29

Lock-based Concurrent Data Structures

Before moving beyond locks, we'll first describe how to use locks in some common data structures. Adding locks to a data structure to make it usable by threads makes the structure **thread safe**. Of course, exactly how such locks are added determines both the correctness and performance of the data structure. And thus, our challenge:

> CRUX: HOW TO ADD LOCKS TO DATA STRUCTURES 怎么给不同的 DS 上锁？
>
> When given a particular data structure, how should we add locks to it, in order to make it work correctly? Further, how do we add locks such that the data structure yields high performance, enabling many threads to access the structure at once, i.e., **concurrently**?

Of course, we will be hard pressed to cover all data structures or all methods for adding concurrency, as this is a topic that has been studied for years, with (literally) thousands of research papers published about it. Thus, we hope to provide a sufficient introduction to the type of thinking required, and refer you to some good sources of material for further 调查，查究 inquiry on your own. We found Moir and Shavit's survey to be a great source of information [MS04].

29.1 Concurrent Counters

One of the simplest data structures is a counter. It is a structure that is commonly used and has a simple interface. We define a simple non-concurrent counter in Figure 29.1.

Simple But Not Scalable

As you can see, the non-synchronized counter is a trivial data structure, requiring a tiny amount of code to implement. We now have our next challenge: how can we make this code **thread safe**? Figure 29.2 shows how we do so.

```
typedef struct __counter_t {
    int value;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
}

void increment(counter_t *c) {
    c->value++;
}

void decrement(counter_t *c) {
    c->value--;
}

int get(counter_t *c) {
    return c->value;
}
```

Figure 29.1: **A Counter Without Locks**

基本要求是达到了，

This concurrent counter is simple and works correctly. In fact, it follows a design pattern common to the simplest and most basic concurrent data structures: it simply adds a single lock, which is acquired when calling a routine that manipulates the data structure, and is released when returning from the call. In this manner, it is similar to a data structure built with **monitors** [BH73], where locks are acquired and released automatically as you call and return from object methods.

At this point, you have a working concurrent data structure. The problem you might have is performance. If your data structure is too slow, you'll have to do more than just add a single lock; such optimizations, if needed, are thus the topic of the rest of the chapter. Note that if the data structure is *not* too slow, you are done! No need to do something fancy if something simple will work.

但还想更好。

继续优化！

To understand the performance costs of the simple approach, we run a benchmark in which each thread updates a single shared counter a fixed number of times; we then vary the number of threads. Figure 29.5 shows the total time taken, with one to four threads active; each thread updates the counter one million times. This experiment was run upon an iMac with four Intel 2.7 GHz i5 CPUs; with more CPUs active, we hope to get more total work done per unit time.

From the top line in the figure (labeled 'Precise'), you can see that the performance of the synchronized counter scales poorly. Whereas a single thread can complete the million counter updates in a tiny amount of time (roughly 0.03 seconds), having two threads each update the counter one million times concurrently leads to a massive slowdown (taking over 5 seconds!). It only gets worse with more threads.

```
typedef struct __counter_t {
    int             value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

Figure 29.2: **A Counter With Locks**

最理想的情况

Ideally, you'd like to see the threads complete just as quickly on multiple processors as the single thread does on one. Achieving this end is called **perfect scaling**; even though more work is done, it is done in parallel, and hence the time taken to complete the task is not increased.

Scalable Counting

Amazingly, researchers have studied how to build more scalable counters for years [MS04]. Even more amazing is the fact that scalable counters matter, as recent work in operating system performance analysis has shown [B+10]; without scalable counting, some workloads running on Linux suffer from serious scalability problems on multicore machines.

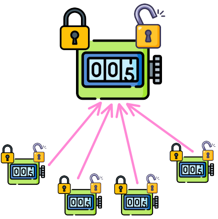
Many techniques have been developed to attack this problem. We'll describe one approach known as an **approximate counter** [C06].

其中一种方法

The approximate counter works by representing a single logical counter via numerous *local* physical counters, one per CPU core, as well as a single *global* counter. Specifically, on a machine with four CPUs, there are four

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	$5 \rightarrow 0$	1	3	4	5 (from L_1)
7	0	2	4	$5 \rightarrow 0$	10 (from L_4)

Figure 29.3: **Tracing the Approximate Counters**



local counters and one global one. In addition to these counters, there are also locks: one for each local counter[1], and one for the global counter.

The basic idea of approximate counting is as follows. When a thread running on a given core wishes to increment the counter, it increments its local counter; access to this local counter is synchronized via the corresponding local lock. Because each CPU has its own local counter, threads across CPUs can update local counters without contention, and thus updates to the counter are scalable.

However, to keep the global counter up to date (in case a thread wishes to read its value), the local values are periodically transferred to the global counter, by acquiring the global lock and incrementing it by the local counter's value; the local counter is then reset to zero.

How often this local-to-global transfer occurs is determined by a threshold S. The smaller S is, the more the counter behaves like the non-scalable counter above; the bigger S is, the more scalable the counter, but the further off the global value might be from the actual count. One could simply acquire all the local locks and the global lock (in a specified order, to avoid deadlock) to get an exact value, but that is not scalable.

To make this clear, let's look at an example (Figure 29.3). In this example, the threshold S is set to 5, and there are threads on each of four CPUs updating their local counters L_1 ... L_4. The global counter value (G) is also shown in the trace, with time increasing downward. At each time step, a local counter may be incremented; if the local value reaches the threshold S, the local value is transferred to the global counter and the local counter is reset.

The lower line in Figure 29.5 (labeled 'Approximate', on page 6) shows the performance of approximate counters with a threshold S of 1024. Performance is excellent; the time taken to update the counter four million times on four processors is hardly higher than the time taken to update it one million times on one processor.

[1] We need the local locks because we assume there may be more than one thread on each core. If, instead, only one thread ran on each core, no local lock would be needed.

```
typedef struct __counter_t {
    int             global;         // global count
    pthread_mutex_t glock;          // global lock
    int             local[NUMCPUS]; // per-CPU count
    pthread_mutex_t llock[NUMCPUS]; // ... and locks
    int             threshold;      // update freq
} counter_t;

// init: record threshold, init locks, init values
//       of all local counts and global count
void init(counter_t *c, int threshold) {
    c->threshold = threshold;
    c->global = 0;
    pthread_mutex_init(&c->glock, NULL);
    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;
        pthread_mutex_init(&c->llock[i], NULL);
    }
}

// update: usually, just grab local lock and update
// local amount; once it has risen ’threshold’,
// grab global lock and transfer local values to it
void update(counter_t *c, int threadID, int amt) {
    int cpu = threadID % NUMCPUS;
    pthread_mutex_lock(&c->llock[cpu]);
    c->local[cpu] += amt;
    if (c->local[cpu] >= c->threshold) {
        // transfer to global (assumes amt>0)
        pthread_mutex_lock(&c->glock);
        c->global += c->local[cpu];
        pthread_mutex_unlock(&c->glock);
        c->local[cpu] = 0;
    }
    pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount (approximate)
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);
    int val = c->global;
    pthread_mutex_unlock(&c->glock);
    return val; // only approximate!
}
```

Figure 29.4: **Approximate Counter Implementation**

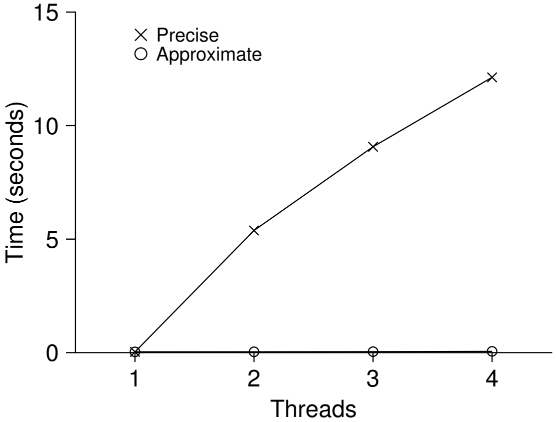


Figure 29.5: **Performance of Traditional vs. Approximate Counters**

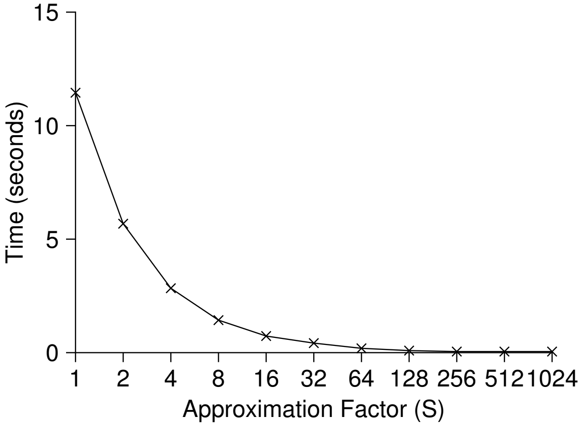


Figure 29.6: **Scaling Approximate Counters**

Figure 29.6 shows the importance of the threshold value S, with four threads each incrementing the counter 1 million times on four CPUs. If S is low, performance is poor (but the global count is always quite accurate); if S is high, performance is excellent, but the global count lags 延迟 (by at most the number of CPUs multiplied by S). This accuracy/performance tradeoff is what approximate counters enable.

A rough version of an approximate counter is found in Figure 29.4 (page 5). Read it, or better yet, run it yourself in some experiments to better understand how it works.

TIP: MORE CONCURRENCY ISN'T NECESSARILY FASTER

If the scheme you design adds a lot of overhead (for example, by acquiring and releasing locks frequently, instead of once), the fact that it is more concurrent may not be important. Simple schemes tend to work well, especially if they use costly routines rarely. Adding more locks and complexity can be your 衰败 downfall. All of that said, there is one way to really know: build both alternatives (simple but less concurrent, and complex but more concurrent) and measure how they do. In the end, you can't cheat on performance; your idea is either faster, or it isn't.

29.2 Concurrent Linked Lists

We next examine a more complicated structure, the linked list. Let's start with a basic approach once again. For simplicity, we'll omit some of the obvious routines that such a list would have and just focus on concurrent insert and lookup; we'll leave it to the reader to think about delete, etc. Figure 29.7 shows the code for this 初步的 rudimentary data structure.

As you can see in the code, the code simply acquires a lock in the insert routine upon entry, and releases it upon exit. One small tricky issue arises if `malloc()` happens to fail (a rare case); in this case, the code must also release the lock before failing the insert.

This kind of exceptional control flow has been shown to be quite error prone; a recent study of Linux kernel patches found that a huge fraction of bugs (nearly 40%) are found on such rarely-taken code paths (indeed, this observation sparked some of our own research, in which we removed all memory-failing paths from a Linux file system, resulting in a more robust system [S+11]).

简单的做法连最基本的要求都满足不了

Thus, a challenge: can we rewrite the insert and lookup routines to remain correct under concurrent insert but avoid the case where the failure path also requires us to add the call to unlock?

改进

The answer, in this case, is yes. Specifically, we can rearrange the code a bit so that the lock and release only surround the actual critical section in the insert code, and that a common exit path is used in the lookup code. The former works because part of the insert actually need not be locked; assuming that `malloc()` itself is thread-safe, each thread can call into it without worry of race conditions or other concurrency bugs. Only when updating the shared list does a lock need to be held. See Figure 29.8 for the details of these modifications.

As for the lookup routine, it is a simple code transformation to jump out of the main search loop to a single return path. Doing so again reduces the number of lock acquire/release points in the code, and thus decreases the chances of accidentally introducing bugs (such as forgetting to unlock before returning) into the code.

```
// basic node structure
typedef struct __node_t {
    int                 key;
    struct __node_t         *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
    node_t                *head;
    pthread_mutex_t     lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        pthread_mutex_unlock(&L->lock);
        return -1; // fail
    }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

Figure 29.7: **Concurrent Linked List**

```
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return -1;
    }
    new->key = key;
    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Figure 29.8: **Concurrent Linked List: Rewritten**

Scaling Linked Lists

Though we again have a basic concurrent linked list, once again we are in a situation where it does not scale particularly well. One technique that researchers have explored to enable more concurrency within a list is something called **hand-over-hand locking** (a.k.a. **lock coupling**) [MS04].

其中一种方法

The idea is pretty simple. Instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then releases the current node's lock (which inspires the name hand-over-hand).

TIP: BE WARY OF LOCKS AND CONTROL FLOW

A general design tip, which is useful in concurrent code as well as elsewhere, is to be wary of control flow changes that lead to function returns, exits, or other similar error conditions that halt the execution of a function. Because many functions will begin by acquiring a lock, allocating some memory, or doing other similar stateful operations, when errors arise, the code has to undo all of the state before returning, which is error-prone. Thus, it is best to structure code to minimize this pattern.

Conceptually, a hand-over-hand linked list makes some sense; it enables a high degree of concurrency in list operations. However, in practice, it is hard to make such a structure faster than the simple single lock approach, as the overheads of acquiring and releasing locks for each node of a list traversal is prohibitive. Even with very large lists, and a large number of threads, the concurrency enabled by allowing multiple ongoing traversals is unlikely to be faster than simply grabbing a single lock, performing an operation, and releasing it. Perhaps some kind of hybrid (where you grab a new lock every so many nodes) would be worth investigating.

29.3 Concurrent Queues

"基本款"读者自证不难

As you know by now, there is always a standard method to make a concurrent data structure: add a big lock. For a queue, we'll skip that approach, assuming you can figure it out.

其中一种方法

Instead, we'll take a look at a slightly more concurrent queue designed by Michael and Scott [MS98]. The data structures and code used for this queue are found in Figure 29.9 (page 11).

If you study this code carefully, you'll notice that there are two locks, one for the head of the queue, and one for the tail. The goal of these two locks is to enable concurrency of enqueue and dequeue operations. In the common case, the enqueue routine will only access the tail lock, and dequeue only the head lock.

One trick used by Michael and Scott is to add a dummy node (allocated in the queue initialization code); this dummy enables the separation of head and tail operations. Study the code, or better yet, type it in, run it, and measure it, to understand how it works deeply.

Queues are commonly used in multi-threaded applications. However, the type of queue used here (with just locks) often does not completely meet the needs of such programs. A more fully developed bounded queue, that enables a thread to wait if the queue is either empty or overly full, is the subject of our intense study in the next chapter on condition variables. Watch for it!

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t              *head;
    node_t              *tail;
    pthread_mutex_t     head_lock, tail_lock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->head_lock, NULL);
    pthread_mutex_init(&q->tail_lock, NULL);
}

void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tail_lock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tail_lock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->head_lock);
    node_t *tmp = q->head;
    node_t *new_head = tmp->next;
    if (new_head == NULL) {
        pthread_mutex_unlock(&q->head_lock);
        return -1; // queue was empty
    }
    *value = new_head->value;
    q->head = new_head;
    pthread_mutex_unlock(&q->head_lock);
    free(tmp);
    return 0;
}
```

Figure 29.9: **Michael and Scott Concurrent Queue**

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++)
        List_Init(&H->lists[i]);
}

int Hash_Insert(hash_t *H, int key) {
    return List_Insert(&H->lists[key % BUCKETS], key);
}

int Hash_Lookup(hash_t *H, int key) {
    return List_Lookup(&H->lists[key % BUCKETS], key);
}
```

Figure 29.10: **A Concurrent Hash Table**

29.4 Concurrent Hash Table

We end our discussion with a simple and widely applicable concurrent data structure, the hash table. We'll focus on a simple hash table that does not resize; a little more work is required to handle resizing, which we leave as an exercise for the reader (sorry!).

留作习题答案略

This concurrent hash table (Figure 29.10) is straightforward, is built using the concurrent lists we developed earlier, and works incredibly well. The reason for its good performance is that instead of having a single lock for the entire structure, it uses a lock per hash bucket (each of which is represented by a list). Doing so enables many concurrent operations to take place.

Figure 29.11 (page 13) shows the performance of the hash table under concurrent updates (from 10,000 to 50,000 concurrent updates from each of four threads, on the same iMac with four CPUs). Also shown, for the sake of comparison, is the performance of a linked list (with a single lock). As you can see from the graph, this simple concurrent hash table scales magnificently; the linked list, in contrast, does not.

极好地

29.5 Summary

无处不在的

We have introduced a sampling of concurrent data structures, from counters, to lists and queues, and finally to the ubiquitous and heavily-used hash table. We have learned a few important lessons along the way: to be careful with acquisition and release of locks around control flow

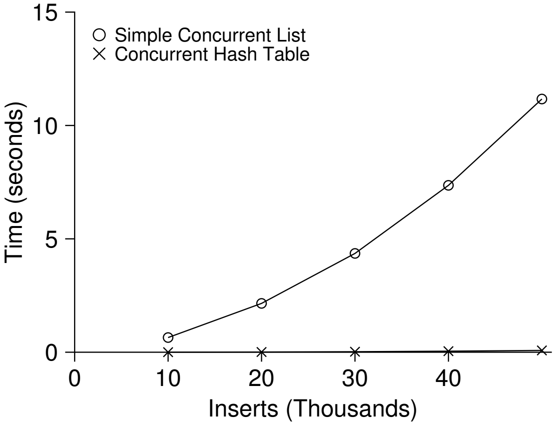


Figure 29.11: **Scaling Hash Tables**

changes; that enabling more concurrency does not necessarily increase performance; that performance problems should only be remedied once they exist. This last point, of avoiding **premature optimization**, is central to any performance-minded developer; there is no value in making something faster if doing so will not improve the overall performance of the application.

纠正

Of course, we have just scratched the surface of high performance structures. See Moir and Shavit's excellent survey for more information, as well as links to other sources [MS04]. In particular, you might be interested in other structures (such as B-trees); for this knowledge, a database class is your best bet. You also might be curious about techniques that don't use traditional locks at all; such **non-blocking data structures** are something we'll get a taste of in the chapter on common concurrency bugs, but frankly this topic is an entire area of knowledge requiring more study than is possible in this humble book. Find out more on your own if you desire (as always!).

TIP: AVOID PREMATURE OPTIMIZATION (KNUTH'S LAW)

When building a concurrent data structure, start with the most basic approach, which is to add a single big lock to provide synchronized access. By doing so, you are likely to build a *correct* lock; if you then find that it suffers from performance problems, you can refine it, thus only making it fast if need be. As **Knuth** famously stated, "Premature optimization is the root of all evil."

Many operating systems utilized a single lock when first transitioning to multiprocessors, including Sun OS and Linux. In the latter, this lock even had a name, the **big kernel lock** (**BKL**). For many years, this simple approach was a good one, but when multi-CPU systems became the norm, only allowing a single active thread in the kernel at a time became a performance bottleneck. Thus, it was finally time to add the optimization of improved concurrency to these systems. Within Linux, the more straightforward approach was taken: replace one lock with many. Within Sun, a more radical decision was made: build a brand new operating system, known as Solaris, that incorporates concurrency more fundamentally from day one. Read the Linux and Solaris kernel books for more information about these fascinating systems [BC05, MM00].

References

[B+10] "An Analysis of Linux Scalability to Many Cores" by Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich . OSDI '10, Vancouver, Canada, October 2010. *A great study of how Linux performs on multicore machines, as well as some simple solutions. Includes a neat* ***sloppy counter*** *to solve one form of the scalable counting problem.*

[BH73] "Operating System Principles" by Per Brinch Hansen. Prentice-Hall, 1973. Available: `http://portal.acm.org/citation.cfm?id=540365`. *One of the first books on operating systems; certainly ahead of its time. Introduced monitors as a concurrency primitive.*

[BC05] "Understanding the Linux Kernel (Third Edition)" by Daniel P. Bovet and Marco Cesati. O'Reilly Media, November 2005. *The classic book on the Linux kernel. You should read it.*

[C06] "The Search For Fast, Scalable Counters" by Jonathan Corbet. February 1, 2006. Available: `https://lwn.net/Articles/170003`. *LWN has many wonderful articles about the latest in Linux. This article is a short description of scalable approximate counting; read it, and others, to learn more about the latest in Linux.*

[L+13] "A Study of Linux File System Evolution" by Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu. FAST '13, San Jose, CA, February 2013. *Our paper that studies every patch to Linux file systems over nearly a decade. Lots of fun findings in there; read it to see! The work was painful to do though; the poor graduate student, Lanyue Lu, had to look through every single patch by hand in order to understand what they did.*

[MS98] "Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared-memory Multiprocessors" by M. Michael, M. Scott. Journal of Parallel and Distributed Computing, Vol. 51, No. 1, 1998. *Professor Scott and his students have been at the forefront of concurrent algorithms and data structures for many years; check out his web page, numerous papers, or books to find out more.*

[MS04] "Concurrent Data Structures" by Mark Moir and Nir Shavit. In Handbook of Data Structures and Applications (Editors D. Metha and S.Sahni). Chapman and Hall/CRC Press, 2004. Available: `www.ostep.org/Citations/concurrent.pdf`. *A short but relatively comprehensive reference on concurrent data structures. Though it is missing some of the latest works in the area (due to its age), it remains an incredibly useful reference.*

[MM00] "Solaris Internals: Core Kernel Architecture" by Jim Mauro and Richard McDougall. Prentice Hall, October 2000. *The Solaris book. You should also read this, if you want to learn about something other than Linux.*

[S+11] "Making the Common Case the Only Case with Anticipatory Memory Allocation" by Swaminathan Sundararaman, Yupu Zhang, Sriram Subramanian, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau . FAST '11, San Jose, CA, February 2011. *Our work on removing possibly-failing allocation calls from kernel code paths. By allocating all potentially needed memory before doing any work, we avoid failure deep down in the storage stack.*