores* are difficult to solve

- A lot of previous exam tasks are modified versions of the problems described in the book
  - Especially tasks where you are asked to implement the solution using higher order synchronization mechanisms from Ada or Java

- Difficult problems we have encountered
  - The Barrier: Easy with Ada, difficult with semaphores
  - Flexible allocation of resources: *Allocating A, B or both*
  - Priority based resource allocation: The example from earlier

Conditional Critical Region (CCR)

- Semaphores give the ability to provide mutual exclusion over a critical section
  - Conditional Critical Regions is a more structured synchronization primitive that provides this mutual exclusion directly

- A Conditional Critical Region is a named section of code that is guaranteed to be executed in mutual exclusion

- Additionally regions of the same name are mutually exclusive

- Guards provide the conditional synchronization on the region

- Conditional Critical Regions are not supported directly by any mainstream languages
  - (Imagined) language support for mutual exclusion
  - Protected Objects in Ada provides much of the same functionality

Bounded buffer with CCR

```
task producer;
   loop
      region buf when buffer.size < N do
         -- place char in buffer etc
      end region
   end loop;
end producer

task consumer;
   loop
      region buf when buffer.size > 0 do
         -- take char from buffer etc
      end region
   end loop;
end consumer
```

Drawbacks of Conditional Critical Regions

- CCR are more structured than utilizing semaphores directly, but still not proper modules since they are still dispersed throughout the code

- There are performance issues connected to re-evaluation of guards
  - Every time a thread leaves a CCR containing the resource being tested by the guard, the guard needs to be re-evaluated
  - A suspended task must become executable again, and retest the guard. If still false, must return to suspended state
  - Leads to big system overheads, and wasted context switches

Monitors

- Monitors provide more structured control regions compared to CCR
  - And provide more efficient implementations of condition synchronization
- The intended critical region is a collection of procedures and local variables encapsulated together in a single module called a monitor
- All procedure calls to the monitor are guaranteed to execute under mutual exclusion
  - Each thread must implicitly acquire the monitor lock when executing in the monitor
- Condition variables are used as a synchronization primitive instead of guards
  - Condition variables are acted upon by the operators `wait` and `signal`
    - `wait`: blocks the thread and places it in a queue associated with that condition variable (similar to suspend)
    - `signal`: awakens *one* waiting thread from the associated queue
  - Demonstrated unsafe last lecture, why is it safe now?
    - Code runs inside environment already protected from interruption
- What happens when a thread is blocked on a condition variable?
  - The thread "temporarily" releases the monitor lock
  - Other procedures can now run while we wait for condition variable
  - All procedures need the monitor lock to run

Bounded buffers with monitors

```
procedure put(e);
begin
    if NumberInBuffer = size then
        wait(spaceavailable);
    end if;
    // Enter e into buffer
    signal(itemavailable)
end put;

Element procedure get();
begin
    if NumberInBuffer = 0 then
        wait(itemavailable);
    end if;
    // get an element from the buffer and return it.
    signal(spaceavailable);
end get;
```

Monitor race conditions

- Monitors can still be affected by race conditions
- Another thread might change a variable `i` we're working on while we're waiting on a condition variable `C`

```
procedure increment_i(){
    t = i
    wait(C)
    i = t+1
}
```

```
procedure set_t(){
    t = 10
}
```

- Usually not a problem, obvious where to put `wait` to avoid these problems

Nested monitors

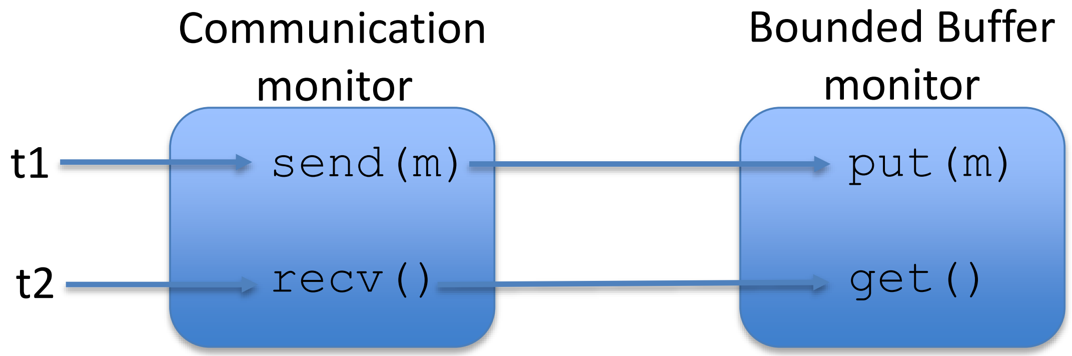
- Consider a Communication monitor that performs inter-process communication via a Bounded Buffer implemented as a monitor

- The Communication monitor has the following procedures calls
  - `send(m)`: calls the `put(m)` procedure in the Bounded Buffer monitor to store the message in the communication queue
  - `recv()`: calls the `get()` procedure in the Bounded Buffer monitor to retrieve the next message from the communication queue

- What happens when the Bounded Buffer is full and the `put(m)` procedure gets suspended?
  - The Bounded Buffer monitor lock is released

- What happens with the Communication monitor lock, should it be released?
  - No, if we release the lock, we release outside of expected waits.
  - Need explicit control of where interleaving can occur to avoid race conditions

- This leads to deadlock, `t2()` can't get Communication monitor lock to get message from buffer
  - It is possible to write code with nested monitors, but you must be very careful



Drawbacks with Monitors

- Monitors are the state of the art of synchronization primitive
  - Examples from *The Little Book of Semaphores* are much easier solved by monitors

- Still they have several drawbacks
  - Destroys composition
    - Impossible to make supermodules out of submodules
    - At least very difficult to do right
  - Deadlocks and race conditions still present
    - Easier to implement for larger systems than semaphores, but go far enough and you encounter similar problems
  - Deadlocks require global analyses
    - Example: Nested monitors
  - More complex than semaphores

Synchronization mechanisms in C/Real-Time POSIX

- Mutexes and condition variables provides the functionality of a monitor, but with a procedural interface

- Each monitor has an associated `mutex`, and all operations on the monitor are surrounded by calls to lock and unlock the mutex

- Condition synchronization is provided by associating condition variables with the monitor `mutex`
  - While waiting on the condition variable, the specified monitor `mutex` can be temporarily released
  - The thread will regain the lock once it's awakened

- Contrary to the general monitor design, signal() awakens all waiting threads
  - Each thread must again test on the condition that caused it to wait()
  - Can be achieved with `while`-loops

Bounded buffers with POSIX

```
void put( int e ) {
    pthread_mutex_lock( &mutex );
    while ( numberInBuffer == size )
        pthread_cond_wait( &spaceAvailable, &mutex );
    // do work
    pthread_cond_signal( &itemAvailable );
    pthread_mutex_unlock( &mutex );
}

int get() {
    pthread_mutex_lock( &mutex );
    while ( numberInBuffer == 0 )
        pthread_cond_wait( &itemAvailable, &mutex );
    // do work
    pthread_cond_signal( &spaceAvailable );
    pthread_mutex_unlock( &mutex );
    return /* ... */;
}
```

Synchronization mechanisms in Java

- Every object can be implemented as a monitor

- They are implicitly declared with the keyword `synchronized` in front of any method
  - Any such methods will be called with mutual exclusion
  - Implemented as one mutex owned by all `synchronized` methods in the object

- `Wait`()
  - Can be called from inside `synchronized` method
  - Suspends the current accessing thread
  - Other threads can then access the same/other `synchronized` methods

- `Notfiy`()/`notifyAll`()
  - Awakens one thread from `Wait`, allowing it to resume
  - `notifyAll()` awakens all waiting threads, still must wait for lock in order
  - Threads may also be awakened by other threads calling `this.interrupt()`

- Objects may have a mix of `synchronized` and regular methods
  - May break mutual exclusion if variables in critical sections are manipulated in regular methods
  - The main rule is not to mix these different method types

- The lock associated with an object can also be manipulated using `synchronized(this){ }`
  - Any object can be used as a mutex this way
  - Can lead to deadlocks and race conditions if you're mixing internal and external locking
  - Avoid this by using empty objects as mutexes this way, and not monitors

Bounded Buffer in Java

```
public synchronized void put(int item) throws InterruptedException
{
    while (numberInBuffer == size) wait();
    last = (last + 1) % size ;
    numberInBuffer++;
    buffer[last] = item;
    notifyAll();
}

public synchronized int get() throws InterruptedException
{
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size ;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
}
```

Java discussion points

- `Notfiy`() vs `notifyAll`()
  - Why do we use `notifyAll` and while loops around wait?
  - “Conditions” are nameless in Java, if there are more conditions for blocking, we would not know that notify wakes the right one
  - Advice: always use `notifyAll` and `while` loops around the `wait()`
  - Also better from a Code Quality perspective, it makes it explicit what we are waiting for (the test within the `while` loop)

- Inheritance anomaly
  - The fact that monitors are full fledged objects, means that they can be used for inheritance
  - Can lead to strange race conditions due to the problem with overloading `synchronized` methods

Inheritance Anomaly example (page 178 in B&W)

```
public class LockableBoundedBuffer extends BoundedBuffer {
	boolean prohibited;

	// Incorrect code

	LockableBoundedBuffer(int length) {
		super(length);
		prohibited = false;
	}

	public synchronized void prohibitAccess() throws Interruptedexception {
		while(prohibited) wait();
		prohibited = true;
	}

	public synchronized void allowAccess() throws AccessError {
		if(!prohibited) throw new AccessError();
		prohibited = false;
		notifyAll();
	}

	public synchronized void put(int item) throws Interruptedexception {
		while(prohibited) wait();
		super.put(item);
	}


	public synchronized int get() throws Interruptedexception {
		while(prohibited) wait();
		return(super.get());
	}
}
```