Chapter 6
Process Synchronization

Cooperating Process
- process that can affect or be affected by other processes
- directly share a logical address space (threads)
- be allowed to share data via files or messages
- concurrent access to shared data may result in data inconsistency

Race Condition
- Multiple processes access and manipulate a variable counter concurrently
- Outcome depends upon the particular order in which the access takes place

Process Synchronization & Coordination

Critical Section Implementation

do {
    Entry Section
    Critical Section
    Exit Section
    Remainder Section
} while(true);

Operating System Critical Sections (Race Conditions)
- Updating kernel data structures
  - Open Files List
  - Memory Allocation Structures
  - Active Process Lists
  - Inactive Process Lists
  - Interrupt Handling

Preemptive Kernels
- allows a process to be preempted while running in kernel mode
- may be subjected to race conditions
- difficult to design & implement in SMP
- highly desired in Real-Time Systems

Non-preemptive Kernels
- does not allow a process to be preempted while running in kernel mode
- process will run until it
  - exits kernel mode
  - blocks
  - voluntarily yields control of the CPU
- not subject to race conditions
Critical Section Requirements

- **Mutual Exclusion**
  - only one process may be in its critical section

- **Progress**
  - no process is in its critical section
  - one or more processes wants to enter its critical section
  - only those processes not in their remainder sections can decide which process can enter its critical section
  - the decision cannot be postponed indefinitely

- **Bounded Waiting**
  - there is a bound on the number of times other processes may enter their critical sections
  - after a process has made a request to enter its critical section

Peterson’s Solution == software solution
fails because of pipelines & parallel instruction streams

Synchronization Hardware
Atomic Hardware Instructions

1. TestAndSet(&lock);

   **Hardware Definition**

   ```
   boolean TestAndSet( boolean *target );
   {
       boolean rv = *target;
       *target = TRUE;
       return rv;
   }
   ```

   **Software Implementation**

   ```
   lock = false;
   do
   {
       while(TestAndSet(&lock));
       // critical section
       lock = FALSE;
       // remainder section
   }while(TRUE);
   ```

   Effect:
   lock = FALSE;
   executes TestAndSet(&lock);
   P₀ gains access to its critical section; lock = TRUE;
   P₁ executes TestAndSet(&lock);
   P₁ hangs in the while(TestAndSet(&lock)) loop, i.e.,
   spin cycle since (lock == TRUE) until P₀ exits its
   critical section

   ```
   P₀
   do
   {
       while(TestAndSet(&lock));
       // critical section
       lock = FALSE;
       // remainder section
   }while(TRUE);
   ```

   ```
   P₁
   do
   {
       while(TestAndSet(&lock));
       // critical section
       lock = FALSE;
       // remainder section
   }while(TRUE);
   ```
2. **Swap(&lock, &key)**
   
   **Hardware Definition**
   
   ```c
   void Swap(boolean *a, boolean *b)
   {
      boolean temp = *a;
      *a = *b;
      *b = temp;
   }
   ```
   
   **Software Implementation**
   
   ```c
   do {
      key = TRUE;
      while(key == TRUE)
         Swap(&lock, &key);
      // critical section
      lock = FALSE;
      // remainder section
   } while(TRUE);
   ```

   **Notes:**
   - **P0** executes `key = TRUE`;
   - **P0** executes while(`key == TRUE`) and then
     - **P0** executes `Swap(&lock, &key)``
     - **P0** executes in its critical section
   - **P1** executes `key = TRUE`;
   - **P1** executes while(`key == TRUE`) and spinlocks because its key is `FALSE`
   - **P0** executes `lock = FALSE`;
   - **P1** executes `Swap(&lock, &key)``
   - **fails the bounded waiting requirement**
3. TestAndSet(&lock)

Software Implementation which satisfies Mutual Exclusion, Progress, & Bounded Waiting

\[ P_i \text{ for } i \leq n, \text{ i.e., there are } n+1 \text{ processes} \]

\[ lock = \text{false}; \text{waiting}[i] = \text{false}; \]

\[ P_0 \]
boolean key = FALSE;
do{
    waiting[0] = TRUE;
    key = TRUE;
    while(waiting[0] && key)
        key = TestAndSet(&lock);
    waiting[0] = FALSE;
    // critical section
    j = (i+1)\%n;
    while((j != 1) && !waiting[j])
        j = (j+1)\%n;
    if(j == 0) lock = FALSE;
    else waiting[j] = FALSE;
    // remainder section
}while(TRUE);

\[ P_1 \]
boolean key = FALSE;
do{
    waiting[1] = TRUE;
    key = TRUE;
    while(waiting[1] && key)
        key = TestAndSet(&lock);
    waiting[1] = FALSE;
    // critical section
    j = 2;
    while((j != 1) && !waiting[j])
        j = (j+1)\%2;
    if(j == 2) lock = FALSE;
    else waiting[j] = FALSE;
    // remainder section
}while(TRUE);

\[ P_i \]
boolean key = FALSE;
do{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    // critical section
    j = (i+1)\%n;
    while((j != i) && !waiting[j])
        j = (j+1)\%n;
    if(j == i) lock = FALSE;
    else waiting[j] = FALSE;
    // remainder section
}while(TRUE);

\[ P_0 \]
starts waiting[0] = TRUE, \( P_0 \) key = TRUE
\( P_0 \) enters the while loop and executes TestAndSet(&lock)

\[ \Rightarrow \text{lock = true & & key = FALSE & & waiting[0] = false} \]
\( P_0 \) drops thru the loop and enters its critical section

\[ P_1 \]
starts waiting[1] = TRUE, \( P_1 \) key = TRUE
\( P_1 \) drops thru while loop and executes TestAndSet(&lock)

\[ \Rightarrow \text{lock = true & & key = true} \Rightarrow \text{\( P_1 \) hangs in the while loop} \]

\[ P_i \]
enters Critical Section only if waiting[i] == false; or key == false;
**Semaphores**

API construct – implementation via synchronization hardware

S: semaphore – integer variable
- directly initialized, i.e., assignment statement
- accessed only via atomic operations
  - `wait( )` – proberen (P) “to test”
  - `signal( )` – verhofen (V) “to increment”

```plaintext
Definition -- wait( )
wait(S)
{
    while (S <= 0);
    S--; 
}

Definition -- signal(S)
signal(S)
{
    S++; 
}
```

Counting Semaphore – unrestricted domain – $-\infty < CS < +\infty$

Binary Semaphore domain restricted to $[0, 1]$

**Mutex Locks – Binary Semaphores – provide Mutual Exclusion**
- critical section – multiple processes
  - $P_i$: processes $0 < i < n+1$
  - `mutex mx = n;`
  - gain access to a resource, process executes `wait(mx);`
  - releasing resource, process executes `signal(mx);`
  - `mx == 0` → no resources available

**Synchronization**

Process $P_1$ with statement $S1$
Process $P_2$ with statement $S2$
synchronize statements: $S1$ must execute before $S2$
`mutex synch = 0;`

**Busy Waiting**

CPU cycles that could have been spend doing productive work are wasted processing waits

**Spinlock – multiprocessor systems – no context switch**

One thread can spin on one processor while another thread performs a critical section on another processor
Blocking
- process finds mutex mx <= 0 \(\Rightarrow\) process executes block( ) operation, i.e.,
  - block( ) places process on queue associated with the mutex mx
  - process state is changed to the waiting state
- scheduler selects another process to execute
- executing process executes signal( ) operation
- process at head of wait queue is restarted with a wakeup( ) operation
- reawakened process is placed in the ready queue

Blocking Semaphore

typedef struct
{   int value;
    struct process *list;
} semaphore;

semaphore S;

wait(semaphore *S)
{
    S \rightarrow value--;  
    if ( S \rightarrow value < 0 )
    {
        // add the calling process to S \rightarrow list;
        block( );
    }
}

signal(semaphore *S)
{
    S \rightarrow value++;
    if(S \rightarrow value <= 0)
    {
        // remove a process, P, from S \rightarrow list;
        wakeup(P);
    }
}

Deadlocks

\[
P_0 \quad \quad \quad \quad \quad P_1
\]
wait(S); \quad wait(Q);
wait(Q); \quad wait(S);
|       |       |
signal(S); \quad signal(Q);
signal(Q); \quad signal(S);

block( ) & wakeup( ) are basic system calls, they must also be atomic, i.e., must not be interruptible
- single processor: disable interrupts
- multiprocessor: spinlock

linkedList of process control blocks

Starvation
linked list is a LIFO queue, i.e., stack

resource acquisition and release

The proper use of semaphores is under the control of each individual programmer – one error and the system fails
Priority Inversion
- higher priority process blocked by lower priority process modifying kernel data

Priority Inheritance Protocol
- processes accessing resources which are controlled by a higher priority, e.g., kernel resources, inherit the higher priority until they have finished

Monitor
ADT (class-object) construct that encapsulates the proper sequences of wait( ) and signal( ) to ensure correct usage of semaphores
- monitor may contain multiple (user) procedures
- monitor ensures that only one procedure will be active at any one time
- condition variables
  - condition x;
- condition operations
  - wait( ) & signal( )
    - process $P_0$ invokes x.wait( ); $P_0$ suspended until another process $P_1$ issues x.signal( )
  - x.signal( ) resumes exactly one suspended process
    - if there are no suspended processes, x.signal( ) has no effect

- process $P_1$ issues x.signal( ) $\Rightarrow$ $P_0$ resumes
  - $P_1$ waits until $P_0$ leaves the monitor
  - $P_1$ waits for another condition
  - $P_0$ waits until $P_1$ leaves the monitor
  - $P_0$ waits for another condition

signal( ) used in semaphores always has an effect
Dining-Philosophers Monitor

```cpp
monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5]; // possible states
    condition self[5]; // provides a wait condition

    void pickup(int i)
    {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait();
    }

    void putdown(int i)
    {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i)
    {
        if (!((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) && ((state[(i + 1) % 5] != EATING))
            { state[i] = EATING;
              self[i].signal();
            }
    }

    initialization_code()
    {
        for (int i = 0; i < 5; i++) state[i] = THINKING;
    }
}
```

Usage

```cpp
dp.pickup(i);
...
// Philosopher i eats until full
...
dp.putdown(i);
```

Implementing a Monitor using Semaphores
see Silberschatz pages 250-252
Java Monitors

- `java.util.concurrent`
- synchronized methods – lock acquisition
- lock’s entry set – set of threads waiting for the lock to become available
- `wait()` & `notify()` i.e., `signal()`
- semaphores, condition variables, mutex locks

```java
public class SimpleClass {
    ...
    public synchronized void safeMethod() {
        ...
    }
    ...
}
```

SimpleClass sc = new SimpleClass();

In order to execute `sc.safeMethod()` the thread on which object `sc` is running must own the lock; lock is available ➞ calling thread becomes the owner of the lock

if the lock is owned by another thread, the `sc.safeMethod()` thread blocks and is put on the entry set for the lock

thread exits the method ➞ lock is made available & a thread from the entry set is selected as the new owner of the lock
Solaris

- semaphores (long code segments)
- condition variables (long code segments)
- adaptive mutex (short code segments)
  - standard semaphore – spinlock
  - data locked
    - lock held by thread running on another CPU, spin until released
    - thread holding lock is not running, blocks until awakened by lock release
- reader-writer locks (long code segments)

- turnstile
  - list of threads waiting to acquire
    - adaptive mutex
    - reader-writer lock
  - is a queue structure holding threads blocked on a lock

- turnstile acquisition
  - each synchronized object with a thread blocked on a lock requires its own turnstile
  - each kernel thread has a turnstile

  - the kernel turnstile on which the first thread blocks becomes the turnstile for the synchronized object
    - subsequent threads blocking on this same lock will be added to the same kernel turnstile

  - thread releases the lock
    - releasing object is assigned a new thread from a free turnstile list maintained by the kernel
    - kernel selects a new owner of the lock from the thread’s turnstile

- priority-inheritance protocol
  - lower-priority thread currently holds a lock
  - higher-priority thread is currently blocked on the same lock
  - lower-priority thread acquires the priority of the higher-priority thread until it releases the lock
Windows XP
- multithreaded kernel
- supports Real-Time processes
- multiple processors

Kernel Threads

**UNIPROCESSOR SYSTEMS**
- kernel thread accesses a global resource → disables all interrupt handlers with access to that resource

**MULTIPROCESSOR SYSTEMS**
- kernel thread accesses a global resource
  - uses spinlocks to protect access to that resource
  - ensures that threads holding spinlocks are not preempted

User Threads

Dispatcher Objects
- **Signaled State**
  - object is available;
  - thread will not block when acquiring the object

- **Nonsignaled State**
  - object is not available;
  - thread will block when attempting to acquire the object

- synchronizes threads by using
  - mutexes
  - semaphores
  - events
  - timers

Events
- notify waiting threads that a desired condition has occurred

Times
- notify thread(s) that a specified time interval has expired

Actions
- thread blocks on a nonsignaled dispatcher object
  - thread state changes to waiting
  - thread is placed on a wait queue for that object

- dispatcher object changes state to signaled
  - if there are threads in the object’s wait queue, the kernel moves thread(s) to the ready state; if the queue is for
    - an Event Object → kernel moves all threads waiting for that event
    - a Mutex → kernel moves one thread to the ready state

```
thread attempts to acquire a mutex dispatcher object in a nonsignaled state;
thread is suspended and placed in a wait queue for the mutex object; another
thread releases the mutex lock; thread at the front of the mutex queue has its
state changed to ready and it acquires the mutex lock
```
Linux (starting with Version 2.6) pre-emptive kernel

Kernel Threads
- Spinlocks
- Semaphores
- Reader-Writer Spinlocks
- Reader-Writer Semaphores

**SYMMETRIC MULTIPROCESSOR SYSTEMS**
- acquiring/releasing short duration spinlocks

**UNIPROCESSOR SYSTEMS**
- enabling/disabling kernel preemption
  - preempt_disable()
  - preempt_enable()
- kernel-mode task is holding a lock ➔ kernel is not preemptible

task ↔ thread-info structure
- preempt_count – number of locks held by the task
  - lock acquired ➔ preempt_count ++;
  - lock released ➔ preempt_count --;
- preempt_count > 0 ➔ not safe to preempt kernel
- no outstanding call to preempt && preempt_count == 0 ➔ safe to preempt

Pthreads API
- Mutex Locks
  o pthread_mutex_t first_mutex;
  o pthread_mutex_t second_mutex;
  o pthread_mutex_init (&first_mutex, NULL);
  o pthread_mutex_init (&second_mutex, NULL);
  o pthread_mutex_acquire();
  o pthread_mutex_lock();
  o pthread_mutex_unlock();

- Condition Variables
- Read-Write Locks
- Spinlocks – not all versions may be portable

POSIX SEM
- Semaphores