
EE108B
Lecture 13

Caches Wrap Up
Processes, Interrupts, and Exceptions

Christos Kozyrakis
Stanford University
<http://eeclass.stanford.edu/ee108b>

Announcements

- Lab3 and PA2.1 are due ton 2/27
- Don't forget to study the textbook
- Don't forget the review sessions

Measuring Performance

- Memory system
 - Stalls include both cache miss stalls and write buffer stalls
 - Cache access time often determines the overall system clock cycle time since it is often the slowest functional unit in the pipeline
- Memory stalls hurt processor performance
 - $\text{CPU Time} = (\text{CPU Cycles} + \text{Memory Stall Cycles}) \cdot \text{Cycle Time}$
- Memory stalls caused by both reading and writing
 - $\text{Mem Stall Cycles} = \text{Read Stall Cycles} + \text{Write Stalls Cycles}$

Memory Performance

- Read stalls are fairly easy to understand
 - $\text{Read Cycles} = \text{Reads/prog} \cdot \text{Read Miss Rate} \cdot \text{Read Miss Penalty}$
- Write stalls depend upon the write policy
 - Write-through
 - $\text{Write Stall} = (\text{Writes/Prog} \cdot \text{Write Miss Rate} \cdot \text{Write Miss Penalty}) + \text{Write Buffer Stalls}$
 - Write-back
 - $\text{Write Stall} = (\text{Writes/Prog} \cdot \text{Write Miss Rate} \cdot \text{Write Miss Penalty})$
- “Write miss penalty” can be complex:
 - Can be partially hidden if processor can continue executing
 - Can include extra time to writeback a value we are evicting

Worst-Case Simplicity

- Assume that write and read misses cause the same delay

$$\text{Memory Stall Cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$\text{Memory Stall Cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

- In a multi-level cache be careful about *local* miss rates
 - Miss rate for 2nd level cache is often “high” if you look at the local miss rate (misses per reference into the cache)
 - But for misses per instruction, it is often much better

Cache Performance Example

- Consider the following
 - Miss rate for instruction access is 5%
 - Miss rate for data access is 8%
 - Data references per instruction are 0.4
 - CPI with perfect cache is 2
 - Read and write miss penalty is 20 cycles
 - Including possible write buffer stalls
- What is the performance of this machine relative to one without misses?

Performance Solution

- Find the CPI for the base system without misses

- $CPI_{\text{no misses}} = CPI_{\text{perfect}} = 2$

- Find the CPI for the system with misses

- Misses/instr = I Cache Misses + D Cache Misses

- $= 0.05 + (0.08 \cdot 0.4) = 0.082$

- Memory Stall Cycles = Misses/instr • Miss Penalty

- $= 0.082 \cdot 20 = 1.64$

- $CPI_{\text{with misses}} = CPI_{\text{perfect}} + \text{Memory Stall Cycles}$

- $= 2 + 1.64 = 3.64$

- Compare the performance

$$n = \frac{\text{Performance}_{\text{no misses}}}{\text{Performance}_{\text{with misses}}} = \frac{CPI_{\text{with misses}}}{CPI_{\text{no misses}}} = \frac{3.64}{2} = 1.82$$

Another Cache Problem

- Given the following data
 - 1 instruction reference per instruction
 - 0.27 loads/instruction
 - 0.13 stores/instruction
 - Memory access time = 4 cycles + # words/block
 - A 64KB S.A. cache with 4 W block size has a miss rate of 1.7%
 - Base CPI of 1.5
- Suppose the cache uses a write through, write-around write strategy without a write buffer. How much faster would the machine be with a perfect write buffer?

$$\text{CPU}_{\text{time}} = \text{Instruction Count} \cdot (\text{CPI}_{\text{base}} + \text{CPI}_{\text{memory}}) \cdot \text{Clock Cycle Time}$$

Note: Red arrows point from the numbers 1, 1.5, and 1 above to the terms Instruction Count, CPI_{base}, and Clock Cycle Time respectively.

Performance is proportional to $\text{CPI} = 1.5 + \text{CPI}_{\text{memory}}$

Another Cache Problem

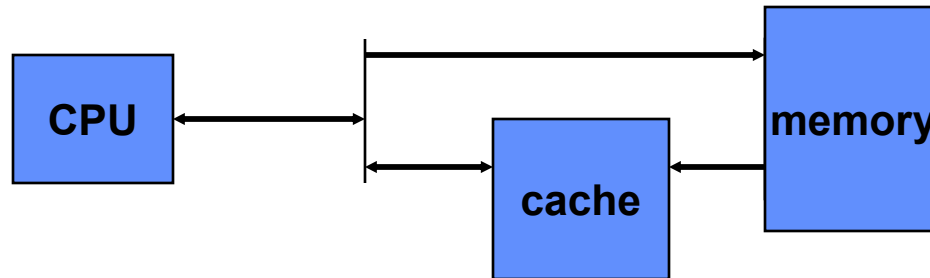
- Given the following data
 - 1 instruction reference per instruction
 - 0.27 loads/instruction
 - 0.13 stores/instruction
 - Memory access time = 4 cycles + # words/block
 - A 64KB S.A. cache with 4 W block size has a miss rate of 1.7%
 - Base CPI of 1.5
- Suppose the cache uses a write through, write-around write strategy without a write buffer. How much faster would the machine be with a perfect write buffer?

$$\text{CPU}_{\text{time}} = \text{Instruction Count} \cdot (\text{CPI}_{\text{base}} + \text{CPI}_{\text{memory}}) \cdot \text{Clock Cycle Time}$$

(Note: In the original image, red arrows point from the numbers 1, 1.5, and 1 above to the terms Instruction Count, CPI_{base}, and Clock Cycle Time respectively.)

Performance is proportional to $\text{CPI} = 1.5 + \text{CPI}_{\text{memory}}$

No Write Buffer



$$\text{CPI}_{\text{memory}} = \text{reads/instr.} \cdot \text{miss rate} \cdot \text{read miss penalty} \\ + \text{writes/instr.} \cdot \text{write penalty}$$

$$\text{read miss penalty} = 4 \text{ cycles} + 4 \text{ words} \cdot 1 \text{ word/cycle} = 8 \text{ cycles}$$

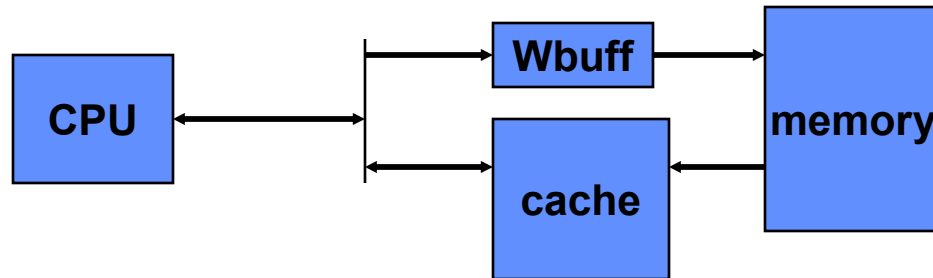
$$\text{write penalty} = 4 \text{ cycles} + 1 \text{ word} \cdot 1 \text{ word/cycle} = 5 \text{ cycles}$$

$$\text{CPI}_{\text{memory}} = (1 \text{ if} + 0.27 \text{ ld}) (1/\text{instr.}) \cdot (0.017) \cdot 8 \text{ cycles} + (0.13 \text{ st})(1/\text{instr.}) \cdot 5 \text{ cycles}$$

$$\text{CPI}_{\text{memory}} = 0.17 \text{ cycles/instr.} + 0.65 \text{ cycles/instr.} = 0.82 \text{ cycles/instr.}$$

$$\text{CPI}_{\text{overall}} = 1.5 \text{ cycles/instr.} + 0.82 \text{ cycles/instr.} = 2.32 \text{ cycles/instr.}$$

Perfect Write Buffer



$$\text{CPI}_{\text{memory}} = \text{reads/instr.} \cdot \text{miss rate} \cdot 8 \text{ cycle read miss penalty} \\ + \text{writes/instr.} \cdot (1 - \text{miss rate}) \cdot 1 \text{ cycle hit penalty}$$

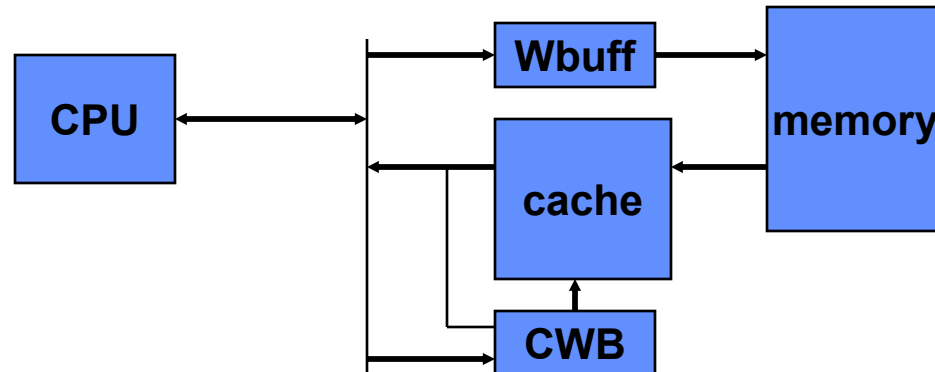
- A hit penalty is required because on hits we must:
 - 1) Access the cache tags during the MEM cycle to determine a hit
 - 2) Stall the processor for a cycle to update a hit cache block

$$\text{CPI}_{\text{memory}} = 0.17 \text{ cycles/instr.} + (0.13 \text{ st})(1/\text{instr.}) \cdot (1 - 0.017) \cdot 1 \text{ cycle}$$

$$\text{CPI}_{\text{memory}} = 0.17 \text{ cycles/instr.} + 0.13 \text{ cycles/instr.} = 0.30 \text{ cycles/instr.}$$

$$\text{CPI}_{\text{overall}} = 1.5 \text{ cycles/instr.} + 0.30 \text{ cycles/instr.} = 1.80 \text{ cycles/instr.}$$

Perfect Write Buffer + Cache Write Buffer



$$CPI_{\text{memory}} = \text{reads/instr.} \cdot \text{miss rate} \cdot 8 \text{ cycle read miss penalty}$$

- Avoid a hit penalty on writes by:
 - ✓ Add a one-entry write buffer to the cache itself
 - ✓ Write the **last** store hit to the data array during **next** store's MEM
 - ✓ Hazard: On loads, must check CWB along with cache!

$$CPI_{\text{memory}} = 0.17 \text{ cycles/inst.}$$

$$CPI_{\text{overall}} = 1.5 \text{ cycles/inst.} + 0.17 \text{ cycles/inst.} = 1.67 \text{ cycles/inst.}$$

Cache Review

- Uses locality to make slow cheap memory look fast
- Principle of locality
 - Temporal locality
 - Recently accessed data is likely to be accessed again soon
 - Spatial locality
 - Nearby data is also likely to be accessed soon
- Design decisions
 - Size, associativity, block size, write policy, replacement policy
 - Access time, miss rate
- Three major categories of cache misses
 - Compulsory – First time access
 - Conflict – Two items map to the same address
 - Capacity – Insufficient cache size

Cache Organization Options

256 bytes, 16 byte block, 16 blocks

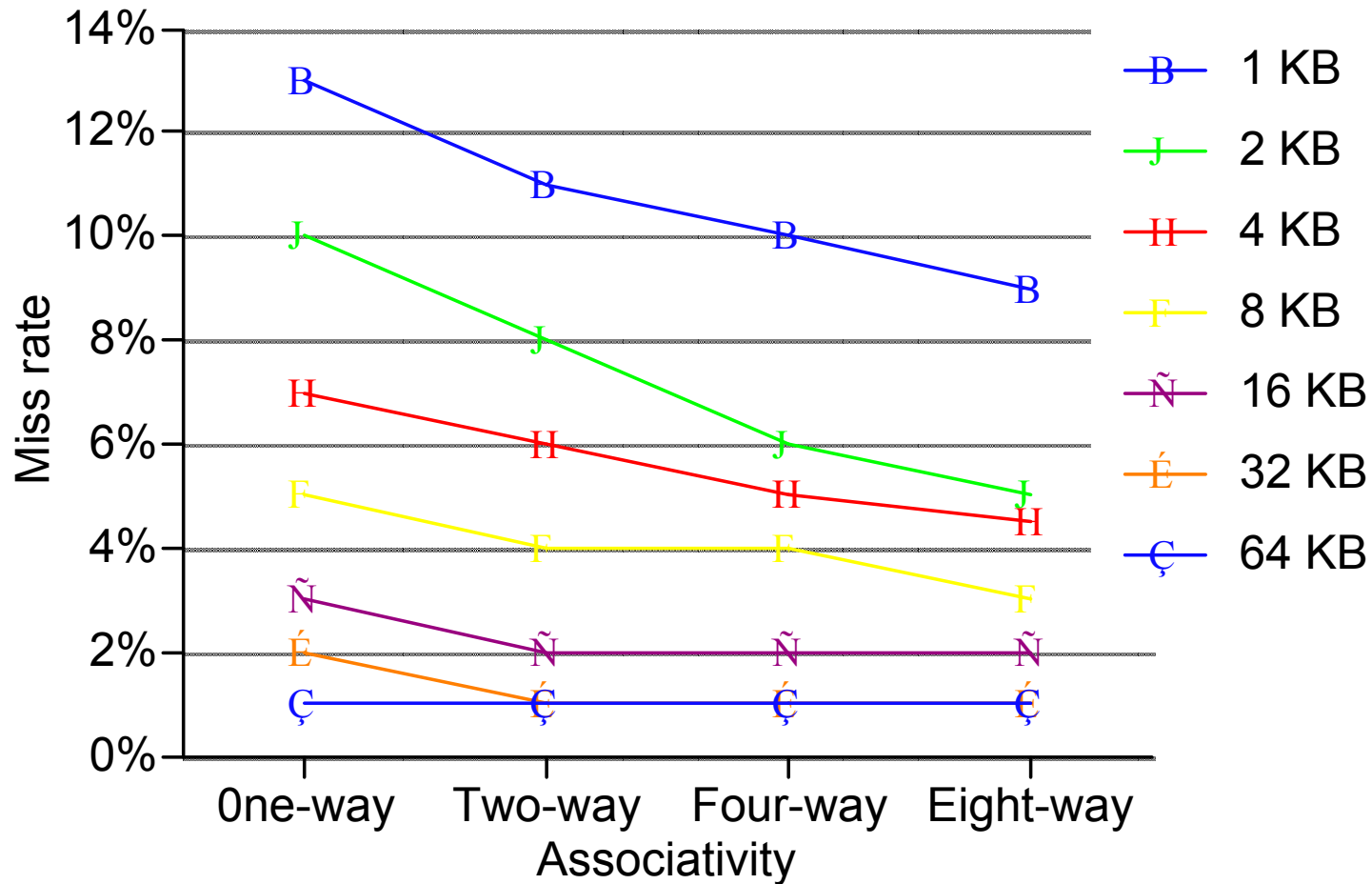
Organization	# of sets	# blocks / set	12 bit Address						
Direct mapped	16	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">tag</td> <td style="text-align: center;">index</td> <td style="text-align: center;">blk off</td> </tr> <tr> <td style="text-align: center;">4</td> <td style="text-align: center;">4</td> <td style="text-align: center;">4</td> </tr> </table>	tag	index	blk off	4	4	4
tag	index	blk off							
4	4	4							
2-way set associative	8	2	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">tag</td> <td style="text-align: center;">index</td> <td style="text-align: center;">blk off</td> </tr> <tr> <td style="text-align: center;">5</td> <td style="text-align: center;">3</td> <td style="text-align: center;">4</td> </tr> </table>	tag	index	blk off	5	3	4
tag	index	blk off							
5	3	4							
4-way set associative	4	4	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">tag</td> <td style="text-align: center;">ind</td> <td style="text-align: center;">blk off</td> </tr> <tr> <td style="text-align: center;">6</td> <td style="text-align: center;">2</td> <td style="text-align: center;">4</td> </tr> </table>	tag	ind	blk off	6	2	4
tag	ind	blk off							
6	2	4							
8-way set associative	2	8	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">tag</td> <td style="text-align: center;">i</td> <td style="text-align: center;">blk off</td> </tr> <tr> <td style="text-align: center;">7</td> <td style="text-align: center;">1</td> <td style="text-align: center;">4</td> </tr> </table>	tag	i	blk off	7	1	4
tag	i	blk off							
7	1	4							
16-way (fully) set associative	1	16	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">tag</td> <td style="text-align: center;">blk off</td> </tr> <tr> <td style="text-align: center;">8</td> <td style="text-align: center;">4</td> </tr> </table>	tag	blk off	8	4		
tag	blk off								
8	4								

Cache Design Summary

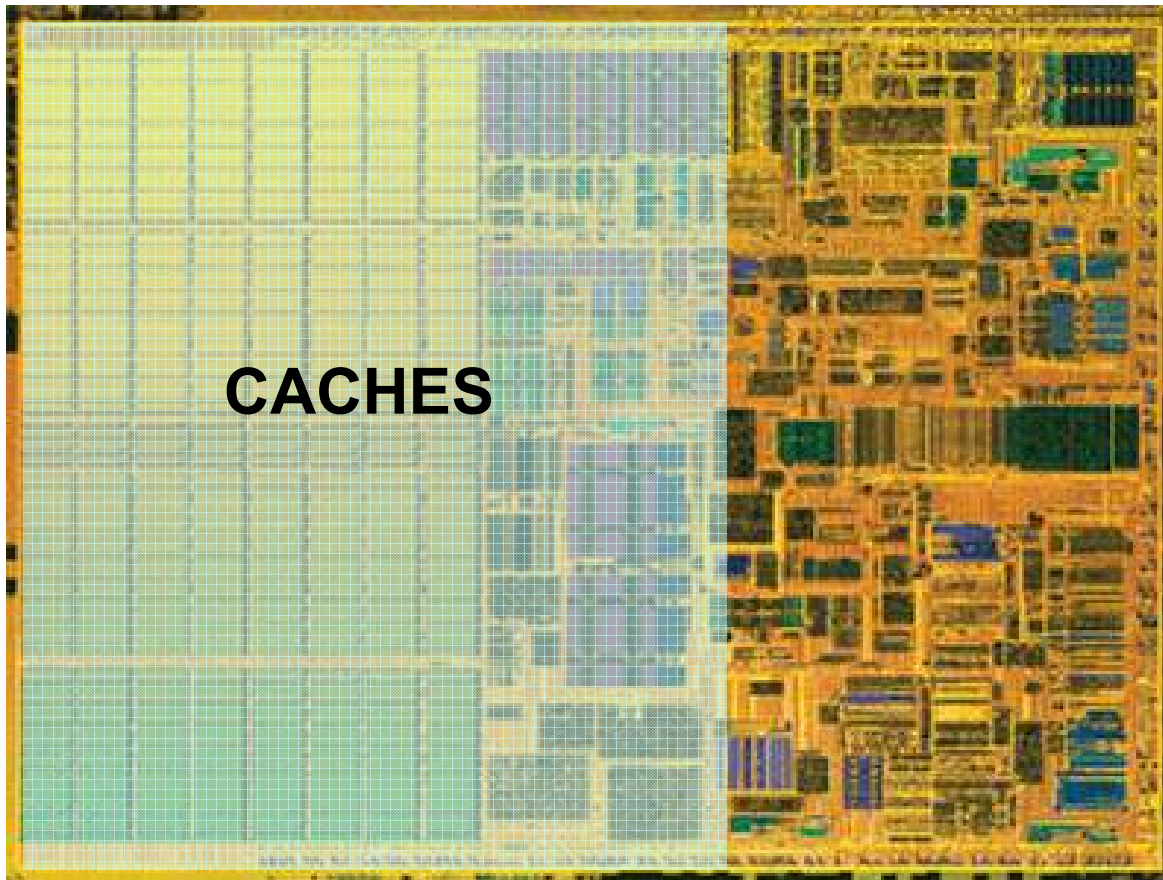
- $AMAT = \text{hit time} + \text{miss rate} \cdot \text{miss penalty}$

	Hit time	Miss rate	Miss penalty
Larger cache	-	+	
Larger block size		+	-
Higher associativity	-	+	
Second level cache			+

Cache Miss Rates for SPEC2000



Intel Pentium M Memory Hierarchy



- 32 K I-cache
 - 4 way S.A., 32 B block
- 32 KB D-cache
 - 4 way S.A., 32 B block
- 1 MB L2 cache
 - 8 way S.A., 32 B block
- 1 GB main memory
 - 3.2 GB/s bus to memory interface chip
- More than half of the die is cache
 - $10.6 \times 7.8 = 83 \text{ mm}^2$
 - 77 M transistors
 - 50 M in cache

Fallacy: Caches are Transparent to Programmers

- Example
 - Cold cache, 4-byte words, 4-word cache blocks
 - C stores arrays in row-major order
 - Assume $N \gg 4$

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate =

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

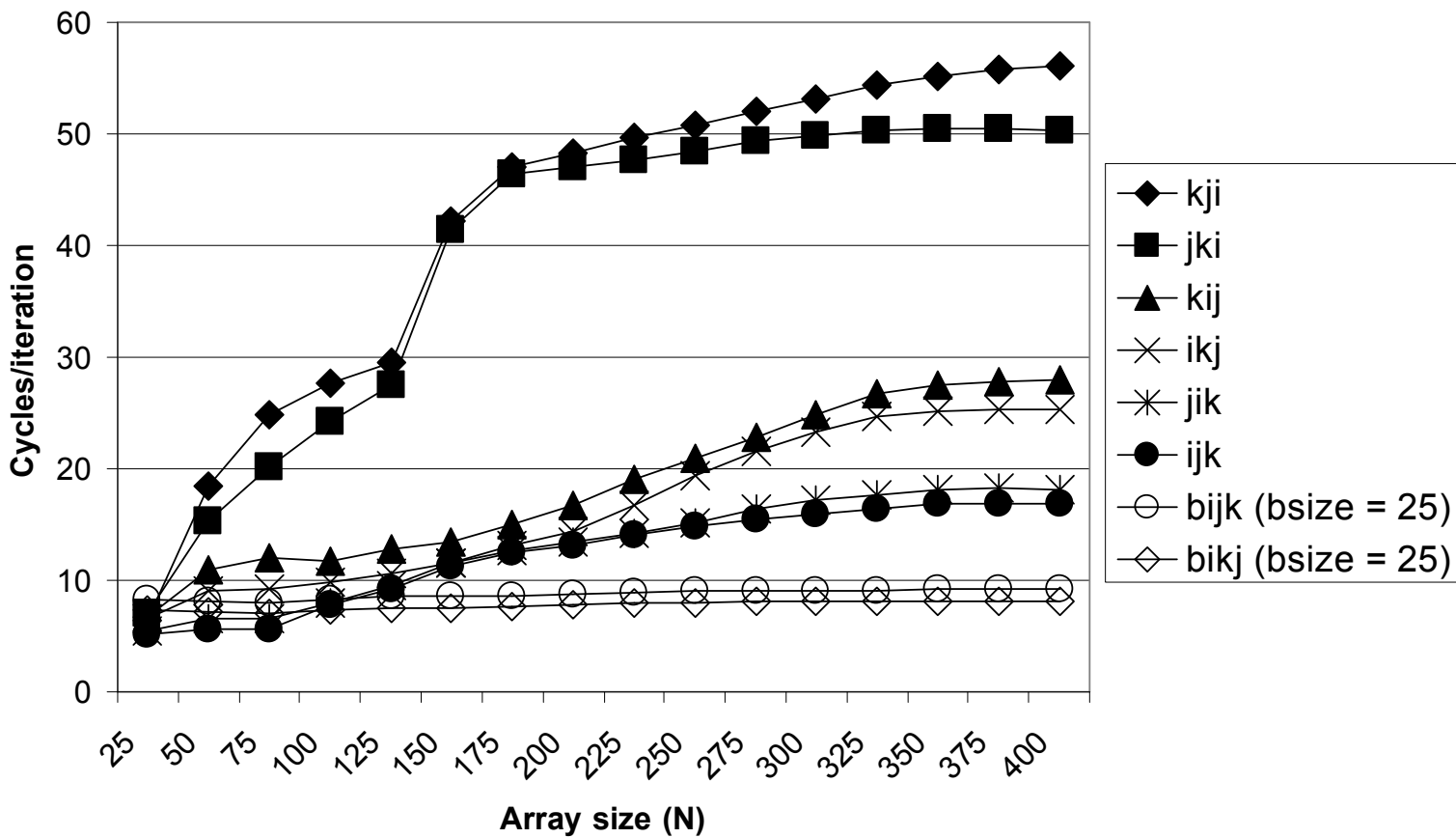
Miss rate =

Matrix multiplication example

- Description:
 - Multiply $N \times N$ matrices
 - $O(N^3)$ total operations
 - Accesses
 - N reads per source element
 - N values summed per destination
- Writing cache friendly
 - Repeated references to variables are good (temporal locality)
 - Stride-1 reference patterns are good (spatial locality)

```
/* ijk */
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```

Pentium blocked matrix multiply performance



Processes, Exceptions, Interrupts, and Virtual Memory

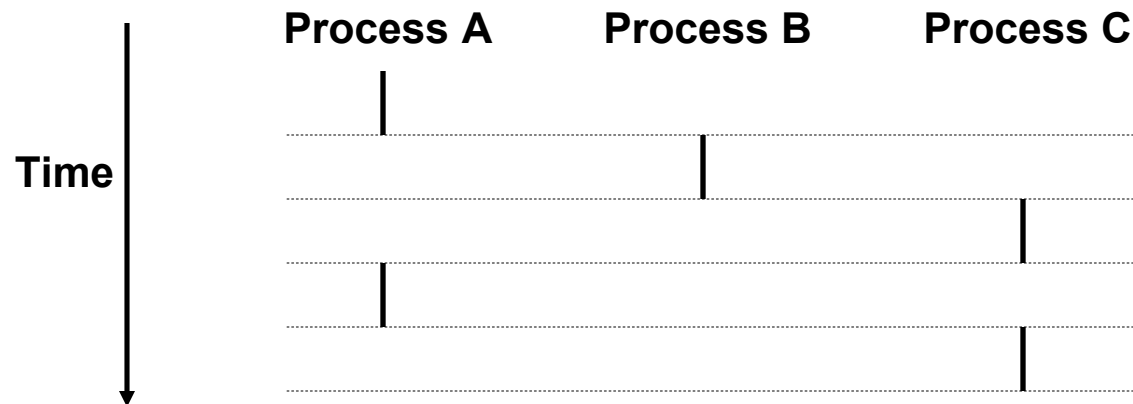
- Operating system
 - Manages hardware resources (CPU, memory, I/O devices)
 - On behalf of user applications
 - Provides protection and isolation
 - Virtualization
 - Processes and virtual memory
- What hardware support is required for the OS?
 - Kernel and user mode
 - Kernel mode: access to all state including privileged state
 - User mode: access to user state ONLY
 - Exceptions/interrupts
 - Virtual memory support

Processes

- Definition: A process is an instance of a running program.
 - Virtualization of CPU and memory
- Process provides each program with two key abstractions:
 - Logical control flow
 - Gives each program the illusion that it has exclusive use of the CPU
 - Private set of register values
 - Private address space
 - Gives each program the illusion that has exclusive use of main memory of infinite size
- How is this illusion maintained?
 - Process execution is interleaved (multitasking)
 - Address space is managed by virtual memory system

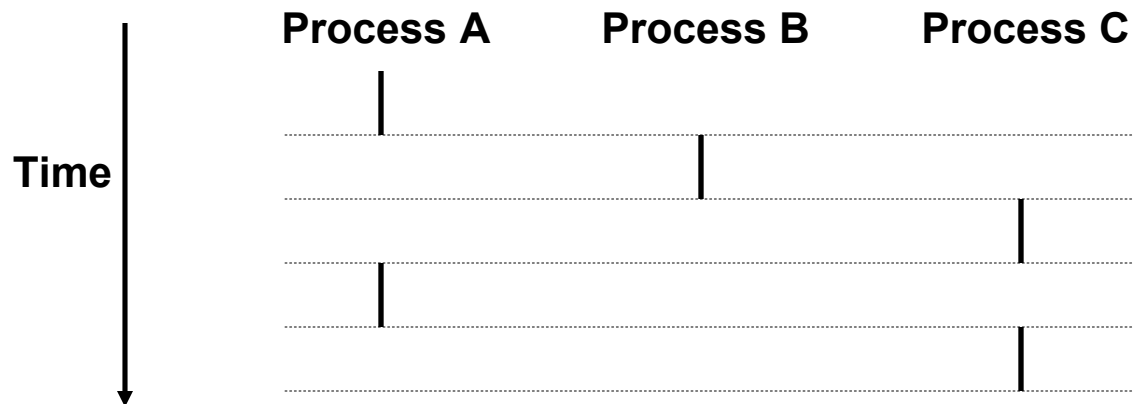
Logical Control Flows

Each process has its own logical control flow



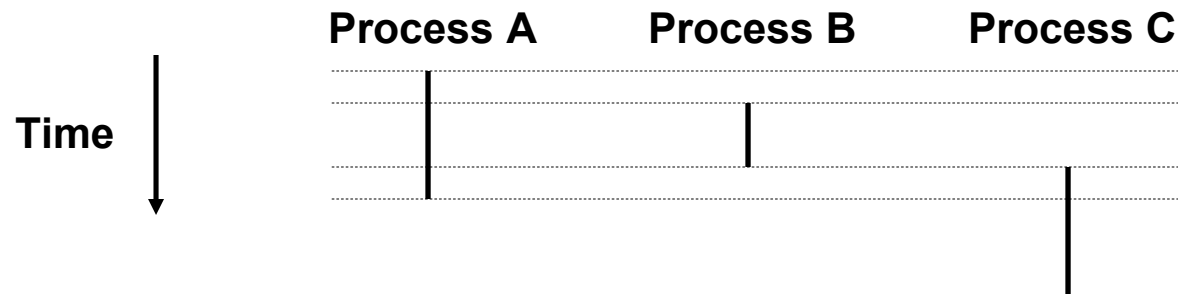
Concurrent Processes

- Two processes run concurrently (are concurrent) if their flows overlap in time.
- Otherwise, they are sequential.
- Examples:
 - Concurrent: A & B, A&C
 - Sequential: B & C



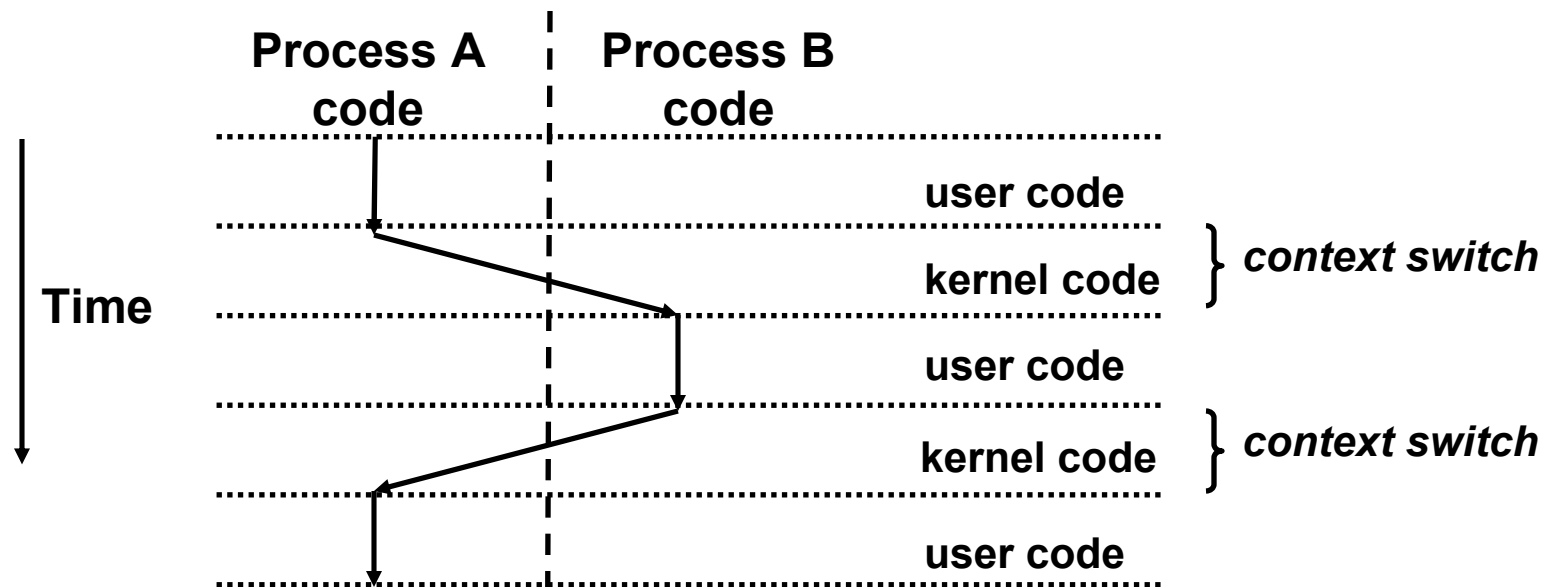
User view of concurrent processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes are running in parallel with each other



Context Switching

- Processes are managed by a shared chunk of OS code called the kernel
 - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a context switch
- Time between context switches 10–20 ms



fork: Creating new processes

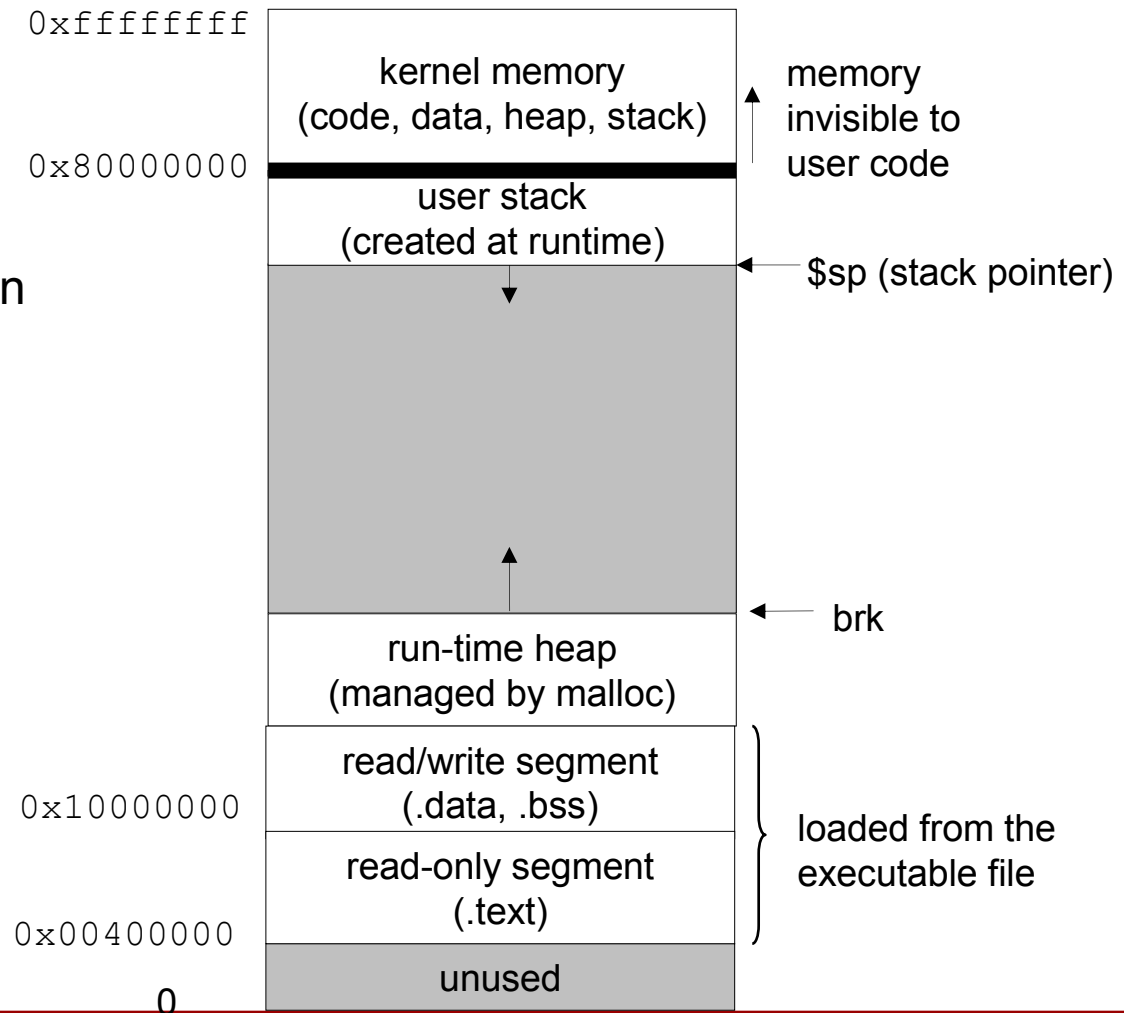
- `int fork(void)`
 - creates a new process (child process) that is identical to the calling process (parent process)
 - returns 0 to the child process
 - returns child's `pid` to the parent process

```
if (fork() == 0) {  
    printf("hello from child\n");  
}  
else {  
    printf("hello from parent\n");  
}
```

Fork is interesting
(and often confusing)
because it is called
once but returns *twice*

Process Address Space

- Each process has its own private address space
- User processes cannot access top region of memory



Altering the Control Flow

- We've discussed two mechanisms for changing the control flow:
 - Jumps and branches
 - Call and return
 - Both implement expected changes in program state
- System call instruction (`syscall`)
 - A jump to operating system code
 - If user process decides to request kernel services
 - Switches mode from user to kernel
 - Jumps to a pre-defined place in the kernel program
 - Still an expected change

Unexpected Changes in Control Flow

- Difficult for the CPU to react to unexpected changes in system state
 - Data arrives from a disk or a network adapter
 - Instruction divides by zero
 - User hits `ctl-c` at the keyboard
 - System timer expires
- CPU needs mechanisms for “exceptional control flow”

Exceptions and Interrupts

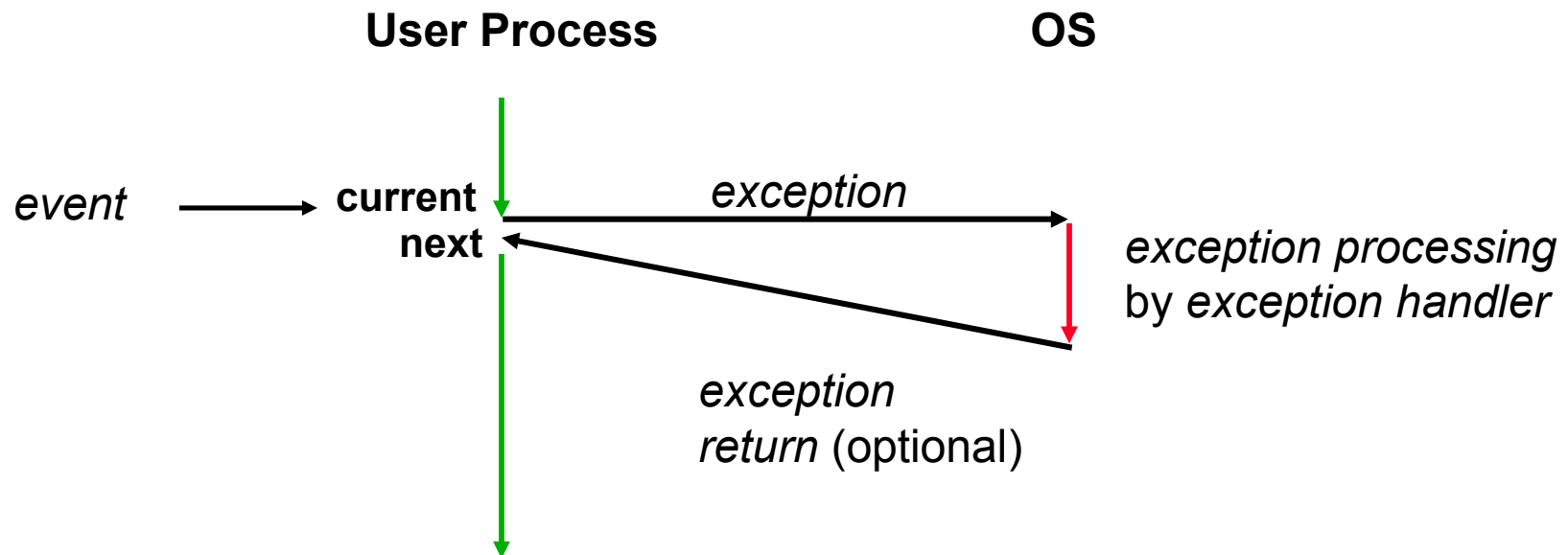
- Change in control flow in response to a system event
- Switch CPU mode from user to kernel

- Interrupts – External CPU events
 - I/O device (disk, keyboard)
 - Clock

- Exceptions– Internal CPU events
 - Arithmetic overflow
 - Illegal instruction
 - Virtual memory address exception
 - Syscall can be viewed as an exceptional event as well
- Implemented as a combination of both hardware and OS software

Exceptional Control Flow

- An exception is a transfer of control to the OS in response to some *event*



- Precise exceptions/interrupts
 - Difficult for deeply pipelined processors
 - Could get exceptions out of order

Reminder: Precise Exceptions

- Definition:
 - All instructions before one that causes exception are fully done
 - The faulting instruction and all that follow do not change program state
 - As if they never executed at all
- Why precise exceptions are difficult with pipelining
 - Imagine a 5 state pipeline during cycle i
 - Stage MEM: a ld causes a virtual memory exception
 - Stage EX: an add causes an arithmetic overflow
 - Stage ID: an invalid instruction exceptions
 - Stage IF: a virtual memory exceptions due to instruction fetch
 - Concurrent or out-of-order exceptions!

Precise Exceptions for Pipelined Processors

- Solution: defer exception handling until WB stage
 - If instruction raised in previous stage, note it in pipeline register, and turn instruction to nop
 - When instruction reaches WB stage: flush pipeline & raise exception
- This works because:
 - Instructions go to WB stage one at the time
 - Instructions go to the WB stage in order
- External events (e.g. IO interrupts)
 - Attach them to a random instruction & proceed as shown above

MIPS Interrupts

- What does the CPU do on an exception?
 - Set `EPC` register to point to the restart location
 - Change CPU mode to kernel and disable interrupts
 - Set `Cause` register to reason for exception also set `BadVaddr` register if address exception
 - Jump to interrupt vector
- Interrupt Vectors
 - `0x8000 0000` : TLB refill
 - `0x8000 0180` : All other exceptions
- Privileged state
 - `EPC`
 - `Cause`
 - `BadVaddr`

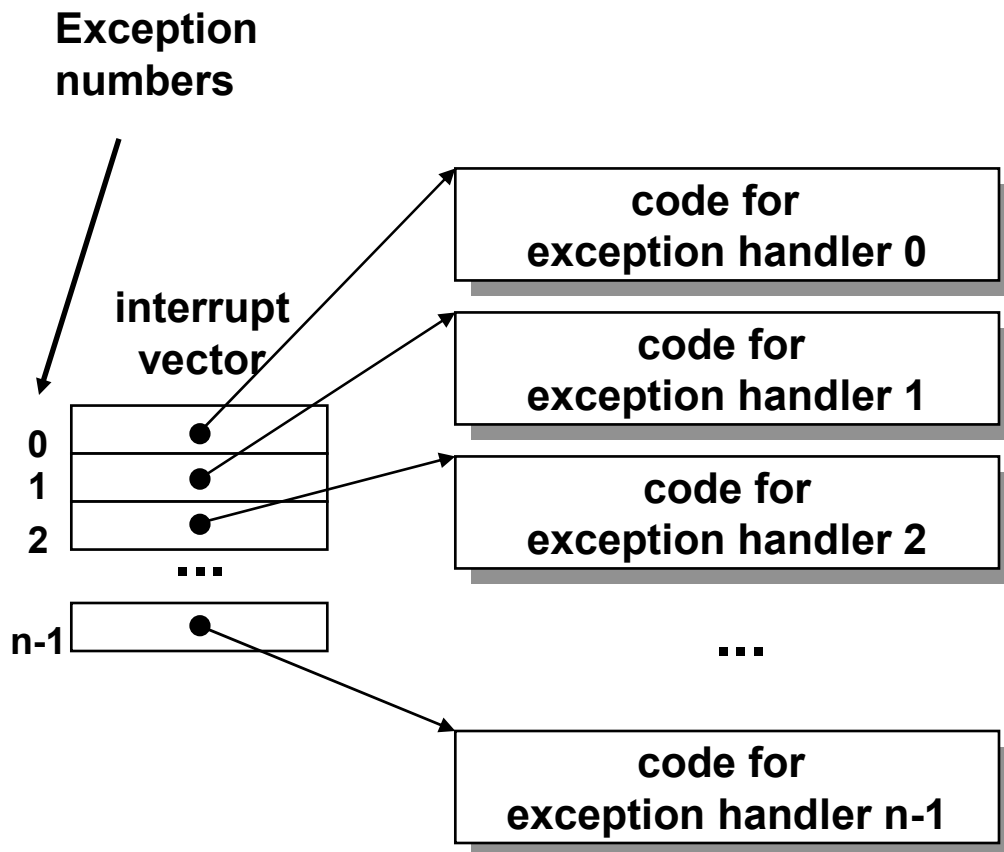
A Really Simple Exception Handler

```
xcpt_cnt:
    la        $k0, xcptcount    # get address of counter
    lw        $k1, 0($k0)       # load counter
    addu      $k1, 1            # increment counter
    sw        $k1, 0($k0)       # store counter
    eret                       # restore status reg: enable
                                # interrupts and user mode
                                # return to program (EPC)
```

- Can't survive nested exceptions
 - OK since does not re-enable interrupts
- Does not use any user registers
 - No need to save them

Interrupt vectors

Accelerating handler dispatch



- 1. Each type of interrupt has a unique exception number k
- 2. Jump table (interrupt vector) entry k points to a function (exception handler).
- 3. Handler k is called each time exception k occurs

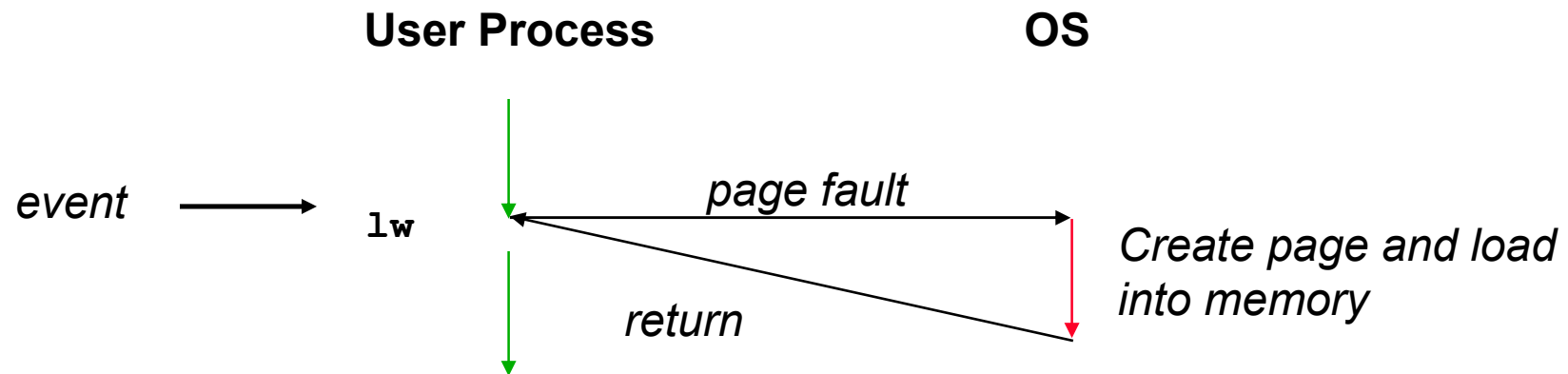
But the CPU is much faster than I/O devices and most OSs use a common handler anyway

Make the common case fast!

Exception Example #1

- Memory Reference
 - User writes to memory location
 - That portion (page) of user's memory is currently not in memory
 - Page handler must load page into physical memory
 - Returns to faulting instruction
 - Successful on second try

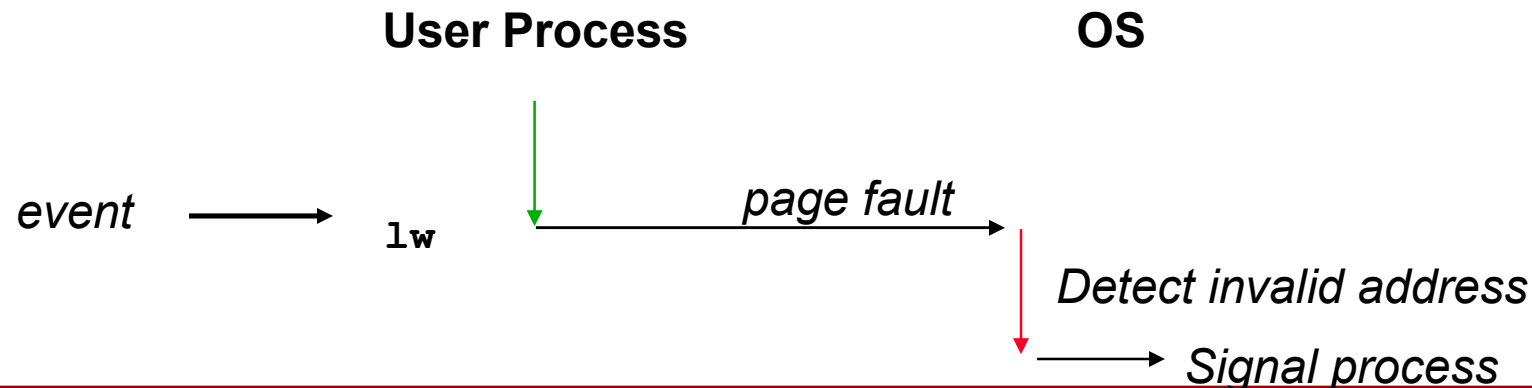
```
int a[1000];
main ()
{
    a[500] = 13;
}
```



Exception Example #2

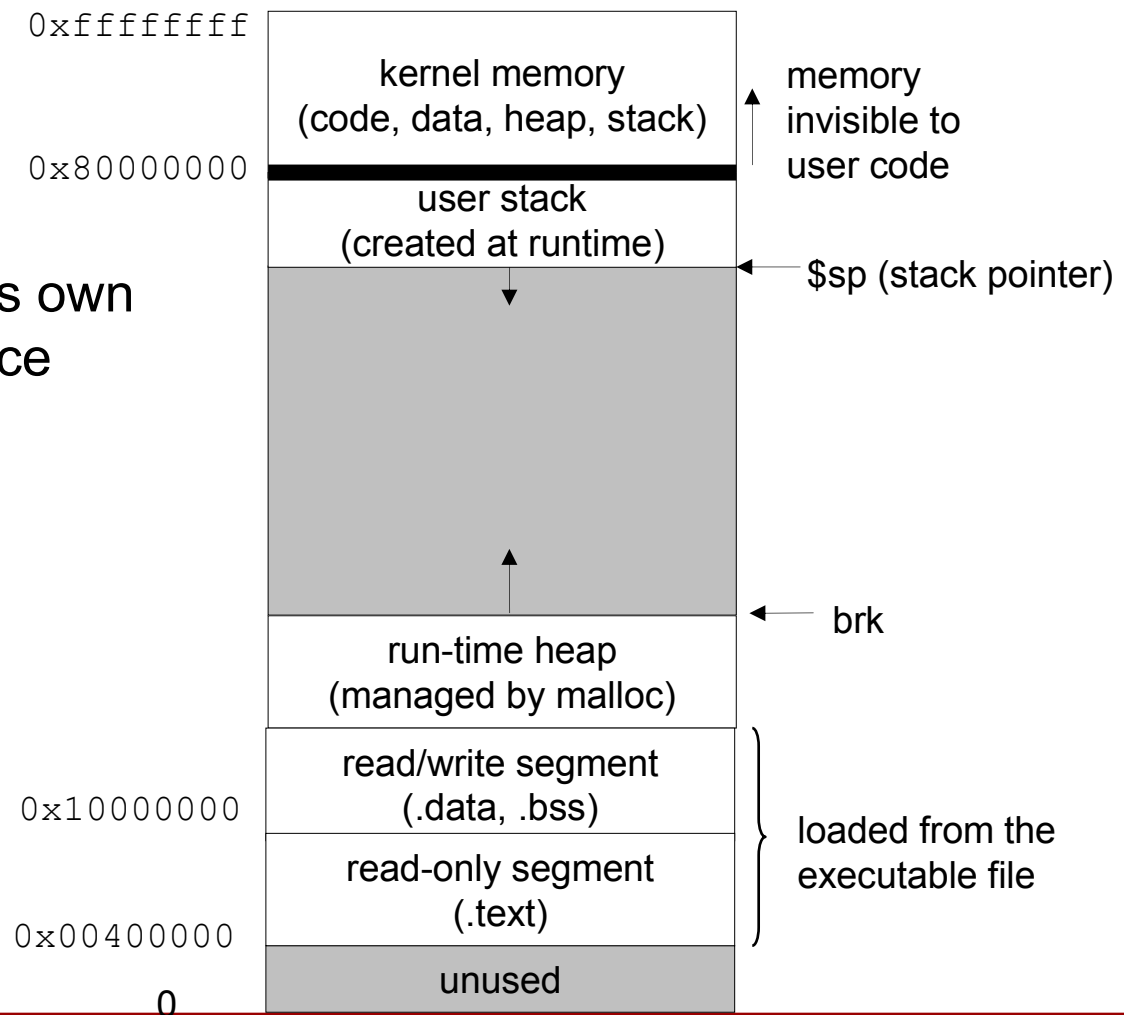
- Memory Reference
 - User writes to memory location
 - Address is not valid
 - Page handler detects invalid address
 - Sends `SIGSEGV` signal to user process
 - User process exits with “segmentation fault”

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



Process Address Space

- Each process has its own private address space

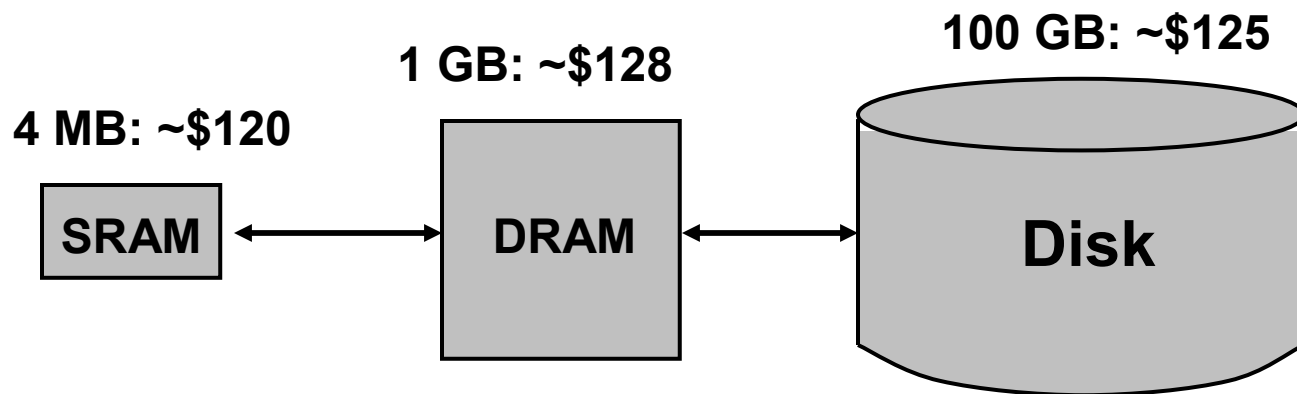


Motivations for Virtual Memory

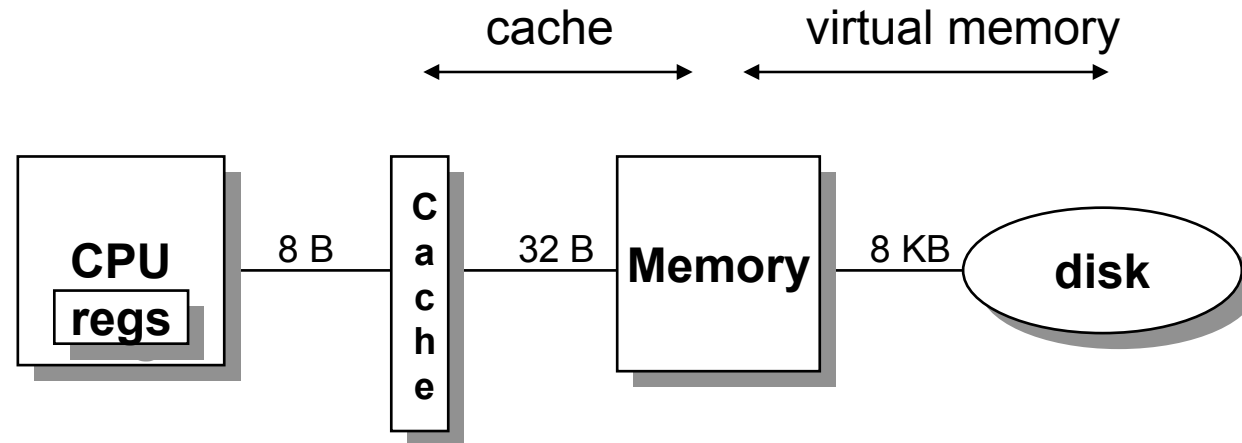
- #1 Use Physical DRAM as a Cache for the Disk
 - Address space of a process can exceed physical memory size
 - Sum of address spaces of multiple processes can exceed physical memory
- #2 Simplify Memory Management
 - Multiple processes resident in main memory.
 - Each process with its own address space
 - Only “active” code and data is actually in memory
 - Allocate more memory to process as needed.
- #3 Provide Protection
 - One process can't interfere with another.
 - because they operate in different address spaces.
 - User process cannot access privileged information
 - different sections of address spaces have different permissions.

Motivation #1: DRAM a “Cache” for Disk

- Full address space is quite large:
 - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
 - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- Disk storage is ~100X cheaper than DRAM storage
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy

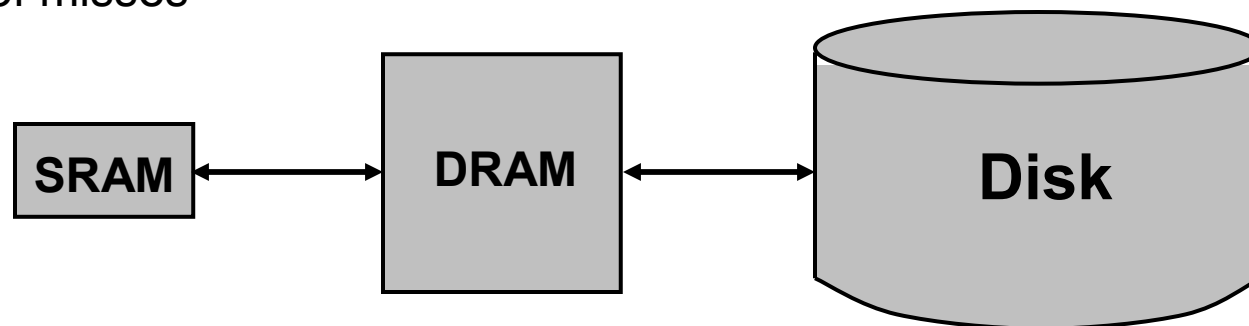


	Register	Cache	Memory	Disk Memory
size:	128 B	<4MB	< 16 GB	> 100 GB
Speed(cycles):	0.5-1	1-20	80-100	5-10 M
\$/Mbyte:		\$30/MB	\$0.128/MB	\$0.001/MB
line size:	8 B	32 B	8 KB	

larger, slower, cheaper 

DRAM vs. SRAM as a “Cache”

- DRAM vs. disk is more extreme than SRAM vs. DRAM
 - Access latencies:
 - DRAM ~10X slower than SRAM
 - Disk ~**100,000X** slower than DRAM
 - Importance of exploiting spatial locality:
 - First byte is ~**100,000X** slower than successive bytes on disk
 - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
 - Bottom line:
 - Design decisions made for DRAM caches driven by enormous cost of misses

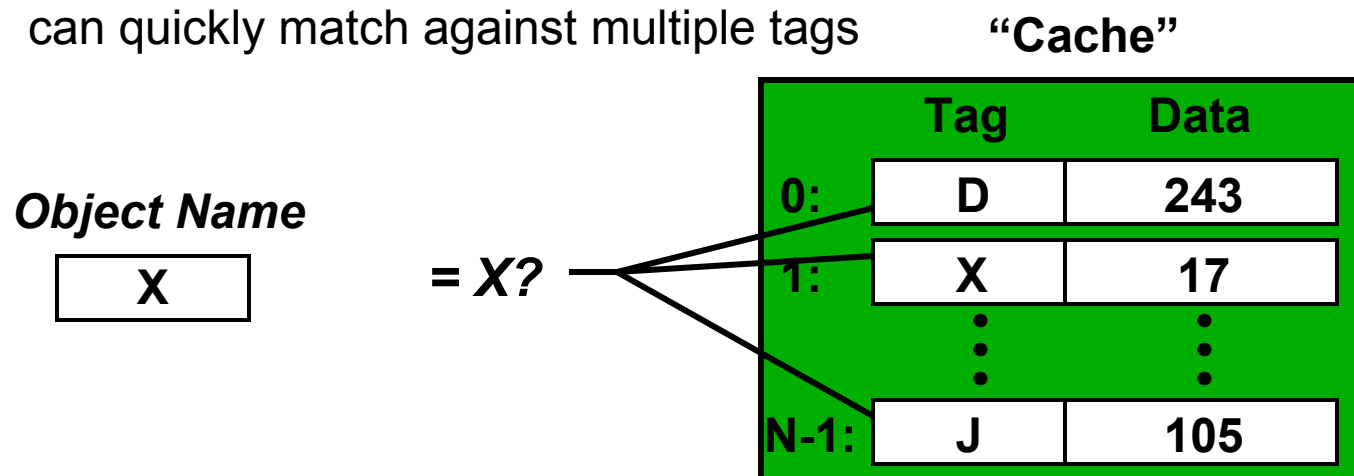


Impact of These Properties on Design

- If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?
 - Line size?
 - Large, since disk better at transferring large blocks
 - Associativity?
 - High, to minimize miss rate
 - Write through or write back?
 - Write back, since can't afford to perform small writes to disk
- What would the impact of these choices be on:
 - miss rate
 - Extremely low. $\ll 1\%$
 - hit time
 - Must match cache/DRAM performance
 - miss penalty
 - Very high. $\sim 20\text{ms}$
 - tag storage overhead
 - Low, relative to block size

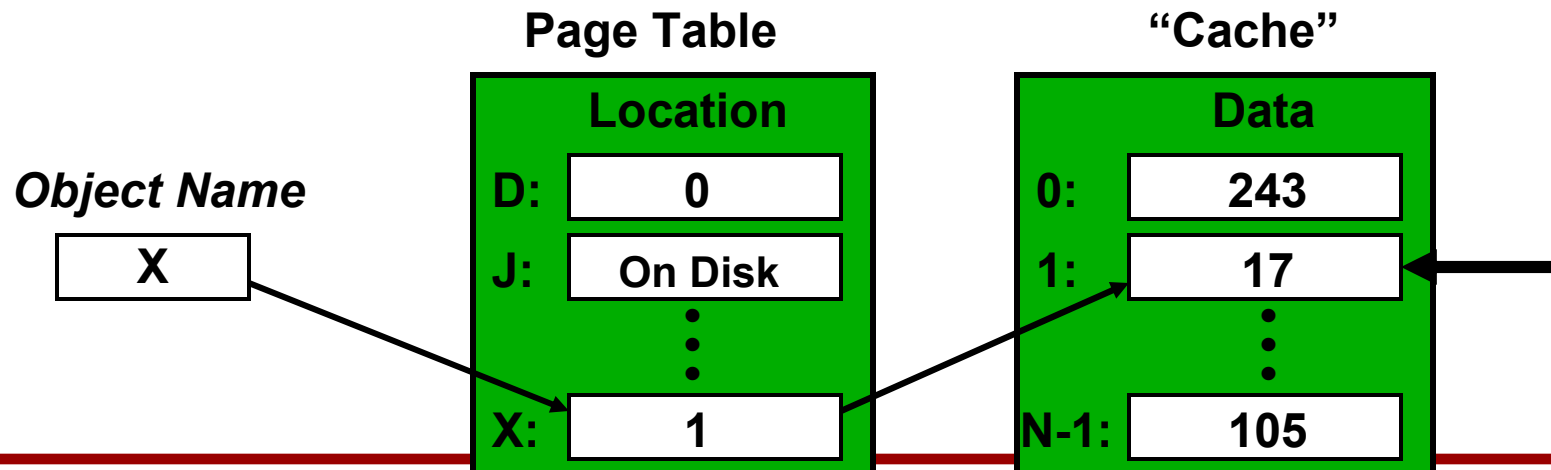
Locating an Object in a “Cache”

- SRAM Cache
 - Tag stored with cache line
 - Maps from cache block to memory blocks
 - From cached to uncached form
 - No tag for block not in cache
 - Hardware retrieves information
 - can quickly match against multiple tags



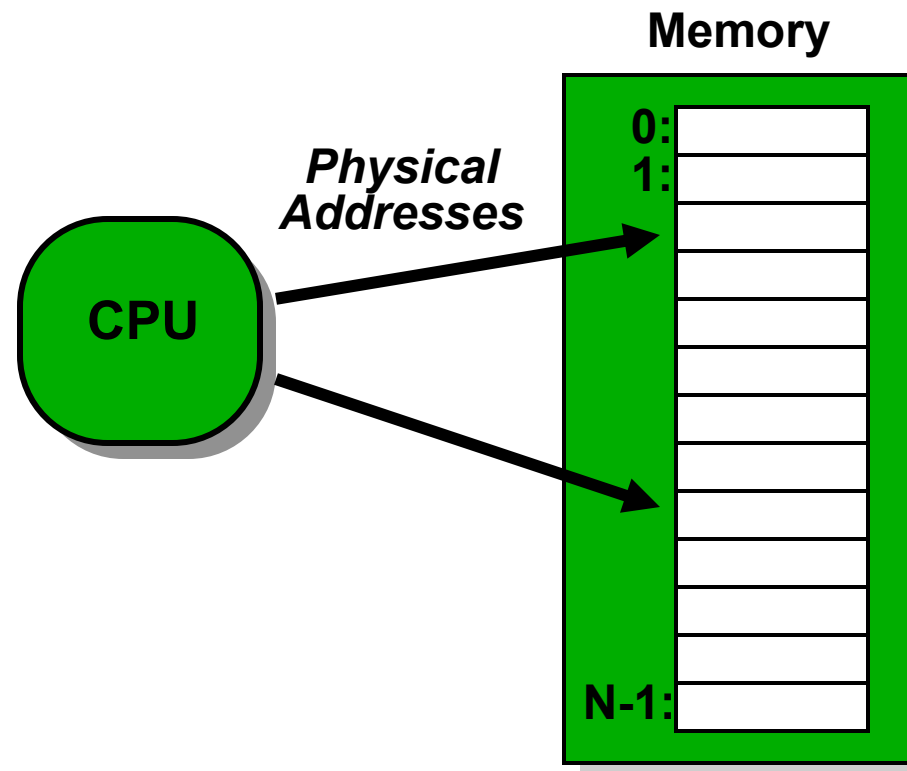
Locating an Object in a “Cache” (cont.)

- DRAM Cache
 - Each allocated page of virtual memory has entry in page table
 - Mapping from virtual pages to physical pages
 - From uncached form to cached form
 - Page table entry even if page not in memory
 - Specifies disk address
 - OS retrieves information



A System with Physical Memory Only

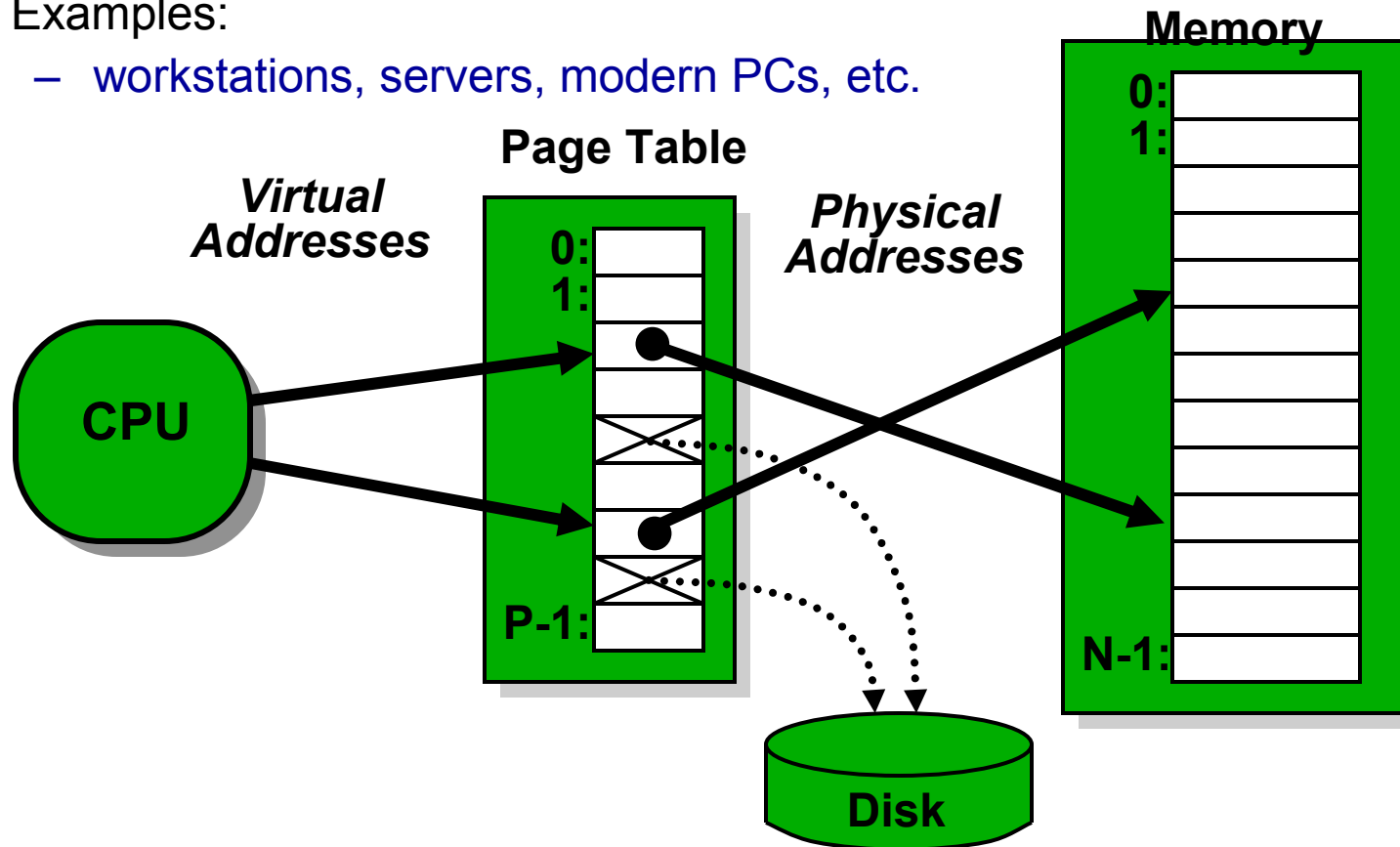
- Examples:
 - Most early Cray machines, early PCs, nearly all embedded systems, etc.



Addresses generated by the CPU point directly to bytes in physical memory

A System with Virtual Memory

- Examples:
 - workstations, servers, modern PCs, etc.

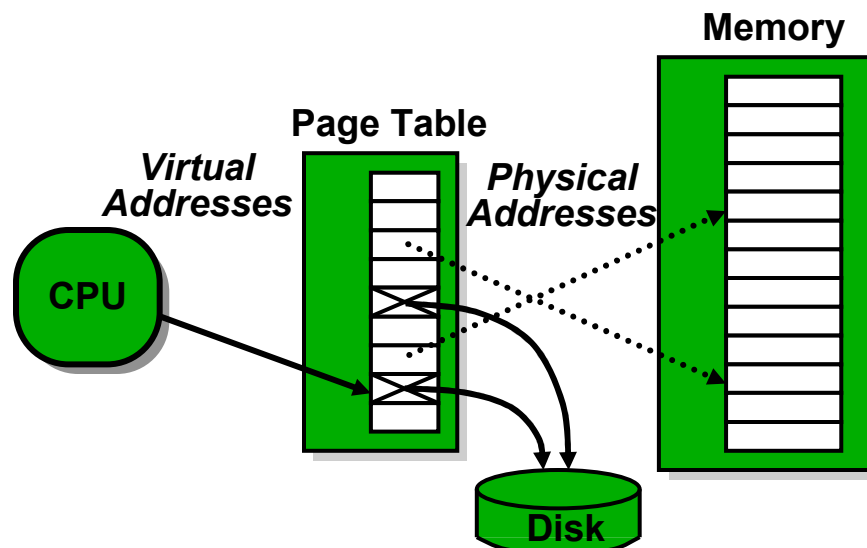


Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

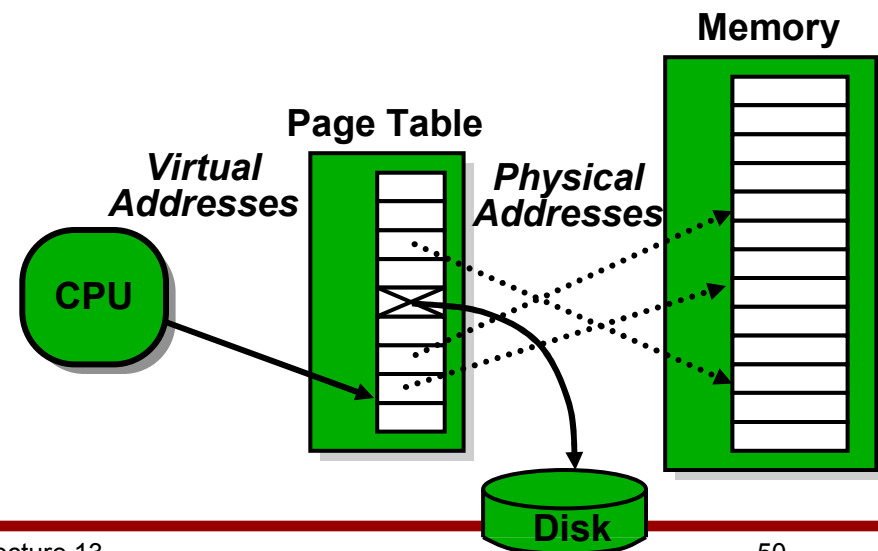
Page Faults (Similar to “Cache Misses”)

- What if an object is on disk rather than in memory?
 - Page table entry indicates virtual address not in memory
 - OS exception handler invoked to move data from disk into memory
 - current process suspends, others can resume
 - OS has full control over placement, etc.

Before fault

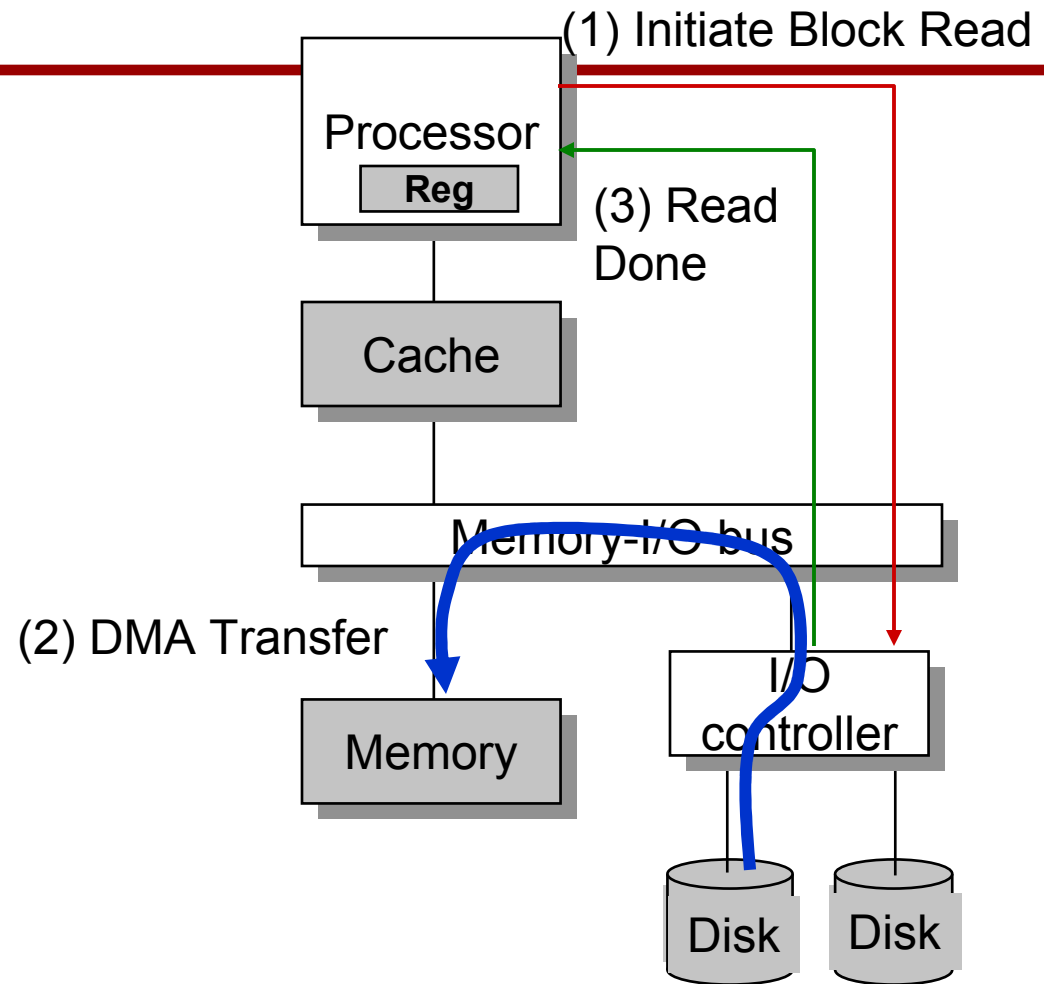


After fault



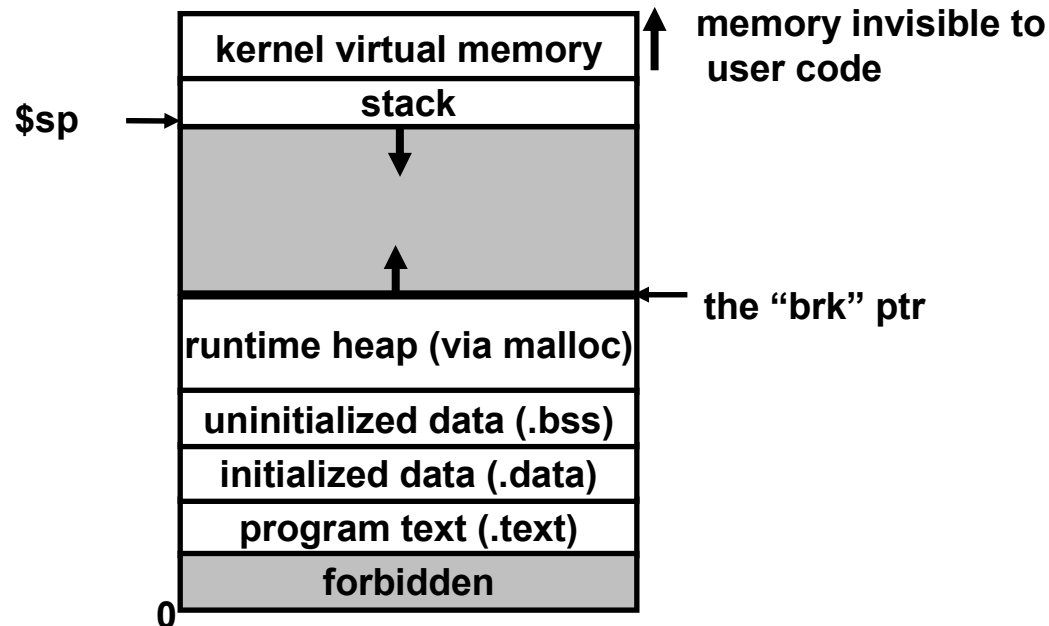
Servicing a Page Fault

- Processor Signals Controller
 - Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
 - Direct Memory Access (DMA)
 - Under control of I/O controller
- I / O Controller Signals Completion
 - Interrupt processor
 - OS resumes suspended process



Motivation #2: Memory Management

- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
 - what if two processes access something at the same address?



Solution: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
 - blocks are called “pages” (both virtual and physical)
- Each process has its own virtual address space
 - operating system controls how virtual pages are assigned to physical memory

