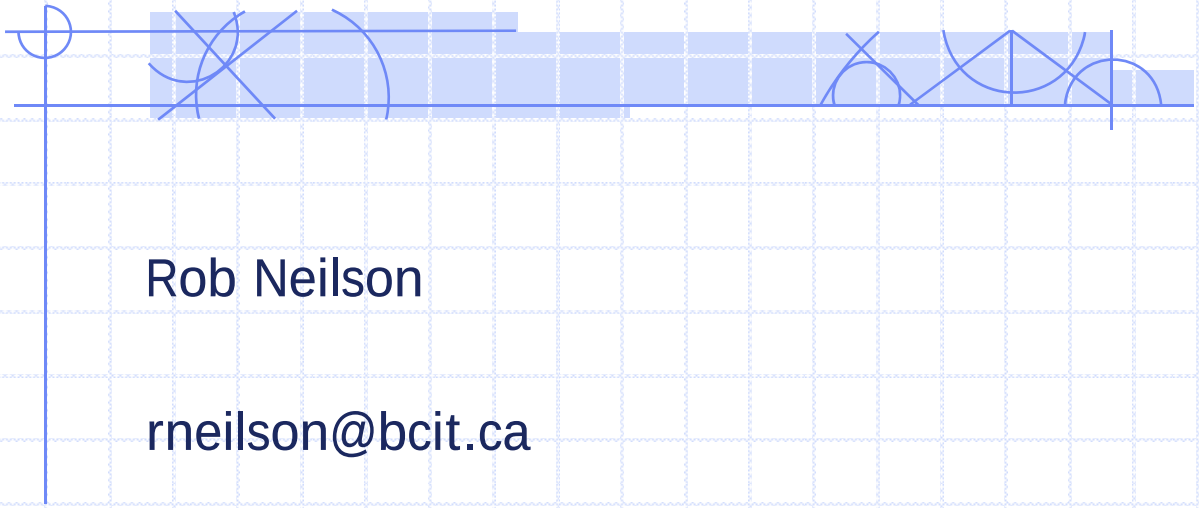
COMP 4735: Operating Systems

Lesson 8.2: Implementing Mutual Exclusion



Rob Neilson

rneilson@bcit.ca

Administration stuff …

- This weeks reading: Chp 2.3
- Next week
  - no quiz next week
  - IPC is a pretty big topic; we will have the quiz in two weeks

Agenda: Thursday Feb 4

- Strict Alternation
- Peterson's Solution
- The TSL Instruction
- Review

Recall the Critical Section Problem

- we have looked at a few different ways to solve this problem
- the last thing we looked at was a locking mechanism that disabled interrupts to make the test and set and atomic operation
- we needed to do this because both threads could enter their critical sections at the same time if a race condition developed to set the lock ...

Code for t_1	Code for t_2
`while(lock) {};` `lock = TRUE;`	`while(lock) {};` `lock = TRUE;`

- *OK - so what would happen if we changed the while condition so that only one thread could get the loc at any point in time ... ?*

Solution 3: Strict Alternation

Idea:

- allocate a shared variable that is a ‘lock’
- design the entry to the critical regions so that one thread enters when the lock is set, and the other when it is not set
- in other words, force the threads to ***TAKE TURNS***

We make the threads take turns by only letting a thread enter its critical section when the other thread says it is OK to do so!

Strict Alternation Example (page 122)

```
shared boolean turn = 1;


Code for t1

while (TRUE) {
   while(turn != 0) {};      // wait for my turn
   critical_region();        // enter critical region
   turn = 1;                 // tell other thread it
  ...
                             // ... is its turn now
   noncritical_region();     // do other stuff
}
```

Other thread will do the same, but use opposite values in *turn* variable.

Strict Alternation Example (page 122)

```
shared boolean turn = 1;
```

Code for t_1

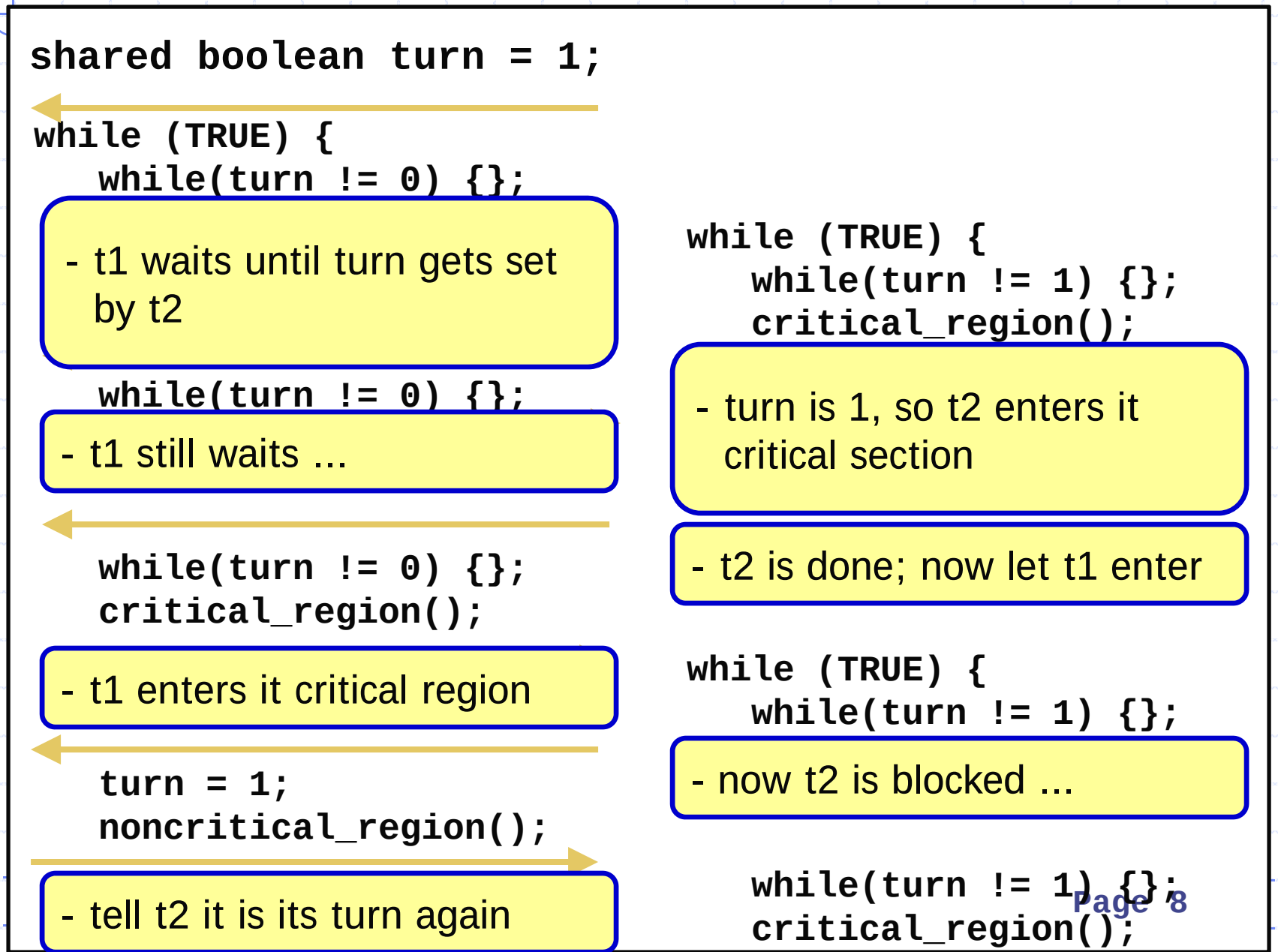
```
while (TRUE) {
   while(turn != 0) {};
   critical_region();
   turn = 1;
   noncritical_region();
}
```

Code for t_2

```
while (TRUE) {
   while(turn != 1) {};
   critical_region();
   turn = 0;
   noncritical_region();
}
```

Will this work?

Strict Alternation Ex. (Timing Diag)



Strict Alternation: Does it work?

- Yes, it works
- Threads will always take turns
- Cannot get lock (enter critical region) until other thread says it is OK
- The setting of turn is done at end of critical region which means
  - other thread cannot also be trying to update the variable - it must be waiting, or doing something non-critical

Strict Alternation: A Problem

- There is still a problem:
  - t1 thread could be blocked, and cannot enter its region until t2 says it is OK
  - but t1 could wait a long time as t2 might be doing something non-critical
  - essentially:
    - t1 wants to run twice in a row and t2 doesn't want to enter its own critical region
    - therefore t1 blocks t2 and does not let it proceed, until after it gets its next turn
    - this violates one of the four premises for mutual exclusion

Peterson's Solution

- this solution combines the idea of taking turns with the idea of lock variables

```
shared int turn = 1;
shared int interested[2];          // 2 procs: 0 and 1

void enter_region(int process)     // call with your own ID
{
   int other = 1 - process;        // calc ID of other process
   interested[process] = TRUE;     // indicate 'I want in'
   turn = process;                 // say 'it is my turn next'

   while (turn == process && interested[other] == TRUE);
}
```

- enter if: other process is not interested, or
  other process is waiting for its turn

```
void leave_region(int process)
{  interested[process] = FALSE; } // let other process enter
```

Test and Set Lock

- locks give us a solution that allows
  - mutual exclusion
  - synchronization
- we have seen two solutions that work
  - Peterson's solution
  - simple locks with enable/disable interrupts
- since locks are used quite a bit, many processors include a special machine instruction that effectively implements an atomic test and set operation

`TSL REGISTER, LOCK`

This instruction does two things:

1. copies current contents of LOCK (a memory location) to REGISTER
2. sets the value of LOCK to be non-zero (1)

Test and Set Lock (how to use)

- if an OS wants to use TSL, the OS developers need to implement some lock and unlock primitives using TSL
- the basic idea is:
  - when a process wants to get the lock it
    - execute TSL
    - look at the value in REGISTER after the TSL
    - if the value is non-zero, another process has the lock, and we wait and try again later
- here is the code:

```
enter_region:
   TSL REGISTER, LOCK
   CMP REGISTER,#0
   JNE enter_region
   RET
```

```
leave_region:
   MOV LOCK,#0
   RET
```

Locks - an Example

- OK - so we have locks; how do we use them
- let us assume that we have primitives:
  - enter_region(lock)
  - exit_region(lock)
- we the use them around our shared variables, for example:

```
shared boolean lock = FALSE;
shared double balance;
```

Code for p_1

```
...
enter_region(lock);
balance = balance + amount;
exit_region(lock);
...
```

Code for p_2

```
...
enter_region(lock);
balance = balance - amount;
exit_region(lock);
...
```

Review (what we have learned so far) - 1

1. Why do modern computers use concurrent execution models?

   Concurrent execution of processes/threads is required to maximize use of the processor and exploit potential parallel operations.

2. What is a shared variable, and why do we need such things?

   Shared variables can be read/written by multiple threads/processes.

   They enable IPC (inter-process communication) by allowing us to share information between threads/processes.

3. What is thread synchronization?

   This is when we have multiple threads and we want them to coordinate their actions so that the result of the concurrent execution of these threads is deterministic.

Review (what we have learned so far) - 2

4. What is a Race Condition?

   A race condition is an error situation that can occur when two threads or processes try to update the same shared variable. It occurs because thread can be preempted when it is only part way through a high level programming language instruction.

5. What is Mutual Exclusion?

   Mutual exclusion is when we (programmers) force the execution of a section of code to proceed sequentially so as to control access to shared variables. Such a section of code is called a “critical section”.

6. What is the Critical Section Problem?

   The critical section problem is the problem of coming up with OS constructs and mechanisms to enable critical sections of code to execute in a mutually exclusive manner.

Review (what we have learned so far) - 3

7. How does disabling interrupts solve the critical section problem?

   By disabling interrupts at the start of a critical section, and enabling at the end of the critical section, we prevent the CPU from interrupting the thread during the critical section.

8. How can we use a lock variable to solve the critical section problem?

   We can define a variable (*lock*) that is shared between the processes. Any process that wants to update another shared variable must first set the lock – which will prevent other processes from accessing the shared variable. The lock should be removed when the first process is finished updating the shared variable.

Review (what we have learned so far) - 4

9. What type of problem can be induced if we use divisible Test and Set instructions in the implementation of locks?

   This can case a race condition, as a process can be preempted after it tests the lock, but before it sets it.

10. What is an Atomic Test and Set, and where would we use it?

    Atomic test and set is a routine or instruction that can be used to test a shared variable (such as a lock) and set it to a value. The test and set actions are performed as if they were one operation (ie: they operate in a mutually exclusive manner).

    We can use atomic test and set to implement locks to solve the critical section problem.

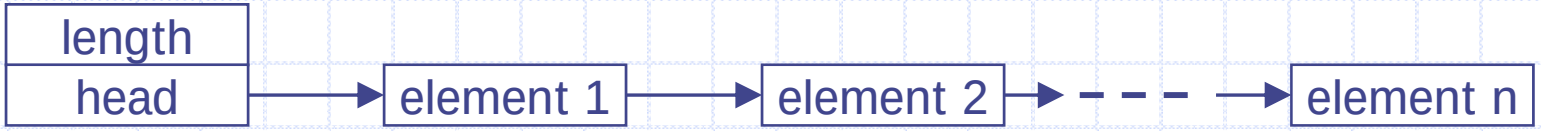
Locks and Multiple Resources

- in the last couple of lessons we came up with a locking solution to ensure sequential access to shared resources
- our solution involved locks, specifically
  - execute enter(lock) before updating the shared variable
  - execute exit(lock) immediately after updating the shared variable
- let's see what happens if we try and use this approach on a problem that includes more than one shared variable ...

Multiple Resource Problem (1)

Sample problem:

- assume two threads exist to manipulate a common linked list
  - thread 1 is responsible for deleting elements
  - thread 2 is responsible for adding elements
- each thread needs access to two shared variables
  - the list header, which contains the length, and
  - the list itself (to add or delete nodes)



delete: remove element from list, update length
add: update length, add element to end of list

Solution attempt 1: hold one lock at a time

```
shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;
```

thread 1: delete

```
  . . .
enter(lock1);
  <delete element>
exit(lock1);
  <other code>;
enter(lock2);
  <update length>
exit(lock2);
  . . .
```

thread 2: add

```
  . . .
enter(lock2);
  <update length>
exit(lock2);
  <other code>
enter(lock1);
  <add element at end>
exit(lock1);
  . . .
```

Basic idea: get the lock when you need to update a shared var, release lock as soon as you are done.

Solution attempt 1: hold one lock at a time (2)

thread 1: delete

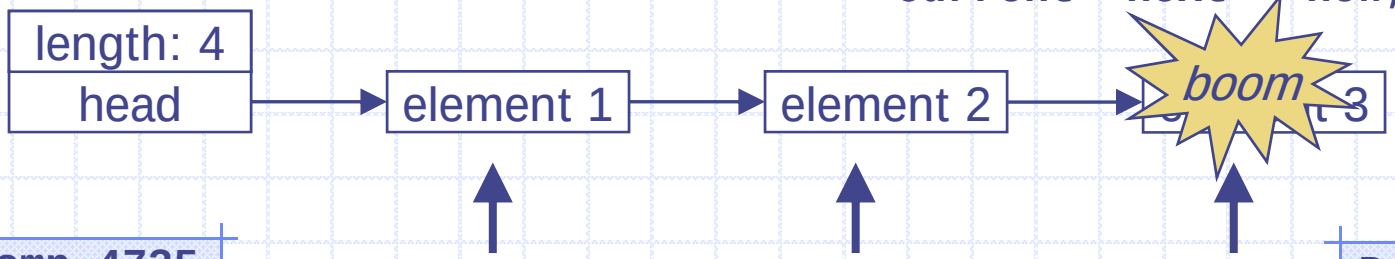
thread 2: add

ts=1

```
enter(lock1);
  <delete element 3>
exit(lock1);
  <other code>;
```

ts=2

```
enter(lock2);
  <length++>
exit(lock2);
  <other code>
enter(lock1);
  <add element at end>
  for i=1 to length-1
    current = current->next;
  current->next = new;
```



Solution attempt 2: holding two locks

```
shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;
```

thread 1: delete

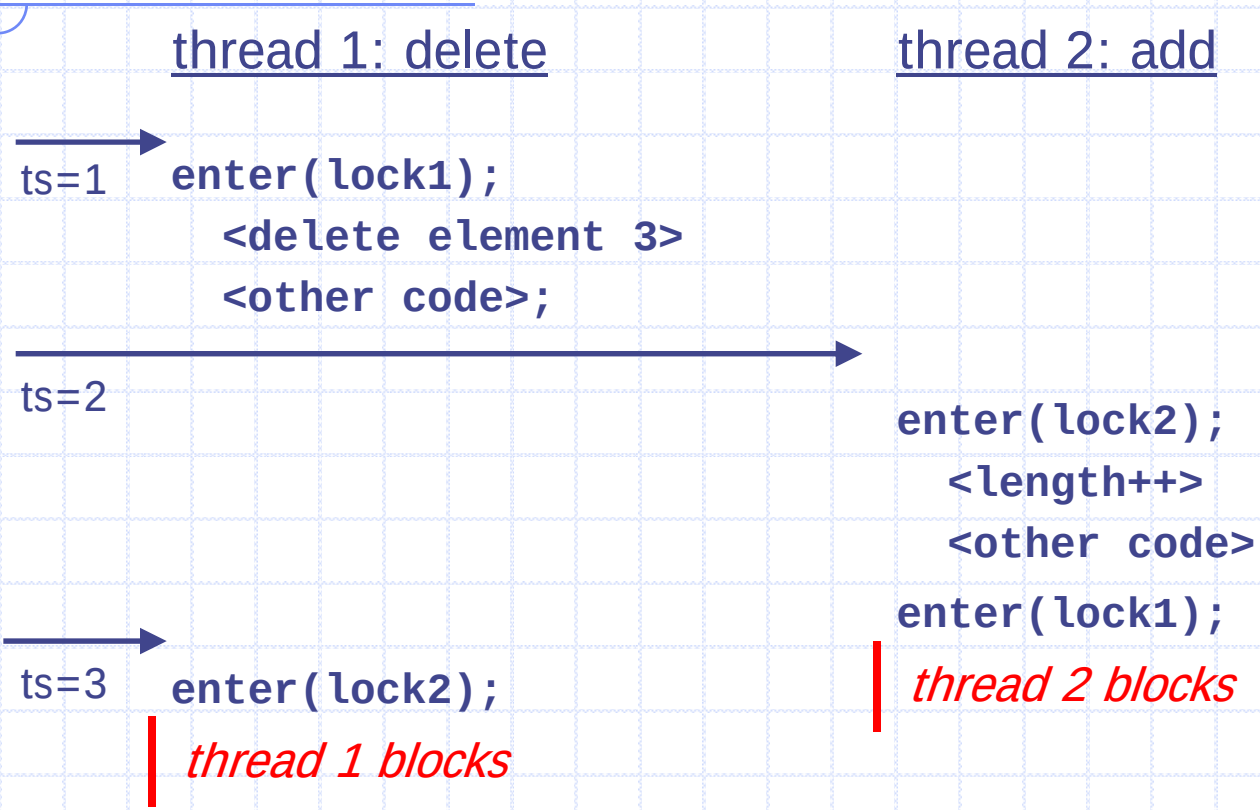
```
  . . .
enter(lock1);
  <delete element>
  <other code>;
enter(lock2);
  <update length>
exit(lock1);
exit(lock2);
  . . .
```

thread 2: add

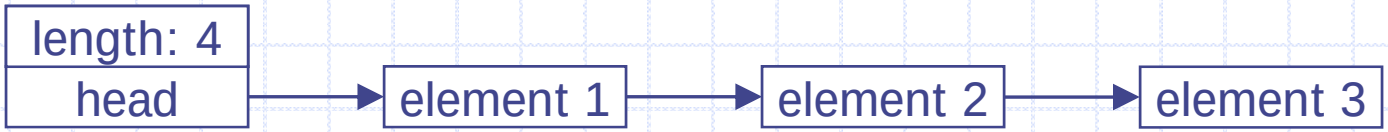
```
  . . .
enter(lock2);
  <update length>
  <other code>
enter(lock1);
  <add element at end>
exit(lock2);
exit(lock1);
  . . .
```

Basic idea: get the lock when you need to update a shared var, hold until entire operation is done.

Solution attempt 2: holding two locks (2)



Now we are in a deadlock situation – neither thread can progress any further!



The Deadlock Problem

- deadlock is a situation that occurs when threads are waiting on each other – but they each have a resource that the other needs
- for deadlock to occur, we need at least two resources
  - each thread will have one resource locked while it is waiting on another
- *we need an approach to synchronization that provides a solution to the deadlock problem*

Solution 3: get all locks before beginning

```
shared boolean lock1 = FALSE;
shared boolean lock2 = FALSE;
shared list L;
```

thread 1: delete

```
  . . .
enter(lock1);
enter(lock2);
  <delete element>
  <other code>;
  <update length>
exit(lock2);
exit(lock1);
  . . .
```

thread 2: add

```
    . . .
enter(lock1);
enter(lock2);
  <update length>
  <other code>
  <add element at end>
exit(lock2);
exit(lock1);
    . . .
```

This approach will work – but not without shortcomings.

What do you think could be problematic with this approach?

The End