Monitors & Critical Regions

May 10, 2000

1 Introduction

Given that we have semaphores, why do we need another programming language feature for dealing with concurrency?

- Semaphores are low-level.
- Omitting a wait breaches safety — we can end up with more than one process inside a critical section.
- Omitting a signal can lead to deadlock.
- Semaphore code is distributed throughout a program, which causes maintenance problems.
- Better to use a high-level construct/abstraction.

2 Critical Regions

- A **critical region** is a section of code that is always executed under mutual exclusion.
- Critical regions shift the responsibility for enforcing mutual exclusion from the programmer (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
  1. Variables that must be accessed under mutual exclusion.
  2. A new language statement that identifies a critical region in which the variables are accessed.

Example: *This is only pseudo-Pascal-FC — Pascal-FC doesn't support critical regions*

```
var
v : shared T;

...

region v do
    begin
    ...
    end;
```

All critical regions that are 'tagged' with the same variable have compiler-enforced mutual exclusion so that only one of them can be executed at a time:

```
Process A:
    region V1 do
        begin
        { Do some stuff. }
        end;

    region V2 do
        begin
        { Do more stuff. }
        end;
```

```
Process B:
    region V1 do
        begin
        { Do other stuff. }
        end;
```

Here process A can be executing inside its `V2` region while process B is executing inside its `V1` region, but if they both want to execute inside their respective `V1` regions only one will be permitted to proceed.

Each shared variable (`V1` and `V2` above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

3 Conditional Critical Regions

Critical regions aren't equivalent to semaphores. As described so far, they lack condition synchronization. We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can't do this with critical regions.

Conditional critical regions provide condition synchronization for critical regions:

```
region v when B do
    begin
    ...
    end;
```

where `B` is a boolean expression (usually `B` will refer to `v`).

Conditional critical regions work as follows:

1. A process wanting to enter a region for `v` must obtain the mutex lock. If it cannot, then it is queued.
2. Once the lock is obtained the boolean expression `B` is tested. If `B` evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest `B`.

3.1 Implementation

Each shared variable now has two queues associated with it. The **main queue** is for processes that want to enter a critical region but find it locked. The **event queue** is for the processes that have blocked because they found the condition to be false. When a process leaves the conditional critical region the processes on the event queue join those in the main

queue. Because these processes must retest their condition they are doing something akin to busy-waiting, although the frequency with which they will retest the condition is much less. Note also that the condition is only retested when there is reason to believe that it may have changed (another process has finished accessing the shared variable, potentially altering the condition). Though this is more controlled than busy-waiting, it may still be sufficiently close to it to be unattractive.

3.2 Limitations

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation. Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

4 Monitors

- Consist of private data and operations on that data.
- Can contain types, constants, variables and procedures.
- Only the procedures explicitly marked can be seen outside the monitor.
- The monitor body allows the private data to be initialized.
- The compiler enforces mutual exclusion on a particular monitor.
- Each monitor has a **boundary queue**, and processes wanting to call a monitor routine join this queue if the monitor is already in use.
- Monitors are an improvement over conditional critical regions because all the code that accesses the shared data is localized.

4.1 Condition synchronization

Just as plain critical regions needed to be extended to conditional critical regions, so do monitors (as described thus far) need to be extended to make them as applicable as semaphores. **Condition variables** allow processes to block until some condition is true and then be woken up:

```
var
c : condition;
```

Two operations are defined for condition variables:

(i) **delay**

(ii) **resume**

4.1.1 Delay

- Similar to the semaphore wait operation.
- `delay(c)` blocks the calling process on `c` and releases the lock on the monitor.

4.1.2 Resume

- Similar to the semaphore signal operation.
- `resume(c)` unblocks a process waiting on `c`.
- *resume* is a nop if no processes are blocked (compare to *signal*, which always has an effect).

But once *resume* has been called we have (potentially) two processes inside the monitor:

- The process that called *delay* and has been woken up.
- The process that called *resume*.

Solutions:

- **resume-and-continue** The woken process must wait until the one that called *resume* releases the monitor.
- **immediate resumption** The process that called *resume* must immediately leave the monitor (this is what Pascal-FC uses).

Resume-and-continue means that processes that call *delay* must use

```
while not B do          instead of          if not B then
    delay(c);                                   delay(c);
```

because a resumer might carry on and alter the condition after calling *resume* but before exiting the monitor.

4.2 Immediate resumption variants

1. Resume and exit: The resumer is automatically forced to exit the monitor after calling *resume*.
2. Resume and wait: The resumer is put back on the monitor boundary queue. When it gets back in, it is allowed to continue from where it left off.
3. Resume and urgent wait: The resumer is put on a second queue that has priority over the monitor boundary queue.

4.3 Nested monitor calls

A **nested monitor call** is when a procedure in monitor A calls a procedure in a different monitor, say monitor B. They are potentially troublesome: think about what happens if the procedure in monitor B contains a *delay* statement.

There are several approaches that can be taken when faced with this situation:

1. Retain the lock on A when calling the procedure in B, release the lock on B when calling *delay* in B.
2. Retain the lock on A when calling the procedure in B, release both locks when calling *delay* in B.
3. Release the lock on A when making a nested call.
4. Ban nested calls altogether!

Pascal-FC chooses the first of these alternatives.