Synchronization with HW support

• Disable interrupts
  – Good for short time wait, not good for long time wait
  – Not good for multiprocessors
    • Interrupts are disabled only in the processor used

• Test-and-set instruction (etc)
  – Good for short time wait, not good for long time wait
  – Nor so good in single processor system
    • May reserve CPU, which is needed by the process holding the lock
  – Waiting is usually “busy wait” in a loop

• Good for mutex, not so good for general synchronization
  – E.g., “wait until process P34 has reached point X”
  – No support for long time wait (in suspended state)

• Barrier wait in HW in some multicore architectures
  – Stop execution until all cores reached barrier_wait instruction
  – No busy wait, because execution pipeline just stops
  – Not to be confused with barrier_wait thread operation
Semaphores

- Dijkstra, 1965, THE operating system
- Protected variable, abstract data type (object)
  - Allows for concurrency solutions if used properly
- Atomic operations
  - Create (SemaName, InitValue)
  - P, *down, wait, take, pend, passeren, proberen, try, prolaad, try to decrease*
  - V, *up, signal, release, post, vrijgeven, verlagen, verhoog, increase*
(Basic) Semaphore

- **P(S)**
  - If value > 0, deduct 1 and proceed
  - o/w, wait suspended in list (queue?) until released
- **V(S)**
  - If someone in queue, release one (first?) of them
  - o/w, increase value by one
General vs. Binary Semaphores

• **General Semaphore**
  – Value range: 0, 1, 2, 3, ….
    • nr processes doing P(S) and advancing without delay
    • Value: “Nr of free units”, “nr of advance permissions”

• **Binary semaphore (or “mutex”)**
  – Value range: 0, 1
    • Mutex lock (with suspended wait)
    • V(S) can (should!) be called only when value = 0
      – By process in critical section (CS)
    – Many processes can be in suspended in list
    – At most one process can proceed at a time
• Someone must create S
  – Value initialized to 1
• Possible wait in suspended state
  – Long time, hopefully at least 2 process switches

Some (operating) systems have “semaphores” with (optional) busy wait (i.e., busy-wait semaphore).
Beware of busy-wait locks hidden in such semaphores!
General Semaphore Implementation

• P(S)
  
  if (S.value > 0)
  S.value = S.value - 1
  else
  suspend calling process P
  place P (last?) in S.list
  call scheduler()

• V(S)
  
  if (S.list == empty)
  S.value = S.value + 1
  else
  take arbitrary (or 1st?) process Q from S.list
  move Q to ready-to-run list
  call scheduler()
Semaphore Implementation

- Use HW-supported busy-wait locks to solve mutex-problem for semaphore operations
  - Short waiting times, a few machine instructions

- Use OS suspend operation to solve semaphore synchronization problem
  - Possibly very long, unlimited waiting times
  - Implementation at process control level in OS
  - This is the resume point for suspended process
    - Deep inside in privileged OS-module
Semaphore Implementation Variants

• Take **first** process in S.list in V(S)?
  – Important semantic change, affects applications
  – Fairness
  – **Strong** semaphore
    (vs. **weak semaphore** with no order in S.list)

• Add to/subtract from S.value **first** in P(S) and in V(S)?
  – Just another way to write code

• Scheduler call every time or sometimes at P or V end?
  – Semantic change, may affect applications
  – Execution turn may (likely) change with P even when process is not suspended in wait
  – Signalled process may start execution immediately
Semaphore Implementation Variants

• S.value can be negative
  – Negative S.value gives the number of waiting processes?
  – Makes it easier to poll number of waiting processes
    • New user interface to semaphore object

• Busy-wait semaphore
  – Wait in busy loop instead of in suspended state
  – Really a busy-wait lock that looks like a semaphore
  – Important semantic change, affects applications

n = value(s);
Blocking Semaphore

- **Blocking**
  - Normal (counting) semaphore with initial value \( = 0 \)
  - **First** \( P(S) \) will block, unless \( V(S) \) was executed first

Example: synchronization between two processes

- Wait for \( Q \)
- \( R \)
- \( Q \)
- Signal \( R \)
- \( R \) \( Q \)
- Wait for \( Q \)
- \( R \)
- Signal \( R \)

- \( P(S) \)
- \( V(S) \)

\( \text{(no wait)} \) \quad \text{Will block if executed first}
Producer-Consumer Problem

- Synchronization problem
- Correct execution order
- Producer places data in buffer
  - Waits if finite size buffer full
- Consumer takes data from buffer
  - Same order as they were produced
  - Waits if no data available
- Variants
  - Cyclic finite buffer – usual case
  - Infinite buffer
    - Realistic sometimes!
      - External conditions rule out buffer overflow?
      - Can be implemented with finite buffer!
  - Many producers and/or many consumers
**Algorithm 6.6: Producer-consumer (infinite buffer)**

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>infinite queue of dataType buffer ← empty queue</td>
<td></td>
</tr>
<tr>
<td>semaphore notEmpty ← (0, Ø)</td>
<td></td>
</tr>
<tr>
<td>dataType d</td>
<td>dataType d</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: d ← produce</td>
<td>q1: wait(notEmpty)</td>
</tr>
<tr>
<td>p2: append(d, buffer)</td>
<td>q2: d ← take(buffer)</td>
</tr>
<tr>
<td>p3: signal(notEmpty)</td>
<td>q3: consume(d)</td>
</tr>
</tbody>
</table>

- Synchronization only one way (producer never waits)
  - Synchronization from producer to consumer
- Counting split semaphore notEmpty
  - Split = “different processes doing waits and signals”
  - Value = nr of data items in buffer
- Append/take might need to be indivisible operations
  - Protect with semaphores or busy-wait locks?
  - Not needed now? Maybe not? (only one producer/consumer)
### Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

finite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, Ø)
semaphore notFull ← \((N, Ø)\)

<table>
<thead>
<tr>
<th>producer</th>
<th>consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataType d</td>
<td>dataType d</td>
</tr>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: d ← produce</td>
<td>q1: wait(notEmpty)</td>
</tr>
<tr>
<td>p2: wait(notFull)</td>
<td>q2: d ← take(buffer)</td>
</tr>
<tr>
<td>p3: append(d, buffer)</td>
<td>q3: signal(notFull)</td>
</tr>
<tr>
<td>p4: signal(notEmpty)</td>
<td>q4: consume(d)</td>
</tr>
</tbody>
</table>

- Synchronization both ways, both can wait
- New semaphore notFull: value = nr of free slots in buffer
- Split semaphore notEmpty & notFull
  - \(\text{notEmpty}.\text{value} + \text{notFull}.\text{value} = N\) in (p1, q4, …)
  - When both at the beginning of loop, outside wait-signal area
    - wait(notFull)…signal(notEmpty), wait(notEmpty)…signal(notFull)
Size N buffer
One producer
One consumer

\[
\begin{array}{cccccccccc}
\text{b[0]} & \text{b[1]} & \text{b[2]} & \text{b[3]} & \text{b[4]} & \text{b[5]} & \cdots & \text{b[n-1]} \\
\end{array}
\]

\text{front} \quad \text{rear}

typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;

process Producer {
    while (true) {
        ...
        produce message data
        P(empty);
        buf[rear] = data;
        rear = (rear+1) % n;
        V(full);
    }
}

process Consumer {
    while (true) {
        fetch and consume:
        P(full);
        result = buf[front];
        front = (front+1) % n;
        V(empty);
        ...
    }
}

Does it work with one producer and one consumer? Yes.
Mutex problem? No. Why not?

Does it work with many producers or consumers? No.
Semaphores or busy wait?

Prod/Consumers
Size N buffer
Many producers
Many consumers

Semaphore full for synchronization
Semaphore mutexF for mutex problem

Why separate mutexD and mutexF?

(Andrews, Fig. 4.5)
Barz’s General Semaphore Simulation

• Starting point
  – Have binary semaphore
  – Need counting semaphore
  – Realistic situation

• Operating system or programming language library may have only binary semaphores

\[
k = 4
\]
4 in CS, 2 in gate
1 completes CS
What now?
2 complete CS?

\[
\text{loop forever}
\]
non-critical section
p1: wait(gate)
p2: wait(S)
p3: count ← count − 1
p4: if count > 0 then
p5: signal(gate)
p6: signal(S)
critical section
p7: wait(S)
p8: count ← count + 1
p9: if count = 1 then
p10: signal(gate)
p11: signal(S)
Udding’s No-Starvation Critical Section with Weak Split Binary Semaphores

- **Weak semaphore**
  - Set, not a queue in wait

- **Split binary semaphore**
  - $0 \leq gate1 + gate2 \leq 1$

- **Batch arrivals**
  - Start service only when no more arrivals
  - Close gate1 during service

- **No starvation**
  - gate1 opened again only after whole batch in gate2 is serviced

```
semaphore gate1 ← 1, gate2 ← 0
integer numGate1 ← 0, numGate2 ← 0

p1: wait(gate1)

p2: numGate1 ← numGate1 + 1

p3: signal(gate1)

p4: wait(gate1)

p5: numGate2 ← numGate2 + 1
    numGate1 ← numGate1 - 1

p6: if numGate1 > 0

p7: signal(gate1)

p8: else signal(gate2)

p9: wait(gate2)

p10: numGate2 ← numGate2 - 1

p11: if numGate2 > 0

p12: signal(gate2)

p13: else signal(gate1)  # last in batch

 Alg 6.14
```
Semaphore Features

• Utility provided by operating system or programming language library
• Can be used solve almost any synchronization problem
• Need to be used carefully
  – Easy to make profound errors
    • Forget V
    • Suspend process in critical section
      – No one can get CS to resume suspended process
      – Someone may be waiting in busy-wait loop
    • Deadlock
  – Need strong coding discipline
Trivial Solution #1

```
/* program diningphilosophers */
semaphore fork [5] = {1}; /* mutex, one at a time */
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]); /* left fork */
        wait (fork [(i+1) mod 5]); /* right fork */
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
                philosopher (3), philosopher (4));
}
```

• Possible deadlock – not good
  – All 5 grab left fork “at the same time”
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4}; /* only 4 at a time, 5th waits */
int i;
void philosopher (int I)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
### Algorithm \textbf{AS} : Dining philosophers (good solution)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore array [0..4] fork ← [1,1,1,1,1]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: think</td>
</tr>
<tr>
<td>p2: wait(fork[i])</td>
</tr>
<tr>
<td>p3: wait(fork[i+1])</td>
</tr>
<tr>
<td>p4: eat</td>
</tr>
<tr>
<td>p5: signal(fork[i])</td>
</tr>
<tr>
<td>p6: signal(fork[i+1])</td>
</tr>
</tbody>
</table>

### philosopher 4

<table>
<thead>
<tr>
<th>loop forever</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: think</td>
</tr>
<tr>
<td>p2: wait(fork[0])</td>
</tr>
<tr>
<td>p3: wait(fork[4])</td>
</tr>
<tr>
<td>p4: eat</td>
</tr>
<tr>
<td>p5: signal(fork[0])</td>
</tr>
<tr>
<td>p6: signal(fork[4])</td>
</tr>
</tbody>
</table>

- No deadlock, no starvation
- No extra blocking
- **Asymmetric** solution – not so nice…
  - All processes should execute the same code
- Simple primitives, must be used properly

Even numbered philosophers?  
or  
This way with 50% chance?  
or  
This way with 20% chance?  
Etc. etc.
void semaphore_server() {
    message m;
    int result;
    /* Initialize the semaphore server. */
    initialize();
    /* Main loop of server. Get work and process it. */
    while(TRUE) {

        /* Block and wait until a request message arrives. */
        ipc_receive(&m);

        /* Caller is now blocked. Dispatch based on message type. */
        switch(m.m_type) {
            case UP:    result = do_up(&m);    break;
            case DOWN:  result = do_down(&m);  break;
            default:    result = EINVAL;
        }

        /* Send the reply, unless the caller must be blocked. */
        if (result != EDONTREPLY) {
            m.m_type = result;
            ipc_reply(m.m_source, &m);
        }
    }
}

Minix Semaphore P

```c
int do_down(message *m_ptr) {
    /* Resource available. Decrement semaphore and reply. */
    if (s > 0) {
        s = s - 1;       /* take a resource */
        return(OK);      /* let the caller continue */
    }
    /* Resource taken. Enqueue and block the caller. */
    enqueue(m_ptr->m_source); /* add process to queue */
    return(EDONTREPLY); /* do not reply in order to block the caller */
}
```

Suspend in message queue!
int do_up(message *m_ptr) {
    message m;  /* place to construct reply message */
    /* Add resource, and return OK to let caller continue. */
    s = s + 1;  /* add a resource */

    /* Check if there are processes blocked on the semaphore. */
    if (queue_size() > 0) {  /* are any processes blocked? */
        m.m_type = OK;
        m.m_source = dequeue();  /* remove process from queue */
        s = s - 1;  /* process takes a resource */
        ipc_reply(m.m_source, m); /* reply to unblock the process */
    }
    return(OK);  /* let the caller continue */
}
Semaphores in Linux

http://fxr.watson.org/fxr/source/include/asm-sh/semaphore.h?v=linux-2.4.22

- semaphore.h
- Low level process/thread control
- In assembly language, in OS kernel
- struct semaphore {
  atomic_t count;
  int sleepers;
  wait_queue_head_t wait;
}
- sema_init(s, val)
- init_MUTEX(s), init_MUTEX_LOCKED(s)
- down(s), int down_interruptible(s), int down_trylock(s)
- up(s)
Semaphores in BACI with C--

• Weak semaphore
  – S.list is a set, not a queue
  – Awakened process chosen in random

• Counting semaphore: \( semaphore \, count; \)

• Binary semaphore: \( binarysem \, mutex; \)

• Operations
  – \textit{Initialize} \( (count, \, 0); \)
  – \( P() \) and \( V() \)
  – Also \( wait() \) and \( signal() \) in addition to \( P() \) and \( V() \)
  – Value can be used directly: \( n = count; \) \( cout \, count; \)

\textbf{current value of semaphore count}
Semaphore Example

`semexample.cm`

```c
semaphore count;       // a "general" semaphore
binarysem output;     // a binary (0 or 1) semaphore for unscrambling output

main()
{
    initialisem(count, 0);
    initialisem(output, 1);
    cobegin {
        decrement(); increment();
    }
} // main

void increment()
{
    p(output);           // obtain exclusive access to standard output
    cout << "before v(count) value of count is " << count << endl;
    v(output);
    v(count);            // increment the semaphore
} // increment

void decrement()
{
    p(output);           // obtain exclusive access to standard output
    cout << "before p(count) value of count is " << count << endl;
    v(output);
    p(count);            // decrement the semaphore (or stop -- see manual text)
} // decrement
```

(BACI C- - User’s Guide)
C- - Semaphore Example

- 3 possible outcomes
  - how?
    
    ```
    Executing PCODE ...
    before v(count) value of count is 0
    before p(count) value of count is 1
    ```
  - how?
    
    ```
    Executing PCODE ...
    before p(count) value of count is 0
    before v(count) value of count is 0
    ```
  - how?
    
    ```
    Executing PCODE ...
    before v(count) value of count is 0
    before p(count) value of count is 0
    ```
  - Why no other possible outcome?

(BACI C- - User’s Guide)
Semaphores in Java

- Class `Semaphore` in package `java.util.concurrent`
  - Permit value can be positive and negative
  - Permits can be initialized to negative numbers
  - Semaphore type
    - fair (= strong) & nonfair (≈ busy-wait ??), default

- Wait(S):
  ```java
  try {
    s.acquire();
  }
  catch (InterruptedException e) {} 
  ```

- Signal(S):
  ```java
  s.release();
  ```

- Many other features
Java Example

• Simple Java-solution with semaphore

  vera: javac Plusminus_sem.java
  vera: java Plusminus_sem

  http://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Plusminus_sem.java

• Still fairly complex
  – Not as streamlined as P() and V()

• How does it *really* work?
  – Busy wait or suspended wait?
  – Fair queueing?
  – Overhead when no competition for CS?
Semaphore Summary

- Most important synchronization primitive
  - Implementation needs OS assistance
- Can do anything
  - Just like assembly language coding…
- Many variants
  - Counting, binary, split, neg. values, mutex
- Programming language interfaces vary