Lecture 14: Concurrency – Lock and Conditional Variable

The Course Organization (Bottom-up)

Concurrency
- Race Conditions, Lock/Semaphore
- Thread

Virtualization
- Memory Management — HW#4
- Process and CPU Scheduling — HW#3

Persistence
- IO Devices and Storage
- File System

HW#2 & Project

System Calls (User-level Programming) — HW#1

Course Overview

Part I: Lock

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);
```

Locks: The Basic Idea

- Lock variable holds <u>the state of </u>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.

The semantics of the lock()

- `lock()`
  - **Try to** acquire the lock.
  - If <u>no other thread holds</u> the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be <u>the owner of</u> the lock.
  - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

Pthread Locks - mutex

- The name that the POSIX library uses for a lock.
  - Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

  - We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

Building A Lock

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the **hardware** and the **OS**.

Evaluating locks – Basic criteria

- **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)
- **Performance**
  - The time overheads added by using the lock

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

Why hardware support needed?

- **First attempt**: 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;
}
```

Why hardware support needed? (Cont.)

- **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**: <u>Spin-waiting</u> 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 (Atomic Exchange)

- 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 to by the `ptr`.
- *Simultaneously* **update**(setting) said value to `new`.
- This sequence of operations is performed atomically.

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 <u>a preemptive scheduler</u>.

Evaluating Spin Locks

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

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;
}
```

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- **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`.

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.

Part II. Condition Variables

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()`.

Condition Variables (Cont.)

A Parent Waiting For Its Child

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

What we would like to see here is:

```
parent: begin
child
parent: end
```

Parent waiting fore 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
  - **Waiting** on the condition
    - <u>An explicit queue</u> 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 said state*, can wake one of those waiting threads and allow them to continue.

Definition and Routines

- Declare condition variable

```
pthread_cond_t c;
```

  - Proper initialization is required.

- Operation (the POSIX calls)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

  - The wait() call takes a <u>mutex</u> 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.

Parent waiting for Child: Use a 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);
}

```

Parent waiting for Child: Use a condition variable

```
(cont.)
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;
}
```

Parent waiting for Child: Use a condition variable

- **Parent:**
  - Create the child thread and continues running itself.
  - Call into `thr_join()` to wait for the child thread to complete.
    - Acquire the lock
    - Check if the child is done
    - Put itself to sleep by calling `wait()`
    - Release the lock
- **Child:**
  - Print the message “child”
  - Call `thr_exit()` to wake the parent thread
    - Grab the lock
    - Set the state variable `done`
    - Signal 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);
}
```

`thr_exit()` and `thr_join()` without variable `done`

- Imagine the case where the *child runs immediately*.
  - The child will signal, but there is <u>no thread asleep</u> 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 <u>interrupted</u> 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.

The Producer / Consumer (Bound Buffer) 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 in to a work queue
  - *Consumer threads* take requests out of this queue and process them

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.

The Put and Get Routines (Version 1)

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

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

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

- 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 Threads (Version 1)

```
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);
    }
}
```

- **Producer** puts an integer into the shared buffer loops number of times.
- **Consumer** gets the data out of that shared buffer.

Producer/Consumer: Single CV and If Statement

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

```
cond_t cond;
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
```

Producer/Consumer: Single CV and If Statement

```
            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	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in …
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		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 → <u>Mesa semantics</u>.
    - Virtually every system ever built employs *Mesa semantics*.
  - <u>Hoare semantics</u> 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.

```
cond_t cond;
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
    }
}
```

Producer/Consumer: Single CV and While

```
(Cont.)
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	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

Thread Trace: Broken Solution (Version 2) (Cont.)

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
…	…	…	…	…	…	…	(*cont.*)
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep …**

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

The single Buffer Producer/Consumer Solution

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

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

```

The single Buffer Producer/Consumer Solution

```
(Cont.)
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

The Final Producer/Consumer Solution

- 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;
}
```

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
cond_t empty, full;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                    // p1
        while (count == MAX)                           // p2
            Pthread_cond_wait(&empty, &mutex);         // p3
        put(i);                                        // p4
        Pthread_cond_signal(&full);                    // 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(&full, &mutex);          // c3
        int tmp = get();                               // c4
```

The Final Producer/Consumer Solution (Cont.)

```
(Cont.)
                Pthread_cond_signal(&empty);           // c5
                Pthread_mutex_unlock(&mutex);          // c6
                printf("%d\n", tmp);
            }
        }
```

The Final Working Solution (Cont.)

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

Covering Conditions

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);        // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads.**
    - <u>Cost</u>: too many threads might be woken.
    - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep.

Summary

- Lock
  - Pthread lock
  - Implementation
    - Test and Set
    - Spin lock based on test-and-set
- Condition Variable
  - Pthread condition variable
  - Consumer/Producer
- Next: Semaphore