Sammanfattning EDA040

Felix Mulder

17 oktober 2013

1 Deadlock Analysis

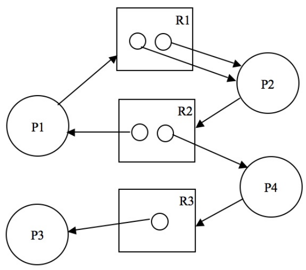
Resource allocation graphs are used to determine if a program can deadlock. For a program to end up in a deadlock there are a few requirements.

- Mutual exclusion: at least one resource is held in a non-shareable mode.
- Hold and wait: there must exist a process that is holding at least one resource and simultaneously waiting for resources that are held by other processes.
- No preemption: resources cannot be preempted; the resource can only be released voluntarily by the resource holding it.
- Circular wait: There must exist a set of processes waiting for each other in a circular structure. I.e: p1 waits for p2, p2 waits for p3, p3 waits for p1.

To draw a resource allocation graph from source code:

1. Draw boxes for each resource.
2. For each thread (i) and line (j), draw a bubble with T_{ij}. If a thread takes, then draw a line to the resource. For $T_{i(j+1)}$ draw a line from the resource to the thread.
3. If T_{ij} only emits or only absorbs arrows, you don't have to keep it in the graph.
4. For resources that exist as multiple instances, draw dots inside the resource. If a cycle exists containing a multiple instance resource, then it may be a false cycle.

Cycles in the graph indicate the possibility of deadlocks.



2 Process synchronization

The critical section problem could be solved simply by disallowing interrupts on a single core cpu. With multiple cores, however, disabling these interrupts will be too time consuming.

2.1 Dekker's Algorithm

Dekker's algorithm solves the process synchronization problem with busy waits. Meaning: using the below specified code results in a correctl handling of critical areas. Alas, the threads spend CPU cycles in the while loop, needlessly. If we implement Dekker's we should compliment it with wait/notify functionality. Without this improvement the semaphores can be referred to as spinlocks. The only advantage with using spinlocks is that there is no context switch required.

Listing 1: Dekker's Algorithm

```
public class Dekkers extends MutualExclusion {
  public Dekkers () {
    flag[0] = false;
    flag[1] = false;
    turn = TURN_0;
  }

  public void enteringCriticalSection (int t) {
    int other;

    other = 1 - t;

    flag[t] = true;
    turn = other;

    while ((flag[other] == true) && (turn == other)) {
      Thread.yield();
    }
  }

  public void leavingCriticalSection (int t) {
    flag[t] = false;
  }

  private volatile int turn;
  private volatile boolean[] flag = new boolean[2];
}
```

2.2 Race condition

A race condition is when multiple threads access and manipulate the same data concurrently, and where outcome of the execution depends on the particular order in which access takes place.

2.3 Mutual Exclustion

If thread T_i is executing in its critical section, then no other threads can be executing in their critical sections.

2.4 Progress

If no thread is executing in its critical section and there exist threads that wish to enter their critical sections, then only the threads not executing in their critical section get to partake in the process of deciding which thread gets to execute its critical section next.

2.5 Starvation

When some threads are allowed to execute and make progress, but others are left "starving."

2.6 Livelock

No thread makes progress, but they keep executing.

2.7 Bounded waiting

There exists a limit to the amount of times a thread will wait for other threads before its request to enter a critical area is granted. (This prevents starvation in a single thread.)

2.8 Drifting

The following piece of code will cause accumulative drift.

Listing 2: Drift example

```
while (!isInterrupted()) {
              sleep(100); foo.bar();
}
```

Sleep specifies a minimun time to sleep, and a context switch may have ocurred after sleep and before the method call thus drift is accumulated.

Listing 3: Drift fixed

```
long t = System.currentTimeMillis();
while (!isInterrupted()) {
  foo.bar();
  t += 100;
  long diff = t - System.currentTimeMillis();
  if (diff > 0) sleep(diff);
}
```

Even with this fix, sleep still causes a minimum busy wait.

2.9 volatile, transient keywords

Volatile means that the compiler is not allowed to cache the value of this variable. It should be updated before evaluation.

Transient means that the variable has no meaning outside of its current context if the variable is passed along with a serialized object over a network, it gets set to "null" or its equivalence.

3 Scheduling

3.1 Priority inversion phenomenon

Occurs when a low priority thread manages to lock a resource and this thread is then interrupted by a higher priority thread. When the thread requests the same resource, the lower thread blocks the higher thread and can thus resume its execution. (Despite being lower prioritized than the other thread.) If we called the highest prioritized thead A, and call the lowest Z. If Z blocks A, and Z is interrupted by a higher prioritized thread (M?) that doesn't share its resources, then this thread (M) also blocks A. This is called a *prioriy inversion* since Z and M will execute before A.

3.2 Priority inheritence protocol

The basic idea is to modify the priority of the tasks causing the blocking. In particular when Z blocks higher prioritized tasks, it temporarily inherits the highest priority of the blocked tasks. This prevents medium prioritized threads from preempting Z and prolonging the blocking duration.

3.2.1 Basic

Raises the priority of the low priority thread temporarily

3.2.2 Priority Ceiling Protocol

To bound the priority inversion phenomenon and preent the formation of deadlocks and chained blocking; PCP extends the priority inheritence protocol with a rule for granting a lock request.

When a job enters a critical section it receives the *priority ceiling* equal to the highest prioritized job able to access said resource, meaning that once it enters the critical region. This means that the only time it is interrupted is when a job with higher priority needs to run. (The interrupting job doesn't access the lower prioritized job's resource.)

If a job with higher priority than the currently running job in the semaphore tries to gain access, its priority is transferred to the lower prioritized job ensuring that a job won't be interrupted again by some job of said priority or lower.

1. Each semaphore is assigned a priority ceiling equal to the highest priority of jobs that can lock it.
2. The job with the highest priority gets to run first.
3. The job running locks a semaphore.
4. Another job tries to interrupt the currently running job, but if said job has a lower priority than the priority ceiling, the first job continues to run. If the interrupting job had a higher priority than the job running inside the semaphore, its priority is transferred to the currently running job. (Priority inheritence)
5. When no others jobs are blocked by the thread it resumes its original priority, i.e. its "nominal priority."

6. Priority inheritence is transitive. I.e. if job J_3 blocks J_2 which in turn blocks J_1 then J_3 may inherit the priority from J_1.

http://fileadmin.cs.lth.se/cs/Education/EDA040/lecture/rtpL5.pdf

3.2.3 Immediate inheritence

The priority of the thread running in a semaphore is immediately raised to the ceiling priority.

3.3 Direct blocking

Occurs when a higher-priority job tries to acquire resources held by a job with lower priority.

3.4 Push-through blocking

Occurs when a medium priority job is blocked by a lower priority job that has inherited a higher priority from a job it directly blocks. This is necessary to avoid unbounded priority inversion.

3.5 Rate monotonic scheduling

Scheduling by rate of occurrence. I.e. a job that has a high occurrence rate is highly prioritized. A set of tasks is said to be schedulable by the rate monotonic algorithm if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

The tasks are also schedulable if $R_i \leq C_i$ for all jobs. We define R_i as:

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \lceil \frac{R_i}{T_j} \rceil C_j$$

3.6 Chained blocking

When a highly prioritized job needs to collect resources held by two or more lower prioritized jobs, meaning it has to wait for a series of lower prioritized jobs to finish before managing to complete its own tasks.