

Synchronization I

Jo, Heeseung

Today's Topics

Synchronization problem

Locks

Synchronization

Threads cooperate in multithreaded programs

- To **share** resources, access shared data structures
- Also, to **coordinate** their execution

For correctness, **we have to control this cooperation**

- Must assume threads interleave executions arbitrarily and at different rates
 - **Scheduling is not under application writers' control**
- We control cooperation using **synchronization**
 - Enables us to restrict the interleaving of execution
- (Note) This also applies to processes, not just threads
 - And it also applies across machines in a distributed system

The Classic Example (1)

Withdraw money from a bank account

- Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000 won
- What happens if both go to separate ATM machines, and simultaneously withdraw 100,000 won from the account?



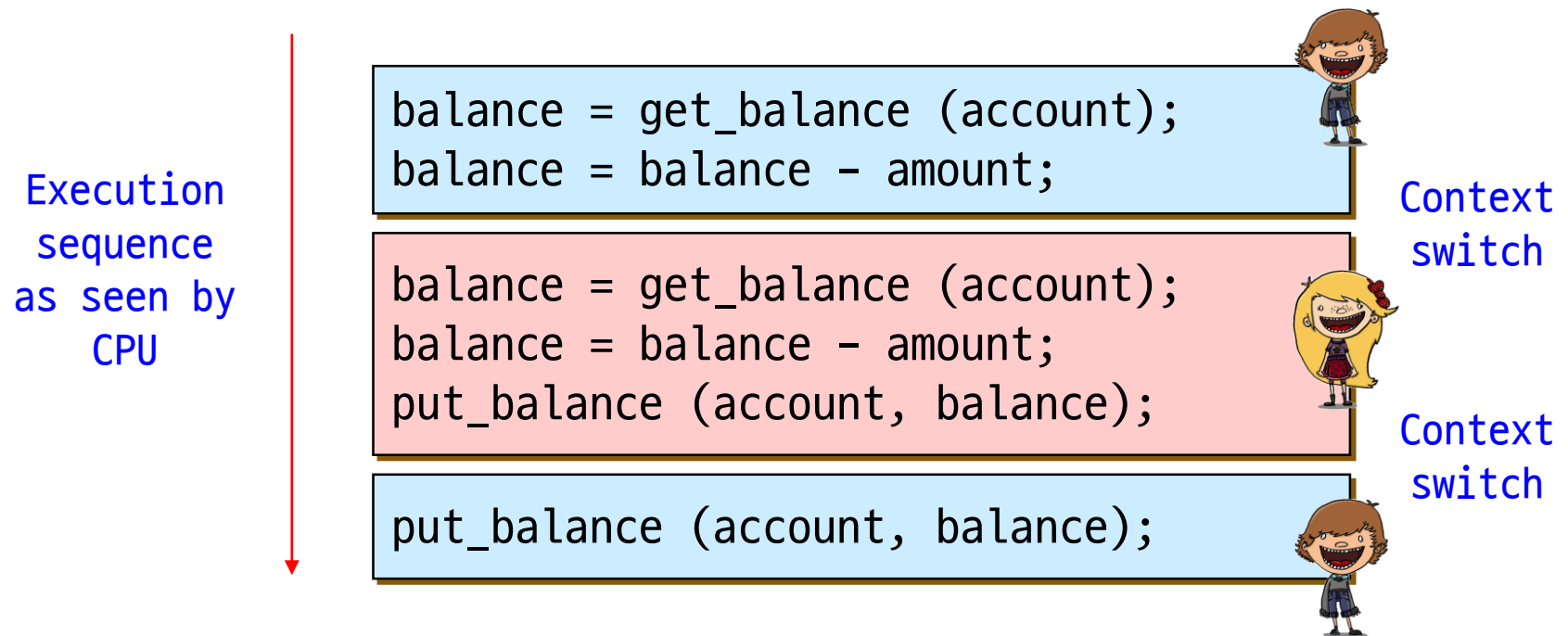
```
int withdraw (account, amount)
{
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    return balance;
}
```



The Classic Example (2)

Interleaved schedules

- Represent the situation by creating a separate thread for each person to do the withdrawals
- The execution of the two threads can be interleaved, assuming preemptive scheduling:



Synchronization Problem

Problem

- Two concurrent threads (or processes) access a **shared resource** without any synchronization
- Creates a **race condition**:
 - The situation where several processes access and manipulate shared data concurrently
 - The result is **non-deterministic** and depends on timing
- Need mechanisms for controlling access to shared resources
 - So that we can reason about the operation of programs
- **Synchronization** is necessary for any shared data structure
 - buffers, queues, lists, etc.

Threads using shared data

```
#include <pthread.h>
#define MAX_THREAD 20

void *threadcount(void *data) {
    int *count = (int *)data;
    int i;
    for (i=0; i<100; i++) {
        *count = *count+1;
    }
}

int main(int argc, char **argv) {
    pthread_t thread_id[MAX_THREAD];
    int i = 0;
    int count = 0;
    for(i = 0; i < MAX_THREAD; i++) {
        pthread_create(&thread_id[i], NULL, threadcount, (void *)&count);
    }
    for(i = 0; i < MAX_THREAD; i++) {
        pthread_join(thread_id[i], NULL);
    }
    printf("Main Thread : %d\n", count);
    return 0;
}
```

```
# gcc ex.c -lpthread
```

```
# ./a.out
```

```
Main Thread : 2000
```

```
# ./a.out
```

```
Main Thread : 1957
```

Threads using shared data

`count=count+1` could be implemented as

- `register1 = count` (LOAD R1, MEM_count)
- `register1 = register1 + 1` (ADD R1, R1, 1)
- `count = register1` (STORE R1, MEM_count)

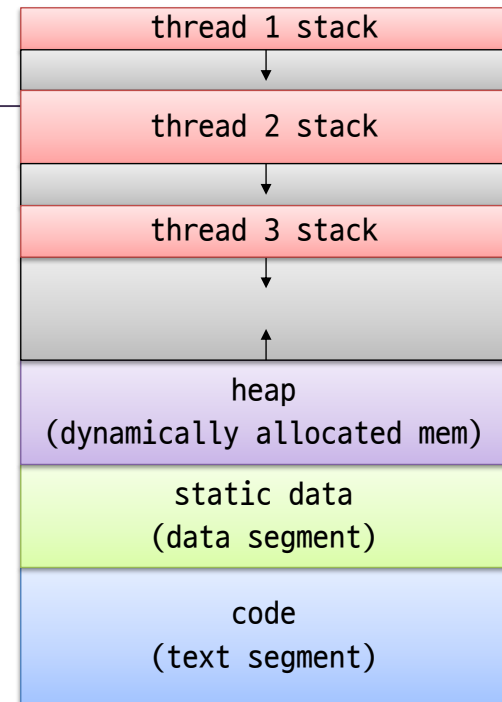
Consider this execution interleaving with "count = 5" initially:

1. T1: `register1 = count` {register1 = 5}
2. T1: `register1 = register1 + 1` {register1 = 6}
3. T2: `register2 = count` {register2 = 5}
4. T2: `register2 = register2 + 1` {register2 = 6}
5. T1: `count = register1` {count = 6}
6. T2: `count = register2` {count = 6}

Sharing Resources

Between threads

- Local variables are not shared
 - Refer to data on the stack
 - Each thread has its own stack
 - Never pass/share/store a pointer to a local variable on another thread's stack
- Global variables are shared
 - Stored in static data segment, accessible by any thread
- Dynamic objects are shared
 - Stored in the heap, shared through the pointers



Between processes

- Shared-memory objects, files, etc. are shared

Critical Sections (1)

Critical sections

- Parts of the program that access shared resources
 - Shared files, shared memory(variable), etc.
- Use **mutual exclusion** to synchronize access to shared resources in critical sections
 - Only one thread at a time can execute in the critical section
 - All other threads are forced to wait on entry
 - When a thread leaves a critical section, another can enter
- Otherwise, critical sections can lead to **race conditions**
 - The final result depends on the sequence of execution of the processes

```
int withdraw (account, amount)
{
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    return balance;
}
```

Critical Sections (2)

Requirements

- **Mutual exclusion**
 - At most one thread is in the critical section
- **Progress**
 - If thread T is inside the critical section, T must finish the critical section within reasonable time
 - If thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
- **Bounded waiting** (no starvation)
 - If thread T is waiting on the critical section, then T will eventually enter the critical section
- **Performance**
 - The overhead of entering and exiting the critical section is small with respect to the work being done within it

Critical Sections (3)

Mechanisms for building critical sections

- Locks
 - Very primitive, minimal semantics, used to build others
- Semaphores
 - Basic, easy to get the hang of, hard to program with
- Monitors
 - High-level, requires language support, implicit operations
 - Easy to program with
 - e.g. Java "synchronized"
- Messages
 - Simple model of communication and synchronization based on (atomic) transfer of data across a channel
 - Direct application to distributed systems

Locks

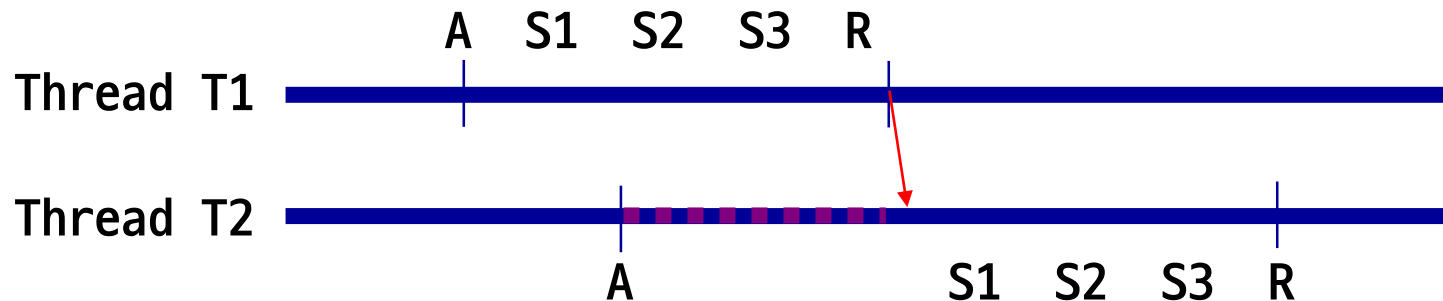
Locks

- A lock is an object (in memory) that provides the following two operations:
 - `acquire()`: wait until lock is free, then grab it
 - `release()`: unlock, and wake up any thread waiting in `acquire()`
- Using locks
 - Lock is initially free
 - Call `acquire()` before entering a critical section, and `release()` after leaving it
 - Between `acquire()` and `release()`, the thread holds the lock
 - `acquire()` does not return until the caller holds the lock
 - At most one thread can hold a lock at a time
- Locks can spin (a `spinlock`) or block (a `mutex`)

Using Locks

```
int withdraw (account, amount)
{
  A    acquire (&lock);
  S1   balance = get_balance (account);
  S2   balance = balance - amount;
  S3   put_balance (account, balance);
  R    release (&lock);
  return balance;
}
```

} Critical
section



Implementing Locks (1)

An initial attempt

```
struct lock { int held = 0; };

void acquire (struct lock *l) {
    while (l->held);
    l->held = 1;
}

void release (struct lock *l) {
    l->held = 0;
}
```

```
int withdraw (account, amount)
{
    acquire (&lock);
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    release (&lock);
    return balance;
}
```



The caller "busy-waits",
or spins for locks to be
released, hence
spinlocks

Implementing Locks (1)

An initial attempt

```
struct lock { int held = 0; }

void acquire (struct lock *l) {
    while (l->held);
    l->held = 1;
}

void release (struct lock *l) {
    l->held = 0;
}
```

```
void acquire (struct lock *l) {
    while (l->held);
    l->held = 1;
}

void release (struct lock *l) {
    l->held = 0;
}
```

```
int withdraw (account, amount)
{
    acquire (&lock);
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    release (&lock);
    return balance;
}
```

Implementing Locks (2)

Problem

- Implementation of locks has a critical section, too!
 - The acquire/release must be atomic
 - A recursion, huh?

Atomic operation

- Executes as though it could not be interrupted
- Code that executes "all or nothing"

Implementing Locks (3)

Solutions

- Software-only algorithms
 - Dekker's algorithm (1962)
 - Peterson's algorithm (1981)
 - Lamport's Bakery algorithm for more than two processes (1974)
- Hardware atomic instructions
 - Test-and-set, compare-and-swap, etc.
- Disable/reenable interrupts
 - To prevent context switches

Software-only Algorithms

Initial algorithm

- Mutual exclusion?
- Progress?
- Bounded waiting?

Thread 0

```
...  
acquire(0)  
do C.S.  
release(0)
```

Thread 1

```
...  
acquire(1)  
do C.S.  
release(1)
```

```
int interested[2];  
interested[0]=FALSE; interested[1]=FALSE;
```

```
void acquire (int process)  {  
    int other = 1 - process;  
    interested[process] = TRUE;  
    while (interested[other]);  
}
```

```
void release (int process)  {  
    interested[process] = FALSE;  
}
```

```
void acquire (int process)  {  
    int other = 1 - process;  
    interested[process] = TRUE;  
    while (interested[other]);  
}
```

```
void release (int process)  {  
    interested[process] = FALSE;  
}
```

Peterson's Algorithm

Solves the critical section problem
for two threads/processes

```
int turn;  
int interested[2];  
interested[0]=FALSE; interested[1]=FALSE;
```

```
void acquire (int process) {  
    int other = 1 - process;  
    interested[process] = TRUE;  
    turn = other;  
    while (interested[other] &&  
           turn == other);  
}
```

```
void release (int process) {  
    interested[process] = FALSE;  
}
```

Thread 0

```
...  
acquire(0)  
do C.S.  
release(0)
```

Thread 1

```
...  
acquire(1)  
do C.S.  
release(1)
```

```
void acquire (int process) {  
    int other = 1 - process;  
    interested[process] = TRUE;  
    turn = other;  
    while (interested[other] &&  
           turn == other);  
}
```

```
void release (int process) {  
    interested[process] = FALSE;  
}
```

Atomic Instructions (1)

Test-and-Set

- Mostly supported by H/W

```
// bts reg/mem, reg
int TestAndSet (int *v)  {
    int rv = *v;
    *v = 1;
    return rv;
}
```

} Atomic

Using Test-and-Set instruction

```
void struct lock { int value = 0; }

void acquire (struct lock *l)  {
    while (TestAndSet (&l->value));
}

void release (struct lock *l)  {
    l->value = 0;
}
```

Atomic Instructions (2)

Swap

- Mostly supported by H/W

```
// cmpxchg reg, reg/mem
void Swap (int *v1, int *v2)  {
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
```

} Atomic

Using Swap instruction

```
void struct lock { int value = 0; }
void acquire (struct lock *l)  {
    int key = 1;
    while (key == 1) Swap(&l->value, &key);
}
void release (struct lock *l)  {
    l->value = 0;
}
```

Problems with Spinlocks

Spinlocks

- Horribly wasteful!
 - CPU cycle is wasted
 - The longer the critical section, the longer the spin
 - Lock holder can be interrupted through involuntary context switch
- Only want to use spinlock as primitives to build higher-level synchronization constructs

Disabling Interrupts (1)

Implementing locks by disabling interrupts

```
void acquire (struct lock *l) {  
    cli();    // local_irq_disable(), disable interrupts;  
}  
void release (struct lock *l) {  
    sti();    // local_irq_enable(), enable interrupts;  
}
```

- Disabling interrupts
 - Blocks notification of external events
 - No context switch (e.g., timer)
- Can two threads disable interrupts simultaneously?

Disabling Interrupts (2)

What's wrong?

- Only available to kernel
 - If OS support these as system calls ?
- Insufficient on a multiprocessor
 - Back to atomic instructions
- What if the critical section is long?
 - Can miss or delay important events
(e.g., timer, I/O)

Like spinlocks, only use to implement higher-level synchronization primitives

Summary

Implementing locks

- Software-only algorithms
- Hardware atomic instructions
- Disable/reenable interrupts

Spinlocks and disabling interrupts are primitive synchronization mechanisms

- They are used to build higher-level synchronization constructs