Lecture 20: Shared-Memory Multiprocessing

1 Shared-Memory Architectures

There are primarily two ways to share memory in a multiprocessing environment: through centralized shared memory or through distributed shared memory. In the centralized approach, which is used in symmetric multiprocessor (SMP) systems and is usually hardware-based, each processor is connected to a shared bus. This shared bus handles all accesses to main memory and I/O. In the distributed approach, each processor has its own local memory and I/O system, and the processors communicate by messages passed through a switched interconnection network. Because of this architecture, distributed shared memory systems are usually software-based.

The centralized approach is often simply called *shared memory*, while the distributed approach is often called *message passing*. In both approaches (as in any modern architecture), each processor has a local cache, which as we will see, introduces interesting problems with coherence in a shared memory environment. See Figure **??** for a schematic of the two approaches. In the future, it is likely that multiprocessing architectures will consist of clusters of SMPs, that is, a message passing system in which each "processor" is actually an SMP.

There are some key tradeoffs between shared memory and message passing systems:

- **Interface to communication.** With shared memory, there is a virtual interface to the communication and processors simply access memory through the shared bus, so communication between processors is implicit and transparent. This offers a high level of abstraction to the programmer. With message passing, processors must explicitly communicate with each other through messages.

- **Architecture complexity.** Shared memory leverages conventional architecture better since existing processors can be added to the shared bus system easily. Message passing, on the other hand, leads to a simpler multiprocessing architecture overall, since there are fewer model assumptions, but also requires code to be rewritten for new platforms due to the explicit interface to communication.

- **Convenience.** Shared memory is "friendly to dusty decks," that is, serial code runs without modification. Message passing libraries, on the other hand, are available for a wide variety of platforms.

- **Protocols.** Since processors do not explicitly communicate with each other in shared memory, the communication protocols are hidden within the system. This means that communication can be close to the hardware (possibly even built into the processor), with the shared bus system deciding how to most efficiently manage communication.

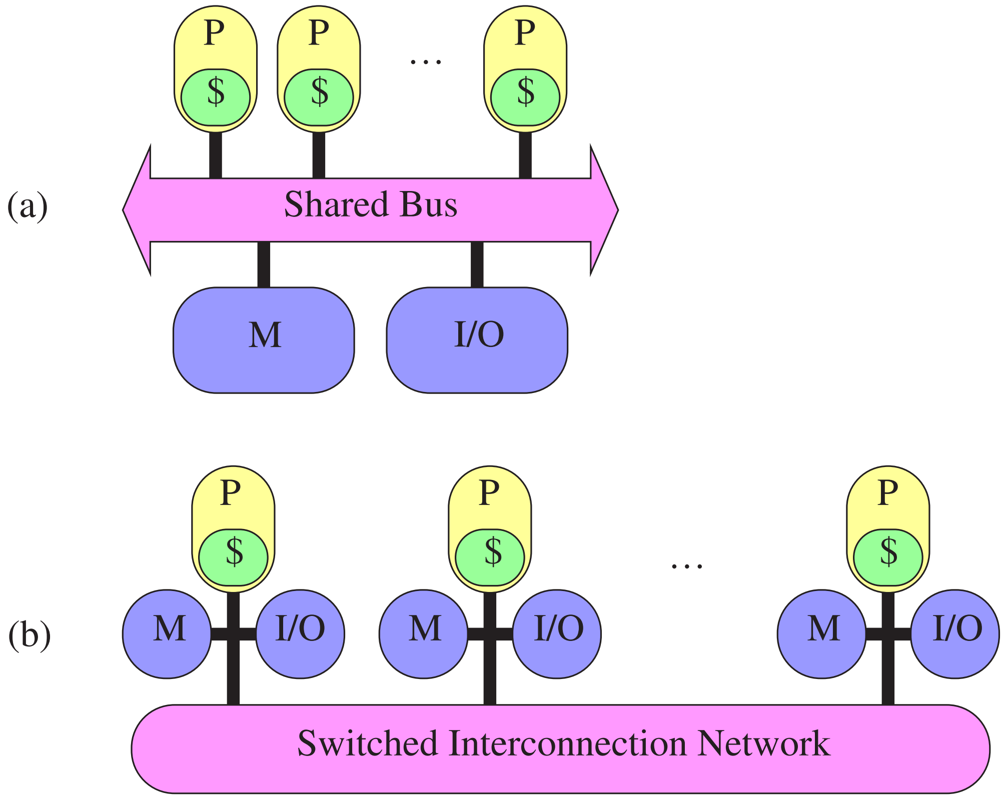


Figure 1: Shared-memory architecures. Processors are denoted by P, caches by $, and memory by M. **(a)** Centralized shared memory in a symmetric multiprocessor system (SMP), where each processor accesses memory through a shared bus. **(b)** Distributed shared memory, where each processor has local memory and processors communicate through an interconnection network.

> In particular, since communication occurs as part of the memory system, a smart shared memory architecure can make communication faster by taking advantage of the memory hierarchy. In a message passing architecture, the communication protocols are fully under user control. The complexities of these protocols are not only a headache for the programmer, but also usually leads to communication being treated as an I/O call for portability reasons. This causes communication to be expensive and slow.

In this lecture we examine three key issues of multiprocessing–cache coherence, memory consistency, and locking—in the context of shared memory (i.e. centralized SMP) architectures.

2 Cache coherence

Recall that the purpose of a cache is to keep the freshest data nearby in fast memory in order to exploit temporal locality in memory accesses. Thus the overall goal of caching is to reduce communication between a processor and the main memory, since this communication is typically slow compared to memory accesses to a cache. When a shared memory architecture is introduced, however, new issues relating to cache coherence emerge.

Suppose we have two processors, A and B, and a variable x stored in shared memory which initially has the value 17 and is present in neither processor's cache. Consider the following sequence of events:

1. Processor A reads x, causing a cache fault, so $x = 17$ is loaded into its cache.
2. Processor B reads x, causing a cache fault, so $x = 17$ is loaded into its cache.
3. Processor A writes a value of 42 to its cache copy of x, without updating main memory.
4. Processor B reads x, causing a cache hit, so it still sees a value of 17.
5. Processor A flushes x from its cache, so it writes back the value of 42 to main memory.
6. Processor B reads x, causing a cache hit, so it still sees a value of 17.

The previous example elucidates a couple of cache coherence issues. At step 4, processor B does not see the new value for x written at step 3 by processor A, because processor A only updates its own cache copy. But even after processor A writes back the new value to main memory in step 5, processor B still sees the old value in step 6 because it reads from its own cache. (This sequence of events is represented in Figure **??**(a).)

The problem highlighted above is one of cache coherence. Even after a processor writes back an updates memory location to main memory, another processor may not see the new value. One simple solution to this problem is to discard caches altogether, but this leads to an unacceptable loss in performance. A more reasonable solution is to use a *coherence protocol* in which each cache line is marked with a state and these states are used to ensure coherence.

For example, a simple protocol might include three cache states:

- INVALID. There is no usable data in the cache line.
- SHARED. The processor only has permission to read the data in the cache line.
- EXCLUSIVE. The processor has read and write permission for the data in the cache line.

The protocol then requires that before a procesor writes a cache line, it must first obtain an exclusive copy. This is accomplished by invalidating all other copies that may exist in the caches of other processors.

There are other protocol requirements, but without specifying these or explaining yet how the protocol is implemented, let us reconsider the previous example with processors A and B sharing a variable x. This time, however, we will apply this simple coherence protocol. Recall that x is initially 17 and is not present in either processor's cache initially.

(a)

Time	A	B	A$	B$	*x*
0			–	–	17
1	`read`		17	–	17
2		`read`	17	17	17
3	`write`		42	17	17
4		`read`	42	17	17
5	`write-back`		–	17	42
6		`read`	–	17	42

(b)

Time	A	B	A$	B$	*x*
0			–	–	17
1	`read`		E 17	–	17
2		`read`	S 17	S 17	17
3	`write`		E 42	–	17
4		`read`	S 42	S 42	17
5	`write-back`		–	E 42	42
6		`read`	–	E 42	42

Figure 2: Cache coherence issues. **(a)** The problem is that processor B still see the old value of x in step 6 even though processor A has written back the new value to main memory in step 5. **(b)** The solution is to use a coherence protocol, which marks each cache line with a state (E for EXCLUSIVE, S for SHARED, and a dash for INVALID) and invalidates processor B's copy of x when processor A writes to it in step 3.

1. Processor A reads x, causing a cache fault, so 17 is loaded into its cache and the line is marked EXCLUSIVE since it is the only processor that has a cache copy of x at this point.

2. Processor B reads x, causing a cache fault, so 17 is loaded into its cache. Since processor A already has a copy of x, both processors mustmark their copies of x as SHARED.

3. Processor A writes a value of 42 to its cache copy of x without updating main memory. Processor B's cache copy of x is invalidated and processor A's copy is marked as EXCLUSIVE.

4. Processor B reads x, causing a cache fault (since its former copy of x has been marked INVALID). It retrieves the new value of 42 from processor A and both copies are marked as SHARED.

5. Processor A flushes x from its cache, so it writes back the value of 42 to main memory. Since processor B is now the only processor with a cache copy of x, it marks its copy as EXCLUSIVE.

6. Processor B reads x, causing a cache hit.

Due to the invalidation of outdated cache copies in step 3 (and the other mechanism of the protocol), processor B never reads an outdated copy of x. (This sequence of events, with the coherence protocol in use, is represented in Figure **??**(b).)

Note that in general, cache coherence protocols are very complex. In real implementations, a cache line can be in one of 1000's of different states. Thus it is no surprise that algorithm verification programs are routinely used to evaluate their correctness.

2.1 Snooping

How would the above protocol be implemented in an SMP? In particular, when processor B reads x in step 4, how does it know to get the new value from processor A's cache? After all, in a shared memory architecture, processors access memory through a shared bus, not by message passing.

One general technique, called *snooping*, takes advantage of the fact that all memory accesses are serialized by the shared bus. With snooping, all the caches of each processor listen to the bus and handle memory requests if appropriate. In step 4 above, processor A's cache notices that processor B is reading x. Since it has an exclusive copy of x and therefore knows that it has the latest value, processor A supplies the value of x to the bus, preempting the main memory (which contains stale data). Furthermore, both processor A and B know to mark their copies of x as SHARED. See Figure **??** for an illustration of this example.

With snooping, the shared bus is treated as a broadcast communication channel. Of course, this communication is handled at the architecture level, so the user need not be aware of it besides relying on the fact that there is in fact cache coherence.

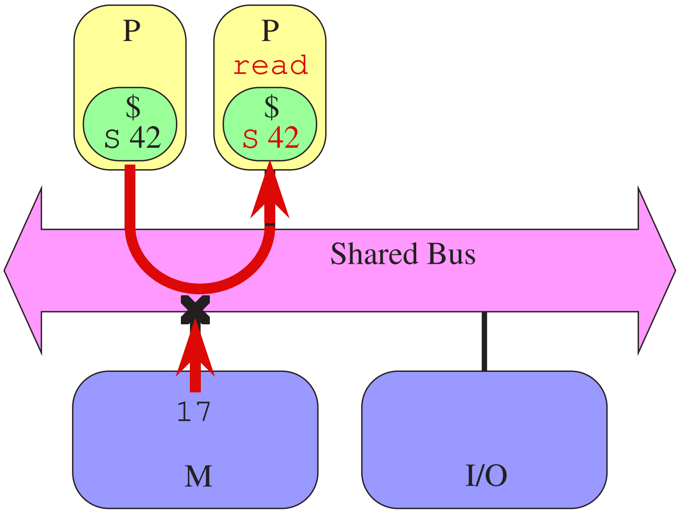


Figure 3: Example of snooping. Processor B issues a read to a location owned by processor A. Processor A "snoops" the read request from the shared bus and supplies the correct value, preempting the outdated value from main memory. Both cache copies are marked as SHARED.

2.2 Directory-based coherence

Another way to implementing a coherence protocol is to use a directory, where a central *home* unit keeps track of which memory locations are being shared in multiple caches and which are held exclusively in a single processor's cache. All memory accesses are handled by the home, which knows based on its directory which caches need to be updated or invalidated on any given memory access.

For instance, suppose a variable is being shared by multiple processors, and one of the processors wants to write a new value to the variable. In a directory-based coherence scheme, the writing processor would issue a write request to home. Then the home, knowing which other processors shared the variable, would send invalidate messages to these processors in order to invalidate their copies of the variable, and grant the exclusive copy to the requesting processor. See Figure **??** for an illustration of this example.

Directory-based coherence is used by all software-based implementations of shared memory. The Alewife machine, for example, uses this cache coherence protocol. One problem associated with directory-based coherence, however, is that this approach is particularly sensitive to the interconnection network speed. Furthermore, because all memory operations are handled centrally, latency is increased for these operations.

2.3 False sharing

Suppose we have two processors, A and B, and that processor A is accessing variable x and processor B is accessing variable y. Since the two processors are accessing different

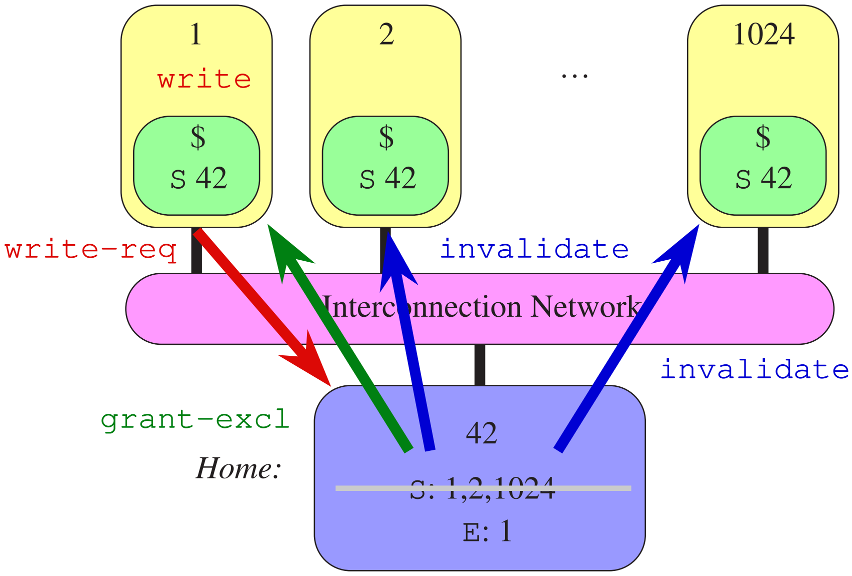


Figure 4: Directory-based coherence. In this example processor 1 issues a write request, causing home to invalidate the processors which have a cache copy of the memory location. The home unit updates its directory, noting the fact the now processor 1 has an exclusive copy, and that the other processors no longer have shared copies.

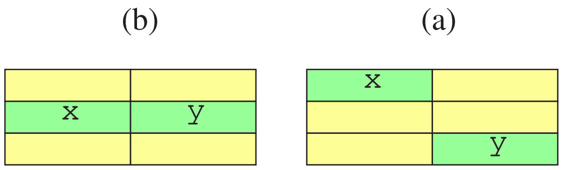
variables, they should each obtain and maintain exclusive copies of these variables, right? Not necessarily. Typical caches contain more than a single word per cache line. So if x and y reside on the same cache line, then the two processors will be sharing a single cache line though they are not actually sharing the same variable.

Since the coherence protocols are based on cache lines, a write by processor A to x will cause processor B's copy of the shared cache line to be invalidated. And likewise a write by processor B to y will cause processor A's copy of the shared cache line to be invalidated. This unfortunate side effect of using coherence protocols with mutiple-word cache lines, which leads to unneeded invalidations, is called *false sharing*. See Figure **??** for an illustration of this phenomenon.

Depending on how a program accesses memory with respect to cache lines and time, false sharing can depreciate the performance of a parallel program significantly. Currently, it is up to the programmer to detect false sharing and to take steps to alleviate the problem. The simple solution to false sharing is to split the data across cache lines, which can be done manually by a knowledgeable programmer.

3 Memory consistency

The *memory consistency model* of a shared memory system specifies how memory will behave in the context of multiple processors reading and writing shared locations. The most basic consistency model, based on the intuition of a single-processor system, is *sequential*



Time	A	B	A\$x	B\$y	A\$x	B\$y
1	read (x)		E	I	E	I
2		read (y)	S	S	E	E
3	write (x)		E	I	E	E
4		write (y)	I	E	E	E
5	write (x)		E	I	E	E
6		write (y)	I	E	E	E

Figure 5: False sharing. **(a)** When x and y do not reside on the same cache line, the two processors acquire and maintain exclusive copies. **(b)** When x and y share a cache line, however, false sharing can occur since writes to one variable cause the other variable to be invalidated since they lie on the same cache line.

consistency [?]. A system is sequentially consistent if the following two conditions are met:

1. The result of any execution is the same as if the operations of all the processors were executed in *some* sequential order.

   One can think of sequential order as a master controller picking one of the processors to run next while halting all of the others. This way, threads can be interleaved, yet only one processor is active at a time.

2. The operations of each individual processor appear in this sequence in the order specified by the program.

Sequential consistency gives the programmer a straightforward, intuitive model of how multiple processors interact regarding shared memory. Unfortunately, it is difficult to implement sequential consistency cheaply.

There are four main types of consistency models:

1. R $\rightarrow$ R

2. W $\rightarrow$ R

3. R $\rightarrow$ W

4. W $\rightarrow$ W

The operations in the first, a read followed by a read, can be performed in any order. For the last three combinations, however, order must always be preserved if the operations are to the same location. As we shall soon see, there are various synchronization methods used to ensure the proper ordering of these operations.

3.1 Relaxed consistency

Programs can often be made to run faster by allowing memory operations to complete in a different order from that specified by the program. For example, *total store order (TSO) consistency* allows a read to bypass an already issued write if the two operations access different locations: given that x and y are distinct memory locations, a write to x can execute after a read to y even if the read is prior to the write in the program order, since there is no semantic difference between the two orderings when run on a single processor. This is an example of a *relaxed consistency model*, in which the requirements of sequential consistency are relaxed in order to realize higher performance.

3.2 Relaxed consistency anomalies

Unfortunately, relaxed consistency can lead to unexpected behavior with multiple processors. For example, consider the following situation:

- Two variables in shared memory, x and y, are both initially 0.
- Processor A runs the following code:

```
x = 1;
if (y == 0)
  foo();
```

- Processor B runs the following code:

```
y = 1;
if (x == 0)
  foo();
```

How many times can the procedure `foo()` be run? It depends on the consistency model. Under sequential consistency, `foo()` can be run either zero or one times. It will run zero times if both x and y are set to 1 before either processor executes the `if` statement. To see that `foo()` will be run at most once, suppose processor A calls `foo()`. Then y must still be 0 and so processor B has not yet assigned 1 to y, and thus, due to the execution order requirement of sequential consistency, processor B has not yet tested the value of x. When processor B does read x, it will see a value of 1, since processor A must have set x to 1 before calling `foo`, again due to the ordering requirement of sequential consistency.

With total store ordering, however, `foo()` could be executed *twice* - once by each processor! This can happen because TSO will allow the reads (which test whether x or y equal 0) to be executed before the writes (which set x or y to 1), despite the specified program order. After all, x and y are different memory locations, so why shouldn't these operations be executed out of order if it leads to higher perfomance? As we see, the reason is that a reordering that is semantically identical in a single processor situation is not necessarily so with multiple processors. (Note that instruction reordering may be performed by a compiler or even by the hardware itself.)

A solution to the above anomaly, where `foo()` can be executed twice despite the intention of the code is to use a *memory fence*. A memory fence is an instruction around which memory accesses are not allowed to be reordered. Any the memory operation before a fence must be executed before any operation after the fence. Operations between fences can still be reodered according to TSO (or whatever other relaxed consistency model in use).

One such memory fence is `membar()`, which causes the write buffer to be emptied before the next read. In the above example, inserting `membar()`'s in the two code fragments between the assignment statement (a write) and the conditional statement (a read), would prevent the reordering that could lead to `foo()` being called twice.

4 Locking

A *lock* is a synchronization variable that can be acquired by only one processor at a time. If a processor is unable to obtain a lock, it *blocks*, waiting for the lock to be freed. Locks are used to guarantee *mutual exclusion*, the condition that a fragment of code will not be executed simultaneously by more than one processor. For example, if two processors A and B both run the following code:

```
lock(S)
foo();
unlock(S);
```

where `S` is a shared lock, then `foo()` (the *critical section*) will be run by at most one of the two processors at any given time.

5 Atomic exchange

A very common way to implement locks is with an *atomic exchange* instruction, which is provided by many machines. The semantics of atomic exchange are simple:

- The values of two memory locations are exchanged.
- The exchange occurs atomically, that is, other processors see either the state of the variables before or after the exchange, never an intermediate state where the exchange is only partially completed.

Given this instruction, locking can be implemented easily:

- The shared lock variable S is initially 1, indicating that the lock has not been acquired by any processor.
- Each processor contending for the lock has a private variable s, initially set to 0, which indicates whether that processor owns the lock.
- When locking, a processor does an atomic exchange between the shared lock variable S and its private indicator s. If S equals 1 and therefore the lock is free to be acquired, the instruction will set $s = 1$ and $S = 0$, giving the locking processor the lock and indicating to other processor that the lock is no longer free. If S equals 0, the processor blocks by looping until S becomes 1 and gets exchanged into s. The code for `lock(S)` is

  ```
  while (s != 1)
    AtomicExchange(s, S);
  ```

- When unlocking, a processor simply does a single atomic exchange between S and s, which appropriately sets the shared lock variable back to 1 and the private indicator back to 0. The code for `unlock(S)` is the one instruction:

  ```
  AtomicExchange(s, S);
  ```

5.1 Load-linked (or load-locked) / store conditional

A more modern approach to implementing locks is based on two new instructions: *load linked* (LL) and *store conditional* (SC). A store conditional is a store from x to y which completes the store only if no change has occured to y since the last time y was loaded with the load linked instruction, with x being set to 0 or 1 to indicate whether the store occured or not.

For example, consider the following code which implements an atomic exchange between the location pointed to by R1 (written as `*R1` and the value of R4:

```
try: MOV   R3, R4
     LL    R2, *R1
     SC    R3, *R1
     BEQZ  R3, try
     MOV   R4, R2
```

The code works as follows:

1. The value of R4 is assigned to a temporary register R3.
2. The load linked instruction is used to load *R1 into R2.
3. The store conditional instruction is used to store R3 (which equals R4) into *R1, with two possible results: *R1 has not changed since the LL instruction, so *R1 gets R3 and R3 gets 1, indicating a successful store; or *R1 has changed since the LL instruction (due to another processor), so *R1 is unchanged and R3 gets set to 0, indicating the store failed.
4. If the store failed (i.e. RC == 0), then the program tries again by branching to the beginning.
5. Otherwise, the exchange is completed by storing the value of R2 (which equals *R1) into R4.

Notice that this code depends on one of the exchanged variables (R4) being a private local register, which cannot be accessed by other processors.

The main advantage to using the load-linked / store conditional scheme for implementing locks is that it exploits *speculative computing*. Speculative computing is when a processor proceeds to execute subsequent instructions even when the work may need to be aborted depending on later results. Improved performance is realized when the processor does not need to abort. If it must abort, then nothing has been lost since the processor does not have any other useful work to do.

In the case of pipelining, for example, when a branch instruction is reached, a processor will choose one of the two branches to follow. If the wrong one is chosen, the instructions after the branch are flushed from the pipeline and the processor backtracks to process the correct branch. The only other alternative is to execute NOP's until the branch instruction is evaluated, which *never* reduces processing time.

Similarly, in the case of load-linked / store conditional, the procedure proceeds assuming that the store will be successful. If it fails, then the code simply aborts to the original state before the exchange attempt. If it succeeds, then it can complete the locking process quickly and proceed with subsequent operations.

5.2 Deadlock and the dining philosophers problem

Once upon a time, before the guilded age of computer science and keen insights into locking, there were five Chinese philosophers sitting at a round table. In front of each philosopher was a bowl of rice, and due to a bamboo shortage due to the emperor's unending war campaigns, the philosophers had only one chopstick each. Deciding that the most enlighted solution to this dilemma was to share chopsticks in a systemmatic way, each philosopher placed his chopstick to the left of his bowl, resulting in a single chopstick between every two adjacent bowls of rice. Then they agreed to the following rule: when someone wanted to eat, he would first grab the chopstick to his left, then the one to his right, and finally replace the two chopsticks after satisfying his hunger.

For many moons the philosophers thought their great thoughts, eating whenever they felt hungry. (The ancient traditions do not tell us why the bowls of rice never emptied...) Their thoughts were so great that the village folk never bothered them, nor wondered that they never showed up for the latern festivals. But one day, one of the village boys accidentally hit a shuttlecock into the house of the philosophers, and after timidly knocking at the door but gaining no response, the boy went in to retrieve his toy. Upon seeing the five philosophers sitting motionless at the table, each man clutching a single chopstick in his left hand, the boy screamed out loud. The philosophers had all starved to death.

What happened? Quite literally, deadlock. One day, the five philosophers had each picked up the chopstick to his left at the very same moment, and then, seeing that the chopstick to his right was not there, began waiting for it to be replaced. Unfortunately, no chopstick was ever put back on the table, since each philosopher was waiting for the philosopher on his right, thus no one could ever get two chopsticks to eat his rice.

What could the philosophers have done to prevent such an unfortunate end? They could have first numbered the chopsticks from 1 to 5, then made the rule that when one of them wanted to eat, he would first pick up the chopstick next to him with the lower number, and then pick up the other chopstick. Such an algorithm would have prevented deadlock because in any situation in which the philosophers contend for the same set of chopsticks, the man who manages to pick up the highest numbered chopstick will always be able to proceed with eating, and will thus eventually replace his chopsticks, allowing others who are waiting for his chopsticks to also proceed.

The moral of the dining philosophers story is that to prevent deadlock, locks need to be ordered, thus breaking the symmetry in the lock-acquiring algorithm. This allows at least one processor to acquire all of the required locks in order to continue processing.

Relating the story to processors and shared resources, the problem of deadlock can be prevented by numbering the shared resources (e.g. the chopsticks), and then having each processor (e.g. the philosophers) acquire locks (e.g. pick up a chopstick) according a specific total ordering, with lowest first. Thus, the invariant is that the processor which acquires the highest numbered resource (according to the total order) will always be able to proceed.

6 Conclusion

Shared memory architectures are becoming the standard approach to multiprocessing, with every major manufacturer offering an SMP product. As we have seen in this lecture, such architectures can be tricky for the user, especially regarding memory consistency and issues of locking. A great step forward in this regard would be high-level programming support (a la Cilk) which frees the programmer from having to use user-level protocols to manage these complexities.

References

[1] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer*, C-28(9):690–691, September 1979.