University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

Operating Systems
Process Synchronisation

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

Concurrent Execution

Concurrent Execution

- Concurrent Execution is when two processes are executing at the **same** time

- Each cooperative processes may affect the execution of the other process

Concurrent Execution Example

```c
int A;
```

Process 1

```c
A = 1;
if(A == 1)
   printf("Process 1 wins");
```

Process 2

```c
A = 2;
if(A == 2)
   printf("Process 2 wins");
```

Concurrent Execution Example

- The variable **A** is shared by both processes

- When both processes are run concurrently on one processor, which one "wins"?
  - the outcome of the concurrent execution depends on which assignment takes place first (**race condition**)

What about threads?

- If we replaced the processes with threads the outcome will be still the same

Atomic Operations

- Is it possible to get a different value in the variable A if we try and execute the two assignments statements at the same time?
- E.g. A = 1 and A = 2 happen at the same time

Atomic Operations

- It is **not** possible.

- References and assignments are all **atomic** in the CPU
  - This means all read and write operations happen as a single step

Atomic Operations

- An atomic operation cannot be interrupted
- This is to prevent illogical things happening

Atomic Instructions

- This atomicity is provided by hardware

- Higher-level constructs are not atomic in general
  - A higher-level construct is any sequence of two or more instructions

Higher-level Example

```
int B;
```

Process 1

```
B = 0;
while(B < 10) B++;
printf("Process 1 finished");
```

Process 2

```
B = 0;
while(B > -10) B--;
printf("Process 2 finished");
```

- Variable B is shared
  - But increments and decrements are atomic
  - Will the process finish?

Higher-level Example

```
int B;
```

Process 1

```
B = 0;
while(B < 10) B++;
printf("Process 1 finished");
```

Process 2

```
B = 0;
while(B > -10) B--;
printf("Process 2 finished");
```

- The **while** sections above are not atomic
  - The process theoretically might never reach 10 or -10

Higher-level Example

```
int B;
```

Process 1	Process 2

`B = 0;`
`while(B < 10) B++;`
`printf("Process 1 finished");`
	`B = 0;`
	`while(B > -10) B--;`
	`printf("Process 2 finished");`

- Process synchronisation is all about making **high-level** constructs **behave atomically**

Mutual Exclusion & Critical Section

The Milk Problem

- Two flat mates sharing a fridge
- They want to have **at most one** bottle of milk in the fridge at any time

Possible Events

Time	Flatmate A	Flatmate B
10:00	Check the fridge: NO MILK	
10:05	Walk to Shop	Check the fridge: NO MILK
10:10	Arrive at the shop; buy milk	Walk to Shop
10:15	Arrive home; put milk in fridge	Arrive at the shop; buy milk
10:20		Arrive home; put milk in fridge
		TOO MUCH MILK

The Milk Problem

- The reason for this problem is that the behaviour of A and B is not atomic
- To solve this problem A and B must **synchronise** (cooperate)

Important Definitions

- **Synchronisation**: ensuring proper cooperation among processes (or threads), by relying on atomic operations

- **Mutual exclusion** (ME): ensuring that only one process at a time holds or modifies a shared resource
    - ME ensures atomicity

Important Definitions

- **Critical section** (CS): a section of code in a program in which shared resources are manipulated
- Mutual exclusion in a critical section requires serialisation of process access to the critical section

Milk Problem

- The critical section is
  - Check fridge, go shopping, put milk in fridge
- Mutual exclusion:
  - Only let one flatmate do this at a time

Locking and Mutual Exclusion

- Achieving mutual exclusion in a critical section always involve some sort of **locking mechanism**
- Locking is where we prevent someone else for doing something with the shared resource

Locking Rules

- Locking involves three rules:

1. You must **lock** before you enter a critical section
2. You must **unlock** when leaving a critical section
3. You must **wait** when trying to enter critical section if it is locked

Milk Problem Lock

1. Leave note before going shopping (“gone shopping”)

2. Remove note after shopping

3. Don’t go shopping if note has been left

Milk Problem (1st Solution Attempt)

- Let both A and B execute (in a closed loop)

```
if(no_milk) {
  if(no_note) {
    put note;
    buy milk;
    remove note;
  }
}
```

Milk Problem (1st Solution Attempt)

- Mutual exclusion is no guaranteed
  - A and B could execute "if(no_note)" before the note is put on the fridge
- This is really hard to debug
  - It might work most of the time but occasionally fail

Milk Problem (2^{nd} Attempt)

- As a workaround, let us change the meaning of the note:
  - B buys if there is a note
  - A buys if there is no note

Milk Problem (2^{nd} Attempt)

Process A

```
if(no_note) {
  if(no_milk) {
    buy milk;
    add note;
  }
}
```

Process B

```
if(note) {
  if(no_milk) {
    buy milk;
    remove note;
  }
}
```

Milk Problem (2nd Attempt)

- Is the critical section mutually exclusive?

- Yes, but what happens if B goes on holidays?
  - A cannot buy any milk!

Milk Problem (2nd Attempt)

- We have the same problem if B is very slow
  - The **relative speed** of processes can be an issue
- Processes must take turns to be in the critical section
  - This means that the same flatmate **cannot** buy milk twice in a row

Milk Problem (3rd Attempt)

- In order to try to avoid the last issue, let us use **two** notes (**note_A** and **note_B**) and some basic courtesy protocol
- Each process can now examine each other’s status, but not modify it.

Milk Problem (3rd Attempt)

Process A

```
put note_A;
if(no_note_B) {
   if(no_milk) {
     buy milk;
   }
}
remove note_A;
```

Process B

```
put note_B;
if(no_note_A) {
   if(no_milk) {
     buy milk;
   }
}
remove note_B;
```

Milk Problem (3rd Attempt)

- Does this work?

- Better than before.
  - Relative speed is still an issue
  - What happens if they both leave notes at the exact same time?

Milk Problem (4^{th} Attempt)

- Two processes use slightly different code
  - One process stays the same as before
  - The other **waits** until the note is removed

Process A

```
put note_A;  if(no_note_B)
{
   if(no_milk)
      buy milk;
 } else {
   while(note_B){ }
   if(no_milk)
      buy milk;
}
remove note_A;
```

Process B

```
put note_B;  if(no_note_A) {
   if(no_milk) {
      buy milk;
   }
 }
remove note_B;
```

Milk Problem (4th Attempt)

- Is this a solution?
  - It was actually the first solution ever given to the mutual exclusion problem
- But it is not satisfactory

Issues with the Solution

- The solution is complicated and just for this problem
  - We need to pay careful attention to make sure that it really works
  - A similar solution might not be good for more complex problems

Issues with the Solution

- The solution is **asymmetric**
  - Not everybody is performing the same steps
- The solution is **not scalable**
  - How would we make it work for more people sharing milk

Issues with the Solution

- The solution is **inefficient**
  - While A is waiting it is consuming processor time
    - Called **busy-waiting**

- The solution is not scalable
  - How would we make it work for more people sharing milk

Consequence

- Because of the problems with the 4^{th} solution it is necessary to have standard synchronisation mechanisms in an operating system

- These can automatically fulfill some minimum requirements

Requirements for True Solution for CS

1. Mutual exclusion
    - One process **at most** inside the CS at any time

2. Bounded waiting (no starvation)
    - A process attempting to enter its CS will eventually do so

Requirements for True Solution for CS

3. Progress
   - A process executing outside a CS cannot prevent another process from entering it
   - If several processes are attempting to enter a CS at the same time the decision on which one goes in cannot be indefinitely postponed
   - A process can't stay inside its CS forever (or exit in there)

Requirements for True Solution for CS

- These conditions are **necessary and sufficient** for process synchronisation
  - Provided that basic operations are **atomic**
- No assumptions are made about: number of processes, relative speed of processes, or underlying hardware

Desirable Properties of a ME Mechanism

- Simple
- Systematic and easy to use
  - E.g. just bracket the critical section
- Easy to maintain

Desirable Properties of a ME Mechanism

- Efficient
  - Does not use a lot of resources while waiting
    - No busy-waiting
  - Overhead due to entering and leaving critical sections has to be small
    - At least smaller than the work inside it
  - Scalable:
    - It should work when many threads share the critical section

Implementations

1. There are three basic mechanisms for implementing mutual exclusion in critical sections

   - Semaphores
     - Simple, but hard to program with
     - Very low level

Implementations

2. Monitors
   - Higher level mechanism
   - Requires higher-level programming languages

3. Messages
   - Synchronisation without shared memory
   - Uses IPC messages instead

Semaphores

Semaphore

- A semaphore is a protected integer variable **S** with an associated queue of waiting processes
- Only two atomic operations **P()** and **V()** can be performed on S

Semaphore Operations

P(S)

```
if(S > 0){
  S--;
} else {
  addToQueue();
}
```

V(S)

```
if(queue_not_empty){
  nextInQueue();
} else {
  S++
}
```

Semaphores

- Initial value of S is **how many** threads can enter critical section at the same time
- Processes in the queue are **blocked**
- The operations are **atomic**, so only one process at a time can execute them

Mutual Exclusion using Semaphores

- To implement mutual exclusion using semaphores, we do the following:
  - Initialise S at 1
  - To enter critical section we execute **P** on its semaphore
  - When leaving the critical section we execute **V** on its semaphore

Counting Semaphores

- If we initialise S > 1, more than one process at a time can get into the CS
  - This means that there is no mutual exclusion
  - This is used in a particular type of semaphores called **counting semaphores**
- A semaphore with the initial value of 1, is also called a **binary semaphore** or **mutex**

Why P and V?

- Dijkstra, who first proposed semaphores, was Dutch
- In Dutch
  - **P**roberen means to probe
  - **V**erhogen means to increment

Creating Semaphores

- Creating semaphores is different for each language
- All will require an initial value
- Other information like who can access may also be required

Semaphores Example

- Atomically increment a shared variable X by n concurrent processes
- Setup:

```
int x;
semaphore S(1, NULL);
```

- Code in threads:

```
P(S);
x = x + 1;
V(S);
```

Semaphores for Milk problem

- Declare semaphore:

```
semaphore S(1, NULL);
```

- Processes code:

```
P(S);
if(no_milk)
  buy_milk
V(S);
```

The Produce/Consumer Problem

- This is a common synchronisation problem
- Consider a **producer process** supplying a resource to a **consumer process**
    - Producer: creates instances of a resource
    - Consumer: uses up instances of a resource

The Produce/Consumer Problem

- Producer and consumer share a **buffer**
  - The producer put resources into the buffer
  - The consumer takes resources from the buffer
  - The buffer has a **finite** size

Problem Constraints

- The consumer must wait for producer if the buffer is empty
- Producer must wait for consumer if the buffer is full

Synchronisation

- We need to keep the producer and consumer synchronised
  - The buffer is the critical section in this problem
  - We usually need mutually exclusive access to the buffer

P/C Solution with Semaphores

Producer

```
while(true) {
  msg = produce();
  P(S);
  if(buffer_full)
    wait();
  put_msg(msg);
  V(S);
}
```

Consumer

```
while(true) {
  P(S);
  if(buffer_empty)
    wait();
  msg=get_msg();
  V(S);
  consume(msg);
}
```

P/C Solution with Semaphores

- The solution cannot work
  - If the buffer is **empty** the consumer waits and the producer cannot add any items
  - If the buffer is **full** the producer waits and the consumer cannot remove any items
- These situations are called **deadlock**

P/C Solution with Counting Semaphores

- We add two more semaphores to the code:
  - One to count the number of **empty** slots in the buffer
  - One to count the number of **full** slots in the buffer

P/C Solution with Counting Semaphores

- Semaphores:

```
int N = buffer_size;
semaphore S(1, NULL);
semaphore full_s(0, NULL);
semaphore empty_s(N, NULL);
```

Producer

```
while(true) {
   msg=produce();

   P(empty_s);
   P(S);
   put_msg(msg);
   V(S);
   V(full_s);
}
```

Consumer

```
while(true) {
   P(full_s);
   P(S);
   msg=get_msg();
   V(S);
   V(empty_s);


consume_msg(msg);
}
```

Questions

- True or false?
  - The value of a semaphore cannot be modified by an operation other than P or V
  - The initial value of a semaphore cannot be zero
- Is the order of Ps important? And the order of Vs?

Monitors

The Problem with Semaphores

- Semaphores are nice but they have an important problem

  - Using more than one semaphore can lead to **deadlocks**, if order of Ps not correctly set

  - It is difficult to program with semaphores

Monitors

- Monitors are higher-level sync primitives
  - A **monitor** is a collection of procedures, variables and data grouped together in a special kind of structure
  - Their aim is to avoid the problems that can arise when protecting critical sections with semaphores

Example Monitor

```
monitor example {
    int i;                  (Internal data)
    condition c;            (Condition variable)

    void p() {              (Monitor procedure)
        ...
    }
}
```

Monitor Rules

- Processes may call monitor procedures whenever they wish
- Processes can’t access internal monitor data (variables, etc.) using external procedures

Monitor Rules

- Only one process can be active inside a monitor
- Mutual exclusion inside the monitor is guaranteed by the compiler

Blocking in Monitors

- Mutual Exclusion is implemented using internal **condition** variables

- Conditions have two operations
  - Wait
  - Signal

wait(condition)

- This is executed when the monitor discovers that a process cannot proceed
  - The monitor **blocks** a process calling wait() and makes it wait on condition
  - Another process can then be allowed into the monitor
  - A process that calls wait() is **always blocked**

signal(condition)

- This is executed to wake up a process that is waiting on **condition**
- After executing **signal**() the calling process must exit the monitor immediately
    - This guarantees mutual exclusion

Monitor for Producer/Consumer

- Requires two conditions:
  - Buffer is full
  - Buffer is empty

- Requires some internal data so we know when it is empty or full

Monitor for Producer/Consumer

- Requires two operations
  - Put a message in the buffer
  - Remove a message from the buffer

Monitor for Producer/Consumer

```
monitor pr_co {
    int count;
    condition full_s,empty_s;
    void put(msg) {
        if(count == N)
            wait(empty_s);
        put_msg(msg);
        count++;
        if(count == 1)
            signal(full_s);
    }
```

Monitor for Producer/Consumer

```
msg get() {
    if(count==0)
        wait(full_s);
    msg=get_msg();
    count--;
    if(count==N-1)
        signal(empty_s);
  }
}
```

Producer and Consumer Code

Producer

```
while(true) {
  msg=produce();
  pr_co.put(msg);
}
```

Consumer

```
while(true) {
  msg=pr_co.get();
  consume_msg(msg);
}
```

Producer and Consumer Code

- Simplest possible consumer and producer code
- Monitor takes care of any issues, not producer or consumer

Messages

Messages System

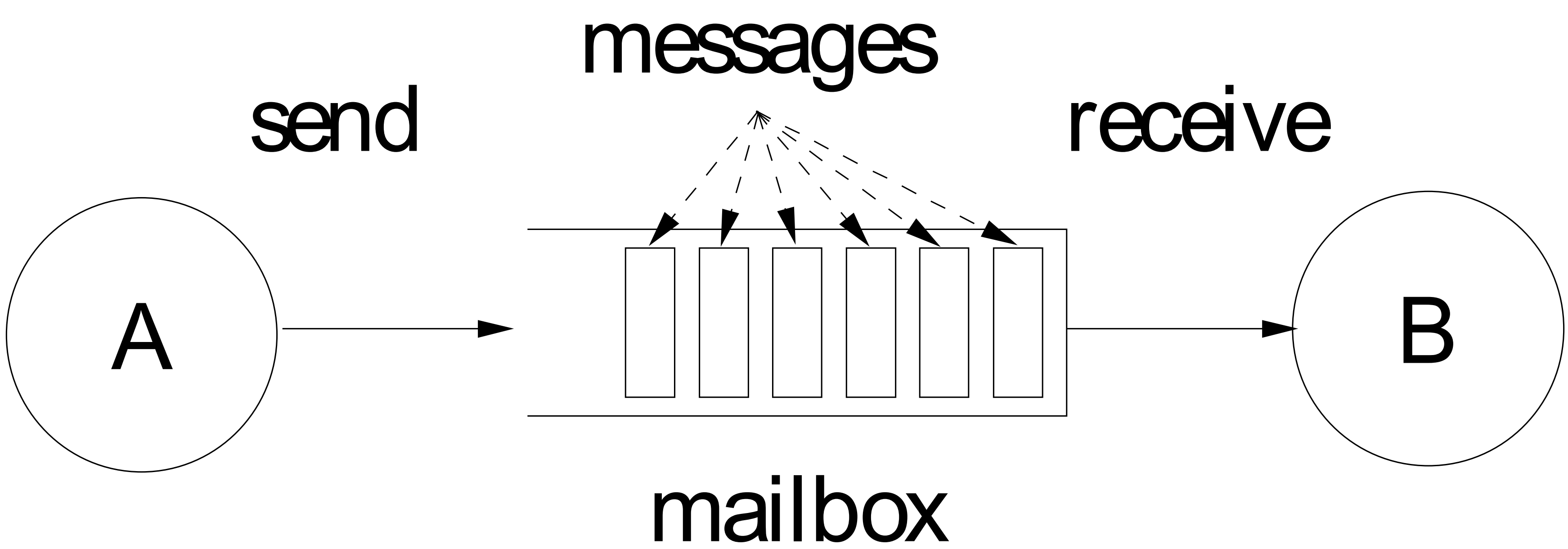
- Synchronisation with both semaphores and monitors relies on **shared memory**
- The message system is a mechanism for inter-process communication (IPC), that can also be relied upon for synchronisation without having to share memory

Message System

- Elements in the message system:
  - **message**: information that can be exchanged between two (or more) processes or threads
  - **mailbox**: a place where messages are stored between the time they are sent and the time they are received

Message System Scheme

- Process A sends messages to process B using a mailbox



Message System Scheme

- One process or the other owns the data
  - Never two at the same time
- If processes need to send messages both ways, we need two mailboxes

Message System Operations

- Two basic operations
  - **send(mailbox, message)**
    - Put a message in the mailbox
  - **receive(mailbox, message)**
    - Remove a message from the mailbox
- In practice we also need the system calls **create(mailbox)** and **delete(mailbox)**

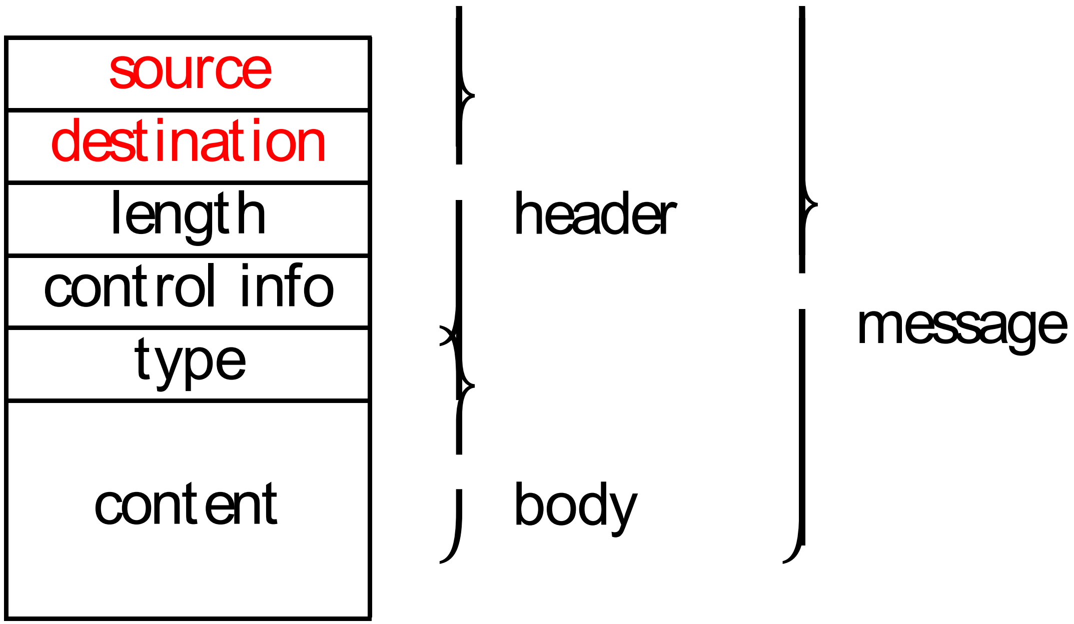
Message System Addressing

- The message identifies **source** and **destination** processes
  - Through their PIDs
- There may be multiple destinations, or destination may be unspecified

Message Format

- Depends on the objectives of the system
  - **fixed-length**: minimise processing and storage overhead
  - **variable-length**: more flexible approach for operating systems

Message Format



Why Use Messages?

- Communicating processes sometimes need to be separate
  - They do not trust each other
  - The programs were written at different times by different programmers who knew nothing about each other
  - They run on different processors
  - Less error-prone without shared memory

Mutual Exclusion with Messages

- Mutual exclusion must be guaranteed solely relying on the mailbox

- Two levels of synchronisation in a message system
  - Producer/Consumer Constraints
  - Execution flow constraints

Producer/Consumer Constraints

- `receive()`
  - Calling thread waits when the mailbox is empty until a message is added
- `send()`
  - Calling thread waits when mailbox is full until there is space for another message

Execution Flow Constraints

- Send and receive: could be blocking or non-blocking
  - *Blocking send*: when a process sends a message it **blocks** until the message is received at the destination.
  - *Non-blocking send*: After sending a message the sender **continues** with its processing without waiting for it to reach the destination

Execution Flow Constraints

- Send and receive: could be blocking or non-blocking
  - *Blocking receive*: When a process executes a receive it waits **blocked** until the receive is completed and the required message is received
  - *Non-blocking receive*: The process executing the receive proceeds **without waiting** for the message(!).

Execution Flow Constraints

- The most common combination is that of
  - *Blocking Receive*
  - *Non-blocking send*

Mutual Exclusion using Messages

- Mutual exclusion can be achieved using a **single shared mailbox**
- Processes can **enter** the critical section whenever they **receive** a message
- Whenever they **leave** the critical section they must **send** a new message to the mailbox

Parent process

```
create_mailbox(mbox);
send(mbox, NULL);
parallel {
  P(1);
  P(2);
    ...
  P(N);
}
```

- Creates N processes
- Empty message required to access critical section

Child Process

```
while(true) {
  receive(mbox,msg);

  Critical Section

  send(mbox,NULL);
}
```

P/C Algorithm Using Messages

- Not only we need to ensure mutual exclusion but also to keep a count
- For this two mailboxes are needed
  - one mailbox for consumers: data produced and consumed
  - one mailbox for producers: accountancy purposes

P/C Algorithm Using Messages

- Producers generate data and send it to the consumers's mailbox
  - Only when the producer’s mailbox indicates there is at least one free slot in the buffer
- Consumers take messages from the consumer’s mailbox when available
  - Then send empty messages to the producer’s mailbox to signal new free slot

Parent Process

```
create mailbox(cons_mbox);
create mailbox(prod_mbox);
for(i=0;i<N;i++)
   send(prod mbox,NULL);

parallel {
   producers();
   consumers();
}
```

Producer Process

```
while(true){
   msg=produce();
   receive(prod_mbox,token);
   send(cons_mbox,msg);
}
```

Consumer Process

```
while(true){
   receive(cons_mbox,msg);
   consume_msg(msg);
   send(prod_mbox,NULL);
}
```

Next Class

- Next Lecture:
  - Deadlock & Starvation
- Review Chapter 5