Locks and Barriers

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems

Uppsala University

Summer 2009

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.

What do you remember from ... yesterday?

Communication

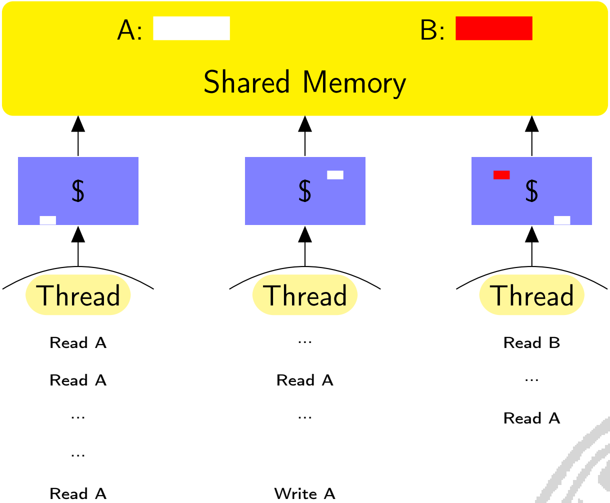
- Reading and Writing shared variables
- Sending and Receiving messages

Communication $\Rightarrow$ Synchronisation

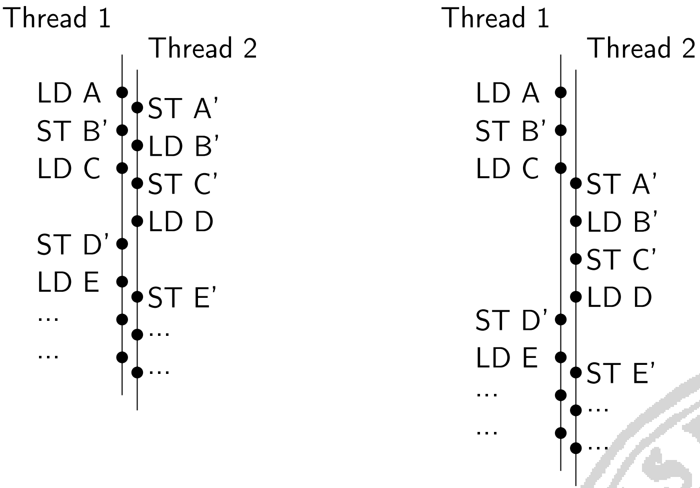
Synchronisation

- Mutual Exclusion
- Condition synchronisation

Cache coherency



Memory ordering



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.

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 keyword in 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:

Check repeatedly a condition until it becomes true.

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

Another solution:

- 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

```
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?

```
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

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

Assumption

A process that enters its CS will eventually exit
$\Rightarrow$ 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: $\neg(\texttt{in1} \wedge \texttt{in2})$

A solution:
wait_until(!`in2`) and then `in1` = `true`; //ATOMICALLY!!
<*wait_until*(!`in2`) and then `in1` = `true`;>

Coarse-grained solution

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

▷*MUTEX:* $\neg(in1 \wedge in2)$

▷*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 $\Rightarrow$ n variables...

Coarse-grained solution

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

```
bool lock = false;
```

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

while (true) { ▷*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

- (TAS):
  Value at Mem[lock_addr] loaded in a specified register.
  Constant "1" atomically stored into Mem[lock_addr]
- :
  Atomically swaps the value of REG with Mem[lock_addr]
- (CAS):
  Swaps if Mem[lock_addr]==REG2
- (FA):
  Increments a value by a given constant and returns the old value

Test And Set

```
bool TAS(bool lock)  {
    <                                        ▷Save the initial value
                                              ▷Set lock
                                   >         ▷Return initial value
}
```

Spin Lock

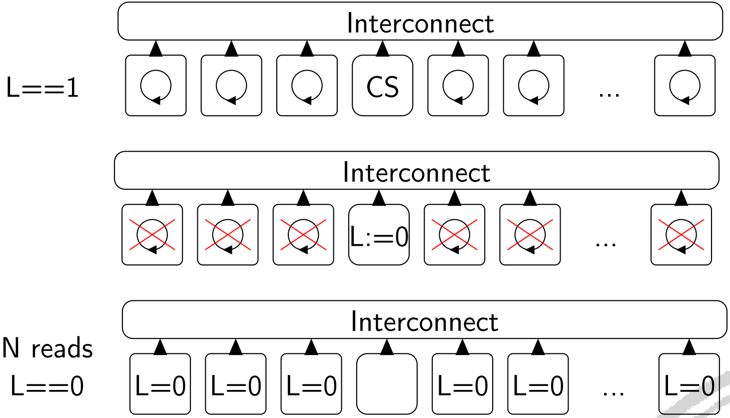
```
bool TAS(bool lock)  {
    < bool initial = lock;      ▷Save the initial value
      lock = true;              ▷Set lock
      return initial; >         ▷Return initial value
}
```

```
lock(lock_variable)  {

        ▷Bang on the lock until free
}
```

```
unlock(lock_variable)  {
                                ▷Reset to the initial value
}
```

Handing over the lock



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){};
    }
}
```

for coherence, but still a lot at handover

Fair solution?

```
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!

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

▷ *Process 1*

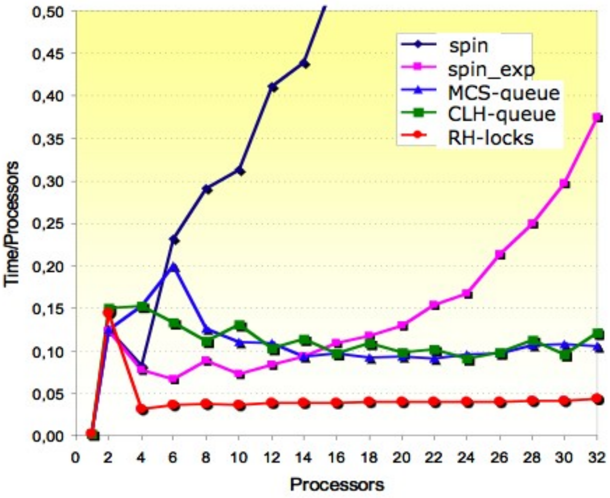
```
while (true) {
   in1=true, last = 1;
   while(in2 and last==1){};
   Do critical section work
   in1=false;
   Do NON-critical section work
}
```

▷ *Process 2*

```
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

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



Benchmark:

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

Ticket-based lock

```
CO [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

```
CO [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

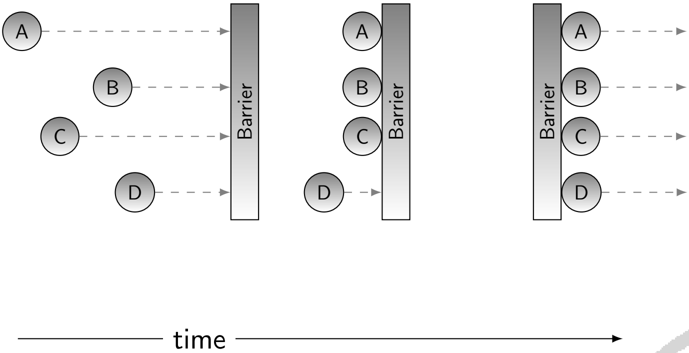
Barrier synchronisation

CO [Process $i = 1$ to n] {
    while (`true`) {
        code for task i
        $\hookrightarrow$ wait for all n tasks to complete $\hookleftarrow$
    }
}

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. . .



Why?

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 an *asynchronous* environment.

How?

Reusable barrier

Wish: employ in order to

On 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,...,n}
shared go                           ▷Atomic bit
local   local.go                    ▷A bit

local.go = go;                      ▷remembers the current value
< counter = counter + 1; >          ▷atomically increment the counter
if ( counter == n ) {   ▷last to arrive at the barrier
    counter = 0;                    ▷reset
    go = 1 - go;                    ▷notify all
} else {
    while(local.go == go){};   ▷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

local.go = 1 - local.go;                ▷toggle its local bit
< counter = counter + 1; >              ▷atomically increment the counter
if ( counter == n ) {                   ▷last to arrive at the barrier
    counter = 0;                        ▷reset
    go = local.go;                      ▷notify all
} else {
    while(local.go ≠ go){};             ▷not the last
}
```

Atomic counter – Local spinning

```
shared counter                          ▷Initially 0, Ranges over {0,...,n}
shared go[1..n]                         ▷array of atomic bit
local  local.go                         ▷A bit

local.go = go[i];                       ▷remembers current value
< counter = counter + 1; >              ▷atomically increment the counter
if ( counter == n ) {                   ▷last to arrive at the barrier
    counter = 0;                        ▷reset
    for (j = 1 to n) {                  ▷notify all
        go[j] = 1 - go[j];              ▷toggling all bits
    }
} else {
    while(local.go == go[i]){};         ▷not the last
}
```

Atomic counter

Without memory initialization

```
shared counter                            ▷Ranges over {0,...,n-1}
shared go                                 ▷Atomic bit
local  local.go                           ▷A bit
local  local.counter                      ▷Atomic register

local.go = go;                            ▷remembers current value
local.counter = counter;                  ▷remembers current value
<counter = counter+1;[n]>                 ▷atomically increment mod n
repeat {
  if( counter == local.counter )          ▷all processes have arrived
  then { go = 1-go; }                     ▷notify all
} until( local.go != go);
```

Who toggles the *go* bit?

Atomic counter – Exercise – Correct?

```
shared counter                         ▷Initially 0, Ranges over {0,...,n}
shared go                              ▷Atomic bit
local   local.go                       ▷A bit

local.go = go;                         ▷remembers the current value
< counter = counter + 1; >             ▷atomically increment the counter
if ( counter == n ) {                  ▷last to arrive at the barrier
    go = 1 - go;                       ▷notify all
    counter = 0;                       ▷reset
} else {
    while(local.go == go){};           ▷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

$\Rightarrow$ is the key!!

Case Study: Gauss-Seidel

Poisson's equation

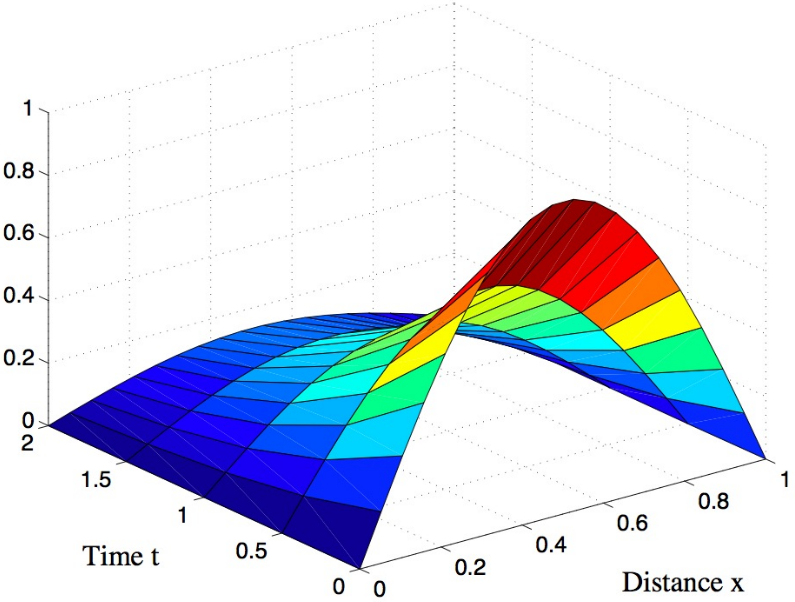
$$\begin{aligned} \Delta\varphi &= f, && \text{in } \Omega \\ \varphi &= 0, && \text{in } \partial\Omega \end{aligned}$$

In 2D cartesian coordinates,

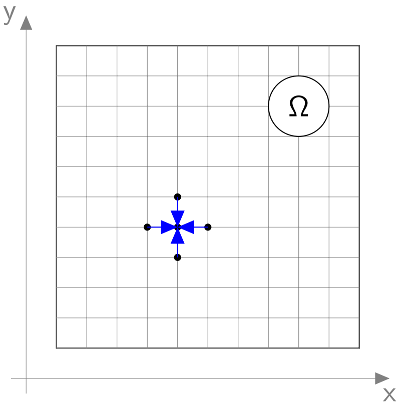
$$\begin{aligned} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) \quad \varphi(x,y) &= f(x,y), && (x,y) \in \Omega \\ \varphi(x,y) &= 0, && (x,y) \in \partial\Omega \end{aligned}$$

Used in fluid theory, electrostatics, ...

▸ To the lab


1
0.8
0.6
0.4
0.2
0
2
1.5
1
0.5
0
Time t
0
0.2
0.4
0.6
0.8
1
Distance x

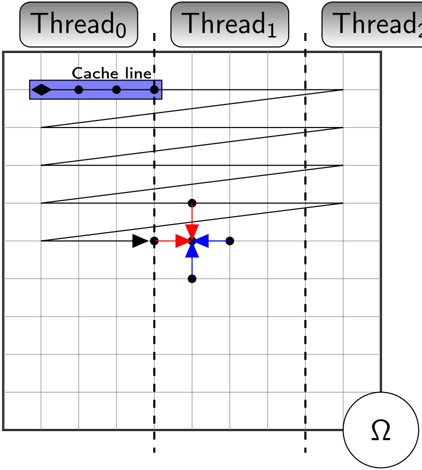
Discretization



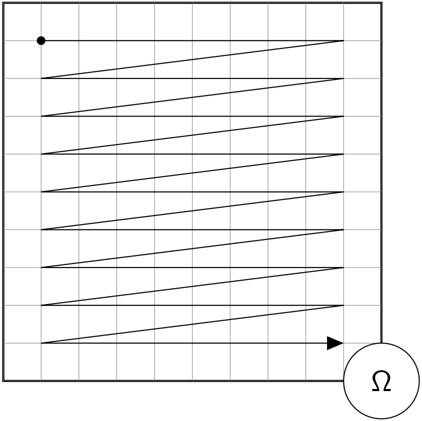
$$\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

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

Discretization – Gauss-Seidel



Sequential Sweep



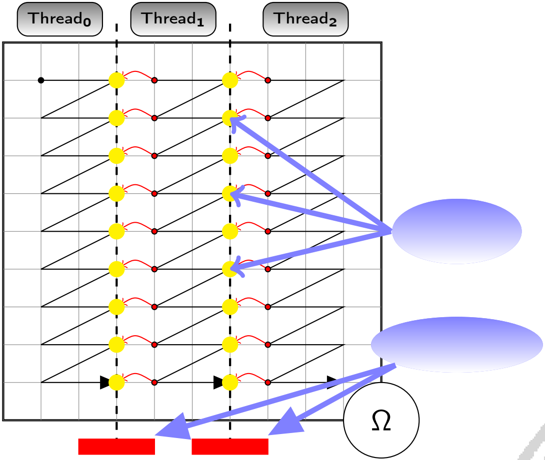
Convergence

while (not converged) {
  Do a sweep;
}

while $(\|M_{new} - M_{old}\| > \epsilon)$ {
  $M_{old} = M_{new}$;
  M_{new} =SWEEP(M_{new});
}

But we simplify: Just do 20 sweeps!

Parallel Sweep



Barrier strategy – Not reausable

Shared counter

```
CO [Process i = 1 to n]  {
      Code to implement task i
      < count = count + 1; >
      < await(count == n); >
}
```

```
FA(count,1);   ▷If no FA, use count++ and mutex
while(count != n ){};
```

Flag

```
row_done[t]=line;   ▷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]