Multi-Average Deadlock Prevention

Walter B. Gress V

Drexel University

Friday, June 7 2013

Mult-Average Deadlock Prevention is a method of processing a critical section without placing a spin lock, semaphore, monitor, or other mechanism to prevent deadlock. In fact it runs over the critical section multiple times on multiple threads, averaging the results to detect the correct configuration.

# introduction

Multi-Average Deadlock Prevention uses a statistical means where the idea is to average over multiple runs, a critical section and its associated variables, instead of simply locking the critical section. The process depends on the theory that over time, or for multiple identical threads, the threads will “average-out” , or that is, the average values for the variables in the critical section will be correct should we plot them over processes.

# present methods of deadlock prevention

## Spinlocks

A spin lock is a method of busy waiting. It causes the thread to wait, consistently polling the locked critical section until it is available.

## Semaphores

They are similar to spinlocks, but semaphores increment or decrement according to the resources being accessed. If a resource is requested the semaphore is decremented. If the semaphore is decremented to negative, then the requester is put on hold.

## Monitors

Monitors use a combination of mutexes and conditions to implement synchronization.

# Advantages AND DISADVANTAGES

With the average deadlock prevention there is no delay in processing. Sections of code are processed without having to wait for other processes to execute their code.

# An example

Here's a real life example. Alice and Bob are poor roommates, and they can't buy a microphone and a pair of headphones each. They decide to share – Alice buys the headphones, Bob buys a microphone. They decide on using the simplest possible tiebreaking rule: "calling it" – whoever says first "I need the microphone" gets the microphone; the same for the headphones. One day, they both decide independently to record a song; both Alice and Bob need the microphone to record, and the headphones to hear what they're recording. They both [race](https://en.wikipedia.org/wiki/Race_condition) to get exclusive locks on both resources. As it happens, Alice gets to call it on the microphone, but Bob calls it first on the headphones. As soon as this happens, they're both in a deadlock: Alice has a lock on the microphone but waits for a lock on the headphones, but Bob doesn't want to give up on his lock on the headphones because he's waiting for a lock on the microphone.

Moving onto the source code level, a deadlock can occur even in the case of a single thread and one resource (protected by a [mutex](https://en.wikipedia.org/wiki/Mutex)). Assume there is a function *f1* which does some work on the resource, locking the mutex at the beginning and releasing it after it's done. Next, somebody creates a different function *f2* following that pattern on the same resource (lock, do work, release) but decides to include a call to *f1* to delegate a part of the job. What will happen is the mutex will be locked once when entering *f2* and then again at the call to *f1*, resulting in a deadlock if the mutex is not [reentrant](https://en.wikipedia.org/wiki/Reentrant_mutex) (i.e. the plain "fast mutex" variety). (Wikipedia)

# Theory

Assume the example above. Both Alice and Bob are running on their own threads. Both need the headphones and mic. The critical section is where Alice and Bob request the headphones and mic. This time however, Alice and Bob are duplicated, that is, there is k Alice threads generated and k Bob threads generated running at the same time. The pairs run simultaneously. In addition the critical variables are duplicated, one set for each thread pair. Each pair, Alice and Bob, operate on their own set of critical variables together.

The result of these threads is examined and the most common value for the critical variables is used.

The assumption is that as the threads run deadlock will occur occasionally. But deadlock will not occur over all thread-pairs. Thus the result of the multiple threads will always hold at least one correct answer.