Synchronization: Critical Sections & Semaphores

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
  - Hardware-supported solutions
- The basic synchronization mechanism: Semaphores
- **Classical synchronization problems**

- Reading: R&R, Ch 14 (and, later, Ch 13)
### Synchronization: Critical Sections & Semaphores

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
  - Hardware-supported solutions
  - The basic synchronization mechanism: Semaphores
- **Classical synchronization problems**

---

#### The Critical Section Problem: Example 1

```c
void echo() {
  char in; /* shared variables */
  input(in, keyboard);
  out := in;
  output(out, display);
}
```

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation:</strong></td>
<td><strong>Operation:</strong></td>
</tr>
<tr>
<td>Echo()</td>
<td>Echo()</td>
</tr>
<tr>
<td>Interleaved execution</td>
<td>Interleaved execution</td>
</tr>
<tr>
<td>/ input(in, keyboard)</td>
<td>/ input(in, keyboard)</td>
</tr>
<tr>
<td>out = in;</td>
<td>out = in;</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>/</td>
<td>/ input(in, keyboard)</td>
</tr>
<tr>
<td>/</td>
<td>out = in;</td>
</tr>
<tr>
<td>/</td>
<td>output(out, display);</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

*Race condition!*
The Critical Section Problem: Example 2

Producer-consumer with bounded, shared-memory, buffer.

Producer:

```c
void deposit(Item * next) {
    while (counter == n) no_op;
    buffer[in] = next;
    in = (in+1) MOD n;
    counter = counter + 1;
}
```

Consumer:

```c
Item * remove() {
    while (counter == 0) no_op;
    next = buffer[out];
    out = (out+1) MOD n;
    counter = counter - 1;
    return next;
}
```

This Implementation is not Correct!

<table>
<thead>
<tr>
<th>Operation</th>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter = counter + 1</td>
<td>reg₁ = counter</td>
<td>reg₂ = counter</td>
</tr>
<tr>
<td>on CPU:</td>
<td>reg₁ = reg₁ + 1</td>
<td>reg₂ = reg₂ - 1</td>
</tr>
<tr>
<td>counter = reg₁</td>
<td>reg₂ = counter + 1</td>
<td>counter = reg₂</td>
</tr>
<tr>
<td>interleaved</td>
<td>reg₁ = counter</td>
<td>reg₂ = reg₂ - 1</td>
</tr>
<tr>
<td>execution:</td>
<td>reg₁ = reg₁ + 1</td>
<td>counter = reg₂</td>
</tr>
<tr>
<td></td>
<td>counter = reg₁</td>
<td></td>
</tr>
</tbody>
</table>

- Race condition!
- Need to ensure that only one process can manipulate variable counter at a time: synchronization.
Critical Section Problem: Example 3

Insertion of an element into a list.

```c
void insert(new, curr) {
    /*1*/ new.next = curr.next;
    /*2*/ new.prev = c.next.prev;
    /*3*/ curr.next = new;
    /*4*/ new.next.prev = new;
}
```

Interleaved Execution causes Errors!

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new1.next = curr.next;</code></td>
<td>...</td>
</tr>
<tr>
<td><code>new1.prev = c.next.prev;</code></td>
<td><code>new2.next = curr.next;</code></td>
</tr>
<tr>
<td>...</td>
<td><code>new2.prev = c.next.prev;</code></td>
</tr>
<tr>
<td><code>curr.next = new1;</code></td>
<td><code>curr.next = new2;</code></td>
</tr>
<tr>
<td><code>new1ext.prev = new1;</code></td>
<td><code>new2.next.prev = new2;</code></td>
</tr>
</tbody>
</table>

- Must guarantee mutually exclusive access to list data structure!
Synchronization: Critical Sections & Semaphores

• Why? Examples
• What? The Critical Section Problem
• How? Software solutions
  • Hardware-supported solutions
• The basic synchronization mechanism:
  Semaphores
• Classical synchronization problems

Critical Sections

• Execution of critical section by processes must be mutually exclusive.
• Typically due to manipulation of shared variables.
• Need protocol to enforce mutual exclusion.

```c
while (TRUE) {
    enter section;
    critical section;
    exit section;
    remainder section;
}
```
Criteria for a Solution of the C.S. Problem

1. **Mutual exclusion**: Only one process at a time can enter the critical section.

2. **Progress**: A process that halts in non-critical section cannot prevent other processes from entering the critical section. When no process is in a critical section, any process that requests to enter the critical section should be permitted to enter without delay.

3. **Bounded waiting**: A process requesting to enter a critical section should not be delayed indefinitely.

Assumptions:
- Make no assumptions about the relative speed of processors (or their number).
- A process remains within a critical section for a finite time only.

Synchronization: Critical Sections & Semaphores

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
  - Hardware-supported solutions
  - The basic synchronization mechanism: Semaphores
  - Classical synchronization problems
A (Wrong) Solution to the C.S. Problem

- Two processes \( P_0 \) and \( P_1 \)
- \texttt{int turn; /* turn == i: \( P_i \) is allowed to enter c.s. : j=1-i;*/}

\begin{verbatim}
\texttt{P_i: while (TRUE) { }
  while (turn != i) no_op;
  critical section;
  turn = j;
  remainder section;
}
\end{verbatim}

What’s Wrong?

- Guarantees mutual exclusion.
- Does not guarantee progress
  - enforces strict alternation of processes entering CS’s.
- Bounded waiting violated
  - suppose one process terminates while its its turn?
Another Wrong Solution
( Remove strict alternation requirement )

```c
bool flag[2];  /* initialize to FALSE */
/* flag[i] == TRUE : P_i intends to enter c.s. */

P_i: while (TRUE) {
    while (flag[j]) no_op;
    flag[i] = TRUE;
    critical section;
    flag[i] = FALSE;
    remainder section;
}
```

What’s Wrong?

- Mutual exclusion violated
- Progress ok.
- Bounded wait ok.
Yet Another Wrong Solution
(Restore mutual exclusion)

```c
bool flag[2]; /* initialize to FALSE */
/* flag[i] == TRUE : P_i intends to enter c.s.*/

while (TRUE) {
    flag[i] = TRUE;
    while (flag[j]) no_op;

    critical section;

    flag[i] = FALSE;

    remainder section;
}
```

What’s Wrong?

- Guarantees mutual exclusion
- Violates progress
  - both processes could set flag and then deadlock on the while.
- Bounded waiting violated
A Combined Solution (Petersen)

```c
int turn;
bool flag[2]; /* initialize to FALSE */

while (TRUE) {
    flag[i] = TRUE;
    turn = j;
    while (flag[j]) && (turn == j) no_op;

    critical section;

    flag[i] = FALSE;

    remainder section;
}
```

Synchronization: Critical Sections & Semaphores

- Why? Examples
- What? The Critical Section Problem
- How? Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism:
  - Semaphores
- Classical synchronization problems
Hardware Support For Synchronization

- **Disallow interrupts**
  - simplicity
  - widely used
  - problem: interrupt service latency
  - problem: what about multiprocessors?

- **Atomic operations:**
  - Operations that check and modify memory areas in a single step (i.e. operation can not be interrupted)
  - Test-And-Set
  - Exchange, Swap, Compare-And-Swap

---

Test-And-Set

```c
bool TestAndSet(bool & var) {
    bool temp;
    temp = var;
    var = TRUE;
    return temp;
}
```

```c
bool lock; /* init to FALSE */
while (TRUE) {
    while (TestAndSet(lock)) no_op;
    critical section;
    lock = FALSE;
    remainder section;
}
```
**Exchange (Swap)**

```c
void Exchange(bool & a, bool & b) {
    bool temp;
    temp = a;
    a = b;
    b = temp;
}
```

```c
bool lock; /*init to FALSE*/
while (TRUE) {
    dummy = TRUE;
    do Exchange(lock, dummy);
    while (dummy);
    critical section;
    lock = FALSE;
    remainder section;
}
```

**Compare-And-Swap**

```c
bool Compare&Swap (Type * x, Type old, Type new) {
    if *x == old {
        *x = new;
        return TRUE;
    } else {
        return FALSE
    }
}
```
Some Fun with Compare-and-Swap: Lock-Free Concurrent Data Structures

Example: Shared Stack

PUSH element C onto stack:

1. Create C
2. C.next = head
3. head = C

Solution: compare-and-swap(head, C.next, C), i.e. compare and swap head, new value C, and expected value C.next.
If fails, go back to step 2.
Synchronization: Critical Sections & Semaphores

- **Why?** Examples
- **What?** The Critical Section Problem
- **How?** Software solutions
  - Hardware-supported solutions
- **The basic synchronization mechanism:** Semaphores
- **Classical synchronization problems**

Semaphores

- Problems with solutions above:
  - Although requirements simple (mutual exclusion), addition to programs complex.
  - Based on busy waiting.
- A Semaphore variable has two operations: wait (or P) and signal (or V)
  - `P(Semaphore * s);`
    /* Decrement value of s by 1. If the value becomes negative, the process invoking the P operation is blocked. */
  - `V(Semaphore * s);`
    /* Increment value of s by 1 in a single indivisible action. If value is not positive, then a process blocked by a P is unblocked*/
- **Binary semaphore:** The value of s can be either 1 or 0 (TRUE or FALSE). a.k.a mutex lock
- **General semaphore:** The value of s can be any integer (counting semaphore)
Effect of Semaphores

- Synchronization using semaphores:
  
  ```
  s.value = 0
  ```

  - Mutual exclusion with semaphores:
  
  ```
  BinSemaphore * s;
  /* init to TRUE*/

  while (TRUE) {
      P(s);
      critical section;
      V(s);
      remainder section;
  }
  ```

Implementation (with busy waiting): Spinlock

- Binary Semaphores:
  
  ```
  P(BinSemaphore * s) {
      key = FALSE;
      do exchange(s.value, key);
      while (key == FALSE);
  }
  ```

  ```
  V(BinSemaphore * s) {
      s.value = TRUE;
  }
  ```

- General Semaphores:
  
  ```
  BinSemaphore * mutex /*TRUE*/
  BinSemaphore * delay /*FALSE*/

  P(Semaphore * s) {
      P(mutex);
      s.value = s.value - 1;
      if (s.value < 0)
          { V(mutex); P(delay); }
      else V(mutex);
  }

  V(Semaphore * s) {
      P(mutex);
      s.value = s.value + 1;
      if (s.value <= 0) V(delay);
      V(mutex);
  }
  ```
### Implementation (“without” busy waiting)

```c
Semaphore * s) {
    while (TestAndSet(lock))
        no_op;
    s.value = s.value - 1;
    if (s.value < 0) {
        append(this_process, s.L);
        lock = FALSE;
        sleep();
    }
    lock = FALSE;
}
```

```c
Semaphore * s) {
    while (TestAndSet(lock))
        no_op;
    s.value = s.value + 1;
    if (s.value <= 0) {
        PCB * p = remove(s.L);
        wakeup(p);
    }
    lock = FALSE;
}
```

### Deadlock and Starvation

- **Deadlock** - Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let $s$ and $q$ be two semaphores initialized to 1.

  - $P_0$
    - wait ($S$);
    - wait ($Q$);
    - ...
    - signal ($S$);
    - signal ($Q$);
  
  - $P_1$
    - wait ($Q$);
    - wait ($S$);
    - ...
    - signal ($Q$);
    - signal ($S$);

- **Starvation** - Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process.
Synchronization: Critical Sections & Semaphores

- Why? Examples
- What? The Critical Section Problem
- How? Software solutions
- Hardware-supported solutions
- The basic synchronization mechanism:
  Semaphores
- Classical synchronization problems

Classical Problems: Producer-Consumer

Semaphore * n; /* initialized to 0 */
BinSemaphore * mutex; /* initialized to TRUE */

Producer:
while (TRUE) {
    produce item;
P(mutex);
deposit item;
V(mutex);
V(n);
}

Consumer:
while (TRUE) {
P(n);
P(mutex);
remove item;
V(mutex);
consume item;
}
Classical Problems:

Producer-Consumer with Bounded Buffer

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>* full; /* initialized to 0 */</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>* empty; /* initialized to n */</td>
</tr>
<tr>
<td>BinSemaphore</td>
<td>* mutex; /* initialized to TRUE */</td>
</tr>
</tbody>
</table>

**Producer:**

```c
while (TRUE) {
    produce item;
P(empty);
P(mutex);
deposit item;
V(mutex);
V(full);
}
```

**Consumer:**

```c
while (TRUE) {
    P(full);
P(mutex);
remove item;
V(mutex);
V(empty);
consume item;
}
```

Classical Problems: Readers/Writers

- Multiple readers can access data element concurrently.
- Writers access data element exclusively.

<table>
<thead>
<tr>
<th>Semaphore</th>
<th>* mutex, * wrt; /* initialized to 1 */</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>nreaders; /* initialized to 0 */</td>
</tr>
</tbody>
</table>

**Reader:**

```c
P(mutex);
nreaders = nreaders + 1;
if (nreaders == 1) P(wrt);
V(mutex);
do the reading ....
P(mutex);
nreaders = nreaders - 1;
if (nreaders = 0) V(wrt);
V(mutex);
```

**Writer:**

```c
P(wrt);
do the writing ...;
V(wrt);
```
Incorrect Implementation of Readers/Writers

```java
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    /* READERS */
    procedure startRead() {
        while (numberOfWriters != 0);
        numberOfReaders = numberOfReaders + 1;
    }
    procedure finishRead() {
        numberOfReaders = numberOfReaders - 1;
    }
    /* WRITERS */
    procedure startWrite() {
        numberOfWriters = numberOfWriters + 1;
        while (busy || (numberOfReaders > 0));
        busy = TRUE;
    }
    procedure finishWrite() {
        numberOfWriters = numberOfWriters - 1;
        busy = FALSE;
    }
};
```

A Correct Implementation

```java
monitor ReaderWriter{
    int numberOfReaders = 0;
    int numberOfWriters = 0;
    boolean busy = FALSE;
    condition okToRead, okToWrite;
    /* READERS */
    procedure startRead() {
        if (busy || (okToWrite.lqueue)) okToRead.wait;
        numberOfReaders = numberOfReaders + 1;
        okToRead.signal;
    }
    procedure finishRead() {
        numberOfReaders = numberOfReaders - 1;
        if (numberOfReaders = 0) okToWrite.signal;
    }
    /* WRITERS */
    procedure startWrite() {
        if (busy || (numberOfReaders > 0)) okToWrite.wait;
        busy = TRUE;
    }
    procedure finishWrite() {
        busy = FALSE;
        if (okToWrite.lqueue) okToWrite.signal;
        else okToRead.signal;
    }
};
```
<table>
<thead>
<tr>
<th>Synchronization: Critical Sections &amp; Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Why?</strong>  Examples</td>
</tr>
<tr>
<td>• <strong>What?</strong> The Critical Section Problem</td>
</tr>
<tr>
<td>• <strong>How?</strong>  Software solutions</td>
</tr>
<tr>
<td>• Hardware-supported solutions</td>
</tr>
<tr>
<td>• The basic synchronization mechanism:</td>
</tr>
<tr>
<td>• Semaphores</td>
</tr>
<tr>
<td>• Classical synchronization problems</td>
</tr>
<tr>
<td>• <strong>More sophisticated synchronization</strong></td>
</tr>
<tr>
<td>• mechanisms: <strong>Monitors</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Higher-Level Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Semaphores as the “GOTO” among the</td>
</tr>
<tr>
<td>synchronization primitives.</td>
</tr>
<tr>
<td>• very powerful, but tricky to use.</td>
</tr>
<tr>
<td>• Need higher-abstraction primitives, for example:</td>
</tr>
<tr>
<td>• Monitors</td>
</tr>
<tr>
<td>• <strong>synchronized</strong> primitive in JAVA</td>
</tr>
<tr>
<td>• Protected Objects (Ada95)</td>
</tr>
<tr>
<td>• Conditional Critical Region</td>
</tr>
<tr>
<td>• ...</td>
</tr>
</tbody>
</table>
Monitors (Hoare / Brinch Hansen, 1973)

- Safe and effective sharing of abstract data types among several processes.
- Monitors can be modules, or objects.
  - local variable accessible only through monitor’s procedures
  - process can enter monitor only by invoking monitor procedure
- Only one process can be active in monitor.
- Additional synchronization through conditions (similar to semaphores)
  
  ```
  Condition c;
  c.cwait() : suspend execution of calling process and enqueue it on condition c. The monitor now is available for other processes.
  c.csignal() : resume a process enqueued on c. If none is enqueued, do nothing.
  ```
  -.cwait/csignal different from P/V: cwait always waits, csignal does nothing if nobody waits.

Structure of Monitor

- local (shared) data
- procedure 1
- procedure 2
- ... (procedures)
- operations
- initialization code
- urgent queue
**Example: Binary Semaphore**

```plaintext
monitor BinSemaphore {
    bool locked; /* Initialize to FALSE */
    condition idle;

e  entry void P() {
        if (locked) idle.cwait();
        locked = TRUE;
    }

e  entry void V() {
        locked = FALSE;
        idle.csignal();
    }
}
```

**Example: Bounded Buffer Producer/Consumer**

```plaintext
monitor boundedbuffer {
    Item buffer[N]; /* buffer has N items */
    int nextin; /* init to 0 */
    int nextout; /* init to 0 */
    int count; /* init to 0 */
    condition notfull; /* for synchronization */
    condition notempty;

e  void deposit(Item x) {
        if (count == N)
            notfull.cwait();
        buffer[nextin] = x;
        nextin = nextin + 1 mod N;
        count = count + 1;
        notempty.csignal();
    }

e  void remove(Item & x) {
        if (count == 0)
            notempty.cwait();
        x = buffer[nextout];
        nextout = nextout + 1 mod N;
        count = count - 1;
        notfull.csignal();
    }
}
```
Monitors: Issues, Problems

- By definition, the above csignal(c) resumes the execution of one suspended process if there is one.
  - the resumed process must be dispatched immediately otherwise other processes may enter the critical section and the condition under which the process was activated could change.
- What happens when the x.csignal() operation invoked by process P wakes up a suspended process Q?
  - Q waits until P leaves monitor?
  - P waits until Q leaves monitor?
    - csignal() vs cnotify()
- Nested monitor call problem.

Synchronization in JAVA

- Critical sections:
  - synchronized statement
- Synchronized methods:
  - Only one thread can be in any synchronized method of an object at any given time.
  - Realized by having a single lock (also called monitor) per object.
- Synchronized static methods:
  - One lock per class.
- Synchronized blocks:
  - Finer granularity possible using synchronized blocks
  - Can use lock of any object to define critical section.
- Additional synchronization:
  - wait(), notify(), notifyAll()
Java Synchronized Methods:
vanilla Bounded Buffer Producer/Consumer

```java
public class BoundedBuffer {
    Object[] buffer;
    int nextin;
    int nextout;
    int size;
    int count;

    synchronized public deposit(Object x) {
        if (count == size) nextin.wait();
        buffer[nextin] = x;
        nextin = (nextin + 1) % N;
        count = count + 1;
        nextout.notify();
    }

    synchronized public Object remove() {
        Object x;
        if (count == 0) nextout.wait();
        x = buffer[nextout];
        nextout = (nextout + 1) % N;
        count = count - 1;
        nextin.notify();
        return x;
    }

    public BoundedBuffer(int n) {
        size = n;
        buffer = new Object[size];
        nextin = 0;
        nextout = 0;
        count = 0;
    }
}
```

Example: Synchronized Block
(D. Flanagan, JAVA in a Nutshell)

```java
public static void SortIntArray(int[] a) {
    // Sort array a. This is synchronized so that
    // some other thread cannot change elements of
    // the array while we are sorting it.
    // At least no other thread that protect their
    // accesses to the array with synchronized.
    // do some non-critical stuff here...
    synchronized (a) {
        // do the array sort here.
    }
    // do some other non-critical stuff here...
}
```