Synchronization

Real-Time & Embedded Systems

Synchronization in real-time systems

Synchronization: the run-time overhead?

• In the potential method pointed for simple data-structures:

\[
\begin{align*}
\text{int } & i = 0; \\
\text{int } & j = 0; \\
\text{int } & k = 0;
\end{align*}
\]

• Depending on the hardware and the compiler, it might be atomic, it might be not.

• Handling certain integers on a 32-bit System will not be atomic, but perhaps it is on 64 bits.

• Any manipulations on the main memory will not be atomic, but perhaps it is a register.

• Broken down to a load-operate-store cycle, the operation will not be atomic, but perhaps the processor supplies atomic operations for the actual case.

• Assuming all ‘perhapses’ are applying: how to expand this code?

• Measures for synchronization are required!

Synchronization by flags

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

• e.g. assigning two values (not under the control of any thread) to a memory cell simultaneously:

\[
\text{lock } i = 0; \quad \text{lock } j = 0; \quad \text{lock } k = 0;
\]

will result in: 

\[
i \neq 0 \quad j \neq 0 \quad k \neq 0
\]

and no other value is ever observable.

Synchronization by semaphores

Assuming further that there is a shared memory between two processes:

• A set of processes agree on a read-only atomic variable operating as a flag to indicate synchronization conditions.

\[
\text{var } 
\text{signal } 
\text{mutex } 
\text{semaphore } 
\text{condition synchronization }
\]

\[
\text{var mutex: semaphore } := 1;
\]

\[
\text{var mutex: semaphore } := 1;
\]

\[
\text{var mutex: semaphore } := 1;
\]

Sequence of operations:

\[
\text{[A] } \text{[X] } \text{[B] } \text{[Y]}
\]

Synchronization in real-time systems

Synchronization: the run-time overhead?

• In the potential method pointed for simple data-structures:

\[
\begin{align*}
\text{int } & i = 0; \\
\text{int } & j = 0; \\
\text{int } & k = 0;
\end{align*}
\]

• Depending on the hardware and the compiler, it might be atomic, it might be not.

• Handling certain integers on a 32-bit System will not be atomic, but perhaps it is on 64 bits.

• Any manipulations on the main memory will not be atomic, but perhaps it is a register.

• Broken down to a load-operate-store cycle, the operation will not be atomic, but perhaps the processor supplies atomic operations for the actual case.

• Assuming all ‘perhapses’ are applying: how to expand this code?

• Measures for synchronization are required!

Types of semaphores:

• General semaphores (counting semaphores): non-negative number (range limited by the system)

• Binary semaphores: restricted to 0 (0) Multiplication (signal) calls have the same effect than 1 call.

• Synchronized methods (counting semaphores): and after semaphore test and signal (which are treated as atomic operations)

• Priorities: semaphore: the increment (increment) and decrement (decrement) for the semaphore is specified at a particular level (level).
Monitors with condition synchronization

backup monitor

procedure take (var I : integer); begin (* initialisation *)
  region critial_buffer_region

  loop
    region critial_buffer_region

without Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;

  private

    in

    general:

    

    procedure Set_False (S : in out Suspension_Object);

    procedure Set_False (S : in out Suspension_Object);

    type Suspension_Object is limited private;

    function Current_State (S : Suspension_Object) return Boolean;

    function Current_State (S : Suspension_Object) return Boolean;

    procedure Suspend_Until_True (S : in out Suspension_Object);

    procedure Suspend_Until_True (S : in out Suspension_Object);

end Ada.Synchronous.Task_Control;

end ADA;

procedure B (var I : integer); begin (* initialisation *)
  region critial_buffer_region

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;

procedure A (var I : integer); begin (* initialisation *)
  region critial_buffer_region

begin

end region

end loop;

end ADA;
• to re-evaluate its entry condition
• to wait for the lock to be released

☞
• There are no explicit conditional variables.
☞
• Methods outside the monitor-object can synchronize at this object.

stopWrite();
stopRead();
// read the shared data only

Intermediate (free) = true;
BEGIN
IF busy THEN WAIT (free) END;
END;
END:
BEGIN
busy := false;
END.

Synchronization
Monitors in ‘C’ / POSIX

typedef … pthread_mutex_t;typedef … pthread_mutexattr_t;
typedef … pthread_cond_t;typedef … pthread_condattr_t;

Interface
Monitors in Modula-1

PRIVATE: boolean busy; SIGNAL free;
PROCEDURE allocate;
PROCEDURE deallocate;

BEGIN      IF busy THEN WAIT (free) END;
END;

Synchronization between POSIX-threads:

int pthread_mutex_init      (      pthread_mutex_t     *mutex,                              const pthread_mutexattr_t    *attr);int pthread_mutex_destroy    (      pthread_mutex_t      *mutex);

int pthread_mutex_lock      (      pthread_mutex_t     *mutex);
int pthread_mutex_unlock    (      pthread_mutex_t     *mutex);
int pthread_mutex_trylock   (      pthread_mutex_t     *mutex);
int pthread_mutex_timedlock (      pthread_mutex_t     *mutex,                              const struct timespec     *abstime);

int pthread_cond_init       (      pthread_cond_t      *cond,                              const pthread_condattr_t    *attr);int pthread_cond_destroy    (      pthread_cond_t      *cond);

int pthread_cond_signal     (      pthread_cond_t      *cond);
int pthread_cond_wait       (      pthread_cond_t      *cond,                              pthread_mutex_t     *mutex);
int pthread_cond_timedwait  (      pthread_cond_t      *cond,                              pthread_mutex_t     *mutex,                              const struct timespec     *abstime);

Monitors in ‘C’ / POSIX

int pthread_mutex_init      (      pthread_mutex_t     *mutex,                              const pthread_mutexattr_t    *attr);int pthread_mutex_destroy    (      pthread_mutex_t      *mutex);

int pthread_mutex_lock      (      pthread_mutex_t     *mutex);
int pthread_mutex_unlock    (      pthread_mutex_t     *mutex);
int pthread_mutex_trylock   (      pthread_mutex_t     *mutex);
int pthread_mutex_timedlock (      pthread_mutex_t     *mutex,                              const struct timespec     *abstime);

int pthread_cond_init       (      pthread_cond_t      *cond,                              const pthread_condattr_t    *attr);int pthread_cond_destroy    (      pthread_cond_t      *cond);

int pthread_cond_signal     (      pthread_cond_t      *cond);
int pthread_cond_wait       (      pthread_cond_t      *cond,                              pthread_mutex_t     *mutex);
int pthread_cond_timedwait  (      pthread_cond_t      *cond,                              pthread_mutex_t     *mutex,                              const struct timespec     *abstime);

monitor calls:

void increment_counter();

void increment_counter() {
  if (mon.ch == current) {
    mon.ch = next;
    mon.counter := mon.counter + 1;
  }
}

Monitors in Real-time Java

Java provides two mechanisms to construct monitors:

Synchronized methods and code blocks:
  All methods and code blocks which are using the synchronized tag are synchronized
  Notify methods and wait methods can be used only in synchronized regions and are
  non-blocking.

Monitors in Real-time Java

Synchronized methods and code blocks:
  All methods and code blocks which are using the synchronized tag are synchronized.
  Notify methods and wait methods can be used only in synchronized regions and are
  non-blocking.

Monitors in Real-time Java

Synchronized methods and code blocks:
  All methods and code blocks which are using the synchronized tag are synchronized.
  Notify methods and wait methods can be used only in synchronized regions and are
  non-blocking.

Monitors in Real-time Java

Synchronized methods and code blocks:
  All methods and code blocks which are using the synchronized tag are synchronized.
  Notify methods and wait methods can be used only in synchronized regions and are
  non-blocking.
Synchronization

Monitors in Real-time Java

- public void StartWrite () throws InterruptedException
  
  synchronized (this)
  
  {             
    if (writing || waitingWriters > 0) {               
      waitingWriters++;               
      OkToWrite.wantToSleep = true; 
      OkToWrite.notify () ; // wakeup one writer
      } 

  
  if (writing | waitingWriters > 0) {               
    waitingReaders++;               
    OkToRead.wantToSleep = true; 
    OkToRead.notify (); 
  } } 
  
  public synchronized void StopWrite()
  
  {             
    writing = false; 
    notifyAll (); 
  } } 
  
  public synchronized void StopRead ()
  
  {             
    readers--;            
    if (readers == 0 & waitingWriters > 0) {               
      waitingWriters--;               
      OkToWrite.notify () ; // wake one writer
    } 

  
  {            
    OkToRead.notifyAll (); 
  } 

"..."
Synchronization by protected objects in Ada95

Barrier evaluation and task activations:
- on a protected entry, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).
- on having a protected procedure or entry, related barriers which task spawned are evaluated (only those parts of the barrier which might have been altered by this procedure or entry which might have changed since the last task evaluation).

Barriers are not evaluated while inside a protected object or on having a protected function.

Further refinements on task control by:
- Entry families (protected entry declarations can contain a discrete subtype selector, which can be evaluated by the barrier, other parameters cannot be evaluated by barriers) and implements an array of protected entries.
- Response facility (protected operations can use 'receive' to collect tasks to either internal, external, or private entries. The current protected operation is finished and the lock of the protected object is released.
- Guards (barriers) (which internal barrier points on the protected object, but caches employed as destinations for request operations).

Shared memory based synchronization

General
- Critical:
  - level of abstraction
  - concurrency/distributed concepts
  - support for consistency and correctness validations
  - error sensitivity
  - reliability
- Efficiency:
  - no more, no less, …

POSIX
- all low level contracts available
- no connection with the kernel data structures
- without constraints introduced by release some semantics of conditional variables (context, signals,

Message-based synchronization

Synchronization model
- Asynchronous
- Synchronous
- Remote execution
- Addressing (name space)
- Direct communication
- Broad-based communication

Message structure
- arbitrary
- traceable to basic types
- originates from any type of communication
Synchronization

Selective waiting in Occam2

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
  or
  when Condition3 => accept do ...
  or
  when Condition4 => accept do ...
end select;

Basic forms of selective synchronization

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Non-determinism in selective synchronizations

formal synchronized selection

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Forms of selective waiting

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Conditional & timed entry-calls

select
  Light Monitor.JOBstrftime( True, Light := True, [] else := False); &
end select;

Shared memory based synchronization

Addition to the set of available synchronization mechanisms

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Message-based selective synchronization in Ada95

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Guarded select-or-terminate

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Message-based selective synchronization in Ada95

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Select-or

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Conditional & timed entry-calls

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Message-based selective synchronization in Ada95

select
  when Condition1 => accept do ...
  or
  when Condition2 => accept do ...
end select;

Shared memory based synchronization