CS5460: Operating Systems

Lecture 7: Synchronization

(Chapter 6)

Multi-level Feedback Queues

- **Multiple queues with different priorities**
  - **Alternative: single priority queue**
- **Round robin schedule processes with equal priorities**
  - **Low priority jobs can "starve" for a while**
- **Adjust priorities based on observed behavior:**
  - **Jobs start with default priority (perhaps modified via "`nice`")**
  - **If time quantum expires, bump job priority down**
  - **If job blocks before end of time quantum, bump priority up (Why?)**
- **Effect:**
  - **The scheduler "figures out" which jobs are interactive and which are CPU-bound**

Synchronization

What is Synchronization?

- **<u>Question</u>: How do you control the behavior of "cooperating" processes that share resources?**

Time	You	Your roomate
3:00	Arrive home	
3:05	Check fridge → no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Buy milk	Check fridge → no milk
3:25	Arrive home, milk in fridge	Leave for grocery
3:30		
3:35		Buy milk
3:40		Arrive home, milk in fridge!

Shared Memory Synchronization

- Threads share memory
- Preemptive thread scheduling is a major problem
  - Context switch can occur at any time, even in the middle of a line of code (e.g., "X = X + 1;")
    - Unit of atomicity → Machine instruction
    - Cannot assume anything about how fast processes make progress
  - Individual processes have little control over order in which processes run
- Need to be paranoid about what scheduler might do
- Preemptive scheduling introduces non-determinism

Race Condition

- Two (or more) processes run in parallel and output depends on order in which they are executed
- ATM Example
  - SALLY: balance += $50; BOB: balance -= $50;
  - Question: If initial balance is $500, what will final balance be?

SALLY	BOB
`r0 ← balance`	
`add r0, r0, $50`	
`balance ← r0`	
	`r0 ← balance`
	`sub r0, r0, $50`
	`balance ← r0`

This (or reverse) is what you'd normally expect to happen.

Net: $500

Race Conditions

- Two (or more) processes run in parallel and output depends on order in which they are executed
- ATM Example
  - SALLY: balance += $50; BOB: balance -= $50;
  - Question: If initial balance is $500, what will final balance be?

However, this (or reverse) can happen due to a race condition.

SALLY	BOB
`r0 ← balance`	
	`r0 ← balance`
`add r0, r0, $50`	
	`sub r0, r0, $50`
`balance ← r0`	
	`balance ← r0`

Net: $450

Synchronization

- **The race condition happened because there were conflicting accesses to a resource**

- **Basic idea behind most synchronization:**
  – **If two threads, processes, interrupt handlers, etc. are going to have conflicting accesses, force one of them wait until it is safe to proceed**

- **Conceptually simple, but difficult in practice**
  – **The problem is that we need to protect all possible locations where two (or more) threads or processes might conflict**

Synchronization Problems

- **Synchronization can be required for different resources**
  - **<u>Memory</u>: e.g., multithreaded application**
  - **<u>OS object</u>: e.g., two processes that read/write same system file**
  - **<u>Hardware device</u>: e.g. two processes that both want to burn a DVD**
- **There are different kinds of synchronization problems**
  - **Sometimes we just want activities to not interfere with each other**
  - **Sometimes we care about ordering**

Synchronization Problems

- **Synchronization may be across machines**
  - **What if some machines are disconnected or rebooting?**
- **Sometimes it’s not OK to block a thread or process**
  - **May have to reserve the “right” do something ahead of time**

Atomic Operations

- **Series of operations that cannot be interrupted**
  – **Some operations are atomic with respect to everything that happens on a machine**
  – **Other atomic operations are atomic only with respect to conflicting processes, threads, interrupt handlers, etc.**
- **On typical architectures:**
  – **Individual word load/stores and ALU instructions**
  – **Synchronization operations (e.g., fetch_and_add, cmp_and_swap)**
- **ATM example → Balance updates were NOT atomic**
  – **Solution: Enforce atomic balance updates**
  – **Question: How?**

More Atomic

- **Atomic operations are at the root of most synchronization solutions**
- **Processor has to support some atomic operations**
  – **If not, we're stuck!**
- **OS uses low-level primitives to build up more sophisticated atomic operations**
  – **For example, locks that support blocking instead of busy-waiting**
  – **We'll look at an example soon**

More Definitions

- **Synchronization (or Concurrency Control):**
  - Using atomic operations to eliminate race conditions
- **Critical section:**
  - Piece of code (e.g., ATM balance update) that must run atomically
  - Mutual exclusion: Ensure at most one process at a time
- **Lock:**
  - Synchronization mechanism that enforces atomicity
  - Semantics:
    - » Lock(L): If L is not currently locked → atomically lock it
      If L is currently locked → block until it becomes free
    - » Unlock(L): Release control of L
  - You can use a lock to protect data: Lock(L) before accessing data, Unlock(L) when done

Fixing the ATM problem

- **Problem:**
  - Balance update not atomic
- **Solution:**
  - Introduce atomic operations
- **Effect:**
  - \+ Eliminates race condition
  - – Increases overhead
  - – Restricts concurrency
- **Open issues:**
  - Where do we use locks?
    - » Avoid deadlocks or livelocks
    - » Ensure fairness
  - How do we implement locks?
  - What other synch ops are there besides locks?

SALLY

```
Lock(L);
r0 ← balance
add r0, r0, $50
balance ← r0
Unlock(L);
```

BOB

```
Lock(L);
r0 ← balance
sub r0, r0, $50
balance ← r0
Unlock(L);
```

Critical Section (the three lines between Lock(L); and Unlock(L);)

Lock Requirements

1. **Must guarantee that only one process / thread is in the critical section at a time**
   – This is obvious
2. **Must guarantee progress**
   – Processes or threads don't have to wait for an available lock
3. **Must guarantee bounded waiting**
   – No process or thread needs to wait forever to enter the critical section
   – Figuring out the bound can be interesting

Implementing Critical Sections

- **Goal:**
  - If milk needed, somebody buys
  - Only one person buys milk
- **Idea: Wait while note is up**
  - "Busy wait" loop
- **Does this work?**
  - Is milk bought?
  - Can both buy?

FAILS: Can both buy milk (How?)

P0:

```
while (Note) { }
Note ← 1;   // leave Note
Milk ← Milk + 1;   // CritSect
Note ← 0;   // remove Note
```

P1:

```
while (Note) { }
Note ← 1;   // leave Note
Milk ← Milk + 1;   // CritSect
Note ← 0;   // remove Note
```

Milk V.1

Implementing Critical Sections

- **Goal:**
  - If milk needed, somebody buys
  - Only one person buys milk
- **Idea: Add per-process flag**
  - Set flag while in critical section
  - Explicit check on other process
- **Does this work?**
  - Is milk bought?
  - Can both buy?

FAILS: Can <u>both</u> buy milk (How?)

```
flag[2] = {0,0};
```

<u>P0</u>:

```
while (flag[1]) { }
flag[0] ← 1;
Milk ← Milk + 1; // Crit sect
flag[0] ← 0;
```

<u>P1</u>:

```
while (flag[0]) { }
flag[1] = 1;
Milk ← Milk + 1; // Crit sect
flag[1] = 0;
```

Milk V.2

Implementing Critical Sections

- **Goal:**
  - If milk needed, somebody buys
  - Only one person buys milk
- **Reverse order in which you set and test flag**
  - Set flag before testing this time
- **Does this work?**
  - Is milk bought?
  - Can both buy?

FAILS: Violates progress and bounded wait

```
flag[2] = {0,0};
```

P0:

```
flag[0] ← 1;
while (flag[1]) { }
Milk ← Milk + 1; // Crit sect
flag[0] ← 0;;
```

P1:

```
flag[1] ← 1;
while (flag[0]) { }
Milk ← Milk + 1; // Crit sect
flag[1] ← 0;
```

Milk V.3

Implementing Critical Sections

- **Goal:**
  - If milk needed, somebody buys
  - Only one person buys milk
- **Idea: Alternating turns**
  - Let one in at a time
  - Wait your turn
- **Does this work?**
  - Is milk bought?
  - Can both buy?

FAILS: Violates progress and bounded waiting

```
turn ← 0;
```

P0:

```
while (turn == 1) { }
Milk ← Milk + 1; // Crit sect
turn ← 1;
```

P1:

```
while (turn == 0) { }
Milk ← Milk + 1; // Crit sect
turn ← 0;
```

Milk V.4

Implementing Critical Sections

- **Goal:**
  - If milk needed, somebody buys
  - Only one person buys milk
- **Idea: Combine approaches**
  - Use flag[ ] to denote interest
  - Use turn to break ties
- **Does this work?**
  - Is milk bought?
  - Can both buy?

SUCCEEDS: Meets all three criteria for locks

```
flag[2] ← {0,0};  turn ← 0;
```

P0:

```
flag[0] ← 1;  turn ← 1;
while (flag[1] && turn == 1) { }
Milk ← Milk + 1; // Crit sect
flag[0] ← 0;
```

P1:

```
flag[1] ← 1;  turn ← 0;
while (flag[0] && turn == 0) { }
Milk ← Milk + 1; // Crit sect
flag[1] ← 0;
```

Milk V.5

Peterson's Algorithm

- Algorithm on previous slide was published by Peterson in 1981
  - Should work on any uniprocessor
    - Relies only on atomicity of memory operations
  - Can be extended to more than 2 threads

- Peterson's algorithm does not work on any modern multicore machine
  - It depends on certain guarantees provided by the memory subsystem, such as not reordering stores
  - Fixing the algorithm is not totally trivial
  - These fixes are not portable to other architectures

Lock Correctness

- **How do I show that a lock implementation is wrong?**
- **How do I argue that a lock implementation is right?**

Summary

- **Critical sections are those that must execute atomically**
  - Locks are a way to get atomicity
  - Locks are implemented using lower-level atomic operations
- **Locks should guarantee mutual exclusion, progress, and bounded waiting**
- **Implementing locks is tricky**
  - Many published solutions have been wrong for years before somebody noticed the problem
  - Even harder on a modern machine
- **In real life, 99.9% of the time you don't implement synchronization operations yourself**

Questions?