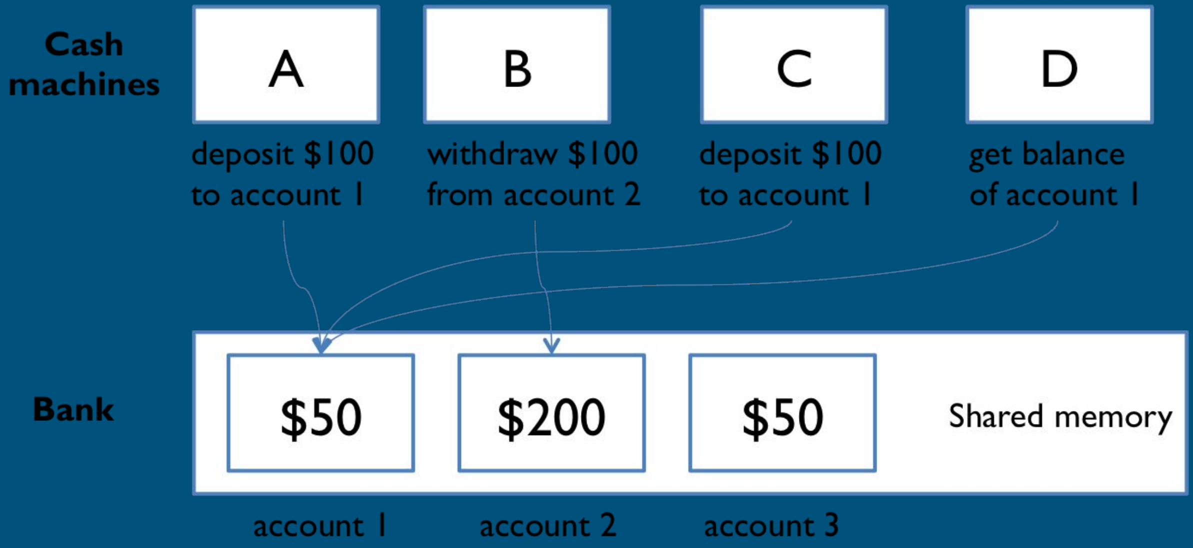
Locks

Operating Systems

What is the Problem?

The bank has several cash machines, all of which can read and write the same account objects in memory.

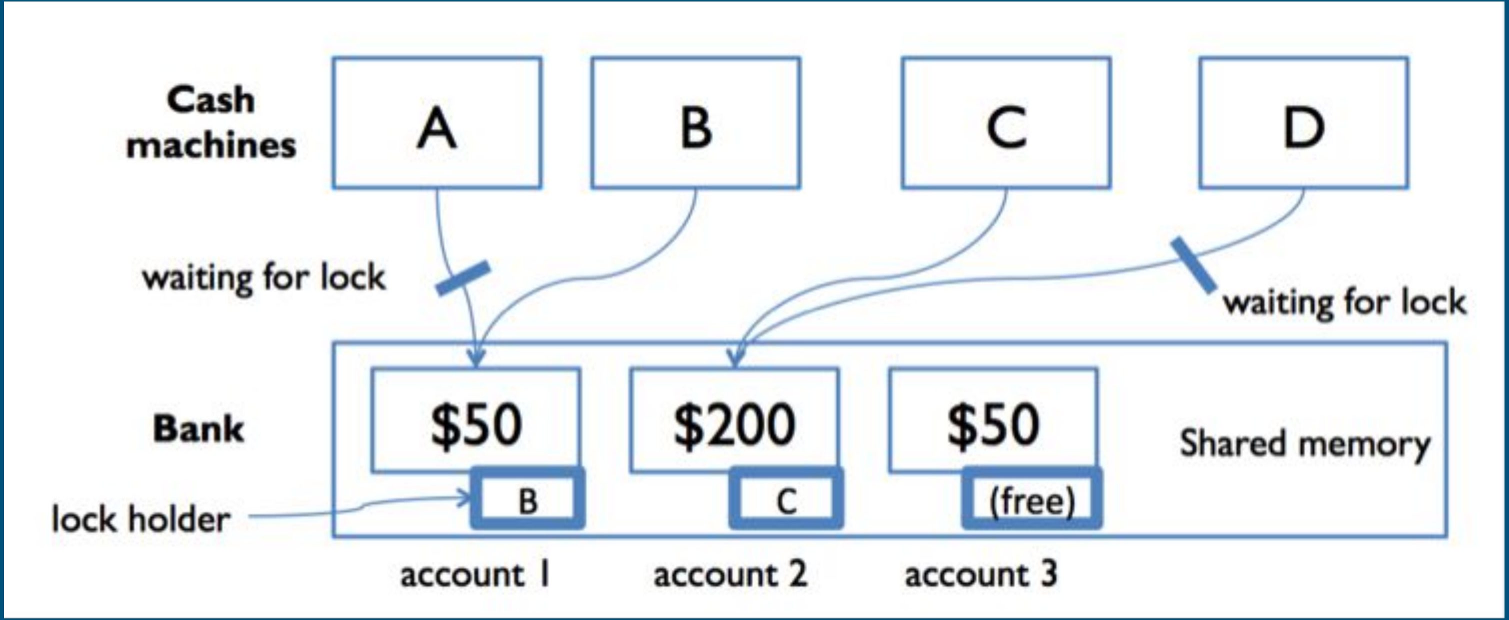


Solution

To solve this problem with locks, we can add a lock that protects each bank account. Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.

In the diagram to the right, both A and B are trying to access account 1. Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

Solution(Continue)



Locks

The same concept applies to the threads. Imagine multiple threads will increase a certain data type(named X) in the given struct. Suppose X has an initial value of 0, so what happens when two threads try to increase the X?

1- X will be 2

2- X will be 1

Which one is possible?

Locks(Continue)

Both answers were correct. Since each thread act independently, it is hard to predict the final value of X. As you know, the correct answer is the first one, but we should use locks to enforce this. Locks allow us to control the critical sections in the source code.

Locks in C

The first type of lock we are going to look at is Mutex. Mutex is one of the popular synchronization methods used in most multithreaded software.

A Mutex is a lock we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of the code. So we see that even if thread 2 is scheduled while thread 1 is not done accessing the shared resource and the code is locked by thread 1 using mutexes, thread 2 cannot even access that region of code. So this ensures synchronized access to shared resources in the code.

Locks in C(Continue)

Here is the API's for the Mutex:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t
*restrict attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Locks in C(Continue)

Let's Rock some codes!!

Locks in C(Continue)

There is another type of lock in C, named semaphore. Actually, the mutex is a subset of semaphore, sometimes called binary semaphore, so what I mean is that with a mutex, you can only lock certain regions and allow only one thread to enter that particular section, so what if we want two or more threads to go in ceratin section but no more than that? This is where semaphores come to play. Although semaphore is usually used for protecting critical areas, this is not their only application(We will see that in a moment).

Locks in C(Continue)

Here is the API's for the Semaphores:

```
sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);

sem_destroy(sem_t *mutex);
```

Locks in C(Continue)

Let's Rock some codes!!

Dark side of Locks

Question:

Imagine we have a simple queue so that we can push or pop elements into the queue. Let’s say we already have 1000 elements inside this queue, so will 3 threads empty the queue faster or one thread?(Suppose we protect pop operation with lock).

Dark side of Locks

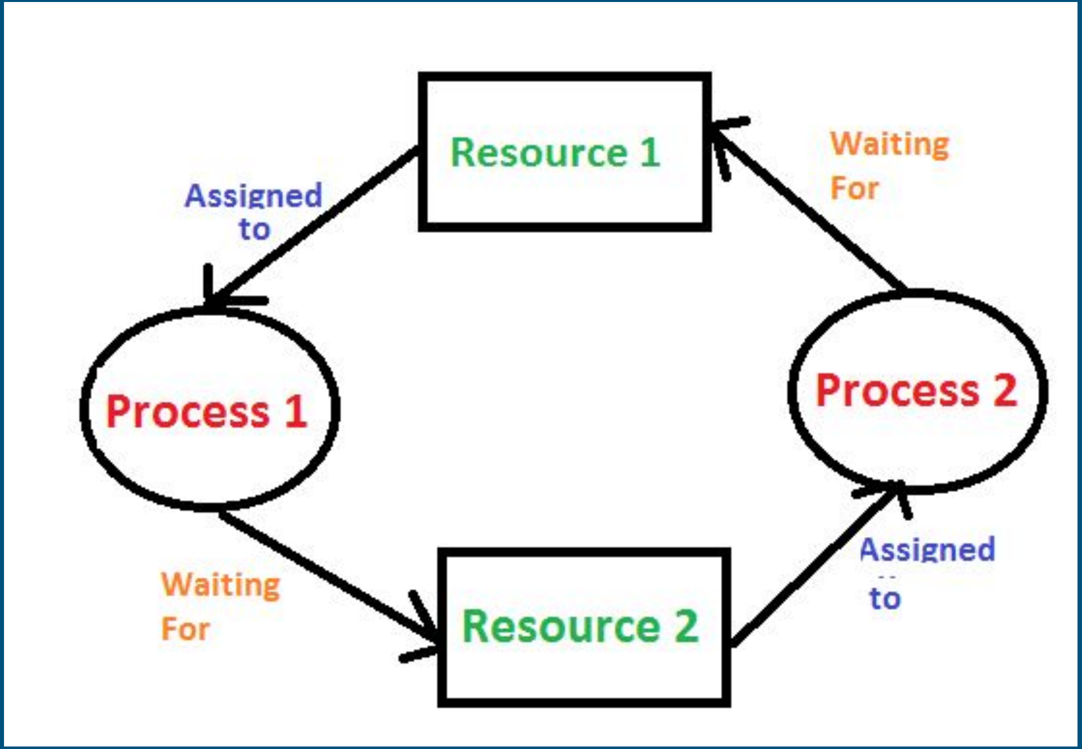
In multithreaded programming, the access time to shared data through multiple threads will never scale to one threaded version because locking mutex is so expensive that we miss so many cycles in the CPU. So the scale in multithreaded programs comes in those areas where threads work independently. Also, remember that debugging multithread software is still challenging because each thread acts independently. Therefore, we face non-deterministic situations, which is why most simulations are single-threaded and adopt an event-based approach for simulation.

Dark side of Locks(continue)

Question:

What is the problem of

the following picture?



Dark side of Locks(continue)

The previous picture illustrates the concept of deadlock. Deadlock is one of the important aspects of multithreaded software. This situation usually appears when two thread wants to acquiesce two locks, but one lock is at the hand of thread1, and the other is at the hand of thread2. Each one of them wants the other lock to continue its operation. Since neither one is willing to release the requisite lock and allow the other one to continue the operation, we face a deadlock where neither of them continues their work, so we are stuck in a dire situation.

Dark side of Locks(continue)

There are some algorithms for preventing or detecting these problems, and Dr.entezari will introduce some of them throughout the course. Still, one the most straightforward algorithm that uses in some database management systems is …?

Can you guess?

Dark side of Locks(continue)

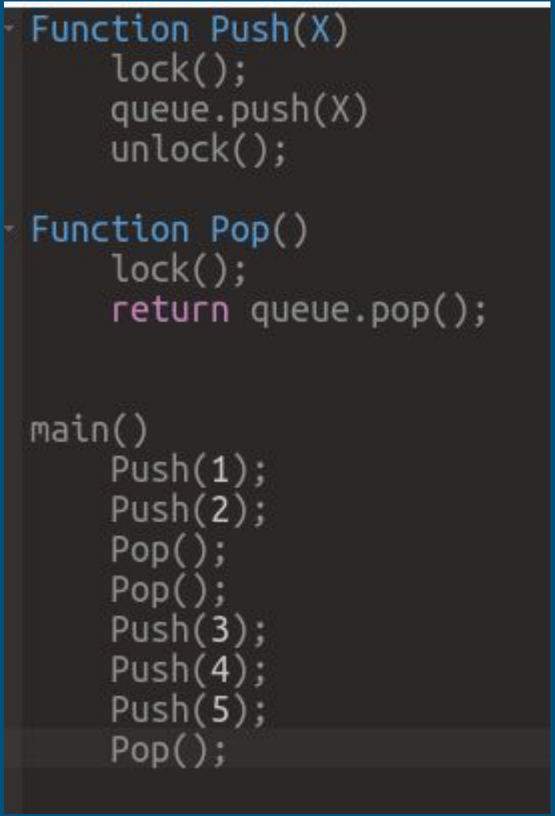
As you saw in the given picture, we can make a directed graph from process and resources, so if we find a cycle in this directed graph(with dfs or bfs), we can assume that we will have a deadlock. Therefore, we can prevent it.

Dark side of Locks(continue)

Let's Rock some codes!!

Dark side of Locks(continue)

Question: What is the problem in the following code?

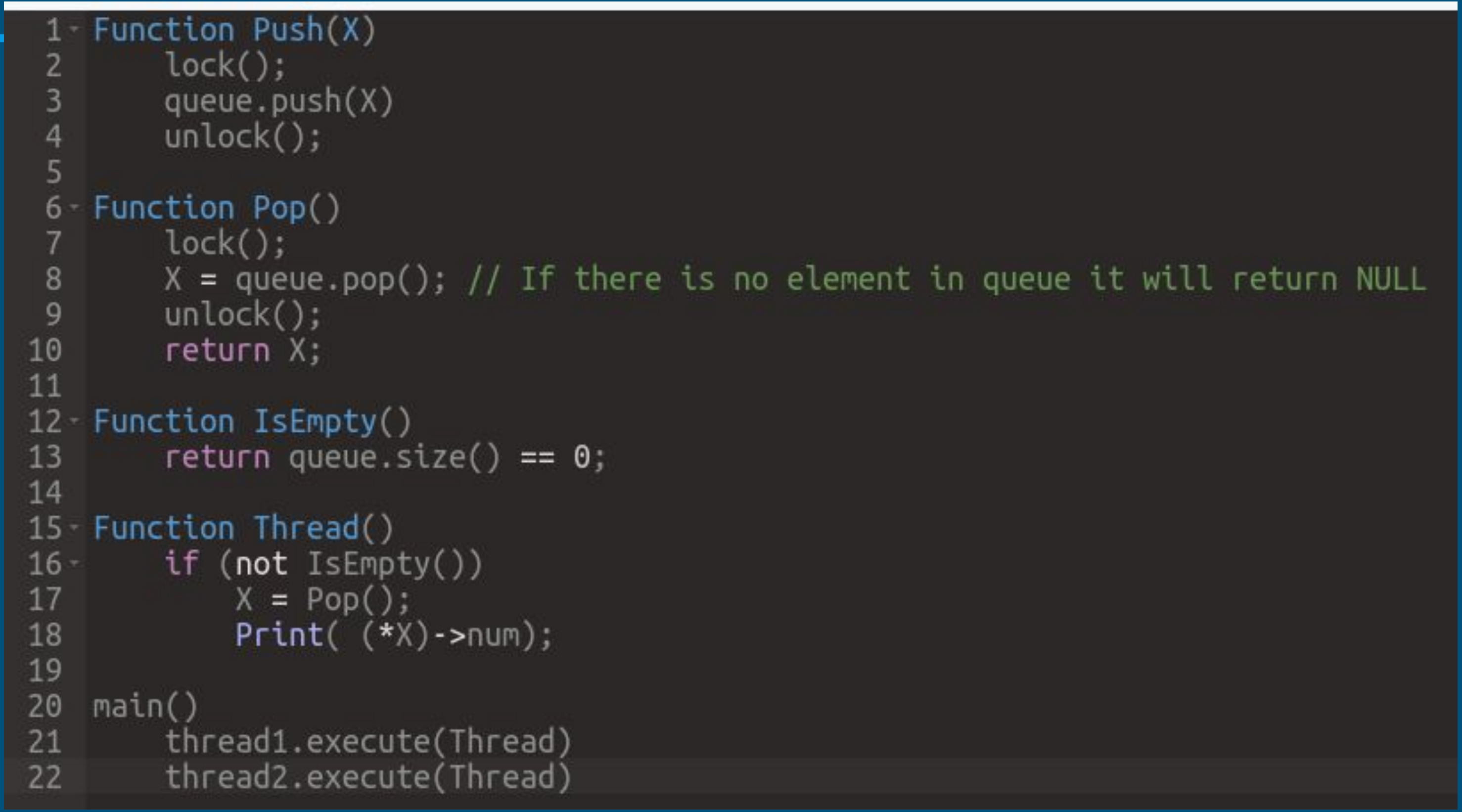


```
Function Push(X)
        lock();
        queue.push(X)
        unlock();

Function Pop()
        lock();
        return queue.pop();


main()
        Push(1);
        Push(2);
        Pop();
        Pop();
        Push(3);
        Push(4);
        Push(5);
        Pop();
```

Dark side of Locks(continue)

Question: What is the problem in the code?

Dark side of Locks(continue)



```
Function Push(X)
    lock();
    queue.push(X)
    unlock();

Function Pop()
    lock();
    X = queue.pop(); // If there is no element in queue it will return NULL
    unlock();
    return X;

Function IsEmpty()
    return queue.size() == 0;

Function Thread()
    if (not IsEmpty())
        X = Pop();
        Print( (*X)->num);

main()
    thread1.execute(Thread)
    thread2.execute(Thread)
```