Lecture - 3 - Task synchronization

TSEA81

Computer Engineering and Real-time Systems

Linköping University
Sweden

This document is released - 2014-11-10 - first version

Author - Ola Dahl, Andreas Ehliar

Asymmetric synchronization

- One-way synchronization
- One task *P1* informs another task *P2* that *P2* can continue its execution
- Can be implemented using a semaphore, where *P1* performs a *Signal*-operation and *P2* performs a *Wait*-operation

Time synchronization

Consider a clock, as used in the first two assignments in the course. The following observations can be done.

- Calling *usleep* makes the calling task wait *relative* to the time of the call. The actual clock period will be larger than the time specified to *usleep*, since the clock task also updates and displays the time.
- Timing can be improved by instead calling *clock_nanosleep* with the TIMER_ABSTIME flag set, which makes the calling task wait until a specified time instant.
- Timing can also be improved by having a higher prioritized task (a tick task) which calls *usleep* and then trigger a clock task using asymmetric synchronization. Here, the tick task does nothing else in its while-loop, and the clock task updates and displays time in its while loop.

Assignment 2 - Alarm Clock

Design and implementation considerations:

- Determine, using pseudocode or drawings, the main actions for each task, to be performed inside the while-loop.
- Think about how the tasks use shared resources.
- There is a difference between an enabled alarm (the alarm is set), and an activated alarm (the alarm is active, i.e. the clock is ringing).

Asymmetric synchronization and interrupt handlers

- Sometimes it is required that a task shall wait for an external event.
- Tasks waiting for time to expire wait for the external event "timer interrupt occurred N times". When this happens, the interrupt handler makes the task ready for execution.
- Asymmetric synchronization using a semaphore, can be implemented between an interrupt handler and a task. The interrupt handler does *Signal* and the task does *Wait*, on the semaphore. In this way, a task can wait for an external event, e.g. the pressing of a button, the arrival of data on a communication channel, etc.

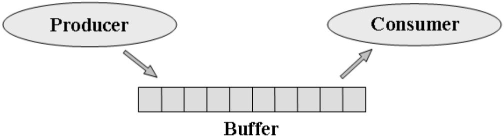
Symmetric synchronization - Tasks wait for each other, e.g. during a data transfer.
Can be implemented using semaphores, as

```
Process P1
{
  while (1)
  {
     .
     .
     Prepare data
     Write data
     Signal(Data_Ready)
     Wait(Data_Received)
     .
  }
}
```

```
Process P2
{
  while (1)
  {
    .
    .
    .
    Wait(Data_Ready)
    Read data
    Signal(Data_Received)
    Process data
  }
}
```

Producers and Consumers

A producer and a consumer, and a buffer used for their communication:



Typical requirements:

- A producer shall wait if the buffer is full.
- A consumer shall wait if the buffer is empty.

A buffer can be declared as

```
/* buffer size */
#define BUFFER_SIZE 10

/* buffer data */
char Buffer_Data[BUFFER_SIZE];

/* positions in buffer */
int In_Pos;
int Out_Pos;

/* number of elements in buffer */
int Count;
```

The buffer can be protected using a mutex.

Conditional critical regions

Conditional critical regions are critical regions associated with conditions.

Requirements on functionality for critical regions:

- It must be possible for a task to *wait* if a given condition is satisfied.
- There must be a mechanism for *activation* of a waiting task, so that a task can re-evaluate a condition, in order to determine if it is allowed to enter its critical region.

Condition variables

- Can be used, together with mutexes, for implementing conditional critical regions.
- Three operations: one operation for initialisation, and two operations denoted *Await* and *Cause*.
- A condition variable is *associated with* a mutex.