

Memory Hierarchy

Main Points

- Memory technologies (just a quick overview)
- Memory hierarchy
- Cache memories
- Measuring and Improving Cache Performance

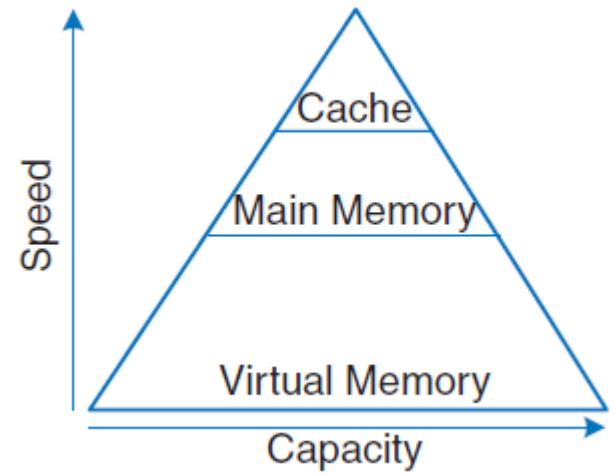
Credits: several figures in these slides come from “Computer Organization and Design. The Hardware/Software Interface” by D.A.Patterson, J.L. Hennessy, 5th edition.

Memory Technologies

- Different memory technologies
- Volatile memories:
 - Latches, flip-flops, register files (or simply *registers*)
 - SRAM (Static Random-Access Memory)
 - DRAM (Dynamic Random-Access Memory)
- Non-volatile memories:
 - ROM
 - NVRAM
 - Flash memory
 - Magnetic disks
 - Optical disks
 - Tape
 - ...

Cost vs Capacity vs Access Time

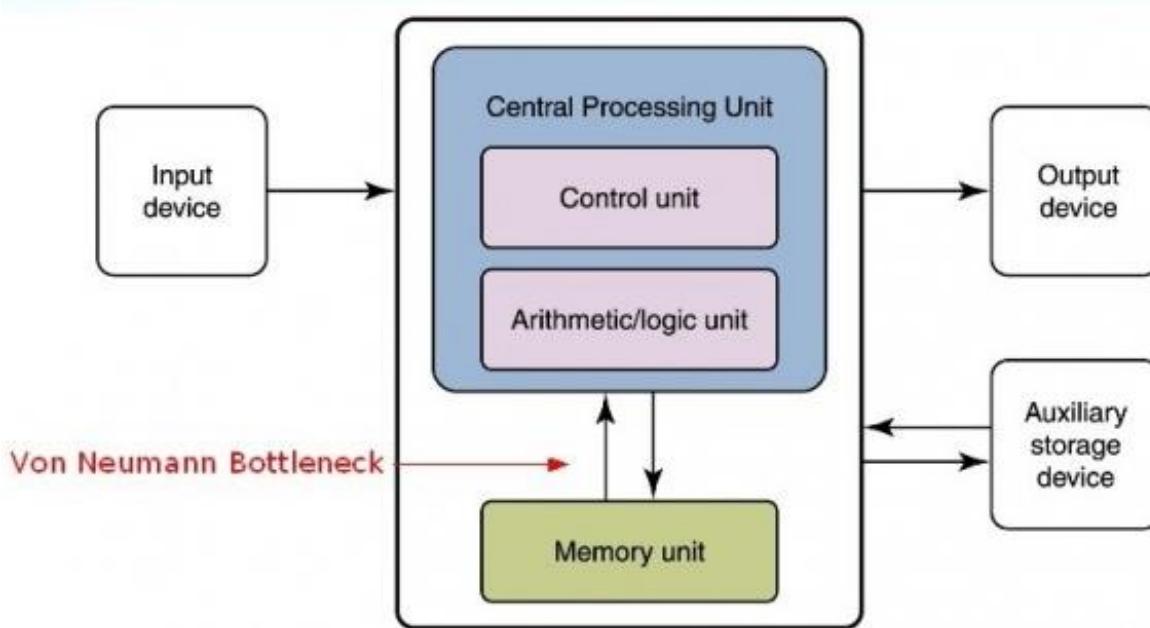
- **SRAM:**
 - Access Time (ns): 0.5 - 1 Bandwidth (GB/s): 25+
 - Price (\$/GB): 5,000
 - Used for registers and caches
- **DRAM**
 - Access Time (ns): 10 - 50 Bandwidth (GB/s): 10
 - Price (\$/GB): 7
 - Used for: RAM
- **Flash memory:**
 - Access Time (ns): 20000 (20 us) Bandwidth (GB/s): 0.5
 - Price (\$/GB): 0.40
 - Used for: SSD disk (secondary and virtual memory – non volatile)
- **Magnetic disks:**
 - Access Time (ns): 5000000 (5 ms) Bandwidth (GB/s): 0.75
 - Price (\$/GB): 0.05
 - Used for: HDD disk (secondary/tertiary storage – non volatile)



Contrasting needs

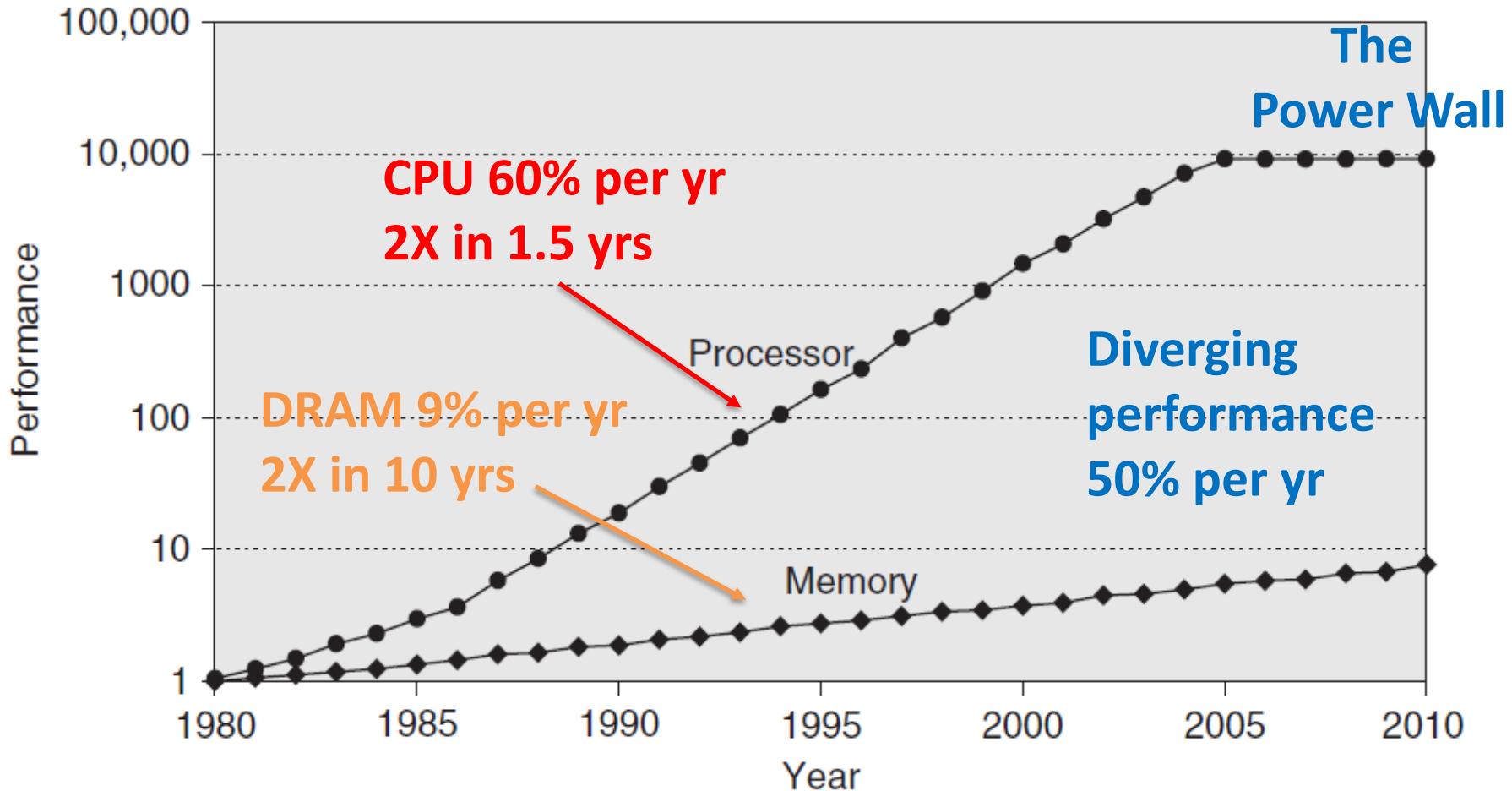
- *General rules:*
 - Larger memory is slower and cheaper
 - Smaller memory is faster and more costly
- Programmers/users need large and fast memories
 - Large enough to fit all needed data
 - Fast to mitigate the *von Neumann bottleneck*

von Neumann architecture



- Computer system performance is limited due to the relative speed of CPUs compared to top rates of data transfer between off-chip memories and CPUs.

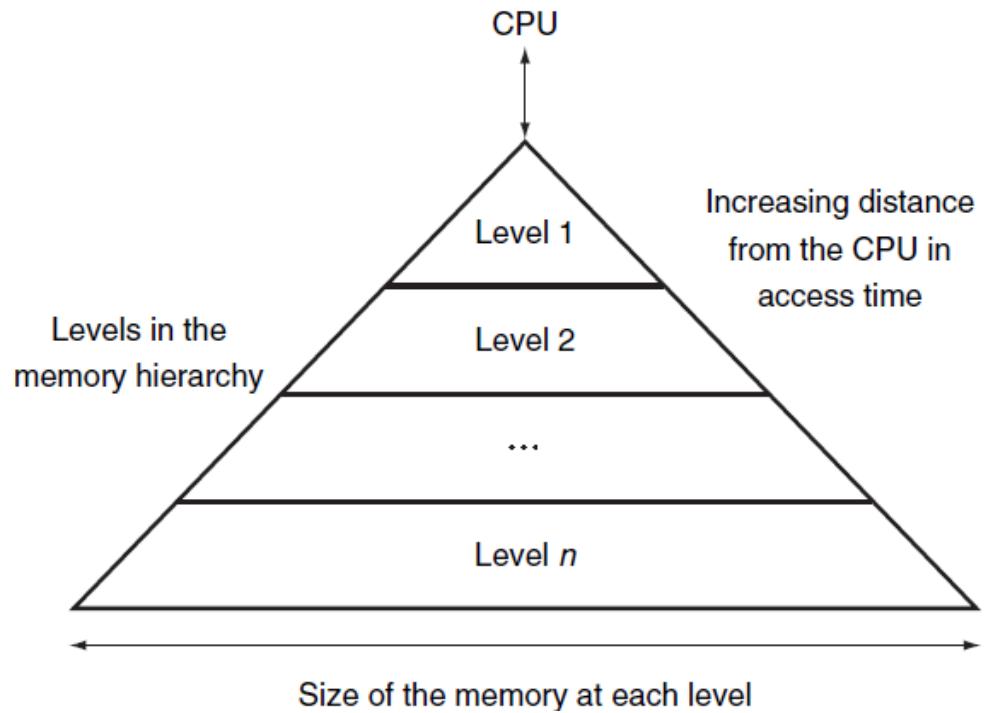
Von Newmann bottleneck



Basic structure of a memory hierarchy

Speed	Processor	Size	Cost (\$/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

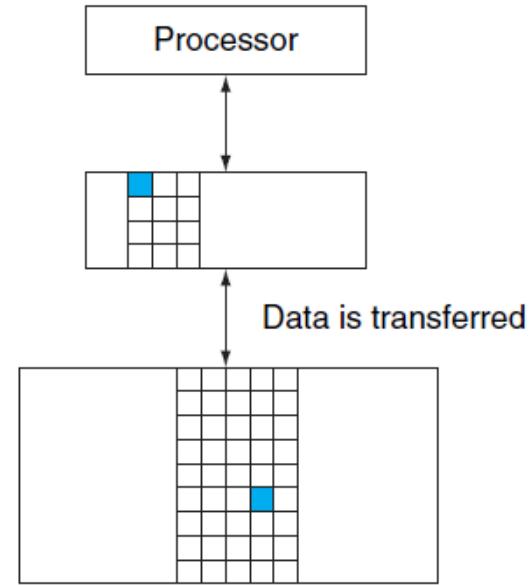
Goal of the memory hierarchy



Goal: By implementing the memory system as a hierarchy, the programmer has the *illusion* of a memory that is as large as the largest level of the hierarchy and (*almost*) as fast as the closest level.

Illusion of large and fast memory

- At the beginning, data resides in the last level of memory
- Referred data moves from level n to level n-1 using different data transfer granularity (e.g., pages, blocks)
- Data movements is transparent to the high-level programmer
- Level n-1 contains a subset of data present in level n
(*inclusive* memory levels)



Questions

- What if the referred data is already present in one memory level ?
 - We need a mechanism to find the right data
- What if one memory level does not have the data and it is full ?
 - We need both mechanisms and policies to replace one of data present to make space for the new one

Terminology

- If the data requested by the processor appears in some block in the closest memory level, this is called a **hit**. Otherwise, the request is called a **miss** and the next memory level is then accessed to retrieve the block containing the requested data.
- The **hit rate** (frequency of successes) is the fraction of memory accesses found in the upper level
 - used as a measure of the performance of the hierarchy
- The **miss rate** is the fraction of memory accesses not found in the upper level
- The **miss penalty** is the time to replace a block in level n with the corresponding block from level n-1.
- The **miss time** is the time to obtain the element in case of miss
 - miss time = miss penalty + hit time

Hit rate, miss rates and AMAT

$$MR = \frac{\text{Number of misses}}{\text{Number of total memory accesses}} = 1 - HR$$

$$HR = \frac{\text{Number of hits}}{\text{Number of total memory accesses}} = 1 - MR$$

$$AMAT = t_{M_0} + MR_{M_0} * (t_{M_1} + MR_{M_1} * (t_{M_2} + MR_{M_2} * (t_{M_3} + \dots)))$$

hit time miss rate miss penalty

- **AMAT** is the *Average Memory Access Time*
 - hit-time + (1 – hit-rate)*miss-penalty
- MR_x is the Miss Rate (probability of miss) at memory level x
- T_x is the access time for a memory hit at level x (**hit time**)

Observation: If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

The locality principle

- *Locality of reference* (or **locality principle**) refers to the phenomena in which a program tends to access *the same set of memory locations for a given time period*.
- It has been observed that, If the program refers a memory location, then
 - *The same memory location* will be used again soon with a high probability
 - The elements “close to” the memory location just accessed will be referenced soon with a high probability

The locality principle

- The *locality principle* is the *driving force* that makes the memory hierarchy work properly
- It increases the probability of reusing data blocks that were previously moved from level n to level n-1 (i.e., closer to the CPU), thus reducing the *miss rate*.

Locality characterization

- **Temporal locality** (or *data reuse*): data items referenced recently are likely to be referenced again soon
 - Examples: instructions in a loop, data variables

```
for(int i=0; i<10; ++i) { s1 += i; s2 -= i; }
```
 - *Memory hierarchies take advantage of data reuse by keeping more recently accessed data items closer to the CPU*

Locality characterization

- **Spatial locality:** data items near those referenced recently are likely to be referenced again soon
 - Examples: sequential instruction accesses, data elements of an array

```
for(int i=0;i<10; ++i) func(A[i]);
```
 - *Memory hierarchies take advantage of spatial locality by moving data items in blocks (i.e., contiguous words in memory) from level n to level n-1*

Data transferring

- Data is transferred between only two adjacent memory levels at a time.
 - The processor accesses only at the first level
- Transfers always happen at ***block*** granularity to exploit spatial locality
- The size of the block may vary
 - For the cache, the block is called *cache line* or *cache block* (typical values are 64 - 128 bytes – 8, 16 memory words)
 - Pages or segments for the RAM
 - Disk block for the disk

Locality examples

- Let's consider the following snippet of C code:

```
// sum and A are global variables  
int i;  
for(i=0, sum=0;i<N;++i)  
    sum += A[i]
```

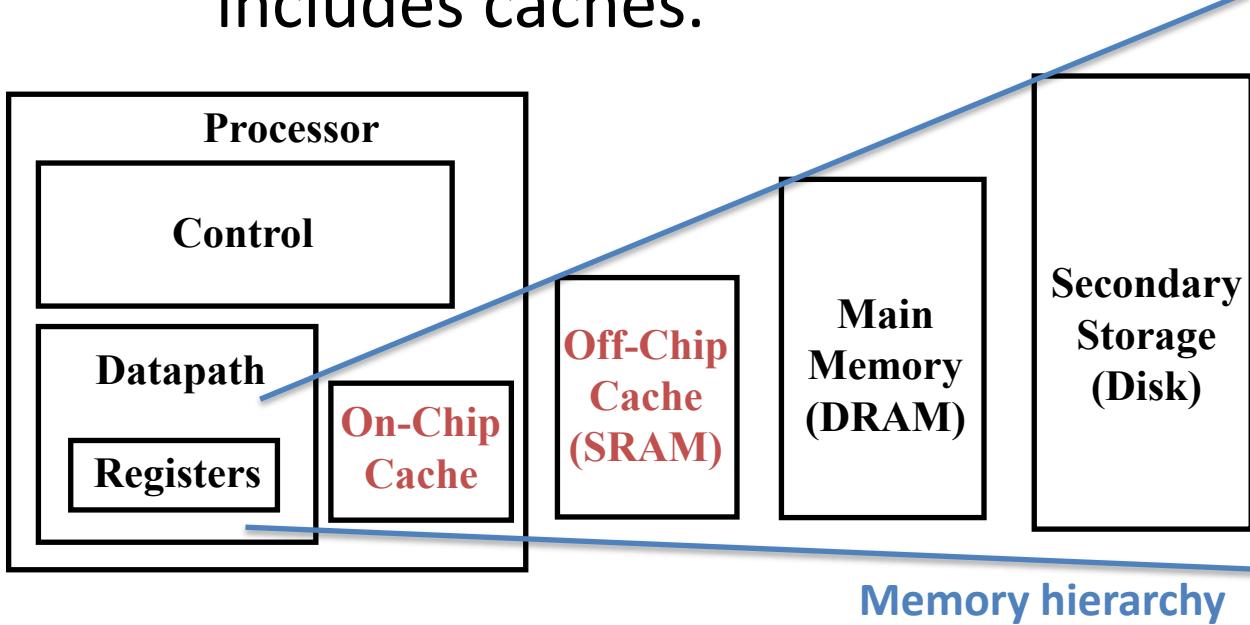
one possible
compilation

```
@ r2= N, r3 = i, r4 = sum  
loop: cmq r3, r2  
      beq end  
      ldr r12, [r0, r3, lsl #2] @ r12 A[i]  
      add r4, r4, r12  
      str r4, [r1]  
      add r3, r3, #1  
      b loop  
end: ...
```

- Loop body executed N times; each **instruction** fetched N times in sequence
→ *both temporal and spatial locality for the instructions*
- sum* is repeatedly read and written over time (the same for the loop index *i*, but it is usually kept in a register) → *sum* exhibits temporal locality
- A* is stored contiguously in memory; *A* exhibits spatial locality because its elements are accessed one after the others
 - $A[0], A[1], \dots, A[i-1], A[i], A[i+1], A[i+2], \dots, A[N-1]$

Cache memories

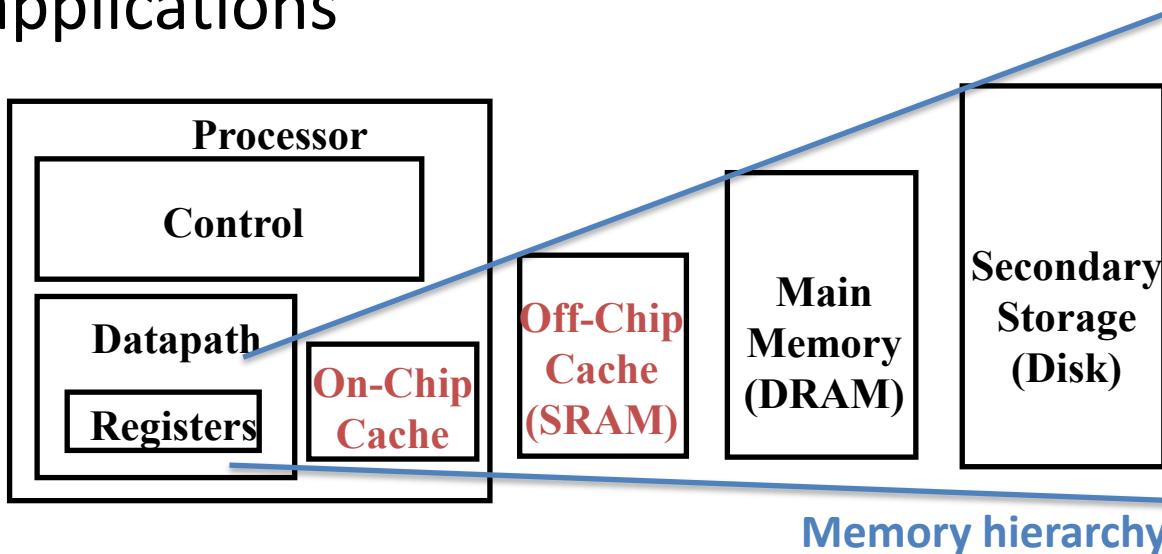
- Cache memory is the level of the memory hierarchy closest to the CPU
 - Every general-purpose computer built today, from servers to low-power embedded processors, includes caches.



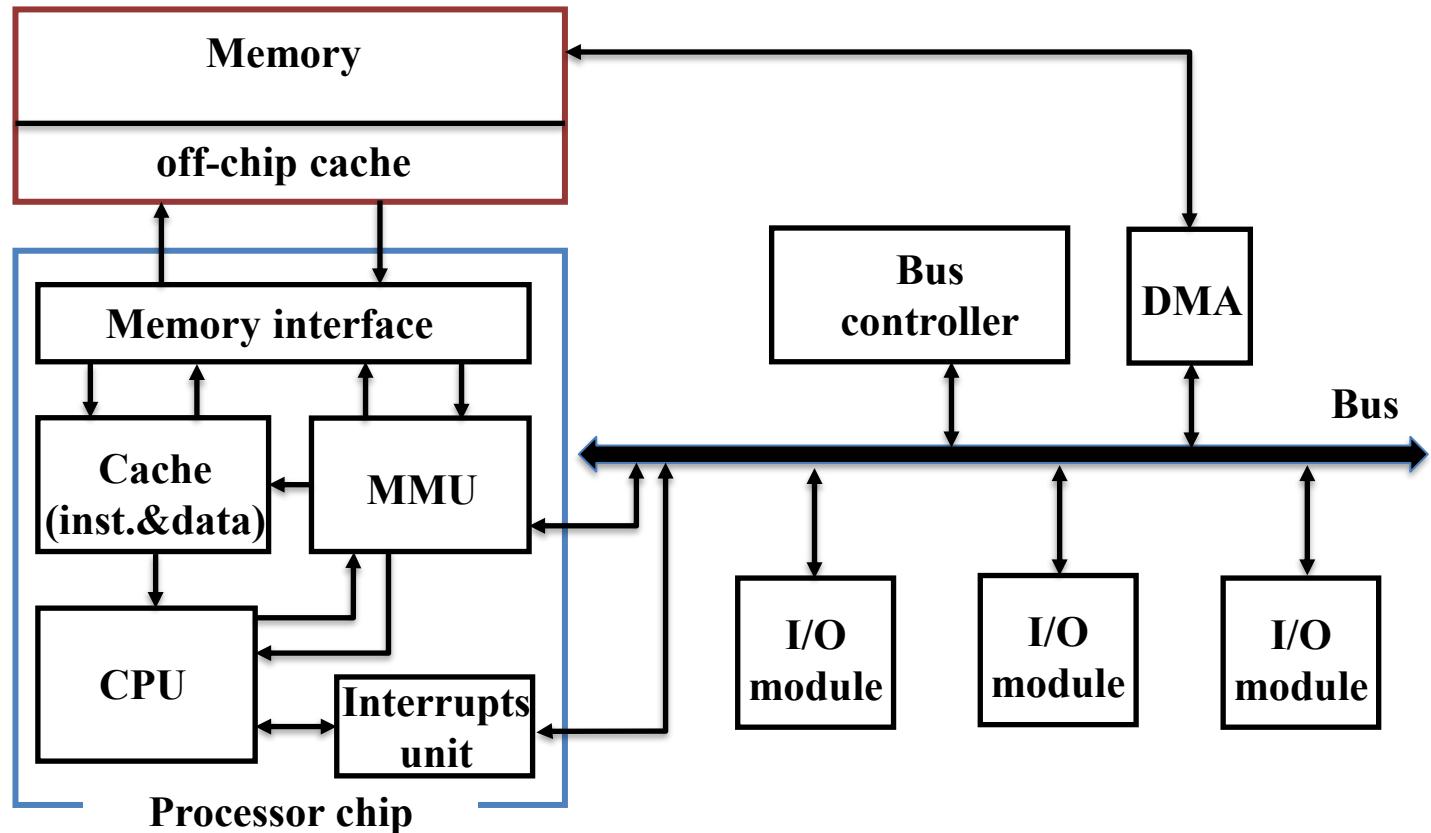
Cache: a hiding place especially for concealing and preserving provisions or implements.
– Merriam Webster Online Dictionary, 2015.
www.merriam-webster.com

Handling data movements

- Between first level cache and registers handled by the compiler
- Between cache levels and between caches and RAM handled by the HW (microarchitecture)
- Between storage devices and RAM handled by the OS and applications

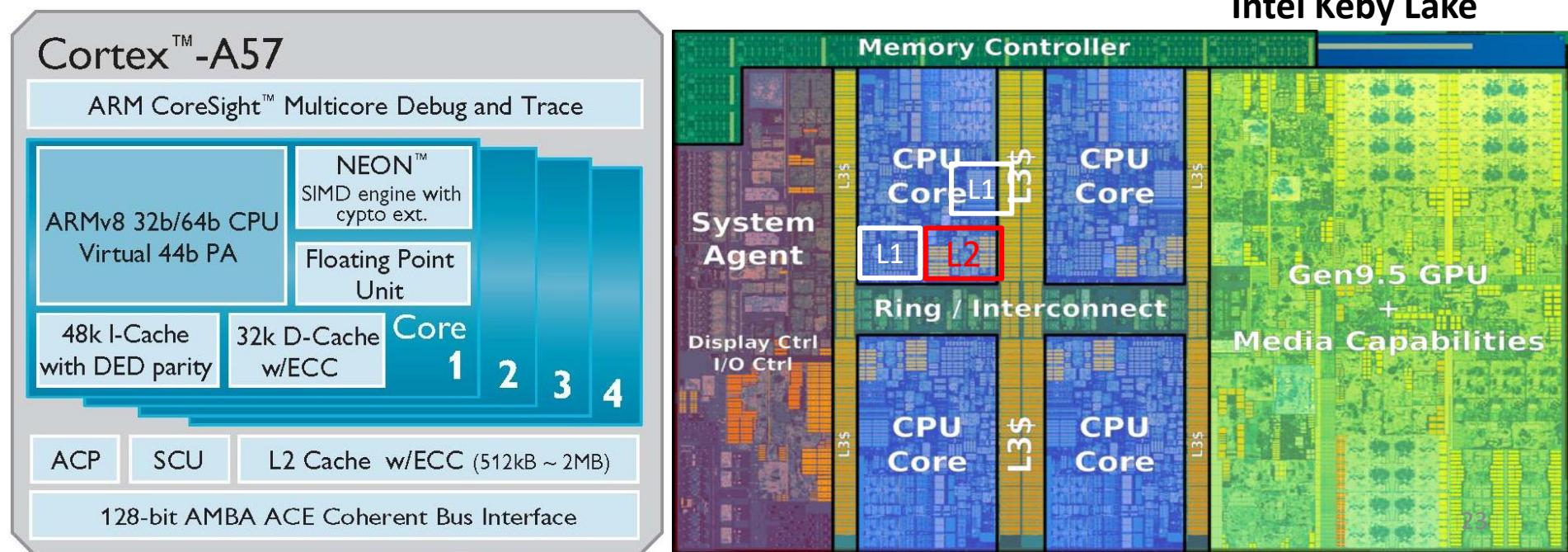


Cache-based architecture (logical scheme)



Caches on modern processors

- Multi-levels on-chip caches (L1, L2, L3)
- Per-core private cache (L1)
- Large on-chip shared cache (L2 or L3)
- Separate data and instruction caches (e.g., L1d, L1i)



Caches usage

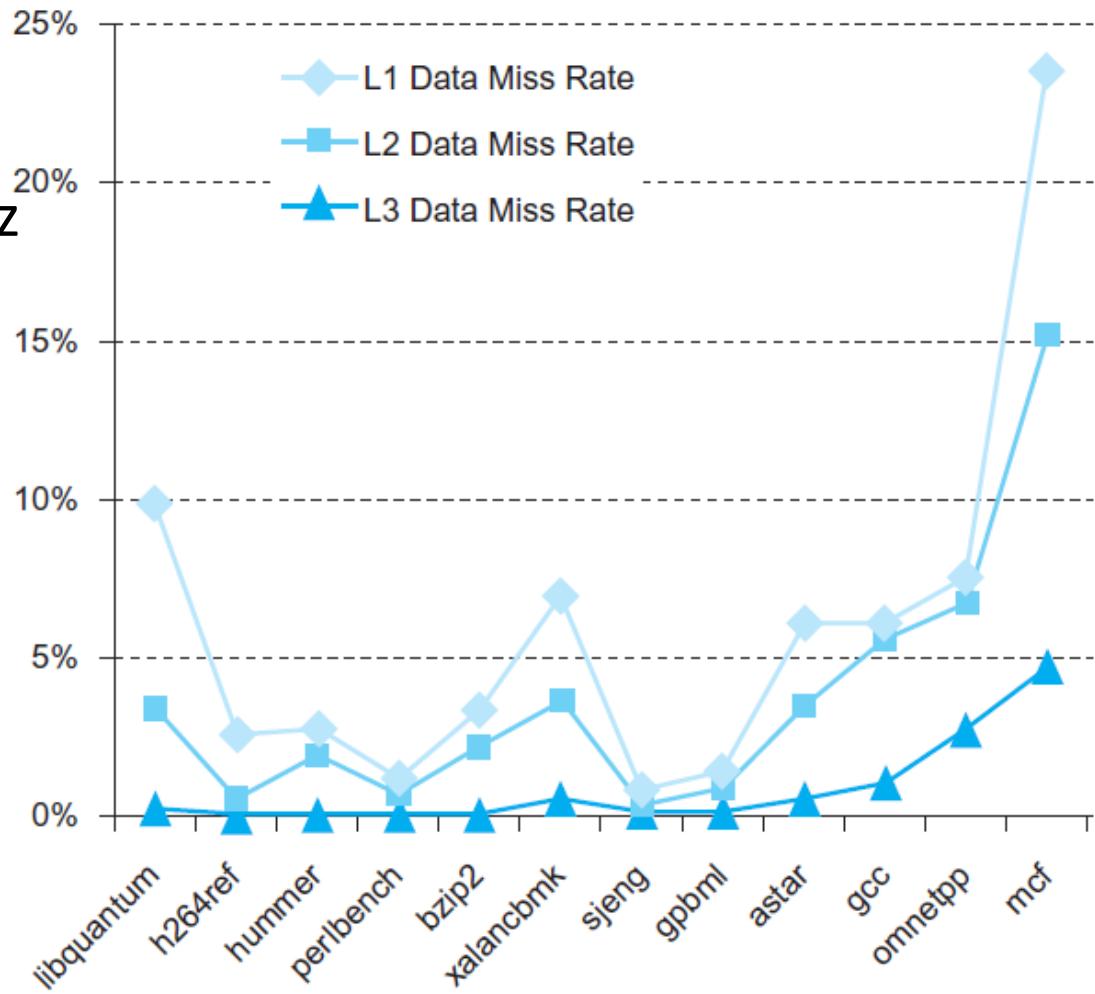
- Caches are organized in *lines*, each line contains a block of memory words (e.g., 8 memory words)
- The first time the processor asks for a memory word a **cache miss** occurs
 - The block containing the memory word is transferred into the cache
- Next requests
 - If the data is present in the block → **cache hit**
 - If the data is not present → **cache miss**
 - The block containing the data is transferred into a cache line

Cache effects on AMAT

- The usage of large caches in a memory hierarchy helps to reduce the von Neumann bottleneck
- Quantitative example. Suppose the following values:
 - $t_M = 50\text{ns}$ (main memory service time)
 - $t_{L1} = 1\text{ns}$ (L1 hit time, i.e., cache hit service time)
 - Miss rates (MR_{L1}): 5%
 1. No cache: $AMAT = 50\text{ns}$
 2. With L1 cache: $AMAT = t_{L1} + MR_{L1} * t_M = 1 + 0.05 * 50 = 3.5\text{ns}$

Cache miss rates

- SPECCPU2006 benchmarks
- Intel Core i7 920 @2.66GHz
 - 4 cores, 8 HW contexts
 - L1d 4x32KB, L1i 4x32KB
 - L2 4x256KB
 - L3 8MB shared
- t_M about 100 cycles
- t_{L1} 4 cycles
- t_{L2} 10 cycles
- t_{L3} 35 cycles



Cache Performance

$$CPU_{time} = ClockCycles * ClockCycleTime = IC * CPI * ClockCycleTime$$

where:

- IC (*Instruction Count*) is the number of program instructions executed
- CPI (*ClockCycles Per Instruction*) is defined as $\frac{ClockCycles}{IC}$
- CPU time can be further divided into the cycles that the CPU spends executing the instructions with no misses ($CPI_{Perfect}$) and the clock cycles that the CPU spends waiting for the memory system (CPI_{Stall} or *Memory-stall ClockCycles*)
$$CPI = (CPI_{Perfect} + CPI_{Stall})$$
- The *Memory-stall clock cycles* can be defined as the sum of the stall cycles coming from reads and writes. For simplicity, let's assume that the read/write miss rates and miss penalties are the same for load and stores
 - Here we also assume that **write buffer** stalls are negligible (see “Optimizing Writes”)

$$CPI_{Stall} = \underbrace{\frac{Memory\ accesses}{Program}}_{Miss\ rate\ per\ memory\ instruction} * Miss\ rate * Miss\ penalty$$

Cache Performance Example

- Assume a miss rate of 2% for the instruction cache and of 4% for the data cache, a miss penalty of 100 cycles for all misses, and a frequency of 36% of loads and stores. If the CPI is 2 without memory stalls (i.e., $CPI_{Perfect}$), determine how much faster the processor runs with a perfect cache that never misses.
- CPI memory stalls for instructions and data accesses:

$$CPI_{Stall-Instr} = 1 * 0.02 * 100 = 2 \text{ cycles}$$

$$CPI_{Stall-Data} = 0.36 * 0.04 * 100 = 1.44 \text{ cycles}$$

$$CPI_{Stall} = 2 + 1.44 = 3.44$$

Therefore, the total CPI including memory stalls is: $CPI = 2 + 3.44 = 5.44$

$$\frac{CPU_{time \ with \ stalls}}{CPU_{time \ perfect}} = \frac{IC * (CPI_{Perfect} + CPI_{Stall}) * ClockCycleTime}{IC * CPI_{Perfect} * ClockCycleTime} = \frac{5.44}{2}$$

- The performance with the perfect cache is better by a factor of 2.72

Designing a caching system

- **Main objective:** minimize AMAT

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

- For minimizing AMAT we can:
 - Reduce the miss rate
 - Reduce the miss penalty
 - Reduce the hit time

Designing a caching system

- Questions:
 - How large is the cache block? How many blocks for each level?
 - How do we know if a data item is present in the cache?
 - We need a mapping function: memory address -> cache block id
 - If the data item is not present, how to retrieve the block?
 - How to deal with block conflicts?
 - Which is the replacement policy?
 - What happen if the block to be replaced has been modified?
 - How to keep consistent different memory levels?
 - What does it mean data consistency?

How is data found?

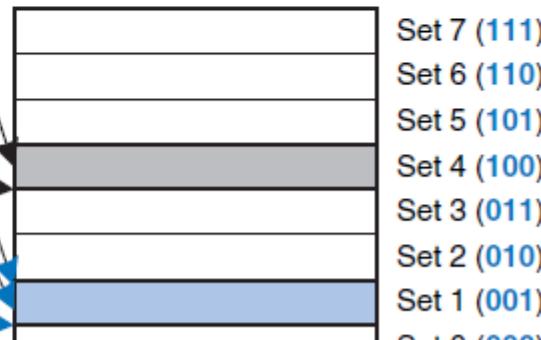
- A cache of capacity C is organized into S sets each one holding B blocks (or *lines*). b is the number of words per block.
- Example $b=4$

cache block (cache line)	Word4	Word3	Word2	Word1
-----------------------------	-------	-------	-------	-------
- The *mapping function* maps one memory address to exactly one set
- Caches can be categorized based on the number of blocks in a set
 - *Direct mapped* cache has $\#S=\#B$
 - *N-way set-associative* cache. Each set contains N blocks. $S= B/N$
 - *Fully associative* cache has $S=1$
- From now on, let's consider a system with 32-bit addresses and 32-bit words → the memory has 2^{30} words.

Direct Mapped Cache

Address	Data
11...11111100	mem[0xFFFFFFFFFC]
11...111111000	mem[0xFFFFFFFFF8]
11...11110100	mem[0xFFFFFFFFF4]
11...11110000	mem[0xFFFFFFFFF0]
11...11101100	mem[0xFFFFFFFEC]
11...11101000	mem[0xFFFFFFF8]
11...11100100	mem[0xFFFFFFE4]
11...11100000	mem[0xFFFFFE0]
⋮	⋮
00...00100100	mem[0x00000024]
00...00100000	mem[0x00000020]
00...00011100	mem[0x0000001C]
00...00011000	mem[0x00000018]
00...00010100	mem[0x00000014]
00...00010000	mem[0x00000010]
00...00001100	mem[0x0000000C]
00...00001000	mem[0x00000008]
00...00000100	mem[0x00000004]
00...00000000	mem[0x00000000]

- $b=1$ (**one memory word per block**), $S=B=8$, $C= B*b=8$
- Since the number blocks is a power of 2, use the low-order bits of a block address to compute its cache location
- In this case we need $\log_2 8 = 3$ bits

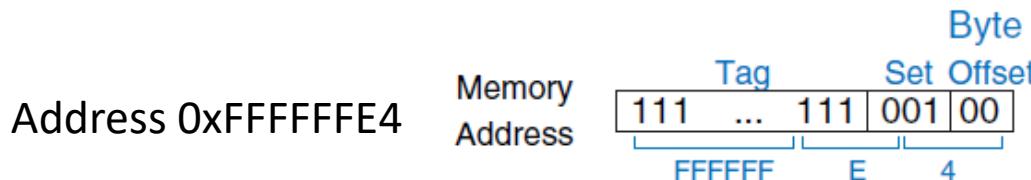


2^{30} -Word Main Memory

2^3 -Word Cache

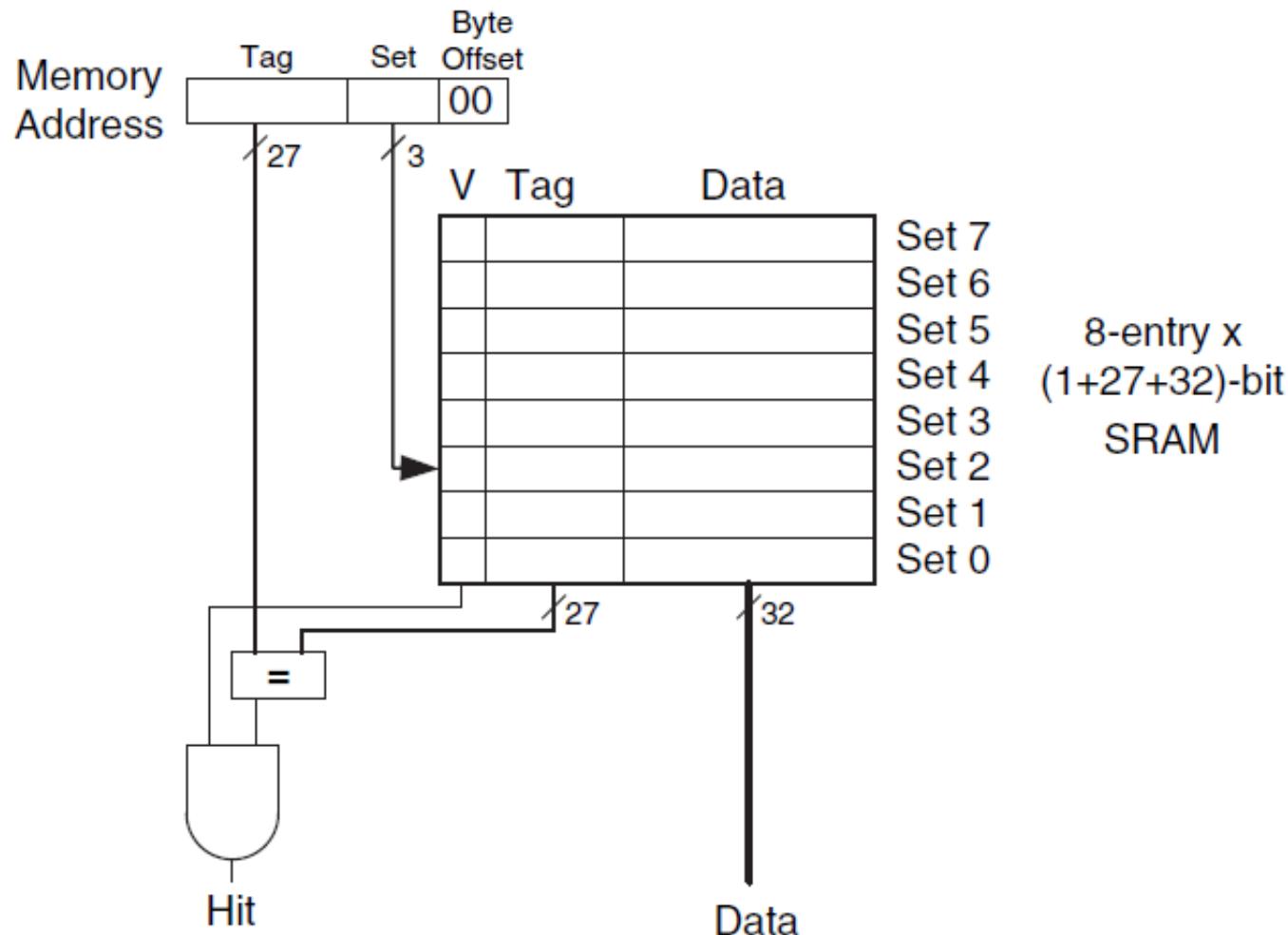
Direct Mapped Cache

- How do we know which block is in a cache location?
 - We need a set of **tags to each cache location** identifying the block address in cache
 - Actually, the tag only needs to contain the higher-order bits of the memory address



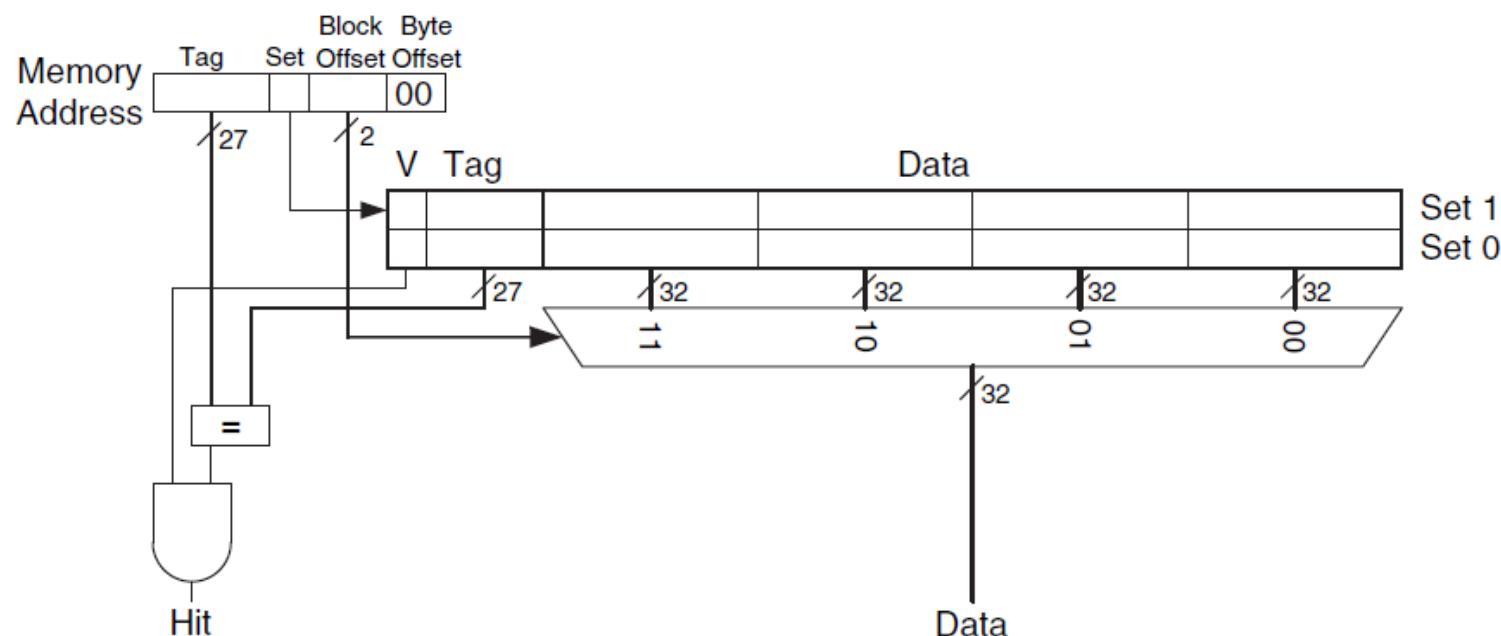
- How do we know whether the data in a cache location is valid?
 - We need a **valid bit to each cache location** (`valid=1; invalid=0`)

Direct Mapped Cache

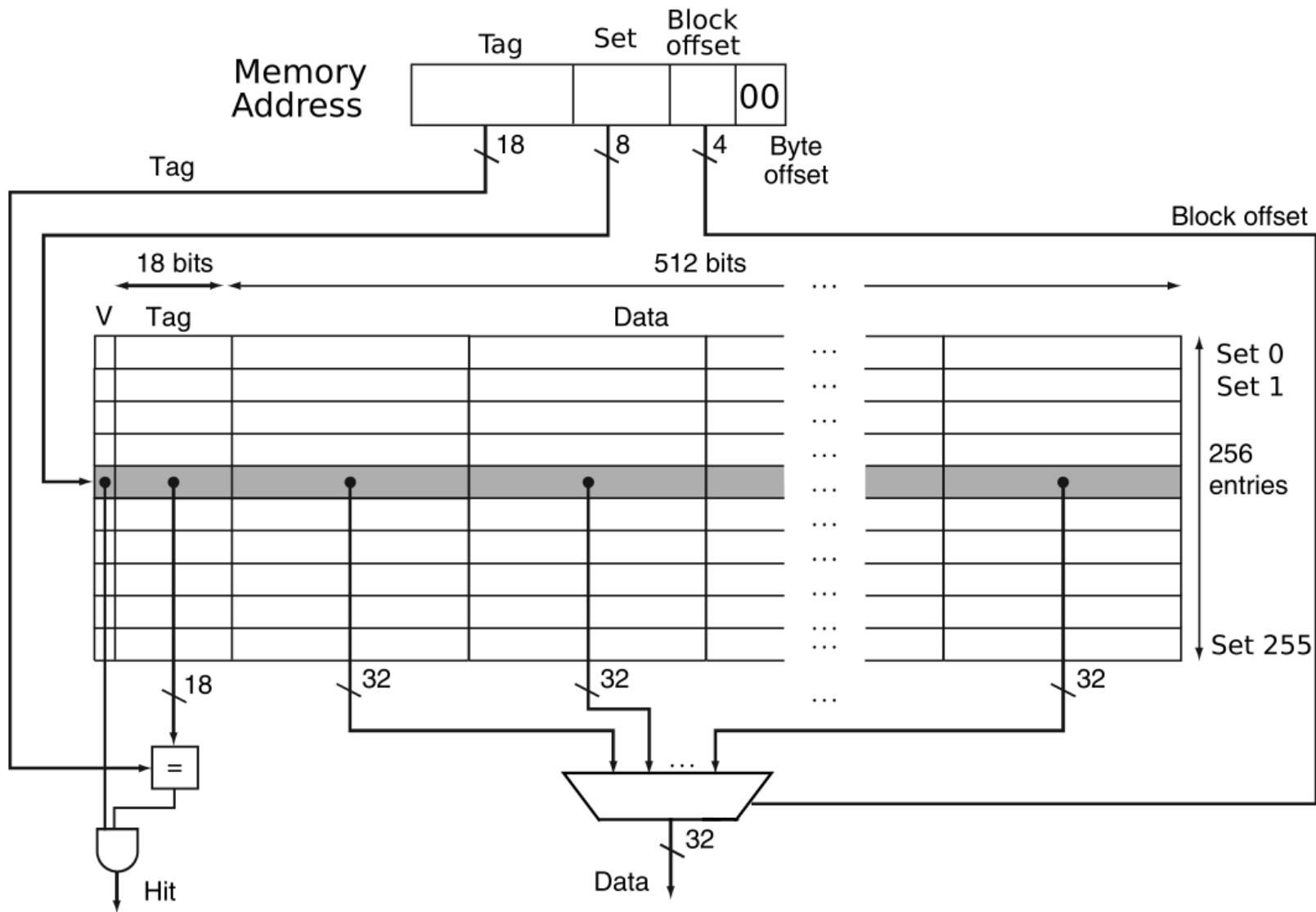


Direct Mapped Cache with $b>1$

- To take advantage of *spatial locality*, a cache uses larger blocks containing several consecutive words.
- Let's consider $C=8$ (as before) and $b = 4$
 - $B = C/b = 2$ (therefore $S=2$)



256x64 Byte Blocks Direct Mapped Cache



Example1

- Let's suppose 32bit addresses, $S=B=128$ and $b=8$ cache. Which is the structure of the address?
- The memory word bytes offset takes 2 bits
- To identify the memory word in a block we need $\log_2 8 = 3$ bits (this is the block offset)
- To address all sets, we need $\log_2 128 = 7$ bits
- The remaining bits are for the TAG (20 bits)



Example2

- Considering S=B=128, b=8, which is the cache line and the offset within the block that holds the address 0xFEAC?
- 0xFEAC -> 1111 **1110** **1010** 1100
- (TAG, IDX, OFF) = (1111, **1110101**, **01100**)
- The cache line has index 117, i.e., the 118th entry
- The offset in the cache block is 3, i.e., the 4th memory word.

TAG	111	110	101	100	011	010	001	000
1111	0xFEBC	0xFEB8	0xFEB4	0xFEBO	0xFEAC	0FEA8	0FEA4	0FEA0

Example3

- Let's consider the following snippet of C code

```
// A,B,C already allocated and A,B initialized  
for(i=0;i<16;i++) C[i] = A[i]+B[i];
```

- Consider a processor running @2GHz with a *direct-mapped L1 data cache* with C=128, b=8; $t_M = 100$ cycles and $t_{L1} = 4$ cycles
 - 'A' starts at address 0x00000000, 'B' starts at address 0x00000040, and 'C' starts at address 0x00000080
 - After t_M cycles all b words have been loaded into the cache block
- Compute the number of cache faults (cache misses) and the AMAT *considering only load instructions.*

Example3 (cont)

```
// A,B,C already allocated and A,B initialized  
for(i=0;i<16;i++) C[i] = A[i]+B[i];
```

- Since A starts at memory address 0, the first 8 words of A will be loaded in the Set0 (000) while the second 8 words in the Set1 (001) of the L1 cache.
- Since B starts at memory address 64, the first 8 words of B will be loaded in the Set2 (010) while the second 8 words Set3 (011) of the L1 cache.
- Therefore, there are no conflict on the sets, and in total we have 4 misses (2 for the two cache lines containing the 16 words of A, and 2 for B) and 32 cache hits (16+16)
- In general, if there is not temporal locality (reuse) the number of misses is $\frac{N}{b}$ where N is the number of load instructions (in this example $N = 2 * 16$)
- $AMAT$ is defined as $HitTime + MissRate * MissPenalty$, in our case the $MissRate = \frac{4}{32}$, thus $AMAT = \left(4 + \frac{4}{32} * 100\right) * ClockCycleTime$, where $ClockCycleTime = \frac{1}{2GHz} = 0.5ns$, $AMAT = 8.25ns$

Direct Mapped Cache pros and cons

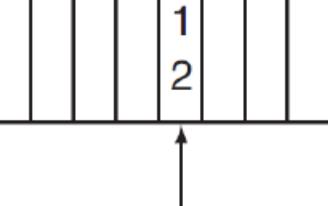
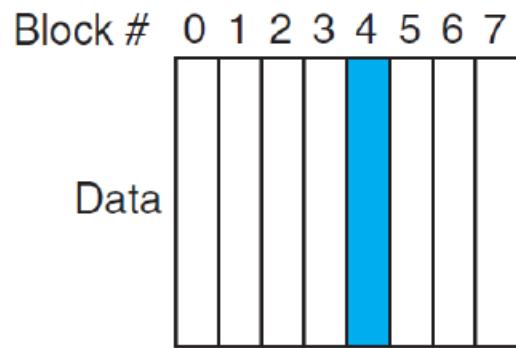
- **Pros:**
 - Simple to realize
 - Very fast in case of cache hits
- **Cons:**
 - Too rigid, a given memory word can be placed in only one single entry of the cache
 - The substitution of a cache line does not consider *temporal locality*
 - This rigidity *may have a big impact on the number of cache conflicts (**thrashing**)* depending on the memory placement and usage of the data structures
 - E.g., suppose A and B of the previous example mapped in the same set

Associative Caches

- **Fully associative cache**
 - Blocks can be placed in any entry in the cache
 - To find a given block, it requires to search in all entries (in parallel)
 - Each entry has a comparator, thus the HW cost significantly increases
- **N-way set-associative cache**
 - Blocks can be placed in a fixed number of N entries (called Set)
 - Each block address is mapped *to exactly one set* (as in the direct mapped cache where $\#S=\#B$), which contains N entries
 - A block can be placed in any entry of the set (as in the fully associative cache)
 - To find a given block, the search is made in parallel in the N entries of the set (using N distinct comparators)

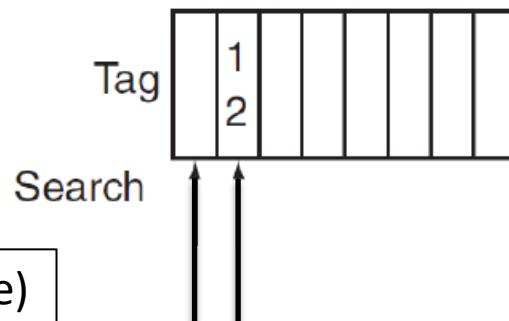
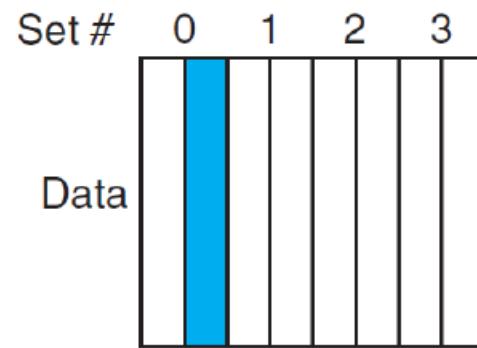
Associative Caches

Direct mapped



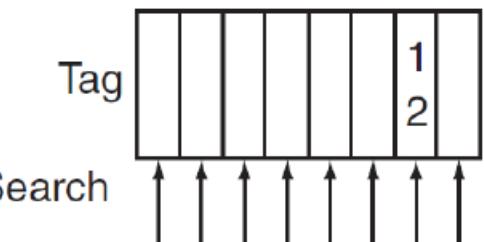
(block address) % (#blocks in cache)
12 % 8 = 4

Set associative

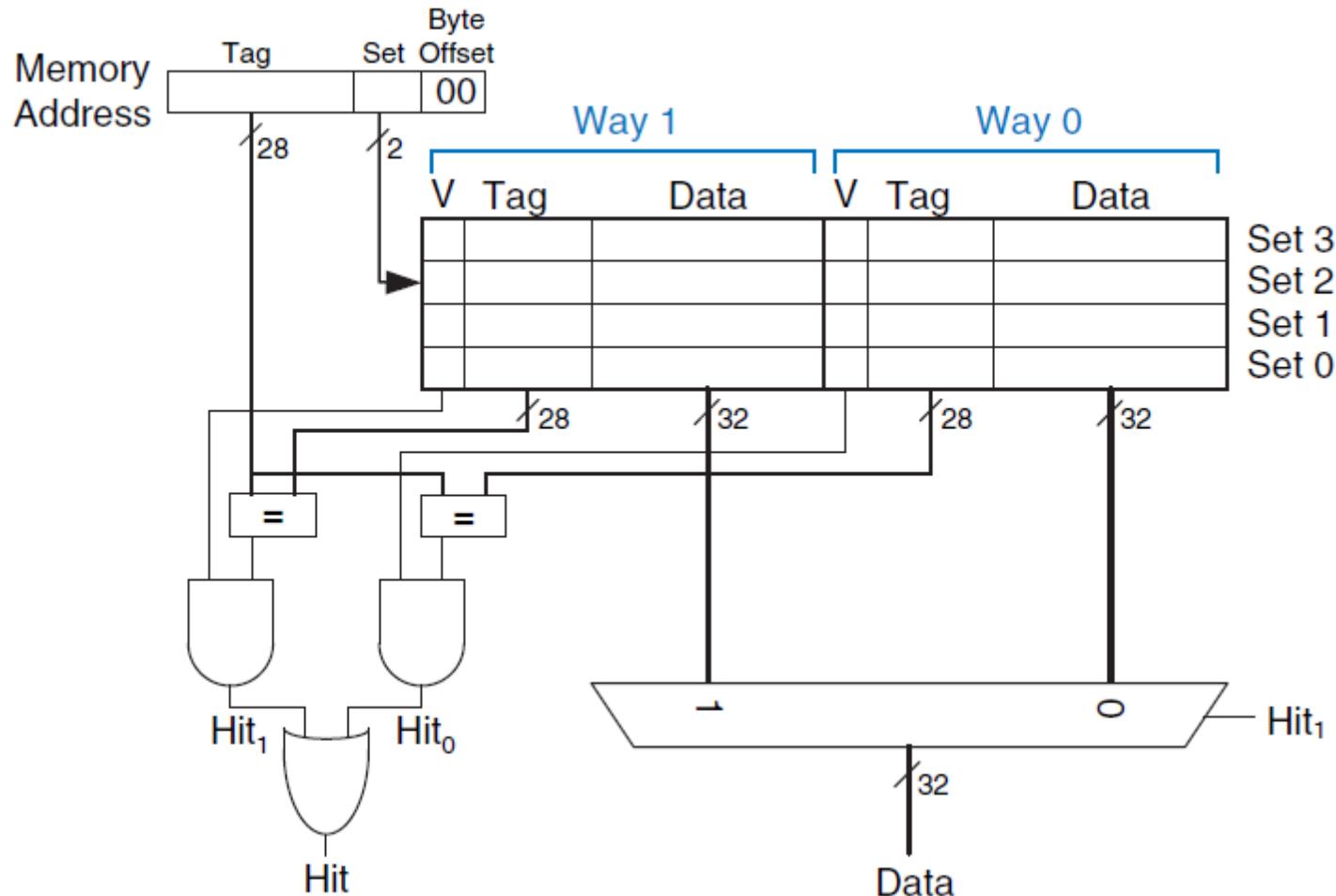


(block address) % (#sets in cache)
12 % 4 = 0

Fully associative

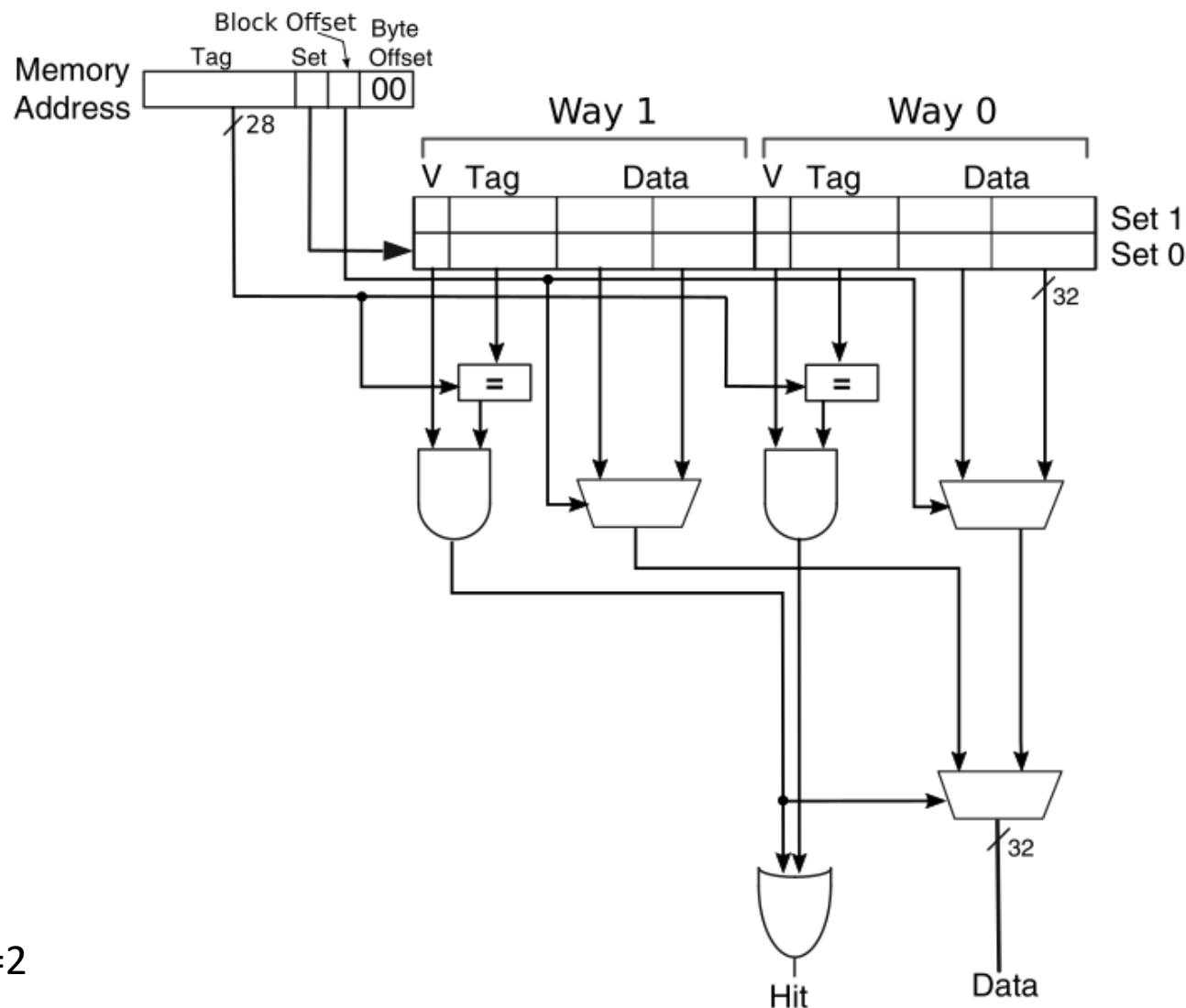


2-way Set-Associative Cache (b=1)



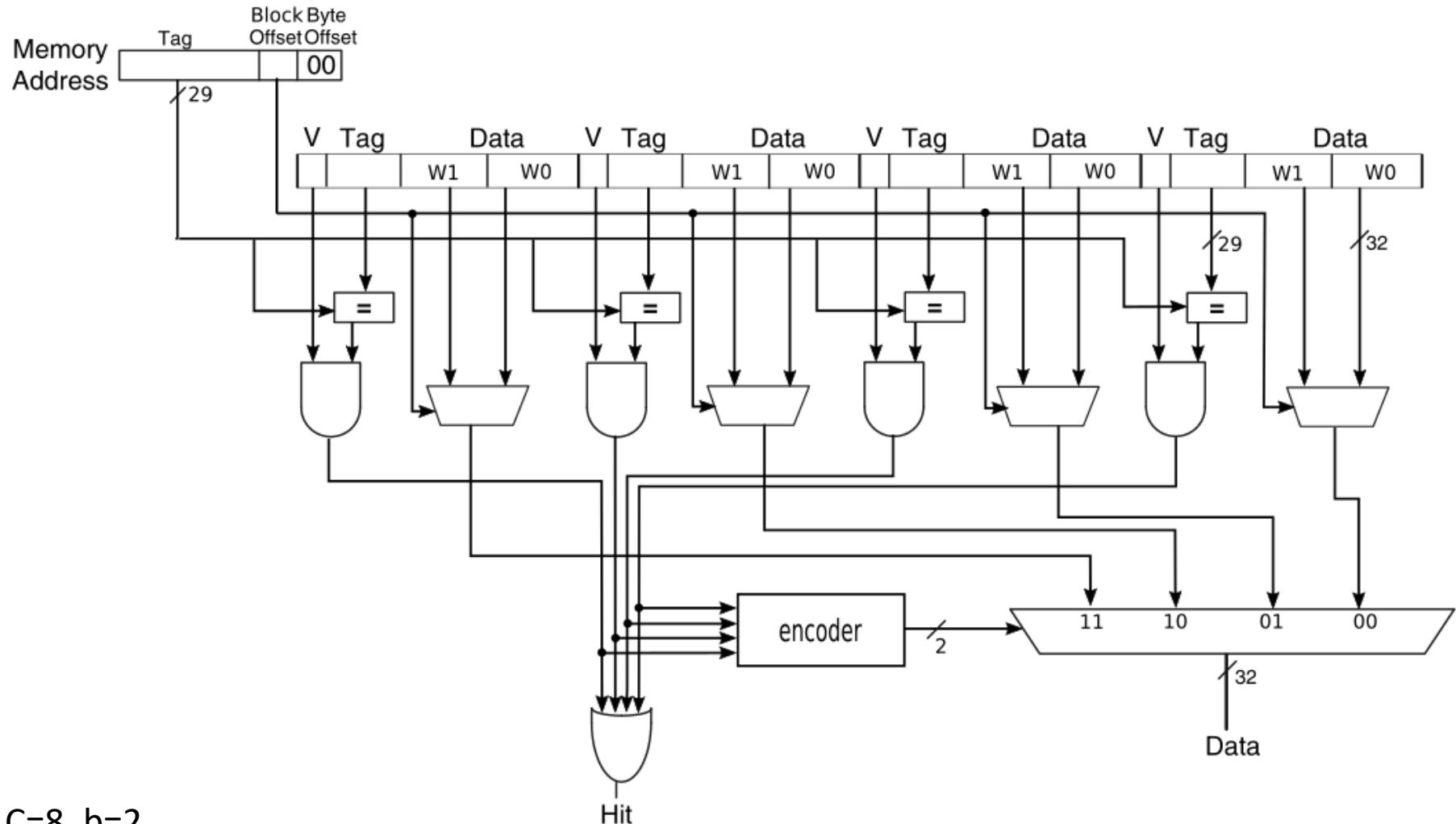
C=8, S=4, b=1

2-way Set-Associative Cache (b=2)



C=8, S=4, b=2

Fully Associative Cache ($b=2$)



$C=8, b=2$

Cache organization summary

- For a given capacity C , we must decide the block size b , the number of blocks B (i.e., cache lines, C/b) and the number of blocks in a Set (N)

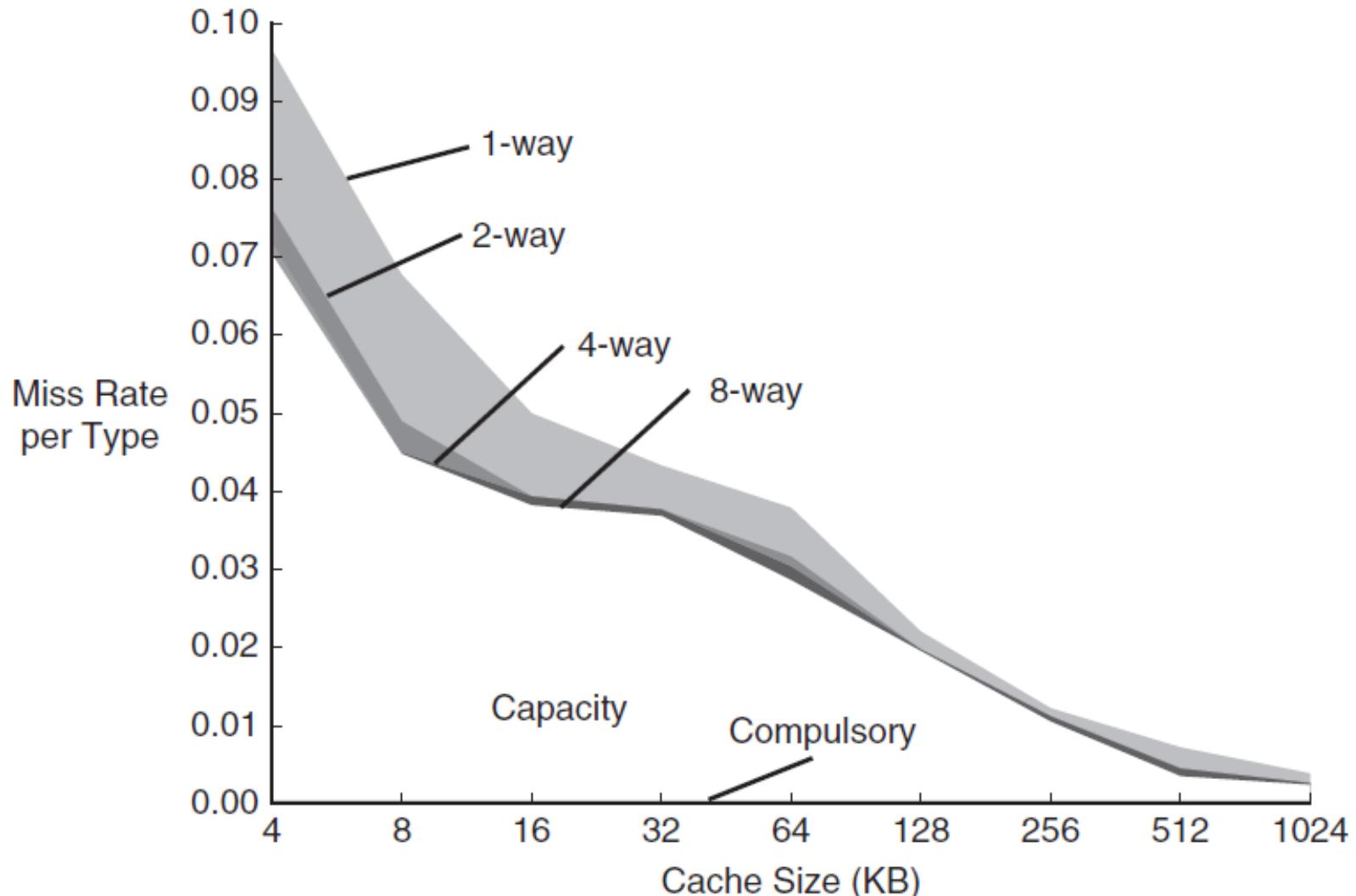
Organization	Number of Ways (N)	Number of Sets (S)
Direct Mapped	1	B
Set Associative	$1 < N < B$	B/N
Fully Associative	B	1

- Cache misses can be reduced by increasing the capacity C , the block size b and the associativity N
 - We are interested in to evaluate the impact of b and N **keeping C fixed**

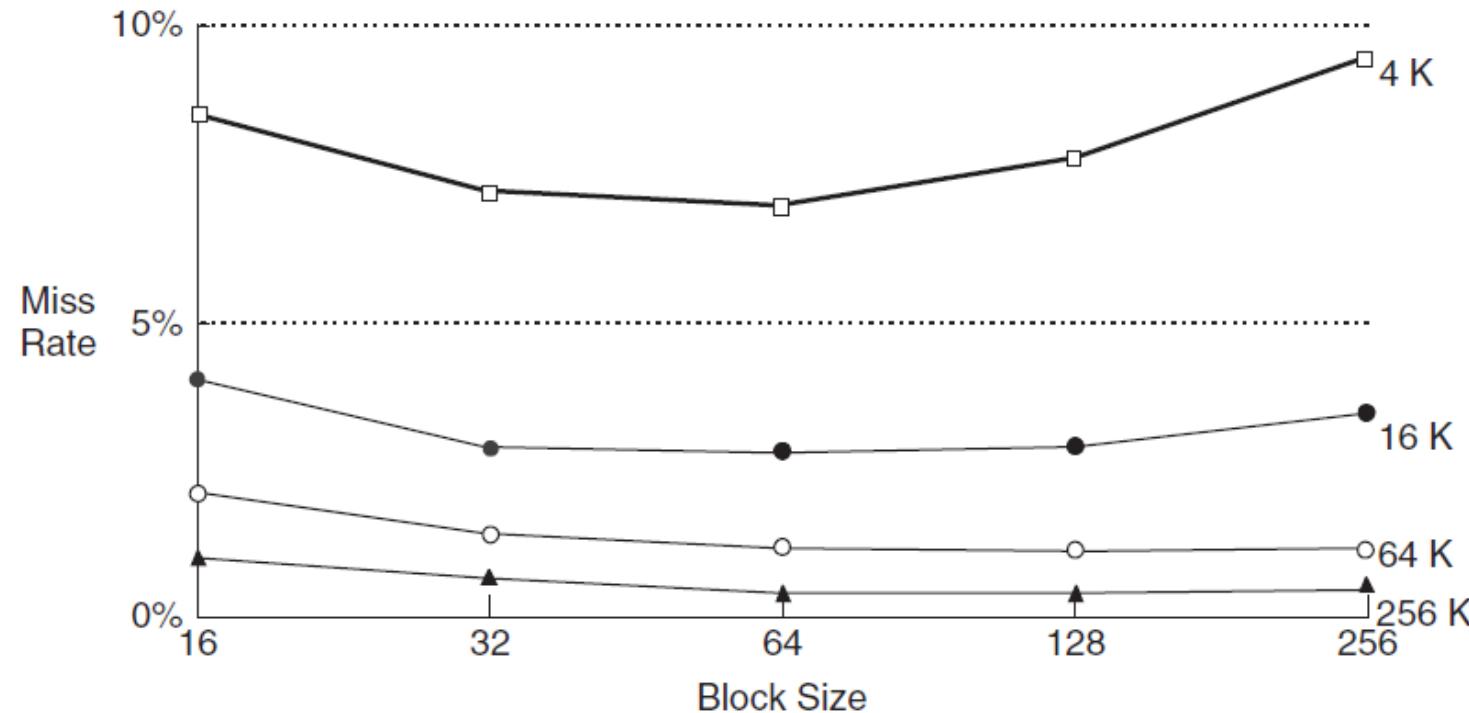
Kinds of cache misses

- **Compulsory misses**
 - These are cache misses caused by the first access to a block that has never been in the cache.
- **Capacity misses**
 - These are cache misses caused when the cache cannot contain all the blocks needed. Some blocks are evicted and later retrieved.
- **Conflict misses**
 - Only for direct mapped and set-associative caches
 - These are cache misses that are eliminated in a fully associative cache of the same size

Miss rate vs cache size and associativity



Miss rate vs block size and cache size



- As block size increases, there are more opportunity to exploit spatial locality, but
 - The miss penalty increases (more time to load a block)
 - Increases the probability of conflict misses because the number of sets decreases

Reducing Miss Rate Summary

- Increase the block size
 - Improve spatial locality but increase the miss penalty
- Increase the associativity
 - Less conflicts but higher hit time
- Increase the case size
 - Less capacity miss and conflicts but higher hit time
- Cache-oblivious algorithm (not yet examined)
 - Impact on the complexity of compilers
 - Impact on programmers

Handling cache misses

- A cache miss *stalls the entire processor*, freezing the contents of all registers while waiting for memory
 - Indeed, more advanced processor allows ***out-of-order execution*** of other instructions while waiting for memory
- The steps taken for a *cache miss* are (***fault management***):
 1. Tell the next memory level to *read* the missing value
 2. Wait for the memory to respond (this can take multiple cycles)
 3. Update the corresponding cache line with the data received
 4. Restart the instruction execution (now it is a cache hit)

Handling writes

- **Write hits (same TAG)**
 - If the store instruction writes the data only into the cache, then cache and memory would have different values (cache and memory are **inconsistent** – the main memory contains stale data)
 - We need to define a strategy for write hits; two options:
 - 1) **Write-Through**
 - 2) **Write-Back**
- **Write misses (different TAG)**
 - Should we fetch the block from memory to cache and then overwrite the missing word?
 - This policy is called **write-allocate**. The block is loaded into the cache followed by a write hit (mainly used in the **Write-Back** policy)
 - Should we write the word directly to the next memory level?
 - This policy is called **no-write-allocate**, the block is not loaded into the cache (mainly used in the **Write-Through** policy)

Write-Through

- Write hits always update both the cache and the next memory level
- Pros
 - Simple solution, easy to implement
 - Data is ***always consistent*** between the two memory levels
- Cons
 - The speed of writes depends on the write speed of the lower memory level
 - Higher memory traffic, for every single write there may be multiples writes in each memory level

Write-Back

- Write hits update only the cache, then the modified block is written to the lower memory level when it is replaced
- Pros
 - The speed of writes is that of the cache
 - Lower memory traffic w.r.t. Write-Through, subsequent writes to the same cache block do not produce traffic with the lower memory level
- Cons
 - We need to keep track of modified blocks (**dirty bit**)
 - More complex to implement than Write-Through
 - The cache line replacement is more costly

Write-Through vs Write-Back example

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

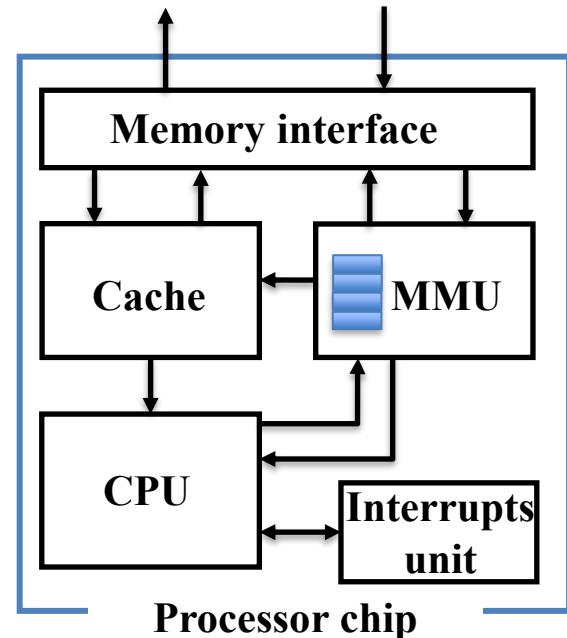
Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
MOV R5, #0  
STR R1, [R5]  
STR R2, [R5, #12]  
STR R3, [R5, #8]  
STR R4, [R5, #4]
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

Optimizing writes

- Since writing to off-chip memory is costly, memory stores are buffered
- A **write buffer** is used to hold the data waiting to be written to memory
- Execution continues immediately after writing the data into the cache **and** into the write buffer
 - Memory stores to main memory are executed in parallel with the CPU computation
- The CPU stalls only if the write buffer is full, therefore the main memory requested bandwidth is a critical factor, *particularly for the Write-Through cache model!*



Write-Back Considerations

- Write-Back can improve performance especially when the CPU generates store instructions faster than what the main memory can handle
- However, the cost of cache writes is higher if a write miss occurs, we first must write the block back to memory (if the dirty bit is 1).
 - This requires at least two cycles even for a write hit: a cycle to check for a hit followed by a cycle to actually perform the write
 - Alternatively, we may use a **write buffer** to temporarily hold the data to write while the cache block is checked for a hit. The processor does the cache lookup and places the data in the write buffer during the normal cache access cycle. Assuming a cache hit, the new data is written from the write buffer into the cache on the next unused cache access cycle (*pipelining of accesses*)

Cache Replacement

- In a *direct mapped* cache, the requested block can go in exactly one position, thus we have no choice
 - If the block to be replaced has been modified and the write policy is Write-Back, we must update the lower-level memory
- In an associative cache, we have a choice of where to place the requested block
 - *Fully associative* cache, all blocks are candidates for replacement
 - *N-way set-associative* cache, we must choose among the N blocks in the selected set

Cache Replacement Policy

- Which replacement policy to adopt?
- The most used scheme is **Least Recently Used (LRU)**
 - It considers temporal locality, the block replaced is the one that has been *unused* for the longest time
 - For a 2-way set-associative cache, it can be implemented with 1 bit (*use bit -- U*), for a 4-way is still doable with 2 bits, for more than 4-ways it becomes quite complicated (*pseudo-LRU*)
- For high associative caches, a **random** policy gives approximately the same performance as LRU

(*) We will discuss cache replacement policies in more details when we will study virtual memory

Example

- Consider a small cache with 4 blocks, $b=1$. Find the number of misses for a **direct-mapped** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8.

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Example

- Consider a small cache with 4 blocks, $b=1$. Find the number of misses for a **2-way set-associative** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8 (**LRU replacement policy**).

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Example

- Consider a small cache with 4 blocks, $b=1$. Find the number of misses for a **fully associative** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8.

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Improving processor performance

- Let's consider again the [cache performance example](#)
- What happens if the processor is made faster with the same memory system?
- Suppose we speed up the clock rate by a factor of 2 (e.g., 2 → 4 GHz)

$$CPI_{Stall-Instr} = 1 * 0.02 * 200 = 4$$

$$CPI_{Stall-Data} = 0.36 * 0.04 * 200 = 2.88$$

$$CPI_{Stall.} = 4 + 2.88 = 6.88$$

$$CPI = 2 + 6.88 = 8.88$$

- The fraction of time spent on memory stalls rises from 63% to 77%
 - $3.44/5.44 = 0.63 \rightarrow 6.88/8.88 = 0.77$
- The execution time improves of a factor 1.23:

$$\frac{CPU_{time1}}{CPU_{time2}} = \frac{IC * CPI_{Stall1} * ClockCycleTime}{IC * CPI_{Stall2} * \frac{ClockCycleTime}{2}} = \frac{\frac{5.44}{8.88}}{\frac{2}{2}} = 1.23$$

Improving processor performance

- The same happens if we improve the processor architectures by a factor of 2 (i.e., the CPI decrease from 2 to 1)
 - for example, because we improved the CPU pipeline organization
- In both situations (i.e., increasing clock rate and decreasing CPI), the time spent on memory stalls takes an increasing fraction of the total execution time, which produces a reduced impact on the overall performance (1.23) compared to the speed up factor of 2
- In other words, the relative efficiency of these optimizations is low:

$$\varepsilon = \frac{1.23}{2} = 61\%$$

- To face this issue almost all modern systems, support *multiple levels of caching*, which *helps reducing the miss penalty factor*

Multi-level caches in modern processors

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Multi-level caches

- The second-level (L2) cache is accessed whenever a miss occurs in the first-level (L1) cache.
- If the L2 cache contains the desired data, the miss penalty for the L1 cache will be essentially the access time of the L2 cache, which will be much less than the access time of main memory
 - different technologies and main memory is off-chip.
- The same happens for a third-level (L3) cache, if it exists.
- Only if L1, L2 and L3 do not contain the data, a main memory access is required, and a larger miss penalty is paid

Multi-level cache and *AMAT*

- Quantitative example:
 - $t_M = 50\text{ns}$ (main memory service time), $t_{L1} = 1\text{ns}$ $t_{L2} = 6\text{ns}$ $t_{L3} = 10\text{ns}$
 - Miss rates: 10%, 1.5% and 0.4% for L1,L2, and L3, respectively
 - 1. No cache: $AMAT= 50\text{ns}$
 - 2. Only L1 cache: $AMAT= 1+0.1*50= 6\text{ns}$
 - 3. L1 and L2 caches: $AMAT= 1+0.1*(6+0.015*50)= 1.675\text{ns}$
 - 4. L1, L2 and L3 caches: $AMAT= 1+0.1*(6+0.015*(10+0.004*50))= 1.6153\text{ns}$

Multi-level cache and CPI

- Remember that

$$CPI_{Stall} = \text{Miss rate memory instr.} * \text{Miss penalty}$$

- With multilevel caches, memory-stall clock cycles can be defined as the sum of the stall cycles coming from all cache levels
 - For simplicity let's assume load/store miss rates and penalties are the same

$$\begin{aligned} CPI_{Stall} = & \text{MissRate}_{L1} * \text{MissPenalty}_{L1} \\ & + \text{GlobalMissRate}_{L2} * \text{MissPenalty}_{L2} \\ & + \text{GlobalMissRate}_{L3} * \text{MissPenalty}_{L3} + \dots \end{aligned}$$

where, for a given level LN (N>1):

$$\text{GlobalMissRate}_{LN} = \text{MissRate}_{L1} * \text{MissRate}_{L2} * \dots * \text{MissRate}_{LN}$$

Multi-level cache and CPI

- Suppose a 2-level cache system with:
 - $\text{MissRate}_{L1} = 2\%$, $\text{MissRate}_{L2} = 20\%$
 - L2 cache access time 20 cycles, main memory access time 200 cyclescompute the $\text{GlobalMissRate}_{L2}$ and the $\text{CPI}_{\text{Stall}}$.

$$\text{GlobalMissRate}_{L2} = \text{MissRate}_{L1} * \text{MissRate}_{L2} = 0.02 * 0.2 = 0.04 \text{ (4%)}$$

A miss in the first-level cache can be satisfied either by the secondary cache or by the main memory. The miss penalty is 20 cycles if we have an hit in L2, and 200 cycles if we have a miss in L2, thus:

$$\text{MissPenalty} = 20 + 0.2 * 200 = 60 \text{ cycles}$$

Therefore, $\text{CPI}_{\text{Stall}} = \text{MissRate}_{L1} * \text{MissPenalty} = 0.02 * 60 = 1.2 \text{ cycles}$

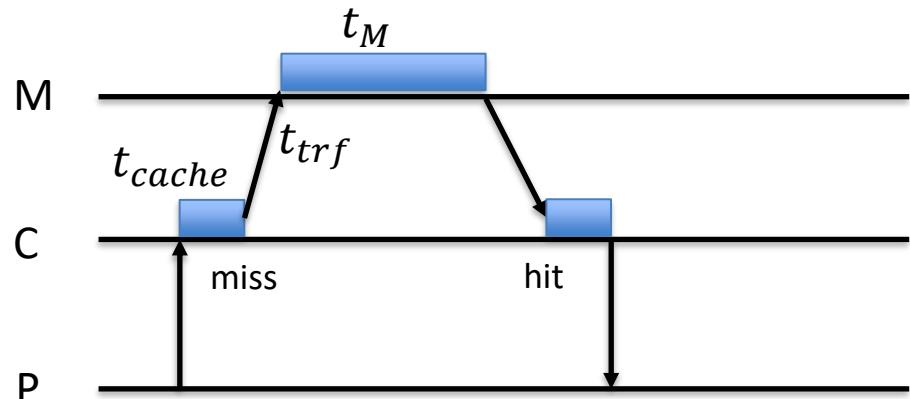
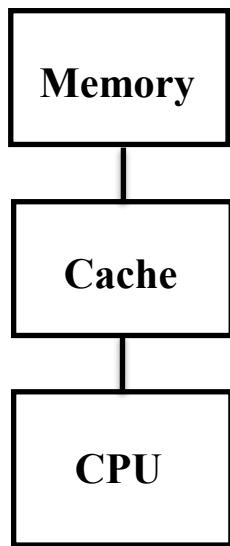
Similarly, by considering $\text{MissPenalty}_{L1} = 20$ and $\text{MissPenalty}_{L2} = 200$ and using the previous formula (previous slide), we have:

$$\text{CPI}_{\text{Stall}} = 0.02 * 20 + (0.02 * 0.2) * 200 = 1.2 \text{ cycles}$$

Designing the Memory System

- The miss penalty can be reduced increasing the bus bandwidth between DRAM and cache
- Three possible organizations:
 - **Simple**: one-word at a time is read from memory
 - **Wide-memory**: N words at a time are read from memory
 - **Interleaved**: K independent memory banks capable to serve K requests simultaneously
- Let's consider the cache block transfer time for *Simple* vs *Interleaved* memory organization

Cache blocks transfer

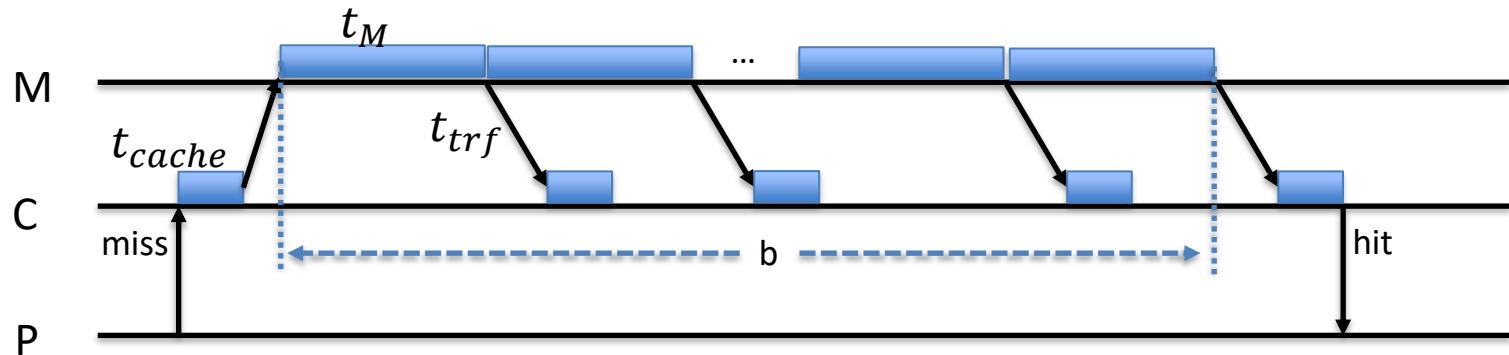


- Suppose a single main memory module
- The *Memory* and *Cache* service time is t_M , t_{cache} , respectively
- The offered memory bandwidth is $B_M = \frac{1}{t_M}$ and b=1
- The memory access time as seen by the CPU is about
$$t_a = 2 t_{trf} + 2 t_{cache} + t_M$$

Cache blocks transfer

- If the cache line holds $b > 1$ memory words, the cost to transfer the entire block in case of a cache miss is (*cache fault management*):

$$t_{miss} = 2t_{trf} + 2t_{cache} + b t_M \approx b t_M$$

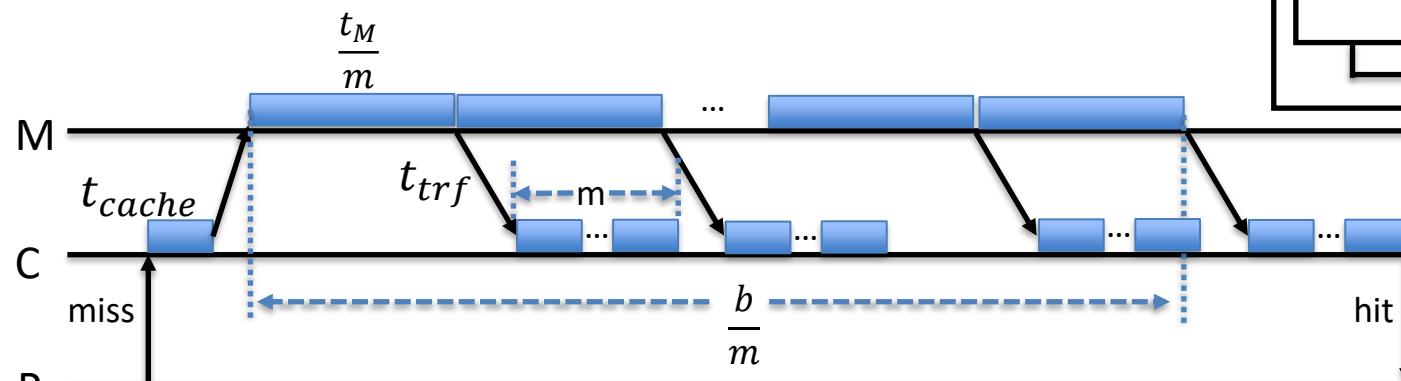


- Considering $b = 8$ and $t_M = 80$, $t_{cache} = 1$ and $t_{trf} = 6$ clock cycles, the cost of the cache miss due to the transferring of all data in the block is very high (>650 clock cycles)
 - The CPU stalls for several hundred cycles!

Cache blocks transfer

- By using modular memory (*memory interleaving*) of m modules we increase the offered bandwidth by a factor of m :

$$B_M = \frac{m}{t_M}$$



$$t_{miss} = 2t_{trf} + (m + 1) t_{cache} + \frac{b}{m} t_M$$

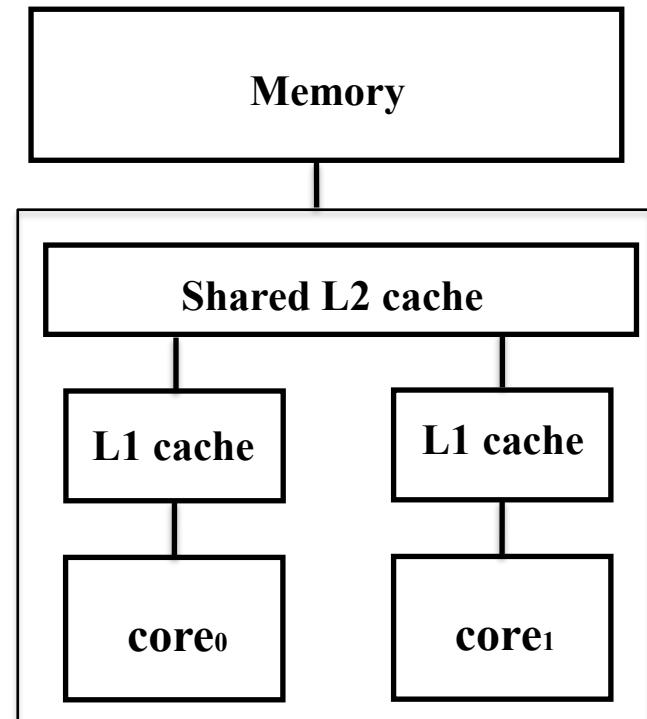
- If we set $m = b$ we have again $t_{miss} \approx t_M$

Cache issues

- Caching is essential to reduce the von Neuman bottleneck and achieve reasonable performance on modern systems
- However, caching introduces some problems in ***multiprocessor systems***
 - **Cache coherence problem**
 - *False sharing* performance degradation in cache-coherent multiprocessors
 - Two unrelated variables are placed in the same cache block and accessed in read/write mode by different threads!

Cache Coherence Problem (brief note)

- Let's suppose a SMP architecture
 - SMP: Symmetric Multiprocessors
- Caching of both private and shared data
- Private core data is cached in L1, thus reducing the AMAT as well as off-chip memory communications.
- When shared data are cached, the shared values may be replicated in multiple private core caches.
 - This also reduce memory contention!
- What happens if shared data are written?

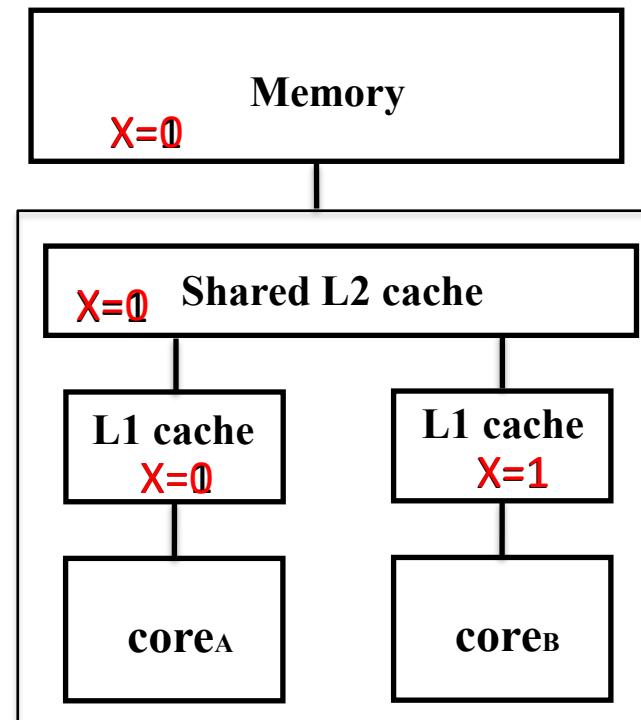


Caching of shared data introduces a new problem: **cache coherence**

Cache Coherence Problem (brief note)

Write-through caches

Time	Event	L1-core1	L1-core2	M
0				X=1
1	A reads X	X=1		X=1
2	B reads X	X=1	X=1	X=1
3	A writes X	X=0	X=1	X=0



Write invalidate protocol (brief note)

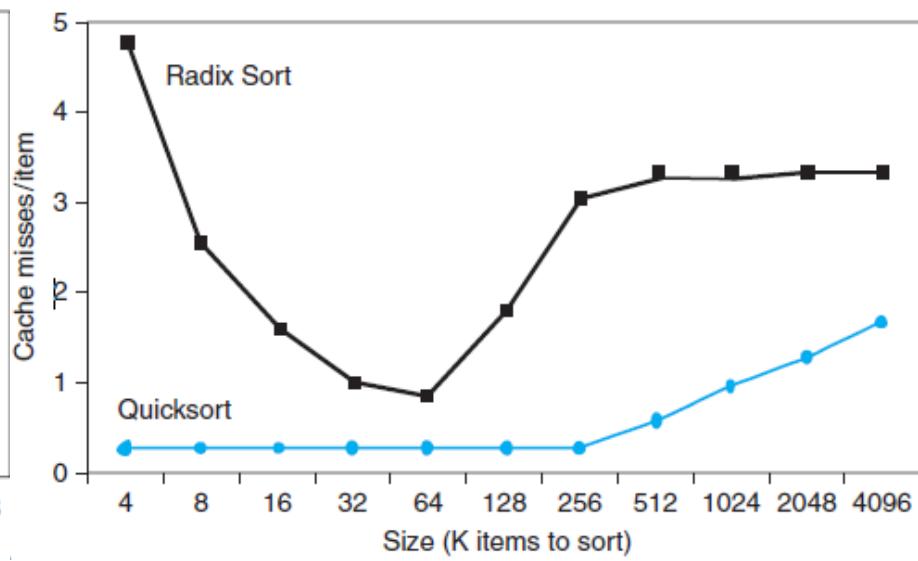
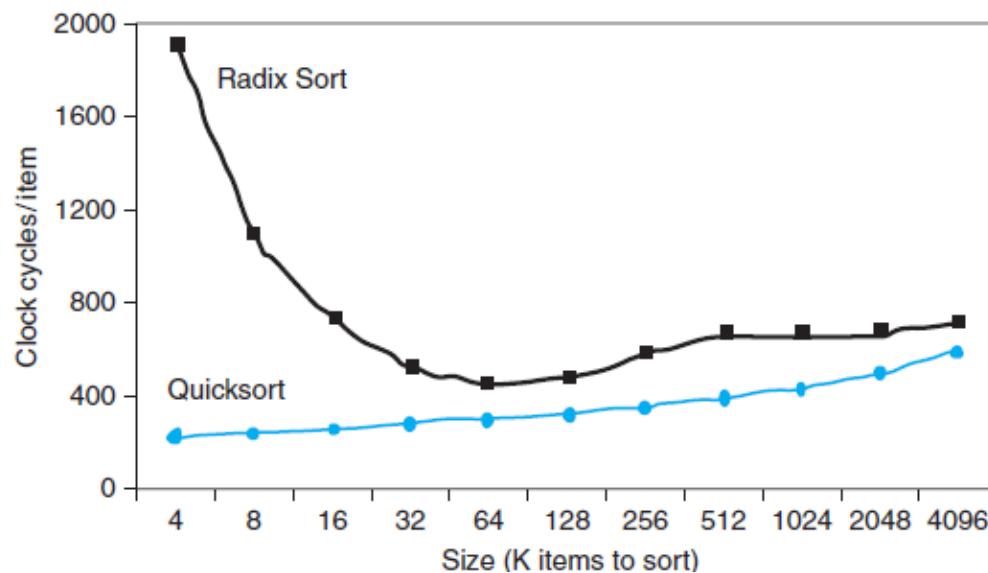
- The *write invalidate protocol* invalidates copies of the data present in other caches on a write operation
- A simple implementation uses a *snooping bus protocol* for ensuring the processor acquires exclusive access to a cache block before it writes into it

Write-back caches

Event	Bus activity	L1-core1	L1-core2	M
				X=1
A reads X	Miss for X	X=1		X=1
B reads X	Miss for X	X=1	X=1	X=1
A writes X	Invalidation for X	X=0	not valid	X=1
B reads X	Miss for X	X=0	X=0	X=0

Cache Interaction with Software

- Cache misses depend on memory access patterns
 - Algorithm behavior
 - Compiler/Programmer **optimizations** for memory access



- *False sharing* can be another important issues in multiprocessors with automatic cache-coherence

Software Optimizations

- **Goal:** maximize cached data reuse by enhancing temporal and/or spatial locality
- Two well-known techniques to reduce data miss rate are:
loop interchange and **data blocking**
- Example: loop interchange (for row-major ordering – C)

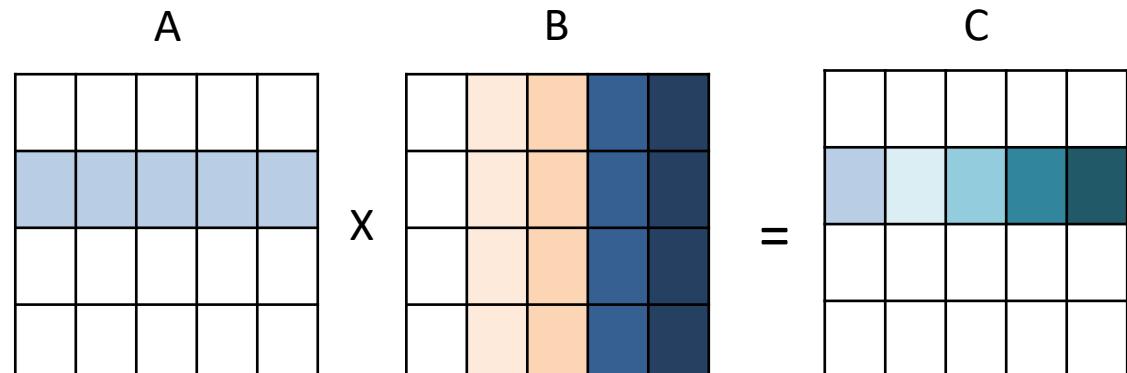
Before	After
<pre>for (j=0;j<100;++j) for(i=0;i<1000;++i) A[i][j] = 2*A[i][j];</pre>	
	<pre>for (i=0; i<1000; ++i) for(j=0; j<100; ++j) A[i][j] = 2*A[i][j];</pre>

Software Optimization via blocking

- Loop interchange improves spatial locality
- Blocking improves temporal locality
- Example: Matrix Multiplication (GEMM algorithm)

Before (textbook algorithm)

```
for (i=0;j<N;++i)
  for(j=0;j<N;++j) {
    c = 0;
    for(k=0;k<N;++k)
      c+= A[i][k] * B[k][j];
    C[i][j] = c;
  }
```

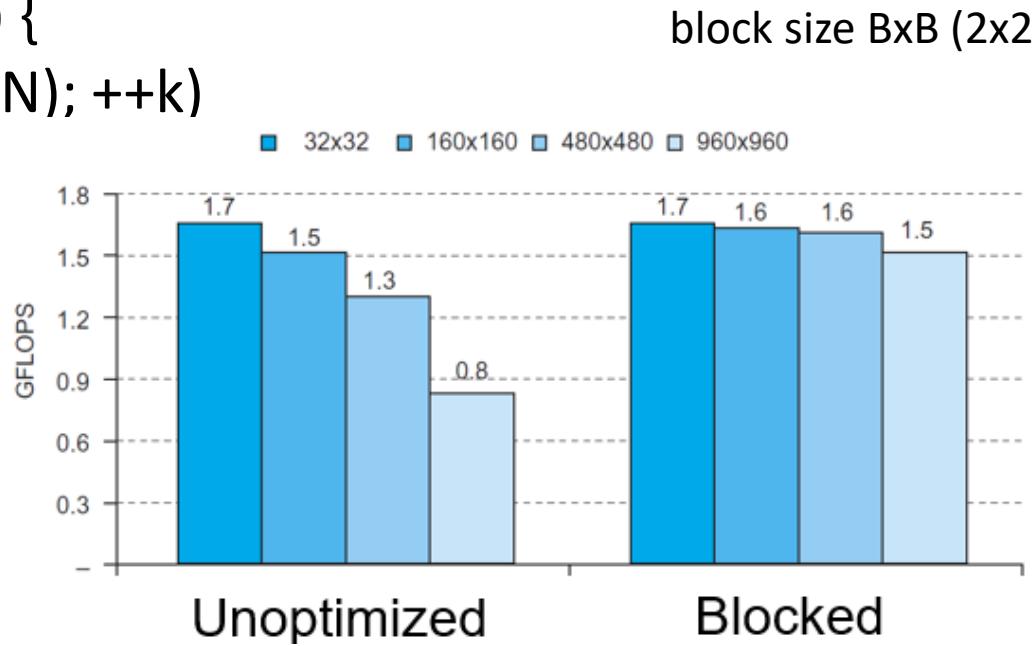
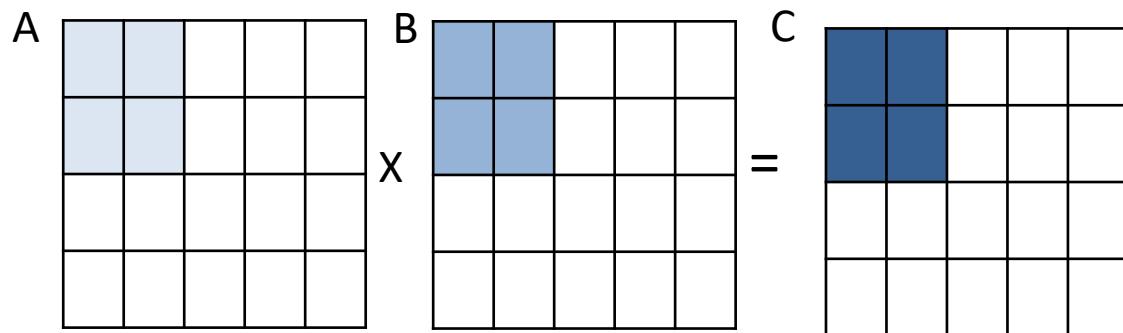


Software Optimization via blocking

- Blocking version of GEMM with block size B (*block factor*)

After

```
for (jj=0; jj<N; jj+=B)
    for(kk=0; kk<N; kk+=B)
        for(i=0;i<N;++i)
            for(j=jj; j < min(jj+B,N); ++j) {
                for(c=0,k=kk; k<min(kk+B,N); ++k)
                    c+= A[i][k] * B[k][j];
                C[i][j] += c;
            }
```



Input Output

Main Points

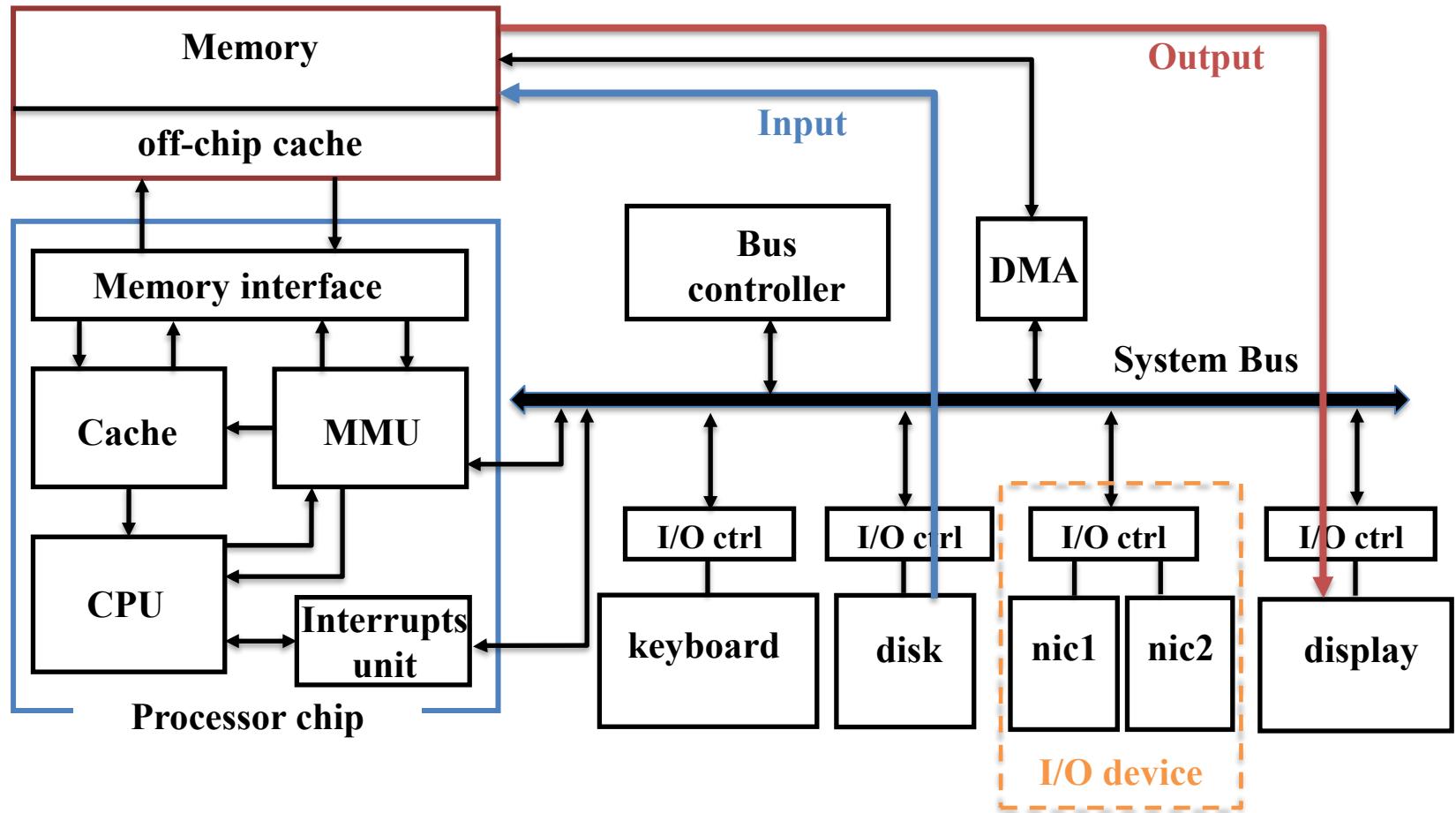
- I/O devices
- Device controller
- Buses
- I/O management
- Device drivers

Credits: some figures in these slides come from “Computer Organization and Architecture” by William Stallings, 10th edition.

Some I/O devices

Device	Behavior	Partner	Data rate (Mbit/sec)
Keyboard	Input	Human	0.0001
Mouse	Input	Human	0.0038
Voice input	Input	Human	0.2640
Sound input	Input	Machine	3.0000
Scanner	Input	Human	3.2000
Voice output	Output	Human	0.2640
Sound output	Output	Human	8.0000
Laser printer	Output	Human	3.2000
Graphics display	Output	Human	800.0000–8000.0000
Cable modem	Input or output	Machine	0.1280–6.0000
Network/LAN	Input or output	Machine	100.0000–10000.0000
Network/wireless LAN	Input or output	Machine	11.0000–54.0000
Optical disk	Storage	Machine	80.0000–220.0000
Magnetic tape	Storage	Machine	5.0000–120.0000
Flash memory	Storage	Machine	32.0000–200.0000
Magnetic disk	Storage	Machine	800.0000–3000.0000

Input/Output and I/O devices



I/O impact

- The I/O impact on program execution time may be quite high
- Let's suppose $T_{exe} = T_{cpu} + T_{I/O}$ and $T_{I/O} = \frac{1}{10}T_{cpu}$, therefore $T_{exe} = \frac{11}{10}T_{cpu}$
- If we speed up T_{cpu} by 10 times leaving unaltered $T_{I/O}$, we have $T_{cpu}^{enhanced} = \frac{1}{10}T_{cpu}$, thus $T_{exe} = \frac{1}{10}T_{cpu} + \frac{1}{10}T_{cpu} = \frac{1}{5}T_{cpu}$
- Let's consider the $Speedup = \frac{\text{Exec.time before enhancement}}{\text{Exec.time after enhancement}}$
- The speedup obtained is $\frac{11}{2} = 5.5$

An optimization of 10-fold on the T_{cpu} produced only about 5-fold enhancement on T_{exe} ! Why?

Amdahl's law

- Proposed by Gene Amdahl in 1967. It deals with the potential maximum speedup attainable by a parallel program when increasing the number of processors from 1 to N
 - *It can be applied to any optimization process*
- Consider a program in which only the fraction f can be optimized (e.g., parallelized using N processors), whereas the fraction $(1-f)$ remain unaltered (e.g., it is inherently sequential).
- $Speedup = \frac{Exec.\text{time before enhancement}}{Exec.\text{time after enhancement}} = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$

The speedup is bound by the sequential fraction $(1-f)$, i.e., by the part of the process that I cannot (or I'm not able) to enhance!

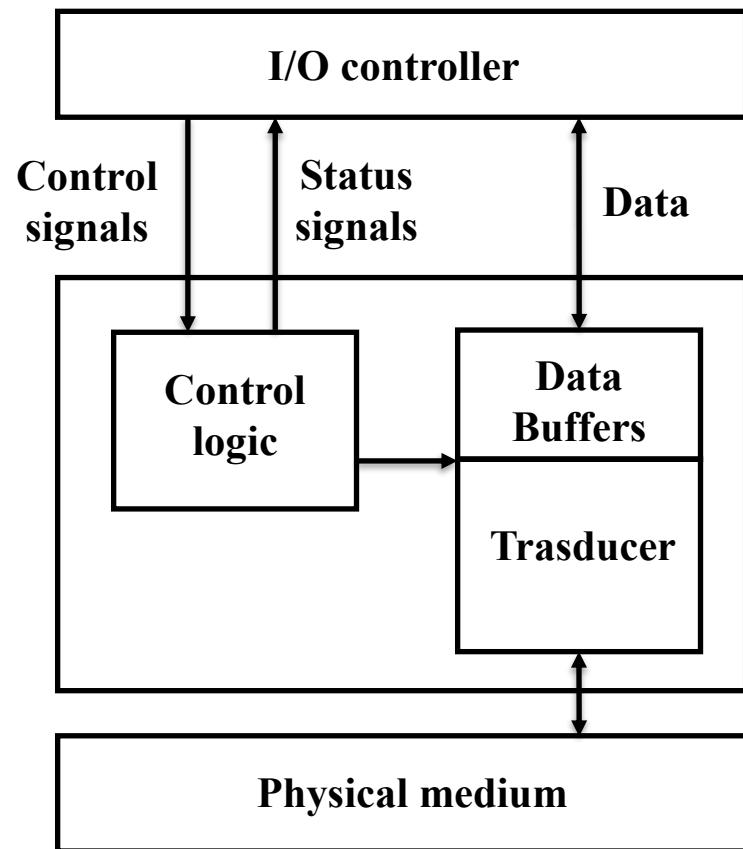
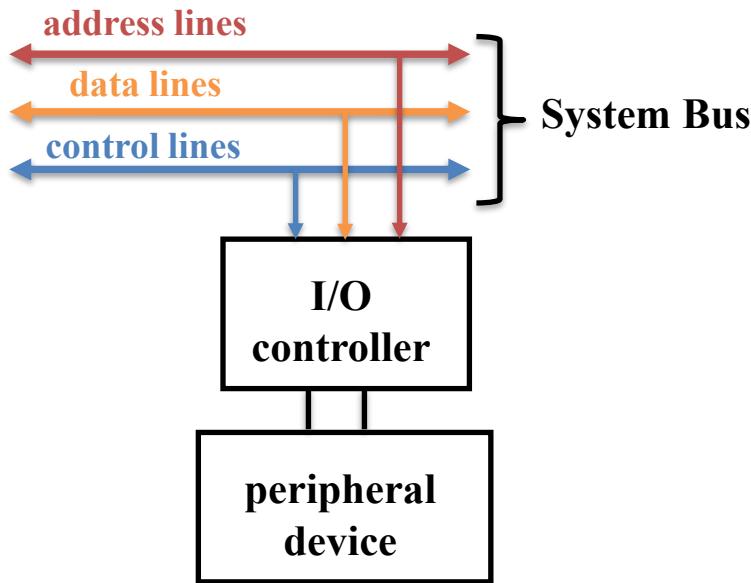
Not only performance

- Performance of the I/O subsystem is very important, but it is not all. Other important aspects are:
 - **Reliability** (affidabilità)
 - Mean Time To Failure (MTTF) metrics
 - Fault avoidance (better components)
 - Fault tolerance (introducing some level of redundancy)
 - **Availability** (disponibilità)
 - Mean Time To Repair (MTTR)
 - $$Availability = \frac{MTTF}{MTTF+MTTR}$$

I/O: control + data

- I/O devices have two ports
 - Control: both commands and status reports
 - How we tell the device what to do
 - How the device tells us about its features
 - How the device tells us about op status
 - Data
 - To/from device memory

Generic I/O device



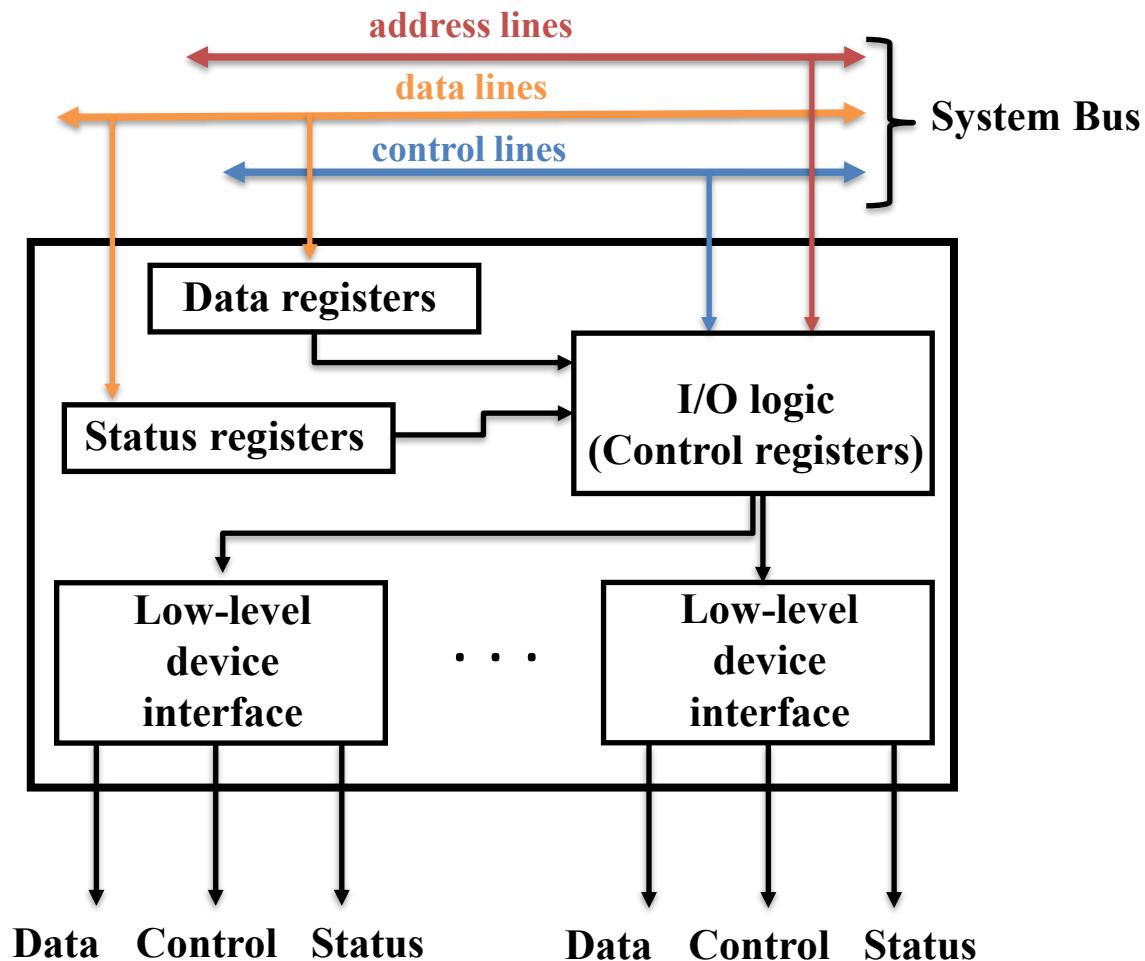
I/O controller functions

- Control and timing
- Processor communication
 - Command decoding (e.g., read sector X)
 - Data exchange
 - Status reporting
 - Address recognition
- Device communication (command, status info, data)
- Data buffering
 - To optimize data transfers
 - To compensate different speeds
- Error detection
 - Transmission errors (parity bits)
 - Electrical/mechanical errors

Logical I/O steps

- Control of the transfer from a device to processor:
 - CPU checks I/O module device status
 - I/O controller returns the status
 - If ready, CPU requests data transfer by means of a command to the controller
 - I/O controller gets data from the peripheral device
 - I/O controller transfers data to processor
- These steps require one or more bus arbitration actions to implement the ***communication protocol***

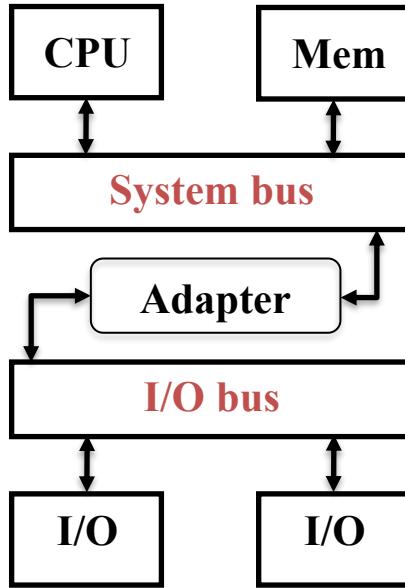
I/O controller diagram



Bus interconnection

- A collection of data lines that is treated together as a single logical signal
 - Also used to indicate a shared collection of lines with multiple connected devices (called *taps*)
 - Performance factors: physical length, number of connected taps
- A bus is a *shared transmission medium*
 - Only one device at a time can successfully transmit
- The **system bus** connects major components (processor, memory, I/O)
 - Hundreds of separate lines
 - Lines classified in three groups: data, address, control

Examples of buses



- **System bus**

Connects processor and memory, I/O interfaced with adapters

- Short, few taps → Fast, high-bandwidth
- Not standard (i.e., system specific)

- **I/O bus**

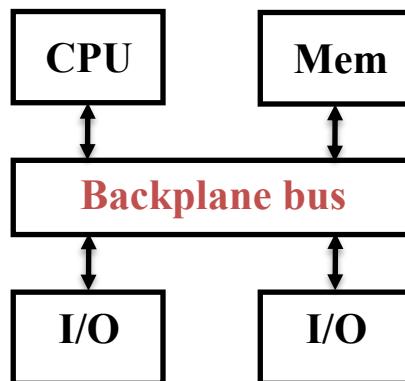
Connects I/O devices, no direct connection with processor and memory

- longer, more taps → slower, lower-bandwidth
- Industry standard (e.g., ATA/SCSI)

- **Backplane bus**

Connects CPU, memory, I/O devices

- Long, many taps → slow, medium/low-bandwidth
- Different devices with different bandwidths
- Industry standard



Bus design

- Goals: high-performance, standardization, low cost
 - System bus emphasizes performance, I/O and Backplain buses emphasizes costs and standardization
- Design issues to address:
 - Are wires shared or separated?
 - Wider buses (i.e., higher bandwidth) are more expensive and more susceptible to skew
 - How is bus control acquired and released?
 - **Atomic transactions:** low utilization, low complexity.
 - **Split-transactions:** Request/replies can be interleaved; this means higher utilization but more complex design (e.g., IDs to identify different requests)
 - Is bus clocked?
 - How do we decide who gets the bus?

Bus clocking

- **Synchronous**

- All devices connected to the bus share the same bus clock. Events happen at the edge of the clock signal
- Possible protocol: at cycle X the I/O unit writes a READ request on the bus; at cycle $X+\Delta$ the unit can read the data from the bus. Δ is the maximum time to WRITE the data on the bus by a connected unit.
- Potential clock skew issue
- Limited to short buses (e.g., system bus)

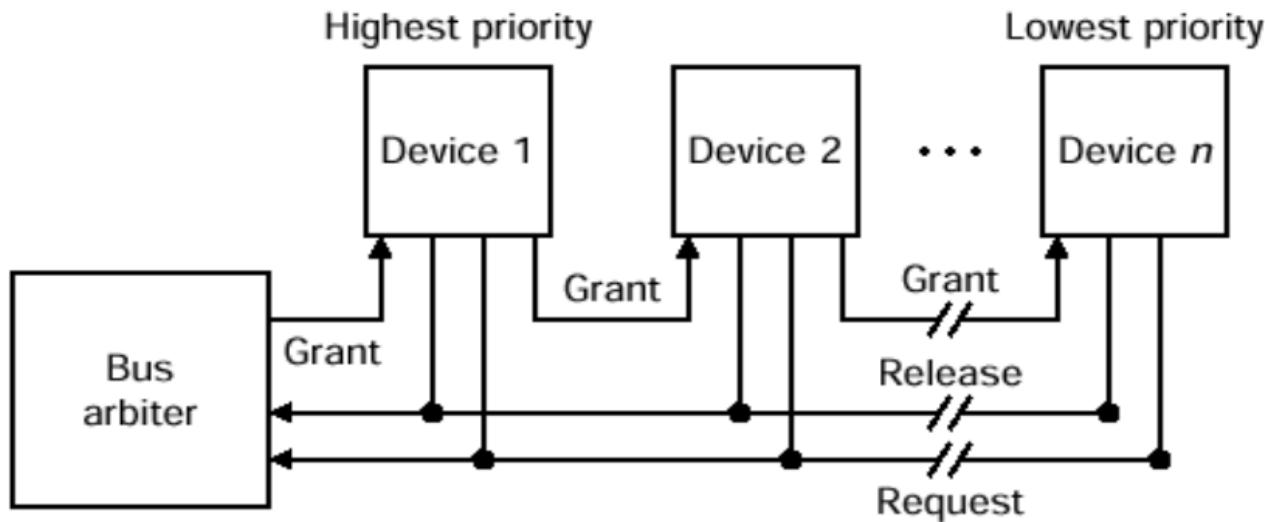
- **Asynchronous**

- The bus has no clock. No clock skew, it deals with devices with different speeds
- Can be longer, thus slower. It requires **handshaking** protocols.
- Possible protocol (3 control lines, 1 data line where data and address are multiplexed)
 1. UIO1 writes a READ (WRITE) request **ReadReq (WriteReq)** in the *control* line and the address in the *data* line
 2. UIO2 reads the address and writes an **ACK** into the control line to UIO1
 3. UIO2 writes the data on the bus and writes the **DataReady** control line to notify UIO1
 4. UIO1 reads the data and sends an **ACK** into the control line to UIO2

Bus arbitration

- Bus communications must be managed by a *communication protocol*
- **Bus master** concept: unit that can initiate a bus request
 - Simplest configuration: only one master
 - Buses typically have several masters
- **Arbitration** concept: choosing a master among multiple requests trying to implement priority and fairness (preventing starvation)
 - Only one master at a time (all others listen to the bus)
 - The role of the arbiter is to manage the bus requests and assign grants considering priorities and fairness.
- Implementation models for arbitration:
 - *Centralized*: a dedicated device (Arbiter) collects requests and then decide (potential *bottleneck problem*)
 - *Daisy-chain* (next slide), not fair
 - With independent request/reply lines
 - *Distributed*: every device sees the requests simultaneously and participates to the selection of the next master (more complex to realize and it needs a lot of control lines)

Daisy-chain arbiter



- Devices connect to the bus in priority order
- High-priority devices intercept/deny requests by low-priority ones
- Simple to implement but slow and cannot ensure fairness
 - Request line: this is a wired-OR line.
 - Grant line: the signal is propagated through all the devices. If a device made a request, it will take control of the bus when it receives the grant signal and then negates the grant line for the next device.

bus examples

Characteristic	Firewire (1394)	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Basic data width (signals)	4	2	2 per lane	4	4
Theoretical peak bandwidth	50 MB/sec (Firewire 400) or 100 MB/sec (Firewire 800)	0.2 MB/sec (low speed), 1.5 MB/sec (full speed), or 60 MB/sec (high speed)	250 MB/sec per lane (1x); PCIe cards come as 1x, 2x, 4x, 8x, 16x, or 32x	300 MB/sec	300 MB/sec
Hot pluggable	Yes	Yes	Depends on form factor	Yes	Yes
Maximum bus length (copper wire)	4.5 meters	5 meters	0.5 meters	1 meter	8 meters
Standard name	IEEE 1394, 1394b	USB Implementors Forum	PCI-SIG	SATA-IO	T10 committee

I/O management

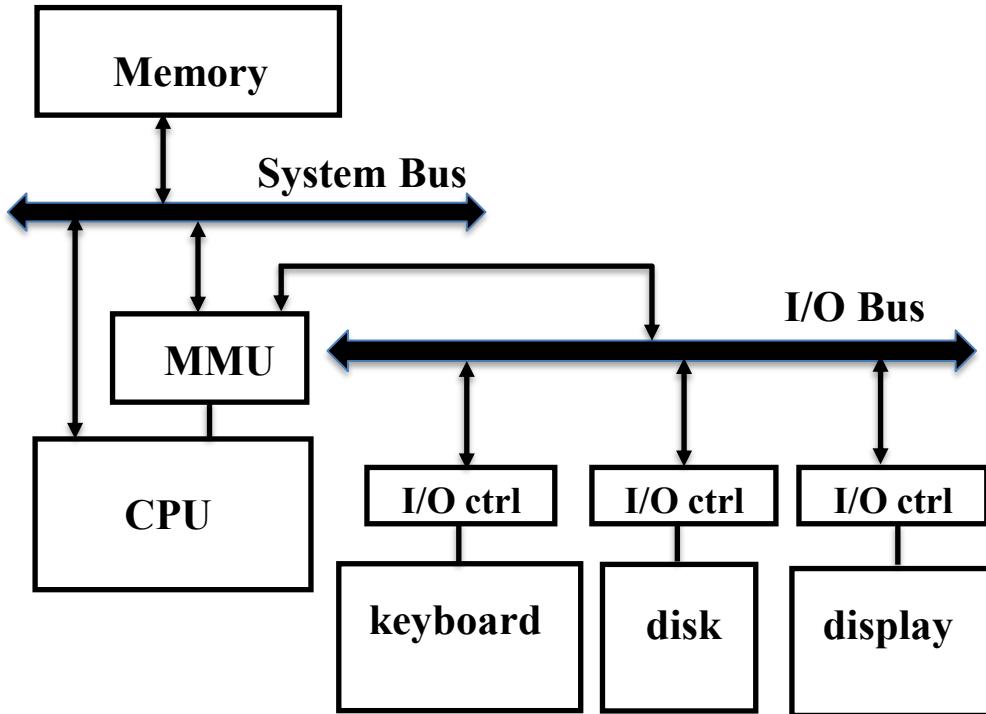
- How does I/O actually happen?
- We have to answer the following questions:
 1. How does CPU give commands to I/O devices?
 2. How does CPU know when I/O devices completed operations?
 3. How do I/O devices execute data transfers?

Sending commands to I/O devices

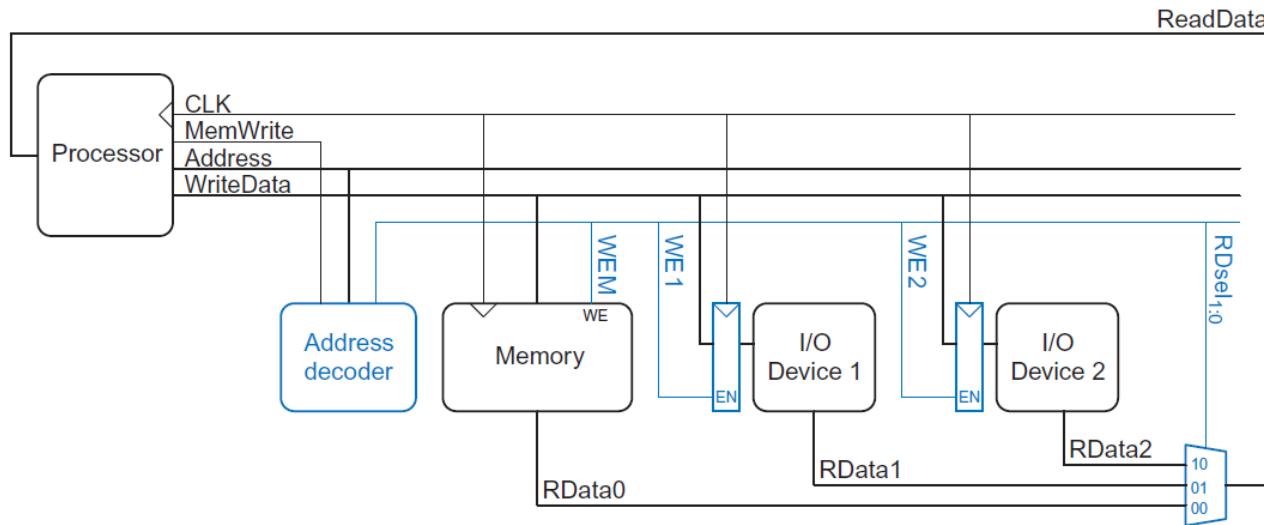
- How does CPU give commands to I/O devices?
- Who can send commands to I/O devices? **Only the OS!**
- How do programs in user space send I/O commands?
 - **By using System Calls!** They demand the task to the OS
- Two options for sending commands to I/O devices:
 - **I/O instructions**
 - ISA instructions that address the registers of the I/O device
 - *Privileged instructions* available only in kernel mode
 - Example architectures: Intel IA-32, IBM370
 - **Memory-mapped I/O**
 - A portion of the ***physical address space*** is reserved for I/O

Memory-mapped I/O

- The internal registers of the devices are mapped to main memory locations (**at reserved physical addresses**)
- I/O commands are standard memory read/write
- Operations to those locations are redirected to the device controllers by the MMU
- User mode accesses to memory-mapped areas generate *memory violation* exceptions



Memory-mapped I/O



Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following assembly code writes the value 7 to I/O Device 1.

```
MOV R1, #7  
LDR R2, =ioadr  
STR R1, [R2]  
ioadr DCD 0x20001000
```

The address decoder asserts WE1 because the address is 0x20001000 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following assembly code.

```
LDR R1, [R2]
```

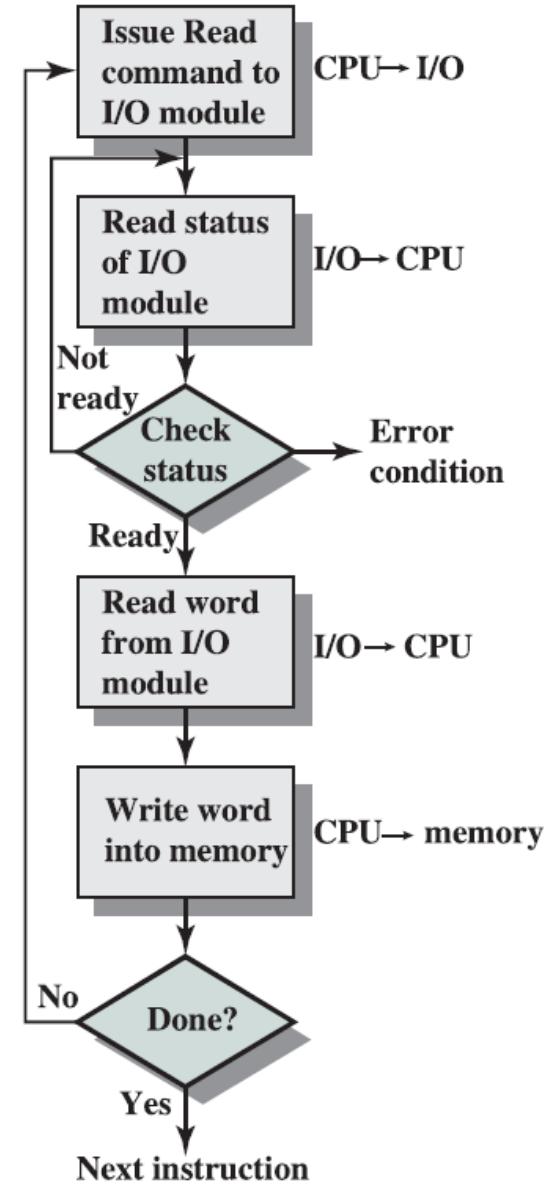
The address decoder sets *RDsel*_{1:0} to 01, because it detects the address 0x20001000 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into R1 in the processor.

Querying I/O status

- How does the CPU know if the operation request has completed?
 - Programmed I/O
 - CPU directly control I/O operation and status
 - Interrupt-driven I/O
 - An **interrupt** is an *asynchronous event coming from a device* (e.g., I/O module, timer, DMA controller)
 - Sometimes **exceptions** are also called interrupts (or comprise interrupts). In general exceptions are *synchronous events* that disrupt program execution (e.g., arithmetic overflow, undefined instruction).

Programmed I/O

- CPU has direct control over I/O
 - Sensing status
 - Read/write commands
 - Data transferring
- The I/O device has a *passive* role
- CPU waits for I/O module to complete operations (*polling*)

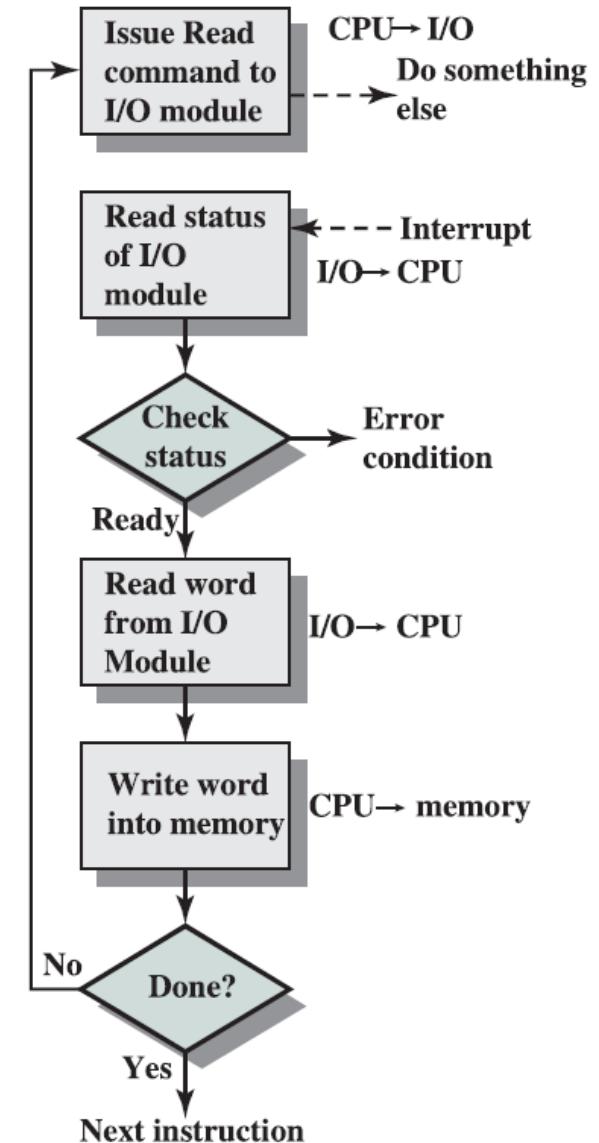


Programmed I/O

- Polling
 - *Processor queries I/O device status register*
 - READ example:
 1. write I/O control register for READ op
 2. while (ready-bit == 0) do; // **busy waiting**
 3. *load the data in memory*
 - *Pros: Simple to implement*
 - *Cons: Waste of processor's cycles*
 - *The CPU is much faster than any I/O device*
 - *The CPU may read the Status register many times only to find that the device has not completed the operation*

Interrupt-driven I/O

- The problem of programmed I/O is CPU waste of time
- **Interrupts:** alternative to polling
 - The CPU issues commands to a device and continues doing other tasks
 - I/O device generates an interrupt when the status changes, i.e., the data is ready
 - I/O interrupts are asynchronous
 - I/O interrupts are prioritized
 - High-bandwidth I/O devices have higher priority than low-bandwidth ones



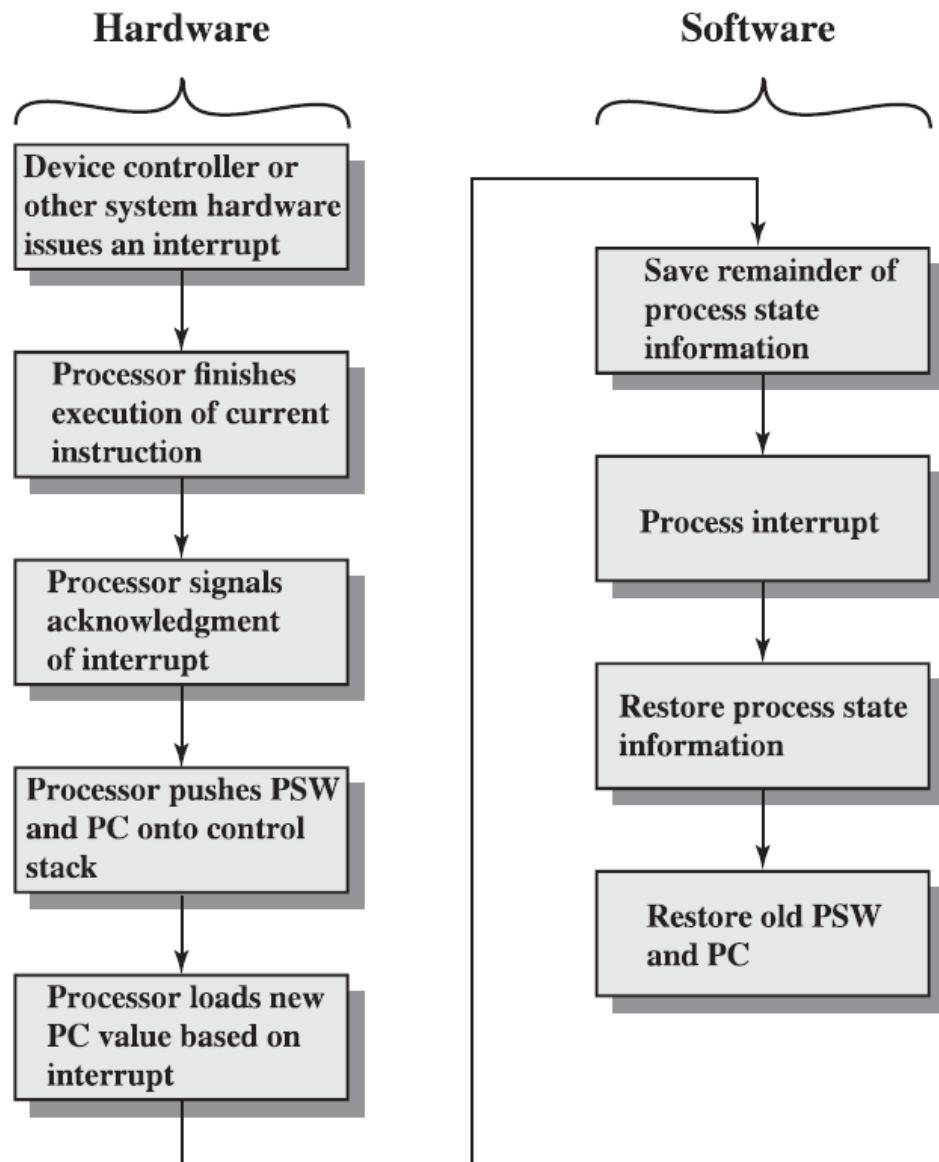
Interrupt-driven I/O

- CPU viewpoint
 - Issue I/O command (e.g., a memory-mapped load)
 - Do other work
 - Check for interrupt at the end of the fetch-execute cycle
 - If interrupt received
 - Save current context (i.e., registers, not necessarily all) and switch to *privileged level* (i.e., PL1 on ARM processor)
 - On the basis of the interrupt ID and interrupt priority the associated interrupt handler is ready to be executed
 - **NOTE:** The previous steps are atomic from the OS viewpoint!
 - Execute the handler (i.e., the OS handles the interrupt)
 - Restore context for executing the next instruction
 - **NOTE:** the restore is an atomic operation for the OS!

Interrupt management steps

PSW is the Program Status Word register, the equivalent of the **CPSR/APSR** register for ARM processors

```
1 while(true) {  
2     try {  
3         IR = fetch(PC);  
4         decode(IR);  
5         execute(IR);  
6         update(PC);  
7     } catch (exception e) {  
8         exception_management();  
9     }  
10    if(interrupt) {  
11        interrupt_management();  
12    }  
13 }
```



Polling overhead

- **Assumptions:** 500MHz CPU, polling cost 400 cycles
- **MOUSE:** slow-bandwidth device
 - we want to poll 30 times per second
 - Cycles per second for polling: $(30 \text{ poll/s}) * 400 = 12000 \text{ cycles/s}$
 - Percentage of cycles spent for polling: $12\text{K}/500\text{M} = 0.002\%$
 - **Acceptable overhead!**
- **DISK:** high-bandwidth device (4MB/s transfer rate, 16B interface)
 - To not miss data we must poll often enough: $(4\text{MB/s})/16\text{B} = 250 \text{ K/s}$
 - Cycles per second for polling: $250 \text{ K/s} * 400 \text{ (cycles/s)} = 100 \text{ M cycles/s}$
 - Percentage of cycles spent for polling: $100 \text{ M}/500\text{M} = 20\%$
 - **Not acceptable!!** We spent 20% of cycles just for checking the I/O registers, not for data transfer

Interrupt overhead

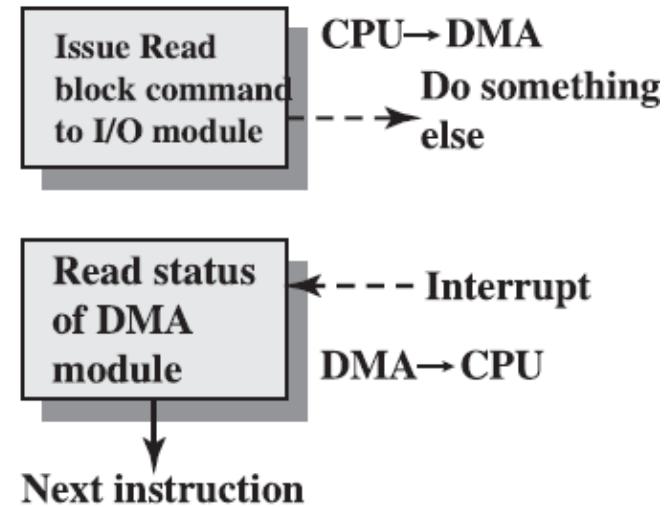
- **Assumptions:** 500MHz CPU, interrupt handling takes 400 cycles
- **DISK (4MB/s transfer rate, 16B interface)**
 - The processor is involved for each data transfer (16 B)
 - $(4M \text{ B/s}) / (16 \text{ B}) * [400 \text{ cycles} / 500M \text{ cycles/s}] = 20\%$
 - Same overhead of polling **if the disk is fully utilized**, but in this case the CPU can execute other instructions
 - In CPU-bound applications, the I/O devices are used for a fraction of time!
- Let's suppose that the data transfer time takes 100 cycles
 - The percentage of time the processor spends transferring data is $(4M \text{ B/s}) / (16 \text{ B}) * [100 \text{ cycles} / 500M \text{ cycles/s}] = 5\%$

Data transfers (DMA)

- **How do I/O devices execute data transfers?**
- Interrupt-driven I/O remove the problems of polling
- Still the OS must transfer data one word at a time. This is ok only for low-bandwidth I/O devices (e.g., mouse)
 - Caveat: in general the cost of managing interrupt is higher than polling
- **Direct Memory Access (DMA)**
 - A mechanisms that provides an I/O device controller with the ability to transfer data directly to and from main memory without involving the CPU
 - DMA decouples the CPU-Memory protocol from the I/O device-memory protocol
 - Interrupts used by the I/O device to communicate with the CPU only on completion of the I/O transfer or when an error occurs.

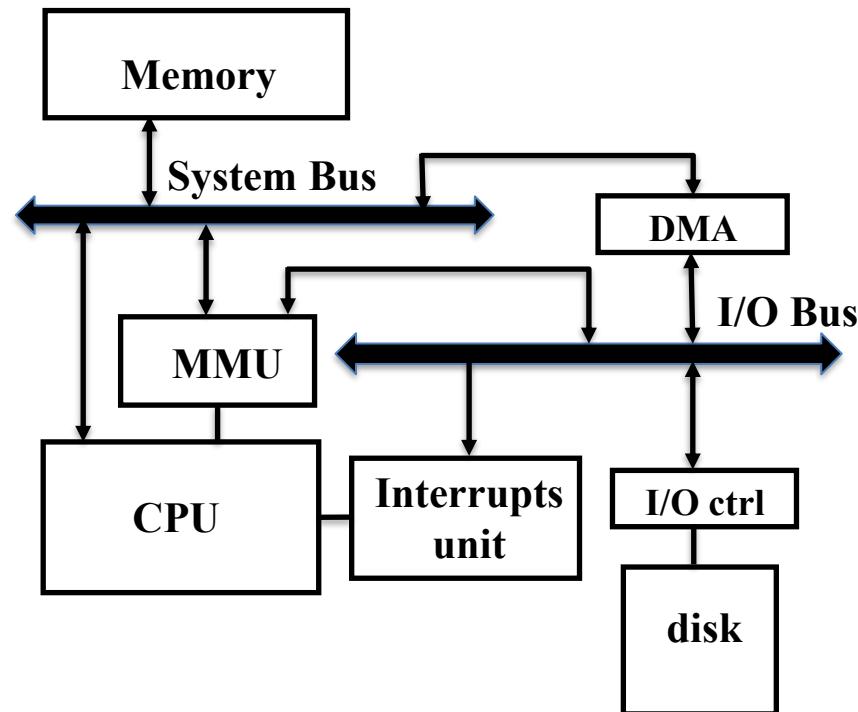
DMA controller

- DMA is implemented with a specialized controller
- To do DMA, an I/O device is attached to a DMA controller
 - Multiple devices can be connected to the same controller
- The controller itself is seen as a memory-mapped I/O device
- The DMA controller takes care of bus arbitration and data transfer
 - It becomes the bus master
- The DMA controller and the CPU contend the memory bus

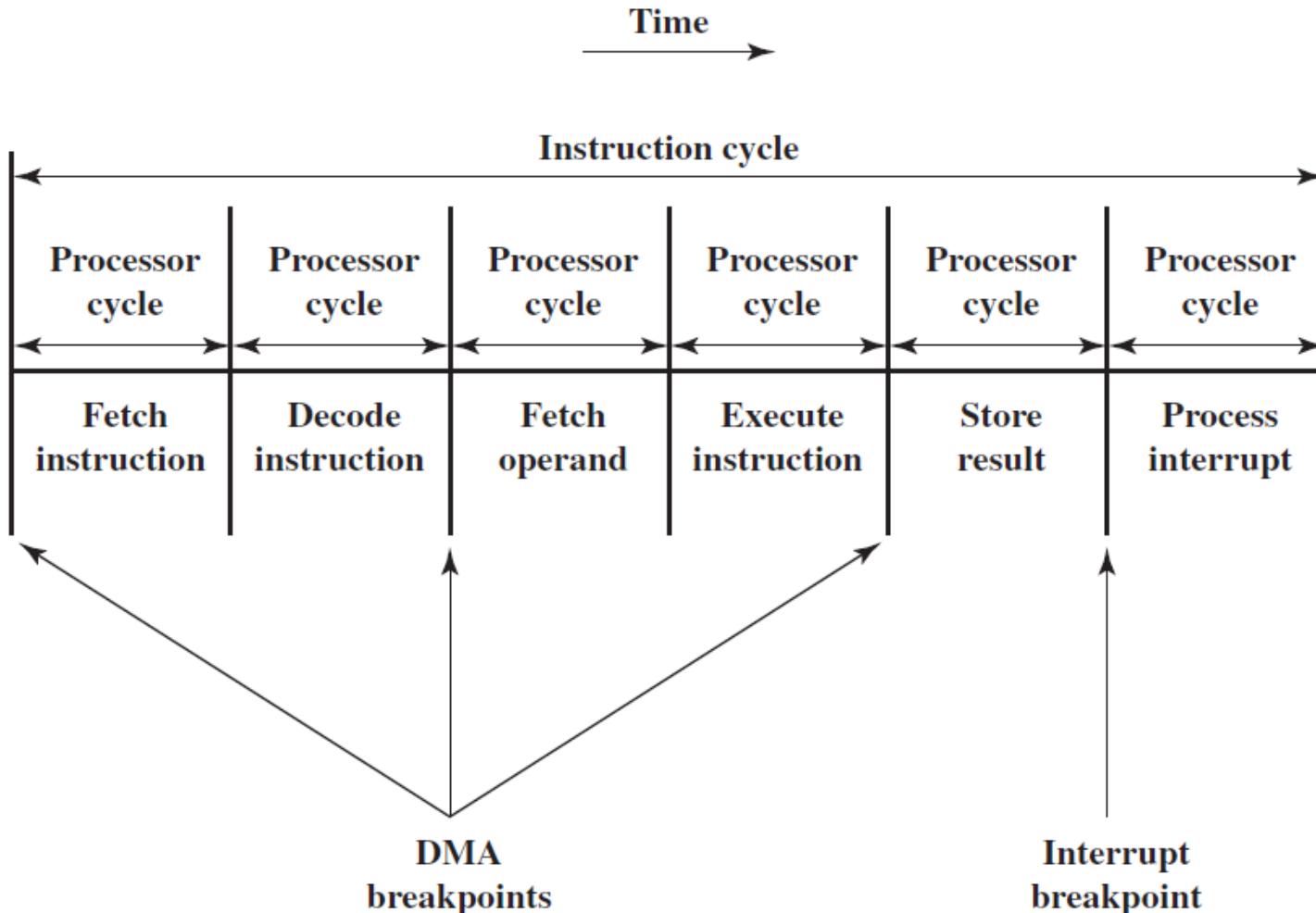


DMA transfer

- There are three steps in a DMA transfer:
 1. CPU sets up DMA providing *identity, op, memory address, and number of bytes to transfer*
 2. DMA starts the op on the device and arbitrates for the connection. It transfers the data from the device or memory
 3. Once the DMA transfer is complete, the controller sends an interrupt to the CPU



DMA and Interrupt Breakpoints



DMA overhead

- **Assumptions:**
 - 500MHz CPU, disk device with 4MB/s transfer rate, 16B interface,
 - 50% utilization
 - Interrupt handling takes 400 cycles
 - Data transfer takes 100 cycles
 - DMA setup takes 1600 cycles, it transfers a 16KB block at a time
- Processor overhead for **interrupt-driven I/O without DMA**
 - The processor is involved for each data transfer (16 B)
 - $0.5 * (4M \text{ B/s}) / (16 \text{ B}) * [(400+100) \text{ cycles} / 500\text{M cycles/s}] = 12.5 \%$
- Processor overhead for **interrupt-driven I/O with DMA transfer**
 - The processor is involved once per block transfer (16 KB)
 - $0.5 * (4M \text{ B/s}) / (16K \text{ B/s}) * [(1600+ 400) / (500 \text{ M cycles/s})] = 0.05 \%$

DMA and Memory Hierarchy

- **Without DMA:** processor initiates all data transfers
 - All transfers go through address translation (MMU)
 - Transfers can be of any size and cross virtual page boundaries
 - No impact on the cache hierarchy, caches never contain stale data
- **With DMA:** the DMA controller initiates data transfers
 - There is another path to the memory system!
 - Potential issues:
 1. Do DMA controllers use *virtual or physical addresses?*
 2. What if they write data to a **cached memory location?**

Virtual vs Physical DMA

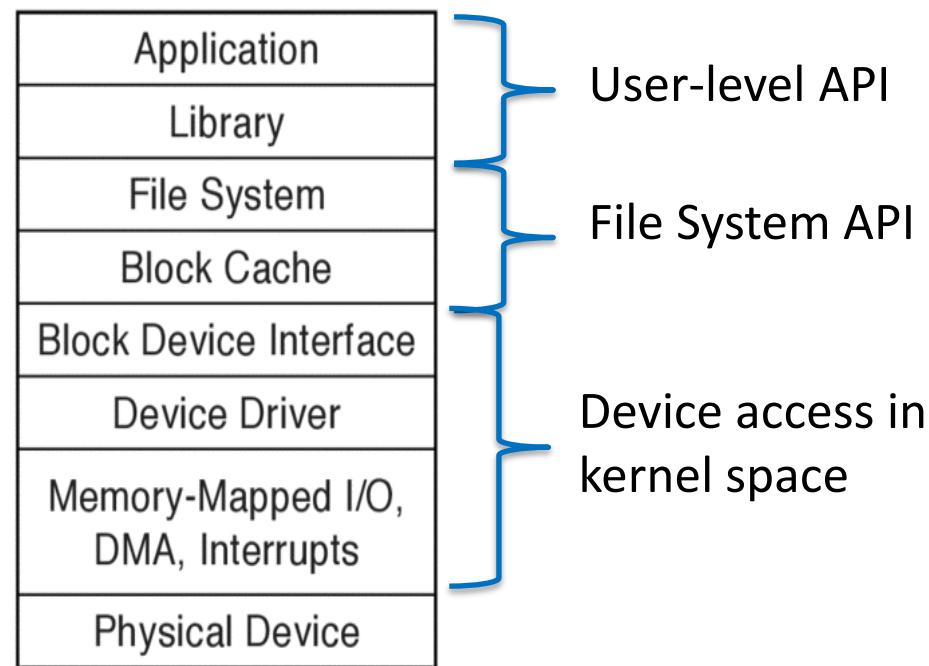
- Which addresses does processor specify to DMA controller?
- **Virtual DMA:**
 - The DMA controller must do address translation internally by using a small cache (TLB) initialized by the OS when it requests an I/O transfer
 - Complex and flexible (e.g., large cross-page transfers)
- **Physical DMA:**
 - The DMA controller works with physical addresses
 - Transfers at page granularity, the OS breaks large transfers into page-size chunks
 - More simple and less flexible

DMA and caching

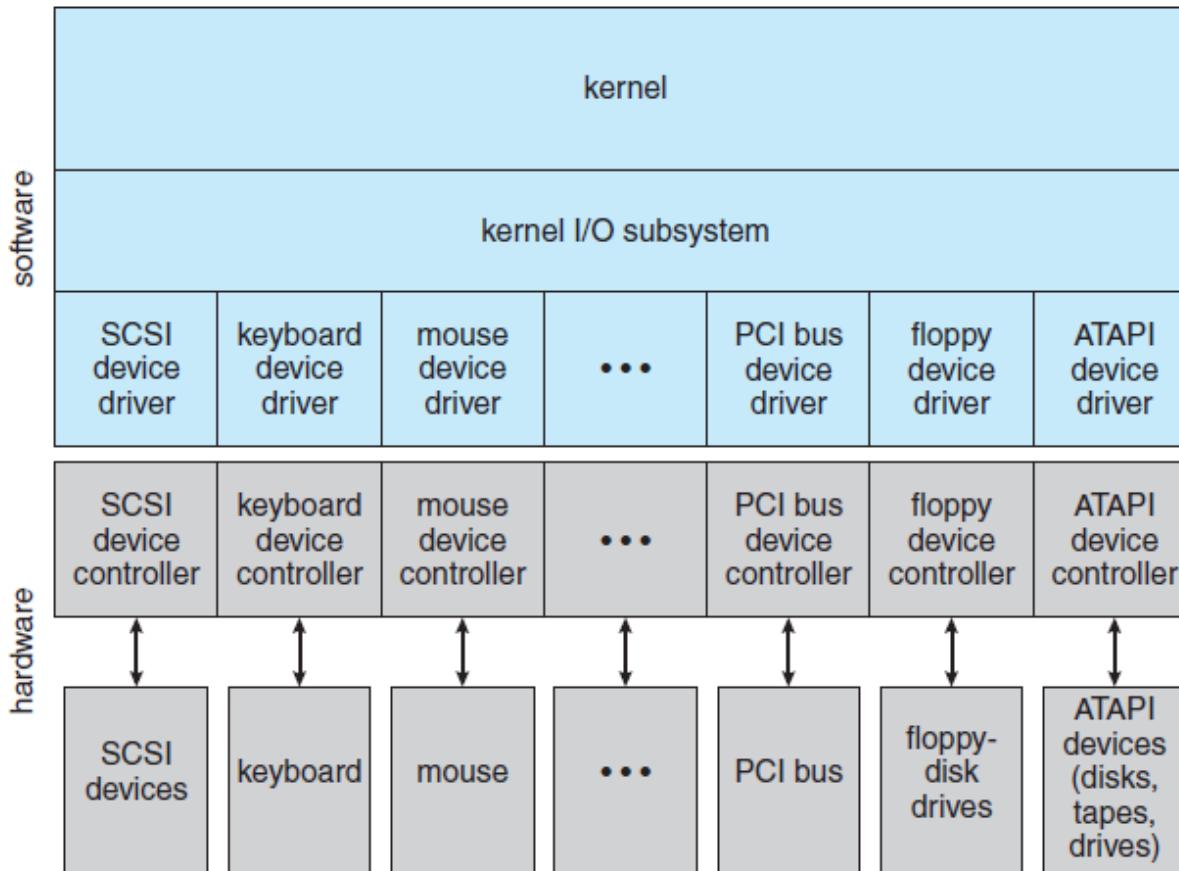
- Caching reduces CPU's instruction and data access latency, and reduces CPU's use of memory, thus leaving more memory/bus bandwidth for DMA transfers
- But caches introduce a **coherence problem** for DMA
 - If the DMA writes into main memory, then cached version of data is stale
 - For Write-Back caches, the DMA controller might read old data values from memory while cached data have the dirty-bit set
- The solution is the selective *flushing/invalidations* of cached lines involved in DMA transfers
 - Requires extra logic for HW cache coherence!

Device Access

- A software stack that provides ways to access a wide range of I/O devices
- A common interface
 - In POSIX equivalent to file access
 - In terms of open/close, read/write system calls
- Device drivers for each specific device
 - Upper part **HW-independent (to enhance portability)**
 - Lower part **HW-dependent**



Kernel I/O structure

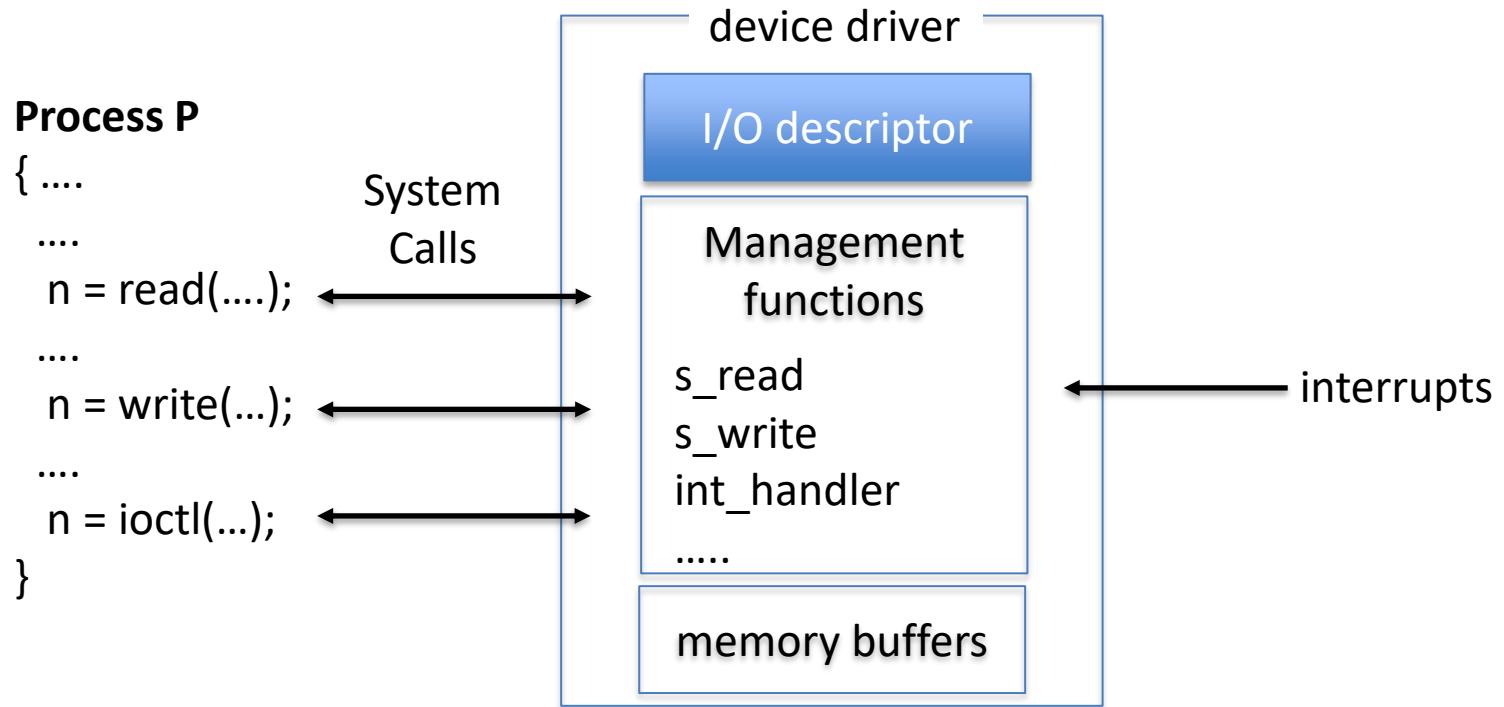


- The purpose of the device driver layer is to hide the differences among device controllers

Device driver

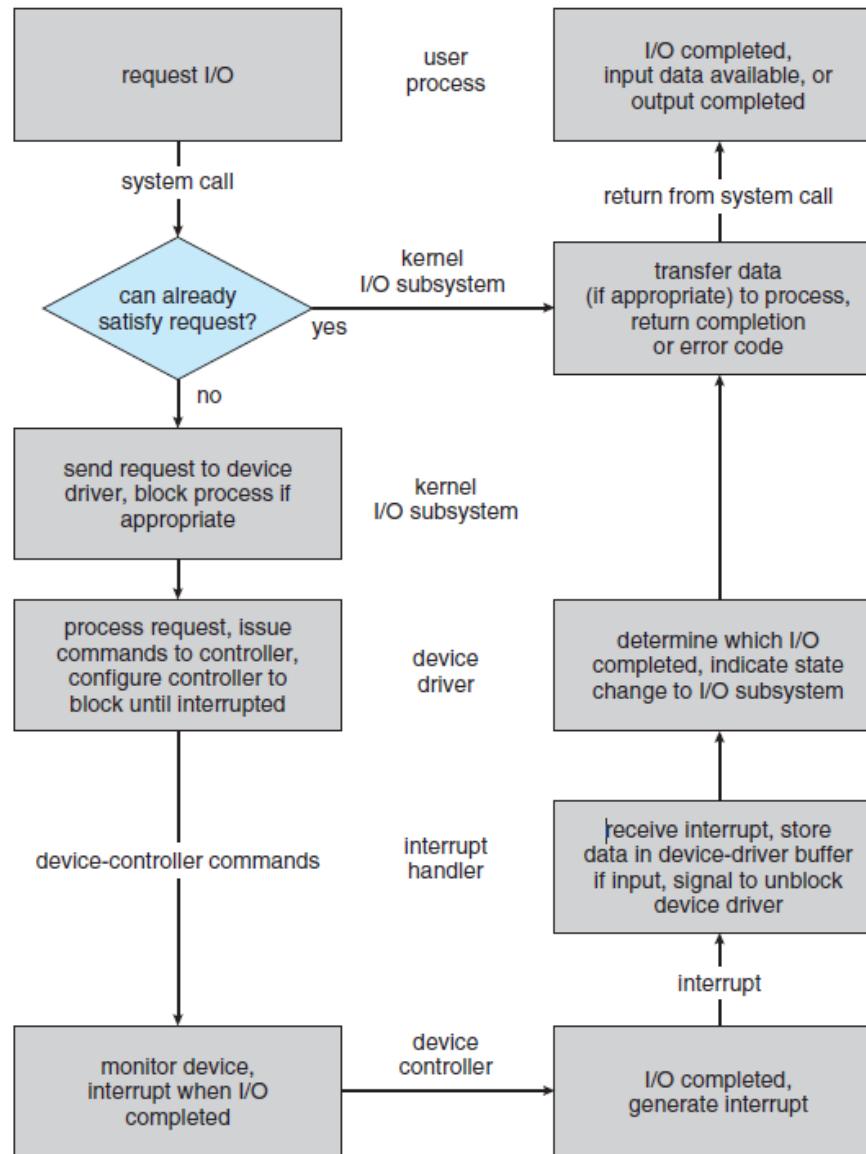
- The file system:
 - Defines a name space for devices and their identifiers (i.e., <major, minor> numbers)
 - Implements caching policies
 - It is HW-independent
- The ***device driver***:
 - It works in kernel space
 - Interfaces with the device controller (HW-dependent)
 - Manages synchronization with the device
 - Manages data transfer to/from device (by interfacing the DMA controller) and to/from user processes
 - Manages device failures

Device driver

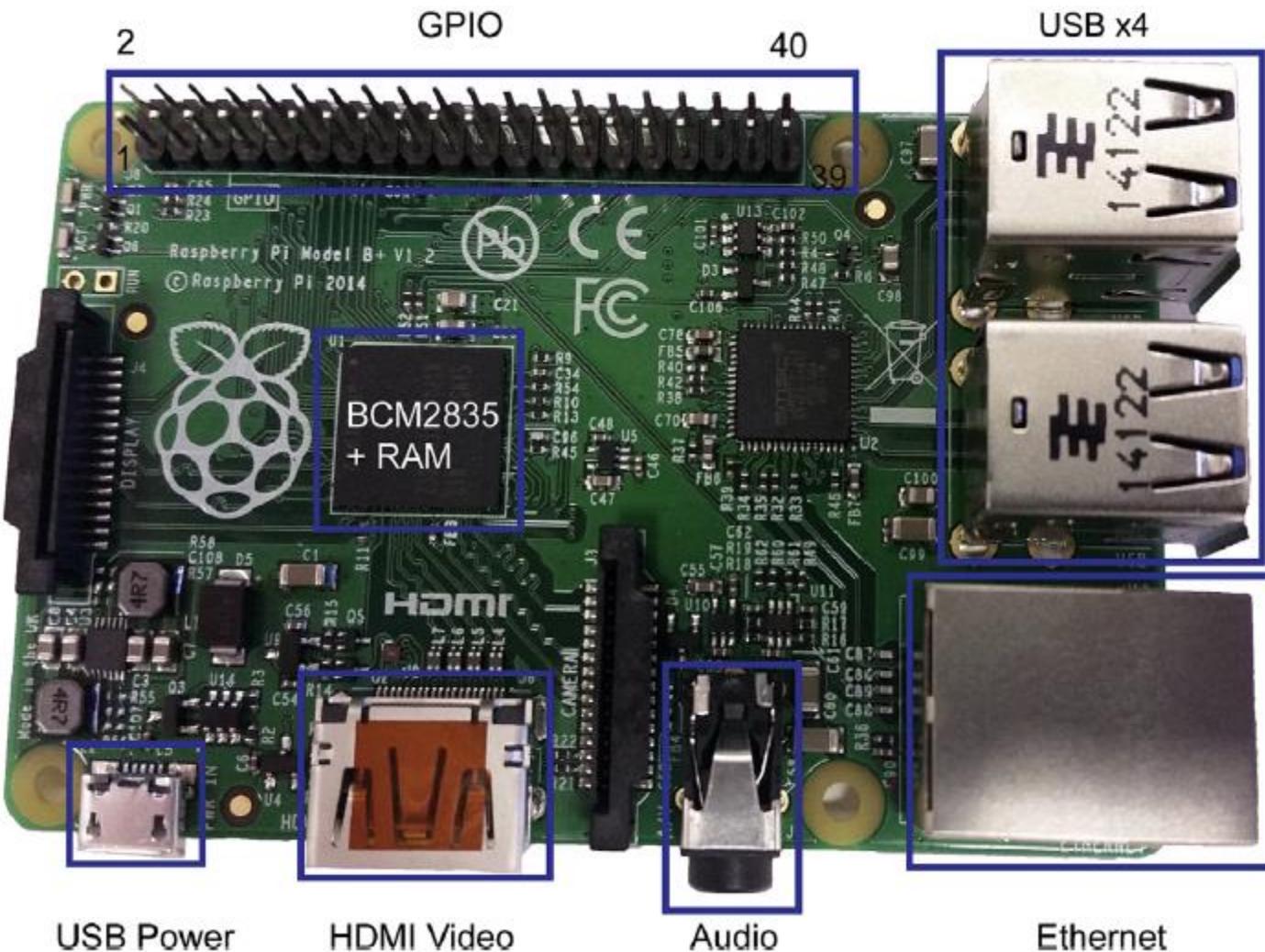


- I/O descriptor contains all information to interact (i.e., communicate and synchronize) with the specific device
 - Buffer pointers, counters, synchronization variables, status, etc.

I/O request life cycle



ARM example



MM I/O on BCM2835

- The memory-mapped I/O on the BCM2835 is found at ***physical addresses*** 0x20000000-0x20FFFFFF
- Loads/stores refers to ***virtual addresses***
- To access memory-mapped I/O memory regions it is *necessary to map the physical addresses to the program's virtual address space.*
 - *mmap system call*

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE 0x20000000
#define GPIO_BASE          (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int *gpio; //Pointer to base of gpio
#define GPLEV0              (* (volatile unsigned int *) (gpio + 13))
#define BLOCK_SIZE           (4*1024)

void pioInit(){
    int mem_fd;
    void *reg_map;

    // /dev/mem is a psuedo-driver for accessing memory in Linux
    mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
    reg_map = mmap(
        NULL,                      // Address at which to start local mapping (null = don't-care)
        BLOCK_SIZE,                // 4KB mapped memory block
        PROT_READ|PROT_WRITE,      // Enable both reading and writing to the mapped memory
        MAP_SHARED,                // Nonexclusive access to this memory
        mem_fd,                   // Map to /dev/mem
        GPIO_BASE);               // Offset to GPIO peripheral

    gpio = (volatile unsigned *)reg_map;
    close(mem_fd);
}
```

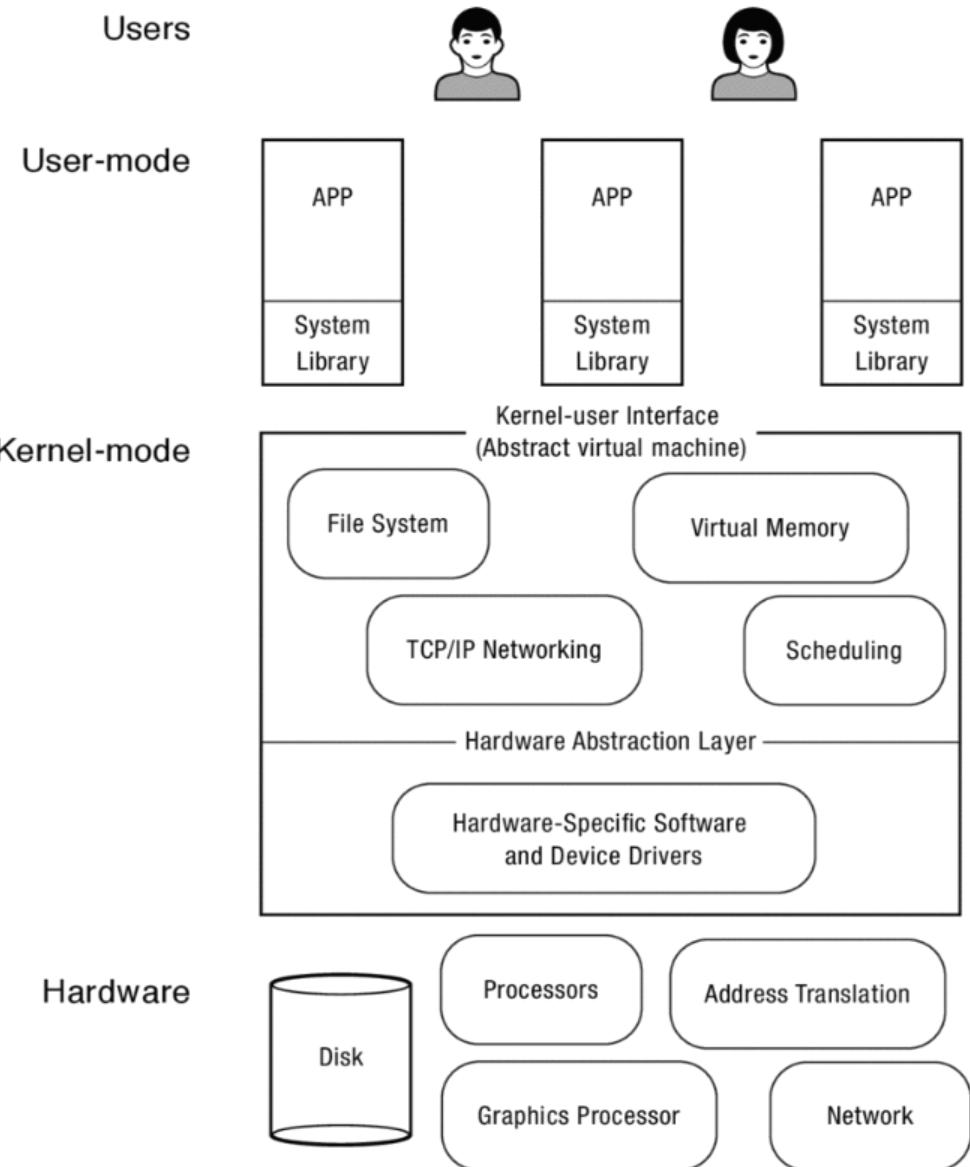
Operating Systems: Principles and Practice

Main Points

- Operating system definition
 - Software to manage a computer's resources for its users and applications
- OS challenges
 - Reliability, security, responsiveness, portability, ...
- OS history
 - Where we are?

What is an operating system?

- Software to manage a computer's resources for its users and applications
- Core functionalities:
 - Scheduling of resources
 - Virtual Memory
 - Inter-Process Communication (IPC) and synchronization mechanisms



Operating System Roles

- Referee:
 - Resource allocation among users, applications
 - Isolation of different users, applications from each other
 - Communication between users, applications
- Illusionist
 - Each application appears to have the entire machine to itself
 - Infinite number of processors, (near) infinite amount of memory, reliable storage, reliable network transport
- Glue
 - Provide common, standard services to applications
 - Simplifies application development
 - Libraries, user interface widgets, ...

Example: File Systems

- Referee
 - Prevent users from accessing each other's files without permission
 - Even after a file is deleted and its space re-used
- Illusionist
 - Files can grow (nearly) arbitrarily large
 - Files persist even when the machine crashes in the middle of a save
- Glue
 - Named directories, printf, ...

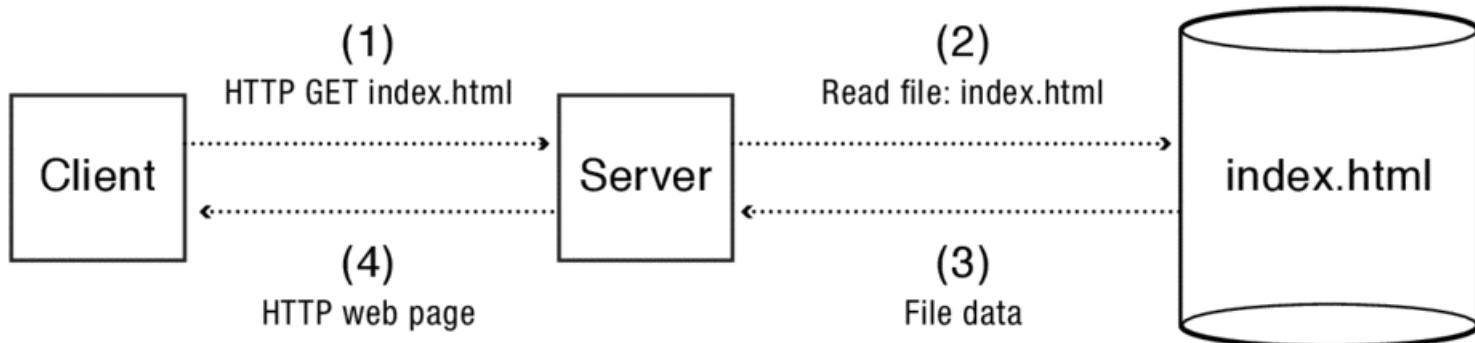
Operating system design patterns

- Cloud computing
 - Referee: how to allocate resources between competing applications in the cloud?
 - Illusionist: computing resources in a cloud evolve continuously, how to isolate applications from this evolution?
 - Glue: how to provide common, standardized access to the cloud services?
- Web services
 - Referee: ensure responsiveness when multiple tabs are opened at the same time
 - Illusionist: web services are geographically distributed for fault tolerance. Mask server failures to the users.
 - Glue: how does a browser achieve portable execution of scripts across different OS and HW platforms?

Operating system design patterns

- Multi-user database systems
 - Referee: how to enforce data access and privacy to different users ?
 - Illusionist: how to mask failures so that data remains consistent and available to users?
 - Glue: what common services to programs development?
- Internet
 - Referee: guarantee differentiated services to users and protect against DoS, spam, phishing etc...
 - Illusionist: internet appears as a unique, world-wide network but it is not!
 - Glue: internet protocols make applications independent of the underlying network architecture

Example: web service



- It defines a simple behavior but ...
- How does the server manage many simultaneous client requests?
- How do we keep the client safe from spyware embedded in scripts on a web site?
- How do make updates to the web site so that clients always see a consistent view?

Example: web service

However:

- Multiple users issue requests at the same time
 - These must be managed simultaneously
- Many requests involve data and computations
 - Think about search engines, a request may involve deep computations over large clusters of machines
- The server uses caches to speed up
 - Cache is shared among users, need for synchronized access mechanisms
- Servers send to clients scripts for pages customization
 - How does the client can protect itself from the execution of third party code that may embed viruses/spyware?

Example: web service

However:

- Web sites need to be updated
 - How to manage consistency with concurrent read requests?
- Client and server may run at different speeds
 - Need for speed decoupling
- Hardware supporting the web site may be updated
 - How to take advantage of this without rewriting the web server code?

Question

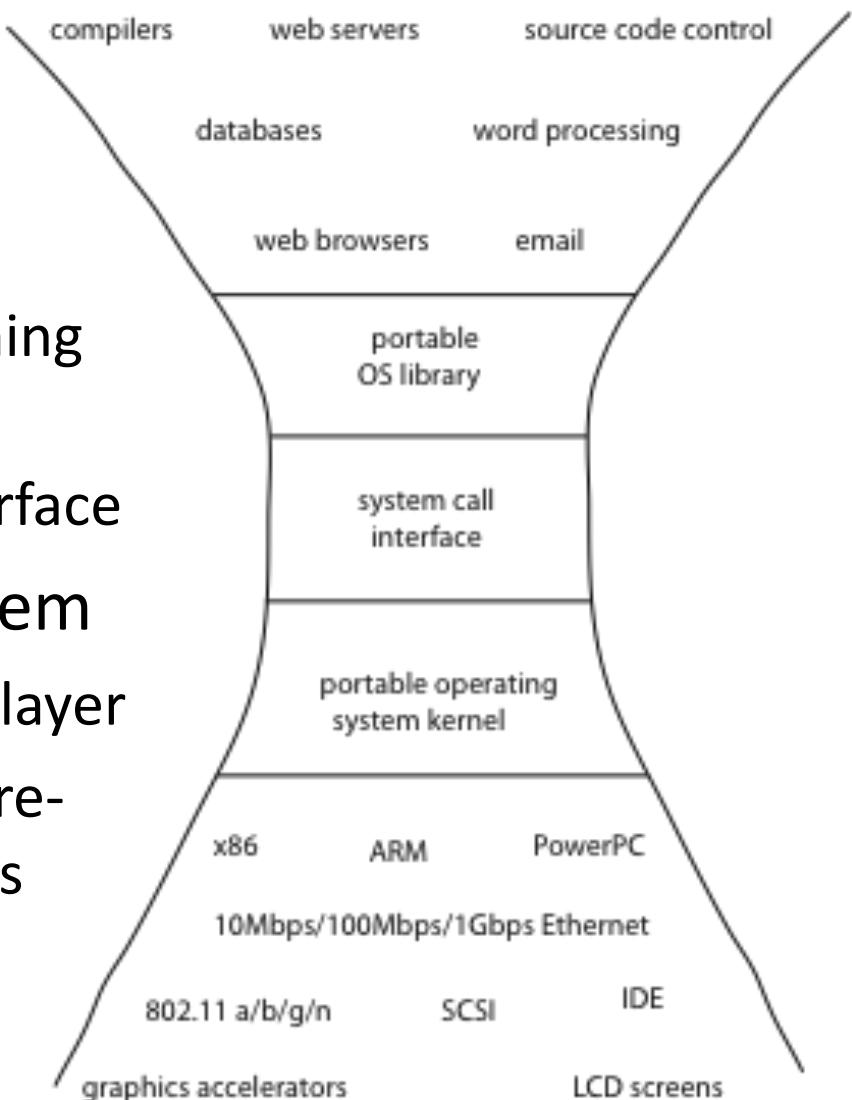
- What (hardware, software) do you need to be able to run an untrustworthy application?

OS Challenges

- Reliability
 - Does the system do what it was designed to do?
 - Availability
 - What portion of the time is the system working?
 - Mean Time To Failure (MTTF), Mean Time to Repair
- Security
 - Can the system be compromised by an attacker?
 - Privacy
 - Data is accessible only to authorized users
- Both require very careful design and code

OS Challenges

- Portability
 - For programs:
 - Application programming interface (API)
 - Abstract machine interface
 - For the operating system
 - Hardware abstraction layer
 - Most OS have hardware-specific kernel routines



OS Challenges

- Performance
 - Latency/response time
 - How long does an operation take to complete?
 - Throughput
 - How many operations can be done per unit of time?
 - Overhead
 - How much extra work is done by the OS?
 - Fairness
 - How equal is the performance received by different users?
 - Predictability
 - How consistent is the performance over time?

OS Structure

- Many dependencies among modules
 - Many parts of the OS depends on synchronization primitives
 - The Virtual Memory system depends on low-level HW support for address translation
 - Both the File System and the Virtual Memory share blocks of physical memory
 - The File System can depend on the network protocol stack
 - ...

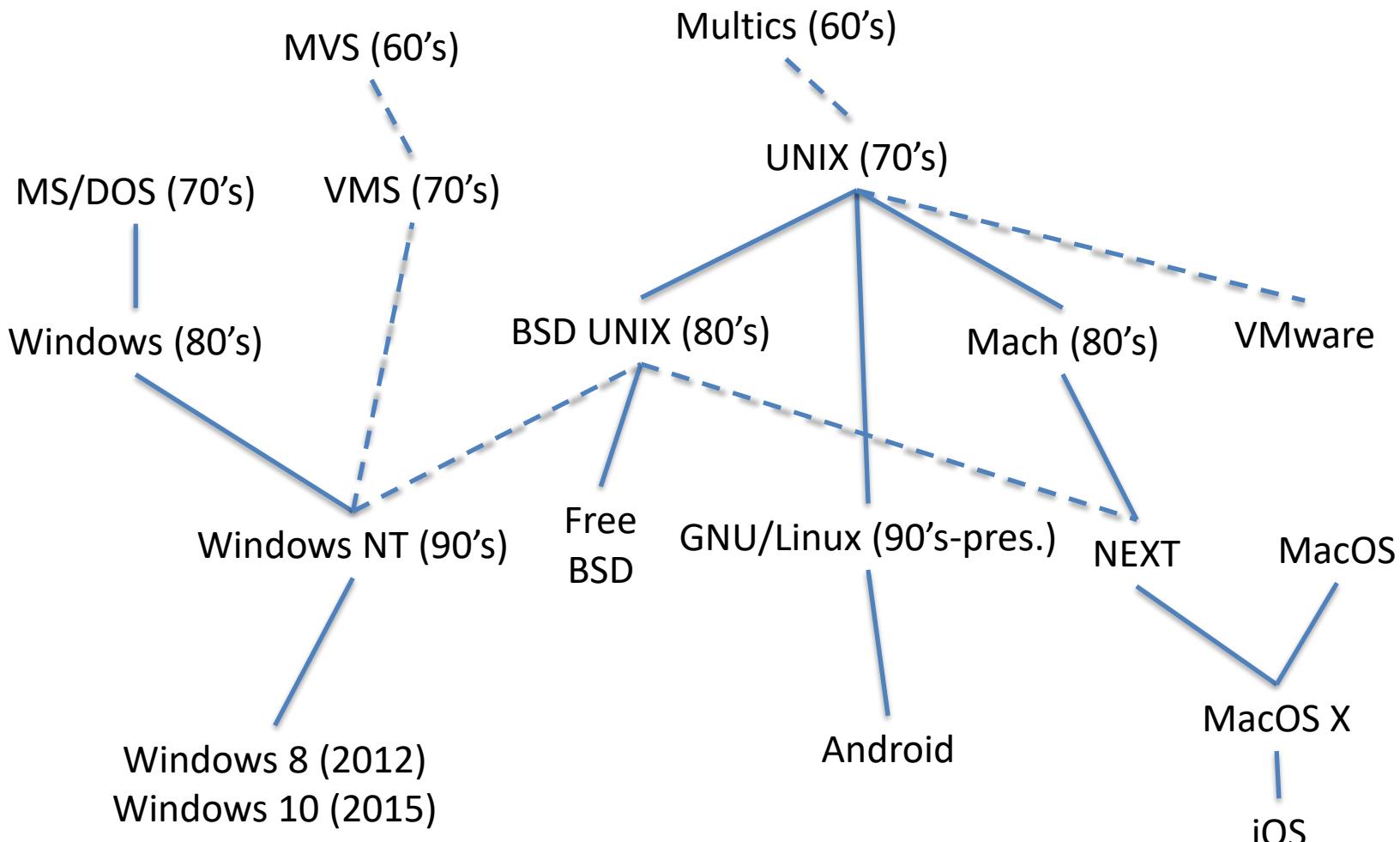
OS Structure

- Tradeoff in the OS design:
 - **Monolithic kernel**
 - Most of the OS functionalities run inside the kernel
 - To improve portability and flexibility monolithic kernels use HAL and dynamically loaded device drivers
 - **Microkernel**
 - In the kernel only core mechanisms, most of the OS functionalities run as user-level servers
 - More reliable due to the higher isolation; easier to debug, maintain and extend
 - **Hybrid model**
 - Combines the best practices of the two previous approaches

OS Adoption

- Adoption is beyond control of an OS
 - Wide availability of applications
 - Wide availability of HW supporting it
- Network effect
 - App stores
 - Example: Android model vs iPhone model
- Proprietary vs open systems
 - Not a clear winner

OS History



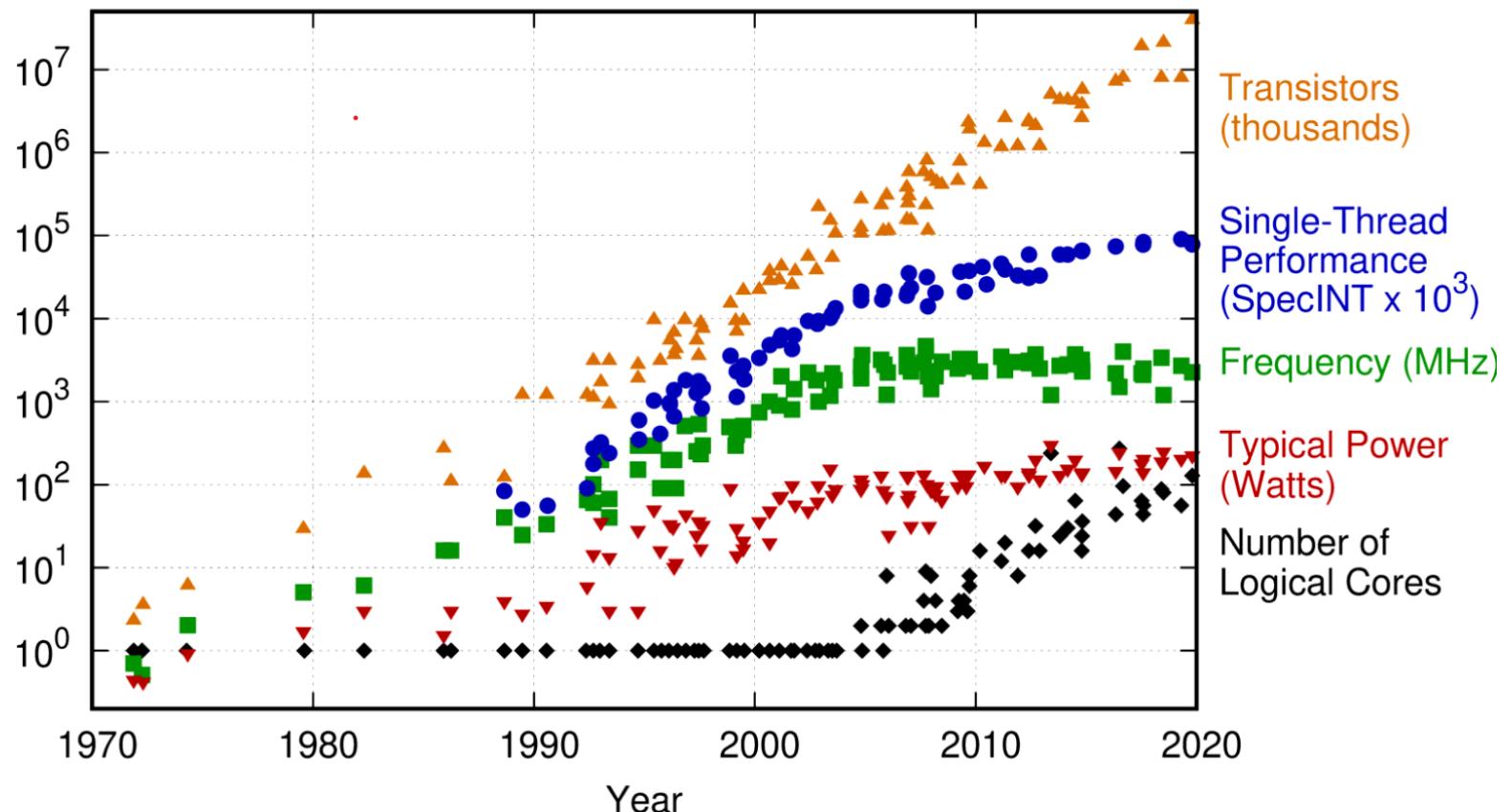
Influence
 Descendant

Computer Performance Over Time

	1981	1997	2014	Factor
Processor Speed MIPS	1	200	2500	2.5K
CPUs per computer	1	1	10+	10+
Processor \$ /MIPS	\$100 K	\$25	\$0.20	500 K
DRAM capacity (MiB/\$)	0.002	2	1K	500 K
Disk capacity (GiB) / \$	0.003	7	25 K	10 M
Home Internet	300 bps	256 Kbps	20 Mbps	100 K
LAN Network	10 Mbps shared	100 Mbps switched	10 Gbps switched	1000+
Users per machine	100:1	1:1	1:several	100+

Microprocessor Trend

48 Years of Microprocessor Trend Data



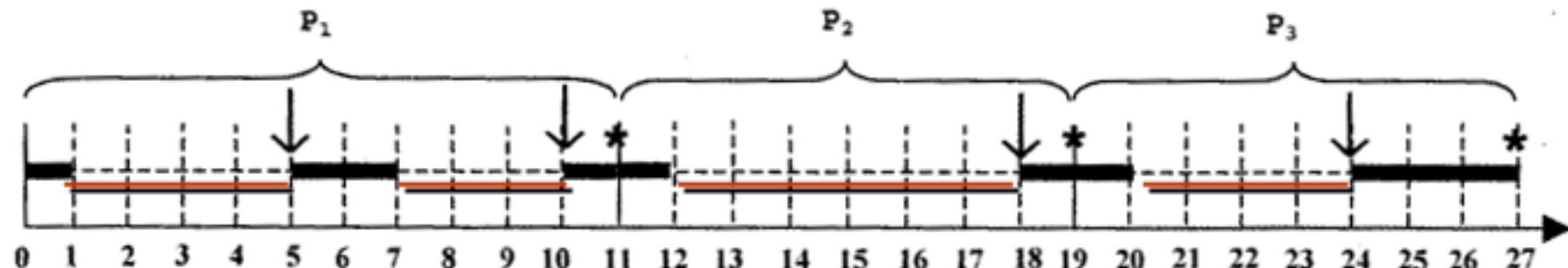
Source: <https://github.com/karlrupp/microprocessor-trend-data>

Early Operating Systems: Computers Very Expensive

- One application at a time
 - Had complete control of hardware
 - OS was runtime library
 - Users would stand in line to use the computer
- Batch systems
 - Keep CPU busy by having a queue of jobs
 - OS would load next job while current one runs
 - Users would submit jobs, and wait, and wait, ...

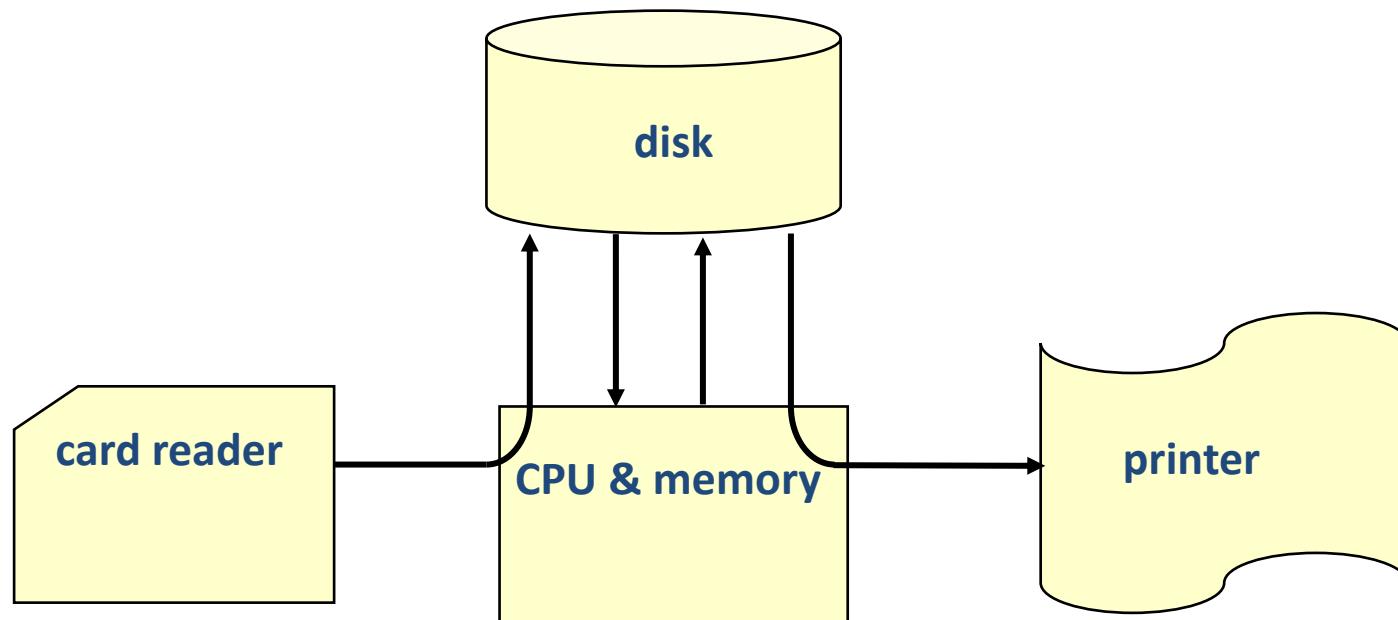
Single task systems

- Sequential execution
 - Black line: execution on the processor
 - Red line: I/O operation – the program stops & waits for the result of the operation



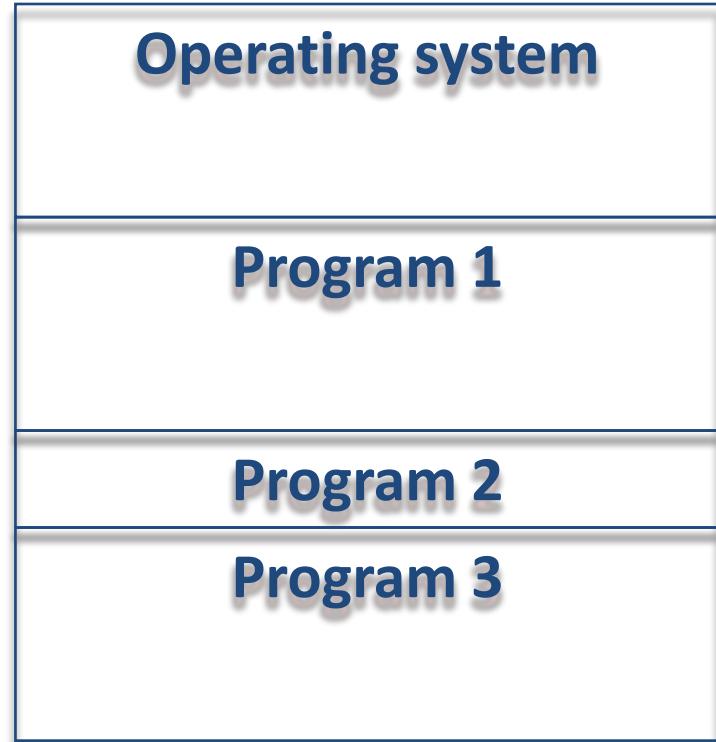
Early batch systems

- SPOOL: Simultaneous Peripheral Operation On-Line

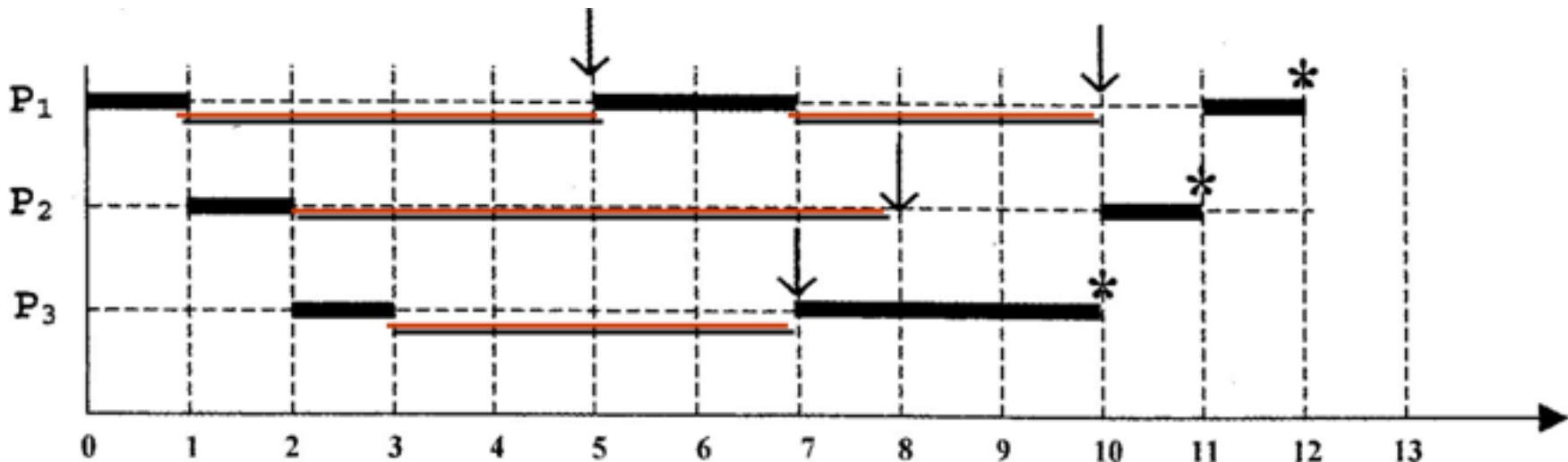
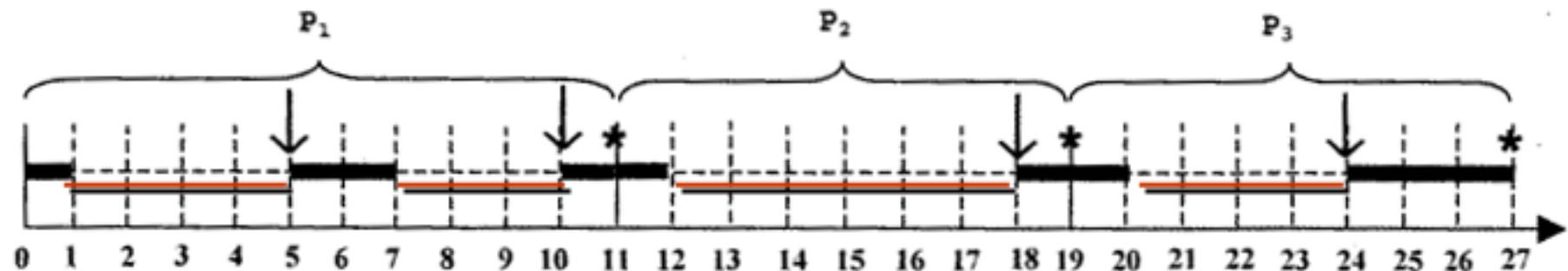


Multi-programmed batch systems

- multi-user system: several programs loaded in memory at the same time
- Spool optimization
- Resource optimization (processor, memory, devices)
 - Response time not important



Multi-tasking vs single-task

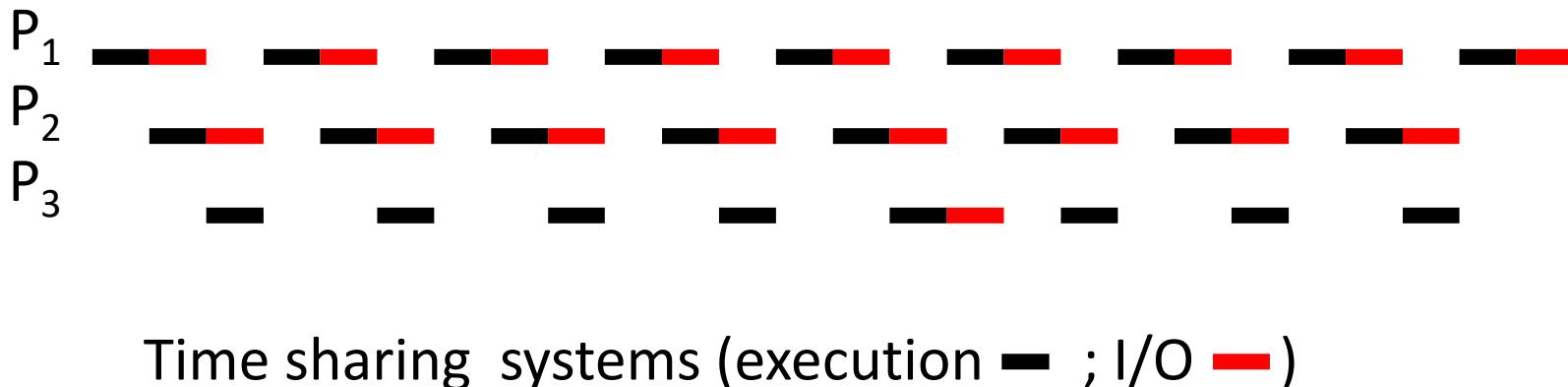
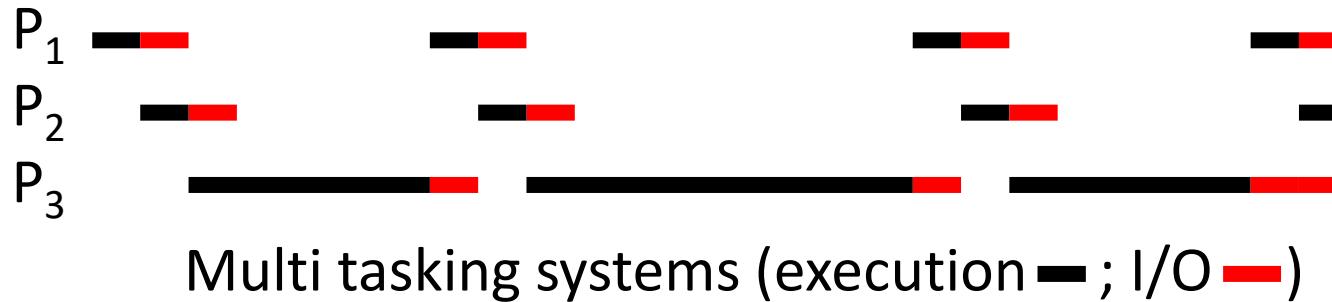


Time-Sharing Operating Systems: Computers and People Expensive

- Multiple users on computer at same time
 - Multiprogramming: run multiple programs at same time
 - Interactive performance: try to complete everyone's tasks quickly
 - As computers became cheaper, more important to optimize for user time, not computer time

Time-Sharing Operating Systems

- time sharing v.s. multitasking



Today's Operating Systems: Computers Cheap

- Smartphones
- Embedded systems
- Laptops
- Tablets
- Virtual machines
- Data center servers

Tomorrow's Operating Systems

- Giant-scale data centers
- Increasing numbers of processors per computer
- Increasing numbers of computers per user
- Very large-scale storage (cloud storage)

New scenarios:

- Edge Computing
- Internet of Things
- Green Computing

Bonus Thought Questions

- How should an operating system allocate processing time between competing uses?
 - Give the CPU to the first to arrive?
 - To the one that needs the least resources to complete? To the one that needs the most resources?
- What if you need to allocate memory?

The Kernel Abstraction

Challenge: Protection

- How do we execute code with restricted privileges?
 - Either because the code is buggy or if it might be malicious
- Some examples:
 - A script running in a web browser
 - A program you just downloaded off the Internet
 - A program you just wrote that you haven't tested yet

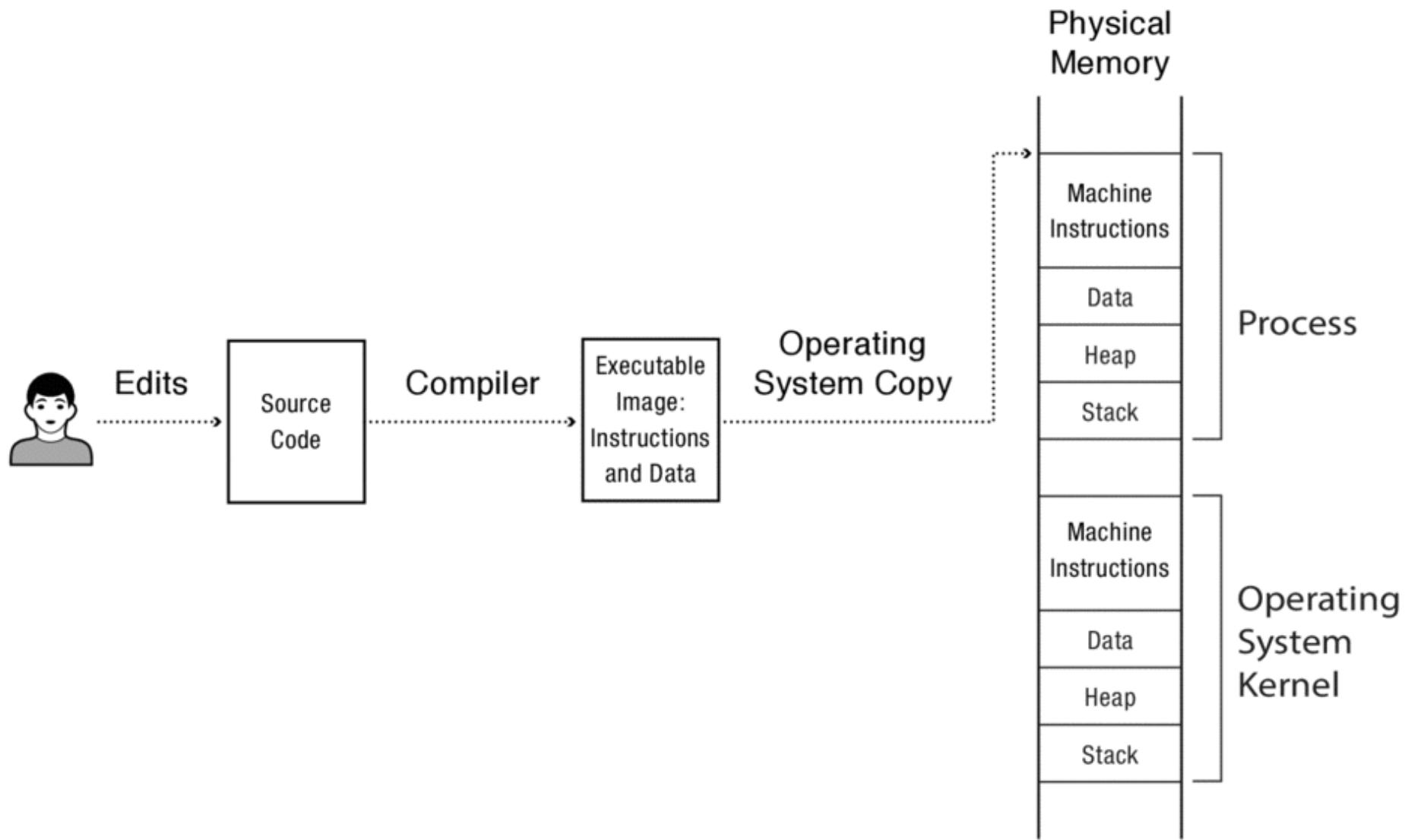
Thought Experiment

- Implementing execution with limited privileges:
 - Execute each program instruction in a simulator
 - If the instruction is permitted, do the instruction
 - Otherwise, stop the process
 - Basic model in Javascript, ...
- How do we go faster?
 - Run the unprivileged code directly on the CPU?

Main Points

- Process concept
 - A process is an OS abstraction for executing a program with limited privileges
- Dual-mode operation: *user* vs. *kernel mode*
 - *Kernel-mode*: execute with complete privileges
 - *User-mode*: execute with fewer privileges
- Safe control transfer
 - How do we switch from one mode to the other?

Process Concept



Process Concept

- Process: a sequence of activities activated by a program, **running with limited rights**
 - Process control block (PCB): the data structure the OS uses to keep track of a process
 - Process Table: contains all PCBs
 - Two elements:
 - Thread: executes a sequence of instructions within a process
 - Potentially many threads per process (for now 1:1)
 - Thread aka lightweight process
 - Address space: set of rights of a process
 - Memory that the process can access
 - Other permissions the process has (e.g., which procedure calls it can make, what files it can access)

Program and Process

- Program: a static sequence of instructions
- Process:
 - a dynamic entity: *a sequence of activities described by a program*
 - executed on a set of CPUs with limited rights
- Several processes can be activated on the same program
 - The processes execute the same code
 - Each process executes the program on different data and/or in different times

Process Control Block (PCB)

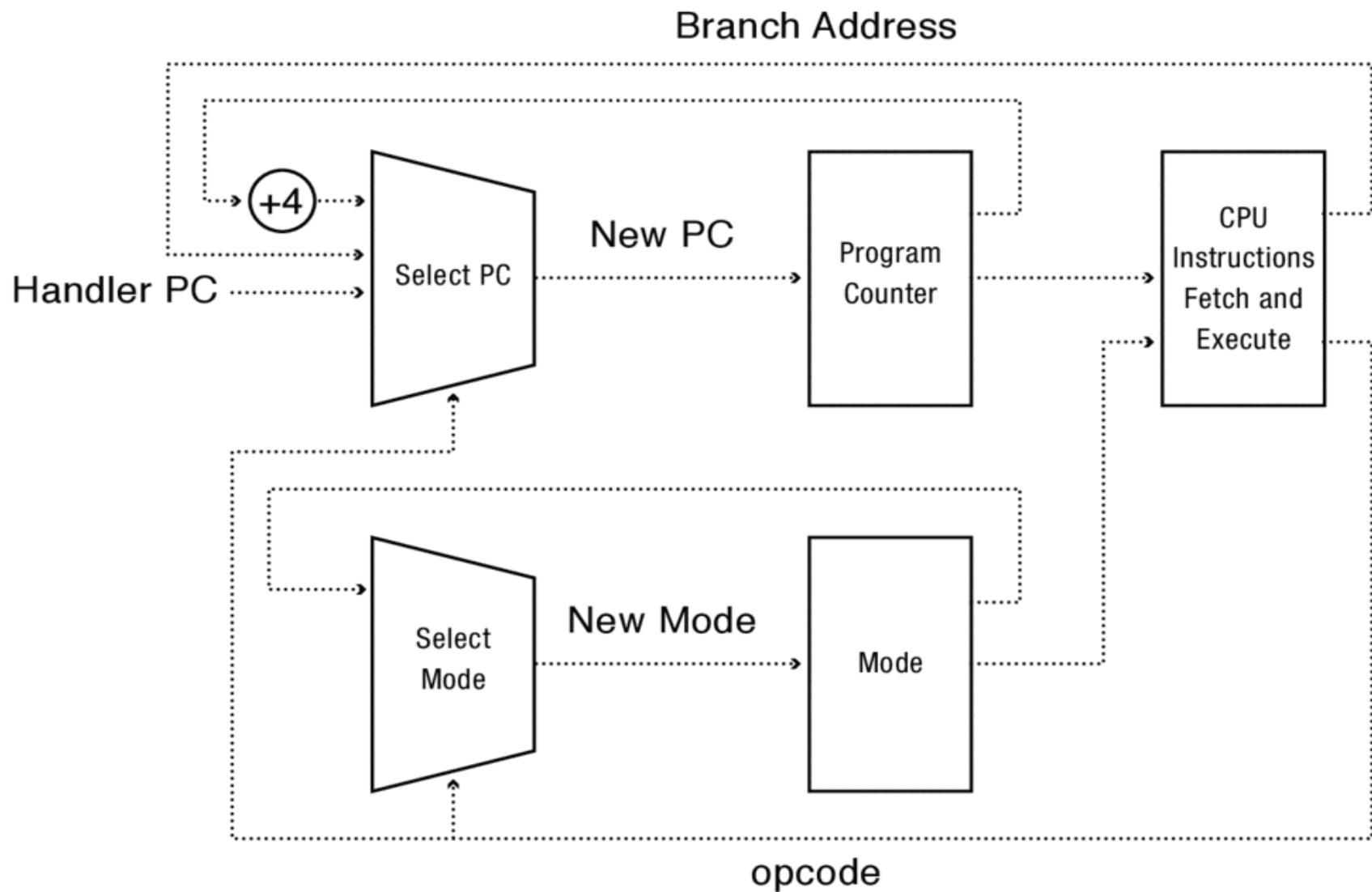
It is a data structure:

- Process name
 - Can be the index of the PCB in the process table
- Process state (new, ready, running, stopped,)
- CPU registers and scheduling information
- Pointers to process threads
- Assigned memory (memory limits)
- Other assigned resources
 - Open Files, devices, I/O status, etc...

Hardware Support: Dual-Mode Operation

- Kernel mode
 - Execution with the full privileges of the hardware
 - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet
- User mode
 - Limited privileges
 - Only those granted by the operating system kernel
- On the x86, mode stored in EFLAGS register
- On the ARM, mode stored in CPSR register
 - In general, in the *Program Status Register* (PSR/PSW)

A CPU with Dual-Mode operation



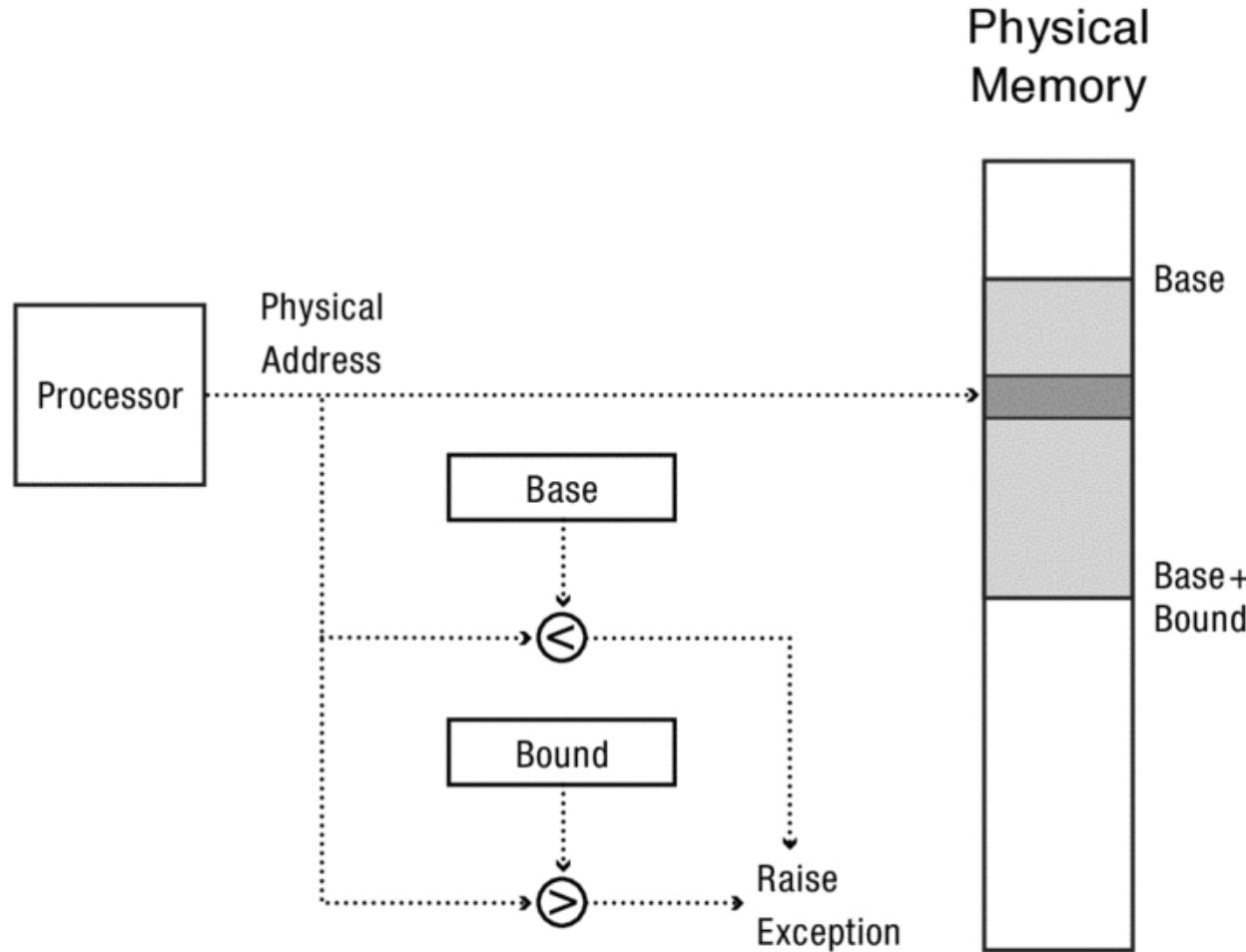
Hardware Support: Dual-Mode Operation

- Privileged instructions
 - Available to kernel
 - Not available to user code
- Limits on memory accesses (memory protection)
 - To prevent user code from overwriting the kernel
- Timer
 - To regain control from a user program in a loop
- Safe way to switch from user mode to kernel mode, and vice versa

Privileged instructions

- Examples?
 - Disabling interrupts
 - Setting the timer
 - Changing the bits in the PSW (CPSR)
 - ...
- What should happen if a user program attempts to execute a privileged instruction?
 - The OS takes over

Base-Bound Memory Protection with Physical addresses

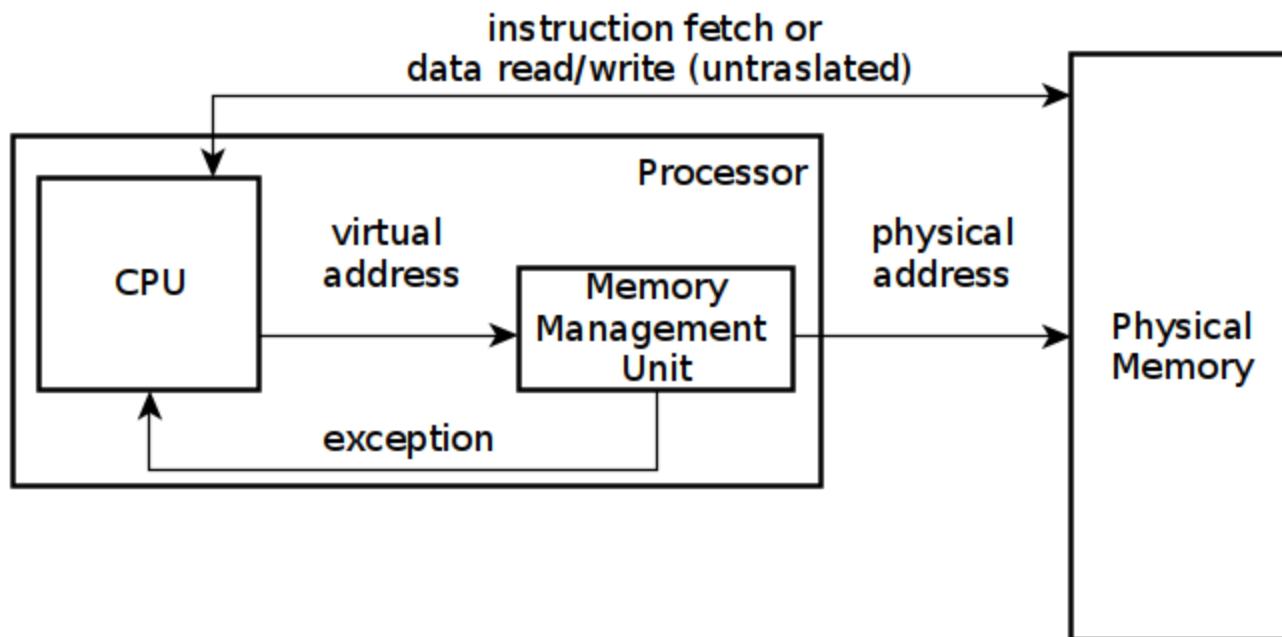


Towards Virtual Addresses

- Problems with base and bounds?
- No or limited expandable heap and stack
- No memory sharing
- Need to work with physical addresses -> hard to move memory around
- Memory fragmentation

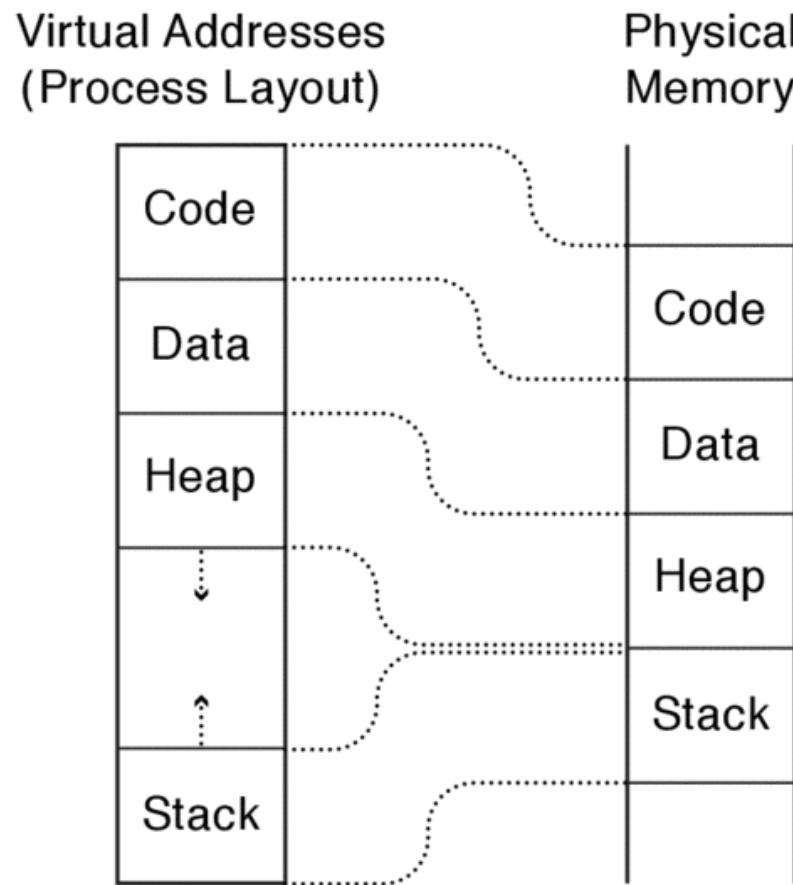
Virtual Addresses

- Translation done in hardware, using a table
- Table set up by the kernel
- Usually, the MMU takes care of address translation



Virtual Address Layout

- Plus shared code segments, dynamically linked libraries, memory mapped files, ...



Example

- What if we run two instances of this program at the same time?

```
int staticVar = 0;    // a static variable
main() {
    int localVar = 0; // a procedure local variable
    staticVar += 1; localVar += 1;
    sleep(10); // sleep causes the program to wait for 10 seconds
    printf ("static address: %p, value: %d\n", &staticVar, staticVar);
    printf ("local address: %p, value: %d\n", &localVar, localVar);
}
```

Produces (*):

static address: 0x100407000, value: 1

local address: 0xfffffc3c, value: 1

(*) Because of the *Address Space Layout Randomization* (ASLR) the output can be different at each run.....

Question

- Suppose we had a perfect object-oriented language and compiler, so that only an object's methods could access the internal data inside an object. If the operating system only ran programs written in that language, would it still need hardware memory address protection?

Hardware Timer

- Hardware device that periodically interrupts the processor
 - Returns control to the *kernel timer interrupt handler*
 - Interrupt frequency set by the kernel
 - Not by user code!
 - Interrupts can be temporarily deferred
 - Not by user code!
 - Interrupt deferral crucial for implementing mutual exclusion (we will see it later on)

Mode Switch

- From user-mode to kernel
 - Interrupts
 - Triggered by timer and I/O devices
 - Exceptions
 - Triggered by unexpected program behavior
 - Or malicious behavior!
 - System calls (aka protected procedure call)
 - Request by program for kernel to do some operation on its behalf
 - Only limited # of very carefully coded entry points
 - Also called Software Interrupts (SWIs)

Mode Switch

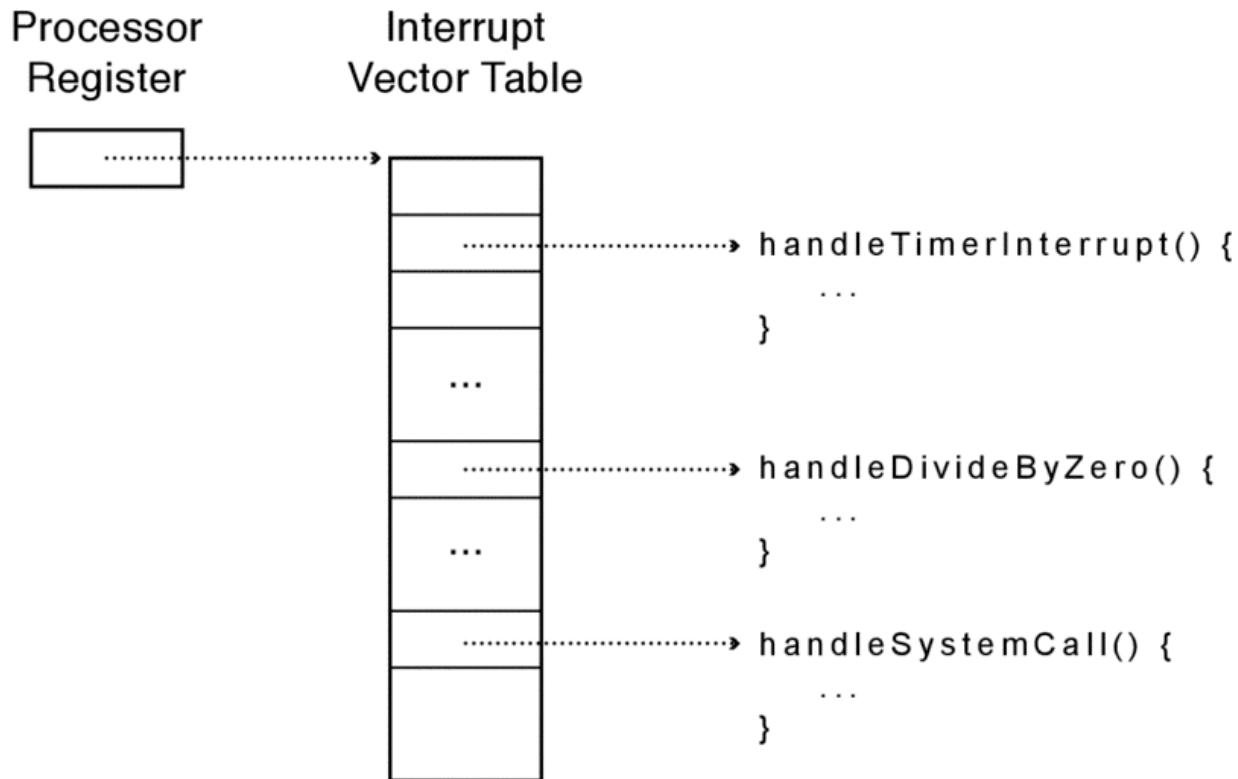
- From kernel-mode to user-mode
 - Return from interrupt, exception, system call
 - Resume suspended execution
 - New process/new thread start
 - Jump to first instruction in program/thread
 - Process/thread context switch
 - Resume some other process
 - User-level upcall
 - Asynchronous notification to user program

How do we take interrupts safely?

- Interrupt vector
 - Limited number of entry points into kernel
- Kernel interrupt stack
 - Handler works regardless of state of user code
- Interrupt masking
 - Handler is non-blocking
- Atomic transfer of control
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
- Transparent restartable execution
 - User program does not know interrupt occurred

Interrupt Vector

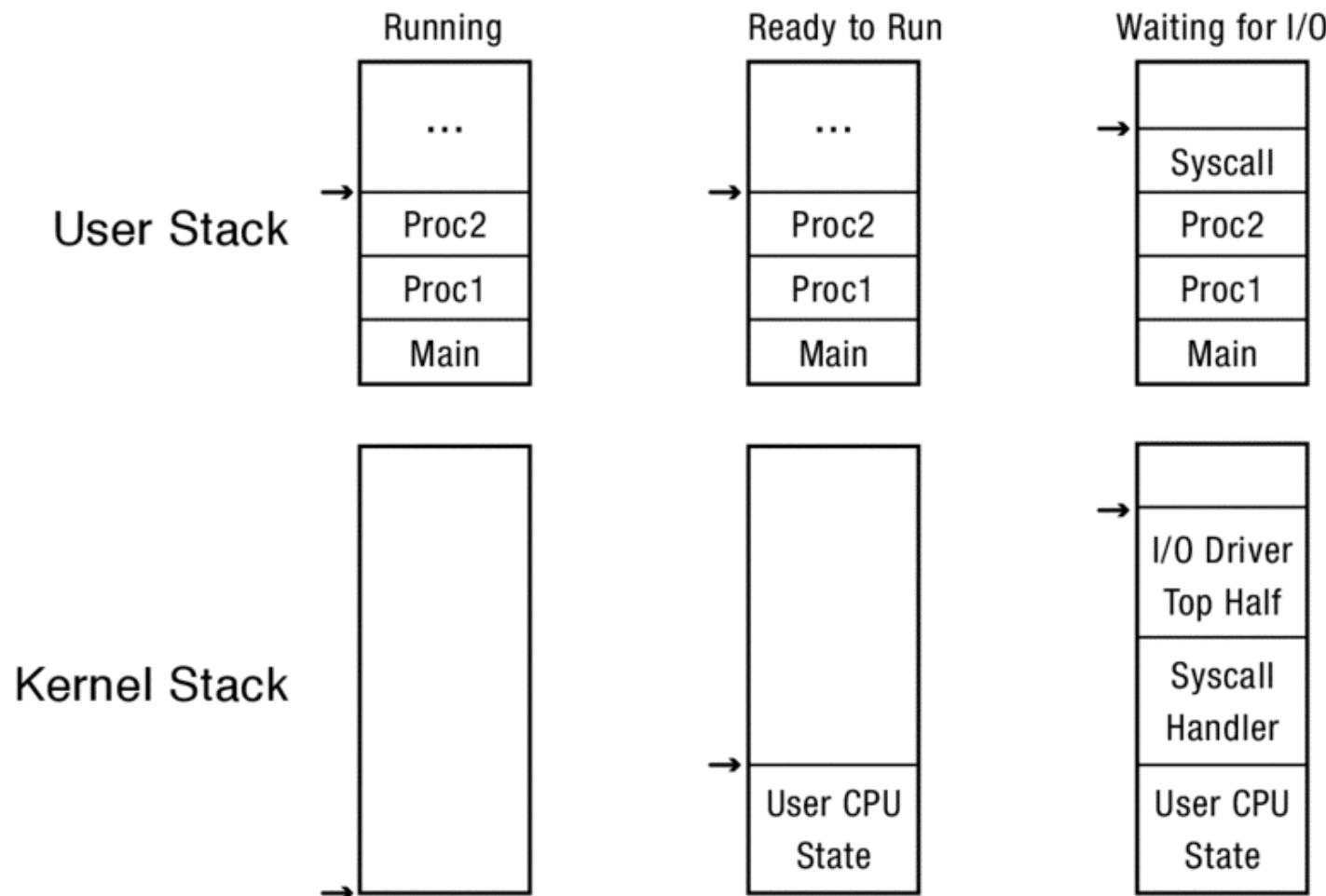
- Table set up by OS kernel; pointers to code to run on different events



Interrupt Stack

- Per-processor, located in kernel (not user) memory
 - Usually a thread has both: kernel and user stack
- Why can't interrupt handler run on the stack of the interrupted user process?

Interrupt Stack



Interrupt Masking

- Interrupt handler runs with interrupts off
 - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
 - Eg., when determining the next process/thread to run
 - If defer interrupts too long, can drop I/O events
 - On x86
 - CLI: disable interrupts
 - STI: enable interrupts
 - Only applies to the current CPU
- Cf. implementing synchronization, chapter 5

Interrupt Handlers

- Non-blocking, run to completion
 - Minimum necessary to allow device to take next interrupt
 - Any waiting must be of limited duration
 - Wake up other threads to do any real work
- Rest of device driver runs as a kernel thread
 - Queues work for interrupt handler
 - (Sometimes) wait for interrupt to occur

Atomic Mode Transfer

- On interrupt (x86)

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Switch to handler, PC & kernel PSW
- Interrupt handler saves registers it might clobber
- Interrupt management

in hardware

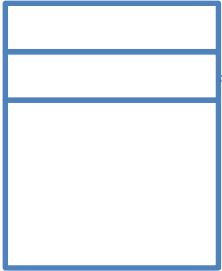
Before

User-level
process

Code:

```
Foo() {  
    while(...) {  
        x=x+1;  
        y=y-2;  
    }  
}
```

Stack:



Registers: (x86 names):

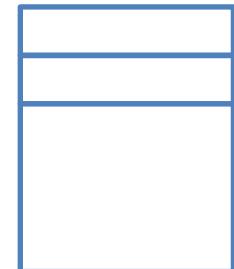
SP	SS:ESP
PC	CS:EIP
PSW	EFLAGS
General registers	

Kernel

Code:

```
handler() {  
    push A  
    ...  
}
```

Exception
Stack:



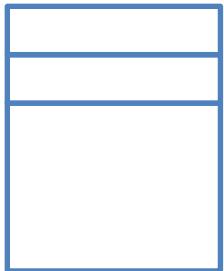
During

User-level
process

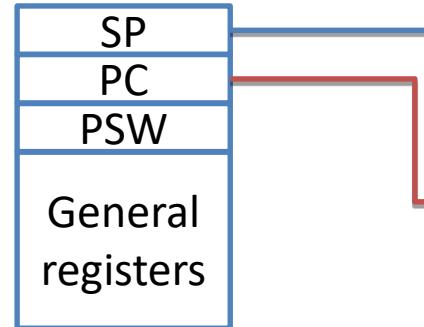
Code:

```
Foo() {  
    while(...) {  
        x=x+1;  
        y=y-2;  
    }  
}
```

Stack:



Registers:

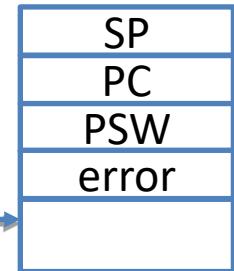


Kernel

Code:

```
handler() {  
    pusha  
    ...  
    popa  
    IRET  
}
```

Exception
Stack:



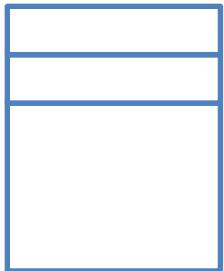
During

User-level
process

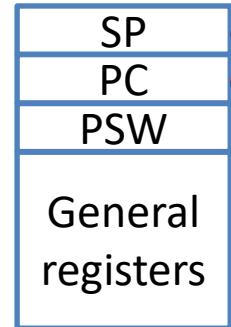
Code:

```
Foo() {  
    while(...) {  
        x=x+1;  
        y=y-2;  
    }  
}
```

Stack:



Registers:

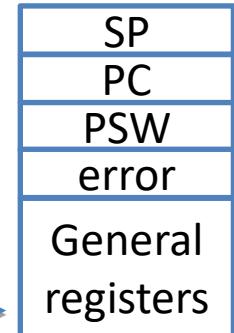


Kernel

Code:

```
handler() {  
    pusha  
    ...  
    popa  
    IRET  
}
```

Exception
Stack:



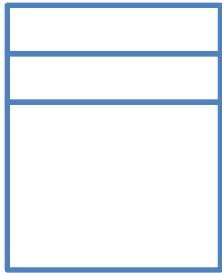
After

User-level
process

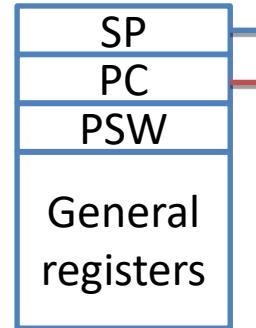
Code:

```
Foo() {  
    while(...) {  
        x=x+1;  
        y=y-2;  
    }  
}
```

Stack:



Registers:

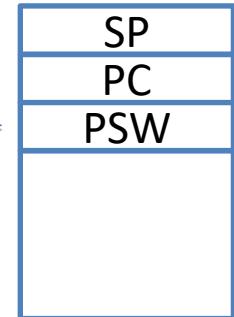


Kernel

Code:

```
handler() {  
    pusha  
    ...  
    popa  
    IRET  
}
```

Exception
Stack:



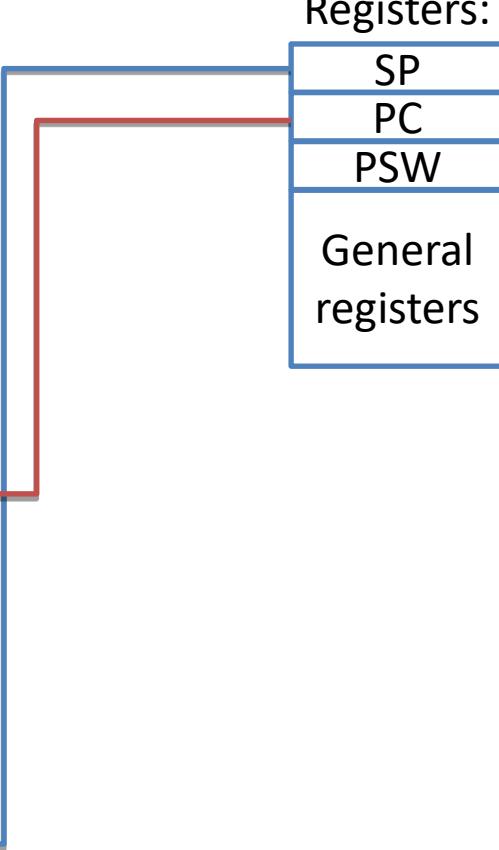
... and after

User-level
process

Code:

```
Foo() {  
    while(...) {  
        x=x+1;  
        y=y-2;  
    }  
}
```

Stack:



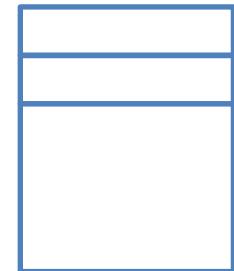
Kernel

Code:

```
handler() {  
    pusha  
    ...  
    popa  
    IRET  
}
```

Exception

Stack:



At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread (IRET instruction in the previous example)
 - Restore program counter
 - Restore program stack
 - Restore processor status word/condition codes
 - Switch to user mode
 - Enable interrupts

A (very) simplified architecture

- Let's suppose a 16-bit processor with a few general registers **R1..Rn**, and three status and control registers **PC**, **SP**, and **PS**
- One single kernel stack for managing interrupts
- Interrupt checked at the end of the fetch-execute cycle (PC already incremented)
- IRET instruction to return from the interrupt handler. It pops PC, SP, and PSW out from the kernel stack and restores them atomically!

Interrupt management: a simplified example

Initial state: interrupt '500' occurs when executing instruction A000

Registri nella CPU

PC	A000
PS	PSW P
SP	FFFC
R1	AAAA
R2	BBBB
...	

Programma P

...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A00C	istr. 4
A010	istr. 5
...	

Memoria

Stack del kernel

2FF8	
2FFC	
3000	
3004	
3008	
300C	

Interrupt Handler

100	Store
104	Save
108	Registers
...	
200	Restores
204	Registers
208	IRET

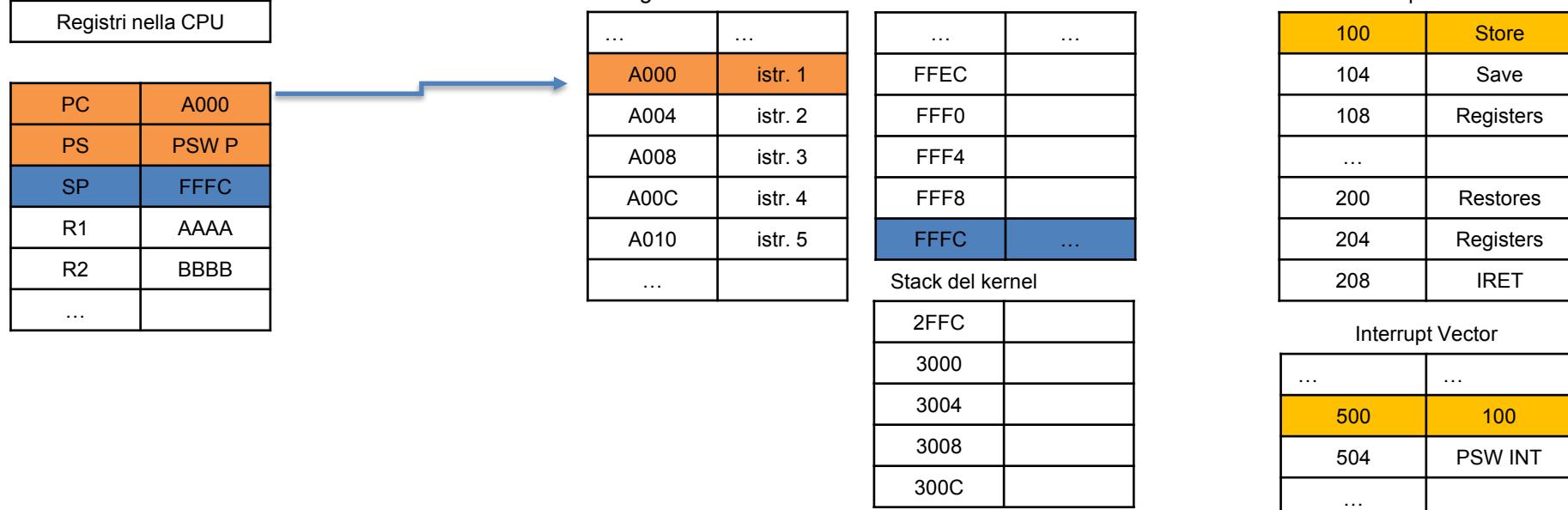
Stack di P

...	...
FFEC	
FFF0	
FFF4	
FFF8	
FFFC	...

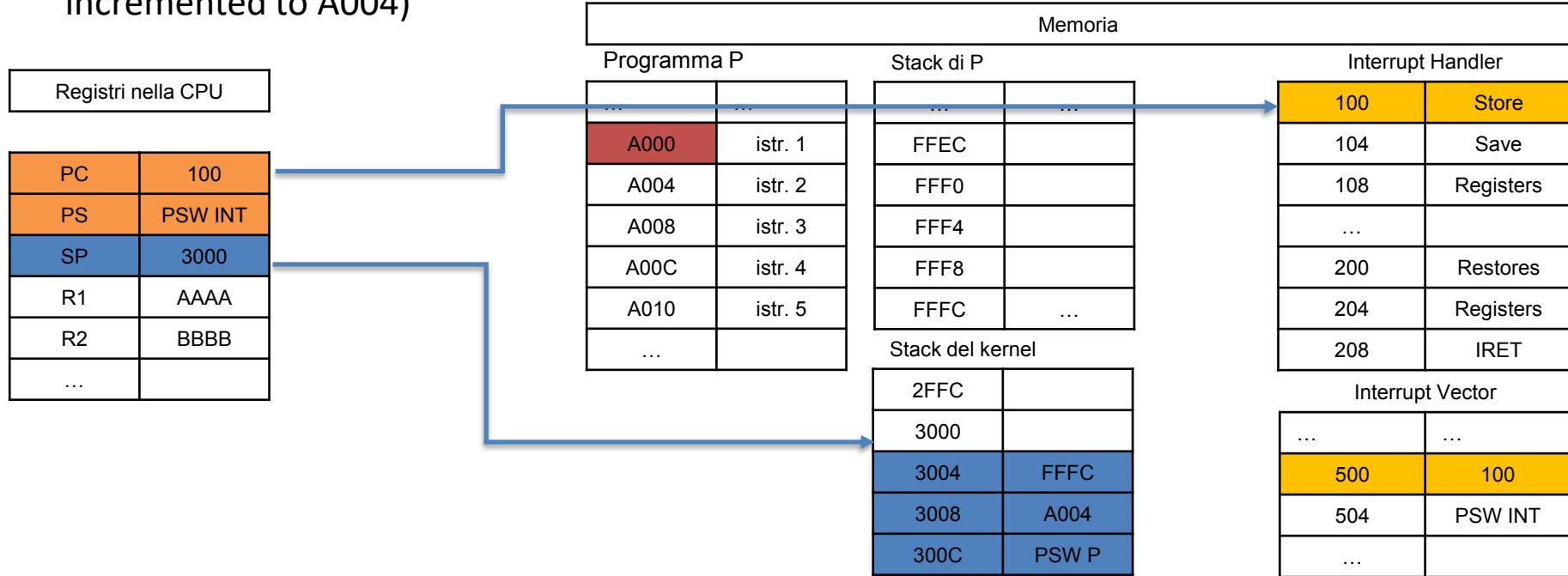
Interrupt Vector

...	...
500	100
504	PSW INT
...	

1) Initial state



2) Interrupt recognized after instruction A000 (PC incremented to A004)



2) Interrupt recognized after instruction A000

Registri nella CPU	
PC	100
PS	PSW INT
SP	3000
R1	AAAA
R2	BBBB
...	

Memoria					
Programma P		Stack di P		Interrupt Handler	
...	100	Store
A000	istr. 1	FFEC		104	Save
A004	istr. 2	FFF0		108	Registers
A008	istr. 3	FFF4		...	
A00C	istr. 4	FFF8		200	Restores
A010	istr. 5	FFFC	...	204	Registers
...		2FFC		208	IRET
Stack del kernel					
2FFC		3000		Interrupt Vector	
3000		3004	FFFC
3004		3008	A004	500	100
3008		300C	PSW P	504	PSW INT
300C		

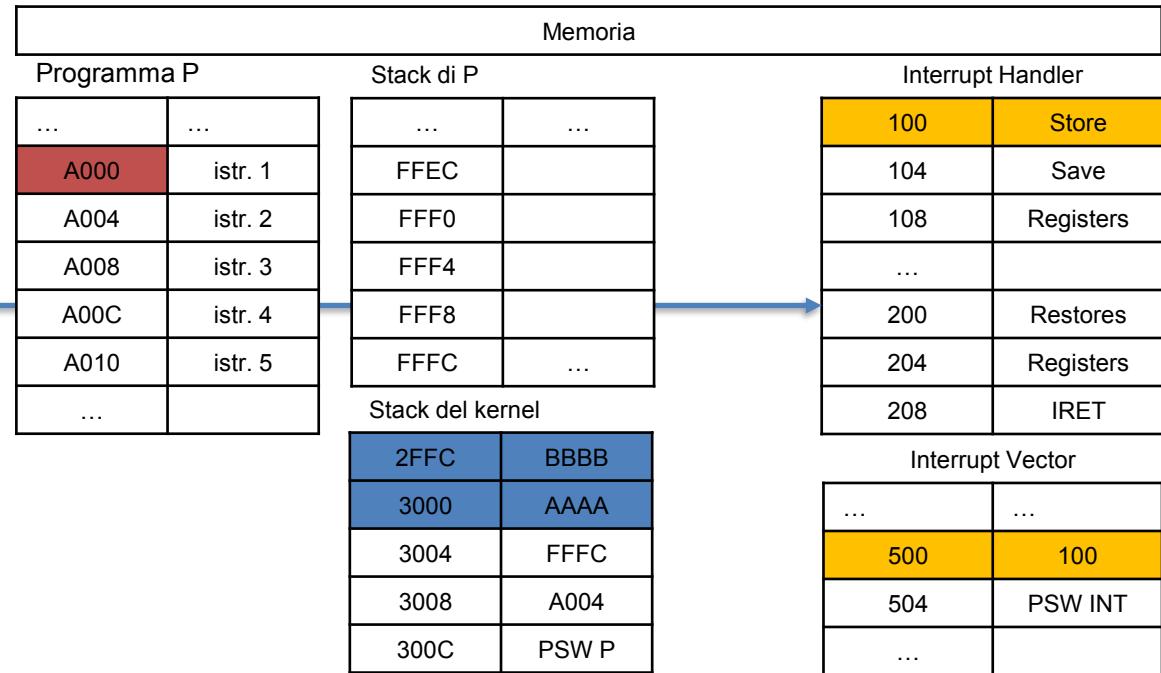
3) Stores general registers

Registri nella CPU	
PC	112
PS	PSW INT
SP	2FF8
R1	AAAA
R2	BBBB
...	

Memoria					
Programma P		Stack di P		Interrupt Handler	
...	100	Store
A000	istr. 1	FFEC		104	Save
A004	istr. 2	FFF0		108	Registers
A008	istr. 3	FFF4		...	
A00C	istr. 4	FFF8		200	Restores
A010	istr. 5	FFFC	...	204	Registers
...		2FFC	BBBB	208	IRET
Stack del kernel					
2FFC	BBBB	3000	AAAA	Interrupt Vector	
3000	AAAA	3004	FFFC
3004	FFFC	3008	A004	500	100
3008	A004	300C	PSW P	504	PSW INT
300C	PSW P	

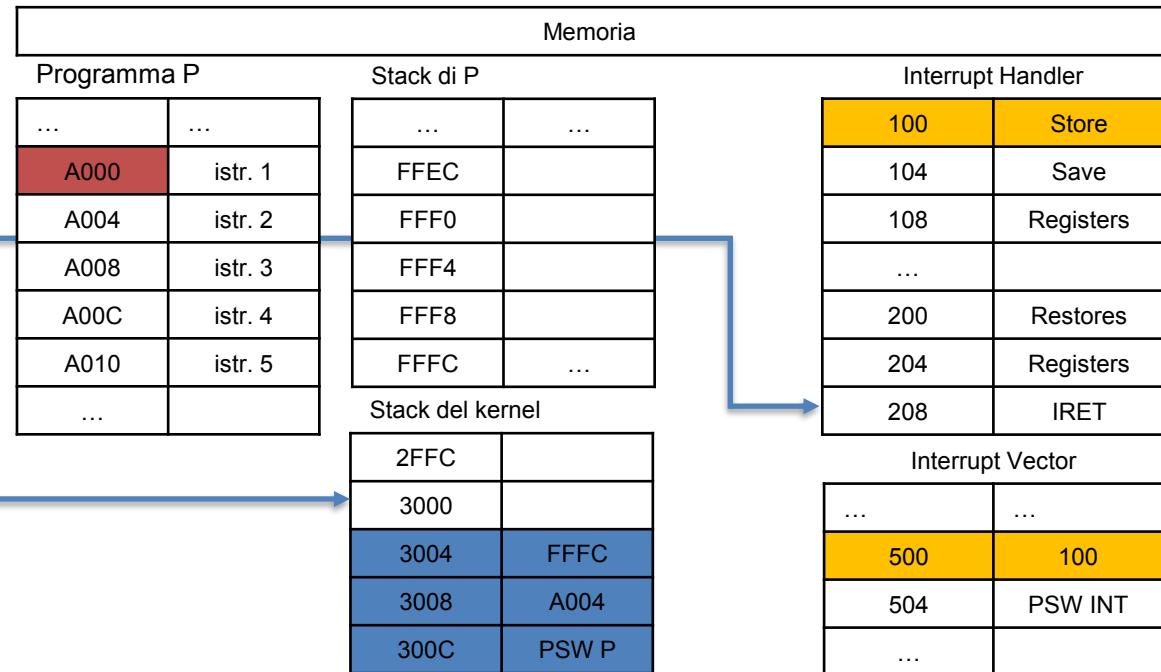
4) Executes interrupt handler

Registri nella CPU	
PC	200
PS	PSW INT
SP	2FF8
R1	??
R2	??
...	



5) Restores general registers

Registri nella CPU	
PC	208
PS	PSW INT
SP	3000
R1	AAAA
R2	BBBB
...	



5) Restores general registers

Registri nella CPU	
PC	208
PS	PSW INT
SP	3000
R1	AAAA
R2	BBBB
...	

Memoria	
Programma P	
...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A00C	istr. 4
A010	istr. 5
...	

Stack di P	
...	...
FFEC	
FFF0	
FFF4	
FFF8	
FFFC	...

Interrupt Handler	
100	Store
104	Save
108	Registers
...	
200	Restores
204	Registers
208	IRET

Stack del kernel	
2FFC	
3000	
3004	FFFC
3008	A004
300C	PSW P

Interrupt Vector	
...	...
500	100
504	PSW INT
...	

6) Executes IRET

Registri nella CPU	
PC	A004
PS	PSW P
SP	FFFC
R1	AAAA
R2	BBBB
...	



Memoria	
Programma P	
...	...
A000	istr. 1
A004	istr. 2
A008	istr. 3
A00C	istr. 4
A010	istr. 5
...	

Stack di P	
...	...
FFEC	
FFF0	
FFF4	
FFF8	
FFFC	...

Interrupt Handler	
100	Store
104	Save
108	Registers
...	
200	Restores
204	Registers
208	IRET

Stack del kernel	
2FFC	
3000	
3004	
3008	
300C	

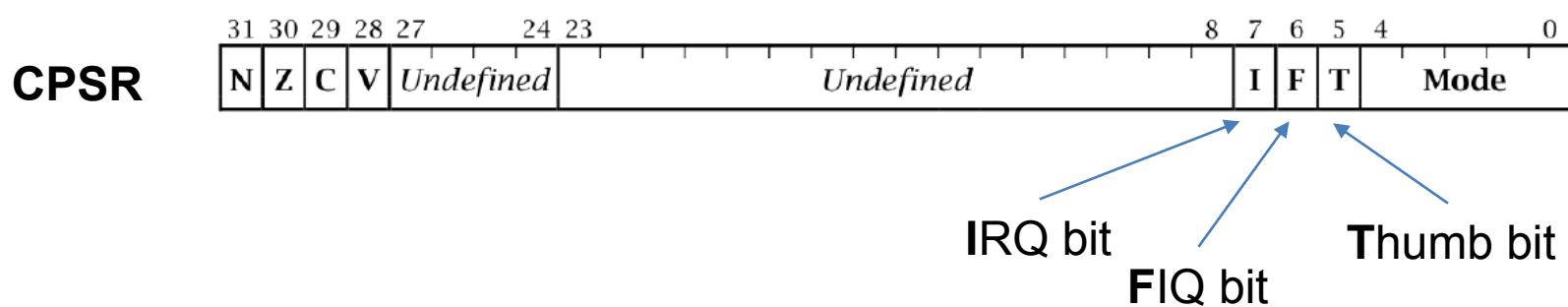
Interrupt Vector	
...	...
500	100
504	PSW INT
...	

Exception handling in ARM

- ARM processor has 6 modes of operation
- Switching between modes can be done by modifying the mode bits in the CPSR register
- User-mode operates at PL0, the other modes at PL1
- All system resources can be accessed in PL1 mode
- In PL1 mode, some banked registers are used to execute the exception handler

Exception mode and CPSR

Mode	CPSR _{4:0}
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001



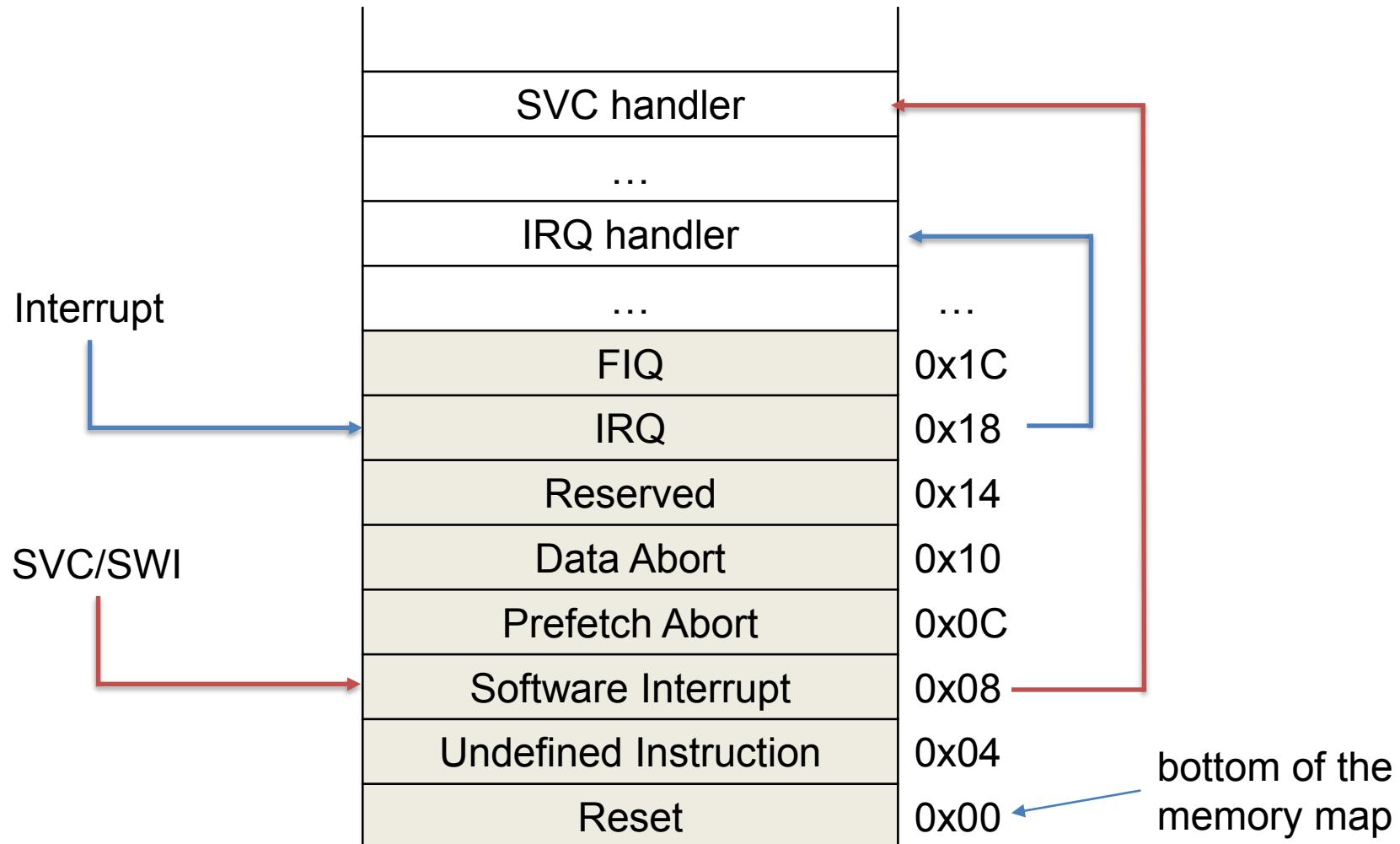
Exception vector table

Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

Priority (1 high, 6 low):

- 1 Reset
- 2 Data Abort
- 3 FIQ
- 4 IRQ
- 5 Prefetch Abort
- 6 Undefined Instruction and Software Interrupt (SWI)

Exception vector table



- Each entry has only 32 bits and contains: a ***branch instruction*** or ***load PC instruction*** to the actual handler

Banked registers

User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12
R13 (SP)	R13 (SP)	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14 (LR)	R14 (LR)	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

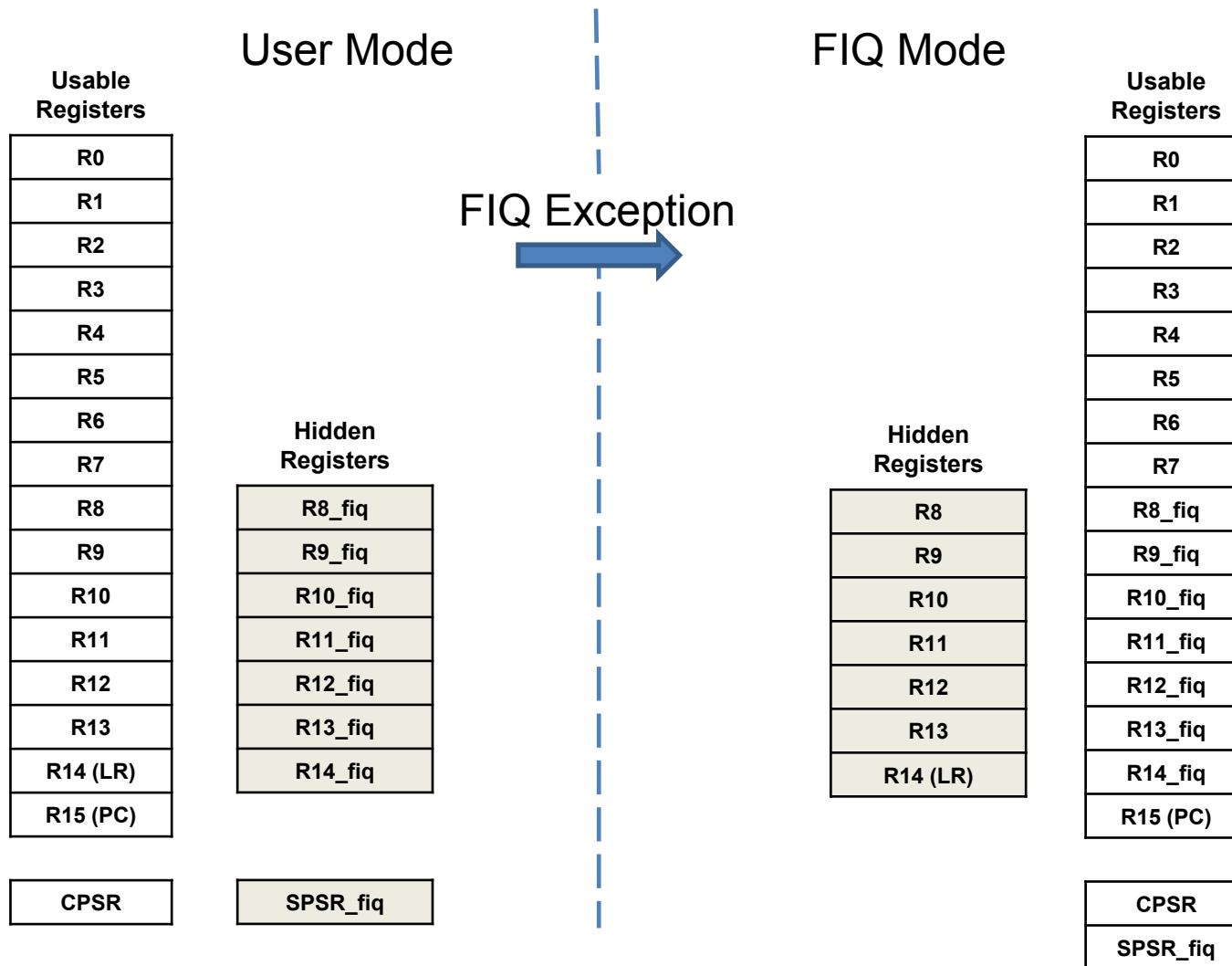
- 37 registers in total (31 general purpose, 6 control registers)
- In PL1, SP LR and CPSR are banked; FIQ mode has 5 additional banked registers

Response to an Exception

- Set the banked LR register to the return address (e.g., PC+4)
- Copies the CPSR into the banked SPSR
 - This save the current mode, the interrupt mask and condition flags
- Change the appropriate CPSR mode bits
- Map in the appropriate banked registers for that mode
 - For FIQ mode there are 5 extra banked registers
- Disable interrupts
 - *IRQs are disabled when any exception occurs*
 - FIQs are disabled when FIQ occurs, and *on reset*
- Set the PC to the vector address for the exception handler

Visibility of banked registers:

FIQ example



Exception management (some cases)

Reset:

```
R14_rst = PC
SPSR_rst = CPSR
CPSR[4:0] = 0x10011 // supervisor mode
CPSR[5] = 0 // ARM state
CPSR[6] = 1 // Disable FIQ
CPSR[7] = 1 // Disable IRQ
PC = 0x00000000
```

IRQ:

```
R14_irq = PC+4
SPSR_irq = CPSR
CPSR[4:0] = 0x10010 // IRQ mode
CPSR[5] = 0 // ARM state
// CPSR[6] unchanged
CPSR[7] = 1 // Disable IRQ
PC = 0x00000018
```

SVC:

```
R14_svc = PC+4 // next instruction
SPSR_svc = CPSR
CPSR[4:0] = 0x10011 // supervisor mode
CPSR[5] = 0 // ARM state
// CPSR[6] unchanged
CPSR[7] = 1 // Disable IRQ
PC = 0x00000008
```

FIQ:

```
R14_fiq = PC+4
SPSR_fiq = CPSR
CPSR[4:0] = 0x10001 // FIQ mode
CPSR[5] = 0 // ARM state
CPSR[6] = 1 // Disable FIQ
CPSR[7] = 1 // Disable IRQ
PC = 0x0000001C
```

Returning from an Exception

- In general, the steps are:
 - Restore the CPSR from the banked SPSR, which restores the execution mode and privilege level
 - Restore the PC from the banked LR
- The above two steps must be simultaneous (i.e., *atomic*)!
- The exception return is carried out by a *data processing instruction* with the ‘S’ flag set, specifically **MOVS** and **SUBS**
 - For example (**if not using the stack**):
 - **MOVS PC, R14_svc** // pc = lr_svc ; cpsr = spsr_svc
 - **SUBS PC, R14_fiq, #4** // pc = lr_fiq-4 ; cpsr = spsr_fiq

Returning from an Exception

- Returning from *SVC*
 - Exception generated by the instruction itself; PC not updated; LR banked already contains PC+4, i.e., next instruction to execute
 - If not using the stack: **MOVS PC, LR_svc**
 - If using the stack:

STMFD SP!, {reglist, LR} // handler entry

....

LDMFD SP!, {reglist, PC}^ // handler exit

(*) The ^ qualifier specifies that the CPSR is restored from the SPSR.
It must be used only from privileged modes

Returning from an Exception

- Returning from FIQ and IRQ
 - Interrupt received only after the execution of an instruction, thus PC already updated to point to the next instruction; LR banked contains PC+4, but PC instruction has not been executed. Therefore, LR must be decremented by 4
 - If not using the stack: **SUBS PC, LR_fiq/irq, #4**
 - If using the stack:

SUB LR, LR, #4 // handler entry

STMFD SP!, {reglist, LR}

....

LDMFD SP!, {reglist, PC}^ // handler exit

Returning from an Exception Summary

	Return instruction	ARM LR_x	THUMB LR_x
SWI	MOVS PC, LR_svc	PC+4	PC+2
UNDEF	MOVS PC, LR_und	PC+4	PC+2
FIQ	SUBS PC, LR_fig, #4	PC+4	PC+4
IRQ	SUBS PC, LR_irq, #4	PC+4	PC+4
PrefetchABT	SUBS PC, LR_abt, #4	PC+4	PC+4
DataABT	SUBS PC, LR_abt, #8	PC+8	PC+8
RESET	-	?	?

Interrupt management example

Step1: Initial state, IRQ occurs when executing instruction 0xA000

Registri nella CPU	
R0	0x0001
..	...
R12	0x0002
SP	0xFFFF4
LR	0xABCD
PC	0xA000
CPSR	User Mode
SPSR	
SP_irq	0xFFFFC
LR_irq	

User stack	
...	...
0xEFFC	
0xF000	
0xF004	
0xF008	??
0xF00C	??

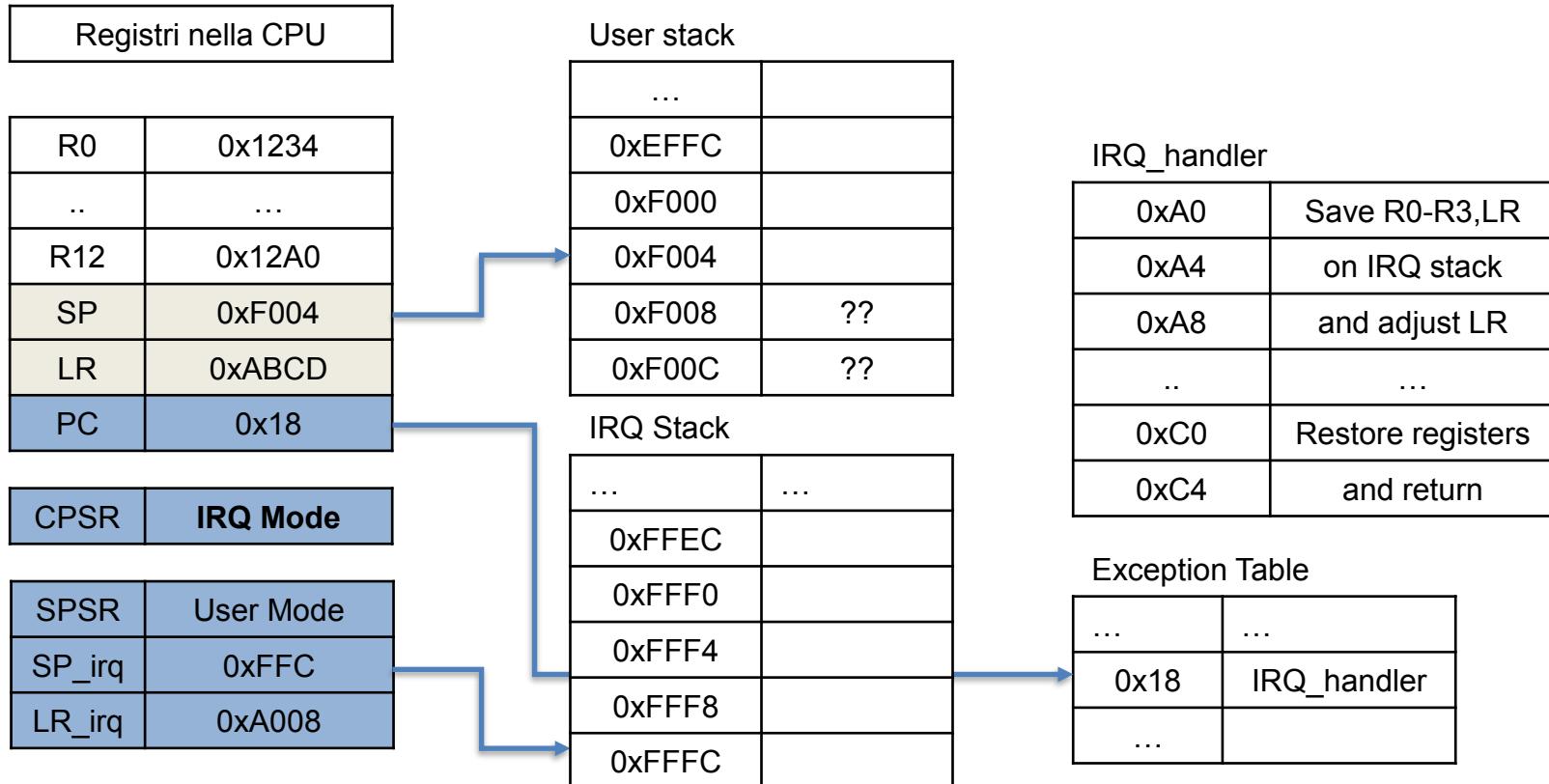
IRQ Stack	
...	
0xFFEC	
0xFFF0	
0xFFF4	
0xFFF8	
0xFFFFC	

IRQ_handler	
0xA0	Save R0-R3,LR
0xA4	on IRQ stack
0xA8	and adjust LR
..	...
0xC0	Restore registers
0xC4	and return

Exception Table	
...	...
0x18	IRQ_handler
...	...

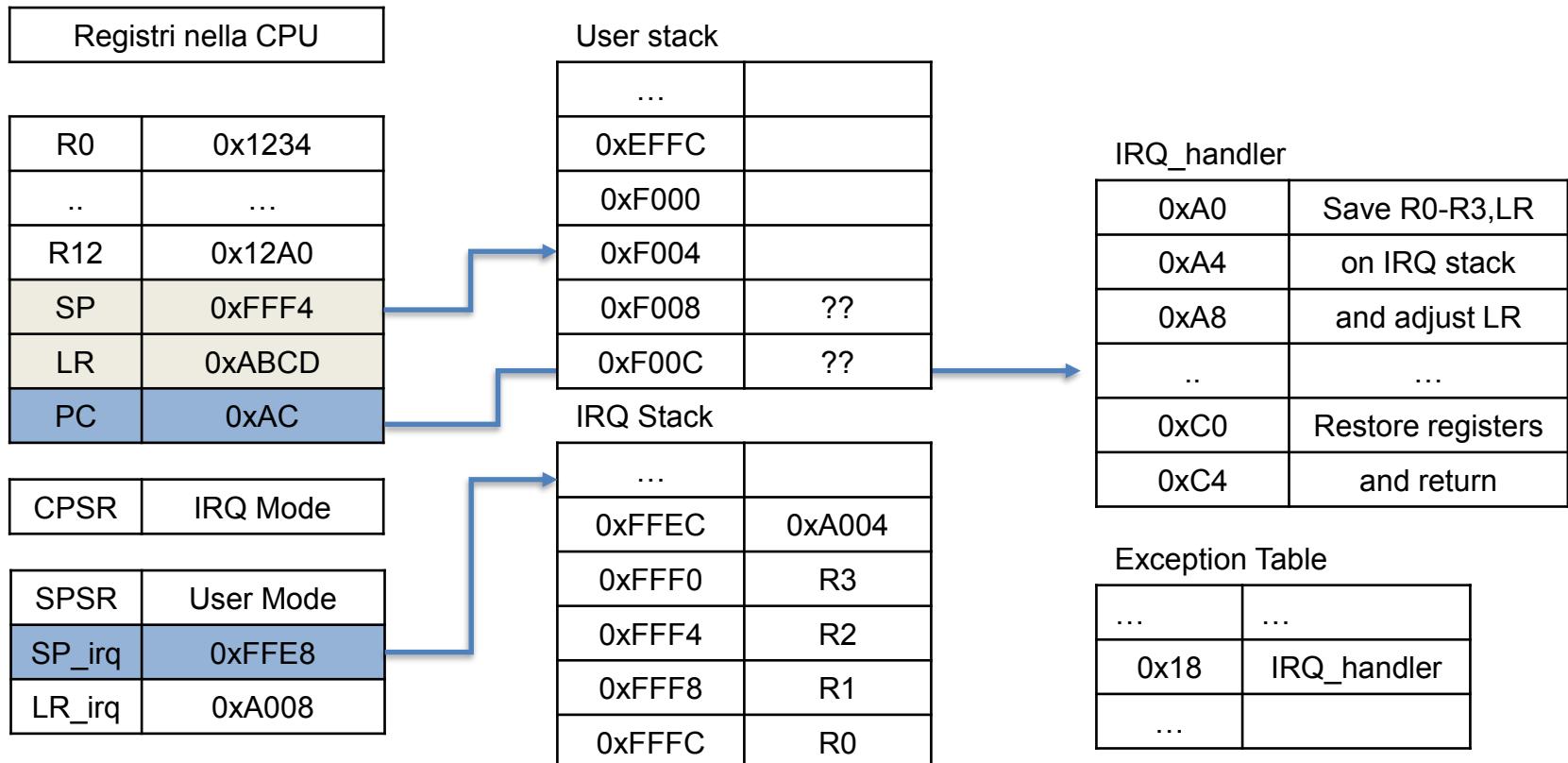
Interrupt management example

Step2: Switching to IRQ Mode, map to banked SP and LR and be prepared to execute the IRQ handler



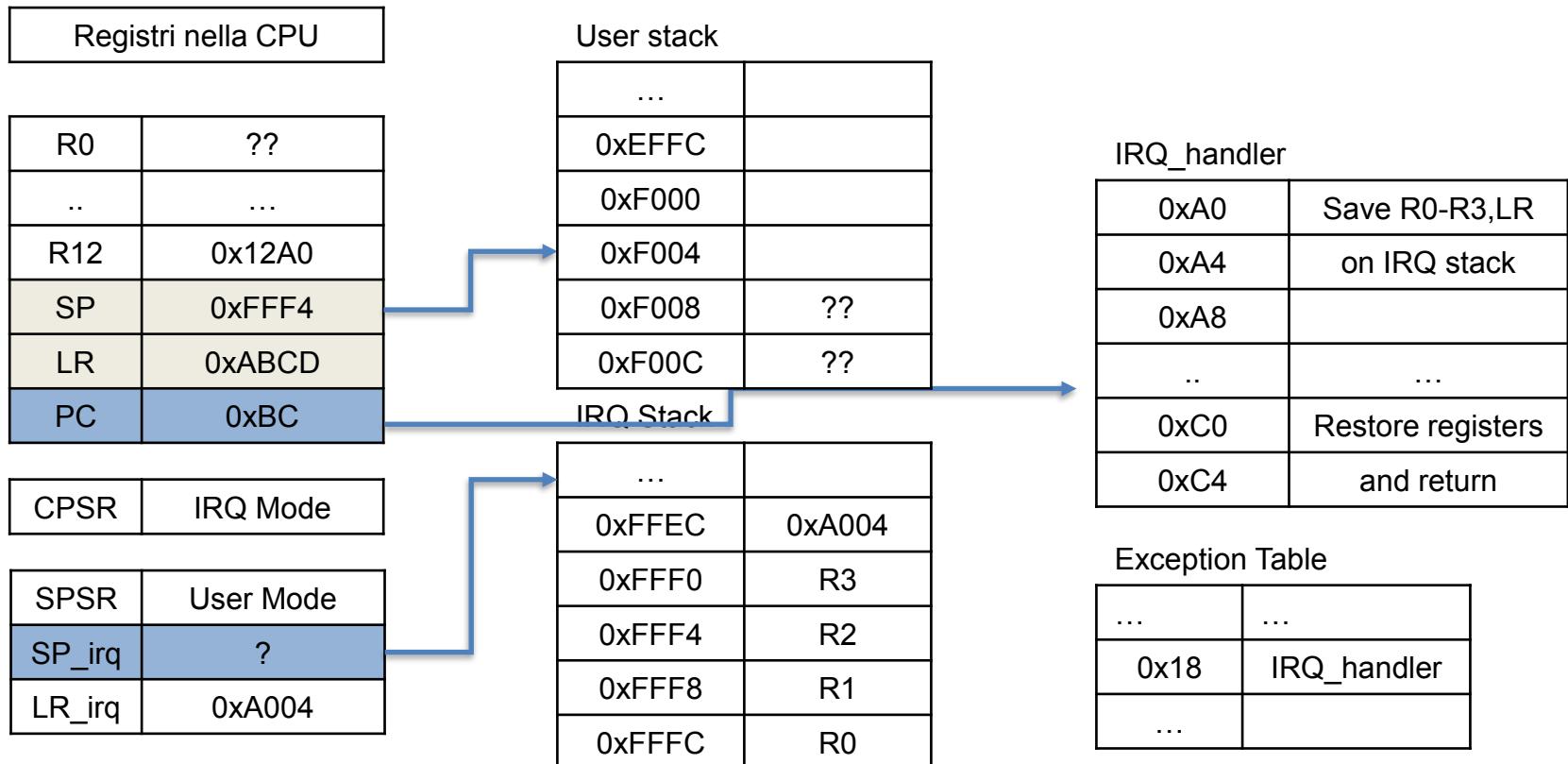
Interrupt management example

Step3: Start executing top-level handler for IRQ



Interrupt management example

Step4: IRQ handled, now we have to restore the state before the IRQ



Interrupt management example

Step4: Restore user registers and switch to User Mode, LR_irq (on the stack) will be copied into

Registri nella CPU

R0	0x1234
..	...
R12	0x12A0
SP	0xFFFF4
LR	0xABCD
PC	0xA004

CPSR	User Mode
------	-----------

SPSR	
SP_irq	0xFFFFC
LR_irq	

User stack

...	
0xEFFC	
0xF000	
0xF004	
0xF008	??
0xF00C	??

IRQ Stack

...	
0xFFEC	
0xFFF0	
0xFFF4	
0xFFF8	
0xFFFFC	

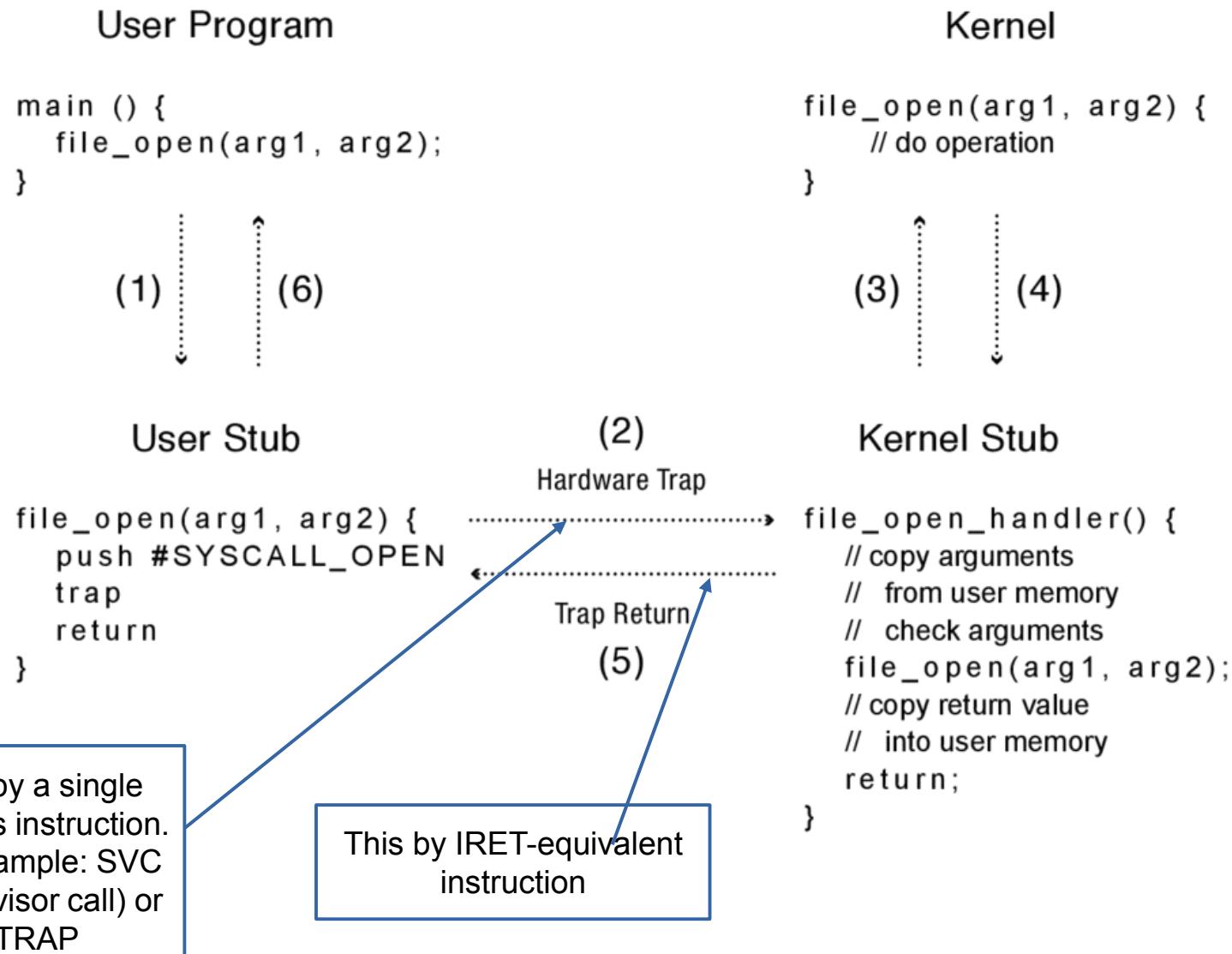
IRQ_handler

0xA0	Save R0-R3,LR
0xA4	on IRQ stack
0xA8	
..	...
0xC0	Restore registers
0xC4	and return

Exception Table

...	...
0x18	IRQ_handler
...	

Software Interrupts aka System Calls



Kernel System Call Handler

- Locate arguments
 - In registers or on user(!) stack
- ***Copy arguments***
 - From user memory into kernel memory
 - Protect kernel from malicious code evading checks
- Validate arguments
 - Protect kernel from errors in user code
- Copy results back
 - into user memory

System Calls in ARM Linux

- The system call handler must know which kind of software interrupt is being called (i.e., SVC number) and where to find the arguments
- man 2 syscall***

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error		
alpha	callsys	v0	v0	a4	a3		
arc	trap0	r8	r0	-	-		
arm/OABI	swi NR	-	a1	-	-		
arm/EABI	swi 0x0	r7	r0	r1	-		
arm64	svc #0	x8	x0	x1	-		
blackfin	excpt 0x0	P0	R0	-	-		
i386	int \$0x80	eax	eax	edx	-		
Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7
alpha	a0	a1	a2	a3	a4	a5	-
arc	r0	r1	r2	r3	r4	r5	-
arm/OABI	a1	a2	a3	a4	v1	v2	v3
arm/EABI	r0	r1	r2	r3	r4	r5	r6
arm64	x0	x1	x2	x3	x4	x5	-
blackfin	R0	R1	R2	R3	R4	R5	-
i386	ebx	ecx	edx	esi	edi	ebp	-

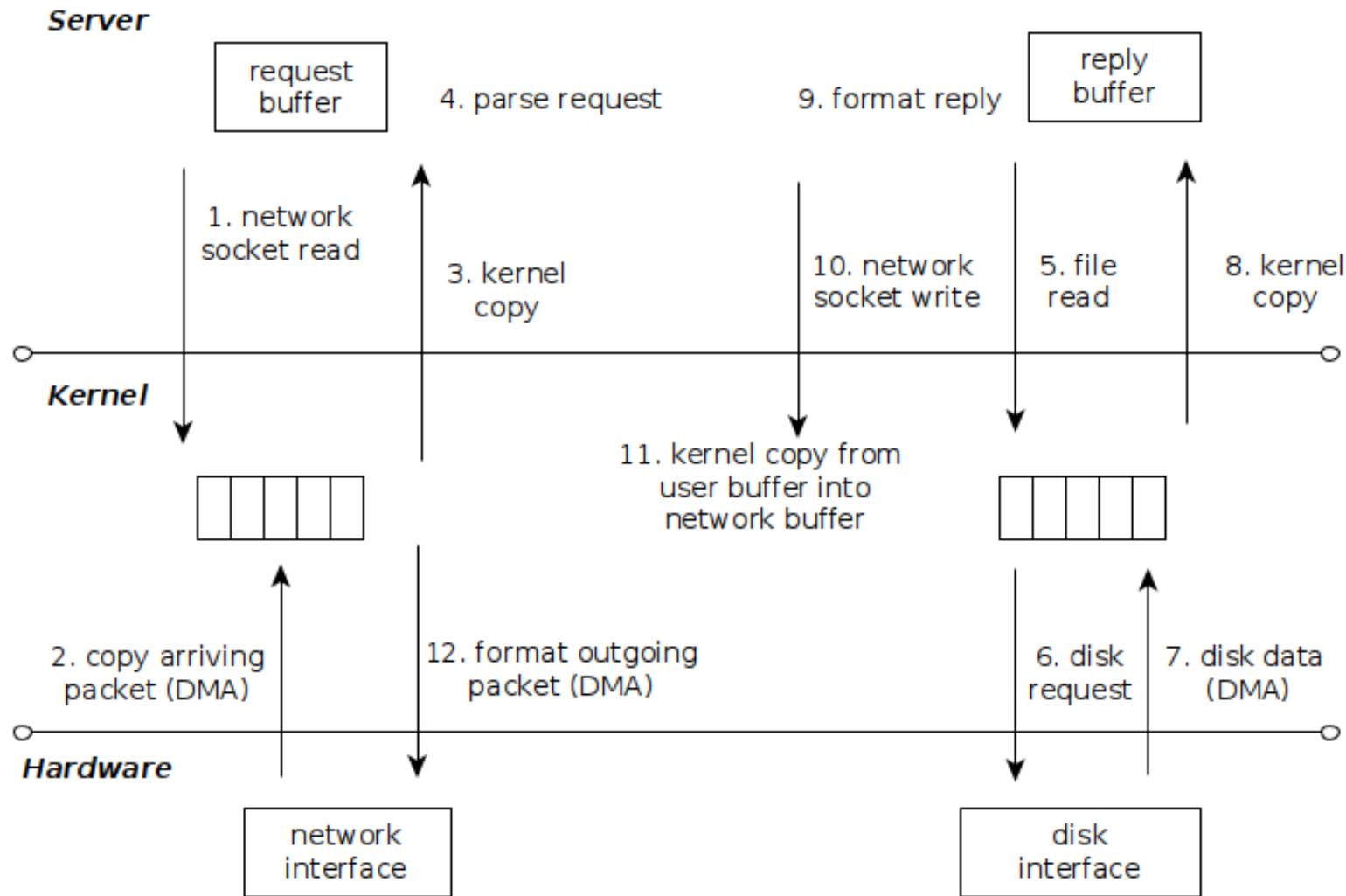
Top-Level SWI handler

```
/* simplified schema of the top-level handler for handling SVCs */
SWI_Handler          ; top-level handler
    STMFD  SP!, {R0-R12, LR}      ; pushes registers into the stack
    ADR R8, SYS_CALL_TABLE       ; load syscall table pointer in R8
    ; R7 contains the SC number
    ADD R7, R7, #_SYSCALL_BASE   ; OS entry of the sys_* routine
    ; sanity checks
    ;
    LDR PC [R8, R7, LSL #2]      ; call sys_* routing
    ; ...
    ; result of the SVC stored in R0
    ; ...
    ; restore user registers
    ;
    MOVS PC, LR                 ; return from handler restoring CPSR
```

Do applications use System Calls?

- Yes, they do!
- Reading/Writing to disks, network cards, screens,, all these ops require system calls.
- Many daily-life applications make large use of system calls
 - For example: a web server
 - Other examples?

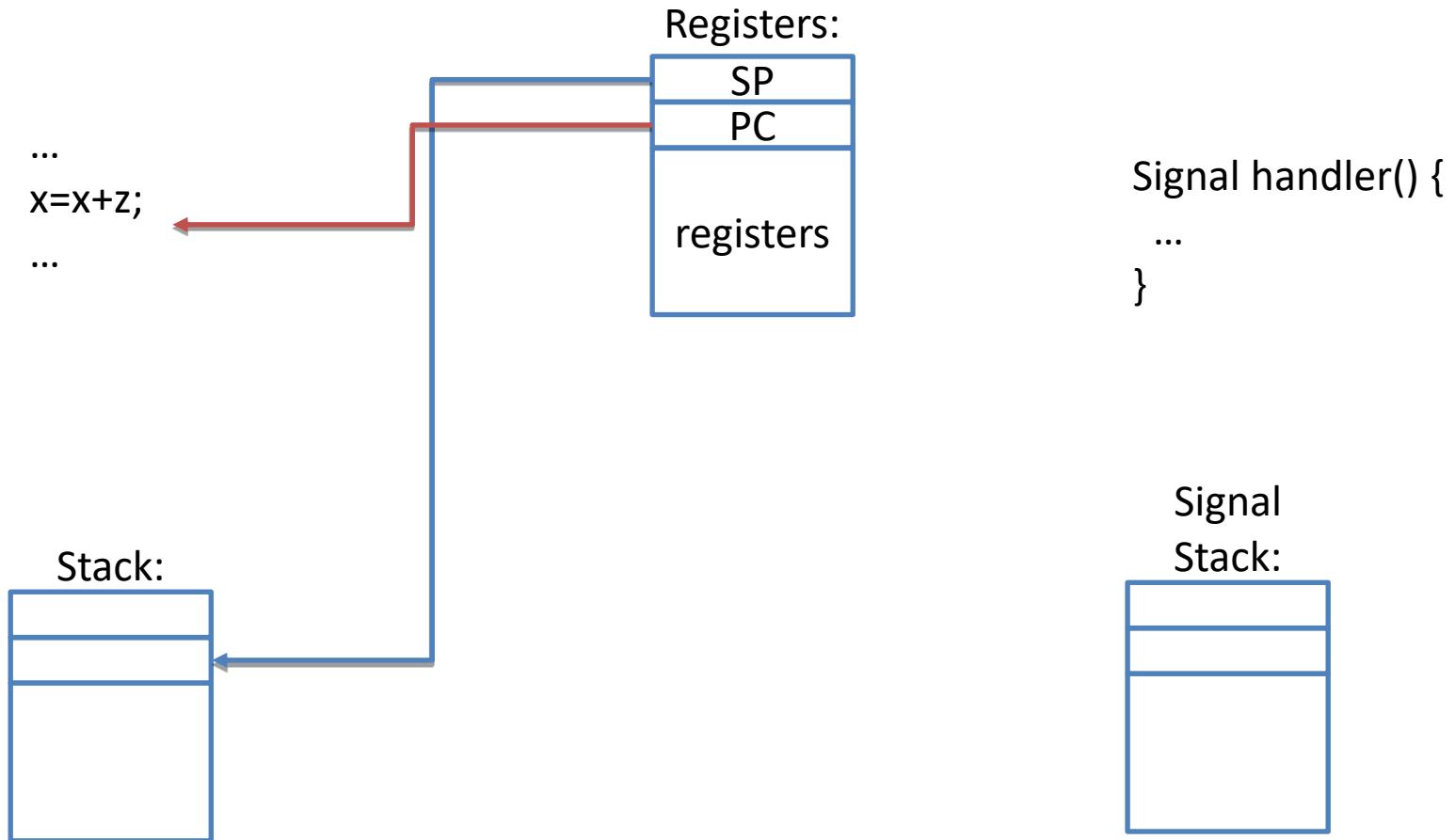
Web Server Example



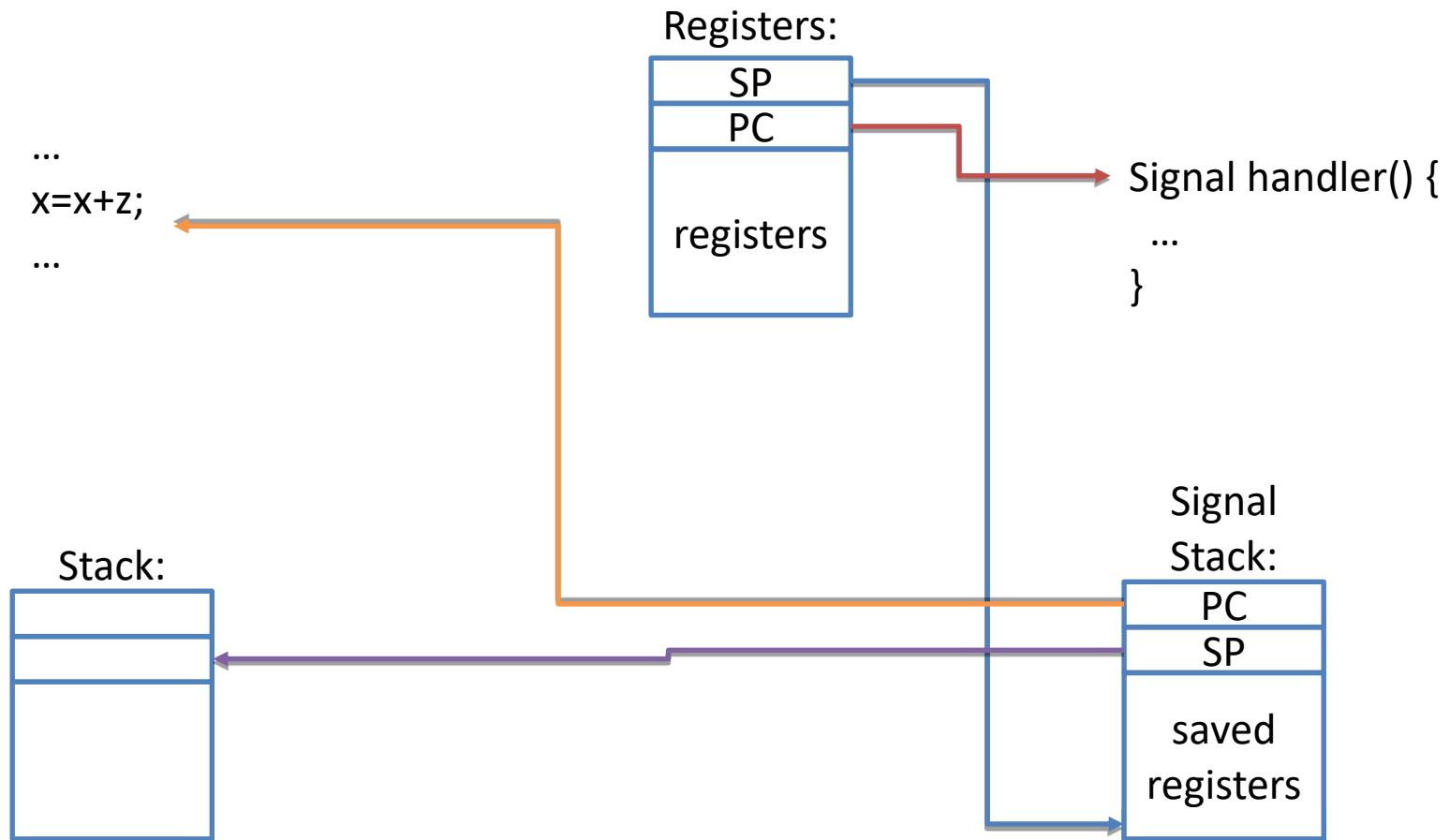
Upcall: User-level interrupt

- Also known as **UNIX *signals***
 - Notify user process of event that needs to be handled right away
 - Time-slice for user-level thread manager
 - Interrupt delivery for VM player (see later)
- Direct analogue of kernel interrupts
 - Signal handlers – fixed entry points
 - Separate signal stack
 - Automatic save/restore registers – transparent resume
 - Signal masking: signal disabled while in signal handler

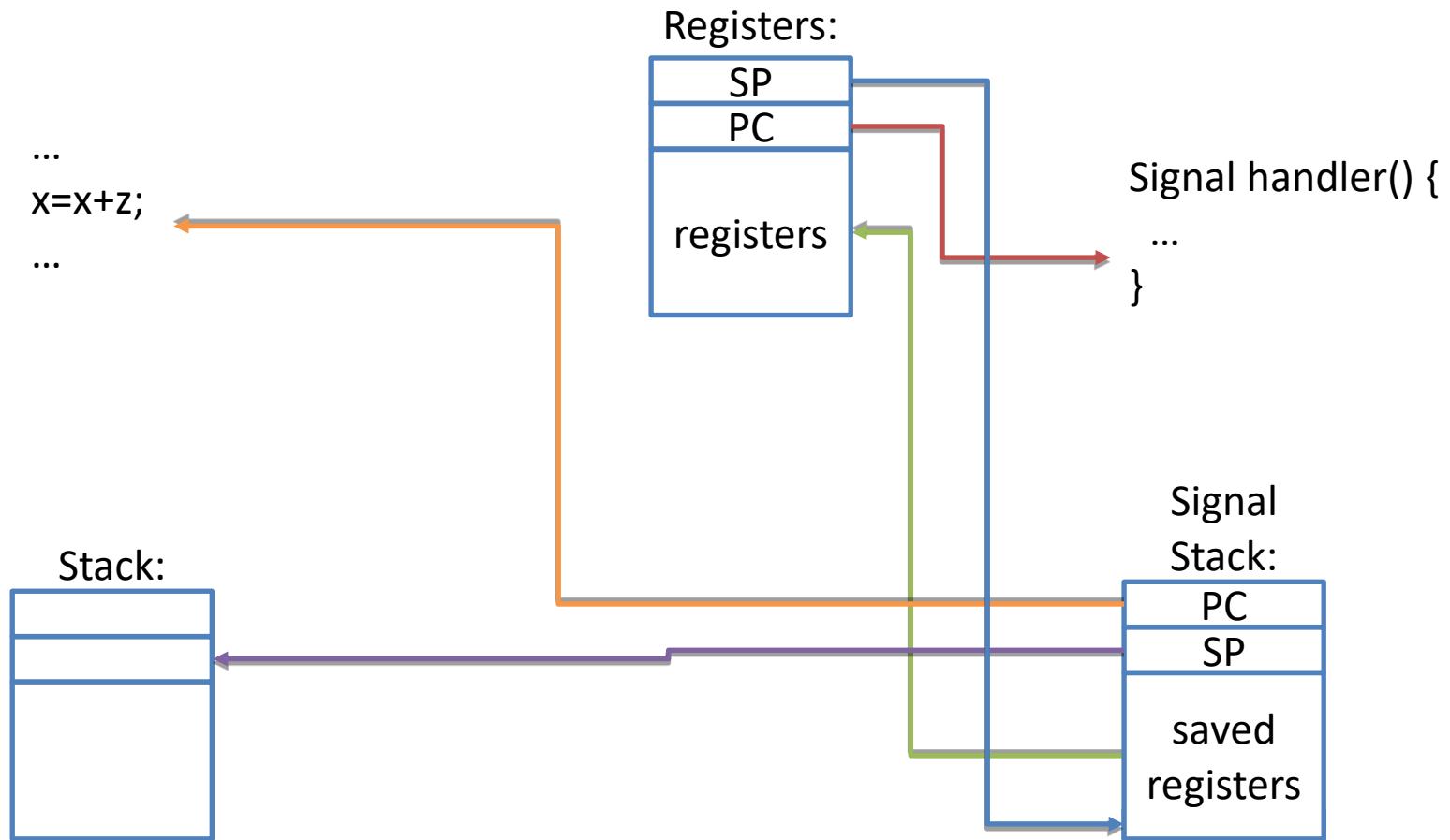
Upcall: Before



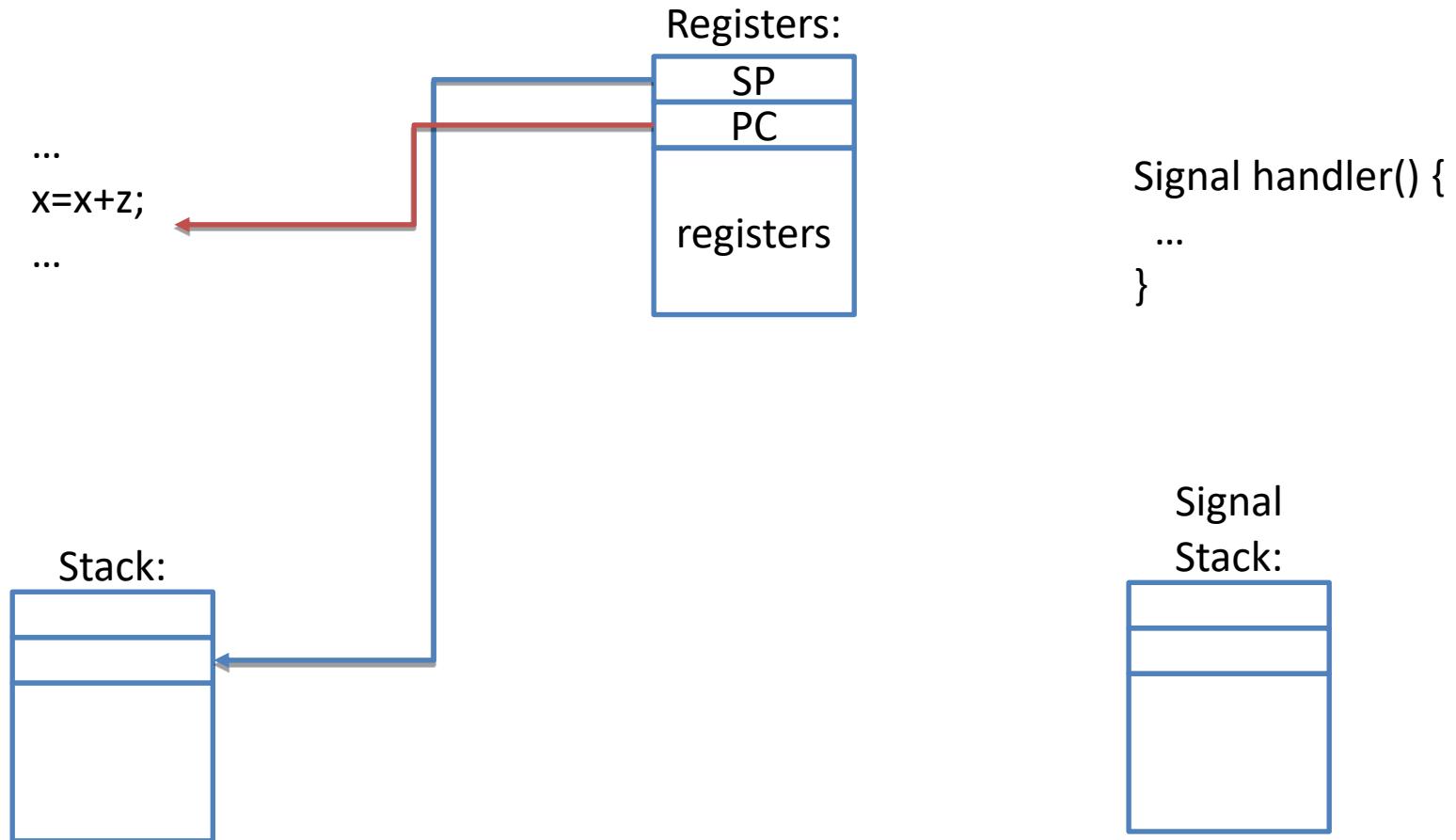
Upcall: During



Upcall: final phase

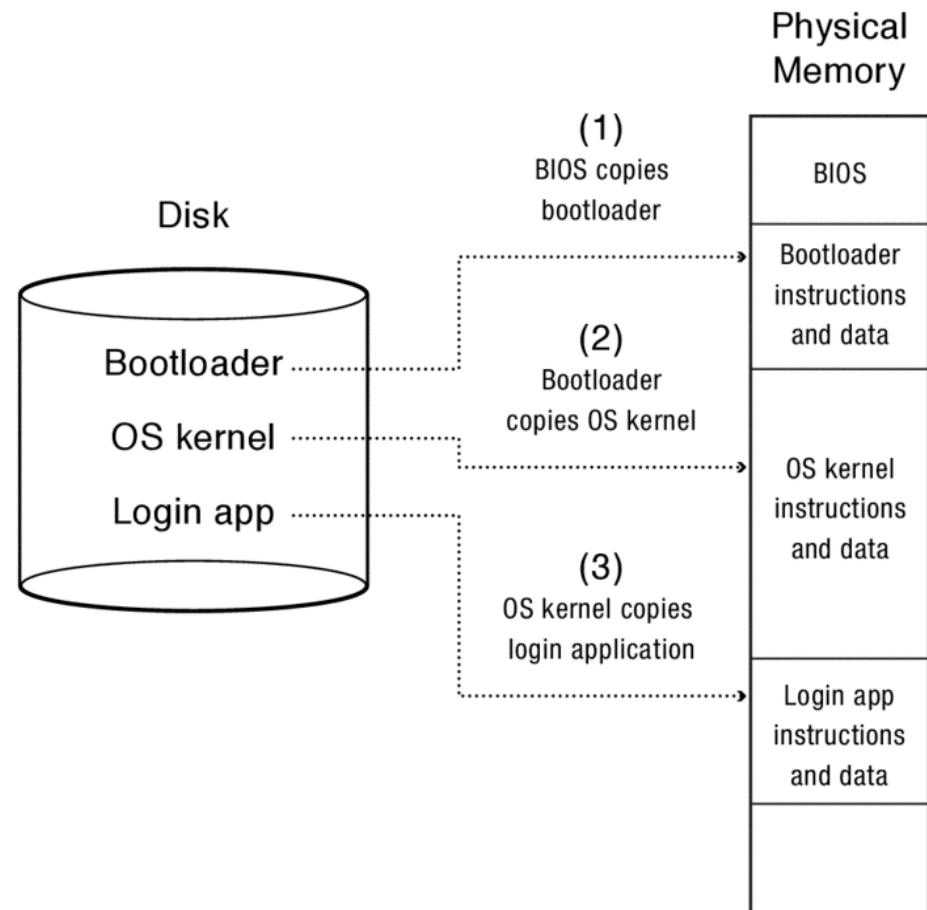


Upcall: end of handler



Kernel Booting

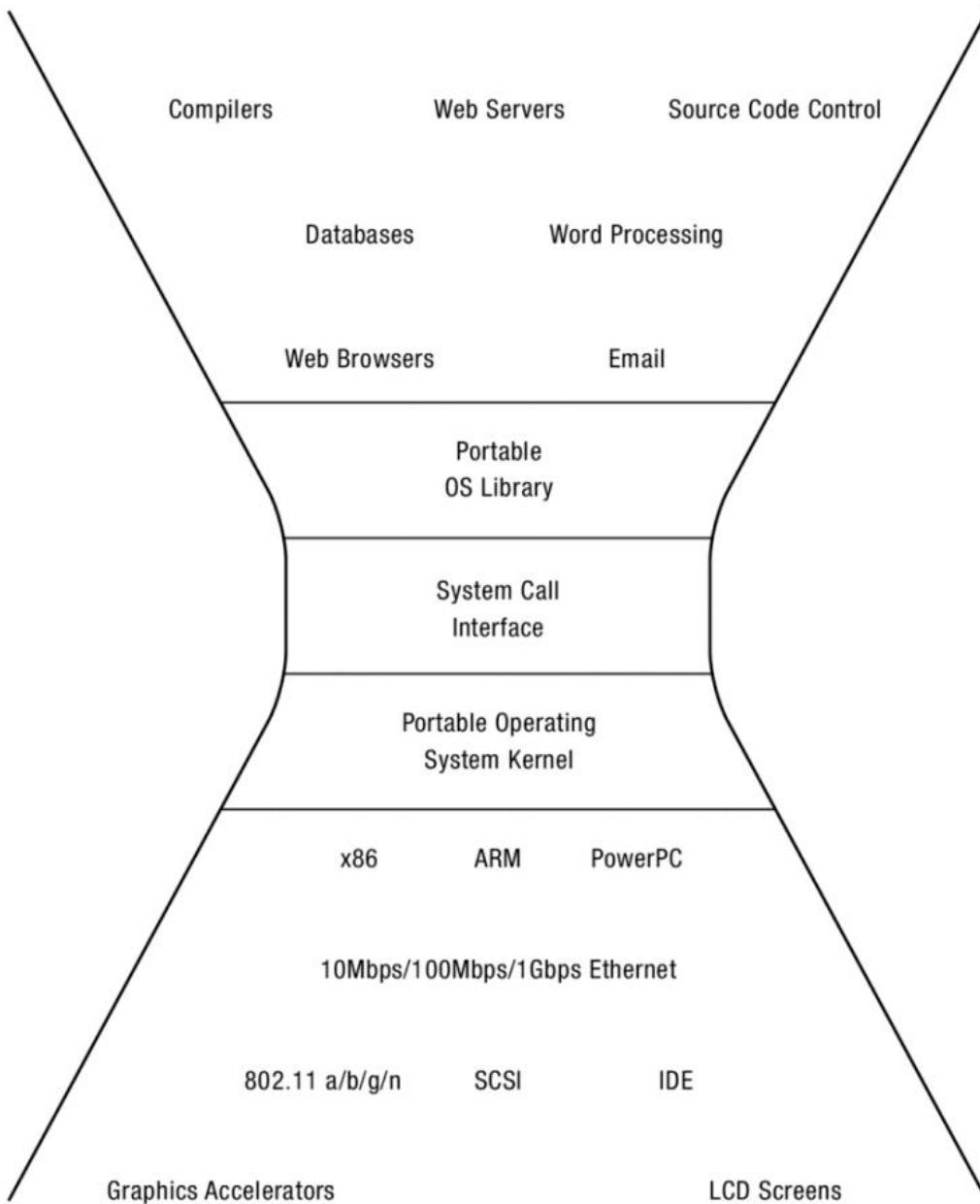
- At boot, the processor executes code in ROM to load the ***first-stage bootloader***.
- The first-stage bootloader initializes the memory controller and a few I/O devices to load in memory a ***second-stage bootloader***.
- The second-stage boot loader loads the kernel and the root file system and then passes control to the kernel



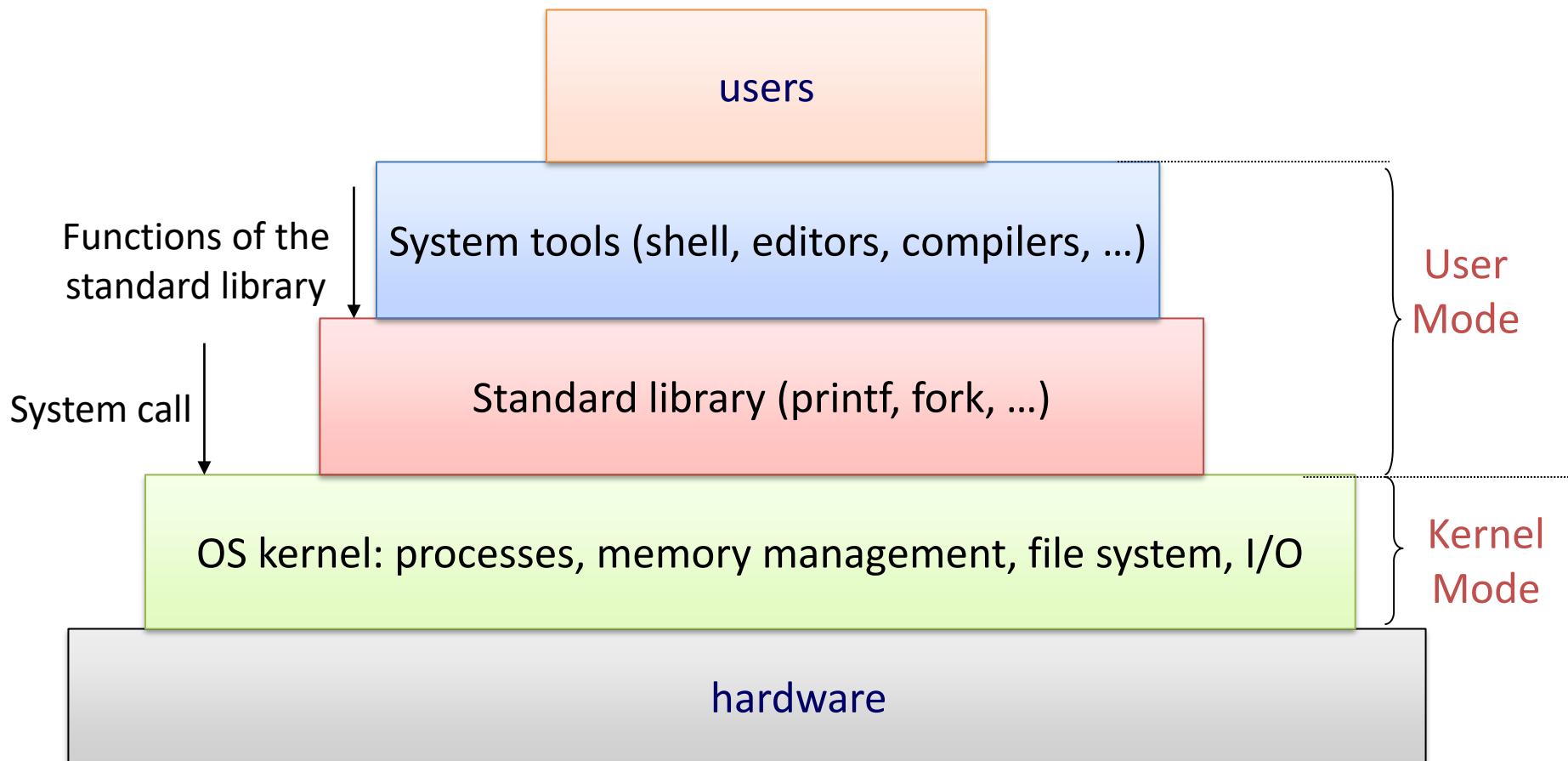
Summary: HW support for OSs

- **Privilege levels**, at least two: user and kernel level.
 - **Privileged instructions**: instructions available only in kernel mode.
 - **Memory translation** prevents user programs from accessing kernel data structures and aids in memory management.
 - **Processor exceptions** trap to the kernel on a privilege violation or other unexpected event.
 - **Timer interrupts** return control to the kernel on time expiration.
 - **Device interrupts** return control to the kernel to signal I/O completion.
 - **Interprocessor interrupts** cause another processor to return control to the kernel.
 - **Interrupt masking** prevents interrupts from being delivered at inopportune times.
 - **System calls** trap to the kernel to perform a privileged action on behalf of a user program.
 - **Return from interrupt**: switch from kernel mode to user mode, to a specific location in a user process.
 - **Boot ROM**: code that loads startup routines from disk into memory.
 - **Support for Virtualization**: hypervisor (aka VMM) and additional privilege levels
- To support threads, we will need one additional mechanism (described later):
- **Atomic read-modify-write instructions** used to implement synchronization in multi-threaded programs.

Structure



Unix architecture



Programming Interface

Main Points

- Creating and managing processes
 - fork, exec, wait (,exit)
- Performing I/O
 - open, read, write, close
- Communicating between processes
 - pipe, dup, select, connect
- Example: implementing a shell

Laboratory part

Shell

- A shell is a job control system
 - Allows programmer to create and manage a set of programs to do some task
 - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
```

Question

- If the shell runs at user-level, what system calls does it make to run each of the programs?
 - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
 - Create and initialize the process control block (PCB) in the kernel
 - Create and initialize a new address space
 - Load the program into the address space
 - Copy arguments into memory in the address space
 - Initialize the hardware context to start execution at ``start''
 - Inform the scheduler that the new process is ready to run

Windows CreateProcess API (simplified)

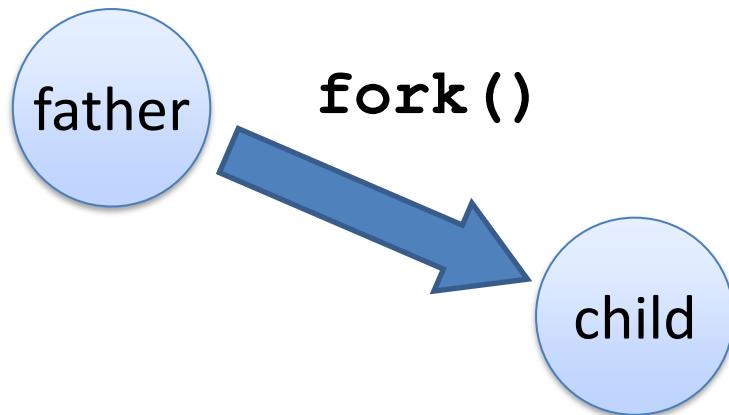
```
if (!CreateProcess(  
    NULL,          // No module name (use command line)  
    argv[1],        // Command line  
    NULL,          // Process handle not inheritable  
    NULL,          // Thread handle not inheritable  
    FALSE,         // Set handle inheritance to FALSE  
    0,             // No creation flags  
    NULL,          // Use parent's environment block  
    NULL,          // parent's starting directory  
    &si,           // Pointer to STARTUPINFO structure  
    &pi            // Pointer to PROCESS_INFORMATION structure  
)
```

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
 - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send notifications among processes

UNIX fork()

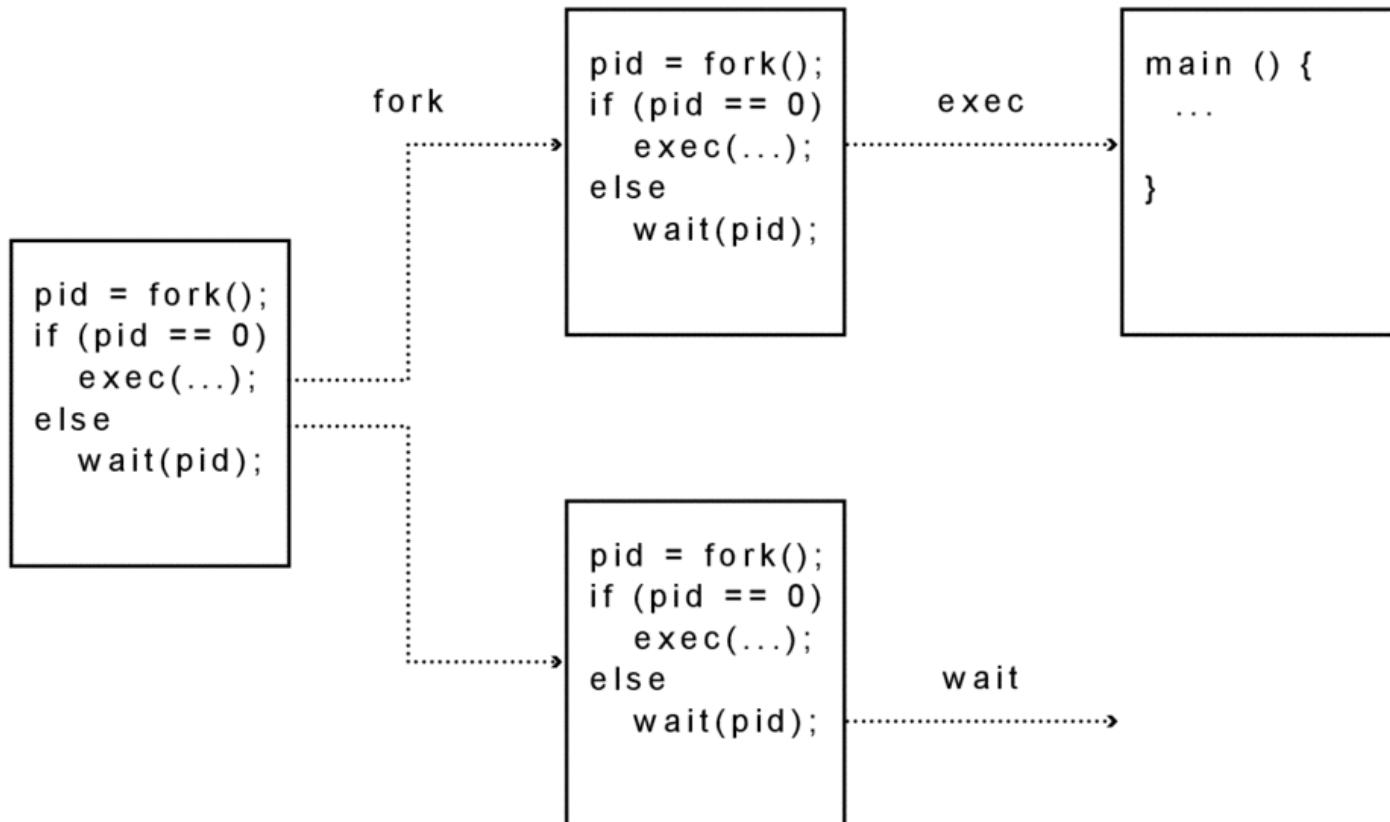
- `fork()` is used to generate a child process:
 - The father and its child share the same code
 - The child process inherits a copy of the kernel and user-data of the father



UNIX fork()

- `fork()` does not take input params
- Returns an integer:
 - For the child it is 0
 - For the father is:
 - A positive value that represents the PID of the child
 - A negative value that represents an error code (the error code is stored into the per-thread *errno* variable)

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {    // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                  // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately?
Why?

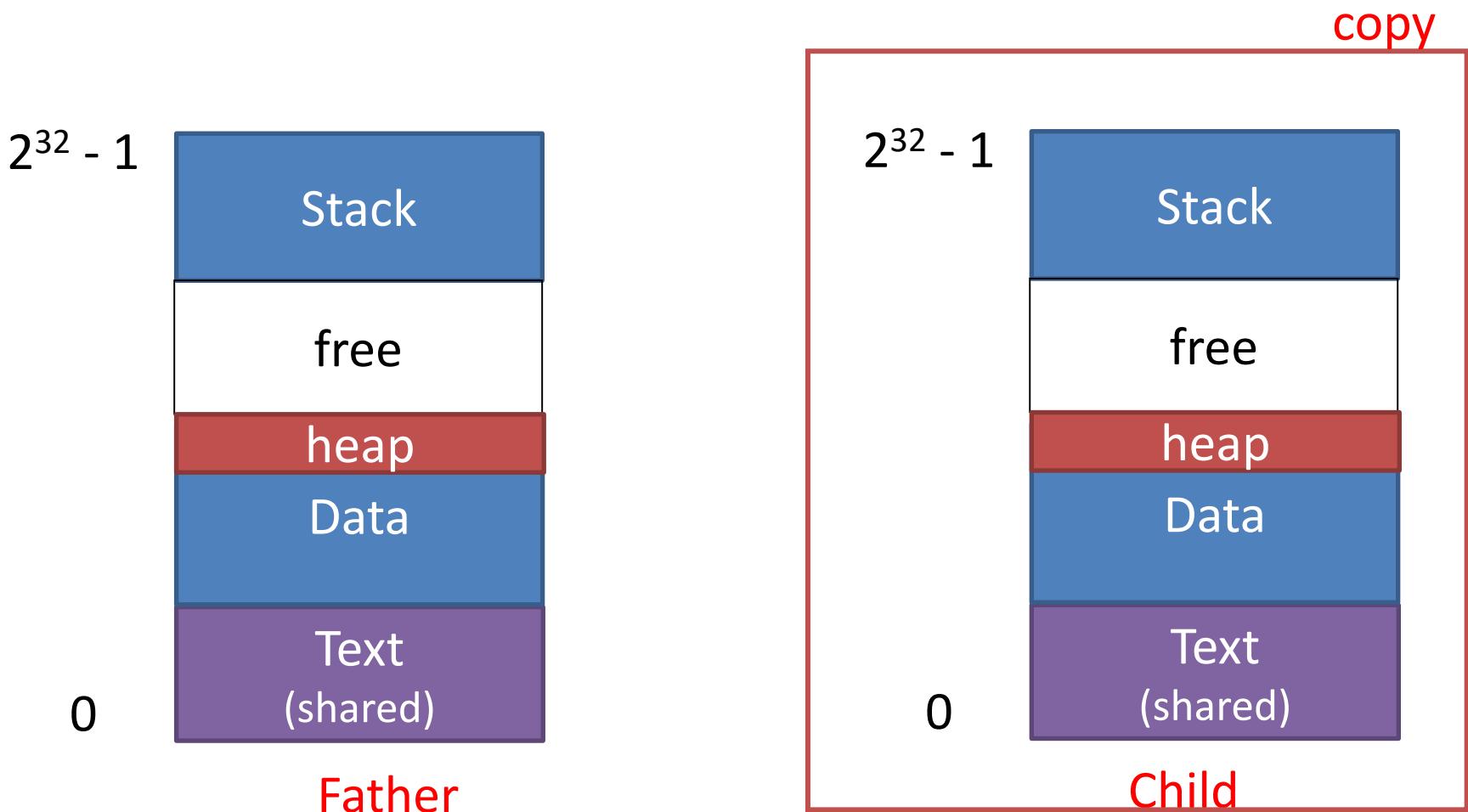
Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
 - Namely, a process structure and a user structure
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Copy arguments into memory in the address space
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

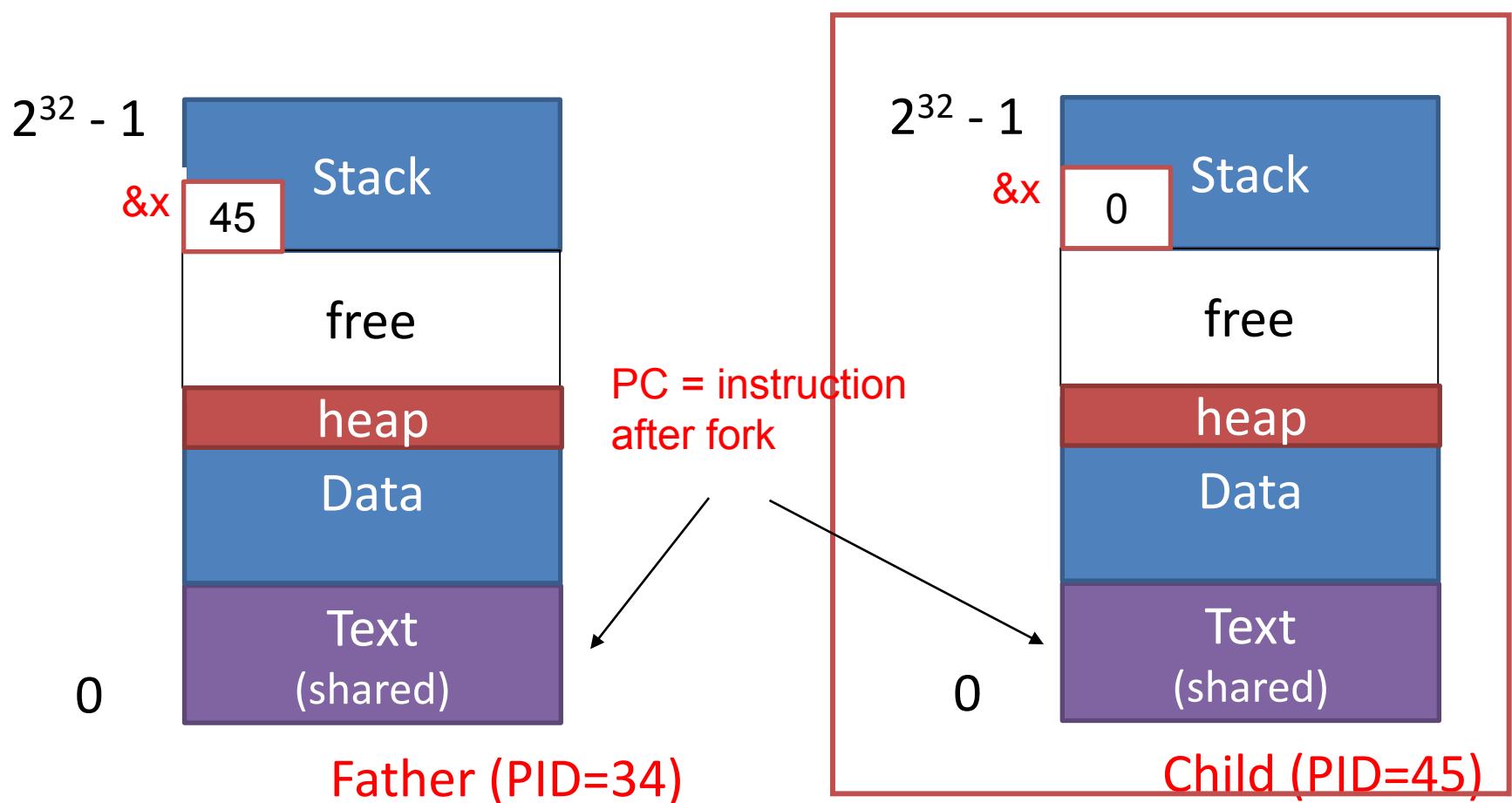
Implementing UNIX fork

Addressing spaces of the father and the child after a successful fork



Implementing UNIX fork

Addressing spaces of the father and the child after a successful fork



UNIX exec

- Replaces the code executed by a process
 - Does not create a new process, inherits PCB and changes address space
- Replaces the data
- Example, execl:

```
int execl(char *pathname, char *arg0, ..., char *argN,  
          (char*) 0)
```

- Pathname is the name of an executable file
- argN[1] ... argN[...] are the arguments passed to the program
- list terminated with NULL ((char*) 0)

UNIX exec

- If it's successful it does not return
 - The process executes another program
- If it fails it returns an error code
- After exec the process:
 - Keeps the PID
 - Keeps the PCB (process and user structures)
 - But it changes references to code and data memory
 - Resets the pending signals (laboratory classes)
 - Keeps the kernel stack
 - Keeps the assigned resources (open files)

Process termination in UNIX

- A process can terminate:
 - Because of an exception due to illegal actions
 - By invoking the system call `exit`
- The terminated process returns an exit value to its father
 - The father receives the value by the system call `wait`
 - If the father didn't already call the `wait`, the terminated process switches to zombie state
 - If the father is already terminated, the init process waits for the termination of the children

exit() and wait()

- `void exit(int status);`
 - Status is the termination code
 - `exit` never returns
 - Frees memory, releases resources
 - If it switches to zombie, keeps the PCB until the father invokes `wait`
- `int wait(int *status);`
 - Status is the PID of the terminated process or an error code

Implementing a (simplified) Shell

```
char *prog, **args;  
int child_pid;  
  
// Read and parse the input a line at a time  
while (readAndParseCmdLine(&prog, &args)) {  
    child_pid = fork();    // create a child process  
    if (child_pid == 0) {  
        exec(prog, args);    // I'm the child process. Run program  
        // NOT REACHED  
    } else {  
        wait(child_pid);    // I'm the parent, wait for child  
        // checking exit status  
    }  
}
```

UNIX I/O

- Uniformity
 - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
 - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
 - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
 - Open the file, return file descriptor
 - Options:
 - if file doesn't exist, return an error
 - If file doesn't exist, create file and open it
 - If file does exist, return an error
 - If file does exist, open file
 - If file exists but isn't empty, nix it then open
 - If file exists but isn't empty, return an error
 - ...

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
    create(name); // can create fail?
fd = open(name); // does the file exist?
```

Concurrency

Motivation

- Operating systems need to be able to handle *Multiple Things At Once* (MTAO)
 - processes, interrupts, background system maintenance
- Servers need to handle MTAO
 - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
 - To achieve better performance
- Programs with user interfaces often need to handle MTAO
 - To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
 - To hide network/disk latency

Motivations

A programmer usually needs to write programs that manage MTAO.

For example, consider a word processor:

- Reads input from keyboard
- Writes on screen
- Formats the data structure
- Periodically saves on disk
- Spell checking
-

Motivations

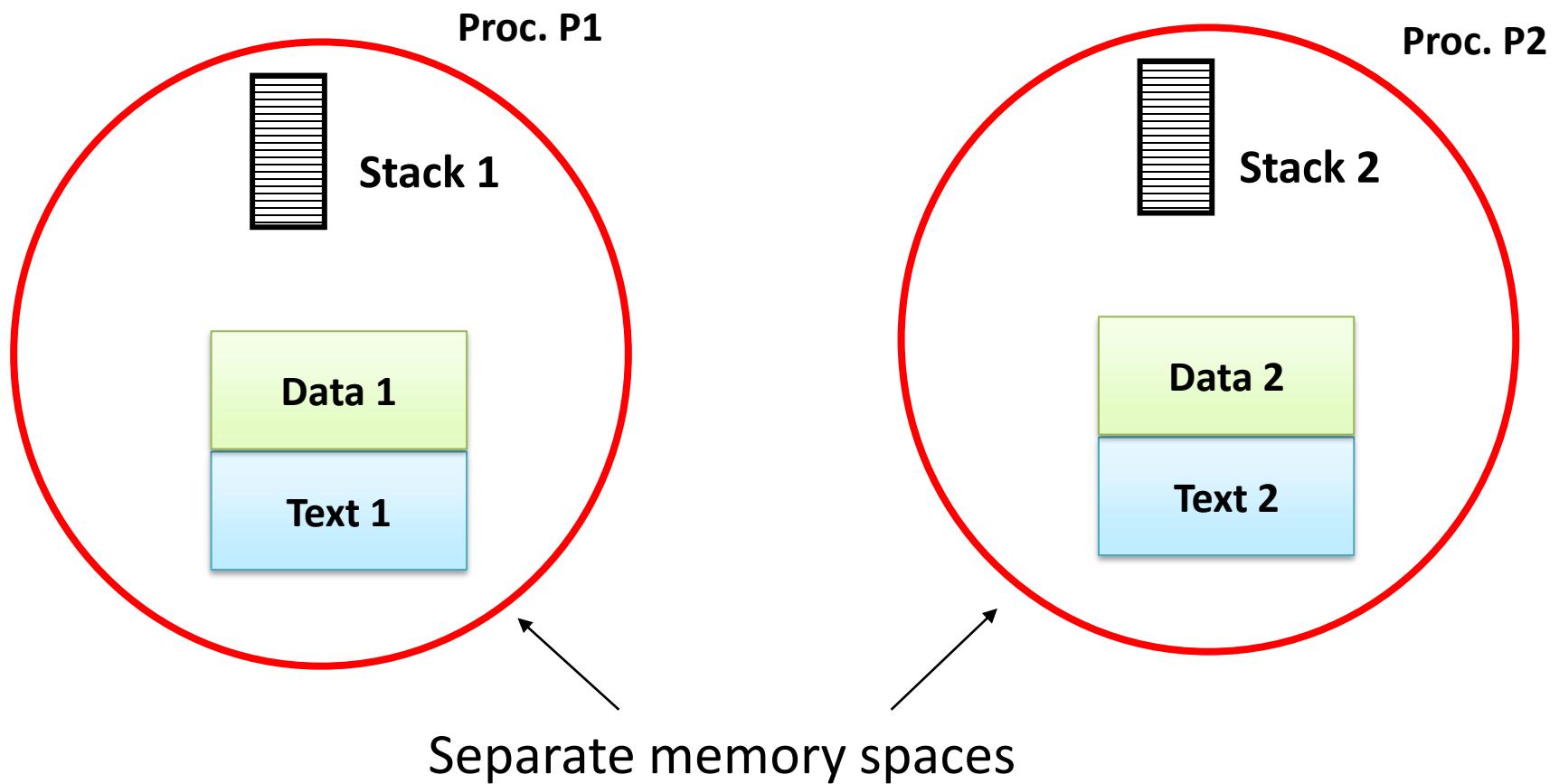
Idea: implement each function in a separate process

However:

- Each process has its own private memory space
- Processes can cooperate by using inter-process communication (IPC) mechanisms offered by the OS:
 - example : pipes, send-receive via sockets, ...
- They have higher communication overhead than using shared-variables!

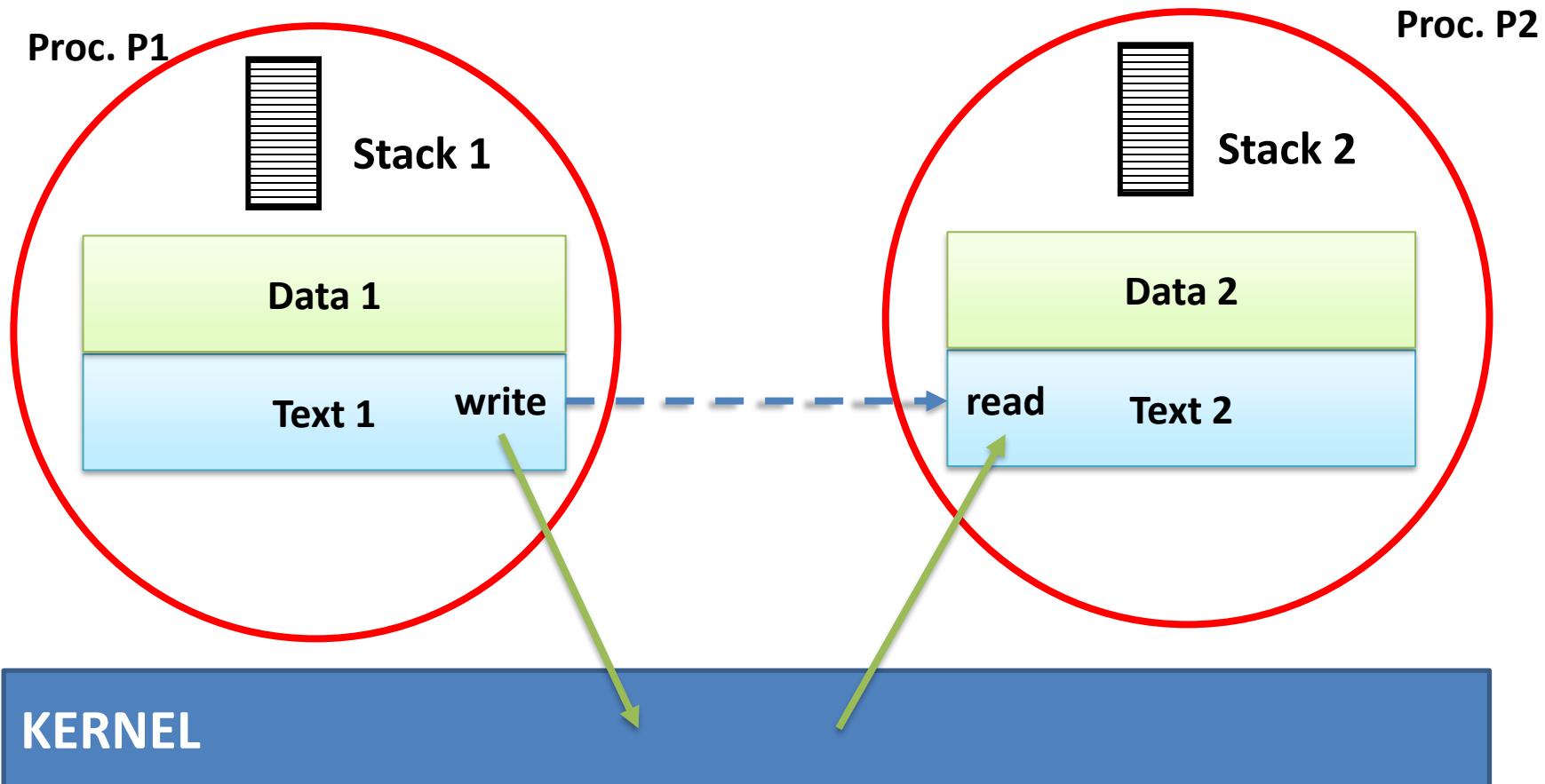
Inter-process communications – IPC

- Two processes do not share memory!

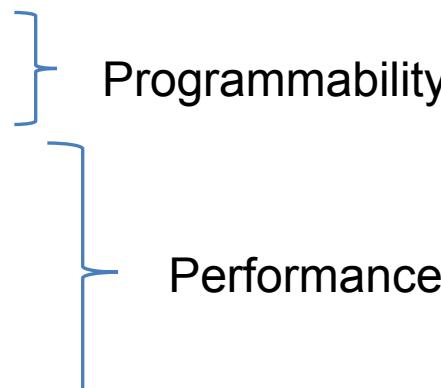


Inter-process communications

- Several mechanisms offered by the kernel
 - eg: write/read on pipes
 - Performance issue: communications should pass through the kernel



Thread abstraction

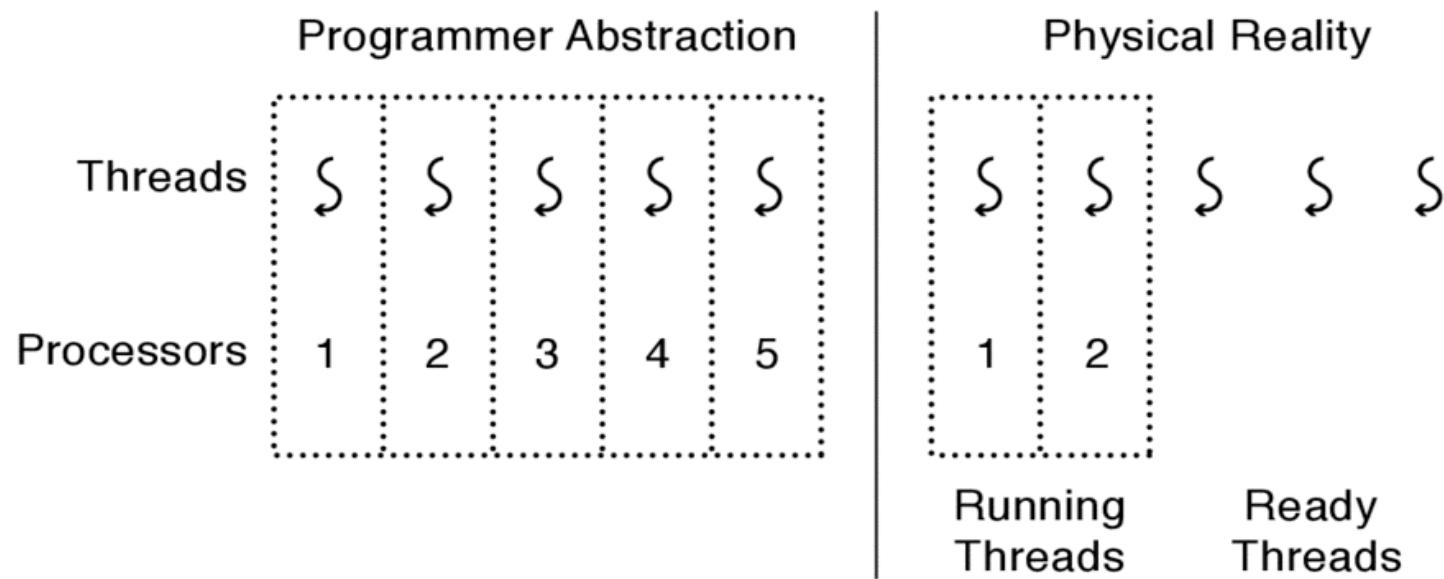
- A thread is a single execution sequence that represents a separately schedulable task
 - Single execution sequence: familiar programming model
 - Separately schedulable: OS can run or suspend a thread at any time
 - Why using threads?
 - Program structure
 - Responsiveness
 - Exploiting multi-cores
 - Interacting with slow I/O devices
- 
- Programmability
- Performance

Thread abstraction

- Protection is an orthogonal concept
 - Can have one or many threads per protection domain (process)
 - Single threaded user program: one thread, one protection domain
 - Multi-threaded user program: multiple threads, sharing same data structures, isolated from other user processes
 - Multi-threaded kernel: multiple threads, sharing kernel data structures, capable of using privileged instructions

Thread Abstraction

- Unlimited number of processors
 - one processor for each thread
- Threads execute with variable speed
 - Programs must be designed to work with any schedule

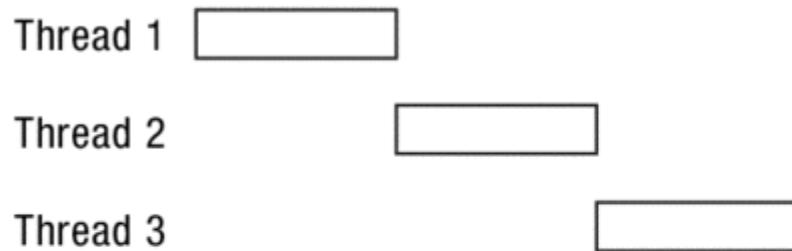


Programmer vs. Processor View

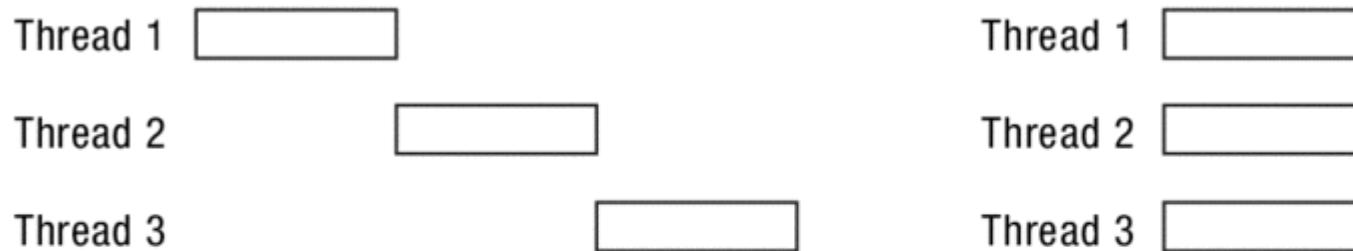
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	Thread is resumed.
		y = y + x;
		z = x + 5y;	z = x + 5y;

Possible Executions

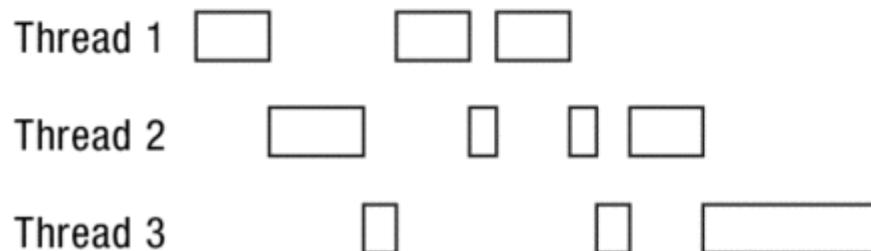
One Execution



Another Execution



Another Execution



Main: Fork 10 threads call join on them, then exit

- What other kinds of interleaving are possible?
- What is the maximum # of threads running at same time?
- Minimum?

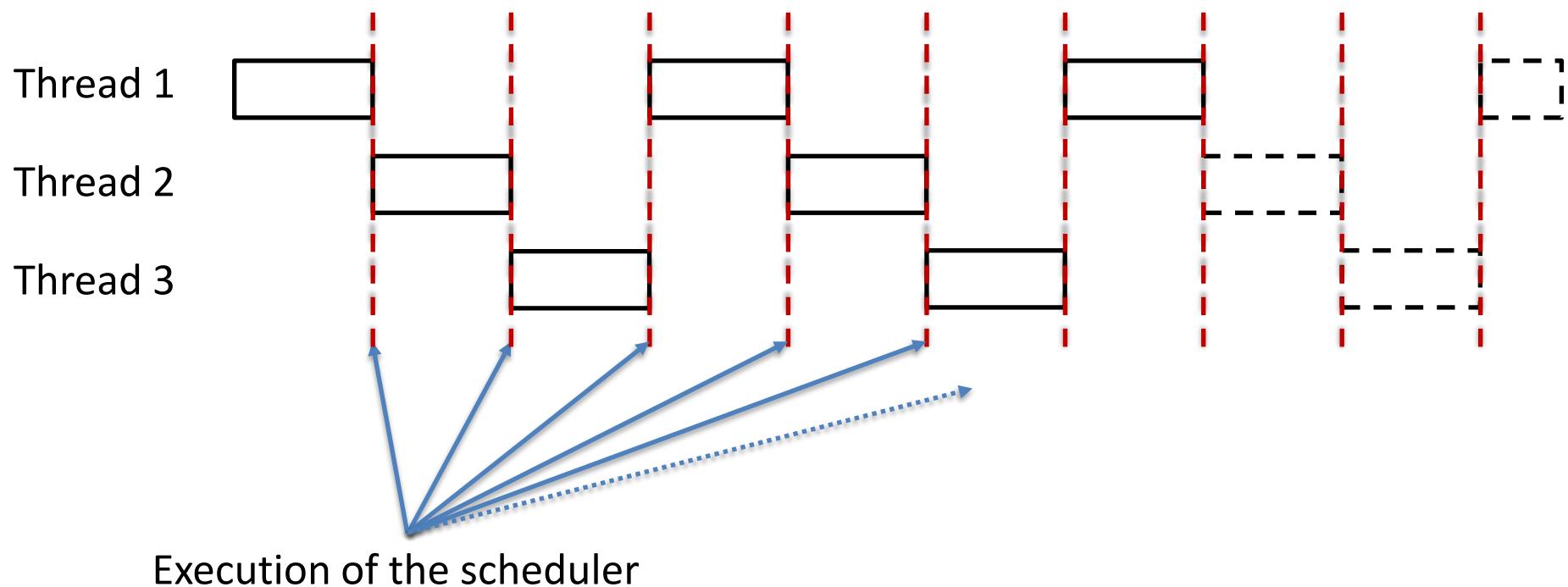
```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Implementing threads

- Thread Control Block (TCB) – data structure that stores information about the thread
- A set of operations on threads
- A scheduler
 - A function of the OS that assigns the processor(s) to the threads

Scheduler in time-sharing systems

* One processor

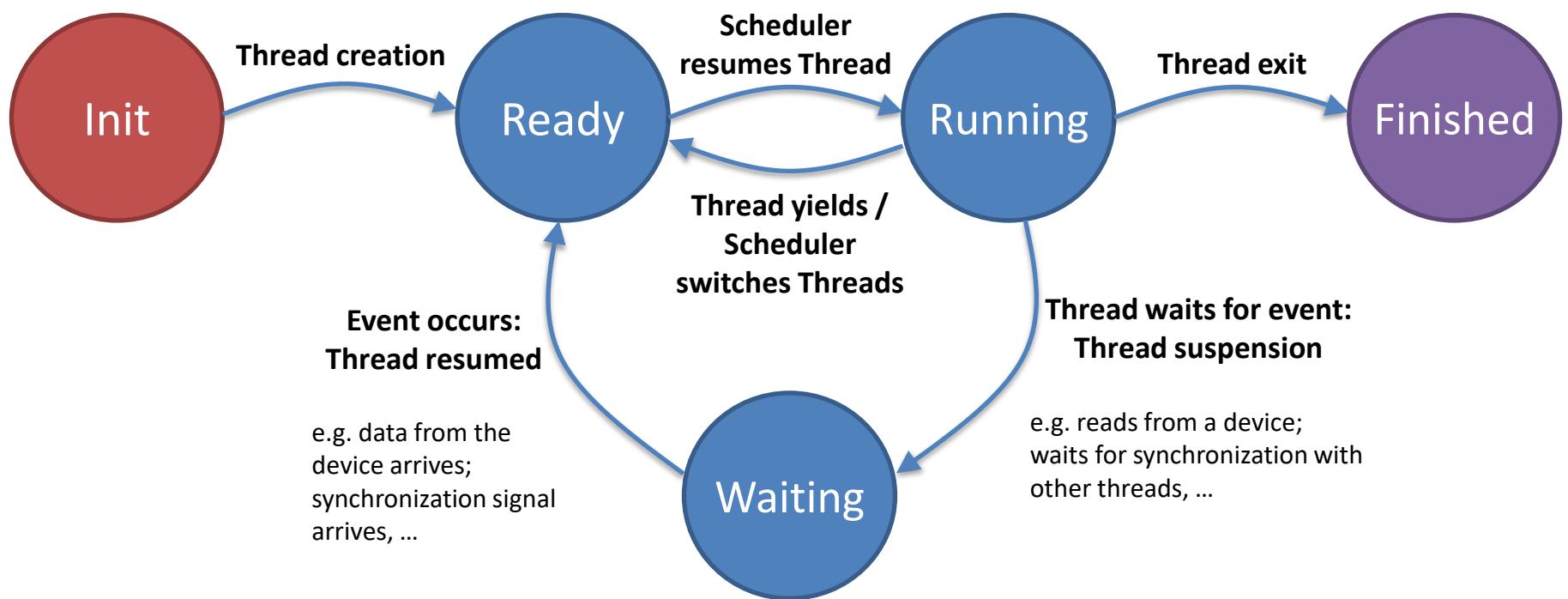


- *Cooperative vs preemptive multi-threading*

Simple Thread API

- `thread_create(thread, func, args)`
 - Creates a new thread, storing information about it in `thread`, to run `func(args)`
- `thread_yield()`
 - The calling thread voluntarily gives up the processor to let some other thread(s) run
- `thread_join(thread)`
 - Waits for `thread` to finish if it has not already done so, then returns the exit status of the thread. It can be called only once for each thread
- `thread_exit(exit_status)`
 - Finishes the current thread. Stores the `exit_status` in the threads's data structure. If another thread is waiting in the `thread_join`, resumes it.

Thread Lifecycle



Location of thread's per thread state

State of thread	Location of TCB	Location of registers
INIT	Being created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List, then Deleted	TCB

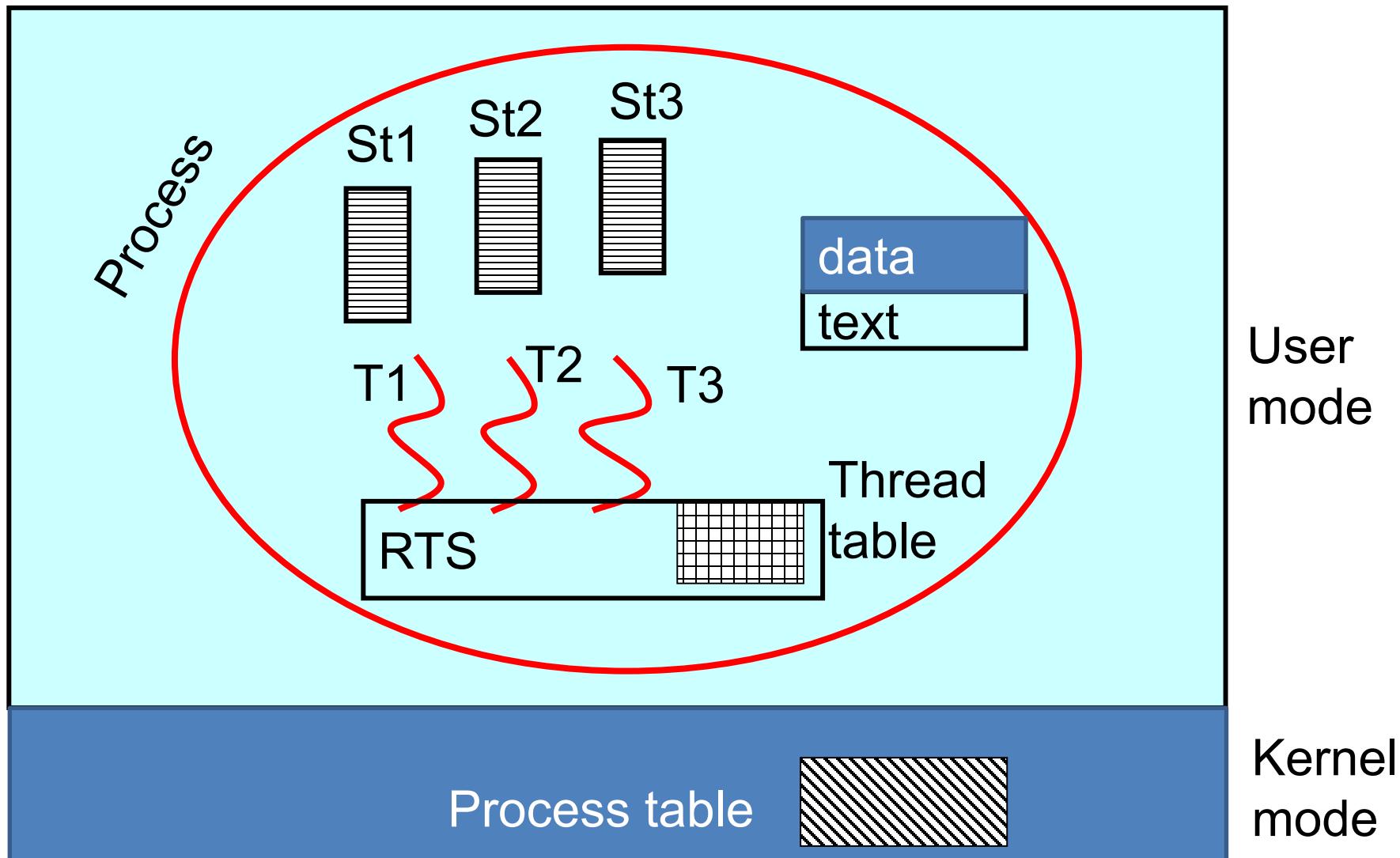
Roadmap

- Threads can be implemented in any of several ways
 - Multiple single-threaded processes (early UNIX)
 - Multiple user-level threads, inside a UNIX process (early Java)
 - Mixture of single and multi-threaded processes and kernel threads (Linux, MacOS, Windows)
 - To the kernel, a kernel thread and a single threaded user process look quite similar
 - Scheduler activations (Windows, see later)

User-level threads

- Threads implemented by means of a user-level library
- O.S. not aware of user level threads
- Thread table within each process (user space)
- Scheduling of the threads implemented by the Run Time Support (RTS) of the process
 - Threads can use *thread_yield()* to release the processor
 - *preemptive scheduler with Scheduler Activations
- An invocation to a blocking system call blocks all the threads of the process

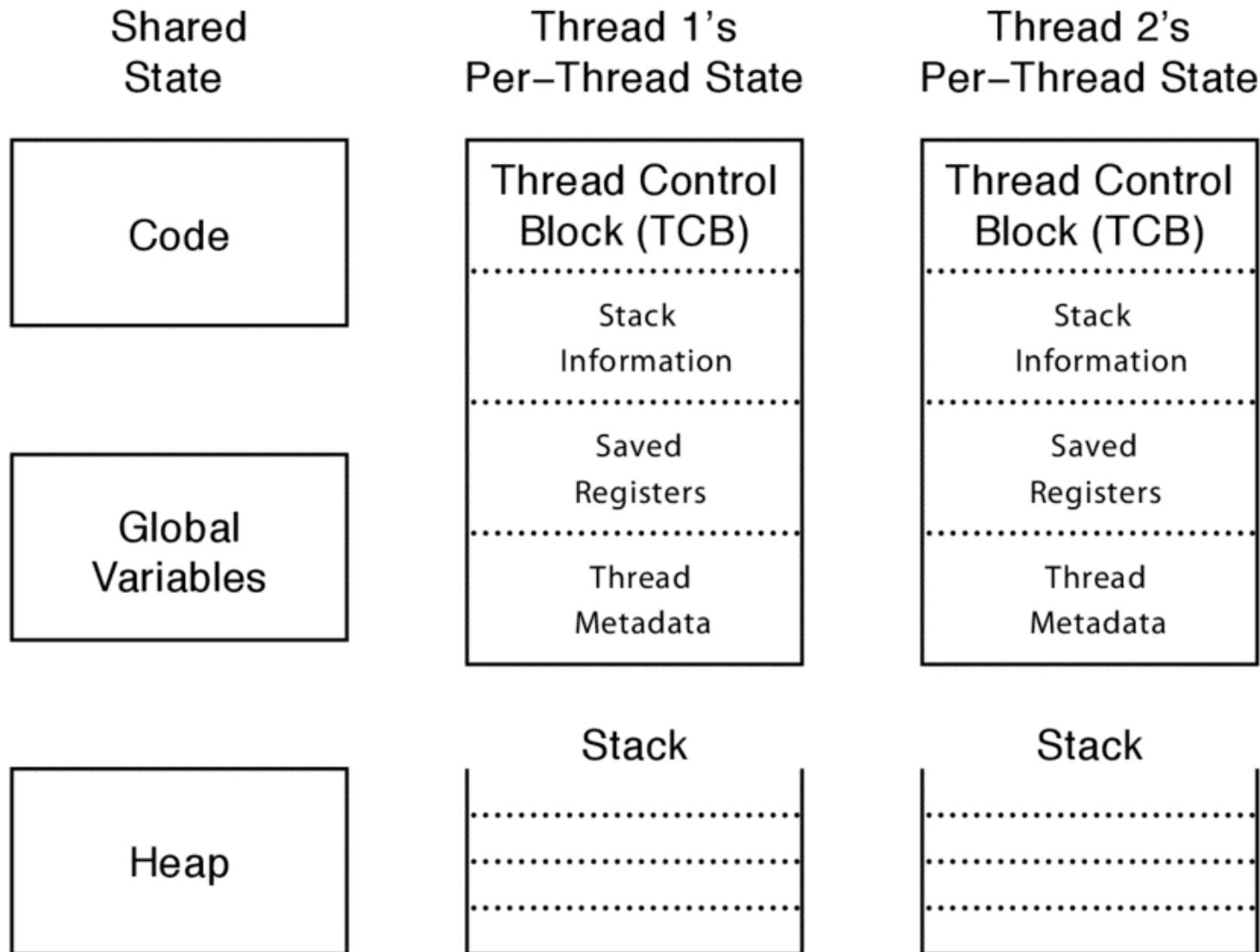
User-level threads



Implementing user-level threads

- `thread_create(thread, func, args)`
 - Allocate thread control block
 - Allocate stack
 - Build stack frame for base of stack (stub)
 - Put func, args on stack
 - Put thread on ready list
 - Will run sometime later (maybe right away!)
- `stub(func, args)`
 - Call `(*func)(args)`
 - Call `thread_exit()`

Shared vs. Per-Thread State



Thread Stack

- What if a thread puts too many procedures on its stack?
 - What should happen?
 - What happens in Java?
 - What happens in Linux?

User-level threads

Pros:

- Creation, termination and context switch very efficient
 - Do not need system call invocations, just calls to the thread library
 - In case of context switch the addressing space remains the same
- Can be implemented on any O.S. that does not support multithreading
 - e.g. early versions of UNIX

User-level threads

Cons (*):

- Blocking system calls block all the user-level threads of a process
- Do not take advantage of multiprocessors architectures
 - All threads of a process are scheduled on the same processor

* this does not apply to scheduler activations (see later)

Kernel-level threads

- Threads implemented in the kernel
 - Thread table in the kernel
 - Creation, termination and context switch activated by system calls
 - Different thread of the same process can run in parallel on different processors

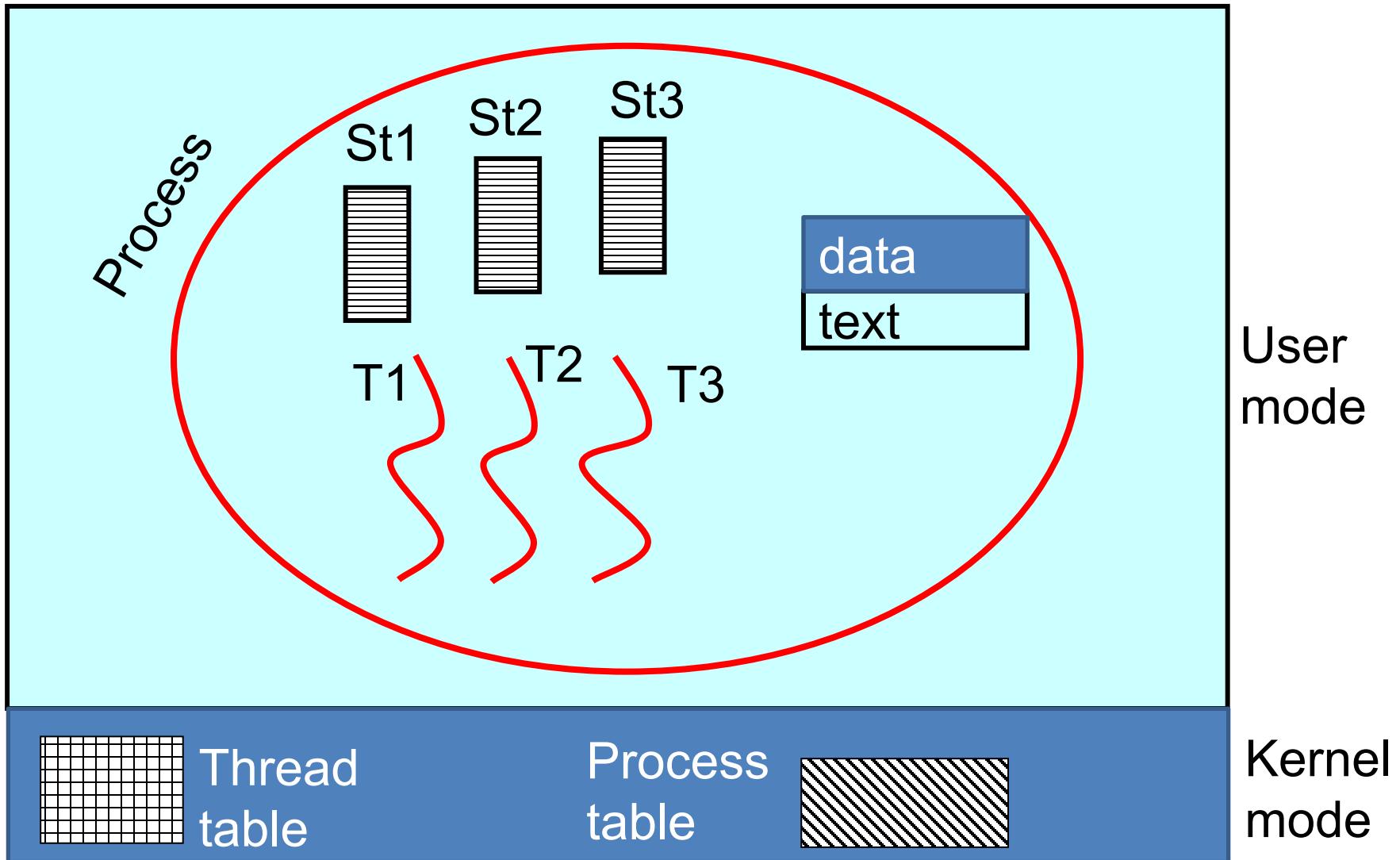
Processes and Threads representation

- Process Control Block (PCB)
 - Data structure associated to each process
- Process Table
 - Contains all PCBs
 - In the kernel, one for the entire system
- Thread Control Block (TCB)
 - One for each thread
- Thread Table
 - One for each process (user-level threads)
 - In the kernel, one for the entire system (kernel-level threads)

TCB & PCB

- PCB:
 - Process name (PID)
 - Assigned memory
 - Other resources
 - Devices, open files, ...
 - Handlers to the process' threads
 - ...
- TCB:
 - Thread ID
 - State
 - Context of the thread
 - Scheduling parameters
 - Reference to the stack(s)
 - ...

Kernel-level threads



Kernel-level threads

- Operations on threads and interactions among threads by means of System Calls
 - More overhead w.r.t. user-level threads
- Thread scheduling implemented by the O.S.
- Threads can invoke blocking system calls
 - Only the invoker gets blocked

Threads in a Process

- Threads are useful in user-level programs:
 - Parallelism, hide I/O latency, interactivity/responsiveness
- Option A (early Java): user-level library, within a single-threaded process
 - Library implements thread context switch
 - Kernel time slices between processes, e.g., on system call I/O
 - Main Issues: blocking system calls, non-preemptive scheduling
- Option B (Linux, MacOS, Windows): use kernel threads
 - System calls for thread fork, join, exit (and lock, unlock,...)
 - Kernel implements context switch (preemptive scheduling)
 - Simple, but a lot of transitions between user and kernel mode

Threads in a Process

- Option C (Windows): *Scheduler Activations*
 - Kernel allocates processors to user-level library
 - Thread library implements context switch
 - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O
- Option E: Threads + Asynchronous I/O

Thread Switch

- Two causes:
 - Voluntary
 - Due to an interrupt/exception
- Almost the same management for the different cases:
 - Kernel/user threads
 - Multithread/singlethread processes

Implementing (voluntary) thread context switch

- User-level threads in a user-level process:
 - Save registers on old TCB (of the running thread)
 - It may temporarily save registers on the stack
 - Switch to new stack & to new thread
 - Restore registers from new thread's TCB
 - return
- Kernel threads
 - Exactly the same, ‘return’ substituted by proper instruction (e.g., IRET or MOVS/SUBS)

Two threads call `thread_yield()`

Thread 1's instructions

call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state

return `thread_yield`
call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state

return `thread_yield`

...

Thread 2's instructions

call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state

return `thread_yield`
call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state

...

Processor's instructions

call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state
call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state
return `thread_yield`
call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state
return `thread_yield`
call `thread_yield`
save state to stack
save state to TCB
choose another thread
load other thread state
return `thread_yield`

...

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
 - Tells OS some other thread should run
- Simple version
 - Interrupt handler first save registers in kernel stack
 - End of interrupt handler calls **switch_threads()**
 - When resumed, return from handler resumes kernel thread or user process/thread
- Faster version...
 - Interrupt handler saves registers directly in TCB
 - Interrupt handler returns to saved state in TCB
 - Resumes a kernel thread or a user process/thread

switch_threads()

- Save registers (context) of the old thread from kernel stack in the TCB
- Move old thread's TCB to Ready List or to a Waiting List
- Select a new thread from the Ready List
 - scheduling problem – discussed later
- Restores new thread's registers from TCB to processor
- Put new thread's TCB in the Running List
- return control to the new thread (IRET...)

Thread switch - overhead

- Due to registers save and restore
- Due to TCB queues management
- Memory cache invalidation
 - Refer to the computer architecture classes...
- Induced operations on the memory manager
 - Address exceptions
 - Page faults
 - MMU invalidation
 - Will be discussed later on

Context switch - example

Let us consider again a simplified processor with special registers PC and PS, the user-level stack pointer SP and just two general registers R1, R2. The interrupt vector is in memory. The system uses a single kernel stack (shared for all threads). IRET to return. When it receives an interrupt, the processor:

- Sets kernel mode;
- Disables interrupts;
- Saves PC & PS & SP on the kernel stack
- Loads the new PC & PS from the interrupt vector
- Consequently it jumps to the interrupt handler in the kernel

Hardware

The IRET instruction:

- Enables interrupts;
- Sets user mode;
- Restores PC, PS & SP from the kernel stack; (consequently jumps back to the address at which the RUNNING thread had been interrupted in the past)

The interrupt handler:

- First saves the general registers on the kernel stack
- At the end, it restores the general registers and then executes IRET

Software

Context switch – example 1

Hyp. A): thread T1 invokes a system call. At the end it remains in RUNNING state

- 1) Initial situation during the execution of SVC instruction (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

base kernel SP	0FFF
----------------	------

Context switch - example

- 1) Initial situation during the execution of SVC instruction (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF
----------------	------

- 2) After interrupt (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5000
PS	AA45
SP	OFFC
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF
----------------	------

Context switch - example

2) After interrupt (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5000
PS	AA45
SP	OFFC
R1	4500
R2	CD31

3) After temporary storage of registers (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

Registers	
PC	5000 + ?
PS	AA45
SP	OFFA
R1	??
R2	??

Context switch - example

4) At the end of the primitive (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

Registers	
PC	5000 + ?
PS	AA45
SP	OFFA
R1	??
R2	??

5) During extraction of IRET at address 5100 (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5100
PS	AA45
SP	OFFC
R1	4500
R2	CD31

Context switch - example

- 5) During execution of IRET at address 5100 (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5100
PS	AA45
SP	OFFC
R1	4500
R2	CD31

- 6) At the end of IRET (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

Context switch – example 2

Hyp. B): thread T1 invokes a system call that blocks T1 and switches T2 in RUNNING state

- 1) Initial situation during the execution of SVC instruction (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

base kernel SP	0FFF
----------------	------

Context switch - example

- 1) Initial situation during the execution of SVC instruction (**USER MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF
----------------	------

- 2) After interrupt (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5000
PS	AA45
SP	OFFC
R1	4500
R2	CD31

address	5000
PS	AA45
Interrupt vector	

Base kernel SP	0FFF
----------------	------

Context switch - example

2) After interrupt (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

Registers	
PC	5000
PS	AA45
SP	OFFC
R1	4500
R2	CD31

3) After temporary storage of registers (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

Registers	
PC	5000 + ?
PS	AA45
SP	OFFA
R1	??
R2	??

Context switch - example

3) After temporary storage of registers (**KERNEL MODE**)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

Registers	
PC	5000 + ?
PS	AA45
SP	OFFA
R1	??
R2	??

4) After storage of registers of T1 and restore of registers of T2 (**KERNEL MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	25CC
0FFB	F012
0FFA	

Registers	
PC	5000+?
PS	AA45
SP	OFFA
R1	??
R2	??

Context switch - example

- 4) After storage of registers of T1 and restore of registers of T2 (**KERNEL MODE**)

TCB T1		TCB T2		Kernel stack		Registers	
State	Waiting	State	Running	0FFF	A12C	PC	5000+?
PC	1880	PC	A12C	0FFE	16F2	PS	AA45
PS	16F2	PS	16F2	0FFD	A275	SP	OFFA
SP	2880	SP	A275	0FFC	25CC	R1	??
R1	4500	R1	25CC	0FFB	F012	R2	??
R2	CD31	R2	F012	0FFA			

- 5) During execution of IRET at address 5100 (**KERNEL MODE**)

TCB T1		TCB T2		Kernel stack		Registers	
State	Waiting	State	Running	0FFF	A12C	PC	5100
PC	1880	PC	A12C	0FFE	16F2	PS	AA45
PS	16F2	PS	16F2	0FFD	A275	SP	OFFC
SP	2880	SP	A275	0FFC		R1	25CC
R1	4500	R1	25CC	0FFB		R2	F012
R2	CD31	R2	F012	0FFA			

Context switch - example

- 5) During execution of IRET at address 5100 (**KERNEL MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	
0FFB	
0FFA	

Registers	
PC	5100
PS	AA45
SP	OFFC
R1	25CC
R2	F012

- 6) At the end of IRET (**USER MODE**)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

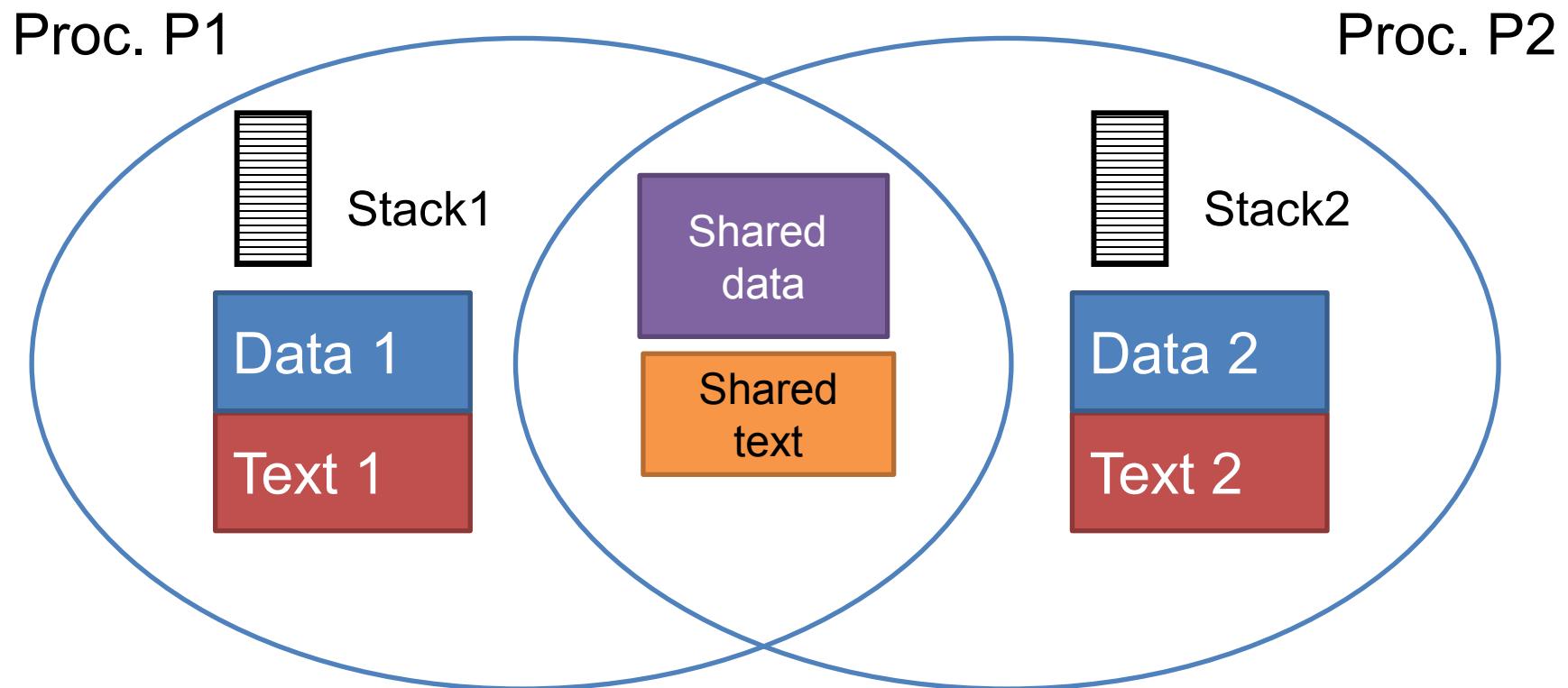
Registers	
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

Cooperation models

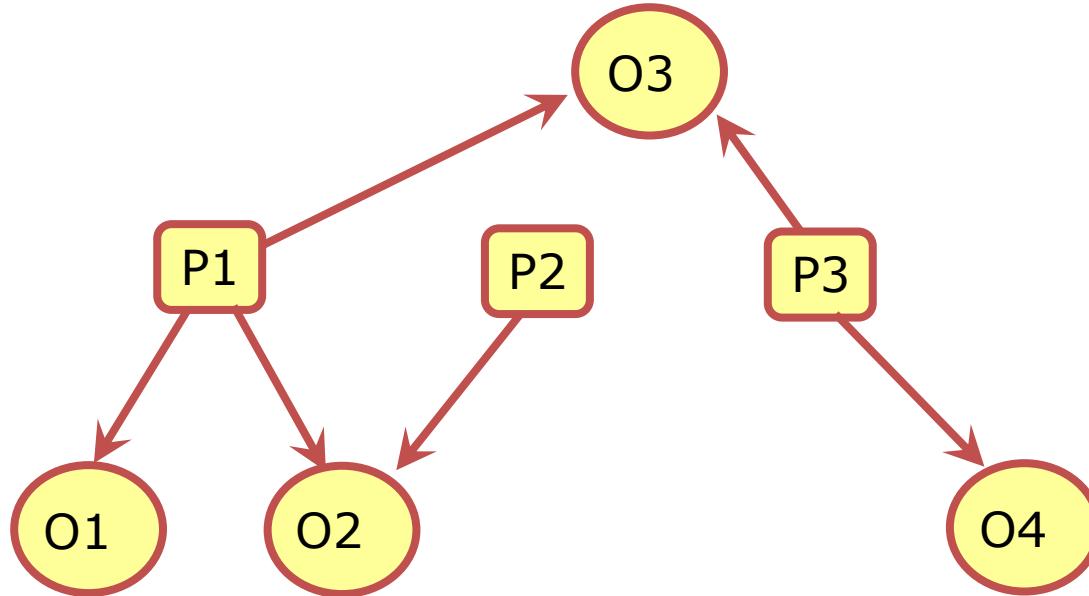
Global vs. local model

- Global environment
 - Processes/threads can share data
 - Cooperation via *shared memory variables*
- Local environment
 - processes/threads do not share data
 - No shared memory
 - Cooperation via *explicit messages*

Global environment



Global environment

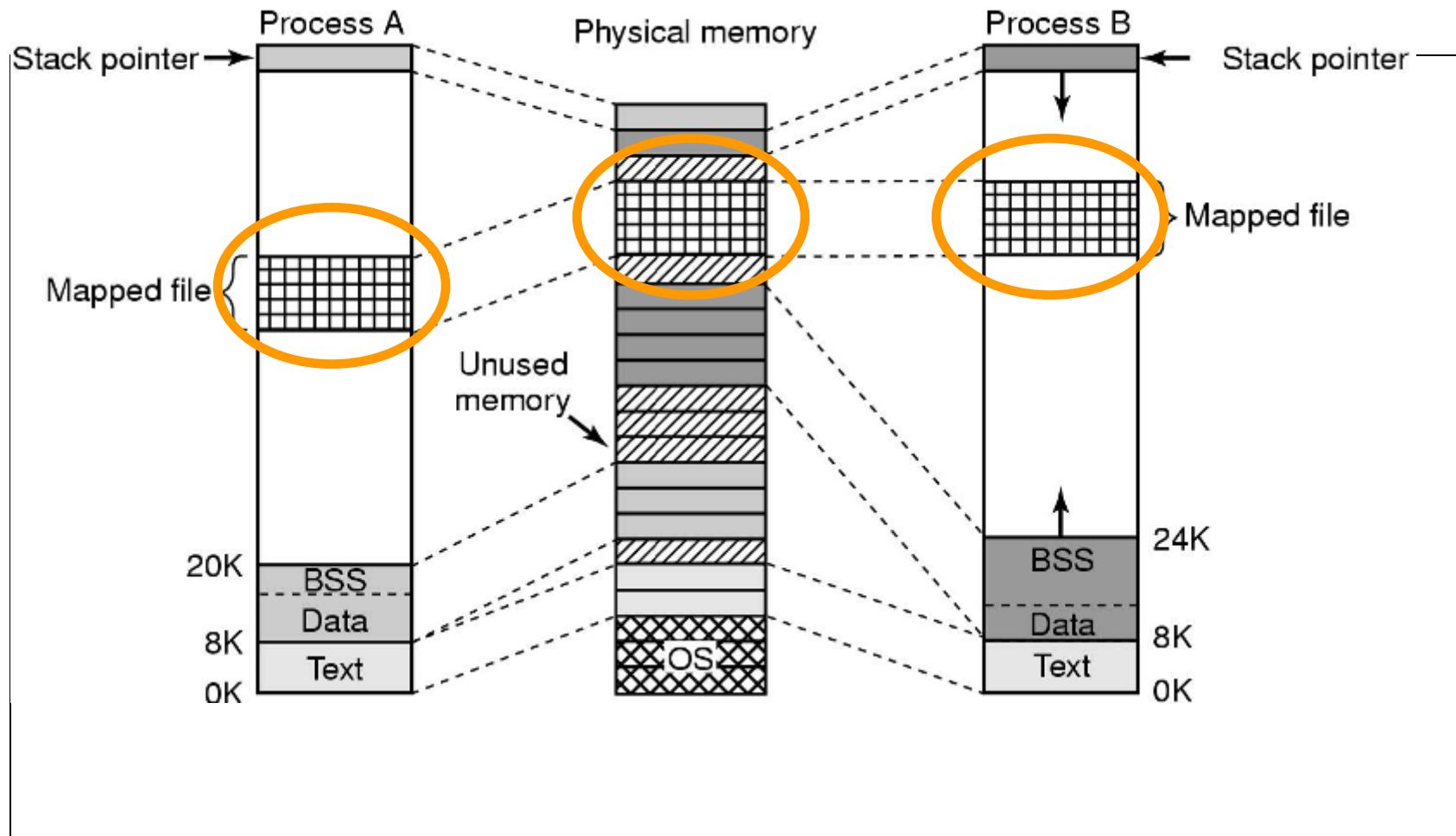


O1, O4 private objects

O2, O3 shared objects

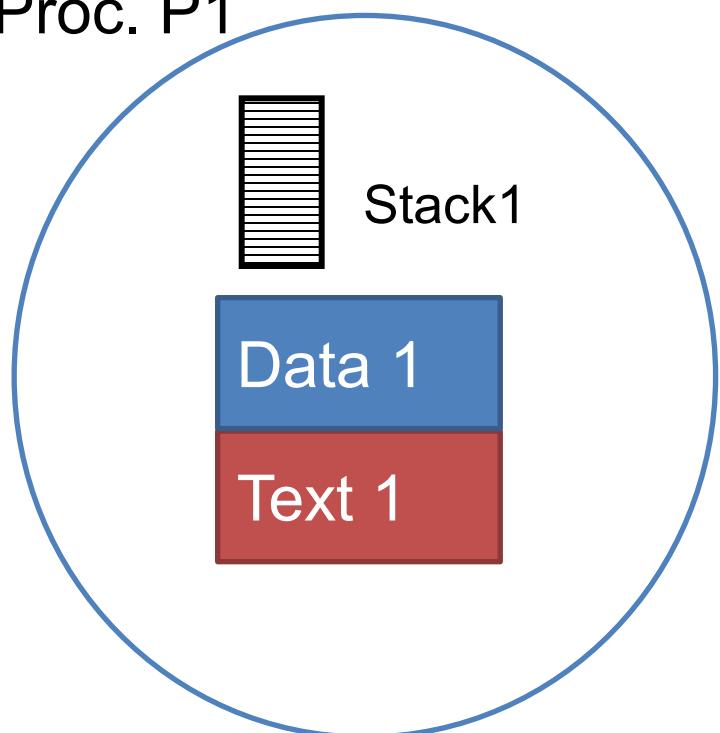
- competition, cooperation

Segment sharing using memory mapped file in Unix

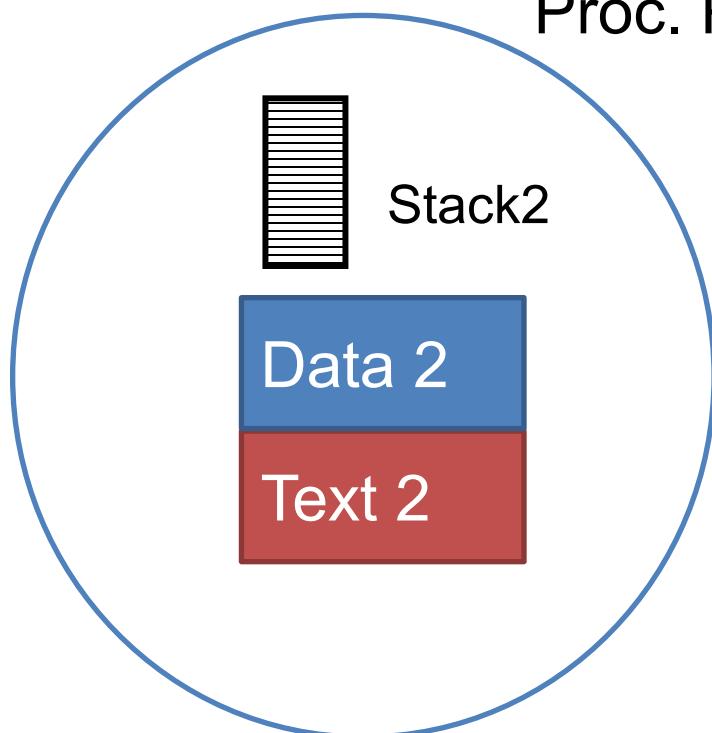


Local environment

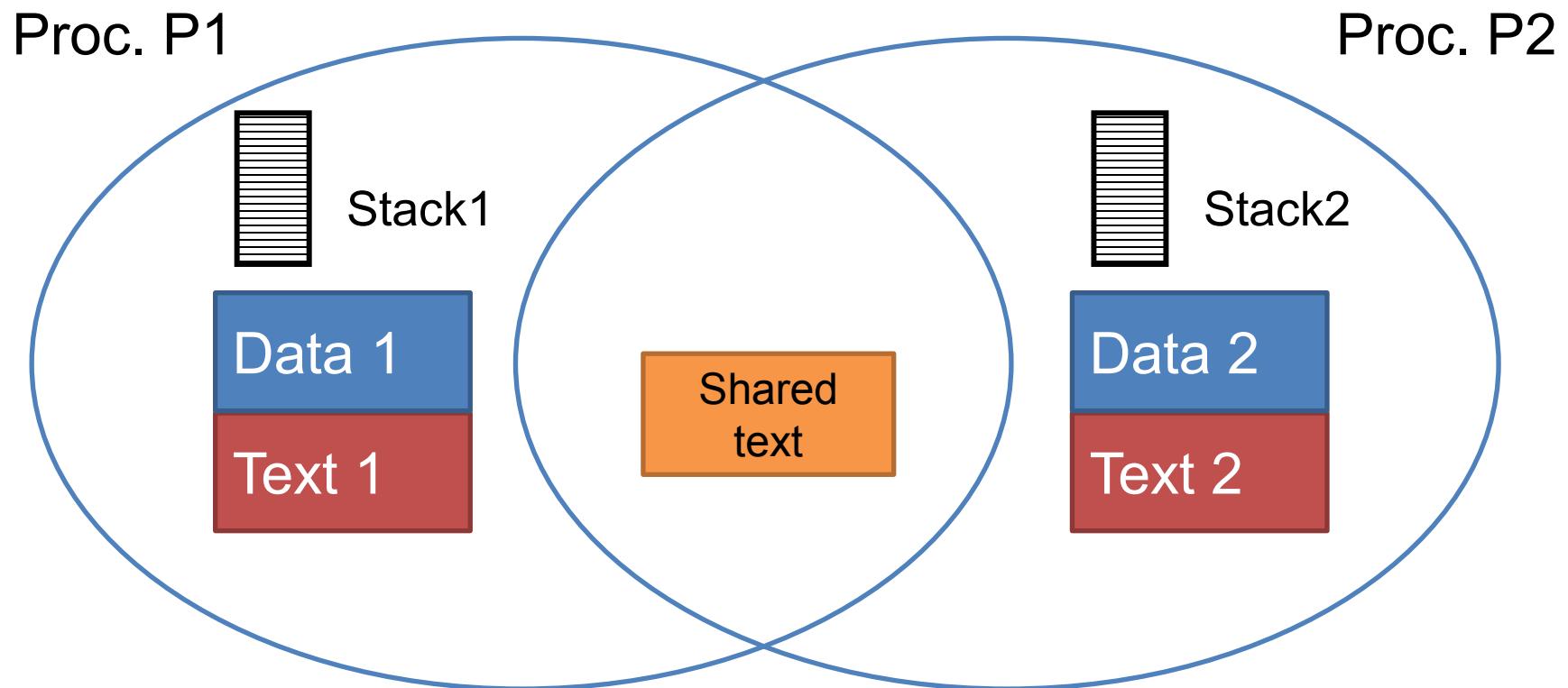
Proc. P1



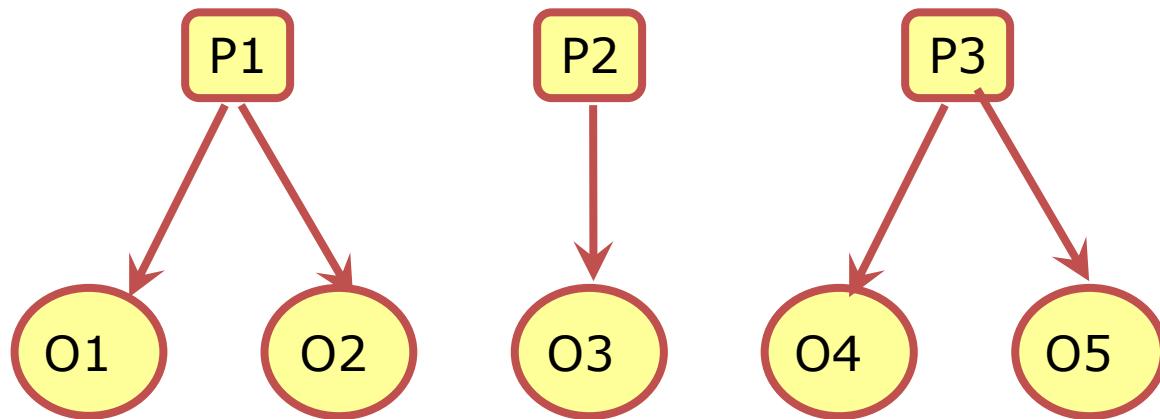
Proc. P2



Local environment



Local environment

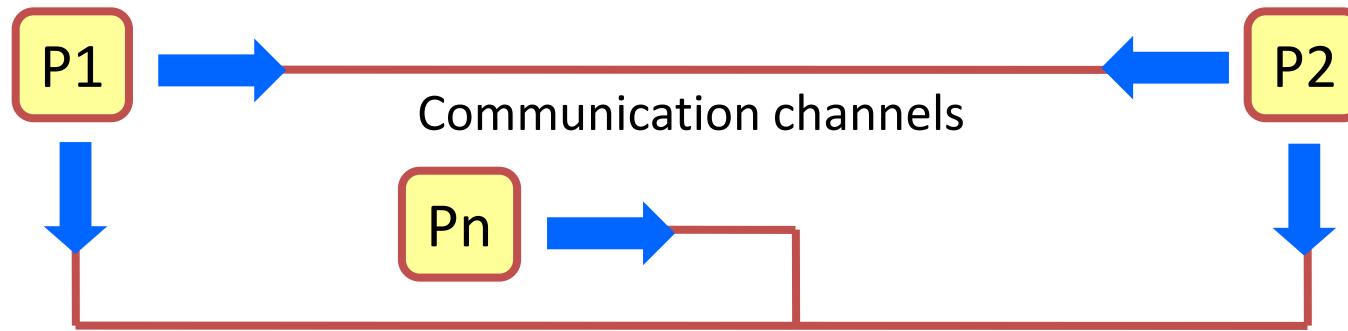


O1-O5 are private objects

Competition through server processes

Cooperation through explicit communication

Local environment



Cooperation (communication, synchronization) by means of message passing

Synchronization

Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
 - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
 - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

Can this panic?

NOTE: all functions do not have side effects, i.e., they are stateless!

Thread 1

```
p = someFn();  
isInitialized = true;
```

Thread 2

```
while (! isInitialized) ;  
q = aFn(p);
```

```
if ( q != aFn(someFn()) )
```

panic

Can this panic?

NOTE: all functions do not have side effects, i.e., they are stateless!

Thread 1

```
p = someFn();  
isInitialized = true;
```

Thread 2

```
while (! isInitialized) ;  
q = aFn(p);
```

```
if ( q != aFn(someFn()) )  
panic
```

Yes, it can panic!!

Compiler and processor may reorder operations!!

Why Reordering?

- Why do compilers reorder instructions?
 - Efficient code generation requires analyzing control/data dependency
 - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
 - Write buffering: allow next instruction to execute while write is being completed
- How to fix it? **Memory barrier** instructions
 - Instruction to compiler/CPU
 - All ops before barrier complete before barrier returns
 - No op after barrier starts until barrier returns
 - **NOTE:** synchronizations (e.g., locks) introduce implicit memory barrier!

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions

- **Race condition** : output of a concurrent program depends on the order of operations between threads (ops interleaving)
- **Mutual exclusion**: only one thread does a particular thing at a time
 - **Critical section**: piece of code that only one thread can execute at once
- **Lock**: Synchronization variable that provides mutual exclusion, preventing someone from doing something
 - Lock before entering critical section, i.e., before accessing shared data
 - Unlock when leaving, after done accessing shared data
 - wait if locked (**all synchronizations involves waiting either active or passive waiting!!**)

Too Much Milk, Try #1

- Correctness property
 - Someone buys if needed (**liveness**)
 - At most one person buys (**safety**)
- Try #1: leave a note...

Too Much Milk, Try #1

Thread A

```
if (!note) {  
    if (!milk)  
        leave note  
        buy milk  
        remove note  
}
```

Thread B

```
if (!note){  
    if (!milk)  
        leave note  
        buy milk  
        remove note  
}
```

Too Much Milk, Try #2

Thread A

```
leave note A  
if (!note B) {  
    if (!milk)  
        buy milk  
}  
remove note A
```

Thread B

```
leave note B  
if (!noteA){  
    if (!milk)  
        buy milk  
}  
remove note B
```

Too Much Milk, Try #3

Thread A

```
leave note A  
while (note B) // X  
    do nothing;  
if (!milk)  
    buy milk;  
remove note A
```

Thread B

```
leave note B  
if (!noteA){ // Y  
    if (!milk)  
        buy milk  
    }  
remove note B
```

Can guarantee at X and Y that either:

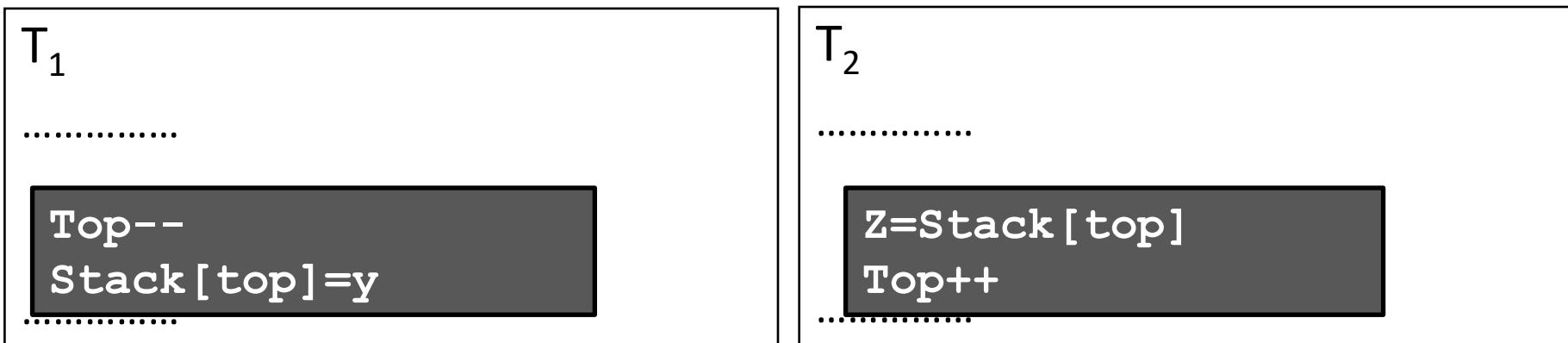
- i) Safe for me to buy
- ii) Other will buy, ok to quit

Lessons

- Solution is complicated
 - “obvious” code often has bugs
 - Demonstrating correctness is not trivial
- Modern compilers/architectures reorder instructions
 - Making reasoning even more difficult
- Generalizing to many threads/processors even more complex
 - Peterson’s algorithm generalization
 - **NOTE:** Peterson’s algorithm is not guaranteed to work on modern processors due to out-of-order execution of instructions. It would require careful insertion of memory barrier instructions (see Lamport’s circular buffer)!

Another example: a stack

- Two threads interact by means of a stack, with push and pop operations
- Expected behaviour:



Critical sections

Another example: a stack

- Here the operations of T_1 are interrupted by a pop executed by T_2
- Possible behaviour:

T_1

.....

1 Top--

3 Stack [top]=y

.....

T_2

.....

2 Z=Stack [top]

Top++

.....

Interference!

Another example: Lamport's circular buffer

- Lamport's Single-Producer Single-Consumer circular buffer uses only load/store instructions (1977). Non-blocking (nbk) algorithm → **busy waiting**

```
nbk_push(data) {  
    if (NEXT(head) == tail)  
        return 0; // buffer full  
    buffer[head] = data;  
    head = NEXT(head);  
    return 1; // OK  
}
```

```
nbk_pop(data) {  
    if (head == tail)  
        return 0; // buffer empty  
    data = buffer[tail];  
    tail = NEXT(tail);  
    return 1; // OK  
}
```

**Doesn't work
on modern
processors!!!**



Non-trivial modifications!

```
nbk_push(data) {  
    if (buffer[tail] != BOTTOM)  
        return 0; // buffer full  
    WMB(); // memory barrier  
    buffer[tail]=data;  
    tail=NEXT(tail);  
    return 1; // OK  
}
```

```
nbk_pop(data) {  
    if (buffer[head] != BOTTOM)  
        return 0; // buffer empty  
    data=buffer[head];  
    buffer[head]=BOTTOM;  
    head=NEXT(head);  
    return 1; // OK  
}
```

**Modified
version for
modern
processors**

BOTTOM is a value that cannot be stored into the buffer.
If we store data pointers into the buffer, *BOTTOM* can be *NULL*

Lock semantics

- lock.acquire()
 - wait until lock is free, then take it
 - lock.release()
 - release lock, *waking up anyone waiting for it*
-
1. At most one lock holder at a time (*safety*)
 2. If no one holding, acquire gets lock (*liveness*)
 3. If lock holder finishes and no higher priority waiters, waiter eventually gets lock (*liveness*)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

lock.acquire()

if (!milk) buy milk

lock.release()

- How do we implement locks? (discussed later)
 - We need HW support for RMW instructions

Lock Example: Malloc/Free

```
char *malloc (n) {  
    heaplock.acquire();  
    p = allocate memory  
    heaplock.release();  
    return p;  
}
```

```
void free(char *p) {  
    heaplock.acquire();  
    put p back on free list  
    heaplock.release();  
}
```

(*) real implementation of malloc and free more complex than this example

Rules for Using Locks

- Lock is initially free
- ***Always*** acquire before accessing shared data structure
 - Beginning of procedure!
- ***Always*** release after finishing with shared data
 - End of procedure!
 - DO NOT throw lock for someone else to release
- ***Never access shared data without lock***
 - Danger!

Will this code work?

[...]

```
if (p == NULL) {  
    lock.acquire();  
    if (p == NULL) {  
        p = newP();  
    }  
    lock.release();  
}  
use p->field1
```



Where:

```
newP() {  
    p = malloc(sizeof(p));  
    p->field1 = ...  
    p->field2 = ...  
    return p;  
}
```

Lock example: Bounded Buffer

```
tryget() {  
    item = NULL;  
    lock.acquire();  
    if (nelem>0) {  
        item = buf[front];  
        front = (front++)%size;  
        nelem --;  
    }  
    lock.release();  
    return item;  
}
```

```
tryput(item) {  
    r=false;  
    lock.acquire();  
    if (nelem < size) {  
        buf[last] = item;  
        last = (last ++)%size;  
        nelem ++; r=true;  
    }  
    lock.release();  
    return r;  
}
```

Initially: $nelem = front = last = 0$; $lock = FREE$;
 $size$ is buffer capacity

Questions

- If *tryget* returns NULL, do we know the buffer is empty?
 - *No, we only know the buffer was empty*
- If we poll *tryget* in a loop, what happens to a thread calling *tryput*?
 - *It could be delayed in acquiring the lock (probably it will be)*

Condition Variables

- Waiting inside a critical section
 - Synchronization without busy waiting can be achieved using *condition variables and mutex locks*
- Operations: *wait*, *signal*, *broadcast*

Called only when holding a lock!
- ***wait***: **atomically** releases lock and relinquishes processor until signaled
- ***signal***: wake up a waiter, if any
- ***broadcast***: wake up all waiters, if any

Condition Variable Design Pattern

```
methodThatWaits() {  
    lock.acquire();  
    // Read/write shared state  
  
    while (!testSharedState()) {  
        cv.wait(&lock);  
    }  
  
    // Read/write shared state  
    lock.release();  
}
```

```
methodThatSignals() {  
    lock.acquire();  
    // Read/write shared state  
  
    // If testSharedState is now true  
    cv.signal(&lock);  
  
    // Read/write shared state  
    lock.release();  
}
```

Classic problem: Producer-Consumer

Definition:

- One (or more) producer (s) deposits messages in a *shared buffer*
- One (or more) consumer (s) extracts messages from the shared buffer

Requirements:

- Each produced message must be consumed exactly once.

Variants:

- The shared buffer can have a bounded or unbounded capacity

Example: Bounded Buffer

```
get() {  
    lock.acquire();  
    while (nelem == 0)  
        empty.wait(lock);  
    item = buf[front];  
    front = (front++) % size;  
    nelem --;  
    full.signal();  
    lock.release();  
    return item;  
}
```

```
put(item) {  
    lock.acquire();  
    while (nelem == size)  
        full.wait(lock);  
    buf[last] = item;  
    last = (last++) % size;  
    nelem ++;  
    empty.signal();  
    lock.release();  
}
```

Initially: $nelem = front = last = 0$; $size$ is buffer capacity
 $empty/full$ are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
 - $\text{nelem} \geq 0$ (and $\text{front} \leq \text{last}$, without considering the module operation)
 - $\text{nelem} \leq \text{size}$ (and $\text{front} + \text{size} \geq \text{last}$, without considering the module operation)
 - (also true on return from wait)
- Also true at lock release!
- Allows for proof of correctness

Condition Variable Design Pattern

```
funcThatWaits() {  
    lock.acquire();  
    // Pre-condition: State is consistent  
  
    // Read/write shared state  
  
    while (!testSharedState()) {  
        cv.wait(&lock);  
    }  
    // WARNING: shared state may  
    // have changed! But  
    // testSharedState is TRUE  
    // and pre-condition is true  
  
    // Read/write shared state  
    lock.release();  
}  
  
funcThatSignals() {  
    lock.acquire();  
    // Pre-condition: State is consistent  
  
    // Read/write shared state  
  
    // If testSharedState is now true  
    cv.signal(&lock);  
  
    // NO WARNING: signal keeps lock  
  
    // Read/write shared state  
    lock.release();  
}
```

Condition Variables

- **ALWAYS** hold lock when calling wait, signal, broadcast
 - Condition variables provide synchronization FOR shared state
 - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
 - If signal when no one is waiting, no op
 - If wait before signal, waiter wakes up
- Wait atomically releases lock
 - What if wait, then release?
 - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
 - Signal/broadcast put thread on ready list
 - When lock is released, anyone might acquire it
- Wait **MUST** be in a loop

```
while (needToWait())
    condition.Wait(lock);
```
- Simplifies implementation
 - Of condition variables and locks
 - Of code that uses condition variables and locks

Java Manual

When waiting upon a Condition, a “*spurious wakeup*” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition *should always be waited upon in a loop*, testing the state predicate that is being waited for.

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
 - Grab lock on start to every method/procedure
 - Release lock on finish
- If need to wait
 - `while(needToWait()) { condition.Wait(lock); }`
 - Do not assume, when you wake up, that signaller just ran
- If do something that might wake someone up
 - Signal or Broadcast
- Always leave shared state variables in a consistent state
 - When lock is released, or when waiting

Remember the rules

- Use consistent structure
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- *Always wait in while loop*
 - *Waiters may be awakened by spurious wake up!*
- Never spin in sleep()

Mesa vs. Hoare semantics

- Mesa (in textbook, Hansen)
 - Signal puts waiter on ready list
 - Signaler keeps lock and processor
- Hoare
 - Signal gives processor and lock to waiter
 - When waiter finishes, processor/lock given back to signaller
 - Nested signals possible!

Mesa & Hoare semantics

- The bounded buffer solution adopted for the Producer-Consumer problem works well with both Mesa & Hoare semantics
- However, it does not make any assumption on the order in which:
 - The producers that are waiting are woken up and deposit their messages
 - The consumers that are waiting are woken up

FIFO Bounded Buffer (Hoare semantics)

```
get() {                                put(item) {  
    lock.acquire();                      lock.acquire();  
    while (nelem == 0)                  while (nelem == size)  
        empty.wait(lock);                full.wait(lock);  
    item = buf[front];                   buf[last] = item;  
    front= (front++) % size;            last = (last++) % size;  
    nelem --;                          nelem++;  
    full.signal(lock);                 empty.signal(lock);  
    lock.release();                   // CAREFUL: someone else ran  
    return item;                      lock.release();  
}  
}
```

Initially: $nelem = front = last = 0$; $size$ is buffer capacity
 $empty/full$ are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!
- Easily extends to case where queue is LIFO, priority, priority donation, ...
 - With Hoare semantics, not as easy

FIFO Bounded Buffer (Mesa semantics, put() is similar)

```
get() {  
    lock.acquire();  
    if ( nelem == 0 ||  
        !nextGet.empty() ) {  
        self = createCondition();  
        nextGet.Append(self);  
        do self.wait(lock);  
        while (nelem == 0);  
        nextGet.Remove(self);  
        destroyCondition(self);  
    }  
    item = buf[front];  
    front= (front++) % size;  
    nelem --;  
    if (!nextPut.empty())  
        nextPut.first()->signal();  
    lock.release();  
    return item;  
}
```

Initially: $nelem = front = last = 0$; $size$ is buffer capacity
 $nextGet, nextPut$ are queues of Condition Variables

Implementing Synchronization

Concurrent Applications

Shared Objects

Bounded Buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt Disable Test-and-Set

Hardware

Multiple Processors Hardware Interrupts

Implementing Synchronization

Take 1: SW implementation using memory load/store

- See “too much milk” solution / Peterson’s algorithm

Take 2:

```
lock.acquire() { disable interrupts }
```

```
lock.release() { enable interrupts }
```

Take 3: SW implementation with HW support...

Lock Implementation, Uniprocessor

```
LockAcquire() {  
    disableInterrupts();  
    if (value == BUSY) {  
        waiting.add(myTCB);  
        suspend(); /*  
    } else {  
        value = BUSY;  
    }  
    enableInterrupts();  
}  
* Invokes the scheduler,  
  context switch & enables  
  interrupts
```

```
LockRelease() {  
    disableInterrupts();  
    if (!waiting.Empty()) {  
        thTCB = waiting.Remove();  
        readyList.Append(thTCB);  
    } else {  
        value = FREE;  
    }  
    enableInterrupts();  
}
```

LOCK threw to the
awakened thread!

Multiprocessor

- In multiprocessor, disabling interrupts is not enough
- We need Read-Modify-Write (RMW) instructions
 - Atomically read a value from memory, operate on it, and then write it back to memory
 - Intervening instructions prevented in hardware
- Examples
 - Test-and-set and Compare-and-Swap (CAS) operations
 - Intel: xchgb, lock prefix
 - ARM: LDREX/STREX (“Load-link/Store-conditional”-like ops)
- Does it matter which type of RMW instructions we use?
 - Not for implementing locks and condition variables!

Spinlocks

A Spinlock is a Lock where the processor waits in a loop for the lock to become free (**active waiting!**)

- Assumes lock will be held for a short time
- Used to protect ready list and to implement locks

```
SpinlockAcquire() {  
    while (TestAndSet(&spinLockValue) == BUSY)  
        ;  
}
```

```
SpinlockRelease() {  
    spinLockValue = FREE;  
    memory_barrier();  
}
```

Test-and-Set example

- Let's suppose we have the following *Test-and-Set Lock* (TSL) instruction

```
TSL REG, &ADDR      // REG ← MEMORY[ADDR]  
                      // MEMORY[ADDR] ← #BUSY
```

- The two memory accesses are atomic. Atomicity granted by the HW memory controller.
- TSL is provided in different ways in real processors

Spinlocks implementation with TSL

```
// &spinLockValue is a memory cell containing a binary value: FREE (0) or BUSY (1)
// TSL R, &spinLockValue :
    writes the content of &spinLockValue in R and writes BUSY (1) in
    &spinLockValue

spinlockAcquire(&spinLockValue) {
    Loop:   TSL R, &spinLockValue
            CMP R, #BUSY
            BEQ Loop           // jump if last comparison was successful
            END                // at this point &spinLockValue == BUSY!!!!
}

spinlockRelease(&spinLockValue) {
    MOV #FREE, &spinLockValue // this unlocks a thread in the loop, if any
    MFENCE                  // memory barrier
}
```

Spinlock implementation in ARM Linux Kernel

```
static inline
void arch_spin_lock(arch_spin_lock *lock){
    unsigned long tmp;
    __asm__ __volatile__(
        "1:    ldrex   %0, [%1]\n"
        "       teq     %0, #0\n"
        WFE("ne")
        "       strexeq %0, %2, [%1]\n"
        "       teqeq   %0, #0\n"
        "       bne     1b"
        : "&r" (tmp)
        : "&lock->lock", "r" (1)
        : "cc");
    smp_mb();
}
```

```
static inline
void arch_spin_unlock(arch_spinlock_t *lock){
    smp_mb();
    __asm__ __volatile__(
        "       str     %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");
    dsb_sev();
}
```

Spinlock implementation in ARM Linux Kernel

```
static inline
void arch_spin_lock(arch_spin_lock *lock){
    unsigned long tmp;
    __asm__ __volatile__(
        "1: ldrex %0, [%1]\n"
        "      teq   %0, #0\n"
        "      WFE(\"ne\")\n"
        "      strexeq %0, %2, [%1]\n"
        "      teqeq %0, #0\n"
        "      bne   1b"
        : "<=r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc");
    smp_mb();
}
```

```
static inline
void arch_spin_unlock(arch_spinlock_t *lock){
    smp_mb();
    __asm__ __volatile__(
        "      str   %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");
    dsb_sev();
}
```

Load lock->lock from memory and check whether it is 0.
LDREX (*load exclusive*) loads from memory to register and marks memory location as exclusive.

Try to store 1 into lock->lock to acquire the exclusive ownership of the memory location. If the returned value is not 0 we jump again to the LDREX.

If not 0, WFE sets CPU into power saving mode until it receives an interrupt or an event

Memory barrier

Spinlock implementation in ARM Linux Kernel

```
static inline
void arch_spin_lock(arch_spin_lock *lock){
    unsigned long tmp;
    __asm__ __volatile__(
        "1:    ldrex   %0, [%1]\n"
        "       teq    %0, #0\n"
        WFE("ne")
        "       strexeq %0, %2, [%1]\n"
        "       teqeq  %0, #0\n"
        "       bne    1b"
        : "&r" (tmp)
        : "&lock->lock", "r" (1)
        : "cc");
    smp_mb();
}
```

```
static inline
void arch_spin_unlock(arch_spinlock_t *lock){
    smp_mb();
    __asm__ __volatile__(
        "       str    %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");
    dsb_sev();
}
```

Memory barrier operation to commit all previous changes in memory

Stores 0 to lock->lock for releasing the it

Sends an event to other processors which are waiting at WFE

Lock Implementation, Multiprocessor

```
LockAcquire(){  
    disableInterrupts();  
    spinLockAcquire(&spinLock);  
    if (value == BUSY){  
        waiting.add(myTCB);  
        sched.suspend(&spinLock);  
    } else {  
        value = BUSY;  
        spinLockRelease(&spinLock);  
    }  
    enableInterrupts();  
}
```

scheduler: marks thread as waiting;
releases spinlock; schedules next thread;

```
LockRelease() {  
    disableInterrupts();  
    spinLockAcquire(&spinLock);  
    if (!waiting.Empty()){  
        thTCB = waiting.Remove();  
        sched.resume (thTCB,);  
    } else value = FREE;  
    spinLockRelease(&spinLock);  
    enableInterrupts();  
}
```

scheduler: marks thread as ready,
puts it in the ready list.

What thread is currently running?

- Thread scheduler needs to find the TCB of the currently running thread
 - To suspend and switch to a new thread
 - To check if the current thread holds a lock before acquiring or releasing it
- On a uniprocessor, easy: just use a global
- On a multiprocessor, various methods:
 - Compiler dedicates a register
 - If hardware has a special per-processor register, use it
 - Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
 - Find it by masking the current stack pointer

Lock Implementation, Linux

- Fast path
 - If lock is FREE, and no one is waiting, TestAndSet
- Slow path
 - If lock is BUSY or someone is waiting, see previous slide
- User-level locks
 - Fast path: acquire lock using test&set
 - Slow path: system call to kernel, to use kernel lock

Semaphores

- Semaphore has a non-negative integer value
 - P() atomically waits for value to become > 0 , then decrements
 - V() atomically increments value (if no waiter is present); else it wakes up one waiter
- Semaphore's data structures are integer + queue:
 - Only operations are P and V
 - Operations are atomic
 - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
 - Unlocked wait: interrupt handler, fork/join

P&V Implementation, Multiprocessor

```
P(sem){  
    disableInterrupts();  
    spinLockAcquire(&spinLock);  
    if (sem.value == 0){  
        waiting.add(myTCB);  
        suspend(&spinLock);  
    } else {  
        sem.value --;  
        spinLockRelease(&spinLock);  
    }  
    enableInterrupts();  
}
```

Invokes scheduler; context switch;
enables interrupts

```
V(sem) {  
    disableInterrupts();  
    spinLockAcquire(&spinLock);  
    if (!waiting.Empty()){  
        thTCB = waiting.Remove();  
        readyList.Append(thTCB);  
    }  
    else sem.value ++;  
    spinLockRelease(&spinLock);  
    enableInterrupts();  
}
```

Throws semaphore to
the awakened thread!

Semaphore Bounded Buffer

```
get() {  
    empty.P(); ←  
    mutex.P(); →  
    item = buf[front];  
    front= (front+1) % size;  
    mutex.V(); ←  
    full.V(); ←  
    return item;  
}  
  
put(item) {  
    full.P(); →  
    mutex.P(); ←  
    buf[last] = item;  
    last = (last +1) % size;  
    mutex.V(); ←  
    empty.V(); ←  
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to **0** and **size**)
mutex is a semaphore initialized to **1**

Implementing Condition Variables using Semaphores (Take 1)

```
wait(lock) {           signal() {  
    lock.release();      sem.V();  
    sem.P();            }  
    lock.acquire();  
}  
}
```

Implementing Condition Variables using Semaphores (Take 2)

```
wait(lock) {           signal() {  
    lock.release();     if semaphore not empty  
    sem.P();            sem.V();  
    lock.acquire();     }  
}  
}
```

Implementing Condition Variables using Semaphores (Take 3)

```
wait(lock) {  
    sem =  
        createSemaphore();  
    // queue of waiting threads  
    queue.Append(sem);  
    lock.release();  
    sem.P();  
    destroySemaphore(sem);  
    lock.acquire();  
}
```

```
signal() {  
    if !queue.Empty() {  
        sem = queue.Remove();  
        sem.V(); // wake up waiter  
    }  
}
```

Monitors brief note

- To avoid potential issues related to wrong usage of semaphores and/or locks+CV, the ***monitor*** concept have been proposed as high-level language construct (e.g., java *synchronized* objects)
- A monitor is an ADT including a set of programmer-defined functions that execute in mutual exclusion within the monitor
- The monitor construct ensures that only one thread at a time is active within the monitor
- A monitor can be implemented with locks+CV or with semaphores
 - From our standpoint monitors are just “*syntactic sugar*”
- In C a monitor can be emulated with a struct with synchronization variables (e.g., lock+CV) and state variables as fields. Then, only a set of well-defined functions operate on the struct’s fields.
 - Consider get and put methods of a bounded buffer!

Synchronization Summary

- Use consistent structure
- Always use locks and condition variables (in user-code)
 - “Semaphores considered harmful” Dijkstra, 1968
 - ... still, they are widely used mainly in OSs
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Multi-Object Synchronization

Main Points

- Problems with synchronizing multiple objects
- Definition of deadlock
 - Circular waiting for resources
- Conditions for its occurrence
- Solutions for avoiding and breaking deadlock

Large Programs

- What happens when we try to synchronize across multiple objects in a large program?
 - Each object with its own lock, condition variables
 - Is concurrency modular?
- Deadlock
- Performance
- Semantics/correctness

Deadlock Definitions

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
 - Preemptable: can be taken away by OS (and later given back)
 - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
 - Deadlock => starvation, but not vice versa

Assumptions

We consider re-usable resources

- example: critical sections, printers, ...
- Request
 - example: lock.acquire, cond.wait(&lock), ...
 - hypothesis: *blocking request*
- Assignment and use
 - If need to wait, *keep the resource*, i.e., *wait while holding* the resource (non-preemptable resource)
- Release
 - example: lock.release, cond.signal(), ...

Example: two locks

Thread A

```
lock1.acquire();
```

...

```
lock2.acquire();
```

```
lock2.release();
```

```
lock1.release();
```

Thread B

```
lock2.acquire();
```

...

```
lock1.acquire();
```

```
lock1.release();
```

```
lock2.release();
```

Two locks and a condition variable

Thread A

```
lock1.acquire();
...
lock2.acquire();
while (need to wait)
    condition.wait(lock2);
lock2.release();
...
lock1.release();
```

Thread B

```
lock1.acquire();
...
lock2.acquire();
.....
condition.signal();
lock2.release();
...
lock1.release();
```

Bidirectional Bounded Buffer

Thread A

```
buffer1.put(data);
```

```
buffer1.put(data);
```

Thread B

```
buffer2.put(data);
```

```
buffer2.put(data);
```

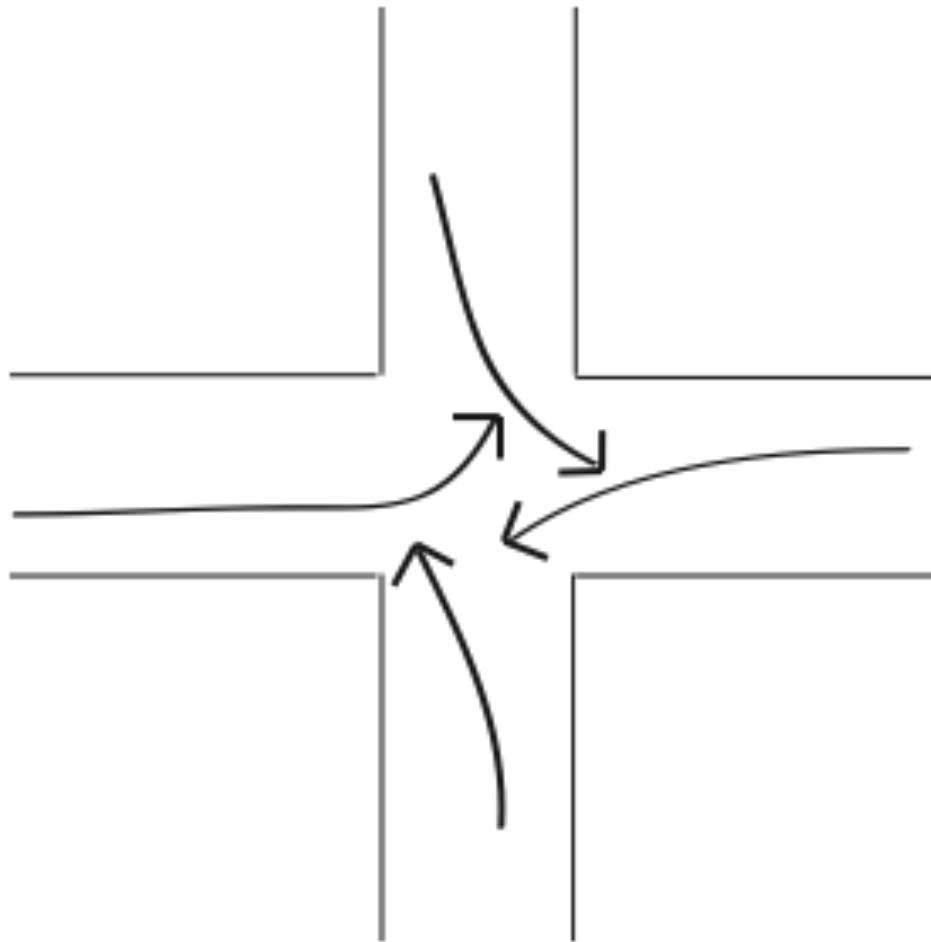
```
buffer2.get();
```

```
buffer2.get();
```

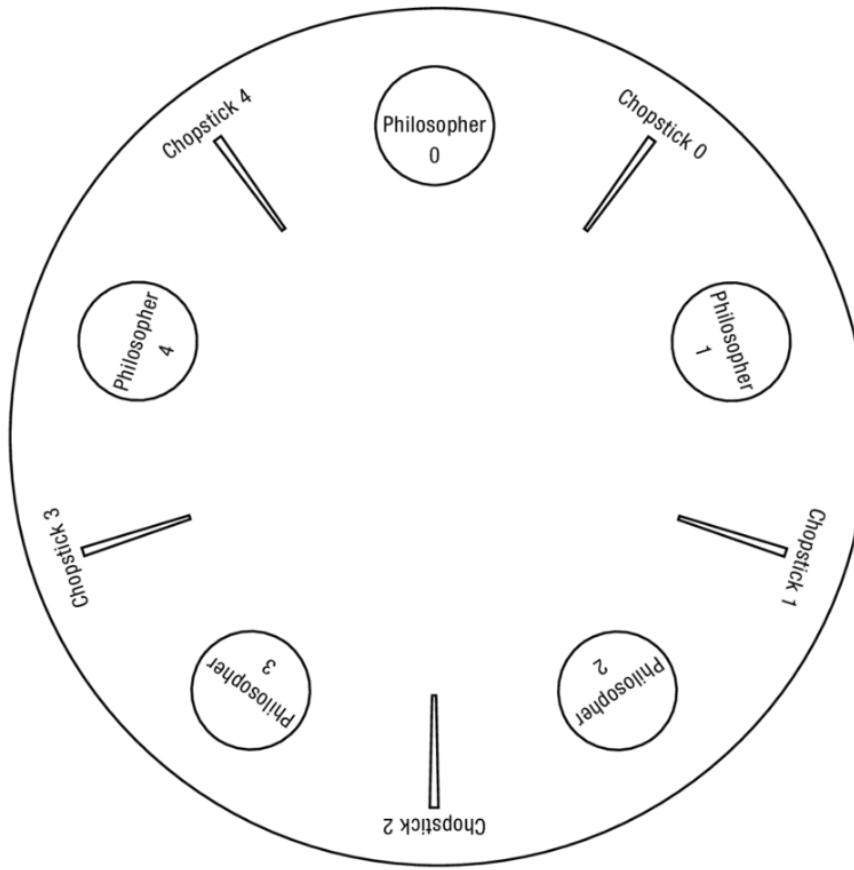
```
buffer1.get();
```

```
buffer1.get();
```

Yet another Example



Classic Problem: Dining Philosophers

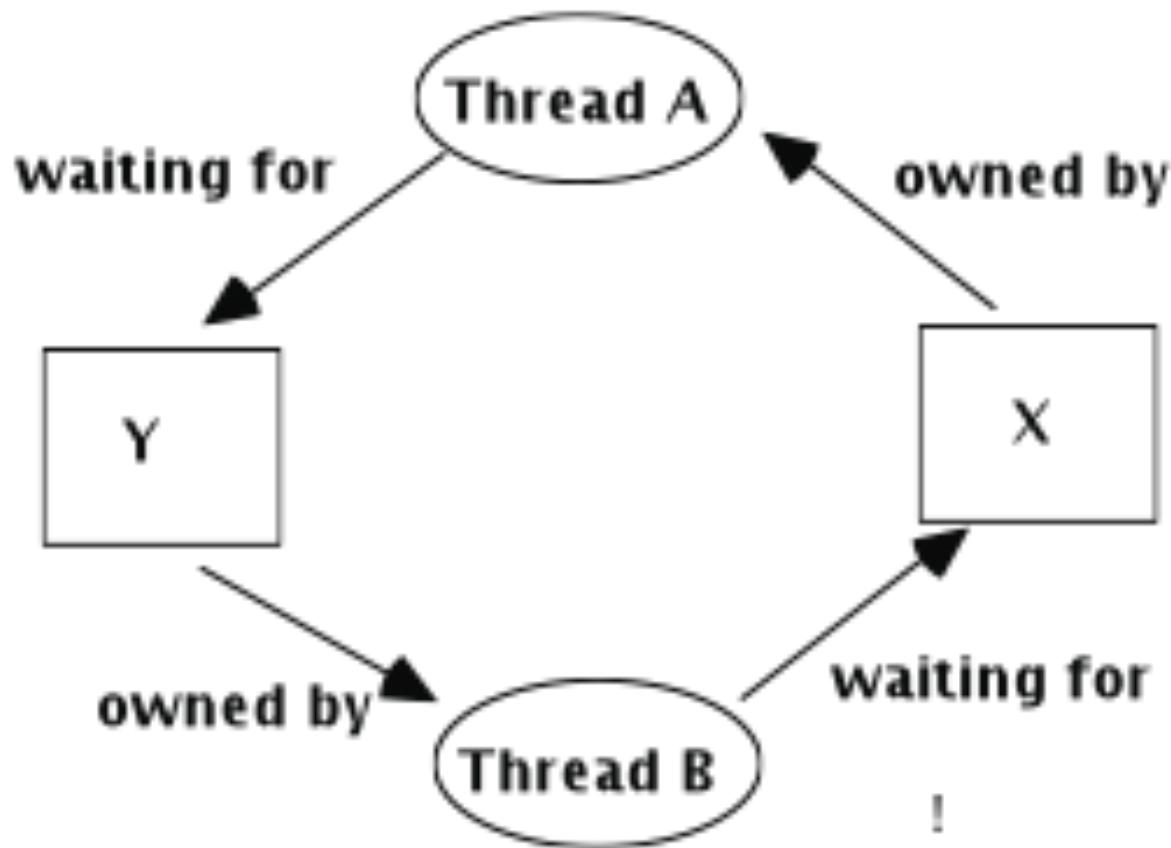


Each philosopher needs two chopsticks to eat.
Each grabs chopstick on the right first.

Conditions for Deadlock

- Limited access to resources
 - A limited number of threads can use simultaneously a resource
 - If at most one: mutual exclusion
 - If “infinite” resources (virtualized), no deadlock!
- No preemption
 - If resources are virtual, can break deadlock
- Wait while holding
 - A thread holds the assigned resources while waiting for another one (aka multiple independent requests)
- Circular chain of requests

Circular Waiting



Example: sequence NOT leading to a deadlock

A requests R

C requests T

A requests S

C requests R

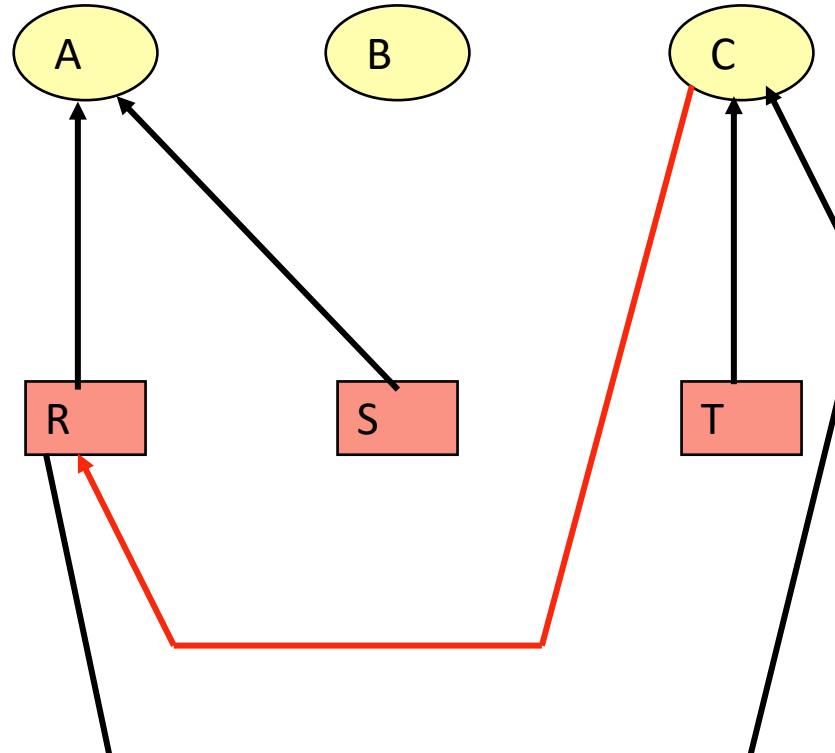
A releases R

A releases S

...

hyp: A needs only R and S

C needs only T and R



Example: sequence leading to a deadlock

A requests R

B requests S

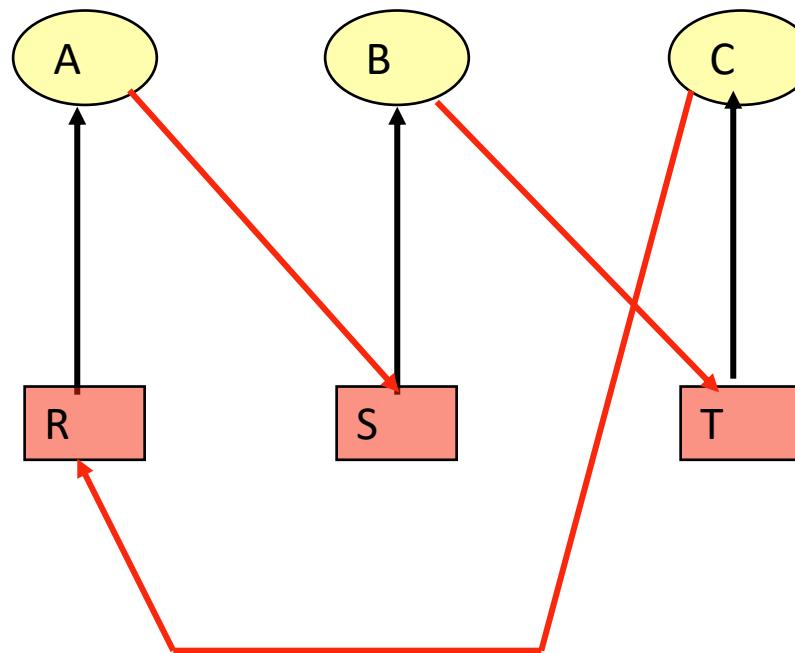
C requests T

A requests S

B requests T

C requests R

Deadlock!



Methods to deal with deadlock

- Detect and fix
- Static prevention
- Dynamic prevention (banker's algorithm)

In practice many systems do not adopt any solution

- The case of Unix, Windows,...
- Make sense if:
 - The cost for prevention or deadlock fixing is too high
 - The system is plentiful of resources (and the chance for a deadlock is very low)

Solution #1: Detect and Fix

- Algorithm
 - Scan the graph
 - Detect cycles
 - Fix cycles
- How?
 - Remove one thread, reassign its resources
 - Requires exception handling code to be very robust
 - Roll back actions of one thread
 - Databases: all actions are provisional until committed

Solution #1: Detect and Fix

- Thread suppression
 - Very simple, rather rough
 - The resources assigned to the suppressed thread can be re-assigned
 - Needs for criteria for the selection of the thread
 - “minimum impact”

Solution #1: Detect and Fix

- Roll-back actions:
 - Restore a previous state of the processes
 - The processes start again
 - Luckily, the deadlock will not appear again
- Requires check-pointing
 - Periodic storage of the internal state of the processes
 - Memory, registers, files,...
- No surprise OSs generally prefer thread suppression

Solution #2: Deadlock Prevention

Eliminate one of the four conditions for deadlock

- Lock ordering (avoids circular waiting)
 - Always acquire locks in the same order
 - Example: acquire resources in alphabetical order (always!!)
 - Widely used in OS kernels
- Design system to release resources and retry if need to wait
 - No “wait while holding”
 - Example: telephone circuit setup. If busy, retry!
- Infinite resources? Not always possible!
- Acquire all needed resources in advance
 - No “wait while holding”
 - Ex: UNIX reserves a process for the sysadmin to run “kill”

Solution #2: Deadlock Prevention

No “wait while holding”

- A process should know in advance all the resources it needs
- It requests in advance the resources
 - If not available -> wait *without holding!*
- Drawbacks
 - Need to know many things...
 - Locks resources that can be used by other processes

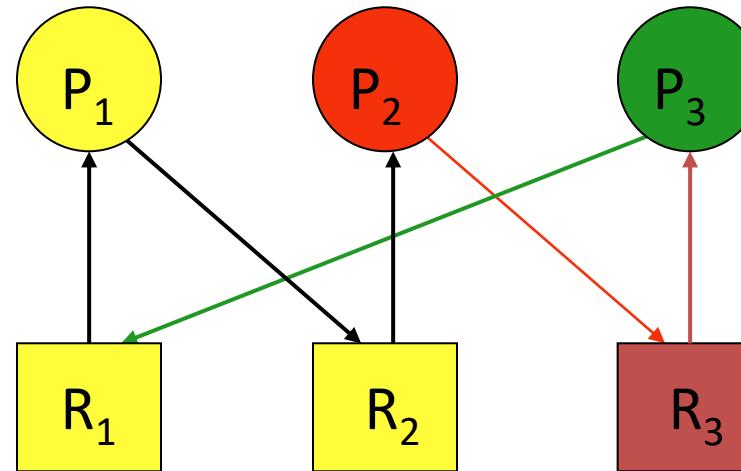
Solution #2: Deadlock Prevention

- Some devices can be managed with a spool
- Example: a printer
 - The spool virtualizes the printer (creates infinite printers)
 - Each process prints on a virtual printer
 - In practice, requests the print to the spool
 - There is no longer mutual exclusion: the physical printer is assigned to the spool process
- Solves the problem, but:
 - Can be applied only in some cases
 - There still can be deadlocks in the buffer used by the spool manager

Solution #2: Deadlock Prevention

Lock ordering

- Requests to resources must be ordered



R3: Resource with maximum index

P_3 violates the constraint of sorted requests

P_3 causes circular waiting

Solution #2: Deadlock Prevention

Summary

Condition	Method
Limited access to resources:	Spool / virtualization
“wait while holding”:	Request all resources in advance
Circular waiting:	Lock ordering

Multiplicity of resources

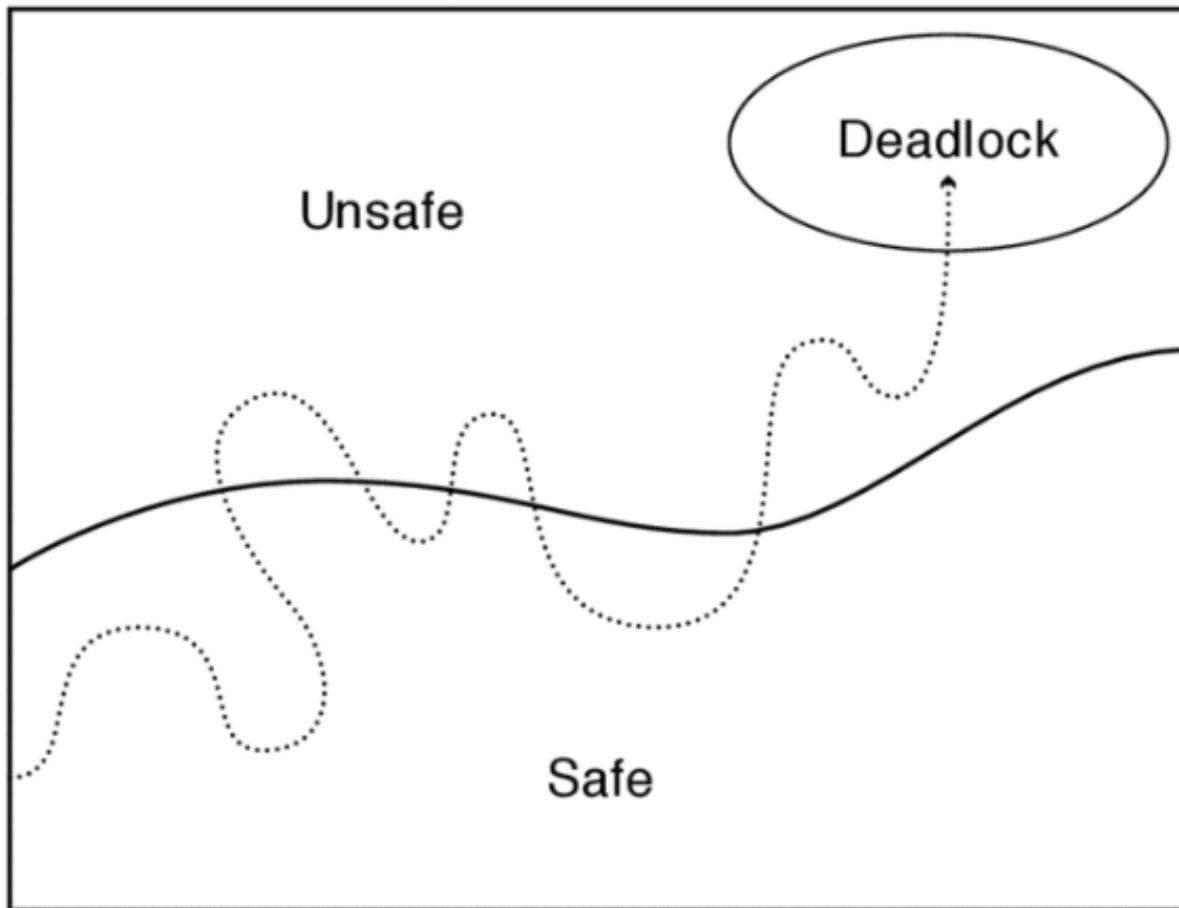
- Resources classified according to their type
 - Multiplicity M:
 - M is the number of resources of a given type
 - Availability D:
 - Number of resources of a given kind that are currently available
- Method of request:
 1. Single resource
 2. Multiple resources
 - A process requests k resources of a given kind
 - If $k \leq D$ then all k resources are assigned
 - If $k > D$ then no resource is assigned (and the process waits)

Solution #3: Banker's Algorithm

- Banker's algorithm
 - State maximum resource needs in advance
 - Allocate resources dynamically when resource is needed -- wait if granting request would lead to deadlock
 - Request can be granted if some sequential ordering of threads is deadlock free

The banker is the resource manager!

Possible System States

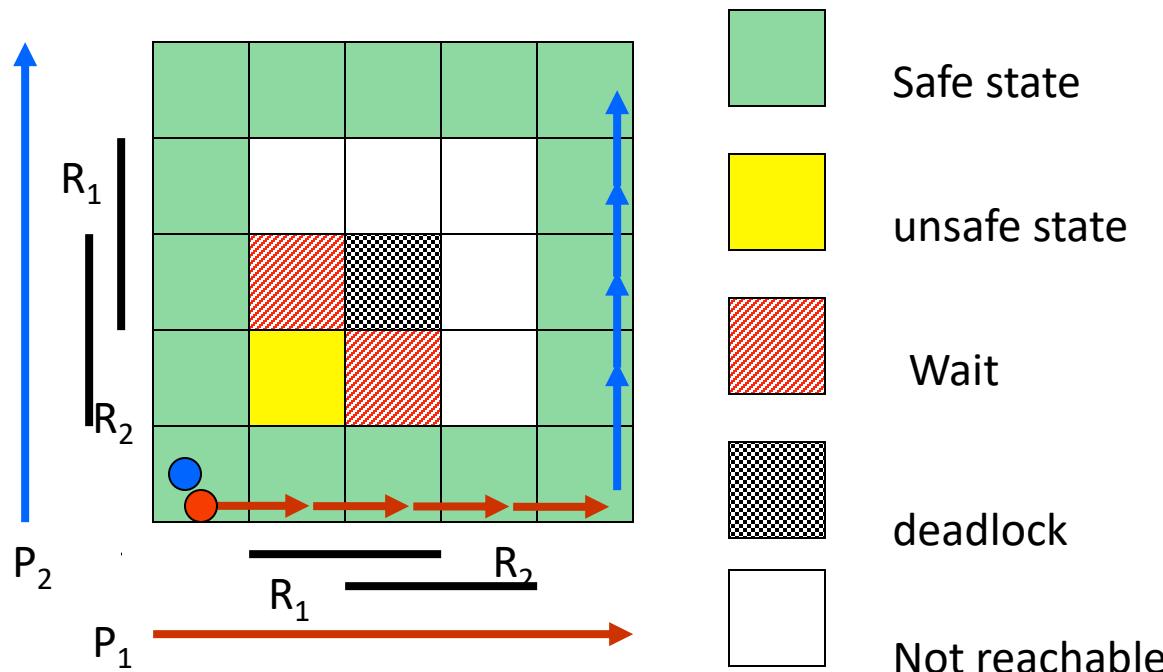


Definitions

- Safe state:
 - For any possible sequence of future resource requests, it is possible to eventually grant all requests*
 - * not necessarily in the same order they are requested...
 - And thus to make all the processes *end* correctly
 - May require waiting even when resources are available!
- Unsafe state:
 - Some sequence of resource requests can result in deadlock
- Doomed state:
 - All possible computations lead to deadlock

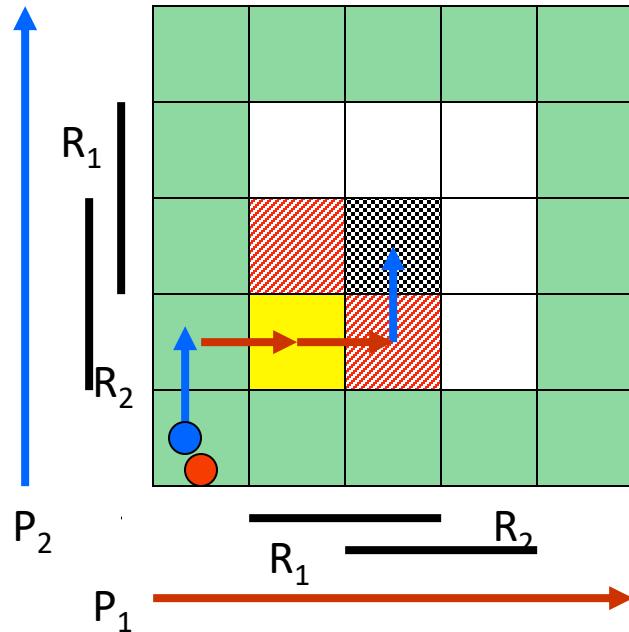
Safe state

1) System that evolves in safe states



Safe state

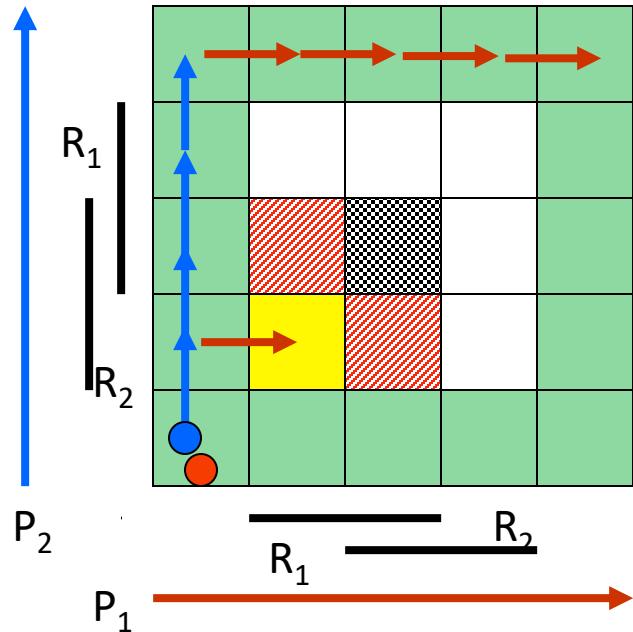
2) A system in an unsafe state can reach a deadlock



- Safe state
 - Unsafe state
 - Wait resource
 - Deadlock
 - Not reachable
- P2 requests R2
- P1 requests R1:
Unsafe state
- P1 requests R2:
wait
- P2 requests R1:
deadlock

With the banker

3) The banker does not accept requests that lead to unsafe states



	Safe state	P2 requests R2
	Unsafe state	P1 requests R1: Not assigned; P1 waits
	Wait resource	P2 releases R1: P2 releases R2
	Deadlock	P1 obtain R1: P1 continue and ends
	Not reachable	

Banker's Algorithm

- Grant request iff(*) result is a safe state
- Sum of maximum resource needs of current threads can be greater than the total resources
 - Provided there is some way for all the threads to end without getting into deadlock
- Example: proceed iff
 - # free resources \geq max remaining that might be needed by this thread in order to finish
 - Guarantees this thread can finish

(*) iff, “if and only if” abbreviation

Banker's Algorithm, resources of the same type

- First each process declares the number of resources it needs
- At a request of process P the banker checks whether the resource assignment keeps a safe state. To this purpose:
 - Considers the state S reached if request is granted
 - For each process computes the residual requirement R (the number of resources it still needs)
 - Sorts the processes for an increasing value of R
 - Executes the algorithm (see next slide)
 - If at all processes are marked at the end of algorithm the state is safe
- If the state S is safe then the request can be granted
- Otherwise the process P waits until there are enough resources to let it proceed

Banker's Algorithm, resources of the multiple types

D : availability vector

For each resource $R_k : D_k$ number of available units of R_k

For each process P_j :

- A_j : assignment vector;
- E_j : vector of residual requirements; ($E_j \leq D$ if $E_{jk} \leq D_k$ for each k)

Initially each process P_j is not marked

```
while ( $\exists$  non marked processes) {
    if ( $\exists$  a non-marked  $P_j$  that satisfies  $E_j \leq D$ ) {
        mark  $P_j$ ;
         $D = D + A_j$  ;
    } else ends while, the state is not safe;
}
```

success: the initial state is safe

Banker's Algorithm, Examples

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.1)

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	1	1	2
P3	0	1	0	2
P4	0	2	5	2

Stato (**sicuro**) raggiunto dal sistema al tempo t :

ASSEGNAZIONE
ATTUALE →

	MOLTEPLICITA' →			
	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILTA'
ATTUALE

	R1	R2	R3	R4
	0	1	1	0

	R1	R2	R3	R4
	4	5	5	5

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

ESIGENZA RESIDUA

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 1)

Stato raggiunto dal sistema al tempo t :

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
MOLTEPLICITA'	→			
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4
0	1	1	0

R1	R2	R3	R4
4	5	5	5

R1	R2	R3	R4
P1	0	2	0
P2	0	1	0
P3	0	0	0
P4	0	0	3

ESIGENZA RESIDUA

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
MOLTEPLICITA'	→			
P1	2	1	1	1
P2	-	-	-	-
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4
2	1	2	2

R1	R2	R3	R4
4	5	5	5

R1	R2	R3	R4
P1	0	2	0
P2	-	-	-
P3	0	0	0
P4	0	0	3

ESIGENZA RESIDUA

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 2)

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	-	-	-	-
P3	0	1	0	0
P4	0	2	2	2

MOLTEPLICITA' →

R1	R2	R3	R4
P1	4	5	5
P2	-	-	-
P3	0	0	0
P4	0	0	3

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4
P1	2	1	2
P2	-	-	-
P3	0	0	0
P4	0	0	3

R1	R2	R3	R4
P1	0	2	0
P2	-	-	-
P3	0	0	0
P4	0	0	3

ESIGENZA RESIDUA

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 2) l'esigenza di P3 può essere soddisfatta e P3 può terminare

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

MOLTEPLICITA' →

R1	R2	R3	R4
P1	4	5	5
P2	-	-	-
P3	-	-	-
P4	0	0	3

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4
P1	2	2	2
P2	-	-	-
P3	-	-	-
P4	0	0	3

R1	R2	R3	R4
P1	0	2	0
P2	-	-	-
P3	-	-	-
P4	0	0	3

ESIGENZA RESIDUA

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 3)

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 2) l'esigenza di P3 può essere soddisfatta e P3 può terminare

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →	R1 R2 R3 R4 4 5 5 5
P1 2 1 1 1 P2 - - - - P3 - - - - P4 0 2 2 2	DISPONIBILTA' ATTUALE	R1 R2 R3 R4 2 2 2 2
		R1 R2 R3 R4 0 2 0 0
		R1 R2 R3 R4 - - - -
		R1 R2 R3 R4 0 0 3 0
		ESIGENZA RESIDUA

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
- 3) l'esigenza di P1 può essere soddisfatta e P1 può terminare

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →	R1 R2 R3 R4 4 5 5 5
P1 - - - - P2 - - - - P3 - - - - P4 0 2 2 2	DISPONIBILTA' ATTUALE	R1 R2 R3 R4 4 3 3 3
		R1 R2 R3 R4 - - - -
		R1 R2 R3 R4 - - - -
		R1 R2 R3 R4 0 0 3 0
		ESIGENZA RESIDUA

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 4)

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
- 3) l'esigenza di P1 può essere soddisfatta e P1 può terminare

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →	R1 R2 R3 R4 4 5 5 5
DISPOSIBILITA' ATTUALE	R1 R2 R3 R4 4 3 3 3	R1 R2 R3 R4 4 - - -
	R1 R2 R3 R4 0 2 2 2	
		R1 R2 R3 R4 0 0 3 0
		ESIGENZA RESIDUA

Verifica dello stato sicuro:

- 1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
- 3) l'esigenza di P1 può essere soddisfatta e P1 può terminare
- 4) l'esigenza di P4 può essere soddisfatta e P4 può terminare

STATO SICURO !

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →	R1 R2 R3 R4 4 5 5 5
DISPOSIBILITA' ATTUALE	R1 R2 R3 R4 4 5 5 5	R1 R2 R3 R4 - - - -
	R1 R2 R3 R4 - - - -	
		R1 R2 R3 R4 - - - -
		ESIGENZA RESIDUA

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.2)

Stato (**sicuro**) raggiunto dal sistema al tempo t :

MOLTEPLICITA' →				
	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA' ATTUALE				
	R1	R2	R3	R4
	0	1	1	0

ESIGENZA RESIDUA				
	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

Il processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

MOLTEPLICITA' →				
	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA' ATTUALE				
	R1	R2	R3	R4
	0	0	1	0

ESIGENZA RESIDUA				
	R1	R2	R3	R4
P1	0	1	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.3)

Il processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

ASSEGNAZIONE ATTUALE →					MOLTEPLICITA' →	R1	R2	R3	R4
P1	2	2	1	1		4	5	5	5
P2	2	0	1	2					
P3	0	1	0	0					
P4	0	2	2	2					

DISPONIBILITA' ATTUALE					R1	R2	R3	R4	ESIGENZA RESIDUA
0	0	1	0						

Verifica dello stato sicuro:

- 1) l'esigenza di P1 non può essere soddisfatta
- 2) l'esigenza di P2 non può essere soddisfatta
- 3) l'esigenza di P3 non può essere soddisfatta
- 4) l'esigenza di P4 non può essere soddisfatta

LO STATO NON E' SICURO, LA RICHIESTA INIZIALE DI P1 NON PUO' ESSERE SODDISFATTA!

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.1)

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	0	1	3
P3	0	1	0	2
P4	0	2	3	2

Stato raggiunto dal sistema al tempo t :

		MOLTEPLICITA' →				ESIGENZA RESIDUA			
ASSEGNAZIONE ATTUALE →		R1	R2	R3	R4	R1	R2	R3	R4
P1	2	1	1	1		4	5	5	5
P2	2	0	1	2					
P3	0	1	0	0					
P4	0	2	2	2					
DISPONIBILITA' ATTUALE		R1	R2	R3	R4				
		0	1	1	0				

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.2)

Stato raggiunto dal sistema al tempo t :

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
MOLTEPLICITA'	→			
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4	
ESIGENZA RESIDUA				
0	1	1	0	

R1	R2	R3	R4
4	5	5	5

R1	R2	R3	R4	
ESIGENZA RESIDUA				
P1	0	2	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

Al tempo t processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

ASSEGNAZIONE
ATTUALE →

	R1	R2	R3	R4
MOLTEPLICITA'	→			
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILITA'
ATTUALE

R1	R2	R3	R4	
ESIGENZA RESIDUA				
0	0	1	0	

R1	R2	R3	R4
4	5	5	5

R1	R2	R3	R4	
ESIGENZA RESIDUA				
P1	0	1	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.3)

Al tempo t processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →																																				
	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>2</td> <td style="background-color: #ADD8E6;">2</td> <td>1</td> <td>1</td> </tr> <tr> <td>P2</td> <td>2</td> <td>0</td> <td>1</td> <td>2</td> </tr> <tr> <td>P3</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>P4</td> <td>0</td> <td>2</td> <td>2</td> <td>2</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	2	2	1	1	P2	2	0	1	2	P3	0	1	0	0	P4	0	2	2	2	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>4</td> <td>5</td> <td>5</td> <td>5</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	4	5	5	5
	R1	R2	R3	R4																																	
P1	2	2	1	1																																	
P2	2	0	1	2																																	
P3	0	1	0	0																																	
P4	0	2	2	2																																	
	R1	R2	R3	R4																																	
P1	4	5	5	5																																	
	DISPONIBILITA' ATTUALE	R1 R2 R3 R4																																			
		<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>0</td> <td style="background-color: #ADD8E6;">0</td> <td>1</td> <td>0</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	0	0	1	0																									
	R1	R2	R3	R4																																	
P1	0	0	1	0																																	
		ESIGENZA RESIDUA																																			
		<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>0</td> <td style="background-color: #ADD8E6;">1</td> <td>0</td> <td>0</td> </tr> <tr> <td>P2</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>P3</td> <td>0</td> <td>0</td> <td>0</td> <td>2</td> </tr> <tr> <td>P4</td> <td>0</td> <td>0</td> <td>1</td> <td>0</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	0	1	0	0	P2	0	0	0	1	P3	0	0	0	2	P4	0	0	1	0										
	R1	R2	R3	R4																																	
P1	0	1	0	0																																	
P2	0	0	0	1																																	
P3	0	0	0	2																																	
P4	0	0	1	0																																	

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare

ASSEGNAZIONE ATTUALE →	MOLTEPLICITA' →																																				
	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>2</td> <td>2</td> <td>1</td> <td>1</td> </tr> <tr> <td>P2</td> <td>2</td> <td>0</td> <td>1</td> <td>2</td> </tr> <tr> <td>P3</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>P4</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	2	2	1	1	P2	2	0	1	2	P3	0	1	0	0	P4	-	-	-	-	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>4</td> <td>5</td> <td>5</td> <td>5</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	4	5	5	5
	R1	R2	R3	R4																																	
P1	2	2	1	1																																	
P2	2	0	1	2																																	
P3	0	1	0	0																																	
P4	-	-	-	-																																	
	R1	R2	R3	R4																																	
P1	4	5	5	5																																	
	DISPONIBILITA' ATTUALE	R1 R2 R3 R4																																			
		<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>P2</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>P3</td> <td>0</td> <td>0</td> <td>0</td> <td>2</td> </tr> <tr> <td>P4</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	0	1	0	0	P2	0	0	0	1	P3	0	0	0	2	P4	-	-	-	-										
	R1	R2	R3	R4																																	
P1	0	1	0	0																																	
P2	0	0	0	1																																	
P3	0	0	0	2																																	
P4	-	-	-	-																																	
		ESIGENZA RESIDUA																																			
		<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th></th> <th>R1</th> <th>R2</th> <th>R3</th> <th>R4</th> </tr> </thead> <tbody> <tr> <td>P1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>P2</td> <td>0</td> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>P3</td> <td>0</td> <td>0</td> <td>0</td> <td>2</td> </tr> <tr> <td>P4</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> <td style="background-color: yellow;">-</td> </tr> </tbody> </table>		R1	R2	R3	R4	P1	0	1	0	0	P2	0	0	0	1	P3	0	0	0	2	P4	-	-	-	-										
	R1	R2	R3	R4																																	
P1	0	1	0	0																																	
P2	0	0	0	1																																	
P3	0	0	0	2																																	
P4	-	-	-	-																																	

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.4)

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare

	MOLTEPLICITA' →					
ASSEGNAZIONE ATTUALE →	P1	R1 2	R2 2	R3 1	R4 1	
	P2	R1 2	R2 0	R3 1	R4 2	
	P3	R1 0	R2 1	R3 0	R4 0	
	P4	R1 -	R2 -	R3 -	R4 -	
DISPONIBILITA' ATTUALE	R1 0	R2 2	R3 3	R4 2		
	R1 4	R2 5	R3 5	R4 5		
	R1 0	R2 1	R3 0	R4 0		
	R1 0	R2 0	R3 0	R4 1		
R1 0	R2 0	R3 0	R4 2			
R1 -	R2 -	R3 -	R4 -			
ESIGENZA RESIDUA						

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
- 2) l'esigenza di P1 può essere soddisfatta e P1 può terminare

	MOLTEPLICITA' →					
ASSEGNAZIONE ATTUALE →	P1	R1 -	R2 -	R3 -	R4 -	
	P2	R1 2	R2 0	R3 1	R4 2	
	P3	R1 0	R2 1	R3 0	R4 0	
	P4	R1 -	R2 -	R3 -	R4 -	
DISPONIBILITA' ATTUALE	R1 2	R2 4	R3 4	R4 3		
	R1 4	R2 5	R3 5	R4 5		
	R1 -	R2 -	R3 -	R4 -		
	R1 0	R2 0	R3 0	R4 1		
R1 0	R2 0	R3 0	R4 2			
R1 -	R2 -	R3 -	R4 -			
ESIGENZA RESIDUA						

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.5)

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
- 2) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

	R1	R2	R3	R4
P1	-	-	-	-
P2	2	0	1	2
P3	0	1	0	0
P4	-	-	-	-

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	-	-	-	-
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

DISPONIBILTA' ATTUALE

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
- 2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
- 3) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	1	0	0
P4	-	-	-	-

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

DISPONIBILTA' ATTUALE

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.6)

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
- 2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
- 3) l'esigenza di P2 può essere soddisfatta e P2 può terminare

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	1	0	0
P4	-	-	-	-

MOLTEPLICITA' →

	R1	R2	R3	R4
P1	4	5	5	5
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

DISPONIBILTA' ATTUALE

	R1	R2	R3	R4
P1	4	4	5	5
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

Verifica dello stato sicuro:

- 1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
- 2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
- 3) l'esigenza di P2 può essere soddisfatta e P2 può terminare
- 4) l'esigenza di P3 può essere soddisfatta e P3 può terminare

STATO SICURO ! LA RICHIESTA INIZIALE DI P1 PUO' ESSERE SODDISFATTA.

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

MOLTEPLICITA' →

	R1	R2	R3	R4
P1	4	5	5	5
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

DISPONIBILTA' ATTUALE

	R1	R2	R3	R4
P1	4	5	5	5
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

Scheduling

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run
 - Or multiple packets to send, or web requests to serve, or ...
- Definitions
 - response time, throughput, predictability
- Uniprocessor policies
 - FIFO, SJF, round robin, optimal
 - multilevel feedback as approximation of optimal
- Multiprocessor policies
 - Affinity scheduling, Gang scheduling

Definitions

- Task/Job
 - User request: e.g., mouse click, web request, shell command, ...
- Latency/response time
 - How long does a task take to complete?
- Throughput
 - How many tasks can be done per unit of time?
- Overhead
 - How much extra work is done by the scheduler?
- Fairness
 - How equal is the performance received by different users?
- Predictability
 - How consistent is the performance over time?

More Definitions

- Workload
 - Set of tasks for system to perform
- Preemptive scheduler
 - If we can take resources away from a running task
- Work-conserving
 - Resource is used whenever there is a task to run
- Scheduling algorithm
 - takes a workload as input
 - decides which tasks to do first
 - Performance metric (throughput, latency) as output
 - Only work-conserving schedulers to be considered

First In First Out (FIFO)

- Schedule tasks in the order they arrive
 - Continue running them until they complete or give up the processor
- On what workloads is FIFO particularly bad?

Shortest Job First (SJF)

Two forms:

- non pre-emptive (“senza pre-rilascio”):
 - take the task with the shortest remaining work to do and run it up to its end or until it releases the processor
- pre-emptive (“con pre-rilascio”):
 - run the task with the shortest remaining amount of work to do
 - If a new shorter task wakes up: context switch and run the new task
 - Often called Shortest Remaining Time First (SRTF)

Shortest Job First (SJF)

- Suppose we have five tasks that arrive one right after each other, but the first one is much longer than the others
 - Which completes first in FIFO? Next?
 - Which completes first in SJF? Next?

FIFO vs. SJF

Tasks

FIFO

(1)



(2)



(3)



(4)



(5)



SJF

(1)



(2)



(3)



(4)



(5)



Time

Shortest Job First

- Claim: SJF is optimal for average response time
 - Why?
- For what workloads is FIFO optimal?
- Pessimal?
- Does SJF have any downsides?

Shortest Job First

SJF minimizes the average response time (turnaround):

4 jobs A,B,C,D with execution time: a, b, c, d

Scheduling sequence: $a \rightarrow b \rightarrow c \rightarrow d$

- turnaround(A) -- a
- turnaround(B) -- $a + b$
- turnaround(C) -- $a + b + c$
- turnaround(D) -- $a + b + c + d$

Average turnaround: $(4a + 3b + 2c + d) / 4$

Minimized iff a, b, c, d are sorted in increasing order

iff = “if and only if”

Shortest Job First

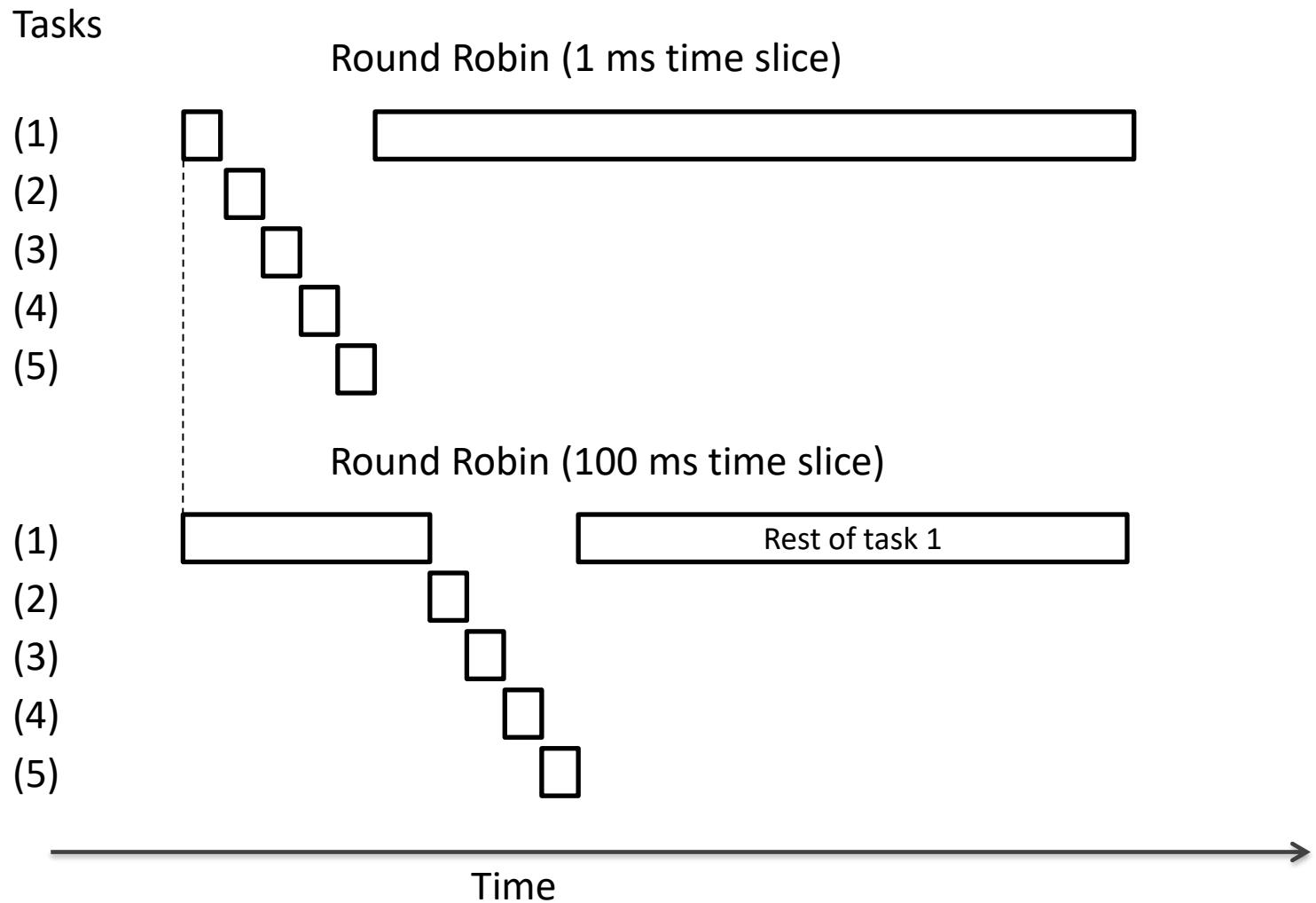
Downsides:

- Starvation ... some task might take forever?
- Variance in response time

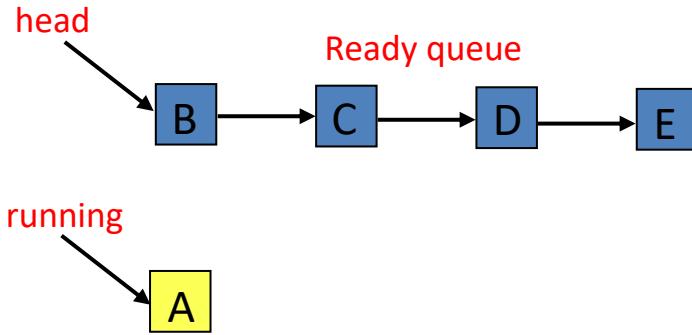
Round Robin

- Each task gets resource for a fixed period of time (time quantum)
 - If task doesn't complete, it goes back in line
- Need to pick a time quantum
 - What if time quantum is too long?
 - Infinite? Then, it falls into FIFO....
 - What if time quantum is too short?
 - One instruction? Too much overhead....

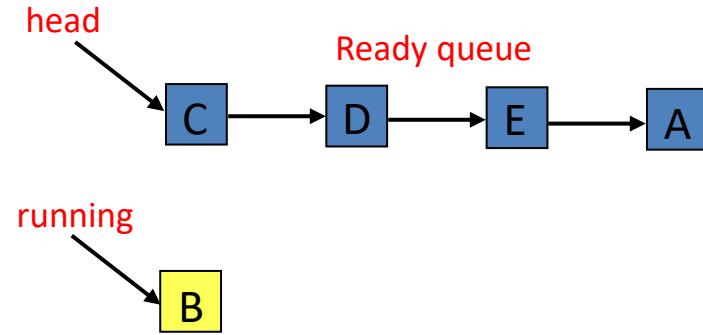
Round Robin



Round Robin



(a) A in execution



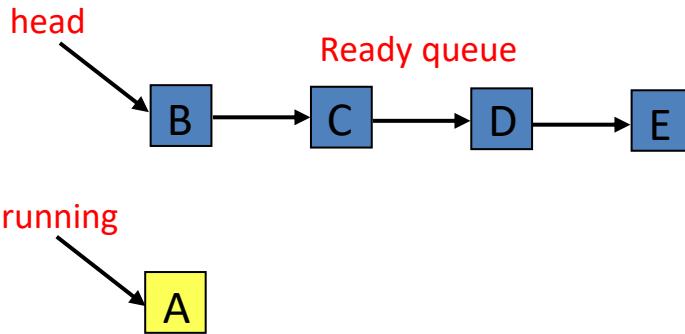
(b) A completes its time share

- Proportionality: turnaround (time spent in the system) proportional to task length
- Response time upper-bounded by the number of processes
 - # ready processes * time share

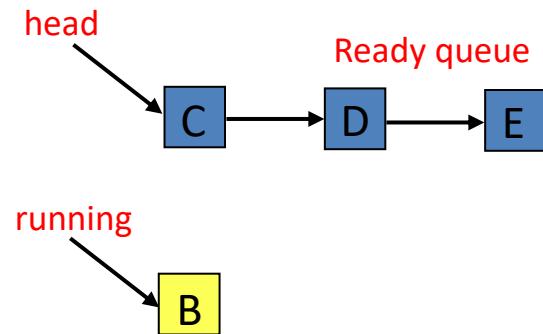
Round Robin

Management of waiting threads

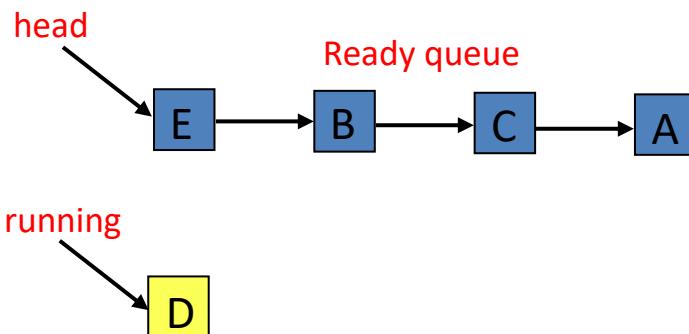
(a) A in execution



(b) A waiting



(c) A resumed



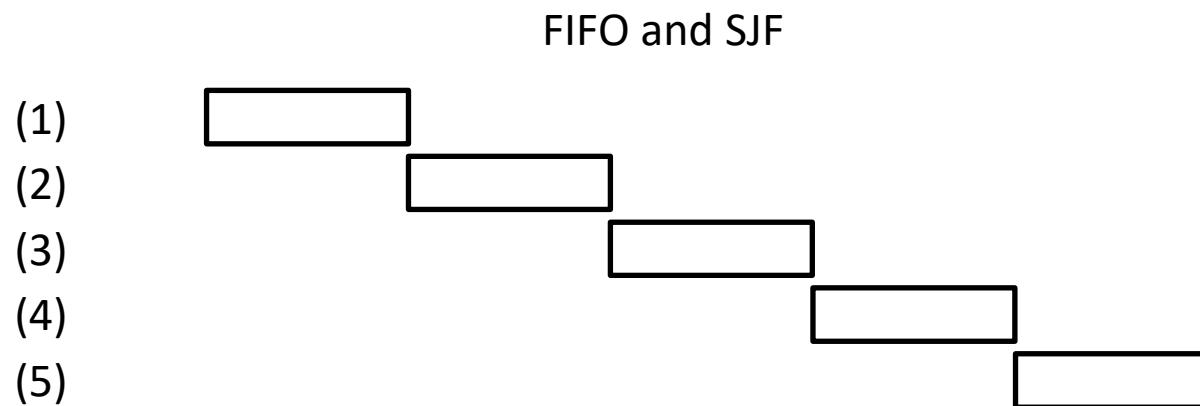
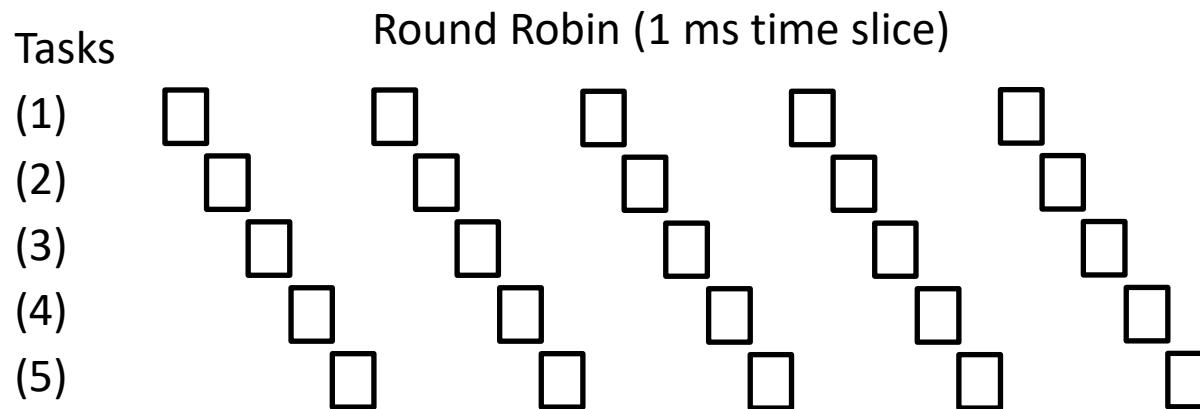
Round Robin

- End of time share signaled by the timer
 - Timer interrupt causes the activation of the scheduler
 - The scheduler restarts the timer
- Scheduler takes over also in case of suspension of the running process
 - Reassigns the CPU and restarts timer
- In current systems time share around 20-120 msec

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?

Round Robin vs FIFO



Time

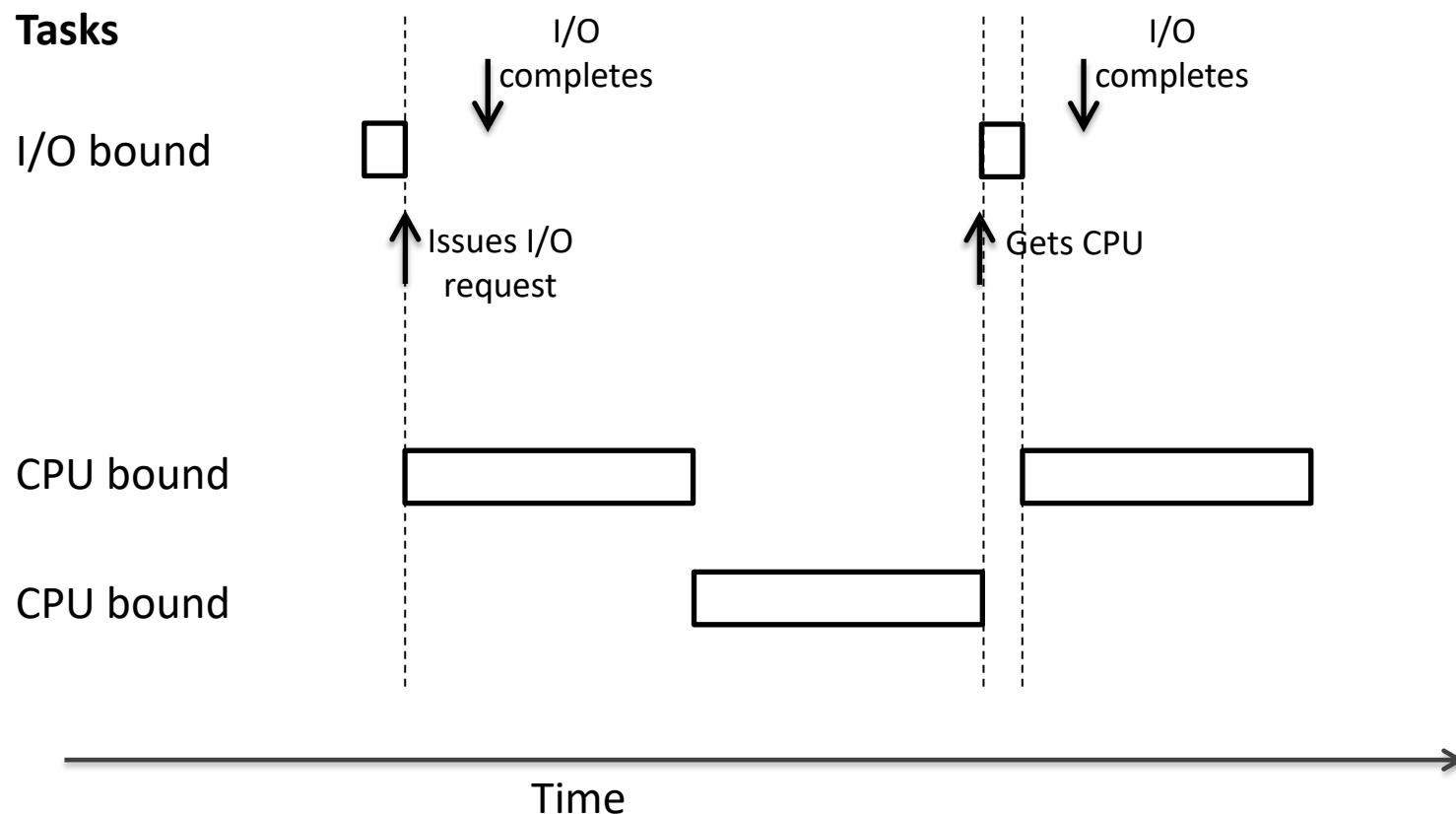
Round Robin = Fairness?

- Is Round Robin always fair?

Mixed Workload

- Let's consider two kind of processes:
 - CPU-bound – long CPU bursts with sparse I/O
 - E.g., number crunching computation
 - I/O bound – short CPU bursts with frequent I/O
 - E.g., highly interactive computation

Mixed Workload



Max-Min Fairness

- How do we balance a mixture of repeating tasks:
 - Some I/O bound, need only a little CPU
 - Some compute bound, can use as much CPU as they are assigned
- Approach: *maximize the minimum allocation given to a task*
 - Schedule the smallest task first
 - Split the remaining time using max-min
 - If all remaining tasks need at least equal share, split evenly

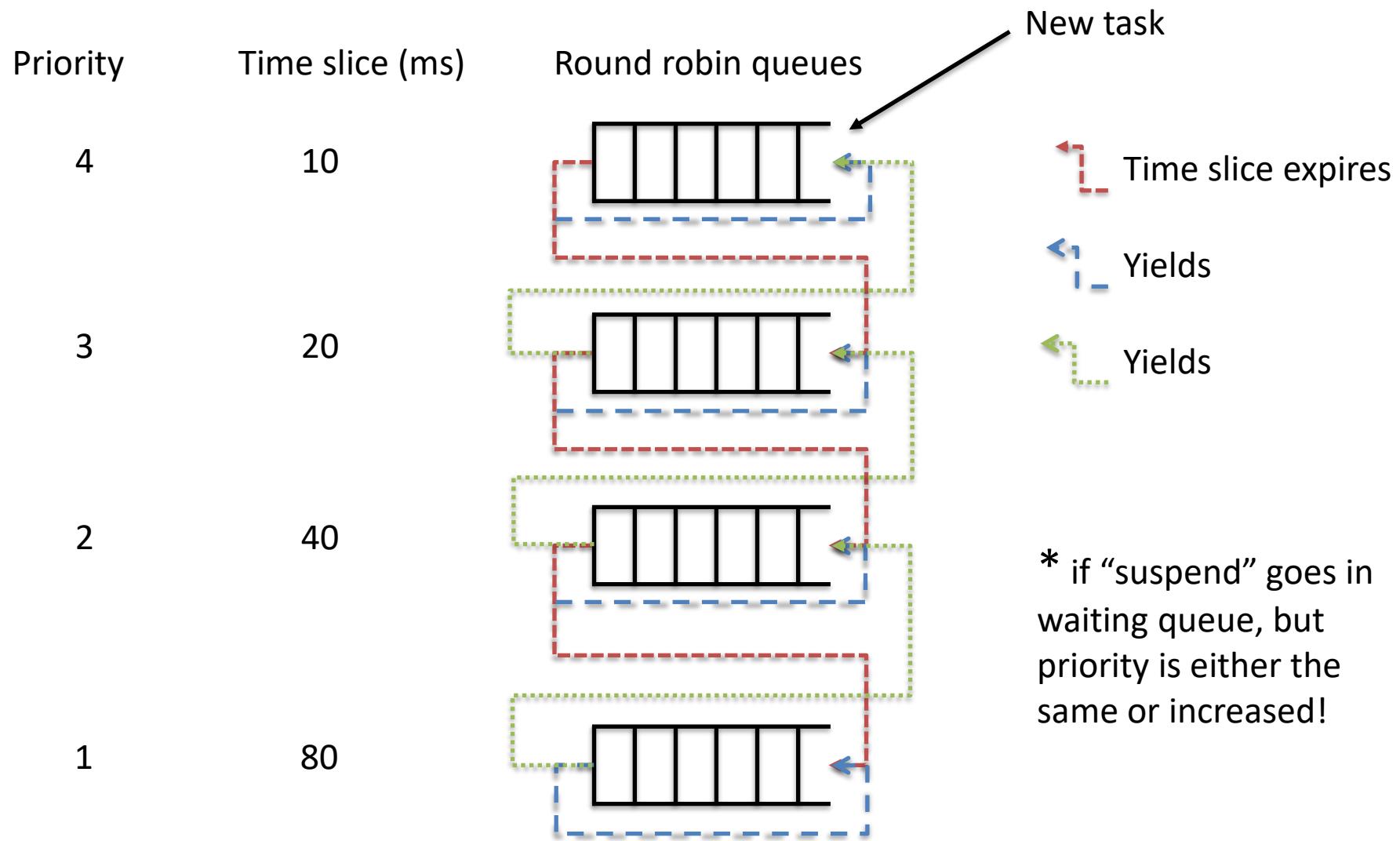
Multi-level Feedback Queue (MFQ)

- Goals:
 - Responsiveness
 - Low overhead
 - Starvation freedom
 - Some tasks are high/low priority
 - Fairness (among equal priority tasks)
- Not perfect at any of them!
 - Used in Linux, Windows, MacOS

MFQ

- Set of Round Robin queues
 - Each queue has a different priority
- High priority queues have short time slices
 - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
 - Round robin in each queue

MFQ



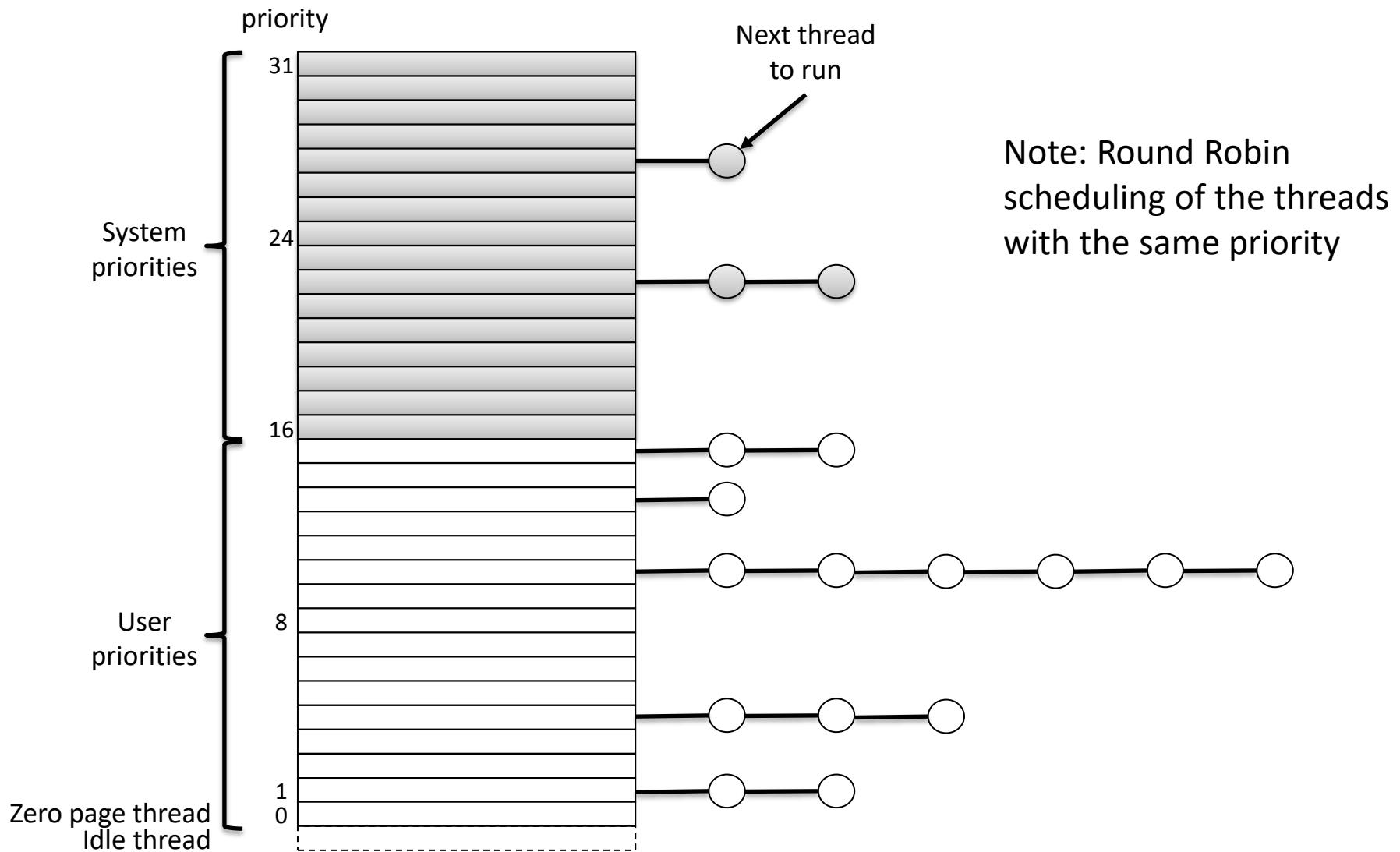
MFQ

- Tasks start in highest priority queue
- If time slice expires, task drops one level
 - i.e. the task has used entirely its time share
 - Task priority is reduced
- If task suspends or yields the processor, it remains in its level (or is bumped up)
 - Task priority is increased
- dynamic priority : tasks may change “habits”: from CPU-bound to I/O-bound and vice-versa

MFQ

- Starvation with MFQ
 - If the upper queues are always full of I/O-bound processes
- Need for policies to raise up the priority of CPU-bound processes that are starving

Example: scheduling in Windows



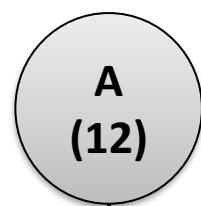
Example: scheduling in Windows

- A new thread starts with priority 8
- Priority raised up if :
 - thread reactivated after I/O operation (disk : +1, Serial line: +6, Keyboard: +8, Audio card: +8, ...)
 - thread reactivated after waiting on a mutex/semaphore (+1 if in background, +2 if in foreground)
 - Thread didn't run for a given amount of time (priority goes to 15 for two time shares)
 - against inversion of priority (see next slide)
- Priority lowered if thread uses all time share (-1)
- When a window goes in foreground the time share of its threads is enlarged

Example: scheduling in Windows

Inversion of priority

1)



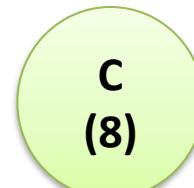
A executes
P(sem) and
waits



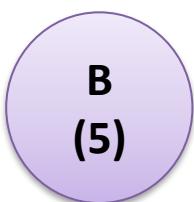
2)

A waiting on sem

running



ready;
Should execute V(sem)
But does not have the CPU



Uniprocessor Summary (1)

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Uniprocessor Summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O do poorly under Round Robin.
- Max-min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

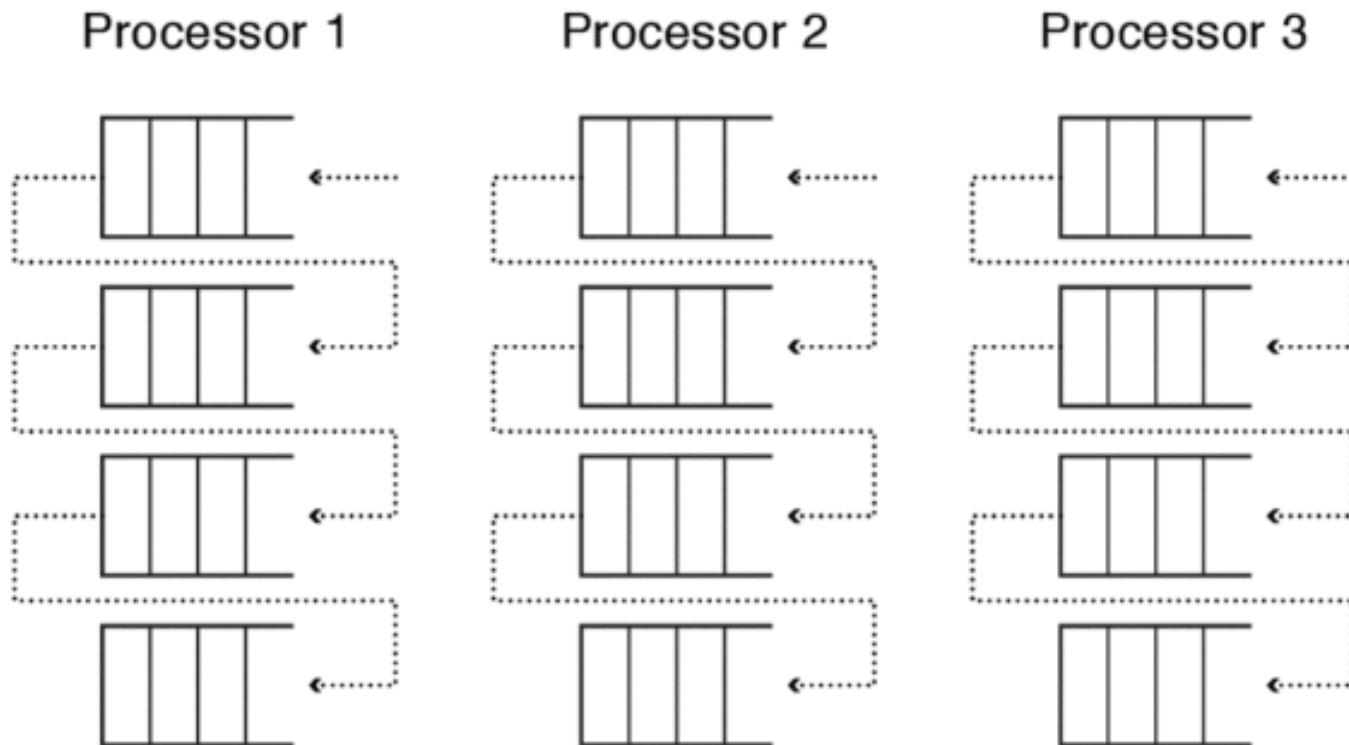
Multiprocessor

- Today, most general-purpose computers are multiprocessors
 - In 2014 we entered in the so-called “multi-core era”
- Many high-end servers have multiple CPUs, each with several cores
 - Today, a single server has $O(100)$ cores
 - Example: IBM Power10 CPU has 15 cores SMT8 (=120 logical cores for the OS). A server can be equipped with several CPUs.
- **Main point for OS:**
 - **How do we make effective use of multiple cores for running both sequential tasks as well as highly parallel applications?**

Multiprocessor Scheduling

- What would happen if we used MFQ on a multiprocessor?
 - Contention for scheduler spinlock
 - Cache slowdown due to ready list data structure pinging from one CPU to another
 - Limited cache reuse: thread's data from last time it ran is often still in its old cache

Per-Processor Multi-level Feedback with Affinity Scheduling



Per-Processor Affinity Scheduling

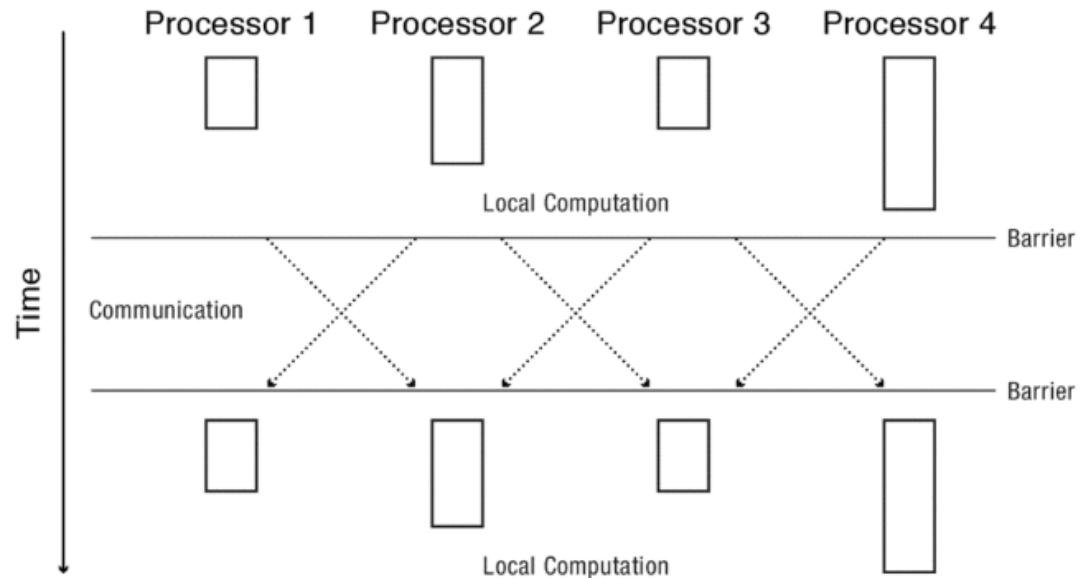
- Each processor has its own (multi-level) ready list
 - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run
 - Ex: when I/O completes, or on CV signal
- Idle processors can steal work from other processors
 - How many jobs is convenient to steal? From whom?
 - When is it convenient to rebalance the workload?

Scheduling Parallel Programs

- What happens if one thread gets time-sliced while other threads from the same program are still running?
 - Assuming program uses locks and condition variables, it will still be correct
 - What about performance?

Some Parallel Patterns

Bulk Synchronous Parallel (BSP)



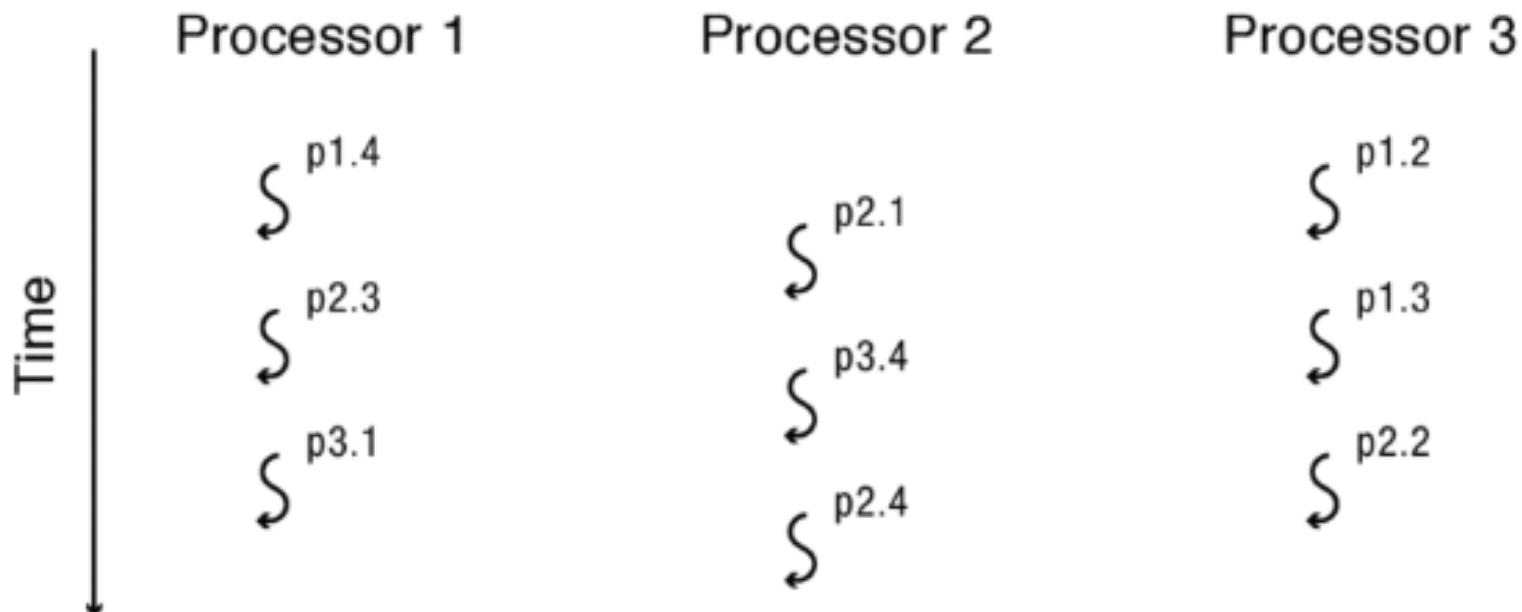
Producer-Consumer pipeline



Preempting one processor can stall all processors!

Scheduling Parallel Programs

Oblivious Scheduling: each processor time-slices its ready list independently of the other processors



$px.y = \text{Thread } y \text{ in process } x$

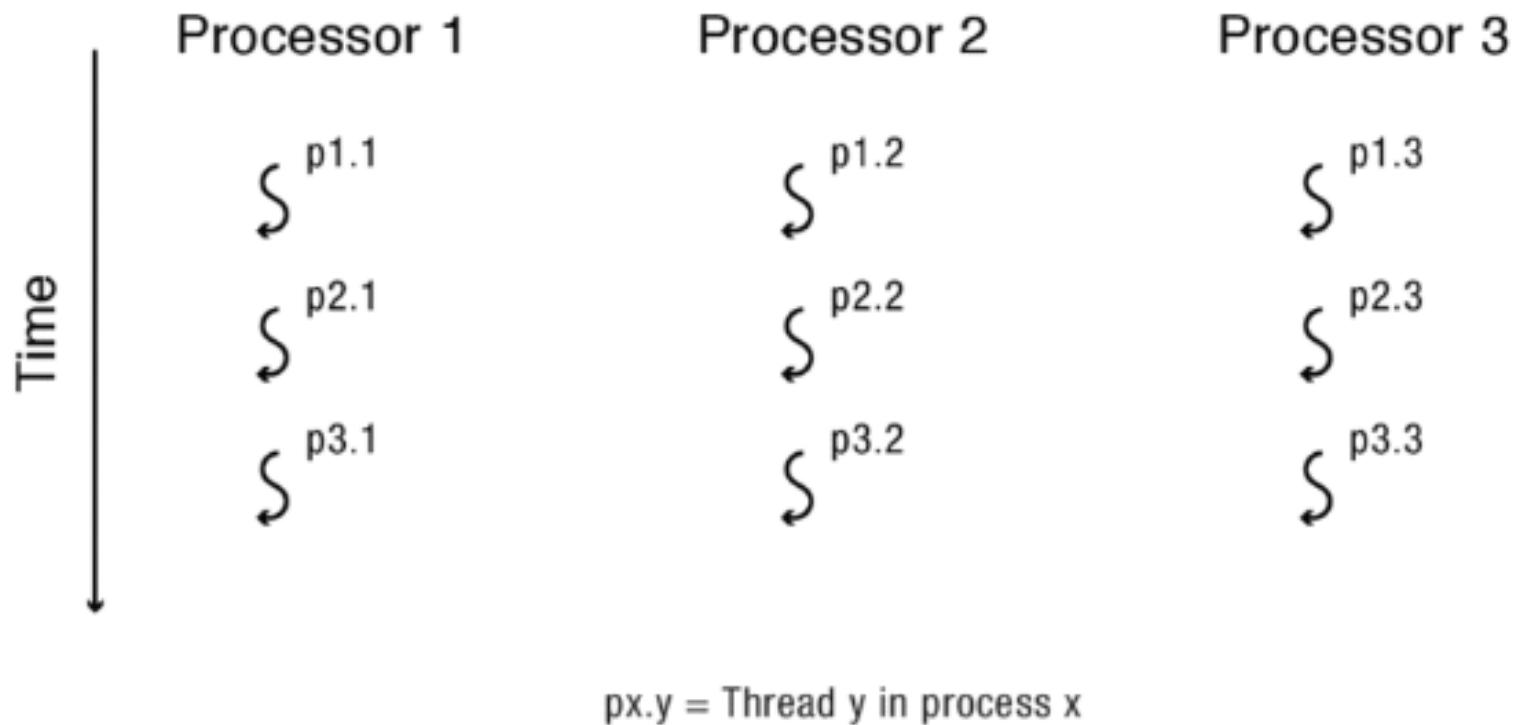
Oblivious Scheduling

Oblivious scheduling potential issues:

- Delay in BSP-like computations
- Delay in Producer-Consumer-like computations
- Critical path delay
- Preemption of lock holder
- I/O delay of some threads

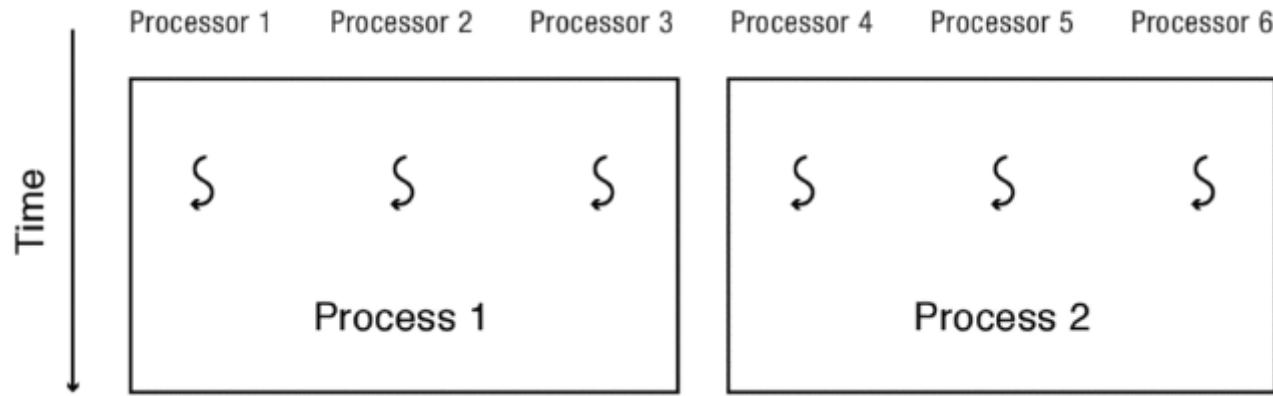
Gang Scheduling

- One solution is to schedule all of the tasks of a program together



Space Sharing

- However, gang scheduling can be inefficient for multiplexing multiple parallel applications



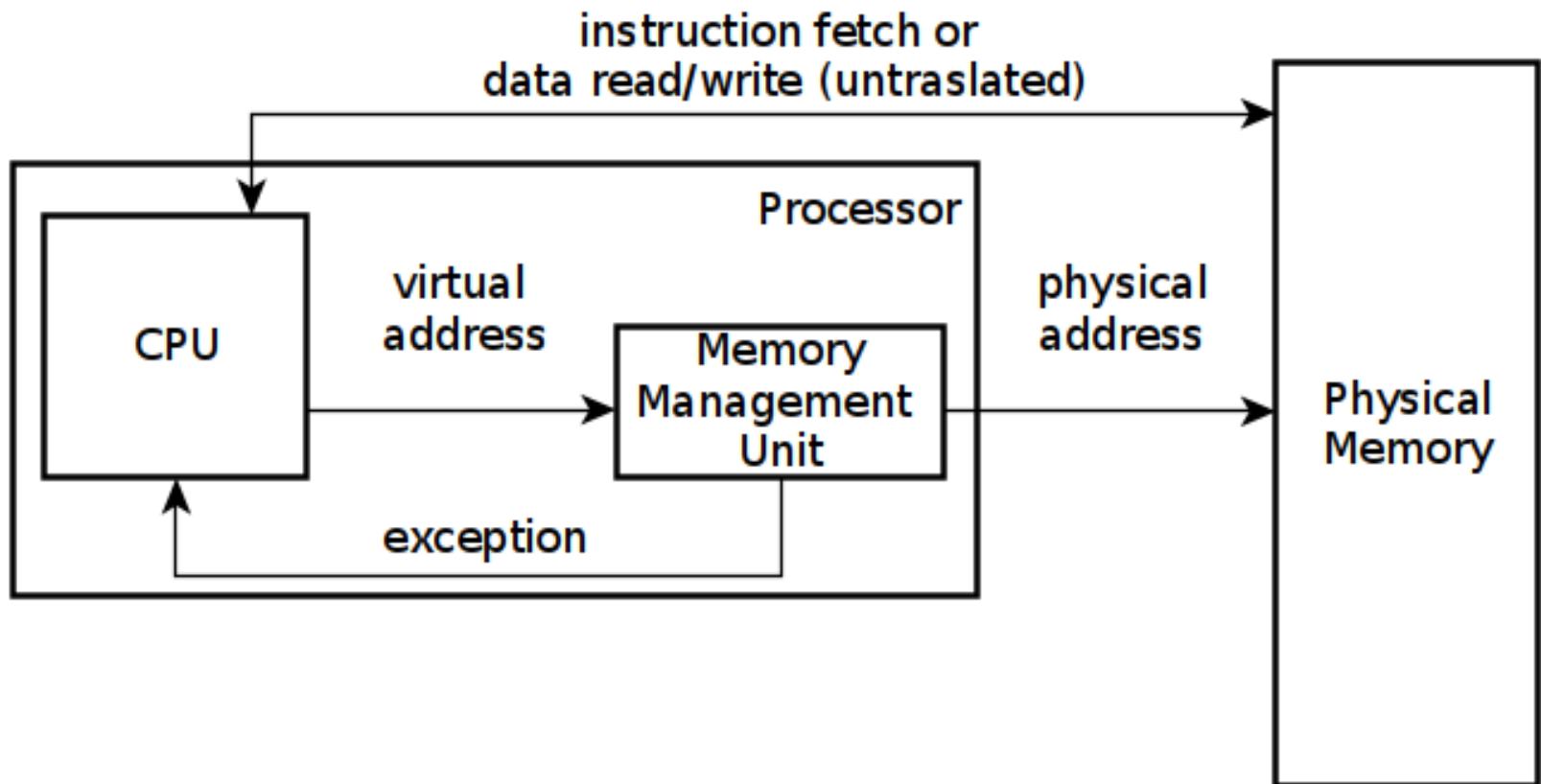
- Allocating different processors to different tasks

Address Translation

Main Points

- Address Translation Concept
 - How do we convert a virtual address to a physical address?
- Flexible Address Translation
 - Base and bound
 - Segmentation
 - Paging
 - Multilevel translation
- Efficient Address Translation
 - Translation Lookaside Buffers (TLB)
 - Virtually and Physically Addressed Caches

Address Translation Concept



Address Translation Goals

- Memory protection
- Memory sharing
- Flexible memory placement
- Sparse addresses
- Runtime lookup efficiency
- Compact translation tables
- Portability

Address Translation

- What can you do if you can (selectively) gain control whenever a program reads or writes a particular memory location?
 - With hardware support
 - With compiler-level support
- Memory management is one of the most complex parts of the OS
 - Serves many different purposes
 - ... implements a virtual memory

Address Translation Uses

- Process isolation
 - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
 - Shared regions of memory between processes
- Shared code segments
 - E.g., common libraries used by many different programs
- Program initialization
 - Start running a program before it is entirely in memory
- Dynamic memory allocation
 - Allocate and initialize stack/heap pages on demand

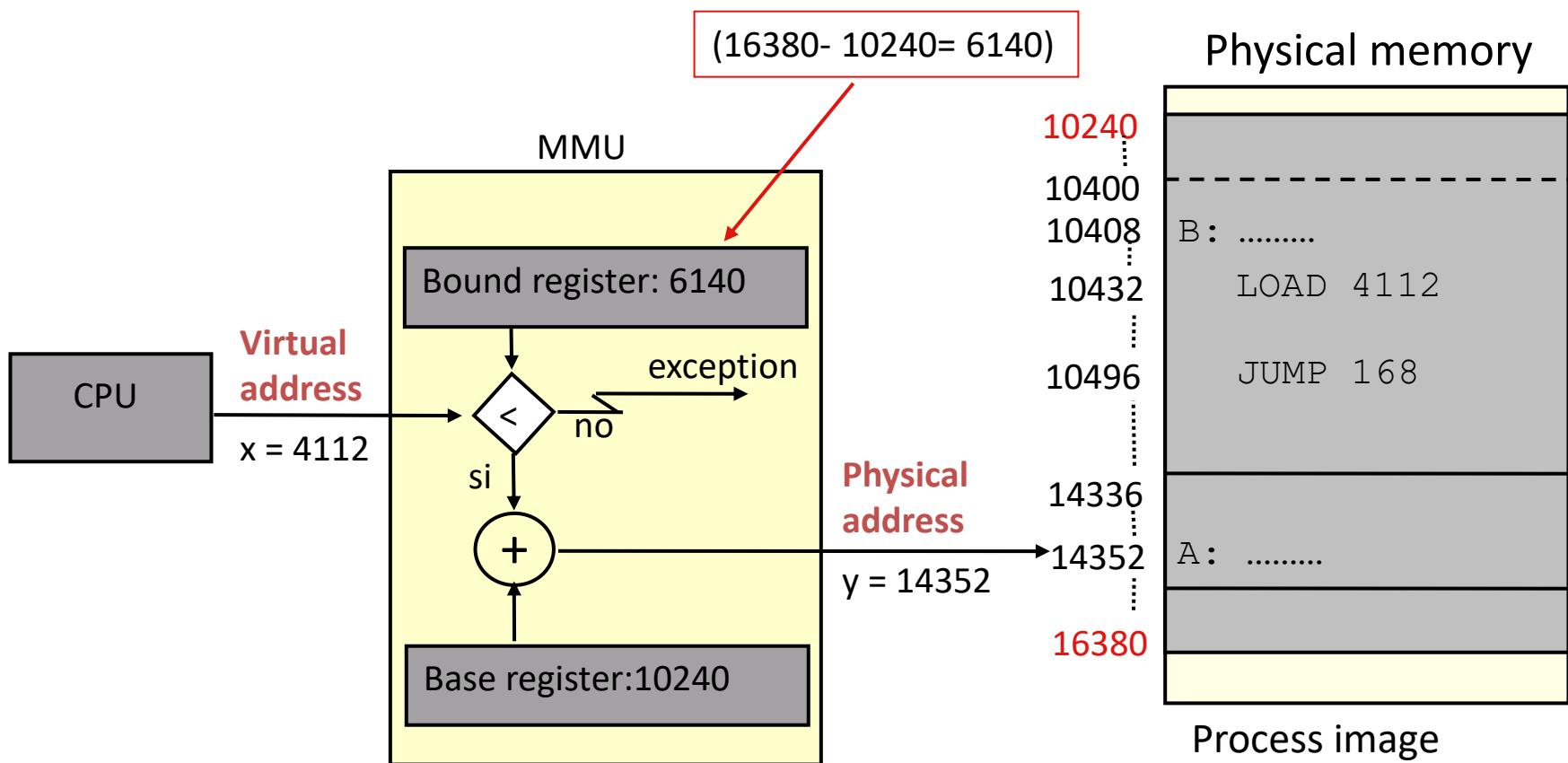
Address Translation (more)

- Cache management
 - Page coloring
- Program debugging
 - Data breakpoints when address is accessed
- Zero-copy I/O
 - Directly from I/O device into/out of user memory
- Memory mapped files
 - Access file data using load/store instructions
- Demand-paged virtual memory
 - Illusion of near-infinite memory, backed by disk or memory on other machines

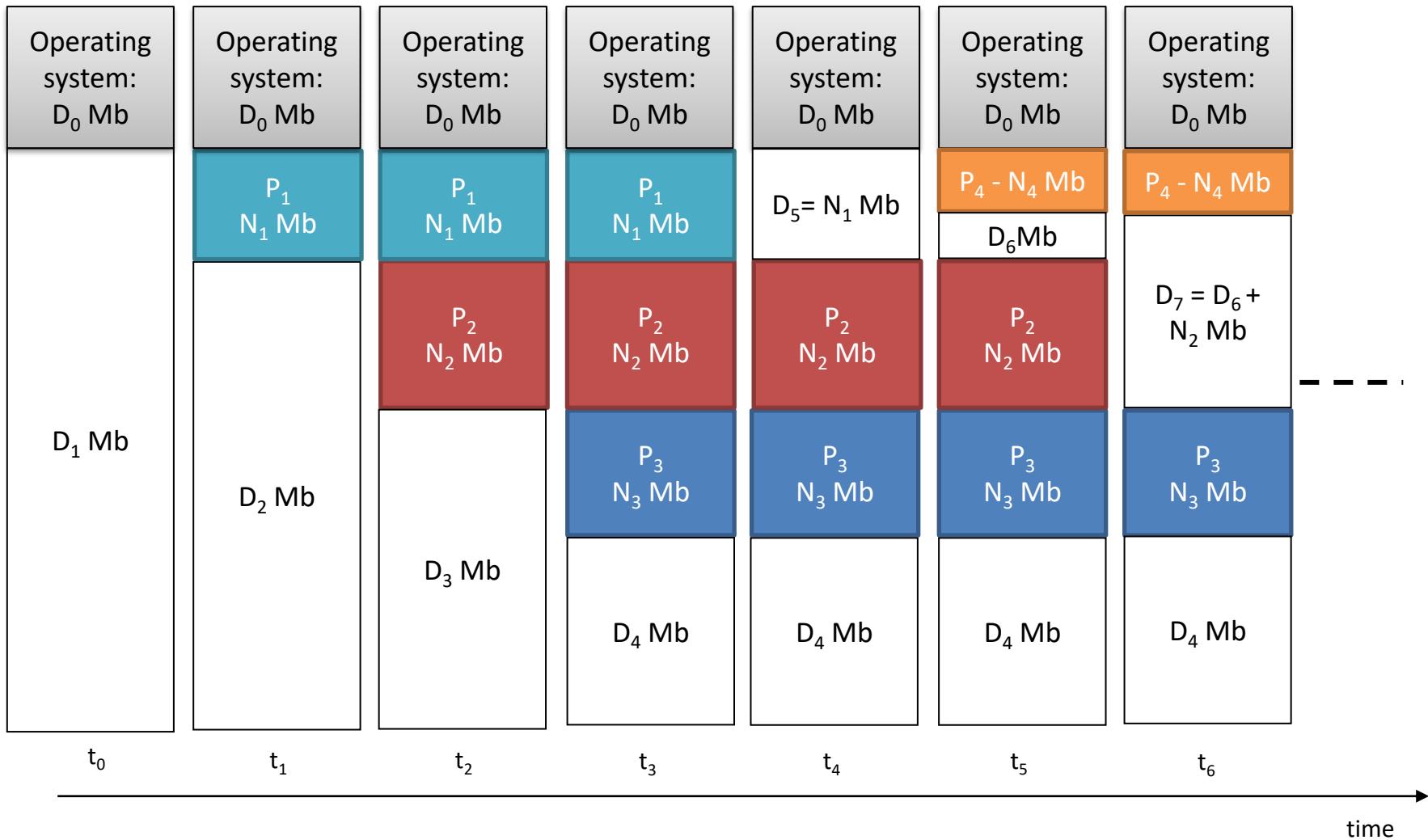
Address Translation (even more)

- Checkpointing/restart
 - Transparently save a copy of a process, without stopping the program while the save happens
- Persistent data structures
 - Implement data structures that can survive system reboots
- Process migration
 - Transparently move processes between machines
- Information flow control
 - Track what data is being shared externally
- Distributed shared memory
 - Illusion of memory that is shared between machines

Virtual Base and Bounds



Variable partitions



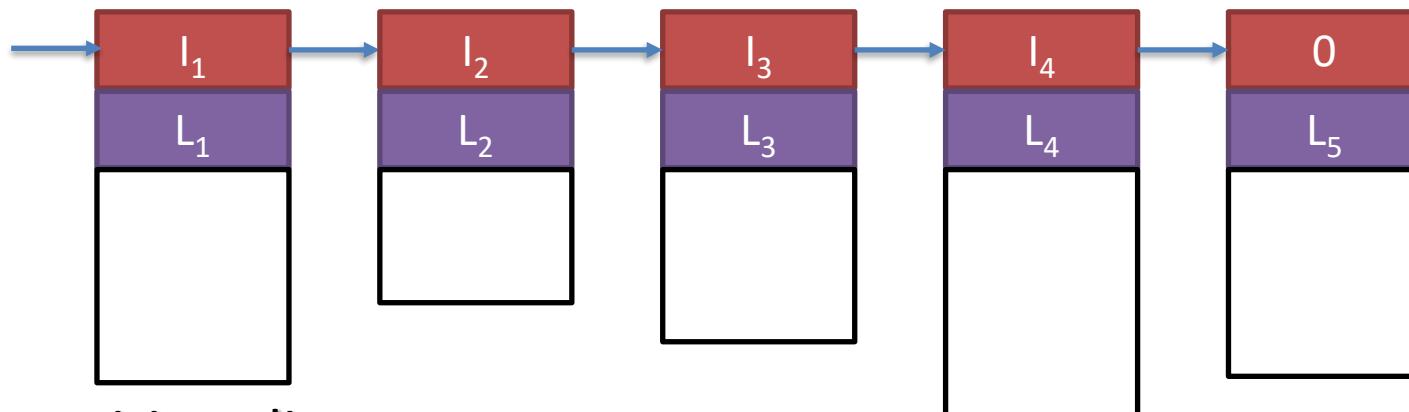
Allocation of a new partition

- **First-fit**

among all free partitions take the first one that is large enough to allocate the new partition

- **Best-fit**

among all free partitions take the smallest one that is large enough to allocate the new partition

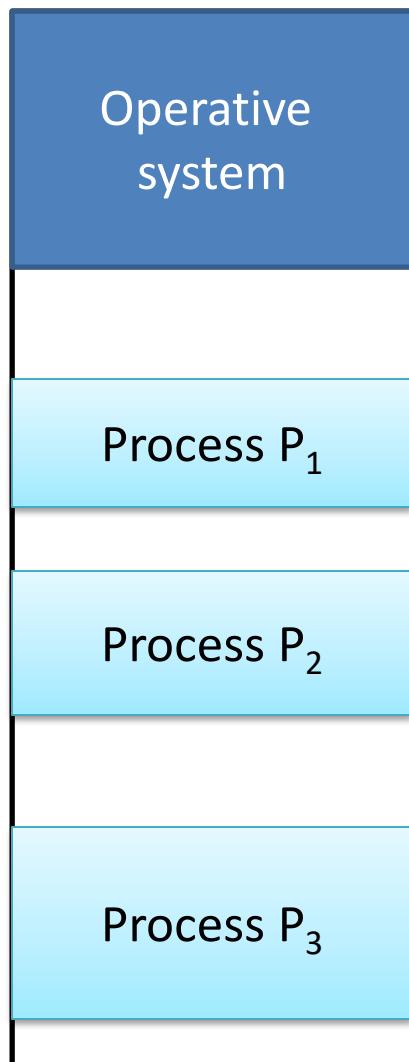


Free partitions list

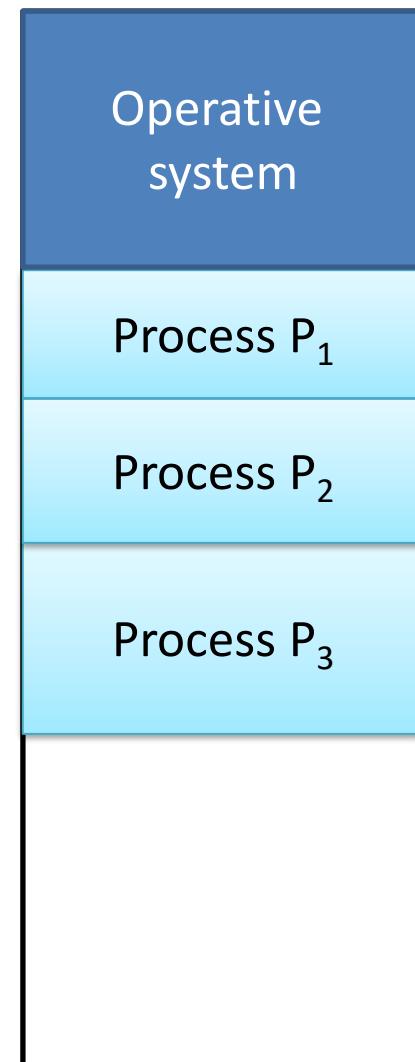
Fragmentation

- **Internal fragmentation**
 - memory allocated to a partition but not used/not necessary to the process.
 - Happens if fixed partitions are used.
- **External fragmentation**
 - the free memory partitions are too small to be used for other memory allocations.
 - Even if the overall free memory might be sufficient...
 - Occurs with variable partitions.

Memory de-fragmentation



Fragmented memory



De-fragmented memory

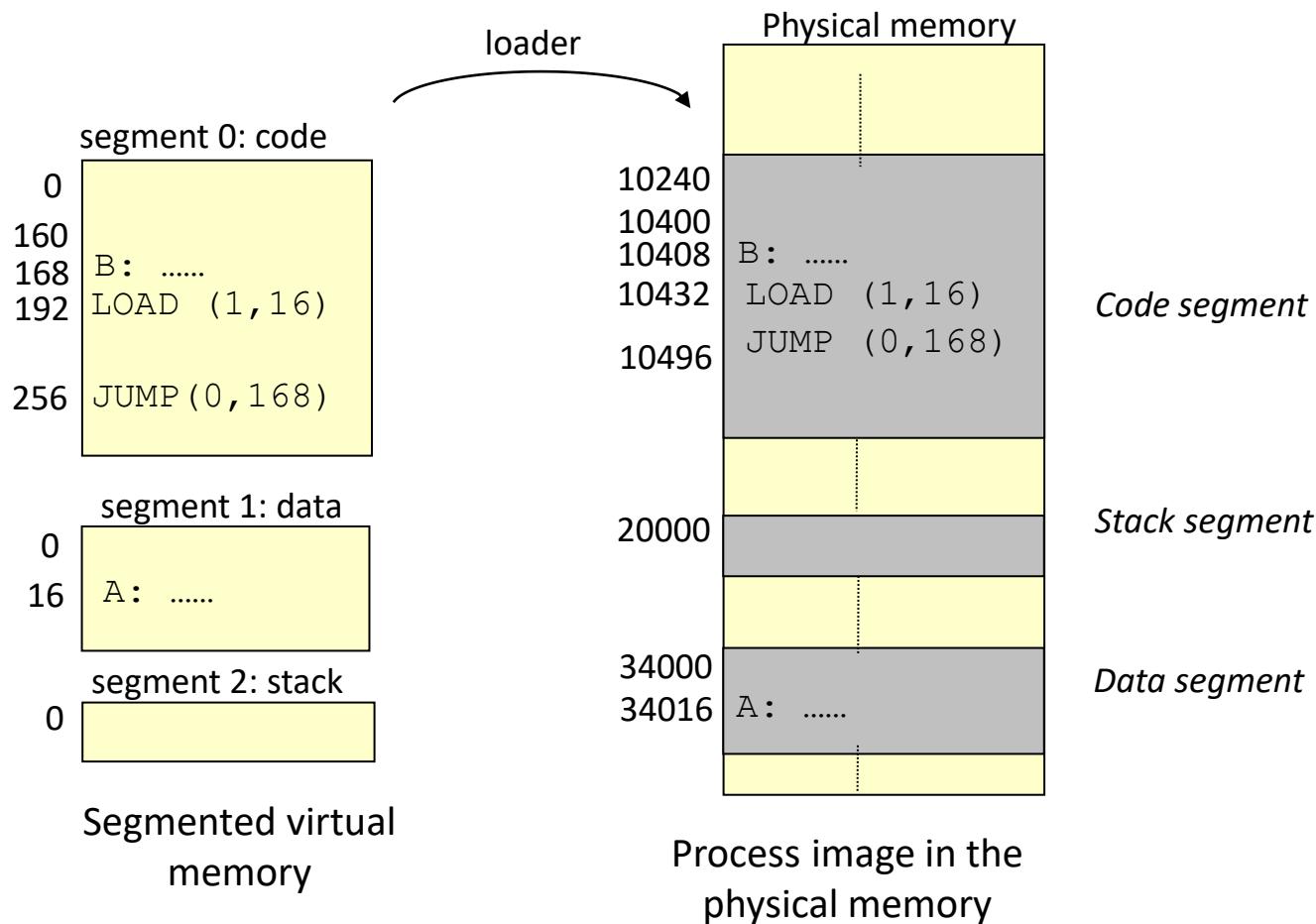
Virtual Base and Bounds

- Pros?
 - Simple
 - Fast (2 registers, adder, comparator)
 - Can relocate in physical memory without changing process
- Cons?
 - Can't keep program from accidentally overwriting its own code
 - Can't share code/data with other processes
 - Can't grow stack/heap as needed

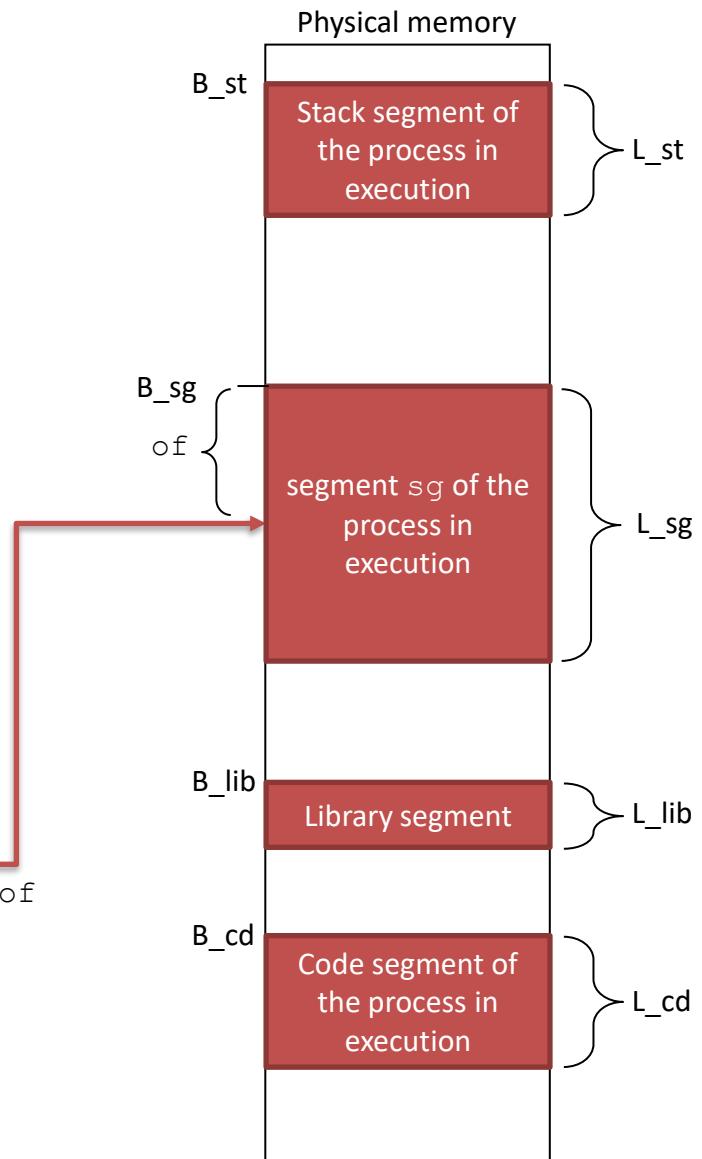
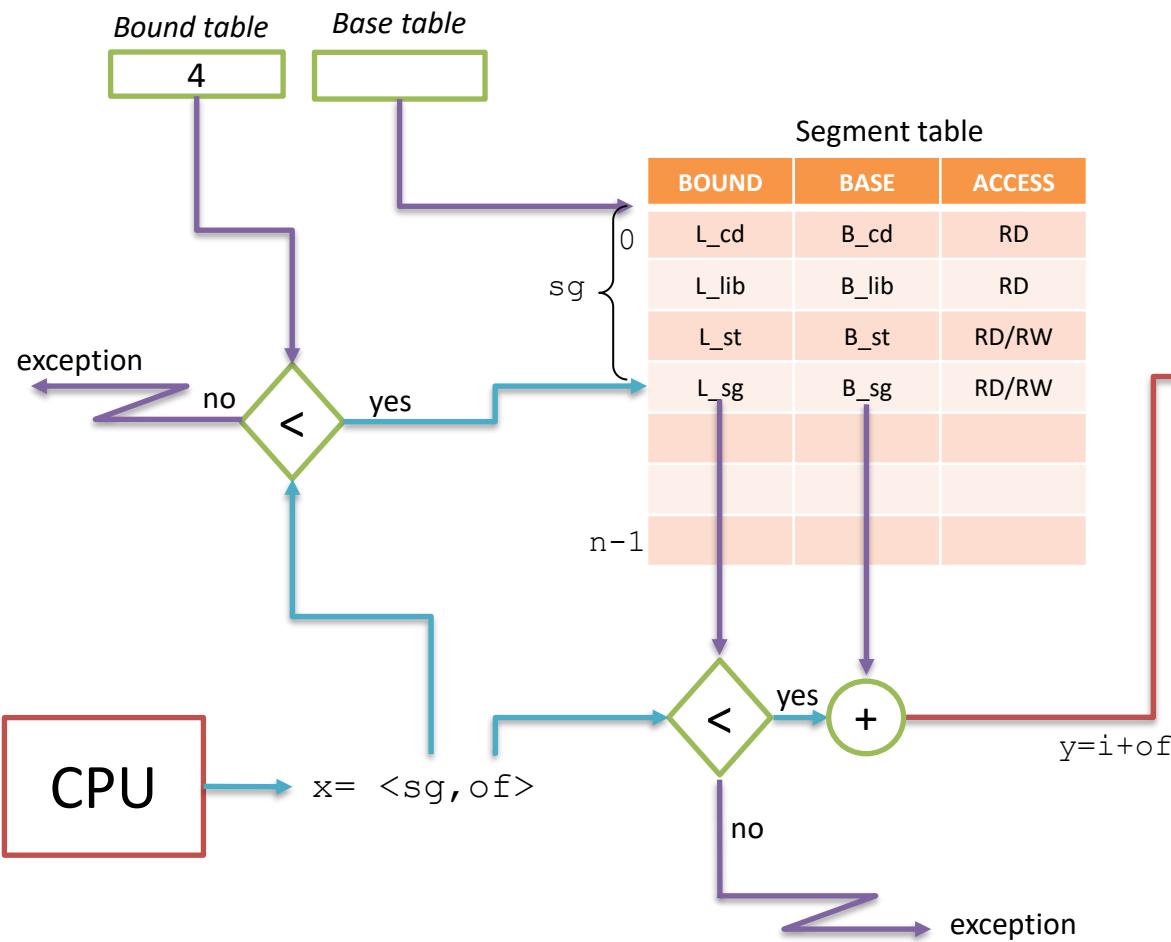
Segmentation

- Segment is a contiguous region of memory
 - Virtual or (for now) physical memory
- Each process has a segment table (in hardware)
 - Segment table = array of segment entries
 - Entry in table = segment (i.e., a base and bound pair)
- Segment can be located anywhere in physical memory
 - Start
 - Length
 - Access permission
- Processes can share segments
 - Same start, length, same/different access permissions

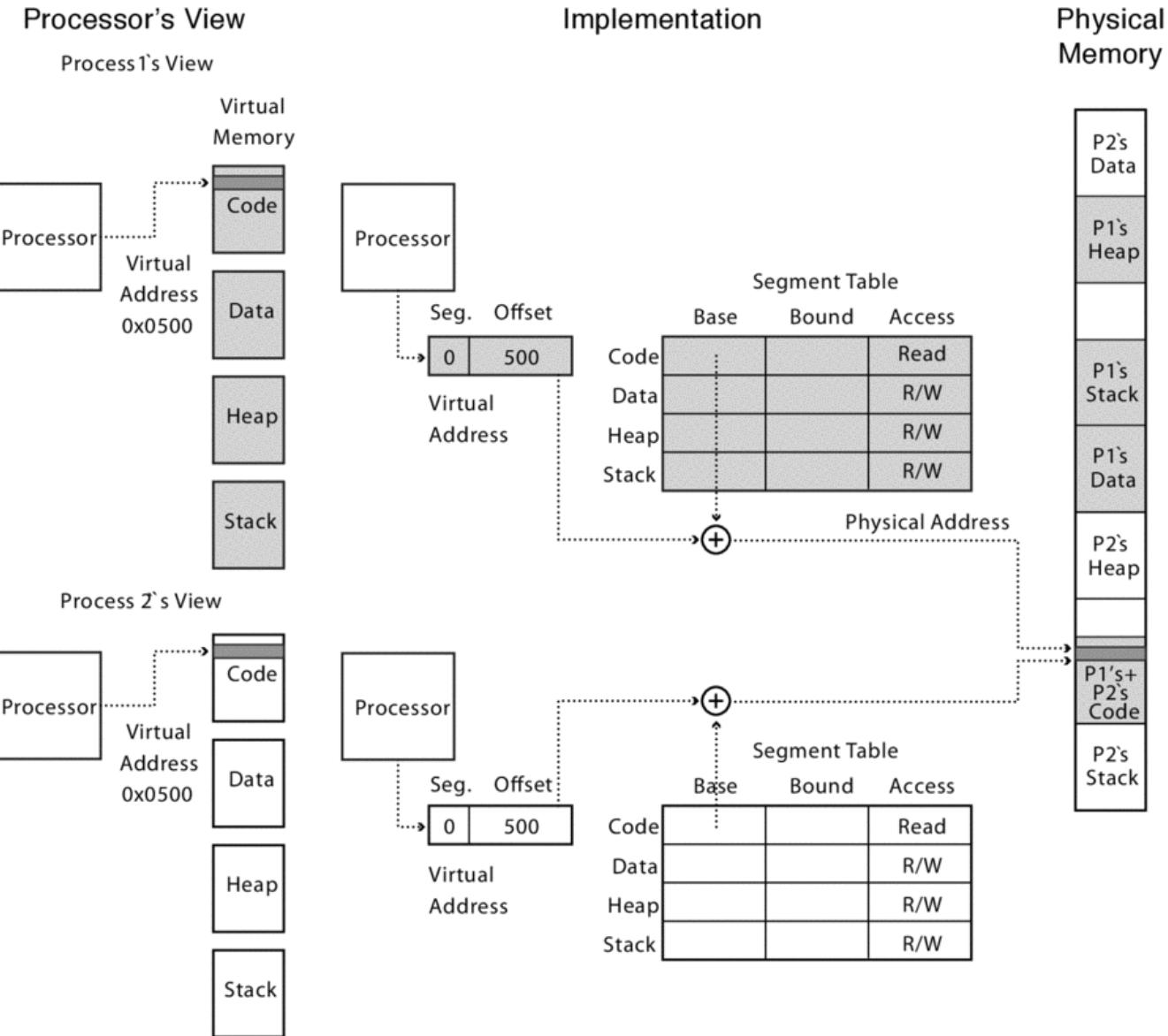
Segmentation



Segmentation: address translation



Segments sharing



UNIX fork and Copy on Write

- UNIX fork
 - Makes a complete copy of a process
- Segments allow a more efficient implementation
 - Copy segment table into child
 - Mark parent and child segments read-only
 - Start child process; return to parent
 - If child or parent writes to a segment, will trap into kernel
 - make a copy of the segment and resume

Zero-on-Reference

- How much physical memory do we need to allocate for the stack or heap?
 - Zero bytes!
- When program touches the heap
 - Segmentation fault into OS kernel
 - Kernel allocates some memory
 - How much?
 - Zeros the memory
 - avoid accidentally leaking information!
 - Restart process

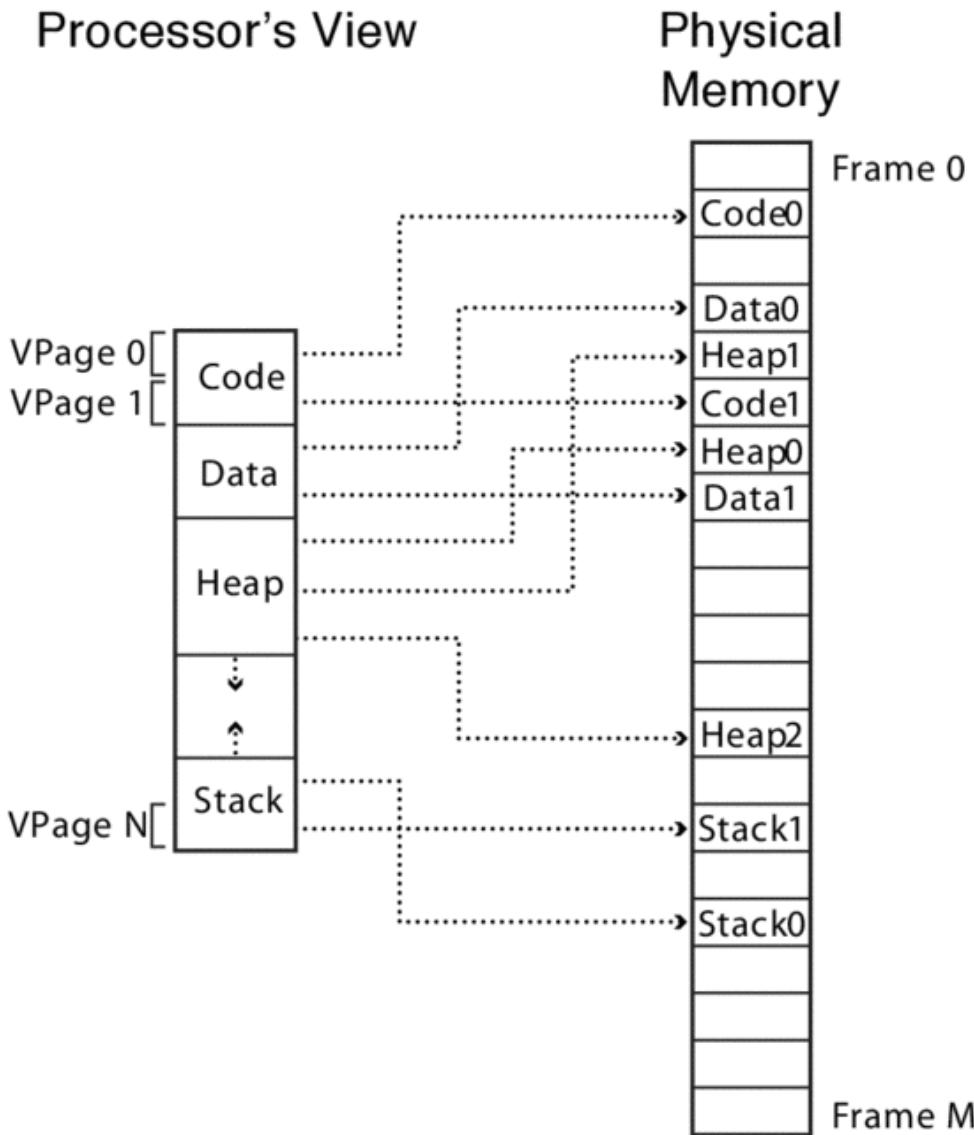
Segmentation

- Pros?
 - Can share code/data segments between processes
 - Can protect code segment from being overwritten
 - Can transparently grow stack/heap as needed
 - Can detect if need to copy-on-write
- Cons?
 - Complex memory management (mainly allocation)
 - Need to find chunk of a particular size
 - May need to rearrange memory from time to time to make room for new segment or growing segment
 - External fragmentation: wasted space between chunks

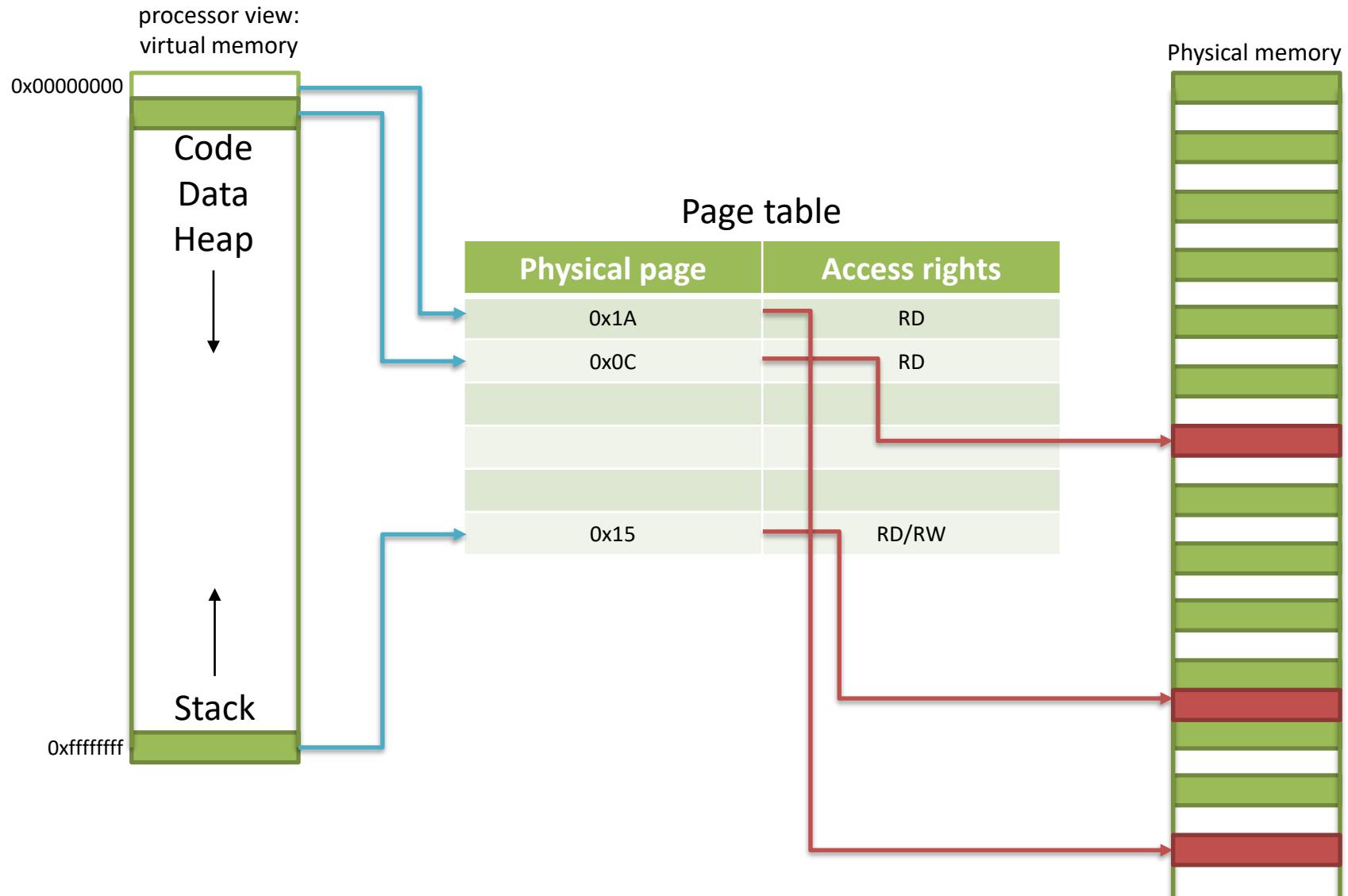
Paged Translation

- Manage memory in *fixed size units (page frames)*
- Finding a free page is easy
 - Bitmap allocation: 001111100000001100
 - Each bit represents one physical page frame
- Each process has its own page table
 - **Page table stored in physical memory.** Why?
 - We need two registers
 - pointer to page table start
 - page table length

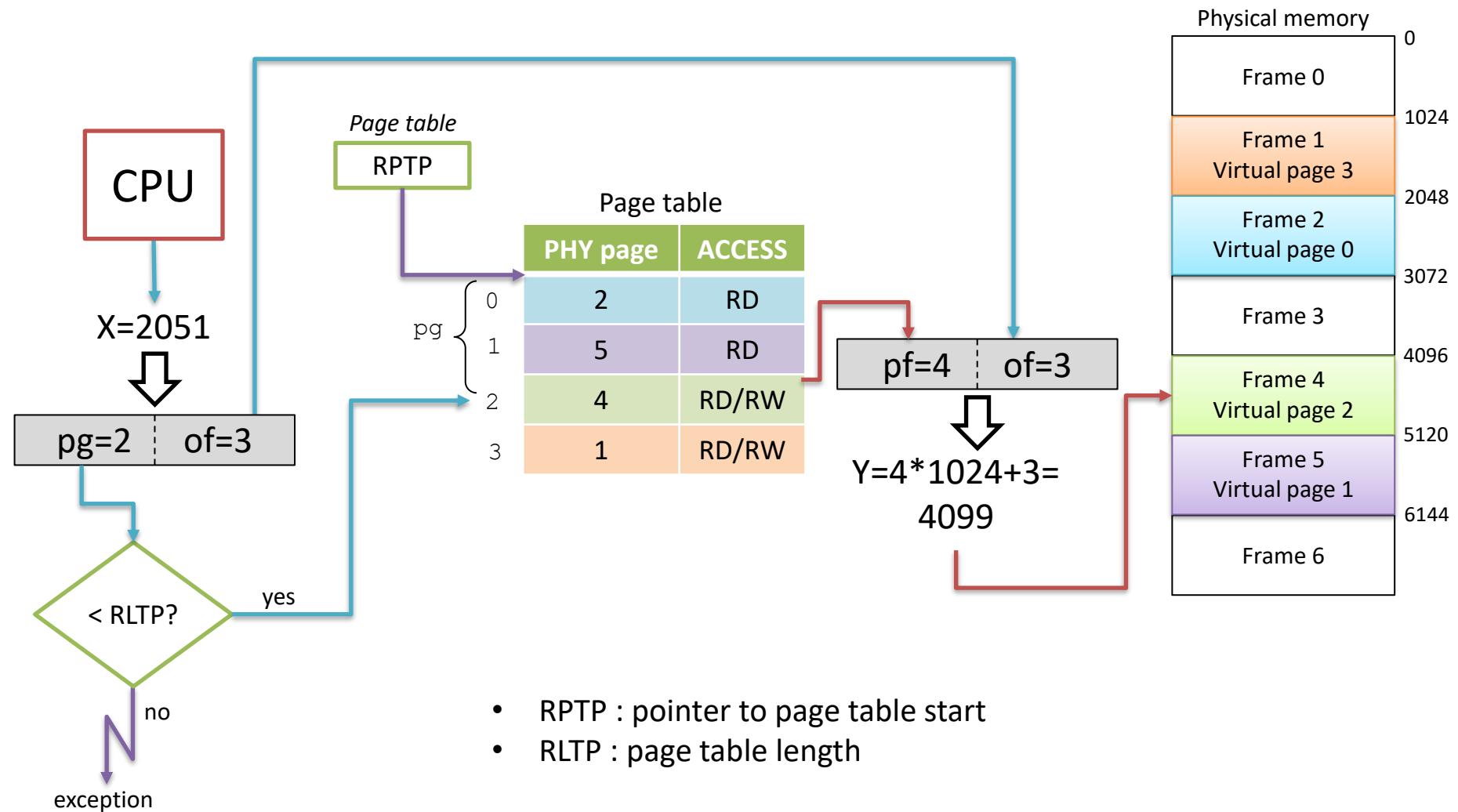
Virtual and physical spaces



Virtual and physical spaces



Paged translation



Paging Questions

- What must be saved/restored on a process context switch?
 - Pointer to page table/size of page table
 - Page table itself is in main memory
- What if page size is very small?
 - Huge page table
- What if page size is very large?
 - Internal fragmentation: if we don't need all the space inside a fixed size chunk

Page Sharing and Copy on Write

- Can we share memory between processes?
 - Set entries in both page tables to point to the same page frames
 - Need ***core map*** of page frames, a data structure to track which processes are pointing to which page frames
- UNIX fork with Copy on Write (CoW) at page granularity
 - Copy page table entries to new process
 - Mark all pages as read-only
 - Trap into kernel on write (in child or parent)
 - Copy page and resume execution

Paging and Fast Program Start

- Can I start running a program before its code is in physical memory?
 - Set all page table entries to invalid
 - When a page is referenced for first time
 - Trap to OS kernel
 - OS kernel brings in page
 - Resumes execution
 - Remaining pages can be transferred in the background while program is running

Sparse Address Spaces

- Might want many separate segments
 - Per-processor heaps
 - Per-thread stacks
 - Memory-mapped files
 - Dynamically linked libraries
- What if virtual address space is sparse?
 - On 32-bit UNIX, code starts at 0
 - Stack starts at $2^{32}-1$
 - 4KB pages => 1M page table entries
 - 64-bits => ~4 quadrillion page table entries

Multi-level Translation

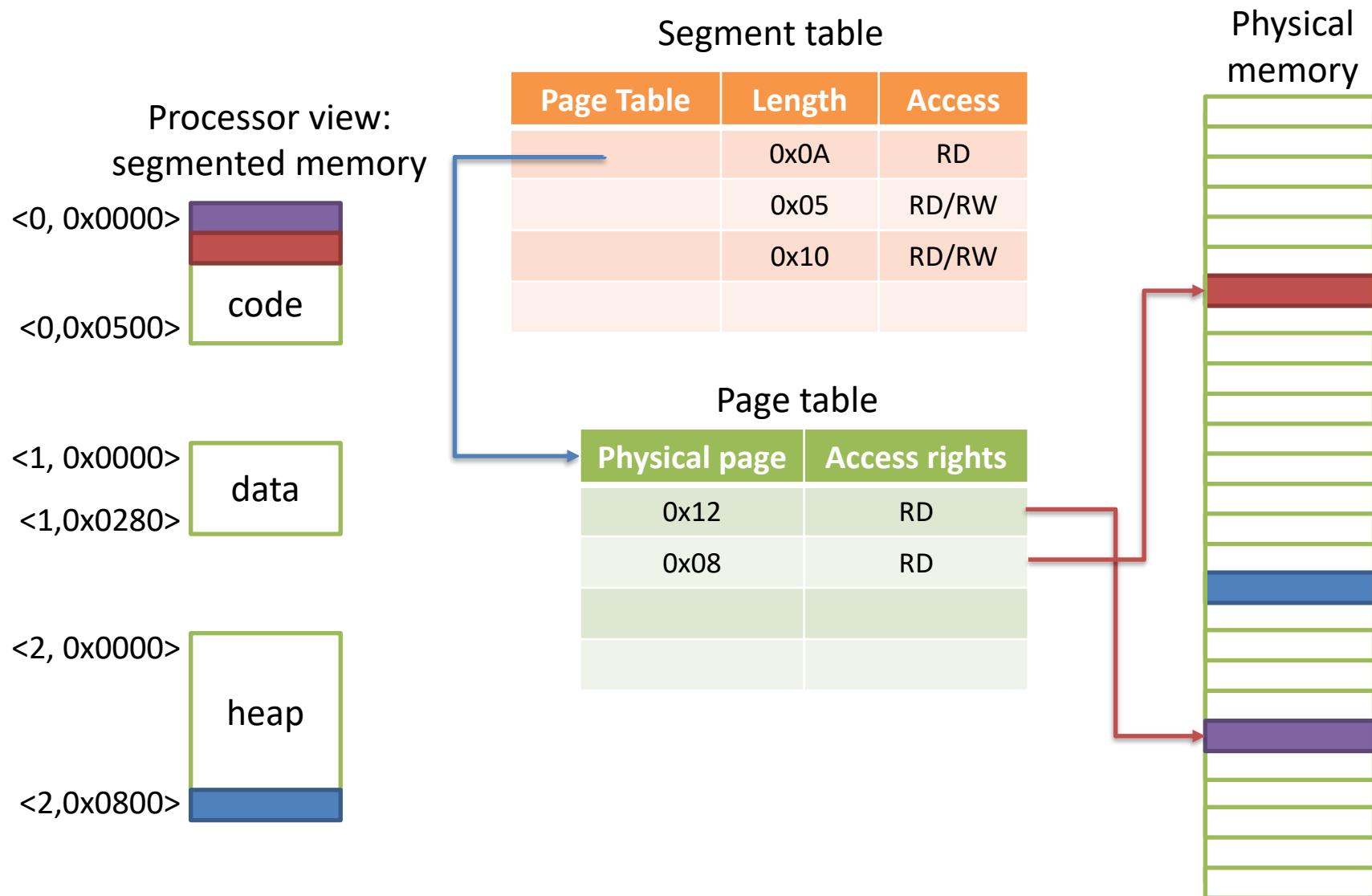
- Tree of translation tables
 1. *Paged segmentation*
 2. *Multi-level page tables*
 3. *Multi-level paged segmentation*
- All these approaches: fixed size page as lowest level unit
 - Efficient memory allocation
 - Efficient disk transfers
 - Easier to build Translation Lookaside Buffers (TLBs)
 - Efficient reverse lookup (from physical -> virtual)
 - Page granularity for protection/sharing

Paged Segmentation

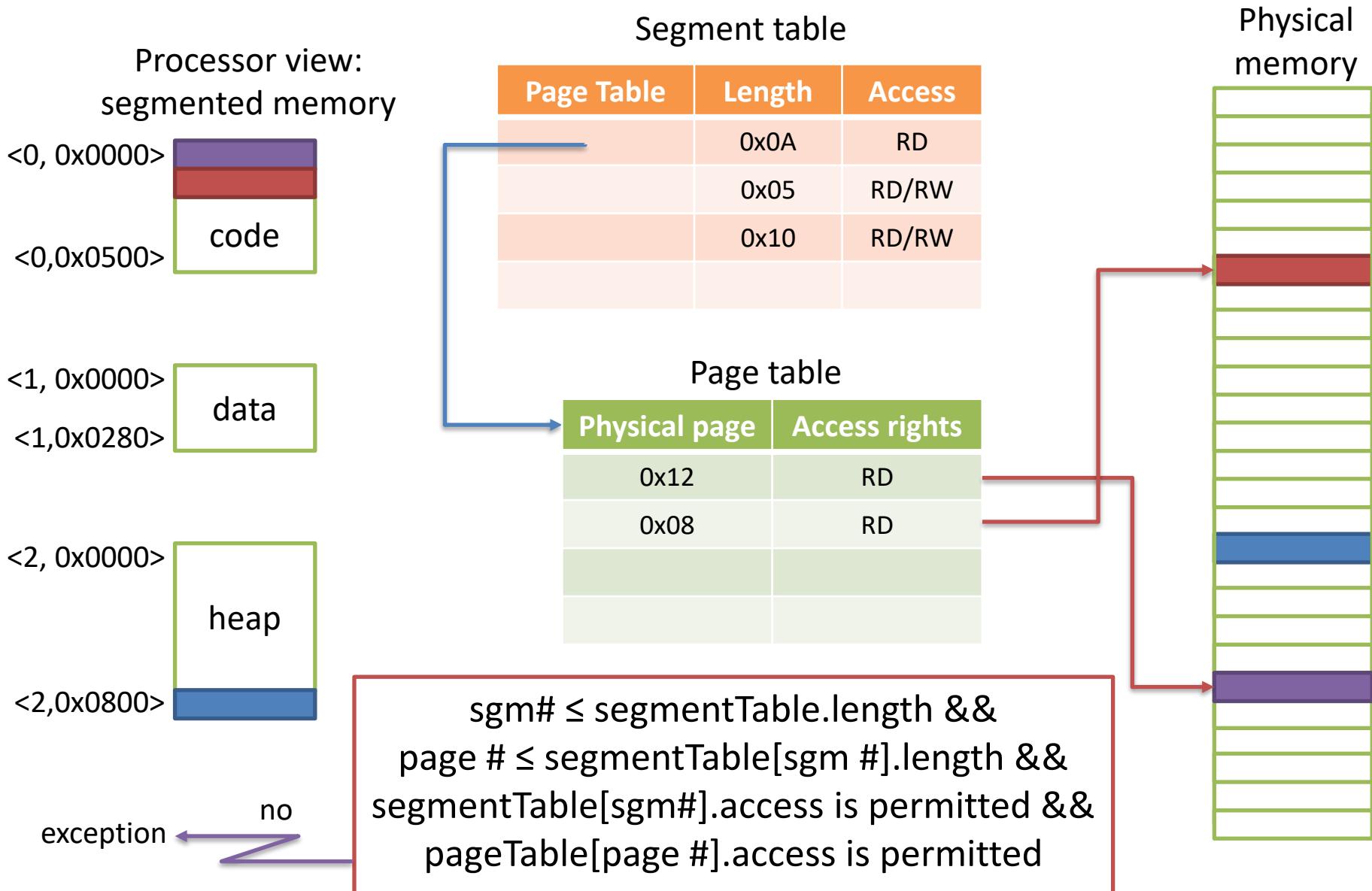
- Process memory is segmented
- The entry of the segment table is:
 - Pointer to page table
 - Page table length (# of pages in segment)
 - Access permissions
- The entry of the page table is:
 - Page frame
 - Access permissions
- Share/protection at either page or segment-level

Virtual address	segment #	page #	page offset
-----------------	-----------	--------	-------------

Physical address	SegmentTable[segment #].PageTable[page #]	page offset
------------------	---	-------------



Virtual address	sgm #	page #	page offset
Physical address	segmentTable[sgm #].pageTable[page #]		page offset



Question

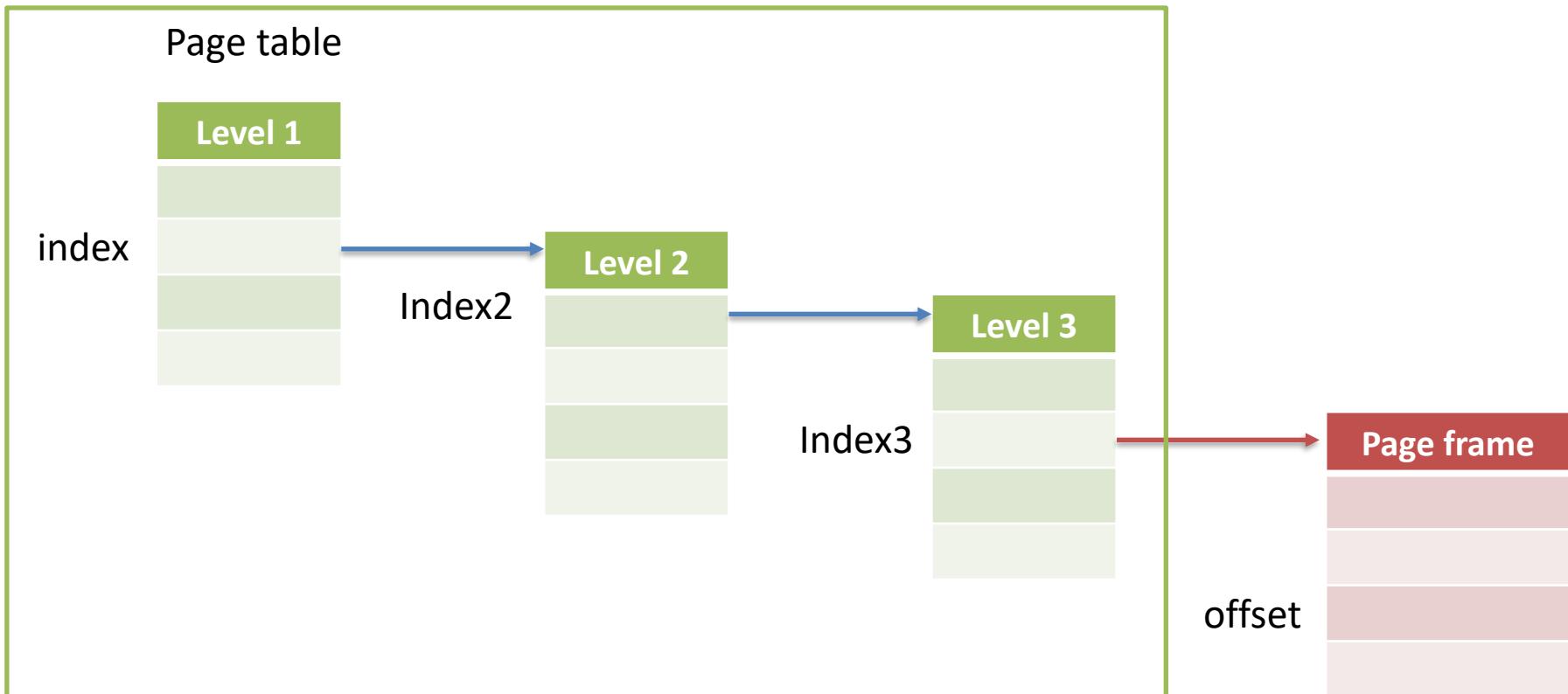
- With paged segmentation, what must be saved/restored across a process context switch?

Multilevel Paging

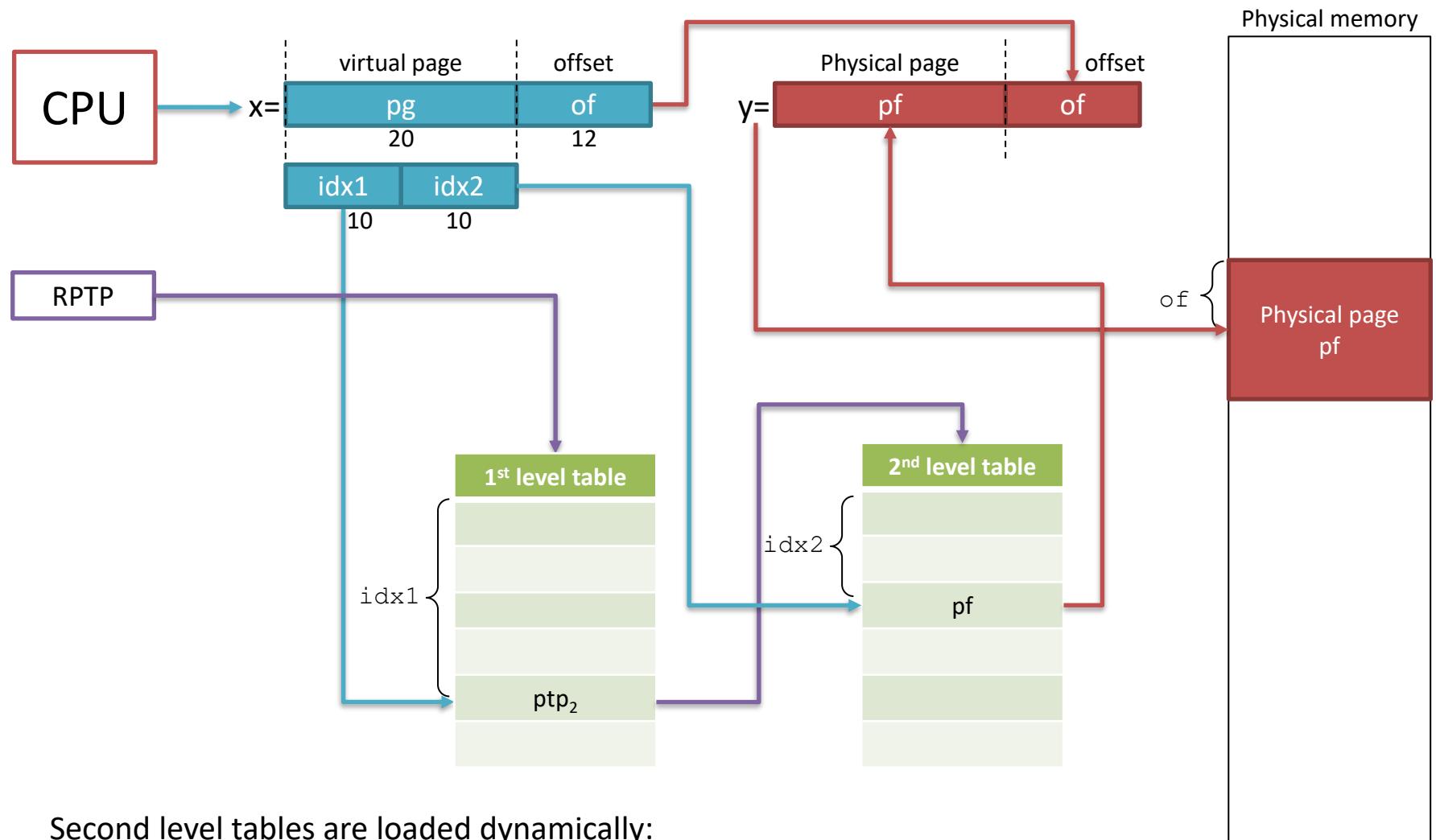
Virtual address



physical address=pageTable[index].pageTable[index2].pageTable[index3] | page offset



Two-level paging



Second level tables are loaded dynamically:

- reduce memory occupation

Comparing page table size

- Hypothesis:
 - 32 bit address;
 - logical and physical pages of 4K (\rightarrow 12 bit for the offset)
 - Page table entry of 4 byte
 - 3 bytes for block address
 - 1 byte for flags
- Page table size one single page level:
 - Number of table entries: 2^{20}
 - Space in bytes: $2^{20} * 4 = 2^{22}$ (4MB)
- Max physical memory size:
 - 3 bytes for the block address means 2^{24} pages \rightarrow 64GB

Comparing page table size

- Page table size two-level paging (same hypothesis):
 - Of the 20 bits, 10 bits are for index1 and 10 bits for index2
 - We have 2^{10} second level tables
 - First and second level page table size is $2^{10} * 4 = 2^{12}$ (4K)
 - Best case: only 1 table in the second level → size = 8 K
 - Worst case: all tables in the second level → size = 4K + 4M
 - In general $4K + n * 4K$ where n is the number of second level tables
- Max physical memory size:
 - 3 bytes for the block address means 2^{24} pages → 64GB

x86 Multilevel Paged Segmentation

- Global Descriptor Table (segment table)
 - Pointer to page table for each segment
 - Segment length
 - Segment access permissions
 - Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)
- Multilevel page table
 - 4KB pages; each level of page table fits in one page
 - Only fill page table if needed
 - 32-bit: two level page table (per segment)
 - 64-bit: four level page table (per segment)

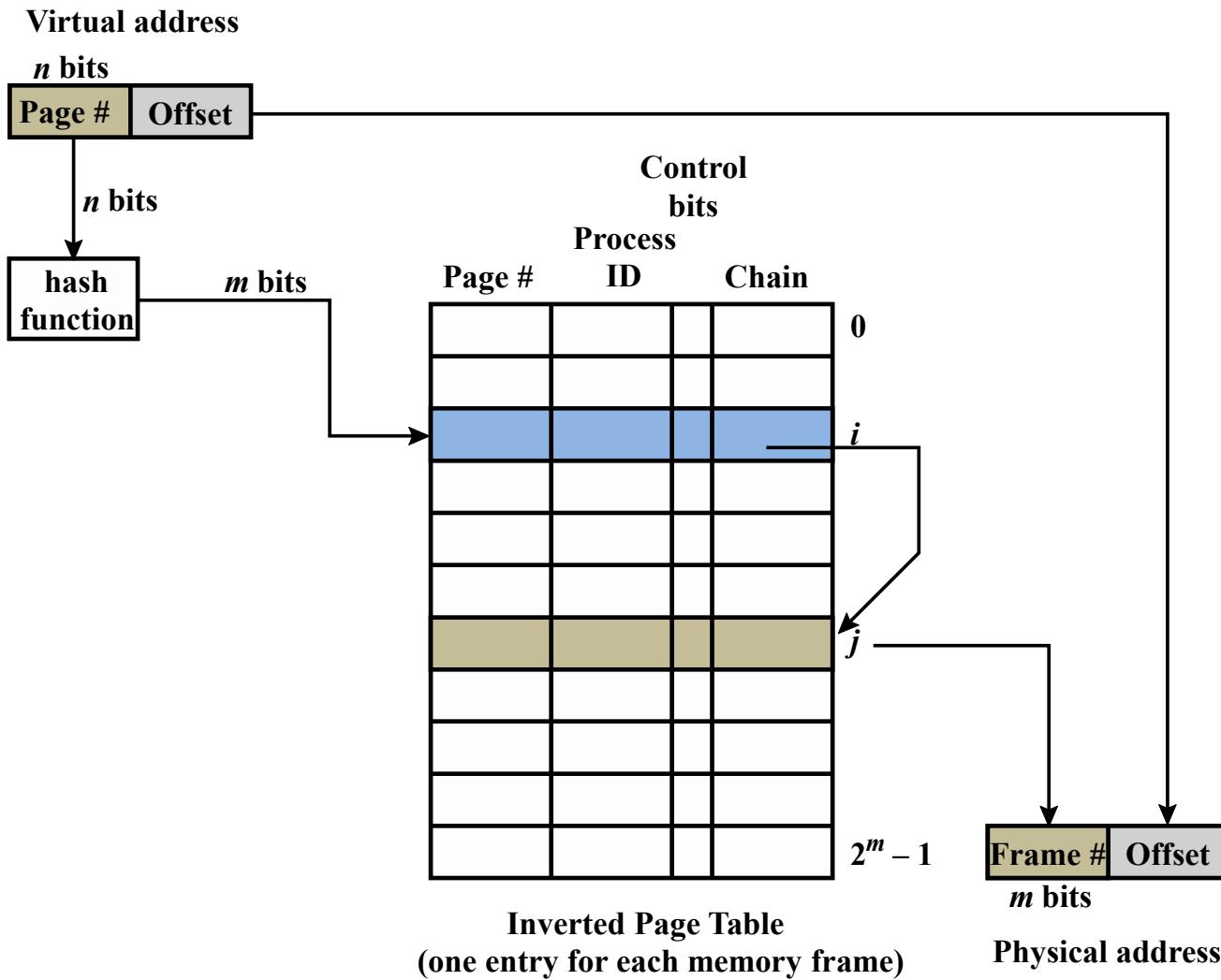
Multilevel Translation

- Pros:
 - Allocate/fill only as many page tables as used
 - Simple memory allocation
 - Share at segment or page level
- Cons:
 - Two or more lookups per memory reference

Portability

- Many operating systems keep their own memory translation data structures
 - List of memory objects (segments)
 - Virtual -> physical
 - Physical -> virtual (core map)
- Simplifies porting from different processors (x86, ARM, ...) from 32 bit to 64 bit
- *Inverted page table*
 - Hash from virtual page -> physical page
 - Space proportional to # of physical pages

Inverted Page Table



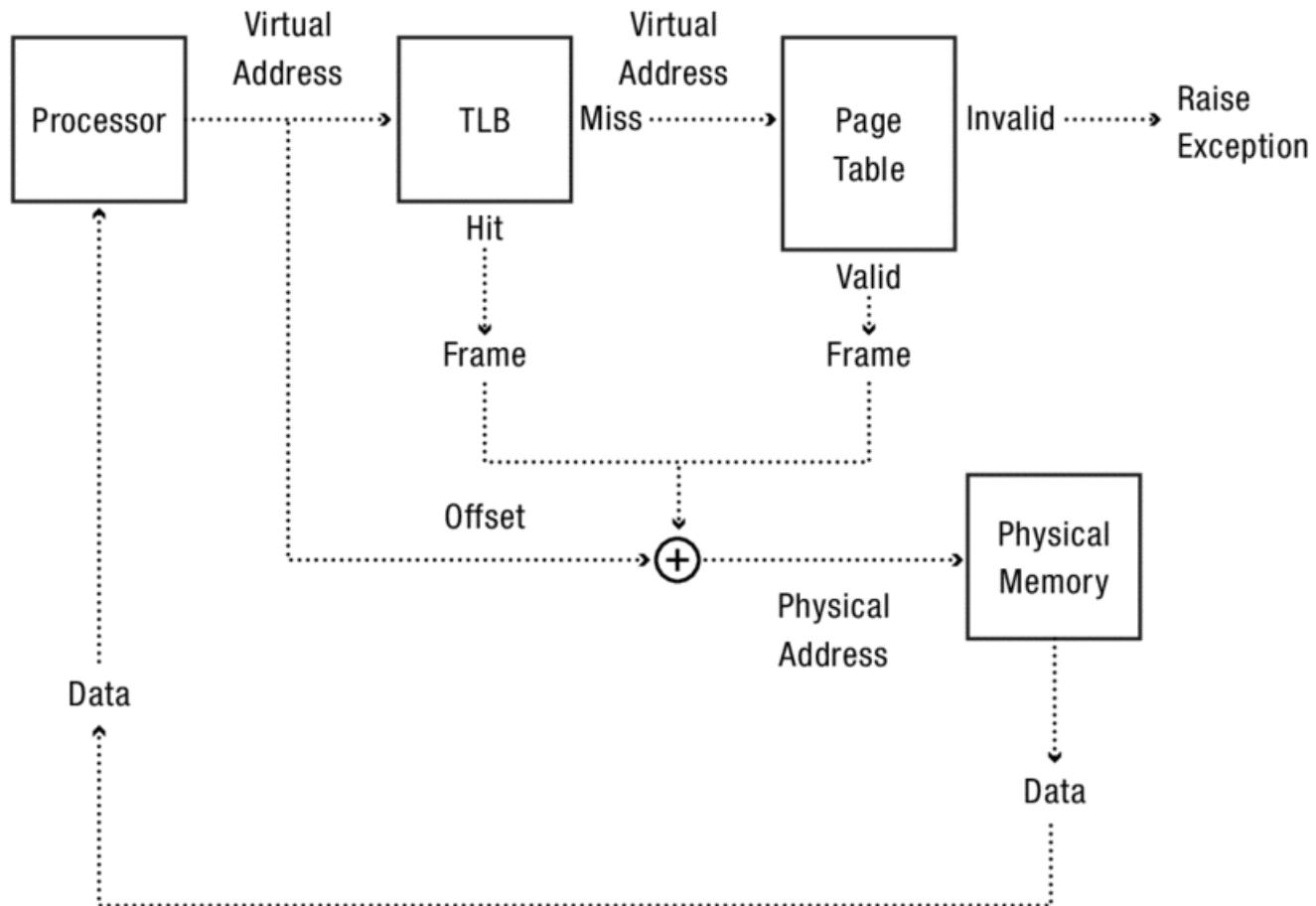
Do we need multi-level page tables?

- Use inverted page table in hardware instead of multilevel tree
 - IBM PowerPC
 - Hash virtual page # to inverted page table bucket
 - Location in Inverted Page Table => physical page frame
- Pros/cons?

Efficient Address Translation

- Access to page tables has good *locality*
- ***Translation Lookaside Buffer*** (TLB)
 - It is a cache of recent virtual page -> physical page translations
 - If cache hit, use translation available in the TLB entry
 - If cache miss, walk multi-level page table
- TLB cache is logically placed in the MMU
- Typical values for a TLB are:
 - Size: 16—512 entries; Block size: 1—2 page table entries
 - Hit time: 0.5—1 clock cycles; Miss Penalty: 10—100 clock cycles
 - Miss rate: 0.01%-1%
- We may have multiple TLB levels

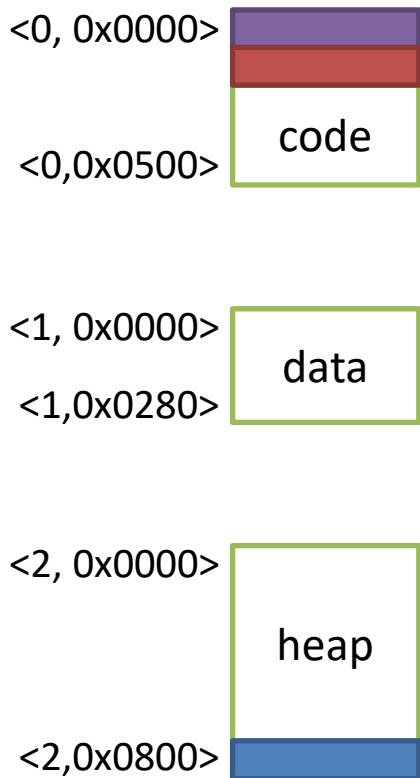
Address Translation with TLB



$$\text{Cost of Address Translation} = TLB_{hit} + TLB_{miss} * \text{CostFullTranslation}$$

Virtual address	Virtual page #	page offset
Physical address	TLBlookup[virtual page #].pageFrame	page offset

Processor view:
segmented memory



Translation Lookaside Buffer (TLB)

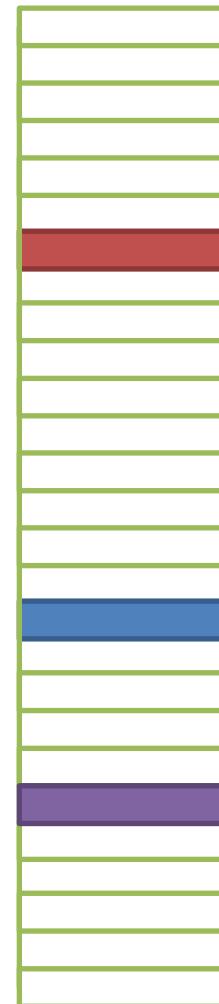
	virtual Page	Page Frame	Access
=?	0x0000	0x0A	RD
=?	0x40FF	0x05	RD/RW
=?	0x0001	0x10	RD/RW
=?			invalid

TLB contains Virtual page #?

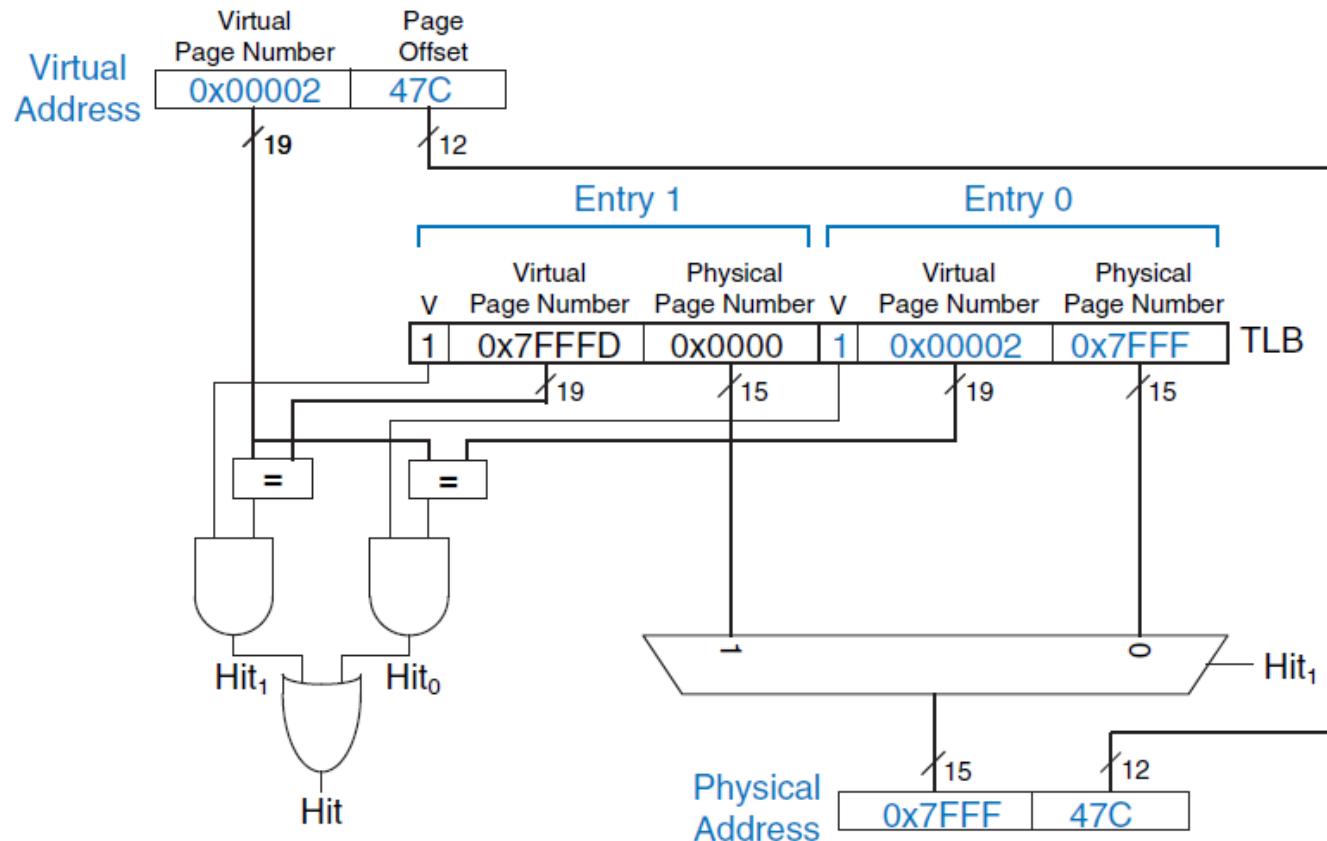
no

do page table lookup

Physical
memory



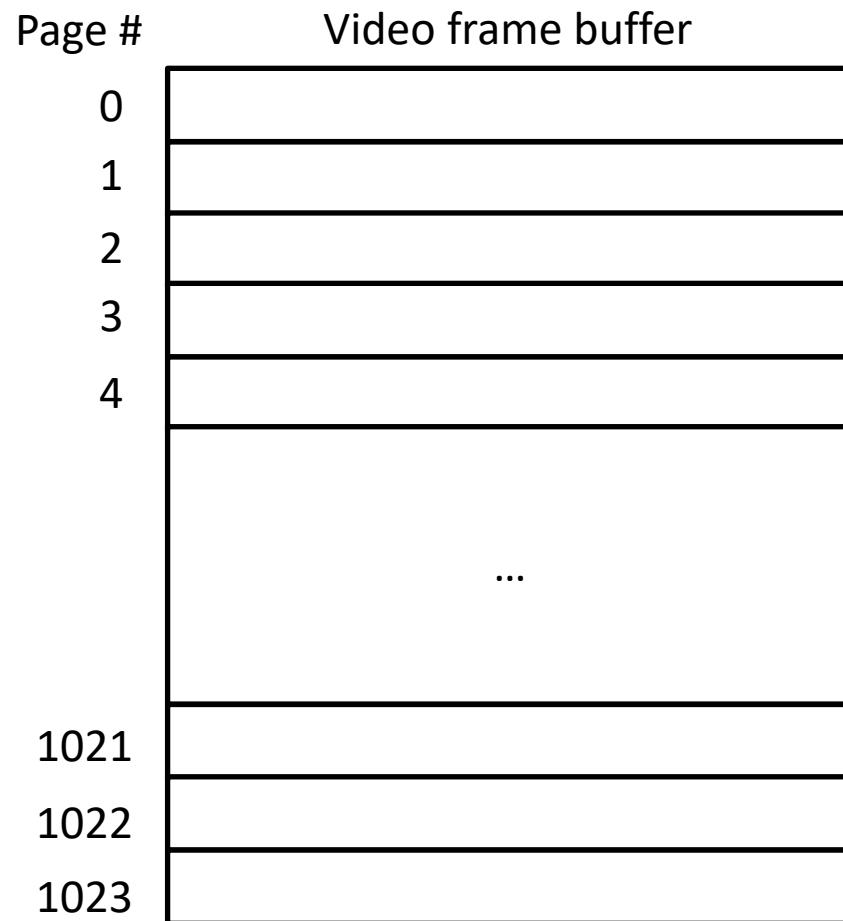
Two-entry TLB schema



Software Loaded TLB

- Do we need a page table at all?
 - MIPS processor architecture
 - If translation is in TLB, ok
 - If translation is not in TLB, trap to kernel
 - Kernel computes translation and loads TLB
 - Kernel can use whatever data structures it wants
- Pros/cons?

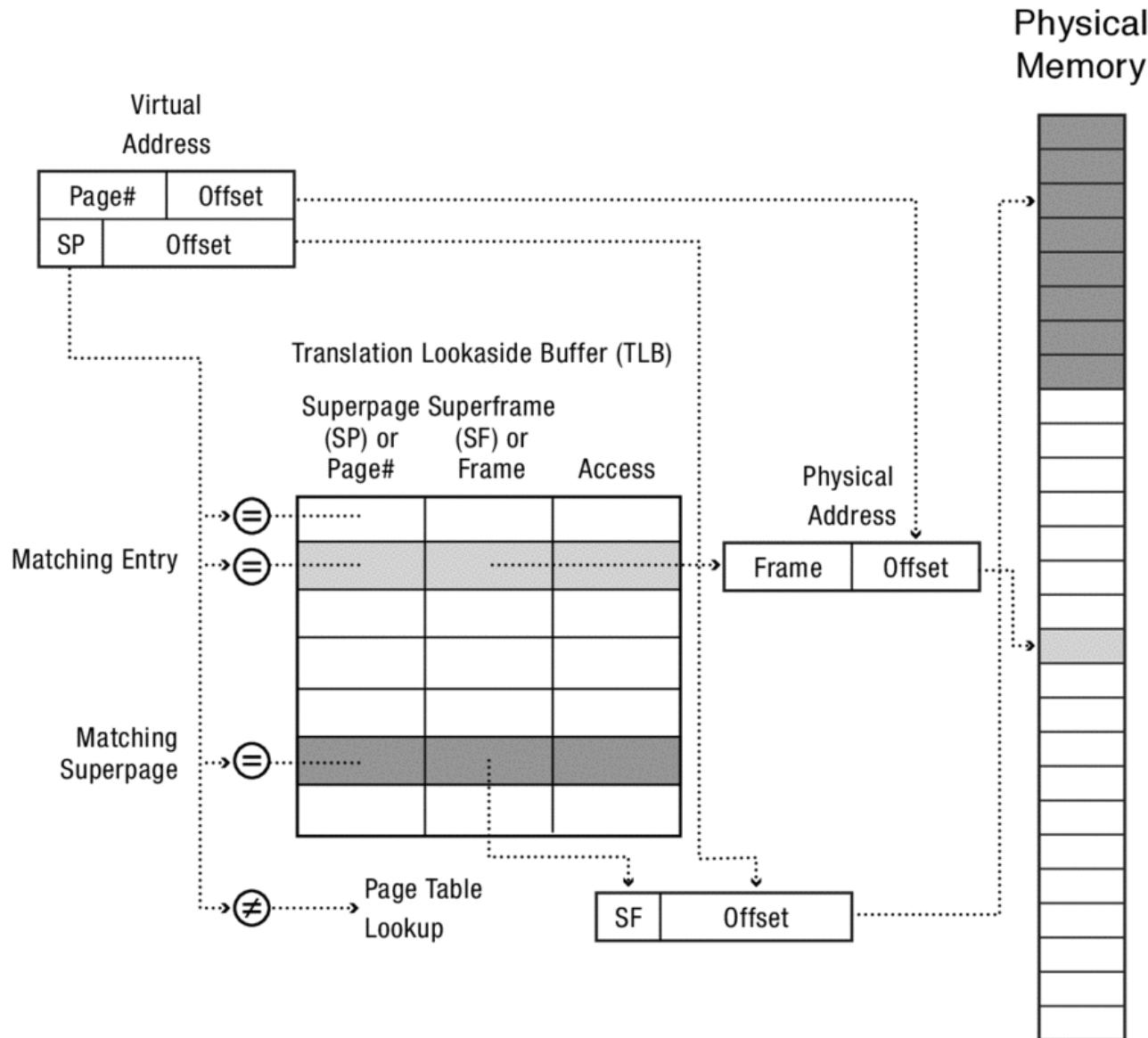
When Do TLBs Work/Not Work?



Superpages

- TLB entry can be
 - A page
 - A superpage: a set of contiguous pages
 - x86: superpage is set of pages in one page table
 - x86 TLB entries
 - 4KB
 - 2MB
 - 1GB

Superpages

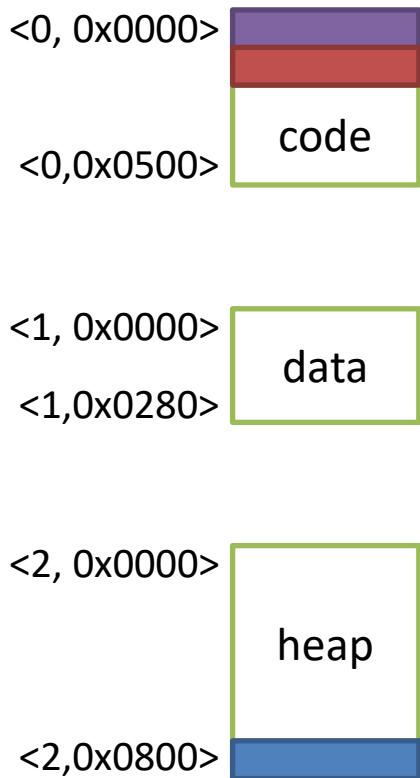


When Do TLBs Work/Not Work, part 2

- What happens on a context switch?
 - Reuse TLB?
 - Discard TLB?
- Motivates hardware tagged TLB
 - Each TLB entry has process ID
 - TLB hit only if process ID matches current process

Virtual address	Virtual page #	page offset
Physical address	TLBlookup[virtual page #].pageFrame	page offset

Processor P view:
segmented memory



Translation Lookaside Buffer (TLB)

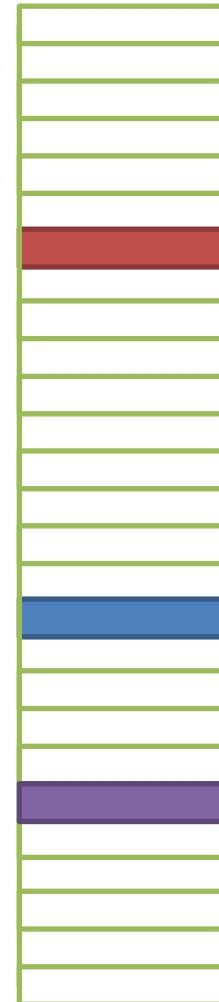
PID	virtual Page	Page Frame	Access
=?	0	0x0000	RD
=?	1	0x40FF	RD/RW
=?	1	0x0001	RD/RW
=?	0	0x003F	RD

TLB contains Virtual page #
For the current process P?

no

do page table lookup

Physical
memory

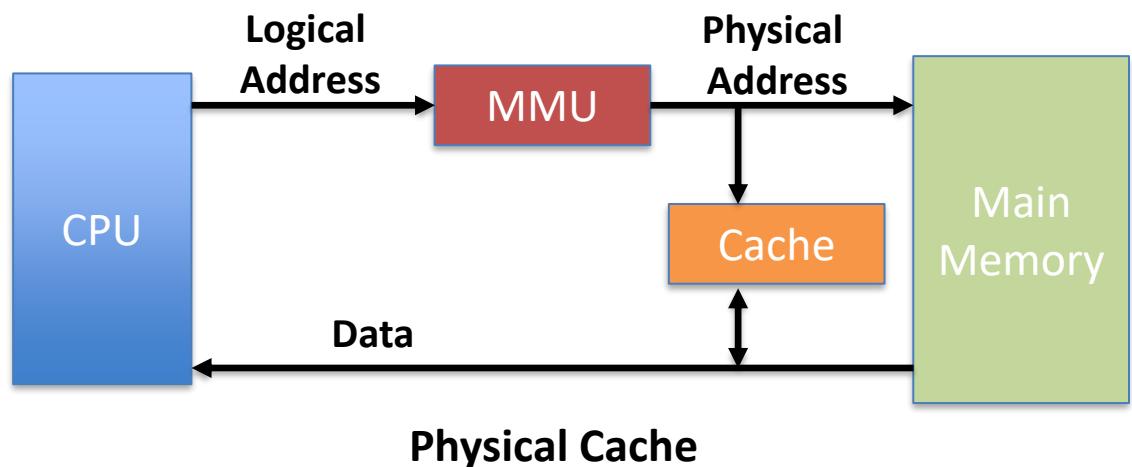
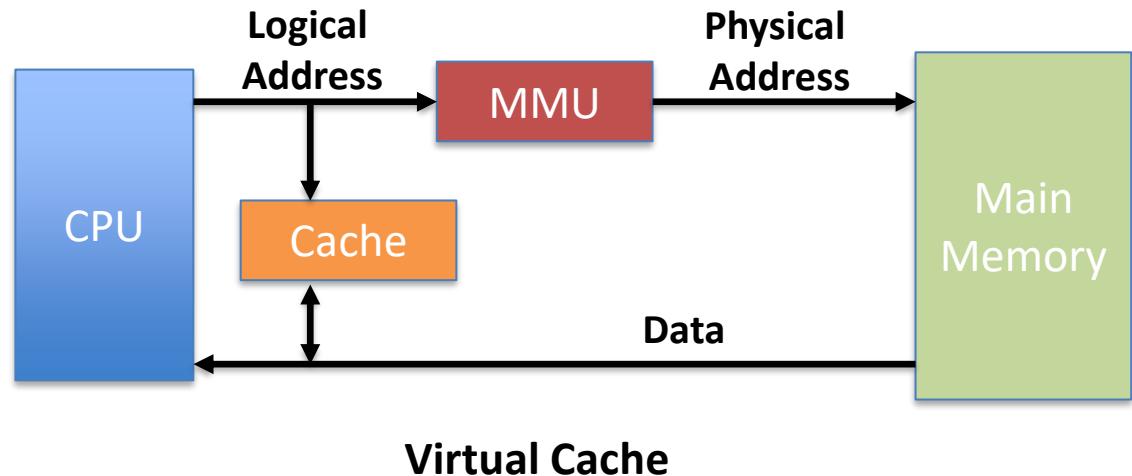


When Do TLBs Work/Not Work, part 3

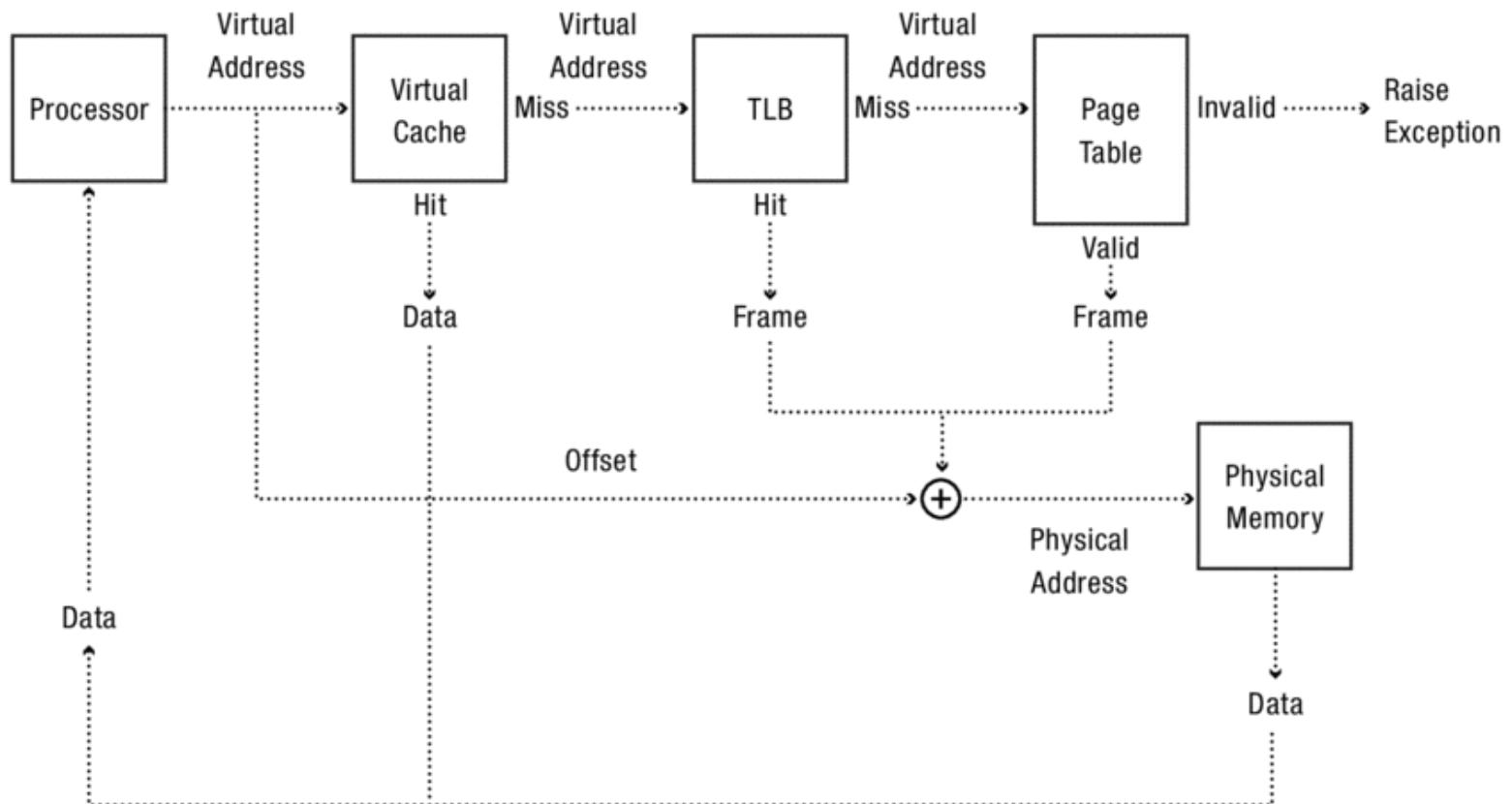
- What happens when the OS changes the permissions on a page?
 - For demand paging, copy on write, zero on reference, ...
- TLB may contain old translation
 - OS must ask hardware to purge TLB entry
- On a multicore, ***TLB shootdown*** phenomenon
 - OS must ask each CPU to purge TLB entry

Virtual and Physical Cache

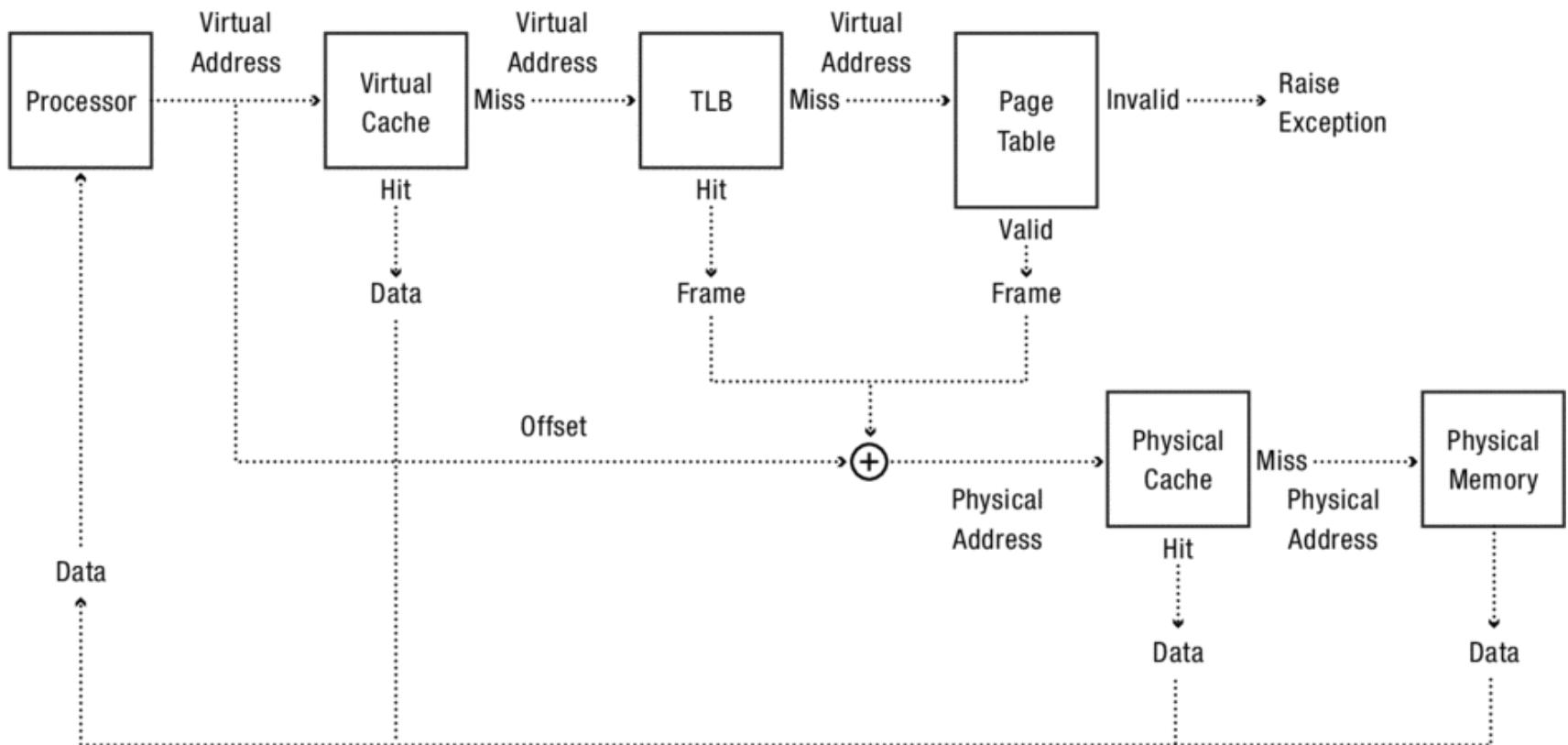
- Virtual cache faster because the cache can respond before the MMU performs an address translation
- But it requires to tag cache entries (same virtual address space for processes), thus more complexity and more space needed



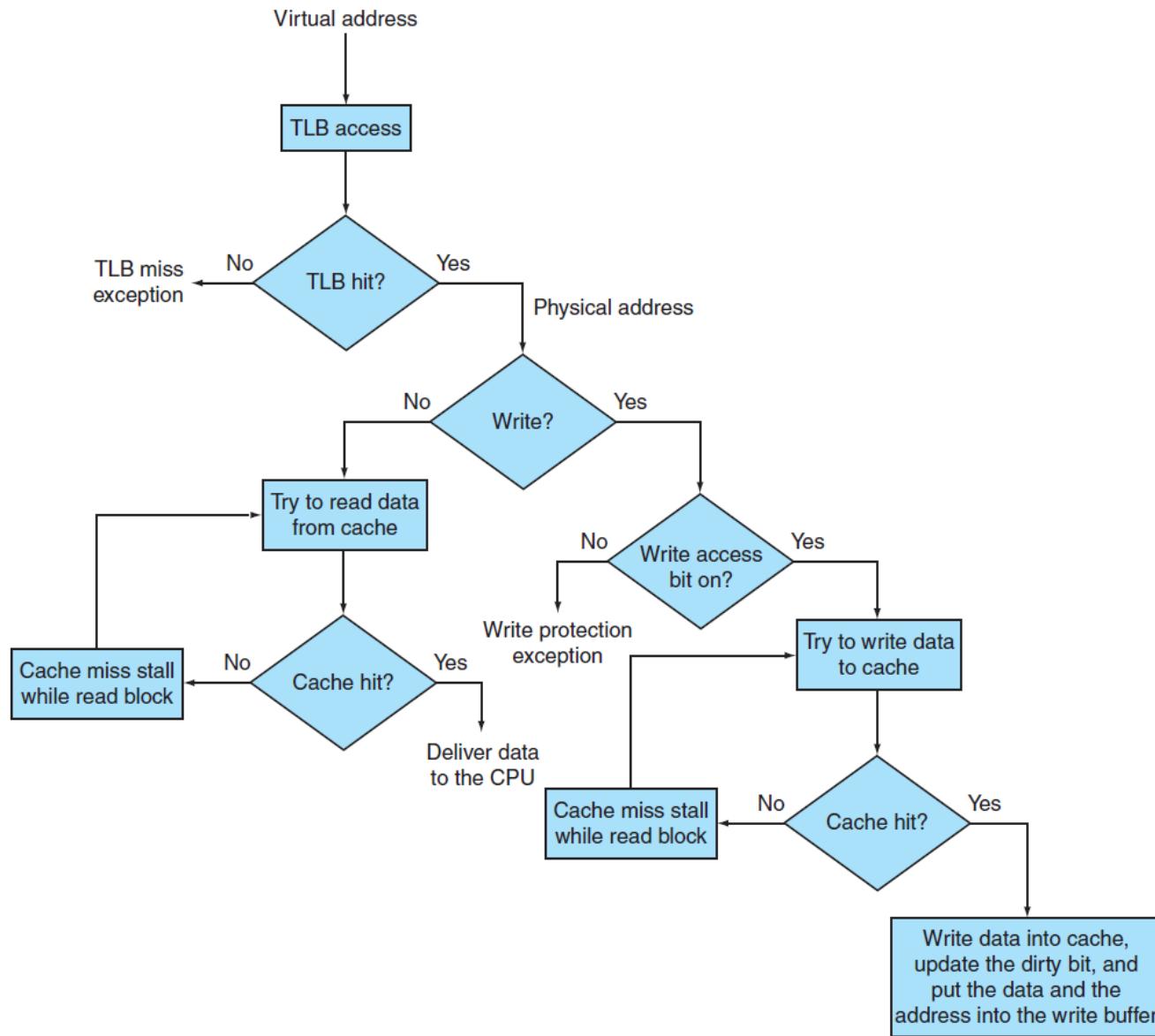
Virtually addressed Cache



Physically addressed Cache



TLB and Cache Interaction



Demand-Paged Virtual Memory and Page Caching

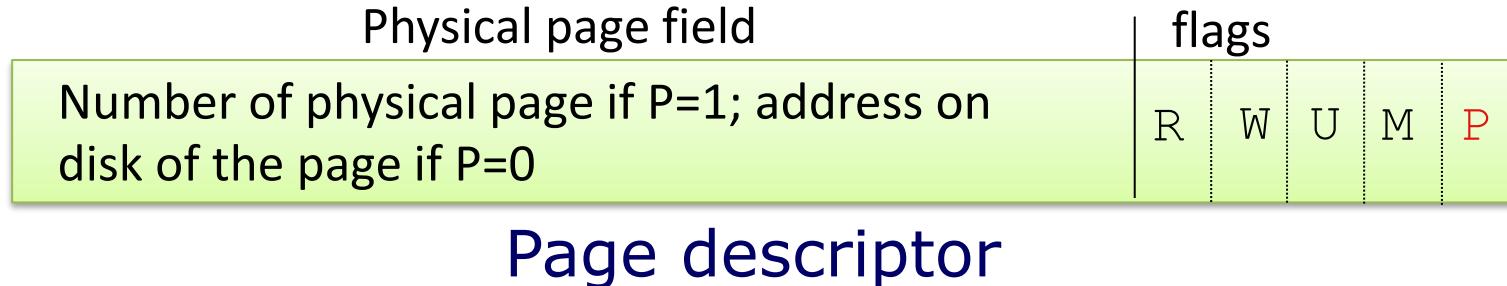
Main Points

- Demand-paged Virtual Memory
- Cache Replacement Policies
 - FIFO, MIN, LRU, LFU, Clock
- Memory-mapped files
- Use cases

Virtual Memory

- The OS provides applications the abstraction of more memory than is physically present
- The abstraction is implemented considering the main memory ***as a cache for disk(s)***
- Application pages that fits in memory have valid page table entries
- Pages not resident in memory have invalid page table entries and are loaded *on-demand* upon access by the OS
 - Such process is called ***demand paging***

Demand Paging

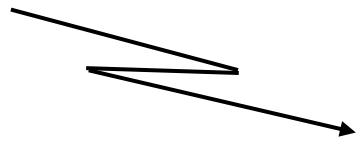


The page table contains a page descriptor for each page

Beyond the information for translation of the address, the descriptor contains some flags:

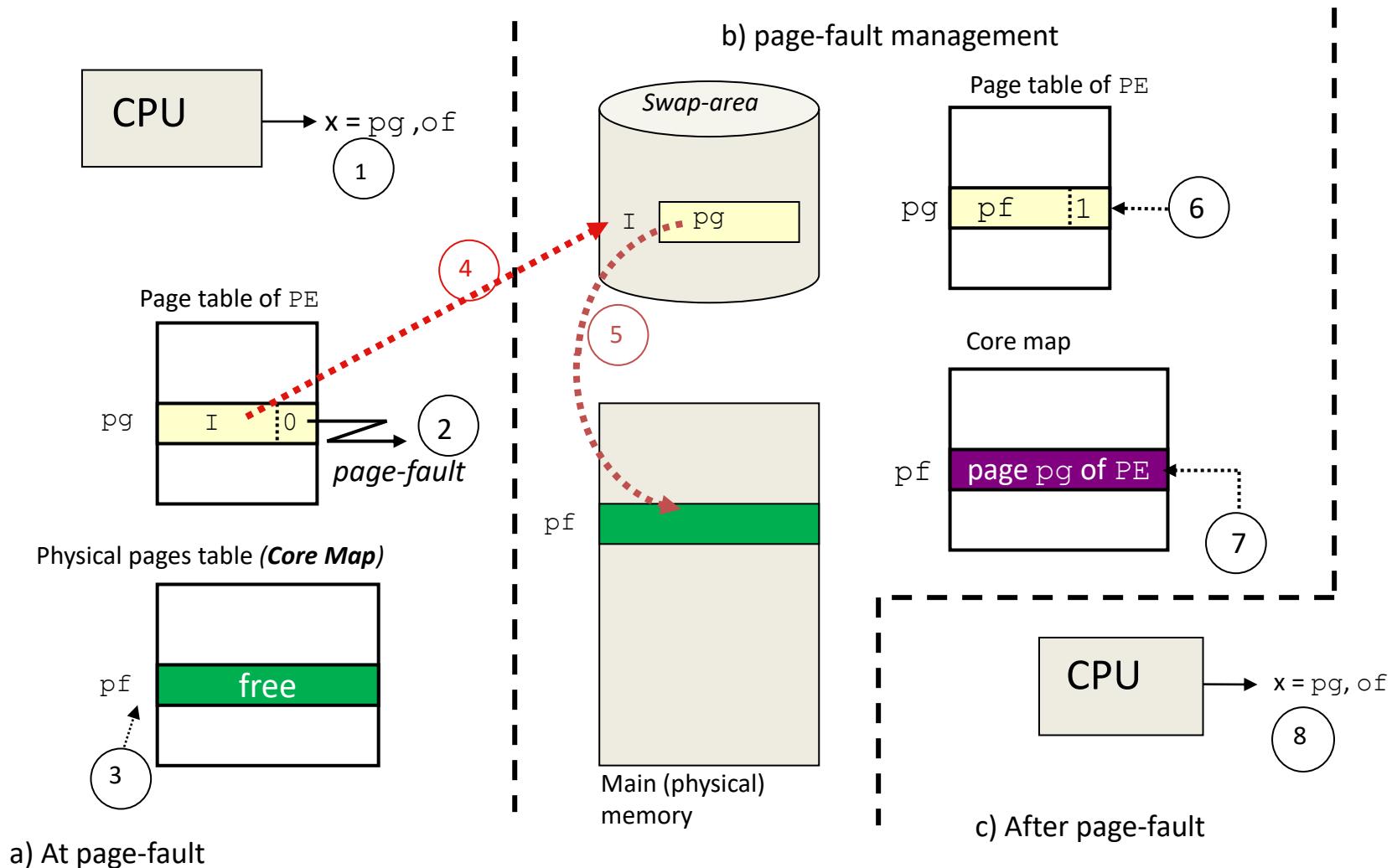
- R, W: read/write access rights
- M, U: modified/use bits (for the page replacement algorithms)
- P: presence bit (page is invalid if not present in main memory)

- P = 1: page in main memory
- P = 0: page not in main memory



page-fault

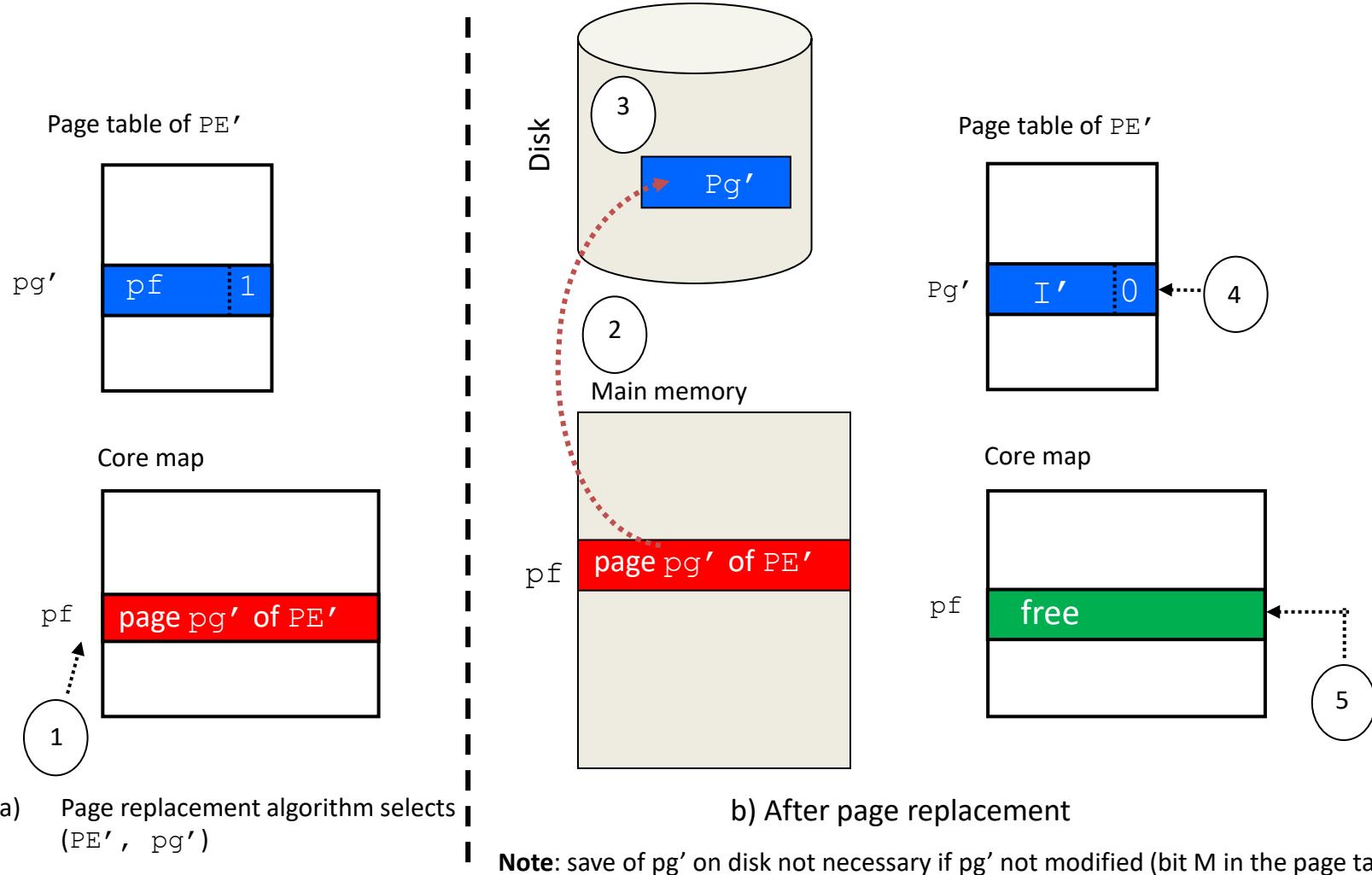
Page fault management



On-demand Paging

1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert address to file + offset
6. Allocate page frame
 - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction

Page replacement



Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
 - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
 - Copies of now invalid page table entry
- Write changes to page to disk, if necessary
 - i.e. if the page had been modified

How do we know if page has been modified?

- Every page table entry has some bookkeeping
 - Has page been modified?
 - Set by hardware on *store* instruction to page
 - In both TLB and page table entry
 - Has page been used?
 - Set by hardware on *load* or *store* instruction to page
 - In page table entry on a TLB miss
- Can be reset by the OS kernel
 - When changes to page are flushed to disk
 - To track whether page is recently used

TLB

	dirty = 0

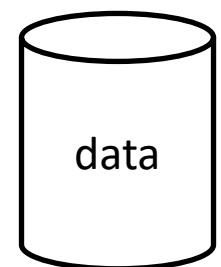
Page Table

Frame	Access
	dirty = 0

Physical memory
page frames



disk



TLB

	dirty = 1

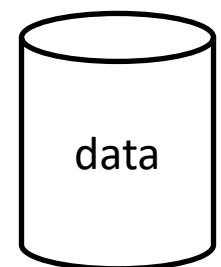
Page Table

Frame	Access
	dirty = 1

Physical memory
page frames



disk



Emulating M and U bits

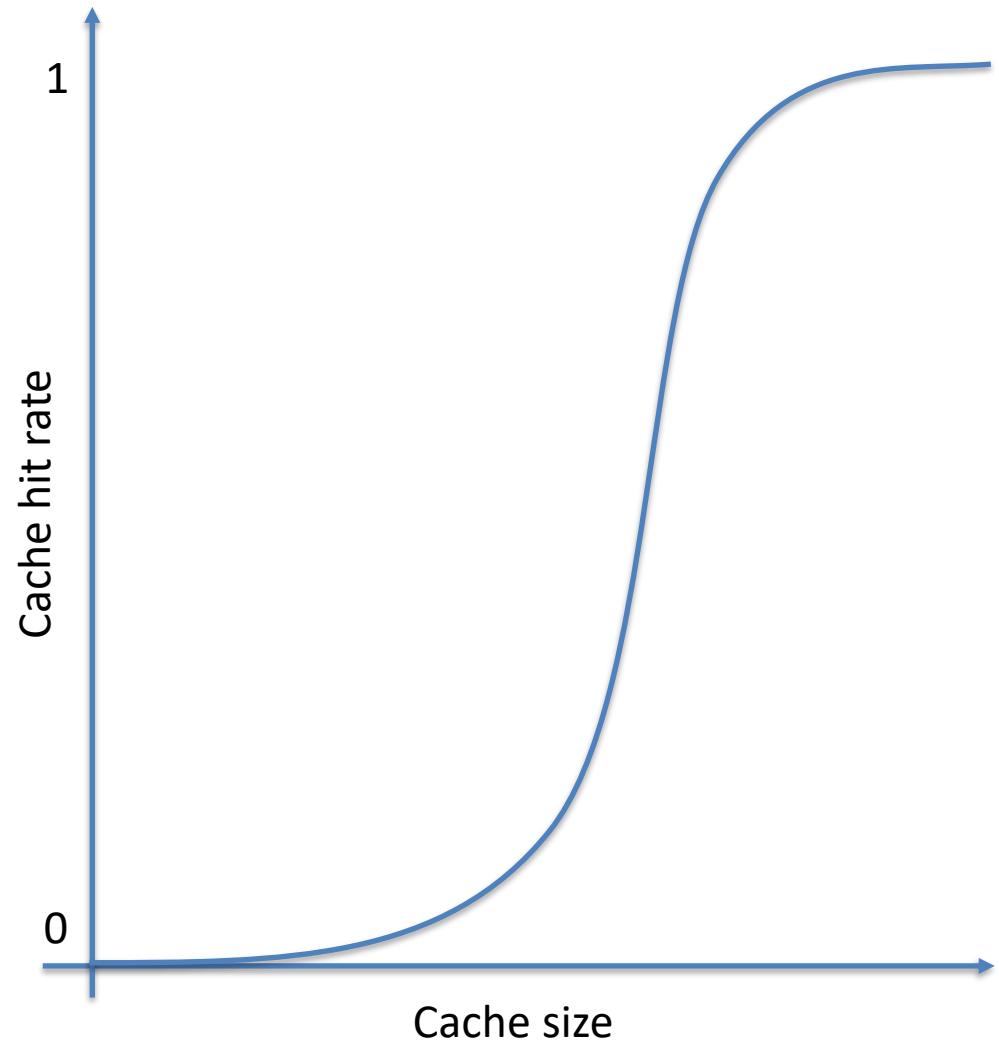
- Some processor architectures do not keep a use bit in the page table entry
 - Extra bookkeeping and complexity
- OS can emulate a use bit:
 - For M bit: set all clean pages as read-only
 - For U bit: set all unused pages as invalid
 - On first write (M) or read/write (U), take page fault to kernel
 - From info in the ***core map*** the kernel knows what to do
 - Kernel sets M bit and marks page as read-write
 - Kernel sets U bit and marks page as read or read/write

Demand-paged Virtual Memory

- Problem:
 - If the OS pages out to disk (swap) a page just before it is about to be used, then it will have to get that page again almost immediately
 - Too much of this leads to a condition known as ***thrashing***
 - Thrashing: the system spends most of the time swapping pages to disk rather than executing application's instructions

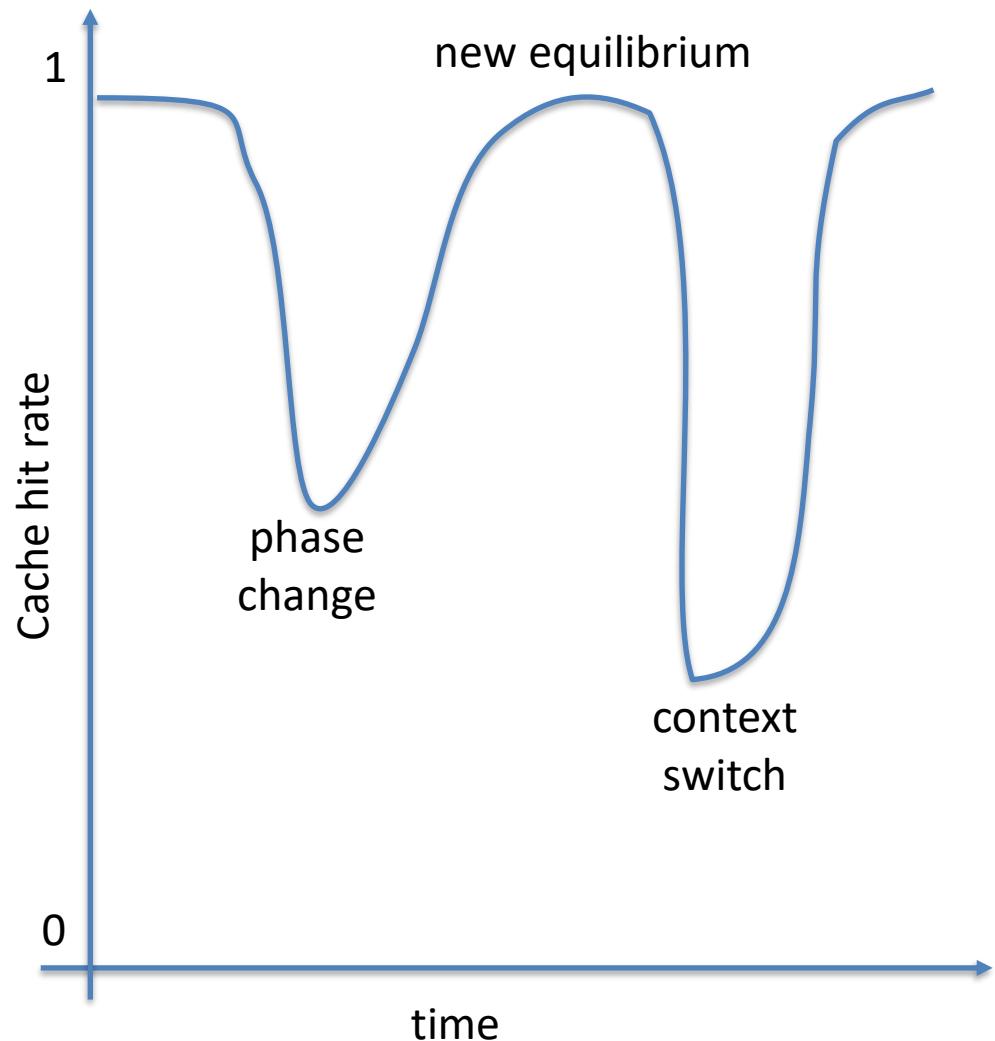
Working Set

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- Thrashing happens when system has a too small cache



Phase Change Behavior

- Programs may change their working set over time
- Context switches also change working set



Cache Replacement Policy

- On a cache miss, how do we choose which entry to replace?
 - Assuming the new entry is more likely to be used soon
- Policy goal: *reduce cache misses*
 - Improve expected case performance
 - Also: reduce likelihood of very poor performance

A Simple Policy

- Random?
 - Replace a random page
- FIFO?
 - Replace the page that has been in the cache (main memory) the longest time
 - What could go wrong?

FIFO in Action

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E			D		
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is if program strides through memory that is larger than the main memory

- working set larger than cache capacity

MIN, LRU, NRU

- MIN (ideal, optimal)
 - Replace the page that will not be used for the longest time into the future (i.e., whose ***future distance*** is maximum)
 - Optimality proof based on exchange: if evict a page used sooner, that will trigger an earlier page fault (=cache miss)
- Least Recently Used (LRU)
 - Replace the page that has not been used for the longest time in the past (i.e., whose ***past distance*** is maximum)
 - Approximation of MIN
- Not Recently used (NRU)
 - Replace one of the pages that have not been used recently
 - Relax the requirement of LRU
 - Approximation of LRU, easier to implement
 - LFU (Least Frequently Used) replace the cache entry used the least often
 - Examples: second chance, working set algorithms

LRU/MIN for Sequential Scan

LRU

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

MIN

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A					+					+				
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

LRU

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A			+				+				+			+
2		B			+								+		
3				C					E			+			
4					D			+		+					C

FIFO

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A			+				+		E			+		
2		B			+						A			+	
3				C									B		
4					D			+		+					C

MIN

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A			+				+				+			+
2		B			+								+		
3				C					E			+			
4					D			+		+					C

Belady's Anomaly

FIFO (3 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+	D		

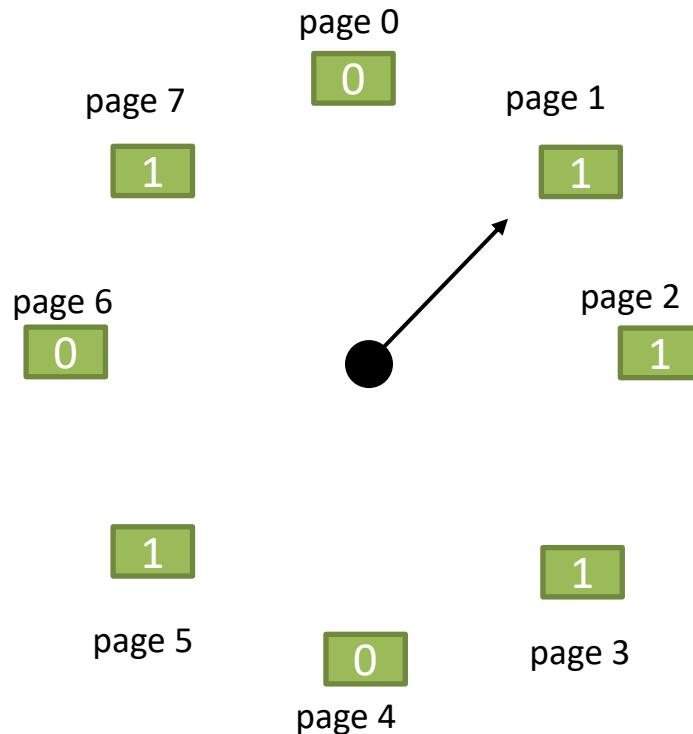
FIFO (4 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

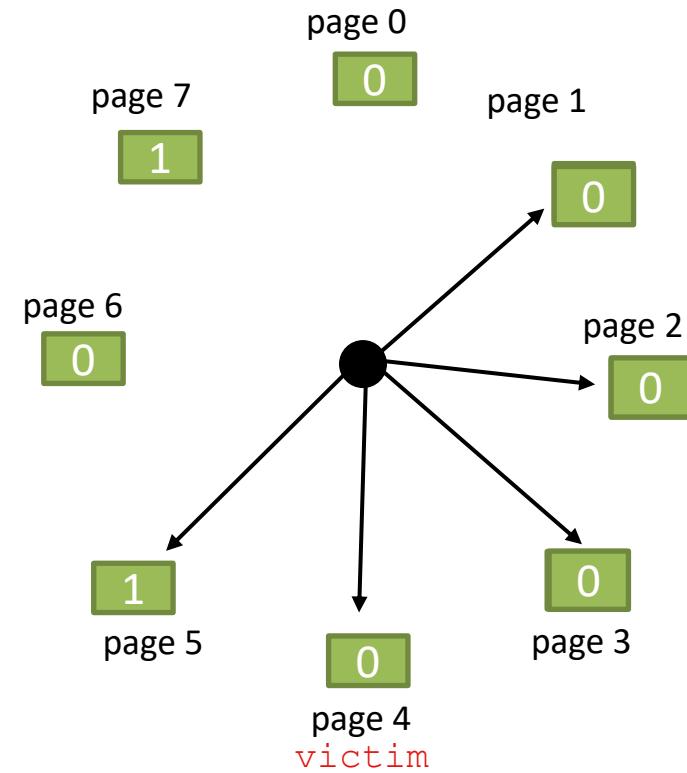
Clock Algorithm: estimating LRU

- Periodically, sweep through all pages
- If page is unused, reclaim
- If page is used, mark as unused

Second-Chance (clock algorithm)



a) Initial state



b) Next states of second chance

N^{th} Chance: generalization of second chance

- Instead of one bit per page, keep an integer
 - `notInUseSince`: # of sweeps since last use
- Periodically, sweep through all page frames
- If page hasn't been used in any of the past N sweeps, reclaim
- If page is used, mark as unused and set as active in current sweep

```
if (page is used) {  
    notInUseSince = 0;  
} else if (notInUseSince < N) {  
    notInUseSince++;  
} else {  
    reclaim page;  
}
```

Local and global page replacement

- Global algorithms:
 - The page selected for removal is selected among all pages in main memory
 - Irrespective of the owner
 - “past distance” of a pages defined based on a global time (absolute clock)
 - May result in trashing of slow processes
- Local algorithms
 - The page selected for removal belongs to the process that caused the page fault
 - Fair with “slow” processes about trashing
 - Past distance of a page based on relative time
 - The time a process has spent in running state

Local vs global page replacement

T: time of last reference

a)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

b)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

c)

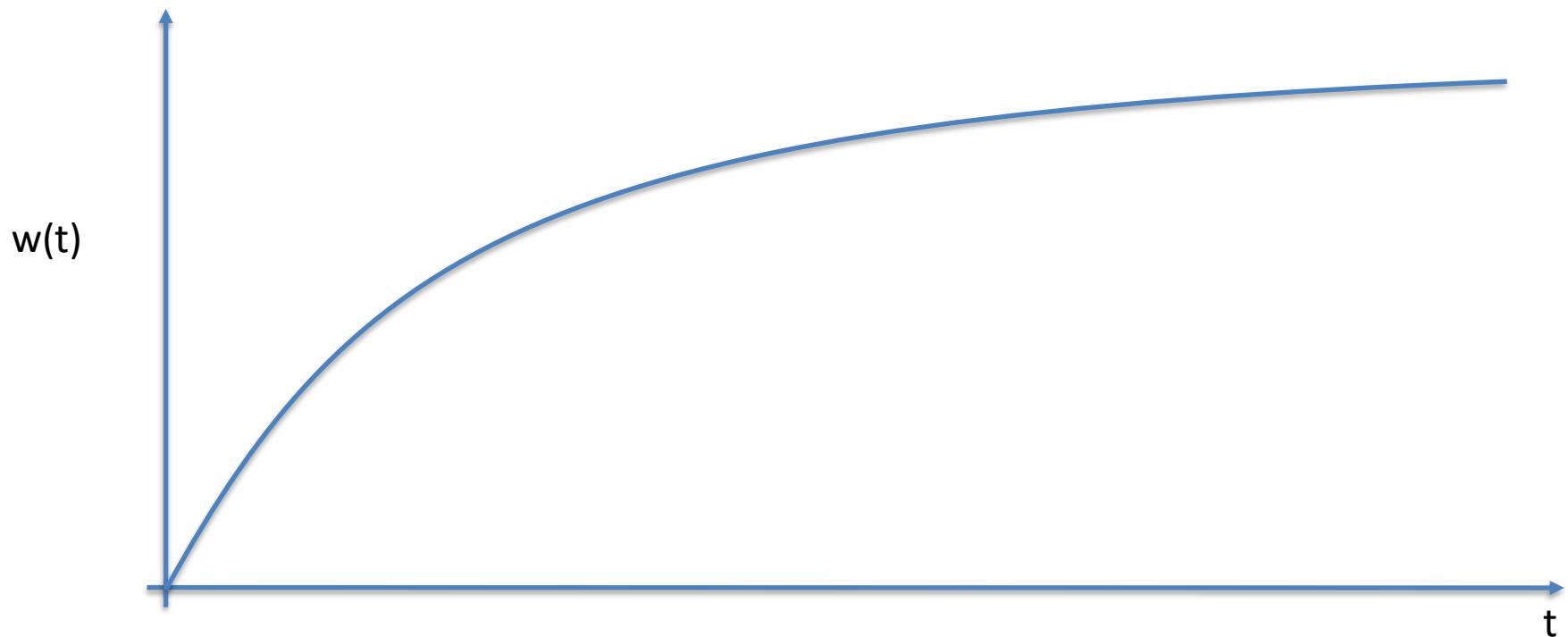
	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

- a) Initial configuration
- b) Page replacement with a local policy (WS, LRU, sec. chance)
- c) Page replacement with a global policy (LRU, sec. chance)

Working set algorithm

- Keep in memory the *working set of a process*:
 - the pages that the process is currently using
- Working set defined as:
 - The set of pages referred in the last k memory accesses
 - difficult and costly to implement
 - The set of pages referred in the last period T
 - usually implemented in this way, using the «use» bit

Working set



- working set: The set of pages referred in the last k memory accesses
- $w(t)$ is the size of the working set as function of time

Working set algorithm

- Each process has a number of physical pages reserved to upload its working set
 - WS replacement policy is inherently local
- Resident set:
 - is the actual set of virtual pages in main memory
 - some of them may be out of the working set
 - Resident set \neq working set

Working set algorithm

- WS defined as the set of pages referred in the last period T
 - T is a parameter of the WS algorithm
- For each page:
 - *bit R* (called “referred” or “use” bit) indicates whether the page had been referred in the last time tick
 - *TLR* an approximation of the time of last reference to the page
 - At the end of each time tick resets bit R for each page and updates the approximation of time of last reference
 - *age* of a page defined as the difference between current time and time of last reference

Working set algorithm

When it is executed (either at page fault or in a periodic daemon):

- For each page considers its values of R , TLR and age
 - If $R=0$: $age = current_time - TLR$
 - if $R=1$: $TLR = current_time$ ($age = 0$) and resets R
- The pages with $age < T$ (referred in the last period T) are in the working set and (if possible) are not removed

Working set algorithm

Current virtual time: 2204

Page table

age	R	...
2084	1	
2003	0	
1980	1	
1213	0	
2014	1	
2020	1	
1604	0	

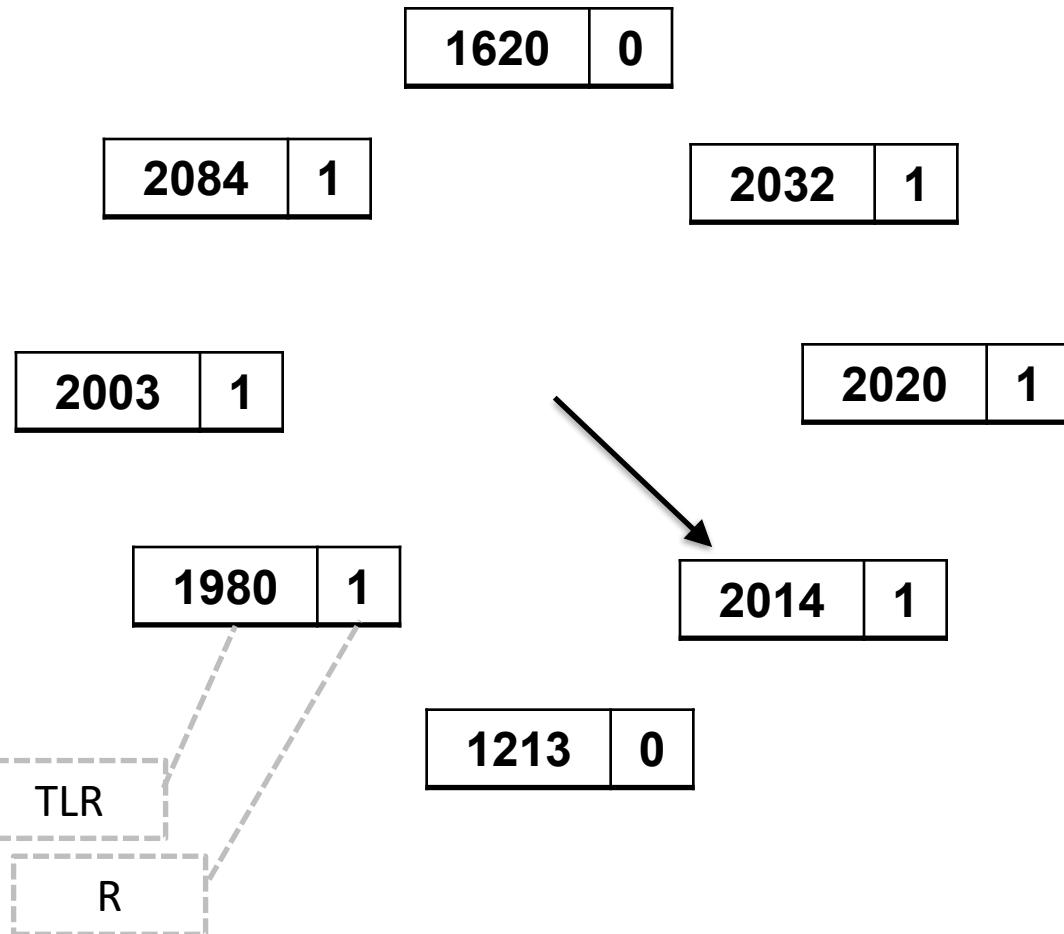
[age: current_time – TLR]

For each page: {
 if (R==0)
 age = current_time – TLR;
 else if (R==1) {
 TLR= current_time; R=0; age=0;
 }
 if ((age>T)
 removes the page
 }

if (age<=T for each page)
 removes the page with higher age

WSClock (working set clock)

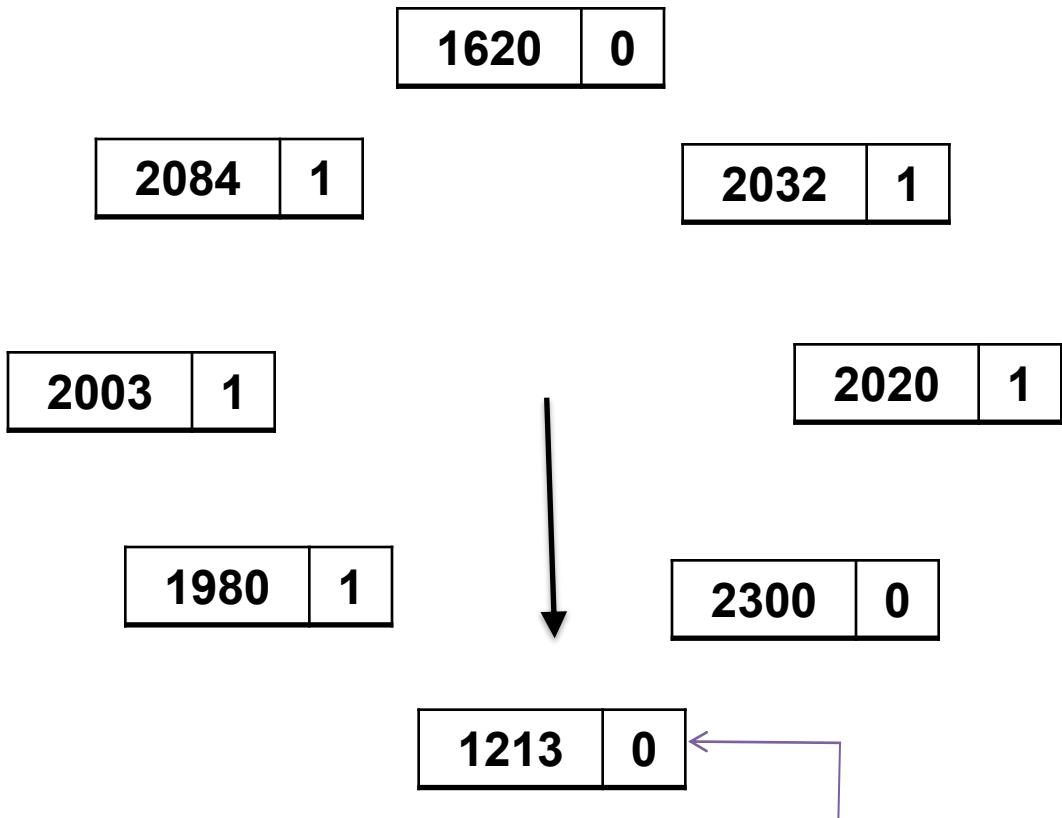
current_time = 2300; T=1000



- Considers only the pages in main memory
 - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
 - Better if not “dirty”
 - If selects a dirty page, the page is saved before its actual removal

WSClock (working set clock)

current_time = 2300; T=1000

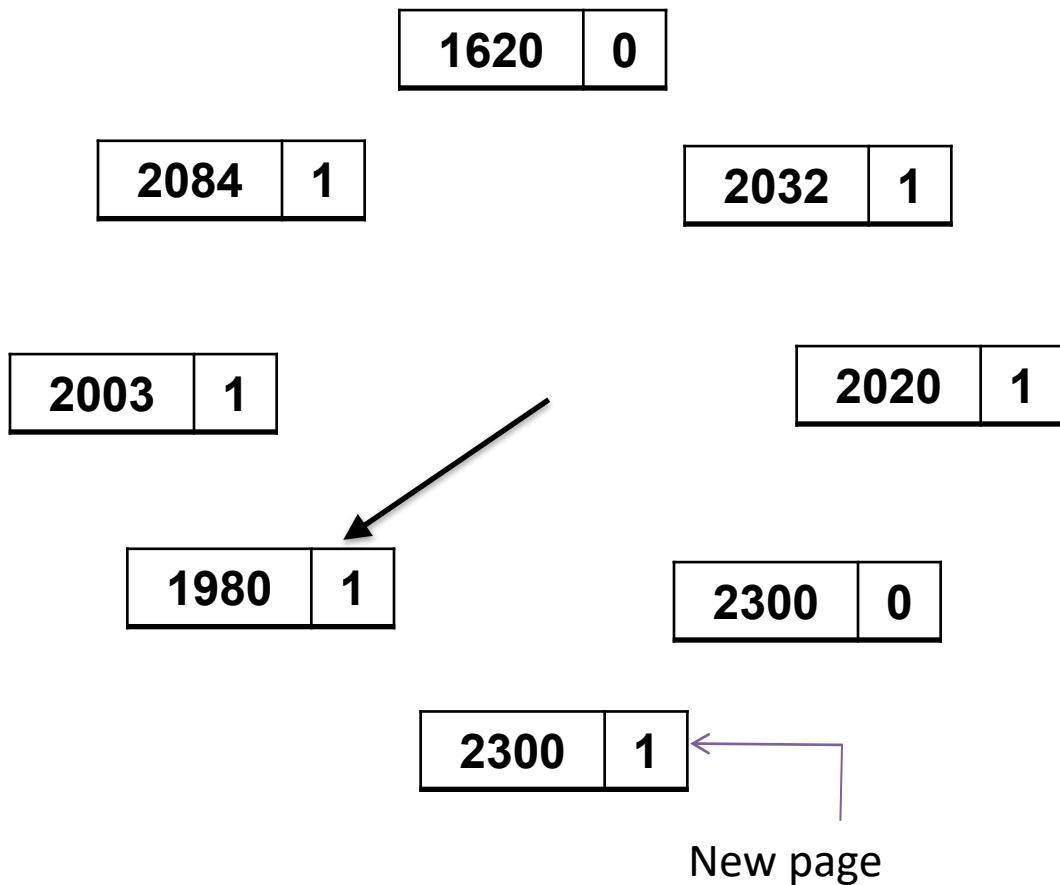


- Considers only the pages in main memory
 - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
 - Better if not “dirty”
 - If selects a dirty page, the page is saved before its actual removal

age > T : removes this page

WSClock (working set clock)

current_time = 2300; T=1000



- Considers only the pages in main memory
 - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
 - Better if not “dirty”
 - If selects a dirty page, the page is saved before its actual removal

Working set algorithm

- In practice, WS and all page replacement algorithms are executed in advance
- Guarantees free physical pages in case of page fault
 - To speed up the page fault
- Details in the case studies (Unix & Windows)

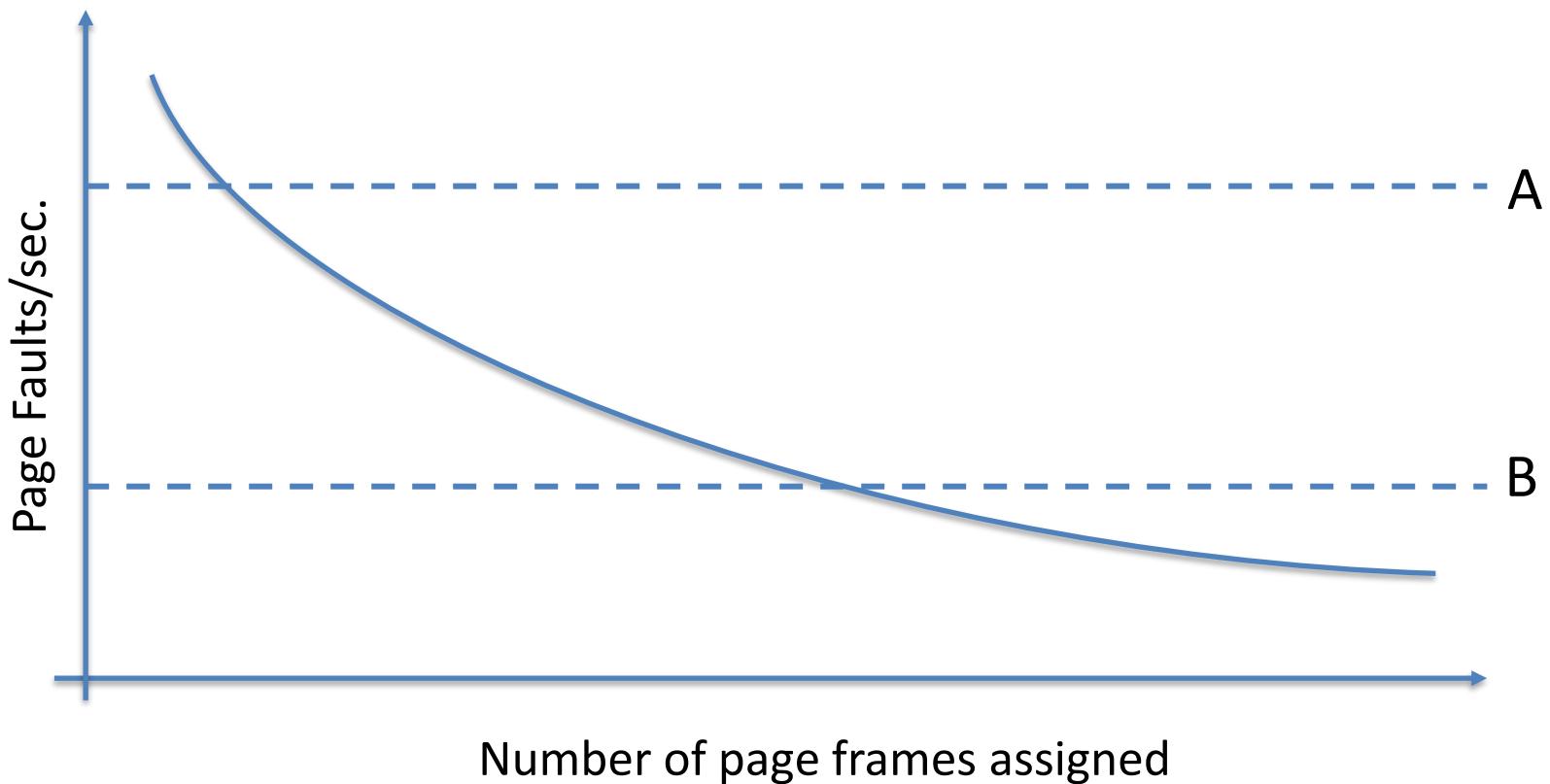
Paging and WS algorithm

- On demand paging:
 - Initially no page of the process is loaded in memory
 - Pages loaded by the process by generating page faults
 - Initially the number of page faults is high
 - When the working set had been loaded the number of page faults reduces
- Prepaging
 - A new process becomes ready when all its pages in the working set are loaded in main memory
 - Need to know (or to predict) what pages will be in the working set initially
 - not easy, can be done for some pages

Working set algorithm

- What should be the number of physical pages allocated to a process?
 - i.e. what should be the (maximum) size of the resident set of a process?
- Not easy to say in advance
- Page allocation algorithms
 - Static algorithms do not work
 - Dynamic algorithms: Page Fault Frequency (PFF)

Page Fault Frequency



Number of page faults per unit of time, depending on the number of physical pages assigned to the process

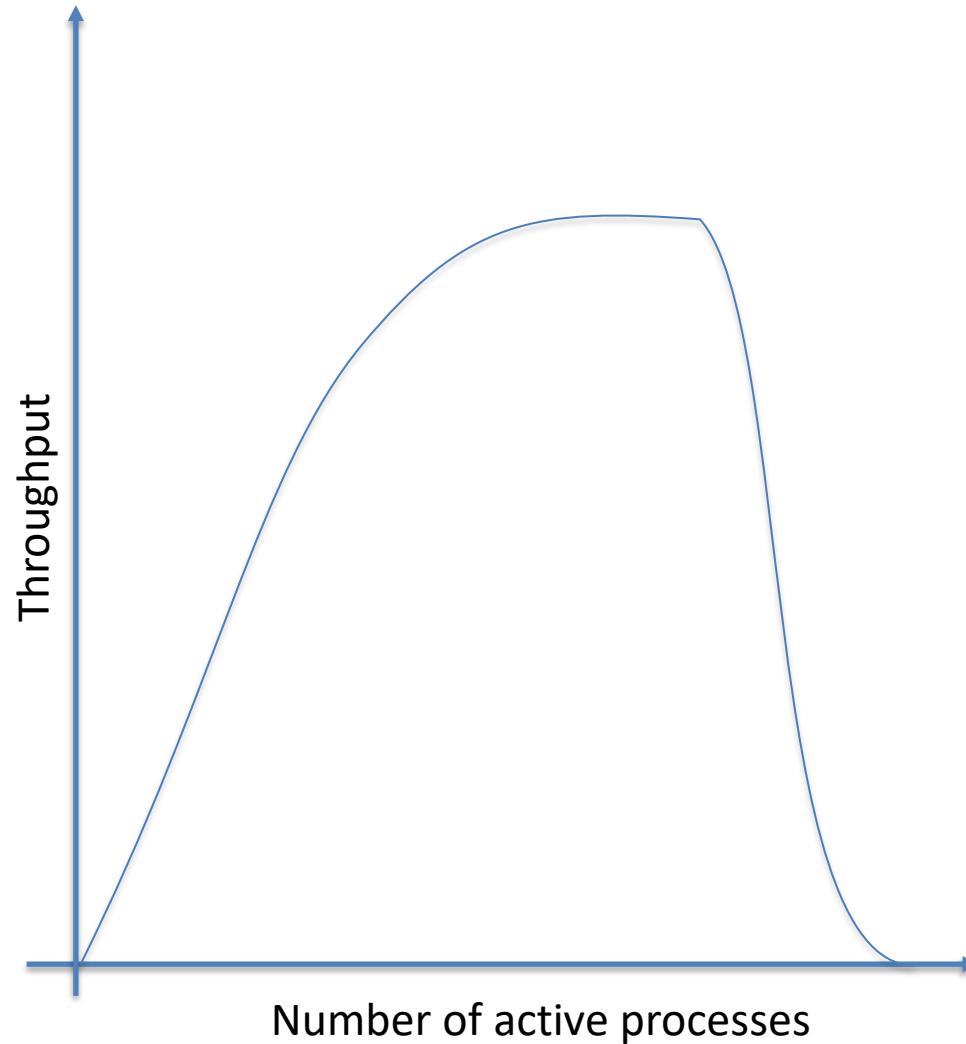
Page Fault Frequency

- Dynamically determines the number of physical pages assigned to a process
 - Guarantees that resident set \geq working set
- When frequency of page faults \gg «natural frequency»
 - Increases the size of the resident set
- When frequency of page faults \ll «natural frequency»
 - Reduces the size of the resident set

Question

- What happens to system performance as we increase the number of processes?
 - If the sum of the working sets becomes bigger than the physical memory?

Thrashing



Thrashing

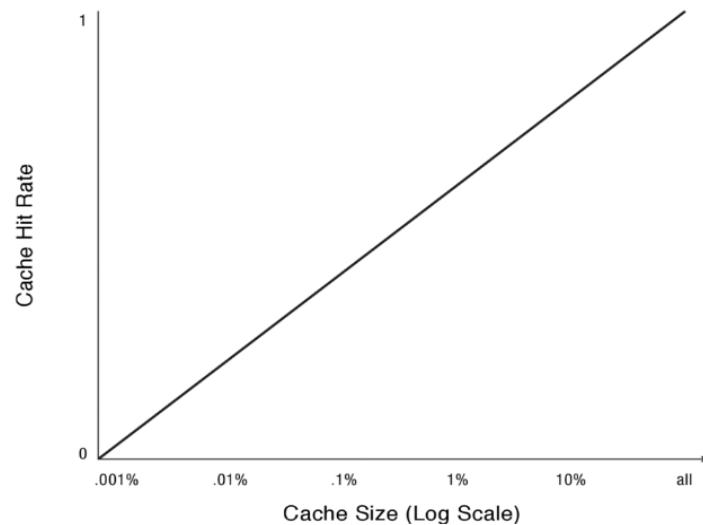
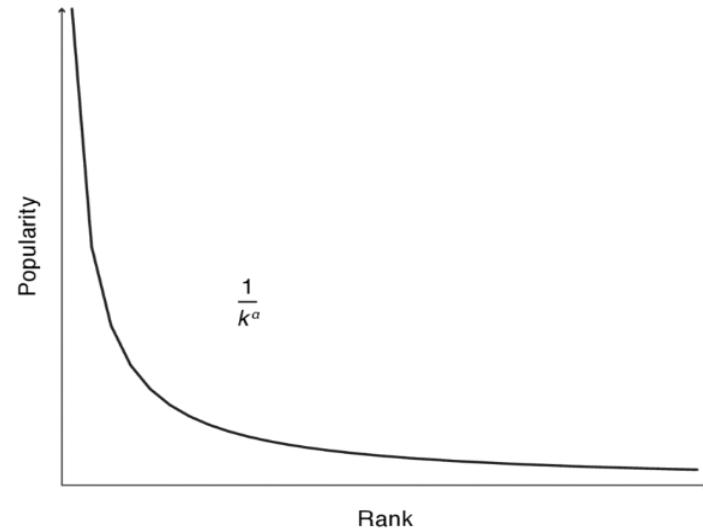
- When, from the PFF algorithm, comes out:
 - Some processes require more memory
 - No process requires less memory
- The number of page faults raises up
 - The system almost halts...
- Solution: reduce the number of processes in main memory
 - Reduces competition for memory (reduces the degree of multiprogramming)
 - Swaps out some process on disk

Where pages are stored

- Every process segment backed by a file on disk
 - Code segment -> code portion of executable
 - Data, heap, stack segments -> temp files
 - Shared libraries -> code file and temp data file
 - Memory-mapped files -> memory-mapped files
 - When process ends, delete temp files
- Provides the illusion of an infinite amount of memory to programs

Zipf Model

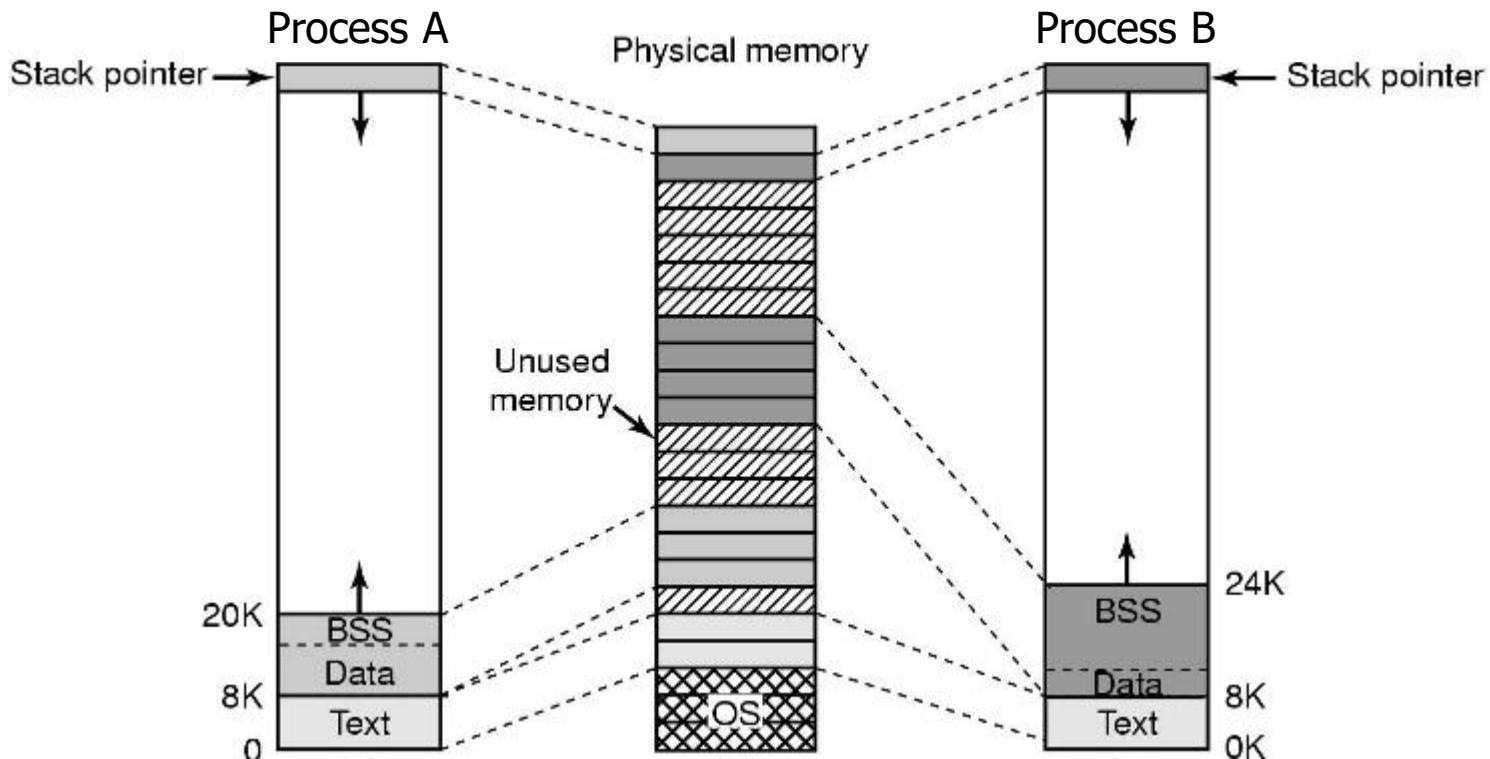
- Caching behavior of many systems are not well characterized by the working set model (e.g., web server)
- An alternative is the Zipf distribution model
 - Popularity $\sim \frac{1}{k^c}$ for k-th most popular item ($1 < c < 2$)
 - Examples: web pages, movies, words in text, etc..



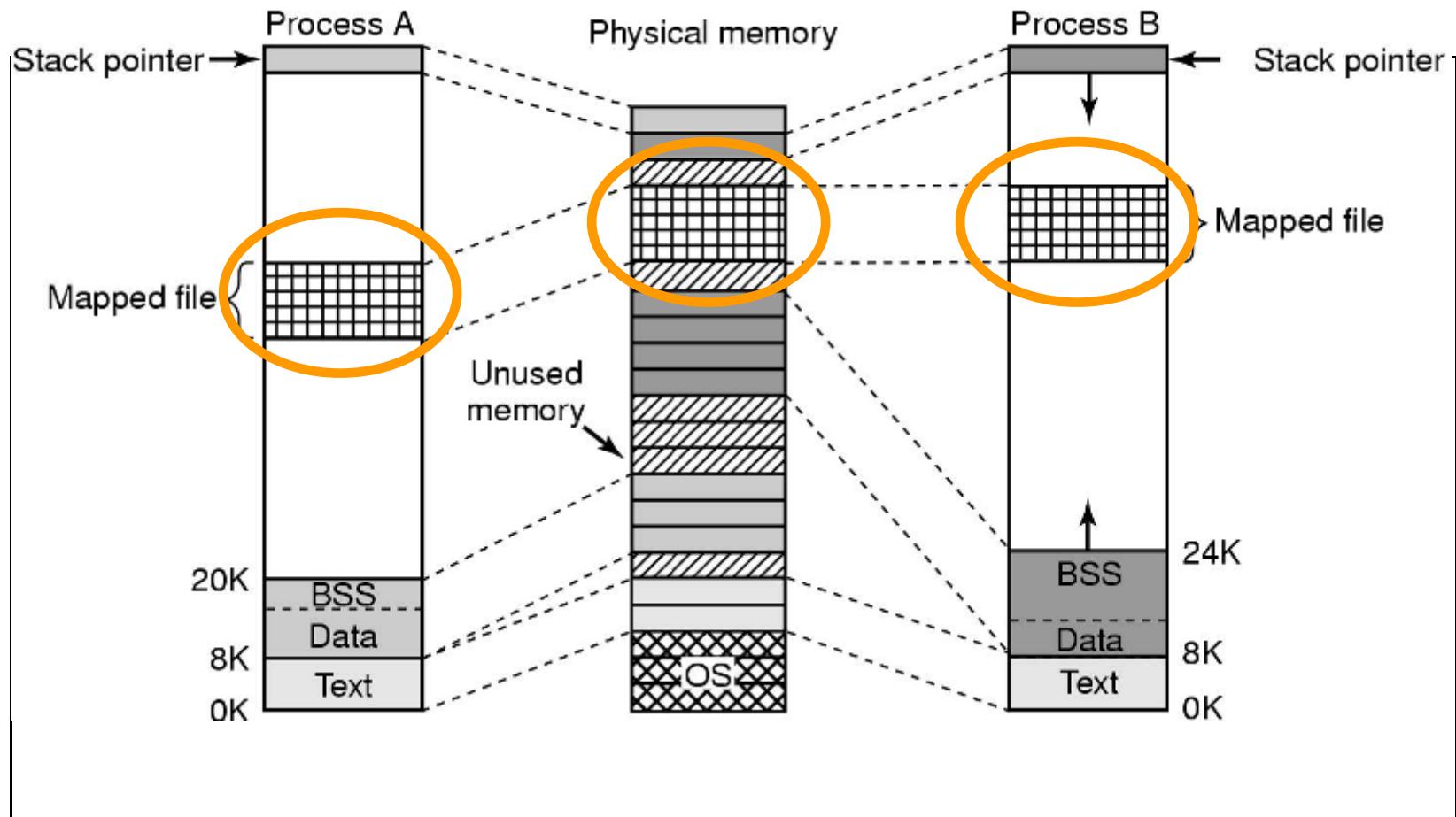
Memory management in Unix

- Since BSD v.3:
 - Paged segmentation
 - **Virtual memory** based on on-demand paging
- On demand paging:
 - ***Core map***:
 - kernel data structure that contains the allocation of the physical blocks
 - Used in case of page fault
 - **Page replacement algorithm**: Second Chance

Unix: memory organization



Unix: sharing memory mapped files



Models for Application File I/O

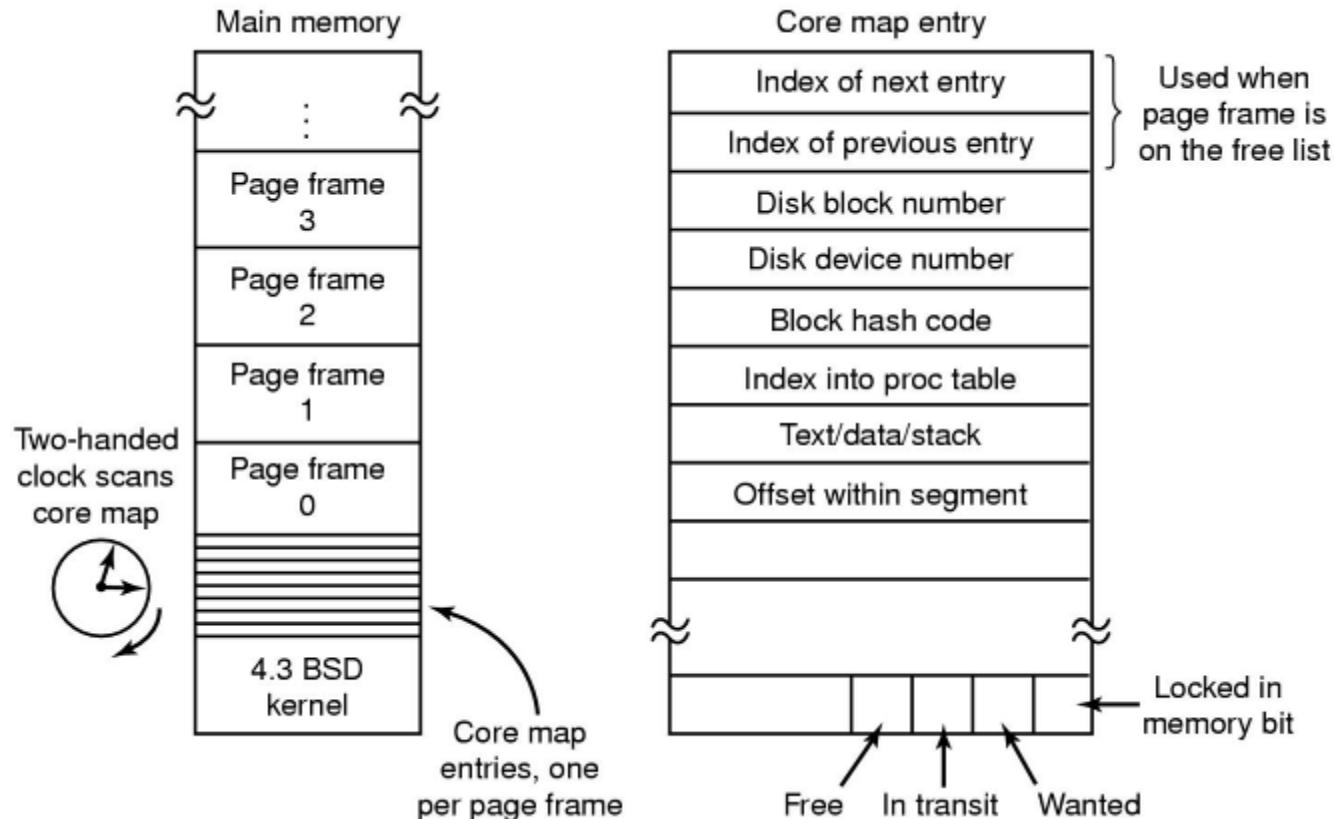
- Explicit read/write system calls
 - Data copied to user process using system call
 - Application operates on data
 - Data copied back to kernel using system call
- Memory-mapped files
 - Open file as a memory segment
 - Program uses load/store instructions on segment memory, implicitly operating on the file
 - Page fault if portion of file is not yet in memory
 - Kernel brings missing blocks into memory, restarts process

Advantages to Memory-mapped Files

- Programming simplicity, especially for large file
 - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
 - Data brought from disk directly into page frame
- Pipelining
 - Process can start working before all the pages are populated
- Interprocess communication
 - Shared memory segment vs. temporary file

Paging in Unix BSD

with an inverse page table (core map):



Page replacement in Unix (BSD) - I

Page replacement algorithm:

- Second chance (global)
- or variants (e.g. the two-handed clock algorithm)

Page replacement executed periodically by the *Page Daemon*:

- Uses parameters: *lotsfree*, *desfree*, *minfree*, with:

$$\textit{lotsfree} > \textit{desfree} > \textit{minfree}$$

Page replacement in Unix (BSD) - II

PageDaemon algorithm (sketch):

- *if (#freeblocks \geq lotsfree) return //no operation required*
- *if (minfree \leq #freeblocks $<$ lotsfree) or
(#freeblocks $<$ minfree **and**
Average[#freeblocks, Δt] \geq desfree))*
 replace pages until #freeblocks = lotsfree + k (with k>0)
- *if ((#freeblocks $<$ minfree **and**
Average[#freeblocks, Δt] $<$ desfree))*
 swapout processes

Page replacement in Unix (BSD) - III

Relationship with the working set theory:

- If $\#freeblocks < minfree$:
 - High number of page faults from the last execution of *Page Daemon*
 - There exist processes that have $WS \not\subset RS$ (not included) that cause thrashing
 - RS : resident set, i.e. set of pages resident in memory
 - WS : working set
- If $\text{average}[\#freeblocks, \Delta t] < desfree$:
 - The problem has been there for a while
 - The swapout of some processes will free resources and avoid thrashing
 - The processes with $WS \not\subset RS$ will expand their resident set, as consequence of their future page faults

Swapping of processes in Unix

Swapout:

```
if (#freeblocks < minfree) and  
  (Average[#freeblocks, Δt] < desfree)  
  // The average is computed over a given time frame Δt
```

The *Page Daemon* selects «victim» processes based on :

- Priority
- Elapsed time without being executed
- Amount of memory required
- ...

Swaps out one or more victim process
until $\#freeblocks \geq lotsfree + k$ (with $k>0$)

Swapping of processes in Unix

Swapin:

If the number of free blocks is large enough,
the *Page Daemon* selects one or more processes
based on:

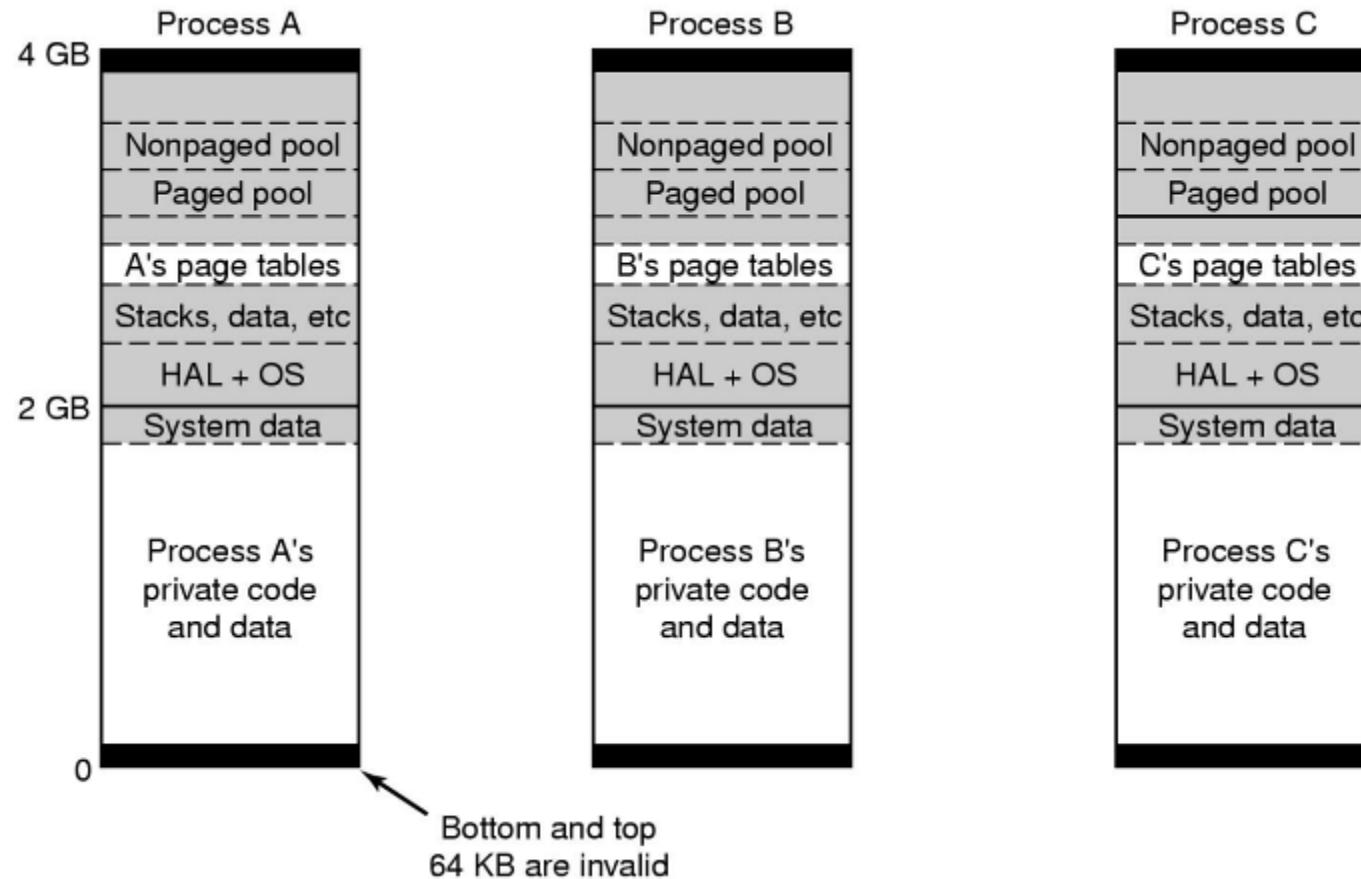
- Time spent in swapped-out state
- Amount of memory required
- ...

Swapin of one or more processes provided
 $\#freeblocks \geq lotsfree + k$ (with $k > 0$)

Gestione della memoria in Windows

- Dimensione della memoria virtuale: 4 Gbyte (indirizzo virtuale di 32 bit).
- Memoria virtuale paginata (paginazione a domanda) con pagine di dimensioni fisse (le dimensioni della pagina dipendono dalla particolare macchina fisica).
- Spazio virtuale suddiviso in due sottospazi di 2 Gbyte ciascuno
 - il sottospazio virtuale inferiore è privato di ogni processo
 - il sottospazio virtuale superiore è condiviso tra tutti i processi e mappa il sistema operativo.

Struttura della memoria virtuale in Windows



Le aree bianche sono private; le aree scure sono condivise

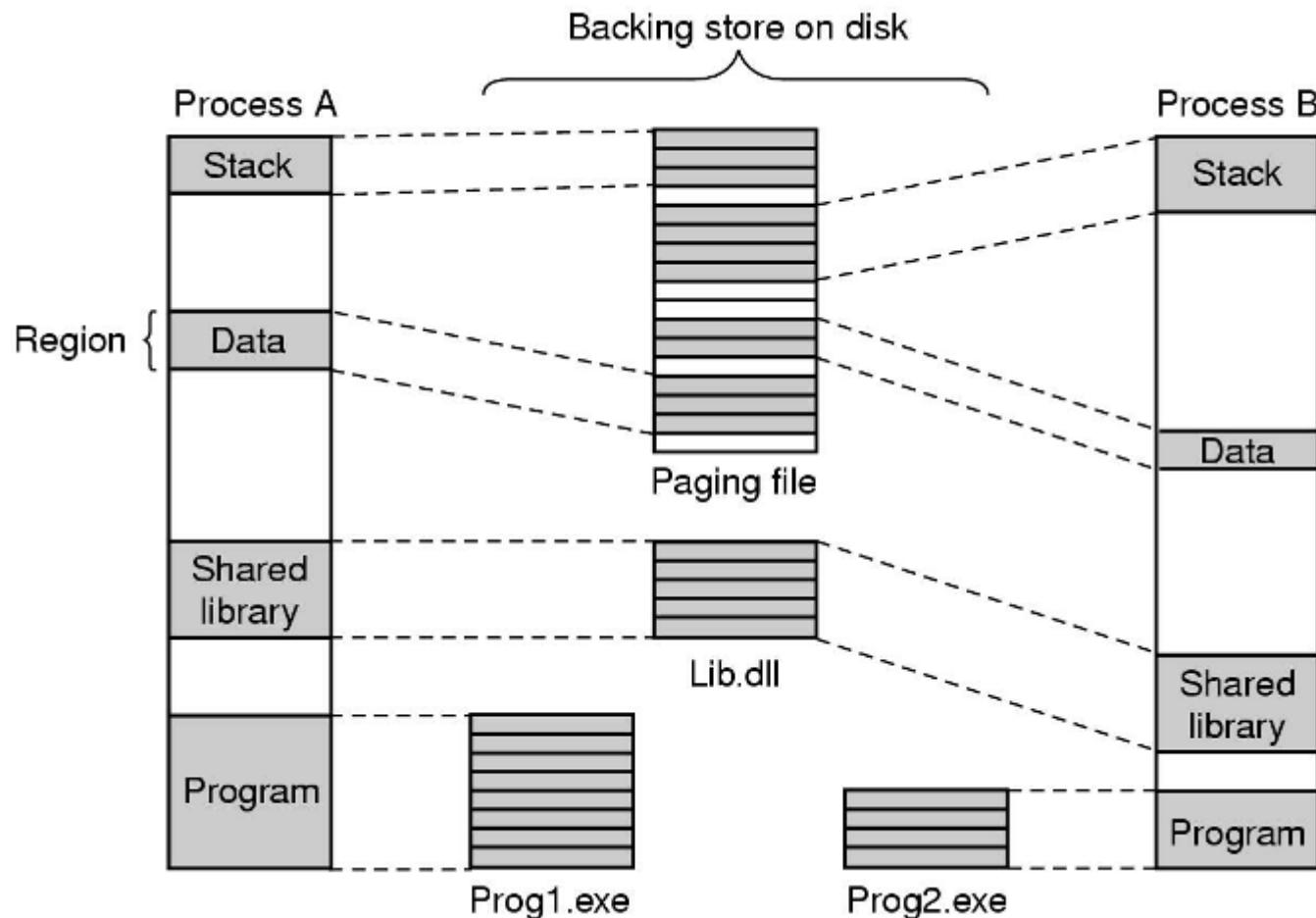
Memoria Virtuale in Windows

Spazio virtuale unico, suddiviso in *regioni*

Ogni pagina logica può essere :

- *free* : se non è assegnata a nessuna regione
 - un accesso a una pagina free determina page fault non gestibile
- *reserved* : è una pagina non ancora in uso ma che è stata riservata per espandere una regione
 - ==> non mappata nella tabella delle pagine
 - esempio: riservata per l'espansione dello stack
 - non utilizzabile per mappare nuove regioni
 - un accesso a una pagina reserved determina page fault gestibile
- *committed* : se appartiene a una regione già mappata nella tabella delle pagine
 - un accesso a una pagina *committed* non presente in memoria risulta in un page fault, *che determina il caricamento della pagina solo se questa non si trova in una lista di pagine eliminata dal working set*

Windows: backing store



Windows: page fault management (1)

- Windows adopts a working set algorithm
 - Local algorithm
 - Operates on the set of *resident pages* (*RS*)
 - For a given process, $x = \#RS$ ranges in $[min, max]$
 - *min* and *max* are initially set to default values...
 - ... but they vary during the life of a process, to adapt to its memory needs

Windows: page fault management (2)

- At page fault of process P , the requested page is always loaded in a free block in memory
- Hence the size of the resident set of P increases ($x=x+1$)
- If $x \geq max$ the pages of P in excess will be removed by the working set manager

Windows: page fault management (3)

Windows ensures there are always free blocks in memory by means of two processes:

- balance set manager
 - Executes periodically
 - If free memory is scarce: evicts pages identified by the working set manager
- working set manager
 - Implements the page replacement policy

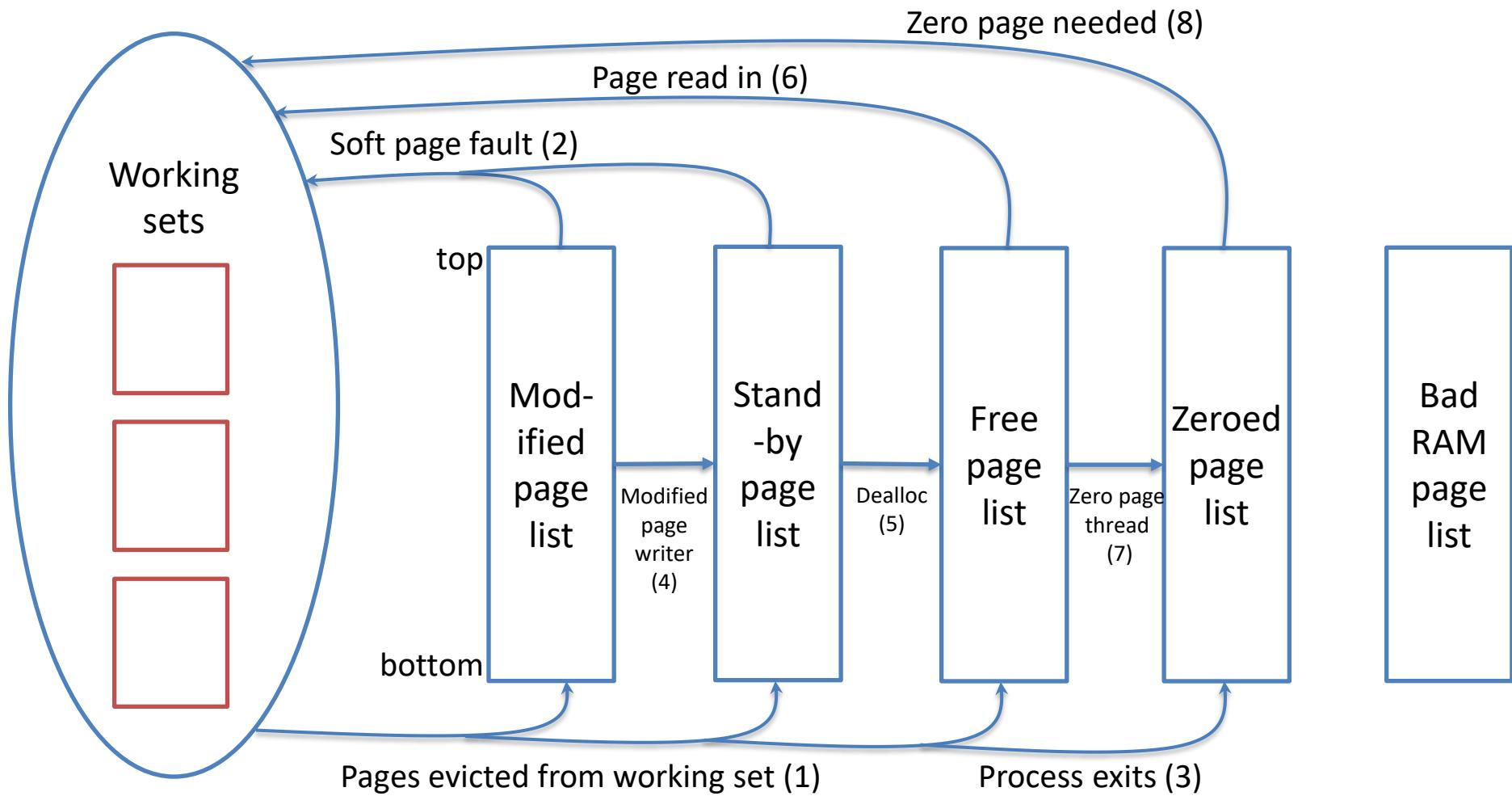
Windows: working set manager

Adopts a working set page replacement policy

- For each process with $x > max$:
 - For each page p with *reference bit* $R = 1$: resets R and $count(p)$
 - For each page p with $R = 0$: increase $count(p)$
 - $count(p)$ is an approximation of the time of last reference (past distance) of a page
 - Set for removal $x - max$ pages in decreasing order of $count(p)$
- If the number of free blocks is still low: remove pages also of processes with $x > min$

Windows: management of pages

lists of pages and transitions



Storage Systems

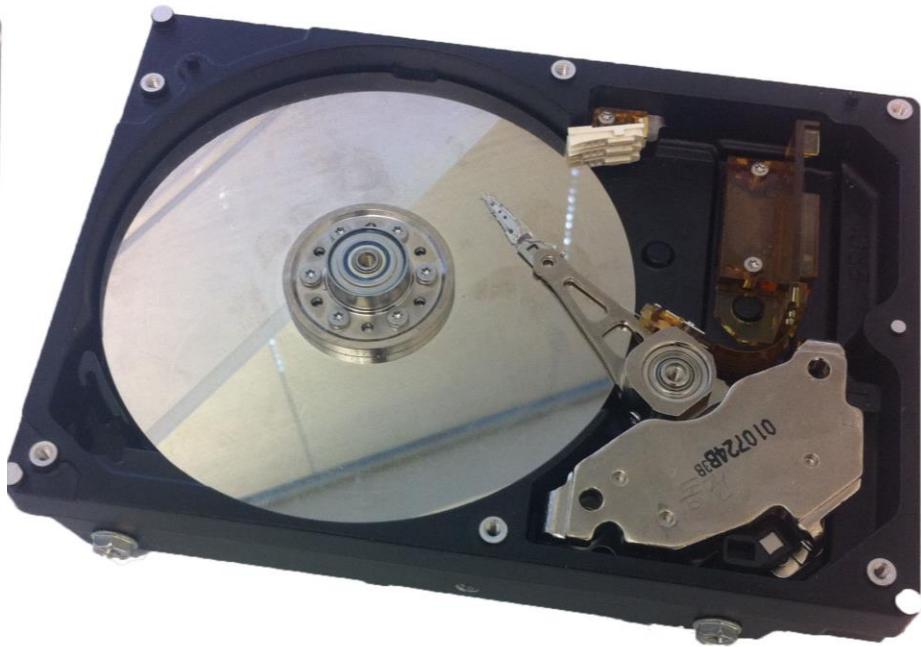
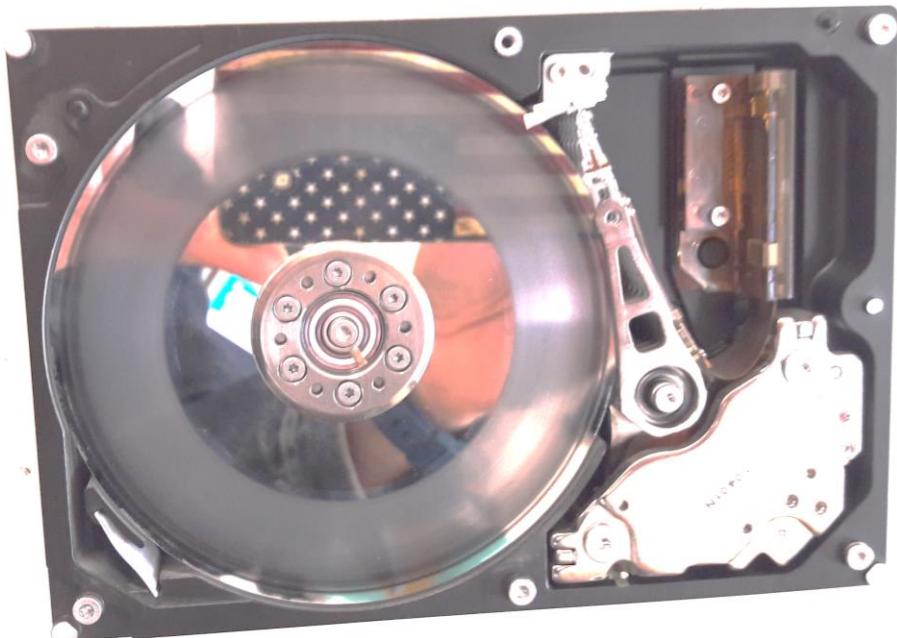
Main Points

- Storage hardware characteristics
 - Magnetic disks
 - Flash memory
 - RAID storage brief overview

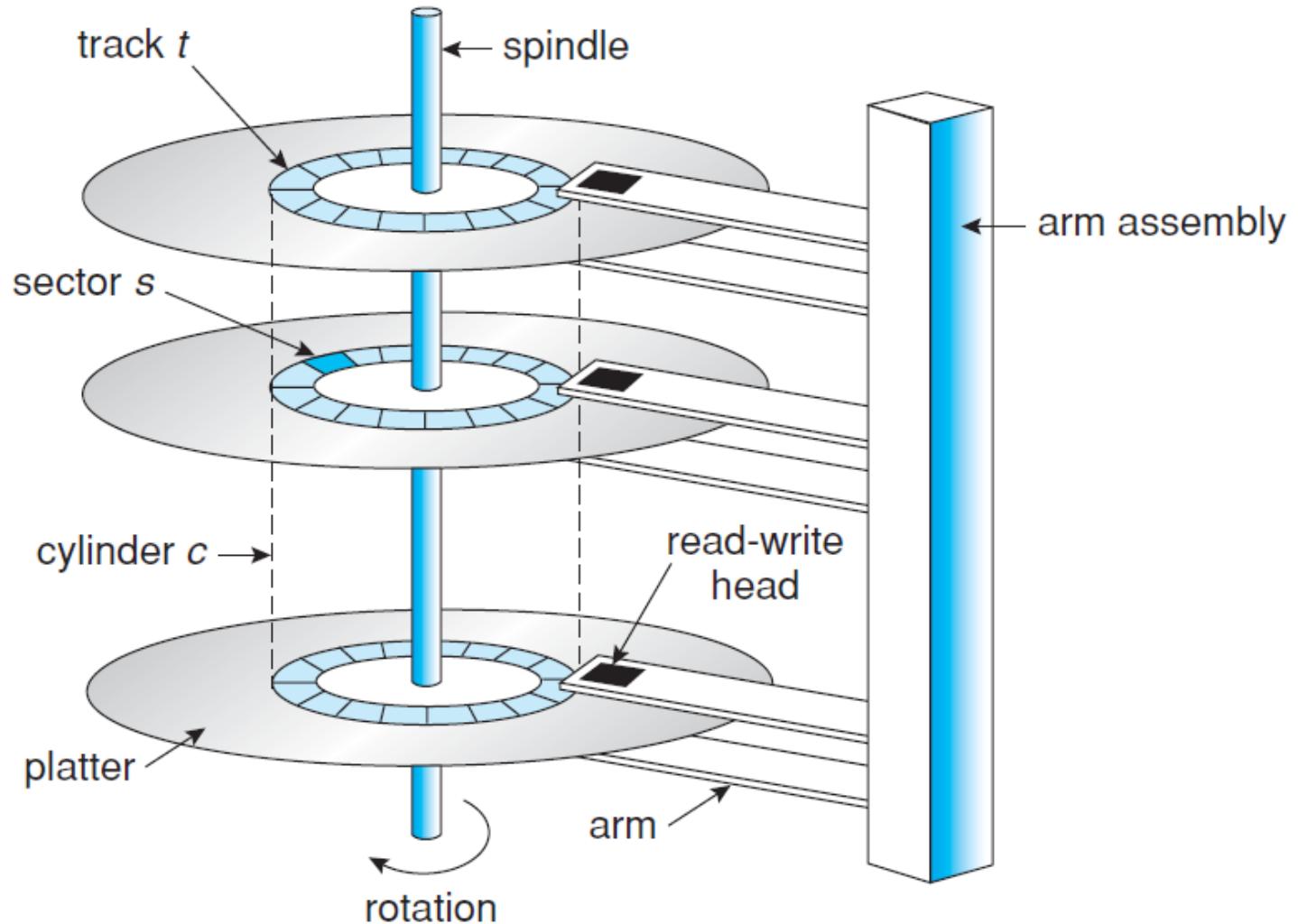
Storage Devices

- Magnetic disks
 - Storage that rarely becomes corrupted
 - Large capacity at low cost
 - Block level random access
 - Slow performance for random access
 - Better performance for streaming access
- Flash memory
 - Storage that rarely becomes corrupted
 - Good capacity at intermediate cost ($2 \times$ disk)
 - Block level random access
 - Good performance for reads; worse for random writes

Magnetic Disk



Magnetic Disk Components



Disk Tracks

- ~ 1 micron wide
 - Wavelength of light is ~ 0.5 micron
 - Resolution of human eye: 50 microns
 - 100K on a typical 2.5" disk
- Separated by unused guard regions
 - Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)
- Track length varies across disk
 - Outside: More sectors per track, higher bandwidth
 - Disk is organized into regions of tracks with same # of sectors/track
 - Only outer half of radius is used
 - Most of the disk area in the outer regions of the disk

Sectors

Sectors contain sophisticated error correcting codes

- Disk head magnet has a field wider than track
- Hide corruptions due to neighboring track writes
- Sector sparing
 - Remap bad sectors transparently to spare sectors on the same surface
- Slip sparing
 - Remap all sectors (when there is a bad sector) to preserve sequential behavior
- Track skewing
 - Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops

Sectors & blocks

- **At low level** the controller accesses individual sectors
 - Typical sector size is 256/512 bytes
 - Identified by a triple: <#cylinder, #face, #sector>
- **At the higher level**, the disk driver groups a *set of contiguous sectors in a block*
 - Typical block size is 2/4/8 Kbytes
 - identified by a pointer on a contiguous address space (from 0 to max-blocks)

Sectors & blocks

Given a sector number b , and a triple $\langle c, f, s \rangle$:

$$b = c(\#faces \cdot \#sectors) + f(\#sectors) + s$$

- * assumption 1 block corresponds to 1 sector
- $\#faces$ is the number of faces in the disk
- $\#sectors$ is the number of sectors per track

Consequently:

$$c = b \text{ div } (\#faces \cdot \#sectors)$$

$$f = (b \text{ mod } (\#faces \cdot \#sectors)) \text{ div } \#sectors$$

$$s = (b \text{ mod } (\#faces \cdot \#sectors)) \text{ mod } \#sectors$$

Disk Performance

Disk Latency = Seek Time + Rotation Time + Transfer Time

Seek Time: time to move disk arm over track (1-20ms)

Fine-grained position adjustment necessary for head to “settle”

Head switch time ~ track switch time (on modern disks)

Rotation Time: time to wait for disk to rotate under disk head

Disk rotation: 4 – 15ms (depending on price of disk)

Transfer Time: time to transfer data onto/off of disk

Disk head transfer rate: 50-100MB/s (5-10 usec/sector)

Host transfer rate dependent on I/O connector (USB, SATA, ...)

Disk Information (2008)

Size	
Platters/heads	2/4
Capacity	320 GB
Performance	
Spindle speed	7200 RPM
Average seek time read/write	10.5 ms / 12.0 ms
Maximum seek time	19 ms
Track-to-track seek time	1 ms
Transfer rate (surface to buffer)	54-128 MB/s
Transfer rate (buffer to host)	375 MB/s
Buffer memory	16 MB
Power	
Typical	16.35 W
Idle	11,68 W

Question

- How long to complete 500 random disk reads, in FIFO order?

Question

- How long to complete 500 random disk reads, in FIFO order?
 - Seek: average 10.5 msec
 - Rotation: average 4.15 msec
 - Transfer: 5-10 usec
- $500 * (10.5 + 4.15 + 0.01)/1000 = 7.3$ seconds

Question

- How long to complete 500 sequential disk reads?

Question

- How long to complete 500 sequential disk reads?
 - Seek Time: 10.5 ms (to reach first sector)
 - Rotation Time: 4.15 ms (to reach first sector)
 - Transfer Time: (outer track)
 $500 \text{ sectors} * 512 \text{ bytes} / 128\text{MB/sec} = 2\text{ms}$

Total: $10.5 + 4.15 + 2 = 16.7 \text{ ms}$

Might need an extra head or track switch (+1ms)

Track buffer may allow some sectors to be read off disk out of order (-2ms)

Disk Scheduling

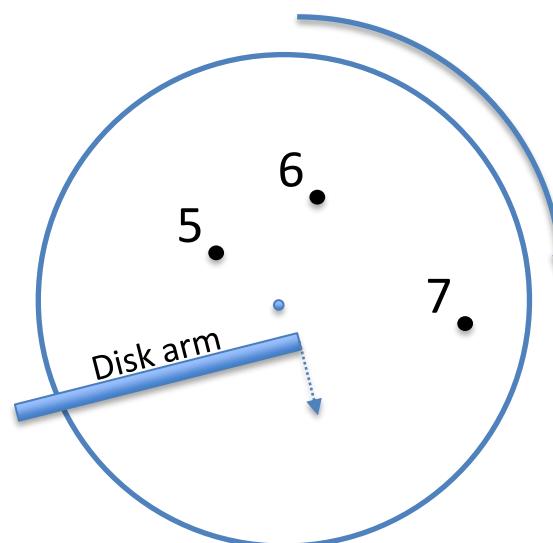
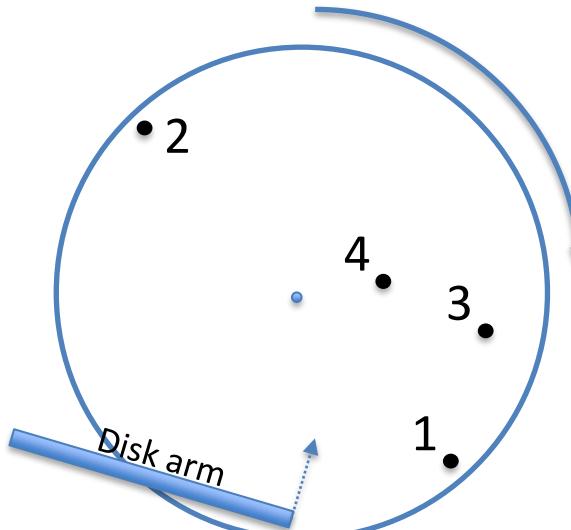
- FIFO (aka First-Come First-Served – FCFS)
 - Schedule disk operations in order they arrive
 - Downsides?

Disk Scheduling

- Shortest Seek Time First
 - Not optimal in response time
 - ... suppose two clusters of requests at far end of disk
 - Downsides?

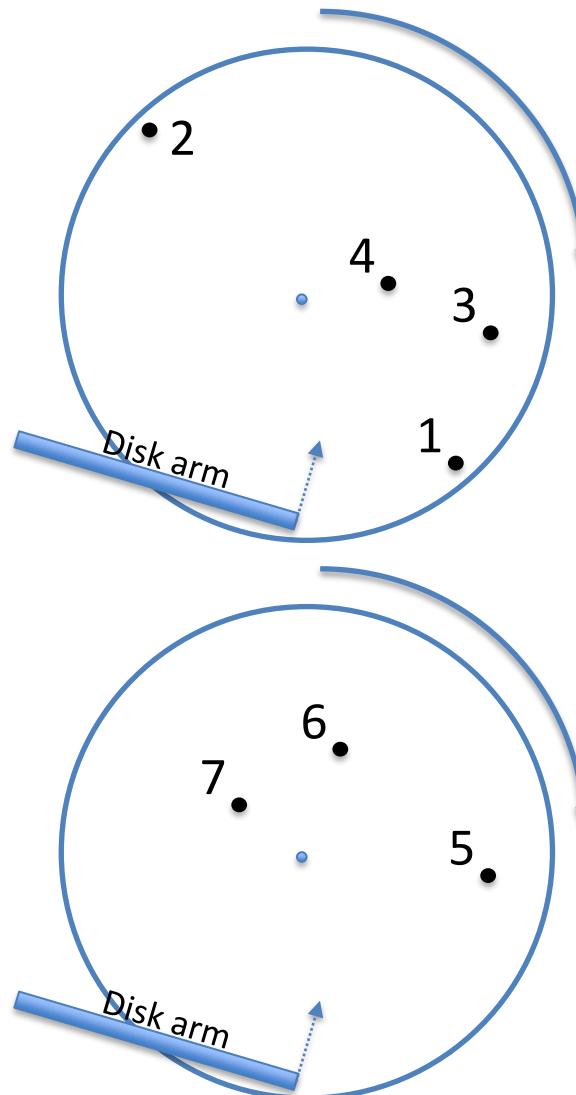
Disk Scheduling

- SCAN: move disk arm in one direction, until all requests satisfied, then reverse direction



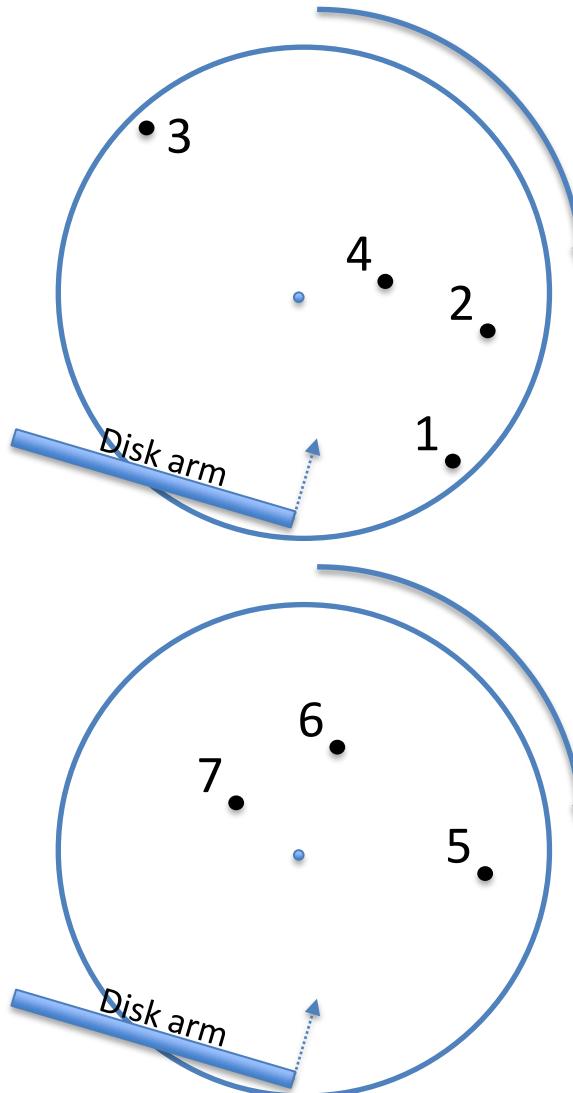
Disk Scheduling

- CSCAN: move disk arm in one direction, until all requests satisfied, then start again from farthest request



Disk Scheduling

- R-CSCAN: CSCAN but take into account that short track switch is < rotational delay



Question

- How long to complete 500 random disk reads, in any order?

Question

- How long to complete 500 random disk reads, in any order?
 - Disk seek: 1ms (most will be short)
 - Rotation: 4.15ms
 - Transfer: 5-10usec
- Total: $500 * (1 + 4.15 + 0.01) = 2.2$ seconds
 - Would be a bit shorter with R-CSCAN
 - vs. 7.3 seconds if FIFO order

Question

- How long to read all of the bytes off of a disk?

Question

- How long to read all of the bytes off of a disk?
 - Disk capacity: 320GB
 - Disk bandwidth: 54-128MB/s
- Transfer time =

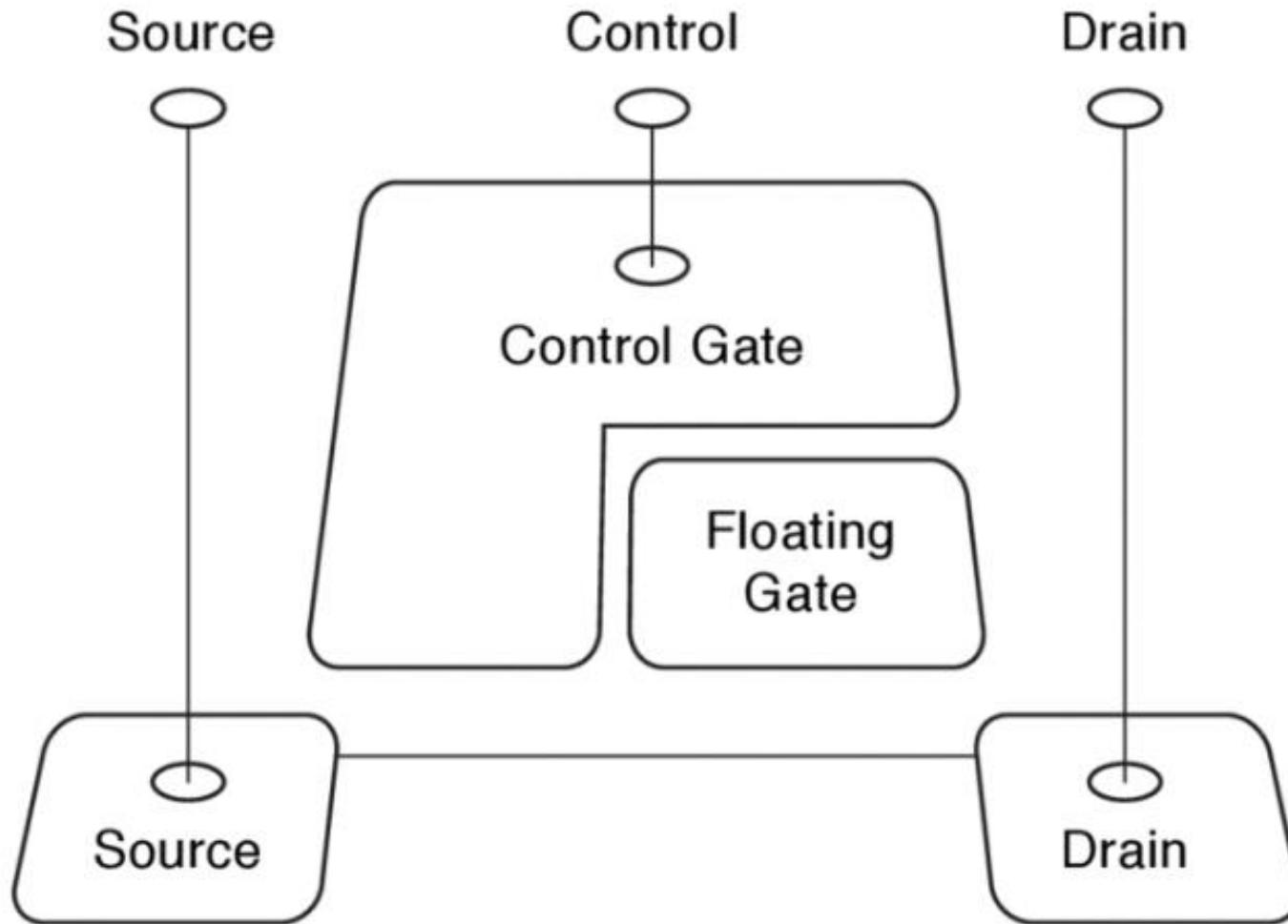
Disk capacity / average disk bandwidth

~ 3500 seconds (1 hour)

SSD Disk

- A Solid State Drive (SSD) is a non-volatile storage device based on flash-memory technology
- SSDs are more reliable and faster than hard disks (HDs) because they have no moving parts and no seek time
 - SSDs commonly use a simple FCFS scheduling policy.
- They consume less power and are more expensive per MB of data. The capacity of HD is usually larger.
- In some systems, SSDs are used as an additional cache tier in the memory hierarchy

Flash Memory in a nutshell



Flash Memory

- Writes must be to “clean” cells; no update in place
 - Large block erasure required before write
 - Erasure block: 128 – 512 KB
 - Erasure time: Several milliseconds
- Write/read page (2-4KB)
 - 50-100 usec

SSD disk information (2011)

Size	
Capacity	300 GB
Page size	4 KB
Performance	
Bandwidth (sequential reads)	270 MB/s
Bandwidth (sequential writes)	210 MB/s
Read/write latency	75 μ s
Random reads per second	38,500
Random writes per second	2,000 (2,400 with 20% space reserve)
interface	SATA 3GB/s
Endurance	
Endurance	1.1 PB (1.5 PB with 20% space reserve)
Power	
Power consumption(active/idle)	3.7 W / 0.7 W

Question

- Why are random writes so slow?
 - Random write: 2000/sec
 - Random read: 38500/sec

Flash Translation Layer

- To avoid the cost of an erase for each write, pages are erased in advance
 - Clean pages always available for a new write
- This means that a write cannot be directed to an arbitrary page in the memory, but only to one previously erased page
- What happens if you rewrite a block of a file?
 - The page storing the block cannot be rewritten immediately...
 - It need to be erased first but with surrounding pages!
 - A clean page is used to apply the write, but this page is somewhere in the disk...
 - The old page goes to garbage for recycling
- How to know where the pages of my file are?

Flash Translation Layer

- Flash device firmware maps logical page # to a physical location
 - Move live pages as needed for erasure
 - Garbage collect empty erasure block by copying live pages to new location
 - Wear-levelling
 - Can only write each physical page a limited number of times
 - Avoid pages that no longer work
- Transparent to the device user

File System – Flash

- File systems on magnetic disks do not need to tell the disk what blocks are in use:
 - When a block is no longer used it is marked free in the bitmap
 - The file system reuse it whenever it wants
- When these FS were used first on flash drives the performances decayed dramatically over time

File System – Flash

- The Flash Translation Layer got busy on garbage collection:
 - Live blocks must be remapped to a new location ...
 - ... to compact the free pages in order to proceed with block erasure
- This even with a large amount of free space
 - For example, if the FS move a large file from a range of blocks to another one ...
 - ... the storage does not know that the old blocks can go to garbage ...
 - ... unless the FS tells it!

File System – Flash

- TRIM command
 - File system tells device when pages are no longer in use
 - Helps the File Translation Layer to optimize garbage collection
 - Introduced in between 2009/2011 in most OSs

Dischi RAID

Redundant Array of Independent Disks

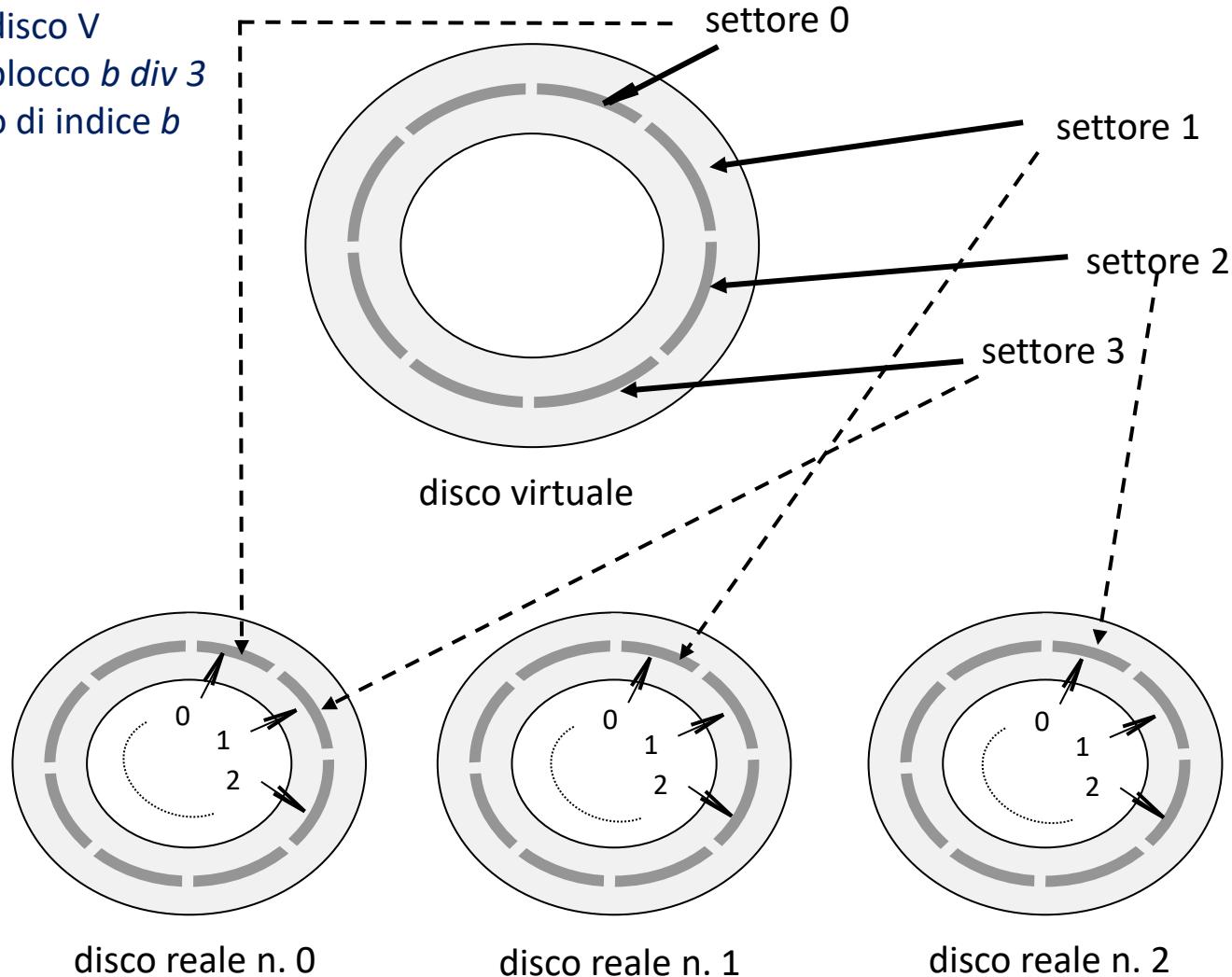
Architettura RAID:

- Realizza un *disco virtuale* di capacità superiore a quella dei singoli dischi
 - l'interfaccia è quella di un unico disco
- Sfrutta il parallelismo per ottenere un accesso più veloce
 - i blocchi consecutivi di uno stesso file sono distribuiti sui dischi dell'array in modo da permettere operazioni contemporanee
- Sfrutta la ridondanza per accrescere l'affidabilità
 - la ridondanza permette di correggere gli errori di certe classi

Diversi livelli di architettura RAID, con diversi livelli di ridondanza

Dischi RAID

Blocco b del disco V
mappato nel blocco $b \text{ div } 3$
del disco fisico di indice $b \text{ mod } 3$.



Dischi RAID

Livello 0: Dischi asincroni, nessuna ridondanza (*striping*)

- Si possono effettuare contemporaneamente operazioni indipendenti
- Anche detto JBOD (just a bunch of disks)

Livello 1: Dischi asincroni, disco con copie ridondanti (*mirror*)

- Si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 2: Dischi sincroni, i dischi ridondanti contengono codici per la correzione degli errori

- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 3: Dischi sincroni, un solo disco ridondante

- contiene la parità del contenuto degli altri dischi
- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Dischi RAID

Livello 4: Dischi asincroni; 1 disco ridondante

- contiene la parità del contenuto degli altri dischi
- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori
- Il disco ridondante è sovraccarico nei piccoli aggiornamenti

Livello 5: Come livello precedente, ma parità distribuita tra tutti i dischi

- permette un miglior bilanciamento del carico tra i dischi
- minimo 3 dischi

Livello 6: Come livello precedente, ma parità doppia distribuita tra tutti i dischi

- minimo 4 dischi, permette di tollerare fino al fallimento di 2 dischi

Livello 10: Dischi asincroni, Mirror di stripes

Livello 01: Dischi asincroni, Stripe di mirror

Dischi RAID

Dischi **asincroni**:

- vengono distribuite le *stripes* (singoli settori o sequenze di settori contigui)

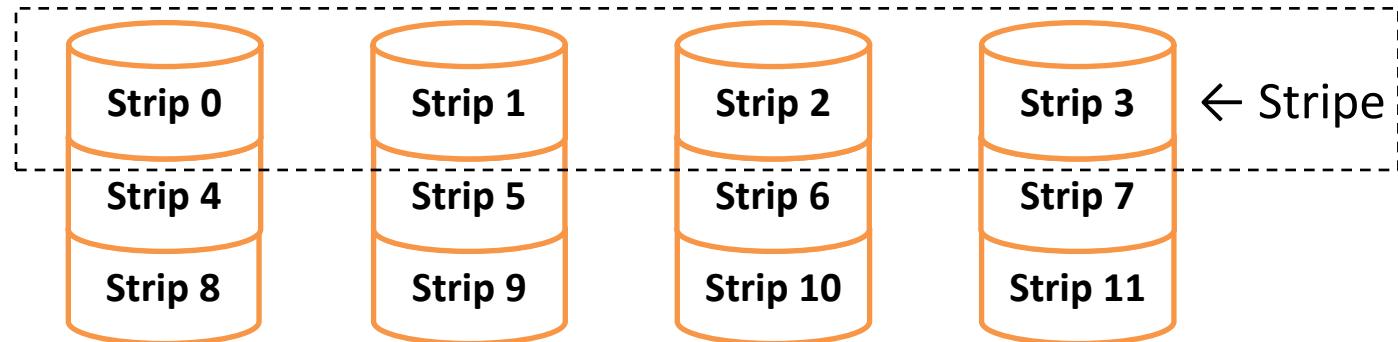
Livello 0: Nessuna ridondanza

- possibilità di eseguire contemporaneamente operazioni indipendenti

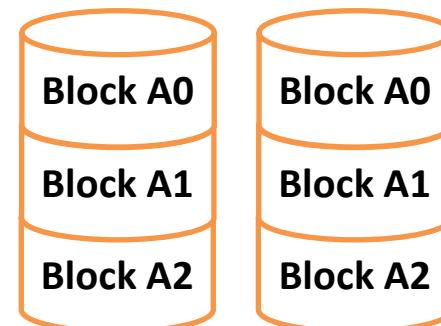
Livello 1: Disco con copia ridondante (mirror). No striping.

- Esecuzione contemporanea di operazioni indipendenti e correzione di *crash faults*
singoli

Raid level 0



Raid level 1



Dischi RAID

Dischi **asincroni**, vengono distribuite le *stripe*

Livello 4: Ridondanza con *strip* di parità

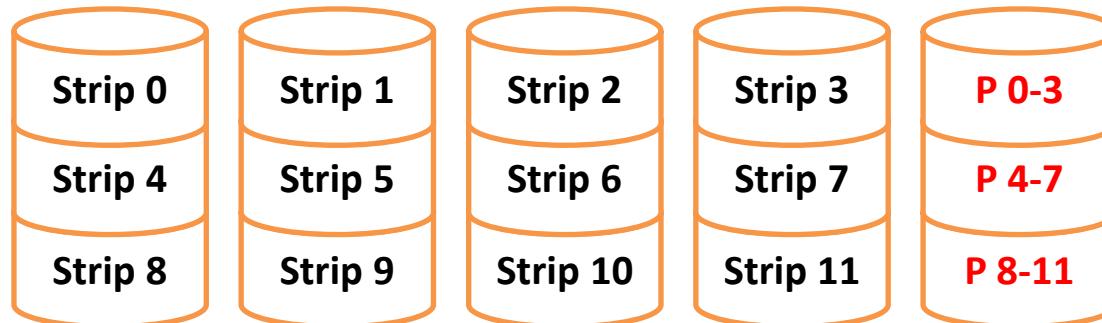
Esempio: disco 4 contiene le strip di parità delle strips omologhe dei dischi 0, 1, 2, 3

- esecuzione contemporanea di operazioni indipendenti e *correzione di crash faults singoli*

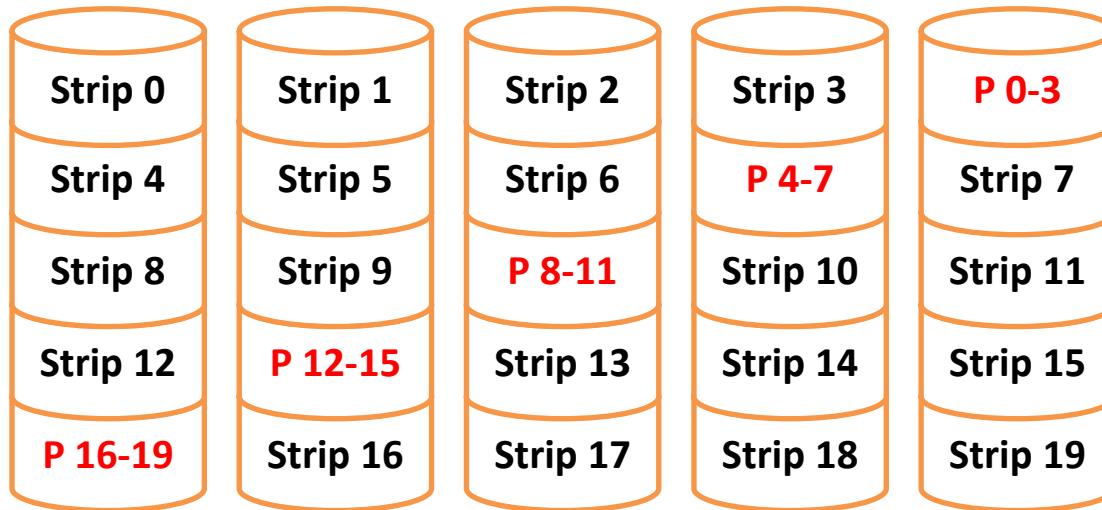
Livello 5: Come livello 4, ma strip di parità distribuite nei vari dischi

- migliore bilanciamento del carico tra i dischi

Raid level 4



Raid level 5



Dischi RAID

Dischi asincroni, organizzazione in cluster

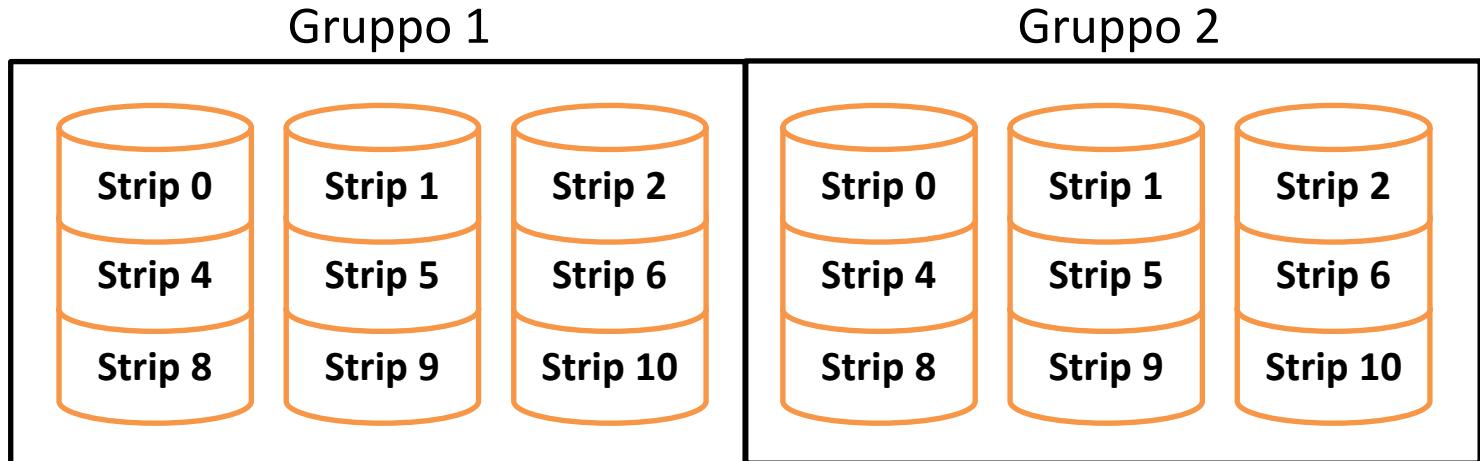
Livello 0+1 (o 01): Mirror di stripes

- Organizzato in gruppi: ogni gruppo è un mirror di un RAID 0.

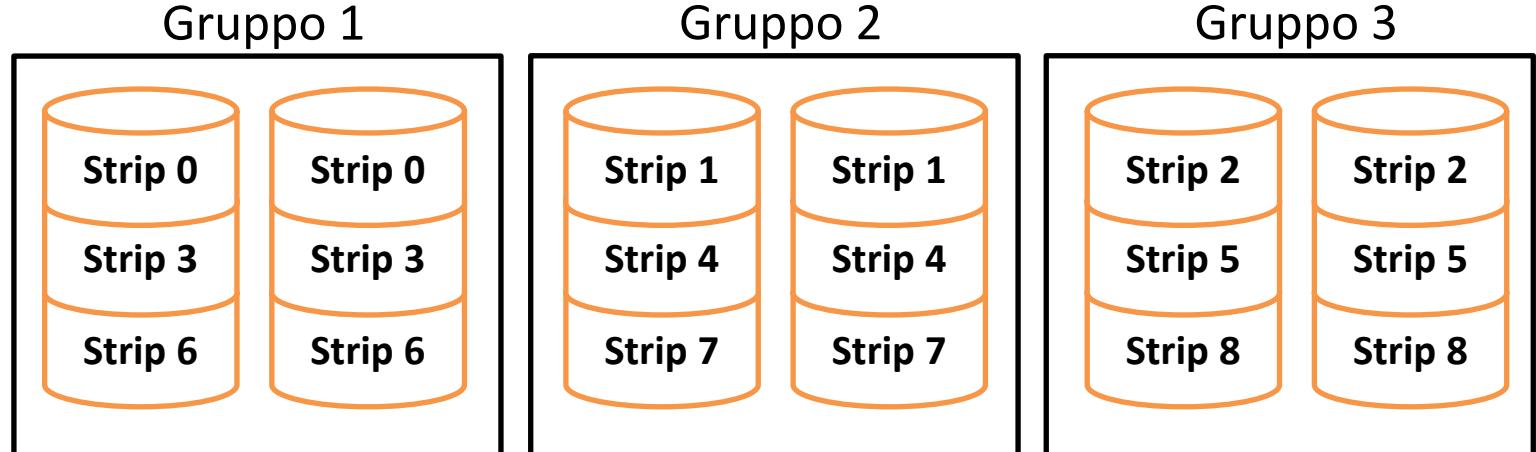
Livello 1+0 (o 10): Stripe di mirror

- Organizzato in gruppi, ogni gruppo è un RAID 1. I gruppi realizzano un RAID 0
- Miglior tolleranza ai guasti del 01: se si guastano due dischi in gruppi diversi il sistema può ancora funzionare (i dischi in un gruppo non sono indipendenti)

Raid level 01



Raid level 10



File Systems

Main Points

- File system:
 - Useful abstractions on top of physical devices
 - Usage patterns
 - File layout
 - Directory layout

File Systems

- Abstraction on top of persistent storage
- Devices provide
 - Storage that (usually) survives across machine crashes
 - Block level (random) access
 - Large capacity at low cost
 - Relatively slow performance
 - Magnetic disk read takes 10-20M processor instructions

File System as Illusionist: Hide Limitations of Physical Storage

- Persistence of data stored in file system:
 - Even if crash happens during an update
 - Even if disk block becomes corrupted
 - Even if flash memory wears out
- Naming:
 - Named data instead of disk block numbers
 - Directories instead of flat storage
 - Byte addressable data even though devices are block-oriented
- Performance:
 - Cached data
 - Data placement and data structure organization
- Controlled access to shared data

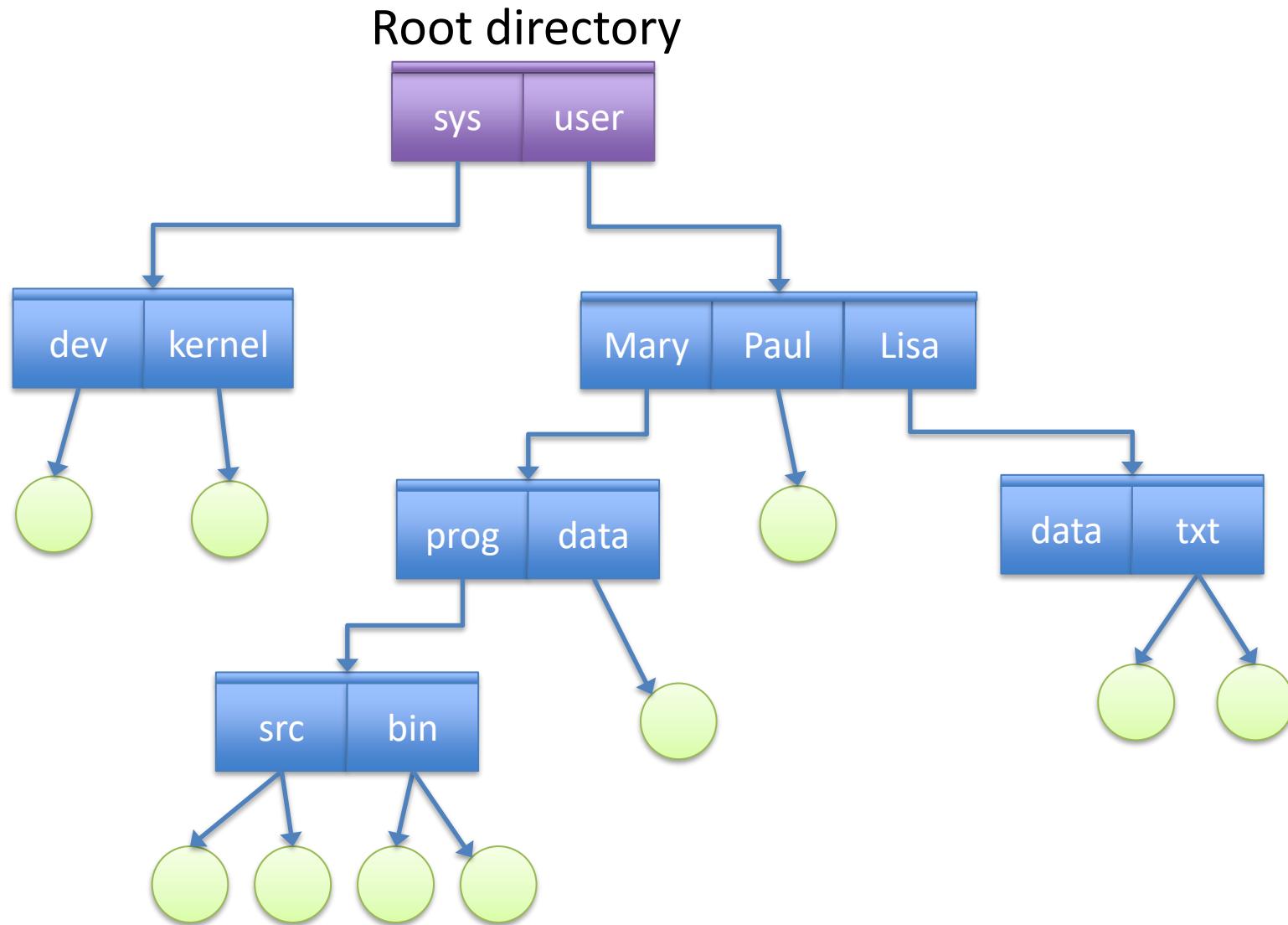
File System Abstraction

- File system
 - Hierarchical organization (directories, subdirectories)
 - Access control on data
 - File: named collection of data
 - Linear sequence of bytes (or a set of sequences)
 - Read/write or memory mapped
 - Crash and storage error tolerance
 - Operating system crashes (and disk errors) leave file system in a valid state
 - Performance
 - Achieve close to the hardware limit* in the average case
- (*) considering both the device and the memory subsystem

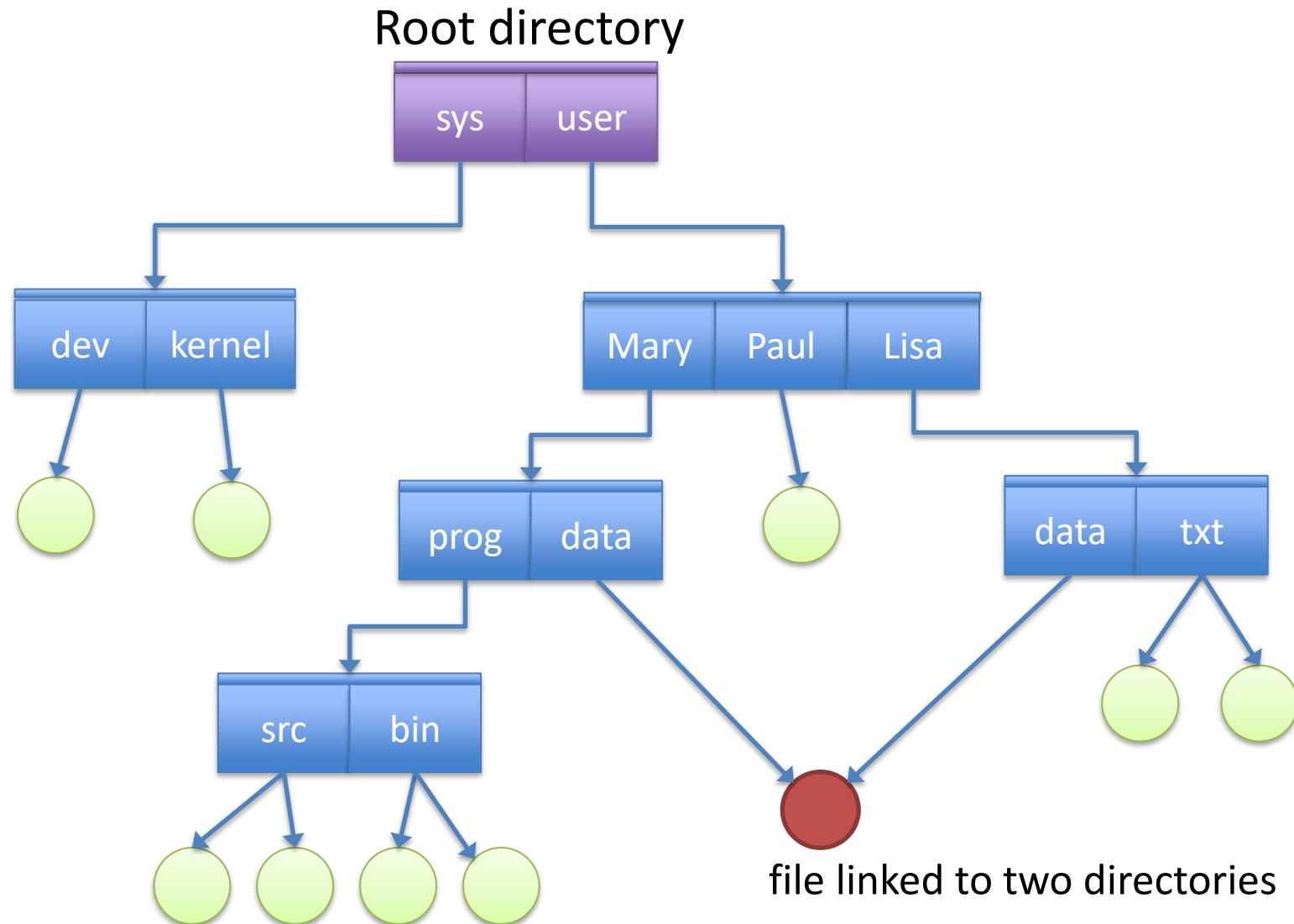
File System Abstraction

- Directory
 - Group of named files or subdirectories
 - Mapping from file name to file metadata location
- Path
 - String that uniquely identifies file or directory
 - Ex: /cse/www/education/courses/cse451/12au
- Links
 - Hard link: link from name to metadata location
 - Soft link: link from name to alternate name
- Mount
 - Mapping from name in one file system to root of another

File System: tree structure



File System: acyclic graph structure

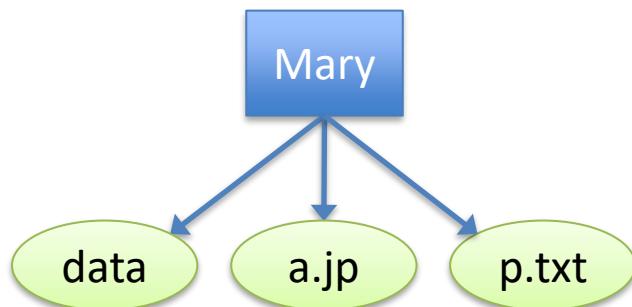


Data structures: Directories

- Each file (and directory) belongs to a directory;
- Each directory is a data structure that links file names to file attributes
 - Example of attributes: file size, address on disk, access rights, time of last access, time of creation, ...

A possible implementation (FAT file systems):

- The directory is a table, it associates each file name to its file descriptor (which includes all attributes of the file)



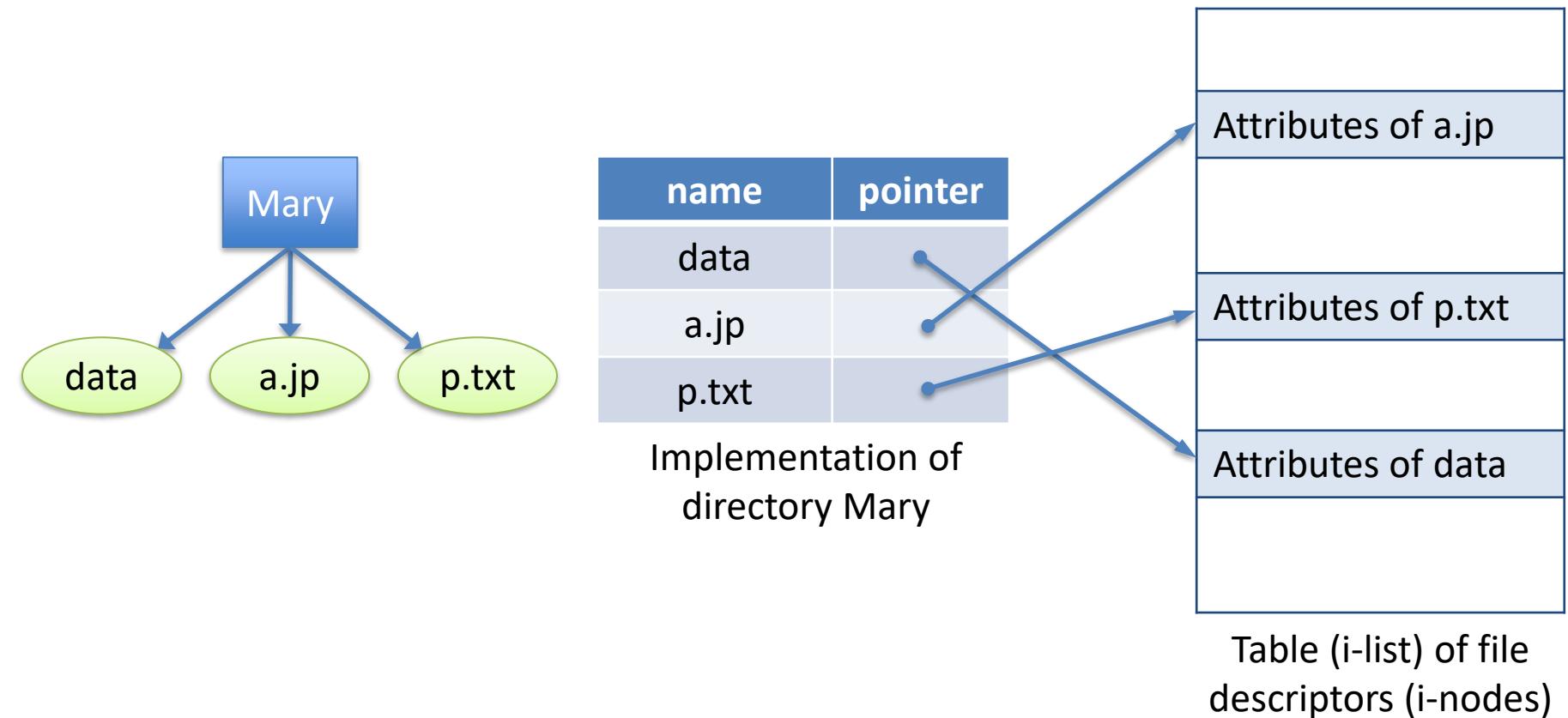
name	descriptor
data	Attribute: type, address, etc.
a.jp	Attribute: type, address, etc.
p.txt	Attribute: type, address, etc.

Implementation of directory Mary

Data structures: Directories

Implementation of directories in Unix:

- The directory is a table that includes the references to the file descriptors (i-nodes), which are stored in a separate data structure in the disk



Access to files

- File access operations:

- Read a logical record from a file.
- Write logical records on a file

To execute these operations on a file it is necessary to retrieve the attributes of the file:

- The addresses on disk of the logical records
- Access rights
- etc.....

Too expensive to read these data from the disk at each access:

- they are read once for all before accessing the file, by means of the **open** SC
- the **close** SC is then necessary to deallocate such information from memory

Access methods

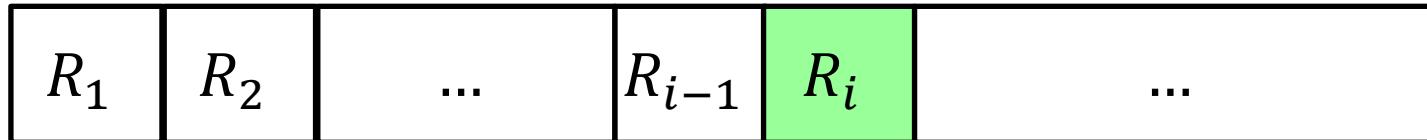
- File access can be:
 - Sequential
 - Direct

The access method:

- independent of the physical device
- independent of the allocation method of the files on the device

Sequential access

- The file is a sequence of logical records $[R_1, R_2, \dots, R_N]$:
 - To access each logical record R_i it is necessary to access first all the previous $(i - 1)$ records:



- Access operations:
 - *readnext*: read next logical record
 - *writenext*: write next logical record
 - Each operation (read/write) positions an access pointer on the next logical record

Direct Access

- The file is a set $\{R_1, R_2, \dots, R_N\}$ of logical records
 - The user can directly access each logical record
 - Access operations:
 - $read(f, i, \&V)$: read logical record i of file f ; the read data is stored in buffer V ;
 - $write(f, i, \&V)$: write the content of buffer V on the logical record i of file f .

UNIX File System API

- `create, link, unlink, mkdir, rmdir`
 - Create file, link to file, remove link
 - Create directory, remove directory
- `open, close, read, write, seek (mmap, munmap)`
 - Open/close a file for reading/writing
 - Seek resets current position
- `fsync`
 - File modifications can be cached
 - `fsync` forces modifications to disk (like a memory barrier)

File System Interface

- UNIX file open is a Swiss Army knife:
 - Open the file, return file descriptor
 - Options:
 - if file doesn't exist, return an error
 - If file doesn't exist, create file and open it
 - If file does exist, return an error
 - If file does exist, open file
 - If file exists but isn't empty, nix it then open
 - If file exists but isn't empty, return an error
 - ...

Interface Design Question

- Why not separate syscalls for open/create/exists?
 - Would be more modular!
 - But rather boring as well...

```
if (!exists(name))
    create(name); // can create fail?
fd = open(name); // does the file exist?
```

File System Workload

- File sizes
 - Are most files small or large?
 - Which accounts for more total storage: small or large files?

File System Workload

- File sizes
 - Are most files small or large?
 - SMALL
 - Which accounts for more total storage: small or large files?
 - LARGE

File System Workload

- File access
 - Are most accesses to small or large files?
 - Which accounts for more total I/O bytes: small or large files?

File System Workload

- File access
 - Are most accesses to small or large files?
 - SMALL
 - Which accounts for more total I/O bytes: small or large files?
 - LARGE

File System Workload

- How are files used?
 - Most files are read/written sequentially
 - Ex. Image files, executables
 - Some files are read/written randomly
 - Ex: database files, swap files
 - Some files have a pre-defined size at creation
 - Some files start small and grow over time
 - Ex: program stdout, system logs

File System Design

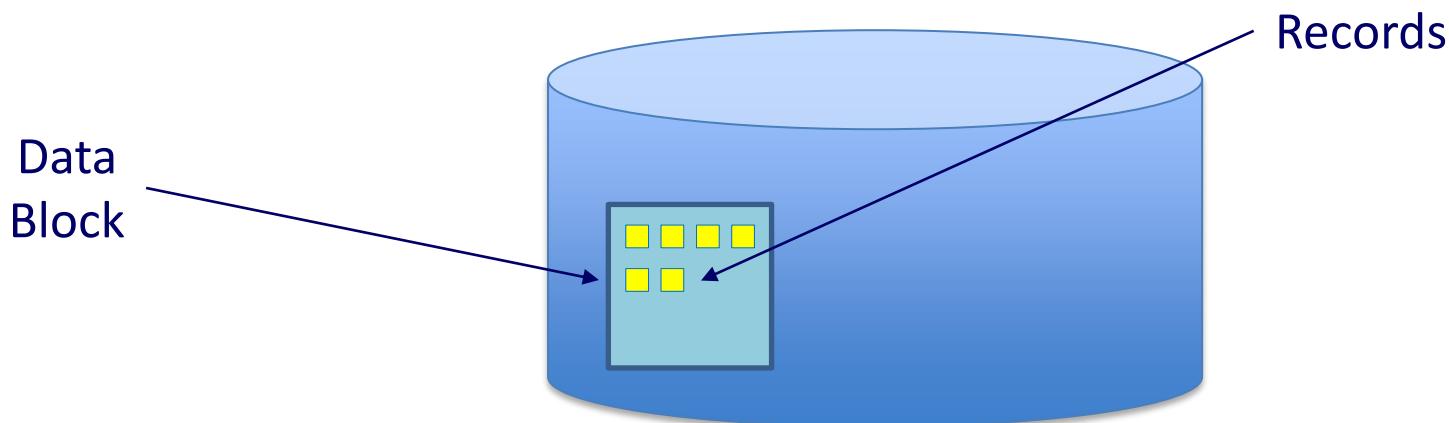
- For small files:
 - Small blocks for storage efficiency
 - Concurrent ops more efficient than sequential
 - Files used together should be stored together
- For large files:
 - Large blocks for storage efficiency
 - Contiguous allocation for sequential access
 - Efficient lookup for random access
- May not know at file creation
 - Whether file will become small or large
 - Whether file is persistent or temporary
 - Whether file will be used sequentially or randomly

File System Design

- Data structures
 - Directories: file name -> file metadata
 - Store directories as files
 - File metadata: how to find file data blocks + other properties of files
 - Free map: list of free disk blocks
- How do we organize these data structures?
 - Device has non-uniform performance

Data blocks and records

- Data in files are accessible in records
 - e.g., in Unix a record is a single byte
- Data are physically stored (and accessed) in blocks
- Usually block size >> record size



Design Challenges

- Index structure
 - How do we locate the blocks of a file?
- Index granularity
 - What block size do we use?
- Free space
 - How do we find unused blocks on disk?
- Locality
 - How do we preserve spatial locality?
- Reliability
 - What if machine crashes in middle of a file system op?

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym.)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

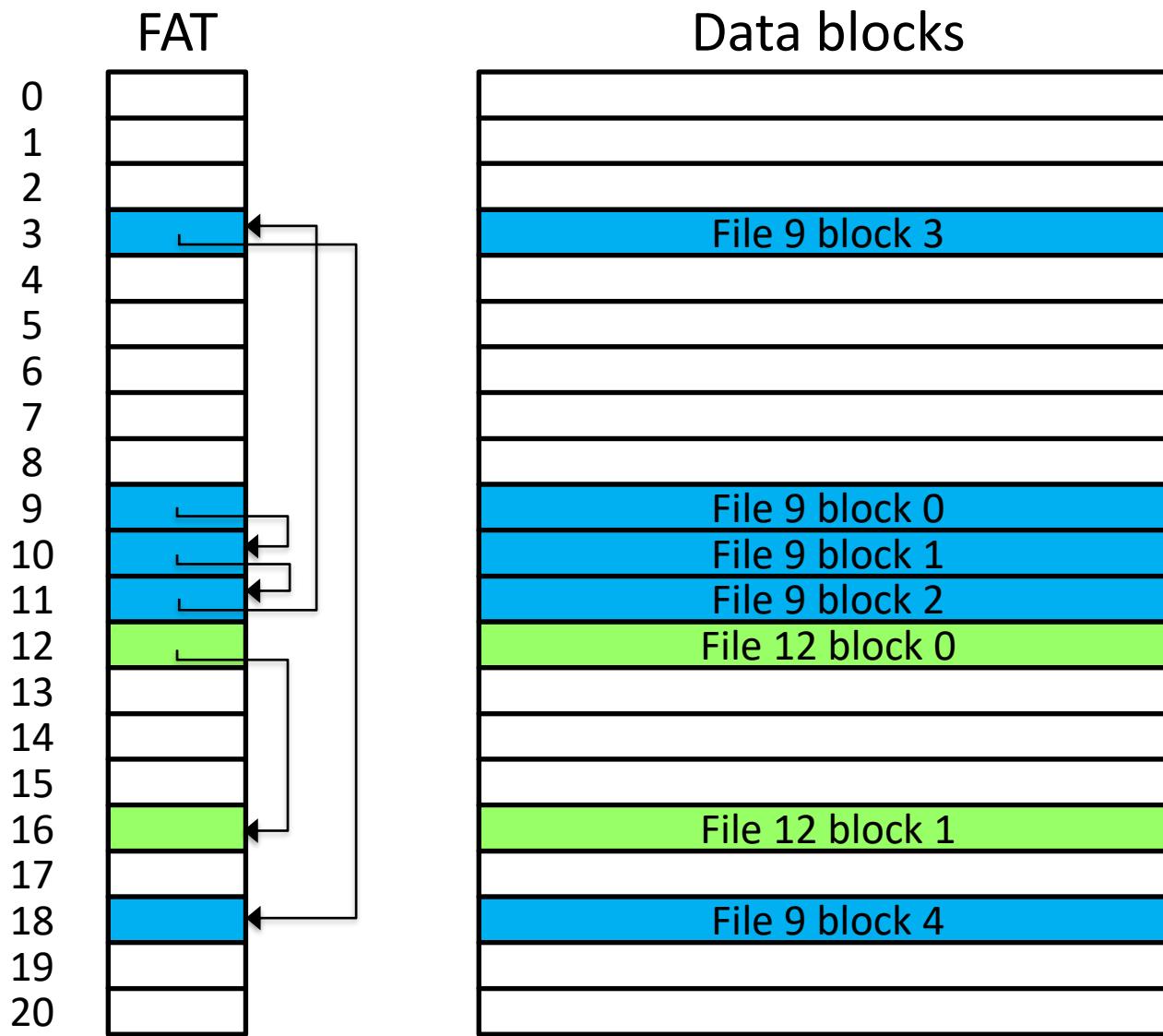
Named Data in a File System



Microsoft File Allocation Table (FAT)

- Linked list index structure
 - Simple, easy to implement
 - Still widely used (e.g., thumb drives)
- File table:
 - Linear map of all blocks on disk
 - Each file a linked list of blocks

FAT file system



FAT

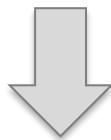
- Pros:
 - Easy to find free block
 - Easy to append to a file
 - Easy to delete a file
- Cons:
 - FAT size
 - Should be uploaded in main memory
 - Limitation to file system size
 - Limited metadata and no protection
 - Random access is slow
 - Fragmentation
 - File blocks for a given file may be scattered
 - Files in the same directory may be scattered
 - Problem becomes worse as disk fills

Limitazioni del file system FAT

Posto:

- L lunghezza (in bit) degli elementi della FAT
- B la dimensione(in byte) dei blocchi del disco

Il numero di blocchi indirizzabili è 2^L (capacità del disco, o partizione)



la massima estensione del file system è:

2^L blocchi, ovvero a $B \cdot 2^L$ byte.

se ogni elemento occupa N byte (solitamente L è multipla del byte), la FAT occupa complessivamente: $N \cdot 2^L$ byte

Limitazioni del file system FAT

Esempio: con $N=2$ (\rightarrow FAT 16) e $B = 2^{10}$ (blocchi di 1 Kbyte)

La massima estensione del file system è 2^{16} blocchi (2^{26} byte = 64MB)

La FAT occupa complessivamente $2 * 2^{16}$ byte = 128 Kbyte

- Sul disco la FAT occupa 128 KB, cioè 128 blocchi.
- con una memoria paginata e pagine di 1Kbyte, la FAT occupa 128 pagine in memoria principale.

Dato che gli elementi che descrivono un file possono essere distribuiti su molte pagine diverse, possono verificarsi frequenti errori di pagina quando si percorre un file.

→ Per realizzare file systems più estesi si usano blocchi di dimensioni maggiori.

Limitazioni dei file systems FAT

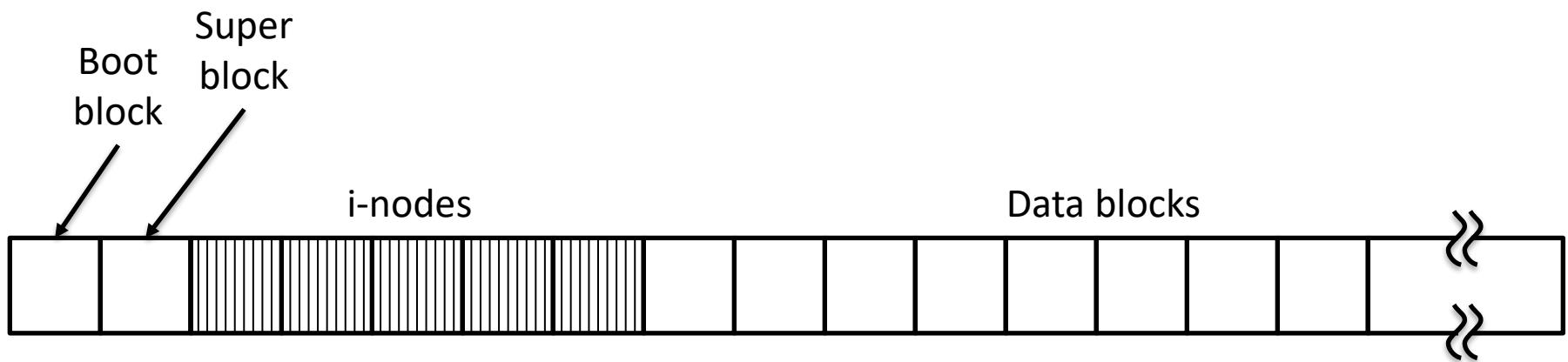
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

- Massima dimensione del File System per diverse ampiezze dei blocchi

Berkeley UNIX FFS (Fast File System)

- inode table (i-list)
 - “Analogous” to FAT table, but does much more...
- inode
 - Metadata
 - File owner, access permissions, access times, ...
 - Set of pointers to data blocks

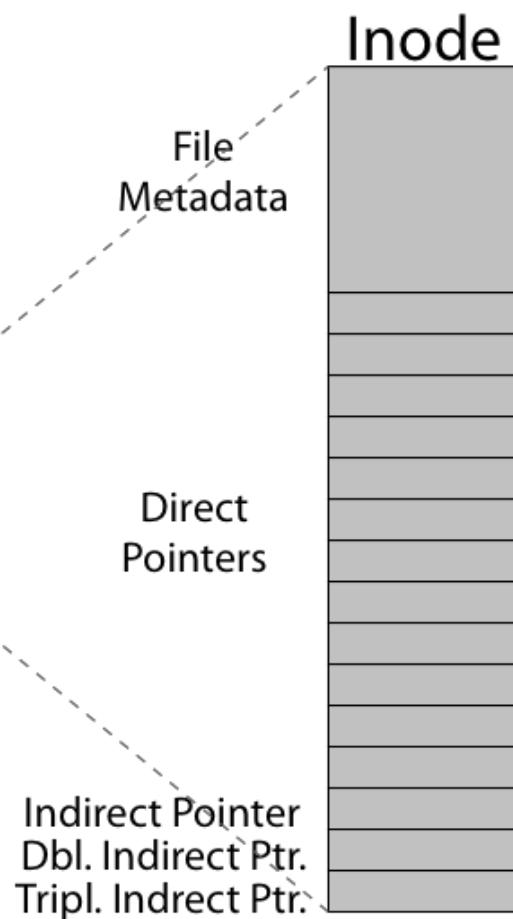
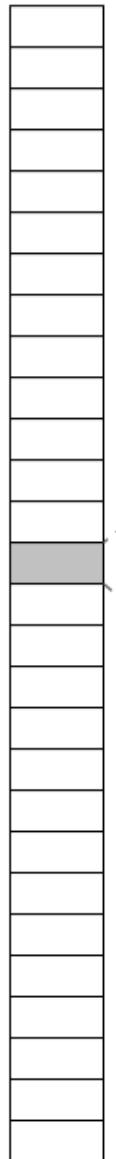
Physical disk organization in UNIX



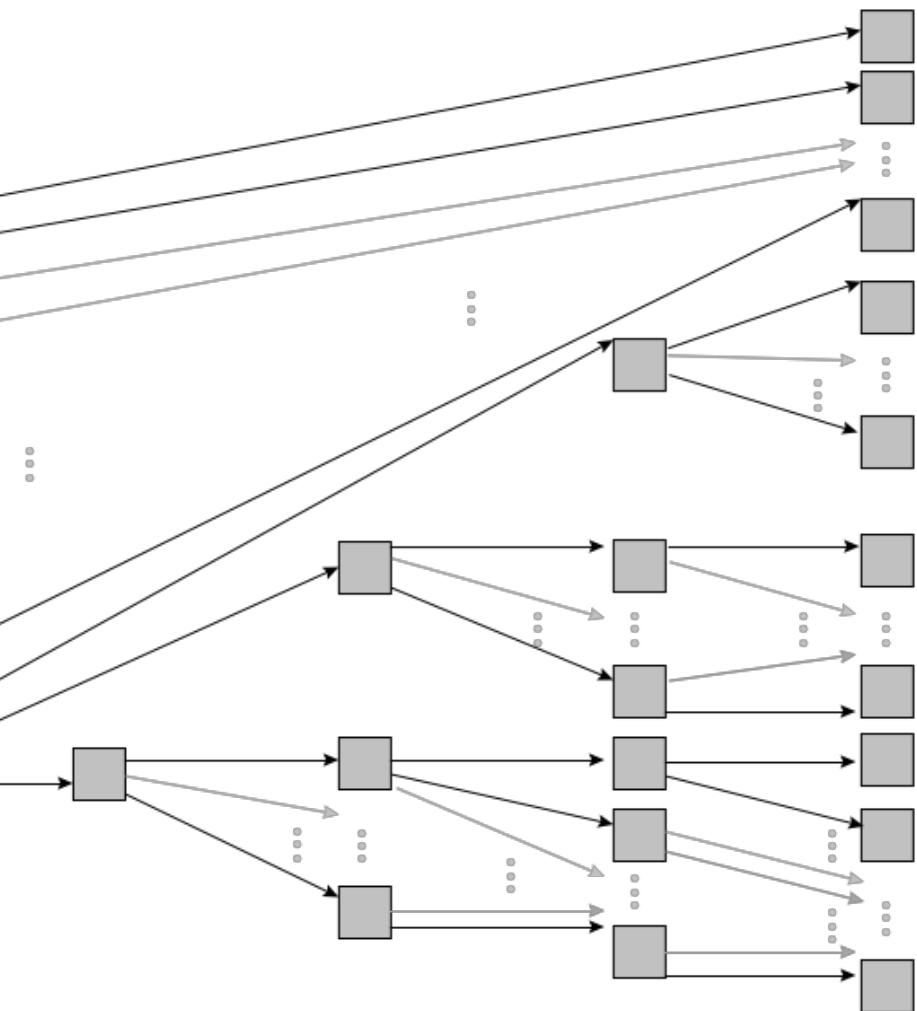
FFS inode

- Metadata
 - File owner, access permissions, access times, ...
- Set of 12 data pointers (of 32 bits each)
 - With 4KB blocks => max size of 48KB
- Indirect block pointer
 - pointer to disk block of data pointers
 - 4KB block size => 1K pointers to data blocks => 4MB
- Doubly indirect block pointer
 - Doubly indirect block => 1K indirect blocks
 - 4GB (+ 4MB + 48KB)
- Triply indirect block pointer
 - Triply indirect block => 1K doubly indirect blocks
 - 4TB (+ 4GB + 4MB + 48KB)

Inode Array



Triple Double
Indirect Indirect Indirect Data
Blocks Blocks Blocks Blocks

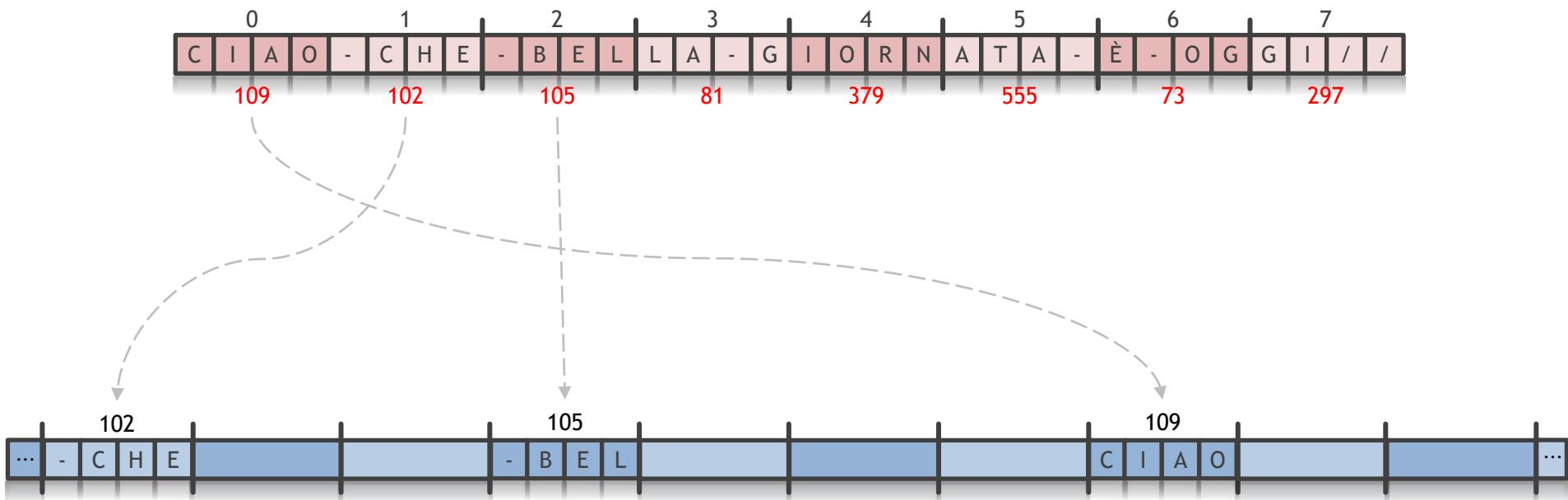


Esempio (semplificato) di i-node

- Blocchi di 4 byte
- Puntatori di 2 byte
- File di 30 byte (quindi 8 blocchi)
- I node con:
 - 2 puntatore diretti
 - 1 puntatore indiretto singolo
 - 1 puntatore indiretto doppio

Esempio (semplificato) di i-node

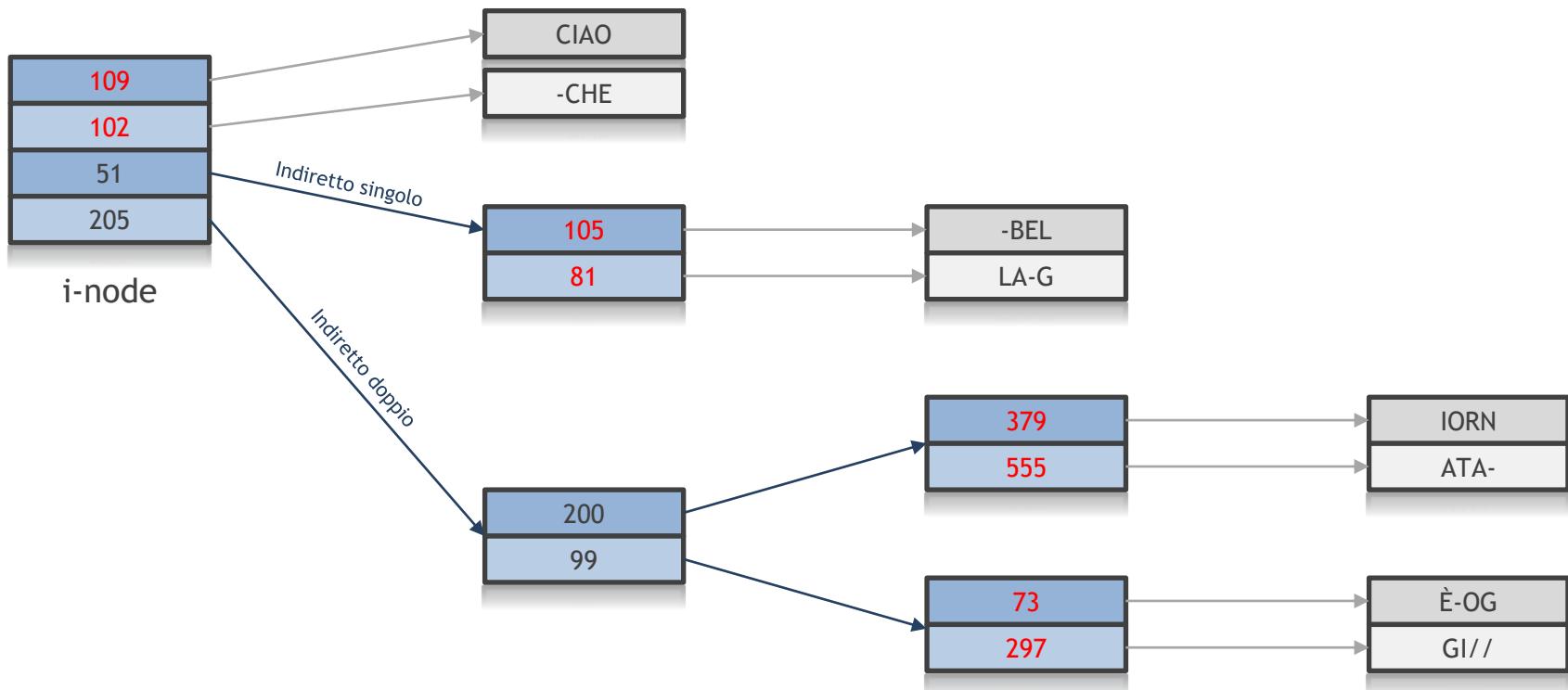
Supponiamo di avere il seguente file
Suddiviso in blocchi da 4 byte ciascuno:



Ogni blocco **logico** del file corrisponde
Ad un blocco **fisico** nel disco

Esempio (semplificato) di i-node

Supponiamo adesso che gli indirizzi siano a 2 byte



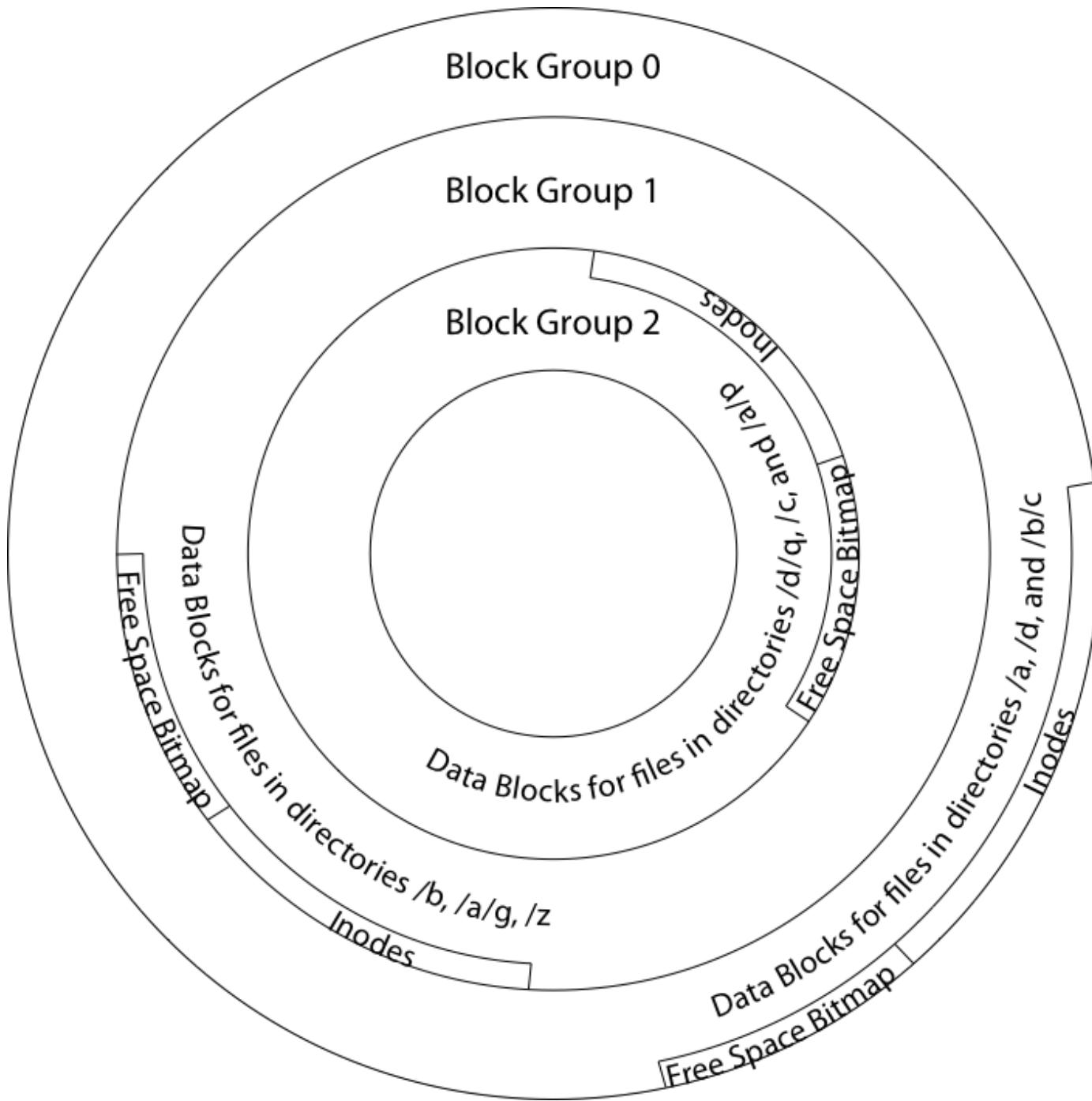
Quanti blocchi sono indirizzabili da ogni puntatore?
4Byte/2Byte = 2

FFS Asymmetric Tree

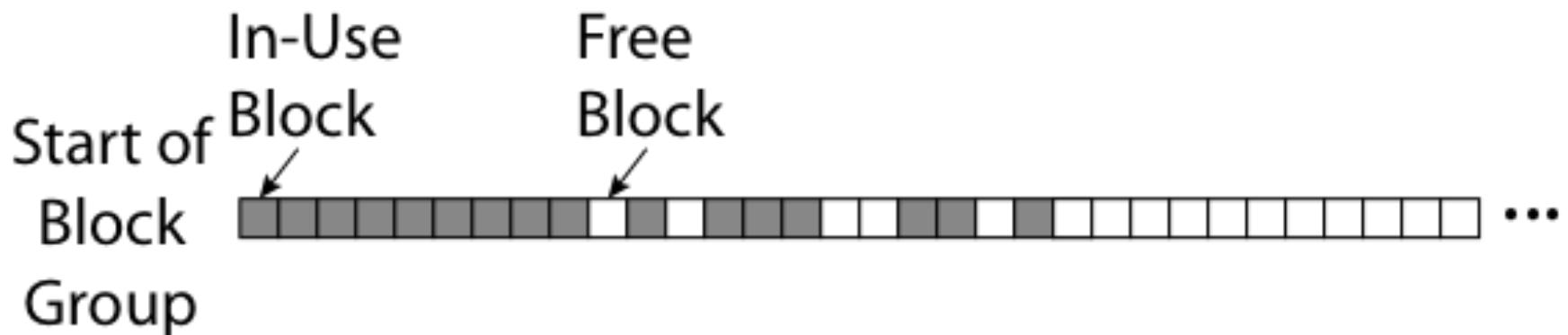
- Small files: shallow tree
 - Efficient storage for small files
- Large files: deep tree
 - Efficient lookup for random access in large files
- Sparse files: only fill pointers if needed

FFS Locality

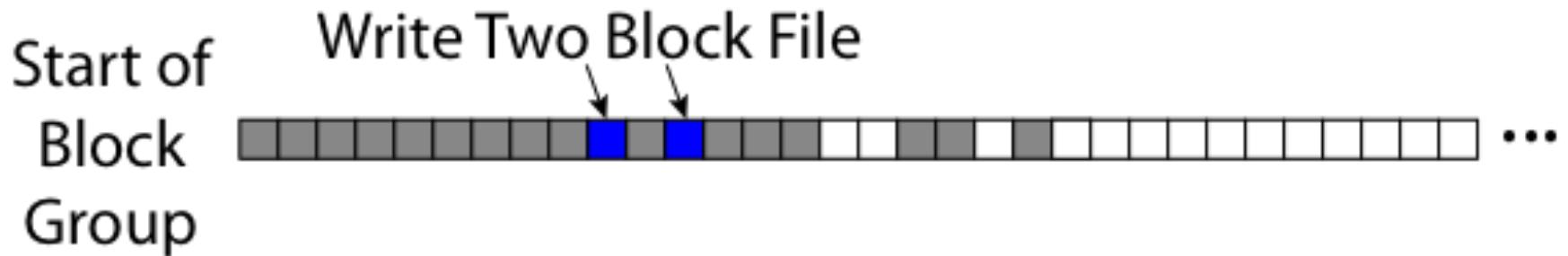
- Block group allocation
 - Block group is a set of nearby cylinders
 - Files in same directory located in same group
 - Subdirectories located in different block groups
- inode table spread throughout disk
 - inodes, bitmap near file blocks
- First fit allocation
 - Small files fragmented; large files contiguous (if disk not close to full utilization...)



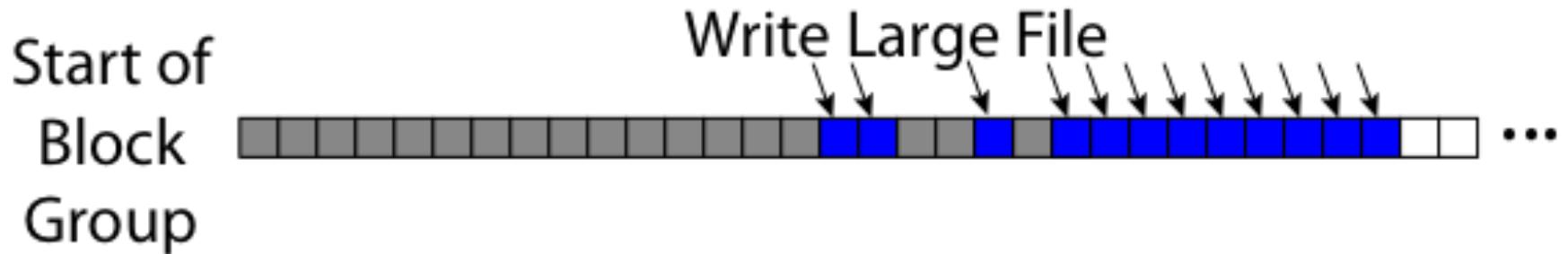
FFS First Fit Block Allocation



FFS First Free Block Allocation



FFS First Free Block Allocation



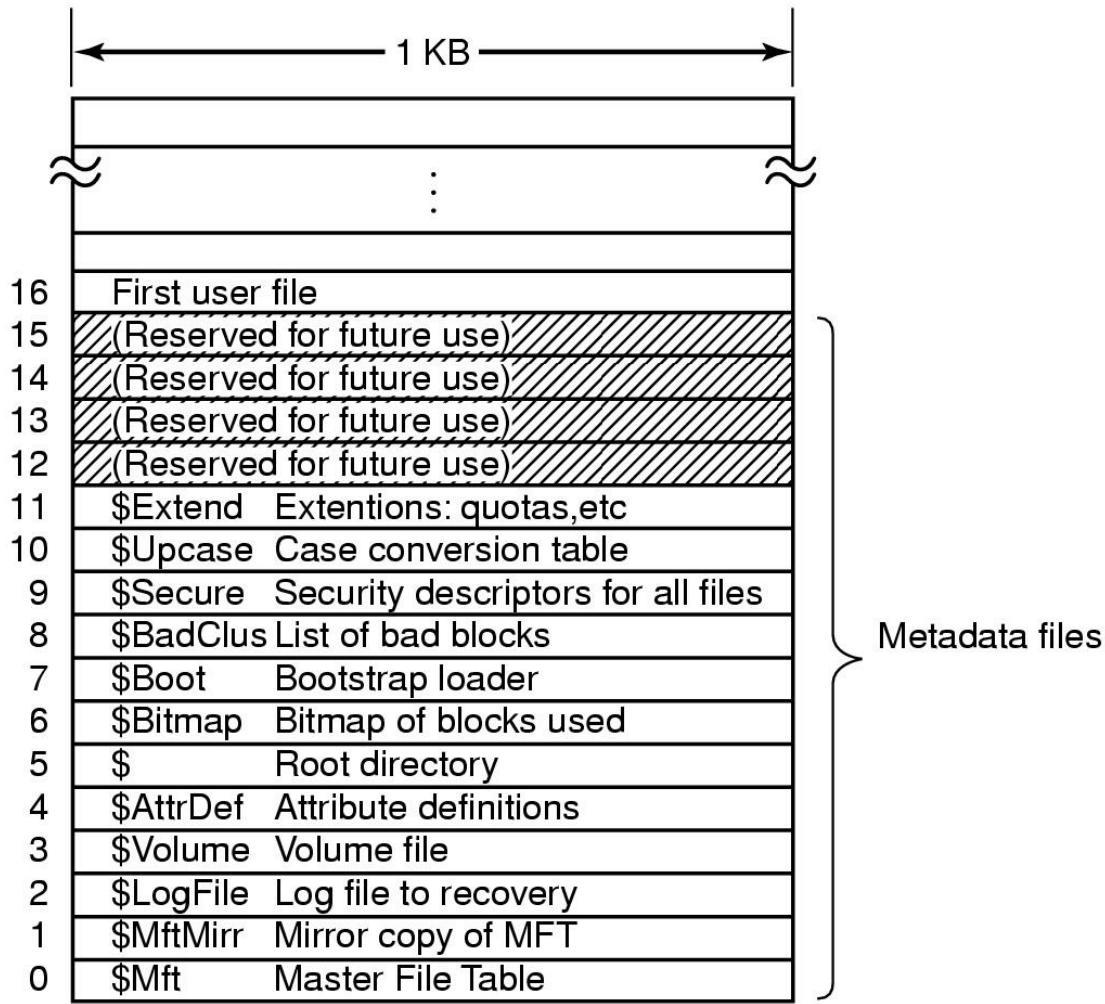
FFS

- Pros
 - Efficient storage for both small and large files
 - Locality for both small and large files
 - Locality for metadata and data
- Cons
 - Inefficient for tiny files (a 1 byte file requires both an i-node and a data block)
 - Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)
 - Need to reserve 10-20% of free space to prevent fragmentation

NTFS

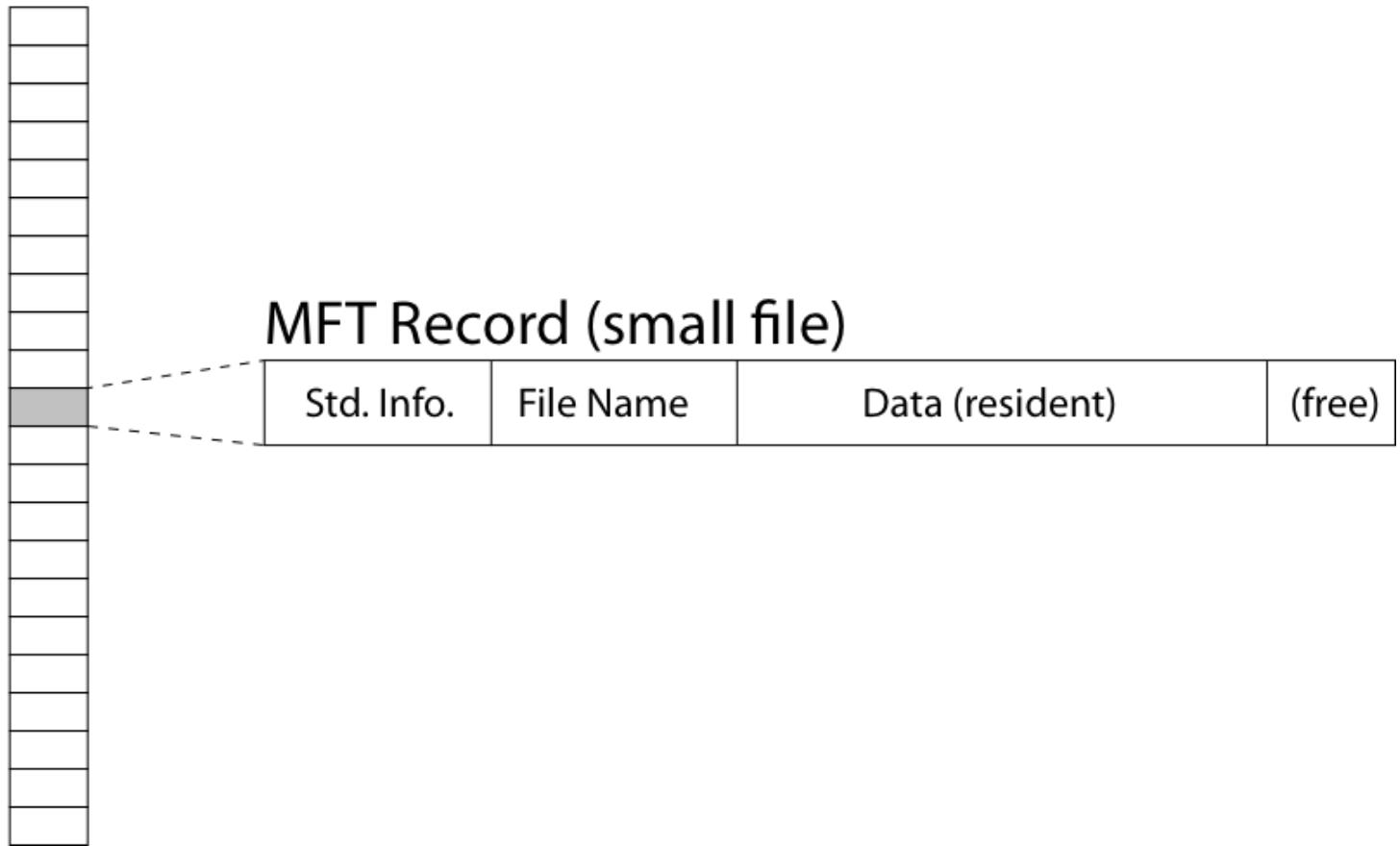
- Master File Table
 - A table of records (one x file)
 - Flexible 1KB storage for a file metadata and data
 - MFT itself is a file!
- Extents
 - Block pointers cover runs of blocks
 - Linux (ext4) adopts a similar approach
 - File create can provide hint as to size of file
- Journalling for reliability
 - Will not be discussed!

MFT



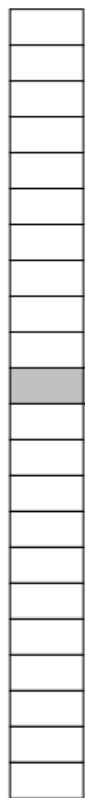
NTFS Small File

Master File Table

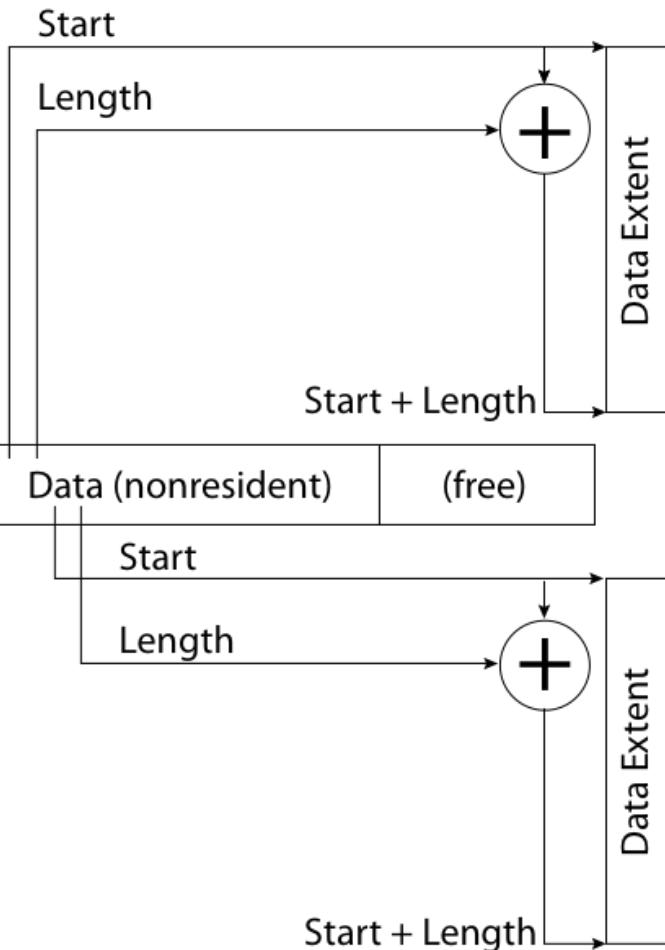


NTFS Medium File

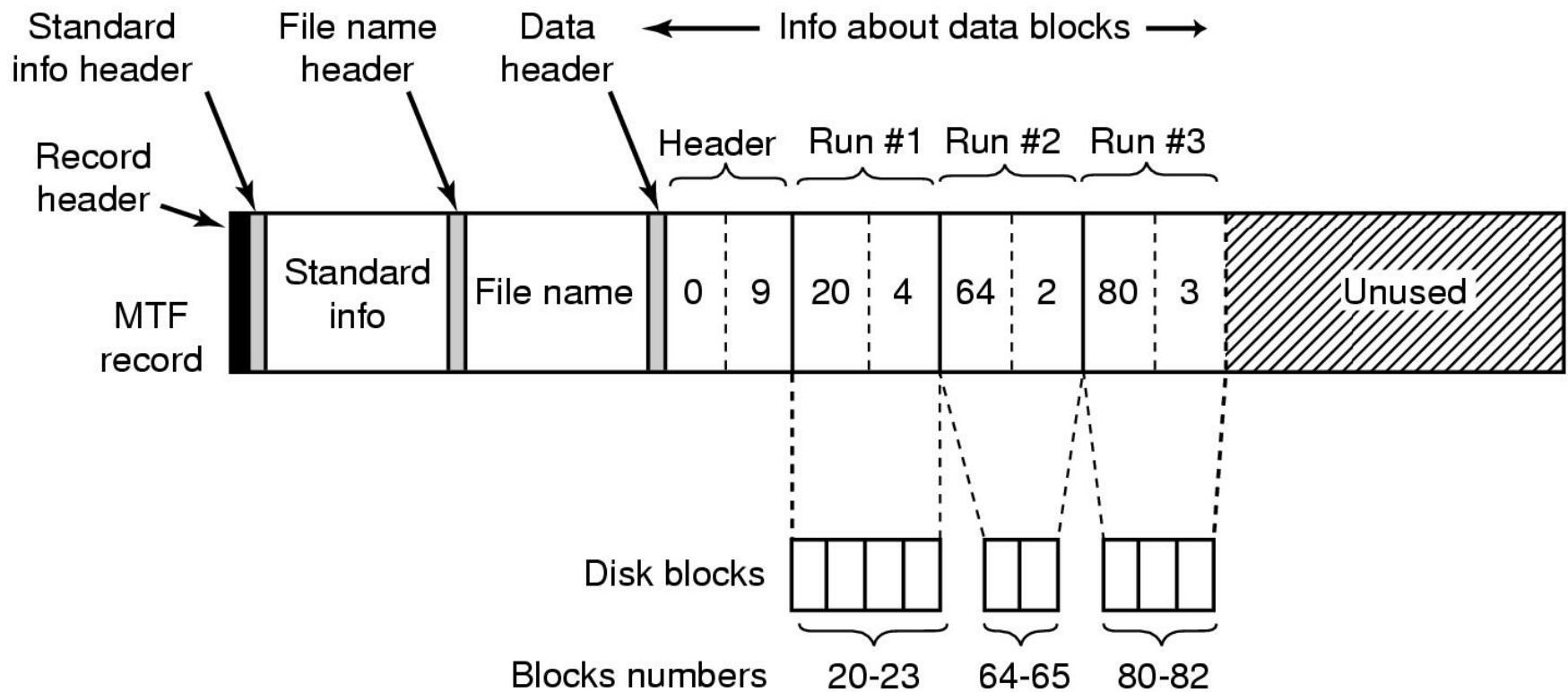
Master File Table



MFT Record

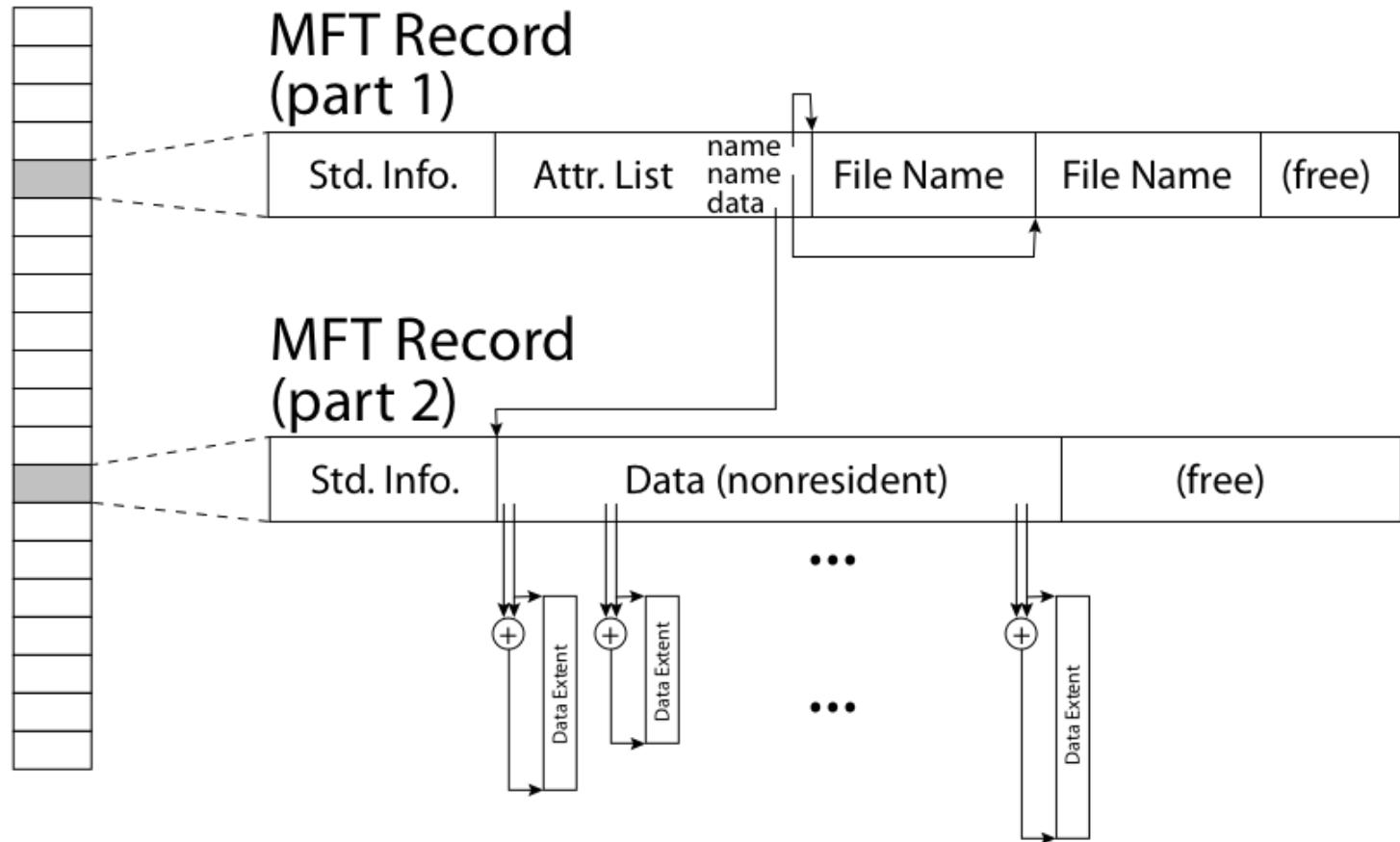


NTFS Medium File

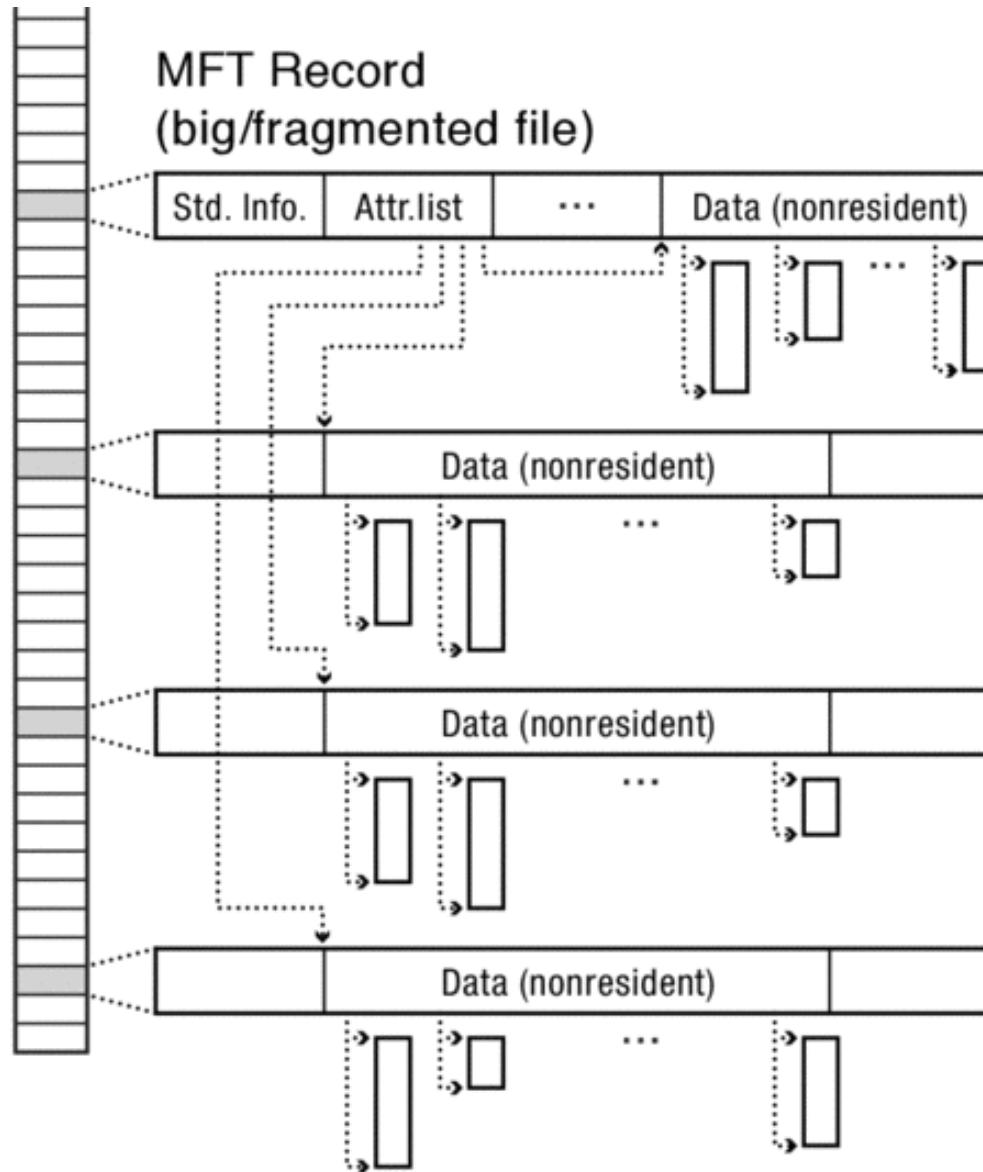


NTFS Indirect Block

Master File Table



NTFS Multiple Indirect Blocks



MFT Record (huge/badly-fragmented file)



Extent with part of attribute list

Data (nonresident)

Data (nonresident)

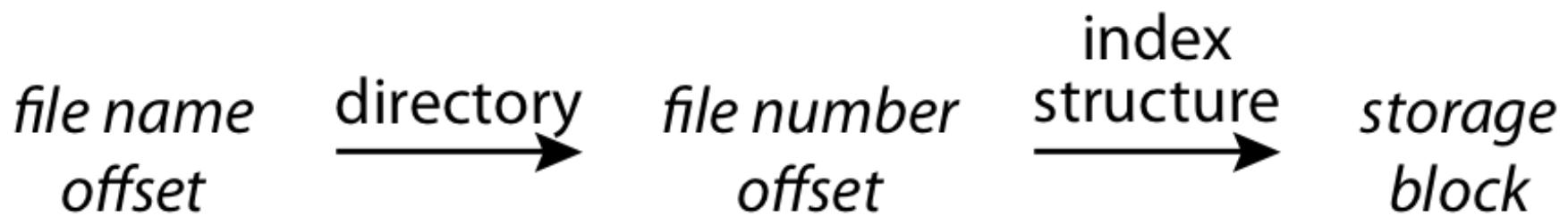
Data (nonresident)

Extent with part of attribute list

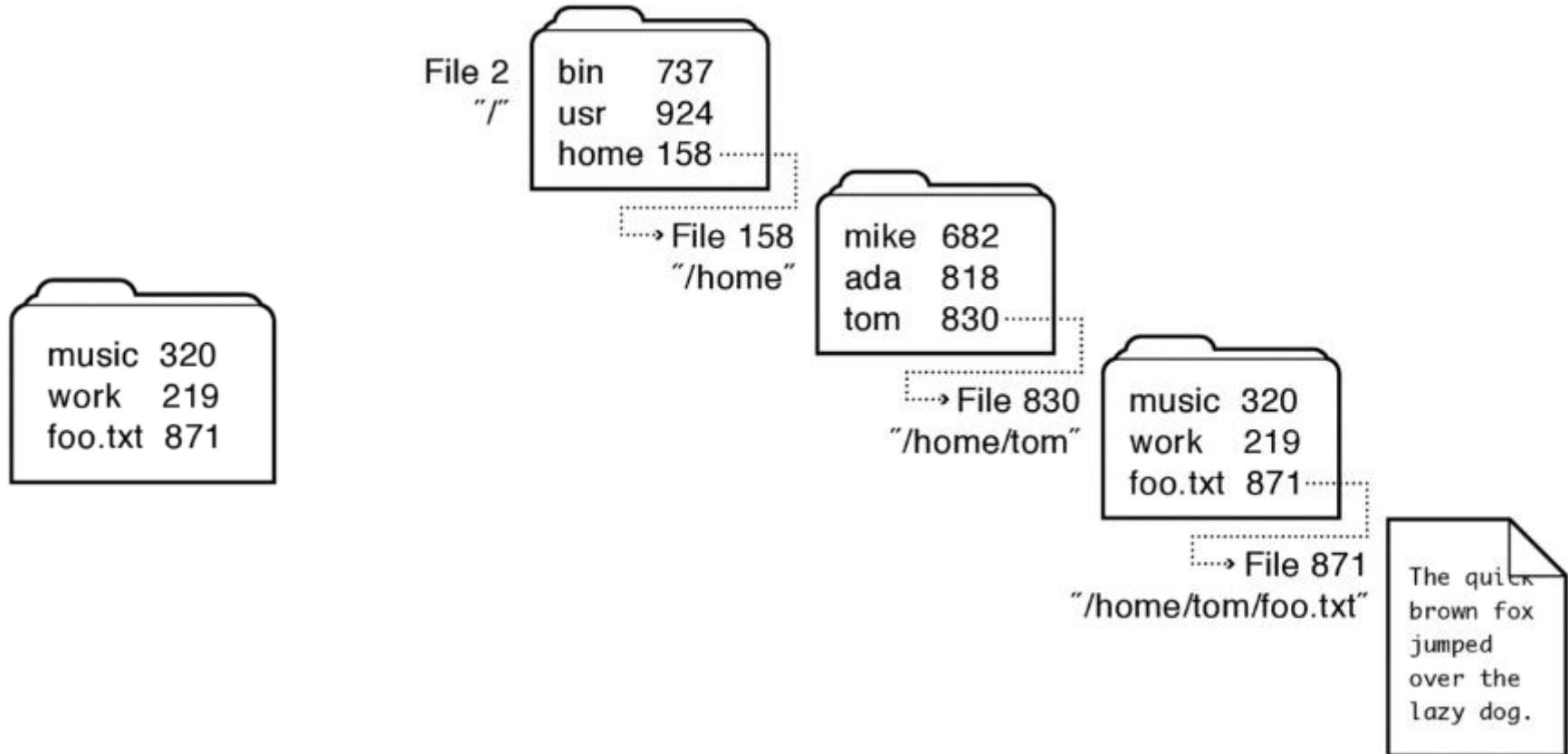
Data (nonresident)

Data (nonresident)

Named Data in a File System (remind)

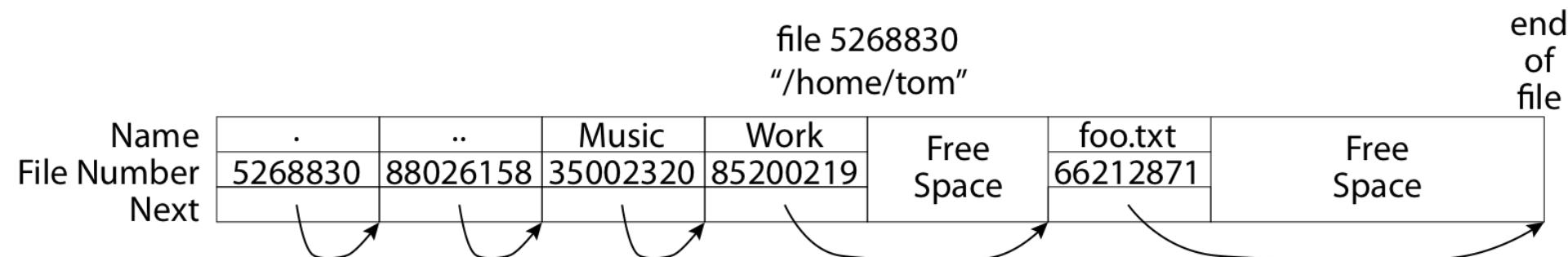


Named Data in a File System



Directories

- Directories are files
 - Map file name to file number (MFT #, inode num)
- Table of file name -> file number
 - Small directories: linear search



Large Directories: B-Trees

