Operating Systems

Thread Synchronization: Implementation

Thomas Ropars

thomas.ropars@imag.fr

Équipe ERODS – LIG/IM2AG/UJF

2015

References

The content of these lectures is inspired by:

- The lecture notes of Prof. André Schiper.
- The lecture notes of Prof. David Mazières.
- The lectures notes of Arnaud Legrand.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau
- *A Primer on Memory Consistency and Cache Coherence* by D. Sorin, M. Hill and D. Wood.

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

Agenda

Consistency and Coherence

Memory models

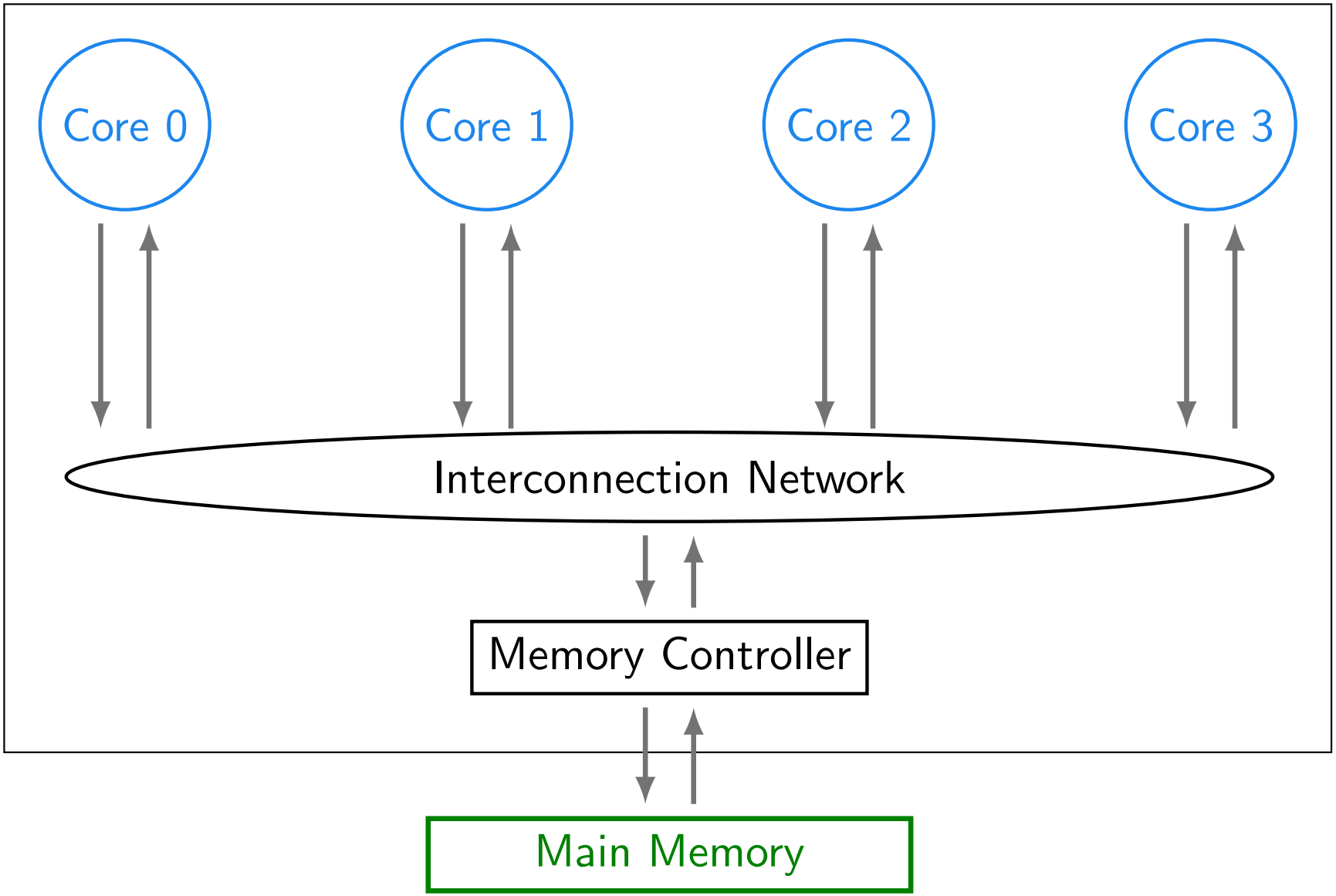
Data Races

Atomic Operations

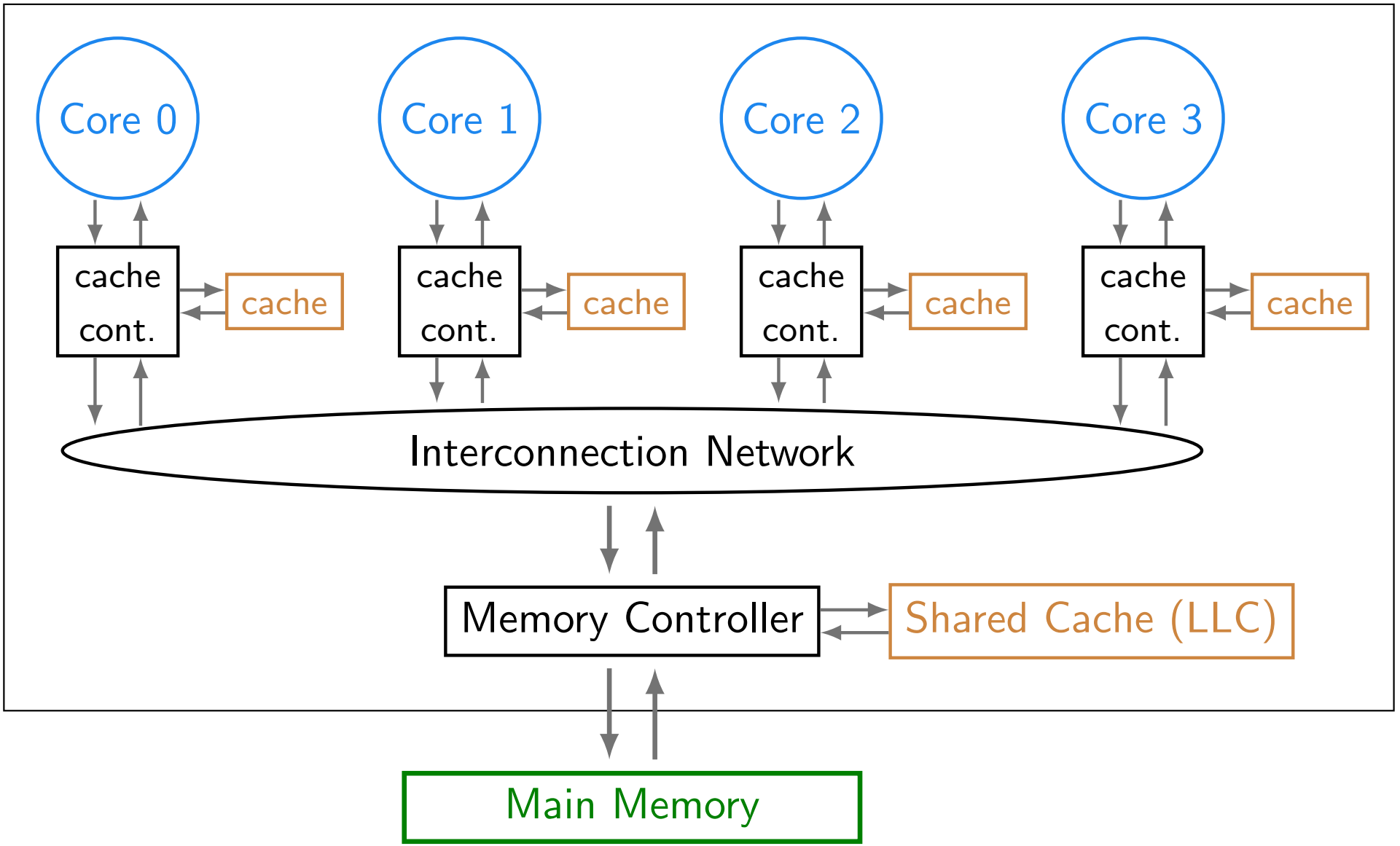
Spinlock

Sleeping Locks

A modern multicore processor



A modern multicore processor with caches



A modern multicore processor with caches

Cache

- Goal: Reducing memory access latency
- Mandatory for good performance
- Stores a copy of the last accessed cache lines
- Multiple level of private cache + a larger shared cache

Network interconnect

- Legacy architectures based on a bus
- Modern CPUs use a multi-dimensional network (mesh)

Memory system properties

Basic properties

- Each core read (load) and write (store) to a single shared address space.
- Size of loads and stores is typically 1 to 8 bytes on 64-bit systems.
- Cache blocks size is 64 bytes on common architectures (cache line).

Questions

- How is *correctness* defined for such a system?
- How do caches impact correctness?

Memory system properties

Consistency (Memory Model)

- Defines correct shared memory behavior in terms of *loads* and *stores*
  - What value a *load* is allowed to return.
- Specifies the allowed behavior of multithreaded programs executing with shared memory
  - Pb: multiple executions can be correct
- Defines ordering across memory locations

Memory system properties

Consistency (Memory Model)

- Defines correct shared memory behavior in terms of *loads* and *stores*
  - What value a *load* is allowed to return.
- Specifies the allowed behavior of multithreaded programs executing with shared memory
  - Pb: multiple executions can be correct
- Defines ordering across memory locations

Coherence

- Ensures that *load*/*store* behavior remains consistent despite caches
  - The cache coherence protocol makes caches logically invisible to the user.
- Concerns accesses to a memory block

Cache coherence

Sorin, Hill and Wood

Single-Writer, Multiple-Reader (SWMR) invariant

For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) (read-write epoch) or some number of cores that may read it (read epoch).

Data-Value Invariant

The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch.

Implementation

- Invalidate protocols (outside the scope of this course)

Agenda

Consistency and Coherence

Memory models

Data Races

Atomic Operations

Spinlock

Sleeping Locks

Illustrating the problem

Core C0	Core C1
`S1: Store data = NEW;`	
`S2: Store flag = SET;`	`L1: Load r1 = flag;`
	`B1: if(r1!=SET) goto L1;`
	`L2: Load r2 = data;`

Final value of `r2`?

Illustrating the problem

Core C0	Core C1
`S1: Store data = NEW;`	
`S2: Store flag = SET;`	`L1: Load r1 = flag;`
	`B1: if(r1!=SET) goto L1;`
	`L2: Load r2 = data;`

Final value of `r2`?

- I don't know

Illustrating the problem

Core C0	Core C1
`S1: Store data = NEW;`	
`S2: Store flag = SET;`	`L1: Load r1 = flag;`
	`B1: if(r1!=SET) goto L1;`
	`L2: Load r2 = data;`

Final value of `r2`?

- I don't know
- It depends what re-ordering between operations the hardware can do

Re-ordering of operations to different addresses

- Store-store reordering
  - Non-FIFO write buffer
  - $S1$ misses while $S2$ hits the local cache
- Load-load reordering
  - Out-of-order cores
- Load-store/store-load reordering
  - Out-of-order cores

Re-ordering of operations to different addresses

- Store-store reordering
  - Non-FIFO write buffer
  - $S1$ misses while $S2$ hits the local cache
- Load-load reordering
  - Out-of-order cores
- Load-store/store-load reordering
  - Out-of-order cores

If Store-store or Load-load reordering is allowed, the answer to the previous question can be not NEW!

Re-ordering of operations to different addresses

- Store-store reordering
  - Non-FIFO write buffer
  - $S1$ misses while $S2$ hits the local cache
- Load-load reordering
  - Out-of-order cores
- Load-store/store-load reordering
  - Out-of-order cores

If Store-store or Load-load reordering is allowed, the answer to the previous question can be not NEW!

The memory consistency model defines what behavior the programmer can expect and what optimizations might be used in hardware

Re-ordering of operations to different addresses

- Store-store reordering
  - Non-FIFO write buffer
  - $S1$ misses while $S2$ hits the local cache
- Load-load reordering
  - Out-of-order cores
- Load-store/store-load reordering
  - Out-of-order cores

If Store-store or Load-load reordering is allowed, the answer to the previous question can be not NEW!

The memory consistency model defines what behavior the programmer can expect and what optimizations might be used in hardware

Note that with a single thread executing on one core, all these re-orderings are fully safe.

Sequential Consistency (SC)

Definition

The result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program. – Lamport

This boils down to two requirements:

- All operations of a core appear in program order from memory point of view
- All write operations appear atomic

Sequential Consistency (SC)

		Second op	
		Load	Store
First op	Load	X	X
	Store	X	X

Table: SC ordering rules

With SC, the answer to previous question is $r2 = NEW$

Sequential Consistency (SC)

		Second op	
		Load	Store
First op	Load	X	X
	Store	X	X

Table: SC ordering rules

With SC, the answer to previous question is $r2 = NEW$

Problem

Some hardware optimizations do not work efficiently or are complex to implement with SC

Total Store Order (TSO)

Definition

Same as SC except that:

- A load is allowed to return before an earlier store to a different location.
- A core is allowed to read its own *writes* before their are made visible to all cores.

TSO appears to be the memory model of x86 architectures (no formal specification).

Write Buffer

Each core has a write buffer where are stored pending write operations. It prevents the core from stalling if the core does not have read/write access to a cache line.

- TSO formalizes the behavior observed when write buffers are used

Write Buffer

Each core has a write buffer where are stored pending write operations. It prevents the core from stalling if the core does not have read/write access to a cache line.

- TSO formalizes the behavior observed when write buffers are used

		Second op	
		Load	Store
First op	Load	X	X
	Store	B	X

Table: TSO ordering rules

B = Bypassing: No ordering guaranty except that bypassing from the write buffer is required if both operations are to the same address.

Impact of TSO

Core C0	Core C1
`S1: Store data = NEW;`	
`S2: Store flag = SET;`	`L1: Load r1 = flag;`
	`B1: if(r1!=SET) goto L1;`
	`L2: Load r2 = data;`

Final value of `r2`?

Impact of TSO

Core C0	Core C1
`S1: Store data = NEW;`	
`S2: Store flag = SET;`	`L1: Load r1 = flag;`
	`B1: if(r1!=SET) goto L1;`
	`L2: Load r2 = data;`

Final value of `r2`?

- `r2 = NEW` !
- TSO behaves like SC for most programs
- You can assume SC except if you start designing some low level synchronization
- Some architectures (eg, ARM) have a weaker memory model (no guaranty on the order of store operations)

TSO versus SC

Core C0	Core C1
	`/* x=0, y=0 initially*/`
`S1: Store x = NEW;`	`S2: Store y = NEW;`
`L1: Load r1 = y;`	`L2: Load r2 = x;`

Is the final result r1=0, r2=0 possible?

- With SC?
- With TSO?

TSO versus SC

Core C0	Core C1
	`/* x=0, y=0 initially*/`
`S1: Store x = NEW;`	`S2: Store y = NEW;`
`L1: Load r1 = y;`	`L2: Load r2 = x;`

Is the final result r1=0, r2=0 possible?

- With SC? No
- With TSO?

TSO versus SC

Core C0	Core C1
	`/* x=0, y=0 initially*/`
`S1: Store x = NEW;`	`S2: Store y = NEW;`
`L1: Load r1 = y;`	`L2: Load r2 = x;`

Is the final result r1=0, r2=0 possible?

- With SC? No
- With TSO? Yes

TSO versus SC

Core C0	Core C1
	`/* x=0, y=0 initially*/`
`S1: Store x = NEW;`	`S2: Store y = NEW;`
`L1: Load r1 = y;`	`L2: Load r2 = x;`

Is the final result r1=0, r2=0 possible?

- With SC? No
- With TSO? Yes

Fence

A memory fence (or memory barrier) can be used to force the hardware to follow program order.

`S1: Store x = NEW;`	`S2: Store y = NEW;`
`FENCE`	`FENCE`
`L1: Load r1 = y;`	`L2: Load r2 = x;`

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

- No compiler optimization can be applied to this variable (typically because it it accessed by multiple threads)

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

- No compiler optimization can be applied to this variable (typically because it it accessed by multiple threads)

Can both critical sections run?

- with SC?
- with TSO?

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

- No compiler optimization can be applied to this variable (typically because it it accessed by multiple threads)

Can both critical sections run?

- with SC? No
- with TSO?

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

- No compiler optimization can be applied to this variable (typically because it it accessed by multiple threads)

Can both critical sections run?

- with SC? No
- with TSO? Yes

Another Example (1)

```
volatile int flag1 = 0;
volatile int flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main() {
  tid id = thread_create(p1, NULL);
  p2();
  thread_join (id);
}
```

What does `volatile` mean?

- No compiler optimization can be applied to this variable (typically because it it accessed by multiple threads)

Can both critical sections run?

- with SC? No
- with TSO? Yes
- Peterson's algorithm doesn't work as is with TSO!

Another Example (2)

```
volatile int data = 0;
volatile int ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

Can `use` be called with value 0?

- with SC?
- with TSO?

Another Example (2)

```
volatile int data = 0;
volatile int ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

Can `use` be called with value 0?

- with SC? No
- with TSO?

Another Example (2)

```
volatile int data = 0;
volatile int ready = 0;

void p1 (void *ignored) {
  data = 2000;
  ready = 1;
}

void p2 (void *ignored) {
  while (!ready)
    ;
  use (data);
}

int main () { ... }
```

Can `use` be called with value 0?

- with SC? No
- with TSO? No

Another Example (3)

```
volatile int flag1=0;
volatile int flag2=0;

int p1 (void){
  int f, g;
  flag1 = 1;
  f = flag1;
  g = flag2;
  return 2*f + g;
}

int p2 (void){
  int f, g;
  flag2 = 1;
  f = flag2;
  g = flag1;
  return 2*f + g;
}

int main () { ... }
```

Can both return 2?

- with SC?
- with TSO?

Another Example (3)

```
volatile int flag1=0;
volatile int flag2=0;

int p1 (void){
  int f, g;
  flag1 = 1;
  f = flag1;
  g = flag2;
  return 2*f + g;
}

int p2 (void){
  int f, g;
  flag2 = 1;
  f = flag2;
  g = flag1;
  return 2*f + g;
}

int main () { ... }
```

Can both return 2?

- with SC? No
- with TSO?

Another Example (3)

```
volatile int flag1=0;
volatile int flag2=0;

int p1 (void){
  int f, g;
  flag1 = 1;
  f = flag1;
  g = flag2;
  return 2*f + g;
}

int p2 (void){
  int f, g;
  flag2 = 1;
  f = flag2;
  g = flag1;
  return 2*f + g;
}

int main () { ... }
```

Can both return 2?

- with SC? No
- with TSO? Yes

Agenda

Consistency and Coherence

Memory models

Data Races

Atomic Operations

Spinlock

Sleeping Locks

Data races and race conditions

Data race

A data race is when two threads access the same memory location, at least one of these accesses is a *write*, and there is no synchronization preventing the two accesses from occurring concurrently.

Race condition

A race condition is a flaw in a program that occurs because of the timing or ordering of some events.

Data races and race conditions

Data race

A data race is when two threads access the same memory location, at least one of these accesses is a *write*, and there is no synchronization preventing the two accesses from occurring concurrently.

Race condition

A race condition is a flaw in a program that occurs because of the timing or ordering of some events.

- A data race may lead to a race condition but not always
- A race condition might be due to a data race but not always
- Peterson's synchronization algorithm is an example where data races do not lead to race conditions

Data races and race conditions

```
Transfer_Money(amount, account_from, account_to){



  if(account_from.balance < amount){
    return FAILED;
  }

  account_to.balance += amount;



  account_from.balance -= amount;

  return SUCCESS;
}
```

- Data race?
- Dace condition?

Data races and race conditions

```
Transfer_Money(amount, account_from, account_to){



  if(account_from.balance < amount){
    return FAILED;
  }

  account_to.balance += amount;


  account_from.balance -= amount;

  return SUCCESS;
}
```

- Data race? Yes (updates of balance)
- Dace condition? Yes (balance can get below 0)

Data races and race conditions

New try

```
Transfer_Money(amount, account_from, account_to){
  mutex1.lock();
  val=account_from.balance;
  mutex1.unlock();
  if(val < amount){
    return FAILED;
  }
  mutex2.lock();
  account_to.balance += amount;
  mutex2.unlock();
  mutex1.lock();
  account_from.balance -= amount;
  mutex1.unlock();
  return SUCCESS;
}
```

- Data race?
- Dace condition?

Data races and race conditions

New try

```
Transfer_Money(amount, account_from, account_to){
  mutex1.lock();
  val=account_from.balance;
  mutex1.unlock();
  if(val < amount){
    return FAILED;
  }
  mutex2.lock();
  account_to.balance += amount;
  mutex2.unlock();
  mutex1.lock();
  account_from.balance -= amount;
  mutex1.unlock();
  return SUCCESS;
}
```

- Data race? No
- Dace condition? Yes (balance can get below 0)

By the way, how is a lock implemented?

By the way, how is a lock implemented?

Good question!

Recall

- Disabling interrupts
  - Does not work on multi-core systems.
- Peterson's algorithm
  - Requires to know the number of participants in advance
  - Uses only load and store operations

Recall

- Disabling interrupts
  - Does not work on multi-core systems.
- Peterson's algorithm
  - Requires to know the number of participants in advance
  - Uses only load and store operations

To implement a general lock, we need help from the hardware:

- We need atomic operations.

Agenda

Consistency and Coherence

Memory models

Data Races

Atomic Operations

Spinlock

Sleeping Locks

Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Common atomic operations

- test_and_set(type* ptr): sets **ptr* to 1 and returns the previous value in memory

Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Common atomic operations

- test_and_set(type* ptr): sets **ptr* to 1 and returns the previous value in memory
- fetch_and_add(type* ptr, type val): add *val* to **ptr* and returns the previous value in memory

Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Common atomic operations

- test_and_set(type* ptr): sets **ptr* to 1 and returns the previous value in memory
- fetch_and_add(type* ptr, type val): add *val* to **ptr* and returns the previous value in memory
- compare_and_swap(type* ptr, type oldval, type newval): if $*ptr == oldval$, set **ptr* to *newval* and returns `true`; returns `false` otherwise

A concurrent counter

Version with fetch_and_add()

```
int count = 0;
```

Thread 1:

```
for(i=0; i<10; i++){
  fetch_and_add(&count,1);
}
```

Thread 2:

```
for(i=0; i<10; i++){
  fetch_and_add(&count,1);
}
```

Agenda

Recall: lock using busy waiting (attempt)

```
typedef struct{
  int  flag ;
} lock_t ;

void  init ( lock_t  *lock) {
  lock->flag = 0;
}

void  lock( lock_t  *lock) {
  while(lock-> flag == 1){;}
  lock->flag = 1;
}

void  unlock( lock_t  *lock) {
  lock->flag = 0;
}
```

Recall: lock using busy waiting (attempt)

```
typedef struct{
  int  flag ;
} lock_t ;

void  init ( lock_t  *lock) {
  lock->flag = 0;
}

void  lock( lock_t  *lock) {
  while(lock-> flag == 1){;}
  lock->flag = 1;
}

void  unlock( lock_t  *lock) {
  lock->flag = 0;
}
```

- Multiple threads can be in CS at the same time!

Spinlock

A lock using test_and_set()

```
typedef struct{
  int  flag ;
} lock_t ;

void  init ( lock_t  *lock) {
  lock->flag = 0;
}

void  lock( lock_t  *lock) {
  while  ( test_and_set (&lock->flag) == 1){;}
}

void  unlock( lock_t  *lock) {
  lock->flag = 0;
}
```

Spinlock

A lock using test_and_set()

```
typedef struct{
  int  flag ;
} lock_t ;

void  init ( lock_t  *lock) {
  lock->flag = 0;
}

void  lock( lock_t  *lock) {
  while  ( test_and_set (&lock->flag) == 1){;}
}

void  unlock( lock_t  *lock) {
  lock->flag = 0;
}
```

Beware:

- The solution is safe and ensures progress
- The solution does not warrant bounded waiting

Spinlock

A lock using compare_and_swap()

```
typedef struct{
  int  flag ;
} lock_t ;

void  init ( lock_t  *lock) {
  lock->flag = 0;
}

void  lock( lock_t  *lock) {
  while  (!compare_and_swap(&lock->flag,0,1)){;}
}

void  unlock( lock_t  *lock) {
  lock->flag = 0;
}
```

Beware:

- The solution is safe and ensures progress
- The solution does not warrant bounded waiting

About spinlocks

- As the name suggests, it implies busy waiting:
  - Busy waiting not only wastes CPU time, it generates *traffic* on the interconnection network.
- There are more complex solutions that provide bounded waiting
- Spinning can be fine when the number of threads is not more than the number of cores
- Spinlocks are used when the critical section is short

Agenda

Consistency and Coherence

Memory models

Data Races

Atomic Operations

Spinlock

Sleeping Locks

Sleeping instead of spinning

The problem

- Spinning threads might delay the thread currently executing a critical section
- Could we use a `yield()` primitive (explicitly tell the OS that a thread wants to give up the CPU)?

Sleeping instead of spinning

The problem

- Spinning threads might delay the thread currently executing a critical section
- Could we use a `yield()` primitive (explicitly tell the OS that a thread wants to give up the CPU)?
  - Simply moves the caller from the *running* state to the *ready* state
  - Imagine 100 threads competing for the same lock ... still not doing anything useful 99% of the time

Sleeping instead of spinning

The problem

- Spinning threads might delay the thread currently executing a critical section
- Could we use a `yield()` primitive (explicitly tell the OS that a thread wants to give up the CPU)?
  - Simply moves the caller from the *running* state to the *ready* state
  - Imagine 100 threads competing for the same lock ... still not doing anything useful 99% of the time

We need to remove threads from the ready list.

- This is what we call sleeping

Sleeping locks: Mutexes

The mutex data structure is simple:

```
typedef struct mutex {
    bool is_locked;              /* true if locked */
    thread_id_t owner;           /* thread holding lock, if locked */
    thread_list_t waiters;       /* threads waiting for lock */

    lower_level_lock_t lk;       /* Protect above fields */
};
```

Sleeping locks: Mutexes

The mutex data structure is simple:

```
typedef struct mutex {
    bool is_locked;              /* true if locked */
    thread_id_t owner;           /* thread holding lock, if locked */
    thread_list_t waiters;       /* threads waiting for lock */

    lower_level_lock_t lk;       /* Protect above fields */
};
```

- On lock(): If the mutex is locked, the tread is removed from the "ready list" of the kernel (set of threads that are ready to execute), and inserted into the list of waiters of the mutex.
- On unlock(): If the list of waiters is not empty, remove one thread from the list and put it back into the ready list.
- The lower_level_lock_t is used to ensure mutual exclusion when manipulating the list of waiters.

Comments on this Mutex algorithm

Which locking algorithm should we use for the lower level lock?

Comments on this Mutex algorithm

Which locking algorithm should we use for the lower level lock?

- A spinlock can be used as the CS is very short.

Comments on this Mutex algorithm

Which locking algorithm should we use for the lower level lock?

- A spinlock can be used as the CS is very short.

This corresponds to how things work inside the kernel

- The code of the kernel itself has to be thread safe!

Comments on this Mutex algorithm

Which locking algorithm should we use for the lower level lock?

- A spinlock can be used as the CS is very short.

This corresponds to how things work inside the kernel

- The code of the kernel itself has to be thread safe!

How are pthread mutexes implemented?

Comments on this Mutex algorithm

Which locking algorithm should we use for the lower level lock?

- A spinlock can be used as the CS is very short.

This corresponds to how things work inside the kernel

- The code of the kernel itself has to be thread safe!

How are pthread mutexes implemented?

- "Remove thread from the ready list" and "put thread in the ready list" requires a system call
- We would like to avoid always making system calls

User-level mutexes

First try

We should sleep only if there is contention on the lock.

```
struct mutex {
  int busy;
  thread *waiters;
};
void lock (mutex *mtx) {
  while ( test_and_set (&mtx->busy)) {     /* 1 */
    atomic_push (&mtx->waiters, self);     /* 2 */
    sleep ();
  }
}
void unlock (mutex *mtx) {
  mtx->busy = 0;
  wakeup (atomic_pop (&mtx->waiters));
}
```

User-level mutexes

First try

We should sleep only if there is contention on the lock.

```
struct mutex {
  int busy;
  thread *waiters;
};
void lock (mutex *mtx) {
  while ( test_and_set (&mtx->busy)) {     /* 1 */
    atomic_push (&mtx->waiters, self);     /* 2 */
    sleep ();
  }
}
void unlock (mutex *mtx) {
  mtx->busy = 0;
  wakeup (atomic_pop (&mtx->waiters));
}
```

What if the mutex gets released between the execution of lines 1 and 2?

Futex

Linux provides the `futex` system call to solve the problem.

- Ask to sleep if the value of a variable hasn't changed

Interface:

- void futex(void* addr1, FUTEX_WAIT, int val ...)
  - Calling thread is suspended ("goes to sleep") if *addr1 == val
- void futex(void* addr1, FUTEX_WAKE, int val)
  - Wakes up at most val threads waiting on addr1
  - Typical usage: val=1 or val=INT_MAX (broadcast)

See "Futexes are tricky" by U. Drepper for a nice discussion of futexes

User-level mutexes

First try with futexes

```
struct mutex {
  int busy;     /*1 if busy*/
};
void lock (mutex *mtx) {
  while ( test_and_set (&mtx->busy))
    futex(&mtx->busy, FUTEX_WAIT, 1);
}
void unlock (mutex *mtx) {
  mtx->busy = 0;
  futex(&mtx->busy, FUTEX_WAKE, 1);
}
```

User-level mutexes

First try with futexes

```
struct mutex {
  int busy;     /*1 if busy*/
};
void lock (mutex *mtx) {
  while ( test_and_set (&mtx->busy))
    futex(&mtx->busy, FUTEX_WAIT, 1);
}
void unlock (mutex *mtx) {
  mtx->busy = 0;
  futex(&mtx->busy, FUTEX_WAKE, 1);
}
```

`unlock` function makes a system call even when there is no contention

User-level mutexes

Second try with futexes

```
struct mutex {
  int busy;   /* Counts number of contending threads */
};
void lock (mutex *mtx) {
  int c;
  while ((c = fetch_and_add(mtx->busy, 1)))
    futex(&mtx->busy, FUTEX_WAIT, c+1);
}
void unlock (mutex *mtx) {
  if (fetch_and_add(mtx->busy, -1) != 1) {
    mtx->busy = 0;
    futex(&mtx->busy, FUTEX_WAKE, INT_MAX);
  }
}
```

User-level mutexes

Second try with futexes

```
struct mutex {
  int busy;   /* Counts number of contending threads */
};
void lock (mutex *mtx) {
  int c;
  while ((c = fetch_and_add(mtx->busy, 1)))
    futex(&mtx->busy, FUTEX_WAIT, c+1);
}
void unlock (mutex *mtx) {
  if (fetch_and_add(mtx->busy, -1) != 1) {
    mtx->busy = 0;
    futex(&mtx->busy, FUTEX_WAKE, INT_MAX);
  }
}
```

- A wrong interleaving of calls to `FAA` and `FUTEX_WAIT` could lead to have `FUTEX_WAIT` repeatedly failing (and ultimately cause an overflow on `busy`).
- We need to wake up all threads on every unlock() – very costly

User-level mutexes

Third try with futexes (correct solution)

```
struct mutex {
  // 3-state variable: 0=unlocked, 1=locked no waiters, 2=locked+waiters
  int state;
};
void lock (mutex *mtx) {
  int c = 1;
  if (!compare_and_swap(&mtx->state, 0, c)) {
    c= swap(&mtx->state, 2); /*atomically write state=2, return old value*/
    while (c != 0) {
      futex (&mtx->state, FUTEX_WAIT, 2);
      c= swap (&mtx->state, 2);
    }
  }
}
void unlock (mutex *mtx) {
  if (fetch_and_add(mtx->state, -1) != 1) {
    mtx->state = 0;
    futex (&mtx->state, FUTEX_WAKE, 1);
  }
}
```

User-level mutexes

Third try with futexes

Comments

- The 3-state variable allows waking up only when needed without any risk of counter overflow.
- The 3-state variable implies that we use CAS instead of FAA
- The SWAP to `mtx->state` to 2 is announcing that we are waiting
- When c==0 after a SWAP, it means that we grabbed the lock
- busy==2 means that there might be a thread waiting

Performance without contention

- lock: 1 atomic operation + 0 system call
- unlock: 1 atomic operation + 0 system call