Synchronization Basics and Semaphores

CS 4410
Operating Systems
Spring 2017
Cornell University

Lorenzo Alvisi
Anne Bracy

See: Ch 5&6 in OSPP textbook

The slides are the product of many rounds of teaching CS 4410 by Professors Sirer, Bracy, Agarwal, George, and Van Renesse.
Threads and their Data

Threads have:
- Private stack
- Private registers
- Shared globals
- What about data?

2 possibilities:

1. Independent Threads
   - t3 working set
   - t2 working set
   - t1 working set
   (private by agreement)

2. Cooperative Threads
2 Threads, 1 Shared Variable

Code like this:

```
... amount -= 10,000;
...
```

Might execute like this:

```
... r1 = load amount
r1 = r1 - 10,000
store r1 to amount
...
```

```
... r2 = load amount
... r2 = 0.5 * r2
store r2 to amount
... 
```
Race Conditions

When the behavior of a program depends on the interleaving of operations of different threads.  
(Once thread starts, it needs to “race” to finish.)

Number of possible interleavings is huge

• Some interleavings are good
• Some interleavings are bad:
  • Bad interleavings may be rare!
  • *Works 100 times ≠ correct*
  • *Case Study: Therac-25*

*ALL possible interleavings should be safe!*
Problems with Sequential Reasoning

1. Program execution depends on the possible interleavings of threads’ access to shared state.

2. Program execution can be nondeterministic.

3. Compilers and processor hardware can reorder instructions.
CONCURRENT APPLICATIONS

SYNCHRONIZATION VARIABLES
- Locks
- Semaphores
- Condition Variables

ATOMIC INSTRUCTIONS
- Interrupt Disable
- Atomic R/W Instructions

HARDWARE
- Multiple Processors
- Hardware Interrupts
Race Condition Revisited

T1

disable_interrupts();
r1 = load amount
r1 = r1 - 10,000
store r1 to amount
enable_interrupts();

T2

disable_interrupts();
r2 = load amount
r2 = 0.5 * r2
store r2 to amount
enable_interrupts();

That was easy….
class dismissed?
Test and Set

MIPS version:

\[
\text{tas } r1, 0(r2):
\]
\[
r1 \leftarrow 0(r2) \quad \# \text{ test}
\]
\[
0(r2) \leftarrow 1 \quad \# \text{ set}
\]

- atomic hardware primitive
- typically a multi-cycle bus operation that atomically reads and updates a memory location
- supports mutual exclusion
Test and Set to provide Mutual Exclusion

ATOMIC int TestAndSet(int *var) {
    int oldVal = *var;
    *var = 1;
    return oldVal;
}

acquire(int *lock) {
    while(test_and_set_set(lock))
        /* do nothing */;
}

release(int *lock) {
    *lock = 0;
}
Race Condition Revisited

Is this a good solution? ??

acquire();
while(test_and_set(lock))
  while(test_and_set(lock))
    yield()
while(test_and_set(lock))
  r1 = load amount
  r1 = r1 - 10,000
  store r1 to amount
release();

T1

T2
acquire();
r2 = load amount
r2 = 0.5 * r2
store r2 to amount
release();

Now with Locks!
Thou shalt not busy-wait!
CONCURRENT APPLICATIONS

SYNCHRONIZATION VARIABLES

Locks  Semaphores  Condition Variables

ATOMIC INSTRUCTIONS

Interrupt Disable  Atomic R/W Instructions

HARDWARE

Multiple Processors  Hardware Interrupts
Semaphores

Dijkstra introduced in the THE Operating System

• Stateful:
  • a semaphore has a non negative VALUE associated with it
  • value is incremented and decremented atomically

• Interface
  • Two operations: P() and V()
  • No operation to read the value!

[Dijkstra 1962]
Semaphore Operations: $P$ and $V$

$P(S)$:
• wait until value is positive
• when so, atomically decrement VALUE by 1

$V(S)$:
• increment VALUE by 1
• resume a thread waiting on $P$ (if any)

```
P(S) {
    while(S <= 0) ;
    S -= 1;
}
```

```
V(S) {
    S += 1;
}
```

Dutch 4410: $P = $ Probeer (‘Try’) and $V = $ Verhoog (‘Increment’, ’Increase by one’)
Binary Semaphore

Semaphore value is either 0 or 1

- Used for **mutual exclusion** (semaphore as a more efficient lock)
- Initially 1 in that case

Semaphore $S$
$S.init(1)$

T1

P($S$)
CriticalSection()
V($S$)

T2

P($S$)
CriticalSection()
V($S$)
Counting Semaphores

Sema count can be any integer
- Used for signaling or counting resources
- Typically:
  - one thread performs P() to await event
  - another thread performs V() to alert waiting thread that event has occurred

Semaphore packetarrived
packetarrived.init(0)

PacketProcessor():
x = get_packet_from_card()
enqueue(packetq, x);
V(packetarrived);

NetworkingThread():
P(packetarrived);
x = dequeue(packetq);
print_contents(x);
Possible Semaphore implementation

P1 Context: no preemption (threads run until they yield)

\[
\text{P(S)} \{
\begin{align*}
&\text{while}(S \leq 0) \\
&S -= 1;
\end{align*}
\}
\]

\[
\text{V(S)} \{
\begin{align*}
&S += 1;
\end{align*}
\}
\]

\[
\text{P(Sema \,*s)} \{
\begin{align*}
&\text{if count big enough} \\
&\text{decrement count} \\
&\text{else} \\
&\text{make note of thread} \\
&\text{stop thread}
\end{align*}
\}
\]

\[
\text{V(Sema \,*s)} \{
\begin{align*}
&\text{if no one waiting on s} \\
&\text{increment count} \\
&\text{else} \\
&\text{wake an interested thread}
\end{align*}
\}
\]
Producer-Consumer Problem

2+ threads communicate:
some threads **produce** data that others **consume**

Bounded buffer: size $N$
Producer process writes data to buffer
  • Writes to **in** and moves rightwards
  • Don’t write more than $N$!
Consumer process reads data from buffer
  • Reads from **out** and moves rightwards
  • Don’t consume if there is no data!
Example: “pipe” ( | ) in Unix  > cat file | sort | uniq | more
Solution #1: No Protection

```c
// add item to buffer
void produce(int item) {
    buf[in] = item;
    in = (in+1)%N;
}

// remove item
int consume() {
    int item = buf[out];
    out = (out+1)%N;
    return item;
}
```

Problems:
1. Unprotected shared state (multiple producers/consumers)
2. Inventory:
   - Consumer could consume when nothing is there!
   - Producer could overwrite not-yet-consumed data!
Solution #2: Add Mutex Semaphores

Shared:
int buf[N];
int in, out;
Semaphore mutex_prod(1), mutex_cons(1);

// add item to buffer
void produce(int item)
{
    P(mutex_prod);
    buf[in] = item;
    in = (in+1)%N;
    V(mutex_prod);
}

// remove item
int consume()
{
    P(mutex_cons);
    int item = buf[out];
    out = (out+1)%N;
    V(mutex_cons);
    return item;
}
Solution #3: Add Communication Semaphores

Shared:
int buf[N];
int in, out;
Semaphore mutex_prod(1), mutex_cons(1);
Semaphore enoughRoom(N), dataThere(0);

void produce(int item)
{
    P(enoughRoom); // space?
P(mutex_prod);
    buf[in] = item;
in = (in+1)%N;
V(mutex_prod);
V(dataThere); // item!
}

int consume()
{
P(dataThere); // need item
P(mutex_cons);
int item = buf[out];
out = (out+1)%N;
V(mutex_cons);
V(enoughRoom); // space!
return item;
}
DONE FOR TODAY!


**Classic Semaphore Mistakes**

<table>
<thead>
<tr>
<th>P(S)</th>
<th>CS</th>
<th>I stuck on 2nd P(). Subsequent processes freeze up on 1st P().</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>I stuck on 2nd P(). Subsequent processes freeze up on 1st P().</td>
</tr>
<tr>
<td>P(S)</td>
<td>← typo</td>
<td>Undermines mutex:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• J doesn’t get permission via P()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• “extra” V()s allow other processes into the CS inappropriately</td>
</tr>
<tr>
<td>V(S)</td>
<td>← typo</td>
<td>Next call to P() will freeze up. Confusing because the other process could be correct but hangs when you use a debugger to look at its state!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Next call to P() will freeze up. Confusing because the other process could be correct but hangs when you use a debugger to look at its state!</td>
</tr>
<tr>
<td></td>
<td>← omission</td>
<td>Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.</td>
</tr>
</tbody>
</table>
“During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores.”

Semaphores NOT to the rescue!

Semaphores are “low-level” primitives. Small errors:
• Easily bring system to grinding halt
• Very difficult to debug

Two usage models:
• **Mutual exclusion**: “real” abstraction is a critical section
• **Communication**: threads use semaphores to communicate (e.g., bounded buffer example)

**Simplification**: Provide concurrency support in compiler
➡️ Enter Monitors