Semaphores and Monitors

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Summer 2009
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 methods:

\[
\text{down}(\text{Semaphore } s) \begin{cases} \\
  \triangleright \text{Probeer (try) / Passeren (pass) / Pakken (grab)} \\
  \text{wait until } c > 0, \text{ then } c := c - 1; \triangleright \\
  \triangleright \text{must be atomic once } c > 0 \text{ is detected} \\
\end{cases}
\]

\[
\text{up}(\text{Semaphore } s) \begin{cases} \\
  \triangleright \text{Verhoog (increase)} \\
  \text{c = c + 1; } \triangleright \text{must be atomic} \\
\end{cases}
\]

\[
\text{Init}(\text{Semaphore } s, \text{ Integer } i) \begin{cases} \\
  c := i; \end{cases}
\]
Semaphores, what for?

- Critical section: Mutual exclusion
- Barriers: Signaling events
- Producers and Consumers
- Bounded buffers: Resource counting
sem mutex;    Init(sem,1);

while (true) {
    Critical section
    NON-Critical section
}

while (true) {
    Critical section
    NON-Critical section
}
Barriers: Signaling events

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

... ▶ process 1 in section A
▶ signal arrival
▶ Wait for the other process

... ▶ process 2 in section A
▶ signal arrival
▶ Wait for the other process

... ▶ process 1 in section B

... ▶ process 2 in section B
Semaphores: Producers and Consumers

- Producer
- Consumer
- Shared Resource
Semaphores: Producers and Consumers

- **Producer**
- **Consumer**
- **Buffer**

**Split Binary Semaphore**
- Both are *binary semaphores*
typeT buf;  \(\triangleright\) Buffer of some type T
sem sEntry, sExit;
Init(sEntry,1); Init(sExit,0);

while (true) {
  \(\triangleright\) Producer
  ...
  buf = data;
  ...
}

while (true) {
  \(\triangleright\) Consumer
  ...
  result = buf;
  ...
}
In the last example...

- Single communication buffer
- No waiting if data are produced+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
The Buffer

- **Put**: buf[rear] = data; rear = (rear + 1) % n
- **Take**: result = buf[front]; front = (front + 1) % n
Bounded buffer: Resource counting

typeT buf[n]; // Array of some type T

int front=0, rear=0;

sem sEntry, sExit;

Init(sEntry,n); Init(sExit,0);

sem mutexP, mutexT; Init(mutexP,1), Init(mutexT,1);

while (true) {
    down(sEntry);
    down(mutexP);
    buf[rear] = data;
    rear = (rear+1) %n;
    up(mutexP);
    up(sExit);
}

while (true) {
    down(sExit);
    down(mutexT);
    result = buf[front];
    front = (front+1) %n;
    up(mutexT);
    up(sEntry);
}

...
Semaphores

Semi-Conclusion

Critical Section

Blocking semaphore

Split Binary semaphore

Resource Counting

Barriers/Signaling

OS2'09 | Semaphores and Monitors
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 patten 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 none is able to acquire both forks – for example, each holds one fork and refuses to give it up.
Dining Philosophers Problem

while (true) {
    think;
    acquire forks;
    eat;
    release forks;
}

sem fork[5] = {1,1,1,1,1}
while (true) {
    think;
    down(fork[i]); down(fork[i+1]);
    eat;
    up(fork[i]); up(fork[i+1]);
}
Deadlocks: Resource Allocation Graph

Deadlock?
- No cycle $\Rightarrow$ No process is deadlocked
- If cycle, deadlock may exist
Cycle in the Graph?

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

If each resource has SEVERAL instance
- 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
Conditions

**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) {
    think;
    down(fork[i]); down(fork[i+1]);
    eat;
    up(fork[i]); up(fork[i+1]);
}

while (true) {
    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 \( 1, \ldots, M \)

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

Readers – as a group – need to lock out writers

but only the \textit{first} needs to grab the lock (i.e. \text{down}(rw))

Subsequent readers can directly access the database

Writers \( 1, \ldots, N \)

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

\[
\]
Relaxing constraints

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

- **Readers** $1, \ldots, M$

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

- **Writers** $1, \ldots, N$

```c
while (true) {
    . . .
    down(rw);  // grab exclusive access lock
    Write the database
    up(rw);  // release the lock
    . . .
}
```
Readers/Writers using Condition Synchronization

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 ∨ nw == 0) ∧ nw ≤ 1

Readers 1,...,M

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

Writers 1,...,N

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 mname.opname(args)
- Designed in isolation
- Maintain the
Mutual exclusion

Monitor procedures *by definition* execute with
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.

\[
\text{cond cv;}
\]

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

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

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

Producers

put

Buffers

Consumer

take

< Cond Variable >

Buffer

isEmpty

wait

signal

P₀ P₁ P₂ ... ...

Entry

In

Exit

0

P₀ P₁ P₂ ... ...
Bounded Buffer – Code

```c
monitor Bounded_Buffer {

    typeT buf[n]
    int front=0, rear=0, count=0; \( \triangleright \) \( rear = (front+count) \mod n \)

    cond not_full, \( \triangleright \) signaled when count < n
    not_empty; \( \triangleright \) signaled when count > 0

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

    procedure take(typeT &result) {
        while(count == 0)wait(not_empty);
        result = buf[front]; front=(front+1)\mod n; count=count-1;
        signal(not_full);
    }
}
```
Readers/Writers

Shared Resource

- Request_read
- Request_write
- Release_read
- Release_write

isEmpty
wait/signal
grant/release
Readers/Writers – Code

monitor RW_Controller {

    int nr=0, nw=0; \(\triangleright RW: (nr == 0 \lor nw == 0) \land nw \leq 1\)
    cond oktoread, \(\triangleright \text{signaled when } nw == 0\)
    cond oktowrite, \(\triangleright \text{signaled when } nr == 0 \land nw == 0\)

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

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

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

    procedure release_write() {
        nw = nw - 1;
        signal(oktowrite); \(\triangleright \text{awaken one writer}\)
        signal_all(oktoread); \(\triangleright \text{and all readers}\)
    }
}

OS2'09 | Semaphores and Monitors
Conclusion

- Semaphore
  - Fundamental
  - Easy to program mutual exclusion and signaling
  - Easy to make errors
  - Global to all processes:
    \( \Rightarrow \) 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