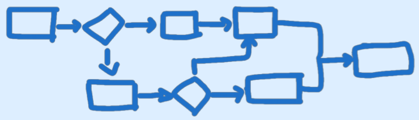
Imperial College London



Operating Systems

Synchronisation I

Course 211
Spring Term 2018-2019

http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

(Slides courtesy of Cristian Cadar)

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

Process Synchronisation

How do processes **synchronise** their operation to perform a task?

Key concepts:

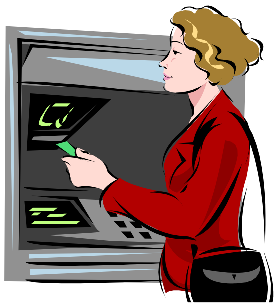
- Critical sections
- Mutual exclusion
- Atomic operations
- Race conditions
- Deadlock
- Starvation
- Synchronisation mechanisms
  - Locks, semaphores, monitors, etc.

Concepts relevant to both **processes** and **threads**

Shared Data Example

Account #1234: £10,000

Extract £1000
from account 1234



Extract £1000
from account 1234

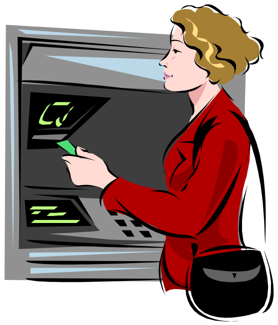
Shared Data Example

```
void  Extract(int acc_no, int sum)
{
  int B = Acc[acc_no];
  Acc[acc_no] = B – sum;
}
```

Acc[1234] 10,000

```
B = 10,000
Acc[1234] = 9000
```

```
B = 9,000
Acc[1234] = 8000
```



```
Extract(1234, 1000)
```

```
Extract(1234, 1000)
```

Shared Data Example

```
void  Extract(int acc_no, int sum)
{
   int B = Acc[acc_no];
   Acc[acc_no] = B – sum;
}
```

Critical section!
Need **mutual exclusion**

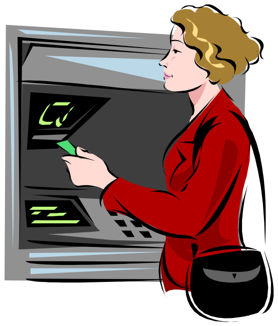
Acc[1234] `10,000`

`B = 10,000`

`B = 10,000`

`Acc[1234] = 9000`

`Acc[1234] = 9000`



`Extract(1234, 1000)`

`Extract(1234, 1000)`

Critical Sections and Mutual Exclusion

Critical section/region: section of code in which processes access a shared resource

Mutual exclusion ensures that if a process is executing its critical section, no other process can be executing it

- Processes must request **permission** to enter critical sections

A **synchronisation mechanism** is required at the entry and exit of the critical section

Requirements for Mutual Exclusion

No two processes may be simultaneously inside a critical section

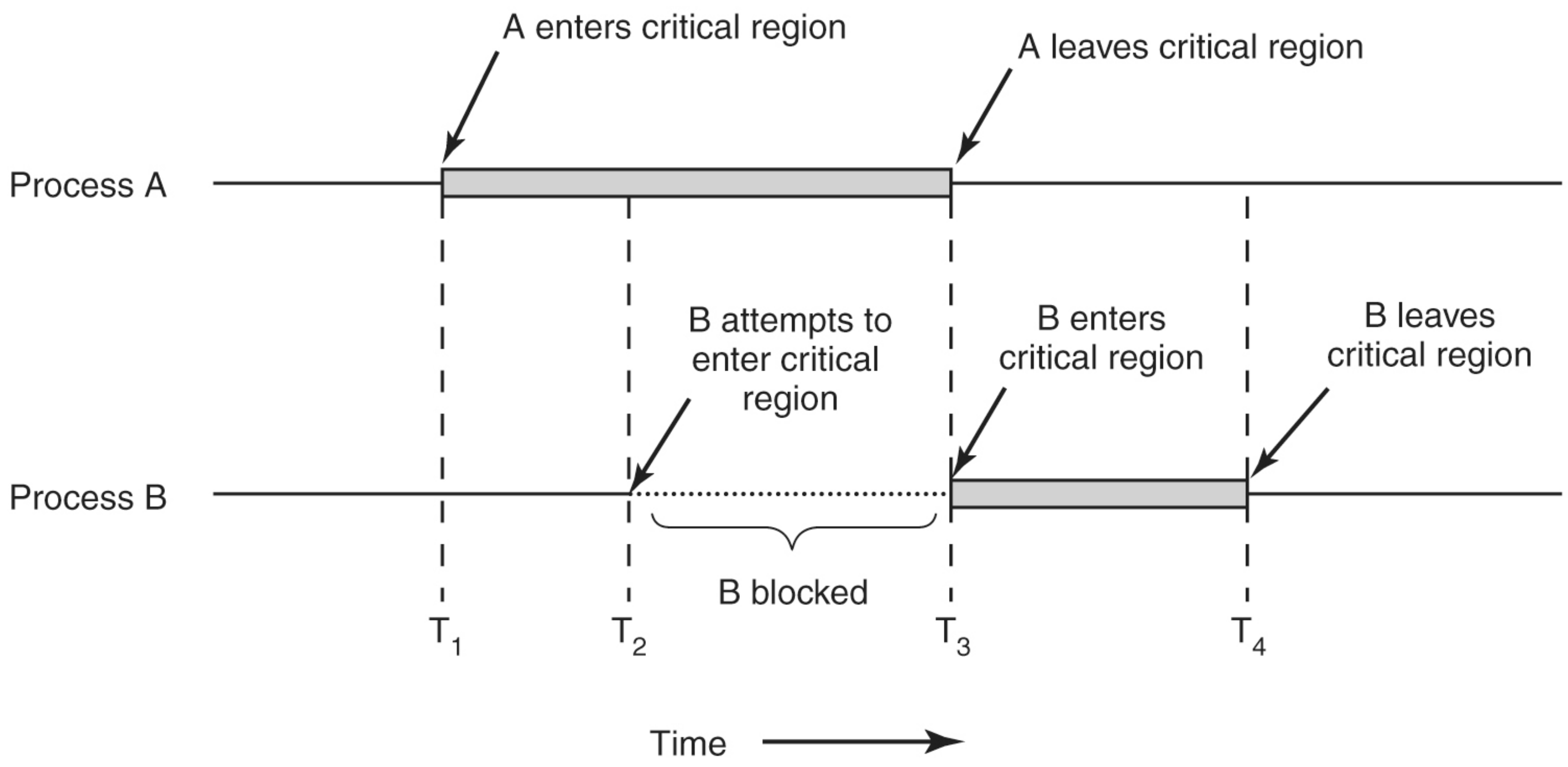
No process running outside the critical section may prevent other processes from entering the critical section

- When no process is inside a critical section, any process requesting permission to enter must be allowed to do so immediately

No process requiring access to its critical section can be delayed forever

No assumptions are made about relative the speed of processes

Critical Sections and Mutual Exclusion



Disabling Interrupts

```
void Extract(int acc_no, int sum)
{
  CLI();
  int B = Acc[acc_no];
  Acc[acc_no] = B - sum;
  STI();
}
```

Works only on single-processor systems

Misbehaving/buggy processes may never release CPU

- Mechanism usually only used by kernel code

Software Solution – Strict Alternation

turn $\boxed{0}$

T_0

```
while (true) {
   while (turn != 0)
     /* loop */ ;
   critical_section()
   turn = 1;

noncritical_section0();
}
```

T_1

```
while (true) {
   while (turn != 1)
     /* loop */ ;
   critical_section()
   turn = 0;

noncritical_section1();
}
```

What happens if T_0 takes a long time in its non-critical section?

– Remember: No process running outside its critical section may prevent other processes from entering the critical section

Can we have T_1 execute its loop twice in a row (w/o T_0 executing in-between)?

Busy Waiting

- Strict alternation solution requires continuously testing the value of a variable
- Called **busy waiting**
  - – Wastes CPU time
  - – Should only be used when the wait is expected to be short

Peterson’s Solution

```
int turn = 0;
int interested[2] = {0, 0};

// thread is 0 or 1
void enter_critical(int thread)
{
   int other = 1 - thread;
   interested[thread] = 1;
   turn = other;
   while (turn == other &&
          interested[other])
      /* loop */ ;
}

void leave_critical(int thread)
{
   interested[thread] = 0;
}
```

T0

```
enter_critical(0);
critical_section();
leave_critical(0);
```

T1

```
enter_critical(1);
critical_section();
leave_critical(1);
```

Peterson’s Solution – Mutual Exclusion Proof

```
int turn = 0;
int interested[2] = {0, 0};

// thread is 0 or 1
void enter_critical(int thread)
{
   int other = 1 - thread;
   interested[thread] = 1;
   turn = other;
   while (turn == other &&
          interested[other])
      /* loop */ ;
}

leave_critical(int thread)
{
   interested[thread] = 0;
}
```

First note that when $\mathbf{T_K}$ tries to enter or is in CS, **`interested[k]=1`**.

Assume both T_0 and T_1 try to enter CS. Then: **`interested[0]=interested[1]=1`** and **`turn`** allows only one thread to enter.

Assume T_0 is in CS. T_1 has to wait for T_0 to set **`interested[0]`** to 0, which only happens when T_0 leaves CS.

Assume T_1 is in CS. T_0 has to wait for T_1 to set **`interested[1]`** to 0, which only happens when T_1 leaves CS.

Atomic Operations

```
void Extract(int acc_no,
             int sum)
{
  int B = Acc[acc_no];
  Acc[acc_no] = B - sum;
}
```

```
void Extract(int acc_no,
             int sum)
{
   Acc[acc_no] -= sum;
}
```

Does this work?

Atomic operation: a sequence of one or more statements that is/appears to be indivisible

Lock Variables

```
void Extract(int acc_no, int sum)
{
  lock(L);
  int B = Acc[acc_no];
  Acc[acc_no] = B - sum;
  unlock(L);
}
```

```
void lock(int L)
{
   while (L != 0)
     /* wait */ ;
   L = 1;
}
```

```
void unlock(int L)
{
   L = 0;
}
```

- Does this work?

TSL (Test and Set Lock) Instruction

Atomic instruction provided by most CPUs

TSL(LOCK)

- Test-and-Set-Lock
- Atomically sets memory location LOCK to 1 and returns old value

```
void lock(int L)
{
   while (TSL(L) != 0)
     /* wait */ ;
}
```

- Locks using *busy waiting* are called **spin locks**

Spin Locks

- Waste CPU
  - Should only be used when the wait is expected to be short
- May run into **priority inversion problem**

Priority Inversion Problem and Spin Locks

Two processes:

- H with high priority
- L with low priority
- H should always be scheduled if runnable

Assume the following scenario:

- H is waiting for I/O
- L acquires lock A and enters critical section
- I/O arrives and H is scheduled
- H tries to acquire lock A that L is holding

What happens?

Lock Granularity

```
void Extract(int acc_no, int sum)
{
   lock(L);
   int B = Acc[acc_no];
   Acc[acc_no] = B - sum;
   unlock(L);
}
```

T1:
```
Extract(1, 40);
```

T2:
```
Extract(2, 40);
```

What happens if there are concurrent accesses to **different** accounts?

Lock Granularity

```
void Extract(int acc_no, int sum)
{
   lock(L[acc_no]);
   int B = Acc[acc_no];
   Acc[acc_no] = B - sum;
   unlock(L[acc_no]);
}
```

T1:
```
Extract(1, 40);
```

T2:
```
Extract(2, 40);
```

Lock granularity: the amount of data a lock is protecting

Lock Overhead and Lock Contention

Lock overhead: measure of cost associated with using locks

- Memory space
- Initialization
- Time required to acquire and release locks

Lock contention: measure of number of processes waiting for lock

- More contention, less parallelism

- **Coarser granularity:**
  - Lock overhead?
  - Lock contention?
  - Complexity?

- **Finer granularity:**
  - Lock overhead?
  - Lock contention?
  - Complexity?

Minimising Lock Contention/Maximising Concurrency

Choose finer lock granularity

- But understand tradeoffs

Release a lock as soon as it is not needed

- Make critical sections small!

```
void AddAccount(int acc_no, int balance)
{
   lock(L_Acc);
   CreateAccount(acc_no);
   lock(L[acc_no]);
   Acc[acc_no] = balance;
   unlock(L[acc_no]);
   unlock(L_Acc);
}
```

Read/Write Locks

```
void ViewHistory(int acc_no)
{
    print_transactions(acc_no);
}
```

T1:
```
ViewHistory(1234);
```

T2:
```
ViewHistory(1234);
```

T3:
```
ViewHistory(1234);
```

Any locks needed?

Read/Write Locks

```
void ViewHistory(int acc_no)
{
   print_transactions(acc_no);
}
```

```
void Extract(int acc_no,
             int sum)
{
  lock(L[acc_no]);
  Acc[acc_no] -= sum;
  add_debit(acc_no, sum);
  unlock(L[acc_no]);
}
```

T1: `ViewHistory(1234);`

T2: `ViewHistory(1234);`

T3: `Extract(1234,500);`

Any extra locks needed?

Read/Write Locks

```
void ViewHistory(int acc_no)
{
    lock(L[acc_no]);
    print_transactions(acc_no);
    unlock(L[acc_no]);
}
```

```
void Extract(int acc_no,
             int sum)
{
  lock(L[acc_no]);
  Acc[acc_no] -= sum;
  add_debit(acc_no, sum);
  unlock(L[acc_no]);
}
```

T1: `ViewHistory(1234);`

T2: `ViewHistory(1234);`

T3: `Extract(1234,500);`

What if later only T1 and T2 run?

Read/Write Locks

```
void ViewHistory(int acc_no)
{
   lock_RD(L[acc_no]);
   print_transactions(acc_no);
   unlock(L[acc_no]);
}
```

```
void Extract(int acc_no,
             int sum)
{
  lock_WR(L[acc_no]);
  Acc[acc_no] -= sum;
  add_debit(acc_no, sum);
  unlock(L[acc_no]);
}
```

Read/write locks:

- `lock_RD(L)` → acquire L in read mode
- `lock_WR(L)` → acquire L in write mode
- In write mode, the thread has exclusive access
- Multiple threads can acquire the lock in read mode!

Race Condition

Occurs when multiple threads or processes read and write **shared data** and the final result depends on the relative timing of their execution

- i.e. on the exact process or thread **interleaving**

E.g. the **`Extract`** example → final value of account 8,000 or 9,000

Thread Interleavings

```
int a, b; // shared
void  T1()                  void T2()
{                           {
   a = 1;                      b = 2;
   b = 1;                      a = 2;
}                           }
```

a = 1	a = 1	a = 1	b = 2	b = 2	b = 2
b = 1	b = 2	b = 2	a = 2	a = 1	a = 1
b = 2	b = 1	a = 2	a = 1	a = 2	b = 1
a = 2	a = 2	b = 1	b = 1	b = 1	a = 2
(2, 2)	(2, 1)	(2, 1)	(1, 1)	(2, 1)	(2, 1)

Tutorial Question

Consider the following three threads:

T1	T2	T3
a = 1;	b = 1;	a = 2;
b = 2;		

How many possible interleaving are there?

4

8

12

16

If all thread interleavings are as likely to occur, what is the probability to have a=1 and b=1 after all threads complete execution?

All Thread Interleavings

Memory models

In this course, we assume **sequential consistency**:

The operations of each thread appear in program order

The operations of all threads are executed in some sequential order atomically

But other memory models (due to hardware behaviour and compiler optimisations) exist!

Sequential Consistency vs Weak Memory Models

```
int flag1 = 0, flag2 = 0; // shared
void  T1()             void T2()
{                      {
 flag1 = 1;             flag2 = 1;
 if (flag2 == 0);       if (flag1 == 0)
    critical()             critical()
}                      }
```

Under sequential consistency, it is impossible for both threads to read flag1 = flag2 = 0 in their if statements

Under weak memory models, this is possible!

Advanced reading:
https://www.cs.princeton.edu/courses/archive/fall10/cos597C/docs/memory-models.pdf

Happens-Before Relationship

Formulated by Leslie Lamport in 1976

Partial order relation between events (e.g. instructions) in a trace

Denoted by $a \rightarrow b$ where a, b are events in a trace

Consider a, b with a occurring before b in the trace:

- If a, b are in the same thread, then $a \rightarrow b$
- If a is unlock(L) and b is lock(L), then $a \rightarrow b$
  - (can generalise for other synchronisation mechanisms)

Irreflexive: $\forall a,\ a \not\rightarrow a$

Antisymmetric: $\forall a, b$: $a \rightarrow b$ then $b \not\rightarrow a$

Transitive: $\forall a, b, c$: $a \rightarrow b \wedge b \rightarrow c$ then $a \rightarrow c$

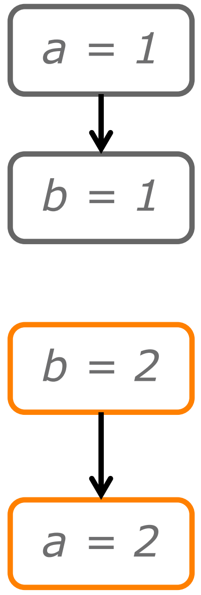
Happens-Before Relationship

A data race occurs between a, b in the trace iff:

- they access the same memory location
- at least one of them is a write
- they are unordered according to happens-before

Happens-Before

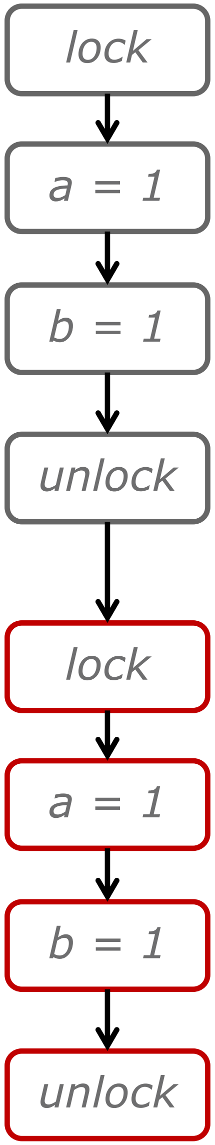
```
int a, b; // shared
void  T1()                     void T2()
{                              {
   a = 1;                         b = 2;
   b = 1;                         a = 2;
}                              }
```



Date race between a =1, a=2
and between b = 1, b =2

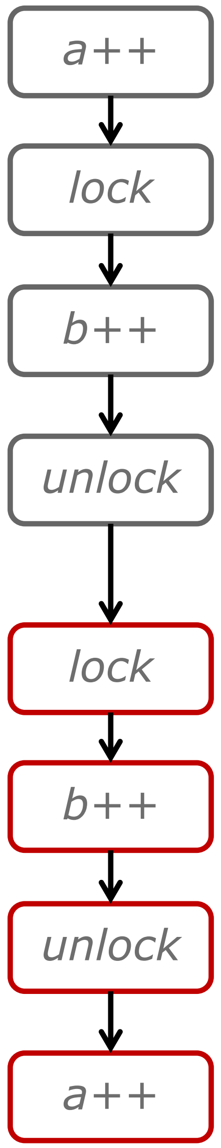
Happens-Before

```
int a, b; // shared
void  T1()               void T2()
{                        {
   lock(L);                 lock(L);
   a = 1;                   b = 2;
   b = 1;                   a = 2;
   unlock(L);               unlock(L);
}                        }
```



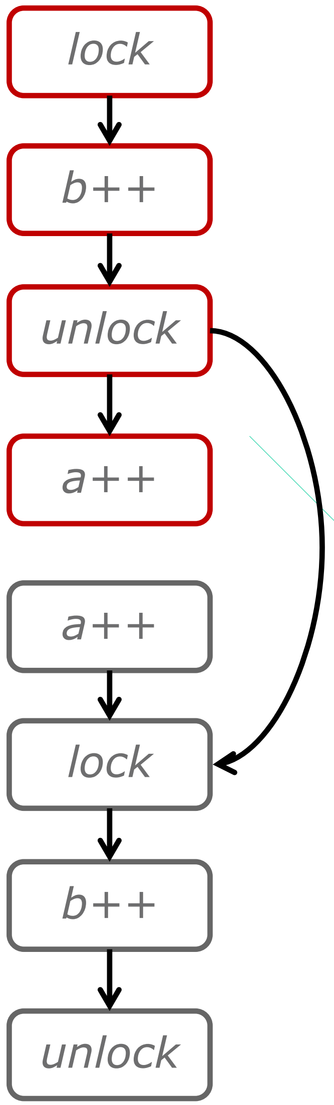
Happens-Before

```
int a, b; // shared
void  T1()                 void T2()
{                          {
   a++;                       lock(L)
   lock(L);                   b++;
   b++;                       unlock(L);
   unlock(L);                 a++;
}                          }
```



Happens-Before

```
int a, b; // shared
void  T1()              void T2()
{                       {
  a++;                    lock(L)
  lock(L);                b++;
  b++;                    unlock(L);
  unlock(L);              a++;
}                       }
```



Order a dynamic execution trace only