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
  - Not 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

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
Lecture 6: 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

Blocking Semaphore

- Blocking
  - Normal (counting) semaphore with initial value = 0
    - First P(S) will block, unless V(S) was executed Q
  - ... Wait for Q (no wait)
  - ... Signal R
  - ... Wait for Q (wait)
  - Signal R

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

Create(S, 0)
P(S) V(S)

Tuottaja-kuluttaja-ongelma
Lecture 6: Semaphores

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

Algorithm 6.5: Producer-consumer (infinite buffer, semaphores)

<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: 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 wait 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 6.6: Producer-consumer (finite buffer, semaphores)

<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: append(d, buffer)</td>
<td>q2: d = take(buffer)</td>
</tr>
<tr>
<td>p3: signal(notEmpty)</td>
<td>q3: consume(d)</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
  - notEmpty value + notFull value = N
  - 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

process Producer()
while (true) {
  produce message data and deposit it in the buffer;
  signal(bufferNotEmpty);
}

process Consumer()
while (true) {
  if (bufferNotEmpty)
    data = (receive from buffer);
  else wait(bufferNotEmpty);
  V(full);
}

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

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

Algorithm 6.5: Producer-consumer (infinite buffer, semaphores)

<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: 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 wait 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)
Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline

Semaphore Features

- Utility provided by operating system or programming language library
- Can be used to 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
    - Deadlock
      - Need strong coding discipline
Minix Semaphore V

```c
int do_upmessage *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 = decrepil; /* remove process from queue */
        s = s - 1;
        /* process takes a resource */
        ipc_needs0,m_source, m, l;
        /* reply to unblock the process */
    }
    return(OK);
    /* let the caller continue */
}
```

Semaphores in Linux

- `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
  - Initialize (count, 0);
  - `P()` and `V()`
  - Also `wait()` and `signal()` in addition to `P()` and `V()`
  - Value can be used directly: n = count; count = n;

C -- Semaphore Example

```c
semaphore count; // a "general" semaphore
binarysem output; // a binary (0 or 1) semaphore for unscrambling output words:
    {
        initcount(0);
        initsemaphore(output);
        collegs {
            decrement();
            increment();
        }
    } // main
void increment()
    {
        p(output); // obtain exclusive access to standard output
        out = "before v(count) value of count is " + count + " end;"
        v(output);
        v(count); // increment the semaphore
    }
void decrement()
    {
        p(output); // obtain exclusive access to standard output
        out = "before p(count) value of count is " + count + " end;"
        v(output);
        p(count); // decrement the semaphore (or stop -- see manual text)
    } // decrement
```

Semaphores in Java

- Class Semaphore in package java.util.concurrent
- `Semaphore` value is `S.permits` in Java
  - 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

Lecture 6: Semaphores
Java Example

- Simple Java-solution with semaphore
  - Simple Java-solution with semaphore
  
  ```
  vera: javac Plusminus_sem.java
  vera: java Plusminus_sem
  [https://www.cs.helsinki.fi/u/kerola/rio/Java/examples/Plusminus_sem.java](https://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