Activity 2# 1 ส่งทุกคน 43 + โปรแกรม ส่งตัวแทนในโปรแกรมให้ ตัวแทนส่ง
// คอมเม้นชื่อ + รหัสทุกคนด้วย

Test Case 2
- report ทำหรือแก้ไขมา .pdf รหัสทุกคน
- programe ที่แก้แล้ว version ที่ทำตนได้ ใช้รหัสนศ.

20 ต.ค. 65 # Synchronization - ไฟล์ .ppt มี/ไม่มีก็ได้

นำเสนอผลงาน test study Case ครั้งที่ 2

ในไฟล์ใส่ชื่อ + รหัสสมาชิกทุกคนด้วย!!

# The Dark Side of Concurrency

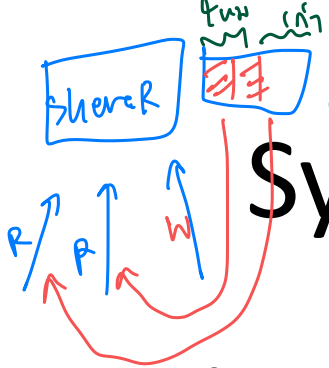
With interleaved executions, the order in which processes execute at runtime is *nondeterministic*. คาดเดาไม่ได้ - เกิด context switch ที่ไหน ( ควบคุมไม่ได้)

- depends on the exact order and timing of process arrivals
- depends on exact timing of asynchronous devices (disk, clock)
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bay doors *before* you release the bomb.
- Two people can't wash dishes in the same sink at the same time.

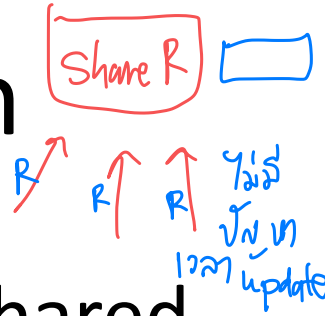
The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.




เกิดปัญหาเวลา update ค่าเก่า/ใหม่ ถ้าอยากให้ถูกต้องupdateเสร็จค่อยอ่าน

# Synchronization Motivation

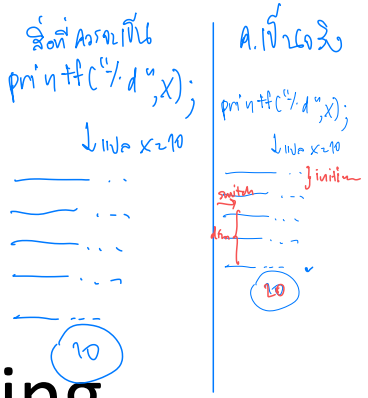
นอกเหนือการควบคุม




- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic

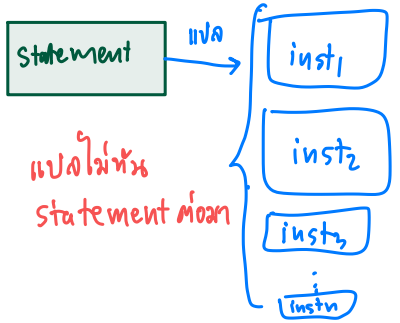
สั่งรอง ของ

switch เมื่อไหร่ → ทำprocessลูกก่อน ค่อยกลับมาทำตัวเอง




Uprocessor fed instruction มีการเรียงลำดับใหม่ (มีการเปลี่ยน)

print(x) แต่ไม่ใช่ประกาศค่า x



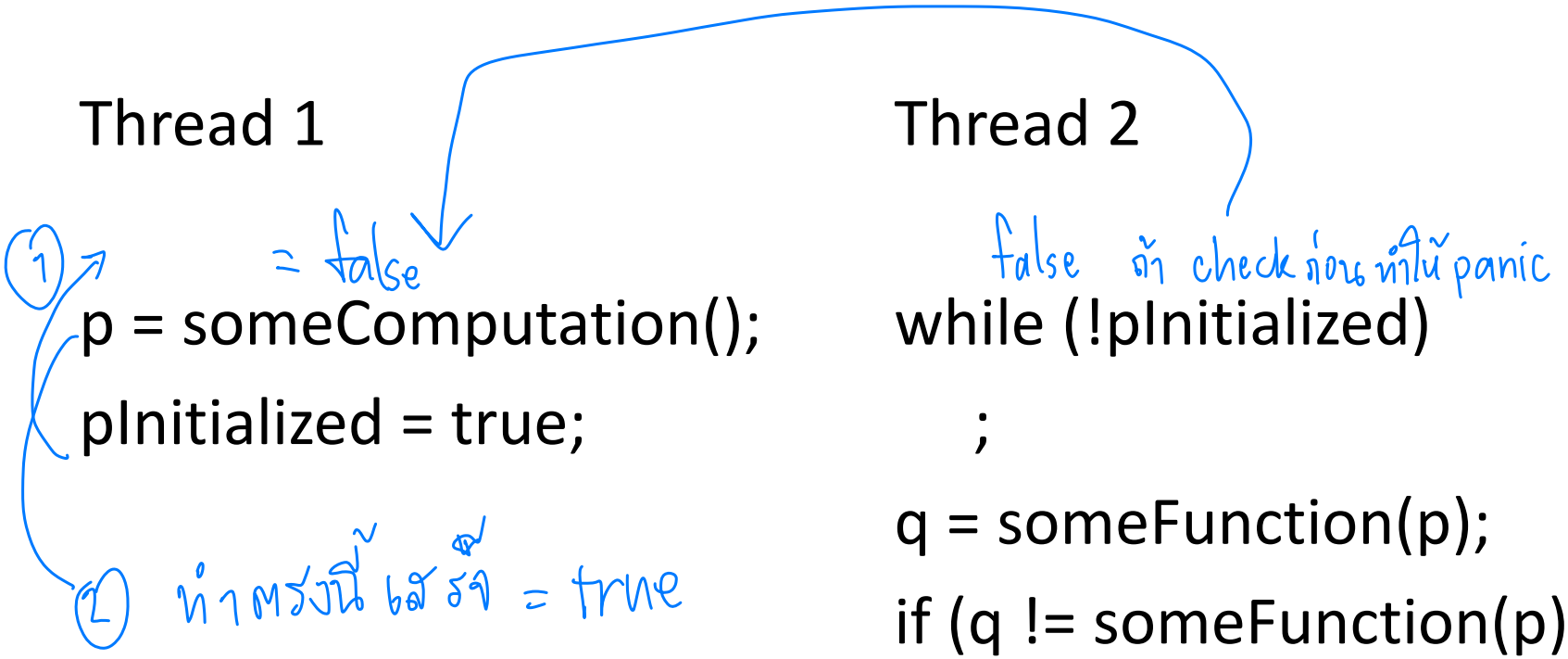
# Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized)
    ;
q = someFunction(p);
if (q != someFunction(p))
    panic
```




อาจจะเกิด panic เนื่องจาก HW/SW สมัย compiler replying ทำให้สลับเรียงลำดับคำสั่งผิดสังผลให้ อาจ จะ panic (reordering instruction)

# Why Reordering?

- Why do compilers reorder instructions?
  - Efficient code generation requires analyzing control/data dependency
  - If variables can spontaneously change, most compiler optimizations become impossible
- Why do CPUs reorder instructions?
  - Write buffering: allow next instruction to execute while write is being completed

Fix: **memory barrier** ใช้เมื่อมี ตัวแปร shared resource

  - Instruction to compiler/CPU
  - All ops before barrier complete before barrier returns
  - No op after barrier starts until barrier returns

# Too Much Milk Example

Shared Resource

| | Person A | Person B |
|---|---|---|
| 12:30 | Look in fridge. Out of milk. | |
| 12:35 | Leave for store. | |
| 12:40 | Arrive at store. | Look in fridge. Out of milk. |
| 12:45 | Buy milk. | Leave for store. |
| 12:50 | Arrive home, put milk away. | Arrive at store. |
| 12:55 | | Buy milk. |
| 1:00 | | Arrive home, put milk away. Oh no! |

แบบนี้เรียกว่า "RACE CONDITION" แย่งกันใช้ทรัพยากรร่วมกัน

# Definitions

**Race condition:** output of a concurrent program depends on the order of operations between threads

**Mutual exclusion:** only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

**Lock:** prevent someone from doing something

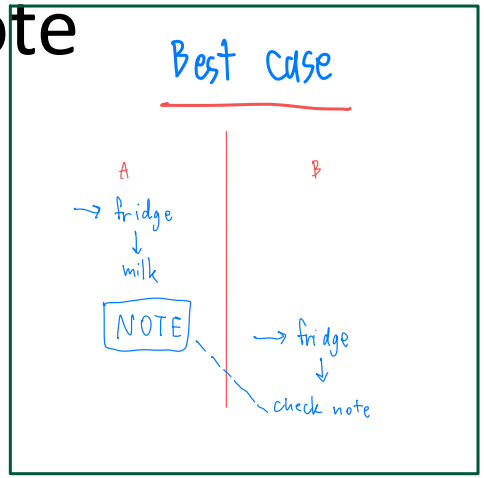
- Lock before entering critical section, before accessing shared data
- Unlock when leaving, after done accessing shared data
- Wait if locked (all synchronization involves waiting!)

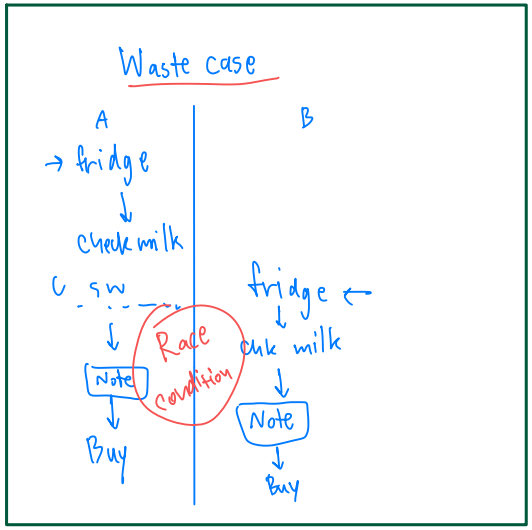
# Too Much Milk, Try #1

ทำให้เกิด Race condition ได้

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
    }
```



# Too Much Milk, Try #2

สรุป starvation ออกแขกนอนทั้งคู่ T_T

แก้โดย แปะ Note ไปไว้ก่อน

Thread A

leave note A → แปะไว้ → ไม่เจอB

if (!note B) {

if (!milk) → check นม

buy milk → ซื้อนม

}

remove note A → ล้อยเอา Note ออก

Thread B

leave note B → แปะไว้ → ไม่เจอB

if (!noteA) {

if (!milk) → check นม

buy milk → ซื้อนม

}

remove note B → ล้อยเอา Note ออก

context switch

ทำงานต่อ เจอโน้ตA ไม่ซื้อ

ทำงานต่อ

# Too Much Milk, Try #3

| Thread A | Thread B |
| --- | --- |
| leave note A | leave note B |
| while (note B) // X | if (!noteA) {  // Y |
| do nothing; | if (!milk) |
| if (!milk) | buy milk |
| buy milk; | } |
| remove note A | remove note B |

เพิ่ม

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

# Lessons

- Solution is complicated
  - "obvious" code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Even more complex: see Peterson's algorithm

# Roadmap

Concurrent Applications

---

Shared Objects

Bounded Buffer  Barrier

---

Synchronization Variables

Semaphores  Locks  Condition Variables

---

Atomic Instructions

Interrupt Disable  Test-and-Set

---

Hardware

Multiple Processors  Hardware Interrupts

---

# Locks

- Lock::acquire
  - wait until lock is free, then take it
- Lock::release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

# Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
lock.release();
```

Critical

Lock เพื่อรัน musual distibution

# Lock Example: Malloc/Free

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Logic ที่ทุกคนตกลงทำกันแบบนี้
ไม่ทำตามๆ = เกิด context switch
เป็นเรื่องที่ทุกคนตกลงร่วมกันว่าจะ
ทำ ถ้าตกลงใช้ Lock ก็ต้อง
ใช้ Lock ทั้งทีม

ตกลงใช้ Lock ใช้เหมือนกัน
ทุกคน

# Rules for Using Locks

- Lock is initially free ไม่มีใครครอบครอง
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

ใครจะใช้ก็แย่ง Apply Lock เพื่อใช้
เพื่อใช้เสร็จคืนสถานะ

อยากใช้งานห้องไปแย่งชิงปุ่มตรงนี้

เข้าสู่ predecount section เพื่อใส่ (apply lock)

ทันทีเสร็จใช้เสร็จ

ใช้ตรงไหน release ตรงนั้น

ปล. ตู้เป็นไม่ได้ล็อค แค่ใช้ปุ่มกดมาดูเฉยๆ

ในfunctionถ้ามีLockเปิด
ต้องมี Lock ปิด

# Will this code work?

อยากทำให้ใช้ได้ ต้อง initialize ให้เป็น null แต่แรกเลย แล้วจะมีกรณี

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = …
    p->field2 = …
    return p;
}
```

ไม่ขึ้นกับ Alm ขึ้นกับ tool ที่ใช้แปลภาษา

กรณีนี้ p ≠ null ในบางอัน ก็ทำให้ ดีใจ

สมมติไม่ มี initialize ใน ใหม่ อาจจะใช้ไม่ได้ ต้นใหม่ แต่ใช้ ได้ในเก่า

# Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

# Condition Variables

- Waiting inside a critical section
  - Called only when holding a lock
- Wait: atomically release lock and relinquish processor
  - Reacquire the lock when wakened
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

# Condition Variable Design Pattern

เริ่ม ready → running ไม่สำเร็จ release คืน ระบบ

X มาใช้

Read Sensor

```
methodThatWaits() {
    lock.acquire();
    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // Read/write shared state
    lock.release();
}
```

```
methodThatSignals() {
    lock.acquire();
    // Read/write shared state
    // If testSharedState is now true
    cv.signal(&lock);
    // Read/write shared state
    lock.release();
}
```

X , X == 0

X เป็น 0

write X = [ ] ≠ 0

① Release } Lock เปลี่ยนจาก running → wait

หยุดรอ เพราะใช้ X = 0 ไม่ได้เชื่องจากมันไม่ใช่ ค่าของ sensor

② acquir Lock ปลุก wait ที่ทำงาน Lock นั้นๆ

* ไม่ release lock ไป & lock ปรับ

ชิง ปัดไม่เสร็จคืนเวลา แล้วเปลี่ยนสถานะ เป็น ready

# C#

lock 1 thread ทำได้ทีละ 1

- Lock

```
lock (_Lock)
{
   …
}
```

- Condition Variable

```
static object _Lock = new object();

lock (_Lock)
{
   …
   Monitor.Wait(_Lock);
   or
   Monitor.Pulse(_Lock);
   Monitor.PulseALL(_Lock);
   …
}
```

# C#

- Semaphore

เข้าไป ทำพร้อมกันได้

P ลดค่า 1

V เพิ่มค่า 1

```
Semaphore s = new Semaphore(1, 1);

s.WaitOne();   // P()
  …
  …
  …
s.Release(); //V()
```

# Race Condition

- ไม่แชร์ Resource ก็ไม่เกิดการแย่งกัน
- ถ้าไม่เข้าถึง Resource ก็ทำอะไรไม่ได้

Share resource

ส่วนใหญ่สุดแล้ว ตัวเขียว

เช่น ซื้อนม → นม

case study → count queue


```
if (LockA.aggr())
{
    if (LockB.aggr())
    {
        if (Lock C.gm())
        {
        }
        else { LockA.relese();
               LockB.rem();
        }
    else..
```

```
else
{ LockA.relese();
}
```

วนลูปจนกว่าจะ acquire ครบทั้งหมด
ถึงจะทำงาน หาก ไม่ได้ release แล้ว รอ ถึงจะ
แย่งเพื่อ acquire ใหม่จนกว่าจะสำเร็จ

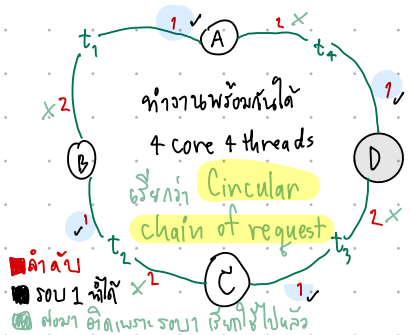
## wait while holding

ถือ Lock เข้าไป wait/sleep ไม่ได้ใช้
คนอื่นมาใช้งานไม่ได้

```
lock (_LockA)
{
    ⋮
    thead.wait();
    ⋮
}
```

```
lock (_LockA)
{
    ⋮
    thead.signal();
    ⋮
}
```

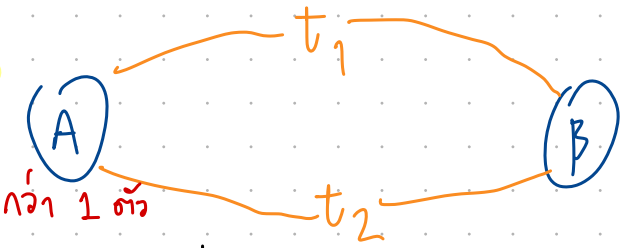
△ ดูเหมือนว่า Dead Lock ค่าสีทรงเปลี่ยน
ตลอดเวลา แต่มีสีทรงค้างค่าไว้ (error)




## อีกอัน

Dead Lock

กรณีมี share resource มากกว่า 1 ตัว




t1 ก่อน →

```
lock(_LockA) ✓ ✓
{
    lock(_LockB) ✗ ✗
    {
        ~~~
    }
}
```

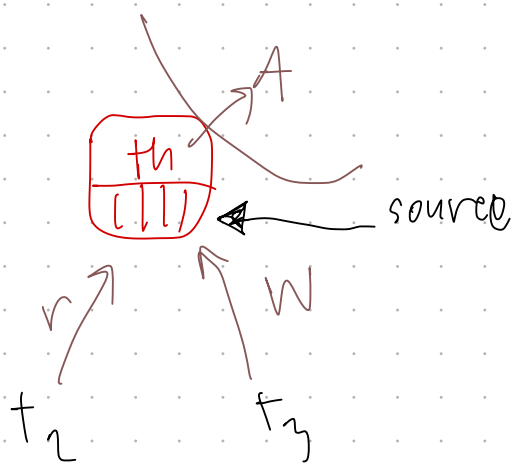
t2

```
lock(_LockB) ✓ ✓
{
    lock(_LockA) ✗ ✓
    {
        ~~~
    }
}
```

t2 thread

แก้โดย acquire ให้ LockA ก่อนแล้ว
B จะได้สำเร็จ หากเกิด context switch

# หลีกเลี่ยงการใช้ Lock




th เข้าถึง Resource A ได้ตัวเดียว

$t_2, t_3$ ต้องติดต่อ th โดยเฉพาะ