Advanced Operating System

Unit – I

Dr.M.Lalli

Dept of CS, Trichy

Motivation for Multiprocessors

- **Enhanced Performance** -
  - Concurrent execution of tasks for increased throughput (between processes)
  - Exploit Concurrency in Tasks (Parallelism within process)
- **Fault Tolerance** -
  - graceful degradation in face of failures

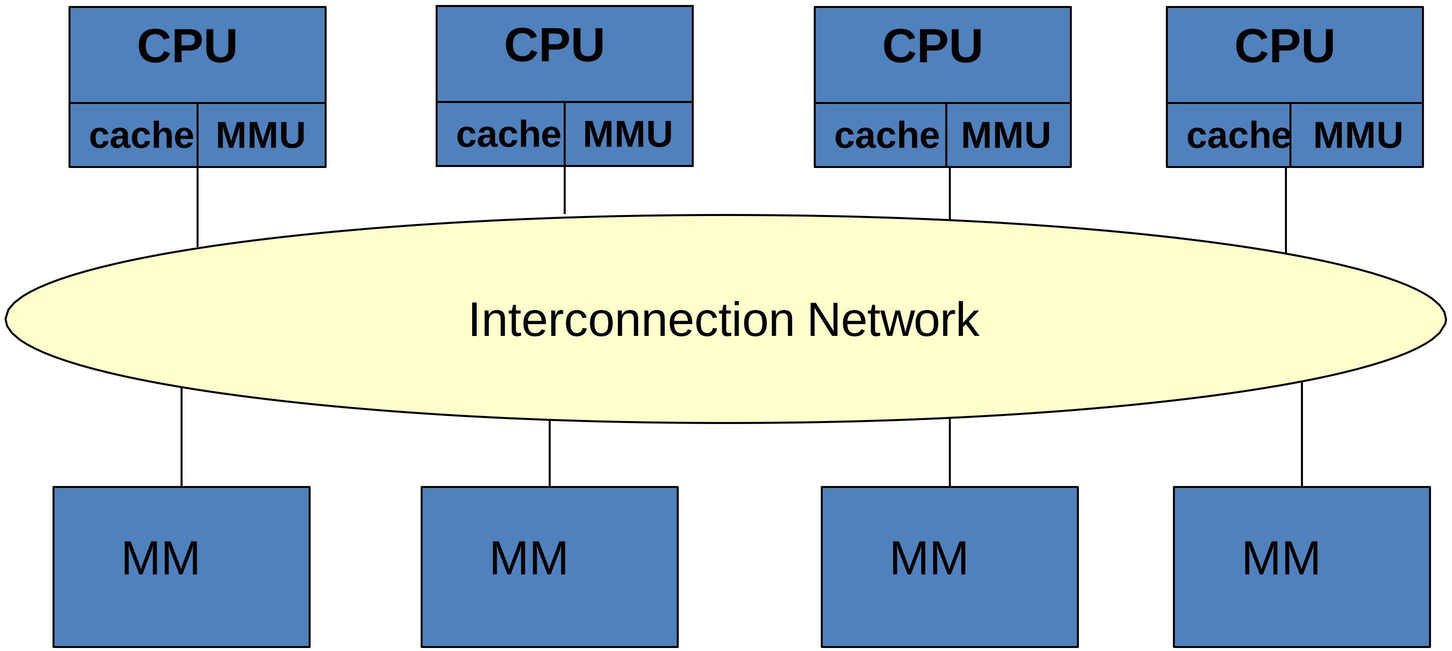
Basic MP Architectures

- Single Instruction Single Data (**SISD**) - conventional uniprocessor designs.
- Single Instruction Multiple Data (**SIMD**) - Vector and Array Processors
- Multiple Instruction Single Data (**MISD**) - Not Implemented.
- Multiple Instruction Multiple Data (**MIMD**) - conventional MP designs

MIMD Classifications

- **Tightly Coupled System** - all processors share the same global memory and have the same address spaces (***Typical SMP system***).
  - Main memory for IPC and Synchronization.
- **Loosely Coupled System** - memory is partitioned and attached to each processor. Hypercube, Clusters (Multi-Computer).
  - Message passing for IPC and synchronization.

MP Block Diagram



Memory Access Schemes

- Uniform Memory Access (UMA)
  - Centrally located
  - All processors are equidistant (access times)
- NonUniform Access (NUMA)
  - physically partitioned but accessible by all
  - processors have the same address space
- NO Remote Memory Access (NORMA)
  - physically partitioned, not accessible by all
  - processors have own address space

Other Details of MP

- Interconnection technology
  - Bus
  - Cross-Bar switch
  - Multistage Interconnect Network
- Caching - *Cache Coherence Problem!*
  - Write-update
  - Write-invalidate
  - bus snooping

MP OS Structure - 1

- Separate Supervisor -
  - all processors have their own copy of the kernel.
  - Some share data for interaction
  - dedicated I/O devices and file systems
  - good fault tolerance
  - bad for concurrency

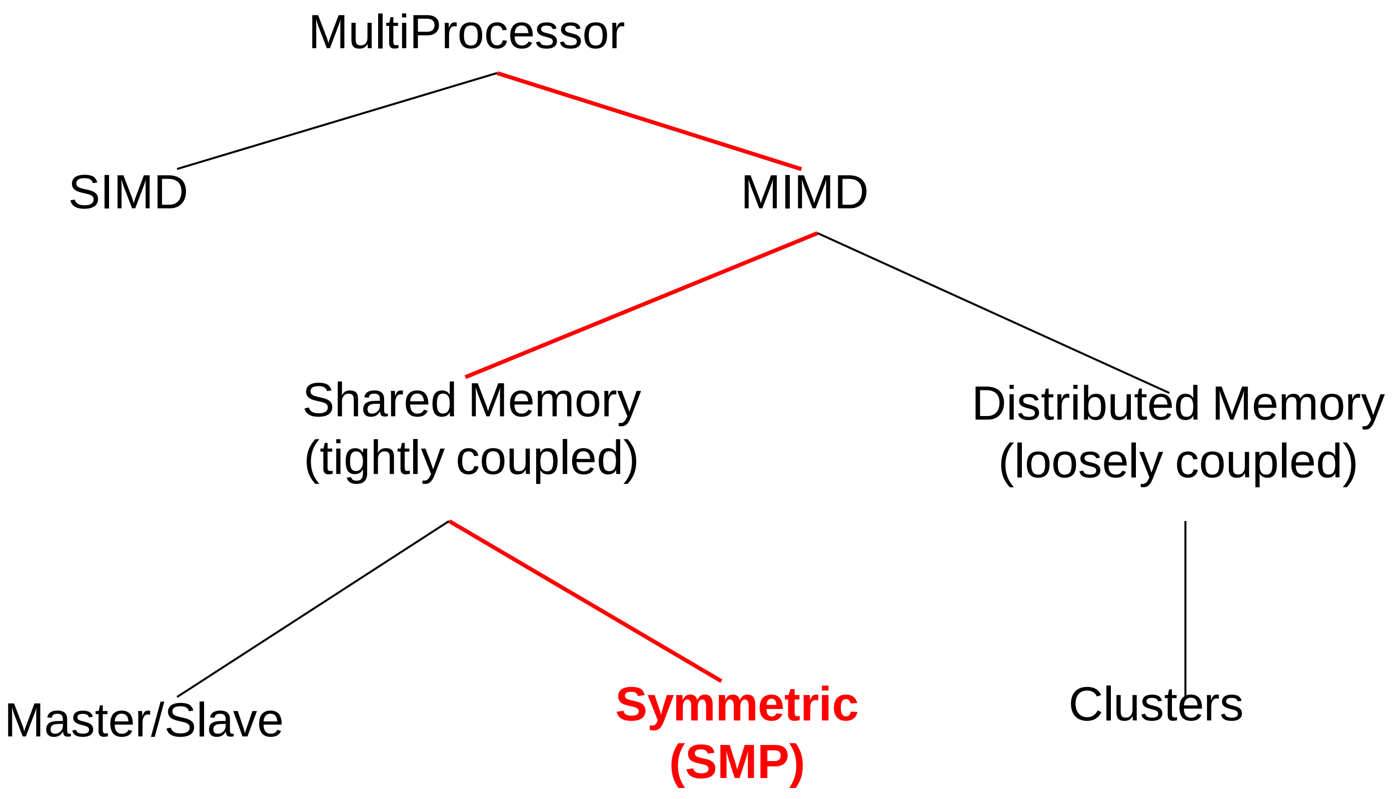
MP OS Structure - 2

- Master/Slave Configuration
  - master monitors the status and assigns work to other processors (slaves)
  - Slaves are a schedulable pool of resources for the master
  - master can be bottleneck
  - poor fault tolerance

MP OS Structure - 3

- Symmetric Configuration - Most Flexible.
  - all processors are autonomous, treated equal
  - one copy of the kernel executed concurrently across all processors
  - *Synchronize access to shared data structures:*
    - Lock entire OS - *Floating Master*
    - Mitigated by dividing OS into segments that normally have little interaction
    - multithread kernel and control access to resources (continuum)

MP Overview


MultiProcessor
SIMD
MIMD
Shared Memory
(tightly coupled)
Distributed Memory
(loosely coupled)
Master/Slave
Symmetric
(SMP)
Clusters

SMP OS Design Issues

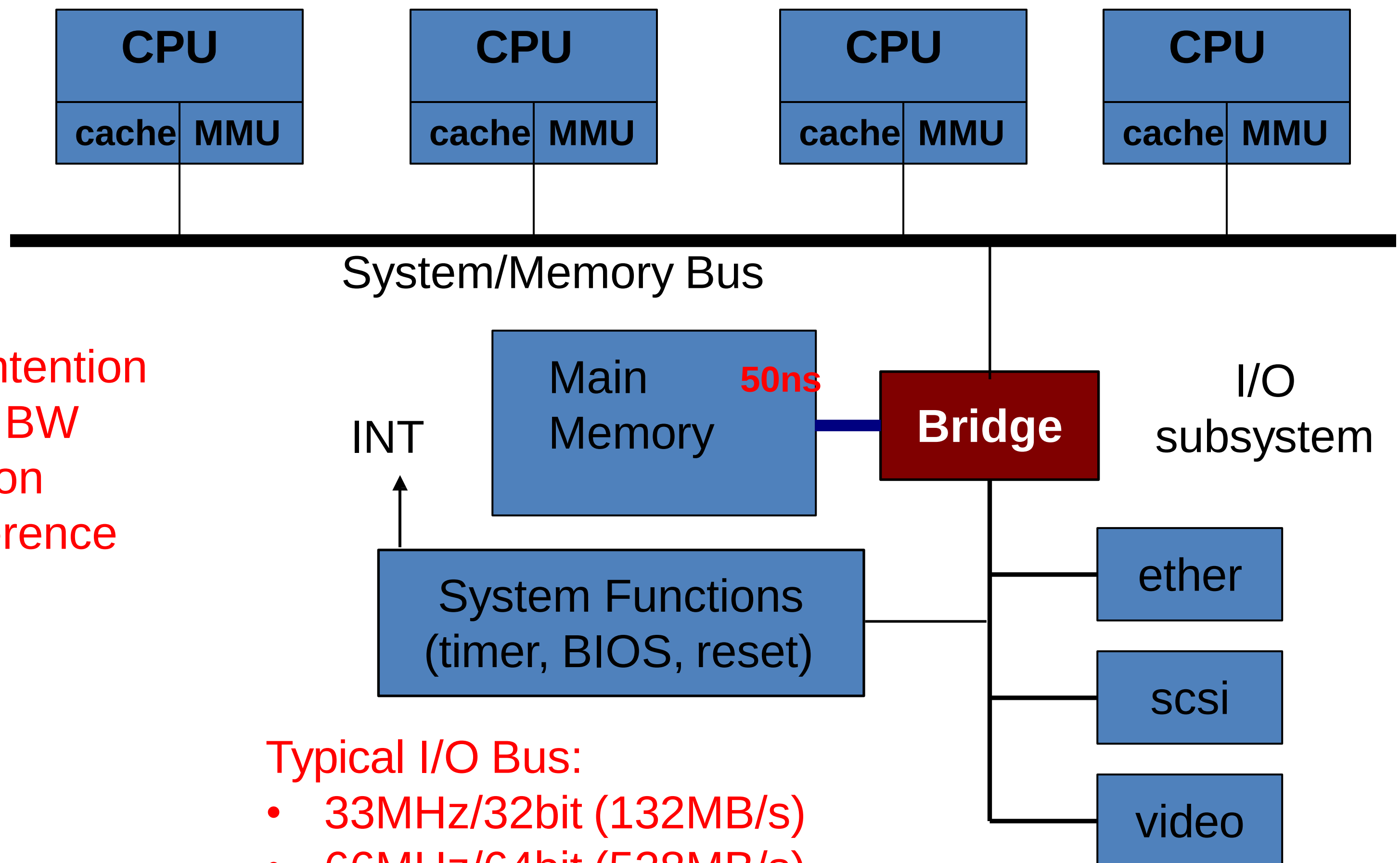
- **Threads** - effectiveness of parallelism depends on performance of primitives used to express and control concurrency.
- **Process Synchronization** - disabling interrupts is ***not*** sufficient.
- **Process Scheduling** - efficient, policy controlled, task scheduling (process/threads)
  - global versus per CPU scheduling
  - Task affinity for a particular CPU
  - resource accounting and intra-task thread dependencies

SMP OS design issues - 2

- **Memory Management** - complicated since main memory is shared by possibly many processors. Each processor must maintain its own map tables for each process
  - cache coherence
  - memory access synchronization
  - balancing overhead with increased concurrency
- **Reliability and fault Tolerance** - degrade gracefully in the event of failures

Typical SMP System

500MHz



Issues:

- Memory contention
- Limited bus BW
- I/O contention
- Cache coherence

Typical I/O Bus:

- 33MHz/32bit (132MB/s)
- 66MHz/64bit (528MB/s)

Some Definitions

- ***Parallelism***: degree to which a multiprocessor application achieves parallel execution
- ***Concurrency***: Maximum parallelism an application can achieve with unlimited processors
- ***System Concurrency***: kernel recognizes multiple threads of control in a program
- ***User Concurrency***: User space threads (***coroutines***) provide a natural programming model for concurrent applications. Concurrency not supported by system.

Process and Threads

- **Process**: encompasses
  - set of threads (computational entities)
  - collection of resources
- **Thread**: Dynamic object representing an execution path and computational state.
  - threads have their own computational state: PC, stack, user registers and private data
  - Remaining resources are shared amongst threads in a process

Process Synchronization:Motivation

- Sequential execution runs correctly but concurrent execution (of the same program) runs incorrectly.
  - Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Let's look at an example: consumer-producer problem.

Producer-Consumer Problem

Producer

```
while (true) {
  /* produce an item and put in nextProduced */
  while (count == BUFFER_SIZE); // do nothing
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
}
```

count: the number of items in the buffer (initialized to 0)

Consumer

```
while (true) {
  while (count == 0); // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
   // consume the item in nextConsumed
}
```

What can go wrong in concurrent execution?

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
    S1: producer execute register1 = register1 + 1 {register1 = 6}
    S2: consumer execute register2 = count {register2 = 5}
    S3: consumer execute register2 = register2 - 1 {register2 = 4}
    S4: producer execute count = register1 {count = 6 }
    S5: consumer execute count = register2 {count = 4}

What are all possible values from concurrent execution?

How to prevent race condition?

- Define a <u>critical section</u> in each process
  - Reading and writing common variables.
- Make sure that only one process can execute in the critical section at a time.
- What sync code to put into the entry & exit sections to prevent race condition?

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

What is the difference between

Progress and Bounded Waiting?

Peterson's Solution

- Simple 2-process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Processor Scheduling

- PS: ready tasks are assigned to the processors so that performance is maximized.
- Cooperate and communicate through shared variables or message passing, PS in multiprocessor system is difficult problem.
- PS is very critical to the performance of multiprocessor systems because a naïve scheduler can degrade performance substantially.

Issues in Processor Scheduling

- 3 major causes of performance degradation are
  - Preemption inside spinlock-controlled critical sections.
    - This situation occurs when a task is preempted inside CS when there are other tasks spinning the lock to enter the same CS.
  - cache corruption
    - Big chunk of data needed by the previous tasks must be purged from the cache and new data must be brought into the cache.
    - Very high miss ratio a processor switched to another task – Cache corrp.
  - context switching overheads
    - Execution of a large no. of instructions to save and store the registers, to initialize the registers, to switch address space, etc.

Distributed Shared Memory in Mach

- The idea is to have a single, linear, virtual address space that is shared among processes running on computers that do not have any physical shared memory. When a thread references a page that it does not have, it causes a page fault. Eventually, the page is located and shipped to the faulting machine, where it is installed so that the thread can continue executing.

Communication in Mach

- The basis of all communication in Mach is a kernel data structure called a port.
- When a thread in one process wants to communicate with a thread in another process, the sending thread writes the message to the port and the receiving thread takes it out.
- Each port is protected to ensure that only authorized processes can send it and receive from it.
- Ports support unidirectional communication. A port that can be used to send a request from a client to a server cannot also be used to send the reply back from the server to the client. A second port is needed for the reply.

Thank U