Monitors: An Operating System Structuring Concept

C.A.R. Hoare
presented by: Ryan O'Connor <rjo@cs.ubc.ca>

September, 21st 2009

Motivation

- Operating systems are designed to share resources amongst competing, unpredictable user programs
- Therefore resource scheduling algorithms are an inportant area of operating system designs
- Idea: create seperate scheduler for each class of resources

Monitors

Monitors are a scheduler with local administrative data and procedures. Monitors have the following properties:

- Any running program may call the monitors' procedures
- Only one program may call a monitor procedure at any given time
- Monitors' procedures should only access variables local to the monitor
- Monitors' local variables should only be accessed from within the monitor

Condition Variables

A scheduler may need to pause a program that wishes to access a resource that is not currently available. To achieve this Hoare introduces the *Condition Variable*.

Programs can:

- *wait* on a condition varaible, which pauses the program.
- *signal* on a condition variable, which immediately resumes a waiting program. No third program can enter the monitor between one program's call to signal and another program's return from wait.

Monitor and Semaphore Equivalence

A monitor that schedules access to a single resource simulates a semaphore. Similarly a monitor can be constructed out of semaphores (overview):

- add semaphore to serialize access to monitors' procedures [*mutex*]
- add semaphore to ensure no third program may intervene between a signal and the resumption of a waiting program
- add semaphore for each condition variable within the monitor

Example: Scheduled Waits

Idea: add parameter when waiting on a condition variable to give the monitor control over which program is resumed by a signal. Hoare then demonstrates a monitor that suspends a program for a given amount of time:

- monitor updates a time counter on each *tick*
- programs call *wakeme* which adds the timecounter value to the requested time and waits with the result as a parameter
- must call signal after each tick

Example: Buffer Allocation

Hoare discusses a solution to buffer allocation, where two or more procucers must allocate buffers.
Problem: If both producers fill buffers at the same rate, but one has a slower consumer, that producer will eventually 'hoard' the buffers.
Solution

- count the number of buffers currently help by each producer and perform a scheduled wait using the count.
- a buffer will then be allocated to the producer with the fewest number of buffers

Example: Readers and Writers

Problem:

- must prevent access to object between readers and writers. Many may read, only one may write.
- writers must not indefinately block readers
- readers must not indefinately block writers

Solution: create a monitor with...

- a varaible to count readers and a boolean to indicate a write in progress
- two condition variables, to indicate that a program may read or write respectively

Conclusion

- "Monitors can be reccomended without reservation" for OS design
- boolean wait conditions may seem attractive, but they are expensive.
- condition variable give program better control over efficiency and scheduling

Monitor Design Principles

- avoid bad scheduling decisions rather than actively seeking optimal decisions
- do not try to present a virtual machine better than the hardware
- use preemptive techniques
- avaid fixed priorities
- aim for graceful degredation
- assume user programs will obey sensible rules regarding monitor calls

Big Picture

Contributions:

- Monitors for OS design
- Seperate resource scheduling algorithms when building an OS
- Condition Variables

Questions

- There are some languages in which specific contracts can be formally stated, such as class invariants. Those invariants can be automatically checked before and after each procedure invocation. Are there any languages where these invariants are also checked automatically when a thread waits on a condition variable?
- Does any language support automatic creation of condition variables, for instance by (1) having the waiter state an actual condition to wait for, rather than a variable name of a condition variable, and (2) having potential signallers check whether each waitable condition is true on every exit from the monitor and signal a waiter if so, rather than having to manually invoke signal() to wake up a waiter at the appropriate time?

Questions

- Not to nitpick, but this has me in a bit of confusion - In the Buffer allocation example, shouldn't the buffer b be added to the freepool in the release procedure? In both the examples, it uses freepool := freepool -b, which doesn't seem to make any sense when releasing a buffer. Or am I missing something?
- Is there any advantage of the signaled thread being given priority over the signaling thread, as opposed to the more intuitive (in my opinion) mechanism that the signaling thread gets priority to finish before the signaled thread starts? It seems to add a fair bit of complexity without any particular discernible benefit.

Questions

- In principle, are there any differences between original monitor propose in this paper and Java synchronized object? What has been improved over these years.
- Are the 8 principles mentioned at the end of this paper still hold? I'm interested in how are monitors used to manage resources in current operation systems?

Questions

- The paper states that "Significant improvements in efficiency may also be obtained by avoiding the use of semaphores, and by implementing conditions directly in hardware" Are there any systems that implement conditions in hardware like this?
- Why is having a signal as the last statement in a monitor a great simplification (aside from omitting two variables)?