Today

Case Studies of parallel OS (Contd.)

* Tornado
* Corey
* Cellular Disco

} Partial reading

Wednesday

Distributed systems

* Watch Lamport clock video

(First TWO videos in Lesson 5 on Udacity)

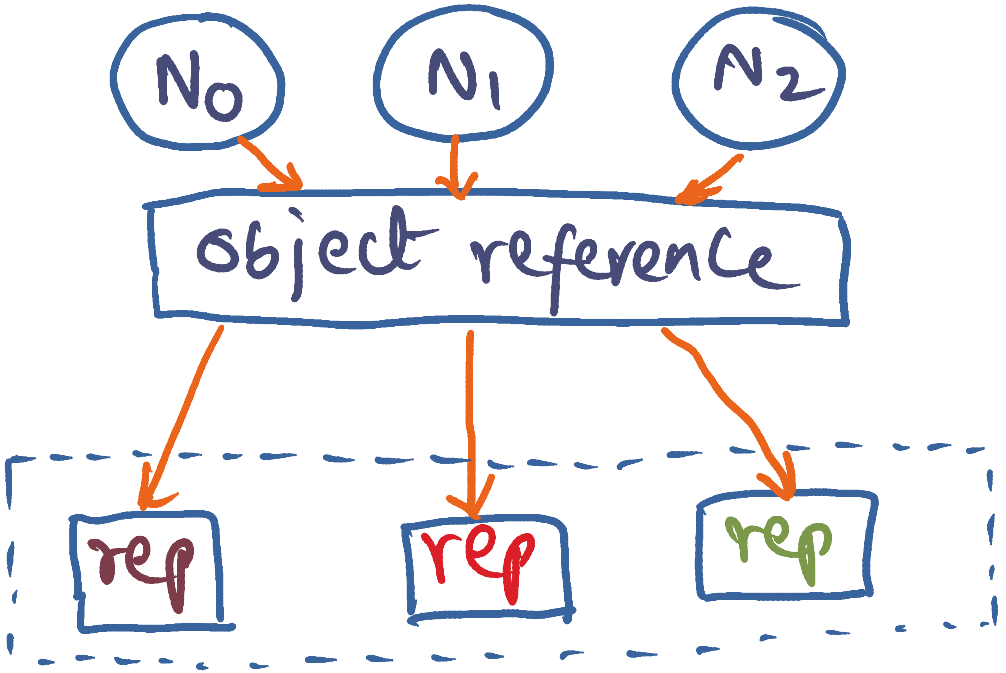
Friday

Townhall (Bring slides to work on in class)

Midterm: Monday 10-5-2015

(in class; closed book + notes)

Advantages of clustered object

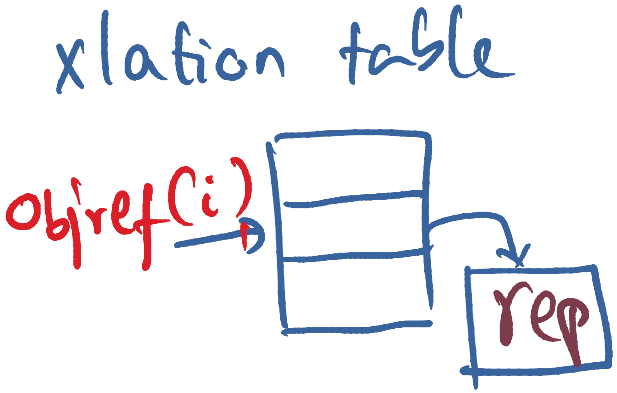


same object reference on all nodes

Allows incremental optimization

* usage pattern determines level of replication

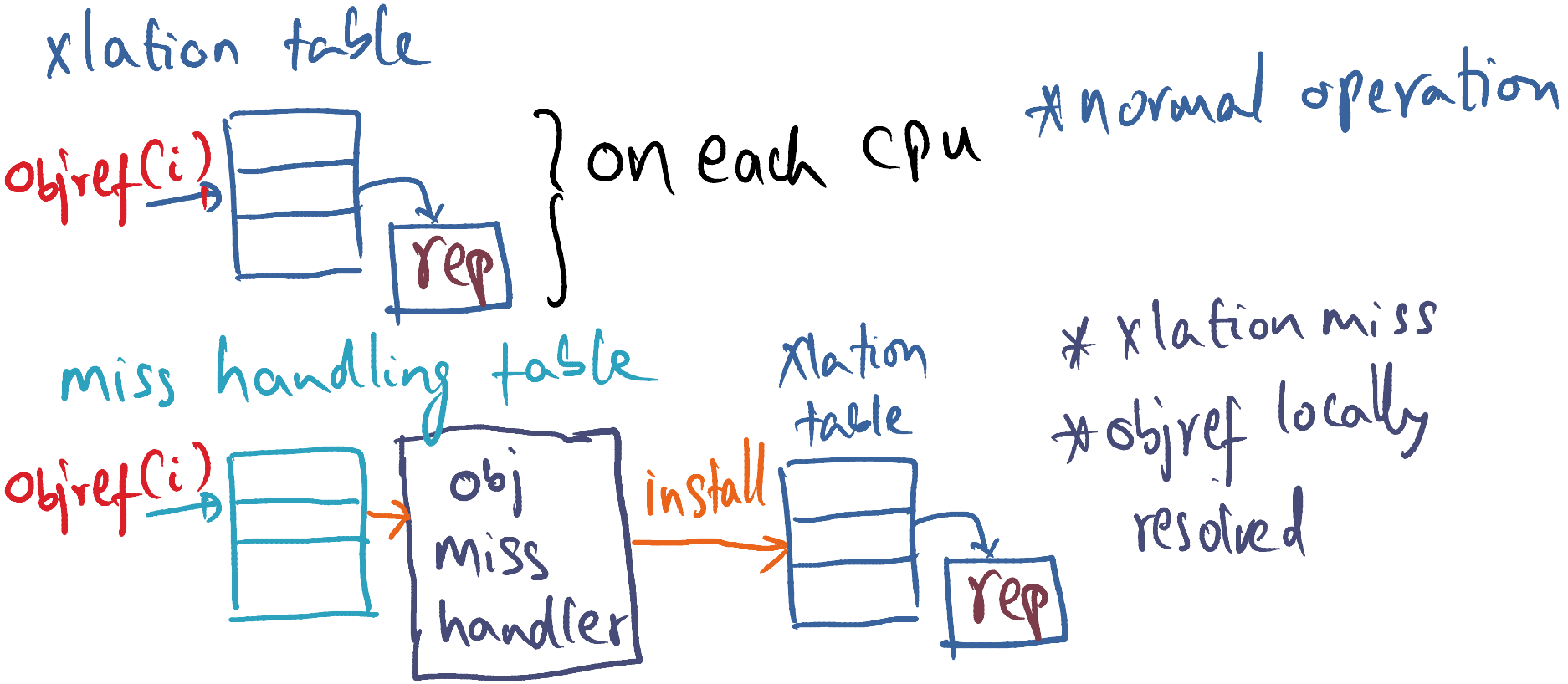
Implementation of clustered object



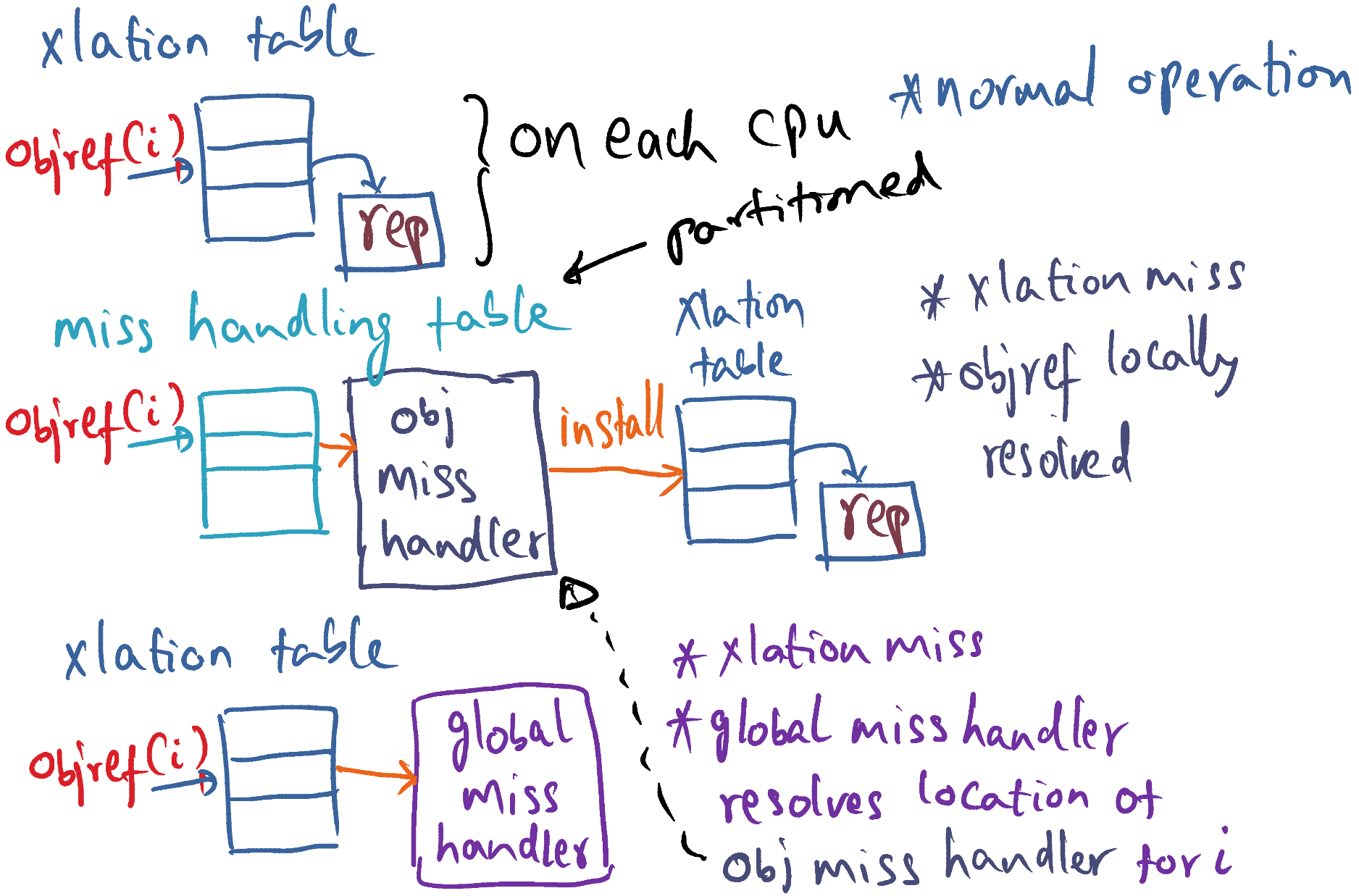
} on each cpu

* normal operation

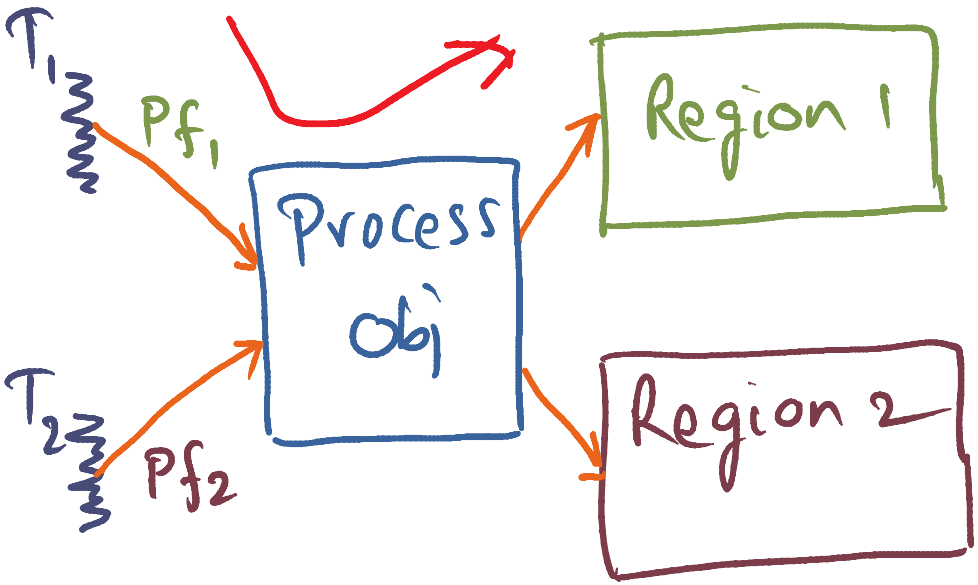
Implementation of clustered object



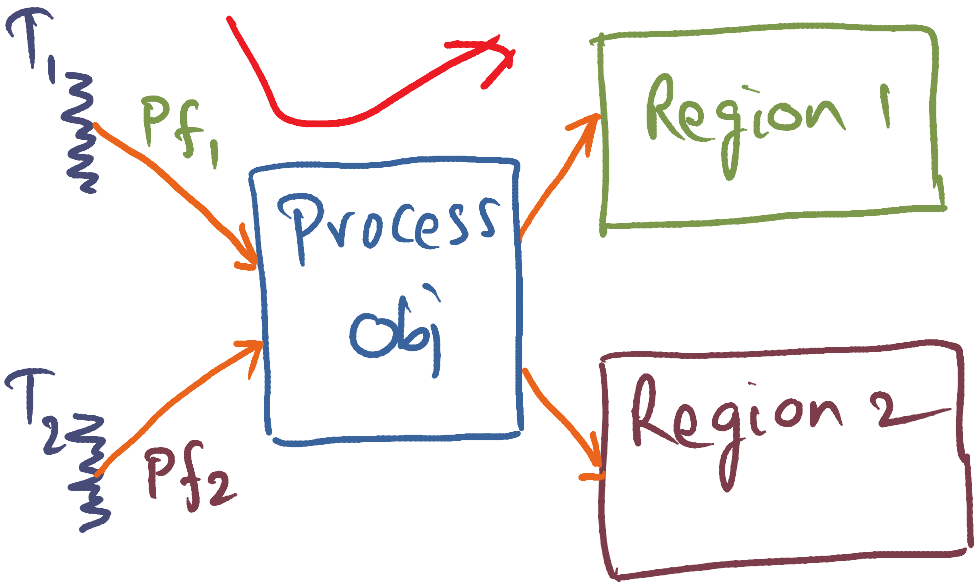
Implementation of clustered object



Non Hierarchical Locking + Existence Guarantee



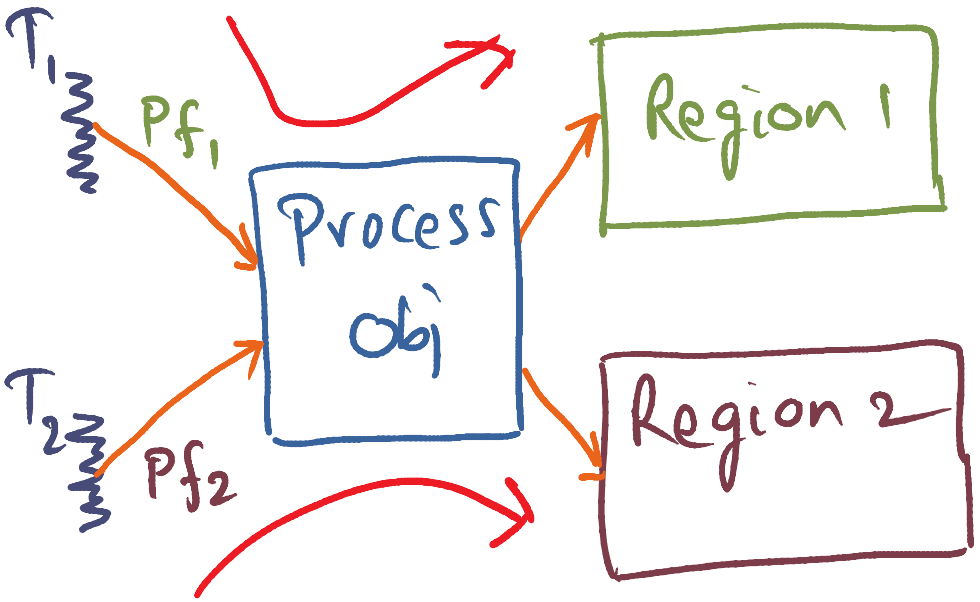
Non Hierarchical Locking + Existence Guarantee



hierarchical locking:

lock(Process)
↓
lock(region)
↓
lock(FCM)
↓
lock(COR)

Non Hierarchical Locking + Existence Guarantee



hierarchical locking:

lock(Process)

↓

lock(region)

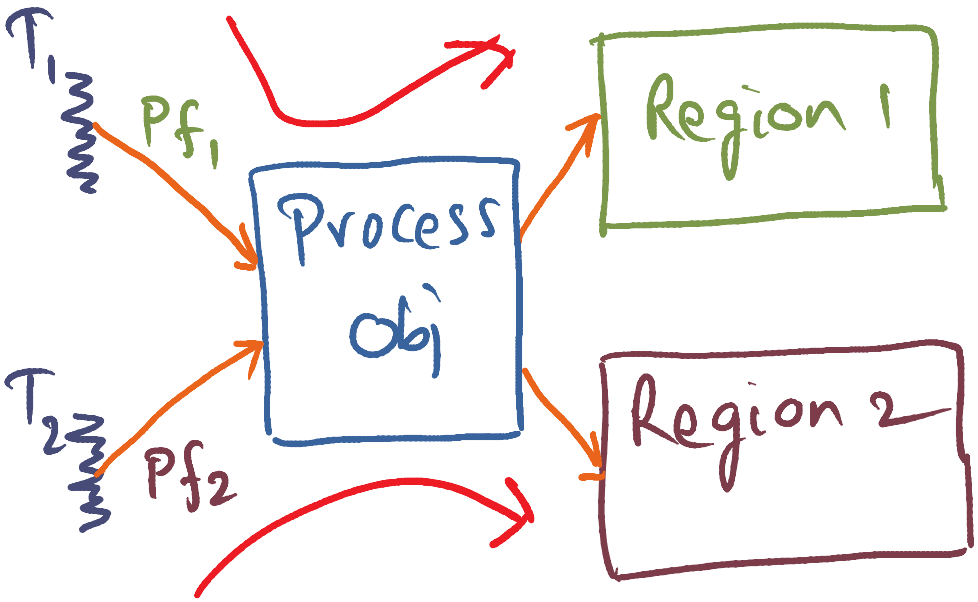
↓

lock(FCM)

↓

lock(COR)

Non Hierarchical Locking + Existence Guarantee



hierarchical locking:

lock(Process)

↓

lock(region)

↓

lock(FCM)

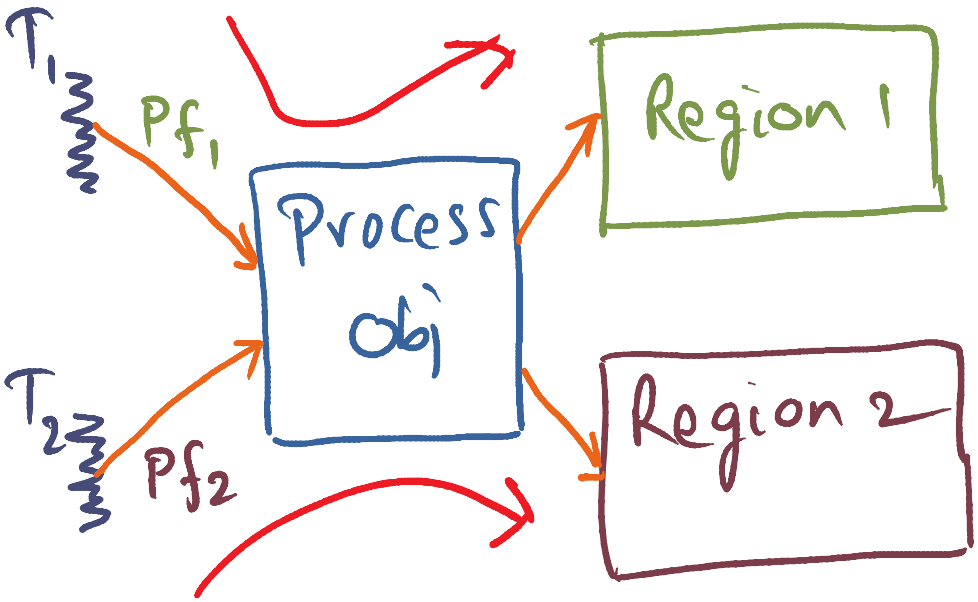
↓

lock(COR)

⇓

→ kills concurrency

Non Hierarchical Locking + Existence Guarantee



hierarchical locking:

lock(Process)

↓

lock(region)

↓

lock(FCM)

↓

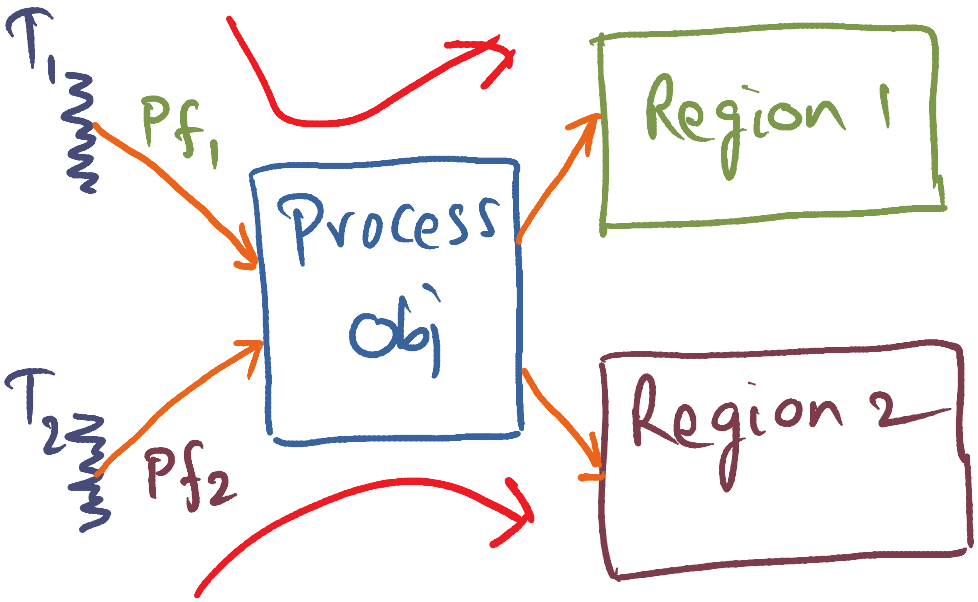
lock(COR)

⇓

- Kills concurrency

⇒ Bad idea

Non Hierarchical Locking + Existence Guarantee



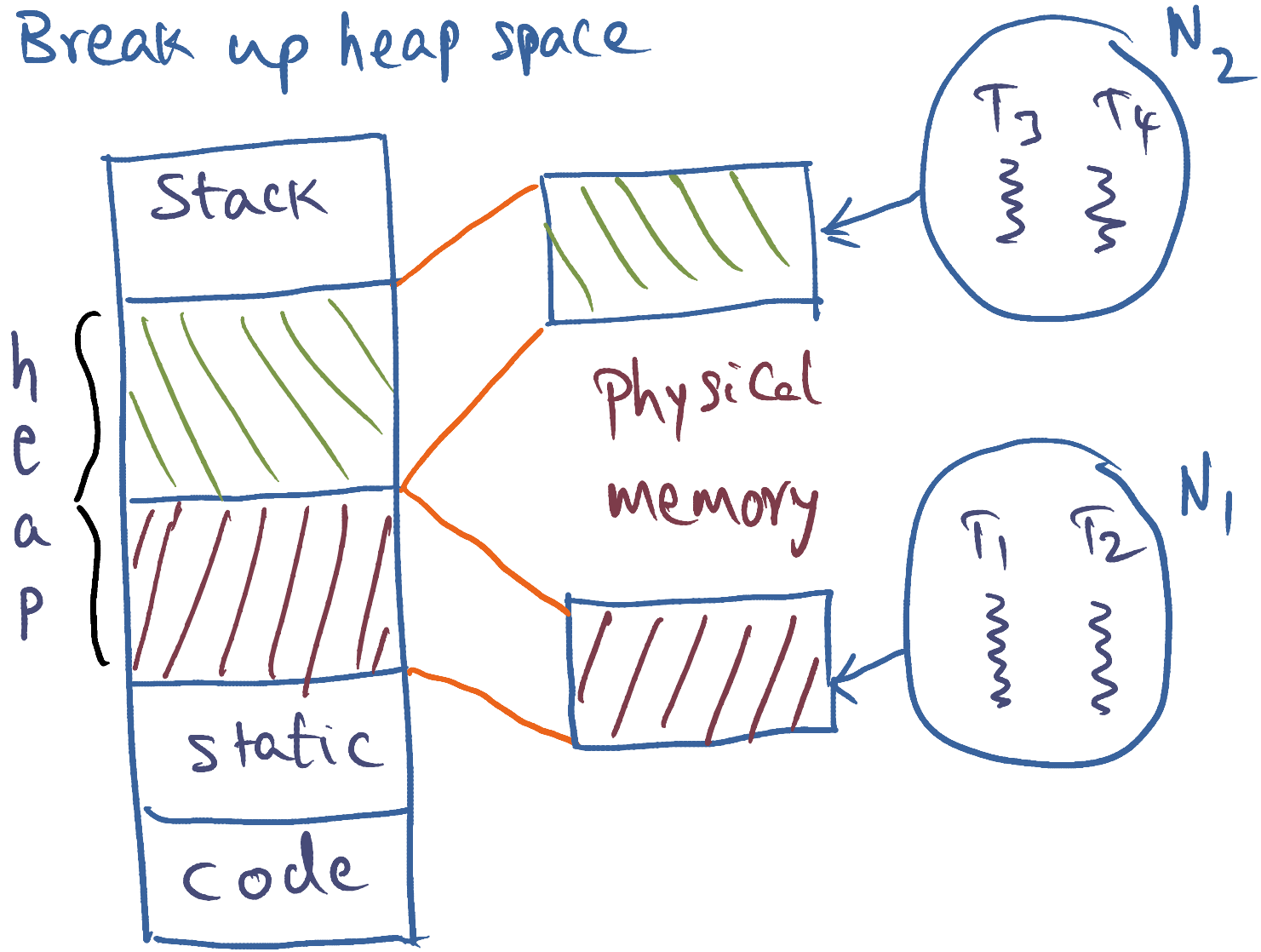
Use refCnt + Existence guarantee instead of hierarchical locking

hierarchical locking:

lock(Process)
↓
lock(region)
↓
lock(FCM)
↓
lock(COR)
⇓
- Kills concurrency
⇒ Bad idea

Dynamic Memory Allocation

Break up heap space



Logical address space of T_1, T_2, T_3, T_4

IPC

Object calls need IPC

- realized by PPC

⇒
- * local PPC no context switch → handoff scheduling
- * remote PPC full context switch

→ Similarity to LRPC

Tornado summary

Object oriented design for scalability

Multiple implementations of OS objects

optimize common case

* p.f. handling vs. region destruction

No hierarchical locking

Limited sharing of os data structures

Corey

- Main principle in structuring OS for shared memory multiprocessors
  - limit sharing kernel data structures, which both limits concurrency and increases contention.
- Corey OS research done at MIT, takes it one step further
  - involve the apps on top of the OS to give hints to the kernel.

Summary of Ideas in Corey System

Address Ranges in an APP

— Similar to "regions" of Tornado

shares

— "intent" to share with other threads

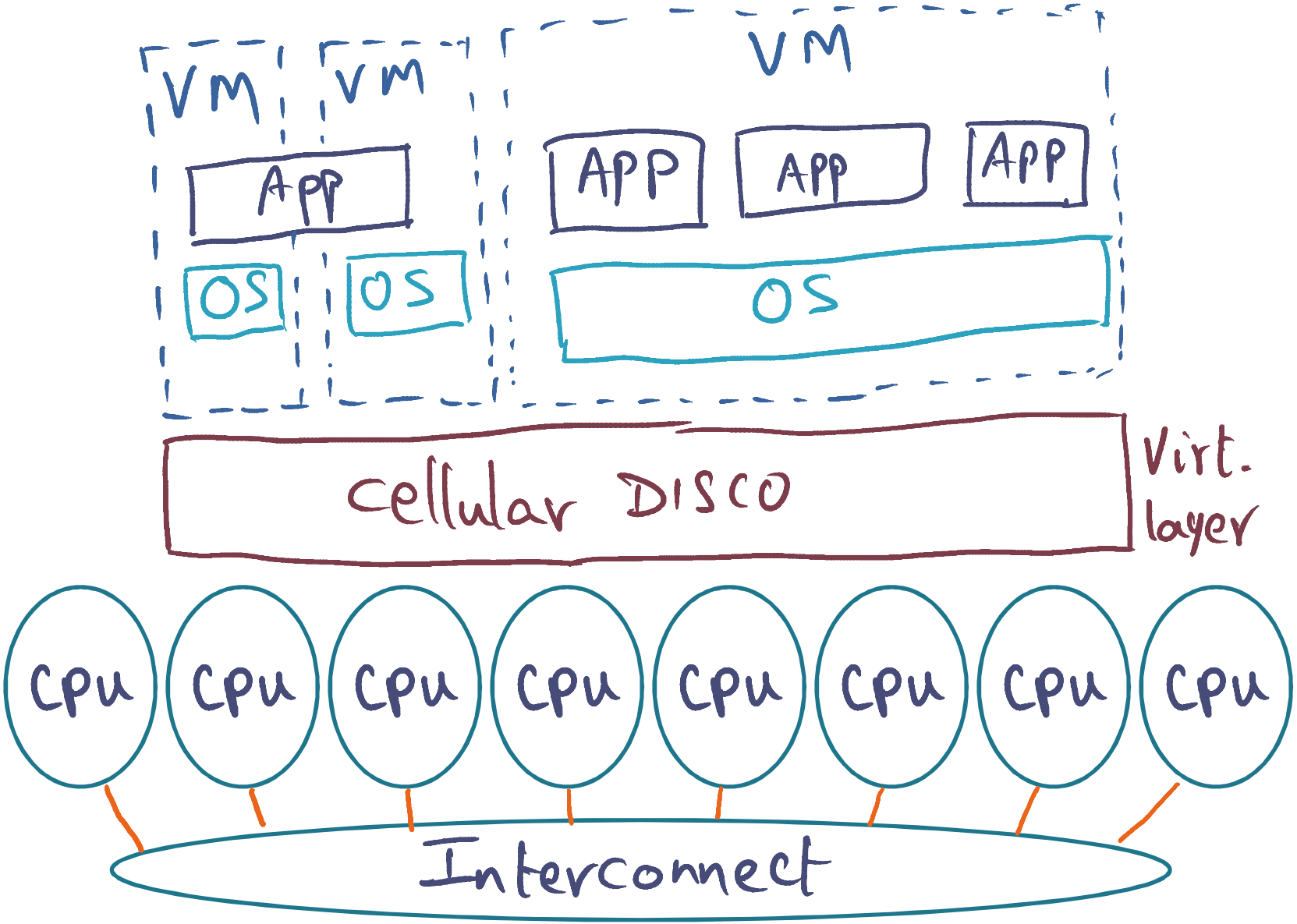
Dedicated Cores for Kernel activity

— Confine "locality" for Kernel threads to few cores

How to amortize OS development pain across generations of MPs?

- Hard work to ensure the scalability of the basic mechanisms
- Done anew for every new parallel architecture
  - different memory hierarchy compared to its predecessors
- Can we reduce this pain point of individually optimizing every OS that runs on the multiprocessor?
- Device drivers that form a BIG part of the code base of an OS
  - Do we have to re-implement them for every flavor of OS that runs on the shared memory multiprocessor?
  - Can we leverage standard third-party device drivers from OEMs to reduce the pain point?

Virtualization to the Rescue

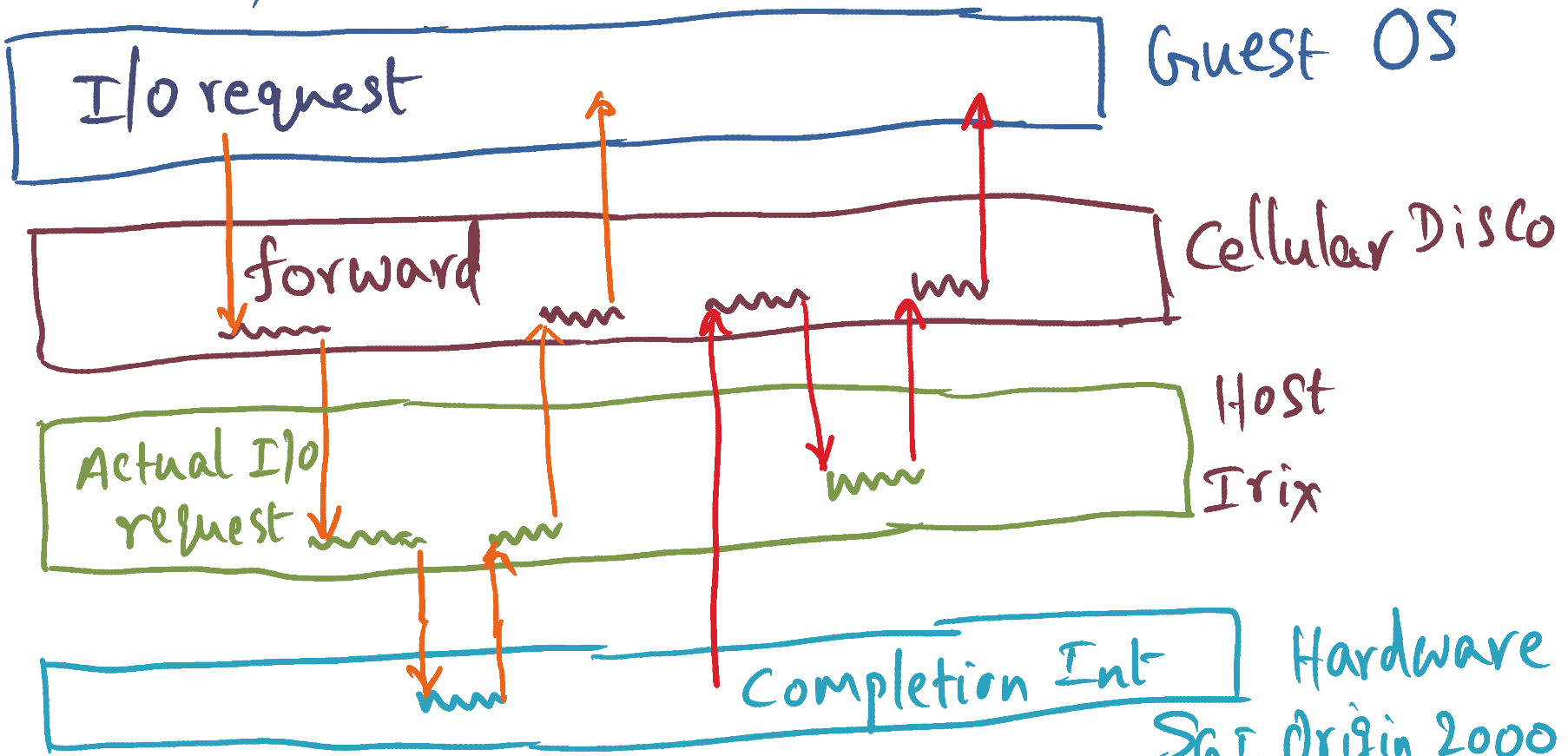


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done

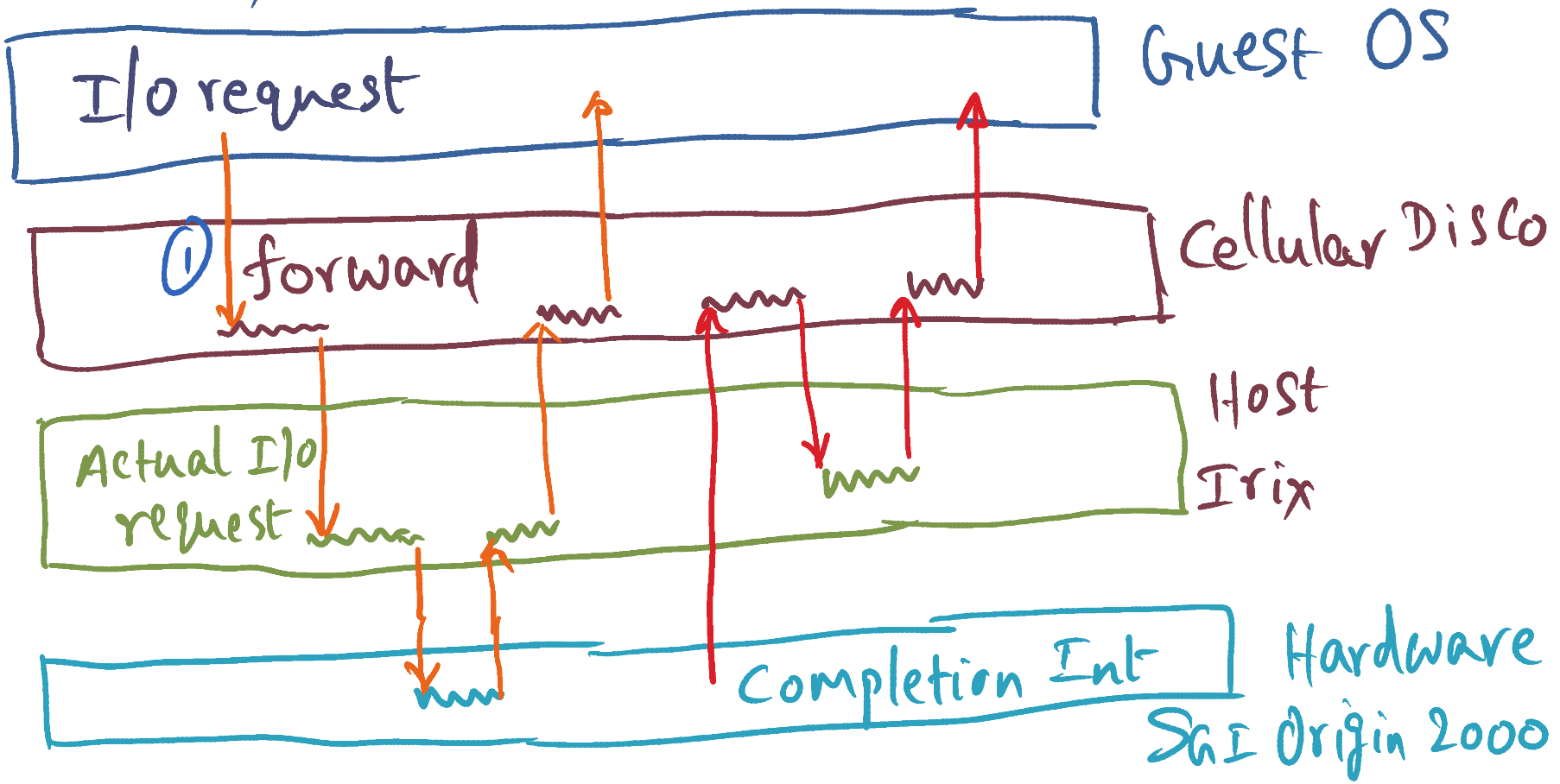


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done

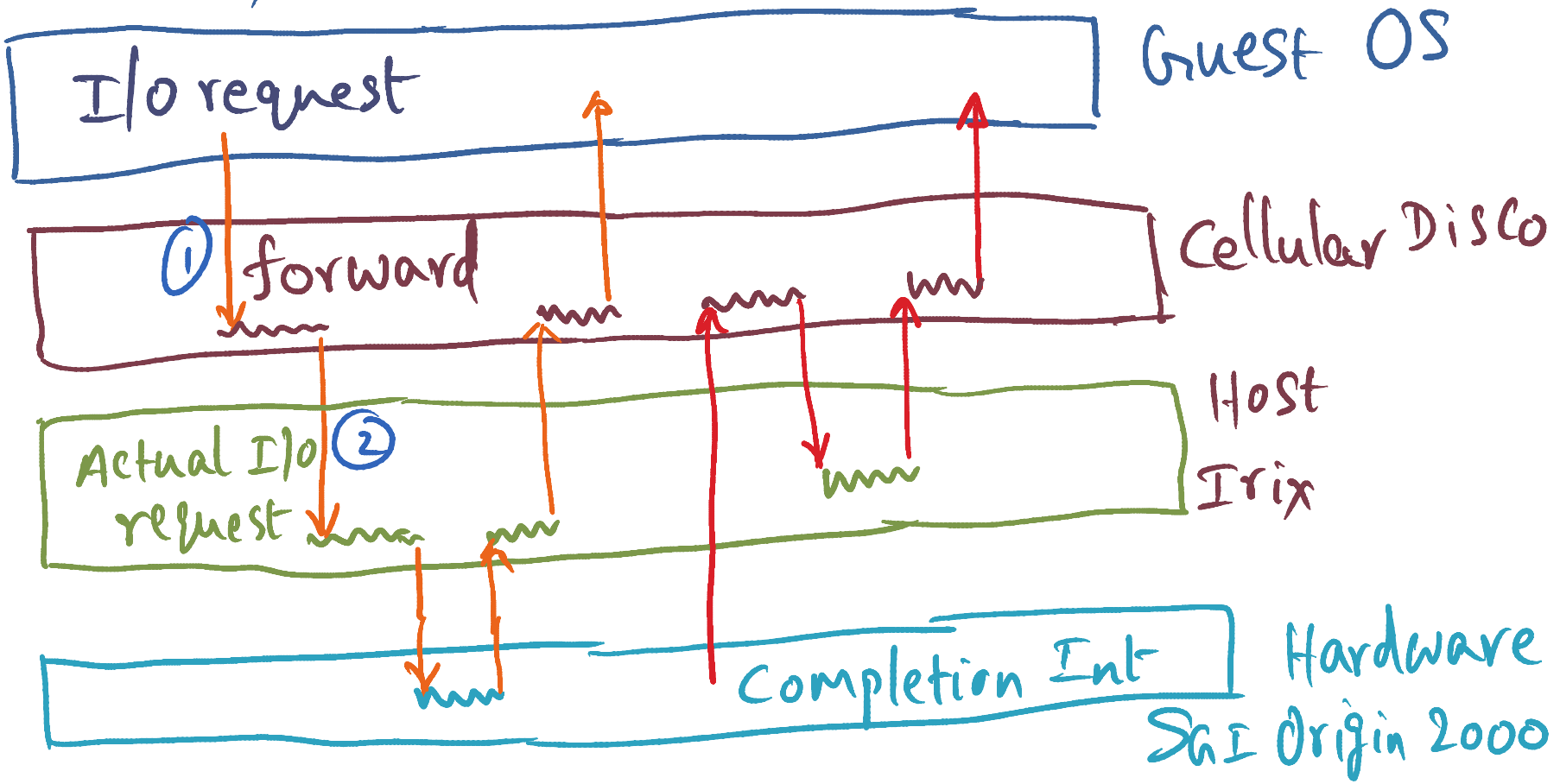


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done

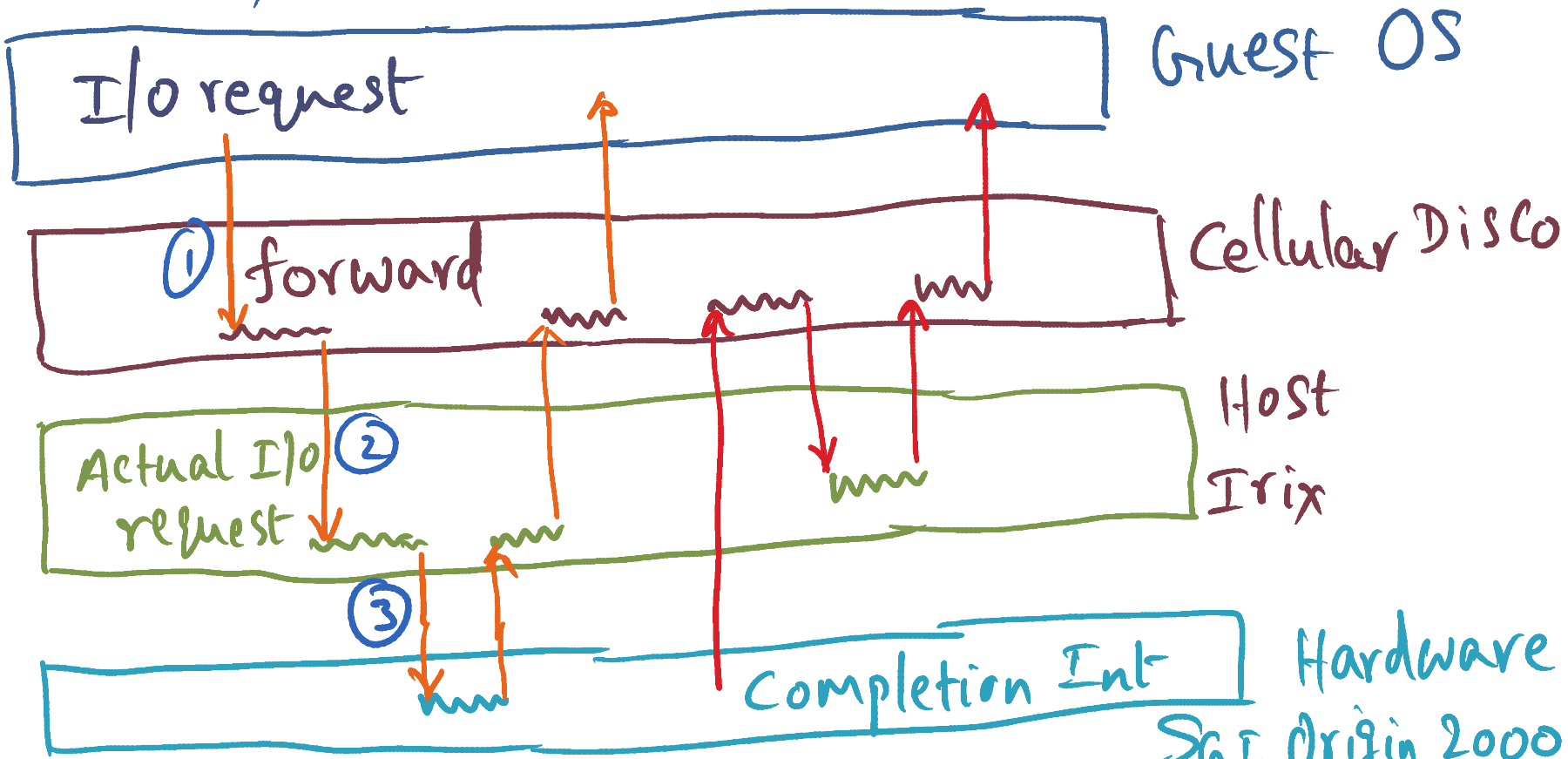


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done

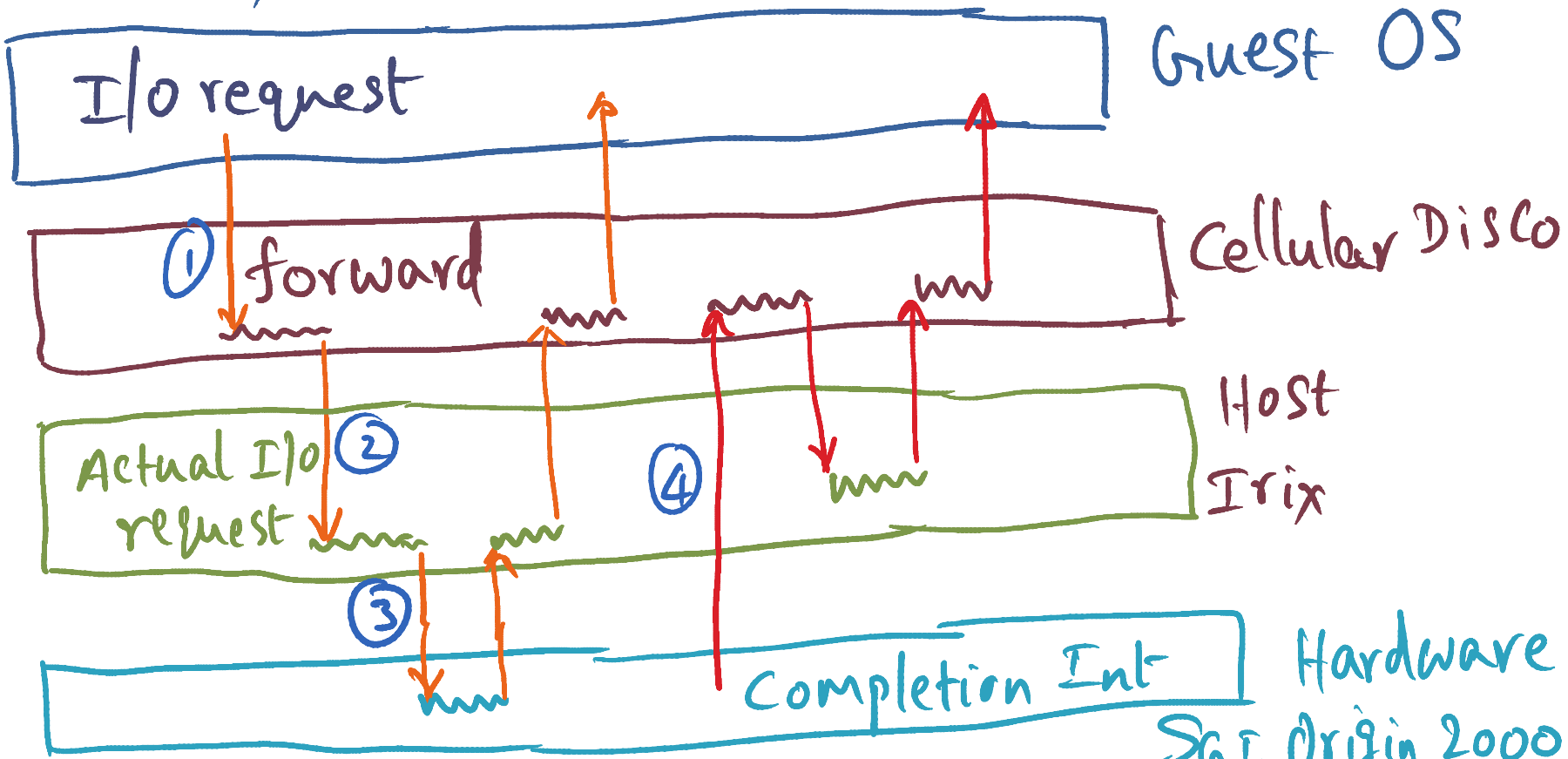


How is it Done?

Standard virtual machine trick

- "trap + emulate"

Shown by construction how it can be done

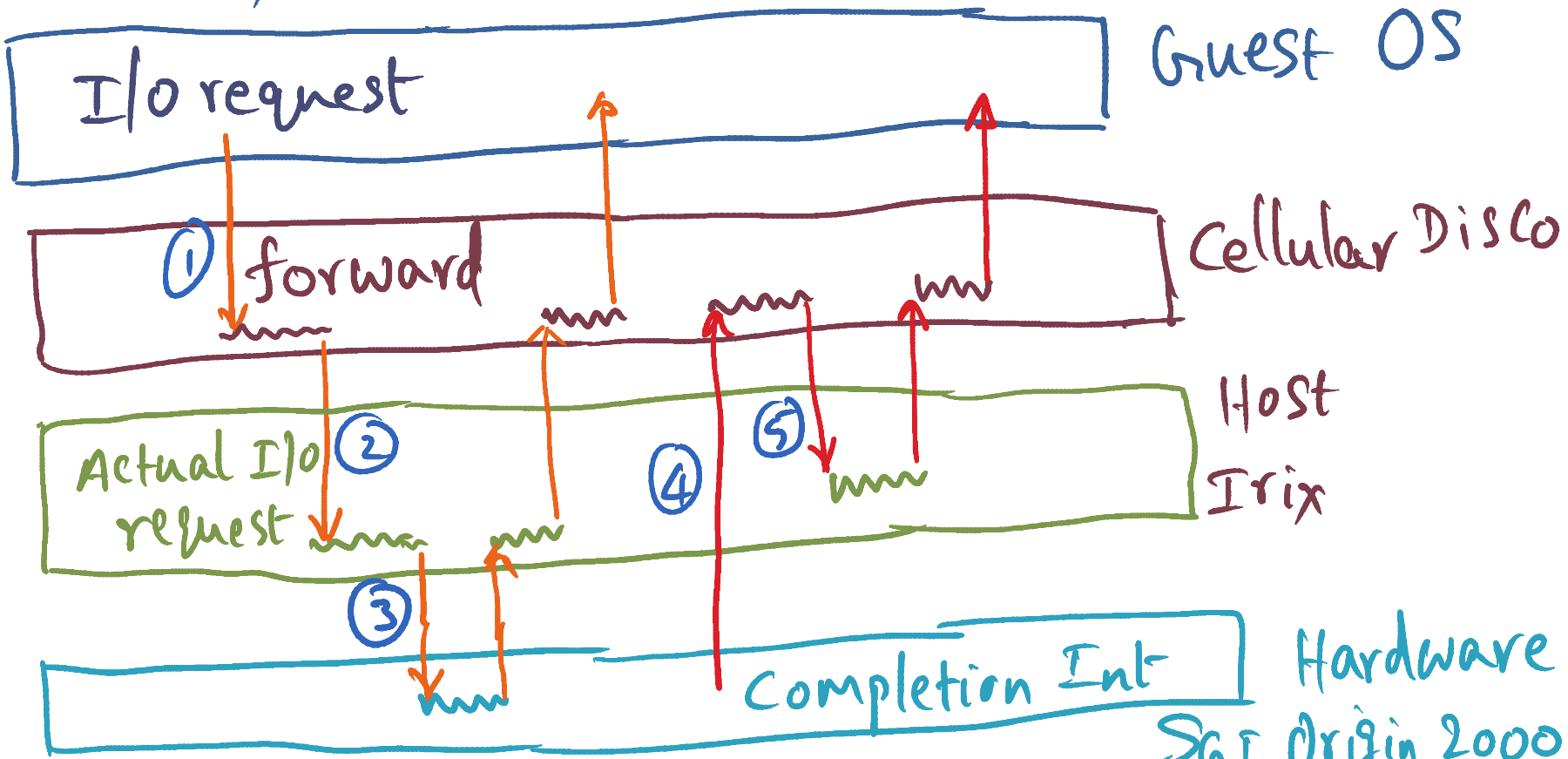


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done

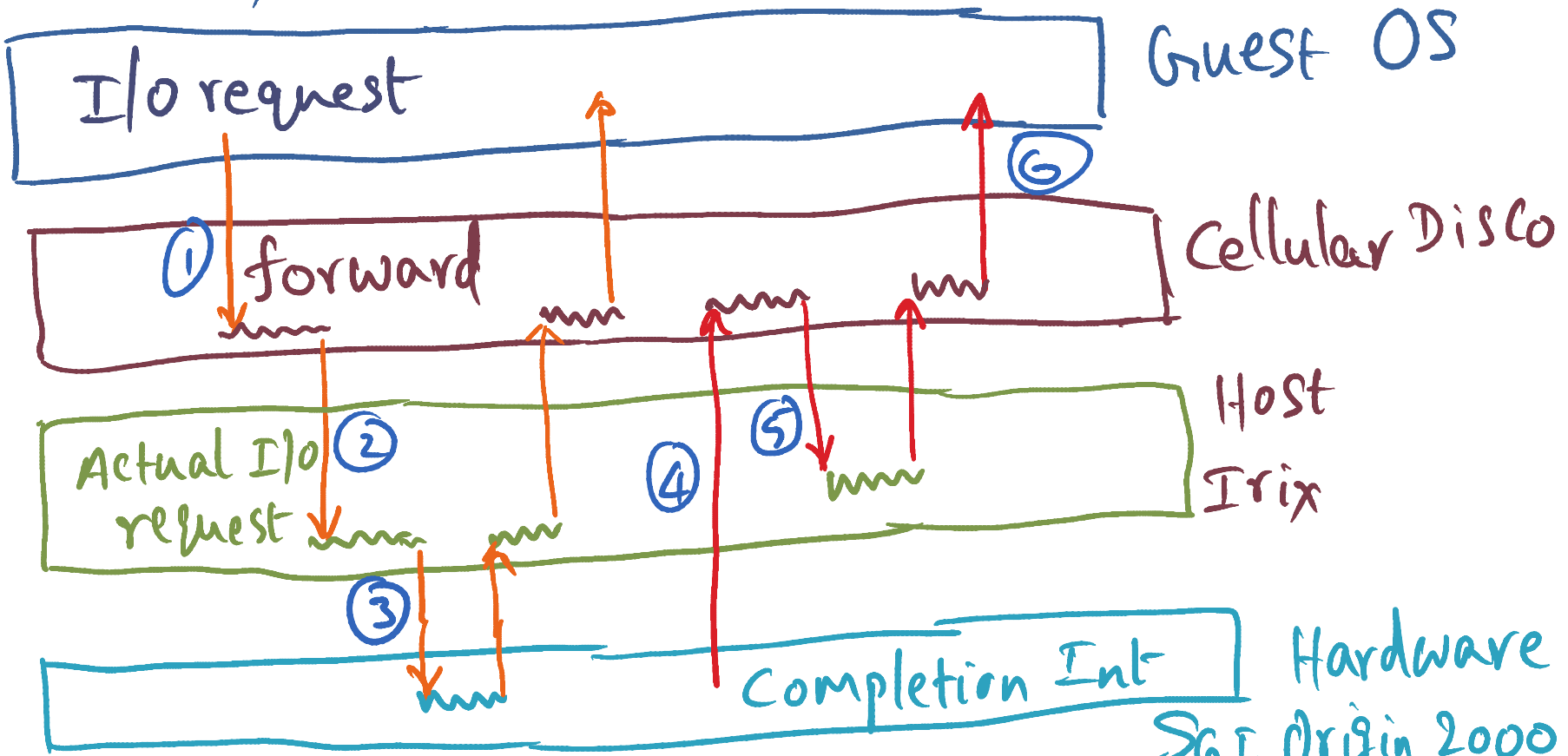


How is it Done?

Standard virtual machine trick

– "trap + emulate"

Shown by construction how it can be done



Steps in handling I/O

CD runs as a multi-threaded kernel process on top of the host OS (Irix in this case)

1. I/O request to CD: check permission; rewrite interrupt vectors
2. CD forwards the request to the dormant host OS
3. Host kernel issues the appropriate I/O request (3)
   - After IRIX initiates the I/O request, control returns to CD
   - puts the host kernel back into the dormant state.
4. Upon I/O completion the hardware raises an interrupt
   - Handled by CD because the interrupt vectors have been overwritten.
5. CD reactivates dormant host, making it look as if the I/O interrupt had just been posted
   - Allows host to properly do any cleanup of I/O completion
6. Finally, Cellular Disco posts a virtual interrupt to the virtual machine to notify it of the completion of its I/O request

Does it work?

Two questions to show the feasibility

- Can a virtual machine monitor manage resources as well as an operating system?
  - Yes (through construction and experimentation)
- Can virtualization overhead be kept low?
  - Usually within 10% (for many multiprocessor real workloads compared to native OS)

Approach similar to Liedtke's by construction

Summary

- Parallel systems complete
- I expect you to carefully read the papers we covered in this module listed
- Read and understand the performance sections of all the papers
- The actual results may not be that relevant