Thank Dr. Dakai Zhu, Dr. Palden Lama, and Dr. Tim Richards (UMASS) for providing their slides.
Lecture Outline

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section
- Synchronization mechanism
  - Hardware supported instructions: e.g., TestAndSet
  - Software solution: e.g., semaphore
- Classical Synchronization Problems
- High-level synchronization structure: Monitor
- Case study for synchronization
  - Pthread library: mutex and conditional variables
  - Java inherit monitor and conditional variable
Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To present both software and hardware solutions of the critical-section problem.
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

Shared Data

- At the same logical address space.
- At different address space through messages (later in DS).
Concurrent Access to Shared Data

- Two threads A and B have access to a shared variable “Balance”

**Thread A:**
Balance = Balance + 100

- A1. LOAD R1, BALANCE
- A2. ADD R1, 100
- A3. STORE BALANCE, R1

**Thread B:**
Balance = Balance - 200

- B1. LOAD R1, BALANCE
- B2. SUB R1, 200
- B3. STORE BALANCE, R1
What is the problem then?

**Observe:** In a *time-shared* system, the *exact instruction execution order* cannot be predicted

- **Scenario 1:**
  - A1. LOAD R1, BALANCE
  - A2. ADD R1, 100
  - A3. STORE BALANCE, R1
  - *Context Switch!*
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1
  - *Sequential correct execution*
  - Balance is effectively decreased by 100!

- **Scenario 2:**
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - *Context Switch!*
  - A1. LOAD R1, BALANCE
  - A2. ADD R1, 100
  - A3. STORE BALANCE, R1
  - *Context Switch!*
  - B3. STORE BALANCE, R1
  - *Mixed wrong execution*
  - Balance is effectively decreased by 200!!!
\[ a = 0; \ b = 0; // Initial state \]

**Thread 1**

- **T1-1:** if (b == 0)
- **T1-2:** a = 1;

**Thread 2**

- **T2-1:** if (a == 0)
- **T2-2:** b = 1;

- \[ \text{99.43\%} \]
- \[ \text{0.56\%} \]
- \[ \text{0.01\%} \]
Race Conditions

- Multiple processes/threads write/read shared data and the outcome depends on the particular order to access shared data are called race conditions
  - A serious problem for concurrent system using shared variables!

How do we solve the problem?!

- Need to make sure that some high-level code sections are executed atomically
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process
Critical-Section (CS) Problem

- Multiple processes/threads compete to use some shared data

- critical section (critical region): a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

- Problem – ensure that only one process/thread is allowed to execute in its critical section (for the same shared data) at any time. The execution of critical sections must be mutually exclusive in time.
Mutual Exclusion

A enters critical region

A leaves critical region

B attempts to enter critical region

B enters critical region

B leaves critical region

Process A

Process B

T₁ T₂ T₃ T₄

B blocked

Time
Solving the Critical-Section Problem

- **Mutual Exclusion**
  - No two processes can simultaneously enter into the critical section.

- **Bounded Waiting**
  - No process should wait forever to enter a critical section.

- **Progress**
  - Non-related process can not block a process trying to enter one critical section

- **Relative Speed**
  - No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed).
Suppose two processes try to open files at the same time, to allocate memory etc.

Two general approach

- **Nonpreemptive** kernel (*free from race RC, easy to design*)
- **Preemptive** kernel (*suffer from race RC, hard to design*)

Why, then, would we want nonpreemptive kernel

- Real-time systems
- Avoid arbitrarily long kernel processes
- Increase responsiveness
General Structure for Critical Sections

\[
\text{do } \{ \\
\quad \ldots \ldots \\
\quad \text{entry section} \\
\quad \text{critical section} \\
\quad \text{exit section} \\
\quad \text{remainder statements} \\
\} \text{ while } (1); \\
\]

In the \textit{entry section}, the process requests \textit{“permission”}. 
Solutions for CS Problem

- Software based
  - Peterson’s solution
  - Semaphores
  - Monitors

- Hardware based
  - Locks
  - disable interrupts
  - Atomic instructions: TestAndSet and Swap
Simple solution for **two threads only**

- **T0 and T1**: alternate between CS and remainder
- Assumption: LOAD & STORE are atomic
  - May not work in modern architectures (e.g., speculation)
- **Shared variables** between T0 and T1
  - `int turn`; → indicate whose turn to enter CS

```c
Thread 0:
-------
while(TRUE) {
    while (turn != 0) ;
    critical section
    turn = 1;
    remainder section
}

Thread 1:
-------
while(TRUE) {
    while (turn != 1) ;
    critical section
    turn = 0;
    remainder section
}
```

Problems??

1. two threads only
2. busy waiting
Peterson’s Solution (Mutual, progress, bounded waiting)

Thread 0:
--------
while(TRUE) {
    flag[0] = 1;  // I am ready
    turn = 1;
    while (flag[1]==1 &&
            turn == 1) ;
    critical section
    flag[0] = 0;  // I am not ready
    remainder section
}

Thread 1:
--------
while(TRUE) {
    flag[1] = 1;
    turn = 0;
    while (flag[0]==1 &&
            turn == 0) ;
    critical section
    flag[1] = 0;
    remainder section

- Boolean flag[2];  → this thread is ready to enter CS
- Mutual  → turn can be only one of them
- Progress, bounded waiting  → no endless while loop
Hardware Solution: Disable Interrupt

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
- Inefficient on multiprocessor systems

```
    do {
        ......
        DISABLE INTERRUPT
        critical section
        ENABLE INTERRUPT
        remainder statements
    } while (1);
```

What are the problems with this solution?

1. Time consuming, decreasing efficiency
2. Clock problem
3. Machine dependent
Hardware Instruction TestAndSet

- The TestAndSet instruction tests and modifies the content of a word **atomically** (non-interruptable)
- Only set the lock to 1 if lock is 0

```c
function LOCK(bool *lock) {
    do {
        while (TestAndSet(lock) == 1);
    } while (Lock(&lock));
    critical section
    //free the lock
    lock = false;
    remainder section
    while(true);
}
```

**What’s the problem?**

1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting
Another Hardware Instruction: Swap

Swap contents of two memory words

```c
void Swap (bool *a, bool *b){
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

bool lock = FALSE;
While(true){
    bool key = TRUE;
    while(key == TRUE) {
        Swap(&key, &lock) ;
    }
    critical section;
    lock = FALSE; //release permission

What's the problem?
1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

LOCK == FALSE
Semaphores

- Synchronization *without* busy waiting
  - Motivation: Avoid busy waiting by *blocking* a process execution until some *condition* is satisfied

- Semaphore $S$ – integer variable

- Two indivisible (atomic) operations: *how*? → *later*
  - `wait(s)` (also called `P(s)` or `down(s)` or `acquire()`);
  - `signal(s)` (also called `V(s)` or `up(s)` or `release()`)
  - User-visible operations on a semaphore
  - Easy to generalize, and less complicated for application programmers
Semaphore Operations

- Semaphore is an integer.
- **wait(s):** //wait until s.value > 0; 
s.value--; /* Executed atomically! */
  - The value of s could be negative → the number of waits
- A process execute the wait operation on a semaphore with value <=0 is **blocked**
  - Blocked → non-runnable state, yield CPU to others
- **signal(s):** s.value++; /* Executed atomically! */
  - If s=0, wake up only one blocked process; which one?!

Definition is a spinlock: waste CPU, but no need for context switches
Semaphore Usage

- **Counting** semaphore – integer value can range over an unrestricted domain
  - Can be used to control access to a given resources with finite number of instances

- **Binary** semaphore – integer value can range only between 0 and 1; Also known as **mutex locks**

```c
S = number of resources
while(1){
    ......
    wait(S);
    use one of S resource
    signal(S);
    remainder section
}
```

```c
mutex = 1
while(1){
    ......
    wait(mutex);
    Critical Section
    signal(mutex);
    remainder section
}
```
Attacking CS Problem with Semaphores

- Shared data
  - semaphore mutex = 1; /* initially mutex = 1 */

- For any process/thread

```c
    do {
        ...
        wait(mutex);
        critical section
        signal(mutex);
        remainder section
    } while(1);
```
Revisit “Balance Update Problem”

- **Shared data:**
  ```c
  int Balance;
  semaphore mutex; // initially mutex = 1
  ```

- **Process A:**
  ```c
  ....
  wait (mutex);
  Balance = Balance - 100;
  signal (mutex);
  ....
  ```

- **Process B:**
  ```c
  ....
  wait (mutex);
  Balance = Balance - 200;
  signal (mutex);
  ....
  ```
Semaphore for General Synchronization

- Execute code $B$ in $P_j$ after code $A$ is executed in $P_i$
- Use semaphore $flag$ : what is the initial value?
- Code

\[
\begin{align*}
\text{Pi} : & & \text{Pj} : \\
& & \\
& \cdot & \cdot \\
& \cdot & \cdot \\
& \cdot & \text{wait}(flag) \\
A & & B \\
\text{signal}(flag) & & \\
\end{align*}
\]

What about 2 threads wait for 1 thread? Or 1 thread waits for 2 threads?
Implement Semaphores

```c
typedef struct {
    int value;
    struct process *list;
} semaphore;

wait(semaphore * s){
    s->value--;
    if (s->value <0){
        enlist(s->list);
        block();
    }
}

signal(semaphore * s){
    s->value++;
    if (s->value <=0)
        delist(P, s->list);
    wakeup(p);
}

Is this one without busy waiting?
```

Two operations by system calls

- **block** – place the current process to the waiting queue.
- **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.
Classical Synchronization Problems

- **Producer-Consumer Problem**
  - *Shared bounded-buffer*
  - Producer puts items to the buffer area, wait if buffer is full
  - Consumer consumes items from the buffer, wait if is empty

- **Readers-Writers Problem**
  - Multiple readers can access **concurrently**
  - Writers mutual exclusive with writes/readers

- **Dining-Philosophers Problem**
  - Multiple resources, get one at each time
Producer-Consumer Problem

With Bounded-Buffer

- Need to make sure that
  - The producer and the consumer do not access the buffer area and related variables at the same time
  - No item is available to the consumer if all the buffer slots are empty.
  - No slot in the buffer is available to the producer if all the buffer slots are full
What Semaphores are needed?

- \texttt{semaphore mutex, full, empty;}

\textit{What are the initial values?}

\texttt{Initially:}

- \texttt{full = 0} /* The number of full buffer slots */
- \texttt{empty = n} /* The number of empty buffer slots */
- \texttt{mutex = 1} /* controlling mutual access to the buffer pool */
Producer-Consumer Codes

**Producer**

```plaintext
do {
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
} while (1)
```

**Consumer**

```plaintext
do {
    ...
    wait(full);
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
} while (1)
```

What will happen if we change the order?

Department of Computer Science @ UTSA
Readers-Writers Problem

- Data (e.g. a file) is shared among concurrent reader/writer
- A writer must have exclusive access to the data object.
- Multiple readers may access the shared data simultaneously without a problem

What semaphores /variable do we need?

Shared data:

```
int readcount; //number of readers
semaphore mutex; //for readers to access ‘readcount’
semaphore wrt; //for writer/reader mutual exclusive
```

Initially

```
mutex = 1, readcount = 0, wrt = 1;
```

Department of Computer Science @ UTSA
Reader

```c
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex);
```

Writer

```c
wait(wrt);
...
writing is performed
...
signal(wrt);
```

Any problem with this solution?!

Starvation for writer
Advanced Reader/Writer Problems

- Preferred Reader: original solution favors readers

- **Preferred Writer Problem**
  - If there is a writer in or waiting, no additional reader in

- **Alternative Reader/Writer Problem**
  - Reader/writer take turn to read/write
Dining-Philosophers Problem

Five philosophers share a common circular table. There are five chopsticks and a bowl of rice (in the middle). When a philosopher gets hungry, he tries to pick up the closest chopsticks.

A philosopher may pick up only one chopstick at a time, and cannot pick up a chopstick already in use. When done, he puts down both of his chopsticks, one after the other.

**Shared data**

```c
semaphore chopstick[5];
```

Initially all semaphore values are 1

A classic example of synchronization problem: allocate several resources among several processes in a deadlock-free and starvation-free manner.
Philosopher’s solution for the $i$’th philosopher:

```c
    do {
        think; // and become hungry
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
        eat
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
    } while (1);
```
Fixing deadlock of dining-philosophers problem

- Allows 4 person only, then one will get all chopsticks
- Pickup both chopsticks atomically

- Asymmetric solution
  - Odd: left, right
  - Even: right, left
Problems with Using Semaphores

Let $S$ and $Q$ be two semaphores initialized to 1

\[
\begin{align*}
T_0 & \quad T_1 \\
\text{wait} \ (S) ; & \quad \text{wait} \ (Q) ; \\
\text{wait} \ (Q) ; & \quad \text{wait} \ (S) ; \\
\ldots & \quad \ldots \\
\text{signal} \ (S) ; & \quad \text{signal} \ (Q) ; \\
\text{signal} \ (Q) ; & \quad \text{signal} \ (S) ;
\end{align*}
\]

**What is the problem with the above code?**
Problems with Using Semaphores (cont.)

- Strict requirements on the sequence of operations
  - Correct: wait (mutex) …. signal (mutex)
  - **Incorrect**: signal (mutex) …. wait (mutex);
    wait (mutex) …. wait(mutex);

- Complicated usages

- Incorrect usage could lead to
  - deadlock
Monitors

- **High-level synchronization construct** (implement in different languages) that provided mutual exclusion within the monitor AND the ability to wait for a certain condition to become true

  ```
  monitor monitor-name{
    shared variable declarations

    procedure body P1 (...) { . . . }
    procedure body P2 (...) { . . . }
    procedure body Pn (...) { . . . }

    {initialization codes; }
  }
  ```
monitors vs. semaphores

- **A Monitor:**
  - An object designed to be accessed across threads
  - Member functions enforce mutual exclusion

- **A Semaphore:**
  - A low-level object
  - We can use semaphore to implement a monitor
Schematic View of a Monitor

- Monitor construct ensures **at most one thread can be active** within the monitor at a given time.
- Shared data (local variables) of the monitor can be **accessed only by local procedures**.
class Account {
    private lock myLock;

    private int balance := 0
    invariant balance >= 0

    public method boolean withdraw(int amount)
    precondition amount >= 0
    {
        int ret;
        myLock.acquire();
        if balance < amount then ret = false
        else { balance := balance - amount ; ret = true }
        myLock.release();
    }

    public method deposit(int amount)
    precondition amount >= 0
    {
        myLock.acquire();
        balance := balance + amount
        myLock.release();
    }
}
Case Study

- Pthread Library: OS-independent
  - mutex locks
  - condition variables
  - barrier
Synchronization in Pthread Library

- Mutex variables
  - `pthread_mutex_t`
- Conditional variables
  - `pthread_cond_t`

- All POSIX thread functions have the form:
  
  `pthread[_object ]_operation`

- Most of the POSIX thread library functions return 0 in case of success and some non-zero error-number in case of a failure.
Mutex Variables: Mutual Exclusion

- A mutex variable can be either *locked* or *unlocked*
  - `pthread_mutex_t lock; // lock is a mutex variable`

- Initialization of a mutex variable by default attributes
  - `pthread_mutex_init( &lock, NULL );`

- Lock operation
  - `pthread_mutex_lock( &lock );`

- Unlock operation
  - `pthread_mutex_unlock( &lock );`
Semaphore vs. Mutex_lock

Definition and initialization

volatile int cnt = 0;
sem_t mutex = 1;

Entering and Exit CS

for (i = 0; i < niter; i++) {
    Wait(&mutex);
    cnt++;
    Signal(&mutex);
}

for (i = 0; i < niter; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
Binary Semaphore and Mutex Lock?

- **Binary Semaphore:**
  - No ownership

- **Mutex lock**
  - Only the owner of a lock can release a lock.
  - Priority inversion safety: potentially promote a task
  - Deletion safety: a task owning a lock can’t be deleted.
Synchronization?

- Synchronization serves two purposes:
  - Ensure safety for shared updates
    - Avoid race conditions
  - Coordinate actions of threads
    - Parallel computation
    - Event notification

- ALL interleavings must be correct
  - there are lots of interleavings of events
  - also constrain as little as possible
Condition Variables

- In a critical section, a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.
  - It will suspend by waiting on a condition variable.
  - It will, however, release the critical section lock.
  - When that condition variable is signaled, it will become ready again; it will attempt to reacquire that critical section lock and only then will be able proceed.

- With POSIX threads, a condition variable can be associated with only one mutex variable.
Condition Variables (cont.)

- `pthread_cond_t   SpaceAvailable;`
- `pthread_cond_init (&SpaceAvailable, NULL );`

- `pthread_cond_wait`
- `pthread_cond_signal`
  - unblock one waiting thread on that condition variable

- `pthread_cond_broadcast`
  - unblock all waiting threads on that condition variable
Synchronization Operations

■ Safety
  ➢ Locks provide mutual exclusion

But, we need more than just mutual exclusion of critical regions…

■ Coordination
  ➢ *Condition variables* provide ordering
Synchronization Problem: Queue

- Suppose we have a thread-safe queue
  - `insert(item)`, `remove()`, `empty()`
  - must protect access with locks

- Options for removing when queue empty:
  - Return special error value (e.g., NULL)
  - Wait for something to appear in the queue
Three Possible Solutions

- **Spin lock**
  - Works?

- **Could release lock**
  - Works?

- **Re acquire Lock**

```c
lock();
while (empty()) {}
unlock();
v = remove();

unlock();
while (empty()) {}
lock();
v = remove();
unlock();
```

Works, but lots of checking...
Solution: Sleep!

- Sleep =
  
  ➢ “don’t run me until something happens”

- What about this?

```c
Dequeue()
lock();
if(queue empty) {
  sleep();
}
take one item;
unlock();
}

Enqueue()
lock();
insert item;
if(thread waiting)
  wake up dequeuer();
unlock();
```

Cannot hold lock while sleeping!
Condition Variables

- Special *pthread* data structure
- Make it possible/easy to go to sleep
  - Atomically:
    - release lock
    - put thread on wait queue
    - go to sleep
- Each CV has a queue of waiting threads
- Do we worry about threads that have been put on the wait queue but have NOT gone to sleep yet?
  - no, because those two actions are atomic
- Each condition variable associated with one lock
Condition Variables

Wait for 1 event, atomically release lock

- `wait(Lock& l, CV& c)`
  - If queue is empty, wait
    - Atomically releases lock, goes to sleep
    - You must be holding lock!
    - May reacquire lock when awakened (pthreads do)

- `signal(CV& c)`
  - Insert item in queue
    - Wakes up one waiting thread, if any

- `broadcast(CV& c)`
  - Wakes up all waiting threads
Condition Variable Exercise

Implement “Producer Consumer”
➢ One thread enqueues, another dequeues

```c
void * consumer (void *){
    while (true) {
        pthread_mutex_lock(&l);
        while (q.empty()){
            pthread_cond_wait(&nempty, &l);
        }
        cout << q.pop_back() << endl;
        pthread_mutex_unlock(&l);
    }
}
```

```c
void * producer(void *){
    while (true) {
        pthread_mutex_lock(&l);
        q.push_front (1);
        pthread_cond_signal(&nempty);
        pthread_mutex_unlock(&l);
    }
}
```

Questions?
➢ Can I use if instead of while (to check cond)?
Barrier

A barrier is used to order different threads (or a synchronization).

- A barrier for a group of threads means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Where it can be used?
- Watching movie.
Barrier Interfaces

- It is used when some parallel computations need to "meet up" at certain points before continuing.

- Pthreads extension includes barriers as synchronization objects (available in Single UNIX Specification)
  - Enable by #define _XOPEN_SOURCE 600 at start of file

- Initialize a barrier for count threads
  - int pthread_barrier_init(pthread_barrier_t *barrier,
    const pthread_barrier_attr_t *attr, int count);

- Each thread waits on a barrier by calling
  - int pthread_barrier_wait(pthread_barrier_t *barrier);

- Destroy a barrier
  - int pthread_barrier_destroy(pthread_barrier_t *barrier);
int main () // ignore arguments {
    time_t now;

    // create a barrier object with a count of 3
    pthread_barrier_init(&barrier, NULL, 3);

    // start up two threads, thread1 and thread2
    pthread_create(NULL, NULL, thread1, NULL);
    pthread_create(NULL, NULL, thread2, NULL);

    time (&now);
    printf("main() before barrier at %s", ctime (&now));

    pthread_barrier_wait(&barrier);

    // Now all three threads have completed.
    time (&now);
    printf("main() after barrier at %s", ctime (&now));

    time (&now);
    printf("T2: after barrier at %s", ctime (&now));

    pthread_exit(NULL);
    return (EXIT_SUCCESS);
}

void * thread1 (void *not_used) {
    time (&now);
    printf("T1 starting %s", ctime (&now));
    sleep (20);
    pthread_barrier_wait(&barrier);
    time (&now);
    printf("T1: after barrier at %s", ctime (&now));
}

void * thread2 (void *not_used) {
    time (&now);
    printf("T1 starting %s", ctime (&now));
    sleep (20);
    pthread_barrier_wait(&barrier);
    time (&now);
    printf("T1: after barrier at %s", ctime (&now));

    time (&now);
    printf("main() before barrier at %s", ctime (&now));
    pthread_barrier_wait(&barrier);

    // Now all three threads have completed.
    time (&now);
    printf("main() after barrier at %s", ctime (&now));

    time (&now);
    printf("T2: after barrier at %s", ctime (&now));
}
Summary

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section
- Synchronization mechanism
  - Hardware supported instructions
  - Software solution: semaphore
- Classical Synchronization Problems
- High-level synchronization structure: Monitor
- Case study: Pthread vs. Java monitor