Synchronization
The Issue...

- 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 Revisited

- Previous, unsynchronized, solution always left one entry empty.
  - Only the producer could access that entry, so no need for synchronization.

- What if we want to use every entry?

- Introduce a count variable that tracks the number of full entries.
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--;
    }
}
Race Condition

• In hardware:
  - `count++` could be implemented as:
    
    ```
    register1 = count
    register1 = register1 + 1
    count = register1
    ```
  - `count--` could be implemented as:
    
    ```
    register2 = count
    register2 = register2 - 1
    count = register2
    ```
Race Condition

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

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6}
  S5: consumer execute count = register2 {count = 4}
Critical Sections

- A region of code in which a process could access or change shared data.
- The critical section problem:
  - Design a protocol that allows a set of processes to cooperate safely.
- Structure of cooperating code:

```c
{
ENTRY SECTION       //Request access to critical section.
CRITICAL SECTION
EXIT SECTION        //Protocol code to exit critical section.
REMAINDER SECTION  //Do the other work this process does.
}
```
Solution Requirements

- **Mutual exclusion** – At most one process in a critical section.
- **Progress** – No process in a remainder section can block another process.
- **Bounded Waiting** – It should not be possible for a process to wait forever in its entry section.
  - I.e. there is a limit on the number of times other processes can enter their critical sections after a process has requested access to its own.
  - Assume that all processes execute at non-zero speed.
  - No assumption about relative speed of processes.
Foolproof Solution

- On a single processor system just turn off interrupts before entering critical section.
- Turn them back on when exiting.
- This approach is not acceptable for user level code.
  - Why not?
- It is often used for kernel code.
Naive Proposal: Lock Variable

while (lock == 1) {;}  
lock = 1;  
CRITICAL SECTION  
lock = 0;

• Does it work?

(Note, some of this discussion is based on Tenenbaum's Modern Operating Systems, 2001)
Another Naive Proposal: Strict Alteration

- **Process 0**
  ```
  while (turn != 1) {;;}
  CRITICAL SECTION
  turn = 1;
  REMAINDER SECTION
  ```

- **Process 1**
  ```
  while (turn != 0) {;;}
  CRITICAL SECTION
  turn = 0;
  REMAINDER SECTION
  ```

- **turn** is initialized to 0.
- Guarantees mutual exclusion!
- Any problems?
Another Naive Proposal: Strict Alteration

- Any problems?
  - Imagine the situation...
  - Both processes are in their remainder section.
  - It is 0's turn.
  - 1 exits its remainder section, and is ready to run.
  - Violates the progress requirement.

```
Process 0
while (turn == 1) {;}
CRITICAL SECTION
turn = 1;
REMAINDER SECTION

Process 1
while (turn == 0) {;}
CRITICAL SECTION
turn = 0;
REMAINDER SECTION
```
Peterson's Solution (1981)

- Two shared data items:
  
  ```java
  int turn;  //Who's turn is it?
  boolean flag[2];  //Readiness state for each process.
  ```
  
  ```java
  {
    flag[i] = true;
    turn = j;
    while (flag[j] == true && turn == j){;}
  }

  CRITICAL SECTION

  flag[i] = false;

  REMAINDER SECTION
  ```
Synchronization Hardware

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

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

void Swap (bool *a, bool *b) {
    bool tmp = *a;
    *a = *b;
    *b = tmp;
}
```
Test and Set Mutual Exclusion

- `lock` is initialized to false.
- Guarantees mutual exclusion.
- Where does it fail?

```c
{ 
    while ( TestAndSet (&lock ) ) {;}
    CRITICAL SECTION
    lock = FALSE
    REMAINDER SECTION
}
```
Test and Set Mutual Exclusion

• Where does it fail?
  – Bounded waiting.
• The book presents a solution that meets all three conditions.
• It's complicated.

```c
{
    while ( TestAndSet (&lock ) ) {;}
    CRITICAL SECTION
    lock = FALSE
    REMAINDER SECTION
}
```
Busy Waiting

• All of the solutions so far have included tight while loops.
  – Busy waiting, spin locks.
  – Why is that a problem?

• The solutions so far have also been complicated for the programmer.

• What is the solution?
Semaphores!

- A semaphore is a counter.
- Two operations can be applied:
  - `wait(S)` – if S is less than or equal to zero, wait.
    - When done waiting decrement S.
  - Otherwise, just keep going.
    - Immediately decrement S.
  - `signal(S)` – Increment S. Give waiting processes a chance to wake up.
- Invented by Dijkstra in '65.
- Wait was called P, signal was called V.
Semaphore Applications: Mutual Exclusion

- Let at most three processes enter a critical region:
  - Initialize $s = 3$.
  ```c
  { 
    wait(S)
    CRITICAL SECTION
    signal(S)
  }
  ```

- A **mutex** is a binary semaphore.
  - Sufficient for mutual exclusion.
Semaphore Applications: Guaranteed Ordering

- Make sure that $S_1$ takes place before $S_2$:
  - Initialize $SEM = 0$

```plaintext
{  
  $S_1$
  signal(SEM)
}

{  
  wait(SEM)
  $S_2$
}
```
Semaphore Implementation

A semaphore may be implemented as a counter and a list of waiting processes:

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

wait (semaphore* S){
    S->value--;
    if (S->value < 0) {
        //add this process to S->list
        block();
    }
}

signal (semaphore* S){
    S->value++;
    if (S->value <= 0) {
        //remove a process P from to S->list
        wakeup(P);
    }
}
```

• Is this implementation OK?
Semaphore Implementation

- No. Modifying and checking the semaphore data needs to be atomic.
- Two possibilities:
  - Turn off interrupts.
  - Spin locks.
- Is there anything special we need to do to ensure bounded waiting?
Semaphore Deadlock

- **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 & \text{wait}(S) \\
& \text{wait}(Q) \\
& ... \\
& ... \\
& \text{signal}(S) \\
& \text{signal}(Q) \\
\end{align*}
```

```
\begin{align*}
P_1 & \text{wait}(Q) \\
& \text{wait}(S) \\
& ... \\
& ... \\
& \text{signal}(Q) \\
& \text{signal}(S) \\
\end{align*}
```
Producer Consumer Solution Using Semaphores

- N entries, 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.
Producer Consumer Solution
Using Semaphores

while (true) {
  // produce an item
  wait (empty);
  wait (mutex);
  // add item to the buffer
  signal (mutex);
  signal (full);
}

while (true) {
  wait (full);
  wait (mutex);
  // remove item from the buffer
  signal (mutex);
  signal (empty);
  // consume the item
}

- Why not just use the mutex?
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 writer at a time.
Readers-Writers Solution

• Shared Data
  – Data set
  – Semaphore `mutex` initialized to 1.
  – Semaphore `wrt` initialized to 1.
  – Integer `readcount` initialized to 0.
Readers-Writers Solution

**Writer**

```java
while (true) {
    wait (wrt);
    // writing is performed
    signal (wrt);
}
```

**Reader**

```java
while (true) {
    wait (mutex);
    readcount++;
    if (readcount == 1) wait (wrt);
    signal (mutex)
    // reading is performed
    wait (mutex);
    readcount--;
    if (readcount == 0) signal (wrt);
    signal (mutex);
}
```

- Any possibility of starvation?
  - (Violations of bounded waiting.)
Readers-Writers Solution

• This is a solution to the *first* readers-writers problem.
  – No reader will be kept waiting unless a writer has already been granted access.

• The *second* readers-writers problem requires that no new readers may be granted access if a writer is waiting.

• Some system provide built in reader-writer locks.
  – Parameter determines what sort of lock is obtained.
  – Linux kernel has these.
Dining Philosophers Problem

- Five philosophers, five chopsticks
- Each philosopher:
  - Picks up one chopstick.
  - Picks up another.
  - Eats a while.
  - Puts down a chopstick.
  - Puts down the other.
  - Thinks for a while.
  - Repeats.

- The problem: how to avoid starvation and deadlock.
Naive Solution

- One semaphore for each chopstick.

```java
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    //EAT
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    //THINK
}
```

- What is the problem?
- Any better ideas?
General Problems With Semaphores

- Coding with semaphores is error prone.
  - `signal (mutex) ... wait (mutex)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both)

- The errors are extremely hard to catch.

- Not easily reproducible.
Monitors

- A programming language construct designed to make synchronization easier.
- Implemented in C#, Java.
- Only one process at a time can be executing monitor code.
- Only monitor code can access local monitor data.
- Very object orientedish.
Monitors

“Syntax”

```plaintext
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (...) { ... }
   procedure Pn (...) { ... }
   initializationCode(...) {...}
}
```

Schematic
A Problem

- Monitors are easy to use, but there is a problem...
- Imagine that in the producer consumer problem, we have a monitor procedure for add.
- What does the procedure do when it finds that the buffer is full?
Condition Variables

- 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) that invoked `x.wait()`.
- Something like a non-counting semaphore.
Producer Consumer With Monitors

```
monitor producer-consumer {
    condition empty, full;
    int count=0;

    void insert(item) {
        if (count == MAX) wait(empty);
        insert_item(item);
        count = count+1;
        signal(full);
    }

    int remove() {
        int ret_val;
        if (count == 0) wait(full);
        ret_val = remove_item();
        count = count-1
        signal(empty);
        return ret_val;
    }
}
```
What Exactly Should Wait/Signal Do?

- Only one process can be in a monitor at a time.
- When a signal occurs and a process is waiting, who gets to stay?
- Simplest solution is to require a signal to be the final instruction in a procedure.
Dining Philosophers Solution

```c
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);
    }
}
```
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;
}
}
Dining Philosophers Solution

- Philosopher i does the following:
  
  ```
  DP.pickup(i);
  EAT
  DP.putdown(i);
  ```
Java Monitors

- Any method can be declared “synchronized”.
- Every object has a single lock.
- Every call to a synchronized method must wait for the lock.
Synchronization Examples

• The book talks about the synchronization mechanisms provided by:
  – Solaris
  – Windows XP
  – Linux
  – pthreads

• Check it out.
Atomic Transactions

• Assures that operations happen as a single logical unit of work, in its entirety, or not at all.
• Related to field of database systems.
• Challenge is assuring atomicity despite computer system failures.
• We might take a closer look at this material when we get to file systems...
Atomic Transactions

- **Transaction** - collection of instructions or operations that performs single logical function.
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of **read** and **write** operations
  - Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation.
  - Aborted transaction must be **rolled back** to undo any changes it performed.
Acknowledgments

• Portions of these slides are taken from PowerPoint presentations made available along with:

• Original versions of those presentations can be found at: