Semaphores

Ch 6 [BenA 06]

Semaphores
Producer-Consumer Problem
Semaphores in C++, Java, Linux, Minix

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

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

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
  - Not to be confused with barrier_wait thread operation

(Basic) Semaphore

public semaphore S
public P(S)
public V(S)
private int S.value
private int S.list

public create (SemaName, InitValue)
public P(S)
public V(S)

private integer value
private int S.value
private int S.list

queue of waiting processes

- P(S)
  - WAIT(S), Down(S)
  - If value > 0, deduct 1 and proceed
  - o/w, wait suspended in list (queue?) until released
- V(S)
  - SIGNAL(S), Up(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

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()
  - 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()

- **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, 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 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

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)
    - Signal R
  - Q
    - Wait for R
    - (wait)
    - Signal R
  - Create( S, 0)
  - R
    - P(S)
  - Q
    - V(S)
  - 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 – producer can not wait
      - External conditions rule out buffer overflow?
    - Can be implemented with finite buffer?
  - Many producers and/or many consumers

Note: This is not a critical section (CS) problem, but CS may be used in the solution.
Requirements are too complex for plain CS.
Lecture 6: Semaphores

- 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

- Synchronization both ways, both can wait

- New semaphore notFull: value = nr of free slots in buffer

- Split semaphore

- 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
  - Does it work with one producer and one consumer? Yes. Why not?
  - 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

- Udding’s No-Starvation Critical Section with Weak Split Binary Semaphores

- Weak semaphore
  - Set, not a queue in wait

- Split binary semaphore
  - 0 ≤ gate1 + gate2 ≤ 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

- Mutex problem?

Discuss
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 (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”

Trivial Solution #1

```c
/* mutex, one at a time */
/* left fork */
/* right fork */

Symbolic solutions?
Even numbered philosophers?
or
This way with 50% chance?
or
This way with 20% chance?
Etc. etc.
```

Minix Semaphore P

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

Suspend in message queue!
Minix Semaphore V

```c
int do_up(message *m_pnt) {
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 */
        inc_queue(m_source, m);  /* reply to unlock 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

```c
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
}
```

```c
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;`

Semaphores in Java

- Class Semaphore in package java.util.concurrent
  - 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): `s.release();`
  - Many other features
Java Example

- Simple Java-solution with semaphore
  - `java javac Plusminus_sem.java`
  - `java Plusminus_sem`

- Still fairly complex
  - Not as streamlined as P() and V()
- How does it really work?
  - Busy wait or suspended wait?
  - Fair queuing?
  - 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

20.1.2012 Copyright Teemu Kerola 2012