Synchronization
(Chapters 4 & 5)

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]
• Foundations
• Semaphores
• Monitors & Condition Variables
Synchronization Foundations

- Race Conditions
- Critical Sections
- Example: Too Much Milk
- Basic Hardware Primitives
- Building a SpinLock
Recall: Process vs. Thread

Process:
- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources
- One or more Threads:
  - Stack
  - Registers
  - PC, SP

Shared amongst threads
Two Threads, One Variable

2 threads updating a shared variable `amount`
- One thread wants to decrement `amount` by $10K
- Other thread wants to decrement `amount` by 50%

What happens when both threads are running?

Memory


t1

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

t2

... amount *= 0.5;
...

amount 100,000
Two Theads, One Variable

Might execute like this:

T1

\[
\begin{align*}
\text{r1} &= \text{load from amount} \\
\text{r1} &= \text{r1} - 10,000 \\
\text{store r1 to amount} \\
\text{...}
\end{align*}
\]

T2

\[
\begin{align*}
\text{r2} &= \text{load from amount} \\
\text{r2} &= 0.5 * \text{r2} \\
\text{store r2 to amount} \\
\text{...}
\end{align*}
\]

Or vice versa (T1 then T2 → 45,000)… either way is fine…
Two Theads, 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” exist (total order on machine instructions)

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.
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 hide bugs

(recall: Therac-25)
Example: Races with Shared Variable

Thread A:

\[
\text{while}(i < 10) \\
i = i + 1; \\
\text{print "A won!"}
\]

Thread B:

\[
\text{while}(i > -10) \\
i = i - 1; \\
\text{print "B won!"}
\]

\(i\) is shared and initialized to 0.

Who wins?
Are there any guarantees about this code?
What if both run on different same-speed cores?
Example: Races with Queues

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

What could possibly go wrong?
Critical Section

Must be atomic due to shared memory access

Goals

**Safety:** 1 thread in a critical section at time

**Liveness:** all threads make it into the CS if desired

**Fairness:** equal chances of getting into CS

... in practice, fairness rarely guaranteed
Too Much Milk: Safety, Liveness, and Fairness with no hardware support
Too Much Milk Problem

2 roommates, fridge always stocked with milk
  • fridge is empty → need to restock it
  • \textit{don’t want to buy too much milk}

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

\[
\text{out\_to\_buy\_milk} = 0
\]

\textbf{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.

Safe? Live? Fair?
Solution #3: add two boolean flags!

one for each roommate (initially false):

blues_got_this, reds_got_this

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 #4: asymmetric flags!

one for each roommate (initially false):

\[ \text{blue\_got\_this}, \text{red\_got\_this} \]

T1
\[
\text{blues\_got\_this} = 1 \\
\text{while } \text{reds\_got\_this}: \\
\quad \text{do\_nothing}() \\
\text{if fridge\_empty}(): \\
\quad \text{buy\_milk}() \\
\text{blues\_got\_this} = 0
\]

T2
\[
\text{reds\_got\_this} = 1 \\
\text{if not blues\_got\_this and fridge\_empty}(): \\
\quad \text{buy\_milk}() \\
\text{reds\_got\_this} = 0
\]

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 turn \{blue, red\}

T1
blues_got_this = 1
turn = red
while (reds_got_this and turn==red):
    do_nothing()
if fridge_empty():
    buy_milk()
blues_got_this = 0

T2
reds_got_this = 1
turn = blue
while (blues_got_this and turn==blue):
    do_nothing()
if fridge_empty():
    buy_milk()
reds_got_this = 0

Safe? Live? Fair?
– complicated (and this is a simple example!)
– hard to ascertain that it is correct
– hard to generalize
Hardware Solution

• HW primitives to provide mutual exclusion
• A **machine instruction** (part of the ISA!) that:
  • Reads & updates a memory location
  • Is atomic because it is a single instruction!
• Example: Test-And-Set
  1 instruction with the following semantics:

```c
ATOMIC int TestAndSet(int *var) {
    int oldVal = *var;
    *var = 1;
    return oldVal;
}
```
sets the value to 1, returns former value
Buying Milk with TAS

Shared variable: `int buyingmilk`, initially 0

T1

```c
while(TAS(&buyingmilk))
    do_nothing();
if fridge_empty():
    buy_milk()
buyingmilk := 0
```

T2

```c
while(TAS(&buyingmilk))
    do_nothing();
if fridge_empty():
    buy_milk()
buyingmilk := 0
```

A little hard on the eyes. Can we do better?
Enter: Locks!

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

release(int *lock) {
    *lock = 0;
}
```
Buying Milk with Locks

Shared lock: `int buyingmilk`, initially 0

T1
```
acquire(&buyingmilk);
if fridge_empty():
    buy_milk()
release(&buyingmilk);
```

T2
```
acquire(&buyingmilk);
if fridge_empty():
    buy_milk()
release(&buyingmilk);
```

Now we’re getting somewhere!
Is anyone not happy with this?
Thou shalt not busy-wait!
Not just any locks: **Spin**Locks

Participants not in critical section must **spin** → **wasting CPU cycles**

- Replace the “do nothing” loop with a “yield()”?  
- Threads would still be scheduled and descheduled (context switches are expensive)

Need a better primitive:
- allows one thread to pass through  
- all others sleep until they can execute again
• Foundations
• **Semaphores**
• Monitors & Condition Variables
Semaphores

- **Definition**
- Binary Semaphores
- Counting Semaphores
- Classic Sync. Problems (w/Semaphores)
  - Producer-Consumer (w/ a bounded buffer)
  - Readers/Writers Problem
- Classic Mistakes with Semaphores
What is a Semaphore? [Dijkstra 1962]

Dijkstra introduced in the THE Operating System

**Stateful:**
- a **value** (incremented/decremented atomically)
- a queue
- a lock

**Interface:**
- **Init**(starting value)
- **P** *(procure)*: decrement, “consume” or “start using”
- **V** *(vacate)*: increment, “produce” or “stop using”

*No operation to read the value!*

Dutch 4410: P = Probeer (‘Try’), V = Verhoog (‘Increment’, ‘Increase by one’)
Semantics of P and V (Part 1)

P():
• wait until value > 0
• when so, decrement VALUE by 1

V():
• increment VALUE by 1

These are the semantics,
but how can we make this efficient?
(doesn’t this look like a spinlock?!?)
Semantics of P and V (Complete)

P():
• block (**sit on Q**) til value $>0$
• when so, decrement VALUE by 1

V():
• increment VALUE by 1
• **resume a thread waiting on Q** (if any)

Okay this looks efficient, but how is this safe?
*(that’s what the lock is for – both P&V need to TAS the lock)*
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
S.P()
CriticalSection()
S.V()

T2
S.P()
CriticalSection()
S.V()
Example: A simple mutex

Semaphore S
S.init(1)

S.P()
CriticalSection()
S.V()

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

V() {
    n += 1;
}
Counting Semaphores

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

```
Semaphore packetarrived
packetarrived.init(0)
```

ReceivingThread:
```
pkt = get_packet()
enqueue(packetq, pkt);
packetarrived.V();
```

PrintingThread:
```
packetarrived.P();
pkt = dequeue(packetq);
pkt = print(pkt);
```
Semaphore’s count:

- must be initialized!
- keeps state
  - reflects the sequence of past operations
  - \( >0 \) reflects number of future P operations that will succeed

Not possible to:

- read the count
- grab multiple semaphores at same time
- decrement/increment by more than 1!
Producer-Consumer Problem

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

Bounded buffer: size $N$ entries

Producer process writes data to buffer
  - Writes to `in` and moves rightwards

Consumer process reads data from buffer
  - Reads from `out` and moves rightwards
Producer-Consumer Applications

• Pre-processor produces source file for compiler’s parser
• Data from bar-code reader consumed by device driver
• File data: computer → printer spooler → line printer device driver
• Web server produces data consumed by client’s web browser
• “pipe” (|) in Unix: `cat file | sort | more`
Starter Code: No Protection

// 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!
Part 1: Guard Shared Resources

Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);

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

// remove item
int consume()
{
    mutex_out.P();
    int item = buf[out];
    out = (out+1)%N;
    mutex_out.V();
    return item;
}
Part 2: Manage the Inventory

Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);
Semaphore space(N), item(0);

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

int consume()
{
    item.P(); // need item
    mutex_out.P();
    int item = buf[out];
    out = (out+1)%N;
    mutex_out.V();
    space.V(); // more space!
    return item;
}
Sanity checks

Shared:
int buf[N];
int in, out;
Semaphore mutex_in(1), mutex_out(1);
Semaphore space(N), item(0);

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

int consume()
{
    item.P(); // need item
    mutex_out.P();
    int item = buf[out];
    out = (out+1)%N;
    mutex_out.V();
    space.V(); // more space!
    return item;
}
Producer-consumer: How did we do?

Pros:
• Live & Safe & Correct
• No Busy Waiting! (*is this true?*)
• Scales nicely

Cons:
• Still seems complicated: is it correct?
• Not so readable
• Easy to introduce bugs
Semaphores

• Definition
• Binary Semaphores
• Counting Semaphores
• Classic Sync. Problems (w/Semaphores)
  – Producer-Consumer (w/ a bounded buffer)
  – Readers/Writers Problem
• **Classic Mistakes with Semaphores**
Classic Semaphore Mistakes

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

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

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!

Conditional code can change code flow in the CS. Caused by code updates (bug fixes, etc.) by someone other than original author of code.
“During system conception … 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!

These 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