Operating Systems

Tutorial 1

Michael Tänzer

os-tut@nhng.de
http://os-tut.nhng.de

7th December 2010

Outline

1. Synchronisation Basics
   - Terms
   - Valid Solutions
2. Concurrent Modifications
   - Many-to-One
3. Synchronisation Primitives
4. Shared Data Structures
5. Semaphores
6. Spinlocks

What is a race condition?

A situation where the correctness of a number of operations depends on the scheduling of the threads they're executed in.

Standard example:

```
current = get_balance();
current += delta;
set_balance(current);
```

How can race conditions be avoided?

How can race conditions be avoided?

- Make sure that “critical” operations are performed atomically

⇒ Use a synchronisation primitive (e. g. lock, semaphore)

⇒ Ensures that at most one thread can be inside a critical section

```
lock(L);
current = get_balance();
current += delta;
set_balance(current);
unlock(L);
```

Explain critical, entry, exit and remainder section

Explain critical, entry, exit and remainder section

Critical Section
- Process accesses common data (e. g. shared variable, file)
- No other process is allowed to execute related critical sections

Entry Section Process requests permission to enter its critical section

Exit Section Process signals completion of critical section

Remainder Section Code after the exit section

Enumerate and explain the requirements for a valid synchronisation solution

Enumerate and explain the requirements for a valid synchronisation solution

Exclusiveness	No two activities can enter a (related) CS
Progress	Activities in their remainder section do not prevent other activites from entering their CS
Bounded Waiting	An activity in its entry section will eventually get into its CS

Why does disabling interrupts for mutual exclusion not work on multiprocessor systems?

Why does disabling interrupts for mutual exclusion not work on multiprocessor systems?

- Interrupts can only be disabled per CPU
- Disabling interrupts doesn’t prevent all kernel entries (e. g. syscalls)

How can mutual exclusion whithin the kernel be guaranteed on multiprocessor systems?

How can mutual exclusion whithin the kernel be guaranteed on multiprocessor systems?

BKL – Big Kernel Lock

- One “big” lock that protects the entire kernel
- \+ Easy to implement $\Rightarrow$ good for porting a kernel from a single- to a multiprocessor system
- – No parallelism within the kernel $\Rightarrow$ limits scalability

Fine-grained locking for each data structure

- \+ Better parallelism
- – More error-prone (forget to acquire/release locks, deadlocks)

Determine the lower and upper bounds of `tally`

```
const int N = 50;
int tally;

void total() {
    for (int i = 0; i < N; ++i)
        tally += 1;
}

int main() {
    tally = 0;

    #pragma omp parallel for
    for (int i = 0; i < 2; ++i)
        total();

    printf("%d\n", tally);
    return 0;
}
```

What if `total` is executed by $t > 2$ threads?

What if `total` is executed by $t > 2$ threads?

Final value of `tally` would be $\in [2, 50 \cdot t]$

Would it make a difference if we used the Many-to-One model?

Would it make a difference if we used the Many-to-One model?

- When using the Many-to-One model threads are scheduled cooperatively

$\Rightarrow$ No real concurrency

$\Rightarrow$ Result always correct

What would be the output if we made the following modification to `total`?

```
void total()
{
    for (int i = 0; i < N; ++i) {
        tally += 1;
        sched_yield();
    }
}
```

What would be the output if we made the following modification to `total`?

```
void total()
{
    for (int i = 0; i < N; ++i) {
        tally += 1;
        sched_yield();
    }
}
```

Many-to-One Still correct as no thread is preempted between reading and writing `tally`

One-to-One Just a lot more context switches but still no guarantee that no thread is interrupted between reading and writing `tally`

Distinguish the types of synchronisation objects

Spinlock

Counting Semaphore

Binary Semaphore

Mutex

Monitor

Condition Variable

Distinguish the types of synchronisation objects

Spinlock	`lock()`/`unlock()`, wastes cycles ⇒ use only for short critical sections
Counting Semaphore	
Binary Semaphore	
Mutex	
Monitor	
Condition Variable	

Distinguish the types of synchronisation objects

Spinlock	`lock()`/`unlock()`, wastes cycles $\Rightarrow$ use only for short critical sections
Counting Semaphore	`wait()`: decrement internal counter counter $< 0 \Rightarrow$ block caller `signal()`: increment counter counter $\leq 0 \Rightarrow$ unblock one of the waiting threads
Binary Semaphore	
Mutex	
Monitor	
Condition Variable	

Distinguish the types of synchronisation objects

Spinlock	`lock()`/`unlock()`, wastes cycles $\Rightarrow$ use only for short critical sections
Counting Semaphore	`wait()`: decrement internal counter counter $< 0 \Rightarrow$ block caller `signal()`: increment counter counter $\leq 0 \Rightarrow$ unblock one of the waiting threads
Binary Semaphore	`wait()`/`signal()`, multiple `signal`s without a `wait` will be lost
Mutex	
Monitor	
Condition Variable	

Distinguish the types of synchronisation objects

Spinlock	`lock()/unlock()`, wastes cycles $\Rightarrow$ use only for short critical sections
Counting Semaphore	`wait()`: decrement internal counter counter $< 0 \Rightarrow$ block caller `signal()`: increment counter counter $\leq 0 \Rightarrow$ unblock one of the waiting threads
Binary Semaphore	`wait()/signal()`, multiple `signal`s without a `wait` will be lost
Mutex	Same as a binary semaphore
Monitor	
Condition Variable	

Distinguish the types of synchronisation objects

Spinlock	`lock()/unlock()`, wastes cycles $\Rightarrow$ use only for short critical sections
Counting Semaphore	`wait()`: decrement internal counter counter $< 0 \Rightarrow$ block caller `signal()`: increment counter counter $\leq 0 \Rightarrow$ unblock one of the waiting threads
Binary Semaphore	`wait()/signal()`, multiple `signal`s without a `wait` will be lost
Mutex	Same as a binary semaphore
Monitor	Language-based $\Rightarrow$ no explicit operation
Condition Variable	

Distinguish the types of synchronisation objects

Spinlock	`lock()/unlock()`, wastes cycles $\Rightarrow$ use only for short critical sections
Counting Semaphore	`wait()`: decrement internal counter counter $< 0 \Rightarrow$ block caller `signal()`: increment counter counter $\leq 0 \Rightarrow$ unblock one of the waiting threads
Binary Semaphore	`wait()/signal()`, multiple `signal`s without a `wait` will be lost
Mutex	Same as a binary semaphore
Monitor	Language-based $\Rightarrow$ no explicit operation
Condition Variable	`wait()/signal()`, similar to binary semaphore but tied to a monitor

Example situation

- *n* threads concurrently access
  a Doubly linked list
  b Binary tree
- Nodes describe customer records and their contents will never change
- Nodes are added to the data structures maintaining sort order
- Before a node is added it's only visible to the thread creating it

Explain how the data structures are vulnerable to race conditions

How to avoid them?

Explain how the data structures are vulnerable to race conditions

How to avoid them?

- Multiple pointers have to be updated $\Rightarrow$ need atomicity
- Alphabetic order leeds to race condition between finding insertion point and the insertion $\Rightarrow$ check again after locking

Must a node have valid data before it's added?

Must a node have valid data before it's added?

- If reading doesn't require a lock $\Rightarrow$ data has to be valid on insertion as readers could access invalid data otherwise
- If reading requires locking $\Rightarrow$ creating thread just acquires that lock before inserting the node and releases it once the data becomes valid

Is there a problem with the following implementation of a generic semaphore?

```
void c_wait(semaphore *s) {
    b_wait(mutex);
    *s -= 1;
    if (*s < 0) {
        b_signal(mutex);
        b_wait(delay);
    } else {
        b_signal(mutex);
    }
}

void c_signal(semaphore *s) {
    b_wait(mutex);
    *s += 1;
    if (*s <= 0) b_signal( delay );
    b_signal( mutex );
}
```

Why does the chess programme run significantly slower if threads T execute concurrently?

- 4 processor SMP system with per processor L1 and L2 cache
- P_4 executes grand master chess programme, working set doesn't fit into L2
- $P_1, \ldots, P_3$ run cooperating treads T which synchronize with spinlocks:

```
do {
    reg = myThreadId;
    /* atomically exchange shared variable 'spinlock' and
    swap(&spinlock, reg);
} while (reg != 0);

/* critical section */

spinlock = 0;
```

How can you improve the software?

How can you improve the software?

```
do
{
    reg = myThreadId;

    /* spinlock can be read from own cache until it
    while( spinlock != 0 )
        ;

    /* now we got spinlock = 0, try to acquire the
    swap( &spinlock, reg );

} while ( reg != 0 ); /* otherwise another thread w

/* critical section */
```

Questions & Comments

Any questions or comments?

Politically Correct UNIX – Part II

System VI Release Notes

- `history` has been completely rewritten and is now called `herstory`
- `quota` can now specify minimum as well as maximum usage, and will be strictly enforced.
- The `abort()` function is now called `choice()`
- The biodegradable `KleeNeX` displaces the environmentally unfriendly `LaTeX`
- To avoid unpleasant, medieval connotations, the `kill` command has been renamed `euthanise`
- From now on, “rich text” will be more accurately referred to as “exploitive capitalist text”
- The term “daemons” is a Judeo-Christian pejorative. Such processes will now be known as “spiritual guides”