# Locks

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2023

# The Classic Example

- Withdrawing money from a bank account
  - Suppose you and your girl (or 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?

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

# The Classic Example: Problem

- The execution of the two threads can be interleaved, assuming preemptive scheduling:

**Execution sequence as seen by CPU**

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

**Context switch**

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

**Context switch**

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

# A Real Example

```
extern long g;

void inc() {
  g++;
}
```

```
ld      a0, 0(s1)
addi    a0, a0, 1
sd      a0, 0(s1)
ret
```

| Thread T1 | | Thread T2 |
|---|---|---|
| `ld   a0, 0(s1)`<br>`addi a0, a0, 1` | | |
| | *context switch* | |
| | | `ld   a0, 0(s1)`<br>`addi a0, a0, 1`<br>`sd   a0, 0(s1)` |
| | *context switch* | |
| `sd   a0, 0(s1)` | | |

# Sharing Resources

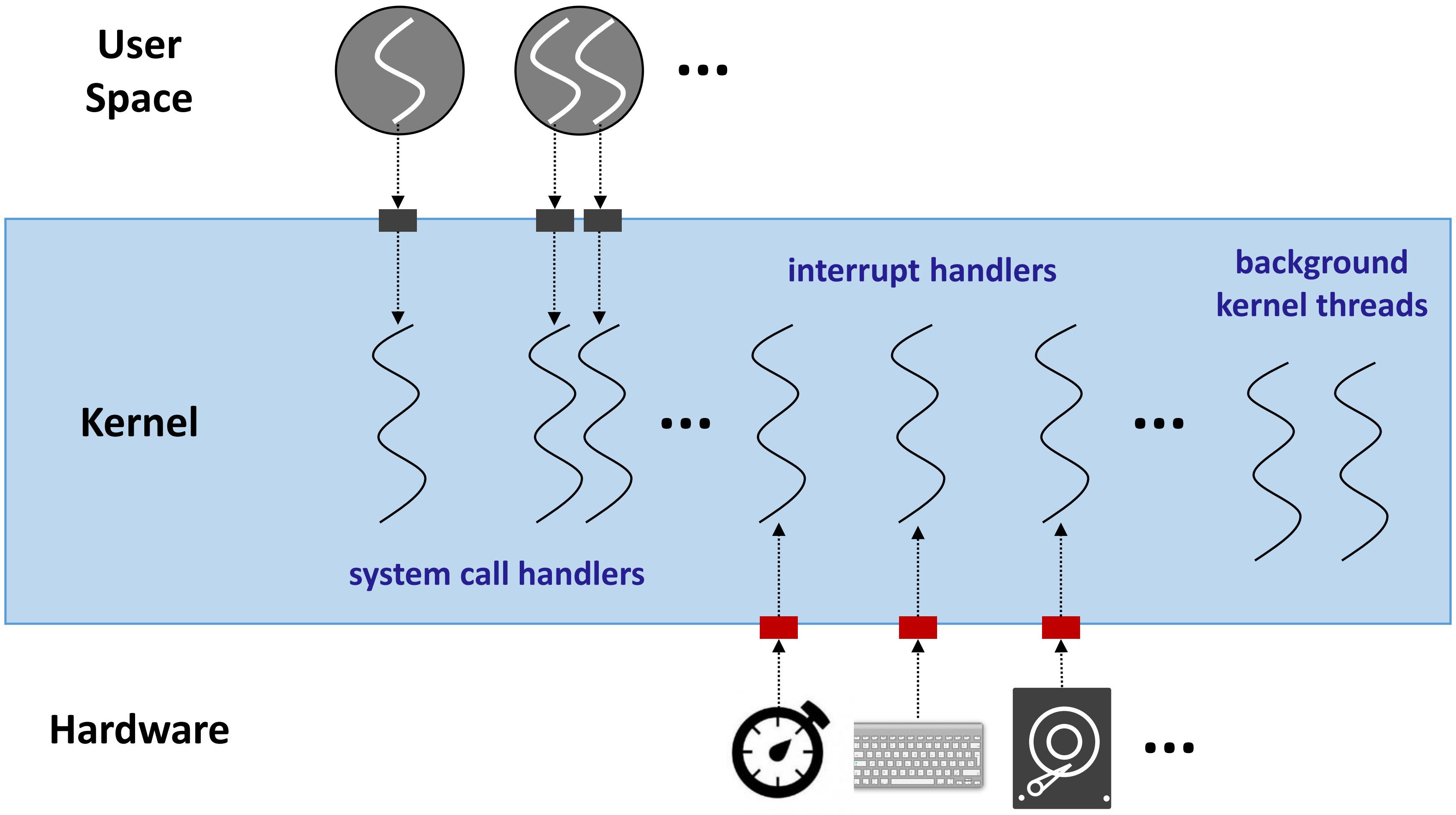
- Local variables are not shared among threads
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack
- Global variables are shared among threads
  - Stored in static data segment, accessible by any thread
-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

# Synchronization Problem

- Concurrency leads to non-deterministic results
  - Two or more concurrent threads accessing a shared resource create a __________ condition
  - The output of the program is not deterministic; it varies from run to run even with same inputs, depending on timing
  - Hard to debug ("Heisenbugs")

- We need synchronization mechanisms for controlling access to shared resources
  - Synchronization restricts the concurrency
  - Scheduling is not under programmer's control

# Concurrency in the Kernel



# Critical Section

- A critical section is a piece of code that accesses a shared resource, usually a variable or data structure

```
ld   a0, 0(s1)
addi a0, a0, 1
sd   a0, 0(s1)
```

critical section

- Need ________________ for critical sections
  - Execute the critical section atomically (all-or-nothing)
  - 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

# Locks

- A lock is an object (in memory) that provides mutual exclusion with the following two operations:
  - `acquire()`: wait until lock is free, then grab it
  - `release()`: unlock and wake up any thread waiting in `acquire()`

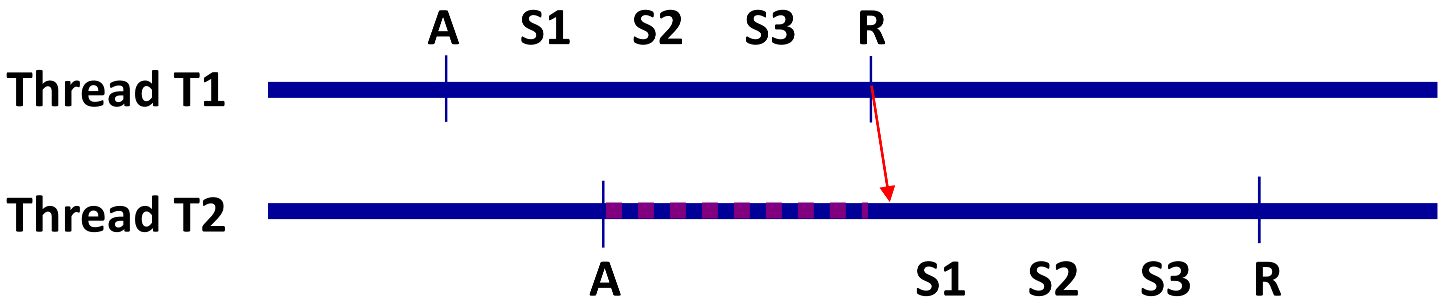
- Using locks
  - Lock is initially free
  - Call `acquire()` before entering a critical section, and `release()` after leaving it
  - `acquire()` does not return until the caller holds the lock
  - On `acquire()`, a thread can spin (spinlock) or block (mutex)
  - At most one thread can hold a lock at a time

# Using Locks

```
A     acquire(lock);
S1    balance = get_balance(account);
S2    balance = balance - amount;
S3    put_balance(account, balance);
R     release(lock);
```

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

**critical section** (S1–S3)



# Requirements for Locks

- Correctness
  - Mutual exclusion: only one thread in critical section at a time
  - __________ (deadlock-free): if several threads want to enter the critical section, must allow one to proceed
  - Bounded waiting (______________): must eventually allow each waiting thread to enter
- Fairness
  - Each thread gets a fair chance at acquiring the lock
- Performance
  - Time overhead for a lock without and with contentions (possibly on multiple CPUs)?

# An Initial Attempt

- An initial implementation of a spinlock

```
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

- Does this work?

# Implementing Locks

- 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
  - Load-Linked (LL) and Store-Conditional (SC)
  - Fetch-And-Add
- Controlling interrupts

# Software-only Algorithm

- The second attempt to implement spinlocks
  - Note: each load and store instruction is atomic

```
int interested[2];

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

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

- Does this work?

# Peterson's Algorithm

- Solves the critical section problem for two processes

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

void acquire(int process) {
  int other = 1 – process;
  interested[process] = TRUE;
  turn = other;
  while (interested[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)

```
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;
}
```

# Test-And-Set

- Atomic instructions
  - read-modify-write operations guaranteed to be executed "atomically"
- Test-And-Set instruction
  - Returns the old value of a memory location while simultaneously updating it to the new value
  - e.g., `xchg` in x86 (`amoswap` in RISC-V): exchange memory with register

```
int TestAndSet(int *v, int new) {
  int old = *v;
  *v = new;
  return old;
}
```

# Using Test-And-Set

- A simple spinlock using Test-And-Set instruction
  - Refer to `spinlock.h` and `spinlock.c` in xv6

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

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

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

→

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

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

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

# Locks with Bounded Waiting

```
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;
}
```

# Compare-And-Swap

- Supported in x86, Sparc, etc.
  - Update the memory location with the new value only when its old value equals to the "expected" value
  - e.g., `cmpxchg` in x86: compare and exchange

```
int CompareAndSwap(int *v, int expected, int new) {
  int old = *v;
  if (old == expected)
    *v = new;
  return old;
}

void acquire(struct lock *l) {
  while (CompareAndSwap(&l->held, _____, _____));
}
```

# LL & SC

- Supported in MIPS, Alpha, PowerPC, ARM, RISC-V, etc.
  - Load-Locked(LL) fetches a value from memory
  - Store-Conditional(SC) succeeds with returning 1 if no intervening store to the address has taken place
  - Otherwise, SC returns 0 without updating the memory

```
void acquire(struct lock *l) {
  while (1) {
    while (LL(&l->held));
    if (SC(&l->held, 1)) return;
  }
}
void release(struct lock *l) {
  l->held = 0;
}
```

# Fetch-And-Add

- Supported in x86, RISC-V, etc.
  - Atomically increments a value while returning the old value
  - e.g., xadd in x86: exchange and add

```
int FetchAndAdd(int *v, int a) {
  int old = *v;
  *v = old + a;
  return old;
}
```

# Ticket Locks Using Fetch-And-Add

- First get a ticket and wait until its turn
- Provides bounded waiting

```
struct lock {
  int ticket = 0;
  int turn = 0;
};

void acquire(struct lock *l) {
  int myturn = FetchAndAdd(&l->ticket, 1);
  while (l->turn != myturn);
}

void release(struct lock *l) {
  l->turn = l->turn + 1;
}
```

# Controlling Interrupts (1)

- Disable interrupts for critical sections

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

  - Disabling interrupts blocks external events that could trigger a context switch
  - The code inside the critical section will not be interrupted
  - There is no state associated with the lock
  - `intr_off()` and `intr_on()` vs. `push_off()` and `pop_off()` in xv6
  - Can two threads disable interrupts simultaneously?

# Controlling Interrupts (2)

- Pros
  - Simple
  - Useful for a single-processor system
- Cons
  - Only available to kernel
    - Why not provide them as system calls?
  - Insufficient on multi-processor systems
    - Back to atomic instructions
  - When the critical section is long, important interrupts can be delayed or lost (e.g., timer, disks, etc.)
  - Slower than executing atomic instructions on modern CPUs

# Summary

- Spinlocks are 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

- Spinlocks (and disabling interrupts on a single CPU) are primitive synchronization mechanisms
  - They are used to build higher-level synchronization constructs