D0003E: real-time systems

Summary

Course focus

- **Concurrency** – how to write programs using parallel threads of execution
- **Reactivity** – how to write programs whose purpose is to react to events (ultimately in the form of interrupts)
- **Real-time** – how to write programs whose correctness depend on their real-time behavior

Course contents

- **Introduction to real-time systems & bare metal programming in C**
  - C vs. Java, pointers, type-casts, the execution stack
- **Bit manipulation & hardware interfacing**
  - (memory-mapped) ports, status changes, busy-waiting
- **Concurrency and mutual exclusion**
  - problems sequential programs, threads, critical sections
- **The inner workings of a kernel**
  - setjmp/longjmp, fresh stacks, the ready queue, yield, interrupts

- **Events, interrupts and reaction**
  - problems with busy-waiting, the CPU as a reactive object
- **A model of reactive objects**
  - state, methods, self, SYNC/ASYNC, cyclic calls, deadlock, idling
- **Clocks, timers and periodic execution**
  - time-stamps, delays, period drift, event baselines + offsets
- **Deadlines and priorities**
  - WCET, timely reaction, (pre-emptive) scheduling, static/dynamic prior

- **Scheduling and feasibility**
  - restricted model, RAM, EDF, optimality, schedulability tests
- **Priority inversion**
  - sporadic tasks, DNA, response times, blocking time, inheritance/ceiling
- **POSIX thread programming**
  - real. OS services, thread creation, mutexes, tasks, POSIX clocks
- **More on inter-process communication**
  - POSIX signals & timers, event-loops, parking, select(), semaphores
- **Real-time languages**
  - monitors, rendezvous, Java threads & monitors, Ada guards & messages

Realtime

- Someone asks about the current outdoor temperature. Which response is better?
  - A correct reading of 20°C delivered 12 hours later
  - An false reading of 10°C delivered immediately
- In a realtime system, a late reponse is just as bad as a wrong one
Threads

- System supporting seemingly concurrent execution
  - Called multi-threaded
- A thread
  - Unique execution of a sequence of machine instructions
  - Can be interleaved with other threads executing on the same machine
- Each thread
  - Its own execution stack, where its local variables, function arguments, and return addresses are stored
- Shared between threads
  - Global variables, so-called static variables
  - Heap-allocated data
  - All other system resources

Critical sections

- Part of code: access to resource
  - Shared between tasks/threads/reactive objects
- Protect
  - Simultaneous access
    - Avoid data corruption
- Mutex
  - Implementatio of critical section
  - Serializes access to resource
  - Only one thread at a time

Mutual exclusion

```c
struct Params p; mutex m;
while (1) {
  lock(&m);
  p.minDistance = e1;
  p.maxSpeed = e2;
  unlock(&m);
}
```

Possible interleaving:

- `p.minDistance = 1;`  
- `p.maxSpeed = 1;`  
- `local_minD = 1;`  
- `local_maxS = 1;`

Critical sections run under mutual exclusion!

Busy-wait: Consequence 1

- Long wait for status change
  - With N threads in the system
  - Delay will be T*(N-1) in the worst case
  - All other threads executed before

Interrupts

- On hardware level
  - Interrupt signals
    - Can be seen as port-initiated write operations
- Interrupts
  - Need for busy-waiting disappears
  - Compare to:
    - Checking into the post-office again and again to see if a delivery has arrived, or
    - Receiving a note in your mailbox that the goods can be picked up
- The CPU reacts to an interrupt signal by executing a designated interrupt service routine (ISR)
  - CPU receives signal
  - Immediately starts executing ISR
    - Regardless of current code executing
  - Later resumes executing regular code
Deadlock

• Deadlock is the result of requesting new exclusive access to something you already have.

• In general, a chain of tasks may actually be involved:

T1 holds m1
T1 wants m2 (waits)

T2 holds m2
T2 wants m3 (waits)

T3 holds m3
T3 wants m1 (waits)

Deadlock: Necessary conditions

• Mutual exclusion
  — only one process at a time can use resource

• Hold and wait
  — processes holding resource
  — … waiting for another resource

• No preemption
  — resource cannot be taken away from process
  — process can only release it voluntarily

• Circular wait
  — chain of processes
  — holds resource
  — waits for another resource
  — which is already held by other process

Deadlock prevention and avoidance

• Prevention:
  — Remove any of the conditions
  — deadlock will not occur

• Avoidance: don’t allow the system to enter an “unsafe” state
  — If it is possible to...
    • allocate resources to each process
    • in some order
    • not enter deadlock state
  — … then we’re always safe!

Periodic execution: Accumulating drift

• With relative delays, each turn in the loop will take at least 100 + x milliseconds, where x is the time taken to perform do_work().
  — Thus, a drift of x milliseconds will accumulate every turn!

Priorities and deadlines: Three issues

• How do we express the real-time constraints?
  • Deadlines!

• How do we construct a scheduler
  • that ensures that those constraints are met?
    • Priority scheduling!

• How do we tell whether the scheduling task is impossible?
  • Ahead of time, or only when it’s too late?
    • Schedulability analysis!

Priorities

• Task/thread/message priorities are integer values that denote the relative importance of each task

• Quite often the priority scale is reversed, meaning that low priority values = high priority

• A priority scheduler always runs the task with the highest priority

• This means that a task can only run after all tasks considered more important have terminated / blocked

• Tasks with identical priorities are sorted according to some secondary scheme, e.g., first-in-first-out
Non-preemptive scheduling

 ISR

 p1 ready

 i/o request syscall

 i/o completes (interrupt)

 p2 terminates

 low-pri: p2

 high pri: p1

 switch to p2

 low-pri: p2

 waiting/ready->running

 running->ready

 ready->running

 running->terminated

 switch to p1

 cannot switch here

 kernel

 Preemptive scheduling

 ISR

 p1 ready

 i/o request syscall

 i/o completes (interrupt)

 p1 terminates

 low-pri: p2

 high pri: p1

 switch to p2

 running->waiting

 waiting->ready

 ready->terminated

 switch to p2

 ready->running

 switch to p1

 Static priorities – method

 • Under the given assumptions, there exists a static priority assignment rule that is really simple:
   – “The shorter the period, the higher the priority”
   • This rule is called Rate Monotonic Priority Assignment, or RM for short
   • For RM, the actual priority values do not matter, only their relative order
   • Because of our inverse priority scale, we can simply implement RM by letting \( P_i = D_i (= T_i) \)

 Dynamic priorities – method

 • Under the given assumptions, there exists a dynamic priority assignment rule that is really simple:
   – “The shorter the time remaining until deadline, the higher the priority”
   • This rule is called Earliest Deadline First, or EDF for short
   • Because EDF will want to distinguish between messages on basis of their absolute deadlines, priority values must use the same units as the system clock
   • Under EDF, each activation \( n \) of periodic task \( i \) will receive a new priority: \( P_{i(n)} = \text{baseline}_{i(n)} + D_i \)

 Optimality

 • Under some given assumptions
   – might be several ways of assigning priorities so that deadlines are met
 • Clearly, a method that only fails if every other method also fails is preferred
   – such a method is called optimal
 • RM is optimal among static priority assignment methods
 • EDF is optimal among dynamic methods
 • However, knowing that a priority assignment is the best one possible is not the same thing as knowing that it is “good enough”; i.e., knowing that deadlines actually will be met

 Schedulability

 • Assume our priority assignment method is optimal
   – like knowing shortest path from A to B
   • but still not knowing if path short enough so that B can be reached in time
 • To answer: Will tasks actually meet their deadlines?
   – determine if task set is schedulable (an optimal priority assignment method will produce a schedule if a schedule exists)
 • Clearly, the question of schedulability must take the WCETs of tasks into account
Blocking: NOT priority inversion

Response time for B increased by the longest time A can run with obj locked!

Priority inversion

Here, blocking time for B involves full execution times of all tasks between high and low, perhaps for multiple periods...

Priority inheritance

With priority inheritance, the blocking time for B is bounded by the time task A locks obj

However, note that task X is now delayed instead!

Priority ceiling protocol

The multi-event-loop pattern

Ideally we would like a parking operation that waits for exactly those events a thread is interested in:

```c
void *fun( void *arg ) {
    INITIALIZE;
    while (1) {
        x = PARK;
        switch (x) {
        case 0: REACT0; break;
        case 1: REACT1; break;
        ... case n: REACTn; break;
        }
    }
}
```

Unfortunately, a truly generic parking op doesn't exist...

Semaphores

- Original process synchronization device (due to Dijkstra 1965)
- Supports two operations: `wait` and `signal` (signal is called `post` in POSIX)
- The general semaphore is `counting`; i.e., it remembers the number of signal calls, and allows the same number of wait calls to succeed without stopping
- A counting semaphore must be initialized with its starting value (the number of initially allowed wait calls)
A semaphore Bounded Buffer

```
#include <semaphore.h>
sem_t   mut;
sem_t   space, items;
int head = 0, tail = 0;
T buf[SIZE];
...
sem_init( &mut, 0, 1 );
sem_init( &items, 0, 0 );
sem_init( &space, 0, SIZE );
```

```c
void put(T item) {
    sem_wait( &mut );
    sem_wait( &space );
    buf[head] = item;
    head = (head + 1) % SIZE;
    sem_post( &mut );
    sem_post( &items );
}
void get(T *item) {
    sem_wait( &items );
    sem_wait( &mut );
    *item = buf[tail];
    tail = (tail + 1) % SIZE;
    sem_post( &mut );
    sem_post( &space );
}
```

Notice the lock/unlock pattern
And the "parking" pattern

Monitors: Things to note

- While mutual exclusion is implicit
  - conditional synchronization still handled explicitly
- condition variables are local to a monitor
  - risk of improper use is reduced
- Observe:
  - convars. are not counting – every wait call blocks
- While a thread blocks on wait
  - monitor is temporarily opened up to some other thread
    - so that someone may be able to call signal eventually

Monitors

- Idea:
  - objects and modules successfully control the visibility of shared variables
  - why not make mutual exclusion automatic for such a construct?
- This is the idea behind monitors, a synchronization construct found in many early concurrent languages (Modula-1, Concurrent Pascal, Mesa)
- Monitors elegantly solve the mutual exclusion problem; for conditional synchronization a mechanism of condition variables is used
- We give an example in C-like syntax; note, though, that neither C nor C++ support monitors directly

A Java BB

```
public class BoundedBuffer {
    private int count = 0, head = 0, tail = 0;
    private T buf[] = new T[SIZE];
    public synchronized void put(T item) {
        if (count == SIZE)
            wait();
        buf[head] = item;
        head = (head + 1) % SIZE;
        count++;
        notify();
    }
    public synchronized T get() {
        if (count == 0)
            wait();
        T item = buf[tail];
        tail = (tail + 1) % SIZE;
        count--;
        notify();
        return item;
    }
}
```

Guards (Conditional critical region)

- Assume monitor
  - if the condition that avoids the wait call is “lifted out” to guard the whole monitor method instead
- That is,
  - instead of letting threads in
    - only to find that some condition doesn’t hold
    - keep the threads out until the condition does hold
- This requires a new language construct
  - allows state-dependent boolean expressions outside the actual methods, but still protected from concurrent access
  - blocked waiting on a boolean expression

Meddelanden
That was all!