CS153: Synchronization

Chengyu Song

Slides modified from Harsha Madhyvasta, Nael Abu-Ghazaleh, and Zhiyun Qian
Administrivia

- **Lab**
  - Lab1: design documents due this Sunday

- **Homework**
  - Homework1 is released, due next Friday
Recap: threads

- Execution and schedule unit
- Tow implementation levels
  - Kernel-level threads vs. user-level threads
- Thread scheduling
  - Non-preemptive vs. preemptive
- Thread interface
  - Create, stop, resume, yield, exit, synchronization
Relationship between threads

- **Independent**: each thread performs a disjoint task, maximizing the utilization of processors (less often)
  - No shared variable, e.g., threads serving different web pages
- **Cooperative**: each thread performs part of a bigger task, usually in a divide-and-conquer manner
  - With shared variables, e.g., sum of a large array
Race condition

• When the output depends on the *sequence* or timing of uncontrollable events
  
  • NOTE: may go wrong but doesn't implies error

• **Basic problem**: two threads are accessing a shared variable, and the variable is modified/written by at least one thread

• Uncontrollable events: threads operations can **interleave arbitrarily** and at **different rates**
  
  • Scheduling (concurrency), parallelism
A classic example

• Suppose we have to implement a function to handle withdrawals from a bank account

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

• And each customer is handled by a different thread and threads are running concurrently
A classic example (cont.)

• Now suppose that you and your father share a bank account with a balance of $1000. Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.

```java
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```
Interleaved schedules

- **Problem**: the execution of the two threads can be interleaved

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

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

put_balance(account, balance);
```

- What is the balance of the account now?
Parallel execution

• **Problem**: the *operations* of the two threads can be interleaved

```java
balance = get_balance(account);
balance = balance - amount;\nput_balance(account, balance);
```

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

• What is the balance of the account now?
Needs for control

• Since the output varies when race condition exists, we need mechanisms to control access to shared resources
• Q1: what resources are shared?
• Q2: how to control the access?
Shared resources

- Process address space
  - Code? -> shared but is read-only
  - Global variables and static objects? -> shared
  - Dynamic objects and other heap objects? -> shared
  - Local variables -> not shared

- Other OS resources?
  - Synchronization in kernel!
Atomic operations

• What we want is atomic
  • The ability to execute a set of operations be executed as an
    uninterruptible entity that appears to the rest of the system as occur
    instantaneously

• What we got
  • A single read or write of a single memory location is atomic
    • Some architecture don't even guarantee this!

• Challenge: how to ridge the gap
Mutual exclusion

- Mutual exclusion restricts thus synchronizes access to shared resources
- Code that uses mutual called a **critical section**
  - 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
  - Example: sharing an ATM with others
Critical section requirements

• Mutual exclusion (mutex)
  • If one thread is in the critical section, then no other is
• Progress
  • A thread in the critical section will eventually leave the critical section
  • If some thread T is not in the critical section, then T cannot prevent
    some other thread S from entering the critical section
Critical section requirements (cont.)

• Bounded waiting (no starvation)
  • If some 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
More about the requirements

• **Safety** property: nothing bad happens
  • Mutex

• **Liveness** property: something good happens
  • Progress, bounded waiting

• **Performance** requirement

• Properties hold for *each run*, while performance depends on *all the runs*
  • Rule of thumb: worry about safety first (but don't forget liveness!).

16
Mechanism to build critical sections

1. Building blocks: atomic read and write
2. Primitive: locks
   - Minimal semantics, used to build others
3. Basic: semaphores
   - Easy to get the hang of, but hard to program with
4. High-level: monitors
   - Requires language support, operations implicit
Mutex: first try

This is called **alternation**

It satisfies **mutex**: if blue is in the critical section, then \( \text{turn} = 1 \) and if yellow is in the critical section, then \( \text{turn} = 2 \). Only one condition can be true at a given time.
Mutex: problem of first try

- Is there anything wrong with this solution?
- It violates **progress**: blue thread could go into an infinite loop outside of the critical section, which will prevent the yellow one from entering
Mutex: Peterson's algorithm

```java
int turn = 1;
bool try1 = false, try2 = false;

while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```java
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```
Mutex: analysis of Peterson's algorithm

```c
int turn = 1;
bool try1 = false, try2 = false;

while (true) {
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
1 try1 = true;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
2 turn = 2;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
3 while (try2 && turn != 1) ;
    { try1 ∧ (turn == 1 ∨ ¬ try2 ∨
                  (try2 ∧ (yellow at 6 or at 7))) }
    critical section
4 try1 = false;
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}

(white at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
 ∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blue at 2 or at 3)))
... ⇒ (turn == 1 ∧ turn == 2)
```

```c
while (true) {
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
5 try2 = true;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
6 turn = 1;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
7 while (try1 && turn != 2) ;
    { try2 ∧ (turn == 2 ∨ ¬ try1 ∨
                  (try1 ∧ (blue at 2 or at 3))) }
    critical section
8 try2 = false;
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}
```
Peterson's algorithm

• Hard to reason

• Simpler ways to implement Mutex exists (see later)
Locks

- Alternation is hard to scale

- A lock is an object in memory providing two operations
  - `acquire()`: before entering the critical section
  - `release()`: after leaving a critical section

- Threads pair calls to `acquire()` and `release()`
  - Between `acquire()`/`release()`, the thread holds the lock
  - `acquire()` does not return until any previous holder releases
Using locks

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

Critical Section

```c
acquire(lock);
balance = get_balance(account);
balance = balance - amount;
```

```c
acquire(lock);
put_balance(account, balance);
release(lock);
```

```c
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);
```
Additional question to ponder

• What can happen if the calls to `acquire()`/`release()` are not paired?
• Why is the `return` outside the critical section? Is this ok?
• What happens when a third thread calls acquire?
Summary

- Race condition can happen between threads with shared data
  - Concurrent
    - At least one write
  - Synchronization is necessary to provide
    - Safety, liveness, performance
- Mechanisms
  - Alternation, lock
For next class ...

• Lock algorithms
• Textbook
  • Lock Algorithms