COMP 4735: Operating Systems

Lesson 8.1: Race Conditions & Critical Sections

Rob Neilson

rneilson@bcit.ca

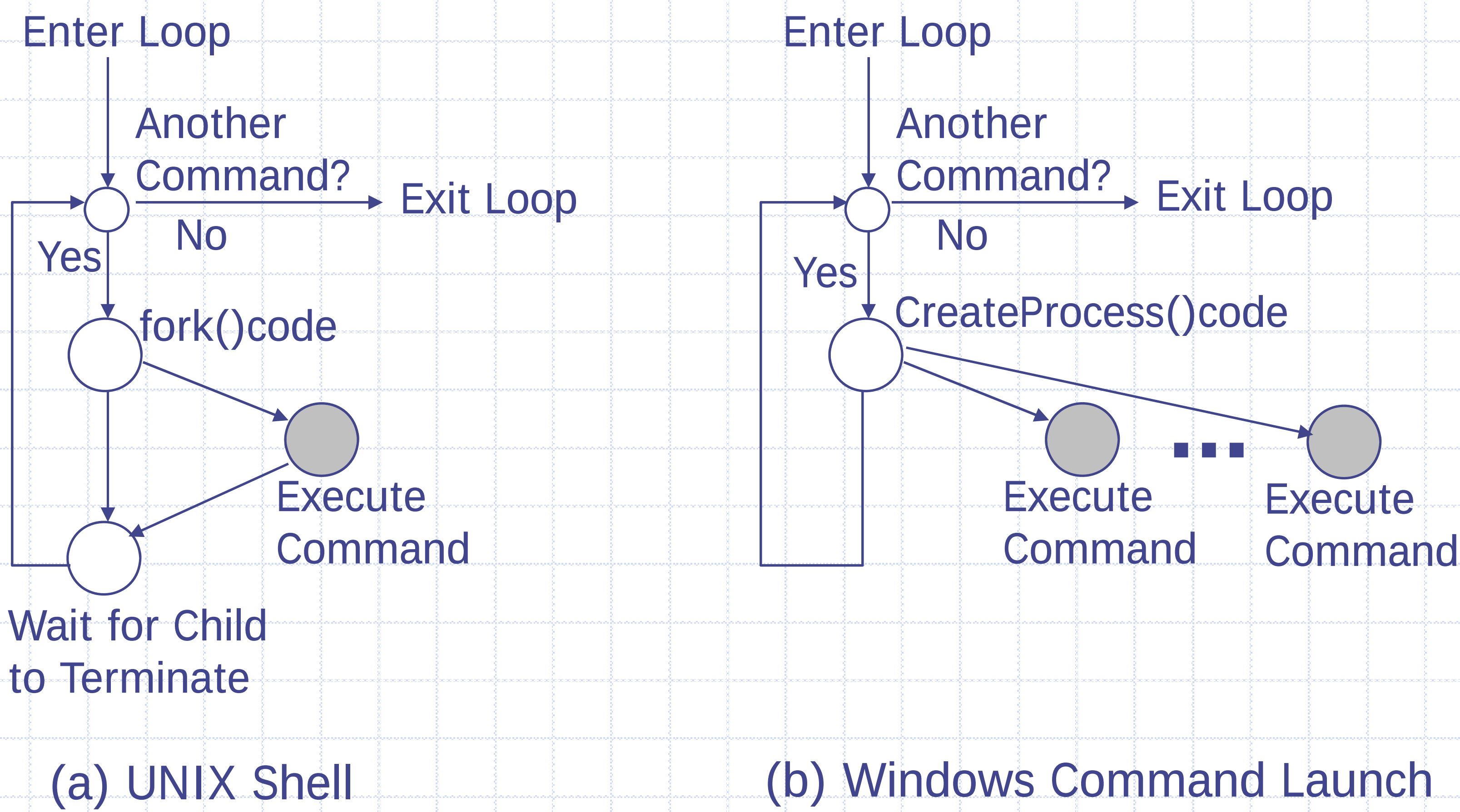
Administration stuff …

- 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: Monday Feb 19

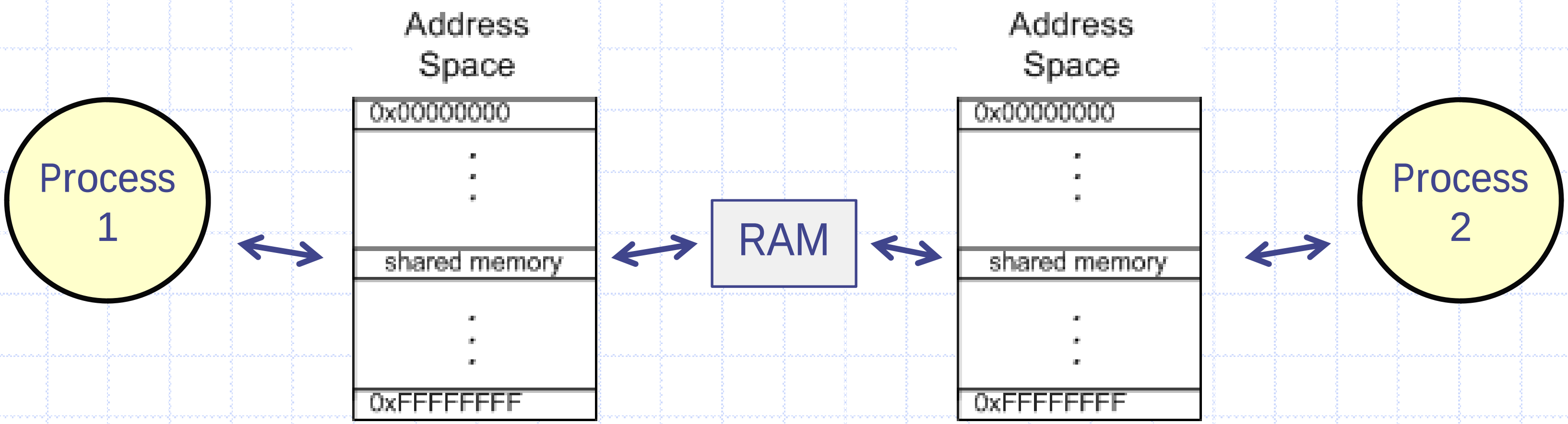
- critical sections
- race conditions
- interrupt-based solutions
- locks and locking

Concept 1: Concurrent Execution



Concept 2: Shared Variables / IPC

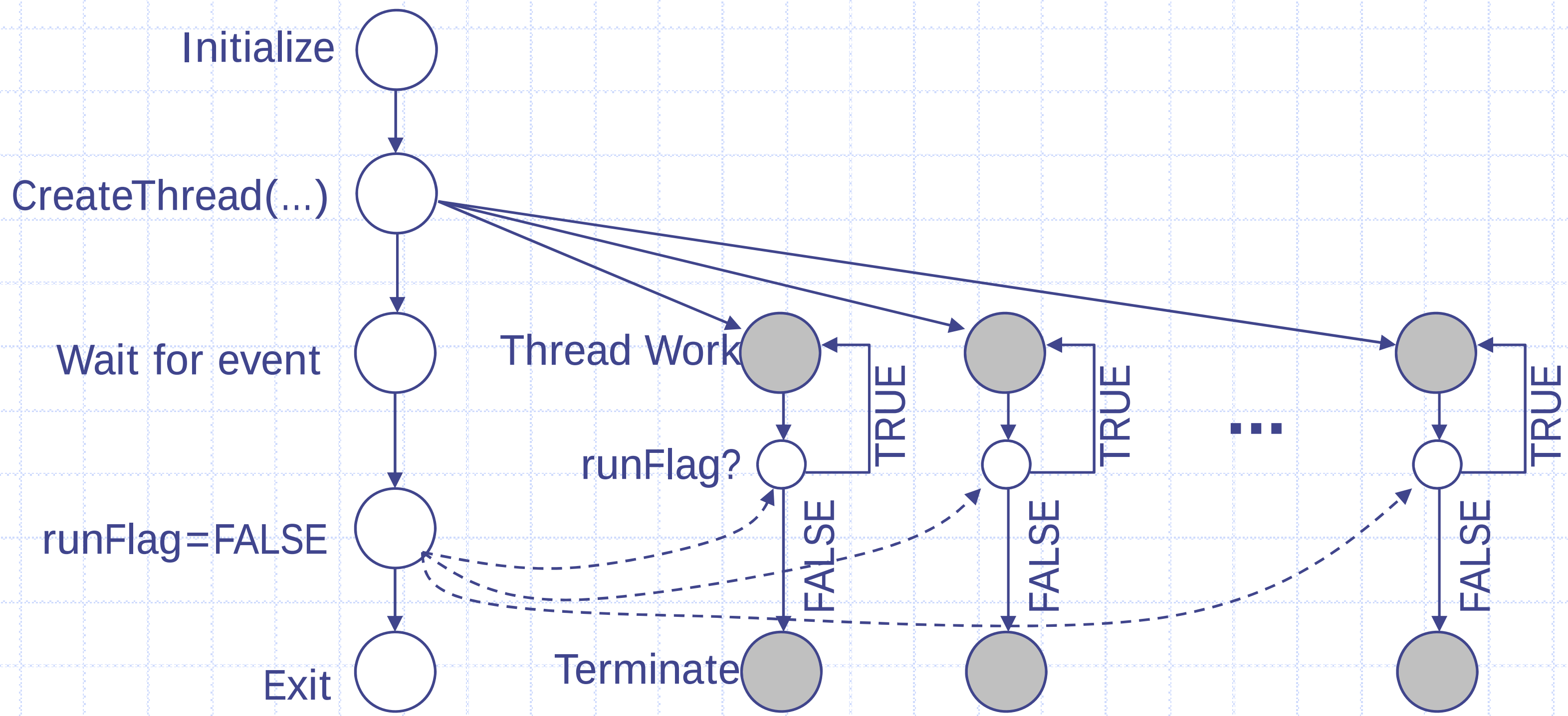
- assume you have two processes or threads
- there are situations where they need to be able to communicate to each other, ie, they need to be able to share information
- a simple way might be to allow both processes to read/write a special variable that is at a pre-defined location in memory
- we call this variable a shared variable



Threads and Shared Variables

- the concept of shared variables is easy with threads, as they are already in the same address space
- simply define a global and all threads can access it

Concept 3: We can use a Shared Variable to Synchronize Threads



- In this example the parent starts a bunch of worker threads.
- The threads work away, periodically checking the runFlag.
- If the parent has set the flag to FALSE, threads terminate.

Problem 1: Race Conditions

- race conditions occur when *two processes* or threads are *attempting to update* a shared variable, and
  - one process gets pre-empted part way through the update operation

- for example, consider the question that was asked on the previous slide. The code for the threads was

```
runFlag++;
```

  - this code might translate into the following machine instructions

```
load  R1, runFlag
incr  R1
store R1, runFlag
```

Race Condition Example 1

- assume two threads want to execute runFlag++, but the first one gets pre-empted part way through …

Execution of t_1

```
…
load  R1, runFlag
```

Timer interrupt

Execution of t_2

```
…
load  R1, runFlag
incr  R1
store R1, runFlag
…
```

Timer interrupt

```
incr  R1
store R1, runFlag
…
```

Race Condition Example 2

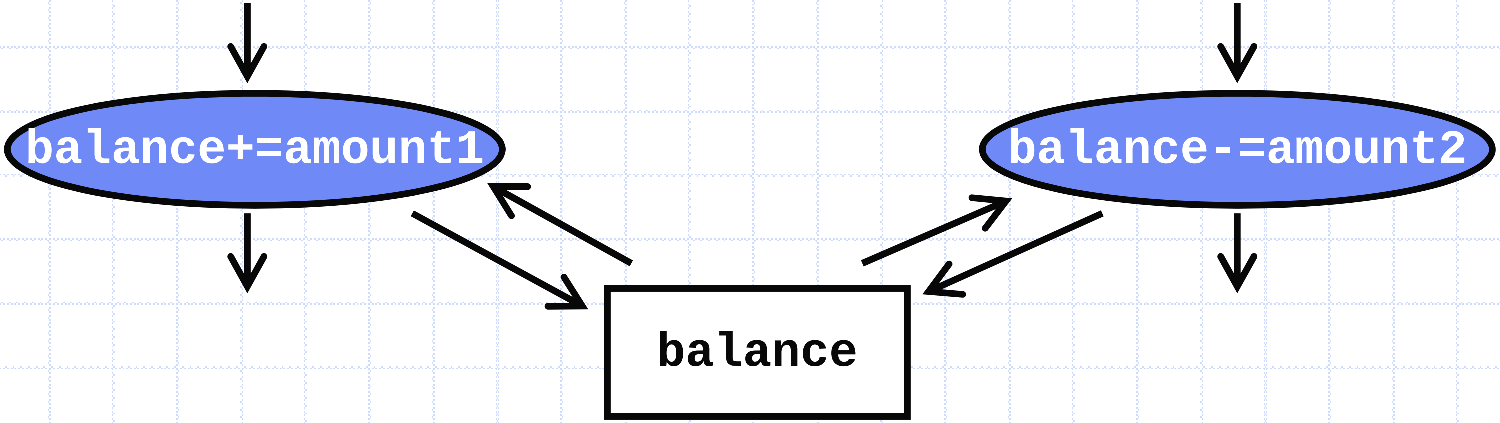
```
shared double balance;
```

<u>Code for p_1</u>

```
  . . .
balance = balance + amount1;
  . . .
```

<u>Code for p_2</u>

```
  . . .
balance = balance – amount2;
  . . .
```



- Assume that balance=100, amount1=3, amount2=2
- *What is the value of balance after this code runs?*

Race Condition Example 2

Execution of p_1

```
…
load  R1, balance
load  R2, amount1
```

Timer interrupt

Execution of p_2

```
…
load  R1, balance
load  R2, amount2
sub   R1, R2
store R1, balance
…
```

Timer interrupt

```
add   R1, R2
store R1, balance
…
```

Concept 4: Mutual Exclusion

- to prevent race conditions, we introduce the idea of *mutual exclusion*

Mutual exclusion:

  - if one process/thread is using a shared variable/file/etc, other processes/threads will be prevented from using the same variable/file/etc at the same time
  - ie: we force the processes/threads to *operate sequentially* on sections of code that could lead to race conditions

- the sections of code that have the potential to cause race conditions are called *critical sections*

Problem 2: Critical Section Problem

To prevent race conditions, we need a mechanism in the OS to enable critical sections of code to be executed such that:

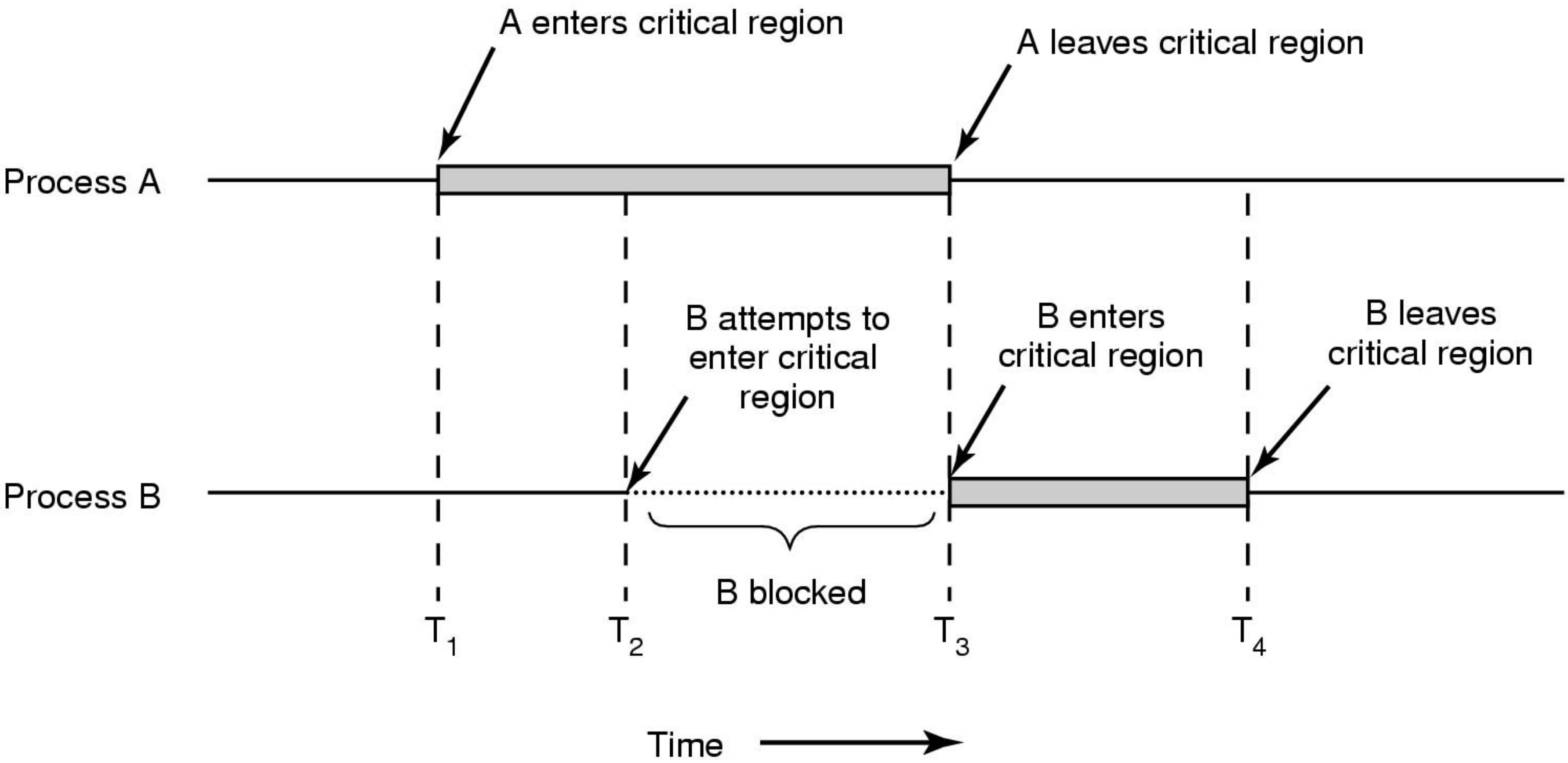
1. no two processes may be simultaneously inside their critical sections
2. no assumptions may be made about speeds or the number of CPUs
3. no process running outside its critical section may block another process
4. no process should have to wait forever to enter its critical section

In-class Exercise: Find the Critical Section

```
shared int childCnt = 0;
shared int parentCnt = 0;
int child;
int i = 0;
while (i<4) {
   i++;
   child = fork();
}
if (child == 0)
   childCnt++;
else {
   parentCnt++;
   wait();
}
while (childCnt < parentCnt)
   sleep(1000);
printf("%d %d\n", childCnt, parentCnt);
exit;
```

1. how many processes are created in total?
2. what should be the value of parentCnt at the end?
3. what happens if the child exits before wait?
4. identify all the critical sections in this program?

Review: Expected Behaviour of Critical Sections



Possible Solutions to the Critical Section Problem

- We can achieve mutual exclusion over sections of code by:
  - Disable interrupts
  - Software solution – locks
  - Test and Set
  - Semaphores / Mutexes
  - Monitors

Solution 1: Disabling Interrupts

Idea: - don't let the process be pre-empted during its critical section.

- use the OSs ability to disable interrupts to achieve this.

How to do it:

1. find the critical sections,
2. encapsulate in disable/enable interrupt system calls.

Consider 'balance update example' from a few slides back ...

```
shared double balance;
```

Code for p_1

```
disableInterrupts();
balance = balance + amount;
enableInterrupts();
```

Code for p_2

```
disableInterrupts();
balance = balance - amount;
enableInterrupts();
```

Disable Interrupt Example

```
shared int childCnt = 0, parentCnt = 0;
int child, i = 0;
while (i<4) {
   i++;
   child = fork();
}
if (child == 0)
   childCnt++;
else {
   parentCnt++;
   wait();
}
while (childCnt < parentCnt)
  sleep(1000);
printf("%d %d\n", childCnt, parentCnt);
exit;
```

Possible Solution

```
shared int childCnt = 0, parentCnt = 0;
int child, i = 0;
while (i<4) {
    i++;
    child = fork();
}
disableInterrupts();
if (child == 0)
    childCnt++;
else {
    parentCnt++;
    wait();
}
enableInterrupts();
while (childCnt < parentCnt)
   sleep(1000);
printf("%d %d\n", childCnt, parentCnt);
exit;
```

Better Solution

```
shared int childCnt = 0, parentCnt = 0;
int child, i = 0;
while (i<4) {
   i++;
   child = fork();
}
if (child == 0) {
   disableInterrupts();
   childCnt++;
   enableInterrupts();
} else {
   disableInterrupts();
   parentCnt++;
   enableInterrupts();
   wait();
}
while (childCnt < parentCnt)
  sleep(1000);
printf("%d %d\n", childCnt, parentCnt);
exit;
```

critical sections:
- these will not be preempted
- execute as single threads

Problems with Disabling Interrupts

- Interrupts could be disabled arbitrarily long
  - we have no control over how long the app programmer disables interrupts
- Incorrect use can lead to error situation such as deadlock
  - example: you disable interrupts but you need something from another thread
- Really only want to prevent p_1 and p_2 from interfering with one another
  - disabling interrupts blocks all p_i
- Enabling & disabling interrupts should only be done by the OS
  - don’t really want to let users control this part of the system directly

Solution 2: Lock Variables

Idea:

- allocate a shared variable that is a ‘flag’ or ‘lock’
- when a process wants to execute a critical section, it
  - checks that the lock is not already set by another process
  - sets the lock
  - executes the critical section
  - resets the lock

Lock Variable Example

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

Code for p_1

```
/* Acquire the lock */
while(lock) {};
lock = TRUE;

/* Execute critical sect */
balance = balance + amount;

/* Release lock */
lock = FALSE;
```

Code for p_2

```
/* Acquire the lock */
while(lock) {};
lock = TRUE;

/* Execute critical sect */
balance = balance - amount;

/* Release lock */
lock = FALSE;
```

Lock Variable Ex. (Timing Diag)

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

```
while(lock) {};           ts:1
lock = TRUE;
```

```
                          ts:2    while(lock) {};
                                      // busy wait
```

```
balance = balance + amount;  ts:3
lock = FALSE;
```

```
                          ts:4    lock = TRUE;
                                  balance = balance - amount;
                                  lock = FALSE;
```

Everything seems to work!

Can you see any potential problems?

Lock Variable (Unsafe)

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

```
while(lock) {};
```

ts:1

ts:2

```
while(lock) {};
lock = TRUE;
```

ts:3

```
  lock = TRUE;
```

Problem 3: Divisible Test and Set

Idea:

- we need to make the setting of the lock an “atomic operation”
- the test and the set need to be implemented such that they are indivisible (ie: no preemption half way through)

Solution:

- *use enable and disable interrupts*
- it is only for a *short amount of time*
- we can do it *in the kernel*, so it can be a privileged instruction

Solution 3: Atomic Lock Manipulation

```
enter(lock)
{
  disableInterrupts();
  // wait if lock is already set
  while(lock) {
    // don't block interrupts
    enableInterrupts();
    disableInterrupts();
  }
  lock = TRUE;
  enableInterrupts();
}
```

```
exit(lock)
{
  disableInterrupts();
  lock = FALSE;
  enableInterrupts();
}
```

```
yield();
```

- this solution *will* work!
- we do need to include the enable/disable part in the test loop, to ensure that we do not block other processes while we are waiting for the lock to be released
- it would be even nicer to put a yield() instruction after the enableInterrupts in the enter(lock) code segment.

Using Enter() and Exit()

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

Code for p_1

```
enter(lock);

/* Execute critical sect */
balance = balance + amount;

exit(lock);
```

Code for p_2

```
enter(lock);

/* Execute critical sect */
balance = balance - amount;

exit(lock);
```

Review

Concept 1: Concurrent Execution

Concept 2: Shared Variables

Concept 3: Synchronization

Problem 1: Race Conditions

Concept 4: Mutual Exclusion

Problem 2: Critical Section Problem

Solution 1: Disable Interrupts

Solution 2: Lock Variables

Problem 3: Divisible Test and Set

Solution 3: Atomic Test and Set

The End