Cooperating process

- Is one that can affect or be affected by other processes executing in a system
- Directly share a logical address space (both code and data)
  - Achieved through the use of lightweight processes or threads
  - Concurrent access to shared data may result in data inconsistency
- Allowed to share data only through files or messages

Discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space

- Maintains data consistency
Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Consumer-producer problem
  - Solution allows at most BUFFER_SIZE – 1 items in the buffer at the same time
    - Buffer is empty when in == out
    - Buffer is full when ((in+1)%BUFFER_SIZE) == out
  - Suppose that we wanted to provide a solution that fills all the buffers
    - Has an integer count that keeps track of the number of full buffers
    - Initially, count is set to 0
    - Incremented by the producer after it produces a new buffer
    - Decremented by the consumer after it consumes a buffer
while (true) {
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE) {
        ; /* do nothing */
    }
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

while (true)
{
    while (count == 0)
    {
        ; /* do nothing */

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;  
        /* consume the item in nextConsumed */
    }
}
Race Condition

Both producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

- `count++` could be implemented as:
  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- `count--` could be implemented as:
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

Consider this execution interleaving with “count = 5” initially:

- T0: producer execute `register1 = counter`  
  `{register1 = 5}`
- T1: producer execute `register1 = register1 + 1`  
  `{register1 = 6}`
- T2: consumer execute `register2 = counter`  
  `{register2 = 5}`
- T3: consumer execute `register2 = register2 - 1`  
  `{register2 = 4}`
- T4: producer execute `counter = register1`  
  `{count = 6 }`
- T5: consumer execute `counter = register2`  
  `{count = 4}`

Race condition

- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.

Process synchronization and coordination.
Critical-Section Problem

- Consider a system consisting of n processes \{P_0, P_1, \ldots, P_{n-1}\}
- Each process has a segment of code, called a critical section
  - Process may be changing common variables, updating a table, writing a file, …
  - No two processes are executing in their critical sections at the same time
- Critical-section problem
  - Design a protocol that the processes can use to cooperate
  - Entry section
    - Each process must request permission to enter its critical section
    - The section of code implementing this request
  - Critical section
  - Exit section
  - Remainder section
Solution to Critical-Section Problem

1. Mutual Exclusion
   - If process \( P_i \) is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress
   - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting
   - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
     - Assume that each process executes at a nonzero speed.
     - No assumption concerning relative speed of the \( N \) processes.
Solution to Critical-Section Problem

- Two general approaches are used to handle critical sections in OSs
  - **Preemptive kernel**
    - Allows a process to be preemted while it is running in kernel mode
    - Is more suitable for real-time programming
      - Allows a real-time process to preempt a process currently running in the kernel
    - Is more responsive
      - There is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes
  - Are especially difficult to design for SMP architectures
    - It is possible for two kernel-mode processes to run simultaneously on different processors
    - Linux 2.6 kernel, Solaris and IRIX
  - **Nonpreemptive kernel**
    - Does not allow a process running in kernel mode to be preempted
      - A kernel-mode process will run until it exits kernel mode
      - Blocks, voluntarily yields control of the CPU
    - Is essentially free from race conditions on kernel data structures
    - Traditional UNIX kernel, prior to Linux 2.6, Windows XP and Windows 2000
Peterson’s Solution

- Classic software-based solution to the critical-section problem
  - Two process solution
  - Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
  - The two processes share two variables:
    - int turn;
    - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process $P_i$ is ready!
Algorithm for Process $P_i$

do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

    CRITICAL SECTION

    flag[i] = FALSE;

    REMAINDER SECTION

} while (TRUE);
Peterson’s Solution

To prove that Peterson’s Solution is correct

- Mutual exclusion is preserved
  - Pi enters its critical region only if either flag[j] == false or turn == 1
  - If both processes can be executing in their critical sections at the same time
    - Flag [0] == flag [1] == true
  - Pi and Pj could not have successfully executed their while statements at about the same time

- The progress requirement is satisfied
- The bounded-waiting requirement is met

To prove properties 2 and 3

- Process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition
  - flag [j] == true and turn == j
  - This loop is the only one possible
Solution to the Critical-section problem Using Locks

do {
    Acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
Synchronization Hardware

- Many systems provide hardware support for critical section code

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words
TestAndSet Instruction

Definition:

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

- Shared boolean variable lock, initialized to false.
- Solution:
  ```
  do {
      while ( TestAndSet (&lock ))
          ; /* do nothing
  //   critical section

      lock = FALSE;
  //   remainder section
  } while ( TRUE);
  ```
Swap Instruction

Definition:

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

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

  ```
  do {
    key = TRUE;
    while ( key == TRUE)
      Swap (&lock, &key );

    //    critical section
  
    lock = FALSE;

    //      remainder section

  } while ( TRUE);
  ```
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;
} while (TRUE);
Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore $S$ – integer variable
- Two standard operations modify $S$: `wait()` and `signal()`
  - Originally called $P()$ and $V()$
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - `wait (S) {
      while $S <= 0$
      ; // no-op
      $S--$;
    }
  
  - `signal (S) {
      $S++$;
    }`
Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain

- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

- Can implement a counting semaphore $S$ as a binary semaphore

- **Mutual-exclusion implementation with semaphore**
  
  ```
  Semaphore mutex;  // initialized to 1
  do {
      wait (mutex);
      // Critical Section
      signal (mutex);
      // remainder section
  } while (TRUE);
  ```
Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time.
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - \textbf{block} – place the process invoking the operation on the appropriate waiting queue.
  - \textbf{wakeup} – remove one of processes in the waiting queue and place it in the ready queue.
Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:
  ```c
  wait(semaphore *S) {
      S->value--;
      if (S->value < 0) {
          add this process to S->list;
          block();
      }
  }
  ```

- Implementation of signal:
  ```c
  signal(semaphore *S) {
      S->value++;
      if (S->value <= 0) {
          remove a process P from S->list;
          wakeup(P);
      }
  }
  ```
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let $S$ and $Q$ be two semaphores initialized to 1

  
  $\begin{align*}
  P_0 & \quad P_1 \\
  \text{wait} (S); & \quad \text{wait} (Q); \\
  \text{wait} (Q); & \quad \text{wait} (S); \\
  \vdots & \quad \vdots \\
  \vdots & \quad \vdots \\
  \text{signal} (S); & \quad \text{signal} (Q); \\
  \text{signal} (Q); & \quad \text{signal} (S);
  \end{align*}$

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- $N$ buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value $N$. 
The structure of the producer process

```c
do {
    // produce an item
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
} while (true);
```
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```c
   do {
      wait (full);
      wait (mutex);

      // remove an item from buffer
      signal (mutex);
      signal (empty);

      // consume the removed item
   } while (true);
```
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write

- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
  - First readers-writers problem
    - No reader should wait for other readers to finish simply because a writer is waiting
  - Second readers-writers problem
    - If a writer is waiting to access the object, no new readers may start reading

- First readers-writers problem
  - Shared Data
    - Data set
    - Semaphore `mutex` initialized to 1.
    - Semaphore `wrt` initialized to 1.
    - Integer `readcount` initialized to 0.
Readers-Writers Problem

- The structure of a writer process

    do {
        wait (wrt) ;

        // writing is performed

        signal (wrt) ;
    } while (true)
Readers-Writers Problem

- The structure of a reader process

```c
int readcount;

do {
    wait (mutex);
    readcount ++ ;
    if (readercount == 1)  wait (wrt) ;
    signal (mutex)

    // reading is performed

    wait (mutex) ;
    readcount -- ;
    if (readcount == 0)  signal (wrt) ;
    signal (mutex) ;
} while (true)
```
Readers-Writers Problem

- If a writer is in the critical section and n readers are waiting
  - One reader is queued on wrt, and n-1 readers are queued on mutex
- When a writer executes signal (wrt)
  - Resume the execution of either the waiting readers or a single waiting writer
- Readers-Writers problem and its solutions has been generalized to provide reader-writer locks on some systems.
  - Acquiring a reader-writer lock requires specifying the mode of the lock
    - Read access
      - When a process only wishes to read shared data, it requests the read-writer lock in read mode
      - Multiple processes are permitted to concurrently acquire a read-writer lock in read mode
    - Write access
      - A process wishing to modify the shared data must request the read-writer lock in write mode
      - Only one process may acquire the lock for writing
Reader-Writer Locks

- In applications where it is easy to identify which processes only read shared data and which threads only write shared data
- In applications that have more readers than writers
  - Increasing concurrency of allowing multiple readers
Dining-Philosophers Problem

- Consider five philosophers who spend their lives thinking and eating
  - Shares a circular table surrounded by five chairs, each belonging to one philosopher
  - The table is laid with five single chopsticks
- Dining-Philosophers Problem is a classic synchronization problem
  - An example of a large class of concurrency-control problems
  - Allocates several resources among several processes in a deadlock-free and starvation-free manner
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
Dining-Philosophers Problem

- The structure of Philosopher $i$:

```c
Do {
    wait (chopstick[i]);
    wait (chopStick[(i + 1) % 5]);

    // eat
    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);

    // think
}

} while (true);
```
Problems with Semaphores

- Correct use of semaphore operations:
  - `signal (mutex) .... wait (mutex)`
    - Several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement
  - `wait (mutex) ... wait (mutex)`
    - A deadlock will occur
  - Omitting of `wait (mutex)` or `signal (mutex)` (or both)
    - Either mutual exclusion is violated or a deadlock will occur
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```plaintext
monitor monitor name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {......}
    Initialization code ( ....) { ... }
    ...
}
```
Schematic view of a Monitor

- Shared data
- Entry queue
- Operations
- Initialization code
Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) the process invoked x.wait ()
Monitor with Condition Variables

- Entry queue
- Shared data
- Queues associated with x, y conditions
- Operations
- Initialization code
Solution to Dining Philosophers

monitor DP
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

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

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
Solution to Dining Philosophers

```c
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;
}
```
Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
  - If a lock will be held for less than a few hundred instructions
- Uses condition variables and semaphores when longer sections of code need access to data
  - If the desired lock is already held, the thread issues a wait and sleeps
  - When a thread frees the lock, it issues a signal to the next sleeping thread in the queue
- Readers-writers locks are used to protect data that are accessed frequently but are usually accessed in a read-only manner
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
Windows XP Synchronization

- A multithreaded kernel provides support for real-time applications and multiple processors
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
    - Protects short code segments
    - A thread will never be preempted while holding a spinlock
- Also provides dispatcher objects for thread synchronization
  - Includes mutexes, semaphores, events, and timers
    - An event acts much like a condition variable, they may notify a waiting thread when a desired condition occurs
    - Timers are used to notify one thread that a specified amount of time has expired
  - Signaled state
    - Indicates that an object is available and a thread will not block when acquiring the object
  - Nonsignal state
    - Indicates that an object is not available and a thread will block when attempting to acquire the object
Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections

- Linux provides:
  - semaphores
  - spin locks
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks
End of Chapter 6