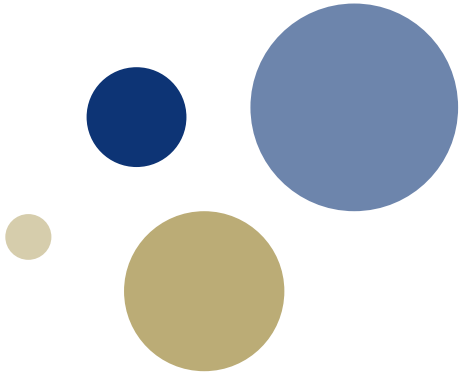
NTNU – Trondheim
Norwegian University of
Science and Technology



TTK4145 – Real-time Programming

Lecture 12 – Race conditions and Real-Time Kernels

Example exam questions

- From 2009:

```
if(getWaiting(A) == 2){  // GetWaiting returns the number of
                         // processes waiting for the semaphore
    // The two others are here already - release and continue.
    signal(A);
    signal(A);
}else{
    // The number of waiting threads is less than two - must wait.
    wait(A)
}
```

This code has a race condition. What is a race condition? What is the problem with the code?

- From 2010:

  Mention some hardware (/assembly) mechanisms that are used for achieving basic synchronization.

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.

Motivation

- In Real-time programming there is a number of timing demands on the system, how do we solve this?
  - Divide the system into threads
  - Each thread responsible for a deadline
  - Choose a predictable scheduler
  - Make assumption on execution time of each thread, ensure that all deadlines are met

- This requires threads interacting with each other through shared memory
  - Synchronizing threads in this matter is a difficult problem
  - Recall the increment/decrement exercise

```
t1(){                    t2(){
  i = i + 1;               i = i - 1;
}                        }
```

  - How do we synchronize the threads such that the resource `i` is not accessed at the same time by different threads?

Repetition

- Synchronization can be achieved with semaphores

```
t1(){
  wait(i_sem);
  i = i + 1;
  signal(i_sem);
}
```

```
t2(){
  wait(i_sem);
  i = i - 1;
  signal(i_sem);
}
```

- Semaphores contain an integer ≥ 0 indicating if the semaphore is blocking or not

- Semaphores can be interacted with through the procedures
  - `wait(`*semaphore*`)`
    - If the internal integer in the semaphore > 0, decrement the value and continue
    - If the integer = 0, the thread suspends and waits until integer > 0
  - `signal(`*semaphore*`)`
    - Increment the internal integer in the semaphore
    - Signals waiting threads that they might be able to proceed

The challenge

- How certain are you that the proposed solution on the previous slide works?
  - There is some uncertainty since the program didn't work before our proposed solution, it might still not work?
- How much time would you have to spend examining the code before making a confident statement about its correctness?
  - What if the correctness of the program involved life or death?
- The number of minutes you spend examining the code before concluding is a metric for the code quality
  - The problems we will be discussing are difficult, and the number will increase rapidly
  - If that number is infinity, the code is bad and should be restructured
- The goal of todays lecture is to be able to identify when a multi-threaded program might not function correctly
  - Extra learning goal: To be able to spot a race condition

The Little Book of Semaphores

- Provides numerous examples on how to solve common problems using semaphores
- The way of thinking is the focus, and each example is iterated upon before reaching the complete solution
- Read through 2 or 3 examples from the book as training in how to identify Race Conditions, and how to solve them using primitive synchronization mechanisms

Race Conditions

What can possibly go wrong?

What exactly is doing the racing?

- Two threads each using the resources `A` and `B`:

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

- Does this work?

- Can cause deadlock if a thread is preempted between locking resources
  - Neither thread can progress
  - Due to interleaving of thread execution from preemptive scheduling

- Example of a Race Condition
  - Recall the definition of Heisenbug from Fault Tolerance
  - A fault that seldom becomes an error, difficult to debug
  - The error depends on the scheduling of the thread executions

Formal definition

From Burns & Wellings (page 143):

> *A data race condition is a fault in the design of the interactions between two or more tasks whereby the result is unexpected and critically dependent on the sequence or timing of accesses to shared data*

Identifying Race Conditions

- Race Conditions are closely related to preemption
  - Ask yourself, what would happen if the program is interrupted exactly here (at this code line)?
- Example, `if`-tests:

```
t1(){
  if(!ready){
    wait(A);
  }
}
```

```
t2(){
  if(ready){
    signal(A);
  }
}
```

```
t3(){
  ready = 0;
  ready = 1;
}
```

  - Not meaningful in multithreaded programs, threads can be preempted right after testing condition
  - The condition might change after preemption
  - Relevant exam question, look for `if`-statements combined with semaphores

Identifying Race Conditions (cont.)

- Example, switching of semaphores

```
t1(){
  wait(A);
  a = a + 1;
  signal(A);
  wait(B);
  b = b + a;
  signal(B);
}
```

```
t2(){
  wait(A);
  a = 10;
  signal(A);
}
```

  - Release one semaphore, lock another
  - Be suspicious, can something go wrong if we get preempted between these two actions?

Challenge for the class

- Consider a multi-threaded program with `N` threads each requiring the resources `A`, `B` or both `A` and `B`
- A thread needing both `A` and `B`, should not reserve `A` if `B` is not available (or vice versa)
  - The thread should be blocked in this case
- How can this be solved with semaphores?
  - Hint: Remember transactions/atomic actions and their use of Lock Managers

Allocating A, B or both A and B

```
allocate(resourceList){
      wait(LockManager);
      if(lm_resoursesAreFree(resourceList))){
            lm_reserve(resourceList);
            signal(LockManager)
            return;
      }else{
            qn = allocateQueueNumber();
            store_request(qn,rList);
            signal(LockManager);
            wait(semQ[qn]);
      }
}

free(rList){
      wait(LockManager);
      lm_unreserve(rList)
      while(Any requests fulfillable){
            set qn & rl for fulfillable request
            lm_reserve(rList);
            signal(semQ[qn])
      }
      signal(LockManager);
}
```

- Threads lock resources by calling the `allocate()` function in the Lock Manager, with their required resources `resourceList`
- Lock Manager locks itself to prevent preemption during allocation
- Internal data structure in `lm_resoursesAreFree()` keeps track of what is reserved
- Each thread has a semaphore for queuing stored in array `semQ`
- Resources are freed by threads calling `free(rList)`, which signals waiting threads

Common Exam question

- Consider a `Person` module with information about persons with the following interface
- The persons are stored in a linked list and the module is protected from simultaneous access of this data structure by the semaphore `P`
- What are the issues with this solution?
  - The function `getNameAndAddress()` deadlocks due to semaphore reservation in nested function calls
- How can we solve this?

```
getName(){
  wait(P);
  …
}

getAddress(){
  wait(P);
  …
}

getNameAndAddress(){
  wait(P);
  getName();
  getAddress();
}
```

Solution to `Person` module deadlock

- Fix the problem by reorganizing the code instead of handling synchronization issues

```
private private_getName(){
  return name;
}
public private_getAddress(){
  return address;
}

public getName(){
  wait(P);
  private_getName();
  …
}

public getAddress(){
  wait(P);
  private_getAddress();
  …
}

public getNameAndAddress(){
  wait(P);
  private_getName();
  private_getAddress();
}
```

Implementing a Barrier

- A barrier is synchronization device that ensures that all threads have arrived at a point before any proceed
- The two-phase commit protocol used in Transactions is an example of this
- Implemented Ada and Java examples of this last week
  - In the Ada example we used an `entry` with a guard that checked how many tasks were blocked by the guard using the `count` parameter
  - The guard opened when the number of waiters equaled the number of tasks, at which point the guard opened and the first task through ensured the guard stayed open for the others
- Now we're going to solve the same problem with semaphores, using an example from The Little Book of Semaphores

Barrier non-solution #1

```
// rendezvous
mutex.wait()
   count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()

// critical point
//…
```

Barrier solution

```
// rendezvous
mutex.wait()
   count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

// critical point
//…
```

Barrier non-solution #2

```
//rendezvous
mutex.wait()
   count = count + 1
   if count == n: barrier.signal()

   barrier.wait()
   barrier.signal()
mutex.signal()

//critical point
//…
```

Reusable barrier non-solution #1 -

```
//rendezvous
mutex.wait()
    count += 1
mutex.signal()

if count == n: turnstile.signal()

turnstile.wait()
turnstile.signal()

//critical point
//…

mutex.wait()
    count -= 1
mutex.signal()

if count == 0: turnstile.wait()
```

Reusable barrier non-solution #2

```
// rendezvous
mutex.wait()
    count += 1
    if count == n: turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

// critical point
//…

mutex.wait()
    count -= 1
    if count == 0: turnstile.wait()
mutex.signal()
```

Reusable barrier solution

```
//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()
```

Basics of a Real-Time Kernel

Motivation

- The use of semaphores for synchronization is dependent on an important property
  - The function calls to `wait()` and `signal()` can not be preempted
- How do we ensure that such interruptions can not happen?
  - Adding semaphores to `wait()` and `signal()`?
  - As with the “Turtles all the way down” expression, this doesn’t solve anything
  - We must tell the hardware to not interrupt during this time, in other words disable interrupts
- To ensure that this communication between hardware and software goes smoothly, a middleman is needed
  - The Real-Time Kernel
  - Which features do we require of a preemptive kernel?

Feature 1: Preemption

- Multithreading is the main feature of a Real-Time Kernel
  - Requires switching between threads preemptively
- Efficient thread switching requires
  - Handling of timer interrupts
    - Hardware on motherboard sends signals to processor at regular intervals
    - The kernel must define what happens in an interrupt procedure
    - Decide whether to preempt the currently executing thread
  - Storing all registers
    - The complete state of the microprocessor must be stored such that the preempted thread can resume work when it regains control
    - This includes the instruction pointer and stack pointer
    - Referred to as the threads context
  - Organize queue of processes
    - The context of runnable threads is stored in a queue ready for being run again
    - The scheduler selects a new thread from this queue to run after preemption
    - The queue contains the entire context of each thread

Synchronization?

- Is it possible to achieve thread synchronization?
  - We don't have access to semaphores, haven't invented them yet
  - Need to ensure mutual exclusion
- What about

```
if(!occupied){
  occupied = 1;
  \\doWork
  occupied = 0;
}
```

  - `If` statements are a bad idea, can be interrupted after the test.
  - A single flag is not enough
- Need active waiting
  - Known as spin locks: `while(occupied){} //doWork`
- How can we solve the synchronization problem with spin locks?
  - Not allowed to turn of preemption, allowed to waste time however
  - Difficult problem, when it was solved in the 60s it resulted in multiple published papers

Synchronization with spin locks

```
flag1 = 0;
flag2 = 0;
turn = 1;

t1(){
  flag1 = 1;
  turn = 2;
  while(flag2 == 1 && turn == 2){}
  //doWork
  flag1 = 0;
}

t2(){
  flag2 = 1;
  turn = 1;
  while(flag1 == 1 && turn == 1){}
  //doWork
  flag2 = 0;
}
```

- First `flag1` is set, indicating that `t1()` intends to access the critical region
- The `turn` variable specifies which thread should be allowed to run if both threads wish to proceed
  - The threads are polite and allows the other to run first
- This prevents deadlocks since each thread yields control to the other
- The `while` loops are spin locks, waiting for the either the `flag` to go down or the other thread to yield priority via the `turn` variable

Making spin locks safer

- Consider the following

```
while(!A){};
A=1;
```

- We wish to ensure that we can't be interrupted between reading and writing A, make the action atomic

- Software engineers asked Intel in the 60s if they could make an assembly instruction so the code could be written safely

- The result was the *test and set* and *swap* instruction
  - They are practically identical, see Burns & Wellings for more details

Swap instruction

- Swap is a single assembly instruction, and can therefore not be preempted

- The instruction swaps a 1 bit with the lock bit and returns the old bit value of the lock
  - A return of 0 means that the lock is available, and is now already reserved for us with the swapped 1
  - A return of 1 means that the lock is in use, and we must wait, no change has been made since we swapped the lock bit value 1 with our swap bit value 1

- Example

```
//Not safe
while(occupied){};
occupied = 1;
//doWork
occupied = 0;
```

```
//Safe
while(swap(occupied,1)){};
//doWork
occupied = 0;
```

Feature 2: Priorities

- The decision to preempt is so far being made using a Round Robin strategy
  - All threads get equal run time
- This works well for desktop computer, but not for Real-Time systems
  - We want tasks to run according to their importance and temporal requirements
- The solution is to assign the tasks priorities
  - High priority means you can skip to the front of the runnable queue
- Sort the process queue
  - Sort by nearest deadline first
  - Or have more queues with different purposes

Feature 3: “Synchronization”

- Without semaphores, synchronization has been solved with spin locks
  - However, this is a waste of CPU time doing idle work
  - If a thread has nothing to do, it shouldn’t run
- The solution is to add another process queue
  - This queue contains all blocked (waiting) processes that are not currently runnable
- Additionally, we add the functions `suspend()` and `resume(T)`
- `suspend()`
  - When called by a thread, stores all registers
  - The context of the thread is then wrapped in a thread object and put into the blocked queue
  - The scheduler then decides which runnable thread gets to run
- `resume(T)`
  - Takes a thread object from the blocked queue and moves it to the runnable queue
- Fixes the bad performance of spin locks

Why the quotes?

- `suspend()` and `resume(T)` are far from an ideal solution
- Example:

```
t1(){                          t2(){
  init();                        if(!initOK){
  initOK = 1;                      suspend();
  resume(t2);                    }
}                              }
```

  - What is the issue?
  - If `t2()` runs first, followed by `t1()`, there is no issue
  - If `t1()` preempts `t2()` after the `if` – test, and runs to completion, `t2()` deadlocks

- Difficult to avoid Race Conditions
  - No operating system or language depends solely on `suspend` and `resume` for mutual exclusion and synchronization
  - Early Java had direct methods for this, but later removed them due to high error rate

Feature 4: Protection of scheduler data

- Previously, only timer interrupts from the motherboard could trigger a preemption
  - With `suspend` and `resume` every thread can trigger a preemption
- The schedulers data structures, the queues, are therefore vulnerable to inconsistent states due to mutual access from several threads
- How do we protect the integrity of the queues?
  - Disable interrupts while a preemption is being handled
  - Can only do this for short operations, since longer operations would effectively kill preemption in the system
  - Could also use spin locks, swap or “test and set” instructions

Feature 5: Better synchronization mechanisms

- At this point, we are longing for semaphores
  - With the tools discussed so far, we are able to implement semaphores
  - Outside the scope of the kernel discussion
- Additionally there are other mechanisms that are of interest
- Events
  - Anonymous suspend and resume
  - We suspend ourselves, and when the event happens, everyone waiting for the event is resumed
- Condition variables
  - Suspend waiting for the condition variable to be true
  - When the condition variable is true, all waiting threads are resumed
- Same drawbacks as suspend and resume, difficult to use

Standard applications of synchronization

- Critical section
  - Code that must not be interrupted
  - Contents of `wait()` and `signal()`
  - Locking operations in atomic actions
- Mutual exclusion
  - Ensures multiple threads not interfering with each other unintended
- Bounded buffer
  - Buffer with full/empty synchronization instead of error return
  - Read from empty buffer, get suspended until element arrives
  - Writing to full buffer, suspended until space is available
  - Will be a recurring example
- Read/Write locks
  - Concurrent readers are ok, preemption during read no problem
  - Mutual exclusion during writing (changing of data)
  - In Ada protected objects, functions are read locks, procedures are write locks

Standard applications of synchronization (cont.)

- Condition synchronization
  - Waiting on event or status
  - Wait for initialization, solved by semaphores
- Resource allocation
  - Letting actions own resources
  - Atomic actions and transactions
  - Higher level of mutual exclusion
- Barrier
  - Everybody waits for everybody
- Communication
  - Writing data to a variable is a kind of communication
  - Receiver wants to know when new data is available