3 - Task synchronization

TSEA81 - Computer Engineering and Real-time Systems

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

Author - Ola Dahl, Andreas Ehliar

Lecture - 3 - Task synchronization

This lecture note starts with a description of asymmetric synchronization, which is a method for synchronization of tasks.

The lecture note also describes a classical problem, denoted producer and consumer, which occurs when a task sends data to another task, via a shared buffer. In connection with this, condition variables are introduced. Condition variables can be used e.g. in real-time programs where tasks communicate via a shared buffer.

The notation [RT] is used to refer to the book Realtidsprogrammering1.

Asymmetric synchronization

Parallel tasks are often cooperating, e.g. when using shared resources. Sometimes it is desirable that the cooperation is performed in a way that synchronizes the execution of the tasks. Asymmetric synchronization is a form of one-way synchronization, where one task, e.g. denoted P1, informs another task, e.g. denoted P2, that P2 shall continue its execution. One reason why this information is given to P2 could be that data which shall be received by P2 have arrived, and that P2 therefore needs to be informed.

Asymmetric synchronization can be implemented using semaphores. The synchronization is performed using a semaphore, e.g. denoted Change_Sem. If the asymmetric synchronization also involves handling of shared data, e.g. data that shall be transferred from one task to another task, an additional semaphore is needed. This semaphore, which e.g. is denoted Mutex, is used to ensure mutual exclusion.

A task P1 which shall transfer data to another task P2 performs a Signal-operation on the Change_Sem semaphore. The task P2, which shall receive data, performs a Wait-operation on the Change_Sem semaphore. A one-way synchronization is then obtained, where the task P1 activates the task P2.

Asymmetric synchronization is described in Section 5.2 in [RT], which contains a program example where data are sent from one task to another task using a shared buffer, and where asymmetric synchronization is used.

Asymmetric synchronization is used in Assignment 2 - Alarm Clock.

1 https://www.studentlitteratur.se/#31445
Time synchronization

In real-time systems, it is often required that specified actions are taken at specified time instants. When using a clock, as is the case in Assignments 1 - Introduction, Shared Resources and 2 - Alarm Clock, a natural requirement is that the clock keeps its time, i.e. that it actually updates its time every second.

The clock update in Assignment 1 - Introduction, Shared Resources is implemented in a task *clock_task*, given by

```c
/* clock_task: clock task */
void clock_task(void)
{
    /* local copies of the current time */
    int hours, minutes, seconds;

    /* infinite loop */
    while (1)
    {
        /* read and display current time */
        get_time(&hours, &minutes, &seconds);
        display_time(hours, minutes, seconds);

        /* increment time */
        increment_time();

        /* wait one second */
        usleep(1000000);
    }
}
```

In this implementation, a call to *usleep* is used, for the purpose of achieving a periodic clock update, every second. The call to *usleep* makes the calling task wait the specified number of milliseconds, relative to the time when the call is done. Before calling *usleep*, there are calls to the functions *get_time* and *display_time*. This means that the actual period of the clock will in this case be the execution time for the calls to *get_time* and *display_time* plus one second.

Different methods for making the clock execution closer to real-time can be considered. One method is to introduce a task, here referred to as *tick_task*, with higher priority than *clock_task*. Further, a semaphore, called *tick_sem* is introduced. The semaphore *tick_sem* is then used in an asymmetric synchronization scenario, between *tick_task* and *clock_task*, where *tick_task* performs a Signal-operation and where *clock_task* performs a Wait-operation. If the *tick_task* is
implemented so that its only actions in its while-loop are the Signal-
operation and the call to usleep, then the resulting system executes
closer to real-time than in the previous implementation, where the
clock task called usleep.

Another method is to use a function which makes the calling
task wait until a specified time instant. In POSIX, this is possible by
using the function clock_nanosleep with the TIMER_ABSTIME flag
set. In addition, a variable is used, for the purpose of storing the
time instant for the next update of the clock. A clock task can be
implemented using this method, as

```c
/* clock_task: clock task */
void clock_task(void)
{
    /* time for next update */
    struct timespec ts;

    /* local copies of the current time */
    int hours, minutes, seconds;

    /* initialise time for next update */
    clock_gettime(CLOCK_MONOTONIC, &ts);

    /* infinite loop */
    while (1)
    {
        /* read and display current time */
        get_time(&hours, &minutes, &seconds);
        display_time(hours, minutes, seconds);

        /* increment time */
        increment_time();

        /* compute time for next update */
        ts.tv_nsec += delay;
        if(ts.tv_nsec >= 1000*1000*1000){
            ts.tv_nsec -= 1000*1000*1000;
            ts.tv_sec++;
        }
        /* wait until time for next update */
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts, NULL);
    }
}
```
Assignment 2 - Alarm Clock

Assignment 2 - Alarm Clock treats implementation of an alarm clock. When implementing the alarm clock, the following considerations may be helpful:

- Regarding the tasks and their (infinite) while-loops, determine, using pseudocode or drawings, the main actions for each task, to be performed inside the while-loop.

- Think about how the different tasks use shared resources.

- Note that 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).

- Note that the requirements regarding text format for the set command is slightly different (now blanks are used between the command and the digits and between the digits) from the format used in the previous assignment (where a colon was used instead).

Asymmetric synchronization and interrupt handlers

Sometimes it is required that a task shall wait for an external event, which is triggered by an interrupt. When an interrupt arrives, an interrupt handler is executed, and it must then be determined how this interrupt handler can interact with the real-time operating system.

For certain types of interrupts, e.g. the periodic timer interrupt used for keeping the time base for the real-time operating system, the interrupt handler may call functions in the real-time operating system which directly manipulate the internal data structures of the real-time kernel. In a typical operating system this is done by calling a function which traverses the list of tasks waiting for time to expire, and if one or more of these tasks are ready for execution (i.e. their time has expired), they are moved to the list of tasks ready for execution.

In other interrupts, it may be possible to use available mechanisms in the real-time kernel, such as semaphores and primitives for message passing. One method, for synchronizing a task and an interrupt handler, is to use asymmetric synchronization, where an interrupt handler performs a Signal-operation and where a task performs a Wait-operation. In this way, the task is able to wait for the external event, which when occurring, triggers the interrupt handler which in turn activates the task by a Signal-operation. In situations like this, questions regarding mutual exclusion must also be considered, so that data shared between a task and an interrupt handler are protected.
Symmetric synchronization

It may be required that tasks, when interacting, shall do this in a way that makes them *wait for each other*. One example is transfer of data, where the sending task may not send data until a receiving task is ready, and a receiving task is required to wait until data is ready to send.

It is possible to implement symmetric synchronization using semaphores.

Consider a situation with two tasks, \( P_1 \) and \( P_2 \). The task \( P_1 \) shall send data to the task \( P_2 \). The data transfer is done by having shared data, which are written by \( P_1 \) and read by \( P_2 \). It is required that the synchronization, when data are transferred, is symmetric, which means that the tasks \( P_1 \) and \( P_2 \) shall wait for each other. For this purpose, two semaphores are used, denoted \( Data\_Ready \) and \( Data\_Received \). Both semaphores are initialized to the value zero. The implementation, using pseudocode, becomes

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

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

As can be seen in the above listing, the task first arriving to the point in its code where data transfer shall take place, automatically waits for the other task to arrive.

It is also possible to implement symmetric synchronization using the message passing paradigm. Symmetric synchronization can be accomplished by having a function for message sending, which when called forces the calling task to wait until a task is ready for receiving the message. This type of communication is further described in Lecture *Message passing*. 
Producers and Consumers

It is often advantageous to use a buffer when transferring data from one task to another task. A communication of this kind is illustrated in Figure ??.

Figure ?? shows a task, denoted Producer, which is a producer. This task writes data to the buffer. Data are read from the buffer by the consumer, denoted Consumer.

There may be different kinds of requirements regarding the synchronization of the producer and the consumer. A common requirement is that the tasks are synchronized so that the producer is required to wait when the buffer is full, and the consumer is required to wait when the buffer is empty. In addition, the buffer must be handled so that mutual exclusion is ensured, i.e. so that only one task at a time is allowed to use the buffer.

There may be more than one producer, and/or more than one consumer, communicating through a shared buffer.

Additional information regarding the producer and consumer problem is found in Section 4.2 in [RT].

Buffers

A buffer can be declared as

```c
/* 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;
```
Protecting a buffer

The Producer task and the Consumer task both contain critical regions, which are defined by the program code where the buffer is used.

A mutex can be used for protection of the buffer. The mutex can be declared as

```c
/* mutex for protection of the buffer */
pthread_mutex_t Mutex;
```

and initialised as

```c
/* initialise mutex */
pthread_mutex_init(&Mutex, NULL);
```

Conditional critical regions

Critical regions may be associated with conditions, expressing requirements regarding task synchronization, e.g. that a producer may not enter its critical region when the buffer is full, and that a consumer may not enter its critical region when the buffer is empty.

Critical regions which are associated with conditions are often referred to as conditional critical regions. A common requirement on a conditional critical region is that there is functionality which makes a task wait if a given condition is satisfied. There must also be functionality 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.

The conditional critical regions are seen in the function `put_item`, as

```c
/* put_item: stores item in the buffer */
void put_item(char item)
{
    /* reserve buffer */
pthread_mutex_lock(&Mutex);

    /* --- Here we need a mechanism, letting the --- */
    /* --- task wait as long as the buffer is full --- */

    /* store item in buffer */

    Buffer_Data[In_Pos] = item;
    In_Pos++;
    if (In_Pos == BUFFER_SIZE)
    {
        In_Pos = 0;
    }
```
Count++;

/* --- Here we need a mechanism for activation --- */
/* --- of waiting tasks. The tasks are waiting --- */
/* --- due to the buffer being empty or full at the --- */
/* --- time of their last access to the buffer --- */

/* release buffer */
pthread_mutex_unlock(&Mutex);
}

and in the function get_item, as

/* get_item: read an item from the buffer */
char get_item(void)
{
    /* item to read from buffer */
    char item;
    /* reserve buffer */
    pthread_mutex_lock(&Mutex);

    /* --- Here we need a mechanism, letting the --- */
    /* --- task wait as long as the buffer is empty --- */

    /* read item from buffer */

    item = Buffer_Data[Out_Pos];
    Out_Pos++;
    if (Out_Pos == BUFFER_SIZE)
    {
        Out_Pos = 0;
    }
    Count--;

    /* --- Here we need a mechanism for activation --- */
    /* --- of waiting tasks. The tasks are waiting --- */
    /* --- due to the buffer being empty or full at the --- */
    /* --- time of their last access to the buffer --- */

    /* release buffer */
    pthread_mutex_unlock(&Mutex);

    /* return the value read */
    return item;
Condition variables

Condition variables can be used, together with mutexes, for implementing conditional critical regions. A condition variable has three operations: one operation for initialisation, and two operations denoted Await and Cause.

A condition variable can be regarded as associated with a mutex. This mutex is used to ensure mutual exclusion for the conditional critical region.

The Await operation is used when a process shall wait, as a result of a condition being satisfied. In pthreads a mutex will be associated with the condition variable when the Await operation is invoked.

The Cause operation is used when waiting processes shall be activated. This operation is used when shared data have been modified, in order to let waiting processes re-evaluate their conditions, to determine if they are allowed to enter the critical region.

Condition variables are available in several real-time operating systems and they are also available when programming with pthreads, e.g. in Linux.

Condition variables are described in Section 6.2 in [RT].

\[\text{https://computing.llnl.gov/tutorials/pthreads/}\]