Synchronization: Recap

- Why?
  - Example
- The Critical Section Problem (recap!)
- Hardware Support for Synchronization
- Lock-free operations
- Semaphores
- Monitors

- Reading: Silberschatz, Ch. 6

Critical Section Problem: Example

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!

Process 1
new1.next = curr.next;
new1.prev = c.next.prev;
...
curr.next = new1;
new1.next.prev = new1;

Process 2
...new2.next = curr.next;
new2.prev = c.next.prev;
curr.next = new2;
new2.next.prev = new2;
...

Must guarantee mutually exclusive access to list data structure!

The Critical Section Problem

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

```
while (TRUE) {
    enter section;
    critical section;
    exit section;
    remainder section;
}
```
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.*/

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

Another Wrong Solution

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

$$P_i: \texttt{while (TRUE) \{ }
\begin{align*}
\texttt{while (flag[j]) no_op; } \\
\texttt{flag[i] = TRUE; } \\
\texttt{critical section; } \\
\texttt{flag[i] = FALSE; } \\
\texttt{remainder section; } \\
\texttt{\}}
$$
Yet Another Wrong Solution

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

P_i: 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 */

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

    critical section;
    flag[i] = FALSE;

    remainder section;
}
```
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
  - Fetch-And-Add
  - Exchange, Swap, Compare-And-Swap
  - Load-Link/Store Conditional

---

Hardware Support: Test-And-Set

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

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

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

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

Hardware Support: Fetch & Add

```c
function FetchAndAdd(&location) {
    int value = location;
    location = value + 1;
    return value;
}

record locktype {
    int ticketnumber; int turn;
}
procedure LockInit( locktype * lock ) {
    lock.ticketnumber = 0;
    lock.turn = 0;
}
procedure Lock( locktype * lock ) {
    int myturn = FetchAndAdd( &lock.ticketnumber );
    while (lock.turn != myturn)
        skip; // spin until lock is acquired
}
procedure UnLock( locktype* lock {
    FetchAndAdd( &lock.turn )
}
```
Hardware Support: Compare-And-Swap

```c
bool CompareAndSwap(Type * x, Type old, Type new) {
    if (*x == old) {
        *x = new;
        return TRUE;
    } else {
        return FALSE;
    }
}
```

```c
bool lock; /*init to FALSE*/
while (TRUE) {
    while(!C&S(&lock, false, true));
    critical section;
    lock = FALSE;
    remainder section;
}
```

Compare-and-Swap: Example
Lock-Free Concurrent Data Structures

**Example:**  
**Shared Stack**

PUSH element `C` onto stack:

1. Create `C`
2. `C.next = head`
3. `head = C`
**Compare-and-Swap: Example**

**Lock-Free Concurrent Data Structures**

**Example:** Shared Stack

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

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

\[ \xrightarrow{\text{context switch}} \]

1. Create \( C' \)
2. \( C'\).next = head
3. head = \( C' \)
   \[ \xrightarrow{\text{context switch back}} \]
4. head = \( C \)

**Solution:** \( \text{compare-and-swap}(\text{head, } C\text{.next, } C) \),
i.e. compare and swap \( \text{head, new value } C \), and expected value \( C\text{.next} \).
If fails, go back to step 2.

---

**Push Operation:**

```cpp
void push(sometype t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!\text{C}\&\text{S}(\&\text{head, } node->\text{next, } node));
}
```
Compare-and-Swap: Example
Lock-Free Concurrent Data Structures

Example: Shared Stack

**Pop Operation:**
```cpp
bool pop(sometype & t) {
    for(;;) {
        Node* ret_ptr = head;
        if (ret_ptr == null) return false;
        Node* next_ptr = ret_ptr->next;
        if(C&S(&head, ret_ptr, next_ptr)) {
            t = current->data;
            return true;
        }
    }
}
```

Compare-And-Swap is “weak”: LL/SC

- CSW does not detect updates if old value has been restored! (so-called ABA problem)
- Solution: “strong” pair of instructions:
  - load-link (LL): returns current value of memory location
  - subsequent store-conditional (SC) stores a new value
    - only if no updates of memory location since LL
    - otherwise SC fails
- Supported on MIPS, PowerPC, Alpha, ARM
- Implementation of LL/SC are often not perfect, e.g.:
  - any exception between LL/SC may cause SC to fail
  - any updates over memory bus may cause SC to fail
Semaphores

• Problems with solutions above:
  - Although requirements simple (mutual exclusion), addition to programs complex.
  - Based on busy waiting.
• A Semaphore variable has two operations:
  - \( V(Semaphore \; * \; s); \)
    /* 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*/
  - \( P(Semaphore \; * \; s); \)
    /* 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

• General Synchronization using semaphores:

```
BinSemaphore * s;
/* init to TRUE*/

while (TRUE) {
    P(s);
    critical section;
    V(s);
    remainder section;
}
```
**Implementation (with busy waiting)**

- **Binary Semaphores:**

  \[
P(BinSemaphore * s) \{
    \text{key} = \text{FALSE};
    \text{do exchange}(s.\text{value}, \text{key});
    \text{while (key == FALSE)};
  \}
\]

  \[
V(BinSemaphore * s) \{
  s.\text{value} = \text{TRUE};
\}
\]

- **General Semaphores:**

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

  \[
V(Semaphore * s) \{
    P(mutex);
    s.\text{value} = s.\text{value} + 1;
    \text{if (s.\text{value} <= 0) V(delay);
    V(mutex); }
  \}
\]

**Implementation (“without” busy waiting)**

```c
Semaphore

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

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

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

**Consumer:**

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

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**
- **procedure k**
- **operations**
- **blocked processes**
- **urgent queue**
- **initialization code**

---

### 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();
  }
}
```

Incorrect Implementation of Readers/Writers

```c
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 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

```java
class BoundedBuffer {
  Object[] buffer;
  int nextin, nextout;
  Object notfull, notempty;
  int size;
  int count;

  synchronized public deposit(Object x) {
    if (count == size) notfull.wait();
    buffer[nextin] = x;
    nextin = (nextin+1) mod size;
    count = count + 1;
    notempty.notify();
  }

  synchronized public Object remove() {
    Object x;
    if (count == 0) notempty.wait();
    x = buffer[nextout];
    nextout = (nextout+1) mod size;
    count = count - 1;
    notfull.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 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...
}
```