Why does cooperation require synchronization?

- Example: Two threads: Maria and Tucker share an account with shared variable ‘balance’ in memory.

- Code to `deposit()`:
  ```c
  void deposit( int amount )
  {
    balance = balance + amount;
  }
  ```

- Compiled to assembly:
  ```assembly
  deposit:
  load RegisterA, balance
  add RegisterA, amount
  store RegisterA, balance
  ```

- Both Maria & Tucker deposit money into account:
  - Initialization: balance = 100
  - Maria: deposit( 200 )
  - Tucker: deposit( 10 )

Example Execution

1. Initialization: balance = 100
2. Maria: `deposit( 200 )`
3. Tucker: `deposit( 10 )`

<table>
<thead>
<tr>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>balance = 100</td>
</tr>
<tr>
<td>2</td>
<td>RegisterA = 0</td>
</tr>
<tr>
<td>3</td>
<td>balance = 100</td>
</tr>
<tr>
<td>4</td>
<td>RegisterA = 100</td>
</tr>
<tr>
<td>5</td>
<td>balance = 100</td>
</tr>
<tr>
<td>6</td>
<td>RegisterA = 300</td>
</tr>
<tr>
<td>7</td>
<td>deposit (Maria): load RegisterA, 100 add RegisterA, 200 store RegisterA, balance</td>
</tr>
<tr>
<td>8</td>
<td>deposit (Tucker): load RegisterA, 300 add RegisterA, 10 store RegisterA, balance</td>
</tr>
<tr>
<td>9</td>
<td>balance = 300</td>
</tr>
<tr>
<td>10</td>
<td>RegisterA = 310</td>
</tr>
</tbody>
</table>

Chapter 6: Process [ & Thread ]
Synchronization

- Why is synchronization needed?
- Synchronization Language/Definitions:
  - What are race conditions?
  - What are critical sections?
  - What are atomic operations?
- How are locks implemented?
Concurrency

What happens if M & T deposit the funds concurrently?

- Strategy: Assume that any interleaving is possible
- No assumption about scheduler

Observation: When a thread is interrupted, the content of registers are saved (and restored) by interrupt handlers (dispatcher/context switcher).

- Initialization: balance = 100
- Maria: deposit(200)
- Tucker: deposit(10)

```
deposit (Maria):
    load RegisterA, balance
    add RegisterA, 200
    store RegisterA, balance
```

```
deposit (Tucker):
    load RegisterA, balance
    add RegisterA, 10
    store RegisterA, balance
```

4. Memory:
   balance = 100
   RegisterA = 100

4. Memory:
   balance = 110
   RegisterA = 110

Race Condition

- Results depends on the order of execution
  - Result in non-deterministic bugs, these are hard to find!
    - Deterministic: Input alone determines results, i.e., the same inputs always produce the same results:
      - Example: Sqrt(4) = 2

- Intermittent –
  - A time dependent `bug`
    - A small change may hide the real bug (e.g., print statements can hide the real bug because they slow down processing time and consequently impacts the timing of the threads).

How to avoid race conditions

- Idea: Prohibit one or more threads from reading and writing shared data at the same time! ⇒ Provide Mutual Exclusion (what?)

- Critical Section: Part of program (or `slice`) where shared memory is accessed

```
void credit( int amount )
{
    int x = 5;
    printf( "Adding money\n" );
    balance = balance + amount;
}
```

```
void debit( int amount )
{
    int i;
    balance = balance - amount;
    for( i = 0; i < 5; i++ );
}
```
THE Critical Section Problem?

- **Problem**: Avoiding race conditions (i.e., provide mutual exclusion) is *not sufficient* for having threads cooperate *correctly (no progress)* and efficiently:
  - What about if *no one* gets into the critical section even if several threads want to get in? (No progress at ALL!)
  - What about if someone waits outside the critical section and never gets a turn? (starvation, NOT FAIR!)

What We Want: Mutual Exclusion (!)

- **Required Properties**:
  - **Mutual Exclusion**:
    - Only *one* thread in the critical section at a time
  - **Progress (e.g., someone gets the CS)**:
    - Not block others out: if there are requests to enter the CS must allow one to proceed
    - Must not depend on threads outside critical section
      - If no one is in CS then someone must be let in...
        - We take no reservations!
  - **Bounded waiting (starvation-free)**:
    - Must eventually allow each waiting thread to enter

Solve: THE Critical Section Problem: “Proper” Synchronization

- **Required “Proper”ties**:
  - **Mutual Exclusion**
  - **Progress (someone gets the CS)**
  - **Bounded waiting (starvation-free, eventually you will run)**

- **Desirable Properties**:
  - **Efficient**:
    - Don’t consume substantial resources while waiting.
      - Example: Do not busy wait (i.e., spin wait)
  - **Fair**:
    - Don’t make some processes wait longer than others
  - **Simple**: Should be easy to reason about and use
Critical Section Problem: Need Atomic Operations

- **Basics:** Need atomic operations:
  - No other instructions can be interleaved (low level)
  - Completed in its entirety without interruption (no craziness)
- **Examples** of atomic operations:
  - Loads and stores of words
    - load register1, B
    - store register2, A
  - Idea: Code between interrupts on uniprocessors
    - Disable timer interrupts, don’t do any I/O
  - Special hardware instructions (later)
    - “load, store” in one instruction
    - Test&Set
    - Compare&Swap

Disabling Interrupts

- **Kernel provides two system calls:**
  - Acquire() and
  - Release()
- **No preemption when interrupts are off!**
  - No clock interrupts can occur
- **Disadvantage:**
  - unwise to give processes power to turn off interrupts
  - Never turn interrupts on again!
  - Does not work on multiprocessors
- **When to use?**
  - But it may be good for kernel itself to disable interrupts for a few instructions while it is updating variables or lists

void Acquire()
{
  disable interrupts
}

void Release()
{
  enable interrupts
}

Software Solutions

- **Assumptions:**
  - We have an atomic load operation (read)
  - We have an atomic store operation (assignment)
- **Notation** [lock=true, lock=false]
  - True: means un-available (lock is set, someone has the lock)
  - False: means available (e.g., lock is not set, as the CS is available, no one is in the CS)

Attempt 1: Shared Lock Variable

- **Single shared lock variable**
  ```
  boolean lock = false; // lock available shared variable
  void deposit(int amount)
  {
    while( lock == true ) {} /* while lock is set : wait */;
    lock = true; /* gets the lock */
    balance += amount; // critical section
    lock = false; /* release the lock */
  }
  ```

- **Uses busy waiting**
- **Does this work?**
  - Are any of the principles violated (i.e., does it ensure mutual, progress and bounded waiting)?
**Attempt 1: Shared Variable**

- M reads lock sees that it is false
- T reads lock sets it as false
- M sets the lock
- T sets the lock

```
boolean lock = false; // shared variable
void deposit(int amount)
{
    while( lock == true ) {} /* wait */
    lock = true;
    balance += amount; // critical section
    lock = false;
}
```

**Problem & Lesson**

- **Problems:**
  - No mutual exclusion: Both processes entered the CS.
- **Lesson learned:** Failed because two threads read the lock variable simultaneously and both thought it was its ‘turn’ to get into the critical section.

**Shared Lock Variable**

<table>
<thead>
<tr>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Idea: Take Turns:
Add a variable that determine if it is its turn or not!

**Attempt 2: Alternate (we want to be fair)**

- **Idea:** Take turns (alternate) via a turn variable that determines which thread’s turn it is to be in the CS
  - (set to thread ID's: 0 or 1). We are assuming only 2 threads!

```
int turn = 0; // shared variable
void deposit( int amount )
{
    while( turn == 1-tid ) {} /* wait */
    balance += amount; // critical section
    turn = 1-tid;
}
```

**Does this work?**

- Mutual exclusion?
- Progress (someone gets the CS if empty)
- Bounded waiting... it will become next sometime?

**Attempt 2: Alternate – Does it work?**

- **Initialize:** Maria is '0' & Tucker is '1'
- M reads turn sees that it is her turn
- M done and change turn to other
- M want to deposit more money...
- T never requests CS no money!

Maria is blocking!

Tucker is not interested in the CS (not deadlocked?)

No progress!
Attempt 2: Strict Alternation

- **Problems:**
  - **No progress:**
    - If no one is in a critical section and a thread wants in -- it should be allowed to enter
  - **Also not efficient:**
    - Pace of execution: Dictated by the slower of the two threads. If Tucker uses its CS only one per hour while Maria would like to use it at a rate of 1000 times per hour, then Maria has to adapt to Tucker’s slow speed.

<table>
<thead>
<tr>
<th></th>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting</th>
<th>No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Strict Alternation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Attempt 3: Check “other thread’s” state/interest then Lock

- **Idea:** Each thread has its own lock; lock indexed by tid (0, 1). Check other’s needs

```java
boolean lock[2] = {false, false} // shared
void deposit( int amount )
{
    while( lock[1-tid] == true ) {} /* wait for other */;
    lock[tid] = true;
    balance += amount; // critical section
    lock[tid] = false;
}
```

- **Does this work?** Mutual exclusion? Progress (someone gets the CS if empty, no deadlock)? Bounded Waiting (no starvation)?

Lessons: Attempt 2: Strict Alternation

- **Problem:** Need to fix the problem of progress!
- **Lesson:** Why did strict alternation fail?
  - Pragmatically: Problem with the turn variable is that we need state information about BOTH processes.
    - We need to know if the other thread is interested in the CS, and [of-course]:
    - We should not wait for a thread that is not interested!

- **Idea:**
  - We need to know the needs of others!
  - Check to see if other needs it.
    - Don’t get the lock until the ‘other’ is done with it.

Attempt 3: Check then Lock

- M checks if Tucker is interested and he isn’t
- T checks if Maria is interested and she isn’t
- Switch back to Maria she now sets his lock
- Switch Back to Tucker he sets his lock
**Attempt 3: Check then Lock**

- **Problems:**
  - No Mutual Exclusion
- **Lesson:** Process locks the critical section AFTER the process has checked it is available but before it enters the section.
- **Idea:** Lock the section first! then lock...

<table>
<thead>
<tr>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting</th>
<th>No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict Alteration</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Check then Lock</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Idea:** Process locks the critical section AFTER the process has checked it is available but before it enters the section.

<table>
<thead>
<tr>
<th>Place limited to slowest process</th>
</tr>
</thead>
</table>

**Attempt 4: Lock then Check**

- **Idea:** Each thread has its own lock; lock indexed by tid (0, 1). Check other’s needs

```java
boolean lock[2] = {false, false} // shared
void deposit(int amount)
{
    lock[tid] = true; /* express interest */
    while (lock[1-tid] == true) {} /* wait */;
    balance += amount; // critical section
    lock[tid] = false;
}
```

- **Mutual Exclusion:** Yes
- **Deadlock:** Each thread waits for the other. Each one thinks that the other is in the critical section
**Attempt 4: Lock then Check**

- **Problems:**
  - No one gets the critical section!
  - Each thread ‘insisted’ on its right to get the CS and did not back off from this position.

- **Lesson:** Again a ‘state’ problem, a thread misunderstood the state of the other thread

- **Idea:** Allow a thread to back off to give the other a chance to enter its critical section.

<table>
<thead>
<tr>
<th></th>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting</th>
<th>No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict Alteration</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Check then Lock</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock then Check</td>
<td>Yes</td>
<td>No (deadlock)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Attempt 5: Defer, back-off lock**

- **Idea:** Add an delay

```java
boolean lock[2] = {false, false} // shared
void deposit( int amount )
{
    lock[tid] = true;
    while( lock[1-tid] == true ) /* spin for other to finish */
    {
        lock[tid] = false;
delay;
        lock[tid] = true;
    }
    balance += amount; // critical section
    lock[tid] = false;
}
```

**Attempt 5: Deferral**

- **Problems:**
  - Mutual Exclusion: Yes
  - Live Lock: sequence can be broken if you are lucky!
    - Not really a deadlock (guaranteed not to be able to proceed)
    - Not starvation - threads starves when a process repeatedly loose to the other threads, here both loose

<table>
<thead>
<tr>
<th></th>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting</th>
<th>No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict Alteration</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Check then Lock</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock then Check</td>
<td>Yes</td>
<td>No (deadlock)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferral</td>
<td>Yes</td>
<td>No (not deadlock)</td>
<td>Not really</td>
<td></td>
</tr>
</tbody>
</table>

Pace limited to slowest process
Lessons

- We need to be able to observe the state of both processes
  » Simple lock is not enough
- We must impose an order to avoid this ‘mutual courtesy’; i.e., after-you-after you phenomena
- Idea:
  » If both threads attempt to enter CS at the same time let only one thread it.
  » Use a turn variable to avoid mutual courtesy
    – Indicates who has the right to insist on entering his critical section.

Dekker’s Algorithm

Mutual Exclusion: Two threads cannot be in the critical region simultaneously – prove by contraction.

- Suppose they are then locks are set according the time line for each point of view (P0, P1).
  » P0:
    - 1. lock[0] = true (sets the lock, then)
    - 2. lock[1] = false (see that lock 1 is false)
  » P1:
    - 3. lock[1] = true
    - 4. lock[0] = false
- Suppose P2 enters CS no later than P1
  » t2 < t4 (so P0 checks lock[1] is false just before entering its CS).
  » t2 < t3
    - after 3. lock[1] = true it remains true so t2 < t3
  » So: t1 < t2 < t3 < t4
- But lock[0] cannot become false until P0 exits and we assumed that both P0 and P1 were in the CS at the same time. Thus it is impossible to have checked flag as false at t4.

Attempt 6: Careful Turns

```
boolean lock[2] = {false, false} // shared
int turn = 0; // shared variable - arbitrarily set
void deposit( int amount )
{
    lock[tid] = true;        // I am interested in the lock take my lock.
    while( lock[1-tid] == true ) // *IS* the OTHER interested? If not get in!
    {
        if( turn == 1-tid )    // if it is it OTHER’s turn then *I* SPIN/DEFER
            lock[tid] = false; // it is - so I will LET him get the lock.
        while( turn == 1 - tid ) {}; // wait to my turn
        lock[tid] = true;        // my turn - still wants the lock
    }
    balance += amount; // critical section
    turn = 1 - tid;    // Set it to the other’s turn so he stops spinning */
    lock[tid] = false;
}
```
**Attempt 7: Peterson’s Simpler Lock Algorithm**

- **Idea**: combines turn and separate locks (recall turn taking avoids the deadlock)

```java
boolean lock[2] = {false, false}; // shared
int turn = 0; // shared variable
void deposit(int amount) {
    lock[tid] = true;
    turn = 1-tid; // set turn to other process
    while(lock[1-tid] == true && turn == 1-tid) {};
    balance += amount; // critical section
    lock[tid] = false;
}
```

- When 2 processes enter simultaneously, setting turn to the other releases the other process from the while loop (one write will be last).

- **Mutual Exclusion**: Why does it work?
  - The Key Observation: Turn cannot be both 0 and 1 at the same time

**Peterson’s Algorithm Intuition (1981)**

- **Mutual exclusion**: Enter critical section if and only if
  - Other thread does not want to enter
  - Other thread wants to enter, but your turn

- **Progress**: Both threads cannot wait forever at while() loop
  - Completes if other process does not want to enter
  - Other process (matching turn) will eventually finish

- **Bounded waiting**: Each process waits at most one critical section

**Summary: Software Solutions**

<table>
<thead>
<tr>
<th></th>
<th>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting</th>
<th>No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Strict Alteration</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Check then Lock</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock then Check</td>
<td>Yes</td>
<td>No (deadlock)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deferral</td>
<td>Yes</td>
<td>No (not deadlock)</td>
<td>Not really</td>
<td></td>
</tr>
<tr>
<td>Dekker</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Peterson</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**2 Processes**

- So far, only 2 processes and it was tricky!
- How about more than 2 processes?
  - Enter Leslie’s Lamport’s Bakery Algorithm

**Pace limited to slowest process**
Lamport’s Bakery Algorithm
(1974)

Idea: Bakery -- each thread picks next highest ticket (may have ties --ties broken by a thread’s priority number)

A thread enters the critical section when it has the lowest ticket.

Data Structures (size N):
- choosing[i] : true iff P, in the entry protocol
- number[i] : value of ‘ticket’, one more than max
- Threads may share the same number

Ticket is a pair: (number[tid], i)

Lexicographical order:
- (a, b) < (c, d) :
  if( a < c) or if( a == c AND b < d )
- (number[j], j) < (number[tid], tid))

Bakery Algorithm

- Pick next highest ticket (may have ties)
- Enter CS when my ticket is the lowest (combination of number and my tid)

choosing[tid] = true; // Enter bakery shop and get a number (initialized to false)
number[tid] = max( number[0], … , number[n-1] ) + 1; /*starts at 0 */
choosing[tid] = false;
for( j = 0; j < n; j++ ) /* checks all threads */
  { while( choosing[j] ){}; // wait until j receives its number
// iff j has a lower number AND is interested then WAIT
  while( number[j]!= 0 && ( (number[j],j) < (number[tid],tid)) );
  balance += amount;
  number[tid] = 0; /* unlocks

Baker’s Algorithm Intuition

- Mutual exclusion:
  - Only enters CS if thread has smallest number
- Progress:
  - Entry is guaranteed, so deadlock is not possible
- Bounded waiting
  - Threads that re-enter CS will have a higher number than threads that are already waiting, so fairness is ensured (no starvation)

choosing[tid] = true;
number[tid] = max( number[0], … , number[n-1] ) + 1;
choosing[tid] = false;
for( j = 0; j < n; j++)
  while( choosing[j] ){}; // wait until j is done choosing
  // wait until number[j] = 0 (not interested) or me smallest number
  while( number[j]!= 0 && ( (number[j],j) < (number[tid],tid)) );
balance += amount;
number[tid] = 0;