Concurrency Control

Slide 1

COMP3231 Operating Systems

2005/S2

CONCURRENCY CONTROL

Concurrency appears in many contexts:

- Multi-threading: concurrent threads share an address space
- Multi-programming: concurrent processes execute on a uniprocessor
- Multi-processing: concurrent processes on a multiprocessor
- Distributed processing: concurrent processes executing on multiple nodes connected by a network

Concurrency is also used in different forms:

- Multiple applications (multiprogramming)
- Structured application (application is a set of concurrent threads or processes)
- Operating-system structure (OS is a set of threads or processes)

Slide 2

WHAT IS CONCURRENCY CONTROL?

Concurrency can be dangerous to the unwary programmer:

- Sharing global resources (order of read and write operations)
- Management of allocation of resources (danger of deadlock)
- Programming errors difficult to locate (Heisenbugs)

Slide 3

Concurrent processes (threads) need special support:

- Communication among processes
- Allocation of processor time
- Sharing of resources
- Synchronisation of multiple processes

Slide 4

CONCURRENT ACCESS TO A GLOBAL QUEUE

Inserting:

1. create new object
2. set last->next to &new
3. set last to &new

Slide 4
CONCURRENT ACCESS TO A GLOBAL QUEUE

Thread A:
- create new object
- set last->next to &new
- set last to &new

Thread B:
- create new object
- set last->next to &new
- set last to new

We can get the same problem with truly parallel threads:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>create new object</td>
<td>create new object</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>last-&gt;next = &amp;new</td>
<td>last-&gt;next = &amp;new</td>
</tr>
<tr>
<td>last = &amp;new</td>
<td>last = &amp;new</td>
</tr>
</tbody>
</table>

Lessons learned:
- We have to control access to shared resource (such as shared variables)
- We can do this effectively by controlling access to the code utilising those shared resources ⇒ critical sections

CONCURRENT ACCESS TO A GLOBAL QUEUE

Only one thread at a time should have write access to the queue:
- Thread A creates new object, sets last->next pointer
- Thread A is suspended
- Thread B is scheduled, calls insert, but since Thread A is currently in insert, has to wait
- Thread A is resumed, the data structure is in the same state as it was when it was suspended.
- Thread A completes operation
- Thread B is allowed to execute insert

CONCURRENCY CONTROL

- Processes can
  - compete for resources
    - Processes may not be aware of each other
    - execution must not be affected by each other
    - OS is responsible for controlling access
  - cooperate by sharing a common resource
    - Programmer responsible for controlling access
    - Hardware, OS, programming language may provide support
- Threads of a process usually do not compete, but cooperate.
- Since process access to shared resources is through OS, problems are the same (although solved on different levels)
  - e.g., kernel threads of different competing processes cooperate
We face three control problems:

1. **Mutual exclusion**: critical resources → critical sections
   - Only one process at a time is allowed in a critical section
   - Example: only one process at a time is allowed to send commands to the printer
2. **Deadlock**: e.g., two processes and two resources
3. **Starvation**: e.g., three processes compete for a resource

Let's look at these problems in turn.

**Mutual exclusion illustrated:**
```c
void proc (int i)
{
    for (;;) {
        entercritical (i); /* blocks if other thread
                           already in critical section */
        <critical section>
        exitcritical (i); /* allow other threads to
                           enter */
        <remainder>
    }
}
void main () {
    parbegin (proc (R_1), proc (R_2), ..., proc (R_n));
}
```

But how can we implement `entercritical()` and `exitcritical()`?

---

**Requirements for Mutual Exclusion**

**Implementation:**
- Only one thread at a time is allowed in the critical section for a resource
- No deadlock or starvation
- A thread must not be delayed access to a critical section when there is no other thread using it
- A thread that halts in its non-critical section must do so without interfering with other thread
- No assumptions are made about relative thread speeds or number of processes

**Usage:**
- A thread remains inside its critical section for a finite time only
- No potentially blocking operations should be executed inside a critical section
- No deadlock or starvation

**Conceptually, there are three ways to satisfy the implementation requirements:**

1. **Software approach**: put responsibility on the processes themselves
2. **Systems approach**: provide support within operation system or programming language
3. **Hardware approach**: special-purpose machine instructions
Software Approaches to Mutual Exclusion

Premises:
- One or more processes with shared memory
- Elementary mutual exclusion at level of memory accesses: simultaneous accesses to the same memory location are serialised

In the following, Dijkstra’s presentation of Dekker’s algorithm (actually, we use Peterson’s algorithm, which is a more elegant variant of Dekker’s)

A First Attempt

The Plan:
- threads take turns in executing critical section
- exploit serialisation of memory access to implement serialisation of access to critical section
- mutual exclusion

We employ a variable (memory location) turn that indicates whose turn it is to enter the critical section:

\[ P_0: \]
\[ \ldots \]
\[ \text{while } (\text{turn} != 0) \]
\[ \text{while } (\text{turn} != 1) \]
\[ \text{do nothing} ; \]
\[ \text{do nothing} ; \]
\[ \text{<critical section>} \]
\[ \text{turn} = 1; \]
\[ \text{turn} = 0; \]
\[ \ldots \]

\[ P_1: \]
\[ \ldots \]

Busy waiting (spin lock):

- Process is always checking to see if it can enter the critical section
- Implements mutual exclusion
- Simple
- Process burns resources while waiting

Other drawbacks of this code:
- Processes must alternate access to the critical section
- If one process fails anywhere in the program, the other is permanently blocked

A Second Attempt

The Problem: turn stores who can enter the critical section, rather then whether anybody may enter the critical section

The New Plan: we store for each process whether it is in the critical section right now (in a Boolean flag):

\[ \text{flag[i]}: \text{Process i is in the critical section} \]

\[ P_0: \]
\[ \ldots \]
\[ \text{while } (\text{flag[1]} \text{) do nothing} ; \]
\[ \text{do nothing } ; \]
\[ \text{<critical section>} \]
\[ \text{flag[0]} = \text{true}; \]
\[ \text{flag[0]} = \text{false}; \]
\[ \ldots \]

\[ P_1: \]
\[ \ldots \]
\[ \text{while } (\text{flag[0]} \text{) do nothing} ; \]
\[ \text{do nothing } ; \]
\[ \text{<critical section>} \]
\[ \text{flag[1]} = \text{true}; \]
\[ \text{flag[1]} = \text{false}; \]
\[ \ldots \]
Is this a good solution?

- If one thread fails
  - ✔ outside of the critical section, the other is not blocked
  - ✗ inside a critical section, other thread is blocked (however, hard to avoid)

- But: it does not even guarantee exclusive access!!!
  ① both flags are set to false
  ② T₀ enters critical section
  ③ T₁ enters critical section
  ④ T₁ sets flag[0]
  ⑤ T₀ sets flag[1]
  ➜ worse if more than two threads involved

Third Attempt

- The Goal: we have to get rid of the gap between toggling the two flags
- Yet Another Plan: move setting the flag before checking whether we can enter

Slide 18

P₀:

\[
\begin{align*}
\text{flag}[0] &= \text{true}; \\
\text{while} \ (\text{flag}[1]) \\
\text{/* do nothing */}; \\
<\text{critical section}> \\
\text{flag}[0] &= \text{false}; \\
\text{...}
\end{align*}
\]

P₁:

\[
\begin{align*}
\text{flag}[1] &= \text{true}; \\
\text{while} \ (\text{flag}[0]) \\
\text{/* do nothing */}; \\
<\text{critical section}> \\
\text{flag}[1] &= \text{false}; \\
\text{...}
\end{align*}
\]

Slide 19

Fourth Attempt

- Previous problem: process sets its own state before knowing the other process’ state and cannot back off
- Our plan: Process retracts its decision if it cannot enter

Slide 19

P₀:

\[
\begin{align*}
\text{flag}[0] &= \text{true}; \\
\text{while} \ (\text{flag}[1]) \\
\text{/* do nothing */}; \\
\text{delay ();} \\
\text{flag}[0] &= \text{true}; \\
<\text{critical section}> \\
\text{flag}[0] &= \text{false}; \\
\text{...}
\end{align*}
\]

P₁:

\[
\begin{align*}
\text{flag}[1] &= \text{true}; \\
\text{while} \ (\text{flag}[0]) \\
\text{/* do nothing */}; \\
\text{delay ();} \\
\text{flag}[1] &= \text{true}; \\
<\text{critical section}> \\
\text{flag}[1] &= \text{false}; \\
\text{...}
\end{align*}
\]

Did we finally make it?

- Close, but we may have a livelock

Tweaking the code:

- We can solve this problem by combining the fourth with the first attempt
- In addition to the flag’s, we use a variable indicating whose turn it is to have precedence in entering the critical section
Instead of Dekker’s original algorithm, let’s consider Peterson’s:

\[ P_0: \]
\[ \ldots \]
\[ \text{flag}[0] = \text{true}; \]
\[ \text{flag}[1] = \text{false}; \]
\[ \text{turn} = 1; \]
\[ \text{while (flag}[1] \]
\[ \text{&& turn} = 1\]
\[ \text{)}\]
\[ \text{/* do nothing */}; \]
\[ <\text{critical section}> \]
\[ \text{flag}[0] = \text{false}; \]
\[ \text{...} \]

\[ P_1: \]
\[ \ldots \]
\[ \text{flag}[1] = \text{true}; \]
\[ \text{turn} = 0; \]
\[ \text{while (flag}[0] \]
\[ \text{&& turn} = 0\]
\[ \text{)}\]
\[ \text{/* do nothing */}; \]
\[ <\text{critical section}> \]
\[ \text{flag}[1] = \text{false}; \]
\[ \text{...} \]

\[ \rightarrow \text{Both processes are courteous and solve a tie in favour of the other} \]
\[ \rightarrow \text{Algorithm can easily be generalised to work with } n \text{ processes} \]