Outline

1. Introduction
2. Configurations
3. Programming
4. Threads
Introduction
What is parallel processing?

- **Parallel processing** (also called **multiprocessing**)
  - situation in which two or more processors operate in unison
  - i.e. two or more CPUs are executing instructions simultaneously
  - each CPU can have a RUNNING process at the same time
  - Process manager must coordinate each processor
  - Process manager must synchronise the interaction among CPUs

- enhance throughput and increase computing power
What is parallel processing?

- **Example: Information Retrieval System**
  - **Processor 1**
    - accepts a query
    - checks for errors
    - passes request to Processor 2
  - **Processor 2**
    - searches database for required information
  - **Processor 3**
    - retrieves data from database (if kept off-line in secondary storage)
    - data placed where Processor 2 can get it
  - **Processor 2**
    - passes information to Processor 4
  - **Processor 4**
    - routes the response back to the originator of the request
Benefits

- Increased reliability
  - more than one CPU
  - if one fails, others can absorb the load
    - failing processor must inform other processors
    - OS must re-structure its resource allocation strategies
- faster processing
  - instructions processed two or more at a time
    - allocate CPU to each job
    - allocate CPU to each working set
    - subdivide individual instructions, called *concurrent programming*
- Challenges:
  - *How to connect the processors in configurations?*
  - *How to orchestrate their interaction?*
Configurations
Master/Slave configuration

- Master/Slave configuration is asymmetric
  - Essentially a single processor with additional “slaves”
  - Master processor responsible for managing entire system
    - maintains status of processes, storage management, schedules work for slave processors, executes all control programs.
  - suited for environments with front-end interactive users, and back-end batch job mode
Master/Slave configuration

- **Advantage:**
  - Simplicity

- **Disadvantage:**
  - Reliability no higher than for single processor (if master fails the whole system fails)
  - Poor use of resources (if master is busy, slave must wait until master becomes free until it can be assigned more work)
  - Increases the number of interrupts (slaves must interrupt the master every time they need OS intervention e.g. IO requests), creating long queues at the master processor
Loosely Coupled configuration

- Loosely Coupled system features several complete computing systems
  - each has its own memory, IO devices, CPU, and OS
  - each processor controls its own resources
  - each processor can communicate and cooperate with others
    - job assigned to one processor, and will remain there until finished
    - job scheduling based on several requirements and policies
      (new jobs may be assigned to the processor with lightest load)
Loosely Coupled configuration

- **Advantage:**
  - Isn’t prone to catastrophic system failures (when a processor fails, others can continue work independently)
- **Disadvantage:**
  - Difficult to detect when a processor has failed
Symmetric configuration

- Symmetric configuration best implemented if processors are the same type
  - Processor scheduling is decentralised
  - Single copy of OS and a global table listing each process and its status (stored in a common area of memory)
  - Each processor uses the same scheduling algorithm
  - If interrupted, processor updates process list and finds another to run (processors are kept busy)
  - Any given job can be executed on several processors
  - Presents a need for process synchronisation
Symmetric configuration

- **Advantage:**
  - Reliable
  - Uses resources effectively
  - Balance loads well
  - Can degrade gracefully in the event of a failure

- **Disadvantage:**
  - Processors must be well synchronised to avoid problems of races and deadlocks
- Multiprocessing can refer to one job using several processors.
- This requires a programming language and computer system that can support it, called **concurrent processing system**.
  - Most programming languages are serial - instructions executed one at a time.
    - To resolve and arithmetic expression, every operation is done in sequence.
**Example:**

- \( A = 3 \times B \times C + \frac{4}{(D + E)} \times (F - G) \)

<table>
<thead>
<tr>
<th>step</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((F - G))</td>
<td>Store difference in (T_1)</td>
</tr>
<tr>
<td>2</td>
<td>((D + E))</td>
<td>Store sum in (T_2)</td>
</tr>
<tr>
<td>3</td>
<td>((T_1) \times (T_2))</td>
<td>Store power in (T_1)</td>
</tr>
<tr>
<td>4</td>
<td>(\frac{4}{(T_1)})</td>
<td>Store quotient in (T_2)</td>
</tr>
<tr>
<td>5</td>
<td>(3 \times B)</td>
<td>Store product in (T_1)</td>
</tr>
<tr>
<td>6</td>
<td>((T_1) \times C)</td>
<td>Store product in (T_1)</td>
</tr>
<tr>
<td>7</td>
<td>((T_1) + (T_2))</td>
<td>Store sum in (A)</td>
</tr>
</tbody>
</table>
1. Arithmetic expressions can be processed differently if we use a language that allows concurrent processing.

2. Define COBEGIN and COEND to indicate to the compiler which instructions can be processed concurrently.

### Example

#### COBEGIN

<table>
<thead>
<tr>
<th>1</th>
<th>$T_1 = 3 \times B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$T_2 = D + E$</td>
</tr>
<tr>
<td>3</td>
<td>$T_3 = F - G$</td>
</tr>
</tbody>
</table>

#### COEND

<table>
<thead>
<tr>
<th>2</th>
<th>$T_4 = T_1 \times C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$T_5 = T_2 \times T_3$</td>
</tr>
</tbody>
</table>

#### COBEGIN

| 3 | $A = T_4 + 4/T_5$ |

#### COEND

<table>
<thead>
<tr>
<th>step</th>
<th>proc.</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$3 \times B$</td>
<td>Store difference in $T_1$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$(D + E)$</td>
<td>Store sum in $T_2$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$(F - G)$</td>
<td>Store difference in $T_3$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$(T_1) \times C$</td>
<td>Store product in $T_4$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$(T_2) \times (T_3)$</td>
<td>Store power in $T_5$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>$4/(T_5)$</td>
<td>Store quotient in $T_1$</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>$(T_4) + (T_1)$</td>
<td>Store sum in $A$</td>
</tr>
</tbody>
</table>
Concurrent Programming Applications

- Increased computational speed
  - increased complexity of programming language
  - increased complexity of hardware (machinery and communication among machines)
  - programmer must explicitly state which instructions are to be executed in parallel, called **explicit parallelism**
    - solution: automatic detection by the *compiler* of instructions that can be performed in parallel, called **implicit parallelism**
Case 1: Array Operations

- To perform an array operation within a loop in three steps, the instruction might say:
  - for(j=1; j<=3; j++)
    a(j) = b(j) + c(j);
- If we use three processors, the instruction can be performed in a single step:
  - Processor #1 performs: a(1) = b(1) + c(1)
  - Processor #2 performs: a(2) = b(2) + c(2)
  - Processor #3 performs: a(3) = b(3) + c(3)
Case 2: Matrix Multiplication

To perform $C = A \times B$ where $A$ and $B$ represent two matrices:

- $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
- Several elements of a row of $A$ are multiplied by the corresponding elements of the column in $B$.
- Serially, the answer can be computed in 45 steps ($5 \times 9$)
- With three processors the answer takes only 27 steps, multiplying in parallel ($3 \times 9$)
Threads
We have considered cooperation and synchronisation of traditional processes (known as heavyweight processes):
- require space in main memory where they reside during execution
- may require other resources such as data files or IO devices
- pass through several states: ready, running, waiting, delayed, blocked

This requires an overhead from swapping between main memory and secondary storage.

To minimise overhead time, implement the use of **threads**
- defined as a smaller unit within a process, that can be scheduled and executed (uses CPU)
- each has its own processor registers, program counter, stack and status, but shares the data area and resources allocated to its process.
The same operations are performed on both traditional processes and threads.

The OS must be able to support:

- Creating new threads
- Setting up a thread so it is ready to execute
- Delaying, or putting to sleep, threads for a specific amount of time
- Blocking, or suspending, threads that are waiting for IO to complete
- Setting threads to wait state until specific event
- Scheduling threads for execution
- Synchronising thread execution using semaphores, events, or conditional variables
- Terminating a thread and releasing its resources

This is done by the OS tracking critical information for each thread.
Thread Control Block

- Just as processes are represented by Process Control Blocks (PCBs), threads are represented by **Thread Control Blocks (TCBs)**:
  - Thread ID: unique identifier assigned by OS
  - Thread State: changes as the thread progresses through execution
  - CPU information: how far the thread has executed, which instruction is being performed, what data is being used
  - Thread Priority: used by Thread Scheduler to determine which thread should be selected for the ready queue
  - Pointer: to the process that created the thread
  - Pointers: to other threads created by this thread