Chapter 6

Synchronization Tools

Yunmin Go

School of CSEE

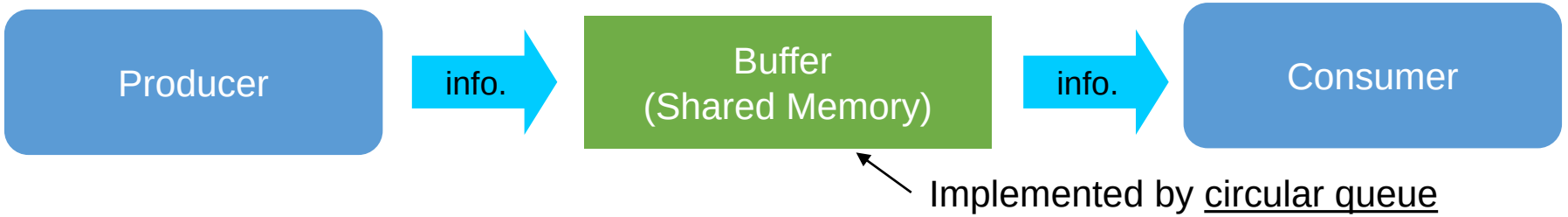
HANDONG GLOBAL UNIVERSITY

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

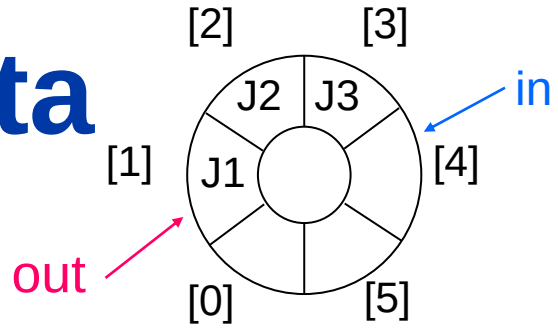
The Producer/Consumer Problem

- Producer
  - Produce data items
  - Wish to place data items in a buffer
- Consumer
  - Grab data items out of the buffer consume them in some way



- Example: Multi-threaded web server
  - A producer puts HTTP requests into a work queue
  - Consumer threads take requests out of this queue and process them

Concurrent Access of Shared Data



- Producer

```
while (true) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFERSIZE;
    counter++;
}
```

- Consumer

```
while (true) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

- Normal situation
  - Initial value of counter is 5
  - Producer increased counter, and consumer decreased counter concurrently.
  - Ideally, counter should be 5. But…

Synchronization Problem

- Implementation of "counter++"

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- Implementation of "counter--"

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Problematic situation: interleaved execution

Producer		Consumer	
$register_1 = counter$	$(register_1 = 5)$		
$register_1 = register_1 + 1$	$(register_1 = 6)$		
		$register_2 = counter$	$(register_2 = 5)$
		$register_2 = register_2 - 1$	$(register_2 = 4)$
$counter = register_1$	$(counter = 6)$		
		$counter = register_2$	$(counter = 4)$

counter can be 4 or 6 !

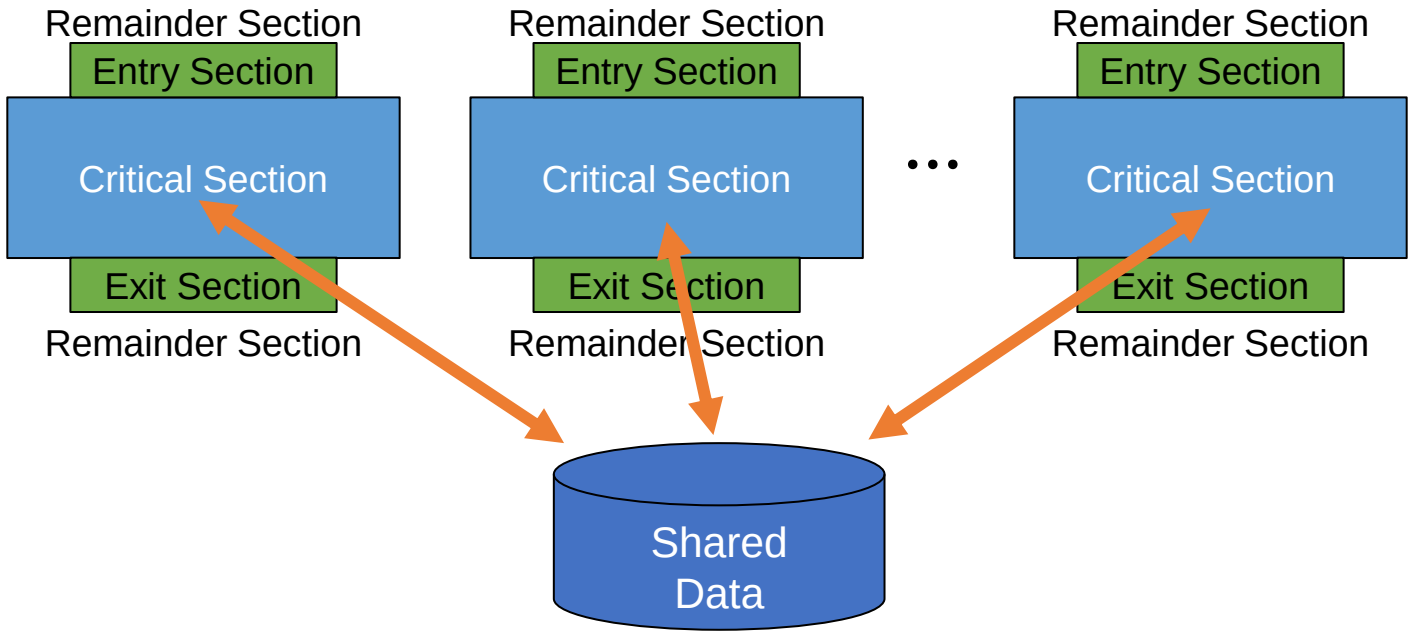
Synchronization Problem

- **Race condition**
  - A situation, where several processes access and manipulate the same data concurrently
  - The outcome of execution depends on the particular order in which the access takes place

- **Synchronization**
  - The coordination of occurrences to operate in unison with respect to time
  - Ensuring only one process can access the shared data at a time

Critical-Section

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section



Critical-Section

- **Critical section problem** is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
// General structure of process Pi
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

- Entry section: code section to request permission to enter critical section
- Critical section: a segment of code which may change shared resources (common variables, file, …)
- Exit section: code section to notice it leaves the critical section
- Remainder section: a segment of code which doesn't change shared resources

Example: Critical Section

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

#define NUM_THREAD    10
void *thread_inc(void * arg);
void *thread_des(void * arg);
long long num = 0;

int main(int argc, char *argv[])
{
    pthread_t thread_id[NUM_THREAD];
    int i;

    printf("sizeof long long: %ld \n", sizeof(long long));
    for (i = 0; i < NUM_THREAD; i++) {
        if (i % 2)
            pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
        else
            pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
    }

    for (i = 0; i < NUM_THREAD; i++)
        pthread_join(thread_id[i], NULL);

    printf("result: %lld \n", num);
    return 0;
}
```

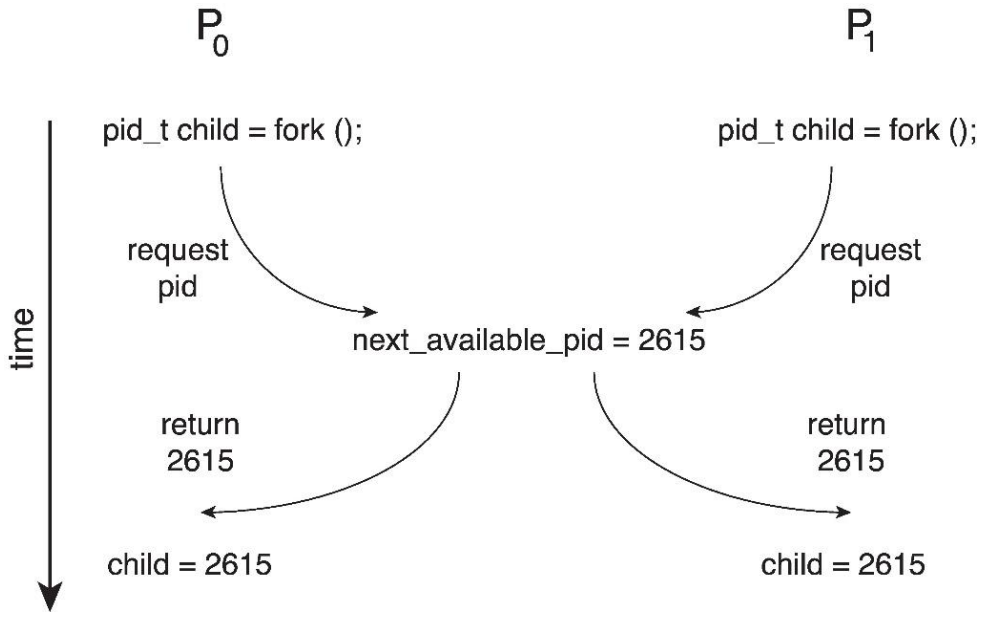
```
void *thread_inc(void * arg) {
    int i;
    for (i = 0; i < 100000; i++)
        num += 1;
    return NULL;
}

void *thread_des(void * arg) {
    int i;
    for (i = 0; i < 100000; i++)
        num -= 1;
    return NULL;
}
```

Result?

Critical-Section Handling in OS

- Example of race condition when assigning a pid
  - Processes P_0 and P_1 are creating child processes using the `fork()` system call
  - Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)
  - Unless there is mutual exclusion, the same pid could be assigned to two different processes!



Critical-Section Handling in OS

- Two general approaches are used to handle critical-section in operating systems
  - Non-preemptive
    - Not allow a process running in kernel mode to be preempted
    - Kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
  - Preemptive kernels
    - Allows preemption of process when running in kernel mode
    - More responsive, since there is less risk a kernel mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes
    - More suitable for real-time process

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

Locks: The Basic Idea

- Ensure that any critical section executes as if it were a single atomic instruction.
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

  - Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

- Lock variable (e.g., `mutex`) holds the state of the lock
  - **Available** (or **unlocked** or **free**): No thread holds the lock
  - **Acquired** (or **locked** or **held**): Exactly one thread holds the lock and presumably is in a critical section

Locks: The Basic Idea

```
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

- `lock()`
  - Try to acquire the lock.
  - If no other thread holds the lock, the thread will acquire the lock.
  - Enter the critical section
    - This thread is said to be the owner of the lock.
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there.
- `unlock()`
  - Once the owner of the lock calls this function, the lock is now available (free) again.

Pthread Locks: Mutex

- Creating and initializing the lock

```
#include <pthread.h>
pthread_mutex_t mutex;                  // global declaration

// create and initialize the mutext lock
pthread_mutex_init(&mutex, NULL);    // call before first lock
or
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Acquiring and releasing the lock

```
pthread_mutex_lock(&mutex);     // acquire the mutex lock

/* critical section */
balance = balance + 1;

pthread_mutex_unlock(&mutex);  // release the mutex lock
```

https://man7.org/linux/man-pages/man3/pthread_mutex_init.3p.html
https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html

Example: Pthread Locks

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

#define NUM_THREAD    10
void *thread_inc(void * arg);
void *thread_des(void * arg);
long long num = 0;
pthread_mutex_t mutex;

int main(int argc, char *argv[]) {
    pthread_t thread_id[NUM_THREAD];
    int i;

    pthread_mutex_init(&mutex, NULL);

    for (i = 0; i < NUM_THREAD; i++) {
        if (i % 2)
            pthread_create(&(thread_id[i]), NULL, thread_inc, NULL);
        else
            pthread_create(&(thread_id[i]), NULL, thread_des, NULL);
    }

    for(i = 0; i < NUM_THREAD; i++)
        pthread_join(thread_id[i], NULL);

    printf("result: %lld \n", num);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

```c
void *thread_inc(void * arg) {
    int i;
    for (i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);
        num += 1;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

void *thread_des(void * arg) {
    int i;
    for (i = 0; i < 100000; i++) {
        pthread_mutex_lock(&mutex);
        num -= 1;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Result?

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

Requirements for Lock

- **Mutual Exclusion**
  - Does the lock work, preventing multiple threads from entering a critical section?
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free?
    - Starvation
    - Progress and Bounded Waiting
- **Performance**
  - The time overheads added by using the lock.

From Operating System Concepts

Mutual Exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Controlling Interrupts

- Disable interrupts for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for **single-processor** systems

```
void lock() {
    DisableInterrupts();
}

void unlock() {
    EnableInterrupts();
}
```

  - Problem:
    - Require too much trust in applications
      - Greedy (or malicious) program could monopolize the processor.
    - Do not work on **multiprocessors**
    - Code that masks or unmasks interrupts be executed slowly by modern CPUs.

Using a Flag

- Using a flag denoting whether the lock is held or not.
  - The code below has problems.

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Using a Flag

- Problem 1: **No Mutual Exclusion** (assume `flag=0` to begin)

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1; // set flag to 1 (too!)`	

- Problem 2: **Spin-waiting** wastes time waiting for another thread.

- So, we need an atomic instruction supported by **Hardware**!
  - *Test-and-Set* instruction, also known as *atomic exchange*

Test-And-Set

- An instruction to support the creation of simple locks

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;      // fetch old value at ptr
    *ptr = new;          // store ‘new’ into ptr
    return old;          // return the old value
}
```

  - Return(testing) `old` value pointed by the `ptr`.
  - Set the value of passed parameter to `new`.
  - This sequence of operations is performed atomically.

Test-And-Set

- A simple spin lock using Test-And-Set

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;       // spin-wait
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Note: To work correctly on a single processor, it requires a preemptive scheduler

Evaluating Spin Locks

- **Correctness**: yes
  - The spin lock only allows a single thread to entry the critical section.
- **Fairness**: no
  - Spin locks don't provide any fairness guarantees.
  - Indeed, a thread spinning may spin *forever*.
- **Performance**:
  - In the single CPU, performance overheads can be quite *painful*.
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Compare-And-Swap

- Test whether the value at the address(`ptr`) is equal to expected.
  - If so, update the memory location pointed to by `ptr` with the `new` value.
  - In either case, return the actual value at that memory location.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
 }
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
 }
```

  - `cmpxchg` (compare-and-exchange) instruction on x86 or x64

Load-Linked and Store-Conditional

- Load-Linked (LL) and Store-Conditional (SC) in MIPS
  - Load-Linked: `ll rt, offset (rs)`
  - Store-Conditional: `sc rt, offset (rs)`

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

    - `Store-Conditional()` only succeeds if no intermittent store to the address has taken place.
      - Success: return 1 and update the value at `ptr` to `value`.
      - Fail: the value at `ptr` is not updates and 0 is returned.

Load-Linked and Store-Conditional

- Using LL/SC to build a lock

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ;         // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return;  // if set-it-to-1 was a success: all done
                       otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- A more concise form of the `lock()` using LL/SC

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
        ;     // spin
 }
```

Fetch-And-Add

- Atomically increment a value while returning the old value at a particular address

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- Ticket lock can be built with Fetch-And-Add.
  - Ensure progress for all threads → fairness

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

So Much Spinning

- Hardware-based spin locks are **simple** and they work.

- In some cases, these solutions can be quite **inefficient**.
  - Any time a thread gets caught spinning, it wastes an entire time slice doing nothing but checking a value.

How to avoid Spinning? We'll need OS Support too!

A Simple Approach: Just Yield

- When you are going to spin, give up the CPU to another thread.
  - OS system call moves the caller from the running state to the ready state.
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield();   // give up the CPU
}

void unlock() {
    flag = 0;
}
```

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;
    *ptr = new;
    return old;
}
```

Using Queues and Sleeping

- **Queue** to keep track of which threads are waiting to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

```
typedef struct __lock_t {
    int flag;      // lock is acquired or not
    int guard;     // to protect the queue
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Wakeup/Waiting Race

- In case of releasing the lock (thread A) just before the call to `park()` (thread B) → Thread B would sleep forever (potentially).

- Solaris solves this problem by adding a third system call: `setpark()`
  - By calling this routine, a thread can indicate it is about to `park()`.
  - If it happens to be interrupted and another thread calls `unpark()` before `park()` is actually called, the subsequent `park()` returns immediately instead of sleeping.

```
// Code modification inside of lock()
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

Futex

- Linux provides a `futex` (is similar to Solaris's `park` and `unpark`).
  - Fast Userspace Mutex
  - `futex`() system call provides a method for waiting until a certain condition becomes true.

  - `futex_wait(address, expected)`
    - Put the calling thread to sleep
    - If the value at `address` is not equal to `expected`, the call returns immediately.
  - `futex_wake(address)`
    - Wake one thread that is waiting on the queue.

Futex

- Snippet from lowlevellock.h in the `nptl` library
  - The high bit of the integer v: track whether the lock is held or not
  - All the other bits : the number of waiters

```
void mutex_lock(int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is the fastpath) */
    if (atomic_bit_test_set(mutex, 31) == 0)
        return;
    atomic_increment(mutex);
    while (1) {
        if (atomic_bit_test_set(mutex, 31) == 0) {
               atomic_decrement(mutex);
               return;
        }
        /* We have to wait now. First make sure the futex value
           we are monitoring is truly negative (i.e. locked). */
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait(mutex, v);
    }
}
```

```
void mutex_unlock(int *mutex) {
    /* Adding 0x80000000 to the counter results in 0
       if and only if there are not other interested threads */
    if (atomic_add_zero(mutex, 0x80000000))
        return;

    /* There are other threads waiting for this mutex,
       wake one of them up */
    futex_wake(mutex);
}
```

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock is about to be released.
  - First phase
    - The lock spins for a while, hoping that it can acquire the lock.
    - If the lock is not acquired during the first spin phase, a second phase is entered,
  - Second phase
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

Condition Variables

- There are many cases where a thread wishes to check whether a condition is true before continuing its execution.
- Example:
  - A parent thread might wish to check whether a child thread has completed.
  - This is often called a `join()`.

Waiting for Child: Spin-based Approach

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

This is hugely inefficient as the parent spins and wastes CPU time.

How to Wait for a Condition

- Condition variable
  - Queue of threads
  - **Waiting** on the condition
    - An explicit queue that threads can put themselves on when some state of execution is not as desired.
  - **Signaling** on the condition
    - Some other thread, when it changes it state, can wake one of those waiting threads and allow them to continue.
- Three in a package
  - Condition variable `c`
  - State variable `m`
  - `lock L;` // to protect state variable

POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion
  - Creating and initializing

```
pthread_mutex_t m;
pthread_cond_t c;

pthread_mutex_init(&m, NULL);
pthread_cond_init(&c, NULL);
```

POSIX Condition Variables

- Operation
  - Thread waiting for the condition `a == b` to become true

```
pthread_mutex_lock(&m);
while (a != b)
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
```

    - The wait() call takes a mutex as a parameter.
      - The wait() call release the lock and put the calling thread to sleep.
      - When the thread wakes up, it must re-acquire the lock.

  - Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&m);
a = b;
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

Waiting for Child: Condition Variable

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
```

```
void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Waiting for Child: Condition Variable

- Parent:
  - Creates the child thread and continues running itself.
  - Calls into `thr_join()` to wait for the child thread to complete.
    - Acquires the lock.
    - Checks if the child is done.
    - Puts itself to sleep by calling `wait()`.
    - Releases the lock.
- Child:
  - Prints the message “child”.
  - Calls `thr_exit()` to wake up the parent thread.
    - Grabs the lock.
    - Sets the state variable done.
    - Signals the parent thus waking it.

The Importance of the State Variable done

```
void thr_exit() {
    pthread_mutex_lock(&m);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void thr_join() {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

- Imagine the case where the child runs immediately.
  - The child will signal, but there is no thread asleep on the condition.
  - When the parent runs, it will call wait and be stuck.
  - No thread will ever wake it.

Another Poor Implementation

```
void thr_exit() {
    done = 1;
    pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        pthread_cond_wait(&c);
}
```

- The issue here is a subtle race condition.
  - The parent calls `thr_join()`.
    - The parent checks the value of done.
    - It will see that it is 0 and try to go to sleep.
    - Just before it calls wait to go to sleep, the parent is interrupted and the child runs.
  - The child changes the state variable done to 1 and signals.
    - But no thread is waiting and thus no thread is woken.
    - When the parent runs again, it sleeps forever.

Agenda

- Synchronization Problem
- Locks
- Building a Lock with Hardware Support
- Building a Lock with OS Support
- Conditional Variables
- Bounded Buffer Problem

Bounded Buffer

- A bounded buffer is used when you pipe the output of one program into another.
  - Example: `grep foo file.txt | wc -l`
    - The `grep` process is the producer.
    - The `wc` process is the consumer.
    - Between them is an in-kernel bounded buffer.
  - Bounded buffer is shared resource → Synchronized access is required!

Producer/Consumer: Version 1

```
int buffer;
int count = 0;     // initially, empty

void put(int value) {
    assert(count == 0);  // abort the program
                         // if assertion is false
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    int i;
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

- Only put data into the buffer when count is zero.
  - i.e., when the buffer is empty.
- Only get data from the buffer when count is one.
  - i.e., when the buffer is full.

Producer/Consumer: Single CV and If

- A single condition variable `cond` and associated lock `mutex`

```
pthread_cond_t cond;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        pthread_cond_signal(&cond);            // p5
        pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        pthread_cond_signal(&cond);            // c5
        pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

- `p1-p3`: A producer waits for the buffer to be empty.
- `c1-c3`: A consumer waits for the buffer to be full.
- With just a single producer and a single consumer, the code works.

If we have more than one of producer and consumer?

Thread Trace: Broken Solution (Version 1)

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

Thread Trace: Broken Solution (Version 1)

- The problem arises for a simple reason:
  - After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer *changed by* T_{c2}.
  - There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
    - Virtually every system ever built employs *Mesa semantics*.
  - Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken.

Producer/Consumer: Single CV and While

- Consumer T_{c1} wakes up and re-checks the state of the shared variable.
  - If the buffer is empty, the consumer simply goes back to sleep.

```
pthread_cond_t cond;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // p1
        while (count == 1)                     // p2
            pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        pthread_cond_signal(&cond);            // p5
        pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        pthread_cond_signal(&cond);            // c5
        pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

  - A simple rule to remember with condition variables is to always use while loops. However, this code still has a bug (next page).

Thread Trace: Broken Solution (Version 2)

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

A consumer should not wake other consumers, only producers, and vice-versa.

Producer/Consumer: Two CVs and While

- Use two condition variables and while
  - **Producer** threads wait on the condition `empty`, and signals `fill`.
  - **Consumer** threads wait on `fill` and signal `empty`.

```
pthread_cond_t empty, fill;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // p1
        while (count == 1)                     // p2
            pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        pthread_cond_signal(&fill);            // p5
        pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            pthread_cond_wait(&fill, &mutex); // c3
        int tmp = get();                       // c4
        pthread_cond_signal(&empty);           // c5
        pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Producer/Consumer: Final Version

- More **concurrency** and **efficiency** → Add more buffer slots.
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

```
pthread_cond_t empty, fill;
pthread_mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // p1
        while (count == 1)                     // p2
            pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        pthread_cond_signal(&fill);            // p5
        pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            pthread_cond_wait(&fill, &mutex); // c3
        int tmp = get();                       // c4
        pthread_cond_signal(&empty);           // c5
        pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

p2: **A producer** only sleeps if all buffers are currently filled.
c2: **A consumer** only sleeps if all buffers are currently empty.

References

- Ch28 and Ch30, Operating Systems: Three Easy Pieces
- https://oslab.kaist.ac.kr/ostepslides/