Operating Systems

Thread Synchronization: Advanced Topics

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2025

References

The content of these lectures is inspired by:

- The lecture notes of Prof. André Schiper.
- The lecture notes of Prof. David Mazières.
- *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.

Included in this lecture

Several concepts related to concurrent programming

- Performance
  - Amdahl's law
- Characteristics of shared memory
  - Coherence
  - Consistency (SC, TSO)
- Correctness of algorithms
  - Data race / race condition
  - Avoiding deadlocks

Agenda

Agenda

First a reminder

Threads have two main purposes:

- Overlapping computation and I/Os
- Taking advantage of multicore (multi-)processors for compute-intensive code

The performance improvement that can be expected from using multiple threads is not infinite.

Amdahl's Law

Speedup of parallel code is limited by the sequential part of the code:

$$T(n) = (1 - p) \cdot T(1) + \frac{p}{n} \cdot T(1)$$

where:

- $T(x)$ is the time it takes to complete the task on x CPUs
- p is the portion of time spent on parallel code in a sequential execution
- n is the number of CPUs

Even with massive number of CPUs:

$$\lim_{n \to \infty} T(n) = (1 - p) \cdot T(1)$$

Amdahl's Law

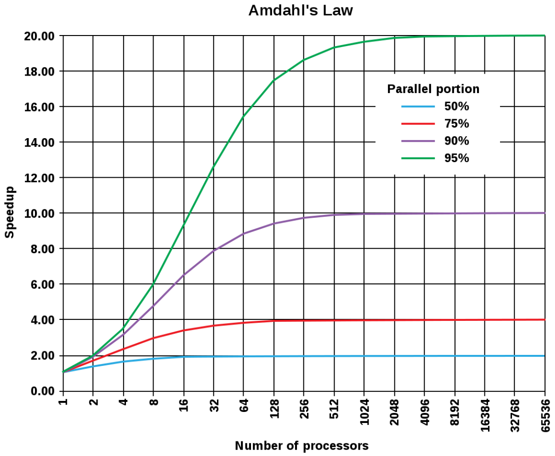


Figure: Speedup as a function of the number of CPUs[1]

[1] "AmdahlsLaw" by Daniels220 at English Wikipedia

Efficient concurrent programs

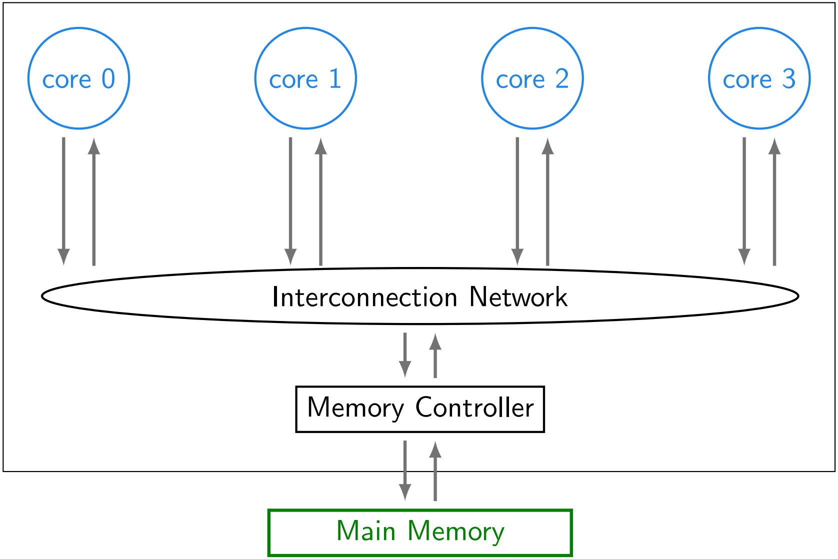
Critical sections are pieces of code that are executed sequentially!

Finding the right granularity for critical sections:

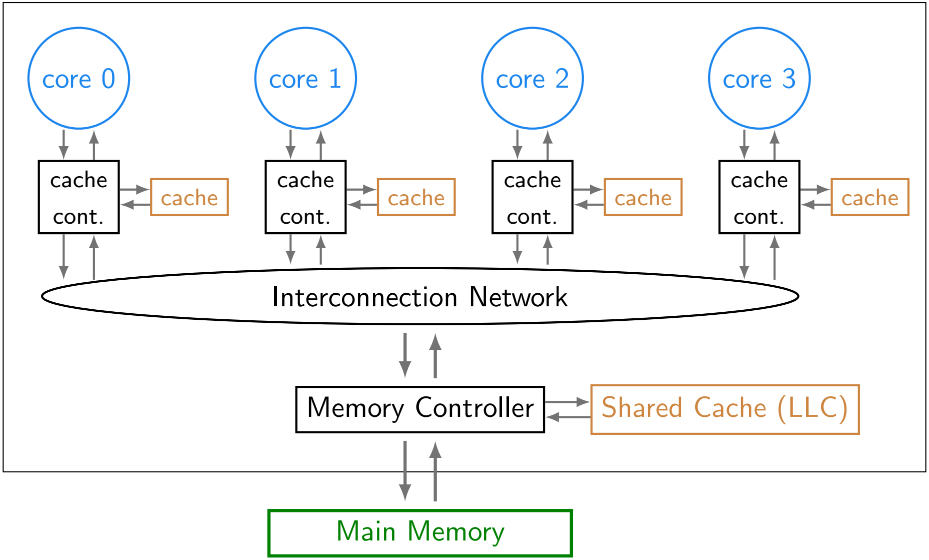
- Correctness always comes first!
- Try to reduce the size of the critical sections that are executed most often
- Do not forget that lock/unlock operations also take time

Agenda

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
- Some 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 multi-threaded programs executing with shared memory
  - Pb: multiple executions can be correct
- Defines ordering across memory addresses

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 multi-threaded programs executing with shared memory
  - Pb: multiple executions can be correct
- Defines ordering across memory addresses

Coherence

- Ensures that *load*/*store* behavior remains consistent despite caches
  - The cache coherence protocol makes caches logically invisible to the user.
- Deals with the case where multiple cores access the same memory address.

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

Illustrating the problem

C0	C1
`data = NEW;`	
`flag = SET;`	
	`while(flag != SET){;}`
	`d = data;`

Illustrating the problem

C0	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

C0	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`?

First guess:

- The obvious answer is $r2 = NEW$
- This assumes the hardware does not re-order load/store operations.

Illustrating the problem

C0	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`?

First guess:

- The obvious answer is $r2 = NEW$
- This assumes the hardware does not re-order load/store operations.

In practice:

- We cannot answer
- It depends what re-ordering between operations the hardware can do

Re-ordering memory accesses

A modern core might re-order memory accesses to different addresses in different ways:

- Store-store reordering
  - Non-FIFO write buffer
  - $S1$ misses while $S2$ hits the local cache
- Load-load reordering
  - Out-of-order cores (dynamic execution) = Executes instructions in an order governed by the availability of input data.
- Load-store/store-load reordering
  - Out-of-order cores

Re-ordering is used to improve performance.

Re-ordering memory accesses

If Store-store or Load-load reordering is allowed, the answer to the previous question can be $r2 \neq NEW$

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.

Some consistency models:

- Sequential consistency
- Total store order

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. – L. 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
  - A write is made visible to all cores at the same time

Sequential Consistency (SC)

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

Table: SC ordering rules. (What ordering is enforced for requests issued by one core)

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. (What ordering is enforced for requests issued by one core)

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 AMD and Intel x86 architectures (no formal specification).

- It is the consequence of introducing write buffers

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 immediate 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 immediate 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	**WB**	X

Table: TSO ordering rules (What ordering is enforced for requests issued by one core)

WB = No ordering guaranty except that most recent value should be returned from write buffer if both operations are to the same address.

Impact of TSO

C0	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

C0	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 (ex: ARM) have a weaker memory model (no ordering enforced for 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 (also called 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)

What does `volatile` mean?

```
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);
}
```

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 [a]

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 [a]

Can both critical sections run?

- with SC?
- with TSO?

[a] The content of a volatile variable can change by means unknown to the compiler – use with care

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 [a]

Can both critical sections run?

- with SC? **No**
- with TSO?

[a] The content of a volatile variable can change by means unknown to the compiler – use with care

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 [a]

Can both critical sections run?

- with SC? **No**
- with TSO? **Yes**

[a] The content of a volatile variable can change by means unknown to the compiler – use with care

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 [a]

Can both critical sections run?

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

[a]The content of a volatile variable can change by means unknown to the compiler – use with care

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

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?
- Race 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)
- Race 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?
- Race 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
- Race condition? Yes (balance can get below 0)

Agenda

Deadlocks (with condition variables)

```
mutex_t m1, m2;
cond_t c1;

void p1 (void *ignored) {
  lock (m1); lock (m2);
  while(!ready){wait(c1,m2);}
  /* do something */
  unlock (m2); unlock (m1);
}
void p2 (void *ignored) {
  lock (m1); lock (m2);
  /* do something*/
  signal (c1)
  unlock (m2);unlock (m1);
}
```

One lesson: Dangerous to hold locks when crossing abstraction limits!

- e.g., `lock(a)` then call function that uses condition variable

Dealing with deadlocks

Prevention

Eliminating one of the conditions:

- Non-blocking (wait-free) algorithms
- Wait on all resources at once
- Use optimistic concurrency control (transactions)
- Always lock resources in the same order

Avoidance

Prevent the system from entering an unsafe state (requires knowing a priori the resource requirements of each process)

Dealing with deadlocks

Detection + corrective action

Problem: What corrective action?

- Process termination
- Rollback (Possible? Cost?)

Ignore the problem

Solution chosen by most OSes (including UNIX systems)

- Assume it occurs rarely enough to avoid the need for a complex solution

Read “Operating Systems: Three Easy Pieces”, chapter 32 for a more detailed discussion on deadlocks

Concurrent algorithms without locks

Non-blocking concurrent algorithms do not rely on locks.

Progress condition

- Lock freedom: If the algorithm runs long enough, it is guaranteed that some operations will finish.
    - Note that with locks, there is no such guarantee: if the thread holding the lock crashes, no progress anymore.
- Wait freedom: Each operation takes a finite number of steps to complete.
    - Newcomers need to help *older* requests before executing their own operation

These algorithms strongly rely on *compare_and_swap()* operations.