CMPT 300 — Operating Systems I

Summer 1999

Segment 6:

Synchronization

Melissa O'Neill

Background

- *Concurrent access to shared data may result in inconsistencies due to race conditions.*
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

The Bounded Buffer Problem

Two processes:

- A **producer**, creating data items
- A **consumer**, using them up

Bounded Buffer, Producer — buffer1.cc

```
const int N = 128;                 // maximum capacity of the buffer
volatile item buffer[N];           // the buffer itself
volatile int in = 0;               // the next position to insert at
volatile int out = 0;              // the next position to remove from

void producer() {
    item produced_item;
    for ( ; ; ) {
        produced_item = produce_item();
        while ((in + 1) % N == out) {
            // buffer full — wait
        }
        buffer[in] = produced_item;
        in = (in + 1) % N;
    }
}
```

Bounded Buffer, Consumer — buffer1.cc

```
void consumer() {
    item consumable_item;

    for ( ; ; ) {
        while ( in == out) {
            // buffer empty — wait
        }
        consumable_item = buffer[out];
        out = (out + 1) % N;
        consume_item(consumable_item);
    }
}
```

This solution to the problem never uses all the available slots in the buffer (when the buffer is "full" one slot remains unused).

Bounded Buffer, Producer — buffer2.cc

```
const int N = 128;              // maximum capacity of the buffer
volatile item buffer[N];        // the buffer itself
volatile int count = 0;         // number of items in the buffer

void producer() {
    int in = 0;
    item produced_item;

    for ( ; ; ) {
        produced_item = produce_item();
        while (count == N) {
            // buffer full — wait
        }
        buffer[in] = produced_item;
        in = (in + 1) % N;
        count = count + 1;
    }
}
```


Bounded Buffer, Consumer — buffer2.cc

```
void consumer() {
    int out = 0;
    item consumable_item;

    for ( ; ; ) {
        while ( count == 0) {
            // buffer empty — wait
        }
        consumable_item = buffer[out];
        out = (out + 1) % N;
        count = count − 1;
        consume_item(consumable_item);
    }
}
```

The statements "count = count + 1" and "count = count − 1" must be executed atomically.

Bounded Buffer, Consumer — buffer2.s

```
_producer:
        pea a6@
        movel sp,a6
        moveml #d2/d3/d4/d5/a2/a3,sp@-
        clrl d3
        lea _produce_item,a3
        movel #_buffer,d4
        lea _count,a2
          ⋮
        movel a2@,d0
        addql #1,d0
        movel d0,a2@
          ⋮
```

Clearly, the code does not atomically change count. (Similar code is present for producer).

The Critical-Section Problem

The problem of the two processes competing to update count is an example of the critical-section problem.

The critical section problem exists where

- *n* processes all competing to use some shared data
- Each process has a code segment, called a **critical section**, in which the shared data is accessed.
- We must ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

The Critical-Section Problem — Structure

```
void foo() {
    for ( ; ; ) {
        // enter critical section
        foo_critical_section_actions();
        // leave critical section
        foo_other_actions();
    }
}

void bar() {
    for ( ; ; ) {
        // enter critical section
        bar_critical_section_actions();
        // leave critical section
        bar_other_actions();
    }
}
```


The Critical-Section Problem — Solution Requirements

Any solution to the critical-section problem must satisfy the following requirements:

- **Mutual Exclusion**. If a process 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 process that will enter its 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.

(Assume processes don't hang/die inside the critical section.)

The Critical-Section Problem — Turn Taking (Solution Attempt)

```
enum { Foo, Bar } turn = Foo;      // shared data — whose turn it is

void foo() {
    for ( ; ; ) {
        while (turn != Foo) {
            // let bar take its turn
        }
        foo_critical_section_actions();
        turn = Bar;
        foo_other_actions();
    }
}
```

The Critical-Section Problem — Turn Taking (contd.)

```
void bar() {
    for ( ; ; ) {
        while (turn != Bar) {
            // let foo take its turn
        }
        bar_critical_section_actions();
        turn = Foo;
        bar_other_actions();
    }
}
```


The Critical-Section Problem — Using Flags (Solution Attempt)

```
bool foo_busy = false;     // set if foo in/entering critical section
bool bar_busy = false;     // set if bar in/entering critical section

void foo() {
    for ( ; ; ) {
        foo_busy = true;
        while (bar_busy == true) {
            // let bar finish
        }
        foo_critical_section_actions();
        foo_busy = false;
        foo_other_actions();
    }
}
```


The Critical-Section Problem — Using Flags

```
void bar() {
    for ( ; ; ) {
        bar_busy = true;
        while (foo_busy == true) {
            // let foo finish
        }
        bar_critical_section_actions();
        bar_busy = false;
        bar_other_actions();
    }
}
```


The Critical-Section Problem — Using Both

```
bool foo_busy = false;     // set if foo in/entering critical section
bool bar_busy = false;     // set if bar in/entering critical section
enum { Foo, Bar } turn = Foo;     // whose turn it is for tiebreaker

void foo() {
    for ( ; ; ) {
        foo_busy = true;
        turn = Bar;
        while (bar_busy == true && turn == Bar) {
            // let bar finish
        }
        foo_critical_section_actions();
        foo_busy = false;
        foo_other_actions();
    }
}
```

The Critical-Section Problem — Using Both

```
void bar() {
    for ( ; ; ) {
        bar_busy = true;
        turn = Foo;
        while (foo_busy == true && turn == Foo) {
            // let foo finish
        }
        bar_critical_section_actions();
        bar_busy = false;
        bar_other_actions();
    }
}
```


Lamport's Bakery Algorithm — Critical-Section Solution for *n* Tasks

Based on the idea of customers taking tickets at a bakery (except that several customers can have the same ticket):

- Before entering its critical section, task receives a number.
- Holder of the smallest number enters the critical section.
- If tasks T_i and T_j receive the same number, if i < j, then T_i is served first.
- The numbering scheme always generates numbers in increasing order of enumeration (e.g., 1,2,3,3,3,3,4,5…)

Lamport's Bakery Algorithm

```
volatile bool choosing[N];        // set while task is taking a number
volatile int numbers[N];          // numbers held by each task

void task(const int i) {
    for ( ; ; ) {
        choosing[i] = true;
        numbers[i] = max(numbers) + 1;
        choosing[i] = false;
        for (int j = 0; j < n; ++j) {
            while (choosing[j]) {
                // wait while task j chooses
            }
            while (number[j] != 0 && ( number[j] < number[i]
                        || (number[j] == number[i] && j < i ))) {
                // wait for other tasks
            }
        critical_section_actions(i);
        numbers[i] = 0;
        other_actions(i);
    }
}
```


Hardware "Solutions" to the Critical-Section Problem — Disabling Interrupts

```
void task(const int i) {
    for ( ; ; ) {
        disable_interrupts();
        critical_section_actions(i);
        enable_interrupts();
        other_actions(i);
    }
}
```

The Test and Set Operation

```
inline bool test_and_set(bool &flag) {
    bool old_value = 0;
    asm {
        tas     flag
        sne     old_value
    }
    return old_value;
    // update flag to hold true, but return the value flag held before
    // it was updated.
}
```


Hardware "Solutions" to the Critical-Section Problem — Test and Set

```
volatile bool lock = false;               // shared mutual exclusion lock

void task(const int i) {
    for ( ; ; ) {
        while (test_and_set(lock) == true) {        // lock set already
            // do nothing — wait for lock to be released
        }
        critical_section_actions(i);
        lock = false;
        other_actions(i);
    }
}
```


Trying to Avoid Busy-Waiting — Test and Set

```
volatile bool lock = false;               // shared mutual exclusion lock

void task(const int i) {
    for ( ; ; ) {
        while (test_and_set(lock)) {              // lock set already
            sched_yield();        // yield the processor to another task
        }
        critical_section_actions(i);
        lock = false;
        other_actions(i);
    }
}
```

Less wasteful on a uniprocessor, but not ideal.

Trivial Process Class

```
class Task {
    static Task * self();   // returns the a pointer to the current task
    void snooze();                              // puts a task to sleep
    void wakeup();                           // wakes up a sleeping task
}
```

Trying to Avoid Busy-Waiting — Test and Set

```
volatile bool lock = false;            // shared mutual exclusion lock
volatile queue<Task *> waiting;        // tasks waiting for the lock

void task(const int i) {
    for ( ; ; ) {
        if (test_and_set(lock)) {         // someone locked it already
            waiting.push(Task.self());    // sleep — they'll wake us up
            Task.self()->snooze();
        }
        critical_section_actions(i);
        if (!waiting.empty()) {                // someone's waiting
            waiting.front()->wakeup();         // wake them up
            waiting.pop();
        } else
            lock = false;
        other_actions(i);
    }
}
```


Mutex Class

```
class Mutex {
public:
    Mutex();                       // initial value, unlocked
    void lock();     // lock the mutex — task sleeps until lock obtained
        // REQUIRE: Not already locked by this task
    void unlock();          // unlock the mutex — can wake up a sleeper
        // REQUIRE: Not already unlocked
private:
        ⋮
}
```

Provided by the operating system!

- OS designers get to decide whether to use busy-waiting or a queue of waiting tasks.

Critical Sections using Mutexes

```
Mutex guard;        // shared mutual exclusion lock, initially unlocked

void task(const int i) {
    for ( ; ; ) {
        guard.lock();
        critical_section_actions(i);
        guard.unlock();
        other_actions(i);
    }
}
```

(Mutexes are sometimes called binary semaphores. When people call them "binary semaphores", they usually intend to use them in "weird" ways.)

Mutex Class, Extended

```
class Mutex {
public:
    Mutex();                       // initial value, unlocked
    void lock();     // lock the mutex — task sleeps until lock obtained
    bool try_lock();  // try to lock the mutex, returns true on success
    void unlock();          // unlock the mutex — can wake up a sleeper
private:
        ⋮
}
```

(See pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, and pthread_mutex_destroy in the Unix manual pages — these interfaces are C based, but it is easy to wrap them into a C++ class).

Semaphores

```
class Semaphore {
public:
    Semaphore(int i = 1);              // initializes counter to i
    void wait();           // --counter, task sleeps until counter > 0
    void signal();    // ++counter, may wake up a task that is sleeping
private:
    int counter;
        ⋮
}
```

(See sem_init, sem_wait, sem_trywait, sem_post, and sem_destroy in the Unix manual pages — these interfaces are C based, but it is easy to wrap them into a C++ class. But you almost never *want* to use semaphores anyway).

Implementing Mutexes using Semaphores

```
class Mutex {
private:
    Semaphore    access;
public:
    Mutex() : access(1) {}     // initializes underlying semaphore to 1
    void lock() {
        access.wait();                 // wait until we can have access
    }
    void unlock() {
        access.signal(); // signal that access by others is now allowed
    }
}
```


Implementing Semaphores using Mutexes

```
class Semaphore {
private:
    int     count;
    Mutex count_guard;
    Mutex access;
public:
    Semaphore(int i = 1) : count(i) {
        access.lock();
    }
    void wait() {
        count_guard.lock();
        --count;
        if (count < 0) {
            count_guard.unlock();
            access.lock();
        }
        count_guard.unlock();
    }
```


Implementing Semaphores using Mutexes (continued)

```
        ⋮
    void signal() {
        count_guard.lock();
        ++count;
        if (count <= 0) {
            access.unlock();
        } else {
            count_guard.unlock();
        }
    }
}
```

(It *should* take you a few minutes to satisfy yourself that this code works.   Yuck!)

Bounded Buffer using Semaphores, Producer — buffer3.cc

```
const int N = 128;                  // maximum capacity of the buffer
volatile item buffer[N];            // the buffer itself
Semaphore empty_slot(N);            // any free slots?
Semaphore filled_slot(0);           // any filled slots?

void producer() {
    int in = 0;
    item produced_item;

    for ( ; ; ) {
        produced_item = produce_item();

        empty_slot.wait();

        buffer[in] = produced_item;
        in = (in + 1) % N;

        filled_slot.signal();
    }
}
```


Bounded Buffer using Semaphores, Consumer — buffer3.cc

```
void consumer() {
    int out = 0;
    item consumable_item;

    for ( ; ; ) {
        filled_slot.wait();

        consumable_item = buffer[out];
        out = (out + 1) % N;

        empty_slot.signal();

        consume_item(consumable_item);
    }
}
```


Readers–Writers Problem

Sometimes a data item has have readers (which don't make changes to the data) and writers (which *do* make changes).

- Many readers can share access to the item, but writers must have exclusive access.
- Solution: Reader–Writer locks.

Implementing Reader–Writer Locks (continued)

```
class ReaderWriterLock {
private:
    Semaphore      access;
    int            readcount;
    Semaphore      readcount_guard;

public:
    ReaderWriterLock() :
        access(),                          // initially, unlocked
        readcount(0),                      // with no readers
        readcount_guard()        // and no one accessing readcount
    { }

    void lockWrite() {
        access.wait();
    }

    void unlockWrite() {
        access.signal();
    }
```

Implementing Reader–Writer Locks (continued)

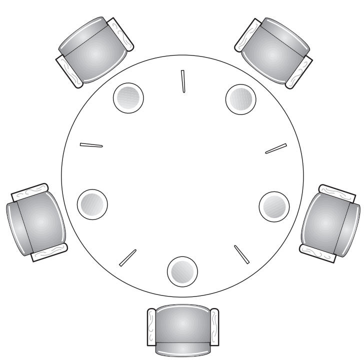
```
    ⋮
void lockRead() {
    readcount_guard.wait();
    ++readcount;
    if (readcount == 1)
        access.wait();
    readcount_guard.signal();
}

void unlockRead() {
    readcount_guard.wait();
    --readcount;
    if (readcount == 0)
        access.signal();
    readcount_guard.signal();
}
```


Class Exercise: Does this code work?

If you think it works, is there room for improvement regarding fairness?

If you don't think it works, what's wrong with it?

The Dining Philosophers Problem



- Each philosopher alternates between periods of:
  - Thinking
  - Eating

The Dining Philosophers Problem

```
const int    N = 5;                   // five philosophers
Semaphore    chopstick[5];   // one semaphore for each chopstick
                                      // initially, set to 1

// Each Philosopher i:
void philosopher(int i) {
    for ( ; ; ) {
        chopstick[i].wait();                // pick up left chopstick
        chopstick[(i+1) % N].wait();        // pick up right chopstick
        eat();
        chopstick[i].signal();              // drop left chopstick
        chopstick[(i+1) % N].signal();      // drop left chopstick
        think();
    }
}
```

The Dining Philosophers Problem (contd.)

Class Exercise: Is this solution to the problem entirely satisfactory?

If not, how would you fix the algorithm?

Monitors

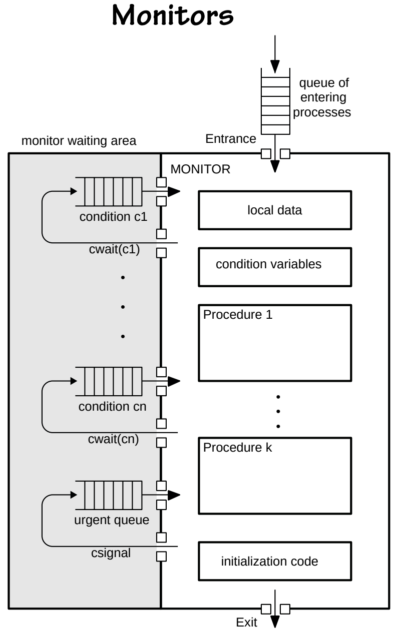
Semaphores are pretty low level and hard to get right.

- High-level synchronization construct, based on classes
- Only one task can be running "inside" the class at a time
- Declare classes like this:

  monitor class *classname* {
  private:
  ... private member declarations ...
  public:
  ... member function declarations only ...
  }

- But how do we handle needing to wait within the monitor?

Monitors



Monitors

- Allow *condition variables*, of class *Condition*:

  Condition tool_ready
- Inspired by semaphores, class *Condition* supports two methods, cwait() and csignal().
  - x.cwait() suspends the process until another process invokes x.signal()
  - x.csignal() resumes *exactly one* suspended process — if no process is suspended, the signal operation has no effect.

Class Exercise: How does this differ from the wait and signal methods of *Semaphores*?

When should the process resumed by x.signal() run? What are the options and what are the tradeoffs?

Monitors — Example, Dining Philosophers

```
monitor class DiningPhilosophers {
public:
    const int       N = 5;                          // five philosophers

private:
    enum { Thinking, Hungry, Eating }  state[N];
    Condition   can_eat[N];             // used to wake up philosophers

public:
    void grab_both_chopsticks (int i) {
        state[i] = Hungry;
        while (!chopsticks_available(i)) {
            can_eat[i].cwait();
        }
        state[i] = Eating;
    }

    void drop_both_chopsticks (int i) {
        state[i] = Thinking;
        might_eat_now((i–1) % N);     // maybe a neighbour can eat
        might_eat_now((i+1) % N);    // now that we're done eating
    }
```


Monitors — Example, Dining Philosophers (contd.)

```
        ⋮

    bool chopsticks_available (int k) {
        return (state[(k–1) % N] != Eating
                    && state[(k+1) % N] != Eating);
    }

    void might_eat_now (int k) {
        if (state[k] == Hungry && chopsticks_available(k) ) {
            can_eat[k].csignal();
        }
    }

    DiningPhilosophers() {
        for (int i = 0; i < 5; ++i) {
            state[i] = Thinking;
        }
    }
}
```


Monitors — Example, Dining Philosophers (contd.)

We run the dining philosophers using:

```
DiningPhilosophers phils;

void philosopher(int i) {

    for ( ; ; ) {

        phils.grab_both_chopsticks(i);

        eat();

        phils.drop_both_chopsticks(i);

        think();

    }
}

void main () {
    parfor (int i = 0; i < DiningPhilosophers::N; ++i)
        philosopher(i);
}
```


Class Exercise: Is this version of the dining philosophers problem satisfactory?

Monitors — Implementation Using Semaphores

- Extra (hidden) variables, added to class

Semaphore	guard;	// *initialized to* 1
Semaphore	insider_can_run;	// *initialized to* 0
int	insider_count;	// *initialized to* 0

- Each public member function becomes:

```
guard.wait();
… original member function body …
if (insider_count > 0)
    insider_can_run.signal();
else
    guard.signal();
```

- Each condition variable, *x*, becomes:

Semaphore	x_happened;	// *initialized to* 0
int	x_waiting_count;	// *initialized to* 0

Monitors — Implementation Using Semaphores (contd.)

- *x*.cwait() *becomes*:

```
++x_waiting_count;
if (insider_count > 0)
    insider_can_run.signal();
else
    guard.signal();
x_happened.wait();
--x_waiting_count;
```

- *x*.signal() *becomes*:

```
if (x_waiting_count > 0) {
    ++insider_count;
    x_happened.signal();
    insider_can_run.wait();
    --insider_count;
}
```


Priority Inversion

Class Exercise: What happens if a low-priority process is in the monitor when a higher-priority thread wants to run (and enter the monitor itself)?

What other synchronization methods does priority inversion apply to?

PThreads Condition Variables

PThreads provides the same kind of condition variables we saw with monitors, but rather than associate a condition with monitor, Pthreads associates them with a guardian mutex.

- pthread_cond_wait(*condition*, *mutex*) — unlocks *mutex* and waits for *condition* to be signalled — when awakened, thread will relock *mutex*
- pthread_cond_signal(*condition*) — signals *condition* to one waiting thread (the guardian mutex should be <u>locked</u> when signalling to prevent lost wakeup bugs)
- pthread_cond_broadcast(*condition*) — signals *condition* to all waiting threads (only one of them will run at a time)
- pthread_cond_init(*condition*) & pthread_cond_init(*condition*) initialize and destroy condition variables

Segment Review

You should be able to:

- Describe the critical-section problem
- Determine whether software-based synchronization algorithms operate correctly (or contain race conditions)
- Describe and implement semaphores, critical regions, and monitors
- Design algorithms using semaphores, critical regions, or monitors for synchronization
- Describe and implement solutions to several classical synchronization problems
- Explain and prevent priority inversion