UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Scheduler Activations

Adopted some slides from www.cs.pdx.edu/~walpole/class/cs533/winter2007/slides/92.ppt

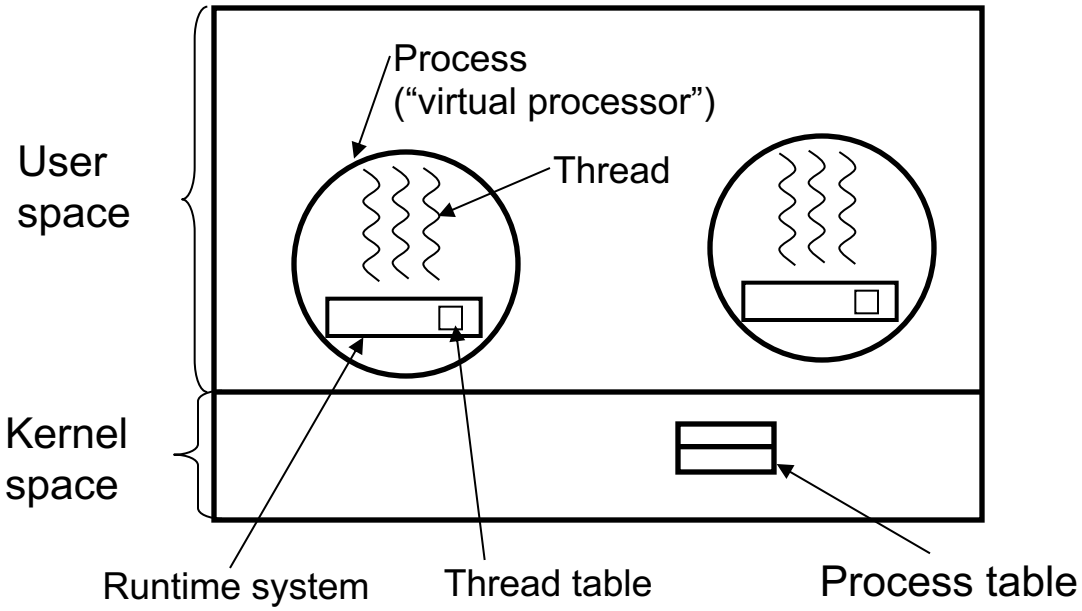
Managing Concurrency Using Threads

UCR

- User-level library
  - Management in application's address space
  - High performance and very flexible
  - Lack functionality
- Operating system kernel
  - Poor performance (when compared to user-level threads)
  - Poor flexibility
  - High functionality
- New system: kernel interface combined with user-level thread package
  - Same functionality as kernel threads
  - Performance and flexibility of user-level threads

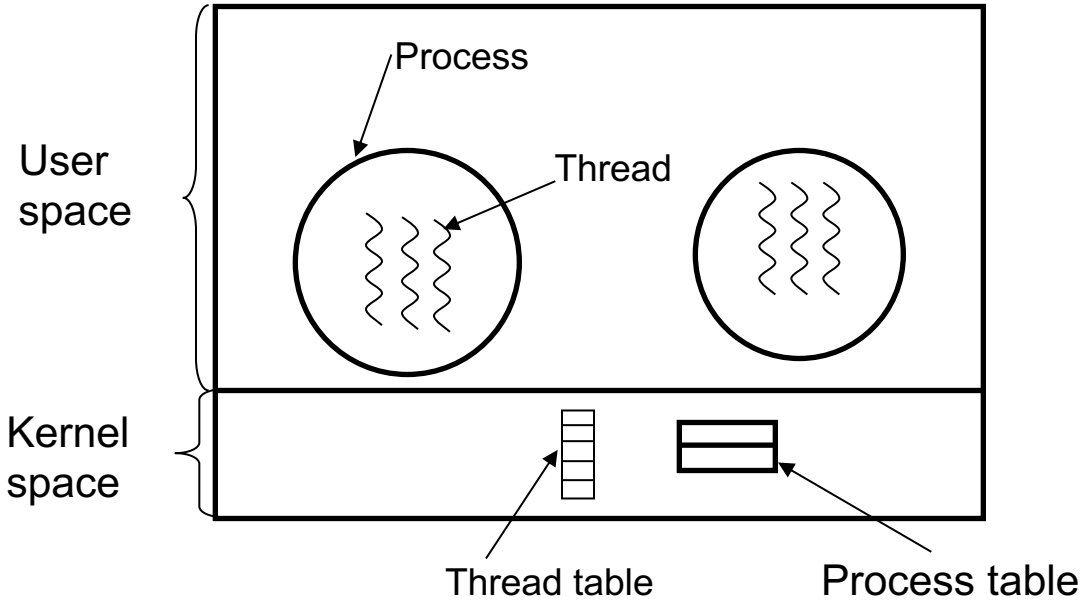
User-level Threads

- Thread management routines linked into application
- No kernel intervention == high performance
- Supports customized scheduling algorithms == flexible
- (Virtual) processor blocked during system services == lack of functionality
  - I/O, page faults, and multiprogramming cause entire process to block



Kernel Threads

- No system integration problems (system calls can be blocking calls) == high functionality
- Extra kernel trap and copy and check of all parameters on all thread operations == poor performance
- Kernel schedules thread from same or other address space (process)
- Single, general purpose scheduling algorithm == lack of flexibility



Kernel Threads Supporting User-level Threads

- **Question: Can we accomplish system integration by implementing user-level threads on top of kernel threads?**
- Typically one kernel thread per processor (virtual processor)
- What about multiple user-level threads run on top of one kernel-level thread?
- **Answer: No**

Goals (from paper)

- Functionality
  - No processor idles when there are ready threads
  - No priority inversion (high priority thread waiting for low priority one) when its ready
  - When a thread blocks, the processor can be used by another thread
- Performance
  - Closer to user threads than kernel threads
- Flexibility
  - Allow application level customization or even a completely different concurrency model

Problems

- User thread does a blocking call?
  - Application loses a processor!
- Scheduling decisions at user and kernel not coordinated
  - Kernel may de-schedule a thread at a bad time (e.g., while holding a lock)
  - Application may need more or less computing
- Solution?
  - Allow coordination between user and kernel schedulers

Scheduler activations

- Allow user level threads to act like kernel level threads/virtual processors
- Notify user level scheduler of relevant kernel events
  - Like what?
- Provide space in kernel to save context of user thread when kernel stops it
  - E.g., for I/O or to run another application

Kernel upcalls

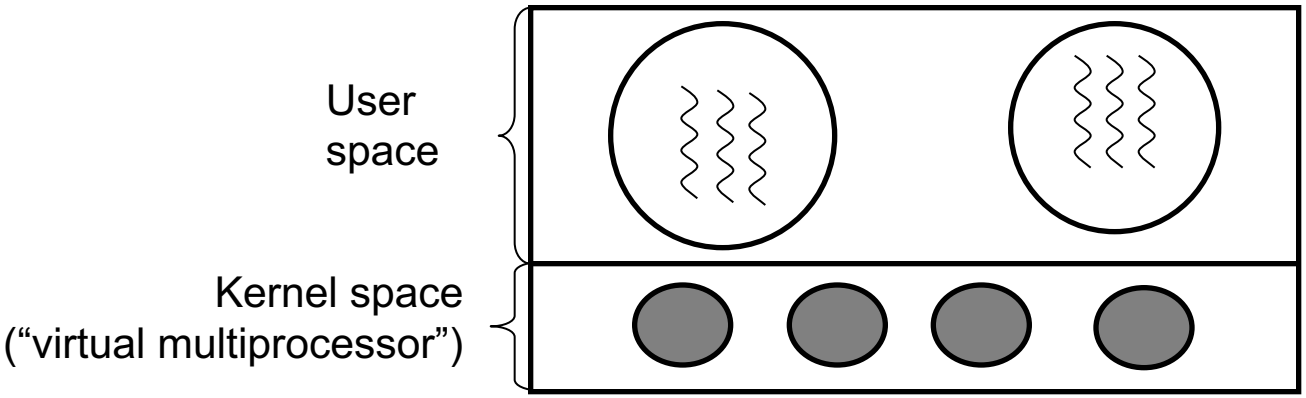
- New processor available
  - Reaction? Run time picks user thread to use it
- Activation blocked (e.g., for page fault)
  - Reaction? Runtime runs a different thread on the activation
- Activation unblocked
  - Activation now has two contexts
  - Running activation is preempted – why?
- Activation lost processor
  - Context remapped to another activation
- What do these accomplish?

Runtime->Kernel

- Informs kernel when it needs more resources, or when it is giving up some
- Could involve the kernel to preempt low priority threads
  - Only kernel can preempt
- Almost everything else is user level!
  - Performance of user-level, with the advantages of kernel threads!

Virtual Multiprocessor

- Application knows how many and which processors allocated to it by kernel.
- Application has complete control over which threads are running on processors.
- Kernel notifies thread scheduler of events affecting address space.
- Thread scheduler notifies kernel regarding processor allocation.



Scheduler Activations

- Vessels for running user-level threads
- One scheduler activation per processor assigned to address space.
- Also created by kernel to perform upcall into application's address space
  - "Scheduler activation has blocked"
  - "Scheduler activation has unblocked"
  - "Add this processor"
  - "Processor has been preempted"
- **Result:** Scheduling decisions made at user-level and application is free to build any concurrency model on top of scheduler activations.

Scheduler activations (2)

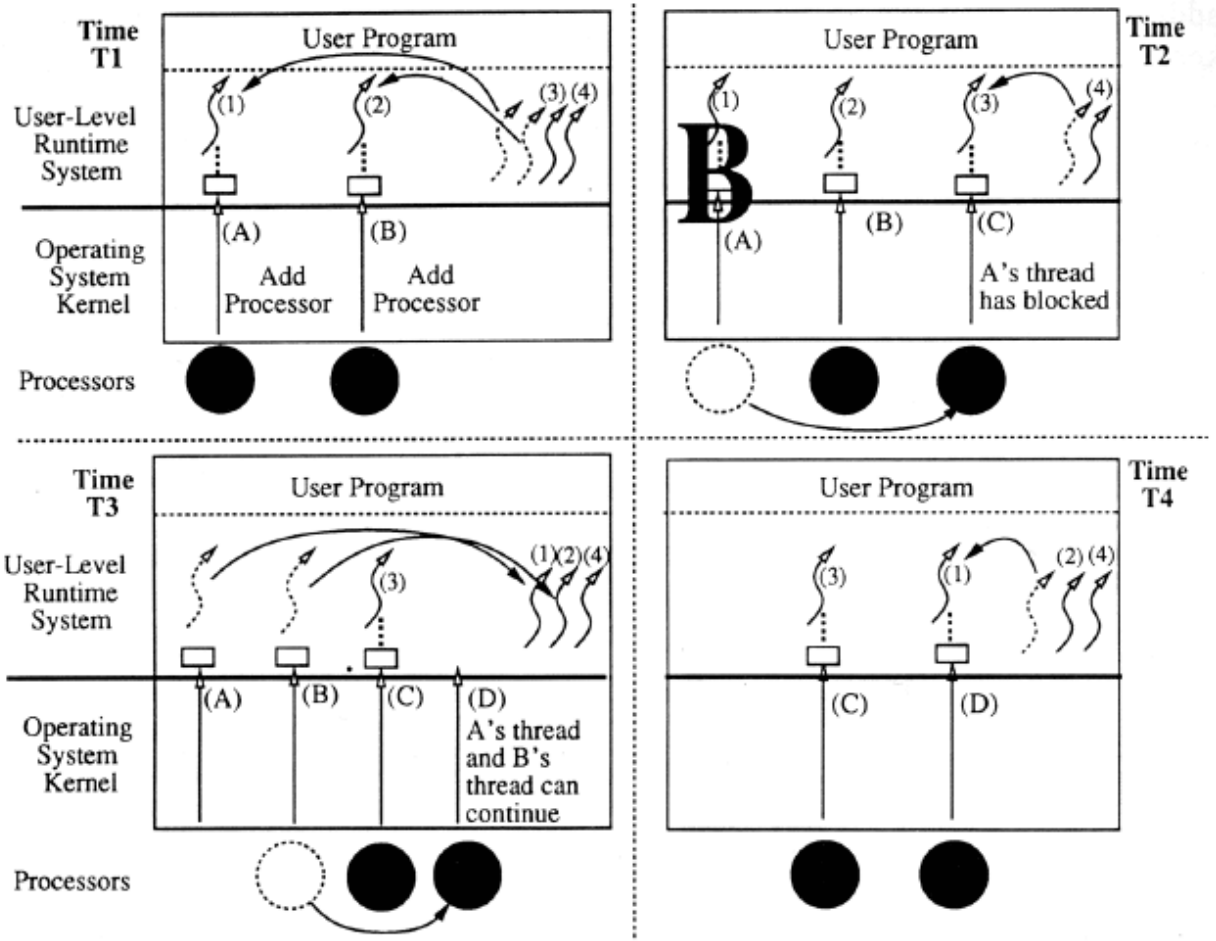


Fig. 1. Example: I/O request/completion.

Preemptions in critical sections

- Runtime checks during upcall whether preempted user thread was running in a critical section
  - Continues the user thread using a user level context switch in this case
    - Once lock is released, it switches back to original thread
    - Keep track of critical sections using a hash table of section begin/end addresses

Implementation

- Scheduler activations added to Topaz kernel thread management.
  - Performs upcalls instead of own scheduling.
  - Explicit processor allocation to address spaces.
- Modifications to FastThreads user-level thread package
  - Processing of upcalls.
  - Resume interrupted critical sections.
  - Pass processor allocation information to Topaz.

Performance

- Thread performance without kernel involvement similar to FastThreads before changes.
- Upcall performance significantly worse than Topaz threads.
  - Untuned implementation.
  - Topaz in assembler, this system in Modula-2+.
- Application performance
  - Negligible I/O: As quick as original FastThreads.
  - With I/O: Performs better than either FastThreads or Topaz threads.

Application Performance (negligible I/O)

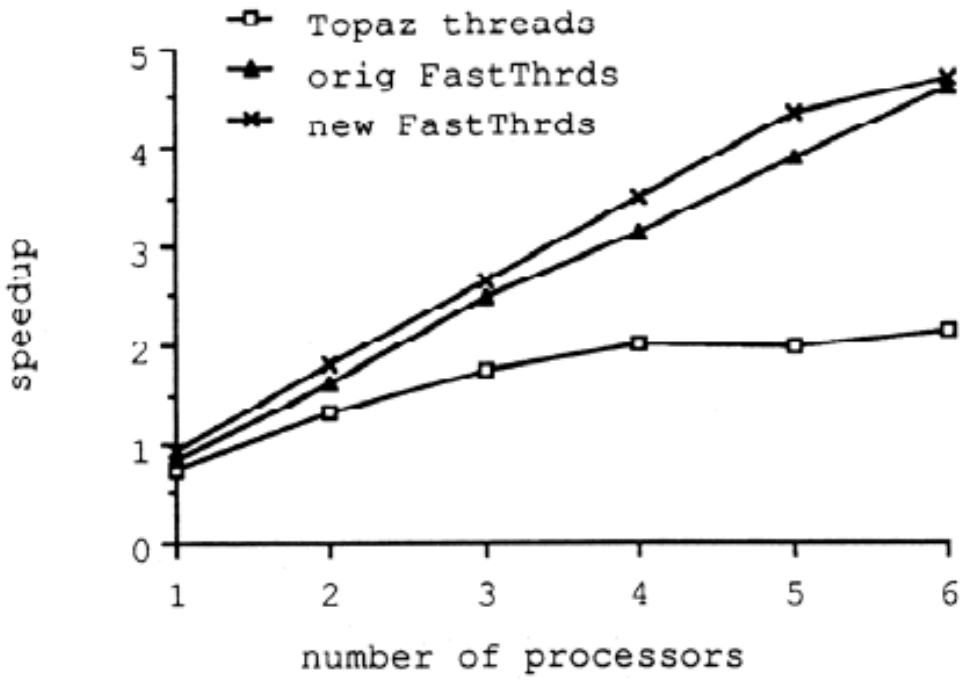


Fig. 2. Speedup of N-Body application versus number of processors, 100% of memory available.

Application Performance (with I/O)

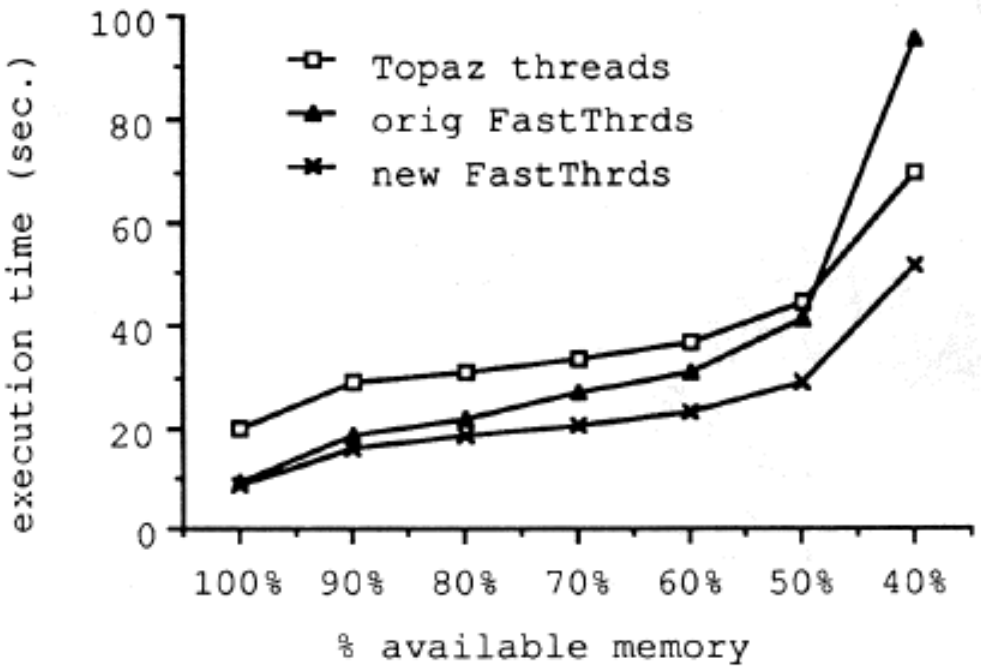


Fig. 3. Execution time of N-Body application versus amount of available memory, 6 processors.

Discussion

- Summary:
  - Get user level thread performance but with scheduling abilities of kernel level threads
  - Main idea: coordinating user level and kernel level scheduling through scheduler activations
- Limitations
  - Upcall performance (5x slowdown)
  - Performance analysis limited
- Connections to exo-kernel/spin/microkernels?

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Advanced Operating Systems (CS 202)

Memory Consistency, Cache Coherence and Synchronization

(some cache coherence slides adapted from Ian Watson; some memory consistency slides from Sarita Adve)

Classic Example

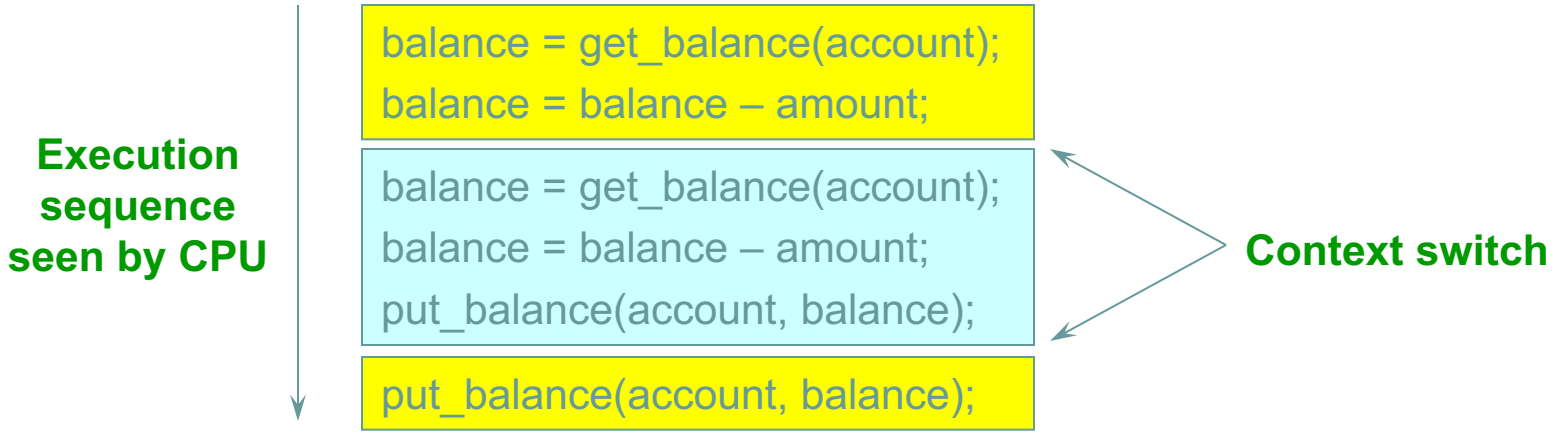
- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

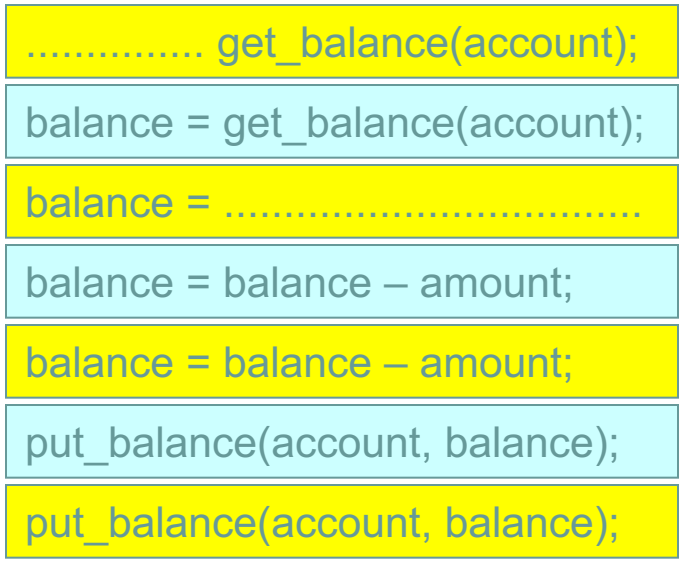


- What is the balance of the account now?

How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever



Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?

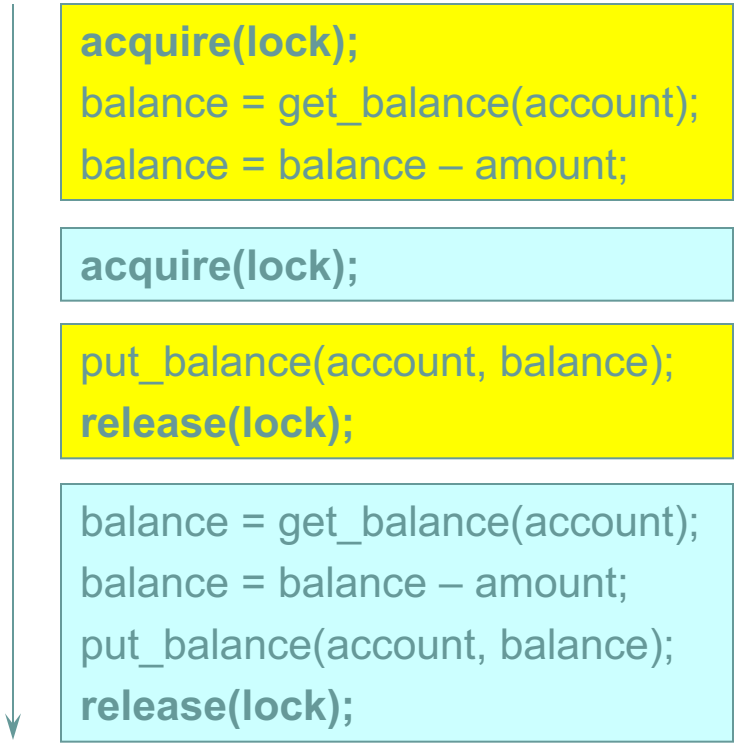
- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others

- What requirements would you place on a critical section?

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section



Using Test-And-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?

Overview

- Before we talk deeply about synchronization
  - Need to get an idea about the memory model in shared memory systems
  - Is synchronization only an issue in multi-processor systems?
- What is a shared memory processor (SMP)?
- Shared memory processors
  - Two primary architectures:
    - Bus-based/local network shared-memory machines (small-scale)
    - Directory-based shared-memory machines (large-scale)

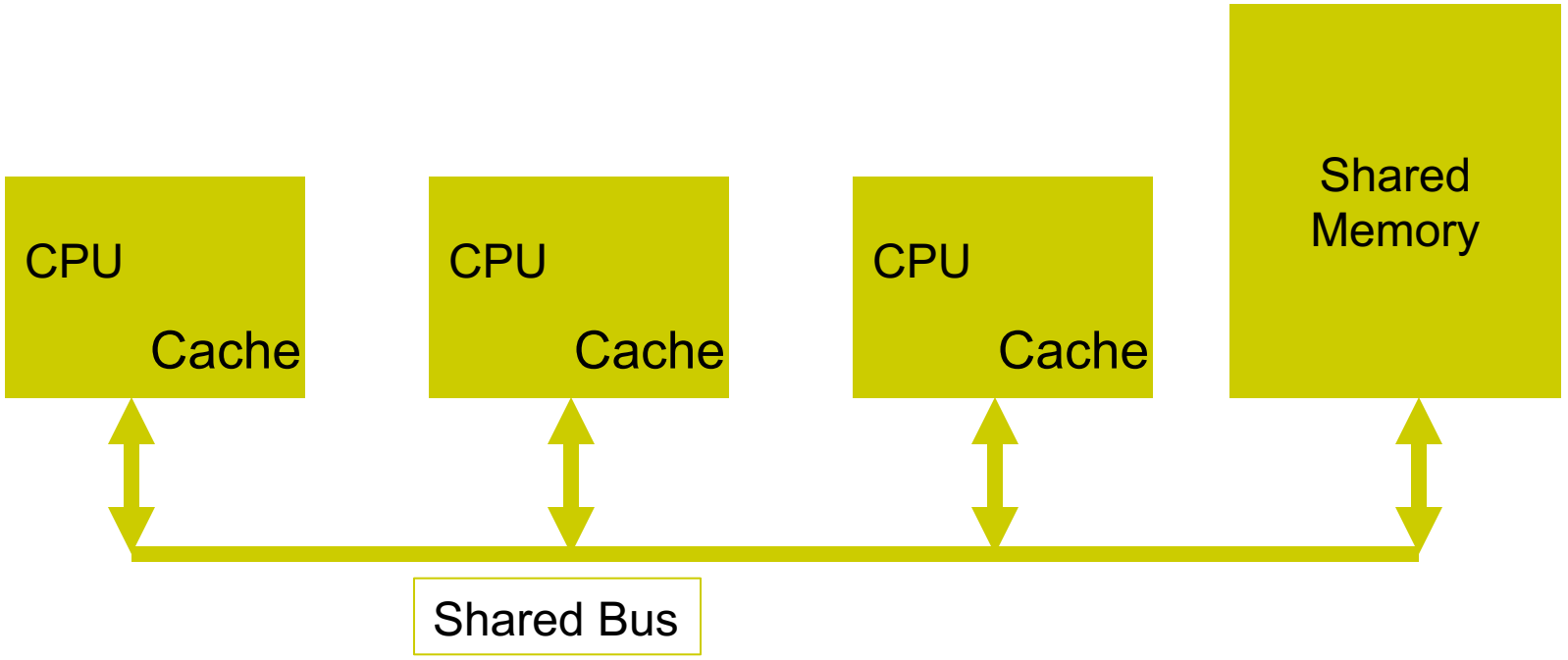
Plan…

- Introduce and discuss cache coherence
- Discuss basic synchronization, up to MCS locks (from the paper we are reading)
- Introduce memory consistency and implications
- Is this an architecture class???
  - The same issues manifest in large scale distributed systems

Crash course on cache coherence

Bus-based Shared Memory Organization

Basic picture is simple :-



Organization

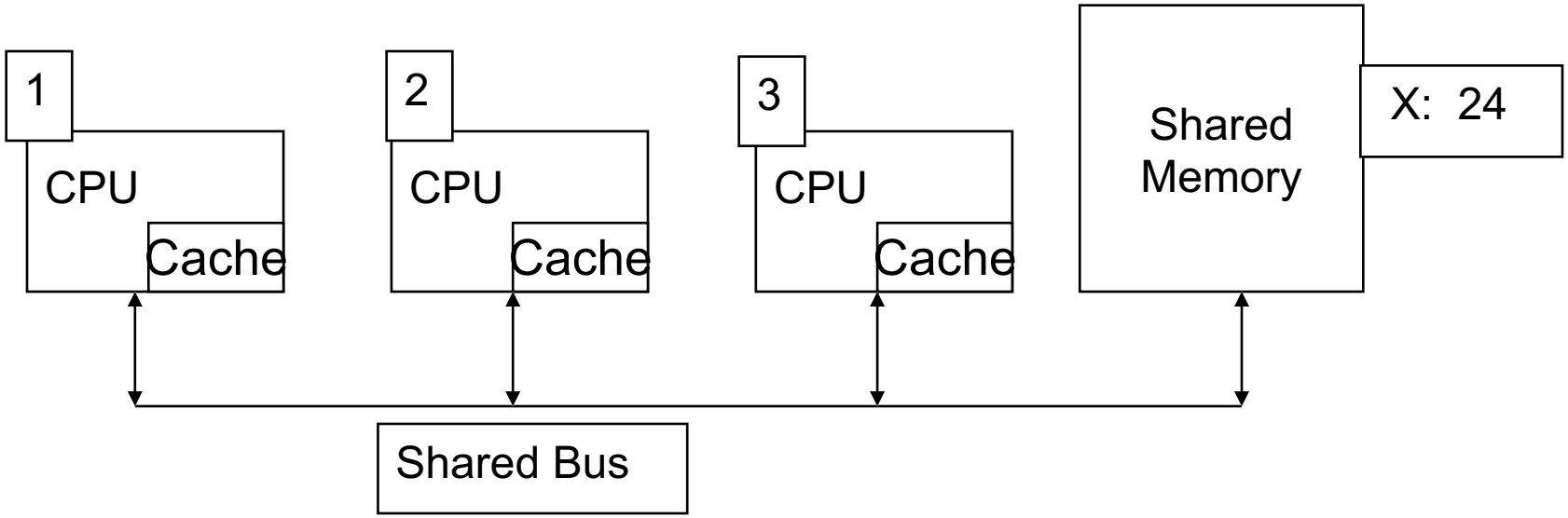
- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- Could be multiple memory elements
- For now, assume that each CPU has only a single level of cache

Problem of Memory Coherence

UCR

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough
- What happens if two updates happen at (nearly) the same time?
  - Can two different processors see them out of order?

Example



Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?
Memory and processor 2 think it is 24
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough

Cache Coherence

- Try to make the system behave as if there are no caches!
- How? Idea: Try to make every CPU know who has a copy of its cached data?
  - too complex!
- More practical:
  - Snoopy caches
    - Each CPU snoops memory bus
    - Looks for read/write activity concerned with data addresses which it has cached.
      - What does it do with them?
    - This assumes a bus structure where all communication can be seen by all.
- More scalable solution: 'directory based' coherence schemes

Snooping Protocols

- Write Invalidate
  - CPU with write operation sends invalidate message
  - Snooping caches invalidate their copy
  - CPU writes to its cached copy
    - Write through or write back?
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.

Snooping Protocols

- Write Update
  - CPU with write updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.
- Harder problem for arbitrary networks

Update or Invalidate?

- Which should we use?
- Bus bandwidth is a precious commodity in shared memory multi-processors
  - Contention/cache interrogation can lead to 10x or more drop in performance
  - (also important to minimize false sharing)
- Therefore, invalidate protocols used in most commercial SMPs

Cache Coherence summary

- Reads and writes are atomic
  - What does atomic mean?
    - As if there is no cache
- Some magic to make things work
  - Have performance implications
  - …and therefore, have implications on performance of programs