Operating Systems

Thread Synchronization: Implementation

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2018

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

Other references:

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

Agenda

Agenda

Previous lecture

Concurrent programming requires thread synchronization.

The problem:

Threads executing on a shared-memory (multi-)processor is an asynchronous system.

- A thread can be preempted at any time.
- Reading/writing a data in memory incurs unpredictable delays (data in L1 cache vs page fault).

Previous lecture

Classical concurrent programming problems

- Mutual exclusion
- Producer-consumer

Concepts related to concurrent programming

- Critical section
- Busy waiting
- Deadlock

Synchronization primitives

- Locks
- Semaphores
- Condition variables

Agenda

High-level goals

How to implement synchronization primitives?

Answering this question is important to:

- Better understand the semantic of the primitives
- Learn about the interactions with the OS
- Learn about the functioning of memory
- Understand the trade-offs between different solutions

Content of the lecture

Solutions to implement mutual exclusion

- Peterson's algorithm
- Spinlocks
- Sleeping locks

Basic mechanisms used for synchronization

- Atomic operations (hardware)
- Futex (OS)

Agenda

A shared counter (remember . . .)

Example seen during the lab

```
int count = 0;
```

Thread 1:

```
for(i=0; i<10; i++){
  count++;
}
```

Thread 2:

```
for(i=0; i<10; i++){
  count++;
}
```

What is the final value of `count`?

- A value between 2 and 20

Explanation (remember . . .)

Let's have a look at the (pseudo) assembly code for `count++`:

```
mov   count, register
add   $0x1, register
mov   register, count
```

A possible interleave (for one iteration on each thread)

Thread 1	Thread 2
mov count, register	
add $0x1, register	
	mov count, register
	add $0x1, register
mov register, count	
	mov register, count

At the end, `count=1` :-(

Implementation: First try (remember . . .)

Shared variables:

```
int count=0;
int busy=0;
```

Thread 1:

```
while(busy){;}
busy=1;
count++;
busy=0;
```

Thread 2:

```
while(busy){;}
busy=1;
count++;
busy=0;
```

This solution violates both safety and liveness.

Critical sections

Thread 1:

```
Enter CS;
count++;
Leave CS;
```

Thread 2:

```
Enter CS;
count++;
Leave CS;
```

How to implement Enter CS and Leave CS?

Disabling interrupts

Description

Prevent a thread from being interrupted while it is in CS

- If a thread is not interrupted, it will (hopefully) execute the CS atomically.

Problems with disabling interrupts

Disabling interrupts

Description

Prevent a thread from being interrupted while it is in CS

- If a thread is not interrupted, it will (hopefully) execute the CS atomically.

Problems with disabling interrupts

- The solution is unsafe:
  - Enabling threads to disable interrupts requires allowing them to run *privileged* operations. (trust ?)
  - Possible attack: disable interrupts and run forever.
- The solution is inefficient:
  - Disabling interrupts is a costly operation.

Disabling interrupts

Solution for user threads (n:1)

- Have a per-thread *Do-Not-Interrupt* (DNI) bit.
- Periodic timer signal caught by thread scheduler.
- Scheduling decisions based on DNI bits.

Disabling interrupts

Solution for user threads (n:1)

- Have a per-thread *Do-Not-Interrupt* (DNI) bit.
- Periodic timer signal caught by thread scheduler.
- Scheduling decisions based on DNI bits.

In any case:

Disabling interrupts does not work on multi-processors!

Peterson's algorithm

Presentation

- Mutual exclusion algorithm solely based on read and write operations to a shared memory
- First correct solution for two threads by Dekker in 1966
- Peterson proposed a simpler solution in 1981

Peterson's algorithm

Solution for 2 threads T_0 and T_1

Algorithm 1 Peterson's algorithm for thread T_i

Global Variables:

```
bool wants[2] = {false, false};
int not_turn; /* can be 0 or 1 */

enter_CS()
    wants[i] = true;
    not_turn = i;
    while wants[1-i] == true and not_turn == i do
        /* do nothing */
    end while

leave_CS()
    wants[i] = false;
```

Peterson's algorithm

A few comments:

- wants: To declare that the thread wants to enter.
- not_turn: To arbitrate if the 2 threads want to enter.
- Line 6: "*The other thread wants to access and not our turn, so loop*".

Correctness of the algorithm

The algorithm is correct. How can it be shown?

- Difficult problem in the general case.

Mathematical Proof

- Reasoning about the properties of the algorithm using classical methods (induction, contradiction, ...).
- Cannot be considered as reliable:
  - We show only the points that we thought about. What if we overlooked a problem?
  - Still increases the confidence of the reader.

Correctness of the algorithm

Model checking

- Description (state space enumeration)
  - Represents the algorithms as a set of states and transitions.
  - Defines a property to be checked (2 threads in CS)
  - Enumerates all possible states to verify the property (here for 2 threads).

- Complex problem:
  - Combinatorial blow up of the state-space (polynomial in number of threads)

Discussion about correctness

- Mutual exclusion: both threads in CS?

- Progress

- Bounded waiting

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from CS
- Progress
- Bounded waiting

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`,
    so `not_turn` would have blocked one thread from CS
- Progress
  - If T_{1-i} doesn't want CS, `wants[1-i] == false`, so T_i won't loop
  - If both threads try to enter, one thread is the `no_turn` thread
- Bounded waiting

Discussion about correctness

- Mutual exclusion: both threads in CS?
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from CS
- Progress
  - If T_{1-i} doesn't want CS, `wants[1-i] == false`, so T_i won't loop
  - If both threads try to enter, one thread is the `no_turn` thread
- Bounded waiting
  - If T_i wants to lock and T_{1-i} tries to re-enter, T_{1-i} will set `not_turn = 1 - i`, allowing T_i in.

Peterson's algorithm – Limits

- Given solution works for 2 threads
- Can be generalized to *n* threads but *n* must be known in advance
- Note that the current version assumes that the memory is *sequentially consistent*. Most processors don't provide sequential consistency.
  - Stay tuned ...

Summary

- 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

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 its previous value
- fetch_and_add(type *ptr, type val): adds *val* to **ptr* and returns its previous value
- compare_and_swap(type *ptr, type oldval, type newval): if $*ptr == oldval$, set **ptr* to *newval* and returns `true`; returns `false` otherwise

A shared counter

With atomic operations

```
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)

```
struct{
    int flag;
} lock_t;

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

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

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

Recall: lock using busy waiting (attempt)

```
struct{
    int flag;
} lock_t;

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

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

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

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

Spinlock with test_and_set()

```
struct{
    int flag;
} lock_t;

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

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

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

Spinlock with test_and_set()

```
struct{
    int flag;
} lock_t;

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

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

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

Beware:

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

Spinlock with compare_and_swap()

```
struct{
    int flag;
} lock_t;

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

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

void unlock(lock_t *L) {
    L->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 cycles, it interferes with the execution of other threads.
  - And what about energy consumption?
- There are more complex algorithms that provide **bounded waiting**
- Spinning may be acceptable when the number of threads is not more than the number of cores
- Spinlocks might be used when the critical section is short

Agenda

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

We need to remove threads from the ready list.

- This is what we call **sleeping**.
- The thread is not eligible anymore to be executed on the CPU.

Sleeping locks (mutexes): High-level description

lock()

If the mutex is locked, remove the calling thread from the “ready list” of the kernel (set of threads that are ready to execute), and insert it into the list of threads waiting on the mutex.

unlock()

If the list of waiting threads is not empty, remove one thread from the list and put it back into the ready list.

Sleeping locks: Design

Discussion on performance

- Manipulating the ready list implies a system call (interaction with the scheduler).
- We should limit the number of system calls (costly)
- The common case is: There is no contention on the lock (a single thread tries to access the CS)
  - We should seek for a solution that is optimized for this case.

User-level mutexes: First try

Assuming a sleep() and a wakeup() system calls are available

```
struct {
    int busy;
    thread *waiters;
} mutex;

void lock (mutex *mtx) {
    while (test_and_set (&mtx->busy)) {
        atomic_put (&mtx->waiters, self); /* waiters protected by a lock */
        sleep ();
    }
}

void unlock (mutex *mtx) {
    mtx->busy = 0;
    wakeup (atomic_get (&mtx->waiters));
}
```

User-level mutexes: First try

Assuming a sleep() and a wakeup() system calls are available

```
struct {
    int busy;
    thread *waiters;
} mutex;

void lock (mutex *mtx) {
    while (test_and_set (&mtx->busy)) {          (1)
        atomic_put (&mtx->waiters, self);         (2)
        sleep ();
    }
}

void unlock (mutex *mtx) {
    mtx->busy = 0;
    wakeup (atomic_get (&mtx->waiters));
}
```

- If unlock() is called between (1) and (2), a thread could sleep forever.
  - Testing `busy` and putting the thread to sleep is not *atomic*.

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 on futexes

User-level mutexes: First try with futexes

```
struct {
    int busy; /*1 if busy*/
} mutex;

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 {
    int busy; /*1 if busy*/
} mutex;

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);
}
```

Opportunity for improvement

- unlock function makes a call to futex (system call) even when there is no thread *waiting*.

User-level mutexes: Second try with futexes

```
struct {
    int busy; /* Counts number of contending threads */
} mutex;

void lock (mutex *mtx) {
    int c;
    while ((c = fetch_and_add(mtx->busy, 1)) != 0)
        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 {
    int busy; /* Counts number of contending threads */
} mutex;

void lock (mutex *mtx) {
    int c;
    while ((c = fetch_and_add(mtx->busy, 1)) != 0)
        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: good solution with futexes

```
struct {
    // 3−state variable: 0=unlocked, 1=locked no waiters, 2=locked+waiters
    int state;
} mutex;

void lock (mutex *mtx) {
    if (!compare_and_swap(&mtx−>state, 0, 1)) {
        int c = swap(&mtx−>state, 2); /*atomically write 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: good solution 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 SWAP, it means that we grabbed the lock
  - `mtx->state==0` means that the lock is not held
- `mtx->state==2` means that there might be a thread waiting
  - When a thread is woken up from `FUTEX_WAIT`, it cannot know if it is the last waiting thread
  - If the lock is released between the call to CAS and the call to SWAP, it might be the case that no thread will be waiting

User-level mutexes: Performance

Performance without contention

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

Hybrid approach: two-phase lock

- If the lock is about to be released, spinning can be more efficient than sleeping.
- Idea: Spin for a few iterations before sleeping
- Corresponds to the current implementation of pthread mutexes.

Implementation of futexes

Required for correctness:

- On FUTEX_WAIT, checking the value and putting the thread to sleep should be done in an atomic step.
  - Otherwise we have the same problem as in Slide 36.
- To ensure this, a lock is used inside the kernel.
  - FUTEX_WAIT and FUTEX_WAKE start by grabbing that lock.

How to implement the low-level lock?

Implementation of futexes

Required for correctness:

- On FUTEX_WAIT, checking the value and putting the thread to sleep should be done in an atomic step.
  - Otherwise we have the same problem as in Slide 36.
- To ensure this, a lock is used inside the kernel.
  - FUTEX_WAIT and FUTEX_WAKE start by grabbing that lock.

How to implement the low-level lock?

- The CS is very short (put/get in a list)
- A spinlock can be used !

Agenda

Reminder

Goals of the lecture

Mutual exclusion: legacy solutions

Atomic operations

Spinlocks

Sleeping locks

About priorities

Problem with priorities: Priority inversion

Processes/threads in a system might have different priorities:

- If a thread with a high priority is ready to execute, it should get the CPU instead of threads with lower priority

Priority inversion

1. 2 threads, 1 CPU: priority(T_1) $>$ priority(T_2)
2. T_1 is interrupted; T_2 starts executing and grab a lock.
3. T_1 resumes and gets the CPU again.
4. T_1 wants to grad the lock: What happens next?

Problem with priorities: Priority inversion

Processes/threads in a system might have different priorities:

- If a thread with a high priority is ready to execute, it should get the CPU instead of threads with lower priority

Priority inversion

1. 2 threads, 1 CPU: priority(T_1) $>$ priority(T_2)
2. T_1 is interrupted; T_2 starts executing and grab a lock.
3. T_1 resumes and gets the CPU again.
4. T_1 wants to grad the lock: What happens next?
   - With a spinlock: deadlock $\rightarrow$ T_1 spins forever
   - With a sleeping lock: ok

Problem with priorities: Priority inversion

Processes/threads in a system might have different priorities:

- If a thread with a high priority is ready to execute, it should get the CPU instead of threads with lower priority

Priority inversion

1. 2 threads, 1 CPU: priority(T_1) $>$ priority(T_2)
2. T_1 is interrupted; T_2 starts executing and grab a lock.
3. T_1 resumes and gets the CPU again.
4. T_1 wants to grad the lock: What happens next?
   - With a spinlock: deadlock $\rightarrow$ T_1 spins forever
   - With a sleeping lock: ok
   - But if you add a third thread with priority(T_1) $>$ priority(T_3) $>$ priority(T_2), even with a sleeping lock T_1 and T_2 might be blocked forever (e.g., if T_3 never tries to grab the lock)

Problem with priorities: Priority inversion

Definition

The problem is called Priority Inversion because the high priority task is indirectly blocked by a low priority task.

- Search "Mars Pathfinder Mission (1997)" for an example

Solutions

- Priority Ceiling: Priority associated with the mutex is assigned to the task grabbing the mutex
  - Priority of the mutex should be equal to that of the task with the highest priority accessing it.
- Priority Inheritance: The low-priority task holding the mutex gets assigned the priority of the high-priority task contending for that mutex.

Additional resources

To complement this lecture, read:

- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau
  - Chapter 28: Locks