Concurrent Programming

Dr. José Luis Zechinelli Martini

joseluis.zechinelli@udlap.mx

LDS-1101

Based on the course of Professor Carlo A. Furia, Chalmers University of Technology – University of Gothenburg

Agenda

- Concurrent programs
- Races
- Synchronization problems:
  - Mutual exclusion
  - Deadlock
  - Starvation and fairness
- Locks

Abstraction of concurrent programs

- We use an abstract notation for multithreaded applications using Java syntax:

```
int counter = 0;
```

thread t	thread u
`int cnt;`	`int cnt;`
`cnt = counter;`	`cnt = counter;`
`counter = cnt + 1;`	`counter = cnt + 1;`

- Each line of code includes exactly one instruction that can be executed atomically

Traces (1)

- A sequence of states gives an execution trace of the concurrent program:

State	t's local	u's local	Shared
1	pc_t: 1, cnt : ⊥	pc_u: 1, cnt : ⊥	counter : 0
2	pc_t: 2, cnt : 0	pc_u: 1, cnt : ⊥	counter : 0
3	done	pc_u: 1, cnt : ⊥	counter : 1
4	done	pc_u: 2, cnt : 1	counter : 1
5	done	done	counter : 2

- The program counter pc points to the atomic instruction that will be executed next

Traces (2)

- A sequence of states gives an execution trace of the concurrent program:

State	t's local	u's local	Shared
1	pc_t: 1, cnt : ⊥	pc_u: 1, cnt : ⊥	counter : 0
2	pc_t: 1, cnt : ⊥	pc_u: 2, cnt : 0	counter : 0
3	pc_t: 1, cnt : ⊥	done	counter : 1
4	pc_t: 2, cnt : 1	done	counter : 1
5	done	done	counter : 2

- The program counter pc points to the atomic instruction that will be executed next

Traces (3)

- A sequence of states gives an execution trace of the concurrent program:

State	t's local	u's local	Shared
1	pc_t: 1, cnt : ⊥	pc_u: 1, cnt : ⊥	counter : 0
2	pc_t: 2, cnt : 0	pc_u: 1, cnt : ⊥	counter : 0
3	pc_t: 2, cnt : 0	pc_u: 2, cnt : 0	counter : 0
4	pc_t: 2, cnt : 0	done	counter : 1
5	done	done	counter : 1

- The program counter pc points to the atomic instruction that will be executed next

Agenda

- ✓ Concurrent programs
- Races
- Synchronization problems:
  - Mutual exclusion
  - Deadlock
  - Starvation and fairness
- Locks

Race conditions

- Concurrent programs are **nondeterministic**:
  - Executing multiple times the same concurrent program with the same inputs may lead to different execution traces
  - This is a result of the nondeterministic interleaving of each thread's trace to determine the overall program trace
  - In turn, the interleaving is a result of the scheduler's decisions
- A **race condition** is a situation where the result of a concurrent program depends on the specific execution:
  - The concurrent counter example has a race condition: in some executions the final value of counter is 2, in some executions the final value of counter is 1
  - Race conditions can greatly complicate debugging!

Data races

- Race conditions are typically due lack of synchronization between threads that access shared memory
- A **data race** occurs when two threads access a shared memory location and:
  - At least one access is a write
  - The relative order of the two accesses is not fixed

```
int counter = 0;
```

thread t	thread u
`int cnt;`	`int cnt;`
`cnt = counter;`	`cnt = counter;`
`counter = cnt + 1;`	`counter = cnt + 1;`

Data races vs. race conditions

- Not every race condition is a data race:
  - Race conditions can occur even when there is no shared memory access
  - For example in filesystems or network access
- Not every data race is a race condition:
  - The data race may not affect the result
  - For example if two threads write the same value to shared memory

Agenda

- ✓ Concurrent programs
- ✓ Races
- Synchronization problems:
  - Mutual exclusion
  - Deadlock
  - Starvation and fairness
- Locks

Push out the races, bring in the speed

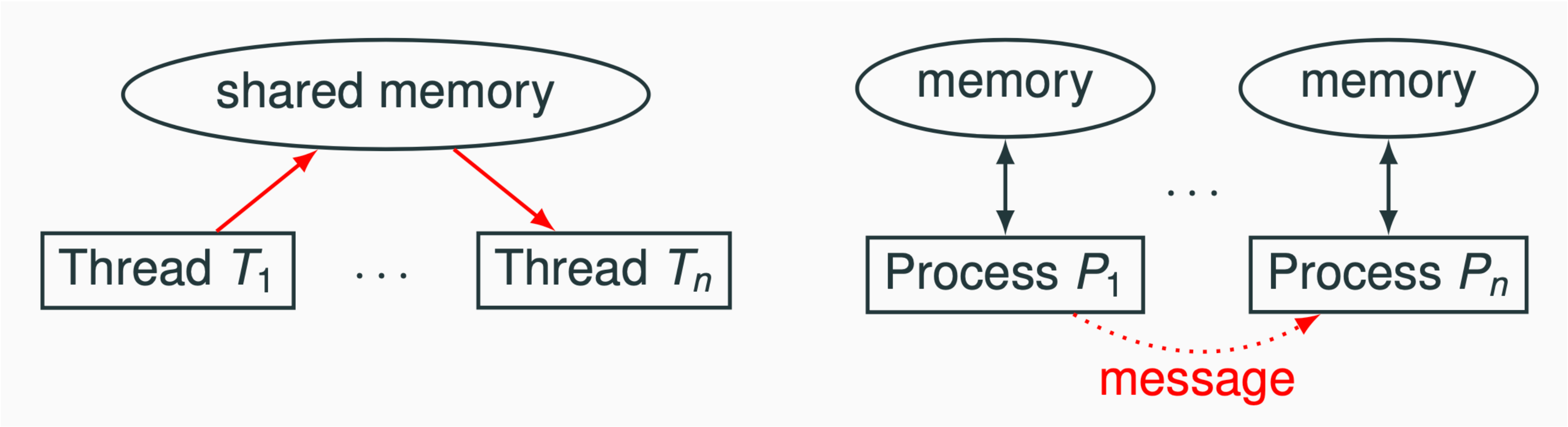
- Concurrent programming introduces:
  - The potential for parallel execution (faster, better resource usage)
  - The risk of race conditions (incorrect, unpredictable computations)
- The main challenge of concurrent programming is thus introducing parallelism without introducing race conditions:
  - This requires to restrict the amount of nondeterminism by synchronizing processes/threads that access shared resources

Synchronization

- We will study several synchronization problems that often appear in concurrent programming, together with their solutions:
  - **Correctness** (that is, avoiding race conditions) is more important than performance: an incorrect result that is computed very quickly is no good!
  - However, we also want to retain as much **concurrency as possible**, otherwise we might as well stick with sequential programming

Share memory vs. message passing synchronization (1)

- **Shared memory synchronization**:
  - Synchronize by writing to and reading from shared memory
  - Natural choice in shared memory systems such as threads
- **Message passing synchronization**:
  - Synchronize by exchanging messages
  - Natural choice in distributed memory systems such as processes



Share memory vs. message passing synchronization (2)

- The two synchronization models **overlap**:
  - send a message by writing to and reading from shared memory (example: message board)
  - share information by sending a message (example: order a billboard)

- However, in the first part of the course we will focus on synchronization problems that arise in shared memory concurrency; in the second part we will switch to message passing

The mutual exclusion problem (1)

- The mutual exclusion problem is a fundamental synchronization problem, which arises whenever multiple threads have access to a shared resource:
  - **Critical section**: the part of a program that accesses the shared resource (for example, a shared variable)
  - **Mutual exclusion property**: no more than one thread is in its critical section at any given time

- The mutual exclusion **problem** devise a protocol for accessing a shared resource that satisfies the mutual exclusion property

The mutual exclusion problem (2)

- Simplifications to present solutions in a uniform way:
  - The critical section is an arbitrary **block** of code
  - Threads **continuously** try to enter the critical section
  - Threads spend a **finite amount of time** in the critical section
  - We **ignore what the threads do outside** their critical sections

The mutual exclusion problem (3)

- The **mutual exclusion** problem devise a protocol for accessing a shared resource that satisfies the mutual exclusion property:

T shared;

---

thread t_j

```
// continuously
while (true) {
    entry protocol
    critical section {
        // access shared data
    }
    exit protocol
} /* ignore behavior
outside critical section */
```

thread t_k

```
// continuously
while (true) {
    entry protocol
    critical section {
        // access shared data
    }
    exit protocol
} /* ignore behavior
outside critical section */
```

The mutual exclusion problem example: Concurrent counter

- Updating a shared variable consistently is an instance of the mutual exclusion problem:

T shared;

---

thread t

```
int cnt;
while (true) {
    entry protocol
    critical section {
        cnt = counter;
        counter = cnt + 1;
    }
    exit protocol
    return;
}
```

thread u

```
int cnt;
while (true) {
    entry protocol
    critical section {
        cnt = counter;
        counter = cnt + 1;
    }
    exit protocol
    return;
}
```

What's a good solution to the mutual exclusion problem?

- A fully satisfactory solution is one that achieves three properties:
  - **Mutual exclusion**: At most one thread is in its critical section at any given time
  - **Freedom from deadlock**: If some threads tries to enter the critical section, some thread will eventually succeed
  - **Freedom from starvation**: Every thread that tries to enter the critical section will eventually succeed

- A good solution should also work for an arbitrary number of threads sharing the same memory

- (Note that freedom from starvation implies freedom from deadlock)

Agenda

- ✓ Concurrent programs
- ✓ Races
- Synchronization problems:
  - ✓ Mutual exclusion
  - Deadlock
  - Starvation and fairness
- Locks

Deadlocks

- A mutual exclusion protocol provides **exclusive access** to shared resources to one thread at a time
- Threads that try to access the resource when it is not available will have to block and **wait**
- Mutually dependent waiting conditions may introduce a **deadlock**

➢ A deadlock is the situation where a group of threads wait forever because each of them is waiting for resources that are held by another thread in the group (circular waiting)

Deadlock: Example

- A **deadlock** is the situation where a group of threads wait forever because each of them is waiting for resources that are held by another thread in the group (circular waiting)
  - A protocol that achieves mutual exclusion but introduces a deadlock
  - **Entry protocol**: wait until all other threads have executed their critical section

Via, resti servita Madama brillante
– E. Tommasi Ferroni, 2012

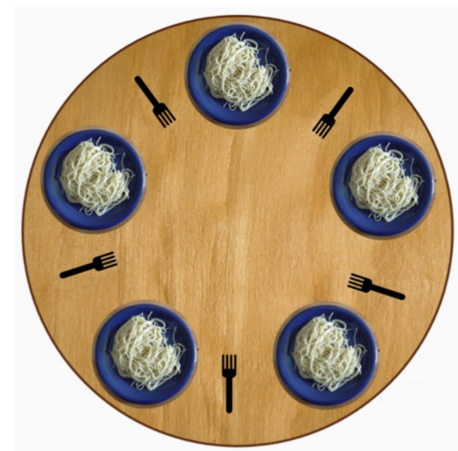
The dining philosophers

- The dining philosophers is a classic synchronization problem introduced by Dijkstra: It illustrates the problem of deadlocks using a colorful metaphor (by Tony Hoare):
  - Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers
  - Each philosopher alternates between thinking (**non-critical section**) and eating (**critical section**)
  - In order **to eat**, a philosopher needs to pick up the **two forks** that lie at the philopher's left and right sides
  - Since the forks are **shared**, there is a **synchronization** problem between philosophers (threads)

Deadlocking philosophers

- An unsuccessful attempt at solving the dining philosophers problem:

```
entry protocol ( Pk ) {
    left_fork.acquire();     // pick up left fork
    right_fork.acquire();    // pick up right fork
}
critical section { eat(); }
exit protocol ( Pk ) {
    left_fork.release();     // release left fork
    right_fork.release();    // release right fork
}
```



- This protocol **deadlocks** if all philosophers get their left forks, and wait forever for their right forks to become available

The Coffman conditions

- Necessary conditions for a deadlock to occur:
  - **Mutual exclusion**: Threads may have exclusive access to the shared resources
  - **Hold and wait**: A thread that may request one resource while holding another resource
  - **No preemption**: Resources cannot forcibly be released from threads that hold them
  - **Circular wait**: Two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding

- Avoiding deadlocks requires to **break one or more** of these conditions

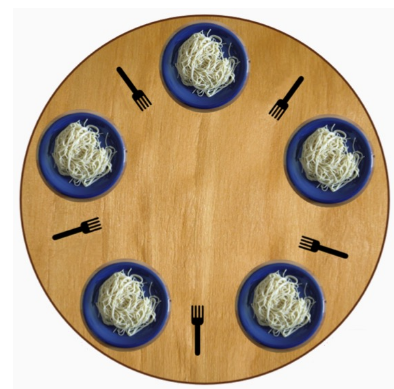
Agenda

- ✓ Concurrent programs
- ✓ Races
- Synchronization problems:
  - ✓ Mutual exclusion
  - ✓ Deadlock
  - ☐ Starvation and fairness
- Locks

Starving philosophers

- A solution to the dining philosophers problem that **avoids deadlock** by breaking hold and wait (and thus circular wait): Pick up both forks at once (atomic operation)

```
entry protocol ( Pk ) {
        forks.acquire();    // pick up left and right fork,
                            // atomically
}
critical section { eat(); }
exit protocol ( Pk ) {
        forks.release();    // release left and right fork,
                            // atomically
}
```



This protocol avoids deadlocks, but it may introduce starvation: a philosopher may never get a chance to pick up the forks

Starvation

- No deadlocks means that the system makes progress as a whole
- However, some individual thread may still make no progress because it is treated unfairly in terms of access to shared resources

- **Starvation** is the situation where a thread is perpetually denied access to a resource it requests
- **Avoiding starvation** requires some assumption on the scheduler, which has to “give a chance to every thread to execute”

Fairness

- **Weak fairness**:
  - If a thread continuously requests (that is, requests without interruptions) access to a resource, then access is granted infinitely often

- **Strong fairness**:
  - If a thread requests access to a resource infinitely often, then access is granted infinitely often

- Applied to a scheduler:
  - Request = A thread is ready (enabled)
  - Fairness = Every thread has a chance to execute

Breaking a circular wait

- Another solution is to pick up first the fork with the lowest id number: This avoids the circular wait because not every philosopher will pick up their left fork first

```
entry protocol ( Pk ) {
    if (left_fork.id() < right_fork.id()) {
        left_fork.acquire();
        right_fork.acquire();
    } else {
        right_fork.acquire();
        left_fork.acquire();
    }
}
critical section { eat(); }
exit protocol ( Pk ) { /* ... */ }
```

Ordering shared resources and forcing all threads to acquire the resources in order is a common measure to avoid deadlocks

Sequential philosophers

- Another solution that avoids deadlock as well as starvation: a (fair) waiter decides which philosopher eats; the waiter gives permission to eat to one philosopher at a time

```
entry protocol ( Pk ) {
    while (!waiter.can_eat(k)) {
        // wait for permission to eat
    }
    left_fork.acquire();
    right_fork.acquire();
}
critical section { eat(); }
exit protocol ( Pk ) { /* ... */ }
```

Having a centralized arbiter avoids deadlocks and starvation, but a waiter who only gives permission to one philosopher a time basically reduces the philosophers to following a sequential order without active concurrency

Agenda

- ✓ Concurrent programs
- ✓ Races
- ✓ Synchronization problems:
  - ✓ Mutual exclusion
  - ✓ Deadlock
  - ✓ Starvation and fairness
- Locks

Lock objects

- A lock is a data structure with interface:

```
interface Lock {
    void lock();        // acquire lock
    void unlock();      // release lock
}
```

  - Several threads share the same object lock of type Lock
  - Multiple threads calling lock.lock() results in exactly one thread T acquiring the lock:
    - T's call lock.lock() returns: T is holding the lock
    - Other threads block on the call lock.lock(), waiting for the lock to
  - Become available
  - A thread T that is holding the lock calls lock.unlock() to release the lock:
    - T's call lock.unlock() returns; the lock becomes available
    - Another thread waiting for the lock may succeed in acquiring it

- Locks are also called **mutexes**; they guarantee mutual exclusion

Using locks

- With lock objects the entry/exit protocols are trivial:
  - **Entry protocol**: call lock.lock()
  - **Exit protocol**: call lock.unlock()

```
int counter = 0;     Lock lock = new Lock();
```

thread t	thread u

`int cnt;`
`lock.lock();`
`  cnt = counter;`
`  counter = cnt + 1;`
`lock.unlock();`
	`int cnt;`
	`lock.lock();`
	`  cnt = counter;`
	`  counter = cnt + 1;`
	`lock.unlock();`

- The implementation of the Lock interface should guarantee mutual exclusion, deadlock freedom, and starvation freedom

Using locks in Java

```
// package with lock-related classes
import java.util.concurrent.locks.*;

// shared with other synchronizing threads
Lock lock;

while(true) {
        lock.lock();   // entry protocol
        try {
                // critical section
                // mutual exclusion is guaranteed
                // by the lock protocol
        } finally {      // lock released even if
                         // an exception is thrown in the critical section
                lock.unlock();     // exit protocol
        }
}
```

Counter with mutual exclusion

```
public class LockedCounter extends CCounter
{
	@Override
	public void run() {
		lock.lock();
		try {
			// int cnt = counter;
			// counter = counter + 1;
			super.run();
		} finally {
			lock.unlock();
		}
	}
	// shared by all threads working on this object
	private Lock lock = new ReentrantLock();
}
```

Built-in locks in Java

- Every object in Java has an implicit lock, which can be accessed using the keyword synchronized:
  - Whole method locking (synchronized methods):

```
synchronized T m() {
        // the critical section
        // is the whole method
        // body
}
```

  - Every call to m implicitly:
    - Acquires the lock
    - Executes m
    - Releases the lock
  - Block locking (synchronized block):

```
synchronized (this) {
        // the critical section
        // is the block's content
}
```

  - Every execution of the block implicitly:
    - Acquires the lock
    - Executes m
    - Releases the lock

Counter with mutual exclusion: with synchronized

- SyncCounter class:

```
public class SyncCounter
        extends CCounter
{
    @Override
    public synchronized
    void run() {
        // int cnt = counter;
        // counter = counter + 1;
        super.run();
    }
}
```

- SyncBlockCounter class:

```
public class SyncBlockCounter
        extends CCounter
{
        @Override
        public void run() {
            synchronized(this) {
                // int cnt = counter;
                // counter = counter + 1;
                super.run();
            }
        }
}
```

Lock implementations in Java

- The most common implementation of the Lock interface in Java is class ReentrantLock
  - **Mutual exclusion**:
    - ReentrantLock guarantees mutual exclusion
  - **Starvation**:
    - ReentrantLock does not guarantee freedom from starvation by default
    - However, calling the constructor with new ReentrantLock(true) “favors granting access to the longest-waiting thread”
    - This still does not guarantee that thread scheduling is fair
  - **Deadlocks**:
    - One thread will succeed in acquiring the lock
    - However, deadlocks may occur in systems that use multiple locks (remember the dining philosophers)

Built-in lock implementations in Java

- The built-in locks — used by synchronized methods and blocks — have the same behavior as the explicit locks of java.util.concurrent.locks
- Built-in locks, as well as all lock implementations in java.util.concurrent.locks, are re-entrant: A thread holding a lock can lock it again without causing a deadlock