IV. Process Synchronisation

Operating Systems

Stefan Klinger

Database & Information Systems Group
University of Konstanz

Summer Term 2009

Background

Multiprogramming – Multiple processes are executed asynchronously.

- Concurrent access to shared resources may result in inconsistency.
  - One printer, many processes that want to print.
    $\Rightarrow$ Resource allocation
  - Access to shared memory.
    $\Rightarrow$ Mutual exclusion
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

$\Rightarrow$ **Process Synchronisation**

Race Condition / Critical Section

Assume a shared integer c, initialised to 42, and one process wanting to increase, the other to decrease the value.

```
P1 {                                         P2 {
  shared int c;                                shared int c;
  c++;                                         c--;
}                                            }
```

Implementation of c++, and c--:

```
  load &c reg1                               load &c reg2
  inc reg1                                   dec reg2
  store reg1 &c                              store reg2 &c
```

Interleaved operation of processes P1 and P2:

```
P1: load &c reg1    →  reg1 = 42
P1: inc reg1        →  reg1 = 43
P2: load &c reg2    →  reg2 = 42
P2: dec reg2        →  reg2 = 41
P2: store reg2 &c   →  c = 41
P1: store reg1 &c   →  c = 43
```

Critical Section / Requirements

```
P1 {                      P2 {
  shared int c;             shared int c;
  c++;                      c--;
}                         }
```

- **Mutual Exclusion** – If a process is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
  - Each process executes at a nonzero speed.
  - No assumption concerning the relative speed of processes.

Synchronisation Hardware

Many systems provide hardware support for critical section code.

- Uniprocessors – could disable interrupts.
    - Currently running code would execute without preemption.
    - Inefficient on multi-processor systems.
    - Not broadly scalable.
- Modern machines provide special atomic hardware instructions

Atomic = non-interruptable

$\Rightarrow$ Test-and-Set, Swap.

TestAndSet Instruction

- Definition:

```
boolean TestAndSet(boolean *target) ATOMIC {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
```

- Solution, `shared` `int lock` is initialised to `FALSE`:

```
...
while ( TestAndSet(&lock) ) ; // do nothing
... // critical section
lock = FALSE;
...
```

Swap Instruction

- Definition:

```
void Swap(boolean *a, boolean *b) ATOMIC {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

- Solution, `shared` `int lock` is initialised to `FALSE`:

```
int key = TRUE;
...
while (key == TRUE) Swap(&lock, &key);
... // critical section
lock = FALSE;
...
```

Semaphore

- Synchronisation tool provided by the operating system.
- May be implemented using TestAndSet or Swap.
- Does not require busy waiting.

Definition: A semaphore is a *shared integer* variable that *cannot drop below zero*.

- `Semaphore semaphore(Name name, unsigned int v)`
  - Returns the semaphore identified by `name`.
  - Creates & initialises semaphore to `v` if it **did not** already exist.
- `void post(Semaphore s)`
  - Increments the semaphore.
- `void wait(Semaphore s)`
  - Blocks until the semaphore is greater than 0,
  - then decrements the semaphore.

All three operations **atomically** manipulate semaphores.

Semaphore (continued)

- Usage

```
Semaphore s = semaphore("mutex", 1);
...
wait(s);
... // critical section
post(s);
...
```

- Common names

Edsger W. Dijkstra	*P proberen*	*V verhogen*
We use	`wait`	`post`
POSIX	`sem_wait`	`sem_post`
Silberschatz	`wait`	`signal`
Tanenbaum	`down`	`up`
Java	`acquire`	`release`

- Kinds
  - **Counting semaphore** – as def. above.
  - **Mutex lock** – *binary semaphore* – counting range is $[0, 1]$

Classical Problem: Bounded-Buffer

a.k.a. Producer-Consumer Problem

Definition:

- n shared **buffers**, each can hold one item,
- **writer** process produces items,
- **reader** process consumes items.

Utilise semaphores to synchronise a reader and a writer process.

Classical Problem: Bounded-Buffer
(continued)

Solution:

- Semaphore `mutex` initialised to the value 1,
- Semaphore `full` initialised to the value 0,
- Semaphore `empty` initialised to the value *n*.

Producer process

```
...
while (1) {
  // produce one item
  wait(empty);
  wait(mutex);
  // store the item
  post(mutex);
  post(full);
}
```

Consumer process

```
...
while (1) {
  wait(full);
  wait(mutex);
  // remove one item
  post(mutex);
  post(empty);
  // consume the item
}
```

Semaphore Implementation: Spinlock

- Must guarantee atomic access to semaphore data.
- Implementation of semaphore interface becomes the critical section problem.

```
void wait(Semaphore s) {
  s--;
  while (s < 0) ; // do nothing
}
void post(Semaphore s) {
  s++;
}
```

- Little busy waiting if critical section rarely occupied.
- Applications may spend lots of time waiting.

Semaphore Implementation without Busy Waiting

Teach scheduler about semaphores:

- Associate waiting queue `s.queue` with each semaphore `s`.
- `void block(Queue q)` – Descheduled calling process and add it to q.
- `void wakeup(Queue q)` – Move one process from q to the scheduler's ready-queue.

```
void wait(Semaphore s) {
  s.val--;
  if (s.val < 0)
    block(s.queue);
}
```

```
void post(Semaphore s) {
  s.val++;
  if (s.val >= 0)
    wakeup(s.queue);
}
```

Only few critical sections remain in semaphore implementation.
⇒ Disable interrupts or use hardware support here.

Semaphore Implementation: Adaptive Mutex

On multi-processor machines an adaptive mutex starts as semaphore implemented as spinlock.

If protected data is locked:

- If thread holding the lock is running on other CPU:
  $\Rightarrow$ Lock likely to be released soon. Keep spinning.
- If thread holding the lock is not running:
  $\Rightarrow$ This might take some time. Goto sleep, avoid busy waiting.

On single-processor machines the other threads always sleep.
$\Rightarrow$ goto sleep.

Deadlock and Starvation

Bad design of synchronisation scheme:

- **Deadlock** – Processes mutually block each other:

```
P1 {                        P2 {
  ...                         ...
  wait(s1);                   wait(s2);
  wait(s2);                   wait(s1);
  ...                         ...
}                           }
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Other problems:

- Error prone.
- Hard to reason about synchronisation scheme.

Monitors

```
monitor Name {
  type var1, var2...;
  Condition c, ...;

  type1 p1(...)
    { ... }
  ...
  typen pn(...)
    { ... }

  Initialisation(...)
    { ... }
}
```

Language construct

- High-level abstraction.
- Provides a convenient and effective mechanism for process synchronization.
- Only one process may be active within the monitor at a time.

Cpecial condition variables:

- `wait(c)` suspends caller.
- `signal(c)` resumes caller.

- Queues manage waiting processes.
- Implementation with semaphores possible.
- **Note:** Implementations with different characteristics: What happens after `signal()` to the caller?

Next level: Transactions

- Known from **database** lectures.
- A transaction is a series of read and write operations. Happens as a **single logical unit of work**, in its entirety, or not at all.
- Assures atomicity despite computer system failures.
- Terminated by
  - commit (transaction successful), or
  - abort (transaction failed) operation.
    Aborted transaction must be rolled back to undo any changes it performed.

$\Rightarrow$ **ACID** properties: Atomic, Consistent, Isolated, and Durable.

Classical Problems

- Bounded-Buffer Problem *a.k.a.* Producer-Consumer Problem
- Readers and Writers Problem
- Dining Philosophers Problem
- Baboon Crossing Problem
- Cigarette Smokers Problem
- ...

Solutions using semaphores:
Allen B. Downey. *The Little Book of Semaphores.*
`http://greenteapress.com/semaphores/`

Classical Problem: Readers-Writers

Definition:

- Given one one shared **buffer**,
- allow **concurrent access by readers**, and
- guarantee **exclusive accesy by writers**.

Utilise semaphores to synchronise reader and writer processes.

Classical Problem: Readers-Writers

Semaphore Pattern: Lightswitch

Solution:

- Shared integer `readers` initialised to 0,
- semaphore `mutex` initialised to 1 protects `readers`,
- semaphore `writable` initialised 1 protects buffer.

The reader process:

```
wait(mutex);
readers++;
if (readers == 1) wait(writable);
post(mutex);
// read buffer
wait(mutex);
readers--;
if (readers == 0) post(writable);
post(mutex);
```

The writer process:

```
wait(writable);
// write buffer
post(writable);
```

Semaphore Pattern: Turnstile

- Use a mutex:
  ```
  Semaphore t = semaphore("turnstile", 1);
  ```
- The turnstile is
  ```
  wait(t); post(t);
  ```
  Only one process can pass the turnstile at the same time.
- A process can lock others from passing the turnstile:
  ```
  wait(t);
  ```
- A process can unlock a turnstile:
  ```
  post(t);
  ```