Real-time Systems D0003E
Lecture 14:
Real-time languages
(Burns & Wellings ch. 8 & 9)

Overview
- A little lab advice...
- Generic parking
- Synchronization
  - semaphores
  - readers-writers problem
  - monitors
  - guards (ccr's)
- Concurrency
  - java concurrency model

Lab advice
- open com port:
  - int COM1;
  - COM1=open("/dev/ttyS0", O_RDWR); // open com port
- have a look at:
  - termios
  - tcgetattr(..., ...);
  - cfsetispeed(..., ...);
  - cfsetospeed(..., ...):

Lab advice
- Blocked wait for both keyboard and com1:
  FD_set rfds; FD_ZERO(&rfds); // empty set
  FD_SET(0, &rfds); // include keyboard
  FD_SET(COM1, &rfds); // include com1
  select(4, &rfds, NULL, NULL, NULL); // wait
  if(FD_ISSET(0, &rfds)) { // handle keypress
    // handle keypress...
  }

Readers-writers problem
- Very common problem in concurrent programming
- Shared resource
- allow several concurrent "readers"
- allow only one "writer" (and no reader)
- A reader/writer is a thread
- How to solve/implement?

Readers-writers solution
- global:
  - readcount=0
- set n=1, w=1
- reader:
  - wait(x)
  - readcount++
  - if readcount=1 then wait(x)
  - signal(x)
    //perform reading
  - wait(x)
- writer:
  - wait(w)
    //perform writing
  - signal(w)

Problems with this?
Readers-writers problem
problems with solution?
- If...
  • reader is reading
  • allow another reader
  • don’t allow writer
  • add a long queue of readers and writers...
  • let all readers in
  • never let writers in
  • always a reader reading
- Hmm...
  • not very good
  • We have starvation of writers

Readers-writers solution
writers preference

global:
readcount=0, writecount=0

reader:
guaranteed writers preference
allow writer?
wait(r)
wait(m1)
wait(m3)
readcount++
if readcount==1
then wait(w)
signal(m1)
signal(r)
signal(m3)
//perform reading
wait(m1)
readcount--
if readcount==0
then signal(w)
signal(m1)

writer:
wait(m2)
writecount++
if writecount==1
then wait(r)
signal(m2)
wait(w)
//perform writing
signal(w)
wait(m2)
writecount--
if writecount==0
then signal(r)
signal(m2)

entry protocol
exit protocol
guarantees writers preference
mutex over rcount
allow reader?
Can safely share thread-local variables, as reactions are automatically serialized

Re: generic parking

Instead of encoding reactive objects as follows...

- Fully generic, can wait for any combination of events
- Can only share global variables, reactions must be manually serialized via a mutex or semaphore

Re: generic parking

We are often forced to resort to the following pattern:

- Not generic, will only wait for some specific events
- Can only share global variables, reactions must be manually serialized via a mutex or semaphore

Criticism of semaphores

- Elegant, but error-prone low-level primitive
- The two uses of semaphores, mutual exclusion and conditional synchronization, are not distinguished
- There is nothing that binds a mutual exclusion semaphore to the shared variables it protects
- A misplaced or forgotten semaphore can lead to total program corruption or deadlock
- Historically important, but mostly inadequate for larger-scale concurrent programming
- Can be used to support other primitives

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 monitor BB

```
monitor boundedBuffer {
    int count = 0, head = 0, tail = 0;
    T buf[SIZE];
    void put(T item) {
        if (count == SIZE) wait(notFull);
        buf[head] = item;
        head = (head + 1) % SIZE;
        count++;
        signal(notEmpty);
    }
    void get(T *item) {
        if (count == 0) wait(notEmpty);
        *item = buf[tail];
        tail = (tail + 1) % SIZE;
        count--;
        signal(notFull);
    }
}
```

These variables are only visible within methods put and get.

Only one thread can be running put or get at any time (the others are queued).

However, a thread can temporarily give up control of the monitor by calling wait.

These variables are only visible within methods put and get

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: cond vars. 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

Concern: who gets to run after a signal?

- The ones "outside of the monitor"
- Let new threads in
- The ones that have called wait?
- Release threads blocked in the monitor
- This is often what we want to do...
- Different monitor variants take different routes...

Monitors and POSIX

- Although C doesn't offer any explicit monitor construct, the POSIX library does provide the building blocks for constructing a monitor
- We have already seen one part of this support: the mutex variables
- The other part is a mechanism for building condition variables:
  - A datatype: pthread cond_t
  - An init operation: pthread cond init(&cv, &attr)
  - A signal operation: pthread cond signal(&cv)
  - A wait operation: pthread cond wait(&cv, &mut)
- Note how the wait call specifies the mutex to be temporarily released

Java concurrency

- The Java threading model views threads and objects as orthogonal concepts (objects are not concurrent, a thread does not encapsulate state)
- However, Java uses objects to represent threads - (just like POSIX/C uses variables of type pthread_t)
- (class MyThread extends Thread {
  public void run() {
    // your code
  }
})
- MyThread t1 = new MyThread();
  t1.start();
  // continues in parallel with code in t1.run()
Java and monitors

- Note that methods in object t1 can still be called from the main thread in a normal fashion. Likewise, thread t1 can freely call methods in any object.
- However, to solve the inevitable synchronization problems, Java takes the route that every object may also double as a monitor!
- Restriction 1: to achieve mutual exclusion, each method of an object must be prepended with the keyword synchronized.
- Restriction 2: only one condition variable per object (monitor) is supported, identified with the object itself.

A Java BB

```java
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;
    }
}
```

Data must be marked private to be truly encapsulated.

Criticism of monitors

- Monitors elegantly solve the mutual exclusion problem, but...
- the conditional synchronization problem is still dealt with using low-level variables.
- All criticisms surrounding semaphores apply to condition variables as well.
- The use of nested monitor calls (monitors that call other monitors) is inherently problematic - only the innermost monitor will open up while the caller waits.
- Can a higher-level alternative to condition variables be devised?

Guards

- 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.

Guards

- Guards also called Conditional critical regions (CCR).
- A critical region with an extra condition.
- We wait (blocked) until the condition is true.
- Very powerful.
- Such a construct can be found in Ada, in the form of guards.

Ada

- Ada provides two relevant constructs for interprocess communication: the protected objects (a monitor-like construct) and a general message-passing mechanism.
- Threads in Ada are called tasks, and are introduced by declaration (imagine a keyword in C indicating that a function body should start executing as a thread, without any call to pthread_create).
- Monitor methods as well as message-passing endpoints are called entries in Ada.
- Ada makes a clear distinction between the interface of a protected object or a task, and its implementation body.
An Ada protected object BB

```ada
protected BoundedBuffer is
  entry put (item : in T;)
  entry get (item : out T;)
  private
    count : Integer := 0;
    head : Integer := 0;
    tail : Integer := 0;
    buf : array (0..SIZE-1) of T;
  end
protected body BoundedBuffer is
  entry put (item : in T) when count /= SIZE is
    begin
      buf(head) := item;
      head := head + 1 mod SIZE;
      count := count + 1;
    end put;
  entry get (item : out T) when count /= 0 is
    begin
      item := buf(tail);
      tail := tail + 1 mod SIZE;
      count := count - 1;
    end get;
end BoundedBuffer;
```

Things to note

- The Ada type system limits the guards to simple boolean expressions.
- In principle the guards should be constantly re-evaluated.
- In practice a good compiler will only insert evaluation code when an entry is called (if the guard depends on an argument) or when an entry exits (if the guard depends on the local state).
- Note that the guards must be run under mutual exclusion if they depend on local state (the locks are handled by the compiler).
- The good thing is that the separation between real code and entry conditions is made explicit.

Message-passing in Ada

- Ada's message-passing mechanism is characterized by:
  - Full type safety (no need for type casts)
  - Selective parking (blocking for several messages)
  - Message guards (like for the protected objects)
  - Sending a message just amounts to calling an entry in a task's exported interface.
  - Receiving a message is done by an accept statement, that mentions the name of an exported entry.

- An infinite loop thread

  ```ada
  task body BoundedBuffer is
    count : Integer := 0;
    head : Integer := 0;
    tail : Integer := 0;
    buf : array (0..SIZE-1) of T;
    begin loop
      select
        when count < SIZE =>
          accept put (item : in T) do
            buf(head) := item;
            head := head + 1 mod SIZE;
            count := count + 1;
          end put;
        or when count > 0 =>
          accept get (item : out T) do
            item := buf(tail);
            tail := tail + 1 mod SIZE;
            count := count - 1;
          end get;
      end select;
    end loop;
  end BoundedBuffer;
  ```

Remarks

- Ada's tasks closely resemble our reactive objects:
  - Entries are sequential, thus local task variables are automatically protected from concurrent access.
  - Any number of unique entries (=methods) can be defined for a task.
  - The select statement can also contain delays and interrupt handlers in parallel with ordinary entries.
  - Asynchronous methods can be mimicked by inserting arbitrary code after an (empty) entry.
  - It seems like select is the generic parking operation we've been looking for!

Criticism of select

- However, notice that there's nothing that guarantees that a task will respond to its entries just because it is parked in a select. The programmer may just as well select not to include a entry.
- As a result, an entry call might freeze indefinitely, until the event occurs that causes the receiving task to do a select that accepts it — i.e., every entry call becomes a potentially parking operation.
- A similar effect is caused by guards, and (in other systems) by condition variables and semaphores.
The perils of silent parking

- Excluding entries (or putting callers on hold using condition variables or semaphores) makes it temptingly easy to write software services like the Bounded Buffer, since calls occurring “at the wrong time” can essentially be ignored.
- However, the downside is that writing clients is made correspondingly harder - if a client must react within a deadline it cannot risk becoming parked on a put() in the middle of a reaction.
- In fact, the whole notion of a reaction becomes blurred if parking may silently occur anywhere - where does one reaction start and where does another one stop?

Our notion of a reaction:

- Avoiding indefinite blocking inside reactions is essential to obtaining predictable reaction times, hence programs that use Ada select / condition variables / semaphores must be written very restrictively.
- The model enforced by our reactive objects directly capture the desired program structure!
- See lecture 9 examples
- Example of reactive objects for a bounded buffer implementation that does not rely on silent parking.

A reaction that parks internally... hmm...

Avoiding indefinite blocking inside reactions is essential to obtaining predictable reaction times, hence programs that use Ada select / condition variables / semaphores must be written very restrictively.

The model enforced by our reactive objects directly capture the desired program structure!

See lecture 9 examples

Example of reactive objects for a bounded buffer implementation that does not rely on silent parking.