Lecture 11
Monitors and Condition Variables

Monitors

- Semaphores can result in deadlock due to programming errors
  - forgot to add a P() or V(), or misordered them, or duplicated them

- To reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*
  - essentially automates insertion of P() and V() for you

  - As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#

  - underneath, the OS may implement monitors using semaphores and mutex locks

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
     // shared local variables

     function f1(...) {
     ...
     }
     ...
     function fN(...) {
     ...
     }
     init_code(...) {
     ...
     }
  }
```

- A monitor ensures that only 1 process/thread at a time can be active within a monitor
  - simplifies programming, no need to explicitly synchronize
- Implicitly, the monitor defines a mutex lock

  semaphore mutex = 1;

- Implicitly, the monitor also defines mutual exclusion around each function
  - Each function's critical code is effectively:

```
function fj(...) {
  P(mutex)
  // critical code
  V(mutex)
}
```

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
     // shared local variables
     int count;
     data_type data[MAX_COUNT];
     function f1(...) {
     ...
     }
     ...
     function fN(...) {
     ...
     }
     init_code(...) {
     ...
     }
  }
```

- The monitor's local variables can only be accessed by local monitor functions

- Each function in the monitor can only access variables declared locally within the monitor and its parameters

Monitors and Condition Variables

- Previous definition of a monitor achieves
  - mutual exclusion
  - hiding of wait() and signal() from user
  - loses the ability that semaphores had to enforce order
    - wait() and signal() are used to provide mutual exclusion
    - but have lost the unique ability for one process to signal another blocked process using signal()
    - there is no way to have a process sleep waiting on the signal

- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, augment monitors with *condition variables*.

Condition Variables

- Augment the mutual exclusion of a monitor with an ordering mechanism
  - Recall: Semaphore P() and V() provide both mutual exclusion and ordering
  - Monitors alone only provide mutual exclusion
- A condition variable provides ordering through Signaling
  - Used when one task wishes to wait until a condition is true before proceeding
    - Such as a queue being full enough or data being ready
  - A 2nd task will signal the waiting task, thereby waking up the waiting task to proceed

Monitors and Condition Variables

condition y;

A condition variable *y* in a monitor allows three operations

- y.wait()
  - blocks the calling process
  - can have multiple processes suspended on a condition variable, typically released in FIFO order
  - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
- y.signal()
  - resumes exactly 1 suspended process.
  - If no process is waiting, then function has *no effect*.
- y.queue()
  - Returns true if there is at least one process blocked on y

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire(int time) {
              if (busy)
                 x.wait(time);
              busy = TRUE;
  }
  void release() {
              busy = FALSE;
              x.signal();
  }
initialization code() {
   busy = FALSE;
  }
}
```

Condition Variables Example

Block Task 1 until a condition holds true, e.g. queue is empty

```
condition wait_until_empty;
```

Task 1:

```
wait_until_empty.wait();
…// proceed after
              queue empty
```

Task 2:

```
…
// queue is empty so signal
wait_until_empty.signal();
```

Problem: If Task 2 signals before Task 1 waits, then Task 1 waits *forever* because CV *has no state*!

Condition Variables vs Semaphores

- Both have wait() and signal(), but semaphore's signal()/V() preserves states in its integer value

```
Semaphore wait_until_empty=0;
```

Task 1:

```
wait(wait_until_empty);
…// proceed after
           queue empty
```

Task 2:

```
…
// queue is empty so signal
signal(wait_until_empty);
```

If Task 2 signals before Task 1 waits, then Task 1 does not wait forever because semaphore has state and remembers earlier signal()

Complex Conditions

- Suppose you want task T1 to wait until a complex set of conditions set by T2 becomes TRUE
  - Use a condition variable

```
condition = x;
int count = 0;
Float f=0.0;
```

```
Task 1
----
…
while (f!=7.0 && count<=0){
      x.wait();
}
… // proceed
```

```
     Task 2
----
…
f=7.0;
count++;
x.signal();
```

Problem: could be a race condition in testing and setting shared variables

Complex Conditions (2)

```
lock mutex;
Condition x;
int count=0;
Float f=0.0;

T1
----
…
Acquire(mutex);
While(f!=7.0 && count<=0) {
       Release(mutex);
       x.wait();
       Acquire(mutex);
}
Release(mutex);
…// proceed
```

- Surround the test of conditions with a mutex to atomically test the set of conditions

```
T2
----
…
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```

Complex Conditions (3)

- Surround the test of conditions with a mutex to atomically test the set of conditions

```
lock mutex;
Condition x;
int count=0;
Float f=0.0;
```

```
pthread_cond_wait(&cond_var,&mutex)
```

```
T1                                  T2
----                                ----
…                                   …
Acquire(mutex);                     Acquire(mutex);
While(f!=7.0 && count<=0) {         f=7.0;
   pthread_cond_wait(&x,&mutex);    count++;
}                                   Release(mutex);
Release(mutex);                     x.signal();
…// proceed
```

Semaphores

```
typedef struct {
    int value;
    PID *list[ ];
} semaphore;
```

Both wait() and signal() operations are atomic

```
wait(semaphore *s) {
    s→value--;
    if (s→value < 0) {
        add this process to s→list;
        sleep ( );
    }
}
```

```
signal(semaphore *s) {
    s→value++;
    if (s→value <= 0) {
        remove a process P from s→list;
        wakeup (P);
    }
}
```

Monitors and Condition Variables

condition y;

A condition variable *y* in a monitor allows three operations

- y.wait()
  - blocks the calling process
  - can have multiple processes suspended on a condition variable, typically released in FIFO order
  - textbook describes another variation specifying a priority p, i.e. call x.wait(p)
- y.signal()
  - resumes exactly 1 suspended process.
  - If no process is waiting, then function has *no effect*.
- y.queue()
  - Returns true if there is at least one process blocked on y

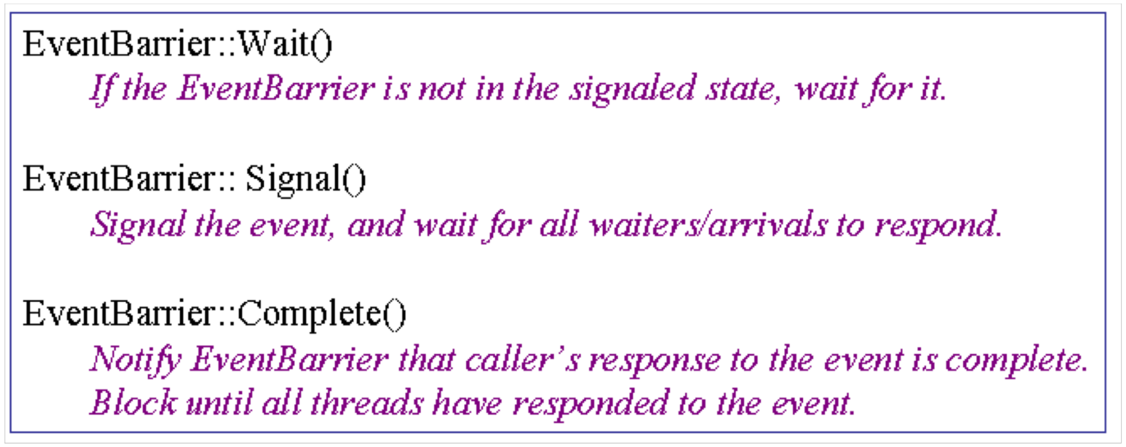
Semaphores vs. Condition Variables

- S.signal vs. C.signal
  - C.signal has no effect if no thread is waiting on the condition
    - Condition Variables are not variables (ha!) They have no value
  - S.signal has the same effect whether or not a thread is waiting
    - Semaphore retains a "memory"

- S.wait vs. C.wait
  - S.signal check the condition and block only if necessary
    - Programmer does not need to check the condition after returning from S.wait
    - Wait condition is defined internally but limited to a counter.
  - C.wait is explicit: It does not check the condition, ever
    - Condition is defined externally and protected by integrated mutex

Other Mechanisms

- EventBarrier
  - Combine semaphores and condition variables
  - Has binary memory and no associated mutex
  - Broadcast to notify all waiting threads
  - Wait until an event is handled



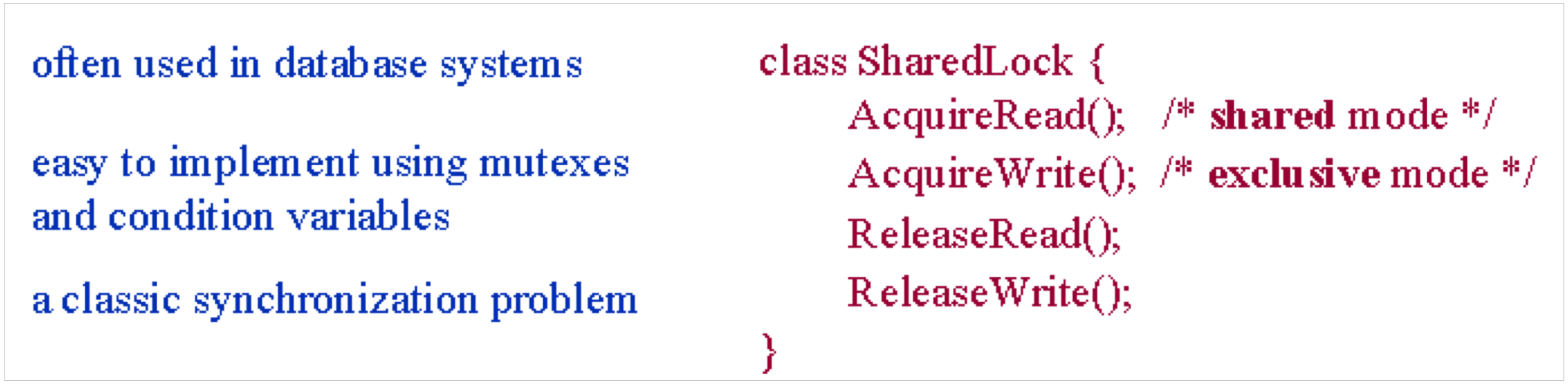
EventBarrier::Wait()
    If the EventBarrier is not in the signaled state, wait for it.

EventBarrier:: Signal()
    Signal the event, and wait for all waiters/arrivals to respond.

EventBarrier::Complete()
    Notify EventBarrier that caller's response to the event is complete.
    Block until all threads have responded to the event.

Other Mechanisms

- SharedLock: Reader/Writer Lock
  - Support Acquire and release primitives
  - Guarantee mutual exclusion when writer is present
  - Provides better concurrency for readers when no writer is present



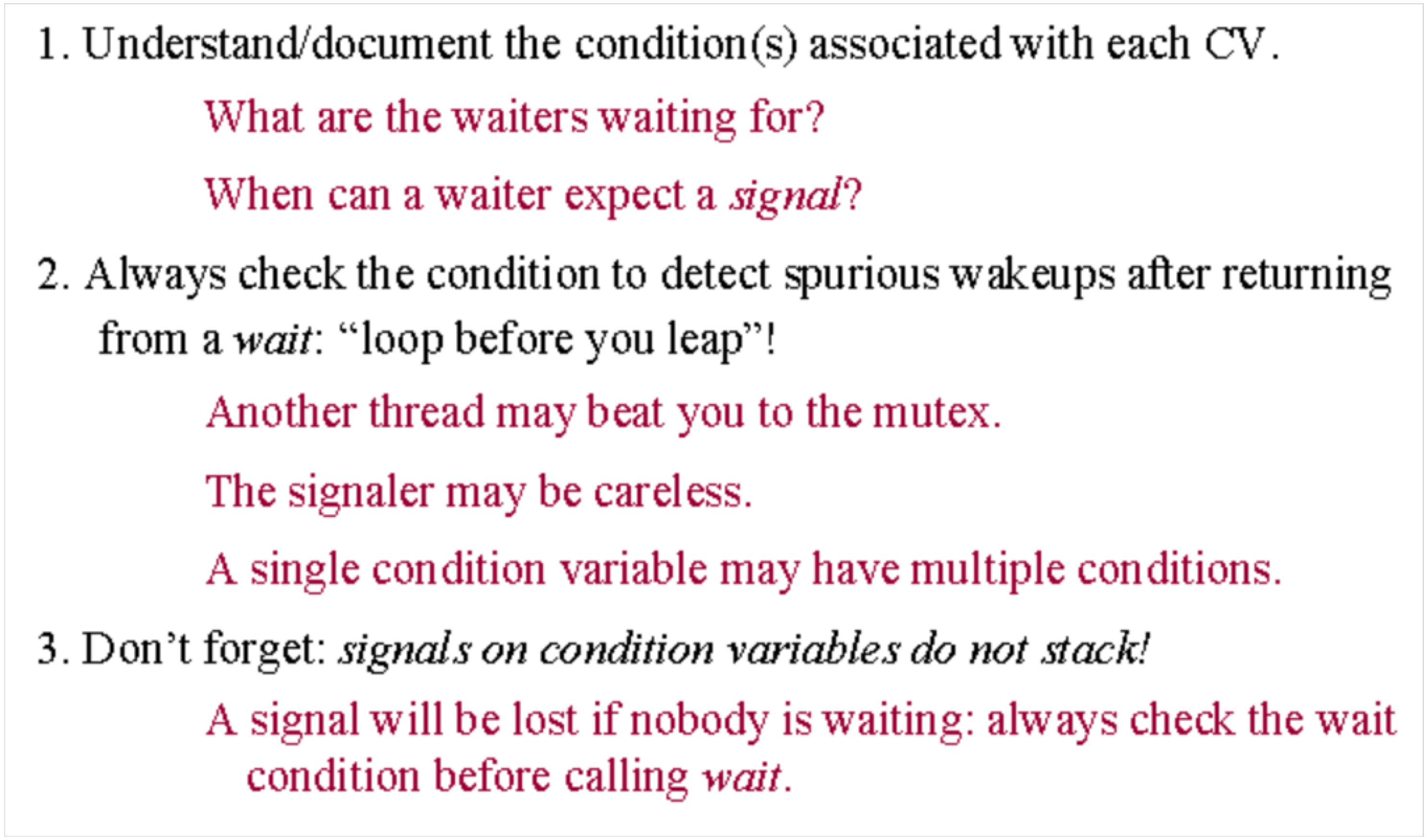
often used in database systems

easy to implement using mutexes and condition variables

a classic synchronization problem

```
class SharedLock {
        AcquireRead();   /* shared mode */
        AcquireWrite();  /* exclusive mode */
        ReleaseRead();
        ReleaseWrite();
}
```

Guidelines for Condition Variables



1. Understand/document the condition(s) associated with each CV.

   What are the waiters waiting for?

   When can a waiter expect a *signal*?

2. Always check the condition to detect spurious wakeups after returning from a *wait*: "loop before you leap"!

   Another thread may beat you to the mutex.

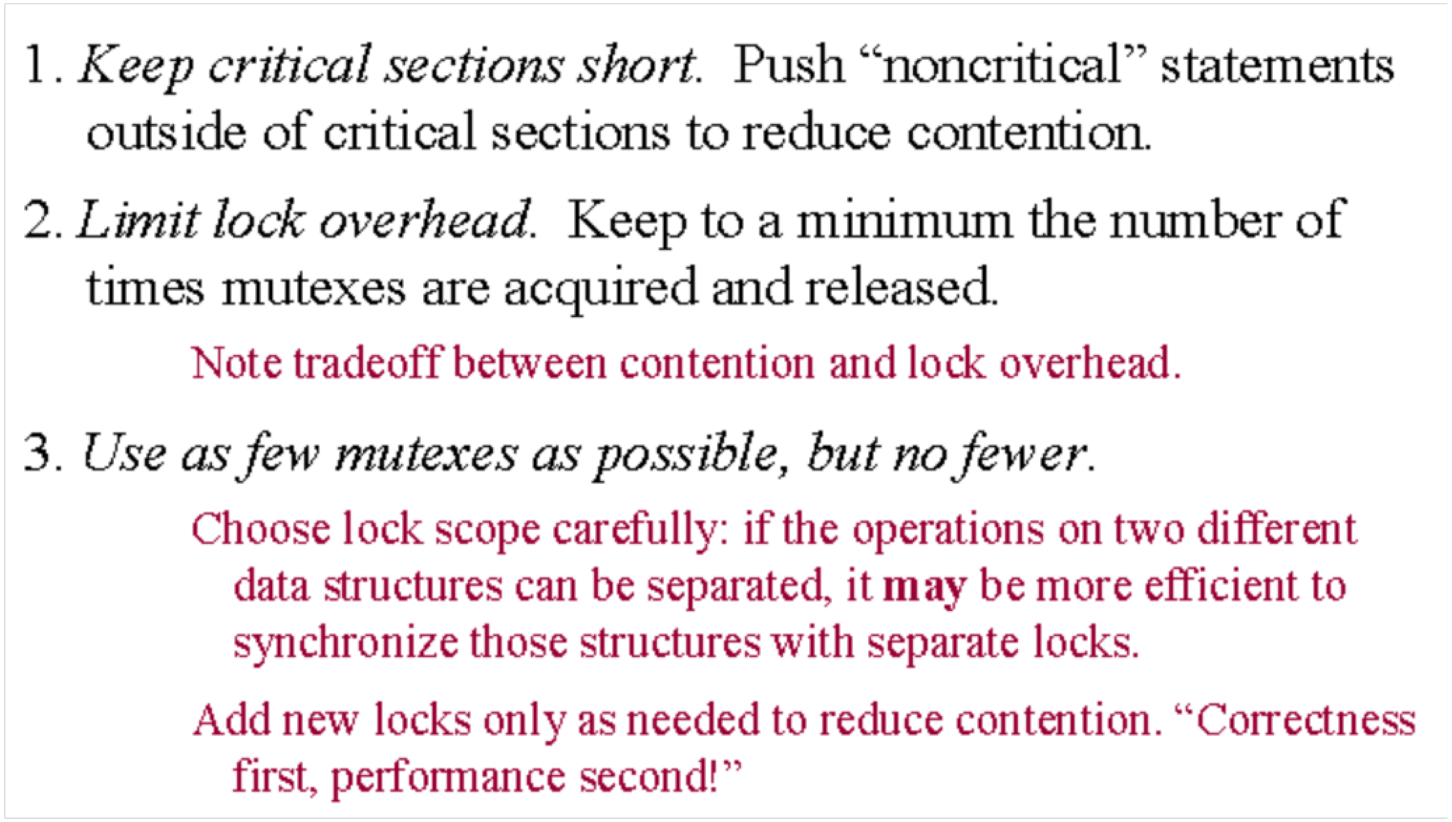
   The signaler may be careless.

   A single condition variable may have multiple conditions.

3. Don't forget: *signals on condition variables do not stack!*

   A signal will be lost if nobody is waiting: always check the wait condition before calling *wait*.

Guidelines for Choosing Lock Granularity



1. *Keep critical sections short.* Push "noncritical" statements outside of critical sections to reduce contention.
2. *Limit lock overhead.* Keep to a minimum the number of times mutexes are acquired and released.

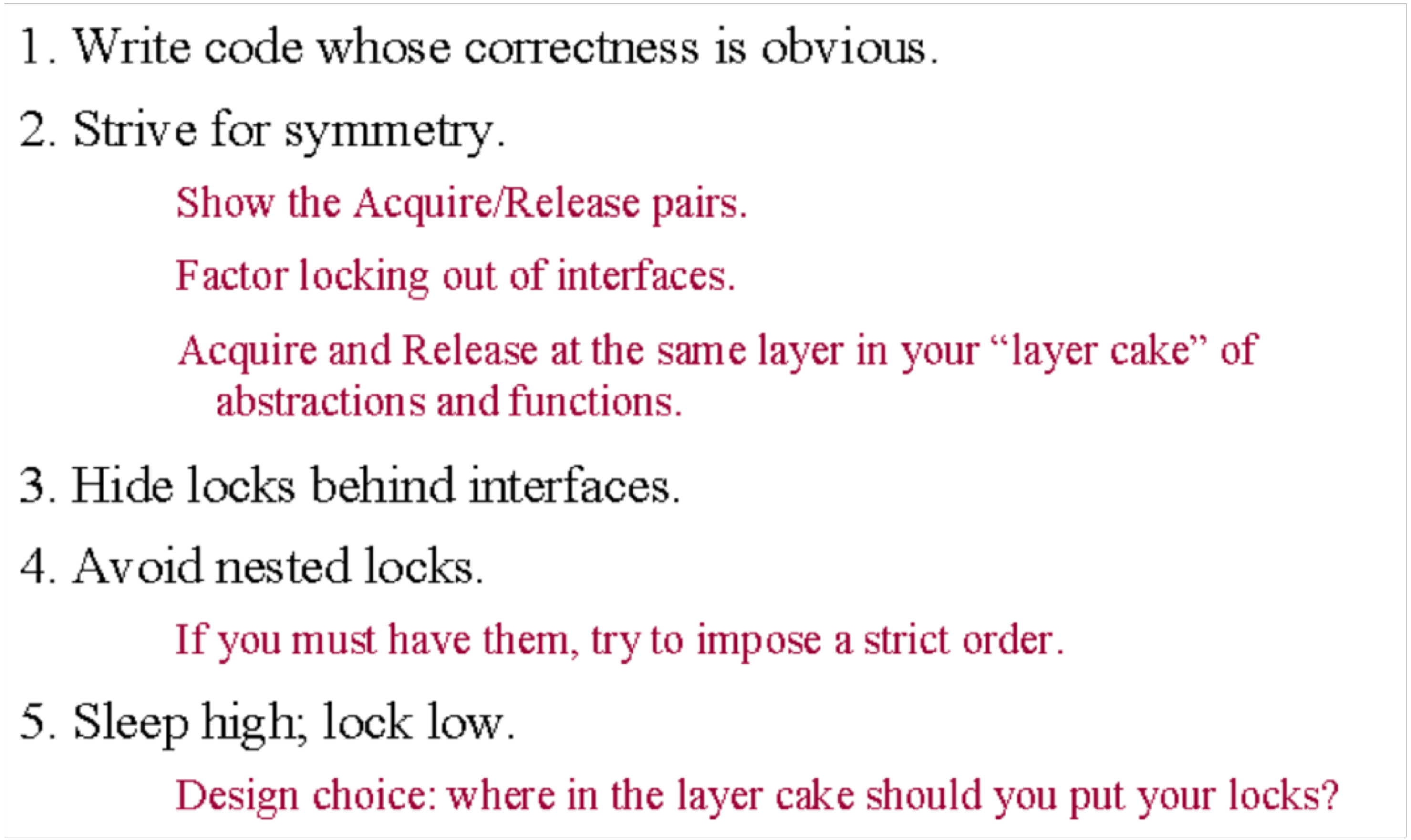
   Note tradeoff between contention and lock overhead.

3. *Use as few mutexes as possible, but no fewer.*

   Choose lock scope carefully: if the operations on two different data structures can be separated, it **may** be more efficient to synchronize those structures with separate locks.

   Add new locks only as needed to reduce contention. "Correctness first, performance second!"

More Guidelines for Locking



1. Write code whose correctness is obvious.
2. Strive for symmetry.

   Show the Acquire/Release pairs.

   Factor locking out of interfaces.

   Acquire and Release at the same layer in your "layer cake" of abstractions and functions.
3. Hide locks behind interfaces.
4. Avoid nested locks.

   If you must have them, try to impose a strict order.
5. Sleep high; lock low.

   Design choice: where in the layer cake should you put your locks?

University of Colorado
Boulder