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)
The Critical Section Problem: Example 1

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

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

**Race condition!**

The Critical Section Problem: Example 2

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

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

```
Consumer:
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></th>
<th>Producer</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>counter = counter + 1</td>
<td>counter = counter - 1</td>
</tr>
<tr>
<td>on CPU</td>
<td>reg₁ = counter</td>
<td>reg₂ = counter</td>
</tr>
<tr>
<td></td>
<td>reg₁ = reg₁ + 1</td>
<td>reg₂ = reg₂ - 1</td>
</tr>
<tr>
<td></td>
<td>counter = reg₁</td>
<td>counter = reg₂</td>
</tr>
<tr>
<td>interleaved execution</td>
<td>reg₁ = counter</td>
<td>reg₂ = counter</td>
</tr>
<tr>
<td></td>
<td>reg₁ = reg₁ + 1</td>
<td>reg₂ = reg₂ - 1</td>
</tr>
<tr>
<td></td>
<td>counter = reg₁</td>
<td>counter = reg₂</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>new1.next = curr.next;</td>
<td>...</td>
</tr>
<tr>
<td>new1.prev = c.next.prev;</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>new2.next = curr.next;</td>
</tr>
<tr>
<td>...</td>
<td>new2.prev = c.next.prev;</td>
</tr>
<tr>
<td>...</td>
<td>curr.next = new2;</td>
</tr>
<tr>
<td>curr.next = new1;</td>
<td>new.next.prev = new2;</td>
</tr>
<tr>
<td>new.next.prev = new1;</td>
<td>...</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. Only one process at a time can enter the critical section.
2. A process that halts in non-critical section cannot prevent other processes from entering the critical section.
3. A process requesting to enter a critical section should not be delayed indefinitely.
4. When no process is in a critical section, any process that requests to enter the critical section should be permitted to enter without delay.
5. Make no assumptions about the relative speed of processors (or their number).
6. 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 $P_1$
- int turn; /* turn == i: $P_i$ is allowed to enter c.s. */

```c
P_i: while (TRUE) {
    while (turn != i) no_op;
    critical section;
    turn = j;
    remainder section;
}
```
Another Wrong Solution

```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;
}
```

Yet Another Wrong Solution

```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;
}
```
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 cannot be interrupted)
  - Test-And-Set
  - Exchange, Swap, Compare-And-Swap

---

Test-And-Set

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

```
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;
}
```

**Mutual Exclusion with Exchange**

```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) {
    atomic!
    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:

```
head ----> A ----> B
   \( \rightarrow \)
    \( C \)
```

1. Create \( C \)
2. \( C.next = \text{head} \)
3. \( \text{head} = C \)

---

Some Fun with Compare-and-Swap: Lock-Free Concurrent Data Structures

Example: Shared Stack

PUSH element \( C \) onto stack: What can go wrong?!

```
head ----> A ----> B
   \( \rightarrow \)
    \( C \)
```

1. Create \( C \)
2. \( C.next = \text{head} \)
   \( \text{context switch} \)
   1. Create \( C' \)
   2. \( C'.next = \text{head} \)
   3. \( \text{head} = C' \)
   \( \text{context switch back!} \)
3. \( \text{head} = C \)

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:
  - \( \text{V(Semaphore } \ast s) \); /* Increment value of \( s \) by 1 in a single indivisible action. If value is not positive, then a process blocked by \( P \) is unblocked*/
  - \( \text{P(Semaphore } \ast s) \); /* Decrement value of \( s \) by 1. If the value becomes negative, the process invoking the \( P \) operation is blocked. */
- Binary semaphore: The value of \( s \) can be either 1 or 0 (TRUE or FALSE).
- General semaphore: The value of \( s \) can be any integer.
Effect of Semaphores

- Synchronization using semaphores:
  - s.value = 0
  - P(s)
  - V(s)
  - P(s)
  - V(s)

- Mutual exclusion with semaphores:
  ```
  BinSemaphore * s;
  /* init to TRUE*/
  
  while (TRUE) {
    P(s);
    critical section;
    V(s);
  }
  ```

Implementation (with busy waiting)

- 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

bool lock;
/* init to FALSE */
int value;
PCBList * l;
```

```
P(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;
}
```

```
V(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;
}
```

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:**

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

**Consumer:**

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

Classical Problems: **Producer-Consumer with Bounded Buffer**

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

**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:**

**The Barbershop**

```c
Semaphore * max_capacity;
/* init to 20 */
Semaphore * sofa;
/* init to 4 */
Semaphore * barber_chair;
/* init to 3 */
Semaphore * coord;
/* init to 3 */
Semaphore * cust_ready;
/* init to 0 */
Semaphore * leave_b_chair;
/* init to 0 */
Semaphore * payment;
/* init to 0 */
Semaphore * receipt;
/* init to 0 */
```

**The Barbershop (cont)**

**Process customer:**

```c
P(max_capacity);
<enter shop>
P(sofa);
<sit on sofa>
P(barber_chair);
<get up from sofa>
V(sofa);
<sit in barber chair>
V(cust_ready);
P(finished);
<leave barber chair>
V(leave_b_chair);
<pay>
V(payment);
P(receipt);
<exit shop>
V(max_capacity);
```

**Process cashier:**

```c
for(;;){
P(payment);
P(coord);
<accept pay>
V(coord);
V(receipt);
}
```

**Process barber:**

```c
for(;;){
P(cust_ready);
P(coord);
<cut hair>
V(coord);
V(finished);
P(leave_b_chair);
P(barber_chair);
V(max_capacity);
```
**The Fair Barbershop**

**Process customer:**
- `P(max_capacity);`
- `<enter shop>`
- `P(mutex1);`
- `custnr := ++count;`
- `V(mutex1);`
- `P(sofa);`
- `<sit on sofa>`
- `P(barber_chair);`
- `<get up from sofa>`
- `V(sofa);`
- `<sit in barber chair>`
- `P(mutex2);`
- `enqueue(custnr);`
- `V(cust_ready);`
- `V(mutex2);`
- `P(finished[custnr]);`
- `<leave barber chair>`
- `V(leave_b_chair);`
- `<pay>`
- `V(payment);`
- `P(receipt);`
- `<exit shop>`
- `V(max_capacity);`

**Process cashier:**
- `for(;;){`
  - `P(payment);`
  - `P(coord);`
  - `<accept pay>`
  - `V(coord);`
  - `V(receipt);`
- `}`

**Process barber:**
- `for(;;){`
  - `P(cust_ready);`
  - `P(mutex2);`
  - `enqueue(b_cust);`
  - `V(mutex2);`
  - `P(coord);`
  - `<cut hair>`
  - `V(coord);`
  - `V(finished[b_cust]);`
  - `P(leave_b_chair);`
  - `V(barber_chair);`
- `}`

---

**Classical Problems: Readers/ Writers**

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

```
Semaphore * mutex, * wrt; /* initialized to 1 */
int nreaders; /* initialized to 0 */

**Reader:**
```
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:**
```
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;
    }
}
```
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
- More sophisticated synchronization mechanisms: Monitors

Higher-Level Synchronization Primitives

- Semaphores as the "GOTO" among the synchronization primitives.
  - very powerful, but tricky to use.

- Need higher-abstraction primitives, for example:
  - Monitors
    - synchronized primitive in JAVA
  - Protected Objects (Ada95)
  - Conditional Critical Region
  - …
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

[Diagram of monitor structure]

- Blocked processes
- Local (shared) data
- Procedure 1
- Procedure 2
- ... procedures
- Operations
- Initialization code
- Urgent queue
Example: Binary Semaphore

```c
monitor BinSemaphore {

    bool locked; /* Initialize to FALSE */
    condition idle;

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

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

Example: Bounded Buffer Producer/Consumer

```c
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;

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

    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**

- 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()`
  - Realized as methods for all objects
Java Synchronized Methods:  
vanilla Bounded Buffer Producer/Consumer

```
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) mod N;
        count = count + 1;
        nextout.notify();
    }

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

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

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

```
public static void SortIntArray(int[] a) {
    // Sort array a. This is synchronized so that
    // some other thread cannot change elements of
    // the array or traverse 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...
}
```