Operating Systems

Synchronization

Lecture 5
Michael O'Boyle

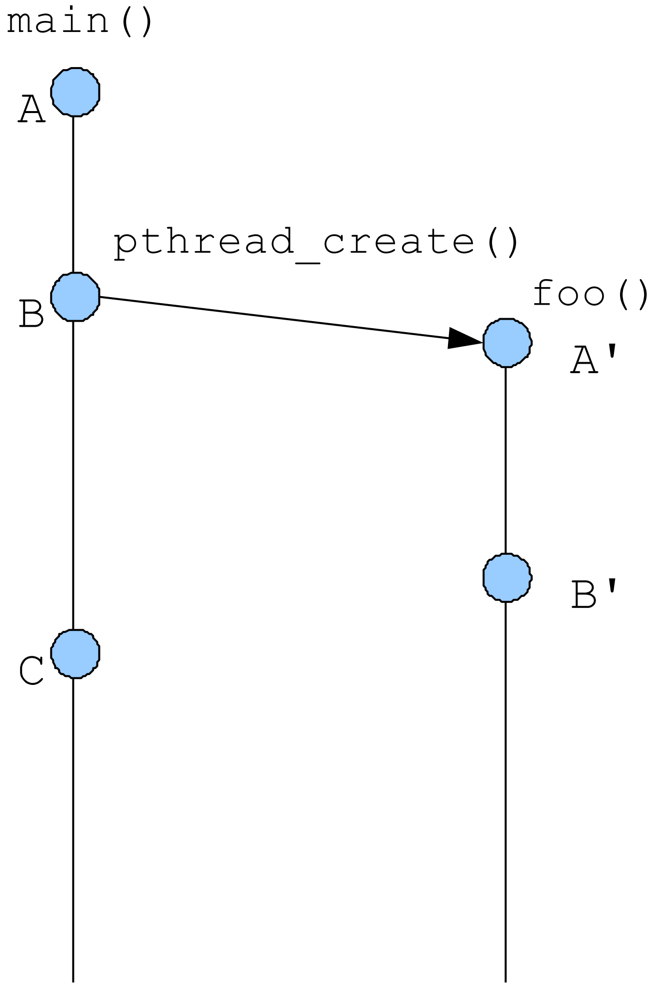
Temporal relations

User view of parallel threads

- Instructions executed by a single thread are totally ordered
  - $A < B < C < \ldots$
- In absence of synchronization,
  - instructions executed by distinct threads must be considered unordered / simultaneous
  - Not $X < X'$, and not $X' < X$

Hardware largely supports this

Example



Y-axis is "time."

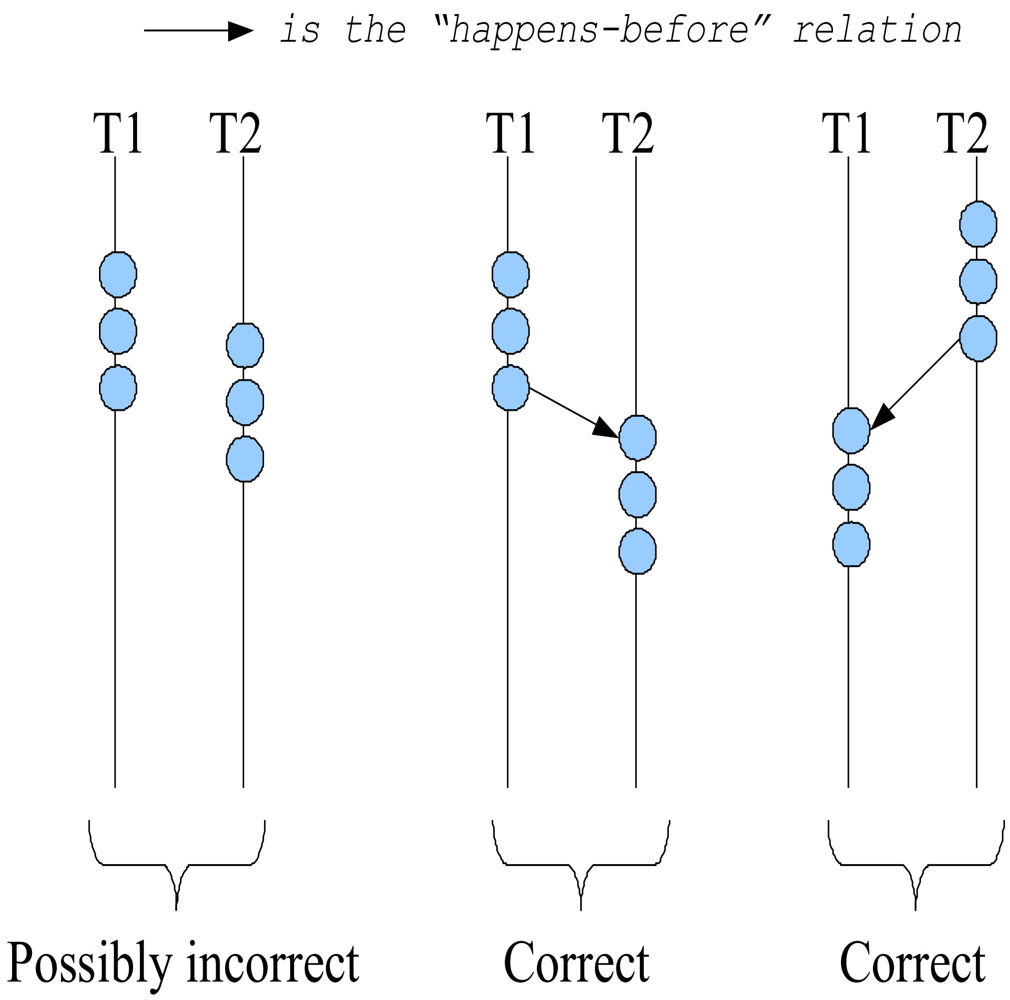
Could be one CPU, could be multiple CPUs (cores).

- A < B < C
- A' < B'
- A < A'
- C == A'
- C == B'

Critical Sections / Mutual Exclusion

- Sequences of instructions that may get incorrect results if executed simultaneously are called critical sections
- Race condition results depend on timing
- Mutual exclusion means “not simultaneous”
  - A < B or B < A
  - We don’t care which
- Forcing mutual exclusion between two critical section executions
  - is sufficient to ensure correct execution
  - guarantees ordering

Critical sections



When do critical sections arise?

- One common pattern:
  - read-modify-write of
  - a shared value (variable)
  - in code that can be executed by concurrent threads
- Shared variable:
  - Globals and heap-allocated variables
  - NOT local variables (which are on the stack)

Race conditions

- A program has a race condition (data race) if the result of an executing depends on timing
  - i.e., is non-deterministic
- Typical symptoms
  - I run it on the same data, and sometimes it prints 0 and sometimes it prints 4
  - I run it on the same data, and sometimes it prints 0 and sometimes it crashes

Example: shared bank account

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
  int balance = get_balance(account);     // read
  balance -= amount;            // modify
  put_balance(account, balance);      // write
  spit out cash;
}
```

- Now suppose that you and your partner share a bank account with a balance of £100.00
  - what happens if you both go to separate CashPoint machines, and simultaneously withdraw £10.00 from the account?

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted

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

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

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence as seen by CPU

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

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
spit out cash;
```

context switch

```
put_balance(account, balance);
spit out cash;
```

- What's the account balance after this sequence?
  - who's happy, the bank or you?
- How often is this sequence likely to occur?

Other Execution Orders

- Which interleavings are ok? Which are not?

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

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

How About Now?

```
int xfer(from, to, machine) {
  withdraw( from, machine );
  deposit( to, machine );
}
```

```
int xfer(from, to, machine) {
  withdraw( from, machine );
  deposit( to, machine );
}
```

- Morals:
  - Interleavings are hard to reason about
    - We make lots of mistakes
    - Control-flow analysis is hard for tools to get right
  - Identifying critical sections and ensuring mutually exclusive access can make things easier

Another example

```
i++;
```

```
i++;
```

Correct critical section requirements

- Correct critical sections have the following 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
      - assumes threads eventually leave critical sections
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

Implementing Mutual Exclusion

How do we do it?

- via hardware: special machine instructions
- via OS support: OS provides primitives via system call
- via software: entirely by user code

Of course, OS support needs internal hardware or software implementation.
How do we do it in software?

We *assume* that mutual exclusion exists in hardware, so that memory access is atomic: only one read or write to a given memory location at a

We will now try to develop a solution for mutual exclusion of two processes, P_0 and P_1. (Let $\hat{\imath}$ mean $1 - i$.)

Mutex – first attempt

Suppose we have a global variable `turn`. We could say that when P_i wishes to enter critical section, it loops checking `turn`, and can proceed iff $\texttt{turn} = i$. When done, flips `turn`. In pseudocode:

```
while ( turn != i ) { }
/* critical section */
turn = î;
```

This has obvious problems:

- processes busy-wait
- the processes must take strict turns

although it does enforce mutex.

Mutex - Second attempt

Need to keep state of each process, not just id of next process.

So have an array of two boolean flags, `flag[`i`]`, indicating whether P_i is in critical. Then P_i does:

```
while ( flag[î] ) { }
flag[i] = true;
/* critical section */
flag[i] = false;
```

This doesn't even enforce mutex: P_0 and P_1 might check each other's flag, then both set own flags to true and enter critical section.

Mutex – Third attempt

Maybe set one's own flag before checking the other's?

```
flag[i] = true;
while ( flag[î] ) { }
/* critical section */
flag[i] = false;
```

This does enforce mutex. (Exercise: prove it.)

But now both processes can set flag to true, then loop for ever waiting for the other! This is *deadlock*.

Mutex – Fourth attempt

Deadlock arose because processes insisted on entering critical section and busy-waited. So if other process's flag is set, let's clear our flag for a bit to allow it to proceed:

```
flag[i] = true;
while ( flag[î] ) {
  flag[i] = false;
  /* sleep for a bit */
  flag[i] = true;
}
/* critical section */
flag[i] = false;
```

OK, but now it is *possible* for the processes to run in exact synchrony and keep deferring to each other – *livelock*.

Peterson's Algorithm

```
flag[i] = true;
turn = î;
while ( flag[î] && turn == î ) { }
/* critical section */
flag[i] = false;
```

Works but we want something easier for programmers

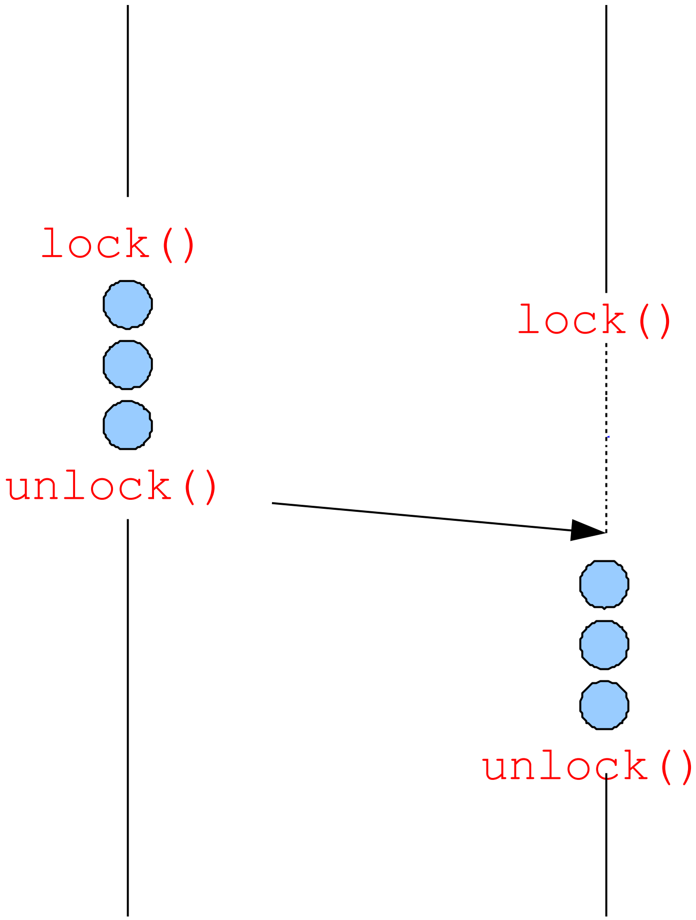
Mechanisms for building critical sections

- Spinlocks
  - primitive, minimal semantics; used to build others
- Semaphores (and non-spinning locks)
  - basic, easy to get the hang of, somewhat hard to program with
- Monitors
  - higher level, requires language support, implicit operations
  - easier to program with; Java "`synchronized()`" as an example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

Locks

- A lock is a memory object with two operations:
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section
- `acquire()` prevents progress of the thread until the lock can be acquired
- Note: terminology varies: acquire/release, lock/unlock

Locks: Example



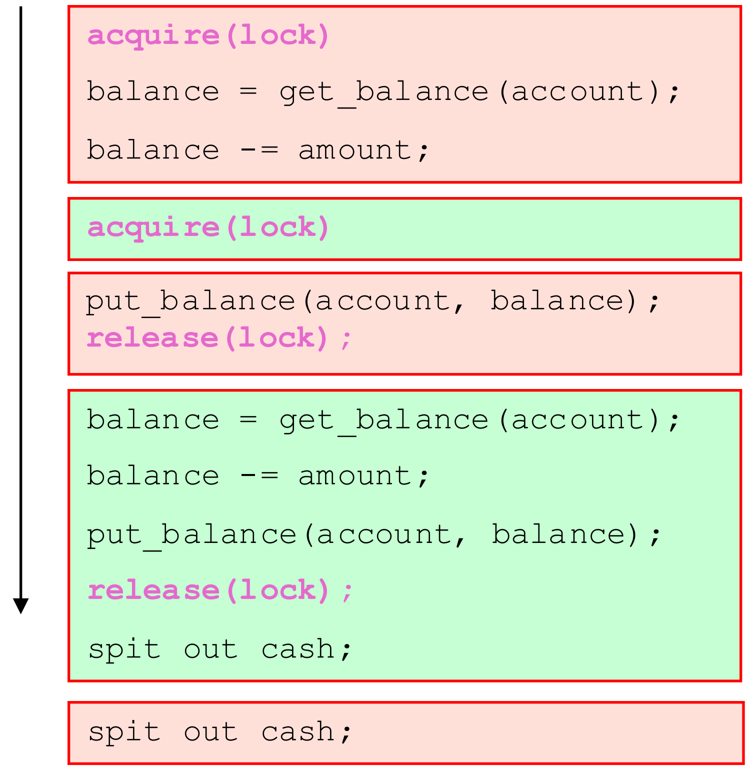
Acquire/Release

- Threads pair up calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread holds the lock
  - `acquire()` does not return until the caller "owns" (holds) the lock
    - at most one thread can hold a lock at a time
- What happens if the calls aren't paired
  - I acquire, but neglect to release?
- What happens if the two threads acquire different locks
  - I think that access to a particular shared data structure is mediated by lock A, and you think it's mediated by lock B?
- What is the right granularity of locking?

Using locks

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  spit out cash;
}
```

critical section



- What happens when green tries to acquire the lock?

Spinlocks

- How do we implement spinlocks? Here's one attempt:

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

the caller "busy-waits", or spins, for lock to be released ⇒ hence spinlock

- Race condition in acquire
- Could use Peterson – but assumes no compiler optimization

Peterson's Algorithm

```
flag[i] = true;
turn = î;
while ( flag[î] && turn == î ) { }
/* critical section */
flag[i] = false;
```

If flag[1-i] hoisted, as loop invariant, then fails

Implementing spinlocks

- Problem is that implementation of spinlocks has critical sections, too!
  - the acquire/release must be **atomic**
    - atomic == executes as though it could not be interrupted
    - code that executes "all or nothing"
  - Compiler can hoist code that is invariant
- Need help from the hardware
  - atomic instructions
    - test-and-set, compare-and-swap, …

Spinlocks: Hardware Test-and-Set

- CPU provides the following as one atomic instruction:

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- Remember, this is a single atomic instruction …

Implementing spinlocks using Test-and-Set

- So, to fix our broken spinlocks:

```
struct lock {
  int held = 0;
}
void acquire(lock) {
   while(test_and_set(&lock->held));
}
void release(lock) {
  lock->held = 0;
}
```

  - mutual exclusion? (at most one thread in the critical section)
  - progress? (T outside cannot prevent S from entering)
  - bounded waiting? (waiting T will eventually enter)
  - performance? (low overhead (modulo the spinning part …))

Reminder of use …

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  spit out cash;
}
```

critical section

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

```
acquire(lock)
```

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

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
spit out cash;
```

```
spit out cash;
```

- How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  - calls yield( ) *(spin-then-block)*
  - there’s an involuntary context switch (e.g., timer interrupt)

Problems with spinlocks

- Spinlocks work, but are wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make progress
    - You'll spin for a scheduling quantum
  - `(pthread_spin_t)`
- Only want spinlocks as primitives to build higher-level synchronization constructs
  - Ok as ensure acquiring only happens for a short time
- We'll see later how to build blocking locks
  - But there is overhead – can be cheaper to spin

Summary

- Synchronization introduces temporal ordering
- Synchronization can eliminate races
- Peterson's Algorithm
- Synchronization can be provided by locks, semaphores, monitors, messages …
- Spinlocks are the lowest-level mechanism
  - primitive in terms of semantics – error-prone
  - implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)