Operating Systems

Autumn 2024

sleeping locks

Recap: Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU

Recap: Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU
- running: executing on the CPU
  - is the thread that currently controls the CPU

Recap: Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU
- running: executing on the CPU
  - is the thread that currently controls the CPU
- waiting: waiting for an event, e.g. I/O
  - cannot make progress until event happens

Spinlock performance

- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again

Spinlock performance

- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later

Spinlock performance

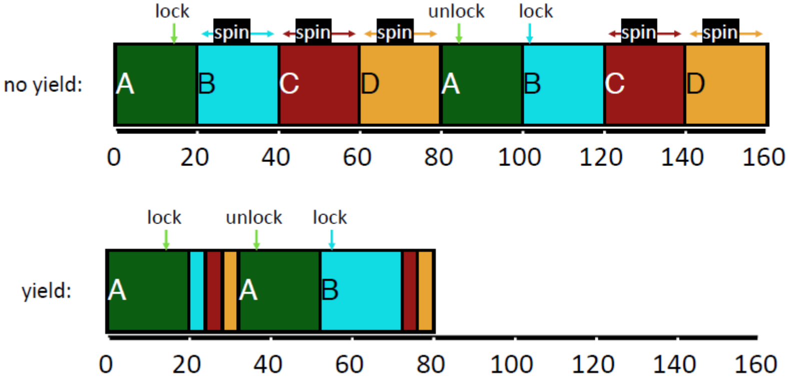
- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
    - `yield` system call moves calling thread from running to ready state

Spinlock performance

- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  - `yield` system call moves calling thread from running to ready state

```
void lock(lock_t *lock) {
    while (xchg(&lock->flag, 1) == 1)
        yield(); // give up the CPU
}
```

Why is yield() useful?



Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)

Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)
  - better than spinlock which wastes 99 time slices spinning

Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)
  - better than spinlock which wastes 99 time slices spinning
- Even with yield, high context switch cost

Blocking locks (aka mutexes): sleeping instead of spinning

- recall that scheduler only runs ready threads

Blocking locks (aka mutexes): sleeping instead of spinning

- recall that scheduler only runs ready threads
- idea: remove waiting threads from the scheduler's ready queue and put them on waiting queue
- no time wasted on threads that are contending on the lock

Blocking locks (aka mutexes): sleeping instead of spinning

- recall that scheduler only runs ready threads
- idea: remove waiting threads from the scheduler's ready queue and put them on waiting queue
- no time wasted on threads that are contending on the lock
- when lock is released, one thread on queue is restarted

Blocking locks: implementation idea

- 1 wait queue and 1 lock variable per blocking lock
- Implementing blocking lock gets tricky
  - Need to enforce mutual exclusion while performing operations on the wait queue (adding/removing) and the lock variable

Blocking locks: implementation idea

- 1 wait queue and 1 lock variable per blocking lock
- Implementing blocking lock gets tricky
  - Need to enforce mutual exclusion while performing operations on the wait queue (adding/removing) and the lock variable
  - how to implement this?

Blocking locks: implementation idea

- 1 wait queue and 1 lock variable per blocking lock
- Implementing blocking lock gets tricky
  - Need to enforce mutual exclusion while performing operations on the wait queue (adding/removing) and the lock variable
  - how to implement this?
  - use spinlock in the implementation of blocking lock!

Blocking locks: implementation idea

- Two separate levels of locking
  - holding spinlock guarding queue/variable from concurrent modification
  - holding actual blocking lock

Blocking locks: data structure

```
typedef struct {
  int flag;
  queue_t *q;
  int guard;
} lock_t;
```

tracks whether any thread has locked and not unlocked

Blocking locks: data structure

```
typedef struct {
    int flag;
    queue_t *q;
    int guard;
} lock_t;
```

list of threads that discovered lock is taken and are waiting for it to be free. these threads are not in scheduler's ready queue

Blocking locks: data structure

```
typedef struct {
  int flag;
  queue_t *q;
  int guard;
} lock_t;
```

used as a spinlock to protect `flag` and `q` manipulation

Blocking locks: OS support

- `park()`
  - Put a calling thread to sleep (remove it from runqueue and ready queue)

Blocking locks: OS support

- `park()`
  - Put a calling thread to sleep (remove it from runqueue and ready queue)
- `unpark(threadID)`
  - Wake a particular thread designated by `threadID`

Blocking locks: OS support

- `park()`
    - Put a calling thread to sleep (remove it from runqueue and ready queue)
- `unpark(threadID)`
    - Wake a particular thread designated by `threadID`

Blocking locks: OS support

- `park()`
  - Put a calling thread to sleep (remove it from runqueue and ready queue)
- `unpark(threadID)`
  - Wake a particular thread designated by `threadID`
- `park`, `unpark` inspired by Solaris

lock() code

```
void lock(lock_t *m) {
    while (xchg(&m->guard, 1) == 1)
        ; // acquire guard lock
    if (m->flag == 0) {
        m->flag = 1; // lock acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
typedef struct {
    int flag;
    int guard;
    queue_t *q;
}
```

- What is protected by the spin lock guard?

lock() code

```
void lock(lock_t *m) {
    while (xchg(&m->guard, 1) == 1)
        ; // acquire guard lock
    if (m->flag == 0) {
        m->flag = 1; // lock acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
typedef struct {
    int flag;
    int guard;
    queue_t *q;
}
```

- What is protected by the spin lock `guard`?
    - the `flag` and queue manipulation code
    - no more than a single thread can ever be active within that code

lock() code

```
void lock(lock_t *m) {
    while (xchg(&m->guard, 1) == 1)
        ; // acquire guard lock
    if (m->flag == 0) {
        m->flag = 1; // lock acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
typedef struct {
    int flag;
    int guard;
    queue_t *q;
}
```

- What is protected by the spin lock `guard`?
  - the `flag` and queue manipulation code
  - no more than a single thread can ever be active within that code
- Why is this better than a simple spin lock?

lock() code

```
void lock(lock_t *m) {
    while (xchg(&m->guard, 1) == 1)
        ; // acquire guard lock
    if (m->flag == 0) {
        m->flag = 1; // lock acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        park();
        m->guard = 0;
    }
}
```

```
typedef struct {
    int flag;
    int guard;
    queue_t *q;
}
```

- Setting `m->guard = 0` after calling park()
- Would it work?

unlock() code

```
void unlock(lock_t *m) {
    while (xchg(&m->guard, 1) == 1)
        ; // acquire guard lock
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock
    else
        // hold lock (for next thread!)
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

```
typedef struct {
    int flag;
    int guard;
    queue_t *q;
} lock_t;
```

- In unlock, there is no setting of flag=0 when we unpark. Why?

Race-condition bug

Thread 1 in `lock`

```
if (m->flag) {
   queue_add(m->q, gettid());
   m->guard = 0;




   park();
}
```

Thread 2 in `unlock`

```
while (xchg(&m->guard,1) == 1)
   ;
if (queue_empty(m->q))
   m->flag = 0;
else
   unpark(queue_remove(m->q));
```

- Assume Thread 2 is holding the lock
- Thread 1 calls lock()
- Before the call to park(), a switch to Thread 2 happens
- Thread 2 calls unlock() and does unpark()
- When Thread 1 calls park(), it sleeps forever (why?)

Solving the race problem: final correct lock

- `setpark()`
  - informs OS of my plan to `park()` myself
- If there is an `unpark()` between my `setpark()` and `park()`, `park()` will return immediately (no blocking)

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard=0;
park();
```

OS Support

- `park`, `unpark` and `setpark` inspired by Solaris
- Other OSes provide different mechanisms to support blocking synchronization
- E.g., Linux has a mechanism called **futex**
  - it renders `guard` and `setpark` unnecessary
- Read more about futex in OSTEP (brief)