FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Real Time Systems – SS2016

Prof. Dr. Karsten Weronek

Faculty 2

Computer Science and Engineering

Semaphores

Mutexes

PIP, PCP

etc.

Necessary and sufficient conditions

For the schedulability tests we had necessary and sufficient conditions.

What are the implications if

- A necessary condition is violated
- A necessary condition is fulfilled
- A sufficent condititition is violated
- A sufficient condition is fulfilled

Example on whiteboard

Exercise in the audience

Dependant Tasks/Processes

When you have to solve a Real-Time problem you probably need a couple of threads or processes that work in parallel.

You have to distinguish the following relationships between parallel tasks:

- **disjunct tasks; they run completely independent**
- **Non-disjunkt tasks; they use common data or resources**
  - competing tasks, that apply for access to the same data,
  - Tasks that are dependent, because one has deliver for the other (producer/consumer-model)

The result of non-disjunct prallel processes without proper synchronisation is often non-predictable and non-reprocucible.

Critical Sections

The part of a code sequences (Thread/Process or Interrupt-service Routines) in which a joint access to common resources may occur are called „Critical Section.

When two or more tasks access common data, this may lead to a „Race Condition“.

The result of a Race Condition depends on the relative progres of the different tasks.

Avoidance of Race Conditions

Race conditions can be avoided by allowing access to a critical section for one task only at a time. This is called mutual exclusion (gegenseitiger Ausschluss).

The mathematical instrument to solve this issue is called a semaphore.

Semaphore (Edsger Dijskstra)

- A semaphore is an integer S
- A semaphore has a Maximum N
- A semaphore is initialised by N

- There is a function P:
  - S = S-1 (passeeren)
  - If S<0 then queue
- There is a function V:
  - S = S+1 (vrejgeben)
  - If S<=0 the take one out of the queue (highest Prio)

Special Semaphore

A Semaphore with N=1 is called **Binary Semaphore** or a **Mutex**.

It tells you only free or not free and

The size of the queue respectively

Example on white board

The little room..

Implementation (pseudo-code)

- P-Operation:

```
S = S - 1;
if (s < 0) {
   sleep_until_semaphore_is_free()
}
```

- V-Operation:

```
S = S + 1;
if (s ≤ 0) {
   wake_up_sleeping_process_with_highest_priority;
}
```

- P- and V-Operations are critical sections as well and are non-interuptable. Therefore semaphore operations are Systemcalls.

Mutex functions in POSIX

```
int pthread_mutex_init (…)
int pthread_mutex_destroy (…)
int pthread_mutex_lock (…)
int pthread_mutex_trylock (…)
int pthread_mutex_unlock (…)
```

C-example (POSIX) for Mutex

```
#include <pthread.h>
#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main() {
  pthread_t thread_id[NTHREADS];
  int i, j;

  for(i=0; i < NTHREADS; i++){
    pthread_create( &thread_id[i], NULL, thread_function, NULL );
  }
  for(j=0; j < NTHREADS; j++) {
    pthread_join( thread_id[j], NULL);
  }
  printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr) {
  printf("Thread number %ld\n", pthread_self());
  pthread_mutex_lock( &mutex1 );
  counter++;
  pthread_mutex_unlock( &mutex1 );
}
```

Deadlocks

Let two tasks A and B:

having to Mutexes S1 and S2; the request sequences are:

Task A: P(S1); P(S2); V(S2); V(S1)

Task B: P(S2); P(S1); V(S1); V(S2)

Task A	Task B
P(S1)	P(S2)
P(S2)	P(S1)
V(S2)	V(S1)
V(S1)	V(S2)

Deadlock occours:
S1 is blocked by Task A
S2 is blocked by Task B

What leads to a deadlock

There are four sufficient conditions (in combination) for deadlocks:

1. Resources are exclusive and are not accessible by multiple task at a time
2. A task uses at least two resources (e.g. CPU and microphone)
3. The resource can not be withdrawn easily from a using task by another task
4. Meanwhile the usage of resources by different task there exists a cyclic chain

The developer has to avoid only one of the four conditions to avoid deadlocks.

This is not easy!

What leads to a deadlock

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

1. Resources are exclusive and are not accessible by multiple task at a time

This is hard to change. For Data/Memory access it is impossible. For most of devices or blocks on the board it is also not possible.

What leads to a deadlock

2. A task uses at least two resources (e.g. CPU and microphone)

You may isolate pure calculating task in a core.

However since most of the task have frequent I/O or use CoProcessors or other blocks this is not an option to change.

Another alternative would be if a task always waits untill all resources are free. This will take a lot of time….

What leads to a deadlock

3. The resource can not be withdrawn easily from a using task by another task

This is dependent of the device. Most of devices have a data loss when you withdraw it from a task.

What leads to a deadlock

4. Meanwhile the usage of resources by different task there exists a cyclic chain

This can be avoided by defining a linear order for the access to the required resources. → Piping. This may be not useful or it may take time.

Write/Read-Locks

When the resource is data it depends if the task wants to read or to write date because reading date by multiple task at the same time is not an issue.

This is realised by a semaphore that allows parallel read-request but allows only a single process to write. See the following cases:

Write/Read-Locks

1. The critical section is free

   → Allow acces

2. There is on or more tasks, who reads the critical section and there is no task to write:

   → A task to read gets access

   → A task to write will be blocked

3. The critical section is accessed by at least one reading task and a writing tasks wants to access

   → Access will be blocked

4. The critical section is accessed by a writing task

   → All other requesting tasks will be blocked.

Preemption-Model

The preemption modell is responible for the safety of critical sections. It devides the operating system in different layers.

The Systemsarchitect need to now it because it influences the real-time behaviour.

The Application Developer need to now it because to identify the critical parts of the code and to be able to assure that they will be secured.

Preemption Model in Linux

The preemption model is devided into four layers:

- The Application-Layer (Userland)
- The Kernel-Layer
- The Soft-IRQ-Layer
- The Interrupt-Layer

Some Details:

On the application layer user programs runs with certain priorities

S

When an application starts a thread on the Kernel-Layer it inherits the priority

Preemptation Model

Some Details:

On the application layer user programs runs with certain priorities

Semaphors are used for critical sections

When an application starts a thread on the Kernel-Layer it inherits the priority

User threads compete with Kernel-Threads. Kernel Threads are not interruptable.

Interrupt Service Routins have the highest (all the same) priority (HW-Priority).

On single core machines no need to protect data.

After having finished the Hardware ISR the soft-IRQ are executed.

Since Interrupts are not allowed for sleep mode, use of semaphores doesn‘t make sense

Priority Inversion

The situation where a high priority task requiring a resource is blocked due to the lock of this ressource by a low priority task.

Then the high priority task is blocked until the low priority task is completed an has released the resource.

This can take a long time, especially when mid-prioritised tasks delay the completion of the low prioritised one.

Priority Inversion

To prevent blocking owing to Priority Inversion

There three possibilities to „heal“ Priority Inversion:

1. NPCS (Non Preemptive Critical Section) „Unterbrechungssperre“
2. Priority Inheritance Protocol
3. Priority Ceiling Protocol

1. Ensures that tasks will be completed wihtout interruption Implemented for each layer of the Preemptation model
2. Avoids blocking by Priority Inheritance however issues like chaining („cascaded delays“) or deadlooks
3. Enhancement of the PIP without the issues

Priority Inheritance Protocol (PIP)

A low priority task holding a critical resource that ,
is requested by a higher prioritised task
inherits the priority of the higher prioritised task.

After completion of the access to the critical resource
the priority will be set back to the initial value.

Priority Ceiling Protocol (PCP)

See Side-document

Deadlocks

Let two tasks A and B

Having to Mutexes S1 and S2 the request sequences are:

Task A: P(S1); P(S2); V(S2); V(S1)

Task B: P(S2); P(S1); V(S1); V(S2)

Task A	Task B
P(S1)	P(S2)
P(S2)	P(S1)
V(S2)	V(S1)
V(S1)	V(S2)

Deadlock occours:
S1 is blocked by Task A
S2 is blocked by Task B