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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
</tr>
</tbody>
</table>

  - Memory: `balance = 300`, `RegisterA = 300`
  - Time: `Maria: deposit(200)`

Another Example

- Who wins?

- **Example**: Maria & Tucker share account: `balance` with registers `RegisterA`

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
</tr>
</tbody>
</table>

  - Memory: `balance = 310`, `RegisterA = 310`
  - Time: `Maria: deposit(200)`

- **Another Example**: Maria & Tucker share account: `balance` with registers `RegisterA`

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
<td>balance = 100</td>
<td>RegisterA = 100</td>
</tr>
</tbody>
</table>

  - Memory: `balance = 310`, `RegisterA = 310`
  - Time: `Maria: deposit(200)`

- Who wins?
Aside: What program data is shared?

- **Local variables** are not shared (private)
  - Each thread has its own stack
  - Local variables are allocated on private stack
  - Weird Bugs: Never pass, share or store a pointer to a local variable on another threads stack
- **Global variables and static objects** are shared
  - Stored in the static data segment, accessible by any threads
- **Dynamic objects** and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete

Critical Section Problem: Properties

- **Mutual Exclusion**: Only one thread in 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 (e.g., no deadlocks).
  - Must not depend on threads outside critical section
    - If no one is in CS then must let someone in.
- **Bounded waiting (starvation-free)**:
  - Must eventually allow each waiting thread to enter

Race Condition

- Results depends on **order of execution**
  - Result in non-deterministic bugs, hard to fine!
    - Deterministic: Input alone determines results, i.e., the same inputs always produce the same results
- **Intermittent** - time depended 'bug' a small change may hide the real bug (e.g., print statements can hide the real bug because the slow down processing and impact 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**
- **Critical Section**: Part of program where shared memory is accessed

```c
void deposit(int amount) {
    balance = balance + amount;
}
void debit(int amount) {
    for (i = 0; i < 5; i++) {
    int i;
    balance = balance - amount;
    }
```

Mutual Exclusion

```c
Process Maria
Maria enters her critical section

Process Tucker
Tucker enters his critical section

void deposit(int amount) {
    balance = balance + amount;
}
void debit(int amount) {
    int i;
    for (i = 0; i < 5; i++) {
    }
}```
Critical Section Problem: Properties

Required Properties:
- Mutual Exclusion
- Progress (someone gets the CS)
- Bounded waiting (starvation-free)

Desirable Properties:
- Efficient:
  - Don’t consume substantial resources while waiting. Do not busy wait (i.e., spin wait)
- Fail:
  - Don’t make some processes wait longer than others
- Simple: Should be easy to reason about and use

Disabling Interrupts

- Kernel provides two system calls:
  - Acquire() and Release()
- No preemption when interrupts are off!
  - No clock interrupts can occur
- Disadvantages:
  - Unwise to give processes power to turn of interrupts
    - Exame Problem: Never turn interrupts on again!
  - Does not work on multiprocessors: Why?
    - Only disables one CPU, the other CPU can still access shared memory
- When to use?:
  - Good for kernel itself to disable interrupts for a few instructions while it is updating variables or lists

Software Solutions

- Assumptions:
  - atomic loads
  - atomic stores
- Notation:
  - True: mean un-available
  - False: means available (e.g., no one is in CS)

Critical Section Problem: Need Atomic Operations

- Basics: Need atomic operations:
  - No other instructions can be interleaved
  - Completed in its entirety without interruption
- Examples of atomic operations:
  - Loads and stores of words
    - load register1, B
    - store register2, A
  - 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

Attempt 1: Shared Lock Variable

- Single shared lock variable

  ```
  void Acquire()
  {
    disable interrupts
  }

  void Release()
  {
    enable interrupts
  }
  ```

- Uses busy waiting
- Does this work?
  - Which principles is violated?

Attempt 1: Shared Variable

- M reads lock sees it as false
- T reads lock sets it as false
- M sets the lock
- T sets the lock
- Two threads in critical Section
Attempt 1: Lock Variable

Problem & Lesson

- **Problems:**
  - No mutual exclusion: Both processes entered the CS.

- **Lessons:** Failed because two threads read the lock variable simultaneously and both thought it was their ‘turn’ to get into the critical section.

<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>Shared Lock Variable</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Attempt 2: Strict Alternation

- **Idea:** Take turns. turn determines which thread can enter (set to thread ID's: 0 or 1).

```java
void deposit(int amount) {
    while (turn != 1-tid) {} /* wait */

    turn = 1-tid;
    balance += amount; // critical section
}
```

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

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

- **Efficiency:**
  - 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>Mutual Exclusion</th>
<th>Progress someone gets the CS</th>
<th>Bounded Waiting No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock Variable</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Attempt 2: Strict Alternation

- **Problem:** Progress

- **Lesson:** Why did strict alternation fail?
  - Pragmatically: Problem with the turn variable is that we need state information about BOTH processes.
    - We should not wait for a thread that does not need if they don't need to get to the critical section

- **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 State 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 */
    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)?
Attempt 3: Check then Lock

- Problems:
  - No Mutual Exclusion
- Lesson: Process lock's 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 No Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Lock Variable</th>
<th>Shared Lock</th>
<th>Strict Alteration</th>
<th>Check then Lock</th>
<th>Lock then Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pace limited to allowed 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

```
boolean lock[2] = {false, false} // shared
void deposit( int amount ) {  
  lock[tid] = true;  
  ... in the critical section
  balance += amount;  
  lock[tid] = false;
}
```

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

Attempt 4: Lock then Check

- Mutual Exclusion: Yes
- Deadlocks: Each thread waits for the other. Each one thinks that the other is in the critical section
Attempt 5: Careful Turns

• Take 'careful' turns

boolean lock[2] = {false, false} // shared
void deposit( int amount )
{  lock[tid] = true;
  while( lock[1-tid] == true )
  {  lock[tid] = false;
      delay;
      lock[tid] = true;
  }
  balance += amount; // critical section
  lock[tid] = false;
}
Attempt 6: Dekker’s Algorithm

- Take ‘careful’ turns

```java
boolean lock[2] = {false, false} // shared
int turn = 0; // shared variable
void deposit( int amount )
{
    lock[0] = true;
    while( lock[1-tid] == true ) // check other
        if( turn == 1-tid ) // other turn to insist
            lock[1-tid] = false; // then I defer
    turn = 1 - tid; // I defer
    while( turn = 1 - tid ) {
        lock[1-tid] = true;
        balance += amount; // critical section
    }
    lock[0] = false;
}
```

- 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

Attempt 7: Peterson’s Algorithm

- Idea: also combines turn and separate locks

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

- When 2 processes enters simultaneously, setting turn to the other releases the ‘other’ process from the while loop (one write will be last).
- Mutual Exclusion: Key Observation: turn cannot be both 0 and 1 at the same time

Peterson’s Algorithm Intuition

- 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</th>
<th>Bounded Waiting</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Lock</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td></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</td>
<td>(deadlock)</td>
<td></td>
</tr>
<tr>
<td>Deferral</td>
<td>Yes</td>
<td>No</td>
<td>Not really</td>
<td></td>
</tr>
<tr>
<td>Peterson</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Lamport’s Bakery Algorithm

- Idea: Bakery – each thread picks next highest ticket (may have ties)
- Enter critical section when have 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

```java
void deposit( int amount )
{
    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
    }
    while( number[j] != 0 )
    {
        if( number[j] < (number[tid],tid))
        {
            balance += amount;
            number[tid] = 0;
        }
    }
    while( number[j] != 0 )
    {
        if( number[j] < (number[tid],tid))
        {
            balance += amount;
            number[tid] = 0;
        }
    }
    balance += amount;
    number[tid] = 0;
}
```
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)

```c
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;
```