Locks and Barriers

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Summer 2008
Scenario

Several cars want to drive from point A to point B.

Sequential Programming
They can compete for space on the same road and end up either:
- following each other
- or competing for positions (and having accidents!).

Parallel Programming
Or they could drive in parallel lanes, thus arriving at about the same time without getting in each other's way.

Distributed Programming
Or they could travel different routes, using separate roads.
Communication
- Reading and Writing shared variables
- Sending and Receiving messages

\[ \text{Communication} \Rightarrow \text{Synchronisation} \]

Synchronisation
- Mutual Exclusion
- Condition synchronisation
Cache coherency

Shared Memory

Thread

Read A
Read A
... 
... 
Read A

$ 

Thread

Read A
Read A
... 
... 
Read A

$ 

Thread

Read B
... 
Read A

- $
Memory ordering

Thread 1

LD A
ST B'
LD C
ST D'
LD E
...
...

Thread 2

ST A'
LD B'
ST C'
LD D
ST E'
...
...

Thread 1

LD A
ST B'
LD C
...
...

Thread 2

ST A'
LD B'
ST C'
LD D
ST D'
...
LD E
ST E'
...

Memory model

Memory model flavors

- **Sequentially Consistent**: Programmer’s intuition
- **Total Store Order**: Almost Programmer’s intuition
- **Weak/Release Consistency**: No guaranty

Memory model is tricky
Dekker’s algorithm, in general

Initially $A = 0, B = 0$

“fork”

$A := 1$
if($B == 0$)print(“A wins”);

$B := 1$
if($A == 0$)print(“B wins”);

Can both $A$ and $B$ win?

The answer depends on the memory model

“Contract” between the HW and SW developers.
Let’s illustrate it – Demo

Log onto simba.it.uu.se

Simba is a 16 processors partition on a Ngorongoro machine.

Ngorongoro = Sun Fire 15k

- 48 UltraSPARC III+ CPUs at 900 MHz
- 8 MB of L2 Cache memory
- 48 GB of primary RAM
- about 3.4 TB of disk storage

- ssh -l <username> simba.it.uu.se
  or ssh <username>@simba.it.uu.se or ssh -Y <username>@simba.it.uu.se
Demo – on Dekker’s algorithm

```
int data = 0; ▶ Shared variable
int n = ...; ▶ Iterations counter

▶ process 1 increments

int a; ▶ local copy
for n iterations {
    a=data;
    a++;
    data=a;
}

▶ process 2 decrements

int b; ▶ local copy
for n iterations {
    b=data;
    b--;  
data=b;
}

if(data==0){print('No problem');}
else{print('Eh??');}
```
Demo – Adding locks

- Declaration: `pthread_mutex_t my_lock;`
- Initialization: `pthread_mutex_init(&my_lock, NULL);`
- Locking: `pthread_mutex_lock(&my_lock);`
- Unlocking: `pthread_mutex_unlock(&my_lock);`
What about the Compiler?

Usage of the *volatile* keyword in C.

```c
int data = 0; // Shared variable

for n iterations {
  data++;
}

for 20 times {
  if (data==0) {
    print("No changes");
  } else {
    print("I saw one");
  }
}
```
Locks How do we make a thread wait?

A solution: Busy waiting

Check repeatedly a condition until it becomes true.

- Virtue: We can implement it using only the machine instructions available on modern processors
- Inefficient for single-processors but powerful for multi-proc
- Even hardware uses busy-waiting (ex: synch of data transfers on memory busses)

Another solution: Blocking

- Waiting threads are de-scheduled
- High overhead
- Allows processor to do other things

Hybrid methods: Busy-wait a while, then block
What for?

Critical Section Problem

```plaintext
LOCK(bank_account) ▶Wait for your turn
if (sum_to_withdraw > account_balance) {
    account_balance = account_balance - sum_to_withdraw;
}
UNLOCK(bank_account) ▶Release the lock
```

Critical Section Problem – Correct?

```plaintext
if (sum_to_withdraw > account_balance) {
    LOCK(bank_account) ▶Wait for your turn
    account_balance = account_balance - sum_to_withdraw;
    UNLOCK(bank_account) ▶Release the lock
}
```
Critical Section

\[ \text{CO} \ [\text{Process } i = 1 \text{ to } n] \{ \]
\[ \text{while (true) } \{ \]
\[ \text{LOCK(resource);} \]
\[ \text{Do critical section work; (using that resource)} \]
\[ \text{UNLOCK(resource);} \]
\[ \text{Do NON-critical section work;} \]
\[ \}
\]

**Assumption**

A process that enters its CS will eventually exit
⇒ A process may only terminate in its NON-critical section
Challenge

Task
Design the LOCK and UNLOCK routines.

Ensuring:
- Mutual Exclusion
- No deadlocks
- No unnecessary delays
- Eventual Entry
LOCK / UNLOCK must ensure:

**Mutual Exclusion**
At most one process at a time is executing its CS.
*Bad state:* 2 processes are in their CS.

**No deadlocks**
If two or more processes are trying to enter their CSs, at least one will succeed.
*Bad state:* all processes are waiting to enter their CS, but none is able to.

**No unnecessary delays**
If a process is trying to enter its CS and the other processes are executing their non-CSs or have terminated, the first process is not prevented from entering its CS.
*Bad state:* A process that wants to enter cannot do so, even though no other process is in its CS.

**Eventual Entry**
A process that is attempting to enter its CS will eventually succeed.
Reformulation

- Let in1 and in2 be boolean variables.
  - in1 is true if Process 1 is in its CS, false otherwise
  - in2 is true if Process 2 is in its CS, false otherwise
- Avoid that both in1 and in2 are true

**MUTEX:** \( \neg (in1 \land in2) \)

**A solution:**

\( wait \_ until(!in2) \) and then in1 = true; //ATOMICALLY!!

\(< wait \_ until(!in2) \) and then in1 = true;>
Recall Demo Locks
Barriers

Coarse-grained solution

```cpp
bool in1 = false, in2 = false;

▷ MUTEX: \( \neg (in1 \land in2) \)

▷ Process 1
while (true) {
    < wait_until(!in2) and then
    in1 = true;
    Do critical section work
    in1 = false;
    Do NON-critical section
}

▷ Process 2
while (true) {
    < wait_until(!in1) and then
    in2 = true;
    Do critical section work
    in2 = false;
    Do NON-critical section
}

But \( n \) processes ⇒ \( n \) variables...
```
Coarse-grained solution

Only 2 interesting states: locked and unlocked
⇒ 1 variable is enough

bool lock = false;

while (true) {
  \( \text{Process 1} \)
  < wait\_until(!lock) and then
  lock = true;
  Do critical section work
  lock = false;
  Do NON-critical section
}

while (true) {
  \( \text{Process 2} \)
  < wait\_until(!lock) and then
  lock = true;
  Do critical section work
  lock = false;
  Do NON-critical section
}
How to?

```< await(!lock) and then lock = true; >
```

Read-Modify-Write atomic primitives

- **Test and Set (TAS):**
  Value at Mem[lock_addr] loaded in a specified register.
  Constant “1” atomically stored into Mem[lock_addr]
- **Swap:**
  Atomically swaps the value of REG with Mem[lock_addr]
- **Compare and Swap (CAS):**
  Swaps if Mem[lock_addr] == REG2
- **Fetch and Add (FA):**
  Increments a value by a given constant and returns the old value
bool TAS(bool lock) {
    bool initial = lock;  // Save the initial value
    lock = true;          // Set lock
    return initial;       // Return initial value
}
bool TAS(bool lock) {
  bool initial = lock;  // Save the initial value
  lock = true;        // Set lock
  return initial;     // Return initial value
}

lock(lock_variable) {
  while(TAS(lock_variable)==true){};
  // Bang on the lock until free
}

unlock(lock_variable) {
  lock_variable:=false; // Reset to the initial value
}
Handing over the lock

High traffic at handover. Even worse with TAS (N writes)
Test and Test and Set

lock(lock_variable) {  
    while(TAS(lock_variable) == true) {};  
}

lock(lock_variable) {  ▶ More optimistic solution  
    while (true) {  
        if(TAS(lock_variable) == false) break;  
        ▶ Bang on the lock once  
        while(lock_variable == true) {};  
    }  
}

Less traffic for coherence, but still a lot at handover
lock(lock_variable) {
    while (true) {
        if(TAS(lock_variable)==false) break; //Bang on the lock once
        while(lock_variable==true){};
    }
}

Can the same thread
- succeed to grab the lock
- perform its critical section
- release the lock
- perform its non-critical section
- and race back to grab the lock again?
Tie Breaker – Petersson’s algorithm

Remember who had the lock latest!

```c
bool in1 = false, in2 = false;
int last = ?;
```

▷ **Process 1**

```c
while (true) {
    in1=true, last = 1;
    while(in2 and last==1){};
    Do critical section work
    in1=false;
    Do NON-critical section work
}
```

▷ **Process 2**

```c
while (true) {
    in2=true, last = 2;
    while(in1 and last==2){};
    Do critical section work
    in2=false;
    Do NON-critical section work
}
Lower traffic at handover

Queue-based locks
Traditional chart for lock performance on a NUMA machine (round-robin scheduling)

Benchmark:

```c
for i = 1 to 10000 {
    lock(L);
    A = A + 1;
    unlock(L);
}
```
Ticket-based lock

C0 [Process $i = 1$ to $n$] {  
  while (true) {  
    < turn[i] = number; number = number+1; >  
    < await(turn[i] == number); >  
    Do critical section  
    < next = next+1; >  
    Do NON-critical section  
  }  
}

Fetch and Add (FA)

Increments a value by a given constant and returns the old value

C0 [Process $i = 1$ to $n$] {  
  while (true) {  
    turn[i] = FA(number,1);  
    while(turn[i] != next){}; ▷ Can even have a back-off  
    Do critical section  
    next = next+1; ▷ Is that safe?  
    Do NON-critical section  
  }  
}
Barriers

Barrier synchronisation

CO \[ \text{Process } i = 1 \text{ to } n \]  
\{ 
  \text{while (true) \{ 
    code for task } i 
    \rightarrow \text{ wait for all } n \text{ tasks to complete} \leftarrow 
  \} 
\}

Definition (A barrier)

coordination mechanism (an algorithm) that forces processes which participate in a concurrent (or distributed) algorithm to wait until each one of them has reached a certain point in its program. The collection of these coordination points is called the barrier. Once all the processes have reached the barrier, they are all permitted to continue past the barrier.
Halt!... Papier, bitte...
Using barriers, often, enables significant simplification of design for concurrent programs.

The programmer may design an algorithm under the assumption that the algorithm should work correctly only when it executes in a *synchronous* environment (where processes run at the same speed or share a global clock).

Then by using barriers for synchronisation, the algorithm can be adapted to work also in an *asynchronous* environment.
Wish: employ local spinning in order to minimize traffic.

On cache-coherent system, local spinning if:

- busy-waits only on locally-cached data
- stops waiting when the data on which it spins change
Atomic counter

- Counter initially set to 0
- As soon as a process reaches the barrier,
  - `counter = counter + 1;`  
  - busy-waits
- when `counter = n`
  - the last process to increment the counter signals the other processes that they may continue to run past the barrier
  - resets to 0 the value of `counter` (← reusable)

Waiting and signaling work on a single bit go. The last process flips the bit.
Atomic counter

- **shared** counter
  - Initially 0, Ranges over \{0,\ldots,n\}
- **shared** go
  - Atomic bit
- **local** local.go
  - A bit

local.go = go;  
\textit{remembers the current value}

\langle \text{counter} = \text{counter} + 1; \rangle  
\textit{atomically increment the counter}

if ( \text{counter} == n ) {
  \text{counter} = 0;  
  \textit{reset}
  \text{go} = 1 - \text{go};  
  \textit{notify all}
} else {
  \textbf{while}(\text{local.go} == \text{go}){\}};  
  \textit{not the last}
}
Atomic counter – a bit better

- **shared** counter
  - Initially 0, Ranges over \{0,...,n\}
- **shared** go
  - Atomic bit, initially 1
- **local** local.go
  - A bit, initially 1

\[
\text{local.go} = 1 - \text{local.go};
\]
- toggle its local bit

\[
< \text{counter} = \text{counter} + 1; >
\]
- atomically increment the counter

\[
\text{if} \ ( \text{counter} == n ) \ { \}
\]
- last to arrive at the barrier
  - counter = 0;
  - reset
  - go = local.go;
  - notify all

\[
\text{else} \ { \}
\]
- not the last
  - while(local.go \neq go){};
Atomic counter – Local spinning

shared counter
shared go[1..n]
local local.go

local.go = go[i];  //remembers current value
< counter = counter + 1; >  //atomically increment the counter
if ( counter == n ) {
    counter = 0;  //reset
    for (j = 1 to n) {
        go[j] = 1 - go[j];  //toggling all bits
    }
} else {
    while(local.go == go[i]){};  //not the last
}
Atomic counter
Without memory initialization

shared counter

shared go

local local.go

local local.counter

\[\text{local.go} = \text{go};\]
\[\text{local.counter} = \text{counter};\]
\[\langle\text{counter} = \text{counter}+1;[n]\rangle\]
repeat {
  if( counter == local.counter )
  then { go = 1-go; }
} until( local.go != go);

Who toggles the go bit?
Atomic counter – Exercise – Correct?

shared counter
shared go
local local.go

local.go = go;
< counter = counter + 1; >
if ( counter == n ) {
    go = 1 - go;
    counter = 0;
} else {
    while(local.go == go){};
}

\(\text{Initially 0, Ranges over } \{0, \ldots, n\}\)
\(\text{Atomic bit}\)
\(\text{A bit}\)

\(\text{remembers the current value}\)
\(\text{atomically increment the counter}\)
\(\text{last to arrive at the barrier}\)
\(\text{notify all}\)
\(\text{reset}\)
\(\text{not the last}\)
Outline

1. Recall
2. Demonstration
3. Locks
4. Barriers
   - Strategies
   - Performance improvement through parallelization
...to multicores

Past
- Minimize communication between processors
- Maximize scalability (thousands of CPUs)

Multicores today
- Communication is “for free”
- Scalability is limited to 32 threads
- The caches are tiny
- Memory bandwidth is scarce

⇒ Data locality is the key!!
Case Study: Gauss-Seidel

Poisson’s equation

\[
\Delta \varphi = f, \quad \text{in } \Omega \\
\varphi = 0, \quad \text{in } \partial \Omega
\]

In 2D cartesian coordinates,

\[
\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \varphi(x, y) = f(x, y), \quad (x, y) \in \Omega \\
\varphi(x, y) = 0, \quad (x, y) \in \partial \Omega
\]

Used in fluid theory, electrostatics, ...
Discretization

$$\nabla^2 u_{i,j} \leftarrow \frac{u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{5}$$

$$\nabla^2 + \frac{\partial^2}{\partial y^2}$$
Discretization – Gauss-Seidel

Thread\(_0\)       Thread\(_1\)       Thread\(_2\)

Cache line

\(\Omega\)
Sequential Sweep
while ( not converged ) {
    Do a sweep;
}

while ( ||M_{new} − M_{old}|| > \epsilon ) {
    M_{old} = M_{new};
    M_{new} = \text{SWEEP}(M_{new});
}

But we simplify: Just do 20 sweeps!
Parallel Sweep
Barrier strategy – Not reausable

Shared counter

\[ \text{CO [Process } i = 1 \text{ to } n] \{ \]

- Code to implement task \( i \)
- \( < \text{count} = \text{count} + 1; > \)
- \( < \text{await} (\text{count} == n); > \)

\[ \}
\]

\[ \text{FA(count,1); } \quad \text{▷If no FA, use count++ and mutex} \]

\[ \text{while(count != n) \{} \]

Flag

\[ \text{row_done[t]=line; } \quad \text{▷Safe, since only one writer} \]

Problem: reset the counter for the barrier

Solution: throw away that counter and use another fresh one at the beginning of each sweep: counter[iter]