Vika 8

TÖL401G - Stýrikerfi

1. Illustrate a race condition and describe the critical-section problem.

- A race condition occurs when two or more threads access shared data concurrently, and at least one of them modifies the data, causing unexpected behavior due to the unpredictable order of execution.
- The critical-section problem refers to the challenge of ensuring that when a thread is executing in its critical section (where shared data is accessed), no other thread can enter its critical section. This helps prevent race conditions and maintain data consistency.

2. Present software, hardware and higher-level solutions to the critical-section problem:

- Software: Peterson's algorithm,
- Hardware: Atomic instructions, including spinlocks,
- Higher-level: Semaphores, monitors and condition variables, message passing.
- Java API for semaphores and monitors with conditions.

2.1. Peterson's Algorithm,

The Peterson algorithm is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981. The algorithm uses two variables, flag and turn, to indicate whether a process is ready to enter the critical section or not.

2.2. Atomic instructions, including spinlocks,

An operation or instruction (or a region of several instructions) is atomic, if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no apparent effect.

2.3. Semaphores,

- A semaphore can be considerd as a special type of variable(or as a special class in case of object-orientation). it consists of:
  - A value,
  - A queue of threads waiting for the value to be greater than zero.
  - Operations that can be applied to it:
    - P (proberen) or wait: decrement the value, and if it is negative, block the thread until it becomes positive again,
    - V (verhogen) or signal: increment the value, and if it was negative, unblock one of the threads waiting for the semaphore.
    - Initialize: set the value to a given number.
    - Wait: decrement the value, and if it is negative, block the thread until it becomes positive again,

Mutual exclution:

```
sem_condition.init(1);

// Process A:
sem_condition.wait();
//<critical section>
sem_condition.signal();

// Process B:
sem_condition.wait();
//<critical section>
sem_condition.signal();
```

2.4. Monitors and condition variables,

- Monitors:
  - A monitor is a programming construct that enforces mutual exclusion by allowing only one thread to execute within its critical section at a time. It is an object that encapsulates both data (shared resources) and the methods (functions) that operate on that data. Monitors ensure that the methods are executed atomically, preventing race conditions.
- Condition variables:
  - Condition variables are used in conjunction with monitors to manage threads that must wait for specific conditions to be met before they can proceed. They allow threads to wait for a certain condition within the monitor and be notified when the condition is met. Condition variables provide an efficient way to manage threads that need to wait and resume execution based on specific conditions.

2.5. Message passing.

Message passing is an alternative approach to shared memory for exchanging data between threads or processes. Instead of using shared memory regions, which require synchronization mechanisms like semaphores, monitors, or condition variables, message passing relies on explicit communication via messages sent between the cooperating entities.

Message passing systems can be implemented using various communication channels like pipes, sockets, message queues, or higher-level APIs provided by programming languages or libraries. These systems can be synchronous (blocking) or asynchronous (non-blocking) based on how communication is managed.

- Encapsulation:
  - Message passing promotes the encapsulation of data, as threads or processes do not directly access shared memory. Instead, they communicate by exchanging messages containing the required data.
- Synchronization:
  - With message passing, synchronization is often implicit, as sending and receiving messages may involve blocking or non-blocking behavior, depending on the implementation. When a sender is blocked until the receiver accepts the message, it enforces a natural synchronization point.
- Scalability:
  - Message passing can be more scalable than shared memory, especially in distributed systems, as it does not rely on a single shared memory region. This makes it more suitable for communication across different systems or networks.
- Ease of reasoning:

- Since message passing avoids direct manipulation of shared memory, it can be easier to reason about the correctness and safety of concurrent programs. However, it may require more explicit communication and handling of message passing events.

2.6. Java API for semaphores and monitors with conditions.

```
import java.util.concurrent.Semaphore;

try {
    semaphore.acquire();
    // start of critical section
    // ...
    // end of critical section
    sem.release();
} catch (InterruptedException e) {
  // i.e. someone called interrupt() while we were waiting in
  // sem.acquire() call
}
```

- Monitor construct is based on the encaptulation of data within an object, and the use of the synchronized keyword to ensure that only one thread can access the object at a time. A methot can be declared as synchronized, or a block of code can be synchronized. And if a thread is executing a synchronized method or block, no other thread can execute any other synchronized method or block on the same object.

3. Present classical synchronisation problems.

- Bounded-buffer problem,
  - same as producer-consumer problem,
- Readers-writers problem,
  - There are at least three variations of the problems, which deal with situations in which many concurrent threads of execution try to access the same shared resource at one time.Some threads may read and some may write, with the constraint that no thread may access the shared resource for either reading or writing while another thread is in the act of writing to it. (In particular, we want to prevent more than one thread modifying the shared resource simultaneously and allow for two or more readers to access the shared resource at the same time).
- Dining philosophers problem.
  - Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks.
    - Deadlocks:
      - Imagine each philosopher picks up their left fork. Then each philosopher will wait forever for their right fork. This is a deadlock.
    - Starvation:
      - Imagine that each philosopher always picks up their left fork first. Then each philosopher will wait forever for their right fork. This is starvation.

Solution (not 100% correct but enough to get the feeling for how it is done):

```
class Philosopher extends Thread {
    private final Semaphore leftFork;
    private final Semaphore rightFork;

    Philosopher(Semaphore leftFork, Semaphore rightFork) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    public void run() {
        while (true) {
            think();
            pickUpForks();
            eat();
            putDownForks();
        }
    }

    private void think() {
        // Philosopher is thinking
    }

    private void pickUpForks() {
        try {
            leftFork.acquire();
            rightFork.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void eat() {
        // Philosopher is eating
    }

    private void putDownForks() {
        leftFork.release();
        rightFork.release();
    }
}

public class DiningPhilosophers {
    public static void main(String[] args) {
        int numberOfPhilosophers = 5;
        Semaphore[] forks = new Semaphore[numberOfPhilosophers];
        Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];

        for (int i = 0; i < numberOfPhilosophers; i++) {
            forks[i] = new Semaphore(1); // Each fork is a semaphore with 1 permit
        }

        for (int i = 0; i < numberOfPhilosophers; i++) {
            Semaphore leftFork = forks[i];
            Semaphore rightFork = forks[(i + 1) % numberOfPhilosophers];
            philosophers[i] = new Philosopher(leftFork, rightFork);
            philosophers[i].start();
        }
    }
}
```

- Sleeping barber problem.
  - Imagine a hypothetical barbershop with one barber, one barber chair, and a waiting room with n chairs (n may be 0) for waiting customers. The following rules apply:
    - If there are no customers, the barber falls asleep in the chair
    - If a customer arrives and the barber is asleep, the customer wakes up the barber.

- When a customer arrives while the barber is cutting someone else's hair,he sit down in one of the chairs in the waiting room.
- If there are no empty chairs, the customer leaves.
- When the barber finishes cutting a customer's hair, they dismiss the customer and return to the barber chair to sleep if there are no other customers waiting.

```
Semaphore ME = new Semaphore(1); // Mutex for the waiting room
Semaphore barberSleep = new Semaphore(0); // Initially asleep
Semaphore barberChair = new Semaphore(0); // Mutex for the barber chair
int numberOfFreeWaitRoomSeats = N; // Number of free seats in the waiting room

// Barber:
void Barber (){
    while (true) {
        barberSleep.wait(); // Try to sleep
        ME.wait(); // Enter the waiting room
        numberOfFreeSeats++; // One chair becomes free
        barberChair.signal(); // Invite customer into the chair
        cutHairOfCustomerOnChair(); // Cut hair
        ME.signal(); // Release the waiting room
    }
}

// Customer:
void Customer(){
    ME.wait(); // Enter the waiting room
    if (numberOfFreeWaitRoomSeats > 0) {
        numberOfFreeWaitRoomSeats--; // Occupy a chair
        barberSleep.signal(); // Wake up the barber if needed
        ME.signal(); // Release the waiting room
        barberChair.wait(); // Wait until invited
        goToBarberChairGetHairCutLeave(); // Get haircut
    } else {
        ME.signal(); // Release the waiting room
        leaveWithoutHaircut(); // No free chairs
    }
}
```