# Sistemi Distribuiti

**I sistemi distribuiti sono caratterizzati da:**

- **Archittettura su cui devono essere eseguiti**
- **Sistema software che lo costituisce**

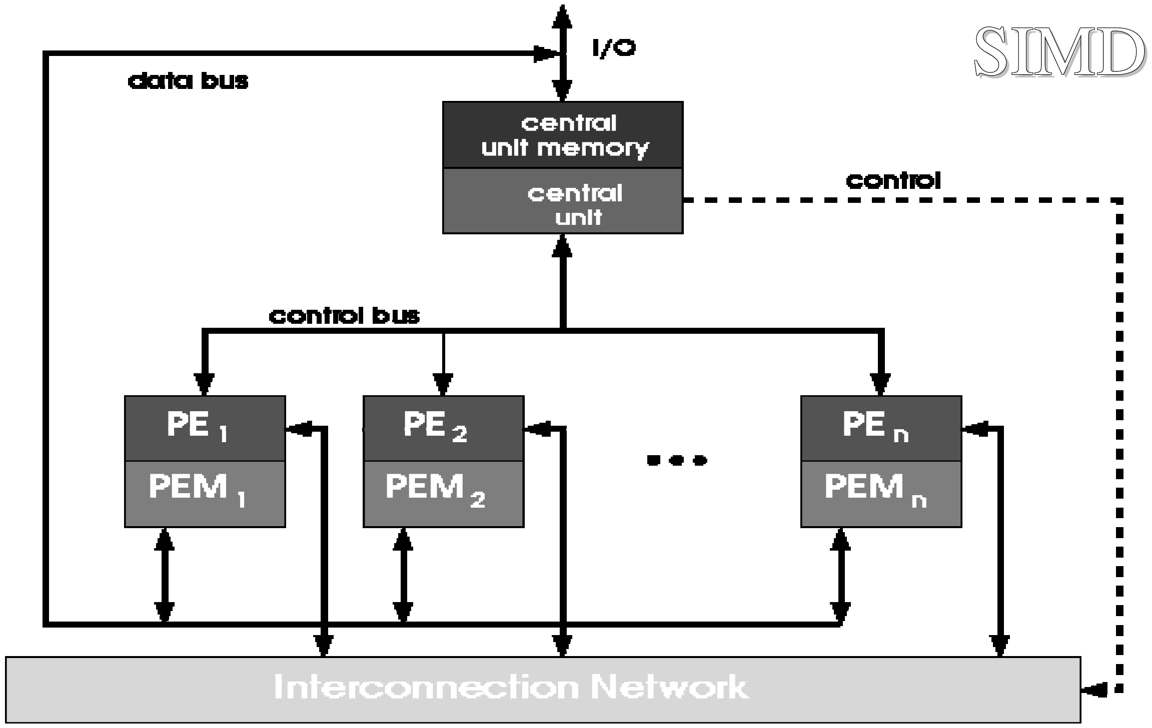
**Noi ci occuperemo dell'aspetto software però per sviluppare tale software è essenziale conoscere l'hardware relativo.**

# Classificazione di Flynn

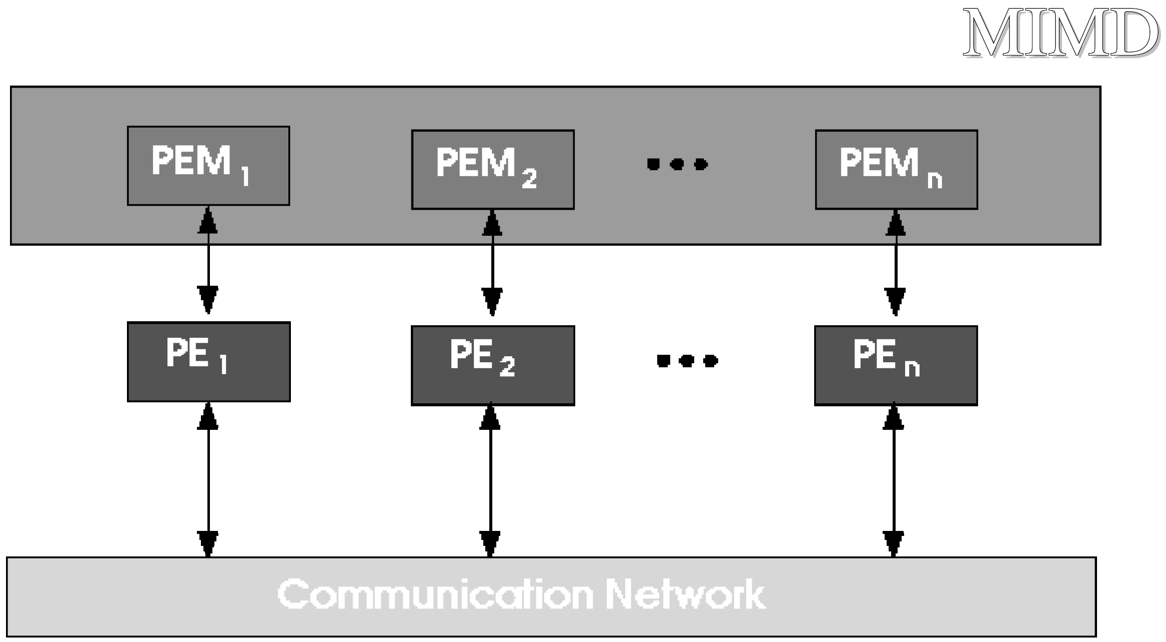
**La tassonomia di Flynn distingue il numero di flussi dati e di controllo in un sistema.**

- **SISD**
- **SIMD**
- **MIMD**

**I computer SISD (single instruction single data stream) sono i computer monoprocessore.**




**$PE_i$ sono delle ALU con registri e memoria ($PEM_i$). L'elaborazione dei $PE_i$ è coordinata dall'unità centrale. Es. Array Processor, Cray.**




**I $PE_i$ sono separati dalla Memoria. La memoria può essere sia distribuita che condivisa. Nessun processore ha un ruolo speciale (es. di coordinare l'operato degli altri processori). I $PE_i$ comunicano tramite la rete.**

**Es. le attuali reti di computer.**

# Tassonomia Alternativa

**La tassonomia di Flynn la più diffusa, ma è decisamente datata (1972).**

**Una tassonomia più moderna divide i sistemi distribuiti in multiprocessori e multicomputer.**

**I n un multiprocessore esiste un'unica memoria centrale condivisa da tutte le CPU; un meccanismo HW coordina gli accessi alla memoria. (Accoppiamento Stretto).**

**In un multicomputer ogni macchina ha una memoria privata (Accoppiamento Lasco, ci sono pochi conflitti e non necessario un coordinatore HW).**

# Sistemi Distribuiti (Aspetti Software)

**Un sistema distribuito è un programma composto da un insieme di processi:**

- **Autonomi**
- **Interagenti**
- **Distribuiti**

# Utilitá dei
# Sistemi Distribuiti

**Un sistema distribuito può essere utilizzato per:**

- **condividere risorse,**
- **incrementare le prestazioni,**
- **Aumentare l'affidabilità,**
- **Aumentare la disponibilità dei servizi offerti**

# Processi

**Def. Un <u>processo</u> è una sequenza di codice in esecuzione assieme al suo ambiente.**

**È un'attività controllata da un programma che si svolge su un processore, quindi, un programma può essere composto da più processi.**

**<u>Nota</u>: I processi sono l'unità di esecuzione in ambiente distribuito.**

# Processi: Caratteristiche

## Distribuiti

**Sono eseguiti su computer e ambienti diversi.**

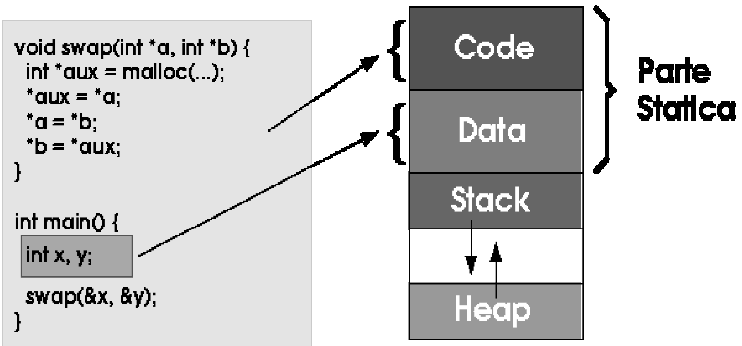
## Autonomi

**Ogni processo ha un proprio ciclo di vita indipendente da quello degli altri processi. Durante il proprio ciclo di vita ogni processo porta a termine il proprio compito.**

## Interagenti

**I processi durante la propria computazione necessitano di servizi offerti da altri processi. Se ci sono molte interazioni tra due processi si parla di accoppiamento forte, altrimenti di accoppiamento lasco.**

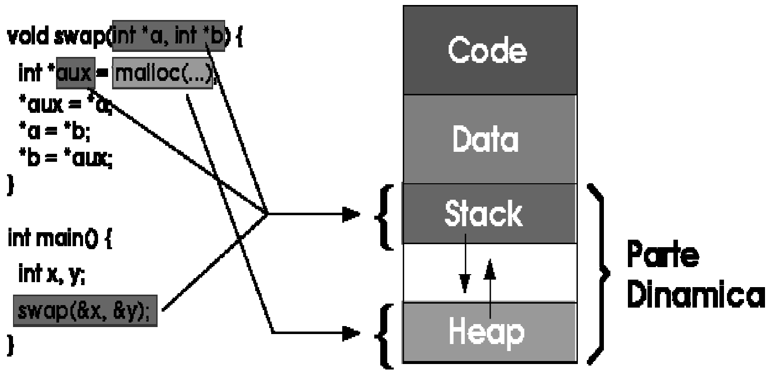
# Struttura dei Processi in Memoria




**Un processo è composto da una parte statica :**

- **Area Codice**
- **Area Dati Globali**

# Struttura dei
# Processi in Memoria




**Un processo è composto da una parte dinamica:**

- **Stack, e**
- **Heap**

# Stato dei Processi

**Lo stato di avanzamento di un processo può essere:**

- **Running: il processore è assegnato al processo che può avanzare.**
- **Blocked: il processo non dispone del processore perché è bloccato in attesa di un servizio.**
- **Ready: il processo è in attesa dell'assegnazione del processore che è occupato da un altro processo.**

## Threads

**Threads (o processi leggeri) sono dei fili di esecuzioni presenti all'interno di un processo.**

**Praticamente il flusso esecutivo di un processo viene scomposto in più flussi concorrenti.**

**I Threads di uno stesso processo condividono l'area dati e codice. È, pertanto, necessaria una sincronizzazione nell'accesso ai dati globali.**

**<u>Vantaggio</u>: il cambio di contesto tra threads è più veloce che tra processi.**

## Threads (Lifecycle)

- **Un thread viene creato ed inizia la propria esecuzione**

  **il nuovo thread avrà una propria identità ed il SO lo tratterà come qualsiasi altro processo, applicandogli le proprie politiche di schedulazione. Il thread è <u>running</u>.**

- **Può mettersi in attesa del verificarsi di una condizione**

  **il passaggio dallo stato di dormiente (<u>sleeping</u> o <u>waiting</u>) a quello di attivo (<u>running</u>) è regolato dal SO tramite segnali.**

- **Conclude la propria computazione**

  **il flusso esecutivo torna a confluire in quello che lo ha originato. Il thread è <u>stopped</u>.**

# Threads in Java

**I l multi−threading permette di scrivere server che gestiscono più richieste contemporaneamente.**

- ***Java ha i thread nativi***

  **i thread sono supportati a livello di linguaggio.**

- ***Java ha la classe Thread per gestire i thread***

  **C'è una corrispondenza tra i thread forniti dal linguaggio e la classe Thread.**

- ***I thread sono controllati tramite i metodi della classe Thread***

# Threads in Java

# Scope delle Variabili nei Threads

- **Variabili locali ad un metodo sono locali ai thread**

  **Modifiche ad una variabile locale ad un thread non sono visibili agli altri thread.**

  **Variabili locali – variabili dichiarate in un metodo.**

**Nota: non sono visibili perchè i thread non condividono lo stack.**

- **Variabili globali sono condivise da tutti i thread**

  **Variabili globali – campi statici o di istanza della classe.**

**Nota: sono condivise perch non sono allocate sullo stack.**

# Scope delle Variabili nei Threads

```
Class Test {
   static int staticW;
   Vector instanceX;

   void aMethod(int paramY) {
      float localZ = staticW;
      instanceX.addElement(new Float(localZ));
   }
}
```

# Sincronizzazione

**La condivisione delle variabili globali da parte di più thread può creare problemi di consistenza.**

**Es.**
```
int balance;
boolean withdraw(int amount) {
   if (balance – amount ≥ 0) {
      balance –= amount;
      return true;
   }  return false;
}
```

| Thread 1 | Thread 2 | Balance |
|---|---|---|
| If (balance – amount ≥ 0) | | 15 |
| | if (balance – amount ≥ 0) | 15 |
| Balance –= 10 | | 5 |
| | Balance –= 10 | –5 |

# Sincronizzazione

- **Ogni oggetto ha un semaforo associato**

  **ogni semaforo ha due operazioni:**

  - **Get, e**
  - **Release**

  **ogni thread quando accede ad una zona critica tenta di prendere possesso del semaforo (Get); se non ci riesce verrà messo a dormire e ci ritenterà quando un altro thread rilascerà (Release) tale semaforo.**

# Sincronizzazione

- **I semafori sono nascosti al programmatore**

**una regione critica viene introdotta dallo statement:**

```
Synchronized (anObject) {
    // regione critica
}
```

**si possono sincronizzare sia classi che metodi.**

**I thread prima di entrare nella regione critica devono ottenere il via libero dal semaforo relativo.**

# Sincronizzazione

**Il metodo withdraw dell'esempio precedente è una sezione critica. Il codice corretto è:**

```
sychronized int balance;
    boolean withdraw(int amount) {
        if (balance – amount ≥ 0) {
            balance –= amount;
            return true;
        }  return false;
    }
```

**in alternativa si poteva sincronizzare solo il corpo del metodo ottenendo lo stesso risultato.**

**<u>Nota</u>: un thread che sta eseguendo una sezione critica e chiede di eseguirne un'altra, ottiene immediatamente l'accesso, evitando i deadlock**

# La Classe Thread

- **È una rappresentazione a run−time di un thread**

  **tramite un'istanza di questa classe è possibile manipolare il thread associato.**

**I metodi principali sono:**

- **Start**
- **Stop**
- **Run**
- **Interrupt**
- **Join**

# I Metodi della Classe Thread

***void run()***

**Viene chiamato automaticamente quando il thread è attivato. Deve essere ridefinito per costruire thread con comportamenti particolari.**

***void start()***

**Permette di attivare un thread. Una chiamata a questo metodo chiama a sua volta il metodo run().**

```
Thread execution = new Thread(anObject);
execution.start();
```

# I Metodi della Classe Thread

***void stop()***

**Permette di fermare un thread. Un thread stoppato non può successivamente essere fatto ripartire. L'uso di questo metodo viene sconsigliato perché può lasciare un oggetto in uno stato inconsistente.**

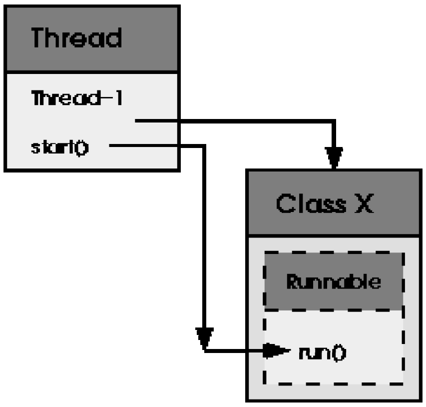
***void interrupt()***

**Permette di interrompere l'esecuzione di un thread, solo quando lo stato dell'oggetto corrispondente è consistente. Cioè quando il thread è dormiente o in attesa di un evento.**

***void join()***

**Permette di bloccare il chiamante fintanto che il thread associato termina la propria esecuzione.**

# Interfaccia Runnable




**Implementare l'interfaccia Runnable è un modo alternativo per rendere una classe multi-thread.**

**Tramite i costruttori della classe Thread, ad una classe X che implementa l'interfaccia Runnable può essere associato un thread.**

**L'attivazione del metodo start() dell'oggetto thread attiverà il metodo run() della classe X.**

# Meccanismi di Notifica

**Alcuni metodi ereditati dalla classe <u>Object</u> permettono di notificare a dei thread l'occorrenza di uno specifico evento.**

***void wait()***

**inviato ad un oggetto, forza il thread chiamante ad attendere che un altro thread invii il messaggio notify() a quell'oggetto.**

***void notify()***

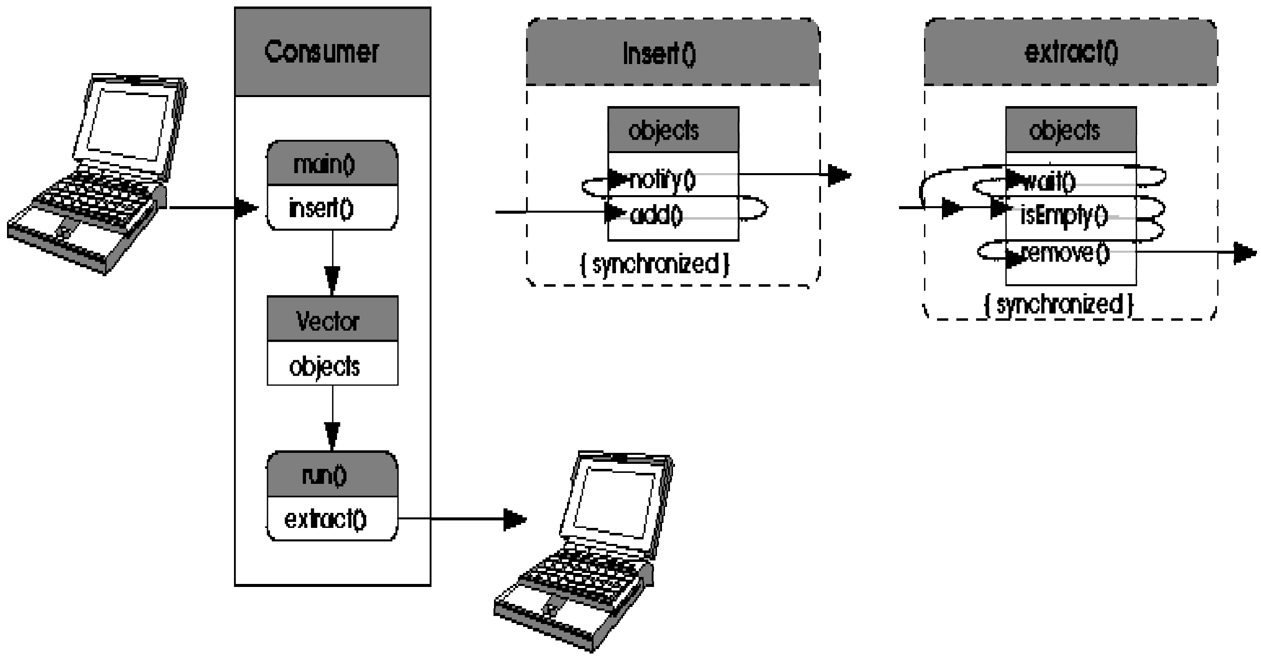
**risveglia un metodo in attesa sull'oggetto destinatario del messaggio.**

***void notifyAll()***

**risveglia tutti i metodi in attesa sull'oggetto destinatario del messaggio.**

**<u>Nota</u>: i thread devono essere sincronizzati sull'oggetto destinatario.**

# Produttore &Consumatore Multi–Threaded



## Struttura Generale

```
public class Consumer extends Thread {
    protected Vector objects;

    public Consumer() {
        objects = new Vector();
    }
                ...
 }
```

## Consumatore

```
Public void run() {
    try {
        while(true) {
            Object object = extract();
            System.out.println(object);
        }
    } catch(InterruptedException ignored) {}
}

protected Object extract() throws InterruptedException {
    synchronized(objects) {
        while(objects.isEmpty())  objects.wait();
        Object object = objects.firstElement();
        return object;
    }
 }
```

# Produttore

```
public void insert(Object object) {
    synchronized(objects) {
        objects.addElement(object);
        objects.notify();
    }
}

public static void main(String[] args) ... {
    Consumer consumer = new Consumer();
    consumer.start();
        ...
    while((line = keyboard.readline()) != null) {
        consumer.insert(line);
        Thread.sleep(1000);
    }
}
```

# Classe ReThread

- **La creazione di thread è inefficiente**

  Per ogni nuovo thread occorre riservare dello spazio in memoria, con tutte le inefficienze del caso.

- **Classe ReThread**

  Permette di riciclare i thread non più usati. Si modifica il metodo start in modo da utilizzare ReThread.

```
public synchronized void start() {
   synchronized (threads) {
      if (threads.isEmpty())    // crea un thread nuovo
      else {
           rethread = (ReThread) threads.lastElement();
           threads.setSize(threads.size()–1);
           rethread.start0(this);
 } } }
```

# Classe ReThread (segue)

```
Protected sychronized void start0( ReThread rethread) {
  this.rethread = rethread;
  target = rethread.target;
  notify();
}
```

- Se interrotti i thread vanno rimessi in circolo.

```
public void run() {
   while (true) {
      target.run();
      target = null;
      threads.addElement(this);
      while (target == null) wait();
   }
}
```

# Riferimenti Bibliografici


- Appunti di Sistemi Operativi su Sistemi Distribuiti (Prof. Gianuzzi).
- Java Network Programming –Hughes et al.