CS/EE3810: Computer Organization

Lecture 17: Cache-Coherence and synchronization

Anton Burtsev

November, 2022

Memory organization

Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)

- Multiple processors connected to a single centralized memory – since all processors see the same memory organization ✉ uniform memory access (UMA)

- Shared-memory because all processors can access the entire memory address space

- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

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

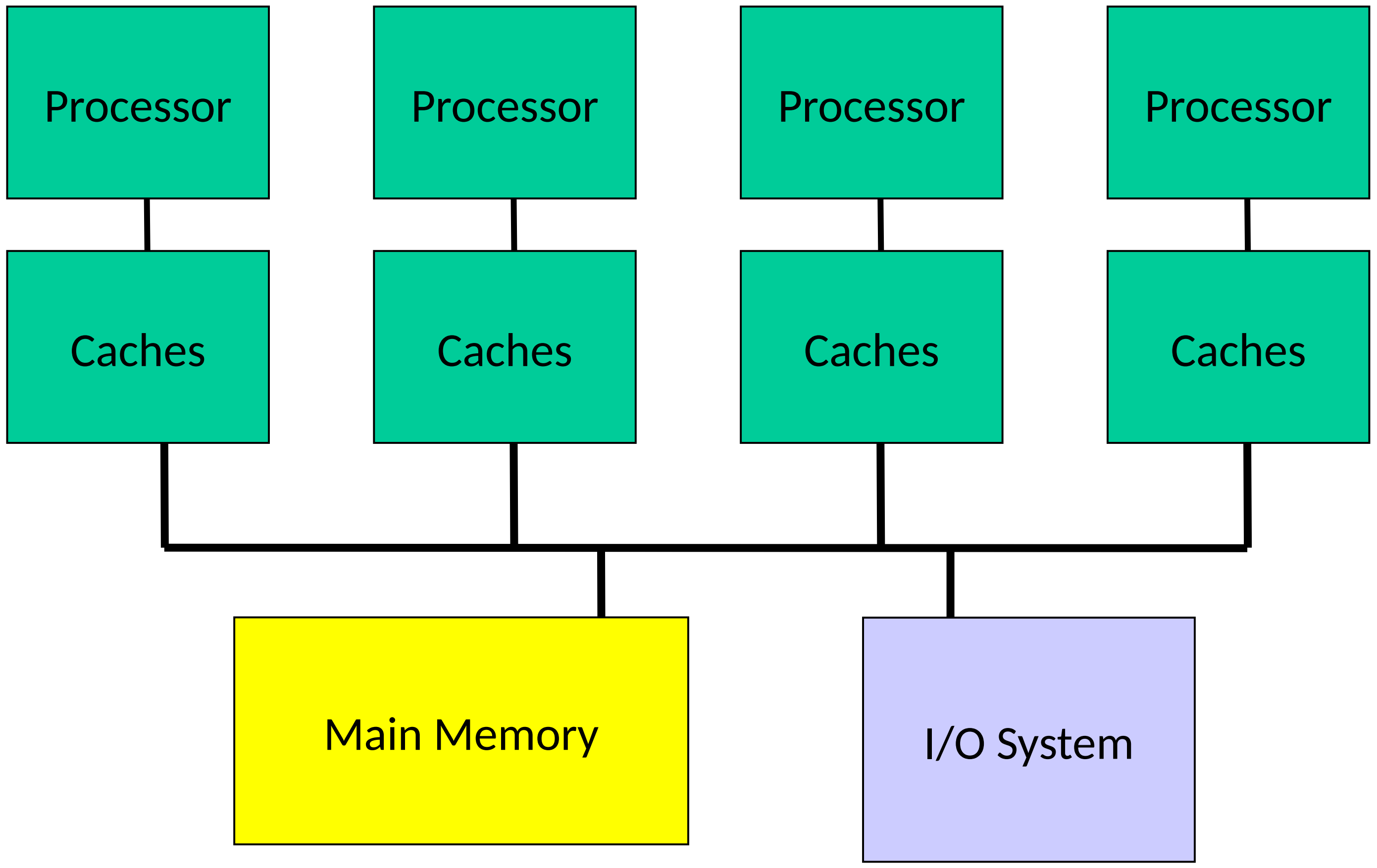
  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

Multiprocs -- Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)
- Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)
- Shared-memory because all processors can access the entire memory address space
- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

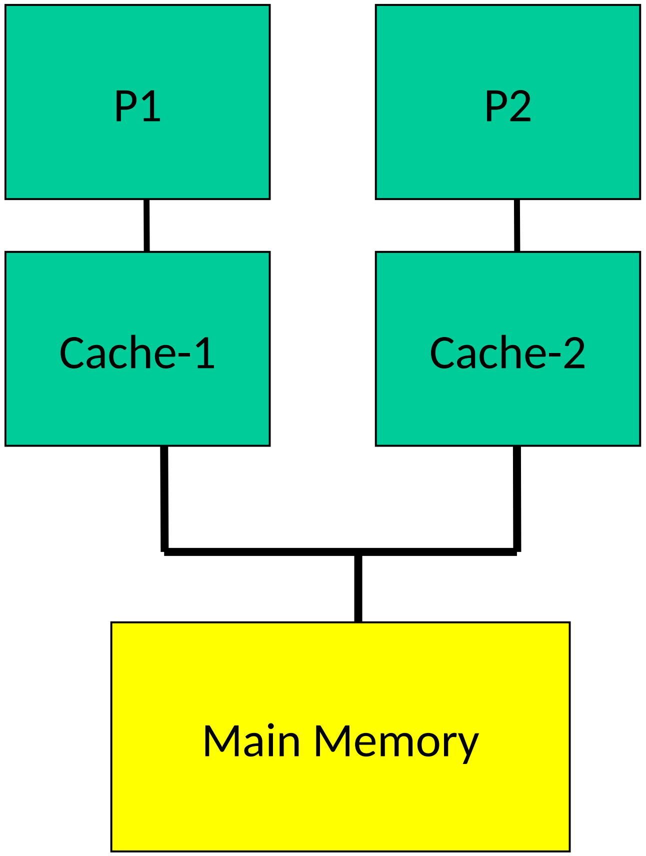
Snooping-Based Protocols

- 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.



Example

- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
- P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state



- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
- P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated

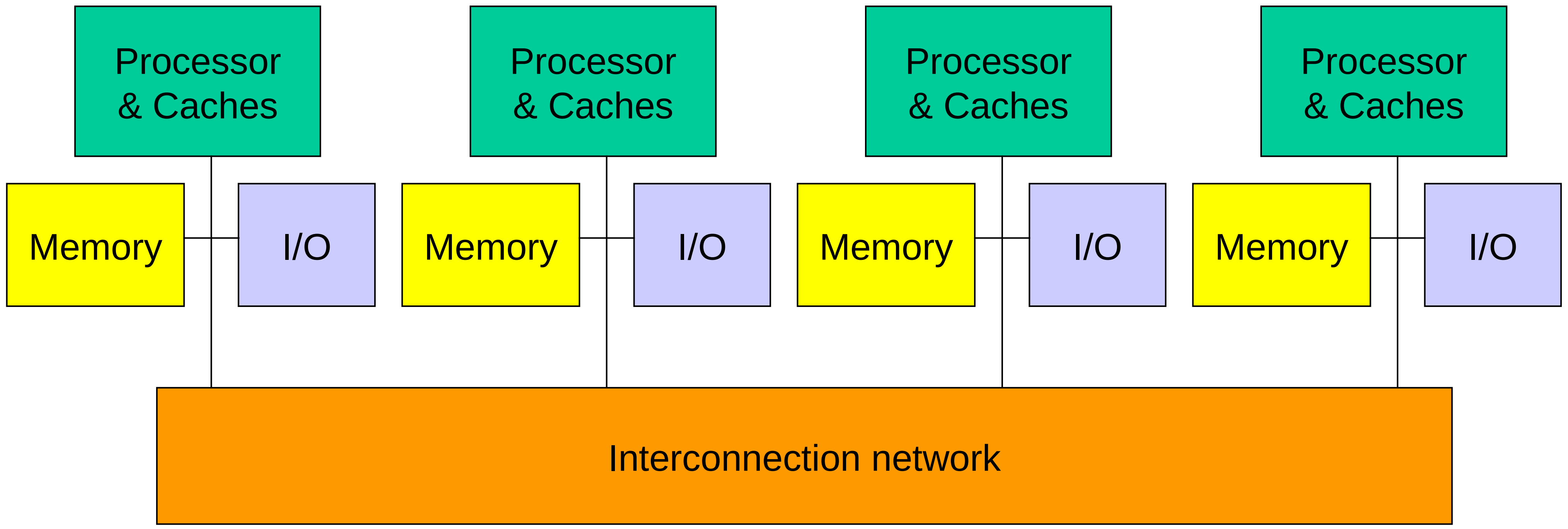
Example

Request	Cache Hit/Miss	Request on the bus	Who responds	State in Cache 1	State in Cache 2	State in Cache 3	State in Cache 4
				Inv	Inv	Inv	Inv
P1: Rd X	Miss	Rd X	Memory	S	Inv	Inv	Inv
P2: Rd X	Miss	Rd X	Memory	S	S	Inv	Inv
P2: Wr X	Perms Miss	Upgrade X	No response. Other caches invalidate.	Inv	M	Inv	Inv
P3: Wr X	Write Miss	Wr X	P2 responds	Inv	Inv	M	Inv
P3: Rd X	Read Hit	-	-	Inv	Inv	M	Inv
P4: Rd X	Read Miss	Rd X	P3 responds. Mem wrtbk	Inv	Inv	S	S

Multiprocs -- Memory Organization - II

- For higher scalability, memory is distributed among processors → distributed memory multiprocessors
- If one processor can directly address the memory local to another processor, the address space is shared → distributed shared-memory (DSM) multiprocessor
- If memories are strictly local, we need messages to communicate data → cluster of computers or multicomputers
- Non-uniform memory architecture (NUMA) since local memory has lower latency than remote memory

Distributed Memory Multiprocessors



Synchronization

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
- Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update

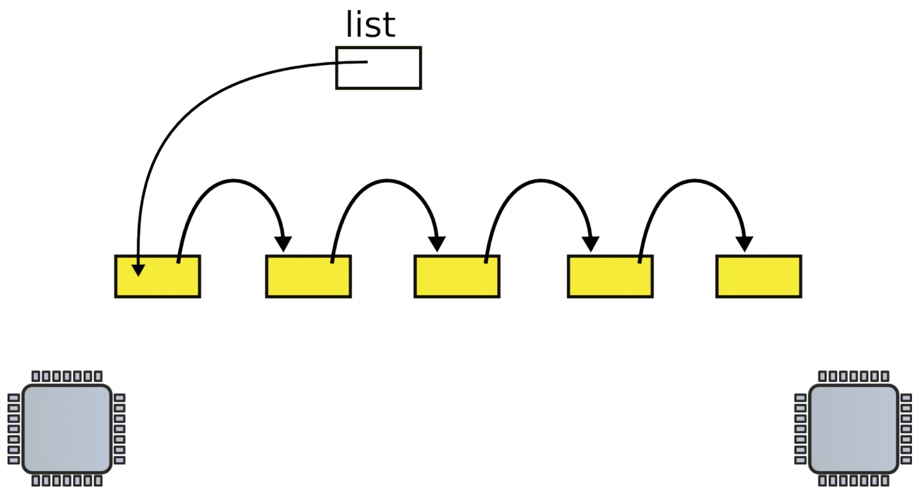
Race conditions

- Example:
  - Global list of outstanding requests
  - Each thread can add requests to the list

List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

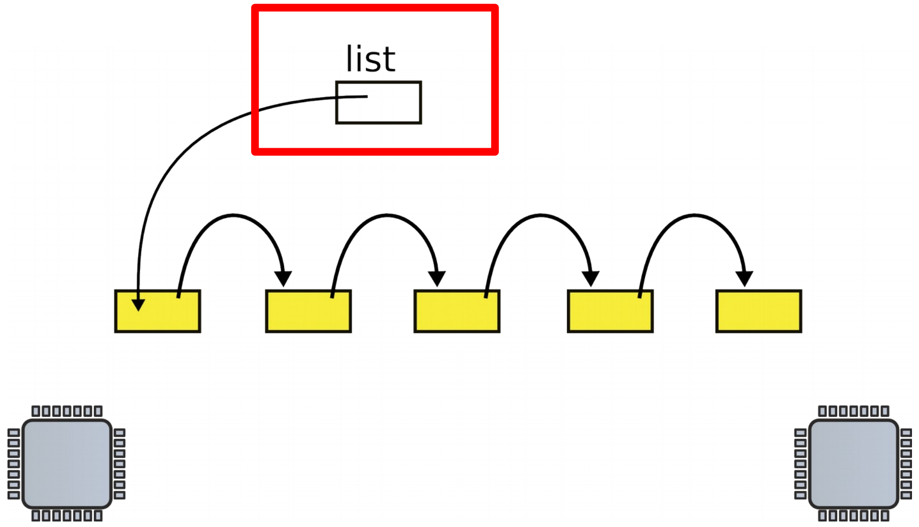
- List
  - One data element
  - Pointer to the next element



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

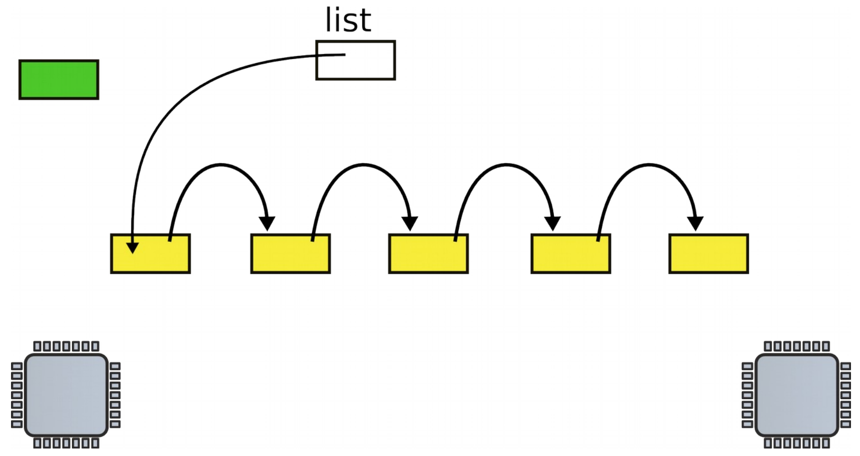
- Global head



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

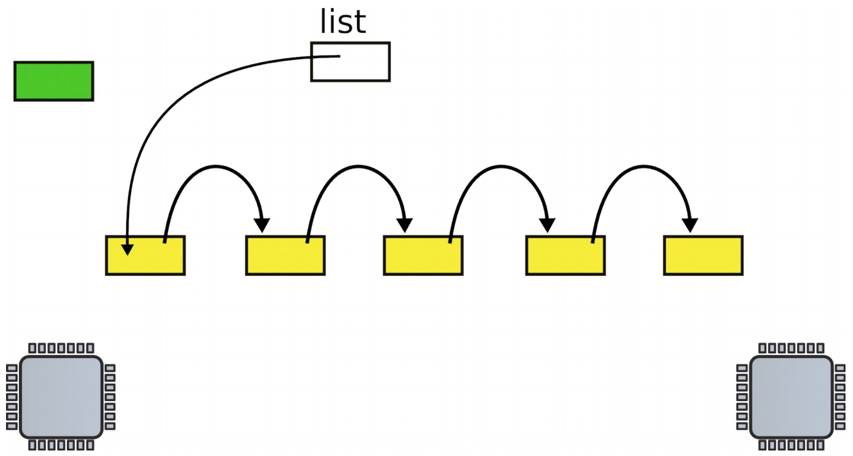
- Insertion
  - Allocate new list element



List implementation (no locks)

```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
  struct list *l;

  l = malloc(sizeof *l);
  l->data = data;
  l->next = list;
  list = l;
}
```

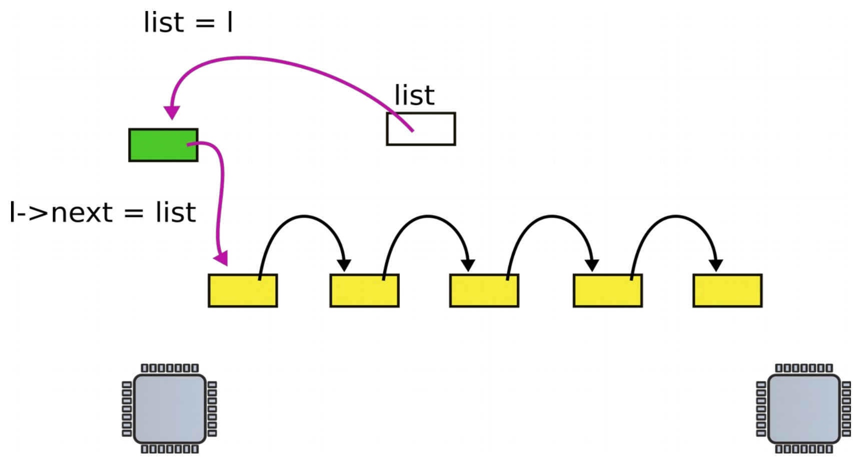
- Insertion
  - Allocate new list element
  - Save data into that element



List implementation (no locks)

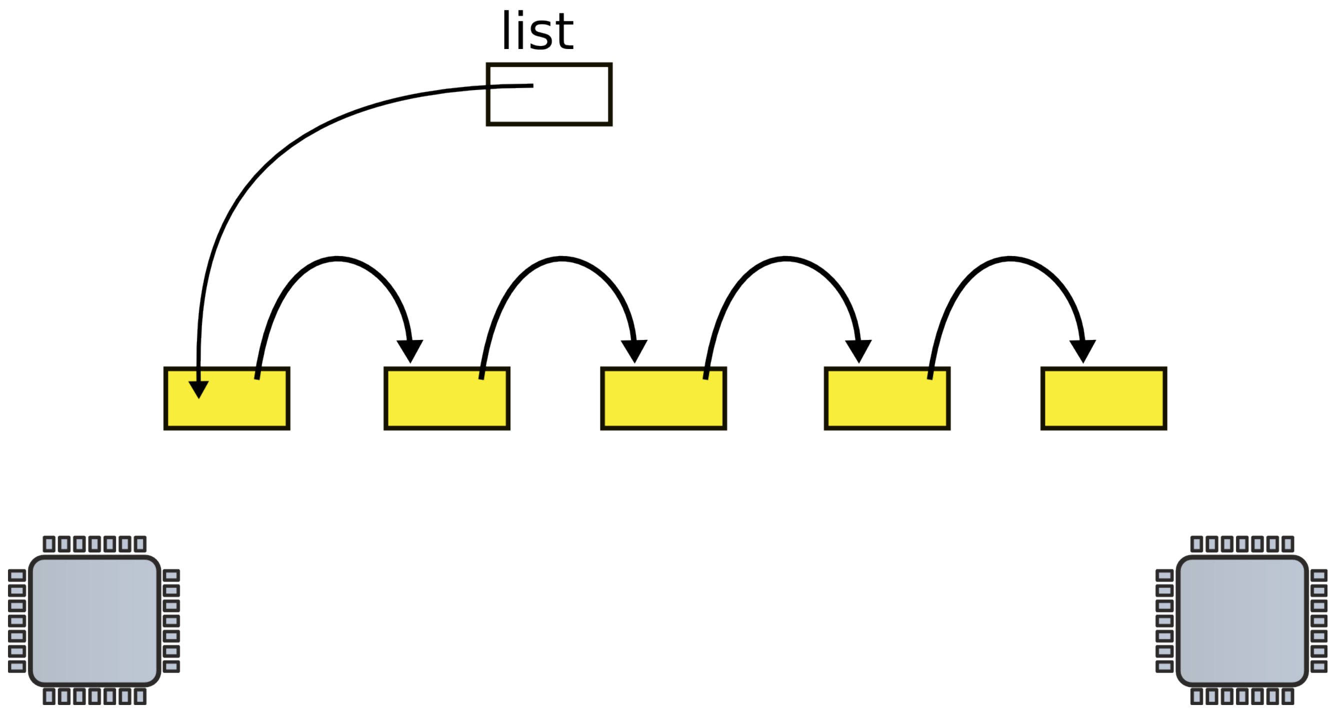
```
struct list {
  int data;
  struct list *next;
};
...
struct list *list = 0;
...
insert(int data)
{
   struct list *l;

   l = malloc(sizeof *l);
   l->data = data;
   l->next = list;
   list = l;
}
```

- Insertion
  - Allocate new list element
  - Save data into that element
  - Insert into the list



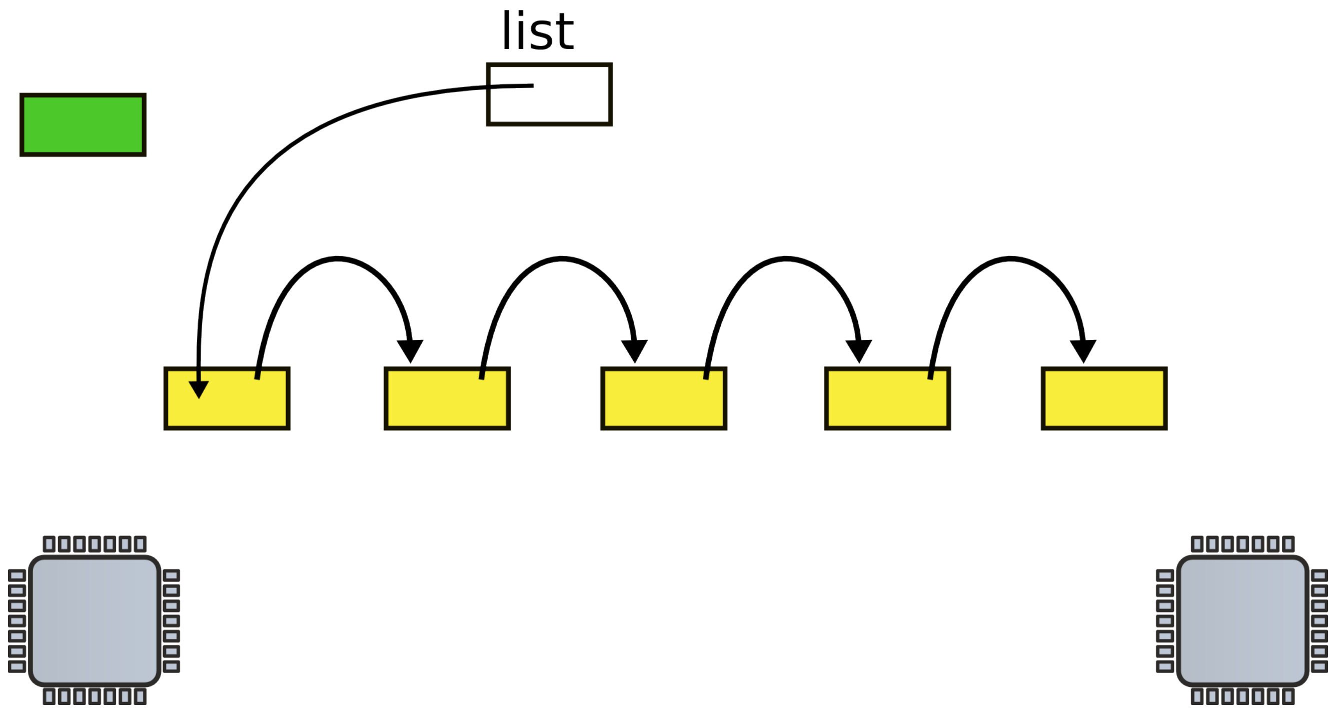
Now what happens when two CPUs access the same list

Request queue (e.g. pending disk requests)

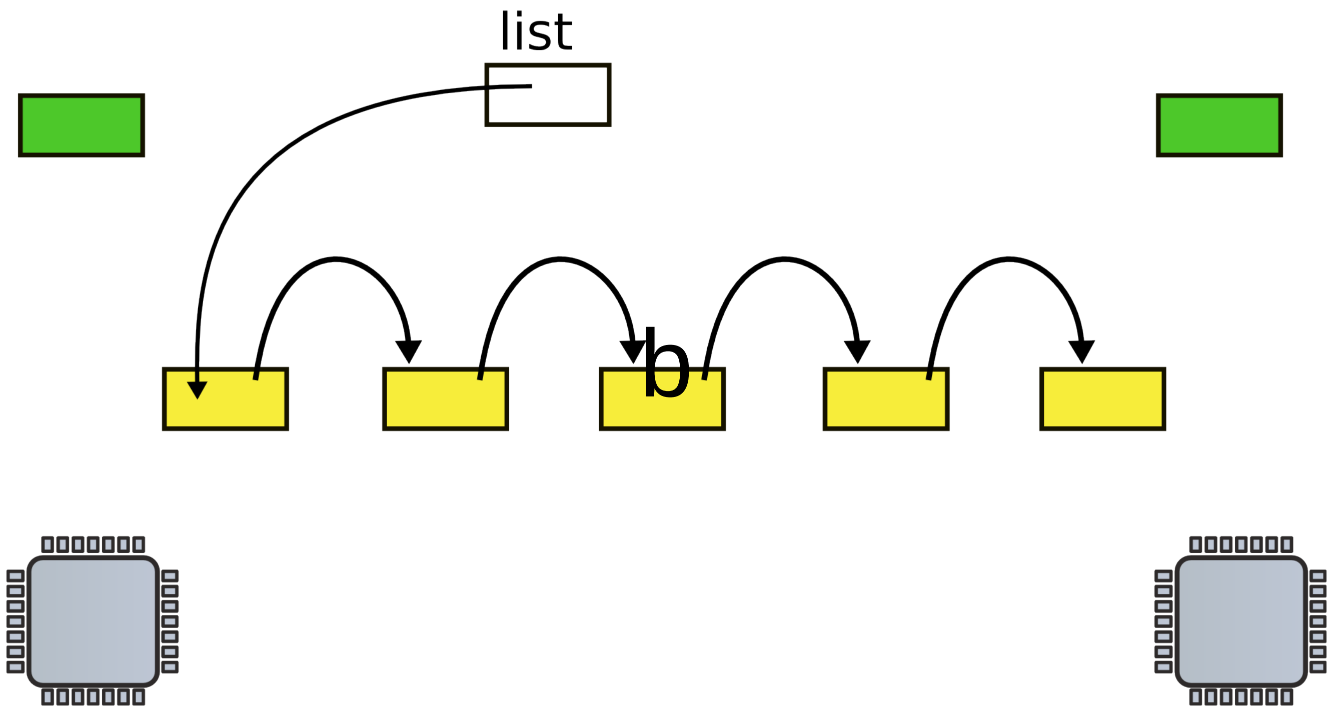


- Linked list, list is pointer to the first element

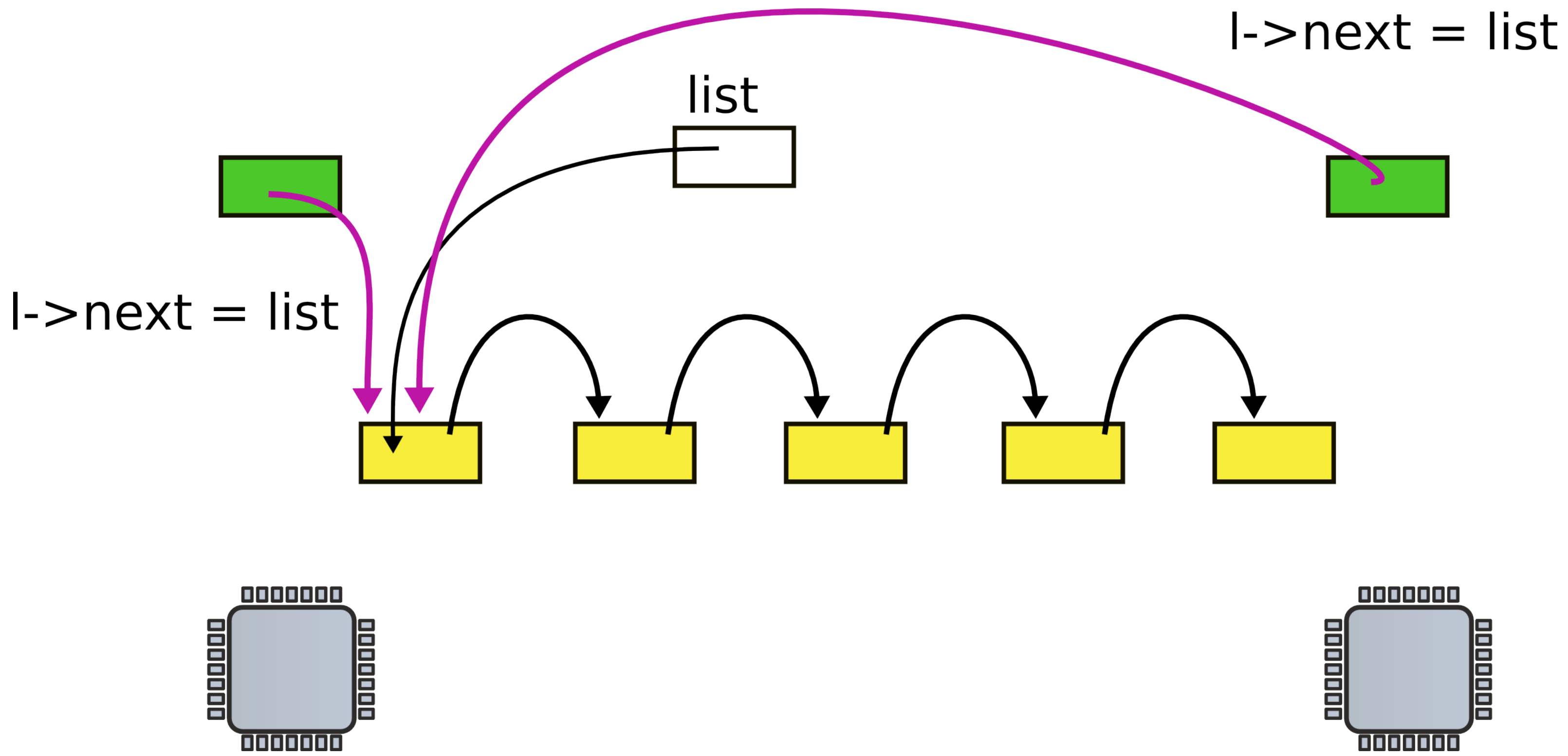
CPU1 allocates new request



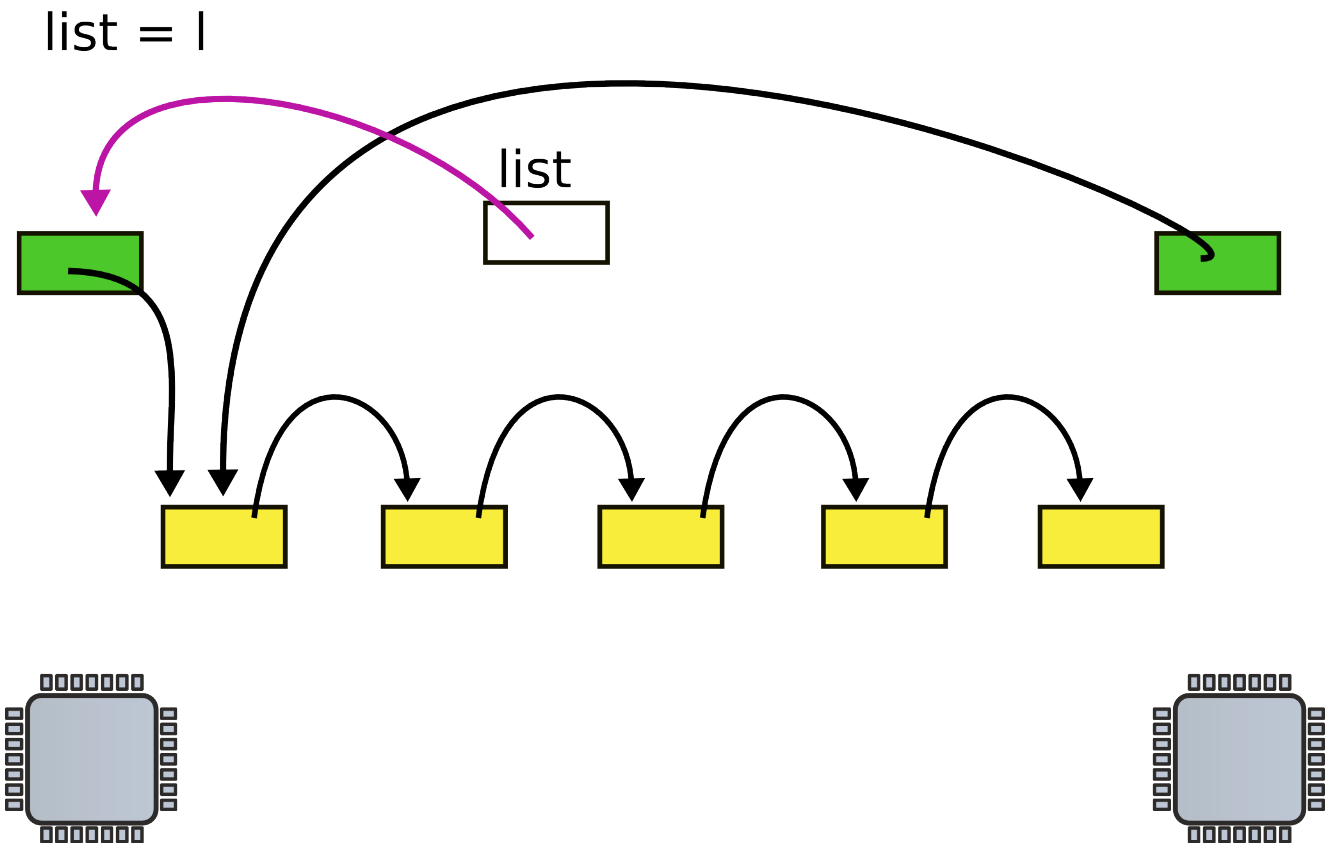
CPU2 allocates new request



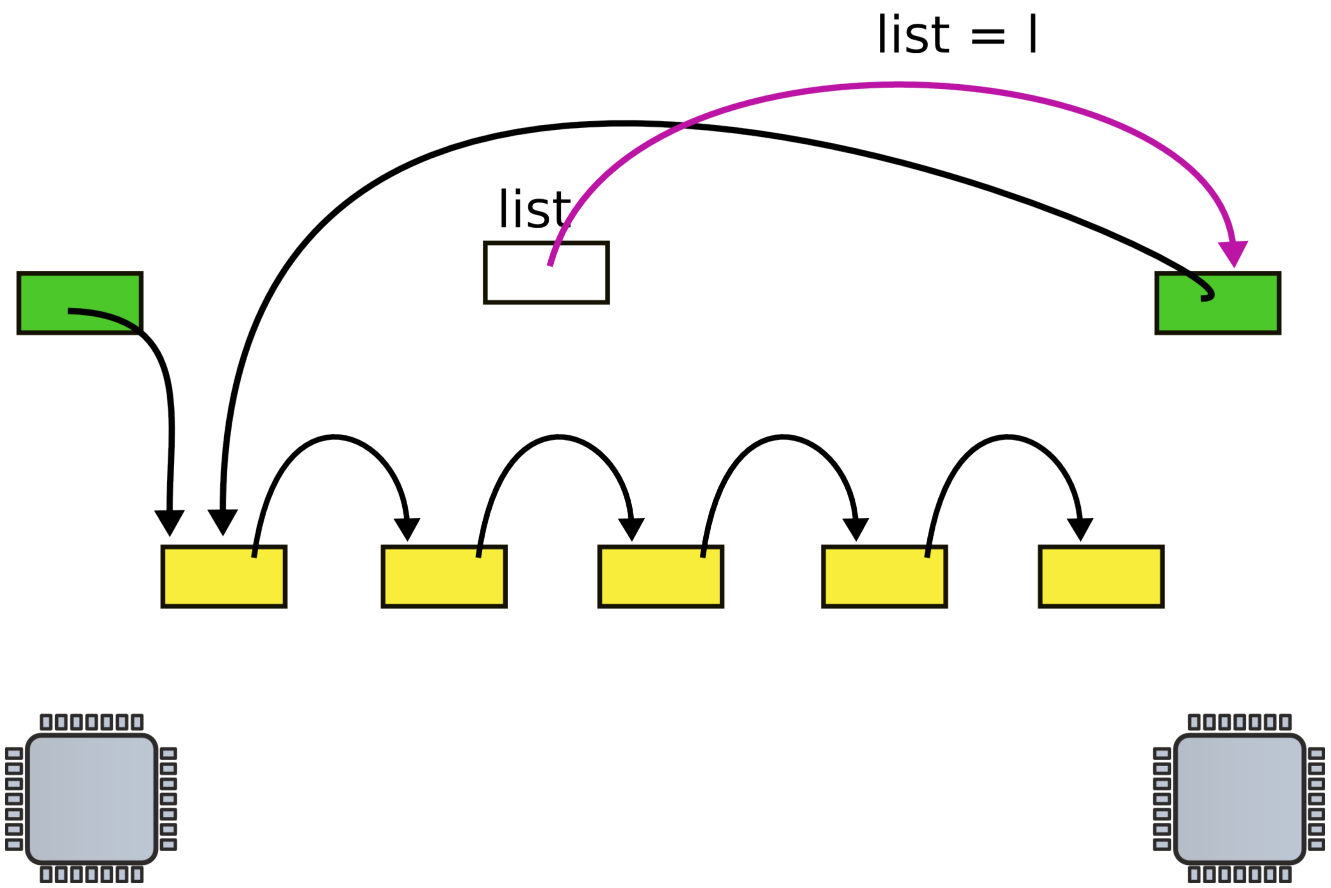
CPUs 1 and 2 update next pointer



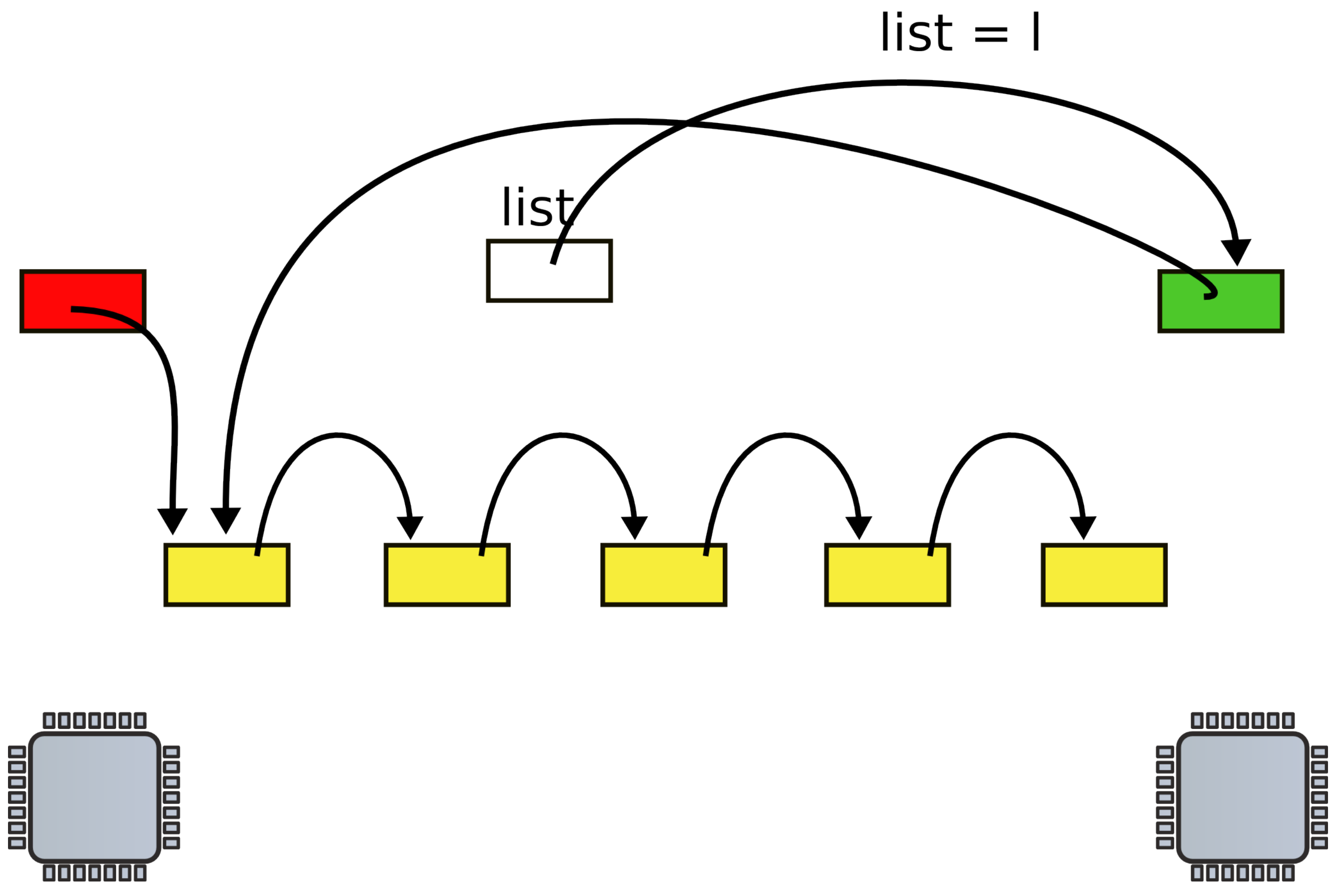
CPU1 updates head pointer



CPU2 updates head pointer



State after the race (red element is lost)



Mutual exclusion

- Only one CPU can update list at a time

List implementation with locks

```
struct list {
  int data;
  struct list *next;
};
struct list *list = 0;
struct lock listlock;
insert(int data)
{
  struct list *l;
  l = malloc(sizeof *l);
  acquire(&listlock);
  l->data = data;
  l->next = list;
  list = l;
  release(&listlock);
}
```

- Critical section

- How can we implement acquire()?

Spinlock

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

- Spin until lock is 0
- Set it to 1

Still incorrect

```
21 void
22 acquire(struct spinlock *lk)
23 {
24   for(;;) {
25     if(!lk->locked) {
26       lk->locked = 1;
27       break;
28     }
29   }
30 }
```

- Two CPUs can reach line #25 at the same time
  - See not locked, and
  - Acquire the lock
- Lines #25 and #26 need to be atomic
  - I.e. indivisible

Synchronization

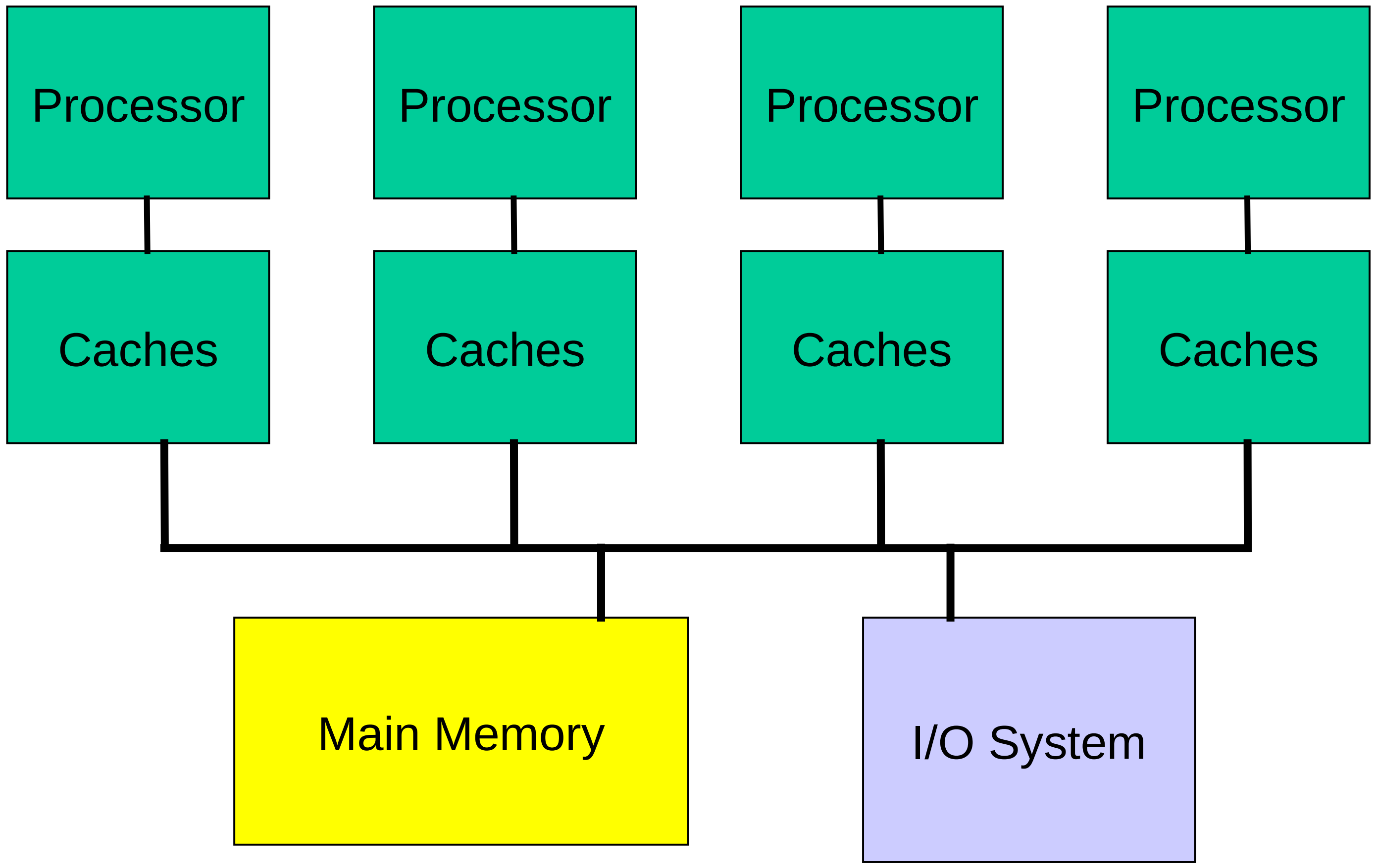
- The simplest hardware primitive that greatly facilitates synchronization implementations (locks, barriers, etc.) is an atomic read-modify-write

- 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

-
  ```
  acquire:   t&s    register, location
             bnz    register, acquire
             CS
  release:   st     location, #0
  ```

SMP/UMA/Centralized Memory Multiprocessor



Thank you!