Semaphore synchronization primitive

- Test And Set are hard to program for end users
- Introduce a simple function called semaphore:
  - Semaphore is an integer, S
  - Only legal operations on S are:
    - Wait() [atomic] - if S > 0, decrement S else loop
    - Signal() [atomic] - increment S
  - \[
  \begin{align*}
  \text{wait (S)} & \{ \\
  & \quad \text{while } S \leq 0 \\
  & \quad \quad ; \ // \ no-op \\
  & \quad \quad S--; \\
  & \} \\
  \text{signal (S)} & \{ \\
  & \quad S++; \\
  & \}
  \end{align*}
  \]
  - Counting (S: is unrestricted), binary (mutex lock) (S: 0, 1)
Semaphore usage example

- Assume synch is initialized to 0
  - **P2:**
    - Wait(synch);
    - Statements2;
  - **P1:**
    - Statements1;
    - signal(synch);
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:
- 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.)

wait (S) {
    value--;
    if (value < 0) {
        add this process to waiting queue
        block();
    }
}

Signal (S) {
    value++;
    if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P);
    }
}
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()`
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

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

- In Java, declaring a method `synchronized` to get monitor like behavior
  - What happens to shared variables which are not protected by this monitor?
Solution to Dining Philosophers using Monitors

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;
}
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

  $$
  P_0 \quad \quad P_1
  $$

  \[
  \begin{align*}
  &\text{wait (S);} \\
  &\text{wait (Q);} \\
  &\text{wait (Q);} \\
  &\text{...} \\
  &\text{signal (S);} \\
  &\text{signal (Q);} \\
  \\
  &\text{wait (Q);} \\
  &\text{wait (S);} \\
  &\text{...} \\
  &\text{signal (Q);} \\
  &\text{signal (S);} \\
  \end{align*}
  \]

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
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
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable
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