Semaphore

• Invented by Edsger Dijkstra in 1962
  • When working on and operating system for Electrologica X which became THE.

• A non-negative, integer, Global variable (S)
  • Initialized at set up time, and
  • 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
  ```
  wait(S) {
    while (S <= 0) ; // busy wait
    S--;
  }
  ```
- Definition of the `signal()` operation
  ```
  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
    • Must 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 Counting Semaphore using Binary Semaphore

• Data structures:
  
  ```
  binary-semaphore S1, S2;
  int C:
  ```

• Initialization:
  
  ```
  S1 = 1
  S2 = 0
  C = initial value of semaphore S
  ```
Implementing *Counting Semaphore*

- **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();
    }
}

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

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

- **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

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

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

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

- **shared data**
- **queues associated with** $x, y$ **conditions**
- **operations**
- **initialization code**
- **entry queue**
Condition Variables Choices

• 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

September 21

Copyright 2018 Silberschatz, Gavin & Gagne
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:

\[
\text{semaphore } x\text{\_sem}; \quad \text{// (initially } = 0) \\
\text{int } x\text{\_count} = 0;
\]

• The operation \( x\text{.wait} \) can be implemented as:

\[
x\text{\_count}++;
\text{if } (next\text{\_count} > 0)
\quad \text{signal}(next);
\text{else}
\quad \text{signal}(mutex);
\text{wait}(x\text{\_sem});
\text{x\_count}--;
\]
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

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

\[
\ldots \\
R.\text{release};
\]

• 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;
    }
}
```