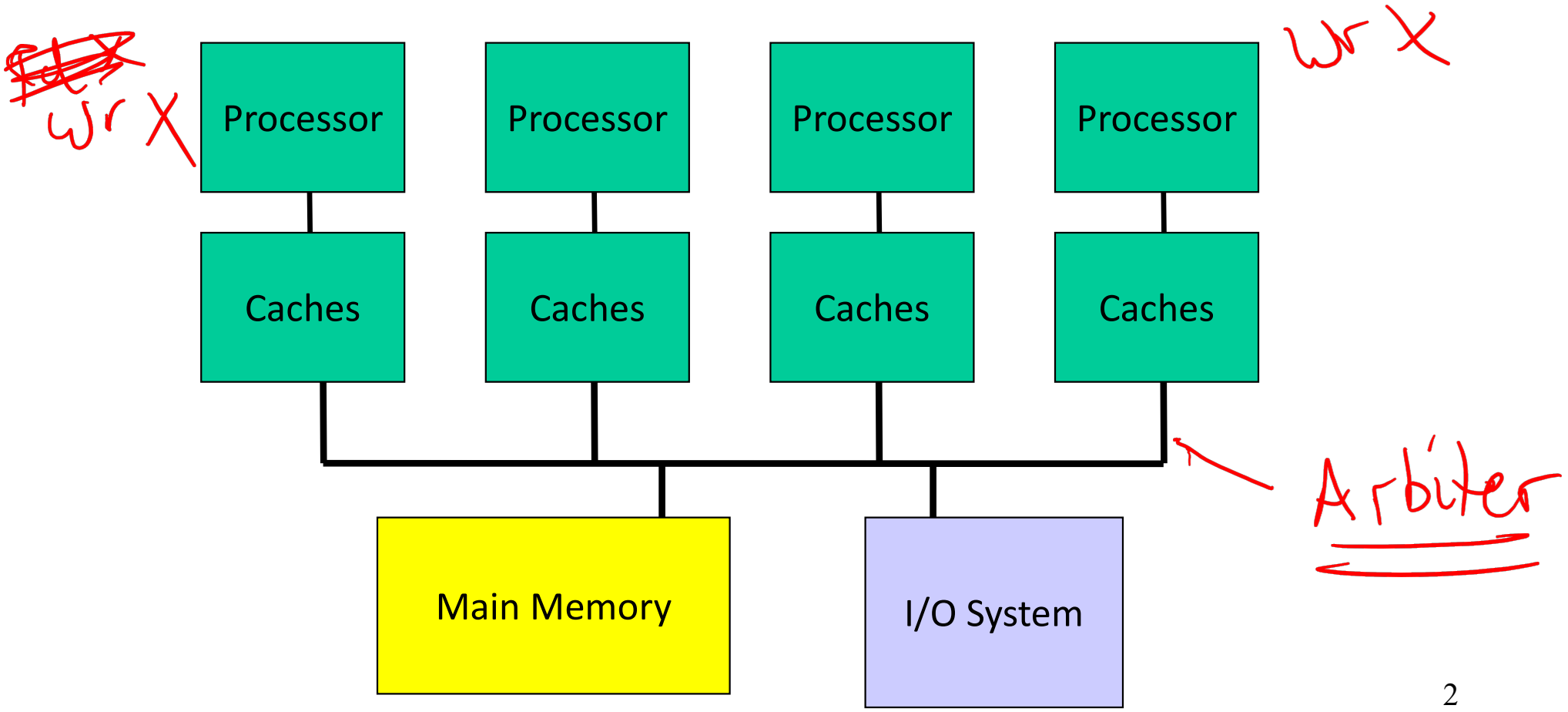
Lecture 25: Synchronization, Consistency, VM

- Today's topics:
  - Synchronization primitives
  - Consistency models
  - Virtual memory basics
  - Midterm 2 score distribution

	A	A-	B+	B	B-	C+/C/C-
%ile	16%	32%	45%	58%	71%	86%
Rank	38	77	108	139	170	206
M2	88	80	73.7	68.7	61.5	49.7
M1	90	84	79.1	75.5	71	62.6
HW	105.1	102.7	99.3	96	92.7	86.5

Snooping-Based Protocols

- Three states for a block: invalid, shared, modified
- A write is placed on the bus and sharers invalidate themselves
- The protocols are referred to as MSI, MESI, etc.



Cache Coherence Protocols

- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

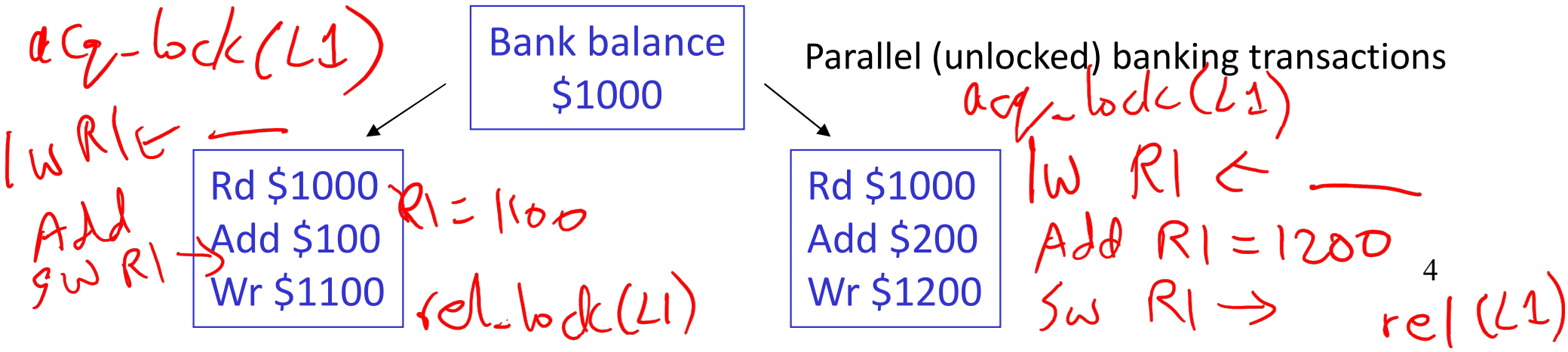
  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

Constructing Locks

- Applications have phases (consisting of many instructions) that must be executed atomically, without other parallel processes modifying the data

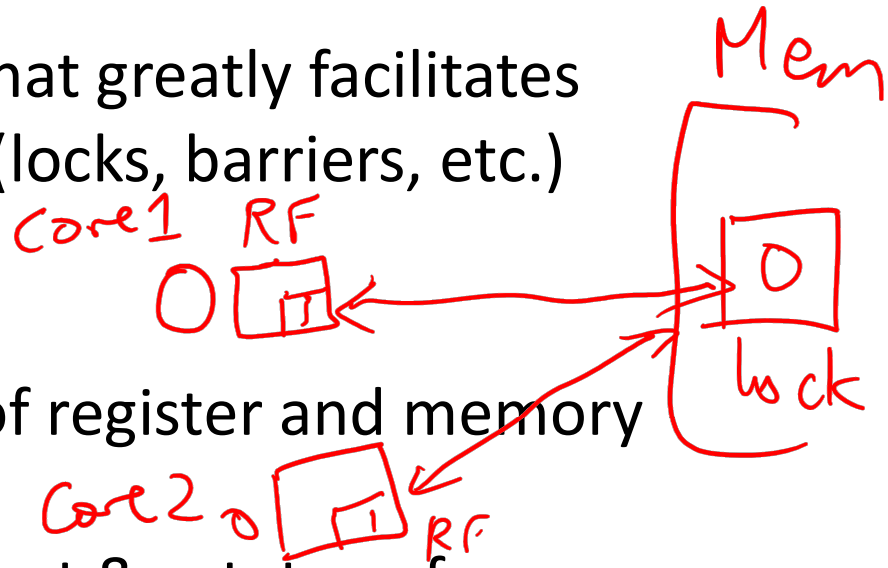
- A lock surrounding the data/code ensures that only one program can be in a critical section at a time

- The hardware must provide some basic primitives that allow us to construct locks with different properties



Synchronization

- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write



- Atomic exchange: swap contents of register and memory
- Special case of atomic exchange: test & set: transfer memory location into register and write 1 into memory (if memory has 0, lock is free)
- lock: t&s register, location
  bnz register, lock
  CS
  st location, #0

lock_acq
critical section
lock_rel
1

When multiple parallel threads execute this code, only one will be able to enter CS

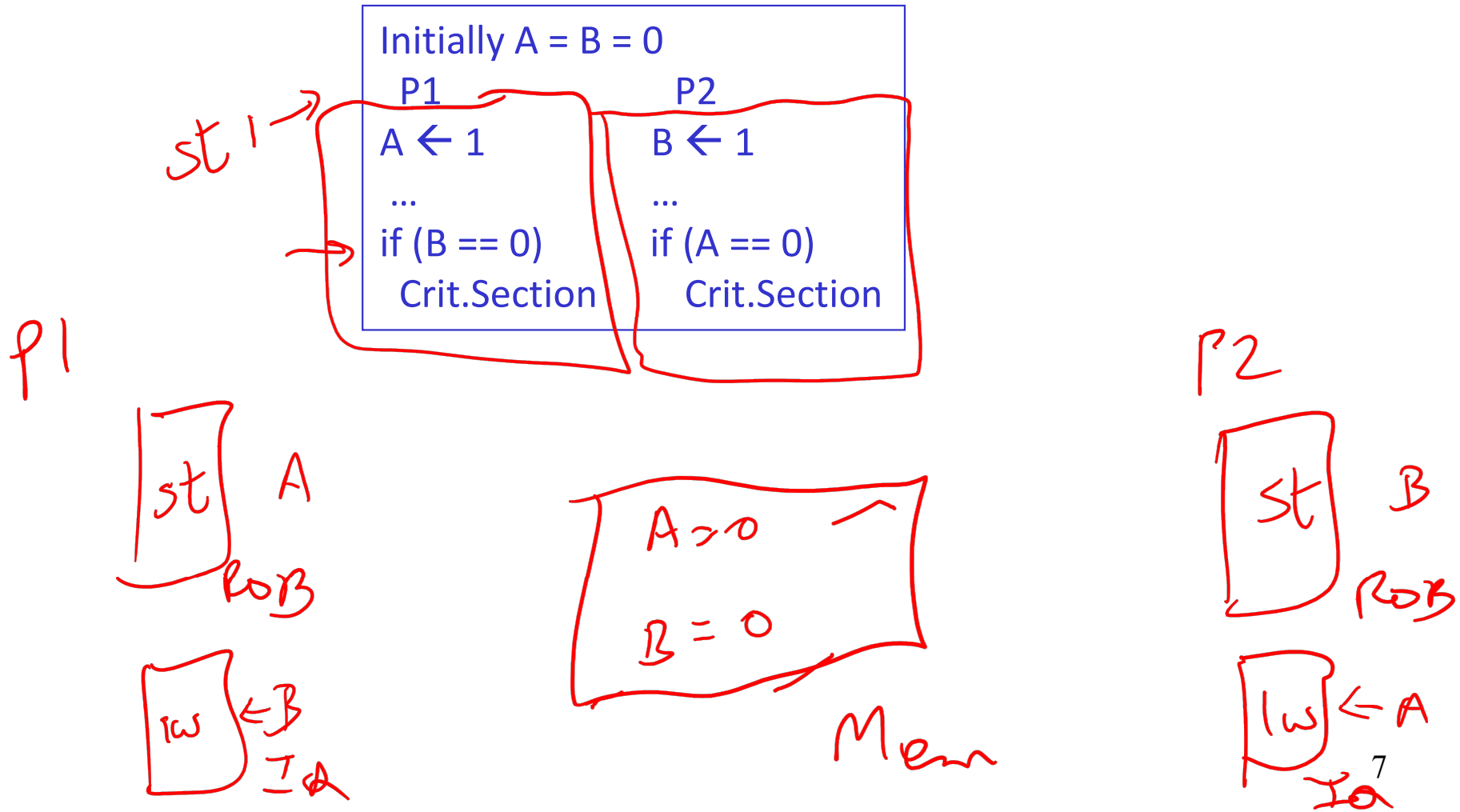
Coherence Vs. Consistency

- Coherence guarantees (i) write propagation (a write will eventually be seen by other processors), and (ii) write serialization (all processors see writes to the same location in the same order)

- The consistency model defines the ordering of writes and reads to different memory locations – the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions

Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence



Consistency Example

- Consider a multiprocessor with bus-based snooping cache coherence

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
 …                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

The programmer expected the above code to implement a lock – because of ooo, both processors can enter the critical section

The consistency model lets the programmer know what assumptions they can make about the hardware's reordering capabilities

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- The multiprocessor in the previous example is not sequentially consistent

- Can implement sequential consistency by requiring the following: program order, write serialization, everyone has seen an update before a value is read – very intuitive for the programmer, but extremely slow

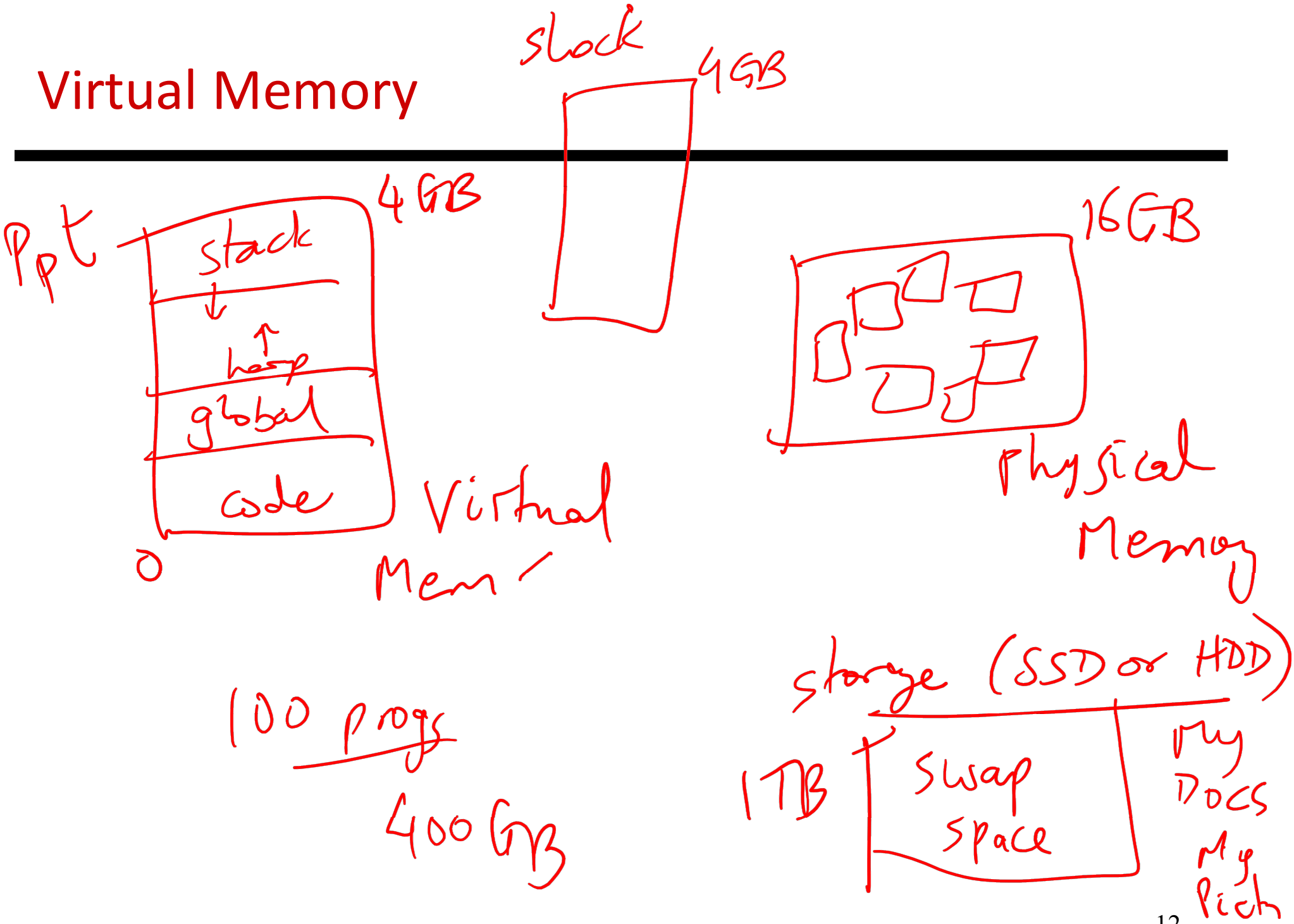
Relaxed Consistency

- Sequential consistency is very slow
- The programming complications/surprises are caused when the program has race conditions (two threads dealing with same data and at least one of the threads is modifying the data)
- If programmers are disciplined and enforce mutual exclusion when dealing with shared data, we can allow some re-orderings and higher performance
- This is effective at balancing performance & programming effort

Virtual Memory

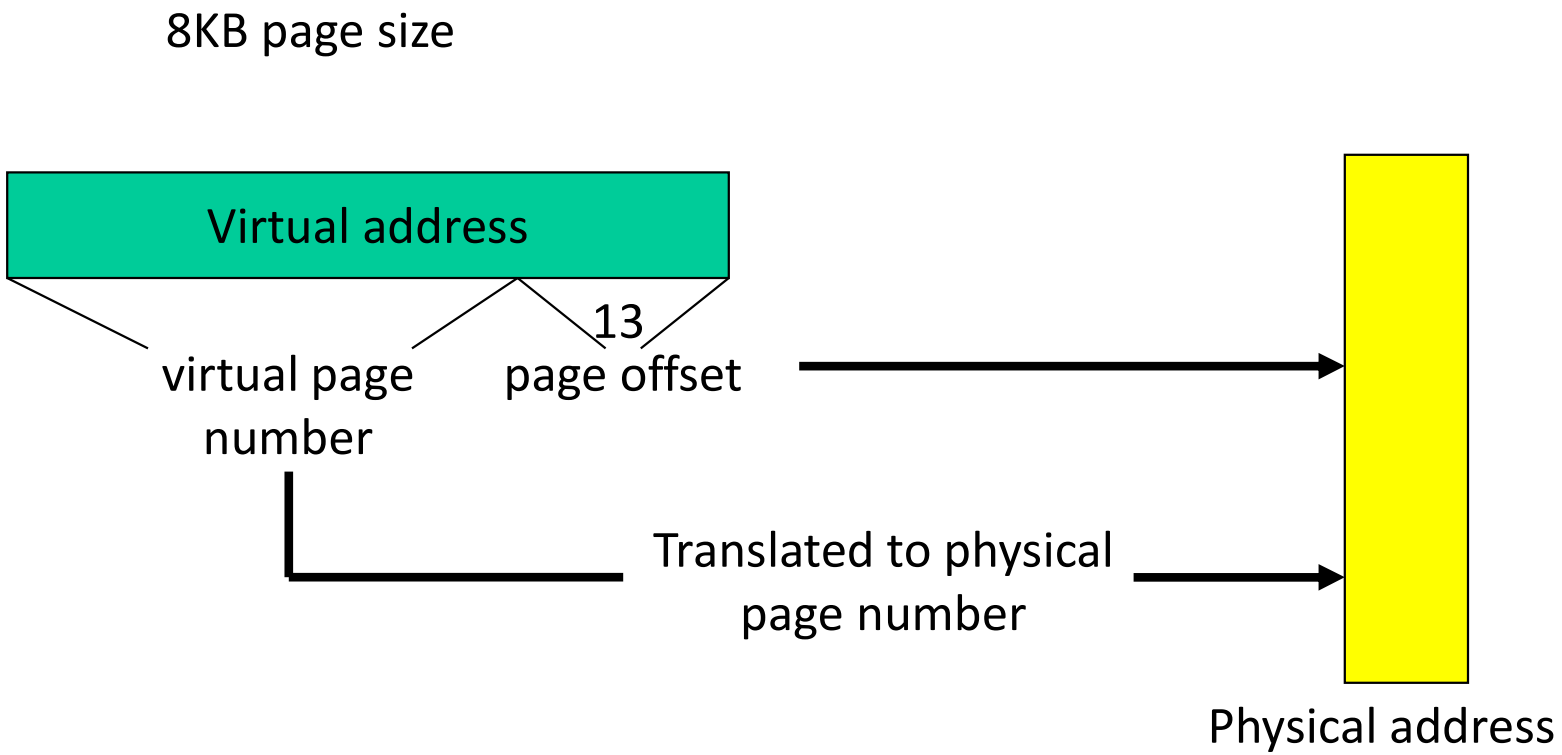
- Processes deal with virtual memory – they have the illusion that a very large address space is available to them
- There is only a limited amount of physical memory that is shared by all processes – a process places part of its virtual memory in this physical memory and the rest is stored on disk (called swap space)
- Thanks to locality, disk access is likely to be uncommon
- The hardware ensures that one process cannot access the memory of a different process

Virtual Memory



Address Translation

- The virtual and physical memory are broken up into pages



Memory Hierarchy Properties

- A virtual memory page can be placed anywhere in physical memory (fully-associative)
- Replacement is usually LRU (since the miss penalty is huge, we can invest some effort to minimize misses)
- A page table (indexed by virtual page number) is used for translating virtual to physical page number
- The page table is itself in memory