Operating System Concepts

Lecture 17: Hardware Support for Synchronization

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Solutions to the mutual exclusion problem
  - Dekker's algorithm
  - Peterson's algorithm
- Hardware support for synchronization

Third attempt

Thread A

```
note[0] = 1;
if(note[1] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

Thread B

```
note[1] = 1;
while(note[0] == 1) {
  ; // spin
}
if(milk == 0) {
  buy_milk();
}
note[1] = 0;
```

Correctness of the third attempt

Thread A

```
   note[0] = 1;
X: if(note[1] == 0) {
     if(milk == 0) {
       buy_milk();
     }
   }
   note[0] = 0;
```

Thread B

```
   note[1] = 1;
Y: while(note[0] == 1) {
     ; // spin
   }
   if(milk == 0) {
     buy_milk();
   }
   note[1] = 0;
```

- at point X either there is a note left by Thread B or not
  - if there is a note, then B is either checking and buying milk as needed, or is waiting for A to remove the note. So in both cases, A must remove its note ASAP
  - if not, B has either bought milk or hasn't started yet. In both cases, A can safely check if milk is needed and buy

Correctness of the third attempt

Thread A

```
   note[0] = 1;
X: if(note[1] == 0) {
     if(milk == 0) {
       buy_milk();
     }
   }
   note[0] = 0;
```

Thread B

```
   note[1] = 1;
Y: while(note[0] == 1) {
     ; // spin
   }
   if(milk == 0) {
     buy_milk();
   }
   note[1] = 0;
```

- at point Y either there is a note left by Thread A or not
  - if there is a note, then A must be checking B's note or buying milk as needed. So B has to wait until there is no longer a note left by A; Once this happens B either finds milk that A bought or buys it if needed
  - if not, it is safe for B to buy milk as needed since A has either not yet started or has quit

Is this a good solution though?

- relies on load and store operations being atomic
- is too complicated — it was hard to convince ourselves this solution works
- is asymmetrical — operations executed by Threads A and B are different
    - adding more threads would require different code for each new thread and modifications to existing threads
- requires busy waiting — Thread B is consuming CPU resources despite the fact that it is not doing any useful work

A better solution: Dekker's algorithm

Thread A

```
lock[0] = 1;
while(lock[1] == 1) {
  if(turn != 0) {
    lock[0] = 0;
    while(turn != 0)
      ; // spin
    lock[0] = 1;
  }
}
<critical section>
turn = 1; // allow B to take a turn
lock[0] = 0;
```

Thread B

```
lock[1] = 1;
while(lock[0] == 1) {
  if(turn != 1) {
    lock[1] = 0;
    while(turn != 1)
      ; // spin
    lock[1] = 1;
  }
}
<critical section>
turn = 0; // allow A to take a turn
lock[1] = 0;
```

```
if(milk == 0)
  buy_milk();
```

- let's use an extra variable which indicates whose turn it is to enter the critical section
- no hardware support is still required

A better solution: Dekker's algorithm

Thread A

```
lock[0] = 1;
while(lock[1] == 1) {
  if(turn != 0) {
    lock[0] = 0;
    while(turn != 0)
      ; // spin
    lock[0] = 1;
  }
}
<critical section>
turn = 1; // allow B to take a turn
lock[0] = 0;
```

Thread B

```
lock[1] = 1;
while(lock[0] == 1) {
  if(turn != 1) {
    lock[1] = 0;
    while(turn != 1)
      ; // spin
    lock[1] = 1;
  }
}
<critical section>
turn = 0; // allow A to take a turn
lock[1] = 0;
```

release the lock until it's your turn again

```
if(milk == 0)
  buy_milk();
```

- let's use an extra variable which indicates whose turn it is to enter the critical section
- no hardware support is still required

A better solution: Peterson's algorithm

Thread A

```
lock[0] = 1;
turn = 1; // give B a turn
while(lock[1] == 1 && turn != 0)
  ; // spin
<critical section>
lock[0] = 0;
```

Thread B

```
lock[1] = 1;
turn = 0; // give A a turn
while(lock[0] == 1 && turn != 1)
  ; // spin
<critical section>
lock[1] = 0;
```

- simplifies Dekker's algorithm
  - the lock array is used to indicate that a thread is **ready** to enter the CS
  - the turn variable is used to indicate whose turn it is to enter the CS
- thread `i` enters CS only if: either `lock[j] = false` or `turn = i`

Peterson’s algorithm

Thread A

```
lock[0] = 1;
turn = 1; // give B a turn
while(lock[1] == 1 && turn != 0)
  ; // spin
```

Thread B

```
lock[1] = 1;
turn = 0; // give A a turn
while(lock[0] == 1 && turn != 1)
  ; // spin
```

Thread A:

```
<critical section>
lock[0] = 0;
```

Thread B:

```
<critical section>
lock[1] = 0;
```

Correctness conditions in general

- Mutual Exclusion (**safety**)
  - if a thread is executing in its critical section, then no other processes can be executing in their critical sections

Correctness conditions in general

- Mutual Exclusion (**safety**)
  - if a thread is executing in its critical section, then no other processes can be executing in their critical sections
- Progress (**liveness**)
  - if no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next <u>cannot be postponed indefinitely</u>

Correctness conditions in general

- Mutual Exclusion (**safety**)
  - if a thread is executing in its critical section, then no other processes can be executing in their critical sections
- Progress (**liveness**)
  - if no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- Bounded Waiting
  - a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - assume that each process executes at a nonzero speed
    - no assumption concerning relative speed of the processes

Implementing critical sections in software is hard

- Peterson’s algorithm works and the solution is symmetric
  - but it is limited to two threads
  - requires busy waiting
  - is not guaranteed to work on modern architectures!
    - processors and/or compilers may reorder operations that have no dependencies.

Thread A

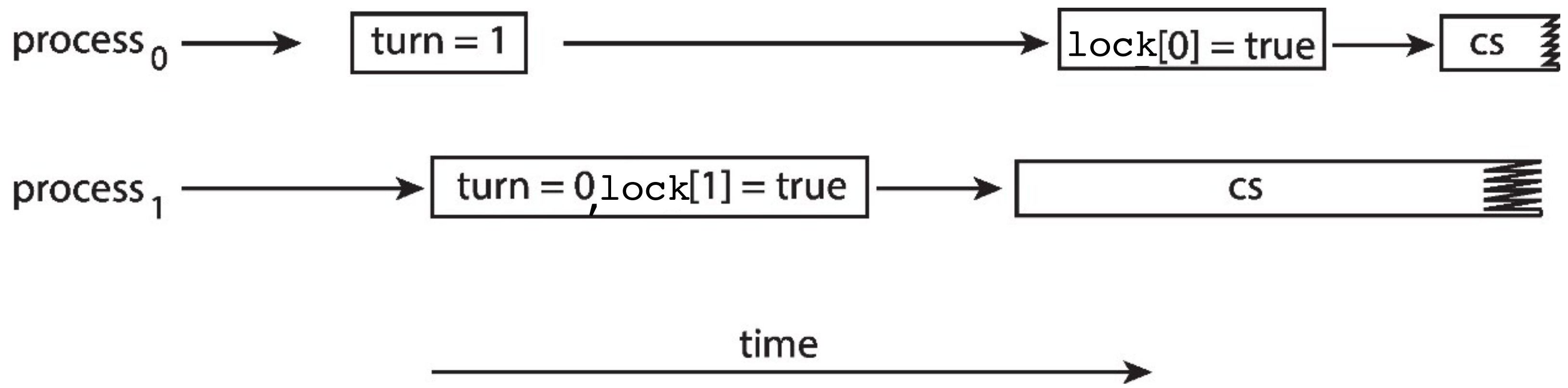
```
while(!flag)
  ;

print x
```

Thread B

```
x = 100
flag = false
```

Implementing critical sections in software is hard

- Peterson's algorithm works and the solution is symmetric
  - but it is limited to two threads
  - requires busy waiting
  - is not guaranteed to work on modern architectures!
    - processors and/or compilers may reorder operations that have no dependencies.
- This allows both threads to be in their critical section at the same time!



Why compilers or processors reorder instructions?

- compilers reorder instructions
  - efficient code generation requires analyzing control/data dependency
- CPUs reorder instructions
  - write buffering: allow next instruction to execute while write is being completed

Software-based solutions are not enough

- how to do better?
    - we can rely on hardware to make synchronization faster (eliminate busy waiting)
    - we can use higher-level programming abstractions to simplify concurrent programming
- a program with lots of concurrent threads can still have poor performance
    - lock contention: only one thread at a time can hold a given lock
    - how to address it? use fine-grained locking, i.e., partition the shared object into subsets, each protected by a lock

Hardware support

- many systems provide hardware support for implementing the critical section code
- on uniprocessor systems — we could disable interrupts
  - no preemption when running a critical section

Hardware support

- many systems provide hardware support for implementing the critical section code
- on uniprocessor systems — we could disable interrupts
    - no preemption when running a critical section
- we will look at three forms of hardware support
    - memory barriers: all operations before barrier complete before barrier returns
        - addresses the reordering problem: `op1; memory_barrier; op2;`
        - forces any change in memory to be propagated (made visible) to all other processors
    - atomic (uninterruptible) hardware instructions
    - atomic variables: providing atomic updates on basic data types such as integers and booleans
        - usually implemented using atomic hardware instructions

Synchronization by disabling interrupts

- two ways for the CPU scheduler to get control:
  - internal events: the thread does something to relinquish control (e.g., I/O)
  - external events: interrupts (e.g., time slice) cause the scheduler to take control away from the running thread

Synchronization by disabling interrupts

- two ways for the CPU scheduler to get control:
  - internal events: the thread does something to relinquish control (e.g., I/O)
  - external events: interrupts (e.g., time slice) cause the scheduler to take control away from the running thread
- on uniprocessors, we can prevent the scheduler from getting control as follows
  - internal events: prevent these by not requesting any I/O operations during a critical section
  - external events: prevent these by disabling interrupts (i.e., tell the hardware to delay handling any external events until after the thread is finished with the critical section)

Synchronization by atomic instructions

- read-modify-write is a class of instructions: atomically read a value from memory into a register and write a new value into it (the new value can be a function of the value read first or independent of it)
  - straightforward to implement on a uniprocessor: simply by adding a new instruction
  - on a multiprocessor, the processor issuing the instruction must be able to **invalidate** any copies the other processes may have in their cache
  - the multiprocessor system must support some type of **cache coherence**
    - keeping track of all copies, invalidating all copies on a write miss, reading the latest copy on a read miss

Synchronization by atomic instructions

- read-modify-write is a class of instructions: atomically read a value from memory into a register and write a new value into it (the new value can be a function of the value read first or independent of it)
  - straightforward to implement on a uniprocessor: simply by adding a new instruction
  - on a multiprocessor, the processor issuing the instruction must be able to **invalidate** any copies the other processes may have in their cache
  - the multiprocessor system must support some type of **cache coherence**
    - keeping track of all copies, invalidating all copies on a write miss, reading the latest copy on a read miss
- multiprocessor cache coherence
  - Thread A modifies data inside a critical section and releases lock; Thread B acquires lock and reads data
  - no problem if all accesses go to main memory
  - but what if new data is cached at processor A or old data is cached at processor B?

Examples of read-modify-write instructions

- test_and_set: read a value, write ‘1’ back to memory
  - BTS instruction in Intel x86

Examples of read-modify-write instructions

- test_and_set: read a value, write ‘1’ back to memory
  - BTS instruction in Intel x86
- exchange: swaps value between register and memory
  - XCHG instruction in Intel x86

Examples of read-modify-write instructions

- test_and_set: read a value, write ‘1’ back to memory
    - BTS instruction in Intel x86
- exchange: swaps value between register and memory
    - XCHG instruction in Intel x86
- compare_and_swap: read value, if value matches register r1 value, exchange it with register r2 value
    - CAS and CAS2 instructions in Motorola 68k
    - CMPXCHG in Intel x86 (with the LOCK prefix)

Test_and_set instruction

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv:
}
```

- returns the original value of the passed parameter
- sets the new value of the passed parameter to **`true`**

Critical section solution using test_and_set

```
while (true) {
    while (test_and_set(&lock))
      ; /* spin */
    /* critical section */
    lock = false;
}
```

Compare_and_swap instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

- operate on a memory word
- check that the value of the memory word hasn't changed from what you expect (**`*value == expected`**)
  - if it has not changed, set the memory word to a new value (**`new_value`**); the swap takes place only under this condition
- return the original value of passed parameter **`value`**

Critical section solution using compare_and_swap

```
while (true) {
    while(compare_and_swap(&lock, 0, 1) != 0)
      ; /* spin */
    /* critical section */
    lock = 0;
}
```

returns 1 when lock is obtained by another thread

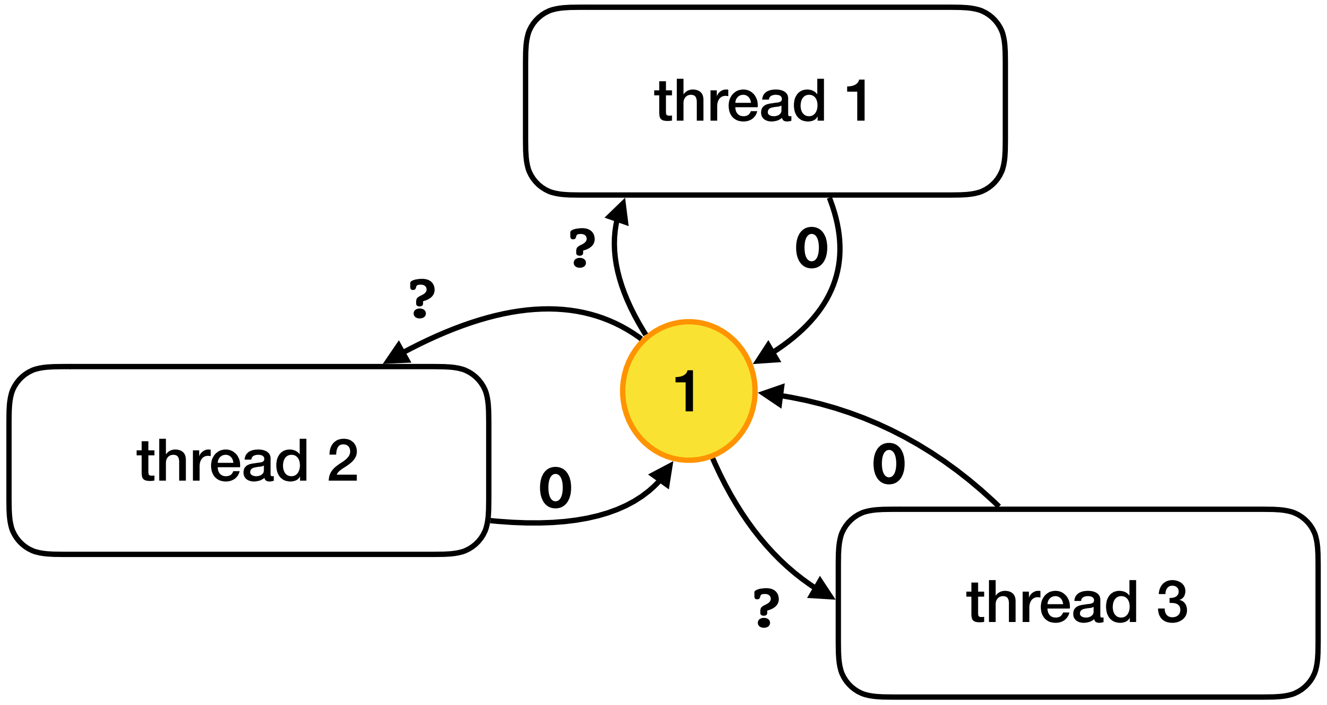
returns 0 when lock is free and sets the lock to 1

Critical section solution using compare_and_swap

```
while (true) {
    while(compare_and_swap(&lock, 0, 1) != 0)
      ; /* spin */
    /* critical section */
    lock = 0;
}
```

returns 1 when lock is obtained by another thread

returns 0 when lock is free and sets the lock to 1

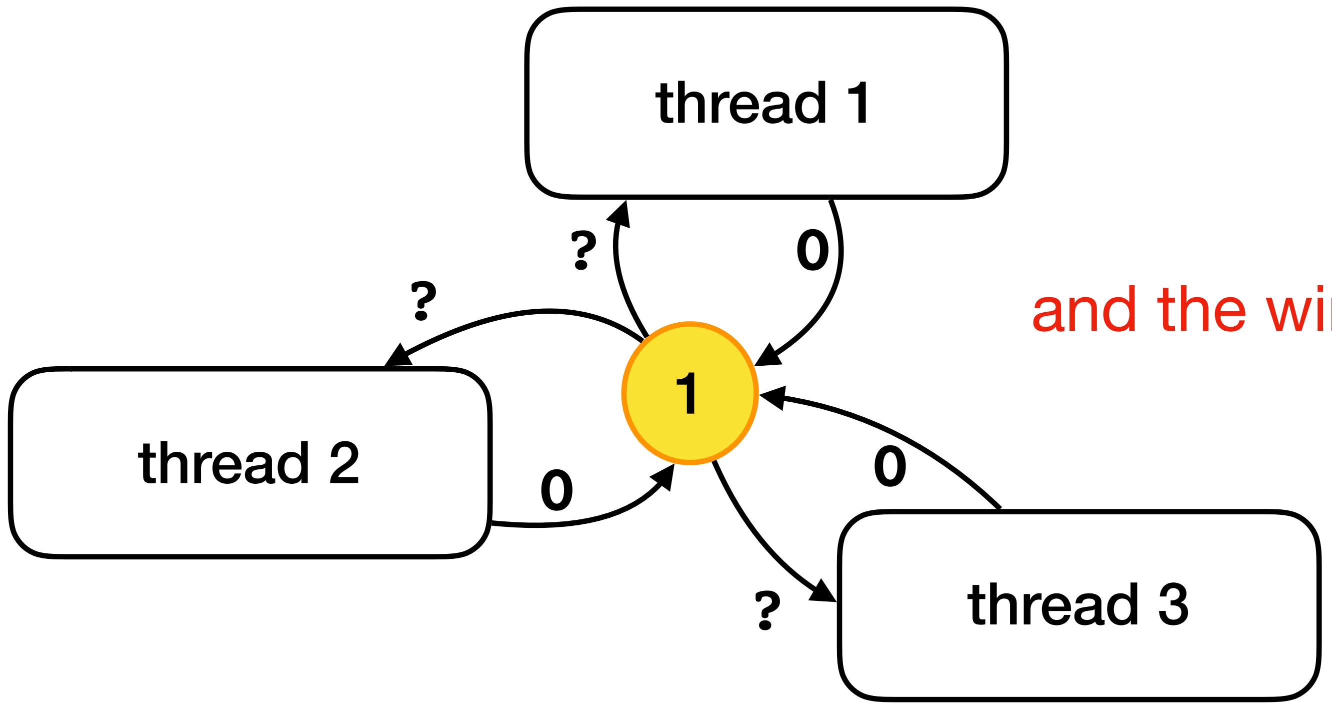


Critical section solution using compare_and_swap

```
while (true) {
    while(compare_and_swap(&lock, 0, 1) != 0)
      ; /* spin */
    /* critical section */
    lock = 0;
}
```

returns 1 when lock is obtained by another thread

returns 0 when lock is free and sets the lock to 1



and the winner is…

Problem with the previous solution?

from the three correctness properties, it doesn't have bounded waiting
- a thread can lose arbitrarily many times

idea: a thread can give the next thread a turn after exiting the CS

Problem with the previous solution?

from the three correctness properties, it doesn't have bounded waiting
- a thread can lose arbitrarily many times

idea: a thread can give the next thread a turn after exiting the CS

```
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
}
```

n is the number of threads

Problem with the previous solution?

from the three correctness properties, it doesn't have bounded waiting
- a thread can lose arbitrarily many times

idea: a thread can give the next thread a turn after exiting the CS

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
}
```

at least one of the two conditions must hold for entering the CS:
1. lock is obtained by Thread `i`
2. waiting[`i`] is false

n is the number of threads

break the loop if `j==i or waiting[j]=true`

let Thread `j` execute the CS if `j!=i` in this case lock shouldn't be released because it's held by Thread j

Atomic variables using compare_and_swap

- `increment( )` is an atomic function which can be implemented using `compare_and_swap`

```
void increment(atomic_int *v) {
  int temp;
  do {
    temp = *v;
  } while(temp != (compare_and_swap(v, temp, temp+1));
}
```

- C++ has atomic class template defined in `<atomic>`
  - defines type aliases for fundamental integral types (int, long, bool, …)
  - provides operations for all atomic types (e.g., load, store, exchange, compare_exchange, add, subtract, and, or, xor)