NTNU – Trondheim
Norwegian University of
Science and Technology

TTK4145 – Real-time Programming

Lecture 14 –

Synchronization Mechanisms continued & Deadlocks

Example exam questions

- 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.

- From 2015:

  Define the terms deadlock and race condition

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.

Introduction

- Todays lecture will continue exploring synchronization mechanisms in Java and Ada

- The focus will be on specific example to show how to utilize different design patterns

- It is expected that you should be able to implement and discuss aspects of similar problems in the exam

- Let’s start with a familiar problem

Reusable barrier with semaphores

```
//rendezvous
mutex.wait()
      count += 1
      if count == n:
            turnstile2.wait() // lock the second
            turnstile.signal() // unlock the first
mutex.signal()

turnstile.wait() // first turnstile
turnstile.signal()

//critical point
//…

mutex.wait()
      count -= 1
      if count == 0:
            turnstile.wait() // lock the first
            turnstile2.signal() // unlock the second
mutex.signal()

turnstile2.wait() // second turnstile
turnstile2.signal()
```

Barrier in Java

- Implementing a Barrier with Semaphores is complicated and error-prone

- The solution required inventing a new design pattern, the turnstile
  - Only one thread can pass through at a time
  - The previous thread signals the next thread

- How can we implement a similar barrier in Java that synchronizes three threads?

One-time barrier in Java

```
void synchronized role1(){
     //doWork

     n_finished++;
     while(n_finished  != N) wait();
     notifyall();

}

void synchronized role2(){
…
}

void synchronized role3(){
…
}
```

Reusable barrier in Java

```
void synchronized role1(){
     //doWork

     n_finished++;
     while(n_finished  != N) wait();
     notifyall();

     release--;
     if(release == 0){
          n_finished = 0;
          release = 3;
}

void synchronized role2(){
…
}

void synchronized role3(){
…
}
```

New example: Read/Write locks

- Mutual exclusion is necessary when several threads wish to make changes to a data structure simultaneous
- However, tasks that only read the value of the data structure have no risk of interfering with each other
- For efficiency, it’s desirable to design a synchronization pattern that allows for mutual exclusion when a thread writes to a variable, but unrestricted access when only reading occurs
- The problem can be solved with semaphores, and is featured in The Little Book of Semaphores

Read/Write locks with semaphores

- Hint:

```
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

- Writer’s solution:

```
roomEmpty.wait()
     //critical section for writers
roomEmpty.signal()
```

- Readers solution:

```
mutex.wait()
     readers += 1
     if readers == 1:
          roomEmpty.wait() // first in locks
mutex.signal()

// critical section for readers

mutex.wait()
     readers -= 1
     if readers == 0:
           roomEmpty.signal() //last out unlocks
mutex.signal()
```

The Lightswitch pattern

- The example introduces a new useful semaphore pattern
- The Lightswitch pattern is used when you have multiple threads accessing a resource, and you wish to exclude other accesses to the resources
- The lock (the light is on) is held until the last thread leaves the critical section, at which point the lock is released (the light is turned off)
- This pattern can easily be turned into a module that accepts a semaphore to lock/unlock as an argument

No-starve Read/Write locks with semaphores

- Hint:

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```

- No-starve writer solution:

```
turnstile.wait()
    roomEmpty.wait()
    // critical section for writers
turnstile.signal()
roomEmpty.signal()
```

- No-starve reader solution:

```
Turnstile.wait()
Turnstile.signal()

readSwitch.lock(roomEmpty)
    // critical section for readers
readSwitch.unlock(roomEmpty)
```

Read/Write locks in Java

- We wish to implement a similar mechanism in Java using a Monitor
- Threads that wish to read/write to the resource needs to call the following functions before/after accessing

```
startRead()
stopRead()
startWrite()
stopWrite()
```

- How can you implement the Read/Write lock in Java such that writers have priority over readers?

Read/Write locks in Java - I

```
public synchronized void StartWrite()
    throws InterruptedException
{
    while(readers > 0 || writing)
    {
        waitingWriters++;
        wait();
        waitingWriters--;
    }
    writing = true;
}

public synchronized void StopWrite()
{
    writing = false;
    notifyAll();
}
```

Read/Write locks in Java - II

```
public synchronized void StartRead()
      throws InterruptedException
{
   while(writing || waitingWriters > 0) wait();
   readers++;
}

public synchronized void StopRead()
{
   readers--;
   if(readers == 0) notifyAll();
}
```

Synchronization mechanisms in Ada

- Ada’s synchronization mechanisms are inspired by the concept of Conditional Critical Regions

- The main synchronization is provided through Protected Objects:
  - Modules
  - (Private) variables
  - Functions
    - Only reads variables, doesn’t change them,
    - Implemented as read locks
  - Procedures
    - Changes variables
    - Implemented as write locks
  - Entries with Guards
    - Criteria for execution in addition to mutual exclusion
  - Mutex is implicit as with Monitors
    - Only 1 thread can execute at the same time

Bounded Buffer in Ada

```
protected body Bounded_Buffer is

    entry Get (Item : out Data_Item) when Num /= 0 is
    begin
        Item := Buf(First);
        First := First + 1;
        Num := Num - 1;
    end Get;

    entry Put (Item : in Data_Item) when Num /= Buffer_Size is
    begin
        Last := Last + 1;
        Num := Num + 1;
        Buf(Last) := Item
    end Put;

end Bounded_Buffer;
```

The Bounded Buffer type

```
Buffer_Size : constant Integer :=10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
   entry Get (Item : out Data_Item);
   entry Put (Item : in Data_Item);
private
   First : Index := Index'First;
   Last : Index := Index'Last;
   Num : Count := 0;
   Buf : Buffer;
end Bounded_Buffer;
```

Read-Write locks in Ada

```
protected body Shared_Data_Item is
   function Read return Data_Item is
   begin
      return The_Data;
   end Read;

   procedure Write (New_Value : in Data_Item) is
   begin
      The_Data := New_Value;
   end Write;

end Shared_Data_Item;
```

Ada challenge

- We have seen the Barrier problem solved with semaphores and in Java
- How would you solve the problem using Ada?
- Hint: Does Ada provide any mechanism for checking how many threads are waiting at a certain point?

One-time barrier in Ada

```
protected body Blocker is
    entry Proceed when Proceed'Count = 5 or Release is
    begin
        Release := True;
    end Proceed;
end Blocker;
```

Reusable barrier in Ada

```
protected body Blocker is
    entry Proceed when Proceed'Count = 5 or Release is
    begin
        if Proceed'Count = 0 then
            Release := False;
        else
            Release := True;
        end if;
    end Proceed;
end Blocker;
```

New example: Update/Modify/Lock/Unlock

- Consider a data structure that the user can
  - Update: Change the value of the variables
  - Modify: Change the data structure itself

- Update should take priority over modify

- Additionally, it should be possible to lock/unlock the data structure from being modified, such that calling threads need to wait until the structure is unlocked

- How would you implement this in Java?

Update/Modify/Lock in Java

```
synchronized Modify(...){
     while(locked || NWaitingUpdaters >0) wait();
          ...
     notifyAll();
}

synchronized Update(...){
     while(locked){
          NWaitingUpdaters++;
          wait();
          NWaitingUpdaters--;
     }
     ...
     notifyAll();
}

synchronized lock(){
     locked = true;
}

synchronized unlock(){
     locked = false;
     notifyAll();
}
```

And in Ada?

Update/Modify/Lock in Ada - I

```
protected Resource_Manager is
   entry Update(...);
   entry Modify(...);
   procedure Lock;
   procedure Unlock;
private
   Manager_Locked : Boolean := False;
   ...
end resource_manager;
```

Update/Modify/Lock in Ada - II

```
protected body Resource_Manager is

     entry Update(...) when not Manager_Locked is
     begin
          ...
     end Update;

     entry Modify(...) when not Manager_Locked and Update'Count = 0 is
     begin
          ...
     end Modify;

     procedure Lock is
     begin
          Manager_Locked := True;
     end Lock;

     procedure Unlock is
     begin
          Manager_Locked := False;
     end Unlock;

end Resource_Manager;
```

Nested Protected Objects

- Monitors in Java do not scale well since blocking inside a submonitor does not release the lock from the supermonitor
  - Can cause deadlocks
- Ada solved this by saying it is impossible, if you block from inside a protected object it is an error
  - Blocking from the inside throws an exception
- Waiting on inner monitor is not blocking!
  - Long procedure/entry calls is an example
  - But waiting for a guard is!

Some issues with entries

- Guards can test only on private variables and not parameters or global variables
  - To ensure efficient testing on guard variables, recall the issues with Conditional Critical Regions
  - The guards are only evaluated on exit from procedures/entries that might have changed the variable
  - An example that demonstrates the issues this can cause is the problem of allocating memory, the guards need to test if the amount of memory available (private variable) is more than the requested memory (parameter)

- Ada provides mechanisms that can be used to (among other things) work around this problem
  - Requeue: Allows for an entry body to complete while redirecting the corresponding entry call to a new (or the same) entry queue
  - Entry families: An array of entries that can be indexed into on entry calls, allows for delegation of work

Handling request parameters in Java

```
public class ResourceManager
{
    private final int maxResources = ...;
    private int resourcesFree;

    public ResourceManager() { resourcesFree = maxResources; }

    public synchronized void allocate(int size)
    {
        while(size > resourcesFree) wait();
        resourcesFree = resourcesFree - size;
    }

    public synchronized void free(int size)
    {
        resourcesFree = resourcesFree + size;
        notifyAll();
    }
}
```

Request parameters in Ada with Requeue - I

```
type Request_Range is range 1 .. Max;
type Resource ...;
type Resources is array(Request_Range range <>) of Resource;

protected Resource_Controller is
   entry Request(R : out Resources; Amount: Request_Range);
   procedure Release(R : Resources; Amount: Request_Range);
private
   entry Assign(R : out Resources; Amount: Request_Range);
   Free : Request_Range := Request_Range'Last;
   New_Resources_Released : Boolean := False;
   To_Try : Natural := 0;
end Resource_Controller;
```

Request parameters in Ada with Requeue - II

```
protected body Resource_Controller is

    entry Request(R : out Resources; Amount: Request_Range) when Free > 0 is
    begin
        if Amount <= Free then
            Free := Free - Amount;
            -- allocate
        else requeue Assign;
        end if;
    end Request;

    procedure Release(R : Resources; Amount: Request_Range) is
    begin
        Free := Free + Amount;
        -- free resources
        if Assign'Count > 0 then
            To_Try := Assign'Count;
            New_Resources_Released := True;
        end if;
    end Release;
```

Request parameters in Ada with Requeue - III

```
    entry Assign(R : out Resources; Amount: Request_Range)
        when New_Resources_Released is
    begin
        To_Try := To_Try - 1;
        if To_Try = 0 then
            New_Resources_Released := False;
        end if;
        if Amount <= Free then
            Free := Free - Amount;
            -- allocate
        else
            requeue Assign;
        end if;
    end Assign;
end Resource_Controller;
```

Request parameters in Ada with Entry Families - I

```
package Resource_Manager is
    Max_Resources : constant Integer := 100;
    type Resource_Range is new Integer range 1..Max_Resources;
    subtype Instances_Of_Resource is Resource_Range range 1..50;

    procedure Allocate(Size : Instances_Of_Resource);
    procedure Free(Size : Instances_Of_Resource);
end Resource_Manager;
```

Request parameters in Ada with Entry Families - II

```
package body Resource_Manager is

    task Manager is
        entry Sign_In(Size : Instances_Of_Resource);
        entry Allocate(Instances_Of_Resource); -- family
        entry Free(Size : Instances_Of_Resource);
    end Manager;

    procedure Allocate(Size : Instances_Of_Resource) is
    begin
        Manager.Sign_In(Size); -- size is a parameter
        Manager.Allocate(Size); -- size is an index
    end Allocate;

    procedure Free(Size : Instances_Of_Resource) is
    begin
        Manager.Free(Size);
    end Free;
```

Request parameters in Ada with Entry Families - III

```
task body Manager is
    Pending : array(Instances_Of_Resource) of Natural := (others => 0);
    Resource_Free : Resource_Range := Max_Resources;
    Allocated : Boolean;
begin
    loop
        select -- wait for first request
            accept Sign_In(Size : Instances_Of_Resource) do
                Pending(Size) := Pending(Size) + 1;
            end Sign_In;
        or
            accept Free(Size : Instances_Of_Resource) do
                resource_free := resource_free + size;
            end Free;
        end select;
```

Request parameters in Ada with Entry Families - IV

```
loop -- main loop
    loop
        -- accept any pending sign-in/frees, do not wait
        select
            accept Sign_In(Size : Instances_Of_Resource) do
                Pending(Size) := Pending(Size) + 1;
            end Sign_In;
        or
            accept Free(Size : Instances_Of_Resource) do
                Resource_Free := Resource_Free + Size;
            end Free;
        else
            exit;
        end select;
    end loop;
```

Request parameters in Ada with Entry Families - V

```
            -- now service largest request
               Allocated := False;
               for Request in reverse Instances_Of_Resource loop
                  if Pending(Request) > 0 and Resource_Free >= Request then
                     accept Allocate(Request);
                        Pending(Request) := Pending(Request) - 1;
                        Resource_Free := Resource_Free - Request;
                        Allocated := True;
                        exit; --loop to accept new sign-ins
                  end if;
               end loop;
               exit when not Allocated;
            end loop;
         end loop;
      end Manager;
end Resource_Manager;
```

Order of Requests in Java

- A pattern for defining custom priority queues for allocation of resources
  - Example uses LIFO priority

```
synchronous allocate(){
    if(busy){
        queue.insertFirst(myThreadId); //LIFO
        while(busy || queue.getFirst() != myThreadId) wait();
        queue.removeFirst();
    }
    busy = true;
    // notifyAll() //? If more can be allocated at the same time.
}

synchronous free(){
    busy = false;
    notifyAll();
}
```

Deadlocks

The most common issue with shared variable synchronization

4 necessary conditions for deadlocks

1. Mutual Exclusion
   – Resources are associated with locks to prevent simultaneous access

2. Hold & Wait
   – Threads maintain their locks on resources while waiting for other resources

3. No Preemption of resources
   – Allocated resources can not be stolen by other threads
   – Locks are only released by the thread holding the resource

4. Circular Wait
   – Threads must be waiting for each other

- These conditions are necessary, but not sufficient

Deadlock prevention

- Deadlock prevention ensures deadlock can not occur by eliminating at least one of the necessary conditions as follows:

1. Optimistic Concurrency Control
   - Assume the probability of simultaneous access is low, and avoid using locks
   - Check after each operation on the resource if its state is consistent and fix any issues

2. Allocate all resources at once
   - Threads allocate all needed resources in a single operation
   - Difficult with semaphores, recall *Allocate A,B or both* example
   - Drawback: Need to know all resources that are required a priori

3. Preemption
   - Allow threads to interrupts each other if a deadlock is detected
   - Threads that are interrupted releases their allocated resources

4. Global allocation order
   - Always allocate resources in a specified order
   - If all threads are required to allocate resource A before B, deadlocks can not occur due to cross reservation of resources
   - Drawbacks:
     - Need to maintain resource locks longer than necessary, maybe affecting concurrency
     - Need to know all resources that are required a priori

Deadlock Avoidance

- Prevention algorithms often have poor resource utilization
- An alternative is to instead of preventing deadlocks all together, avoid situations that might lead to deadlock
- The most common approaches are
  1. Scheduling (Priority Ceiling)
     - Part of the lectures next week
  2. Resource Allocation ("Bankers algorithm")
     - An algorithm for resource allocation and deadlock avoidance
     - Tests all requests made by processes for resources, and checks if the allocation will lead the system to a safe state (non-deadlock)
     - Drawback: Needs to know how much of each resource a process could possibly request

Deadlock Detection

- Previous algorithms require a priori knowledge of resource use in the system
- Instead, treat deadlock as a general software fault, solve it with Fault Tolerance
- First step is detecting the deadlock by either:
  1. Timeout/Watchdog
     - Create a timer for each thread that indicates a reasonable deadline for completing a task
     - If the watchdog times out, abort the thread and free its allocated resources
  2. Analyzing waiters
     - All resource allocations are registered in memory
     - Can have a dedicated module that reads this memory and controls all locks, a Lock Manager
     - Determines which threads owns which resources, and which threads are waiting on what resources
     - If a Circular Wait is detected, abort the threads and free their allocated resources

Deadlock Recovery

- Recovery can be initiated after detection, utilizing the concepts from Fault Tolerance:

1. Break mutual exclusion (forward error recovery)
   - Release all the locks and allow unrestricted access to the shared variables causing the deadlock
   - Can lead to an inconsistent system, the reason we have mutual exclusion in the first place!
   - Handle the aftermath like Optimistic Concurrency Control, check for inconsistent variables and fix the problem

2. Preemption using abort/kill (backwards error recovery)
   - Handle it like a general software fault, abort/kill the tasks that are deadlocked
   - Make sure to reset data structures back to recovery points and release all held locks
   - Remember fairness, could cause starvation if you always kill the same client