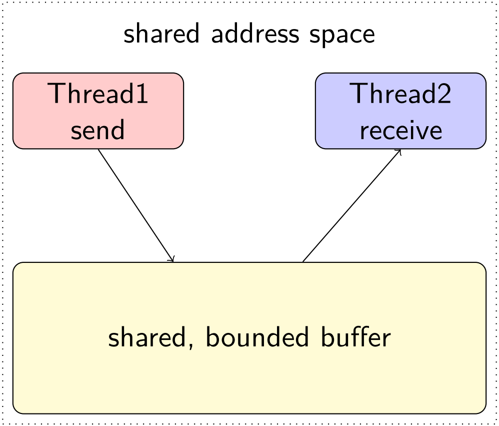
# Operating Systems

## Race Conditions

Hongliang Liang, BUPT

Sep. 2023

# Communication Across Threads



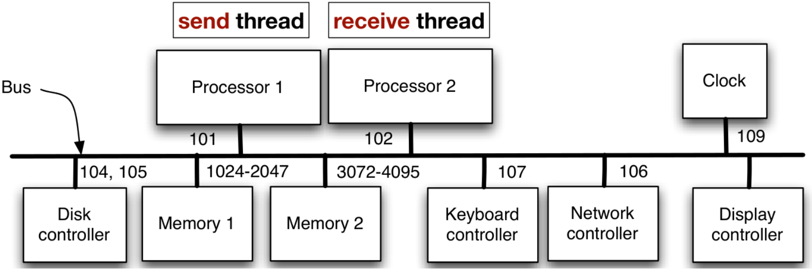
# Communication Across Threads: Primitives

*The OS may provide the following interfaces for **send** and **receive** with bounded buffers:*

- send(message): if there is room in the bounded buffer, insert message in the buffer. If not, **stop the calling thread and wait** until there is room.
- receive(): if there is a message in the bounded buffer, return the message to the calling thread. If not, **stop the calling thread and wait** until another thread sends a message to buffer.

# First assumption: one CPU per thread

*For now, let's assume that there is an available physical CPU for each thread, so we don't need to worry about multiplexing threads on the same CPU*



# The Producer-Consumer Problem

- The problem of sharing a bounded buffer between two threads is an instance of the producer-consumer problem
  - The producer needs to first add a message to the shared buffer before the consumer can remove it
  - The producer needs to wait for the consumer to catch up when the buffer fills up
- Let's try to implement `send()` and `receive()`
  - We can alternatively call them producer() and consumer()

# First implementation of send() and receive()

```
message buffer[N]
int in = 0, out = 0
send(message msg)
  while in - out == N do nothing
  buffer[in % N] = msg
  in = in + 1
  return
message receive()
  while in == out do nothing
  msg = buffer[out % N]
  out = out + 1
  return msg
```

# Assumptions revisited

- There is only one sending thread and one receiving thread
  - Only one thread updates each shared variable
  - Only the receiving thread updates **out**
  - Only the sending thread updates **in**
- **One writer principle**: If each variable has only one writer, coordination becomes easier.

# What if we allow several senders?

- Each sender has its own CPU, but progresses at different paces

```
A buffer is empty        fill entry 0   in=1
---------------------------------------------------->


B    buffer is empty   fill entry 0         in = 2
---------------------------------------------------->
```

- This is called a **race condition**, since it depends on the exact timing of two threads

# Another race condition

```
in = in + 1
  1 LOAD in, R0
  2 ADD R0, 1
  3 STORE R0, in

  read 0
A           1       2  3         in = 1
------------------------------------------------------>


     read 0
B              1          2  3          in = 2
------------------------------------------------------>
```

*Very difficult to debug, as it may rarely occur, and hard to reproduce*

# How do we fix race conditions?

- Race conditions occur whenever the output depends on the precise execution order of threads
- How do we systematically avoid race conditions?

# How do we fix race conditions?

- Intuitively, we need to make threads coordinate with each other to ensure **mutual exclusion** in accessing **critical sections** of code
  - In this case, a critical section defines a multi-step operation on shared data (e.g., variables) that needs to become "before-or-after" **atomic actions**
- We need a lock!

# Shared locks to achieve mutual exclusion

- A **lock** is a shared variable that acts as a flag to coordinate usage of other shared variables
- We introduce two new primitives to work with locks
  - **acquire()** and **release()**
  - Now a thread can acquire a lock, hold it for a while, and then release it
  - When a thread is holding a lock, other threads that attempt to acquire that same lock will wait until the first thread releases the lock
  - Analogy: an exclusive restaurant has a table and a chair. when a customer is served, others must wait outside.
- By surrounding multi-step operations involving shared variables with acquire() and release(), we make sure a multistep operation behaves like a single-step operation

alternatively called **lock()** and **unlock()** (in BLITZ)

# Second implementation of send() and receive()

```
message buffer[N]
int in = 0, out = 0
lock buffer_lock = UNLOCKED // add
send(message msg)
   acquire(buffer_lock)      // add
   while in - out == N do nothing
   buffer[in % N] = msg
   in = in + 1
   release(buffer_lock)      // add
   return
message receive()
   acquire(buffer_lock)      // add
   while in == out do nothing
   msg = buffer[out % N]
   out = out + 1
   release(buffer_lock)      // add
   return msg
```

# Second implementation of send() and receive()

```
message buffer[N]
int in = 0, out = 0
lock buffer_lock = UNLOCKED // add
send(message msg)
   acquire(buffer_lock)      // add
   while in - out == N do nothing
   buffer[in % N] = msg
   in = in + 1
   release(buffer_lock)      // add
   return
message receive()
   acquire(buffer_lock)      // add
   while in == out do nothing
   msg = buffer[out % N]
   out = out + 1
   release(buffer_lock)      // add
   return msg
```

Correct? e.g., sender holding the lock faces the full buffer.

# Third implementation of send() and receive()

```
message buffer[N]
int in = 0, out = 0
lock buffer_lock = UNLOCKED
send(message msg)
   acquire(buffer_lock)
   while in - out == N do
     release(buffer_lock)    //
     acquire(buffer_lock)    //
   buffer[in % N] = msg
   in = in + 1
   release(buffer_lock)
   return
message receive()
   acquire(buffer_lock)
   while in == out do
     release(buffer_lock)    //
     acquire(buffer_lock)    //
   msg = buffer[out % N]
   out = out + 1
   release(buffer_lock)
   return msg
```

# Implementing acquire() and release()

- A correct implementation must enforce the "single-acquire" protocol

  Several threads may attempt to acquire the lock at the same time, but only one should succeed

- Consider the following implementation —

# Implementing acquire() and release()

```
struct lock
  int state

acquire(lock L)
  while L.state == LOCKED do nothing
  L.state = LOCKED

release(lock L)
  L.state = UNLOCKED
```

# Race condition

```
A L.state is UNLOCKED     L.state = LOCKED
--------------------------------------------------------->

B            L.state is UNLOCKED     L.state = LOCKED
--------------------------------------------------------->
```

# Why the race condition?

- The faulty **acquire()** has a multi-step operation on a shared variable (the lock), and we must ensure in some way that acquire() itself is a before-or-after atomic action
- Once acquire() is an atomic action, we can use it to turn arbitrary multi-step operations on shared variables into before-or-after atomic actions
- Did we just go back to the very beginning?

# Why the race condition?

- The faulty **acquire()** has a multi-step operation on a shared variable (the lock), and we must ensure in some way that acquire() itself is a before-or-after atomic action
- Once acquire() is an atomic action, we can use it to turn arbitrary multi-step operations on shared variables into before-or-after atomic actions
- Did we just go back to the very beginning?

- No, we have actually made progress!
- We reduced a more general problem (making multi-step operations on shared variables before-or-after actions) to a narrower problem (making an operation on a single shared lock a before-or-after action)!

# Disabling interrupts?

- Timer interrupts are disabled
- The thread scheduler is not able to run
- No other thread can run on this CPU

# Disabling interrupts?

Problems:

- If we have multiple CPUs, threads running on the other processors can enter the critical section even with interrupts disabled
- Is it fine to trust the user threads for disabling interrupts?
- If so, what if they never re-enable the interrupts?

Hardware support: we need it again

# Hardware support: the TSL instruction

- Test and Set Lock (TSL) instruction (or called Test and Set) from the hardware, modifies the content of a memory location, returning its old value

```
TSL(LOCK)
   do atomic
     1 RX = LOCK
     2 LOCK = LOCKED
   RETURN RX
```

```
bool tas(bool *lock){
  bool old = *lock;
  *lock = true;
  return old;
}
```

- The bus arbiter in hardware that controls the bus connecting processors to the memory must guarantee —
    - the LOAD (line 1) and STORE (line 2) instructions must execute as before-or-after atomic actions
    - By allowing both to be done in a single clock cycle

# Implementing acquire() and release() with TSL

```
acquire(lock L)
  R1 = TSL(L.state)
  while R1 == LOCKED do
    R1 = TSL(L.state)

release(lock L)
  L.state = UNLOCKED
```

# Correctness of the solution

- To see that the implementation is correct, we assume L is UNLOCKED
- If a thread calls **acquire(L)**, then after TSL, L is LOCKED and R1 contains UNLOCKED, so the thread acquired the lock
- The next thread that calls acquire(L) sees LOCKED in R1 after TSL, showing that a thread is holding the lock
- The thread that tried to acquire L will **spin** until R1 contains UNLOCKED
- When releasing a lock, no test is needed, an ordinary **STORE** instruction can do the job without creating a race condition

# Compare-and-swap lock

```
bool cas(T *ptr, T expt, T new) {
  if (*ptr == expt) {
    *ptr = new;
    return true;
  }
  return false;
}
```

The function compares the value at `*ptr` and if it is equal to `expt` then the value is overwritten with `new`. The function returns `true` if the swap happened.

# Compare-and-swap lock

How would you implement the lock `acquire` operation?

# Compare-and-swap lock

How would you implement the lock `acquire` operation?

```
void acquire_cas(bool *lck) {
  while (compare_and_swap(lck, false, true)
        == false);
}
```

# A summary so far

- **Mutual exclusion** from a **critical section** using **shared locks**
  - No concurrent threads are simultaneously in the critical section at the same time
  - Shared locks are usually called **mutex locks** (mutual exclusion locks)
- Two operations to protect the critical section
  - acquire(L): alternatively called lock(L)
  - release(L): alternatively called unlock(L)
- Implementation
  - Using the Test-and-Set-Lock (TSL) or Compare-and-Swap-Lock instruction: an atomic action
- With mutex locks, the producer-consumer problem is solved
  - allowing multiple senders and receivers to access a shared buffer