SCC.211 Operating Systems

Spinlocks and Barrier Synchronization

Dr. Amit Chopra
School of Computing & Communications, Lancaster University, UK
amit.chopra@lancaster.ac.uk

Objectives

Spin locks

- Problems and mitigation
- Blocking vs spinning
- Advantages and drawbacks

Barrier Synchronization

- Example

Spinlock

Each java object has an intrinsic lock

So far we have dealt with threads that block awaiting access to a shared resource

(This is implicit using *synchronized* keyword)

An alternative is to keep the thread active and continuously 'spinning' attempting to acquire the lock

- This is a **spin lock**
- (Potentially) improves threading performance and CPU usage

An Attempted Spinlock Implementation

Lancaster University

***lk == 0**
Lock is free

***lk == 1**
Lock is taken

```
void get_lock (int *lk)
{
        while (*lk ==1); // Do nothing (spin)
        *lk = 1; // Claim the lock
}

void release_lock(int *lk)
{
        *lk = 0; //Let someone else claim lock
}
```

Critical section is not accessed atomically

Contention on variable lk

P1 reads *lk ==0, drops out of while loop

Context switch occurs before P1 sets *lk to 1 (claim lock)

P2 is scheduled, runs through get_lock(), claims lock

P1 resumes, also claims the lock!

Spinlock execution

Lancaster University

*lk = 0;

Context Switch

while (*lk ==1);

while (*lk ==1);

*lk = 1;

*lk = 1;

return

return

Both threads read *lk = 0

Both acquire the lock

Violates mutual exclusion (we have two threads with the same lock!)

Solution

Disable interrupts

Special machine instruction ensuring atomicity

Software-only solution

In Java, would fix this by simply making get_lock() a critical section by using synchronize. Here we are trying to provide a lock with which synchronize might itself by implemented!

Disabling Interrupts

Pre-emptive context switch only happens when interrupt occurs

Could disable interrupts to prevent context switch in critical section

```
void get_lock() {disable_interrupts();}
```

```
void release_lock() {reenable_interrupts();}
```

Disabling Interrupts

While this works, comes with many disadvantages…

1. Interrupts might be disabled frequently and for a long time

– Clock ticks, I/O events could be missed

2. Will not be sufficient when you have more than one processor

- More than once thread could be running concurrently

3. Error proneness

– Forgetting to call release_lock() means interrupts disabled forever

Disabling Interrupts

Slightly better disable-interrupts-based spin lock implementation

```
void get_lock (int *lk)
{
    try_again:
    disable_interrupts();
        if (*lk ==1);                    // Lock taken
        {
          reenable_interrupts();         //permit context switch
          go to try_again;               //spin
        }
      *lk = 1;                           // Claim the lock
      Reenable_interrupts();
}

void release_lock(int *lk)
{
        *lk = 0;                         //Let someone claim lock
}
```

More fine grained
We only disable interrupts while accessing small critical section that reads/updates lk variable

However preceding disadvantages 1 and 2 still apply

Machine Instruction

test-and-set, comp-and-swap, fetch-and-add

Atomic machine instruction

- Sets variable passed to true,
- Tells if variable was true or false before being set to true

If n processes perform instruction, all set target value to true but only one returns false

```
boolean test_and_set(boolean *target)
{
    boolean orig_val = *target;
    *target = TRUE;
    return orig_val;
}
```

Performed atomically with hardware support – this is a software level example!

Machine Instruction: test_and_set() lock

```
void get_lock (boolean *lk)
{
    while(test_and_set(lk) == true);   // wait
}

void release_lock(boolean *lk)
{
        *lk = false;                        //Let someone claim lock
}
```

Software Only – Peterson's Algorithm

Assumes atomic reads and writes

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/synchro/node3.html

- Only works with two threads (can be generalized to n threads)
- Assumes thread ID are 0 and 1

```
int tiebreak = 0;                            /* shared variable */
bool[] interested = {FALSE, FALSE};          /* shared variable */

void get_lock() {

    int self = thread_getid();
    int other = 1 - self;

    interested[self] = TRUE;
    tiebreak = other;
    while(interested[other] && tiebreak == other) ; /* spin */
}

void release_lock() {
    int self = thread_getid()
    interested[self] = FALSE;
}
```

In green and pink are instructions from processes 0 and 1, respectively

Interested[0]=TRUE
Interested[1]=TRUE
tiebreak=1
tiebreak=0
Critical section
Interested[0]=FALSE
Critical section
Interested[0]=TRUE
tiebreak=1
Interested[1]=FALSE

(Continued from left)

Critical section
Interested[1]=TRUE
Interested[0]=FALSE
tiebreak=0
Critical section

…

Blocking vs spinning locks

Lancaster University

Blocking

Scheduler blocks threads while they wait

Good for long critical sections

Frequent queue management if locks accessed frequently

Spinning

Sit in a tight loop until lock acquisition

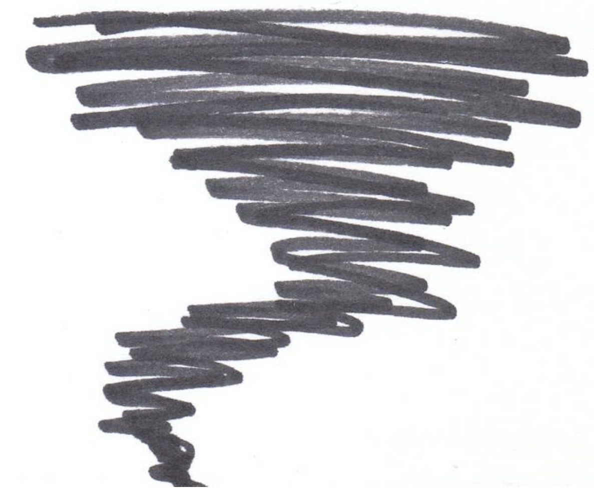
Good for short critical sections

Avoid queue management

Summary

Spin lock implementation

- Interrupt
- Hardware support (this is the most prevalent)
- Software only



Blocking or spinning locks?

- As always, depends on context
- Can result in massive performance degradation
- You’ll want to experiment within your own systems

Barrier Synchronization

- Threads wait for other threads to finish their tasks
  - Example
    - Many Worker threads, each of whom is assigned a file and counts the number of times “Aristotle” appears in the file
    - An Aggregator thread that totals up the counts for all files
    - Problem: If Aggregator totals before all Workers have finished, it will potentially output an incorrect total.
    - Solution: Make Aggregator wait for the Workers to finish their respective tasks
  - In coursework, the main thread must wait for adders and removers to finish before printing the final warehouse inventory

Waiting for a thread to finish: Joining with the thread

```
public static void main(String[] args)
    ...
    Thread c = new Thread(makeCoffee);
    Thread s = new Thread(shower);
    c.join();
    s.join();
    System.out.println("Hello World");
```

- Join is crude (happens when thread terminates)
- Let thread signal when done with task and then continue
  - More sophisticated kind of barriers
  - E.g., in Java, CountDownLatch
    - A latch is initialized with the number of tasks
    - Each Worker calls countDown() when done with its task
    - Aggregator blocks on await(); proceeds when latch count is 0