Operating Systems
Lecture #9: Concurrent Processes

Written by David Goodwin
based on the lecture series of Dr. Dayou Li
and the book *Understanding Operating Systems* 4^{th} *ed.*
by *I.M.Flynn and A.McIver McHoes* (2006)

Department of Computer Science and Technology,
University of Bedfordshire.

Operating Systems, 2013

15^{th} April 2013

Outline

David Goodwin
University of
Bedfordshire

Introduction

What is parallel processing?

- **Parallel processing** (also called **multiprocessing**)
    - situation in which two or more procesors 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

David Goodwin
University of Bedfordshire

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?*

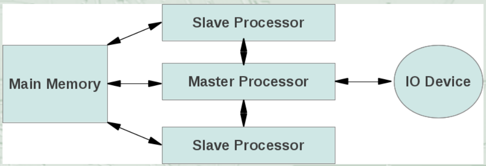
David Goodwin
University of Bedfordshire

Configurations

Master/Slave configuration

David Goodwin
University of Bedfordshire

- 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

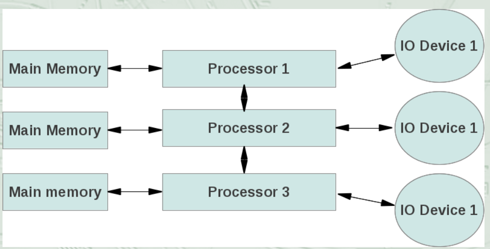


Master/Slave configuration

- Advantage:
  - Simplicity
- Disadvantage:
  - Reliability no higher than for single processor (if master fails the whole system fails)
  - Poor use of resources (if matser 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 requiremenyts and policies (new jobs may be assigned to the processor with lightest load)



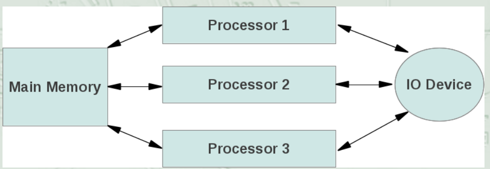
Loosely Coupled configuration

- 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

David Goodwin
University of Bedfordshire

Symmetric configuration

- Symmetric configuration best implimented if processors are the same type
    - Processor scheduling is decentralised
    - Single copy of OS and a globqal 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

David Goodwin
University of
Bedfordshire

Programming

Concurrent Programming Applications

David Goodwin
University of Bedfordshire

- 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

Concurrent Programming Applications

- Example:
  - $A = 3 * B * C + 4/(D + E) * *(F - G)$

step	Operation	Result
1	$(F - G)$	Store difference in T_1
2	$(D + E)$	Store sum in T_2
3	$(T_1) * *(T_2)$	Store power in T_1
4	$4/(T_1)$	Store quotient in T_2
5	$3 * B$	Store product in T_1
6	$(T_1) * C$	Store product in T_1
7	$(T_1) + (T_2)$	Store sum in A

Concurrent Programming Applications

- Arithmetic expressions can be processed differently if we use a language that allows concurrent processing
  - Define COBEGIN and COEND to indicate to the compiler which instructions can be processed concurrently

- COBEGIN
  $T1 = 3 * B$
  $T2 = D + E$
  $T3 = F - G$
  COEND
- COBEGIN
  $T4 = T1 * C$
  $T5 = T2 * *T3$
  COEND
- $A = T4 + 4/T5$

step	proc.	Operation	Result
1	1	$3 * B$	Store difference in T_1
	2	$(D + E)$	Store sum in T_2
	3	$(F - G)$	Store difference in T_3
2	1	$(T_1) * C$	Store product in T_4
	2	$(T_2) * *(T_3)$	Store power in T_5
3	1	$4/(T_5)$	Store quotient in T_1
4	1	$(T_4) + (T_1)$	Store sum in A

Concurrent Programming Applications

David Goodwin
University of Bedfordshire

- Increased computational speed
  - increased complexity of programming language
  - increased complexity of hardware (machinary 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)

David Goodwin
University of Bedfordshire

Case 2: Matrix Multiplication

- To perform $C = A * B$ where A and B represent two matricies:
  - $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
  - 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 (5×9)
- With three processors the answer takes only 27 steps, multiplying in parallel (3×9)

Threads

Threads & Concurrent Programming

- 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, impliment 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 staus, but shares the data area and resources allocated to its process

Threads States

- 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 Contol Blocks (PCBs), threads are represented by **Thread Contol Blocks (TCBs)**:
  - Thread ID: unique identifier assigned by OS
  - Thread State: changes as the thread progresses though execution
  - CPU information: how far the thread has executed, which instruction is being performed, what data is begin 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