Synchronization

CS 4410, Operating Systems

Fall 2016
Cornell University

Rachit Agarwal
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.
Synchronization Motivation & Basics

• Race Conditions
• Critical Sections
• Example: Too Much Milk
• Basic Hardware Primitives
• Building a SpinLock
Threads Share Memory

Threads have:

• Private registers
  Context switching saves and restores registers when switching from thread to thread

• Shared “global” memory
  Global means not stack memory

• Usually private stack
  Pointers into stacks across threads frowned upon
Two Threads, One Variable

2 threads updating a single shared variable amount

- One thread wants to decrement amount by $10K
- Other thread wants to decrement amount by 50%

What happens when two threads execute concurrently?
Two Threads, One Variable

It won’t actually execute like this:

T1

. . .
\[ r1 = \text{load from amount} \]
\[ r1 = r1 - 10,000 \]
store \( r1 \) to amount
. . .

T2

. . .
\[ r2 = \text{load from amount} \]
\[ r2 = 0.5 * r2 \]
store \( r2 \) to amount
. . .

Memory

amount 100,000
**Two Threads, One Variable**

It might execute like this:

- **T1**
  - \( r1 = \text{load from amount}\)
  - \( r1 = r1 - 10,000 \)
  - store \( r1 \) to \( \text{amount} \)
  - . . .

- **T2**
  - . . .
  - \( r2 = \text{load from amount} \)
  - \( r2 = 0.5 \times r2 \)
  - store \( r2 \) to \( \text{amount} \)
  - . . .

Memory

| amount | 100,000 |

Everything works:

although different, either order is correct
Two Threads, One Variable

Or it might execute like this:

T1

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

T2

...  
r2 = load from amount  
...  
r2 = 0.5 * r2  
store r2 to amount  
...  

Lost Update:  
Wrong ..and very difficult to debug
Race Conditions

= timing dependent error involving shared state

• Once thread A starts, it needs to “race” to finish
• Whether race condition happens depends on thread schedule
  • Different “schedules” or “interleavings” (total order on machine instructions)

All possible interleavings should be safe!
Race Conditions are Hard to Debug

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad:
  - But bad interleavings may rarely happen!
  - *Works 100x ≠ no race condition*
- Timing dependent = small changes can hide bug

Case Study: Therac-25
Example: Races with Queues

2 concurrent enqueue() operations?
2 concurrent dequeue() operations?

What could possibly go wrong?
Critical Section

Code that can be executed by only one thread at a time

Goals

Safety: 1 thread in a critical section at time

Liveness: all threads eventually make it into CS if desired

Fairness: all have equal chances of getting into CS

... in practice, fairness rarely guaranteed
Too Much Milk Problem

2 roommates fridge always stocked with milk
  • fridge is empty → need to restock it
  • *don’t want to buy too much milk*

Caveats
  • Only communicate by a notepad on the fridge
  • Notepad has cells with names, like variables:

```
outtobuymilk [0]
```

**TASK:** Write the pseudo-code to ensure that at most one roommate goes to buy milk
Solution #1: No Protection

T1

if fridge_empty():
    buy_milk()

T2

if fridge_empty():
    buy_milk()

Safety: Only one person (at most) buys milk.
Liveness: If milk is needed, someone eventually buys it.
Fairness: Roommates equally likely to go to buy milk.

Safe? Live? Fair?
Solution #2: add a boolean flag

outtobuymilk initially false

Safety: Only one person (at most) buys milk
Liveness: If milk is needed, someone eventually buys it.
Fairness: Roommates equally likely to go to buy milk.
Solution #3: add two boolean flags!

one for each roommate (initially false)

\textcolor{blue}{\text{blueonit}}, \textcolor{green}{\text{greenonit}}

\text{Safety:}\quad \text{Only one person (at most) buys milk}

\text{Liveness:}\quad \text{If milk is needed, someone eventually buys it.}

\text{Fairness:}\quad \text{Roommates equally likely to go to buy milk.}

\text{Safe?  Live?  Fair?}
Solution #4: asymmetric flags!

one for each roommate (initially false)

\textbf{blueonit, greenonit}

\begin{align*}
\text{T1} & \quad \text{blueonit} = 1 \\
& \quad \text{while greenonit:} \\
& \quad \quad \text{do\_nothing()} \\
& \quad \text{if fridge\_empty():} \\
& \quad \quad \text{buy\_milk()} \\
& \quad \text{blueonit} = 0
\end{align*}

\begin{align*}
\text{T2} & \quad \text{greenonit} = 1 \\
& \quad \text{if not blueonit and} \\
& \quad \quad \text{fridge\_empty():} \\
& \quad \quad \text{buy\_milk()} \\
& \quad \text{greenonit} = 0
\end{align*}

\textbf{Safe? Live? Fair?}

– complicated (and this is a simple example!)
– hard to ascertain that it is correct
– asymmetric code is hard to generalize & unfair
**Last Solution: Peterson’s Solution**

another flag \textit{turn} \{blue, green\}

\begin{itemize}
  \item \textcolor{red}{T1} \textcolor{blue}{blueonit} = 1
  \item \textcolor{red}{T1} \textcolor{green}{turn} = \textcolor{green}{green}
  \item \textcolor{red}{T1} while (\textcolor{blue}{greenonit} and \textcolor{blue}{turn}==\textcolor{green}{green}): 
    \textcolor{red}{T1} do\_nothing() 
  \item \textcolor{red}{T1} if fridge\_empty():
    \textcolor{red}{T1} buy\_milk()
  \item \textcolor{red}{T1} \textcolor{blue}{blueonit} = 0
  \item \textcolor{red}{T2} \textcolor{blue}{greenonit} = 1
  \item \textcolor{red}{T2} \textcolor{blue}{turn} = \textcolor{blue}{blue}
  \item \textcolor{red}{T2} while (\textcolor{blue}{blueonit} and \textcolor{blue}{turn}==\textcolor{blue}{blue}): 
    \textcolor{red}{T2} do\_nothing() 
  \item \textcolor{red}{T2} if fridge\_empty():
    \textcolor{red}{T2} buy\_milk()
  \item \textcolor{red}{T2} \textcolor{blue}{greenonit} = 0
\end{itemize}

\textbf{Safe? Live? Fair?}

\begin{itemize}
  \item complicated (and this is a simple example!)
  \item hard to ascertain that it is correct
  \item hard to generalize, inefficient
Hardware Solution

- Hardware primitives to provide mutual exclusion
- Typically relies on a multi-cycle bus operation that atomically reads and updates a memory location
- Example Spec of Test-And-Set:

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

sets the value to 1, returns former value
Spinlocks

```c
SL_acquire(int *lock) {
    while(test_and_set(lock))
        /* do nothing */;
}

SL_release(int *lock) {
    *lock = 0;
}
```
Buying Milk with Spinlock

Shared spinlock: int buyingmilk, initially 0

T1

SL_acquire(&buyingmilk)
if fridge_empty():
  buy_milk()
SL_release(&buyingmilk)

T2

SL_acquire(&buyingmilk)
if fridge_empty():
  buy_milk()
SL_release(&buyingmilk)
SpinLock Issues

Participants not in critical section must spin → wasting CPU cycles
  • Replace the “do nothing” loop with a “yield()”? Processes would still be scheduled and descheduled

Need better primitive:
  • allows one process to pass through
  • all others to sleep until they can be executed again
Semaphores

- Definition
- Binary Semaphores
- Counting Semaphores
- Implementing Semaphores
- Classic Synchronization Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problems
- Classic Semaphore Mistakes
- Semaphores Considered Harmful
What is a Semaphore?

Non-negative integer w/atomic increment, decrement

\[ S = \text{new Semaphore(init)} \quad // \quad \text{must initialize!} \]

Can only be modified by:
- **P(S):** decrement or block if already 0
- **V(S):** increment & wake up any waiting threads
- No interface to read the value

Operations have the following semantics:

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

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

Dutch 4410: \( P = \text{Probeer} \) (‘Try’) and \( V = \text{Verhoog} \) (‘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);
```
Starter Semaphore implementation

P(Sema *s) {
    while(TRUE) {
        if s.count big enough
            break;
    }
    decrement s.count
}

V(Sema *s) {
    increment s.count
}

Assume a thread “stuck” in P() will be eventually interrupted

Do we like this?

1. Shared state is not protected
   Threads can be interrupted at any point? → need a lock

2. Busy-waiting is bad!
   Count not big enough? → make note of thread’s interest and stop it. (have V start it again when count big enough)
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;
}
```

Shared:

- int buf[N];
- int in, out;

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 nRoom(N), NData(0);

void produce(int item)
{
    P(nRoom); // need space
    P(mutex_prod);
    buf[in] = item;
    in = (in+1)%N;
    V(mutex_prod);
    V(nData); // new item!
}

int consume()
{
    P(nData); // need item
    P(mutex_cons);
    int item = buf[out];
    out = (out+1)%N;
    V(mutex_cons);
    V(nRoom); // more space!
    return item;
}
Readers-Writers Problem

Models access to a database: shared data that some threads read and other threads write

Want to allow:
• multiple concurrent readers —OR— (exclusive)
• only a single writer at a time

Example: making an airline reservation
• When you browse to look at flight schedules the web site acts as a reader on your behalf
• When you reserve a seat, web site has to write into database to make the reservation

[Courtois+ 1971]
Readers-Writers Constraints

N threads share 1 object in memory
• Some write: 1 writer active at a time
• Some read: n readers active simultaneously

Insight: generalizes the critical section concept

Questions:
1. Writer active & combo of readers/writers arrive. Who should get in next?
2. Writer waiting & endless of # of readers come. Fair for them to become active?

For now: back-and-forth turn-taking:
• If a reader is waiting, readers get in next
• If a writer is waiting, one writer gets in next
**Readers-Writers Solution**

**Shared:**
```c
int rcount;
Semaphore count_mutex(1);
Semaphore rw_lock(1);
```

**void write()** {
```
P(rw_lock);
... /*perform write */
... V(rw_lock);
```
}

```c
int read()
{
    P(count_mutex);
    rcount++;
    if (rcount == 1)
        P(rw_lock);
    V(count_mutex);
    ... /* perform read */
    ... P(count_mutex);
    rcount--;
    if (rcount == 0)
        V(rw_lock);
    V(count_mutex);
}
```
Readers-Writers: Understanding the Solution

If there is a writer:
- First reader blocks on rw_lock
- Other readers block on mutex

Once a reader is active, all readers get to go through
- Which reader gets in first?

The last reader to exit signals a writer
- If no writer, then readers can continue

If readers and writers waiting on rw_lock & writer exits
- Who gets to go in first?
Readers-Writers: Assessing the Solution

When readers active no writer can enter ✔
  • Writers wait @ P(rw_lock)
When writer is active nobody can enter ✔
  • Any other reader or writer will wait (where?)
Back-and-forth isn’t so fair:
  • Any number of readers can enter in a row
  • Readers can “starve” writers

Fair back-and-forth semaphore solution is tricky!
  • Try it! (don’t spend too much time...)
**Classic Semaphore Mistakes**

- **I**
  - `P(S)`
  - CS
  - `P(S) ← typo`

  I stuck on 2nd `P()`. Subsequent processes freeze up on 1st `P()`.

- **J**
  - `V(S) ← typo`
  - CS
  - `V(S)`

  Undermines mutex:
  - J doesn’t get permission via `P()`
  - “extra” `V()`s allow other processes into the CS inappropriately

- **K**
  - `P(S)`
  - CS
  - `← omission`

  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!

- **L**
  - `if(x) return;`
  - CS
  - `V(S)`

  Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.
Semaphores Considered Harmful

“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