Process Synchronization – Outline

- Why do processes need synchronization?
- What is the critical-section problem?
- Describe solutions to the critical-section problem
  - Peterson’s solution
  - using synchronization hardware
  - semaphores
  - monitors
- Classic Problems of Synchronization
- What are atomic transactions?
Why Process Synchronization?

- Processes may cooperate with each other
  - producer-consumer and service-oriented system models
  - exploit concurrent execution on multiprocessors

- Cooperating processes may share data (globals, files, etc)
  - imperative to maintain data correctness

- Why is data correctness in danger?
  - process run asynchronously, context switches can happen at any time
  - processes may run concurrently
  - different orders of updating shared data may produce different values

- Process synchronization
  - to coordinate updates to shared data
  - order of process execution should not affect shared data

Only needed when processes share data!
## Producer-Consumer Data Sharing

### Producer

```java
while (true) {
    /* wait if buffer full */
    while (counter == 10)  ; /* do nothing */

    /* produce data */
    buffer[in] = sdata;
    in = (in + 1) % 10;

    /* update number of items in buffer */
    counter++;
}
```

### Consumer

```java
while (true) {
    /* wait if buffer empty */
    while (counter == 0)  ; /* do nothing */

    /* consume data */
    sdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of items in buffer */
    counter--;
}
```
Producer-Consumer Data Sharing

**Producer**

```java
while (true) {

    /* wait if buffer full */
    while (counter == 10)  
        ; /* do nothing */

    /* produce data */
    buffer[in] = sdata;
    in = (in + 1) % 10;

    /* update number of items in buffer */
    R1 = load (counter);
    R1 = R1 + 1;
    counter = store (R1);
}
```

**Consumer**

```java
while (true) {

    /* wait if buffer empty */
    while (counter == 0)  
        ; /* do nothing */

    /* consume data */
    sdata = buffer[out];
    out = (out + 1) % 10;

    /* update number of items in buffer */
    R2 = load (counter);
    R2 = R2 - 1;
    counter = store (R2);
}
```
Race Condition

Suppose $counter = 5$

Incorrect Sequence 1

R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 - 1;
counter = store (R1);
counter = store (R2);

Final Value in counter = 4!

Incorrect Sequence 2

R1 = load (counter);
R1 = R1 + 1;
R2 = load (counter);
R2 = R2 - 1;
counter = store (R2);
counter = store (R1);

Final Value in counter = 6!

Race condition is a situation where

- several processes concurrently manipulate shared data, and
- shared data value depends on the order of execution
Critical Section Problem

Region of code in a process *updating* shared data is called a critical region.

Concurrent updating of shared data by multiple processes is dangerous.

Critical section problem
- how to ensure synchronization between cooperating processes?

Solution to the critical section problem
- only allow a single process to enter its critical section at a time

Protocol for solving the critical section problem
- request permission to enter critical section
- indicate after exit from critical section
- only permit a single process at a time
Solution to the Critical Section Problem

- Formally states, each solution should ensure
  - *mutual exclusion*: only a single process can execute in *its* critical section at a time
  - *progress*: selection of a process to enter its critical section should be fair, and the decision cannot be postponed indefinitely.
  - *bounded waiting*: there should be a fixed bound on how long it takes for the system to grant a process's request to enter its critical section

- Other than satisfying these requirements, the system should also guard against *deadlocks*. 
Preemptive Vs. Non-preemptive Kernels

Several kernel processes share data
- structures for maintaining file systems, memory allocation, interrupt handling, etc.

How to ensure OSes are free from race conditions?

Non–preemptive kernels
- process executing in kernel mode cannot be preempted
- disable interrupts when process is in kernel mode
- what about multiprocessor systems?

Preemptive kernels
- process executing in kernel mode can be preempted
- suitable for real-time programming
- more responsive
Peterson’s Solution to Critical Section Problem

- Software based solution
- Only supports two processes
- The two processes share two variables:
  - int turn;
    - indicates whose turn it is to enter the critical section
  - boolean flag[2]
    - indicates if a process is ready to enter its critical section
Peterson's Solution

**Process 0**
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn==1) ;
    // critical section

    flag[0] = FALSE;

    // remainder section
} while (TRUE)

**Process 1**
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn==0) ;
    // critical section

    flag[1] = FALSE;

    // remainder section
} while (TRUE)

Solution meets all three requirements

- P0 and P1 can never be in the critical section at the same time
- if P0 does not want to enter critical region, P1 does no waiting
- process waits for at most one turn of the other to progress
Peterson's Solution – Notes

- Only supports two processes
  - generalizing for more than two processes has been achieved
- Assumes that the LOAD and STORE instructions are atomic
- Assumes that memory accesses are not reordered
- May be less efficient than a hardware approach
  - particularly for >2 processes
Lock-Based Solutions

General solution to the critical section problem

- critical sections are protected by locks
- process must acquire lock before entry
- process releases lock on exit

\[
\text{do } \{ \\
\quad \text{acquire lock;} \\
\quad \text{critical section} \\
\quad \text{release lock;} \\
\quad \text{remainder section} \\
\} \text{ while(TRUE);} 
\]
Hardware Support for Lock-Based Solutions – Uniprocessors

- For uniprocessor systems
  - concurrent processes cannot be overlapped, only \textit{interleaved}
  - process runs until it invokes system call, or is \textit{interrupted}

- Disable interrupts!
  - active process will run without preemption
    
    \begin{verbatim}
    do {
      disable interrupts;
      critical section
      enable interrupts;
    } while(TRUE);
    \end{verbatim}
Hardware Support for Lock-Based Solutions – Multiprocessors

- In multiprocessors
  - several processes share memory
  - processors behave independently in a peer manner

- Disabling interrupt based solution will not work
  - too inefficient
  - OS using this not broadly scalable

- Provide hardware support in the form of atomic instructions
  - atomic test-and-set instruction
  - atomic swap instruction
  - atomic compare-and-swap instruction

- Atomic execution of a set of instructions means that instructions are treated as a single step that cannot be interrupted.
TestAndSet Instruction

Pseudo code definition of TestAndSet

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

```c
int mutex;
init_lock (&mutex);

do {
    lock (&mutex);
    *critical section*
    unlock (&mutex);
    *remainder section*
} while(TRUE);

void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    while(TestAndSet(mutex))
    ;
}

void unlock (int *mutex) {
    *mutex = 0;
}
```
Swap Instruction

Psuedo code definition of swap instruction

```c
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Mutual Exclusion using \textit{Swap}

```c
int mutex;
init_lock (&mutex);

do {
    lock (&mutex);
    \textit{critical section}
    unlock (&mutex);
    \textit{remainder section}
} while(TRUE);

void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    int key = TRUE;
    do {
        Swap(&key, mutex);
    }while(key == TRUE);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```

\textit{Fairness not guaranteed by any implementation !}
Bounded Waiting Solution

Process $i = 0$

\[
\text{do}\{
\text{waiting}[i] = \text{TRUE};
\text{key} = \text{TRUE};
\text{while}(\text{waiting}[i] \&\& \text{key})
\quad \text{key} = \text{TestAndSet}(&\text{lock});
\quad \text{waiting}[i] = \text{FALSE};

\quad \text{// Critical Section}

\quad j = (i + 1) \mod n;
\quad \text{while} ((j \neq i) \&\& !\text{waiting}[j])
\quad \quad j = (j+1) \mod n;

\quad \text{if} (j == i)
\quad \quad \text{lock} = \text{FALSE};
\quad \text{else}
\quad \quad \text{waiting}[j] = \text{FALSE};
\quad \text{// Remainder Section}
\} \text{ while (TRUE)};
\]

Process $i = 1$

\[
\text{do}\{
\text{waiting}[i] = \text{TRUE};
\text{key} = \text{TRUE};
\text{while}(\text{waiting}[i] \&\& \text{key})
\quad \text{key} = \text{TestAndSet}(&\text{lock});
\quad \text{waiting}[i] = \text{FALSE};

\quad \text{// Critical Section}

\quad j = (i + 1) \mod n;
\quad \text{while} ((j \neq i) \&\& !\text{waiting}[j])
\quad \quad j = (j+1) \mod n;

\quad \text{if} (j == i)
\quad \quad \text{lock} = \text{FALSE};
\quad \text{else}
\quad \quad \text{waiting}[j] = \text{FALSE};
\quad \text{// Remainder Section}
\} \text{ while (TRUE)};
\]

Cycle = 0
lock=FALSE, key=FALSE, waiting[0]=0, waiting[1]=0
Bounded Waiting Solution

Process $i = 0$

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);

Process $i = 1$

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);

Cycle $= 1$
llock=FALSE, key=FALSE, waiting[0]=1, waiting[1]=1
Bounded Waiting Solution

**Process i = 0**

```c
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);
```

**Process i = 1**

```c
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);
```

**Cycle = 2**

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1
Bounded Waiting Solution

**Process i = 0**

```c
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);
```

**Process i = 1**

```c
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);
```

**Cycle = 3**

lock=FALSE, key=TRUE, waiting[0]=1, waiting[1]=1
**Bounded Waiting Solution**

**Process i = 0**

```c
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);
```

**Process i = 1**

```c
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);
```

**Process 0 wins the race**

**Cycle = 4**

lock=TRUE, key=FALSE, waiting[0]=1, waiting[1]=1
Bounded Waiting Solution

Process \( i = 0 \)

do{
  \text{waiting}[i] = TRUE; \\
  \text{key} = TRUE; \\
  \text{while}(\text{waiting}[i] && \text{key}) \\
    \text{key} = \text{TestAndSet}(&\text{lock}); \\
  \text{waiting}[i] = FALSE;
  \\
  // Critical Section \\
  j = (i + 1) \% n; \\
  \text{while }((j != i) && \neg \text{waiting}[j]) \\
    j = (j+1) \% n;
  \\
  \text{if } (j == i) \\
    \text{lock} = \text{FALSE}; \\
  \text{else} \\
    \text{waiting}[j] = \text{FALSE}; \\
  // Remainder Section \\
} \text{while } (\text{TRUE});

Process \( i = 1 \)

do{
  \text{waiting}[i] = TRUE; \\
  \text{key} = TRUE; \\
  \text{while}(\text{waiting}[i] && \text{key}) \\
    \text{key} = \text{TestAndSet}(&\text{lock}); \\
  \text{waiting}[i] = FALSE;
  \\
  // Critical Section \\
  j = (i + 1) \% n; \\
  \text{while }((j != i) && \neg \text{waiting}[j]) \\
    j = (j+1) \% n;
  \\
  \text{if } (j == i) \\
    \text{lock} = \text{FALSE}; \\
  \text{else} \\
    \text{waiting}[j] = \text{FALSE}; \\
  // Remainder Section \\
} \text{while } (\text{TRUE});

\textbf{Cycle} = 5

\text{lock}=\text{TRUE}, \text{key}=\text{TRUE}, \text{waiting}[0]=0, \text{waiting}[1]=1
Bounded Waiting Solution

**Process i = 0**

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);

**Process i = 1**

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);

Cycle = 6
lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1
Bounded Waiting Solution

**Process i = 0**

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);

**Process i = 1**

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);

**Cycle = 7**

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1
Bounded Waiting Solution

Process $i = 0$

\[
\text{do}\{
\text{waiting}[i] = \text{TRUE};
\text{key} = \text{TRUE};
\text{while}(\text{waiting}[i] \land \text{key})
    \text{key} = \text{TestAndSet}(&\text{lock});
\text{waiting}[i] = \text{FALSE};
\}
\]

// Critical Section

j = (i + 1) \% n;
while ((j != i) \land \text{!waiting}[j])
    j = (j+1) \% n;

if (j == i )
    \text{lock} = \text{FALSE};
else
    \text{waiting}[j] = \text{FALSE};
// Remainder Section

} \text{ while (TRUE);}

Process $i = 1$

\[
\text{do}\{
\text{waiting}[i] = \text{TRUE};
\text{key} = \text{TRUE};
\text{while}(\text{waiting}[i] \land \text{key})
    \text{key} = \text{TestAndSet}(&\text{lock});
\text{waiting}[i] = \text{FALSE};
\}
\]

// Critical Section

j = (i + 1) \% n;
while ((j != i) \land \text{!waiting}[j])
    j = (j+1) \% n;

if (j == i )
    \text{lock} = \text{FALSE};
else
    \text{waiting}[j] = \text{FALSE};
// Remainder Section

} \text{ while (TRUE);}

Cycle = 8
lock=\text{TRUE}, key=\text{TRUE}, \text{waiting}[0]=0, \text{waiting}[1]=1
Bounded Waiting Solution

**Process i = 0**

```c
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);
```

**Process i = 1**

```c
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);
```

**Cycle = 9**

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=1
Bounded Waiting Solution

**Process i = 0**

```c
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);
```

**Process i = 1**

```c
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);
```

**Cycle = 10**

lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0
Bounded Waiting Solution

**Process i = 0**

```plaintext
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);
```

**Process i = 1**

```plaintext
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);
```

**Cycle = 11**
lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0
Bounded Waiting Solution

**Process i = 0**

```
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);
```

**Process i = 1**

```
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);
```

Cycle = 12
lock=TRUE, key=TRUE, waiting[0]=0, waiting[1]=0
Semaphores

Another solution to the critical section problem

• higher-level than using direct ISA instructions
• similar to locks, but semantics are different

Semaphore (simple definition)

• is an integer variable
• only accessed via `init()`, `wait()`, and `signal()` operations
• all semaphore operations are atomic

Binary semaphores

• value of semaphore can either be 0 or 1
• used for providing mutual exclusion

Counting semaphore

• can have any integer value
• access control to some finite resource
Mutual Exclusion Using Semaphores

```c
int S;
sem_init (&S);

do {
    wait (&S);
    // critical section
    signal (&S);

    // remainder section
}
while(TRUE);

void sem_init (int *S) {
    *S = 0;
}

void wait (int *S) {
    while (*S <= 0)
        ;
    *S-- ;
}

void signal (int *S) {
    *S++;
}
```
Problem With All Earlier Solutions?

- Busy waiting or spinlocks
  - process may loop *continuously* in the entry code to the critical section

- Disadvantage of busy waiting
  - waiting process holds on to the CPU during its time-slice
  - does no useful work
  - does not let any other process do useful work

- Multiprocessors still do use busy-waiting solutions.
Semaphore with no Busy waiting

- Associate waiting queue with each semaphore
- Semaphore (no busy waiting definition)
  - integer value
  - waiting queue

```c
typedef struct {
    int value;
    struct process *list;
} semaphore;
```
Operations on Semaphore with no Busy waiting (2)

• Wait ( ) operation

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

`block ( ) suspends the process that invokes it.`

• Signal ( ) operation

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

`wakeup ( ) resumes execution of the blocked process P.`
Atomic Implementation of Semaphore Operations

- Guarantee that \textit{wait} and \textit{signal} operations are atomic
  - critical section problem again?
  - how to ensure atomicity of \textit{wait} and \textit{signal}? \\
- Ensuring atomicity of \textit{wait} and \textit{signal}
  - implement semaphore operations using hardware solutions
  - uniprocessors – enable/disable interrupts
  - multiprocessors – using spinlocks around \textit{wait} and \textit{signal} \\
- Did we really solve the busy-waiting problem
  - NO!
  - but we shifted its location, only busy-wait around \textit{wait} and \textit{signal}
  - \textit{wait} and \textit{signal} are small routines
Deadlock

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

Example: S and Q be two semaphores initialized to 1

```
P0
  wait (S);
  wait (Q);
  .
  .
  .
  signal (S);
  signal (Q);

P1
  wait (Q);
  wait (S);
  .
  .
  .
  signal (Q);
  signal (S);
```
Starvation and Priority Inversion

- **Indefinite blocking or starvation**
  - process is not deadlocked
  - but is never removed from the semaphore queue

- **Priority inversion**
  - lower-priority process holds a lock needed by higher-priority process!
  - assume three processes L, M, and H
  - priorities in the order L < M < H
  - L holds shared resource R, needed by H
  - M preempts L, H needs to wait for both L and M!!

- solutions
  - only support at most two priorities
  - *priority inheritance protocol* – lower priority process accessing shared resource inherits higher priority
Problem Solving Using Semaphores

- Bounded-buffer problem
- Readers-Writers problem
Bounded-Buffer Problem

Problem synopsis

- A set of resource buffers shared by producer and consumer threads
  - Buffers are shared between producer and consumer
- Producer inserts resources into the buffers
  - Output, disk blocks, memory pages, processes, etc.
- Consumer removes resources from the buffer set
  - Whatever is generated by the producer
- Producer and consumer execute asynchronously
  - No serialization of one behind the other
  - CPU scheduler determines what runs when

Ensure data (buffer) consistency

- Consumer should see each produced item at least once
- Consumer should see each produced item at most once
Bounded Buffer Problem (2)

- Solution employs three semaphores
  - **mutex**
    - allow exclusive access to the buffer pools
    - mutex semaphore, initialized to 1
  - **empty**
    - count number of empty buffers
    - counting semaphore, initialized to n (the total number of available buffers)
  - **full**
    - count number of full buffers
    - counting semaphore, initialized to 0
Bounded Buffer Problem (3)

Semaphore bool mutex;
Semaphore int full, empty;

**Producer**

```c
do {
    Produce new resource
    wait (empty);
    wait (mutex);
    Add resource to next buffer
    signal (mutex);
    signal (full);
} while (TRUE);
```

**Consumer**

```c
do {
    wait (full);
    wait (mutex);
    Remove resource from buffer
    signal (mutex);
    signal (empty);
    Consume resource
    } while (TRUE);
```
Readers – Writers Problem

Problem synopsis

- an object shared among several threads
- some threads only read the object (Readers)
- some threads only write the object (Writers)

Problem is to ensure data consistency

- multiple readers can access the shared resource simultaneously
- only one writer should update the object at a time
- readers should not access the object as it is being updated
- additional constraint
  ‣ readers have priority over writers
  ‣ easier to implement
Readers – Writers Problem (2)

- We use two semaphores
  - *mutex*
    - ensure mutual exclusion for the readcount variable
    - mutex semaphore, initialized to 1
  - *wrt*
    - ensure mutual exclusion for writers
    - ensure mutual exclusion between readers and writer
    - mutex semaphore, initialized to 1
semaphore bool mutex, wrt;
int readcount;

**Writer**

```c
do {
    wait (wrt);
    // . . .
    write object resource
    // . . .
    signal (wrt);
} while (TRUE);
```

**Reader**

```c
Reader
do {
    wait (mutex);
    readcount++;
    if (readcount == 1)
        wait (wrt);
    signal (mutex);
    read from object resource
    wait (mutex);
    readcount--;
    if (readcount == 0)
        signal (wrt);
    signal (mutex);
} while (TRUE);
```
Semaphore – Summary

- Semaphores can be used to solve any of the traditional synchronization problems

- Drawbacks of semaphores
  - Semaphores are essentially shared global variables
    - Can be accessed from anywhere in a program
  - Semaphores are very low-level constructs
    - No connection between semaphore and data controlled by a semaphore
    - Difficult to use
  - Used for both critical section (mutual exclusion) and coordination (scheduling)
  - Provides no control of proper usage
    - User may miss a wait or signal, or replace order of wait, and signal

- The solution is to use programming-language level support.
Monitors

- Monitor is a programming language construct that controls access to shared data
  - synchronization code added by the compiler
  - synchronization enforced by the runtime

- Monitor is an abstract data type (ADT) that encapsulates
  - shared data structures
  - procedures that operate on the shared data structures
  - synchronization between the concurrent procedure invocations

- Protects the shared data structures inside the monitor from outside access.

- Guarantees that monitor procedures (or operations) can only legitimately update the shared data.
Monitor Semantics for Mutual Exclusion

- Only one thread can execute any monitor procedure at a time.
- Other threads invoking a monitor procedure when one is already executing some monitor procedure must wait.
- When the active thread exits the monitor procedure, one other waiting thread can enter.
Monitor Account {
    double balance;
    double withdraw (amount) {
        balance = balance - amount;
        return balance;
    }
}
Monitor for Coordination

- What if a thread needs to wait inside a monitor
  - waiting for some resource, like in producer-consumer relationship
  - monitor with condition variables.

- Condition variables provide mechanism to wait for events
  - resource available, no more writers, etc.

![Diagram of Monitor for Coordination]

- Entry Set
- Wait Set
- Owner
- acquire
- release
- release and exit
- enter
- suspended thread
- active thread
- waiting thread
Condition Variable Semantics

- Condition variables support two operations
  - **wait** – release monitor lock, and suspend thread
    - condition variables have wait queues
  - **signal** – wakeup one waiting thread
    - if no process is suspended, then *signal* has no affect

- Signal semantics
  - **Hoare** monitors (original)
    - *signal* immediately switches from the caller to the waiting thread
    - waiter's condition is guaranteed to hold when it continues execution
  - **Mesa** monitors
    - waiter placed on ready queue, signaler continues
    - waiter's condition may no longer be true when it runs
  - Compromise - signaler immediately leaves monitor, waiter resumes operation
Monitor bounded_buffer {
    Resource buffer[N];
    // condition variables
    Condition empty, full;

    void producer (Resource R) {
        while (buffer full)
            empty.wait( );
        // add R to buffer array
        full.signal( );
    }

    Resource consumer ( ) {
        while (buffer empty)
            full.wait( );
        // get Resource from buffer
        empty.signal( );
        return R;
    }
} // end monitor
Condition Variables

- Condition variables are not booleans
  - "if (condition_variable) then … " is not logically correct
  - wait( ) and signal( ) are the only operations that are correct

- Condition variable != Semaphores
  - they have very different semantics
  - each can be used to implement the other

- **Wait ( )** semantics
  - wait blocks the calling thread, and gives up the lock
  - Semaphore::wait just blocks the calling thread
  - only monitor operations can call wait ( ) and signal ( )

- **Signal ( )** semantics
  - if there are no waiting threads, then the signal is lost
  - Semaphore::signal just increases global variable count, allowing entry to future thread
Monitor with Condition Variables
Dining Philosophers Problem

- Represents need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Problem synopsis
- 5 philosophers, circular table
- 2 states, *hungry* and *thinking*
- 5 single chopsticks
- *hungry*, pick up two chopsticks
  - right and left
- may only pick up one stick at a time
- eat when have both sticks

Problem definition
- allow each philosopher to eat and think without deadlocks and starvation
Dining Philosophers Problem (2)

- Restriction on the problem
  - only pick chopsticks if both are available

- Problem solution
  - use three states, *thinking, hungry, eating*
  - condition variable for each philosopher
    - delay if hungry but waiting for chopsticks
  - invoke monitor operations in the following sequence

```java
DiningPhilosophers.pickup (i);
......
// eat
......
DiningPhilosophers.putdown (i);
```
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((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;
    }
} // end monitor
OS Implementation Issues

- How to wait on a lock held by another thread?
  - sleeping or spin-waiting

- Overhead of spin-waiting
  - a spinning thread occupies the CPU; slows progress of all other threads, including the one holding the lock

- Overhead of sleeping
  - issue a wait and sleep; send signal to sleeping thread; wakeup thread; multiple context switches

- Spin-waiting is used on
  - multiprocessor systems
  - when the thread holding the lock is the one running
  - locked data is only accessed by short code segments
OS Implementation Issues (2)

- Reader-writer locks
  - used when shared data is read more often
  - more expensive to set up than mutual exclusion locks

- Non-preemptive kernel
  - process in kernel mode cannot be preempted
  - used in Linux on single processor machines
  - uses preempt_disable() and preempt_enable() system calls
  - spin-locks, semaphores used on multiprocessor machines
Atomic Transactions

- **Transaction** – collection of instructions that perform a single logical function
- **Atomicity** – execute transaction as one uninterruptible unit
- **Mutual exclusion** – execute critical sections atomically
  - what happens if system fails during a transaction?
  - how to preserve atomicity in the possibility of system failures?
- **Committed** – transaction has completed successfully
- **Aborted** – transaction has failed
  - rollback the transaction to previous consistent state, called recovery

**Strategies**
- log-based recovery
- checkpoints
Serializability – execution of multiple concurrent transactions is equivalent to their execution in an arbitrary order.

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>