Semaphore

• Invented by Edsger Dijkstra in 1962
  • When working on and operating system for Electrologica X which became THE.
• A non-negative integer (S) variable on which two operations are allowed
  • P(S) ------ Wait(S)
    • Decrement S
    • Wait until this operation can be carried out.
  • V(S) ------ Signal(S)
    • Increment S
• Both operations are considered Atomic
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore $S$ – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()`
- Definition of the `wait()` operation
  ```c
  wait(S) {
      while (S <= 0)  // busy wait
          S--;
  }
  ```
- Definition of the `signal()` operation
  ```c
  signal(S) {
      S++;
  }
  ```
Information Implications of Semaphore

- A process has synch points
  - To go past a synch point certain conditions must be true
    - Conditions depend not only on ME but other processes also
    - Have to confirm that the conditions are true before proceeding, else have to wait.
  - P(S) – Wait (S)
    - If can complete this operation
      - Inform others through changed value of S
      - Proceed past the synch point
    - If can not complete
      - Wait for the event when S becomes >0
  - V(S) – Signal (S)
    - Inform others that I have gone past a synch point.
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a *mutex lock*
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$
  - Create a semaphore “*synch*” initialized to 0
    - $P_1$:
      - $S_1$;
      - `signal(synch);`
    - $P_2$:
      - `wait(synch);`
      - $S_2$;
- Can implement a counting semaphore $S$ as a binary semaphore
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 S; // initialized to 1
P(S);
criticalSection();
V(S);
```
Implementing S as a Binary Semaphore

• Data structures:

  binary-semaphore S1, S2;
  int C:

• Initialization:

  S1 = 1
  S2 = 0
  C = initial value of semaphore S
Implementing $S$

- **wait** operation

  ```
  wait(S1);
  C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  signal(S1);
  ```

- **signal** operation

  ```
  wait(S1);
  C ++;
  if (C <= 0)
    signal(S2);
  else
    signal(S1);
  ```
Semaphore Implementation

• Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time

• Thus, the 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

• typedef struct{
    int value;
    struct process *list;
} semaphore;
Implementation with no Busy waiting (Cont.)

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

```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

  
  \[
  \begin{align*}
  &P_0 & & P_1 \\
  &\text{wait}(S); & & \text{wait}(Q); \\
  &\text{wait}(Q); & & \text{wait}(S); \\
  &\ldots & & \ldots \\
  &\text{signal}(S); & & \text{signal}(Q); \\
  &\text{signal}(Q); & & \text{signal}(S);
  \end{align*}
  \]

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

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol
Problems with Semaphores

• Incorrect use of semaphore operations:
  • signal (mutex) .... wait (mutex)
  • wait (mutex) ... wait (mutex)
  • Omitting of wait (mutex) or signal (mutex) (or both)

• Deadlock and starvation are possible.
Monitors

• A high-level abstraction that provides a convenient and effective mechanism for process synchronization

• Abstract data type, internal variables only accessible by code within the procedure

• Only one process may be active within the monitor at a time

• But not powerful enough to model some synchronization schemes

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

  procedure Pn (...) {......}

  Initialization code (...) { ... }
}
```
Schematic view of a Monitor

- shared data
- entry queue
- operations
- initialization code
Condition Variables

• `condition x, y;`
• Two operations are allowed on a condition variable:
  • `x.wait()` — a process that invokes the operation is suspended until `x.signal()`
  • `x.signal()` — resumes one of processes (if any) that invoked `x.wait()`
    • If no `x.wait()` on the variable, then it has no effect on the variable
Monitor with Condition Variables
Condition Variables

• If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  • Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

• Options include
  • **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  • **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  • Both have pros and cons – language implementer can decide
  • Monitors implemented in Concurrent Pascal compromise
    • P executing signal immediately leaves the monitor, Q is resumed
  • Implemented in other languages including Mesa, C#, Java
Monitor Implementation Using Semaphores

• Variables

```c
semaphore mutex;  // (initially = 1)
semaphore next;  // (initially = 0)
int next_count = 0;
```

• Each procedure \( F \) will be replaced by

```c
wait(mutex);
...
    body of \( F \);
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

• Mutual exclusion within a monitor is ensured
Monitor Implementation – Condition Variables

• For each condition variable $x$, we have:

```c
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

• The operation $x$.wait can be implemented as:

```c
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```
Monitor Implementation (Cont.)

• The operation `x.signal` can be implemented as:

```c
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```
Resuming Processes within a Monitor

• If several processes queued on condition x, and x.signal() executed, which should be resumed?
• FCFS frequently not adequate
  • conditional-wait construct of the form x.wait(c)
    • Where c is priority number
    • Process with lowest number (highest priority) is scheduled next
Single Resource allocation

• Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

\[
\text{R.acquire}(t); \\
\ldots \\
\text{access the resource;} \\
\ldots \\
\text{R.release;}
\]

• Where R is an instance of type ResourceAllocator
A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator {
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```
Synchronization Examples

• Classic Problems of Synchronization
• Synchronization within the Kernel
• POSIX Synchronization
• Synchronization in Java
• Alternative Approaches
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
Bounded-Buffer Problem

• $n$ buffers, each can hold one item
• Semaphore $\text{mutex}$ initialized to the value 1
• Semaphore $\text{full}$ initialized to the value 0
• Semaphore $\text{empty}$ initialized to the value $n$
The structure of the producer process

do {
    
    ...  
    /* produce an item in next_produced */
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
Bounded Buffer Problem (Cont.)

• The structure of the consumer process

Do {
    wait(full);
    wait(mutex);
...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
...
    /* consume the item in next consumed */
    ...
} while (true);
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

• Several variations of how readers and writers are considered – all involve some form of priorities

• Shared Data
  • Data set
  • Semaphore `rw_mutex` initialized to 1
  • Semaphore `mutex` initialized to 1
  • Integer `read_count` initialized to 0
Readers-Writers Problem (Cont.)

• The structure of a writer process

    do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
    } while (true);
Readers-Writers Problem (Cont.)

• The structure of a reader process

```c
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

    /* reading is performed */

    ...  
    ...  

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(rw_mutex);
    signal(mutex);
} while (true);
```
Readers-Writers Problem Variations

• **First** variation – no reader kept waiting unless writer has permission to use shared object

• **Second** variation – once writer is ready, it performs the write ASAP

• Both may have starvation leading to even more variations

• Problem is solved on some systems by kernel providing reader-writer locks
Dining-Philosophers Problem

• Philosophers spend their lives alternating thinking and eating
• Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  • Need both to eat, then release both when done
• In the case of 5 philosophers
  • Shared data
    • Bowl of rice (data set)
    • Semaphore chopstick [5] initialized to 1
Dining-Philosophers Problem Algorithm

• The structure of Philosopher $i$:
  
  ```
  do {
    wait (chopstick[i]);
    wait (chopStick[(i + 1) % 5]);
    // eat
    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);
    // think
  } while (TRUE);
  ```

• What is the problem with this algorithm?
Monitor Solution to Dining Philosophers

```c
monitor DiningPhilosophers
{
    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;
}

Solution to Dining Philosophers (Cont.)

• Each philosopher $i$ invokes the operations $\text{pickup()}$ and $\text{putdown()}$ in the following sequence:

\[
\text{DiningPhilosophers.pickup}(i); \\
\text{EAT} \\
\text{DiningPhilosophers.putdown}(i);
\]

• No deadlock, but starvation is possible
A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```
Synchronization Examples

• Solaris
• Windows
• 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
  • Starts as a standard semaphore spin-lock
  • If lock held, and by a thread running on another CPU, spins
  • If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

• Uses condition variables

• Uses 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
  • Turnstiles are per-lock-holding-thread, not per-object

• Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
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
  • atomic integers
  • spinlocks
  • reader-writer versions of both

• On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

• Pthreads API is OS-independent

• It provides:
  • mutex locks
  • condition variable

• Non-portable extensions include:
  • read-write locks
  • spinlocks
Alternative Approaches

• Transactional Memory

• OpenMP

• Functional Programming Languages
• A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```c
void update()
{
    /* read/write memory */
}
```
OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.
Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.

Variables are treated as immutable and cannot change state once they have been assigned a value.

There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.