


CSCI [4|6] 730
Operating Systems

Synchronization Part 1 : The Basics

Chapter 6: Process Synchronization

- Why is synchronization needed?
- Definitions:
  - » What are race conditions?
  - » What are critical sections?
  - » What are atomic operations?
- How are locks implemented?

Why does cooperation require synchronization? (Review)

- **Example: Two threads: `Maria` and `Tucker` share an account with shared variable '`balance`' in memory.**
- **Code to `deposit()`:**

```
void deposit( int amount )
{
balance = balance + amount;
}
```

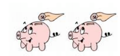
- **Compiled to assembly:**

```
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 )`

Which variables are shared? Which private?

Example Execution



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

```
deposit:
  load  RegisterA, balance
  add   RegisterA, amount
  store RegisterA, balance
```

```
Memory:
  balance = 310
  RegisterA = 310
```

Time

```
deposit (Maria):
  load  RegisterA, 100
  add   RegisterA, 200
  store RegisterA, balance
```

```
deposit (Tucker):
  load  RegisterA, 300
  add   RegisterA, 10
  store RegisterA, balance
```


Concurrency

```
4. Memory:
  balance = 300
  RegisterA = 300
```

320?

```
4. Memory:
  balance = 110
  RegisterA = 110
```

- What happens if M & T deposit "concurrently"?
  - » Assume any interleaving is possible
  - » No assumption about scheduler
  - » Observation: When a thread is interrupted content of registers are saved (and restored) by interrupt handlers.
    - – Initialization: balance = 100
    - – Maria: deposit(200)
    - – Tucker: deposit(10)

```
deposit:
  load  RegisterA, balance
  add   RegisterA, amount
  store RegisterA, balance
```

```
deposit (Maria):
  load  RegisterA, balance

  add   RegisterA, 200

  store RegisterA, balance
```

Time

```
deposit (Tucker):

  load  RegisterA, balance

  add   RegisterA, 10

  store RegisterA, balance
```


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`

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 –
  - » A time dependent `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

Critical Section

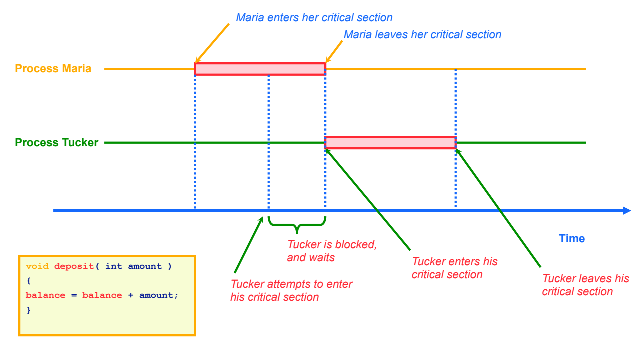
```
void credit( int amount )
{
int x = 5;
printf( "Adding money" );
balance = balance + amount;
}
```

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


Critical Sections

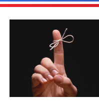
- Problem: Avoiding race conditions (i.e., provide mutual exclusion) is not sufficient for having threads cooperate correctly and *efficiently*
  - » What about if no one gets into the critical section even if several threads wants to get in?
  - » What about if someone waits outside the critical section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

What We Want: *Mutual Exclusion*



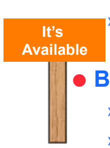
Critical Section Problem: Properties

Memorize



Required 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



Critical Section Problem: Properties

Required "Proper"ties :

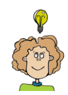
- Mutual Exclusion
- Progress (someone gets the CS)
- Bounded waiting (starvation-free)

Desirable Properties:

- Efficient:
  - » Don't consume substantial resources while waiting. Do not busy wait (i.e., spin wait)
- 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
  - » 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



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 of 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 Aquire()
{
disable interrupts
}
```

```
void Release()
{
enable interrupts
}
```

Who do you trust?
Do you trust your kernel?
Do you trust your friend's kernel?
Do you trust your kernel's friends?

Software Solutions

- **Assumptions:**
  - » We have an atomic load operation.
  - » We have an atomic store operation.
- **Notation:**
  - » True: means un-available
  - » False: means available (e.g., no one is in CS)

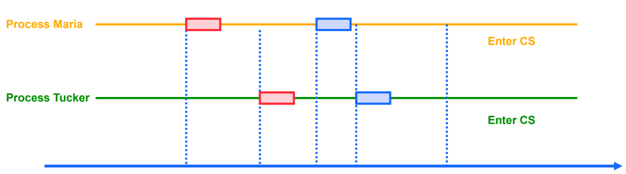
Attempt 1: Shared Lock Variable

- Single shared `lock` variable

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

- 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




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

- 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.
- **Lesson learned**: Failed because two threads read the lock variable simultaneously and both thought it was its 'turn' to get into the critical section

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	X		

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

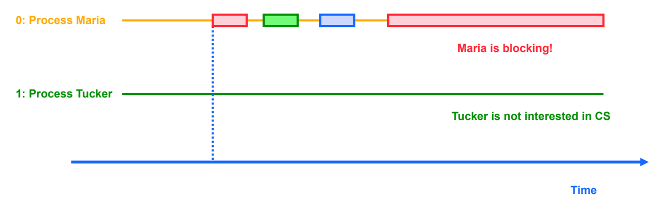
Attempt 2: Alternate (we want to be fair)

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

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

- Does this work?
  - » Mutual exclusion?
  - » Progress (someone gets the CS if empty, no deadlock)?
  - » Bounded waiting… it will become next sometime?

Attempt 2: Alternate – Does it work?




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

- Initialize: Maria is '0' & Tucker is '1'
- M reads turn sees her turn
- M done and change turn to other
- T never requests CS no money!

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.

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	No		
Strict Alteration	Yes	No	No

Pace limited to slowest process

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 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.

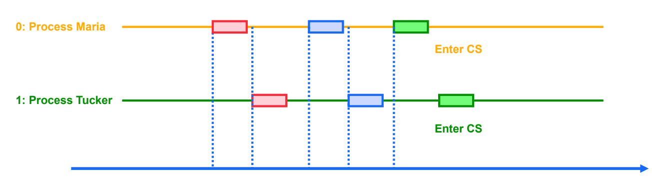
Attempt 3: Check State then Lock

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

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

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

Attempt 3: Check then Lock




```
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;
  }
```

- 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…

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	No		
Strict Alteration	Yes	No	No
Check then Lock	No		

Pace limited to slowest process

Attempt 4: Lock then Check

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

Entry CS: / CS: / Exit CS:

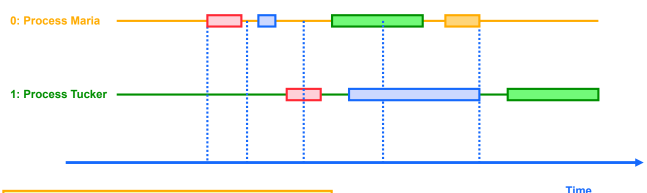
```
boolean lock[2] = {false, false} // shared
void deposit( int amount )
  {
  lock[tid] = true;
  while( lock[1-tid] == true ) {} /* wait */ ;

  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 4: Lock then Check

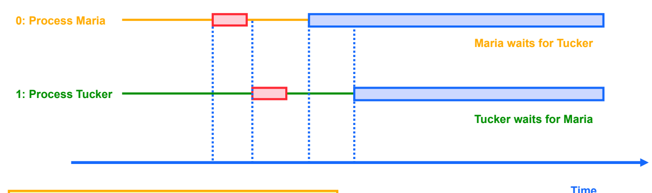



```
boolean lock[2] = {false, false} // shared
void deposit( int amount )
  {
  lock[tid] = true;
  while( lock[1-tid] == true ) {} /* wait */;
  balance += amount; // critical section
  lock[tid] = false;
  }
```

Mutual Exclusion?

- Maria's View: Once Maria sets her lock:
  - » Tucker cannot enter until Maria is done
  - » Tucker already in CS, then Maria blocks until Tucker leaves the CS
- Tucker's View: Same thing
- So yes Mutual Exclusion

Attempt 4: Lock then Check




```
boolean lock[2] = {false, false} // shared
void deposit( int amount )
  {
  lock[tid] = true;
  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.

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	No		
Strict Alteration	Yes	No	No
Check then Lock	No		
Lock then Check	Yes	No (deadlock)	

Pace limited to slowest process

Attempt 5: Defer, back-off lock

- **Idea**: Add an delay

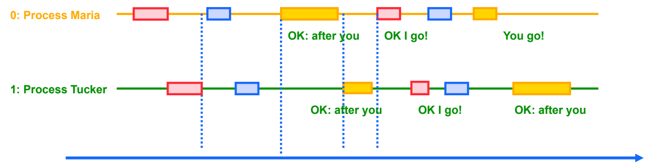
Entry CS: / CS: / Exit CS:

```
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 5: Deferral




```
boolean lock[2] = {false, false}
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;
  }
```

- 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

Attempt 5: Deferral

- Problems:

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	No		
Strict Alteration	Yes	No	No
Check then Lock	No		
Lock then Check	Yes	No (deadlock)	
Deferral	Yes	No (not deadlock)	Not really

Pace limited to slowest process

Lessons

- We need to be able to observe the state of both processes
  - » Lock not enough
- We most impose an order to avoid this 'mutual courtesy'; i.e., after you-after you
- Idea:
  - » use turn variable to avoid mutual courtesy
    - – Indicates who has the right to insist on entering his critical section.

Attempt 6: Careful Turns

```
boolean lock[2] = {false, false} // shared
int turn = 0; // shared variable
void deposit( int amount )
  {
  lock[tid] = true;              // I am interested in the lock
  while( lock[1-tid] == true )   // *IS* the other  interested? If not get in!
    {                            //* WE know he is interested! (we both are)
     if( turn == 1-tid )         // is it his turn to insist to get a turn?
                                  // NOTE if it is MY turn keep the lock
        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;
  lock[tid] = false;
  }
```


Quiz

- Does it work?
- Why does it work

Attempt 7: Peterson's Simpler Lock Algorithm

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

```
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 enters 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? 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

```
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;
  }
```


Summary: Software Solutions

	Mutual Exclusion	Progress someone gets the CS	Bounded Waiting No Starvation
Shared Lock Variable	No		
Strict Alteration	Yes	No	No
Check then Lock	No		
Lock then Check	Yes	No (deadlock)	
Deferral	Yes	No (not deadlock)	Not really
Dekker	Yes	Yes	Yes
Peterson	Yes	Yes	Yes

Pace limited to slowest process

Simpler

Lamport's Bakery Algorithm

- **Idea**: Bakery -- each thread picks next highest ticket (may have ties)
- A thread enters the critical section when it has the lowest ticket.
- Data Structures (size `N`):
  - `choosing[i]` : true iff P_i 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

```
choosing[tid] = true;  // Enter bakery shop and get a number
number[tid] = max( number[0], … , number[n-1] ) + 1;
choosing[tid] = false;
for( j = 0; j < n; j++ )
  {
  while( choosing[j] ){};  // wait until j receives its number

  // wait until number[j] = 0 (not interested) or
  // my number is the lowest
  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;
```