Semaphores

Producer-Consumer Problem

Semaphores in C++, Java, Linux, Minix
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)
    - Usually initial value 1
    - 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 (and just one!) must create $S$
  – Value initialized to 1 (in this example)

• 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)**

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

- **V(S)**

  ```java
  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()
  ```

Atomic operations!
How?
Use HW mutex support!

Tricky part:
section of CS
is in operating system 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
  - Process waits in suspended waiting state
  - 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 or V even when calling process is not suspended in wait
  - Signalled process may start execution immediately
Semaphore Implementation Variants

- **S.value can be negative**
  - $P(S)$ always deducts 1 from S.value
  - Negative S.value gives the number of waiting processes?
  - Makes it easier to poll number of waiting processes
- **New user interface to semaphore object?** $n = \text{value}(s)$

- **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
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

```
R
....
Wait for Q
....
(no wait)
....
Wait for Q
....
(wait)
Q
....
Signal R
....

R Q
....
Signal R
....

Q
....
Signal R
....

R
....
P(S)
....

Q
....
V(S)
....
```

Create( S, 0)

Will block if executed first

3.2.2011 Copyright Teemu Kerola 2011
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 – producer can not wait
      - 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><code>dataType d</code></td>
<td><code>dataType d</code></td>
</tr>
<tr>
<td><code>loop forever</code></td>
<td><code>loop forever</code></td>
</tr>
<tr>
<td><code>p1: d ← produce</code></td>
<td><code>q1: wait(notEmpty)</code></td>
</tr>
<tr>
<td><code>p2: append(d, buffer)</code></td>
<td><code>q2: d ← take(buffer)</code></td>
</tr>
<tr>
<td><code>p3: signal(notEmpty)</code></td>
<td><code>q3: consume(d)</code></td>
</tr>
</tbody>
</table>

- Synchronization only **one way** (producer never waits)
  - Synchronization from producer to consumer
- Counting semaphore `notEmpty`
  - 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)

Discuss
### Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

| finite queue of dataType buffer ← empty queue |
| semaphore notEmpty ← (0, ∅) |
| semaphore notFull ← (N, ∅) |

#### producer
- dataType d
- loop forever
  - p1: d ← produce
  - p2: wait(notFull)
  - p3: append(d, buffer)
  - p4: signal(notEmpty)

#### consumer
- dataType d
- loop forever
  - q1: wait(notEmpty)
  - q2: d ← take(buffer)
  - q3: signal(notFull)
  - q4: consume(d)

- 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 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

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

Does it work with many producers or consumers? No.

```c
  int front, rear;
} typeT;

int 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);
    ...
  }
}
```
Semaphore mutexD for mutex problem

Semaphore full for synchronization

Why separate mutexD and mutexF?

Prod/Consumers
Size N buffer
Many producers
Many consumers

Need mutexes!
Semaphores or busy wait?

typeT buf[n];  /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;  /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion */

process Producer[i = 1 to M] {
    while (true) {
        produce message data and deposit it in the buffer;
        P(empty);
        P(mutexD);
        buf[rear] = data; rear = (rear+1) % n;
        V(mutexD);
        V(full);
    }
}

process Consumer[j = 1 to N] {
    while (true) {
        fetch message result and consume it;
        P(full);
        P(mutexF);
        result = buf[front]; front = (front+1) % n;
        V(mutexF);
        V(empty);
    }
}

(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

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

\[
\text{critical section to implement V}
\]
Udding’s No-Starvation Critical Section with Weak Split Binary Semaphores

- Weak semaphore
  - Set, not a queue in wait
- Split binary semaphore
  \[ 0 \leq \text{gate1} + \text{gate2} \leq 1 \]
- 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
    critical section
p11: if numGate2 > 0
p12: signal(gate2)
p13: else signal(gate1)
```

Copyright Teemu Kerola 2011

3.2.2011
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 (with P)
      - No one can get CS to resume suspended process
      - Someone may be waiting in busy-wait loop
    - Deadlock
  - Need strong coding discipline
Possible deadlock – not good
  – All 5 grab left fork “at the same time”
No deadlock, no starvation
Waiting when resources are available – which scenario? – not good
Algorithm AS: Dining philosophers (good solution)

| semaphore array [0..4] fork ← [1,1,1,1,1] |

loop forever
p1:    think
p2:    wait(fork[i])
p3:    wait(fork[i+1])
p4:    eat
p5:    signal(fork[i])
p6:    signal(fork[i+1])

philosopher 4

loop forever
p1:    think
p2:    wait(fork[0])
p3:    wait(fork[4])
p4:    eat
p5:    signal(fork[0])
p6:    signal(fork[4])

- 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

Symmetric solutions?
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);
        }
    }
}

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**
  - `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;`

`current value of semaphore count`
Semaphore

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

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
}

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)
}

(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*
  
  [http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html)

- S.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
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 high level synchr. primitive
  - Implementation needs OS assistance
  - Wait in suspended state
  - Should wait relatively long time
    - Costs 2 process switches (wait – resume)
- Can do anything
  - Just like assembly language coding…
- Many variants
  - Counting, binary, split, blocking, neg. values, mutex
- Programming language interfaces vary
- No need for shared memory areas
  - Enough to invoke semaphore operations in OS or programming language libraries
Summary

• Semaphore structure, implementation, and use
  – “Busy wait semaphores”
• Producer-Consumer problem and its variants
  – Semaphores for synchronization and for mutex
• Emulate advanced semaphores with simpler ones
  – Barz, Udding
• Semaphores in Linux (C), C++, Java