Operating Systems

Tutorial 2 & 16

Michael Tänzer

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

Calendar Week 49

Outline

True or False

- When using round robin starvation can never happen.
- Shortest job first provides the minimal turnaround time among all non-preemptive scheduling algorithms.
- As multilevel feedback queue is a variation of priority scheduling it is well-suited for real time systems.

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

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?

- Interrupts only 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?

One ‘big’ lock that protects the entire kernel (BKL – Big Kernel Lock)

- \+ Easy to implement ⇒ good for porting a kernel from a single- to a multiprocessor system
- – No parallelism within the kernel ⇒ limits scalability

Fine-grained locking for each data structure

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

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?

- 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?

- 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

Determine the lower and upper bounds of the final value of `tally`

```
const n = 50;
var tally : integer;

procedure total;
var count : integer;
begin
  for count := 1 to n do tally := tally + 1
end;

begin (* main program *)
  tally := 0;
  parbegin
    (* two threads executing in parallel *)
    total; total
  parend;
  write(tally)
end.
```

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`?

```
procedure total;
var count : integer;
begin
  for count := 1 to n do begin
      tally := tally + 1;
      yield
  end
end;
```

⇒ still correct as no thread is preempted between reading and writing `tally`

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 reg *)
    swap(&spinlock, reg);
until (reg = 0);
(* critical section *)
spinlock := 0;
```

How can you improve the software?

```
do
    reg := myThreadId;
    (* spinlock can be read from own cache
    until it is updated by s.o. else *)
    while (spinlock != 0) do (* wait *) od
    (* now we got spinlock = 0,
    try to acquire the lock *)
    swap(&spinlock, reg);
until (reg = 0);
(* otherwise another thread was faster... *)
(* critical section *)
spinlock := 0;
```

What's the problem with the following implementation of a generic semaphore?

```
procedure c_wait(var s:semaphore)
begin
    b_wait(mutex);
    s := s - 1;
    if (s < 0) then begin
        b_signal(mutex);
        b_wait(delay)
    end else
        b_signal(mutex)
end;

procedure c_signal(var s:semaphore)
begin
    b_wait(mutex);
    s := s + 1;
    if (s <= 0) then b_signal(delay);
    b_signal(mutex)
end;
```

Questions & Comments

Any questions or comments?

The End

The End