Sammanfattning EDA040

Felix Mulder

19 oktober 2013

1 Terminology

- *Scheduler:* a scheduler schedules different threads for execution on the CPU. States of threads *running, ready and blocked.* More in the *Scheduling* section.
- *Signalling:* is used when you want to make sure that a thread is running exclusively. I.e. flag passing between threads.
- *Mutual exclusion:* a thread has a critical area and uses a semaphore to lock down a resource for exclusive access. Other threads may still run, just not use the locked resource. I.e. flag passing between threads and resources (not between threads and other threads).
- *Improvement to mutex:* using give could check to make sure that the same entity taking the semaphore is the same entity yielding it.
- *Drift:* when an operation is designed to run at a fixed duration, but due to context switches and least sleep times will accumulate a diff in time. (Accumulative drift)

2 Deadlock Analysis

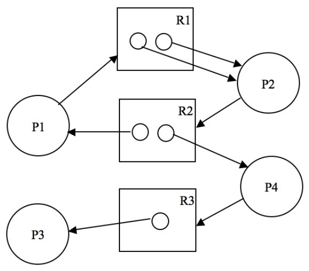
Resource allocation graphs are used to determine if a program can deadlock. For a program to end up in a deadlock there are a few requirements.

- 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.



3 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.

3.1 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];
}
```

3.2 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.

3.3 Mutual Exclustion

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

3.4 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.

3.5 Starvation

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

3.6 Livelock

No thread makes progress, but they keep executing.

3.7 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.)

3.8 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.

3.9 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.

4 Scheduling

4.1 Scheduler

The scheduler schedules different threads for execution time on the CPU. As explained above, there are three different states for the threads/processes:

- *Running:* the thread is executing its instructions on the CPU. The scheduler could decide that the thread has had enough execution time and move the thread *from* running *to* ready.
- *Ready:* The thread has told the scheduler that it is ready for execution on the CPU. When the scheduler decides that the thread should get to execute, it initiates a context switch and transfers it into the running state.
- *Blocked:* the thread has reached a point in its execution where it decides to yield the processor. This could be due to not being able to lock a resource or having to sleep. Important to note is that it is the thread itself that decides when to block. When the thread can continue it is usually due to another thread changing the state of something. It can thus be argued that it is the other thread that takes the first thread from blocked to ready.

4.2 Context switch

A context switch occurs for instance when the scheduler decides that the current thread has run long enough, and allows another thread to execute. The threads each have a call stack. This call stack together with the current registers are saved away and the other thread's call stack and registers are restored.

4.2.1 Faster context switches

Without preemption we can speed up the process of a context switch, since we will then know exactly what needs to get saved away. With preemption we cannot be entirely sure. Ergo we will need to save everything.

4.3 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.

4.4 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.

4.4.1 Basic

Raises the priority of the low priority thread temporarily

4.4.2 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.

Ceiling blocking occurs when the highest prioritized task that can access a resource is blocked by a lower prioritized job using the resource.

4.4.3 Immediate inheritence

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

4.5 Direct blocking

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

4.6 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.

4.7 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

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

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

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

4.8 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.

5 Sceduling Theory

5.1 Sufficient and necessary

The analysis is *sufficient* if, when it answers “YES”, all deadlines are met.

The analysis is *necesary* if, when it answers “NO”, there reallys is a situation where deadlines could be missed.

The analysis is *exact* if it is both necessary and sufficient.

5.2 Fixed priority scheduling

With fixed priority scheduling, where to execute each task is decided dynamically based on which of the currently ready tasks has the highest priority.

5.3 Execution time estimation

5.3.1 Measuring execution times

The main problem with this approach is that it can be overly optimistic. There are no guarantees that the longest execution time has occurred. Testing all combinations of input data is often impossible in practice. Caching and pipelining further exacerbate the estimations, the remedy to this is typically to disable caching and pipelining.

5.3.2 Analyzing execution times

The main problem with the approach is that it can be overly pessimistic, however, this approach is the only one that is formally correct.

- *Compiler dependence:* different compilers generate different code typically analysis tools work with machine code.
- *Data dependant branching:* analysis tool must explore all paths and return the longest.
- *Iterations and recursion:* the programmer must specify the number of times a loop may execute. Also hard to know how deep recursion will go. Therefore it is often disallowed.
- *Dynamic memory allocation:* may invoke garbage collection, which is extremely difficult for analysis tools to handle.

The gap between general purpose hardware and hardware which safely can be used for hard real-time applications i.e. that can be analyzed increases. This makes it important to develop theory and methods that make it possible to use general purpose platforms in hard real-time applications with an acceptable level of performance and quality of service.

5.4 Scheduling methods

5.4.1 Static cyclic scheduling

Static cyclic scheduling is an off-line approach. Contains a table of the order in which to execute the different tasks. The table repeats cyclically.

The scheduler simply starts the first task in the calendar (table), awaits its completion, waits until it's time to start the next task, waits for its completion, and so on. The reason it cannoot start executing the second one immediately is that the schedule has been calculated based on worst running times. Making this rather ineffective but safe. In a slightly more complicated case of preemptive scheduling, the schedule also contains how long each task is allowed to run for.

Limitations:

- *Only periodic tasks:* aperiodic tasks are converted into periodic ones using polling.
- *Long calendars:* The shortest repeating cycle is equal to the least common multiple between the task periods. For example: 5,10,20 ms gives the cycle 20 ms. But in this example: 7,13,23 ms the cycles length is 2093 ms (least common denominator: $7 * 13 * 23 = 2093$)
- *NP-completeness:* generating a schedule is a NP-hard problem.

The advantages:

- *Simple analysis:* only requirement is to run through the calendar and check weather all tasks meet their deadlines.
- *Data sharing:* the scheduling algorithm can make sure that there won't be a context switch between tasks sharing data.
- *Precedence constraints:* Possible to handle precedence constraints. I.e. X should finish before another task Y is allowed to start.

5.5 Earliest Deadline First (EDF) Scheduling

Tasks are scheduled according to which task has the earliest deadline. The approach is dynamic. It is often more intuitive to assign deadlines to tasks than it is to assign priorities. Assigning deadlines only requires local knowledge whereas priorities require a complete understanding, i.e. global knowledge.

- Only periodic tasks
- each task i has a period of T_i
- required computation time C_i (worst case)
- deadline $D_i = T_i$
- no interprocess communication
- an "ideal" real-time kernel

An *ideal kernel* means that context switches and interrupts take zero time. If the *utilization* U is less than 100% then all deadlines will be met:

$$U = \sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq 1$$

The main advantage of EDF is that the processor can be fully utilized and still all deadlines will be met.

5.6 Rate monotonic scheduling

Priorities are set monotonically according to task rate (period). A task with shorter period is assigned a higher priority.

- Only periodic tasks
- $D_i = T_i$
- C_i are known
- No interprocess communication
- Tasks may not suspend themselves
- priorities are unique
- the kernel is ideal

With these assumptions, the following holds:
For a system with n tasks, all tasks will meet their deadlines if:

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

The result is only sufficient. I.e. if the utilization is larger than the bound, the task may still be schedulable. As $n \to \infty$, the utilization bound $U \to 0.693$. Which has led to the simple rule of thumb saying that if the utilization is less than 0.693 all deadlines are met.

5.6.1 Generalized RMS

Restrictive assumptions have been relaxed, in the following way: $D_i \leq T_i$. We can also assume that the worst possible response time will be $\leq T_i$.

Also for a system with utilization larger than 0.693 all deadlines will still be met if:

$$\forall i, R_i \leq D_i$$

where:

$$R_i^{n+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j, R_i^0 = 0$$

Stated earlier RMS didn't take interprocess communication into account. The above, however, includes B_i i.e. blocking time. Thus RMS can be expanded to cover interprocess communication (and release jitter, but fuck that).

For RMS to work all C_i have to be known. Something which is extremely difficult. Worst-case execution times are much larger than those that occur in practice. Thus many cases may be considered to be unschedulable but will work perfectly fine in reality.

Another form of measurement for schedulability using RMS is the *hyperbolic bound* which recently proved that a task is schedulable if:

$$\prod_{i=1}^{n} U_i + 1 \leq 2, \text{ where } U_i = \frac{C_i}{T_i}$$

5.6.2 Calculating B_i

In RMS a thread can only be blocked by a *lower* priority thread. Remember that it can thus be blocked by push-through blocking.

1. The monitor to which the thread is connected, which of the lower priority threads can block it? Calculate max of these.
2. Can it be blocked on a different monitor? The value from 1 added to this.
3. Can it be blocked by push-through blocking? Take this into account.

5.6.3 Calculating C_i

If R_i or T_i and B_i is known we may calculate the smallest possible C_i using the following chain of reasoning:

Example: $R_i = 50$,$B_i = 10$ and i has one higher priority thread, A, with $T_A = 100$ and $C_A = 10$. This gives us:

$$R_i = C_i + B_i + \sum_{\forall k \in hp(k)} \lceil \frac{R_k}{T_j} \rceil C_j = 50$$
$$\Longleftrightarrow$$
$$90 = C_i + 10 + \lceil \frac{90}{100} \rceil 10$$
$$C_i = 70$$

6 Garbage collection

6.1 Manual memory management

Using manual memory management we have to make sure that we get no dangling pointers and no memory leaks. Even if we do this, we can't be sure that the heap is properly defragmented. (GC without compaction also leads to this.)

6.2 Reference Counting

We keep a count to how many references a certain object in memory has. When this counter reaches zero we remove the object.

Advantages: Easy to implement, short pauses. *Disadvantages:*

- DEqueues will have circular references, meaning they won't be marked for removal by the GC.
- No compaction → fragmentation
- Expensive, the counters have to be adjusted for every pointer assignment.

6.3 Traversing algorithms

The idea is to periodically traverse the objects in the heap from a root pointer, usually located in or near main. Objects that are encountered are marked, and the ones not marked are deleted.

6.3.1 Cheney's algorithm

The heap is divided into two equal halves, only one of which is in use at any one time. Garbage collection is performed by copying live objects from one heap to the next, which then becomes current heap. The entire old heap can then be discarded in one piece.

A pointer from the old object location to the new one is kept during copy so that the program can still run during garbage collection.

6.3.2 Generation-based GC

It stands to reason that if objects die young, the ones that don't usually live for a long time.

- Partition heap into several generations
- New objects are allocated into the young generation
- Surviving objects are promoted into the next generation

Pros: Efficient! Most pauses are short. Most garbage collection is done in the youngest generation. *Cons:* Complex. Must keep track of inter-generation pointers.

6.3.3 Idea

By our own Roger Henriksson!

- Avoid doing GC work when high-priority threads execute.
- Perform GC in the pauses. Memory always available.
- Low-priority threads: standard incremental techniques.
- Minimize the cost for pointer operations for the high- priority threads.
- Interruptible garbage collection, minimum locking.
- Theory for a priori schedulability analysis.

6.4 Implementation details

For the garbage collector not getting in the way of things, with a similar scenario, the following should be done:

- Assuming we have one HP thread and one LP thread.
- Garbage collection should be scheduled after the HP thread has run. I.e. the GC should have a medium priority.
- Cleaning after the LP thread should be done in its context. I.e. the GC should have the same priority as the LP for cleaning up after said thread.