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 1,000,000won.
- What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```c
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:

```plaintext
balance = get_balance (account);
balance = balance - amount;

balance = get_balance (account);
balance = balance - amount;
put_balance (account, balance);

put_balance (account, balance);
```
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.
  - We need mechanisms for controlling access to shared resources in the face of concurrency.
    - So that we can reason about the operation of programs.
  - Synchronization is necessary for any shared data structure
    - buffers, queues, lists, etc.
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
  - Critical sections are parts of the program that access shared memory or shared files or other shared resources.
  - We want to 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.
Critical Sections (2)

- **Requirements**
  - **Mutual exclusion**
    - At most one thread is in the critical section.
  - **Progress**
    - 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: 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

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

Critical section

Thread T1

Thread T2
Implementing Locks (1)

- An initial attempt

```c
struct lock { int held = 0; }
void acquire (struct lock *l) {
    while (l->held);
    l->held = 1;
}
void release (struct lock *l) {
    l->held = 0;
}
```

The caller “busy-waits”, or spins for locks to be released, hence spinlocks.

- Does this work?
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) (cf. Exercises)
    - 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

- Wrong algorithm
  - Mutual exclusion?
  - Progress?

```c
int interested[2];

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 processes

```c
int turn;
int interested[2];

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;
}
```
Bakery Algorithm (1)

**Multiple-process solution**

- Before entering its critical section, process receives a sequence number.
- Holder of the smallest number enters the critical section.
- If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e. $1, 2, 3, 3, 3, 4, 4, 5,...$
Bakery Algorithm (2)

```c
int number[N];
int choosing[N];

#define EARLIER(a,b)    ((number[a] < number[b]) || ((number[a] == number[b] && a < b))

int Findmax () {
    int i;
    int max = number[0];
    for (i = 1; i < N; i++)
        if (number[i] > max)
            max = number[i];
    return max;
}

void acquire (int me) {
    int other;
    choosing[me] = TRUE;
    number[me] = Findmax() + 1;
    choosing[me] = FALSE;
    for (other=0; other<N; other++)
    {
        while (choosing[other]);
        while (number[other] && EARLIER(other, me));
    }
}

void release (int me) {
    number[me] = 0;
}
```
Atomic Instructions (1)

- Test-and-Set

```c
int TestAndSet (int *v) {
    int rv = *v;
    *v = 1;
    return rv;
}
```

- Using Test-and-Set instruction

```c
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

```c
void Swap (int *v1, int *v2) {
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
```

- Using Swap instruction

```c
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;
}
Atomic Instructions (3)

- Locks using Test-and-Set with bounded-waiting

```c
struct lock { int value = 0; } int waiting[N];

void acquire (struct lock *l, int me)
{
    int key;

    waiting[me] = 1;
    key = 1;
    while (waiting[me] && key)
        key = TestAndSet (&l->value);
    waiting[me] = 0;
}

void release (struct lock *l, int me)
{
    int next = (me + 1) % N;

    while ((next != me) && !waiting[next])
        next = (next + 1) % N;
    if (next == me)
        l->value = 0;
    else
        waiting[next] = 0;
}
```
### Problems with Spinlocks

**Spinlocks**

- Horribly wasteful!
  - If a thread is spinning on a lock, the thread holding the lock cannot make progress.
  - The longer the critical section, the longer the spin.
  - CPU cycle is wasted.
  - Greater the chances for lock holder to 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

```c
void acquire (struct lock *l) {
    cli(); // disable interrupts;
}

void release (struct lock *l) {
    sti(); // enable interrupts;
}
```

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- There is no state associate with the lock.
- Can two threads disable interrupts simultaneously?
Disabling Interrupts (2)

- **What’s wrong?**
  - Only available to kernel
    - Why not have the 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.