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
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?
Two Theads, One Variable

Might execute like this:

T1

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

T2

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

Or vice versa (T1 then T2 \(\rightarrow 45,000\))… 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
. . .
```

Memory

```
amount  50,000
```

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:

while(i < 10)
    i = i + 1;
print “A won!”

Thread B:

while(i > -10)
    i = i - 1;
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
- don’t want to buy too much milk

Caveats
- Only communicate by a notepad on the fridge
- Notepad has cells with names, like variables:
  
  out_to_buy_milk \[
  \theta
  \]

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

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

\texttt{outtobuymilk} initially false

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

\begin{align*}
\text{T2} & : & \text{while}(& \text{outtobuymilk}): \\
& & \text{do\_nothing}(); \\
& & \text{if} & \text{fridge\_empty}(): \\
& & \text{outtobuymilk} = 1 \\
& & \text{buy\_milk}() \\
& & \text{outtobuymilk} = 0 \\
\end{align*}

\begin{itemize}
\item \textbf{Safety:} Only one person (at most) buys milk
\item \textbf{Liveness:} If milk is needed, someone eventually buys it.
\item \textbf{Fairness:} Roommates equally likely to go to buy milk.
\end{itemize}

\textbf{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):

\[ \textit{blues\_got\_this}, \quad \textit{reds\_got\_this} \]

\begin{columns}
\begin{column}{0.5\textwidth}
\begin{Verbatim}
\begin{align*}
\text{blue\_got\_this} &= 1 \\
\text{while} \quad \text{reds\_got\_this}: \\
&\quad \text{do\_nothing()} \\
\text{if} \quad \text{fridge\_empty}(): \\
&\quad \text{buy\_milk()} \\
\text{blue\_got\_this} &= 0
\end{align*}
\end{Verbatim}
\end{column}
\begin{column}{0.5\textwidth}
\begin{Verbatim}
\begin{align*}
\text{red\_got\_this} &= 1 \\
\text{if} \quad \text{not} \quad \text{blue\_got\_this} \\
&\quad \text{and} \quad \text{fridge\_empty}(): \\
&\quad \text{buy\_milk()} \\
\text{red\_got\_this} &= 0
\end{align*}
\end{Verbatim}
\end{column}
\end{columns}

\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 turn \{blue, red\}

\begin{align*}
T1: \\
\text{blues\_got\_this} &= 1 \\
\text{turn} &= \text{red} \\
\text{while} (\text{reds\_got\_this} \\
\hspace{1cm} \text{and} \hspace{1cm} \text{turn} == \text{red}): \\
\hspace{2cm} &\text{do\_nothing()} \\
\text{if} \hspace{0.5cm} \text{fridge\_empty}(): \\
\hspace{2cm} &\text{buy\_milk()} \\
\text{blues\_got\_this} &= 0
\end{align*}

\begin{align*}
T2: \\
\text{reds\_got\_this} &= 1 \\
\text{turn} &= \text{blue} \\
\text{while} (\text{blues\_got\_this} \\
\hspace{1cm} \text{and} \hspace{1cm} \text{turn} == \text{blue}): \\
\hspace{2cm} &\text{do\_nothing()} \\
\text{if} \hspace{0.5cm} \text{fridge\_empty}(): \\
\hspace{2cm} &\text{buy\_milk()} \\
\text{reds\_got\_this} &= 0
\end{align*}

\textbf{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

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

```
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 */;
}
```

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

Shared lock: `int buyingmilk`, initially 0

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

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 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!_

*[Dijkstra 1962]*

Dutch 4410: **P** = Probeer (‘Try’), **V** = Verhoog (’Increment’, ’Increase by one’)

29
Semantics of P and V (Part 1)

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

V():
• increment VALUE by 1

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

V() {
    n += 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()
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

```latex
\begin{align*}
\text{ReceivingThread:} & \quad \text{PrintingThread:} \\
\text{Semaphore packetarrived} & \quad \text{Semaphore packetarrived} \\
\text{packetarrived.init(0)} & \\
\text{pkt = get_packet();} & \quad \text{packetarrived.P();} \\
\text{enqueue(packetq, pkt);} & \quad \text{pkt = dequeue(packetq);} \\
\text{packetarrived.V();} & \quad \text{print(pkt);} \\
\end{align*}
```
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

```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!
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;
}

1. Is there a V for every P?
2. Mutex initialized to 1?
3. Mutex P&V in same thread?
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
Readers-Writers Problem [Courtois+ 1971]

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

At any time, want to allow:
• multiple concurrent readers — OR — (exclusive)
• only a single writer

Example: making an airline reservation
• Browse flights: web site acts as a reader
• Reserve a seat: web site has to write into database (to make the reservation)
Readers-Writers Specifications

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

Implementation Questions:
1. Writer is active. Combo of readers/writers arrive. Who should get in next?
2. Writer is 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
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

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(S) )</td>
<td>I</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>( P(S) )</td>
<td></td>
</tr>
<tr>
<td>( V(S) )</td>
<td>J</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>( V(S) )</td>
<td></td>
</tr>
<tr>
<td>( P(S) )</td>
<td>K</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>( P(S) )</td>
<td>L</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>\if(x)\ return;</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>( V(S) )</td>
<td></td>
</tr>
</tbody>
</table>

- **I** stuck on 2nd \( P() \). Subsequent processes freeze up on 1st \( P() \).
- **J** didn’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.

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

“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
• Foundations
• Semaphores
• Monitors & Condition Variables
CONCURRENT APPLICATIONS

SYNCHRONIZATION OBJECTS

Locks  Semaphores  Condition Variables  Monitors

ATOMIC INSTRUCTIONS

Interrupt Disable  Atomic R/W Instructions

HARDWARE

Multiple Processors  Hardware Interrupts
Monitors & Condition Variables

• **Definition**
• Simple Monitor Example
• Implementation
• Classic Sync. Problems with Monitors
  – Bounded Buffer Producer-Consumer
  – Readers/Writers Problems
  – Barrier Synchronization
• Semantics & Semaphore Comparisons
• Classic Mistakes with Monitors
Monitor Semantics guarantee mutual exclusion
Only one thread can execute monitor procedure at any time (aka “in the monitor”)

in the abstract:

```plaintext
Monitor monitor_name
{
    // shared variable declarations

    procedure P1() {
    }

    procedure P2() {
    }
    .
    .
    procedure PN() {
    }

    initialization_code() {
    }
}

for example:

Monitor bounded_buffer
{
    int in=0, out=0, nElem=0;
    int buffer[N];

    consume() {
    }

    produce() {
    }
}
```

- can only access shared data via a monitor procedure
- only one operation can execute at a time
One Thread at a Time in the Monitor!

consume() {
}

produce() {
}

out  in
Producer-Consumer Revisited

Problems:

1. Unprotected shared state (multiple producers/consumers)
   
   Solved via Monitor.

   Only 1 thread allowed in at a time.
   - Only one thread can execute monitor procedure at any time
   - If second thread invokes monitor procedure at that time, it will block and wait for entry to the monitor.
   - If thread within a monitor blocks, another can enter

2. Inventory:
   - Consumer could consume when nothing is there!
   - Producer could overwrite not-yet-consumed data!

   What about these?
   → Enter Condition Variables
Condition Variables

A mechanism to wait for events

3 operations on Condition Variable Condition `x`

- `x.wait()`: sleep until woken up (could wake up on your own)
- `x.signal()`: wake at least one process waiting on condition (if there is one). No history associated with signal.
- `x.broadcast()`: wake all processes waiting on condition (useful for resource manager)

!! NOT the same thing as UNIX wait & signal !!
Using Condition Variables

You must hold the monitor lock to call these operations.

To wait for some condition:
```python
while not some_predicate():
    CV.wait()
```
- atomically releases monitor lock & yields processor
- as `CV.wait()` returns, lock automatically reacquired

When the condition becomes satisfied:

```python
CV.broadcast():  # wakes up all threads
CV.signal():    # wakes up at least one thread
```
Condition Variables Live in the Monitor

Abstract Data Type for handling shared resources, comprising:

1. Shared Private Data
   • the resource
   • can only be accessed from in the monitor

2. Procedures operating on data
   • gateway to the resource
   • can only act on data local to the monitor

3. Synchronization primitives
   • among threads that access the procedures

[Hoare 1974]
Types of Wait Queues

Monitors have two kinds of “wait” queues

- **Entry to the monitor**: a queue of threads waiting to obtain mutual exclusion & enter
- **Condition variables**: each condition variable has a queue of threads waiting on the associated condition
Kid and Cook Threads

```c
kid_main() {
    play_w_legos()
    BK.kid_eat()
    bathe()
    make_robots()
    BK.kid_eat()
    facetime_Karthik()
    facetime_oma()
    BK.kid_eat()
}

cook_main() {
    wake()
    shower()
    drive_to_work()
    while(not_5pm)
        BK.makeburger()
    drive_to_home()
    watch_got()
    sleep()
}
```

Monitor BurgerKing {
Lock mlock

    int numbburgers = 0
    condition hungrykid

    kid_eat:
        with mlock:
            while (numburgers==0)
                hungrykid.wait()
                numbburgers -= 1

    makeburger:
        with mlock:
            ++numburger
            hungrykid.signal()

}
Monitors & Condition Variables

- Definition
- Simple Monitor Example
- **Implementation**
- Classic Sync. Problems with Monitors
  - Bounded Buffer Producer-Consumer
  - Readers/Writers Problems
  - Barrier Synchronization
- Semantics & Semaphore Comparisons
- Classic Mistakes with Monitors
Language Support

Can be embedded in programming language:

• Compiler adds synchronization code, enforced at runtime
• **Mesa/Cedar** from Xerox PARC
• **Java**: synchronized, wait, notify, notifyall
• **C#**: lock, wait (with timeouts), pulse, pulseall
• **Python**: acquire, release, wait, notify, notifyAll

Monitors easier & safer than semaphores

• Compiler can check
• Lock acquire and release are implicit and cannot be forgotten
class BK:
    def __init__(self):
        self.lock = Lock()
        self.hungrykid = Condition(self.lock)
        self.nBurgers = 0

    def kid_eat(self):
        with self.lock:
            while self.nBurgers == 0:
                self.hungrykid.wait()
            self.nBurgers = self.nBurgers - 1

    def make_burger(self):
        with self.lock:
            self.nBurgers = self.nBurgers + 1
            self.hungrykid.notify()
class BK:
    def __init__(self):
        self.lock = Lock()
        self.hungrykid = Condition(self.lock)
        self.nBurgers = 0

from rvr import MP, MPthread

class BurgerKingMonitor(MP):
    def __init__(self):
        MP.__init__(self, None)
        self.lock = Lock("monitor lock")
        self.hungrykid = self.Lock.Condition("hungry kid")
        self.nBurgers = self.Shared("num burgers", 0)
Monitors in “4410 Python”: kid_eat

```python
def kid_eat(self):
    with self.lock:
        while self.nBurgers == 0:
            self.hungrykid.wait()
        self.nBurgers = self.nBurgers - 1
```

```python
def kid_eat(self):
    with self.lock:
        while (self.nBurgers.read() == 0):
            self.hungryKid.wait()
        self.nBurgers.dec()
```

We do this for helpful feedback:

- from auto-grader
- from debugger

Look in the A2/doc directory for details and example code.
Monitors & Condition Variables

• Definition
• Simple Monitor Example
• Implementation
• **Classic Sync. Problems with Monitors**
  – Bounded Buffer Producer-Consumer
  – Readers/Writers Problems
  – Barrier Synchronization
• Semantics & Semaphore Comparisons
• Classic Mistakes with Monitors