



Multithread Programming

Alexandre David

1.2.05

adavid@cs.aau.dk



Comparison

- Directive based: OpenMP.
- Explicit parallel programming:
 - pthreads – shared memory – focus on synchronization,
 - MPI – message passing – focus on communication.
 - Both: Specify tasks & interactions.



Programming Models

- Concurrency supported by:
 - Processes – private data unless otherwise specified.
 - Threads – shared memory, lightweight.
 - Directive based programming – concurrency specified as high level compiler directive, OpenMP.
- See OS course.



Threads Basics

- All memory is globally accessible.
- But the stack is considered local.
 - In practice both local (private) and global (shared) memory.
 - Recall that memory is physically distributed and local accesses are faster.
 - Applicable to SMP/multi-core machines.



Why Threads?

- **Software portability** – applications developed and run **without modification** on multi-processor machines.
- **Latency hiding** – recall chapter 2.
- ❓ ■ **Implicit scheduling and load balancing** – specify many tasks and let the system map and schedule them.
- **Ease of programming**, widespread.
- **Performance & correctness issues.**



The POSIX Thread API

- It is a **standard** API (like MPI).
 - Supported by most vendors.
- General concepts applicable to other thread APIs (java threads, NT threads, etc).
- **Low level functions**, API is missing high level constructs, e.g., no collective communication like in MPI.



Thread Life Cycle

- How to create a thread:
 - creation function
 - the thread runs the specified function
- How to destroy a thread:
 - the thread returns from the function or
 - it calls `pthread_exit` or
 - it is cancelled by another thread
- How to clean-up/read status
 - `join`

Thread Creation

```
#include <pthread.h>
```

Header.

```
int pthread_create(
```

```
    pthread_t *thread_handle,
```

Identifier.

```
    const pthread_attr_t *attribute,
```

NULL for default.

```
    void* (*thread_function)(void *),  
    void *arg);
```

Function to call with its argument.

Waiting for Termination

```
int pthread_join(  
    pthread_t thread,  
    void ** ptr);
```

← Thread to wait for.

↑
Threads call **pthread_exit(void*)**.
The caller can read a (void*) at address *ptr*.

The creator process/thread calls this function to wait for its spawned threads.



Misc

```
void pthread_exit(void*);
```

```
pthread_t pthread_self();
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```



Thread Cancellation

- Stop a thread in the middle of its work.
- Function may return before the thread is really stopped!

```
int pthread_cancel(pthread_t thread);
```



Attributes

Thread Attributes

```
pthread_attr_t attr;           // Declare a thread attribute
pthread_t tid;
pthread_attr_init(&attr);      // Initialize a thread attribute
pthread_attr_setdetachstate(&attr, // Set the thread attribute
                          PTHREAD_CREATE_UNDETACHED);
pthread_create(&tid, &attr, start_func, NULL); // Use the attribute
                                                    // to create a thread

pthread_join(tid, NULL);
pthread_attr_destroy(&attr);    // Destroy the thread attribute
```

Notes:

There are many other thread attributes. See a POSIX Threads manual for details.

Threads can be detached/joinable or bound/unbound.



Example

```
#include <pthread.h>
int err;                bad practice
    ...                assume some code here
void main()            should be int main(int argc, char* argv[])
{
    pthread_t tid[MAX];    don't declare MAX as constant
    for(i = 0; i < t; i++)    t undeclared, should be MAX here
    {
        err=pthread_create(&tid[i], NULL, job, i);
    }                    careful with the argument – race condition
    for(i = 0; i < t; i++)
    {
        err=pthread_join_(tid[i], (void**) &status[i])    typos
    }
}
```



Example - Thread

```
void job() wrong signature
{
    int errorcode;

    ...
    if (blabla)
    {
        errorcode = boom;
    }
    pthread_exit(&errorcode); problem with address
}
```



Good Way

```
typedef struct
```

```
{
```

```
    int id; // input data
```

```
    ...
```

```
    int result; // output data
```

```
    int errorcode;
```

```
    ...
```

```
} thread_data_t;
```

```
void* job(thread_data_t *data) or
```

```
{
```

```
    ...
```

```
}
```

```
void* job(void* arg)
```

```
{
```

```
    thread_data_t *data = (thread_data_t*) arg;
```

```
    ...
```

```
}
```



Mutex-Locks

- Implement critical section.
- Mutex-locks can be **locked** or **unlocked**.
 - Locking is atomic.
 - Threads must **acquire a lock** to enter a critical section.
 - Threads must **release their locks** when leaving a critical section.
- Locks represent **serialization points**. Too many locks will **decrease performance**.



Mutex-Lock

```
int pthread_mutex_init(  
    pthread_mutex_t *mutex_lock,  
    const pthread_mutexattr_t *lock_attr);
```

```
int pthread_mutex_lock(  
    pthread_mutex_t *mutex_lock);
```

```
int pthread_mutex_unlock(  
    pthread_mutex_t *mutex_lock);
```



Attribute Objects

- To control threads and synchronization.
 - Change scheduling policy...
 - Specify mutex types.
- Types of mutexes:
 - Normal – 1 lock per thread or deadlock.
 - Recursive – several locks per thread OK.
 - Error check – 1 lock per thread or error.



Overhead of Locking

- Locks represent serialization points.
 - Keep critical sections small.
 - Previous example: create & process tasks outside the section.
- Faster variant:

```
int pthread_mutex_trylock(  
    pthread_mutex_t *mutex_lock);
```

Does not block, returns EBUSY if failed.



Try-lock

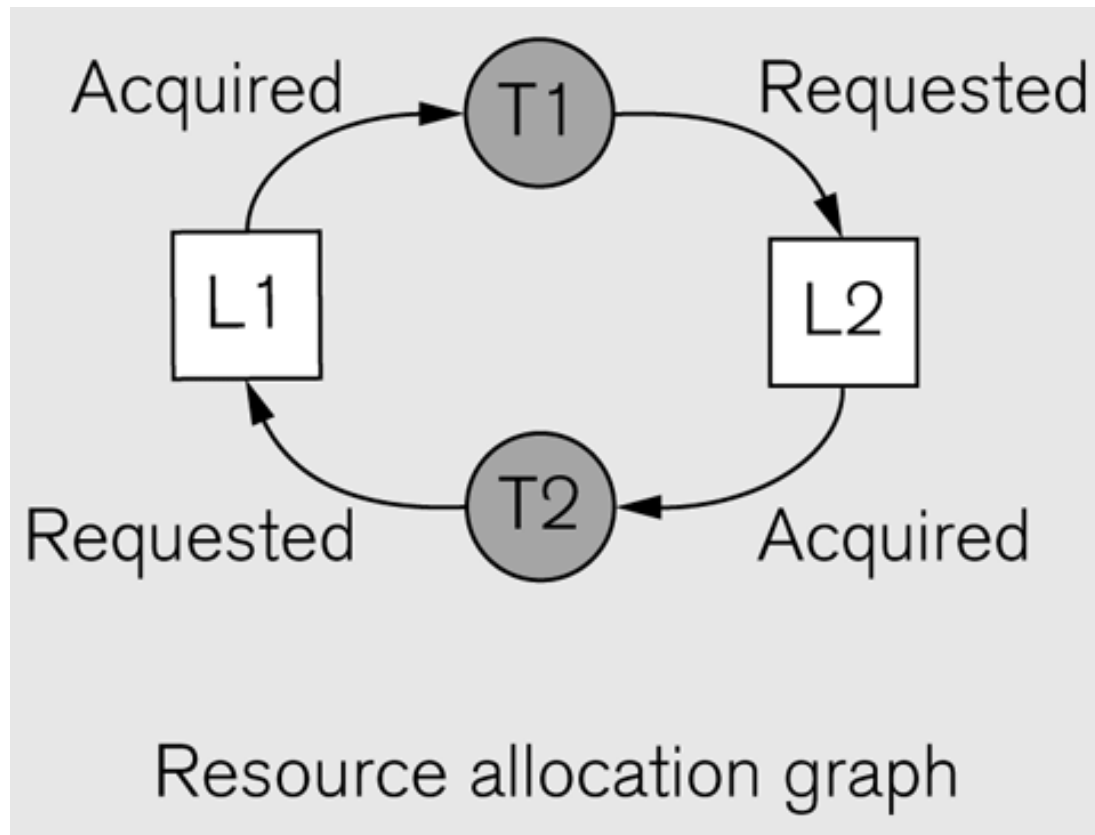
- To reduce idling overheads.
- Good if critical section can be delayed.
- Cheaper call.
 - Although it is polling.



Issues

- Deadlocks
 - A locks M1 and M2, B locks M2 and M1...
 - 4 necessary conditions
 - mutual exclusion – resource assigned to ≤ 1 thread
 - hold and wait – hold resource and wait for another
 - no preemption – only the owner can release its resource
 - circular wait – circular dependency
 - Rule of thumb: Have a global order for locking, unlock in reverse order – **lock hierarchy**.
- Lock non initialized mutex: not good.
- Unlock twice: not good.
- Monitors: better to encapsulate.

Resource Allocation Graph





Condition Variables for Synchronization

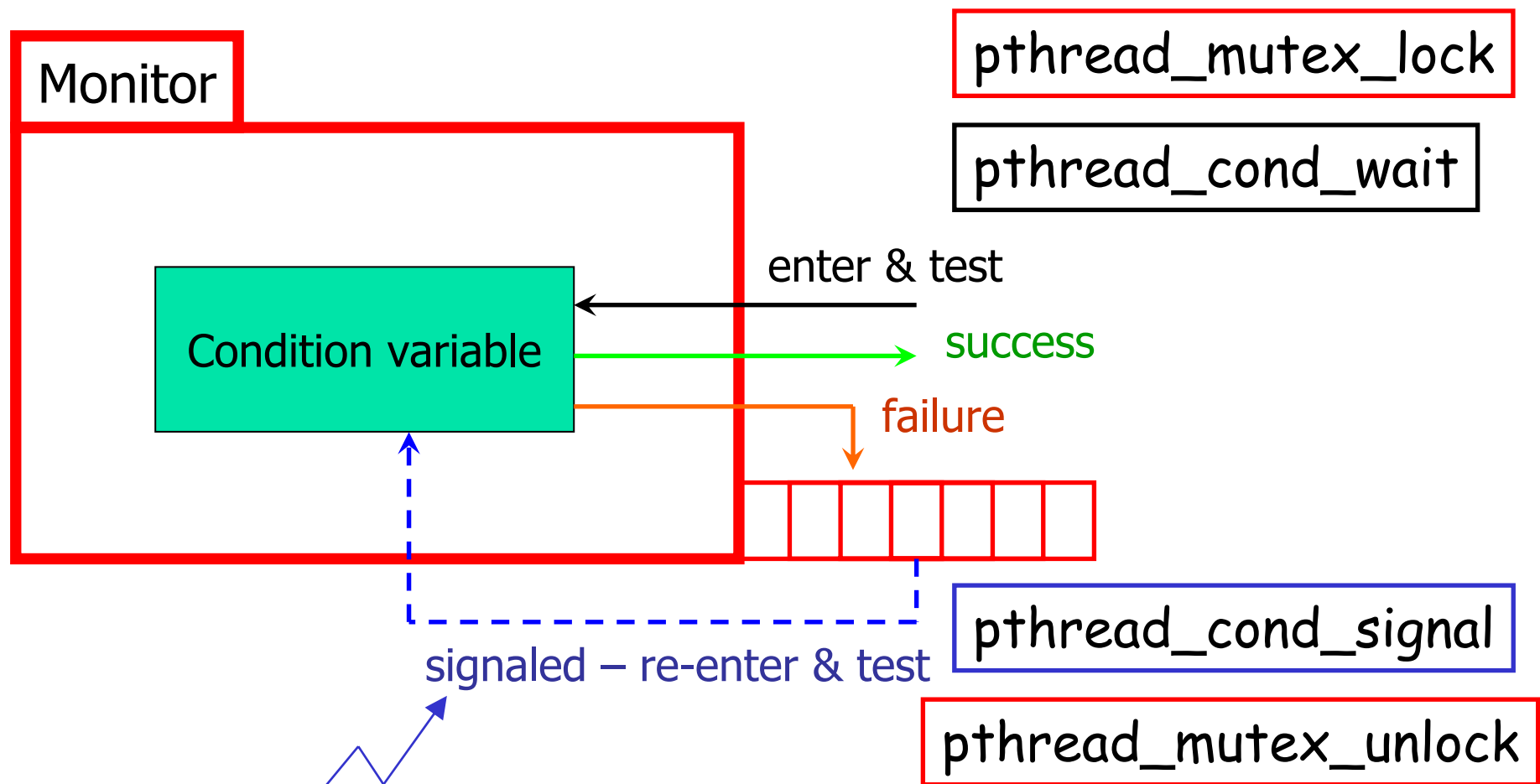
One condition variable \Leftrightarrow one predicate.

- How to implement condition variables with monitors.

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

- A **condition variable** is always associated with a **mutex**.
- Lock/unlock to test & wait, re-lock/unlock to re-test.
- Similar concept of monitors in Java, though implemented differently.

Condition Variables & Monitors





Monitors with Pthread

```
pthread_mutex_lock(&lock);  
while(!predicate) {  
    pthread_cond_wait(&condition, &lock);  
}  
<critical section>  
pthread_cond_signal(&condition);  
pthread_mutex_unlock(&lock);
```



Monitors in Java

```
synchronized void foo() {  
    while(!predicate) wait();  
    <critical section>  
    notify();  
}
```



Monitors in C#

```
using System.Threading;
```

```
...
```

```
void foo() {
```

```
    Monitor.enter(obj);
```

```
    while(!predicate) Monitor.wait(obj);
```

```
    <critical section>
```

```
    Monitor.pulse(obj);
```

```
    Monitor.exit(obj);
```

```
}
```



Calls

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond,  
                     const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```



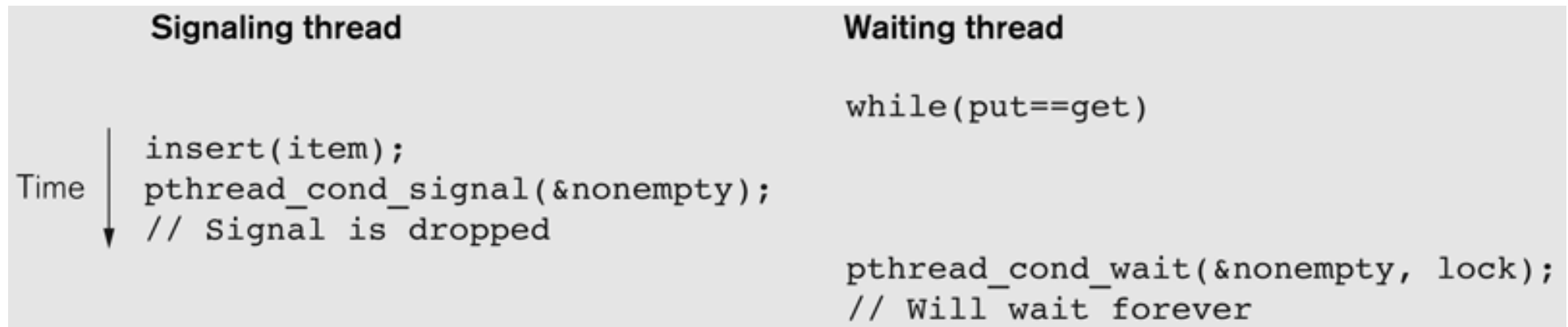
Performance Issues

- Too many locks/conditions – overhead.
- Too few conditions – spurious wake-ups.



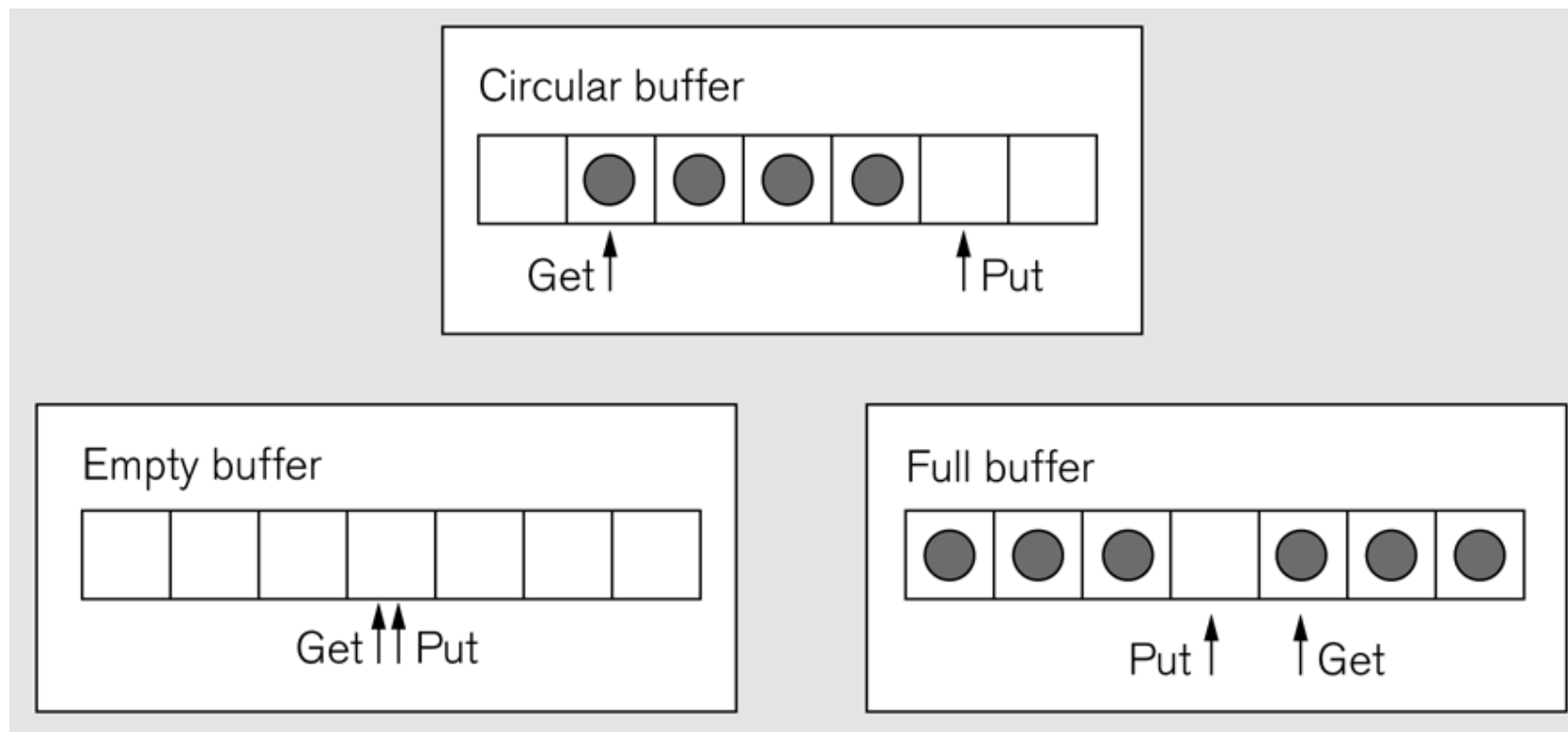
Patterns

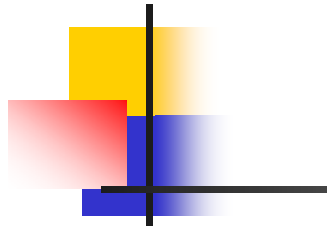
- Follow these patterns.
- Signal if nobody is waiting: nothing.
 - Signal inside the lock.



- Race in testing conditions.
 - while loop.
- Implicit lock.

Example





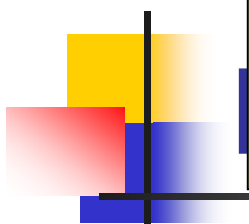
```
1 pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3 pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4 Item buffer[SIZE];
5 int put=0; // Buff index for next insert
6 int get=0; // Buff index for next remove
7
8 void insert(Item x) // Producer thread
9 {
10 pthread_mutex_lock(&lock);
11 while((put>get&&(put-get)==SIZE-1)|| // While buffer is
12 (put<get&&(put+get)==SIZE-1)) // full
13 {
14 pthread_cond_wait(&nonfull, &lock);
15 }
16 buffer[put]=x;
17 put=(put+1)%SIZE;
18 pthread_cond_signal(&nonempty);
19 pthread_mutex_unlock(&lock);
20 }
21
22 Item remove() // Consumer thread
23 {
24 Item x;
25 pthread_mutex_lock(&lock);
26 while(put==get) // While buffer is empty
27 {
28 pthread_cond_wait(&nonempty, &lock);
29 }
30 x=buffer[get];
31 get=(get+1)%SIZE;
32 pthread_cond_signal(&nonfull);
33 pthread_mutex_unlock(&lock);
34 return x;
35 }
```




Example: Producer-Consumer

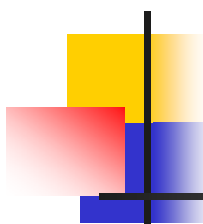
```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;

...
main() {
    ...
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    ... /* create and join producer and consumer threads */
}
```



```
task_available == 0 ⇔ cond_queue_empty  
task_available == 1 ⇔ cond_queue_full
```

```
void *producer(void *producer_thread_data) {  
    int inserted;  
    while (!done()) {  
        create_task();  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (!(task_available == 0)) {  
            pthread_cond_wait(&cond_queue_empty,  
                             &task_queue_cond_lock);  
        }  
        insert_into_queue();  
        task_available = 1;  
        pthread_cond_signal(&cond_queue_full);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
    }  
}
```



```
task_available == 0 ⇔ cond_queue_empty  
task_available == 1 ⇔ cond_queue_full
```

```
void *consumer(void *consumer_thread_data) {  
    while (!done()) {  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (!(task_available == 1)) {  
            pthread_cond_wait(&cond_queue_full,  
                             &task_queue_cond_lock);  
        }  
        my_task = extract_from_queue();  
        task_available = 0;  
        pthread_cond_signal(&cond_queue_empty);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
        process_task(my_task);  
    }  
}
```



Waiting on Multiple Conditions

lock

while not(conjunction of all conditions)

{

 wait(condition 1)

 wait(condition 2)

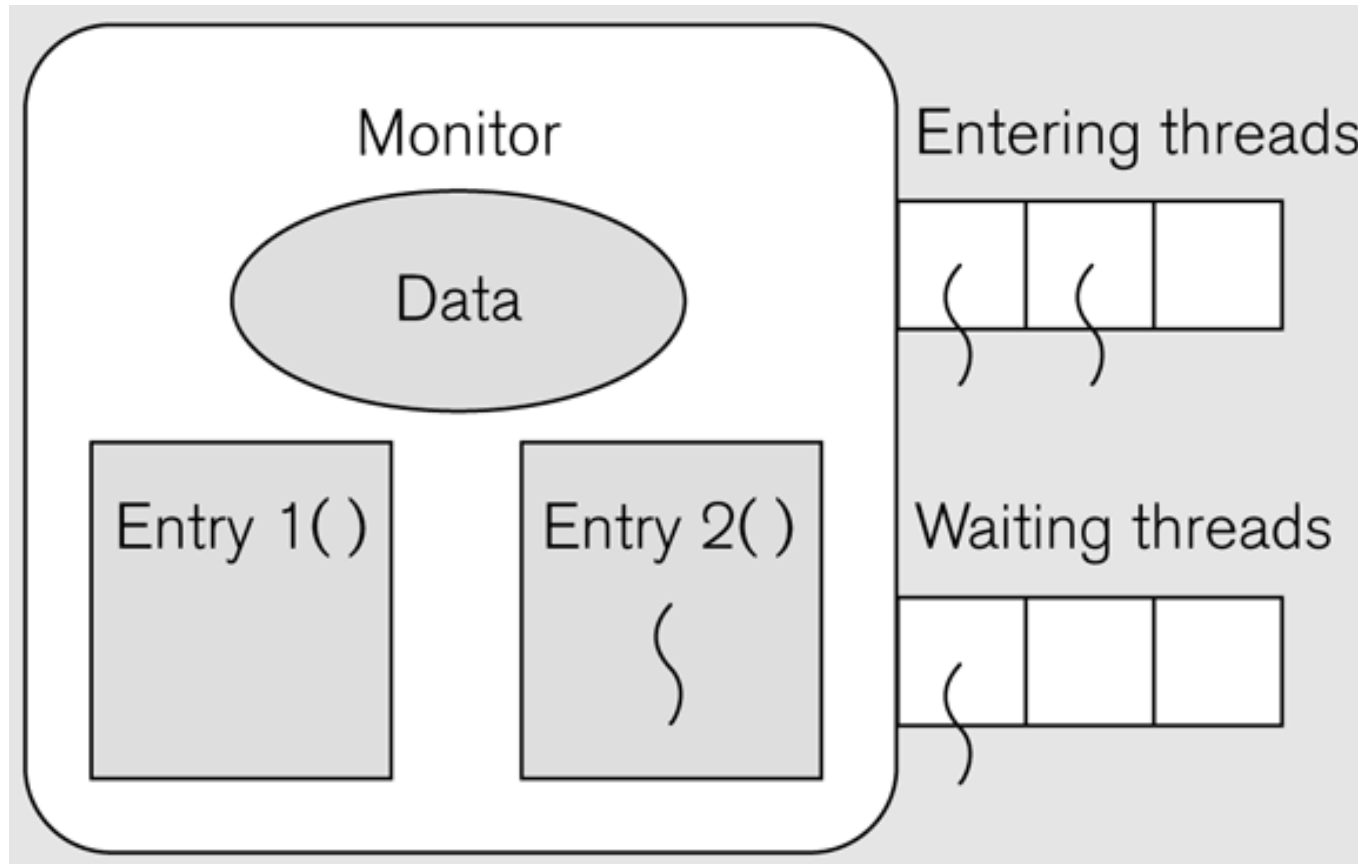
 ...

}

signal if needed

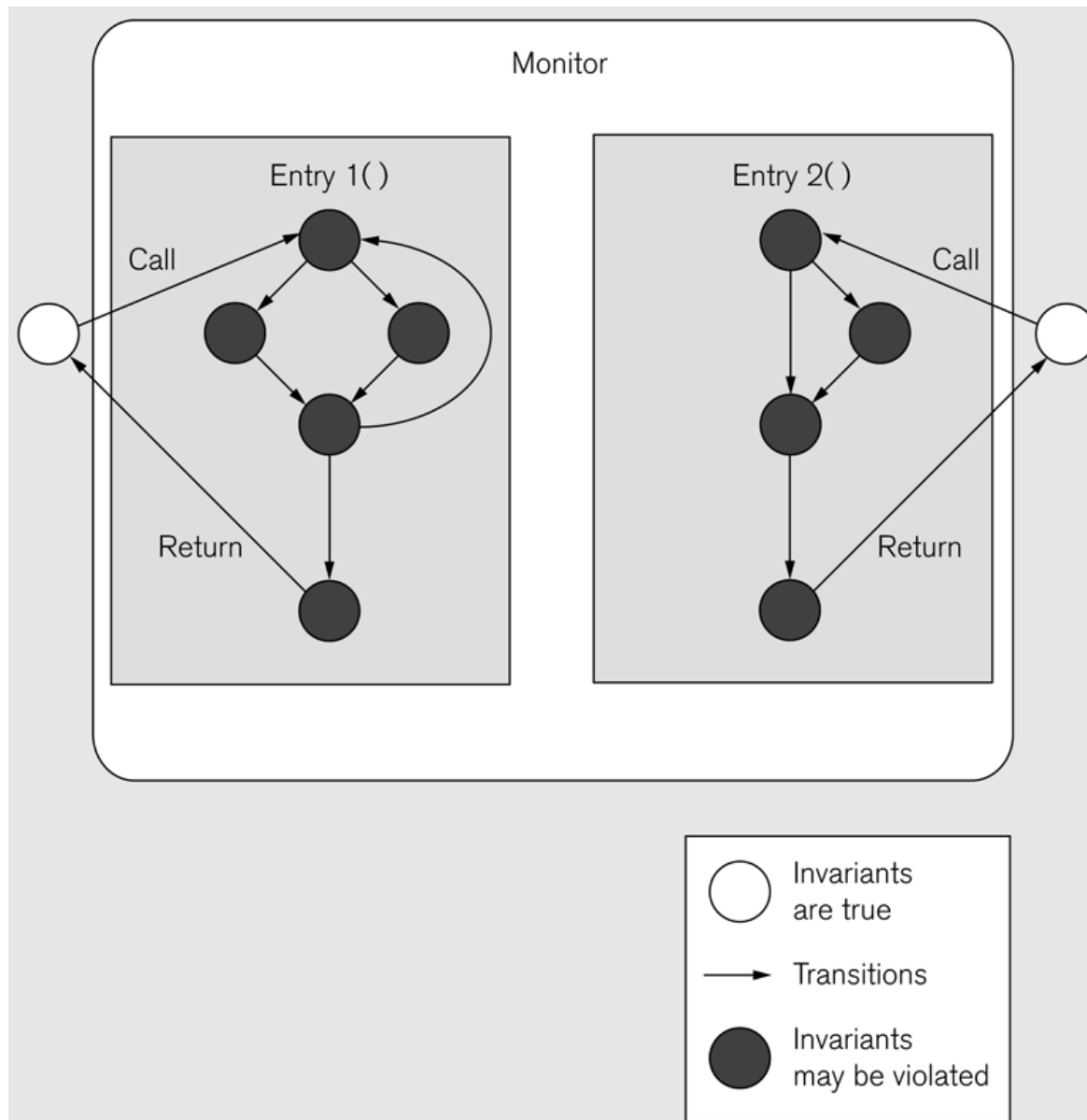
unlock

Practice with Monitors



Encapsulate to preserve **invariants**

Invariants





Error

```
47     in++;
48     pthread_cond_signal(&nonempty);
49     pthread_mutex_unlock(&lock);
50 }
51
52 Item BoundedBuffer::Get()
53 {
54     pthread_mutex_lock(&lock);
55     while(in==out)                // While buffer is empty
56     {
57         pthread_cond_wait(&nonempty, &lock);
58     }
59     x=buffer[out%size];
60     out++;                        overflow
61     pthread_cond_signal(&nonfull);
62     pthread_mutex_unlock(&lock);
63     return x;
64 }
```

RW Example

```
1 int readers; // Neg value=> active writer
2 pthread_mutex_t lock;
3 pthread_cond_t rBusy, wBusy; // Use separate conditional vars
4 // for readers and writers
5 AcquireExclusive()
6 {
7     pthread_mutex_lock(&lock);
8     while(readers !=0)
9     {
10        pthread_cond_wait(&wBusy, &lock);
11    }
12    readers=-1;
13    pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18     pthread_mutex_lock(&lock);
19     readWaiters++;
20     while(readers<0)
21     {
22        pthread_cond_wait(&rBusy, &lock);
23    }
24    readWaiters--;
25    pthread_mutex_unlock(&lock);
26 }
27
28 ReleaseExclusive()
29 {
30     pthread_mutex_lock(&lock);
31     readers=0;
32     pthread_cond_broadcast(&rBusy); // Only w
33     pthread_mutex_unlock(&lock);
34 }
35
36 ReleaseShared(
37 {
38     int doSignal;
39
40     pthread_mutex_lock(&lock);
41     readers--;
42     doSignal=(readers==0)
43     pthread_mutex_unlock(&lock);
44     if(doSignal) // Signal
45     { // of crit
46         pthread_cond_signal(&wBusy); // Wake up
47     }
48 }
```

```
1 int readers;
2 pthread_mutex_t lock;
3 pthread_cond_t rBusy, wBusy;
4
5 AcquireExclusive()
6 {
7     pthread_mutex_lock(&lock);
8     while(readers !=0)
9     {
10        pthread_cond_wait(&wBusy, &lock);
11    }
12    readers=-1;
13    pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18     pthread_mutex_lock(&lock);
19     readWaiters++;
20     while(readers<0)
21     {
```



```

1  int readers; // Neg va:10
2  pthread_mutex_t lock;
3  pthread_cond_t rBusy, wBusy; // Use se:11
4  // for re:12
5  AcquireExclusive()
6  {
7      pthread_mutex_lock(&lock);
8      while(readers !=0)
9      {
10         pthread_cond_wait(&wBusy, &lock);
11     }
12     readers=-1;
13     pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18     pthread_mutex_lock(&lock);
19     readWaiters++;
20     while(readers<0)
21     {
22         pthread_cond_wait(&rBusy, &lock);
23     }
24     readWaiters--;
25     pthread_mutex_unlock(&lock);
26 }
27
28 ReleaseExclusive()
29 {
30     pthread_mutex_lock(&lock);
31     readers=0;
32     pthread_cond_broadcast(&rBusy); // Only w:28
33     pthread_mutex_unlock(&lock);
34 }
35
36 ReleaseShared(
37 {
38     int doSignal;
39
40     pthread_mutex_lock(&lock);
41     readers--;
42     doSignal=(readers==0)
43     pthread_mutex_unlock(&lock);
44     if(doSignal) // Signal
45     { // of crit
46         pthread_cond_signal(&wBusy); // Wake up:36
47     }
48 }

```

```

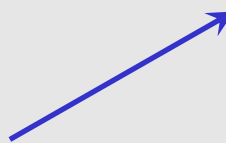
10     pthread_cond_wait(&wBusy, &lock);
11 }
12     readers=-1;
13     pthread_mutex_unlock(&lock);
14 }
15
16 AcquireShared()
17 {
18     pthread_mutex_lock(&lock);
19     readWaiters++;
20     while(readers<0)
21     {
22         pthread_cond_wait(&rBusy, &lock);
23     }
24     readWaiters--;
25     pthread_mutex_unlock(&lock);
26 }
27
28 ReleaseExclusive()
29 {
30     pthread_mutex_lock(&lock);
31     readers=0;
32     pthread_cond_broadcast(&rBusy);
33     pthread_mutex_unlock(&lock);
34 }
35
36 ReleaseShared(
37 {

```

```

1  int readers; // Neg va: 21
2  pthread_mutex_t lock; // Use se: 22
3  pthread_cond_t rBusy, wBusy; // for re: 23
4
5  AcquireExclusive() 24
6  { 25
7      pthread_mutex_lock(&lock); 26
8      while(readers !=0) 27
9      { 28
10         pthread_cond_wait(&wBusy, &lock); 29
11     } 30
12     readers=-1; 31
13     pthread_mutex_unlock(&lock); 32
14 } 33
15
16 AcquireShared() 34
17 { 35
18     pthread_mutex_lock(&lock); 36
19     readWaiters++; 37
20     while(readers<0) 38
21     { 39
22         pthread_cond_wait(&rBusy, &lock); 40
23     } 41
24     readWaiters--; 42
25     pthread_mutex_unlock(&lock); 43
26 } 44
27
28 ReleaseExclusive() 45
29 { 46
30     pthread_mutex_lock(&lock); 47
31     readers=0; 48
32     pthread_cond_broadcast(&rBusy); // Only w:
33     pthread_mutex_unlock(&lock);
34 }
35
36 ReleaseShared(
37 {
38     int doSignal;
39
40     pthread_mutex_lock(&lock);
41     readers--;
42     doSignal=(readers==0)
43     pthread_mutex_unlock(&lock);
44     if(doSignal)
45     {
46         pthread_cond_signal(&wBusy); // Signal
47     } // of crit
48 } // Wake up

```





Spin-locks

- Mutex:
 - Threads block until the lock is acquired.
 - Blocked threads are idle and need to wake up.

- Spin-locks:
 - Threads spin until the lock is acquired.
 - Blocked threads are not idle!
 - Better for quick access of **small** critical sections with low contention.



Pthread spin locks

- Calls:

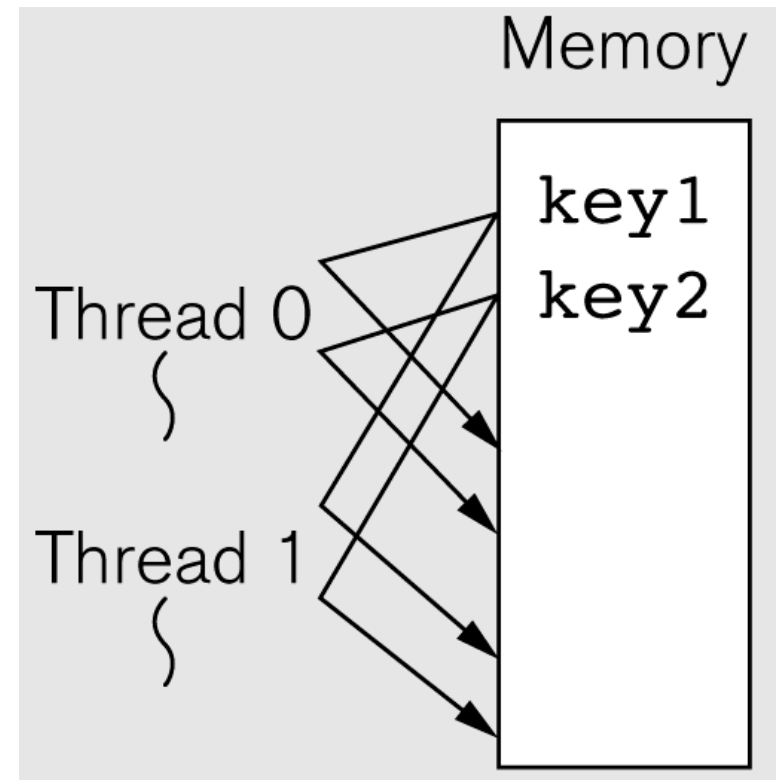
- `pthread_spin_init(pthread_spinlock_t*, int)`
`pthread_spin_destroy(pthread_spinlock_t*)`

- `pthread_spin_lock(pthread_spinlock_t*)`
`pthread_spin_trylock(pthread_spinlock_t*)`
`pthread_spin_unlock(pthread_spinlock_t*)`

- **Not** related to condition variables because threads do not wait and are not woken up!

Thread Specific Data

- Pass through thread index or
- Map keys to values/pointers.





Thread Specific Data

Thread-Specific Data

```
pthread_key_t *my_index;
#define index(pthread_getspecific(my_index))
main()
{
    ...
    pthread_key_create(&my_index, 0);
    ...
}
void start_routine(int id)
{
    pthread_setspecific(my_index, id);
    ...
}
```

Notes:

Avoid accessing `index` in a tight inner loop because each access requires a procedure call.



Composite Synchronization Constructs

- Pthread API offers (low-level) basic functions.
- Higher level constructs built with basic functions.
 - Read-write locks.
 - Barriers.
 - ... well in fact these two are part of the API.



Read-Write Locks (revisited)

- Read often/write sometimes.
 - Multiple reads/unique write.
 - Priority of writers over readers.
- Use condition variables.
 - Count readers and writers.
 - readers_proceed
⇔ pending_writers == 0 && writer == 0.
 - writer_proceed
⇔ writer == 0 && readers == 0.



Read-Write Lock - RLocking

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> pending_writers > 0) || (l -> writer > 0)) {
        pthread_cond_wait(&(l -> readers_proceed),
            &(l -> read_write_lock));
    }
    l -> readers ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```



Read-Write Lock - WLocking

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {  
    pthread_mutex_lock(&(l -> read_write_lock));  
    while ((l -> writer > 0) || (l -> readers > 0)) {  
        l -> pending_writers ++;  
        pthread_cond_wait(&(l -> writer_proceed),  
                        &(l -> read_write_lock));  
        l -> pending_writers --;  
    }  
    l -> writer ++;  
    pthread_mutex_unlock(&(l -> read_write_lock));  
}
```



Read-Write Lock - Unlocking

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l -> read_write_lock));
    if (l -> writer > 0) {
        l -> writer = 0;
    } else if (l -> readers > 0) {
        l -> readers --;
    }
    if ((l -> readers == 0) && (l -> pending_writers > 0)) {
        pthread_cond_signal(&(l -> writer_proceed));
    } else {
        pthread_cond_broadcast(&(l -> readers_proceed));
    }
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```



Barriers

- Encoded with
 - a counter,
 - a mutex, and
 - a condition variable.
- Idea:
 - Count & block threads.
 - Signal them all.
- Linear & log barriers.



Barriers

```
void mylib_barrier(mylib_barrier_t *b, int num_threads) {  
    pthread_mutex_lock(&(b -> count_lock));  
    b -> count ++;  
    if (b -> count == num_threads) { /* last thread */  
        b -> count = 0;  
        pthread_cond_broadcast(&(b -> ok_to_proceed));  
    } else {  
        pthread_cond_wait(&(b -> ok_to_proceed),  
                        &(b -> count_lock));  
    }  
    pthread_mutex_unlock(&(b -> count_lock));  
}
```



Semaphores

- Special counter
 - inc & dec atomic
 - no access to its value
 - wait for counter > 0 & dec
 - inc & signal a blocked thread/process.

- Initial counter
 - if $== 0$, useful for synchronizing.
 - if $== n (> 0)$, useful for allowing at most n threads/processes in a critical section.



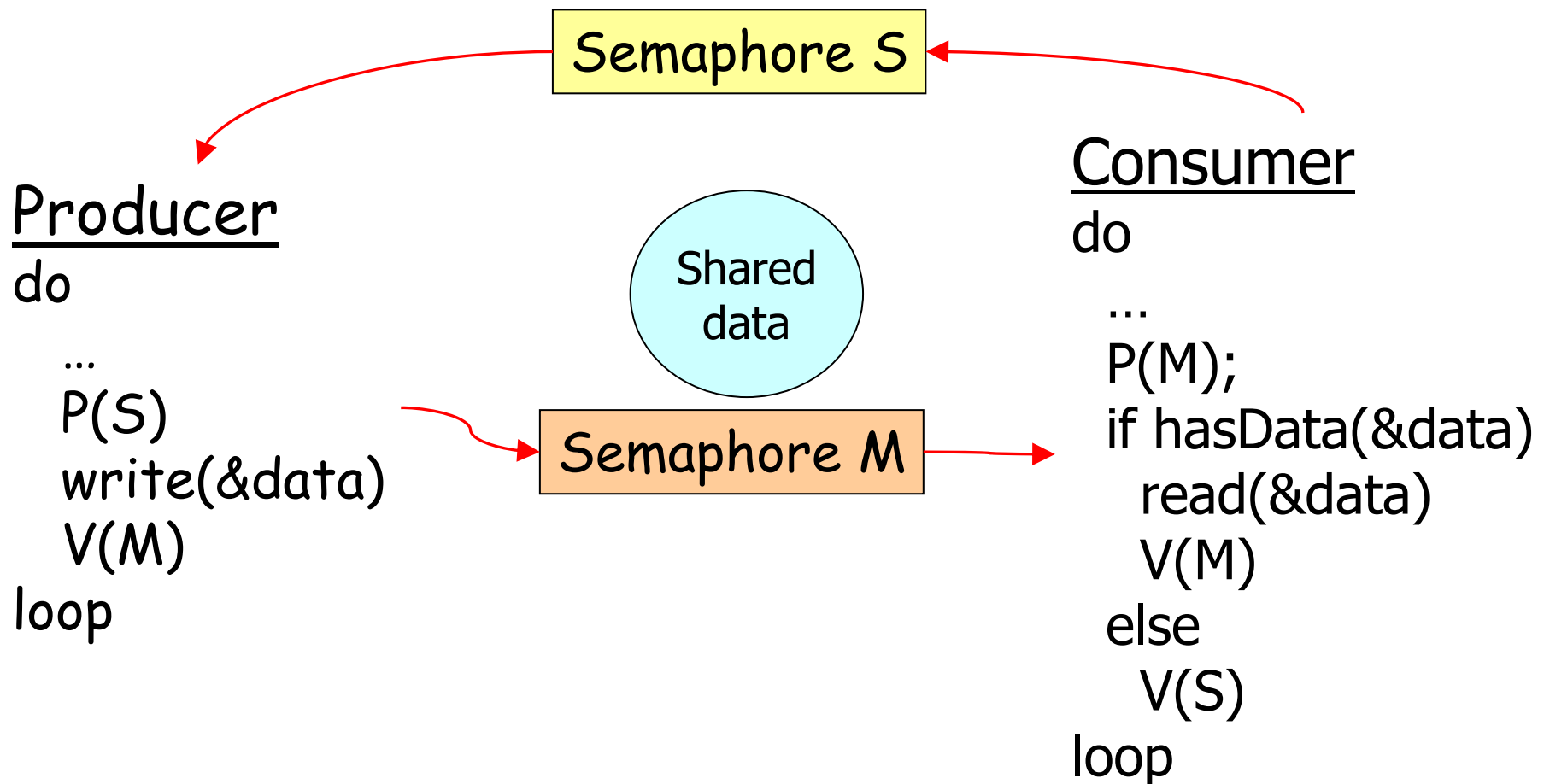
Semaphores

```
#include <semaphore.h>
```

■ Calls

- `sem_init(sem_t*, int, unsigned int value)`
`sem_destroy(sem_t*)`
- `sem_wait(sem_t*)`
`sem_trywait(sem_t*)`
`sem_timedwait(sem_t*, const struct timespec*)`
- `sem_post(sem_t*)`

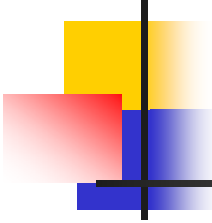
Producer-Consumers with Semaphores





Case Study

- Successive over-relaxation.
 - Iteratively compute average values.
 - Dependency between iteration.
- Partitioning?
 - Identical operations uniformly distributed.
 - Minimize communication, maximize locality.
 - Block decomposition of identical size.
- Synchronization: barrier.

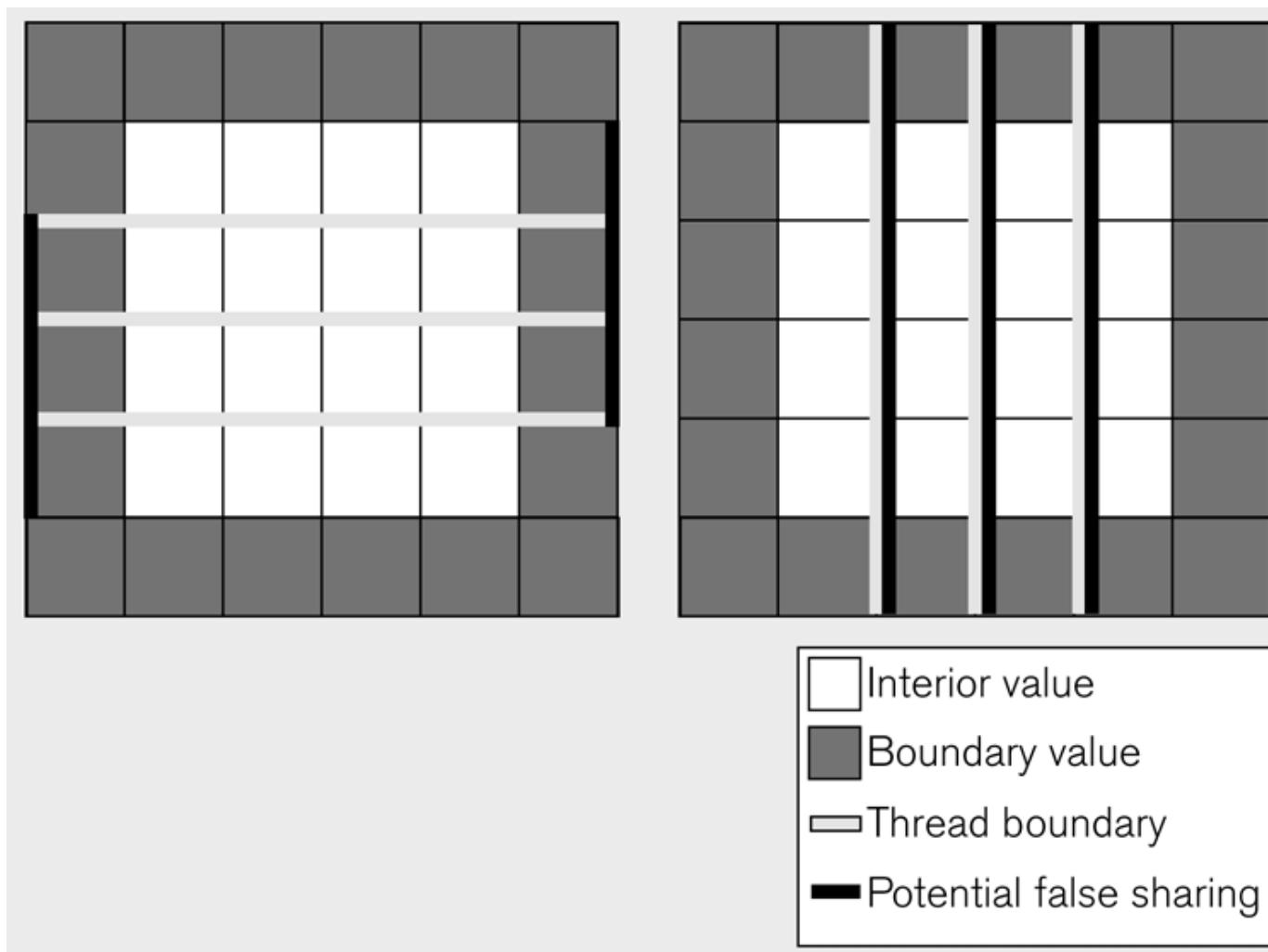


Example

1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0

□ Interior value
■ Boundary value

1-D Partitionings



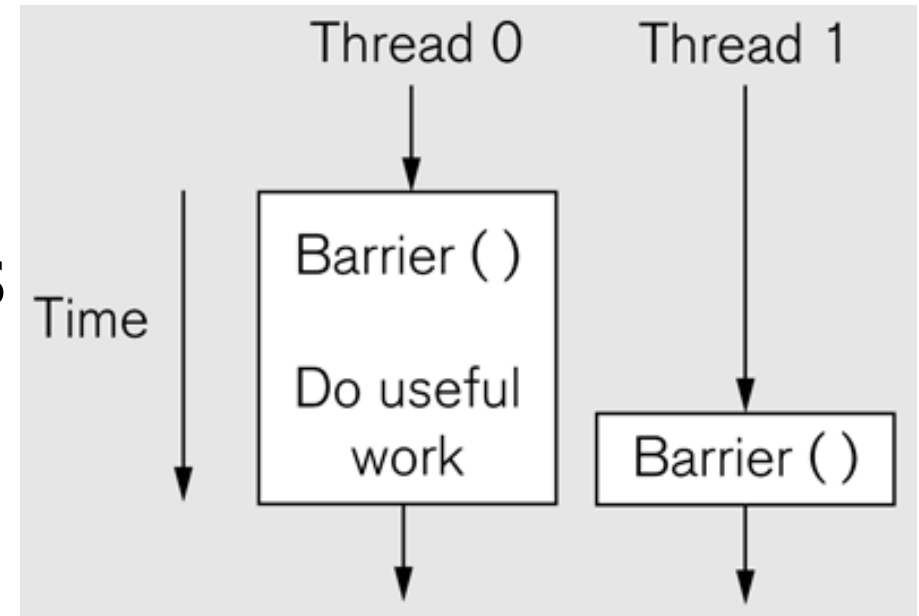


Barrier

```
124     for(i=1; i<n+2; i++)
125     {
126         for(j=0; j<n+2; j++)
127         {
128             val[i][j]=0.0;
129             new[i][j]=0.0;
130         }
131     }
132 }
133
134 void barrier()
135 {
136     pthread_mutex_lock(&barrier_lock);
137     count++;
138     if(count==t)
139     {
140         count=0;
141         pthread_cond_broadcast(&all_here);
142     }
143     else
144     {
145         pthread_cond_wait(&all_here, &barrier_lock);
146     }
147     pthread_mutex_unlock(&barrier_lock);
148 }
```

Decreasing Overhead

- Minimize blocking time:
Block only on what is necessary.
- Split-phase operations
 - initiation
 - completion
- Idea:
 - compute border points for next iteration
 - computer inner points





Split-phase Barrier

```
// Initiate synchronization  
barrier.arrived();  
  
// Do useful work  
  
// Complete synchronization  
barrier.wait();
```



Split-phase Barrier

```
17     int j=start;
18     myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
19     j=start+n_pre_thread -1;
20     myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
21
22     // Start barrier
23     barrier.arrived();
24
25     // Update local interior values
26     for(j=start+1; j<start+n_per_thread-1; j++)
27     {
28         myNew[j]=(myVal[j-1]+myVal[j+1])/2.0;
29     }
30     swap(myNew, myVal);
31
32     // Complete barrier
33     barrier.wait();
34 }
35 }
```



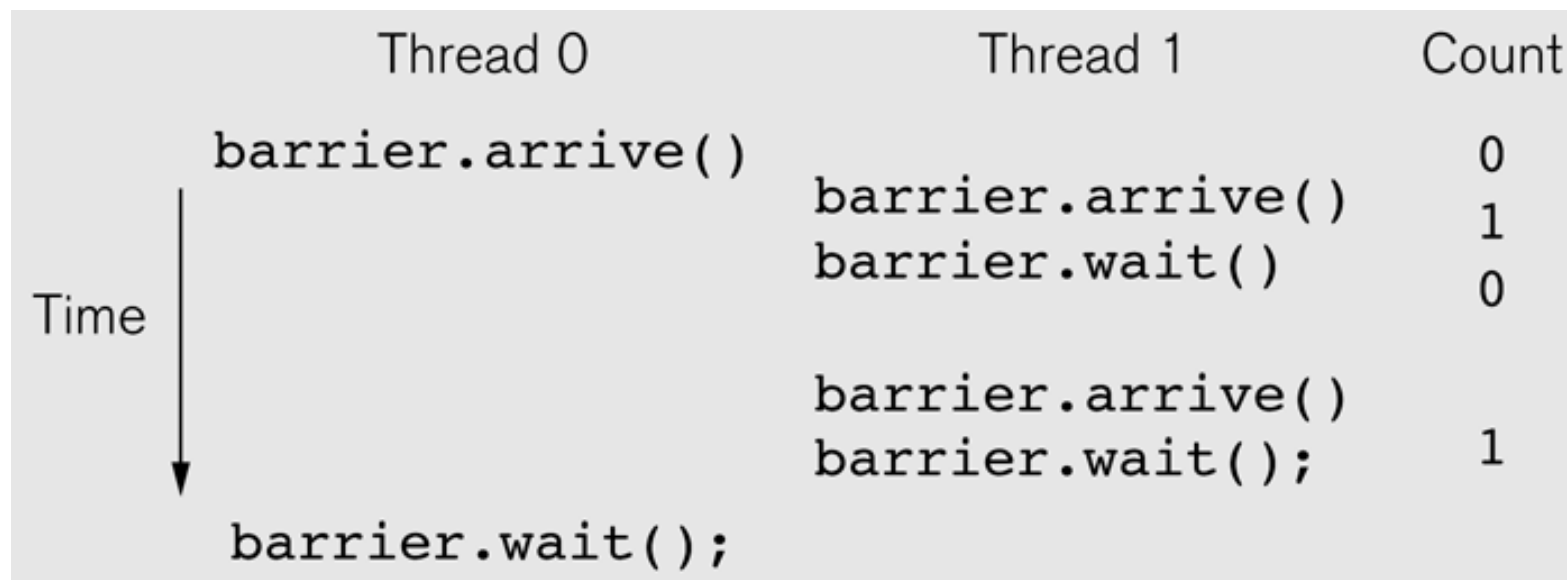
Wrong Implementation

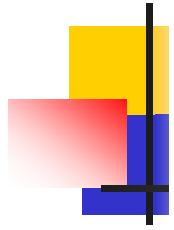
```
24
25
26 void Barrier::arrived(void)
27 {
28     pthread_mutex_lock(&lock);
29     count++ // Another thread has arrived
30
31     // If last thread to arrive, then wake up any waiters
32     if(count==nThreads)
33     {
34         count=0;
35         pthread_cond_broadcast(&all_here);
36     }
37
38     pthread_mutex_unlock(&lock);
39 }
40
41 void Barrier::wait(void)
42 {
43     pthread_mutex_lock(&lock);
44
45     // If not done, then wait
46     if(count !=0)
47     {
48         pthread_cond_wait(&all_here, &lock);
49     }
50
51     pthread_mutex_lock(&lock);
52 }
```

Race between different iterations



The Problem



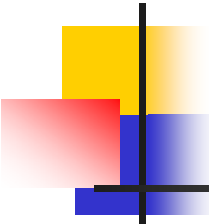


Correction

```
26
27 int Barrier::arrived(void)
28 {
29     int p;
30     pthread_mutex_lock(&lock);
31
32     p=phase;           // Get phase number
33     count++           // Another thread has arrived
34
35     // If last thread, then wake up any waiters, go to next phase
36     if(count==nThreads)
37     {
38         count=0;
39         pthread_cond_broadcast(&all_here);
40         phase=1 - phase;
41     }
42
43     pthread_mutex_unlock(&lock);
44     return p;
45 }
46
```

Correction

```
46
47 void Barrier::wait(int p)
48 {
49     pthread_mutex_lock(&lock);
50
51     // If not done, then wait
52     while(p==phase)
53     {
54         pthread_cond_wait(&all_here, &lock);
55     }
56
57     pthread_mutex_unlock(&lock);
58 }
```



```

2  int n;                // Number of interior values
3  int t;                // Number of threads
4  int iterations        // Number of iterations to perform
5
6  thread_main(int index)
7  {
8      int n_per_thread=n/t;
9      int start=index*n_per_thread;
10     int phase;
11
12     for(int i=0; i<iterations; i++)
13     {
14         // Update local boundary values
15         int j=start;
16         new[j]=(val[j-1]+val[j+1])/2.0;
17         j=start+n_per_thread -1;
18         new[j]=(val[j-1]+val[j+1])/2.0;
19
20         // Start barrier
21         phase=barrier.arrived();
22
23         // Update local interior values
24         for(j=start+1; j<start+n_per_thread-1; j++)
25         {
26             new[j]=(val[j-1]+val[j+1])/2.0;    // Compute average
27         }
28         swap(new, val);
29
30         // Complete barrier
31         barrier.wait(phase);
32     }
33 }

```



Fixed Problem

	Thread 0	Thread 1	Count	Phase
Time ↓	<code>barrier.arrive()</code>		0	0
		<code>barrier.arrive()</code>	1	
		<code>barrier.wait(0)</code>	0	1
		<code>barrier.arrive()</code>		
		<code>barrier.wait(1);</code>	1	1
	<code>barrier.wait(0);</code>			



Improvement

- Dependency is only between adjacent blocks.
 - Current solution is using a global barrier.
 - We can use semaphores between blocks.
 - signal & wait similarly to arrive & wait but to the neighboring threads



Java Threads

The Java Thread Class

```
public synchronized void start()
```

- Starts this Thread and returns immediately after invoking the run() method.
- Throws `IllegalThreadStateException` if the thread was already started.

start

```
public void run()
```

- The body of this Thread, which is invoked after the thread is started.

run

```
public final synchronized void join(long millis)
throws InterruptedException
```

- Waits for this Thread to die. A timeout in milliseconds can be specified, with a timeout of 0 milliseconds indicating that the thread will wait forever.

join

```
public static void yield()
```

- Causes the currently executing Thread object to yield the processor so that some other runnable Thread can be scheduled.

yield

```
public final int getPriority()
```

- Returns the thread's priority.

```
public final void setPriority(int newPriority)
```

- Sets the thread's priority.



Concepts

- Same concepts, different constructs
 - monitors – synchronized methods
 - threads
 - critical section – `synchronized(obj) { .. }`
 - *atomic objects, executor, concurrent collections (Java ≥5)*
- 2 ways to use threads
 - extend Thread
 - implement Runnable
- Semantics issues
 - use **volatile** for shared variables



Atomic Operations

Sample Operations for AtomicInteger

```
boolean compareAndSet(expectedValue, updateValue);
```

- Atomically sets the value to `updateValue` if the current value is the same as `expectedValue`.

```
int getAndIncrement();
```

- Atomically reads the current value and increments the current value by one.

- Implemented using special assembly instructions – more on that later.



Advanced: Futex

- Futex: Fast userspace locking system call.
 - Wait for a value at a given address to change.
 - Wake up anyone waiting on an address.
 - Low-level call usually used to implement locks.
 - Minix specific, available under Linux.



Avoiding Incorrect Code

- Avoid relying on thread inertia.
 - Threads are asynchronous.
 - Initialize data before starting threads.
 - Never assume that a thread will wait for you.
- Never bet on thread race.
 - Assume that at any point, any thread may go to sleep for any period of time.
 - No ordering exists between threads unless you cause ordering.



Avoiding Incorrect Code

- Scheduling is not the same as synchronization.
 - Never use sleep to synchronize.
 - Never try to “tune” with timing.
- Beware of deadlocks & priority inversion.
- One predicate \Leftrightarrow one condition variable.



Avoiding Performance Problems

- Beware of concurrent serialization.
- Use the right number of mutexes.
 - Too much mutex contention or too much locking without contention?
- Avoid false sharing.

- And... don't forget to compile like this:
`gcc -O3 -Wall -o hello hello.c -lpthread`