Semaphore

- Semaphore $S$ – integer variable
- Two standard operations modify $S$: \texttt{wait()} and \texttt{signal()}
  - Originally called \texttt{P()} and \texttt{V()}
- Can only be accessed via two indivisible (atomic) operations
  - \texttt{wait (S) \{ \\
    while S <= 0 \\
    \;
    ; // no-op \\
    \;
    S--; \\
  \}}
  - \texttt{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
- Provides mutual exclusion

```c
Semaphore mutex; // initialized to 1

do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE);
```
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:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - 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

  $P_0$
  
  wait (S);
  wait (Q);
  wait (Q);
  wait (S);
  signal (S);
  signal (Q);
  signal (Q);
  signal (S);

  $P_1$
  
  wait (Q);
  wait (S);
  signal (Q);
  signal (S);

- **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 \texttt{mutex} initialized to the value 1
- Semaphore \texttt{full} initialized to the value 0
- Semaphore \texttt{empty} initialized to the value $N$. 
The structure of the producer process

```c
do {
    // produce an item in nextp
    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 to nextc
    signal (mutex);
    signal (empty);
    // consume the item in nextc
}
```
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

- Shared Data
  - Data set
  - Semaphore **mutex** initialized to 1
  - Semaphore **wrt** initialized to 1
  - Integer **readcount** initialized to 0
Readers-Writers Problem (Cont.)

- The structure of a writer process

```c
    do {
        wait (wrt) ;

        // writing is performed

        signal (wrt) ;
    } while (TRUE);
```
Readers-Writers Problem (Cont.)

- The structure of a reader process

```c
do {
    wait (mutex) ;
    readcount ++ ;
    if (readcount == 1)
        wait (wrt) ;
    signal (mutex)
    // reading is performed

    wait (mutex) ;
    readcount - - ;
    if (readcount == 0)
        signal (wrt) ;
    signal (mutex) ;
} while (TRUE);
```
Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore *chopstick* [5] initialized to 1
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)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait (mutex)` or `signal (mutex)` (or both)
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- 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
**Pthreads Mutex**

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.

- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex.

- Very often the action performed by a thread owning a mutex is the updating of global variables. This is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update.

- When protecting shared data, it is the programmer's responsibility to make sure every thread that needs to use a mutex does so. For example, if 4 threads are updating the same data, but only one uses a mutex, the data can still be corrupted.
Pthreads Condition Variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
  - Without condition variables, the programmer would need to have threads continually polling to check if the condition is met. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.
Main Thread
- Declare and initialize global data/variables which require synchronization
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A
- Do work up to the point where a certain condition must occur
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread Join / Continue
End of Chapter 5