Semaphores and Monitors

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Summer 2008
From locks and barriers...

- Are busy-waiting protocols complex?
- No clear distinction between variables used:
  - for synchronization
  - for computing results
- Busy-waiting is often inefficient
  - Usually more processes/threads than processors
  - Processor executing a spinning process can be more productively employed to execute another process

Semaphores

First synchronisation tool (and remains one of the most important).
⇒ Easy to protect critical sections.
⇒ Included in (almost) all parallel programming libraries.
Semaphore

Shared variable with 2 *atomic* methods:

```java
void down(Semaphore s) {
  \( Probeer \ (try) / Passeren \ (pass) / Pakken \ (grab) \)
  \(< \ wait \ until \ c > 0, \ then \ c := c - 1; > \)
  \( must \ be \ atomic \ once \ c > 0 \ is \ detected \)
}
```

```java
void up(Semaphore s) {
  \( Verhoog \ (increase) \)
  \(< c = c + 1; > \)  \( must \ be \ atomic \)
}
```

```java
void Init(Semaphore s, Integer i) {
  c := i;
}
```
Semaphores, what for?

- Critical section: Mutual exclusion
- Barriers: Signaling events
- Producers and Consumers
- Bounded buffers: Resource counting
Critical section

```
sem mutex; Init(sem,1);

while (true) {
    down(mutex);
    Critical section
    up(mutex);
    NON-Critical section
}
```

```
while (true) {
    down(mutex);
    Critical section
    up(mutex);
    NON-Critical section
}
```
Semaphores

Monitors

Conclusion

Barriers: Signaling events

```c
sem arrive1, arrive2;
Init(arrive1, 0); Init(arrive2, 0);
```

Barrier

Section A

Section B

... ▶️ process 1 in section A
up(arrive1); ▶️ signal arrival
down(arrive2); ▶️ Wait for the other process
... ▶️ process 1 in section B

... ▶️ process 2 in section A
up(arrive2); ▶️ signal arrival
down(arrive1); ▶️ Wait for the other process
... ▶️ process 2 in section B
Semaphores: Producers and Consumers

Producer ➔ Shared Resource ➔ Consumer

OS2’08 | Semaphores and Monitors
Semaphores: Producers and Consumers

Split Binary Semaphore
Both are *binary semaphores*
Semaphores: Producers and Consumers

```c
typedef T buf; // Buffer of some type T
semempty, full;
Init(empty,1); Init(full,0);

while (true) {  // Producer
    ...
    down(empty);
    buf = data;
    up(full);
    ...
}

while (true) {  // Consumer
    ...
    down(full);
    result = buf;
    up(empt
    ...
```
In the last example...

- Single communication buffer
- No waiting if data are produced and consumed at the same rate
- But in general, producer/consumer execution is bursty

Example

- producer produces several items in a quick succession
- does more computation
- produces another set of items

Solution: Increase the buffer capacity
Bounded buffer: Resource counting

Producer → Buffer → Consumer

put

take

Buffer

Empty:

Full:

0

$\text{n}$
The Buffer

- **Put**: \( \text{buf}[\text{rear}] = \text{data}; \text{rear} = (\text{rear} + 1) \mod n \)
- **Take**: \( \text{result} = \text{buf}[\text{front}]; \text{front} = (\text{front} + 1) \mod n \)
Bounded buffer: Resource counting

typeT buf[n];  \(\triangleright Array \text{ of some type } T\)
int front=0, rear=0;
sem empty, full;
Init(empty,n); Init(full,0);
sem mutexP, mutexT; Init(mutexP,1), Init(mutexT,1);

while (true) { \(\triangleright Producer\)

... 
down(empty);
down(mutexP);
buf[rear] = data;
rear = (rear+1) %n;
up(mutexP);
up(full);
...
}

while (true) { \(\triangleright Consumer\)

... 
down(full);
down(mutexT);
result = buf[front];
front = (front+1) %n;
up(mutexT);
up(empty);
...
}
Semi-Conclusion

- Critical Section
- Blocking semaphore
- Split Binary semaphore
- Resource Counting

Barriers/Signaling
Dining Philosophers Problem

Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the center of the table is a large pater of spaghetti. Because the spaghetti is long and tangled – and the philosophers are not mechanically adept – a philosopher must use two forks to eat a helping. Unfortunately, the philosophers can afford only five forks. One fork is placed between each pair of philosophers. And they agree that each will use only the forks to the immediate left and right. The problem is to write a program to simulate the behavior of the philosophers. The program must avoid the unfortunate (and eventually fatal) situation in which all philosophers are hungry but non is able to acquire both forks – for example, each holds one fork and refuses to give it up.
Dining Philosophers Problem

\[
\text{while (true) \{ } \quad \triangleright \text{Philosopher}_i \\
\quad \text{think; } \\
\quad \text{acquire forks; } \\
\quad \text{eat; } \\
\quad \text{release forks; } \\
\text{\} }
\]

\[
\text{sem fork[5] = \{1,1,1,1,1\} } \\
\text{while (true) \{ } \quad \triangleright \text{Philosopher}_0, 1, 2, 3, 4 \\
\quad \text{think; } \\
\quad \text{down(fork[i]); down(fork[i+1]); } \\
\quad \text{eat; } \\
\quad \text{up(fork[i]); up(fork[i+1]); } \\
\text{\} }
\]
Deadlocks: Resource Allocation Graph

Deadlock?

- No cycle ⇒ No process is deadlocked
- If cycle, deadlock may exist
Cycle in the Graph?

If each resource has ONE instance
- cycle $\Rightarrow$ deadlock
- Each process involved in the cycle is deadlocked
- Both necessary and sufficient condition for deadlock

If each resource has SEVERAL instance
- cycle $\not\Rightarrow$ deadlock
- Necessary but not sufficient condition for deadlock
RAG example

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

$P_1, P_2, P_3$ are deadlocked
Mutual exclusion
At least one resource must be nonsharable (only one process can use it)

Hold and wait
At least one process holds at least one resource and waits for more resources which are held by other processes

No preemption
Only the process holding a resource can release it.

Circular wait
A set of processes are waiting for resources held by others in a circular manner $< P_0, ..., P_n >$ where $P_i$ waits for a resource held by $P_{(i+1) \% n}$
Dining Philosophers Starvation ...solved

while (true) {
    //Philosopher_0, 1, 2, 3
    think;
down(fork[i]); down(fork[i+1]);
eat;
up(fork[i]); up(fork[i+1]);
}

while (true) {
    //Philosopher_4
    think;
down(fork[0]); down(fork[4]);
eat;
up(fork[0]); up(fork[4]);
}
Readers/Writers

- Example of selective mutual exclusion
- Example of general condition synchronization
Readers/Writers as an Exclusion Problem

- **First Solution:**
  1. Overconstrain the problem
  2. Relax the constraints

- Let $rw$ be a mutual exclusion semaphore $\Rightarrow \text{Init}(rw,1)$;

Readers $1, \ldots, M$

\[
\text{while (true) } \{ \\
\quad \ldots \\
\quad \text{down}(rw); \; \triangleright \text{grab exclusive access lock} \\
\quad \text{Read the database} \\
\quad \text{up}(rw); \; \triangleright \text{release the lock} \\
\quad \ldots \\
\}\]

- Readers – as a group – need to lock out writers
- but only the *first* needs to grab the lock (i.e. \text{down}(rw))
- Subsequent readers can directly access the database

Writers $1, \ldots, N$

\[
\text{while (true) } \{ \\
\quad \ldots \\
\quad \text{down}(rw); \; \triangleright \text{grab exclusive access lock} \\
\quad \text{Write the database} \\
\quad \text{up}(rw); \; \triangleright \text{release the lock} \\
\quad \ldots \\
\}\]
Relaxing constraints

```c
int nr = 0;
sem rw; Init(rw,1);
sem mutexR; Init(mutexR,1);

Readers 1,...,M

while (true) {
    ...
    down(mutexR);
    nr = nr + 1;
    if (nr == 1) down(rw);  // if first, get lock
    up(mutexR);
    Read the database
    down(mutexR);
    nr = nr - 1;
    if (nr == 0) up(rw);    // if last, release lock
    up(mutexR);
    ...
}

Writers 1,...,N

while (true) {
    ...
    down(rw);  // grab exclusive access lock
    Write the database
    up(rw);    // release the lock
    ...
}
```

▷ number of active readers
▷ lock for reader/writer exclusion
▷ lock for reader access to nr
Second Solution:

1. Count the number of each kind of processes trying to access the database
2. Constrain the values of the counters

Let $nr$ and $nw$ be nonnegative counters;

BAD: $(nr > 0 \land nw > 0) \lor nw > 1$

Symmetrically, good states, $RW = \overline{BAD}$

$RW$: $(nr == 0 \lor nw == 0) \land nw \leq 1$
Coarse-grained solution using Condition Synchronization

```c
int nr = 0, nw = 0;
\[ RW: (nr == 0 \lor nw == 0) \land nw \leq 1 \]

Readers \(1, \ldots, M\)

```c
while (true) {
    ...,
    < await(nw == 0) nr = nr + 1;
    >
    Read the database
    < nr = nr - 1; >
    ...
}
```

Writers \(1, \ldots, N\)

```c
while (true) {
    ...,
    < await(nr == 0 and nw == 0) nw = nw + 1; >
    Write the database
    < nw = nw - 1; >
    ...
}
```
So ... semaphores, really?

- Common use in programming languages that do not intrinsically support other forms of synchronization.
- They are the primitive synchronization mechanism in many operating systems.

The trend in programming language development, though, is towards more structured forms of synchronization, such as monitors.

- Inadequacies in dealing with (multi-resource) deadlocks
- Do not protect the programmer from the easy mistakes of taking a semaphore that is already held by the same process, and forgetting to release a semaphore that has been taken.
Outline

1. Semaphores

2. Monitors
   - ADT
   - Mutual Exclusion
   - Condition Variables
   - Bounded Buffer
   - Readers/Writers

3. Conclusion
Monitor – an abstract data type

- Mutual exclusion is provided implicitly by ensuring that procedures in the same monitor are not executed concurrently
- Easier programming
  - call 
    
  - Designed in isolation
  - Maintain the *monitor invariant*

<table>
<thead>
<tr>
<th>Monitor _mname &gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulated data</td>
</tr>
<tr>
<td>Operations \text{opname}</td>
</tr>
</tbody>
</table>

Operating System

Programmer

Language

Library

Programmer

Language

Library
Mutual exclusion

Monitor procedures *by definition* execute with mutual exclusion
Condition Variables

- used to delay a process that cannot safely continue executing until the monitor’s state satisfies some boolean condition.
- used to awaken delayed processes when the condition becomes true.

```
cond cv;
```

- The value of the condition variable cv is a FIFO queue of delayed processes.
- Hidden to the programmer.
- Somebody is waiting? → Check empty(cv);
- A process blocks on a condition variable by executing wait(cv);
- signal(cv); awakens the front of the queue
Signaling and waking up...Dilemma

- **Signal and Continue** (non-preemptive)
  → The signaler continues and the signaled process executes at some later time.

- **Signal and Wait** (preemptive)
  → The signaler waits until some later time and the signaled process executes now.
Bounded buffer with Monitors

Producer

Buffer

Consumer

< Cond Variable >

Empty

Full

P₀ P₁ P₂ ...

empty

wait

signal

put

take
Bounded Buffer – Code

```plaintext
monitor Bounded_Buffer {

    typeT buf[n]
    int front=0, rear=0, count=0;  ▶ rear = (front + count) % n

    cond not_full,  ▶ signaled when count < n
        not_empty;  ▶ signaled when count > 0

    procedure put(typeT data) {
        while(count == n)wait(not_full);
        buf[rear] = data; rear=(rear+1)%n; count=count+1;
        signal(not_empty);
    }

    procedure take(typeT &result) {
        while(count == 0)wait(not_empty);
        result = buf[front]; front=(front+1)%n; count=count-1;
        signal(not_full);
    }

}
```
Readers/Writers

Shared Resource

< Grant access >

<table>
<thead>
<tr>
<th>R₀</th>
<th>R₁</th>
<th>R₂</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>W₀</td>
<td>W₁</td>
<td>W₂</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

empty
wait/signal
grant/release

request_read
request_write
release_read
release_write
Readers/Writers – Code

monitor RW_Controller {
  int nr=0, nw=0; ▶RW: \((nr == 0 \lor nw == 0) \land nw \leq 1\)
  cond oktoread, ▶signaled when \(nw == 0\)
  cond oktowrite, ▶signaled when \(nr == 0\) and \(nw == 0\)

  procedure request_read() {
    while(nw > 0) wait(oktoread);
    nr = nr + 1;
  }

  procedure release_read() {
    nr = nr - 1;
    if(nr == 0) signal(oktowrite); ▶awaken one writer
  }

  procedure request_write() {
    while(nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }

  procedure release_read() {
    nw = nw + 1;
    signal(oktowrite); ▶awaken one writer
    signal_all(oktoread); ▶and all readers
  }
}

OS2’08 | Semaphores and Monitors
Conclusion

- **Semaphore**
  - Fundamental
  - Easy to program mutual exclusion and signaling
  - Easy to make errors
  - Global to all processes:
    ⇒ Hard to understand the program

- **Monitors**
  - Data structure abstraction
  - Operations are the *only* means to manipulate data
  - Implicit mutual exclusion (*Not* the programmer’s task)
  - Condition variables (FIFO queue)
  - Awaking disciplines