Imperial College London

# Operating Systems

## Synchronisation – Part I

# Process Synchronisation

- How do processes synchronise their operation to perform a task?
- Key concepts:
  - Critical sections
  - Mutual exclusion
  - Atomic operations
  - Race conditions
  - Deadlock
  - Starvation
  - Synchronization mechanisms
    - Locks, semaphores, monitors, etc.
- Concepts relevant to both **processes** and **threads**

Imperial College London

# Shared Data Example

Account #1234: £10,000

Extract £1000
from account 1234

Extract £1000
from account 1234

Imperial College London

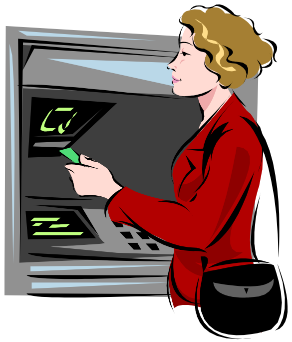
# Shared Data Example

```
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

Imperial College London

# Requirements for Mutual Exclusion

(1) No two processes may be simultaneously inside a critical section

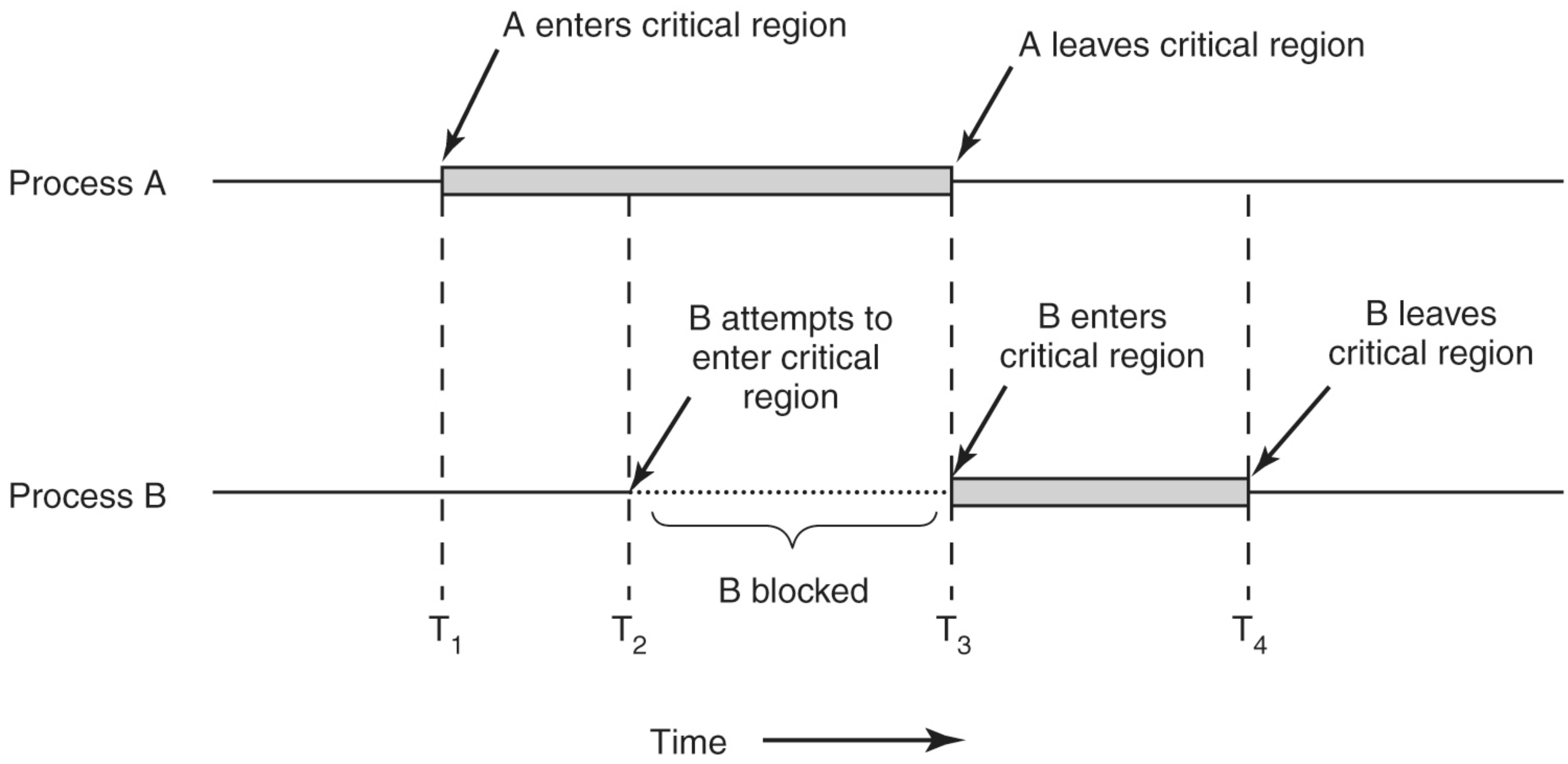
(2) No process running outside the critical section may prevent other processes from entering the critical section
  - When no process is inside a critical section, any process requesting permission to enter must be allowed to do so immediately

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

(4) No assumptions are made about relative the speed of processes

Imperial College
London

# 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

Imperial College London

# 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)?

Imperial College London

# 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

Imperial College London

# 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 $T_K$ 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.

Imperial College London

# 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?
  - Not atomic!

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

Imperial College London

# 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)`
  - Atomically sets memory location `LOCK` to `1` and returns old value

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

*Pseudocode; needs to be written in ASM*

- 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:** a measure of the cost associated with using locks
  - Memory space
  - Initialization
  - ***Time required to acquire and release locks***
- **Lock contention:** a measure of the number of processes waiting for a lock
  - More contention, less parallelism

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

Imperial College London

# Minimizing Lock Contention/Maximizing 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);
}
```

Imperial College London

# 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?
  - They cannot execute concurrently anymore!

Imperial College London

# 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!