- CS250P Computer System Architecture Part II
  - Operating System
    - Old vs Mordern
      - Old OS
      - Mordern OS
  - Architectural Support for OS
    - Privilege Levels
      - RISC-V
      - x86
    - Exceptions and Interrupts
      - Exceptions
      - Interrupts
      - Handling
      - Utilization of Exception
    - Context Switching
      - Stack and Stack Pointer
      - PCB - Process Control Block
    - System Boot Process
    - Virtual Memory
      - Goal
      - Physical Address
      - Virtual Address
      - Segmentation - Old School Solution
        - Basics
        - Swapping
        - Problem - External Fragmentation
      - Paged Virtual Memory
        - Basics
        - Page Table
        - Page Table Entry
        - Pros & Cons
          - Pros
          - Cons
        - Implementation
        - Translation Lookaside Buffer - TLB
          - Basics
          - Virtual Memory Access with TLB
          - Pros with TLB
        - Size of Page
          - Large Page Size
          - Small Page Size
          - Solution

# CS250P Computer System Architecture Part II

---

## Operating System

### Old vs Mordern

Old OS

Old personal operating systems like MS-DOS are basic.

1. OS and user software run together.
2. Software has all access to hardware.
3. Only one software runs at a time.
4. Software failure will lead to system crash.

Mordern OS

1. User Process Isolation
2. OS kernal provides a private address space to each process
3. OS kernal schedules processes into CPU
4. OS kernal let processes invoke system services via system calls

> Process's POV of mordern OS:
>
> 1. Has exclusive access to a contiguous address space
> 2. Can not access memory spaces of other processes or OS
> 3. Only run on CPU for allowed CPU time

## Architectural Support for OS

What ISA needs to provide for mordern OS to function properly.

### Privilege Levels

RISC-V

The implementation of OS privilege levels needs ISA support. Special register "mstatus" and dedicated read & write instrucitons.

1. Machine Level - full access
2. Hypervisor
3. Supervisor
4. User Level

x86

Protection Ring, inner ring kernal level, outer ring less privileged

## Exceptions and Interrupts

Exceptions and interrupts are events that needs to be processed by the OS kernal (in higher privilege than current process). The term exception and interrputs are used interchangeably.

### Exceptions

Events caused by the running process itself.

1. Segmentation fault, illegal memory access
2. Divide by zero
3. System call
4. etc

### Interrupts

Events caused by the outside world.

1. I/O
2. Keystroke
3. Timer
4. etc

### Handling

Handling of exceptions is a purely hardware process, the handling method of each exception is **hardwired** and predefined.
RISC - mtvec - Machine Trap Vector
x86 - IDTR - Interrupt Descriptor Table Register

1. Stop current process at instruction $I_i$, complete all instruction $I_{i-1}$
2. Save pc for instruction $I_i$, save interrupt number in register
3. Enter privilege mode, disable interrupts, transfer control to **predefined** exception handler PC
4. Return control to user process instruction $I_i$

### Utilization of Exception

The interrupts ISA provided can be utilized to achieve bunch of things.

1. CPU Scheduling
   Each process is given a fraction of CPU time. Kernal set timer, raise interrupt after timer runs out. Save process 1, set new timer, load process 2, return control to process 2.

2. Emulating Instructions
   ISA can emulate unsupported instructions with interrupts. For example mul instruction from RISC-V M extension can run on RISC-V. Exception handler is invoked when mul is executed, emulate mul with repeated add in handling function.

3. System Calls
   User process can use system call to request access of system resources.

   - File Access
   - Network Connection
   - Manage Memory
   - Manage Process
   - etc

## Context Switching

**OS SOFTWARE SUPPORT**

On a multitasked system, process execute in small increments on processor. Context switching is required for switching the execution of different processes. Processes are still sharing the same address memory space.

**Context**: the state information of process. All the info need to restore execution of process.
**Context Switching**: Storing context of current process, load context of next process.

### Stack and Stack Pointer

In the interrupt handler, store all register value of the process in the stack.
How to find the stored value after exiting the handler? Store the register address in PCB.

### PCB - Process Control Block

PCB is used to manage context information in the OS. Stores process ID, priority, registers, program counters, process states etc. Each process has its PCB.

## System Boot Process

1. When power on, start executing from address 0.
2. Firmware is located at address 0, which would load bootoader into special address then hand over control.
3. Bootloader loads the actual OS kernal from storage to memory and transfer control.

## Virtual Memory

This topic itself can be a whole section.

### Goal

1. Protection and privacy: processes can not access each other's data (memory space).
2. Abstract memory resource: Provide private address space to processes. From processes' POV, they each have exclusive access to large, uniform address space.

### Physical Address

Actual address on the physical DRAM. Managed & visible to

## Virtual Address

Virtual address generated by a process, points to its private address space. The translation between physical and virtual address is done by memory management unit (MMU hardware in CPU).

## Segmentation - Old School Solution

### Basics

Memory segmentations of **various sizes** are allocated to processes. The segmentation size depends on requirements of different processes. The OS maintains the base addresses and sizes of the segmentations. OS check segmentation bounds when accessing to avoid reading other processes' memory space.

$$\text{Physical Address} = \text{Virtual Address} + \text{Base Address}$$

### Swapping

Entire segmentations are swapped between main memory and main storage to make space for other processes.

### Problem - External Fragmentation

Variable sized sementation efficiently utilizes the memory space within the segmentation. But since the segmentation can't be split up when allocated or swapped in, unused chunks of memory fragments will appear main memory.
Fragementation is hard to deal with and solutions can be costly.

## Paged Virtual Memory

### Basics

Physical memory is divided into **fixed-size** pages.

> The size usually 4KiB, 4096 bytes

For the sake of translating virtual address to physical address, virtual memory address is interpreted as

$$| \text{virtual page number} | \text{page internal offset} |$$

### Page Table

OS uses a page table to translate (page table walk, handled by MMU) virtual page number to physical page number, then add page internal offset to get actual memory address.
**OS** maintains a page table for **each** process.
Physical address space may be larger than virtual address space.

### Page Table Entry

PTE is a single page line, page table is an array of page table entry.
PTE holds the mapping between a single virtual page address and phyical page address.
PTE also holds flags of each page, indicating page properties like read-only.

### Pros & Cons

#### Pros

Pages can be stored non-contiguously thanks to address lookup provided by page table. This greatly reduces external fragmentation.

#### Cons

The storage and maintenance cost of page table is larger than segmentation (32bit base + 32bit size bound for each process). Page table is stored in DRAM.

#### Implementation

Best case two main memory access is required for each virtual memory access.

$$\text{Physical Page Number} = \text{Mem}[\text{Page Table Base} + \text{Virtual Page Numer}]$$

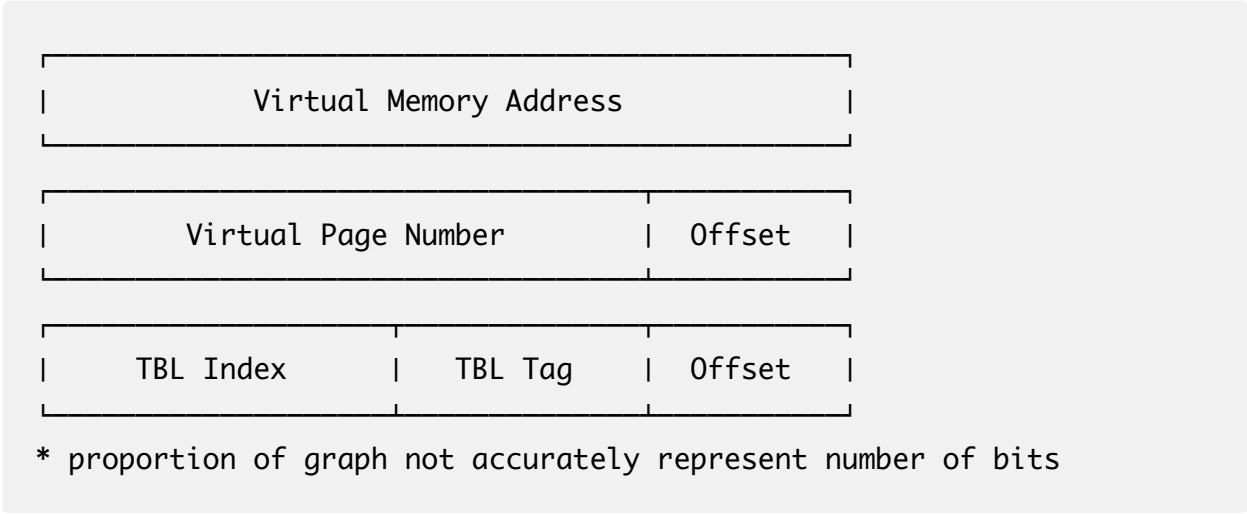
$$\text{Physical Address} = \text{Physical Page Number} + \text{Offset}$$

#### Translation Lookaside Buffer - TLB

One of the eight great ideas "Make the Common Case Fast". As we analysed in the page table implementation, there is a large speed overhead as two main memory accesses are executed for each virtual memory access.

#### Basics

Cache some virtual-to-physical address translations (just like how data cache works) to reduce memory access.
Index by subset of virtual page number.
Tagged by remainder of virtual page numer.
Such mapping scheme has associativity issues.




#### Virtual Memory Access with TLB

1. TLB Hit - immediate address translation
2. TLB Miss - walk entire page table, load page into TLB for future use

#### Pros with TLB

- Low miss rate result from benchmark, except for random access like graph
- Exploits the locality, TLB target is 4KiB page size, which is larger than cache target cache lines.

#### Size of Page

Why 4KiB page size?
4GiB physical address space, 32 bit address
4KiB page size
number of pages (entries) = 4GiB/4KiB = $2^{32} \div 2^{12} = 2^{20}$
page table size = #entries $\times$ entry size
page table size = $2^{20} \times 4\text{bytes} = 2^{22}\text{byte} = 4\text{MiB}$, assume 4 byte page table entry

> There is one full page table for each process. If there are 256 processes in the OS, the total size of page tables would be $4 \times 265 = 1024\text{MiB} = 1\text{GiB}$.

Large Page Size

Smaller number of entries, reduces page table size. Underutilized page internal memory will cause internal fragmentation.

Small Page Size

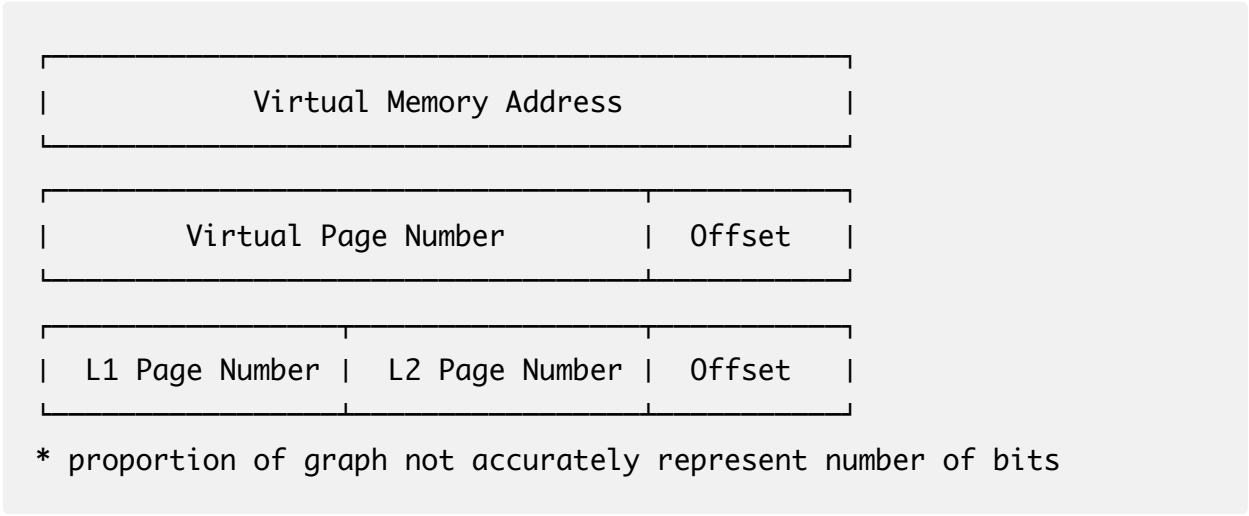
Reduce internal fragmentation. Number of entries will increase, result in large page table size and extra memory cost for maintaining the page table.

Solution

Hierarchical page tables.

Hierarchical Page Table

Most process doesn't use all the virtual memory available, which means most page table entries is not used.
Split page table number into L1 and L2 page number. We can reduce the size of page table by only creating corresponding L2 page tables when L1 page table entry is accessed.
Downside: More memory access if TLB misses.




## Paged Virtual Memory and Multiprocessing

Context Switching with Paged Virtual Memory

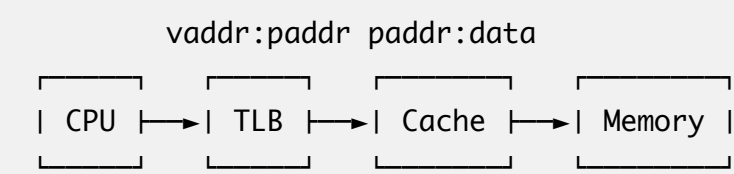
Save the start address of page table (page table base register) as part of the process context.

Caching with Paged Virtual Memory

Cache Based on Physical Address

PIPT, Physically Indexed Physically Tagged Cache. If we use physical address for cache entries, we can basically reuse the old cache scheme. However this solution is slow.

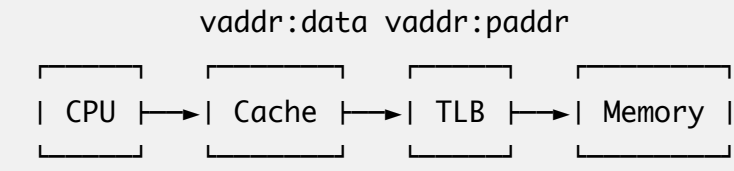
1. Every Cache access requires address translation.
2. If TLB misses, table walk time will overshadow cache benefits.




Cache Based on Virtual Address

VIVT, Virtually Indexed Virtually Tagged Cache. Fast but hard to implements.

1. If Cache hit, immediately get data.
2. If Cache miss, normal TLB process.




Problem:

1. Multiple processes can have the same virtual address mapping to different physical addresses. Same virtual address, different physical address.
2. Shared physical memory may be cached twice. Different virtual address, same physical address.

VIVP Problem Solutions

Naive Solution TLB Flush

Flush the TLB after every context switching. Ensure TLB only has the translation of the virtual address of current running process. Compulsory miss after context switch which sacrifices performance.

Address Space ID

Augment the virtual address with process-specific Address Space ID(ASID), to distinguish virtual addresses of different processes. However the number of processes are now limited by ASID bits. This solution requires hardware and OS support.

Mixed Physical and Virtual Caching

Virtually Indexed Physically Taged (VIPT Cache). Use subset bits of page offset as Cache index. Lookup virtual index in TBL, loopup page offset in Cache, done in parallel. Only works when Cache index smaller than page offset.

Synonym Solution

Problem 2. Update to the physical address from one process may not appear for the other processes sharing this physical address.
Cache needs to ensure the same physical address is not cached twice.

## Demand Paging

Pages can reside in either memory or disk. Add "resident" flag and "dirty" flag in page table.
Page offset can now mean offset in memory or disk.
DRAM now become a full associative cache to disk.

### Writethrough vs Writeback

#### Writethrough

Write to corresponding page in disk whenever there is a write. Disk bandwidth will bottleneck perfomance of damand paging.

#### Writeback

Write to disk only if the on memory page is evicted. Use "dirty" flag to indicate change of page content, which needs a writeback when page is evicted.

### Page Eviction Policy

When we run out of memory, we need to evict some pages to make room for the new pages.
pseudo-LRU

### Handling of the Swap

OS handles the scheduling of pages. MMU checks "resident" flag to see if page is on disk. If so raise exception(needs new hardware support) and let OS handler copy page from disk to memory.

## Copy-on-write Optimization

When fork() is call on a process, make copy of entire process memory, for the new process to use. Some memory is not immediately used, some is never used.

### Optimization

New process copy only the page table. Mark PTE with Read-Only for both new and old process.
Write attempts from both processes arise access faults, only then make new copies of page in the handler.

## On-demand Storage Access

### Example 1

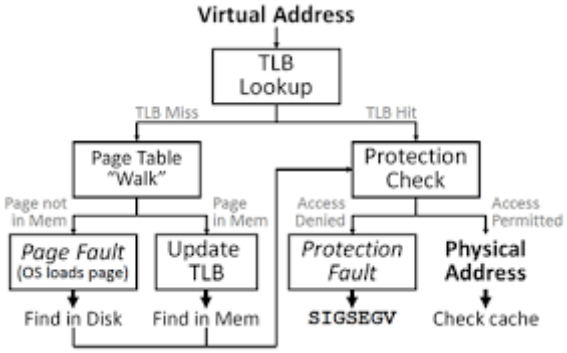
When executing a program, not all parts of the program may be used.
Create page table for the program, and mark most of them non-resident (not in memory).
When access attempts raise page faults, only then load page from storage.

### Example 2

mmap() returns a pointer to the storage, where page is mapped to.
Mark PTE as non-resident.
Read/write to the pointer raises exceptions, which OS handles transparently.

## Summary

The overall picture of paged virtual memory with TLB and demand paging.




# Multiprocessing

## Hardware for Parallelism

1. SISD - Single Instruction Sinlge Data - Single Core Processor
2. SIMD - Single Instruction Multiple Data - GPU, AVX
3. MISD - Multiple Instruction Single Data - Systolic Arrays
4. MIMD - Multiple Instruction Multiple Data - Parallel Processors

## Shared Memory Multiprocessor

Is a Symmetric MultiProcessor(SMP) architecture, the identical processors have their own caches, but caches accross all processors eventually conects to a single physical address space.

Uniform vs Non-uniform

UMA - Uniform Memory Access: Indentical processors have equal access time to memory
NUMA - Non-Uniform Memory Access: Link some of all SMPs, and allow one SMP to directly access memory of other SMPs.
UPI - Ultra Path Interconnect - The link between symetric multiprocessors and memory/shared cache for NUMA.

## Problems

Cache Coherency - The Two CPUs Example

The two CPUs share the same physical address, the data of the address is cached in both $\mathrm{CPU}_A$'s L1 and $\mathrm{CPU}_B$'s L1.
When $\mathrm{CPU}_A$ writes to that address, either with write-though or write-back from cache to memory, the data in $\mathrm{CPU}_A$'s L1 and memory is updated.
However the data cached in $\mathrm{CPU}_B$'s L1 become incorrect.

Memory Consistency - The Two Processes Example

Variable A, B are initially 0.

```
Processor 1:          Processor 2:
1: A = 1;             3: B = 1;
2: print B            4: print A
```

Possible execution outcome:

- 01 if order 1234, 3412
- 11 if order 1324, 3124, 1342, 3142

Impossible outcome:

- 10
- 00

Memory Model defines possible outcome.

## Cache Coherency - Enforced

Read to **each address must** return the most recent value. Which requires writes from each processor must be visible at some point in correct order.

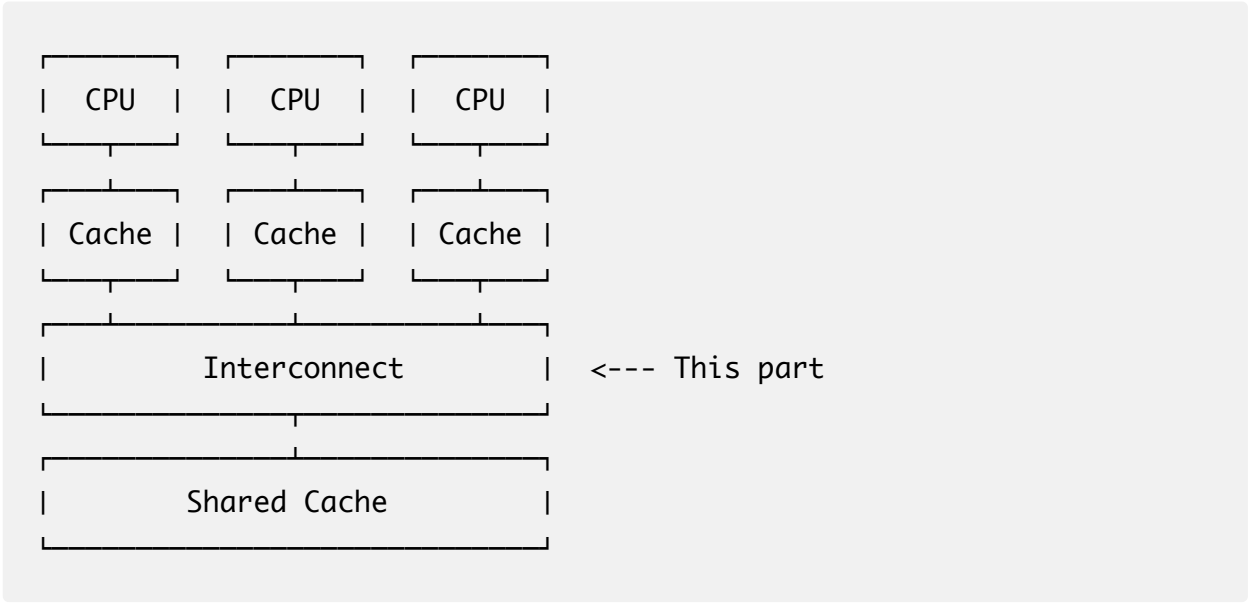
### Problem

All SMP cores have their own cached copies of a shared memory location. Copies of other cores become stale when one core only writes to its own cache. We need to propagate (broadcast) the update to other cores.

### Backgrounds for Possible Solutions

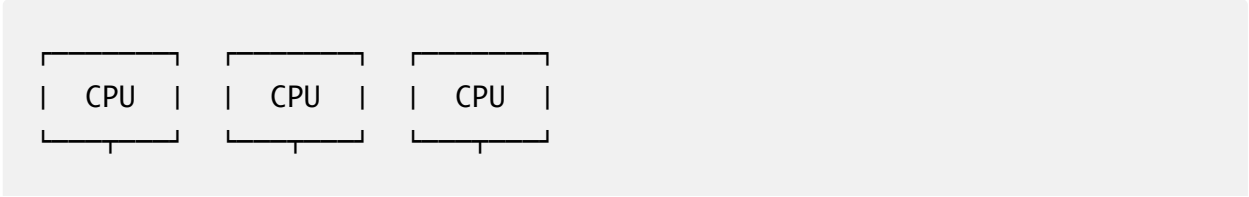
#### On-chip Interconnect

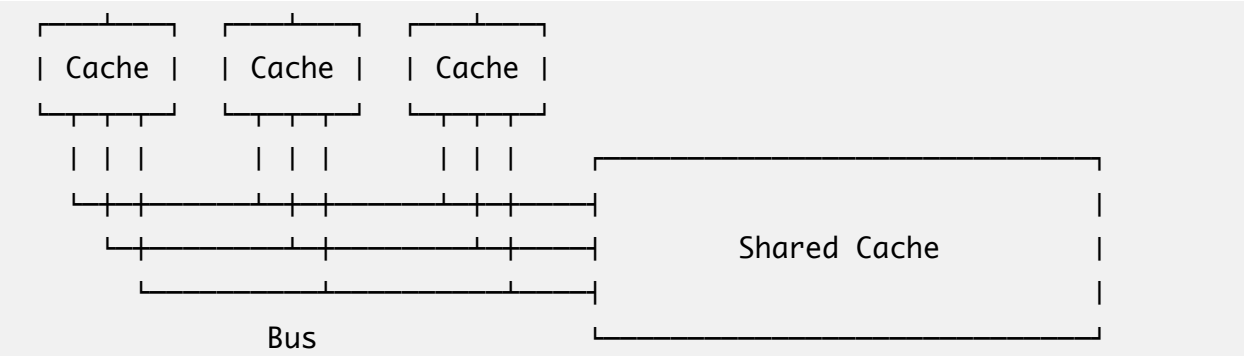
An interconnect fabric (builtin wires of the circuit) connect all cores & their private caches to shared cache levels and main memory.




#### Bus Interconnect

Shared bundle of wires, all data transfers are broadcasted, all entities on the bus can listen to all communications.

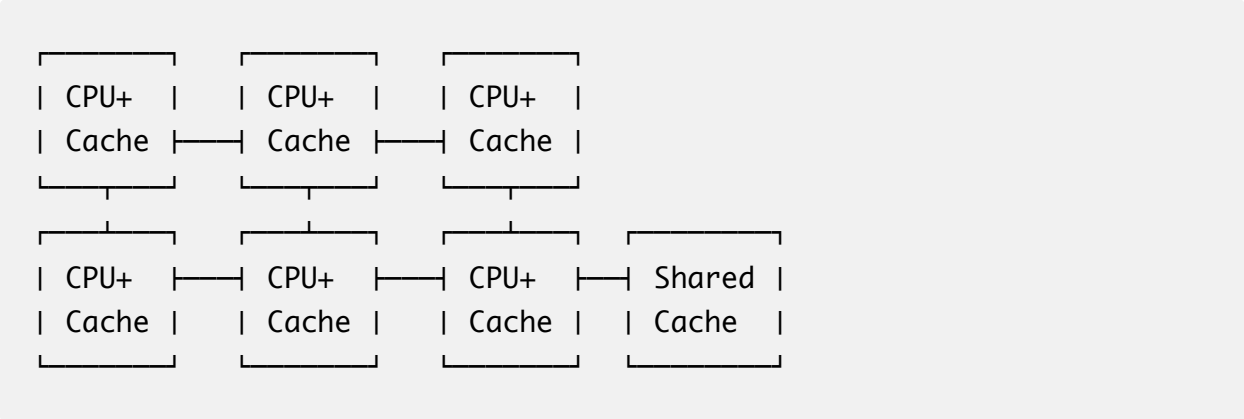





Pros: All communication is immediate single cycle
Cons: Only one entity can transmit data on but in any given cycle, use multi-master to schedule the use of bus

Mesh Interconnect

Interconnect between CPUs, each core acts as a network switch. One cycle to adjacent nodes, multiple cycle with more than one hops.




Pros: Higher bandwidth than bus interconnect, scalability with more cores.
Cons: The latency is variable; Needs more silicon resources to implement.

Solution

Basic idea of cache coherency is:
If cache line is only read, multiple cores can have cache copies of the line.
If cache line is written to, only one cache at a time can have a copy.

Snooping Based

All cores listen (snooping) to requests on shared cache/memory bus. - Cores decide if update cache or not.
Core must broadcast intention before writing into a cache line. Other cores invalidate their cache copies.
MIS:
MIS is one of the algorithms that can make this process efficient.
Mark each cache line with of of the three states:

1. Modified(M): Dirty line, only one copy of this line can exist across cores.
2. Shared(S): Unmodified read-only line. Multiple copies can exist.

3. Invalid(I): Cache line no longer valid.

Problem:
Multiple cores trying to write to the same address will cause ping pong of intention broadcasting.
Only one core can broadcast per cycle, low bandwidth will bottlenect the scalability.

#### Directory Based

All cores consult a separated entity call "Directory" for cache access. - Directory controls access request.
Set a directory entity per core, which manages caches mapped to its own subset.
Add P bit per cache line for P processors, indicates which processor has a copy of the cache line. One additional dirty bit.

#### Synchronization Hardware Support

In parallel software critical sections implemented with lock are essentially for algorithm correctness.

```
Thread 1                  Thread 2
while (lock==False);      while (lock==False);
lock = True;              lock = True;
// critical section       // critical section
lock = False;             lock = False;
```

If thread 2 checked lock before thread 1 set lock to True, both threads will enter critical section.

#### Software Solution - Algorithm

Dekker's algorithm, Lamport's backery algorithm.

#### Hardware Solution - Atomic Instruction

Memory read/write in and single instruction. No other instructions can read/write between the atomic instruction. RISC-V needs an 'A' for atomic extention to support such instructions.

## Memory Consistency - Defined by Architect

**How** updates to **different addresses** become visible to other processors. The two processes example.

#### Consistency Models

Top: more strict, easier to write correct programs
Buttom: exploits more instruction-level parallelism (ILP), higher performance

1. Sequential Consistency
2. Total Store Order
3. Casual Consistency
4. Processor Consistency
5. Release Consistency
6. Eventual Consistency

Sequential Consistency

Results in the memory are the same as sequential order between operations.
Effects of previous instruction is visible to every predecessing instructions in every processor.
Problem:
Write to shared memory must propagate to all other caches/cores (all the way to L1), through TLB and cache.
Archtects then think they can afford to sacrifice consistency for performance.

Total Store Order

Add a small store buffer between each core and its cache.
Writes from core stored in buffer, apply the writes to cache later in order. Write instructions **can not block** following instructions.
Reads first scan buffer then go to L1.
Problem:
Performance improved, single thread is correct by multi-thread is now trippy.
Solution:
Fence instruction, enforce all state changes before it to propagate, restores sequential consistency.

> Write instructions need not block the system, this means 00 and 10 now became possible (acceptable) outcomes for the two processes example, as instruction 3, 4 can now execute before 1, 2.

Amadhl's Law

Speedup of parallelism:

$$\frac{1}{(1-p)+\frac{p}{s}}$$

p portion of parallelized execution time
s speedup of parallelized protion

# GPU

## GPU Architecture

GPUs have thousands of threads running concurrently (SIMD fashion) in GHz. The single thread performance of GPUs are bad, the performance of GPUs comes from the massive parallelism.

GPU Memory Architecture

Not much on-chip memory per thread.
Relatively fast off-chip "Global" memory.
Almost no memory consistency between blocks.

## CUDA

Computer Uniform Device Architecture, run custom programs on massively parallel architecture.

Block

Block is multi-dimensional array of threads. CUDA blocks can be 1D, 2D or, 3D.
Threads in the block and synchronize among themselves and access shared memory.

Grid

Grid is multi-dimensional array of blocks. Blocks in a grid can run in parallel or sequential.

## Application Specific Accelerator

Chip specialization is one of the most prominent solutions to dark silicon.
However, it is not a long-term solution beyond Moore's law.

## Datacenter Architecture

Topics:

- Cost of a datacenter
- Power Supply
- Cooling
- Networking
- Virtualization

## FPGA

Field Programmable Gate Array.
Pros:

- FPGA can emulate any custom ciruit
- Functions can be changed completely

Cons:

- Emulated circuits are less efficient (performance, power consumption)