DCS216 Operating Systems

Lecture 13
Synchronization (2)
Mutex Locks, Semaphores

Apr 10th, 2024

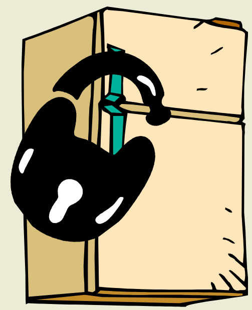
Instructor: Xiaoxi Zhang

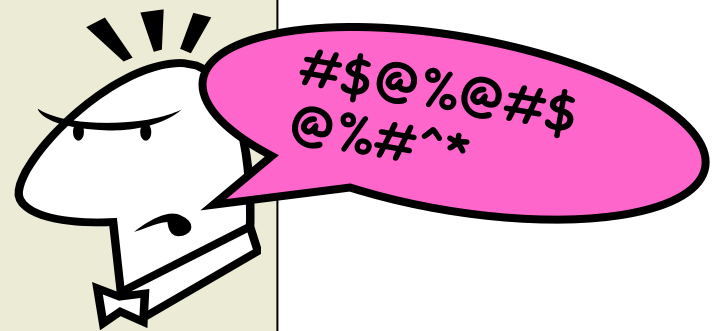
Sun Yat-sen University

Content

- **Content**
  - Mutex Locks
  - Hardware Support for Synchronization
    - Controlling Interrupts
    - Memory Barriers
    - `test_and_set()`
    - `compare_and_swap()` **CAS**
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores

Solution #4 – Mutex Lock






```
/* milk4.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int milk = 0;
pthread_mutex_t lock;

void *threadfun(void *arg) {
    pthread_mutex_lock(&lock);
    if (milk == 0)         // If no milk
        milk++;            // Buy milk
    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}
int main() {
    pthread_t A, B;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&A, NULL, threadfun, NULL);
    pthread_create(&B, NULL, threadfun, NULL);
    pthread_join(A, NULL);
    pthread_join(B, NULL);

    printf("Milk: %d\n", milk);
    return 0;
}
```

By using Mutex, the "Too Much Milk" problem becomes trivial.

The complexity is still there.
It's just hidden away from us in lock() and unlock().

Mutex Lock

- By now, you should have some basic ideas of how a lock works
  - ...from the perspective of a programmer
  - Even a five-year-old can use `pthread_mutex_lock()` and `pthread_mutex_unlock()` to solve Critical-Section Problem.
- But how to build a lock?
  - ...topic of an OS class
  - What hardware support is needed?
  - What OS (software) support is needed?

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

Mutex Lock

- A Mutex Lock (互斥锁) is a high-level software tool to solve the the Critical-Section Problem.
- The term "**Mutex**" is short for **Mut**ual **Ex**clusion.
- A Mutex Lock should ensure the following three properties:
  - **Mutual Exclusion:** At most one thread holds the lock.
  - **Progress:** If no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock.
  - **Bounded Waiting:** If thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does.

Mutex Lock

- **Mutex Lock APIs**
  - `void lock_init(lock_t *lock)` : **Initialize** a mutex lock
  - `void acquire(lock_t *lock)` : **Acquire** a mutex lock
  - `void release(lock_t *lock)` : **Release** a mutex lock

Mutex Lock Implementation #0

- Controlling Interrupts
  - One of the earliest solution used to provide mutual exclusion
  - **Simplicity:** You certainly don't have to scratch your head too hard to figure out why this works: without interruption, a thread can be sure that the code it is executing will not be interfered by other threads.

```
void acquire() {
    disable_interrupts();
}
void release() {
    enable_interrupts();
}
```

  - **Negatives:**
    - works **only** on single-processor systems, not on multi-processors.
    - requires kernel-level privilege (turning interrupts on and off is a privileged operation), typically only used in kernel code.
    - turning off interrupts for extended periods of time can lead to interrupts being lost, which can cause serious systems problems.

Mutex Lock Implementation #1

- A Simple Flag
  - to indicate whether some thread has held a lock.

```c
/* lock1.c */
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (lock->flag == 1)  // TEST flag
        ;                    // spin wait
    lock->flag = 1;          // now SET flag
}

void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

Mutex Lock Implementation #1

- A Simple Flag
  - to indicate whether some thre

```c
/* lock1.c */
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (lock->flag == 1)  // TEST flag
        ;                    // spin wait
    lock->flag = 1;          // now SET flag
}

void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

```c
/* lock1.c */
int max;
static volatile int count = 0;
lock_t lock;

void *threadfun(void *arg) {
    for (int i = 0; i < max; i++) {
        acquire(&lock);
        count++;
        release(&lock);
    }
    pthread_exit(0);
}
int main(int argc, char *argv[]) {
    max = atoi(argv[1]);

    lock_init(&lock);
    pthread_t p1, p2;
    pthread_create(&p1, NULL, threadfun, NULL);
    pthread_create(&p2, NULL, threadfun, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("max: %d, count: %d\n", max, count);
    return 0;
}
```

We'll be using this `main()` **template** to stress-test the our implementation of mutex lock.

Mutex Lock Implementation #1

- A Simple Flag
  - to indicate whether some thre

```c
/* lock1.c */
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (lock->flag == 1)  // TEST flag
        ;                    // spin wait
    lock->flag = 1;          // now SET flag
}

void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

```c
/* lock1.c */
int max;
static volatile int count = 0;
lock_t lock;

void *threadfun(void *arg) {
    for (int i = 0; i < max; i++) {
        acquire(&lock);
        count++;
        release(&lock);
    }
    pthread_exit(0);
}
int main(int argc, char *argv[]) {
    max = atoi(argv[1]);

    lock_init(&lock);
    pthread_t p1, p2;
    pthread_create(&p1, NULL, threadfun, NULL);
    pthread_create(&p2, NULL, threadfun, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("max: %d, count: %d\n", max, count);
    return 0;
}
```

```
$ ./lock1 100000
max: 100000, count: 133746
$ ./lock1 1000000
max: 1000000, count: 1486504
```

Mutex Lock Implementation #1

- A Simple Flag
  - to indicate whether some thread has held a lock.

```
/* lock1.c */
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (lock->flag == 1)  // TEST flag
        ;                    // spin wait
    lock->flag = 1;          // now SET flag
}

void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

`flag == 0` initially...

```
// Thread A                    // Thread B

// call acquire()
while (flag == 1) {
-----------------------------------------------
                               // call acquire()
                               while (flag == 1) {

                               }
                               flag = 1;
                               // Critical Section
-----------------------------------------------
}
flag = 1;
// Set flag (too!)
// Critical Section
```

Thread A got interrupted immediately right after the test `flag == 1`.

Mutex Lock Implementation #1.5

- Peterson's Algorithm
  - An improved **Flag-Based** pure software solution that actually works.

```
/* peterson_lock.c */
typedef struct __lock_t {
    int flag[2];
    int turn;
} lock_t;

void init(lock_t *lock) {
    lock->flag[0] = 0;
    lock->flag[1] = 0;
    lock->turn = 0;
}
void acquire(lock_t *lock, int tid) {
    lock->flag[tid] = 1;    // tid wants to acquire
    int other = 1 - tid;
    lock->turn = other;     // set turn to other
    while (lock->flag[other] && lock->turn == other)
        ;                   // busy wait for its turn
}
void release(lock_t *lock, int tid) {
    lock->flag[tid] = 0;    // tid unlock
}
```

Instruction **interleaving** won't affect the correctness of the algorithm (assuming atomic load and store).

Mutex Lock Implementation #1.5

- Peterson's Algorithm
  - Instruction interleaving won't affect the correctness of the algorithm (assuming atomic load and store).

```c
/* peterson_lock.c */
typedef struct __lock_t {
    int flag[2];
    int turn;
} lock_t;

void init(lock_t *lock) {
    lock->flag[0] = 0;
    lock->flag[1] = 0;
    lock->turn = 0;
}
void acquire(lock_t *lock, int tid) {
    lock->flag[tid] = 1;    // tid wants to acquire
    int other = 1 - tid;
    lock->turn = other;     // set turn to other
    while (lock->flag[other] && lock->turn == other)
        ;                   // busy wait for its turn
}
void release(lock_t *lock, int tid) {
    lock->flag[tid] = 0;    // tid unlock
}
```

```
$ ./peterson_lock 1000
max: 1000, count: 2000

$ ./peterson_lock 10000
max: 10000, count: 19995
Race condition at `count`.

$ ./peterson_lock 100000
max: 100000, count: 199996
Race condition at `count`.

$ ./peterson_lock 1000000
max: 1000000, count: 1999855
Race condition at `count`.
```

Mutex Lock Implementation #1.5

- Peterson's Algorithm
  - Unfortunately, on most systems, atomic loads and stores are not guaranteed by default.

```c
/* peterson_lock.c */
typedef struct __lock_t {
    int flag[2];
    int turn;
} lock_t;

void init(lock_t *lock) {
    lock->flag[0] = 0;
    lock->flag[1] = 0;
    lock->turn = 0;
}
void acquire(lock_t *lock, int tid) {
    lock->flag[tid] = 1;    // tid wants to acquire
    int other = 1 - tid;
    lock->turn = other;     // set turn to other
    while (lock->flag[other] && lock->turn == other)
        ;                   // busy wait for its turn
}
void release(lock_t *lock, int tid) {
    lock->flag[tid] = 0;    // tid unlock
}
```

```
$ ./peterson_lock 1000
max: 1000, count: 2000

$ ./peterson_lock 10000
max: 10000, count: 19995
Race condition at `count`.

$ ./peterson_lock 100000
max: 100000, count: 199996
Race condition at `count`.

$ ./peterson_lock 1000000
max: 1000000, count: 1999855
Race condition at `count`.
```

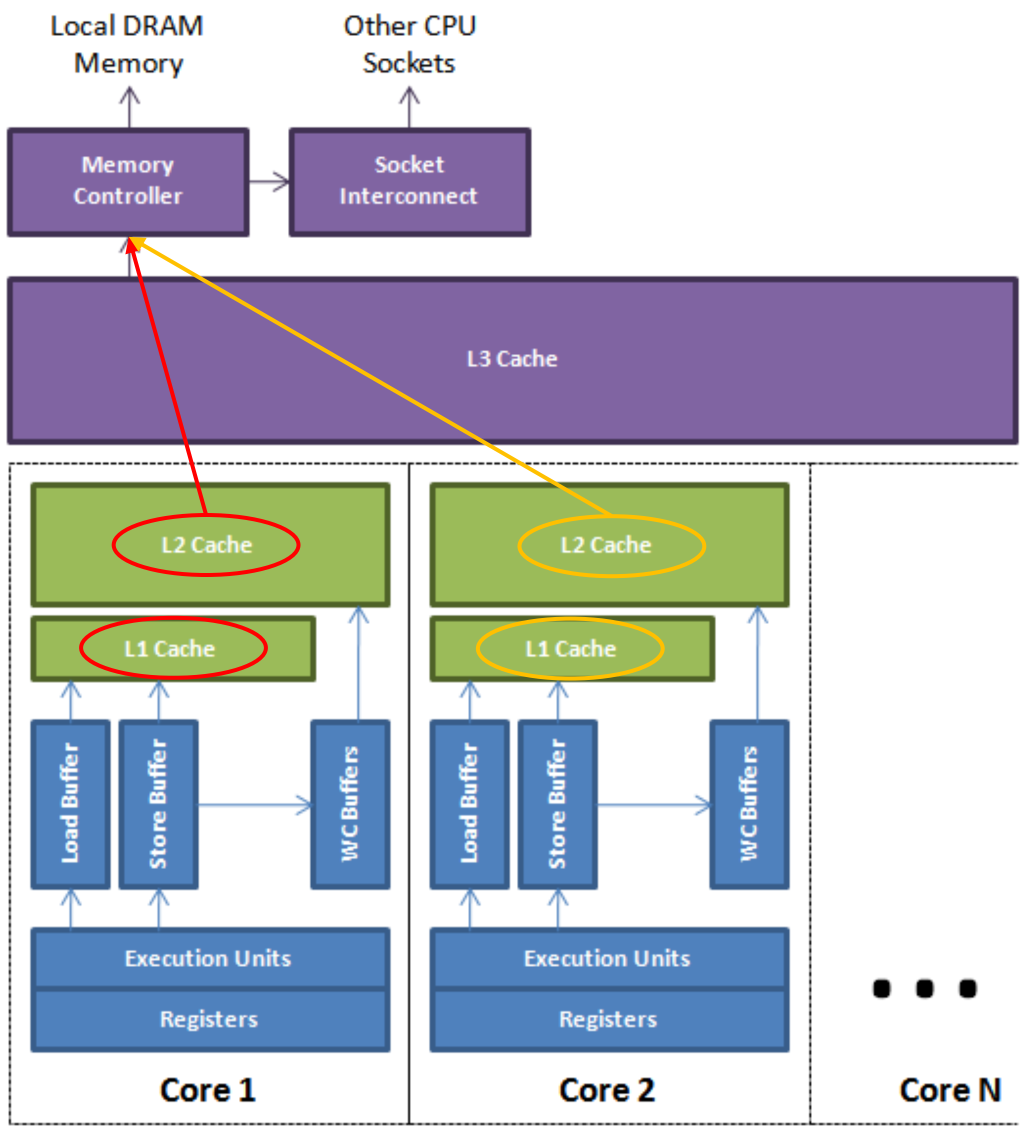
Our implementation **almost** worked! What happened???

Short answer(maybe): Weak memory model; cache inconsistency.

Hardware Support for Synchronization

- **Memory Barriers (内存屏障)**
  - Memory Model: what memory guarantees does a computer architecture provide for an application and how threads interact with memory
    - **Strongly ordered (强内存排序)**: a memory modification on one processor is immediately visible to all other processors
    - **Weakly ordered (弱内存排序)**: modifications to memory on one processor may not be immediately visible to other processors.
  - **Memory orders** vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor.
  - **Memory barriers** (or **memory fences**) are often used to force any change in memory to be propagated to all other processors. (think of it as a `flush()` operation)

Memory Barriers



Mutex Lock Implementation #1.5

- Peterson's Algorithm with atomic load and store
  - *<stdatomic.h>* supports `atomic_store()` and `atomic_load()`.

```c
/* peterson_lock_atomic.c */
typedef struct __lock_t {
    atomic_int flag[2];
    atomic_int turn;
} lock_t;
void init(lock_t *lock) {
    atomic_store(&lock->flag[0], 0);
    atomic_store(&lock->flag[1], 0);
    atomic_store(&lock->turn, 0);
}

void acquire(lock_t *lock, int tid) {
    atomic_store(&lock->flag[tid], 1);
    atomic_store(&lock->turn, 1 - tid);
    while (atomic_load(&lock->flag[1-tid]) &&
           atomic_load(&lock->turn) == 1-tid)
        ;
}
void release(lock_t *lock, int tid) {
    atomic_store(&lock->flag[tid], 0);
}
```

```
$ ./peterson_lock_atomic 1000
max: 1000, count: 2000

$ ./peterson_lock_atomic 10000
max: 10000, count: 20000

$ ./peterson_lock_atomic 100000
max: 100000, count: 200000

$ ./peterson_lock_atomic 1000000
max: 1000000, count: 2000000
```

Memory fences are put in place around every `atomic_store()` and `atomic_load()` operation, ensuring memory consistency among threads running on different cores.

Mutex Lock Implementation #1

- A Simple Flag
  - to indicate whether some thread has held a lock.

```c
/* lock1.c */
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while (lock->flag == 1)  // TEST flag
        ;                    // spin wait
    lock->flag = 1;          // now SET flag
}

void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

`flag == 0` initially...

```c
// Thread A                      // Thread B

// call acquire()
while (flag == 1) {
                                 // call acquire()
                                 while (flag == 1) {

                                 }
                                 flag = 1;
                                 // Critical Section
}
flag = 1;
// Set flag (too!)
// Critical Section
```

Wouldn't it be nice if
- TEST (flag == 1)
- SET (flag = 1)

is uninterruptible/atomic?

Mutex Lock Implementation #2

- Test-And-Set
  - It returns the old value pointed to by `target`
  - and simultaneously updates `*target` to true
  - The `test_and_set` instruction is atomic and uninterruptible, i.e., it is usually performed within a single CPU cycle.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

Mutex Lock Implementation #2

- Test-And-Set
  - Can be used to implement busy wait mutex locks, also called **spin locks.**

```c
/* lock2.c */

typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    // Atomic Test and Set
    while (test_and_set(&lock->flag))
        ;
}
void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

Mutex Lock Implementation #2

- Test-And-Set
  - Since C11, *<stdatomic.h>* provides support for test_and_set().

```
/* lock2.c */
#include <stdatomic.h>
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    // Atomic Test and Set
    while (atomic_flag_test_and_set(&lock->flag))
        ;
}
void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

Mutex Lock Implementation #2

- Test-And-Set
  - Since C11, *<stdatomic.h>* provides support for test_and_set().

```c
/* lock2.c */
#include <stdatomic.h>
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    // Atomic Test and Set
    while (atomic_flag_test_and_set(&lock->flag))
        ;
}
void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

```
$ ./lock2 100000
max: 100000, count: 200000

$ ./lock2 1000000
max: 1000000, count: 2000000

$ ./lock2 10000000
max: 10000000, count: 20000000
```

Mutex Lock Implementation #3

- Compare-And-Swap (**CAS**)
  - like Test-And-Set, but operates on two variables atomically, but uses a different mechanism based on **swapping** the content of two variables

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

Mutex Lock Implementation #3

- Compare-And-Swap (**CAS**)
  - like Test-And-Set, but operates on two variables atomically, but uses a different mechanism based on **swapping** the content of two variables
  - `test_and_set(&lock)`
    is equivalent to
    `compare_and_swap(&lock, 0, 1)`

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic test_and_set() instruction.

Figure 6.7 The definition of the atomic compare_and_swap() instruction.

Mutex Lock Implementation #3

- Compare-And-Swap (**CAS**)
  - like Test-And-Set, but operates on two variables atomically, but uses a different mechanism based on **swapping** the content of two variables
  - `test_and_set(&lock)`
    is equivalent to
    `compare_and_swap(&lock, 0, 1)`

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

Mutex Lock Implementation #3

- Compare-And-Swap (**CAS**)
  - Since C11, *<stdatomic.h>* provides support for compare_and_swap().

```c
/* lock3.c */
#include <stdatomic.h>
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    int expected = 0;
    while (!atomic_compare_exchange_strong(&lock->flag, &expected, 1))
        expected = 0;
}
void release(lock_t *lock) {
    lock->flag = 0;         // RESET flag
}
```

```c
bool atomic_compare_exchange(int *value,
                             int *expected,
                             int new_value) {
    if (*value == *expected) {
        *value = new_value;
        return true;
    } else {
        *expected = *value;
        return false;
    }
}
```

真 Compare-And-**Swap**()

Mutex Lock Implementation #3

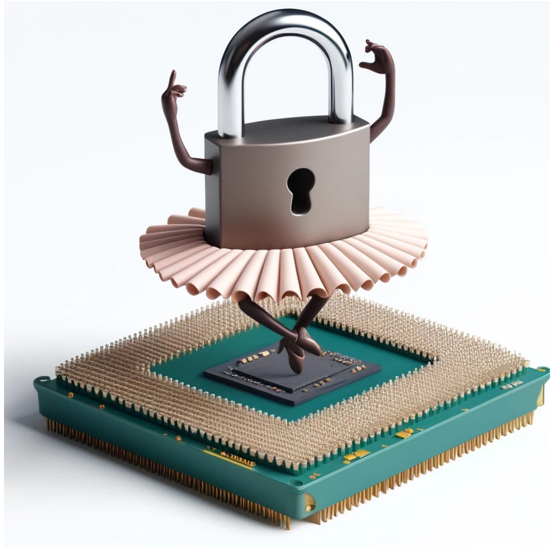
- Compare-And-Swap (**CAS**)
  - Since C11, *<stdatomic.h>* provides support for compare_and_swap().

```c
/* lock3.c */
#include <stdatomic.h>
typedef struct __lock_t {
    int flag;
} lock_t;

void lock_init(lock_t *lock) {
    /* 0 -> lock available (UNLOCKED)
     * 1 -> lock held      (LOCKED)
     */
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    int expected = 0;
    while (!atomic_compare_exchange_strong(&lock->flag, &expected, 1))
        expected = 0;
}
void release(lock_t *lock) {
    lock->flag = 0;          // RESET flag
}
```

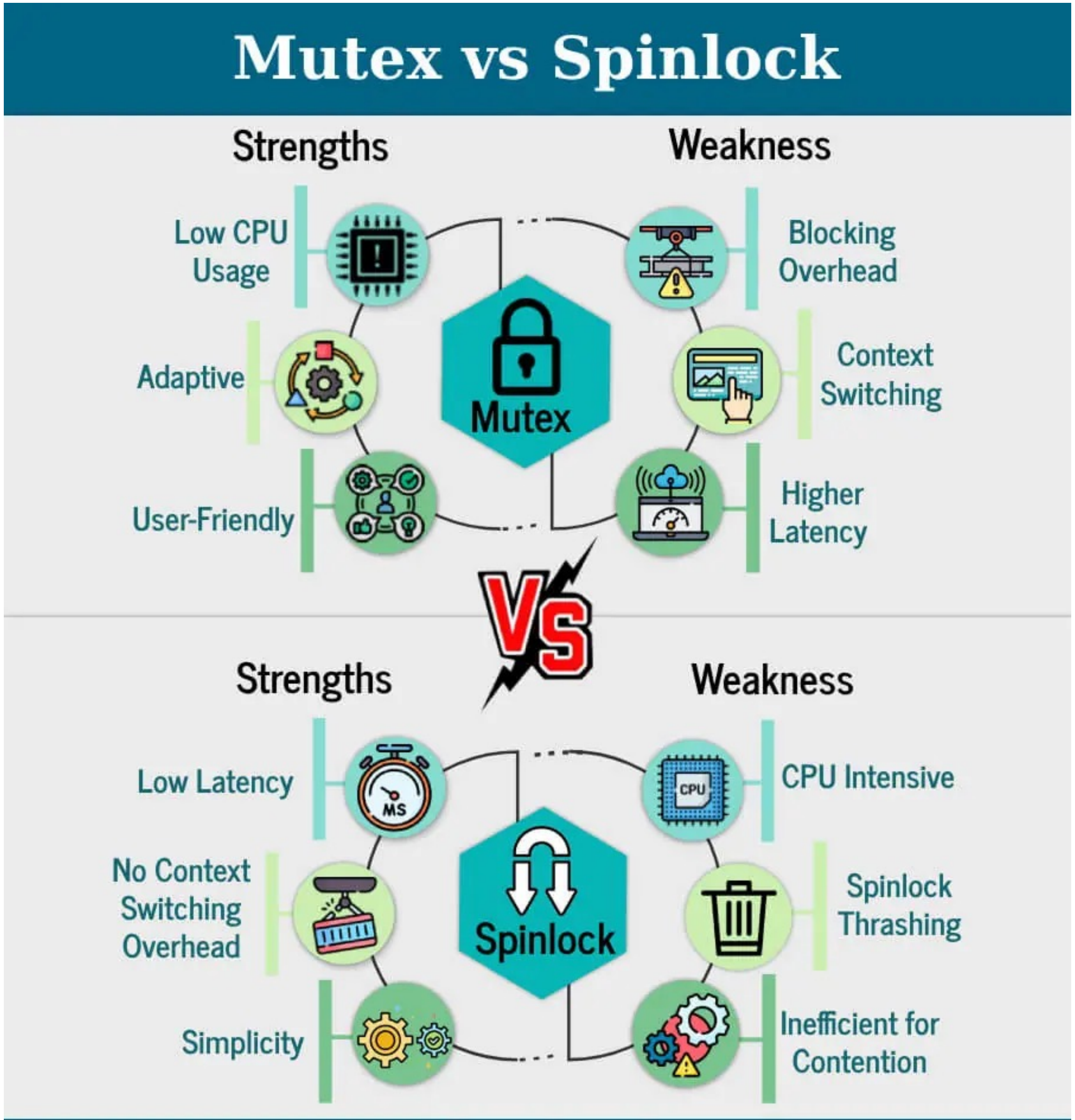
```
$ ./lock3 100000
max: 100000, count: 200000

$ ./lock3 1000000
max: 1000000, count: 2000000

$ ./lock3 10000000
max: 10000000, count: 20000000
```

Spinlock

- The implementation using `` `test_and_set()` `` and `` `compare_and_swap()` `` has a main **disadvantage**: busy waiting.
- The type of mutex lock that perform busy waiting is also called a **spinlock(自旋锁)**, because the thread "spins" while waiting for the lock to become available.
- Spinlocks do have **advantages**, however, in that no **context switch** is required when a thread is waiting for a lock. In certain circumstances, spinlocks are the preferable choice for locking if the critical section is short.



Mutex vs. Spinlock



Semaphores

- Semaphores (信号量) are a kind of **generalized** lock
  - First introduced by **Edsger Dijkstra** in 1960s
  - The main synchronization primitive used in original UNIX
- **Definition:** a Semaphore S is an integer variable that (apart from initialization) is accessed only through two **atomic** operations:
  - `Wait()` / `Down()`/ `P()`: an **atomic** operation that **decrements** semaphore by 1, and waits if value of semaphore is negative
  - `Signal()`/ `Up()` / `V()`: an **atomic** operation that **increments** the semaphore by 1, if there are threads waiting, wake up one of them
  - `Wait()` was originally termed P (from the Dutch ***proberen***, "to test");
  - `Signal()` was originally called V (from Dutch ***verhogen***, "to increment")

Semaphores

- Semaphores (信号量) are a kind of **generalized** lock
  - First introduced by **Edsger Dijkstra** in 1960s
  - The main synchronization primitive used in original UNIX
- **Definition:** a Semaphore S is an integer variable that (apart from initialization) is accessed only through two **atomic** operations:
  - `Wait()` / `Down()`/ `P()`: an **atomic** operation that **decrements** semaphore by 1, and waits if value of semaphore is negative
  - `Signal()`/ `Up()` / `V()`: an **atomic** operation that **increments** the semaphore by 1, if there are threads waiting, wake up one of them

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphores Usage

- **Binary** Semaphore (二进制信号量): **Mutual Exclusion**
  - S value can only be 0 or 1
  - can be used for mutual exclusion (init value = 1), just like a mutex lock

```
P(&S);
// Critical Section
V(&S);
```

- **Counting** Semaphore (计数信号量): **Resource Constraints**
  - S value can be any integer
  - can be used to control access to a given resource consisting of a finite number of instances.
  - The semaphore is initialized to the number of resources available.
  - Each thread that wishes to use a resource performs a `wait()/P()`
  - When a thread releases a resource, it performs a `signal()/V()`
  - When the count for the semaphore goes to `0`, all resources are used.
  - After that, threads that wish to use a resource will block until the count becomes greater than `0`

- **POSIX Semaphores API**
  - Semaphore Type:

    `struct sem_t`
  - Initialization:

    `int sem_init(sem_t *sem, int pshared, unsigned int value);`
    - `pshared == 0: indicates that sem is to be shared between threads of the calling process`
    - `pshared == 1: indicates that sem is to be shared between processes.`
  - `Wait() / Down() / P():`

    `int sem_wait(sem_t *sem);`
  - `Signal() / Up() / V():`

    `int sem_post(sem_t *sem);`

POSIX Semaphores Example

- Binary Semaphore as Mutex Lock

```c
/* sem_mutex.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int max;
static volatile int count = 0;
sem_t mutex;

void* threadfun(void* arg) {
    for (int i = 0; i < max; i++) {
        sem_wait(&mutex);
        // Critical Section Begin
        count++;
        // Critical Section End
        sem_post(&mutex);
    }
    pthread_exit(0);
}
```

```c
int main(int argc, char *argv[]) {
    max = atoi(argv[1]);

    // Binary semaphore (mutex)
    sem_init(&mutex, 0, 1);

    pthread_t p1, p2;
    pthread_create(&p1, NULL, threadfun, NULL);
    pthread_create(&p2, NULL, threadfun, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("max: %d, count: %d\n", max, count);
    if (count != max * 2) {
        printf("Race condition at `count`.\n");
    }

    // Destroy the semaphore
    sem_destroy(&mutex);

    return 0;
}
```

POSIX Semaphores Example

- Binary Semaphore vs Pthread Mutex Lock

```c
/* count_mutex.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int max;
static volatile int count = 0;
pthread_mutex_t mutex;

void* threadfun(void* arg) {
    for (int i = 0; i < max; i++) {
        pthread_mutex_lock(&mutex);
        // Critical Section Begin
        count++;
        // Critical Section End
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

```c
int main(int argc, char *argv[]) {
    max = atoi(argv[1]);

    // Initialize mutex lock
    pthread_mutex_init(&mutex, NULL);

    pthread_t p1, p2;
    pthread_create(&p1, NULL, threadfun, NULL);
    pthread_create(&p2, NULL, threadfun, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("max: %d, count: %d\n", max, count);
    if (count != max * 2) {
        printf("Race condition at `count`.\n");
    }

    // Destroy the semaphore
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

POSIX Semaphores Example

- Binary Semaphore vs Pthread Mutex Lock

```c
/* sem_mutex.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int max;
static volatile int count = 0;
sem_t mutex;

// Binary semaphore (mutex)
sem_init(&mutex, 0, 1);

void* threadfun(void* arg) {
    for (int i = 0; i < max; i++) {
        sem_wait(&mutex);
        // Critical Section Begin
        count++;
        // Critical Section End
        sem_post(&mutex);
    }
    pthread_exit(0);
}
```

```c
/* count_mutex.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

int max;
static volatile int count = 0;
pthread_mutex_t mutex;

// Initialize mutex lock
pthread_mutex_init(&mutex, NULL);

void* threadfun(void* arg) {
    for (int i = 0; i < max; i++) {
        pthread_mutex_lock(&mutex);
        // Critical Section Begin
        count++;
        // Critical Section End
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}
```

Recall: Bounded Buffer with Busy Waiting

Producer-Consumer Problem with Shared Memory

- Producer vs. Consumer

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

- The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`.
  - The variable `in` points to the next free position in the buffer;
  - The variable `out` points to the first full position in the buffer
  - The buffer is empty when `in == out`
  - The buffer is full when `((in + 1) % BUFFER_SIZE) == out`
  - This scheme allows at most `BUFFER_SIZE-1` items in the buffer at the same time. **WHY**?

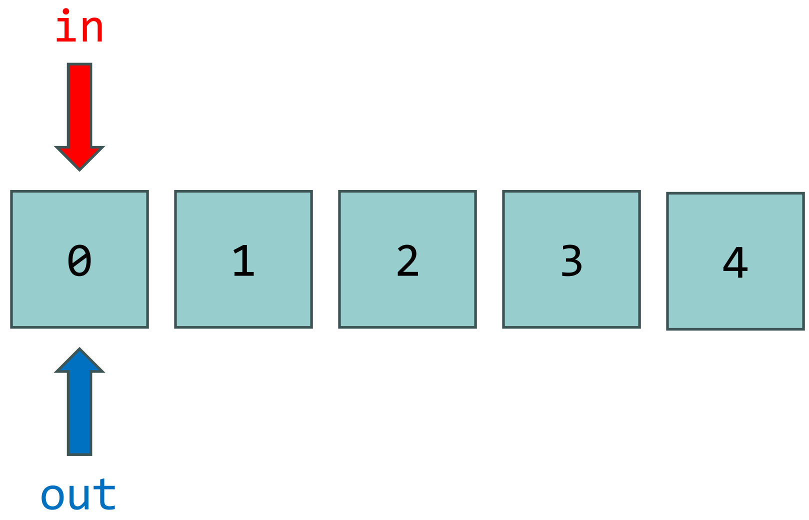
Recall: Bounded Buffer with Busy Waiting

Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 0



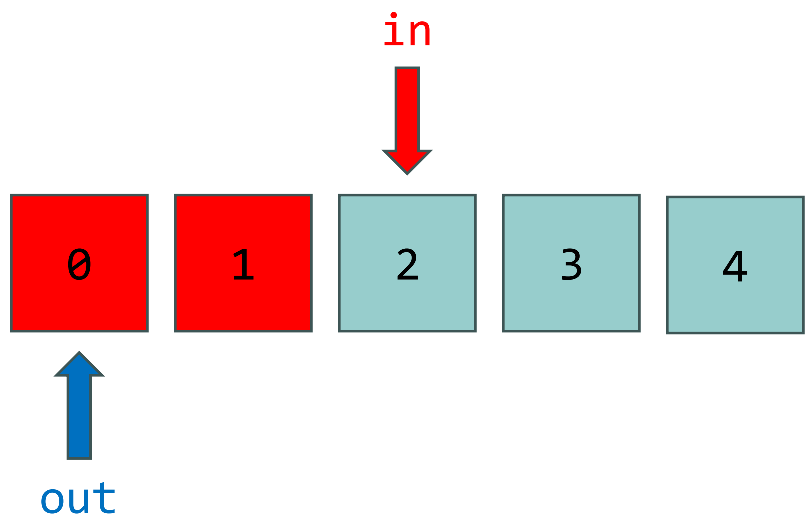
Recall: Bounded Buffer with Busy Waiting

Producer-Consumer Problem with Shared Memory

```
item next_produced;
while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out)
        ;       /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item next_consumed;
while (true) {
    while (in == out)
        ;       /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next_command */
}
```

# of items: 2



Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

Critical Section

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Consumer blocks when buffer is empty, i.e., there're no full (occupied) slots

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

Producer adds item to buffer, signaling non-empty buffer

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Consumer blocks when buffer is empty, i.e., there're no full (occupied) slots

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

Producer adds item to buffer, signaling non-empty buffer

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

- **Bounded Buffer with Semaphores**
  - **Counting** Semaphore
    - semaphore full = 0;   // consumer's constraint
    - semaphore empty = BUFFER_SIZE; // producer's constraint
    - semaphore mutex = 1;   // mutual exclusion
  - Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/
    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

Producer can't proceed if there're no empty slots

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/
    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

Producer can't proceed if there're no empty slots

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Consumer signals new empty slots

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

Consumer signals new empty slots

Bounded Buffer with Semaphores

- **Counting** Semaphore
  - `semaphore full  = 0;           // consumer's constraint`
  - `semaphore empty = BUFFER_SIZE; // producer's constraint`
  - `semaphore mutex = 1;           // mutual exclusion`
- Bounded-Buffer Producer-Consumer
  - No busy waiting caused by while loops...

```
// Producer
while (true) {
    /* produce an item*/

    wait(empty);
    wait(mutex);

    /* add to buffer */

    signal(mutex);
    signal(full);
}
```

```
// Consumer
while (true) {
    wait(full);
    wait(mutex);

    /* remove from buffer */

    signal(mutex);
    signal(empty);

    /* consume item */
}
```

```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```c
/* bounded_buffer_sem.c */
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;
sem_t full;
sem_t mutex;

void printBuffer() {...};
int main() {
    pthread_t prod, cons;

    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE-1);
    sem_init(&full, 0, 0);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

Semaphores


```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

Semaphores


```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

Semaphores

```
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

Semaphores


```c
void* producer(void* arg) {
    int next_produced;
    for (int i = 0; i < 10; i++) {
        next_produced = i;
        sem_wait(&empty); // Decrement empty count
        sem_wait(&mutex);
        // Add the item to the buffer
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        printf("--> Produced %d. ", next_produced);
        printBuffer();

        sem_post(&mutex);
        sem_post(&full);
    }
}
```

```c
void* consumer(void* arg) {
    int next_consumed;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full); // Decrement full count
        sem_wait(&mutex);
        // Remove an item from the buffer
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        printf("<-- Consumed %d. ", next_consumed);
        printBuffer();

        sem_post(&mutex);
        sem_post(&empty);
        sleep(rand() % 5);
    }
}
```

```
$ make bounded_buffer_sem
gcc -g -Wall -pthread -m32 -o bounded_buffer_sem
bounded_buffer_sem.c
$ ./bounded_buffer_sem
--> Produced 0. in: 1, out: 0. Buffer: [ 0 _ _ _ _ ]
--> Produced 1. in: 2, out: 0. Buffer: [ 0 1 _ _ _ ]
--> Produced 2. in: 3, out: 0. Buffer: [ 0 1 2 _ _ ]
--> Produced 3. in: 4, out: 0. Buffer: [ 0 1 2 3 _ ]
<-- Consumed 0. in: 4, out: 1. Buffer: [ _ 1 2 3 _ ]
--> Produced 4. in: 0, out: 1. Buffer: [ _ 1 2 3 4 ]
<-- Consumed 1. in: 0, out: 2. Buffer: [ _ _ 2 3 4 ]
--> Produced 5. in: 1, out: 2. Buffer: [ 5 _ 2 3 4 ]
<-- Consumed 2. in: 1, out: 3. Buffer: [ 5 _ _ 3 4 ]
--> Produced 6. in: 2, out: 3. Buffer: [ 5 6 _ 3 4 ]
<-- Consumed 3. in: 2, out: 4. Buffer: [ 5 6 _ _ 4 ]
--> Produced 7. in: 3, out: 4. Buffer: [ 5 6 7 _ 4 ]
<-- Consumed 4. in: 3, out: 0. Buffer: [ 5 6 7 _ _ ]
--> Produced 8. in: 4, out: 0. Buffer: [ 5 6 7 8 _ ]
<-- Consumed 5. in: 4, out: 1. Buffer: [ _ 6 7 8 _ ]
<-- Consumed 6. in: 4, out: 2. Buffer: [ _ _ 7 8 _ ]
--> Produced 9. in: 0, out: 2. Buffer: [ _ _ 7 8 9 ]
<-- Consumed 7. in: 0, out: 3. Buffer: [ _ _ _ 8 9 ]
<-- Consumed 8. in: 0, out: 4. Buffer: [ _ _ _ _ 9 ]
<-- Consumed 9. in: 0, out: 0. Buffer: [ _ _ _ _ _ ]
```

Summary

Synchronization	Pseudocode Convention	POSIX API
Mutex Lock	`lock_init()`	`pthread_mutex_init()`
	`acquire()`	`pthread_mutex_lock()`
	`release()`	`pthread_mutex_unlock()`
Semaphore	`wait() / Down() / P()`	`sem_wait()`
	`signal() / Up() / V()`	`sem_post()`

Thank you!