Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Bounded Buffer problem (also called producer consumer problem)
**Bounded-Buffer**

- **Shared data**

  ```
  #define BUFFER_SIZE 10
  typedef struct {
      ... 
  } item;
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  int counter = 0;
  ```

- **Producer process**

  ```
  item nextProduced;
  
  while (1) {
      while (counter == BUFFER_SIZE)
      ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
  }
  ```
Bounded-Buffer

- Consumer process
  ```
  int nextConsumed;
  
  while (1) {
    while (counter == 0) {
      ; /* do nothing */
    }
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
  }
  ```

Bounded Buffer

- The statements
  ```
  counter++;
  counter--;
  ```
  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.
Bounded Buffer

- The statement “count++” may be implemented in machine language as:

  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- The statement “count--” may be implemented as:

  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.
Bounded Buffer

- Assume counter is initially 5. One interleaving of statements is:

  - producer: register1 = counter (register1 = 5)
  - producer: register1 = register1 + 1 (register1 = 6)
  - consumer: register2 = counter (register2 = 5)
  - consumer: register2 = register2 − 1 (register2 = 4)
  - producer: counter = register1 (counter = 6)
  - consumer: counter = register2 (counter = 4)

- The value of count may be either 4 or 6, where the correct result should be 5.

Race Condition

- Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be synchronized.
The Critical-Section Problem

- \( n \) processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Mutual Exclusion: Conditions for Solution

Four conditions to provide mutual exclusion
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region
Solutions to the Problem

- General structure of process $P_i$
  
  ```
  do {
    entry section
    critical section
    exit section
    reminder section
  } while (1);
  ```

- Processes may share some common variables to synchronize their actions.

Synchronization Hardware

- Test and modify the content of a word atomically

```java
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
}
```
Mutual Exclusion with Test-and-Set

- Shared data:
  
  ```java
  boolean lock = false;
  ```

- Process $P_i$
  
  ```java
  do {
      while (TestAndSet(lock)) ;
      critical section
      lock = false;
      remainder section
  }
  ```

Semaphores

- The solution we have looked at (TSL instruction) involves **busy waiting**
  - Potential waste of CPU cycles
- Semaphores are synchronization mechanism that does not require busy waiting.
  - Uses **blocking synchronization**
- can only be accessed via two indivisible (atomic) operations: wait() and signal()
- Each semaphore has an integer value and a queue associated with it
Semaphore Implementation

- Define a semaphore as a record
  
  ```c
  typedef struct {
    int value;
    struct process *L;
  } semaphore;
  ```

- Assume two simple operations:
  - `block` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process P.

Implementation

- Semaphore operations defined as

  ```c
  wait(S):
  S.value--;
  if (S.value < 0) {
    add this process to S.L;
    block;
  }

  signal(S):
  S.value++;
  if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
  }
  ```
Critical Section of $n$ Processes

- Shared data:
  ```
  semaphore mutex;  // initially mutex = 1
  ```

- Process $P_i$:
  ```
  do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
  } while (1);
  ```

Implementation cont’d

- Critical aspect of semaphore implementation is that the wait() and signal() operations must be executed atomically
  - need to guarantee that no two processes can execute wait() or signal() at the same time
  - Wait() and signal() have to be executed as critical sections!!

- Uniprocessors – disable interrupts while executing wait() and signal()

- Multiprocessors – disabling interrupts will not work because there are multiple processors
  - Most current CPUs have hardware support available (TSL), use for implementing critical section
Semaphore as a General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore flag initialized to 0
- Code:

```
$P_i$

code

$A$

wait(flag)

signal(flag)  

$P_j$

code

$B$
```

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

```
$P_0$  

wait($S$);  wait($Q$);

wait($Q$);  wait($S$);

.:  

signal($S$);  signal($Q$);

signal($Q$)  signal($S$);

$P_1$
```

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

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

Bounded-Buffer Problem

- Shared data
  
  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1
Bounded-Buffer Problem Producer Process

do {
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
} while (1);

Bounded-Buffer Problem Consumer Process

do {
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while (1);
Readers-Writers Problem

- Shared data

```c
semaophore mutex, wrt;
```

Initially

```c
mutex = 1, wrt = 1, readcount = 0
```

Readers-Writers Problem Writer Process

```c
wait(wrt);
...
writing is performed
...
signal(wrt);
```
Readers-Writers Problem Reader Process

```c
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex);
```

Dining-Philosophers Problem

- Shared data
  ```c
  semaphore chopstick[5];
  Initially all values are 1
  ```
Dining-Philosophers Problem: A non-solution

Philosopher $i$:

```c
    do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
        ...  
        eat
        ...  
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        ...  
        think
        ...  
    } while (1);
```

---

High-level synchronization mechanisms

- Semaphores are a very powerful mechanism for process synchronization, but they are a low-level mechanism
- Several high-level mechanisms that are easier to use have been proposed
  - Monitors
  - Critical Regions
  - Read/Write Locks
- We will study monitors (Java and Pthreads provide synchronization mechanisms based on monitors)
- **NOTE**: high-level mechanisms easier to use but equivalent to semaphores in power
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitored moni
or-name
{
    shared variable declarations
    procedure body P1(…) {
        ...
    }
    procedure body Pn(…) {
        ...
    }
    { initialization code }
}
```

To allow a process to wait within the monitor, a condition variable must be declared, as

```
condition x, y;
```

Condition variable can only be used with the operations `wait` and `signal`.

- The operation `x.wait();` means that the process invoking this operation is suspended until another process invokes `x.signal();`
- The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.
Schematic View of a Monitor

Monitor With Condition Variables
Producer-Consumer using monitors

```plaintext
monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert(item: integer);
begin
  if count = N then wait(full);
  insert_item(item);
  count := count + 1;
  if count = 1 then signal(empty)
end;
function remove: integer;
begin
  if count = 0 then wait(empty);
  remove = remove_item;
  count := count - 1;
  if count = N - 1 then signal(full)
end:
count := 0;
end monitor;
```

Dining Philosophers Example

```plaintext
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)  // following slides
  void putdown(int i)  // following slides
  void test(int i)     // following slides

  void init() {
    for (int i = 0; i < 5; i++)
      state[i] = thinking;
  }
}
```
Dining Philosophers

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);
}

Dining Philosophers

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

- Pthreads
  - Semaphores
  - Mutex locks
  - Condition Variables
  - Reader/Writer Locks

- Java
  - Each object has an (implicitly) associated lock and condition variable

Java thread synchronization calls

thread.join(int millisecs)
Blocks the calling thread for up to the specified time until thread has terminated.

thread.interrupt()
Interrupts thread: causes it to return from a blocking method call such as sleep().

object.wait(long millisecs, int nanosecs)
Blocks the calling thread until a call made to notify() or notifyAll() on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called wait() on object.
Mutual exclusion in Java

class Interfere {
    private int data = 0;
    public synchronized void update() {
        data++;
    }
}

class Interfere {
    private int data = 0;
    public void update() {
        synchronized(this) {
            data++;
        }
    }
}

Producer consumer using Java

public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) {
                item = produce_item(); // producer loop
                mon.insert(item);
            }
        }
    }
    private int produce_item() { ... } // actually produce
    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) {
                item = mon.remove(); // consumer loop
                consume_item(item);
            }
        }
    }
    private void consume_item(int item) { ... } // actually consume
}
Producer consumer using Java cont’d

```java
static class our_monitor {
    // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N; // slot to place next item in
        count = count + 1; // one more item in the buffer now
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer[lo]; // fetch an item from the buffer
        lo = (lo + 1) % N; // slot to fetch next item from
        count = count - 1; // one few items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }

    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {}}
}
```

Pthreads Synchronization Mechanisms

- Mutex Locks
- Condition Variables
- Semaphores
- Read Write Locks
Mutex Locks

- Mutual Exclusion Locks
- Example:

```c
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
int count;

increment_count()
{
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

get_count()
{
  int c;
  pthread_mutex_lock(&count_mutex);
  c = count;
  pthread_mutex_unlock(&count_mutex);
  return(c);
}
```

Condition Variables

- based on monitor condition variables
- Easier to understand and use than semaphores
- cond_wait and cond_signal operations
  - different semantics from semaphore wait and signal operations
- always use in conjunction with a mutex lock
Producer-Consumer using condition variables

char buf[BSIZE];
int occupied;
int nextin;
int nextout;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

producer(char item)
{
    pthread_mutex_lock(&mutex);
    while (occupied == BSIZE)
        pthread_cond_wait(&not_full, &mutex);

    /* insert item */
    buf[next_in++] = item;
    next_in = next_in % BSIZE;
    occupied++;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}

c consumer()
{
    pthread_mutex_lock(&mutex);
    while (occupied == 0)
        pthread_cond_wait(&not_empty, &mutex);

    /* consume item */
    item = buf[next_out++];
    next_out = next_out % BSIZE;
    occupied--;
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);
}