Achieving Synchronization

- OS must provide mechanisms and primitives to achieve synchronization
  - Should support multiple contending processes/threads
  - Should be efficient and fair
  - Should be easy to use

- What are these mechanisms and how to use/implement them?

Critical Section Solution: Locks

- Acquire Lock
- Critical section
- Release Lock
- Remainder section

- How to implement locks?
Disable Interrupts

- Do not preempt process in critical section
- Problems?
  - Inefficient
  - Unfair
  - Infeasible in multiprocessor systems
- Can be used:
  - For small pieces of code
  - On uniprocessors
  - For kernel code

Synchronization Hardware

- Atomic instructions
- Uninterruptible unit of execution
- Finishes completely or not at all
- Concurrent execution is sequentialized
- TestAndSet: Atomically tests the value of a variable and sets its value
- Swap: Atomically swaps the values of two variables

TestAndSet

```c
Boolean TestAndSet(boolean *target) {
    boolean ret = *target;
    *target = TRUE;
    return ret;
}
```

```c
while(1){
    while (TestAndSet(&lock)) /* do nothing */;
    Critical Section
    lock = FALSE;
    Remainder Section
}
```

Swap

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

```c
while(1){
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);
    Critical Section
    lock = FALSE;
    Remainder Section
}
```
**Busy Waiting**

- Waiting process has to loop continuously
- Problems with busy waiting?

**Blocking**

- Block a process if critical section is occupied
  - Context switch to another process
- Wake up a waiting process when critical section becomes free
  - Pick one if multiple waiting processes
  - How to pick?

**Mutex Locks**

- Mutex Lock: Protects access to a shared resource
  - A process locks a resource before accessing it
  - Another process will have to wait for the resource to be unlocked
  - The first process would unlock the resource after accessing it

**Mutex States**

- Locked: A single process holds the mutex
  - Each lock can have only one owner at a time
  - Another process trying to lock mutex will be blocked
- Unlocked: No process holds the mutex
  - A process trying to lock mutex will succeed and get control of the mutex
**Mutex Operations**

- Acquire: Gain control of the mutex
  - Get lock if free
  - Block if already locked
- Release: Release the mutex
  - Unblock a waiting process if any
  - Unblocked process becomes new owner of mutex
- Both these operations are atomic
  - Multiple processes cannot execute these operations on same lock concurrently

**Mutex Example**

```plaintext
mutex mut;

counter=1;

Process A:
acquire(mut);
counter++;
release(mut);

Process B:
acquire(mut);
counter--;
release(mut);
```

**Spin Locks**

- Blocking avoids problems of busy waiting
- What if:
  - Critical section is small?
  - Executing on a multiprocessor system?
- Spin Lock: Busy wait on a lock
  - Might be more efficient than blocking/context-switching
  - Useful for multiprocessors

**Semaphores**

- A powerful software-based synchronization mechanism
  - Allows mutual exclusion as well as ordering-based synchronization
- Semaphore is an integer
  - Initialized with a non-negative value
  - The value determines the access control and synchronization semantics
  - Value can be modified only through two operations: wait, signal
Semaphore Operations

- **Wait**
  - Also called P (proberen), down, lock
  - Either decrements value or spins if value is 0
- **Signal**
  - Also called V (verhogen), up, unlock
  - Increments value
- Operations on value of S are atomic
  - Mutual exclusion while executing these operations

```c
wait(S) {
    while (S<=0)
        /* do nothing */
    S--;
}
signal(S) {
    S++;
}
```

Usage: Mutual Exclusion

- Each process must call wait and signal in correct order
- How many processes can run in critical section if:
  - Initial value of S is 1
  - Initial value of S is 10
  - Initial value of S is 0

Usage: Execution Ordering

<table>
<thead>
<tr>
<th>Process A:</th>
<th>Process B:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement 1; signal(S);</td>
<td>wait(S); Statement 2;</td>
</tr>
</tbody>
</table>

- In which order would the processes run if:
  - Initial value of S is 0
  - Initial value of S is 1

Semaphore Types and Usage

- **Binary semaphores:**
  - Can take only \{0,1\} values
  - Used to provide mutual exclusion on critical sections
- **Counting semaphores:**
  - Can take arbitrary values
  - Used to control access to multi-instance resources
- Can use semaphores to provide:
  - Mutual exclusion
  - Controlled access to finite resources
  - Execution ordering
  - Behavior depends on initial value of semaphore
Semaphore Implementation

- So far:
  - Busy waiting in wait operation
- What if critical section is large?
- Would prefer:
  - Blocking waiting process
  - Switch to another runnable process

Blocking Semaphores

- Semaphore can be thought of as a C struct:
  - Counter
  - Wait list
- Initial: non-negative value
- Wait
  - Either decrements counter or blocks if counter is 0
- Signal
  -Either increments counter or wakes up a blocked process

```
struct {
  int counter;
  process_list list;
}
S->counter--;  
if (S->counter < 0) {
  add process to S->list;
  block;
}
S->counter++;  
if (S->counter <=0) {
  remove a process P from S->list;
  wakeup(P);
}
```

Semaphore Semantics

- Counter can be negative
  - What does the magnitude signify?

```
wait(s)
Critical section
signal(s)
```

- Suppose initial value of S->counter is 1. What if:
  - 2 processes execute wait?
  - 10 processes execute wait?

Implementation Issues

- How to implement wait list?
  - List of PCBs for each semaphore
  - Adding/removing processes from the list?
- How to atomically execute wait and signal?
  - Critical section problem
  - Uniprocessor: Disable interrupts
  - Multiprocessor: Use spinlocks
- Why is it ok to disable interrupts or busy wait here?
Using Semaphores for Classic Synchronization Problems

- Bounded-Buffer Problem
- Readers-Writers Problem

Bounding-Buffer Problem

Producer: while(1) {
    produce(item);
    insert(item, buffer);
}

Consumer: while(1) {
    remove(item, buffer);
    consume(item);
}

Bounded-Buffer Problem With Semaphores

item buffer[N]
Semaphore mutex=1; Semaphore items=0, slots=N;

Producer: while(1) {
    produce(item);
    wait(slots);
    wait(mutex);
    insert(item, buffer);
    signal(mutex);
    signal(items);
}

Consumer: while(1) {
    wait(items);
    wait(mutex);
    remove(item, buffer);
    signal(mutex);
    signal(slots);
    consume(item);
}

Readers-Writers Problem

- Reader: Process that only reads the shared data
- Writer: Process that modifies the shared data

Examples:
- Shared access to a file or database
- Access to a Webpage
Readers-Writers Problem (Contd.)

Reader/writer accesses are not symmetric:
- Any number of readers can be concurrently reading
- Only one writer can be writing at a time (no other readers or writers)

Writer:
while(1) {
    write_data();
}

Reader:
while(1) {
    read_data();
}

Prioritizing Access
- Who should get higher priority of access?
- First RW problem: Readers given higher priority
  - A reader can join if other readers already reading
  - Writer given access only when all readers done
- Second RW problem: Writers given higher priority
  - A reader cannot join if a writer is waiting
  - Writer given access when all existing readers done

First Readers-Writers Problem With Semaphores

Semaphore wrt=1, mutex=1; int readcount=0;

Writer:
while(1) {
    wait(wrt);
    write_data();
    signal(wrt);
}

Reader:
while(1) {
    wait(mutex);
    readcount++;  
    if (readcount==1)
        wait(wrt);
    signal(mutex);
    read_data();
    wait(mutex);
    readcount--;
    if (readcount==0)
        signal(wrt);
    signal(mutex);
}

Reader-writer locks
- Special locks
  - Implement a reader-writer synchronization protocol
  - Acquired in read or write mode
- What types of applications can benefit from these?