CSCI 5103
Operating Systems
Instructor: Abhishek Chandra

Synchronization Tools
- Hardware Synchronization
- Synchronization Mechanisms
  - Mutex Locks
  - Semaphores
- Classic Synchronization Problems

Achieving Synchronization

Entry section
Critical section
Exit section
Remainder section

- How to implement a synchronization protocol?

Synchronization Tools
- 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?
**Synchronization Tools**

- Hardware Synchronization
- Software tools
  - Mutex Locks
  - Semaphores
  - Higher-level tools
    - Monitors

**Hardware Synchronization**

- Use hardware support to achieve synchronization
  - Interrupts
  - Memory barriers
  - Atomic instructions

**Disable Interrupts**

- Do not preempt process in critical section
- How does it achieve synchronization?
- Problems?
  - Inefficient
  - Unfair
  - Infeasible/hard in multiprocessor systems
- Can be used:
  - For small pieces of code
  - On uniprocessors
  - For kernel code

**Memory Barrier**

- Ensures memory operation ordering
  - All prior loads and stores are completed
  - Subsequent loads and stores cannot start earlier

\[
\begin{align*}
x &= 100; \\
\text{memory_barrier();} \\
\text{flag = TRUE; } \\
\text{print flag} \\
\text{memory_barrier();} \\
\text{print x; }
\end{align*}
\]

- Can help with correct implementation of software solutions (like Peterson’s solution)
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
- CompareAndSwap: Atomically compares to a value and swaps the value if true

TestAndSet

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

```c
global boolean lock = FALSE;
while(1){
    while (test_and_set(&lock)) /* do nothing */;
    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?
- Is blocking always better than busy waiting?

**Spin Locks**

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

**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

**Critical Section Solution: Mutex Locks**

1. Acquire lock
2. Critical section
3. Release lock
4. Remainder section

- How to implement locks?
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

```c
counter=1;
mutex mut;

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

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

Synchronization: Other Cases

- Mutual exclusion:
  - Useful to prevent concurrent access/modification
- May also need:
  - Execution ordering
  - Signaling
  - Generalizing mutual execution
  - Allow access to n processes
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

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

- Operations on value of S are atomic
  - Mutual exclusion while executing these operations

Semaphore Usage

- Initialize the semaphore
  - Initializes the internal counter
  - Must be initialized with a non-negative value
- Call wait and signal operations
  - Can be called in any order depending on synchronization requirements
- Cannot directly access or examine counter value
  - Only wait and signal operations allowed

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;</td>
<td>wait(S);</td>
</tr>
<tr>
<td>signal(S);</td>
<td>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

```c
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);
               // }```

```c
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

Bounded-Buffer Problem

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

```c
Consumer: while(1) {
    remove(item, buffer);
    consume(item);
}
```
Bounded-Buffer Problem With Semaphores

```c
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)

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?