

# Synchronization



**Operating Systems**  
**Wenbo Shen**

# Background

---

- Processes/threads can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in **data inconsistency**
  - data consistency requires orderly execution of cooperating processes

# An example

---

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 int counter = 0;
6 static int loops = 1e6;
7 /*pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER; */
8
9 void *worker(void *arg) {
10     int i;
11     printf("%s: begin\n", (char*)arg);
12     for(i = 0; i < loops; i++) {
13         /*pthread_mutex_lock(&pmutex);*/
14         counter++;
15         /*pthread_mutex_unlock(&pmutex);*/
16     }
17     printf("%s: done\n", (char *)arg);
18     return NULL;
19 }
20
21 int main() {
22     pthread_t p1, p2;
23
24     printf("main: begin (counter = %d)\n", counter);
25     pthread_create(&p1, NULL, worker, "A");
26     pthread_create(&p2, NULL, worker, "B");
27     pthread_join(p1, NULL);
28     pthread_join(p2, NULL);
29     printf("main: done with both (counter : %d)\n", counter);
30     return 0;
31 }
```

# Example

---

```
wenbo@parallels:~/os-course$ gcc sync_thread.c -lpthread
wenbo@parallels:~/os-course$ ./a.out
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter : 1057072)
wenbo@parallels:~/os-course$ ./a.out
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done with both (counter : 1031264)
```

Why?

# Uncontrolled Scheduling

- counter = counter + 1

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	(after instruction)		
			PC	%eax	counter
	<i>before critical section</i>		100	0	50
	mov 0x8049a1c, %eax		105	<b>50</b>	50
	add \$0x1, %eax		108	<b>51</b>	50
<b>interrupt</b>					
	<i>save T1's state</i>				
	<i>restore T2's state</i>		100	0	50
		mov 0x8049a1c, %eax	105	<b>50</b>	50
		add \$0x1, %eax	108	<b>51</b>	50
		mov %eax, 0x8049a1c	113	51	<b>51</b>
<b>interrupt</b>					
	<i>save T2's state</i>				
	<i>restore T1's state</i>		108	51	51
	mov %eax, 0x8049a1c		113	51	<b>51</b>

**counter: 51 instead of 52!**

# Race Condition

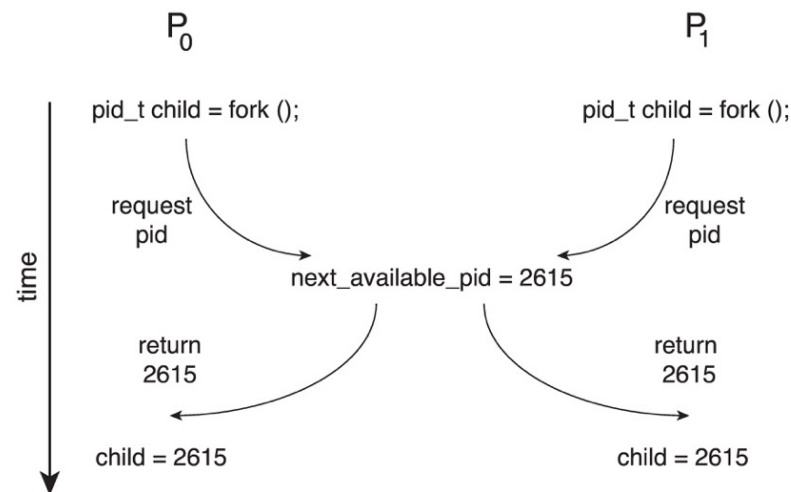
---

- Several processes (or threads) access and manipulate the same data **concurrently** and the outcome of the execution depends on the **particular order** in which the access takes place, is called a **race-condition**

# Race Condition in Kernel

---

- Processes P0 and P1 are creating child processes using the fork() system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

# Critical Section

---

- Consider system of  $n$  processes/threads  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has a critical section segment of code
  - e.g., to change common variables, update table, write file, etc.
- Only one process can be in the critical section
  - when one process in critical section, no other may be in its **critical section**
  - each process must ask permission to enter critical section in **entry section**
  - the permission should be released in **exit section**
  - **Remainder section**



# Critical Section

---

- General structure of process  $p_i$  is

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

# Critical-Section Handling in OS

---

- Single-core system: preventing interrupts
- Multiple-processor: preventing interrupts are not feasible
- Two approaches depending on if kernel is ***preemptive or non-preemptive***
  - Preemptive – allows preemption of process when running in kernel mode
  - Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

# Solution to Critical-Section: Three Requirements

---

- **Mutual Exclusion**
  - only one process can execute in the critical section
- **Progress**
  - if no process is executing in its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely
- **Bounded waiting**
  - There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - It prevents starvation

# Solution to Critical-Section: Three Requirements

---

- **Mutual Exclusion** (互斥访问)
  - 在同一时刻，最多只有一个线程可以执行临界区
- **Progress** (空闲让进)
  - 当没有线程在执行临界区代码时，必须在申请进入临界区的线程中选择一个线程，允许其执行临界区代码，保证程序执行的进展
- **Bounded waiting** (有限等待)
  - 当一个进程申请进入临界区后，必须在有限的时间内获得许可并进入临界区，不能无限等待

# Peterson's Solution

---

- Peterson's solution solves **two-processes/threads** synchronization
  - Only works for two processes case
- It assumes that LOAD and STORE are **atomic**
  - **atomic**: execution cannot be interrupted
  - Usually, the atomicity cannot be guaranteed by hardware automatically
    - Need to use special instructions
- Two processes share two variables
  - boolean **flag[2]**: whether a process is ready to enter the critical section
  - int **turn**: whose turn it is to enter the critical section

# Peterson's Solution

---

```
flag[0] = FALSE;  
flag[1] = FALSE;
```

- **P<sub>0</sub>:**

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while (flag[1] && turn == 1);  
    critical section  
    flag[0] = FALSE;  
    remainder section  
} while (TRUE);
```

Mark self ready

Assert the other one

- **P<sub>1</sub>:**

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] && turn == 0);  
    critical section  
    flag[1] = FALSE;  
    remainder section  
} while (TRUE);
```

# Peterson's Solution

---

- **Mutual exclusion** is preserved
  - P0 enters CS:
    - Either flag[1]=false or turn=0
    - Now prove P1 will not be able to enter CS
  - Case 1: flag[1]=false -> P1 is out CS
  - Case 2: flag[1]=true, turn=1 -> P0 is looping, contradicts with P0 is in CS
  - Case 3: flag[1]=true, turn=0 -> P1 is looping

- **P<sub>0</sub>:**

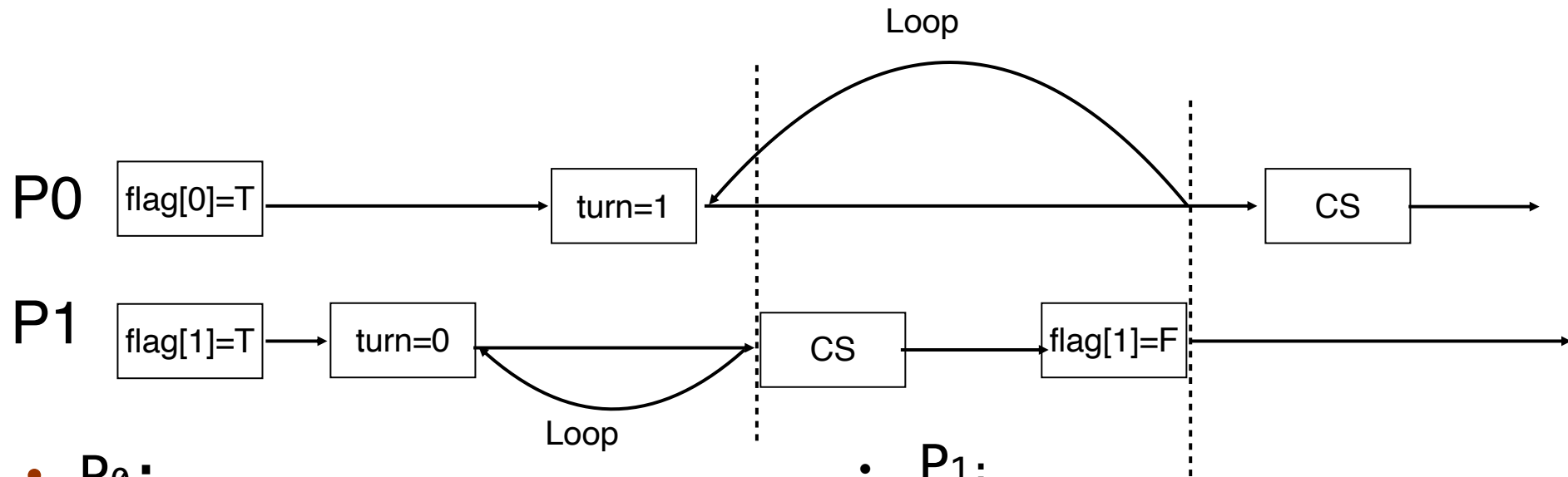
```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while (flag[1] && turn == 1);  
    critical section  
    flag[0] = FALSE;  
    remainder section  
} while (TRUE);
```

- **P<sub>1</sub>:**

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while (flag[0] && turn == 0);  
    critical section  
    flag[1] = FALSE;  
    remainder section  
} while (TRUE);
```

# Peterson's Solution

- Progress requirement



- P<sub>0</sub>:**

```
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = FALSE;
    remainder section
} while (TRUE);
```

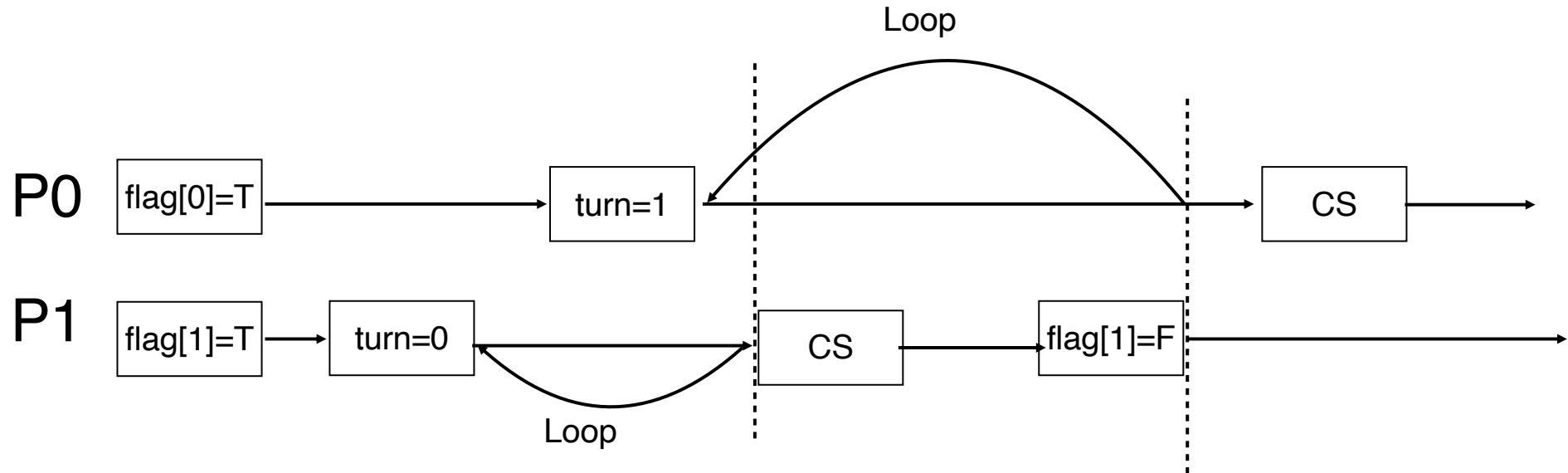
- P<sub>1</sub>:**

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = FALSE;
    remainder section
} while (TRUE);
```



# Peterson's Solution

- Bounded waiting



Whether P0 enters CS depends on P1  
Whether P1 enters CS depends on P0  
not others

P0 will enter CS after **one limited entry** P1

# Peterson's Solution

---

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on **modern architectures**.
- Understanding why it will not work is also useful for better understanding race conditions.
  - Only works for two processes case
  - It assumes that LOAD and STORE are **atomic**
  - Instruction reorder
- To improve performance, processors and/or compilers may **reorder operations** that have no dependencies.
  - For **single-threaded**, this is ok as the result will always be the same.
  - For **multi-threaded**, the **reordering may produce inconsistent** or unexpected results!

# Peterson's Solution

---

Two threads share the data:

```
boolean flag = false;
```

```
int x = 0;
```

- Thread 1 performs
  - while (!flag) ;
  - print x
- Thread 2 performs
  - x = 100;
  - flag = true

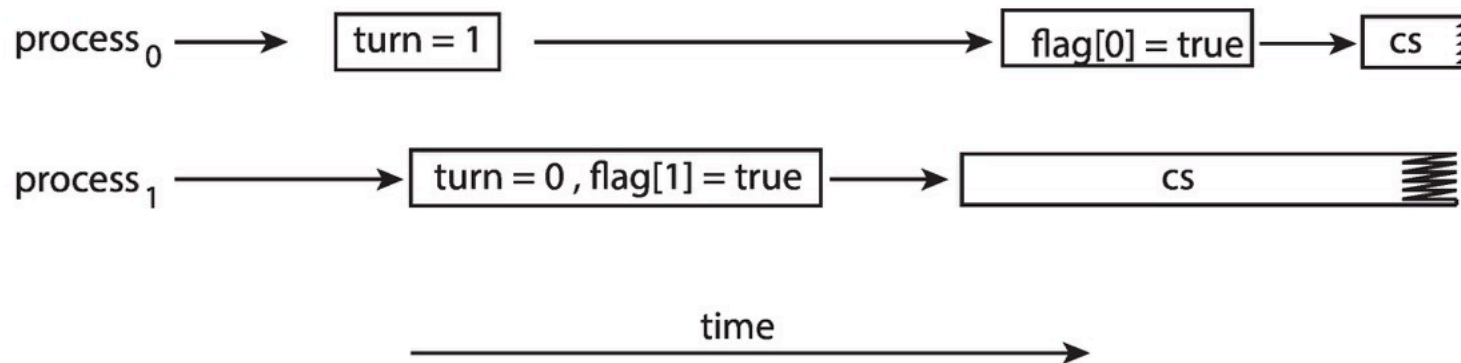
What is the expected output?

# Peterson's Solution

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

```
flag = true;  
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



# Hardware Support for Synchronization

---

- Many systems provide hardware support for critical section code
- **Uniprocessors: disable interrupts**
  - currently running code would execute without preemption
  - generally too inefficient on multiprocessor systems
    - need to disable all the interrupts
    - Other cores may still access critical section
    - operating systems using this not scalable
- Solutions:
  - 1. **Memory barriers**
  - 2. **Hardware instructions**
    - **test-and-set**: either test memory word and set value
    - **compare-and-swap**: compare and swap contents of two memory words
  - 3. **Atomic variables**


# Memory Barriers

---

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

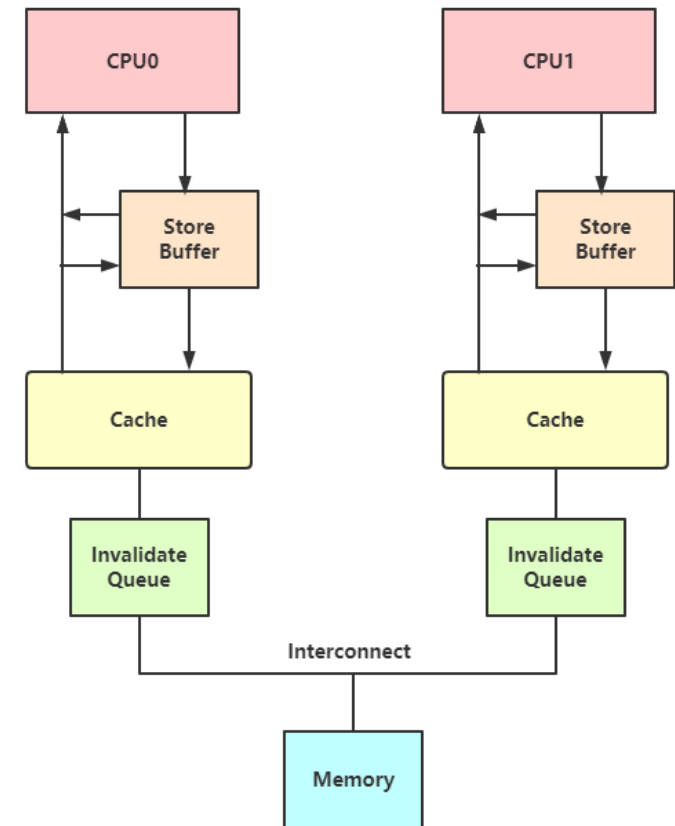
# Memory Barriers

---

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
  - Thread 1 now performs
    - `while (!flag);`
    - `memory_barrier();`
    - `print x`
  - Thread 2 now performs
    - `x = 100;`
    - `memory_barrier();`
    - `flag = true`
- 
- Ensure the value of x is not read prior to seeing the change of flag.
- Ensure that the new value of x is visible to other processors prior to the change of flag.

# Memory Barriers

- Memory barriers with multicore and cache can be very complex
  - <https://javamana.com/2021/02/20210202111159014u.html>
- X86 support
  - 写内存屏障(Store Memory Barrier): 在指令后插入 Store Barrier, 能让写入缓存中最新数据更新写入内存中, 让其他线程可见。强制写入内存, 这种显示调用, 不会让CPU去进行指令重排序
  - 读内存屏障(Load Memory Barrier): 在指令后插入 Load Barrier, 可以让高速缓存中的数据失效, 强制重新从内存中加载数据。也是不会让CPU去进行指令重排
  - <https://www.bsmax.com/A/Vx5MBRyadN/>





# Hardware Instructions

---

- Special hardware instructions that allow us to either test-and-modify the content of a word, or two swap the contents of two words atomically (uninterruptable).
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

# Test-and-Set Instruction

---

- Defined as below, but **atomically**
- Set and return old value

```
bool test_set (bool *target)
{
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

# Lock with Test-and-Set

---

- One example

```
bool lock = FALSE
```

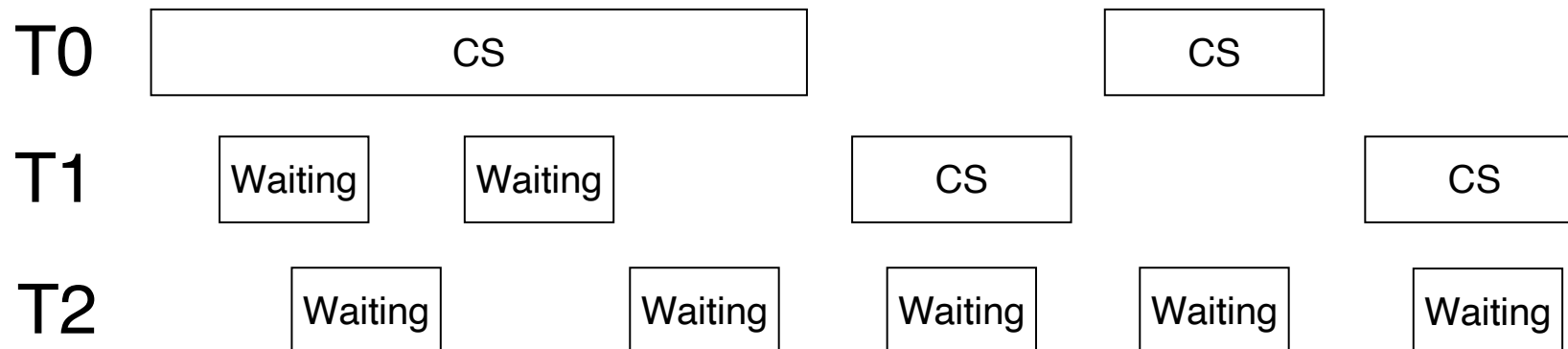
```
do {  
    while (test_set(&lock)); // busy wait  
    critical section  
    lock = FALSE;  
    remainder section  
} while (TRUE);
```

- How about
  - mutual exclusion?
  - progress?
  - bounded-waiting? Why?

# Bounded Waiting for Test-and-Set Lock

```
do {
    while (test_set(&lock));    // busy wait
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Suppose we have three threads



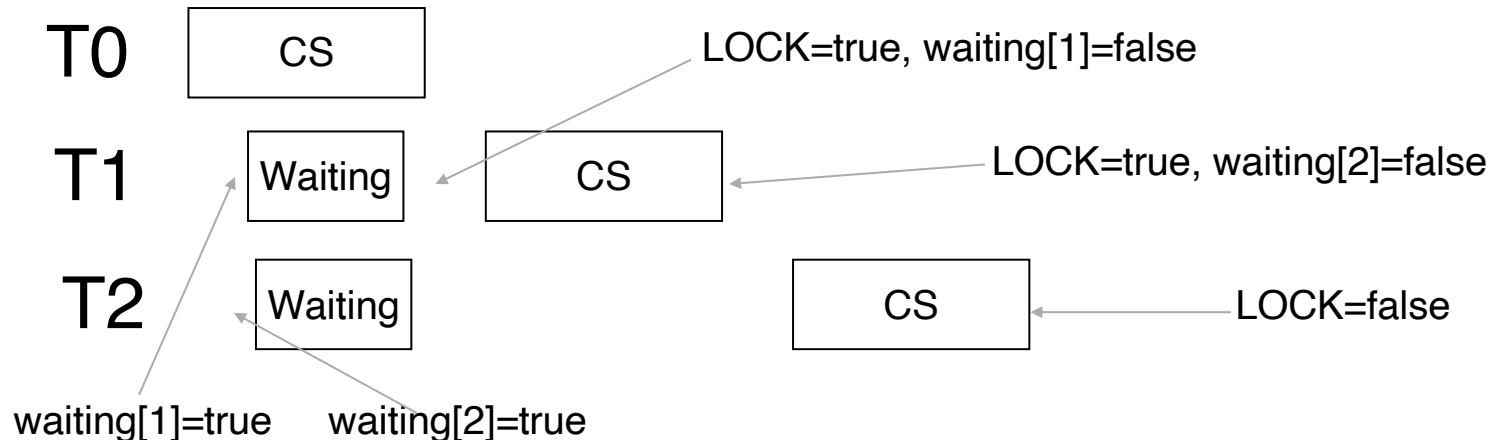
# Bounded Waiting for Test-and-Set Lock

WaitingT1

```
do {  
    waiting[i] = true;  
    while (waiting[i] && test_and_set(&lock)) ;  
    waiting[i] = false;  
  
    /* critical section */  
  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = false;  
    else  
        waiting[j] = false;  
  
    /* remainder section */  
} while (true);
```

Find next waiting = true  
Not setting lock=false

Pass the lock to next



# Compare-and-Swap Instruction

---

- Definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

- Executed atomically
- Returns the original value of passed parameter value
- Set the variable value the value of the passed parameter new\_value but only if \*value == expected is true. That is, the swap takes place only under this condition.

# Solution using Compare-and-Swap

---

- Shared integer lock initialized to 0;
- Solution:

```
while (true)
{
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;
    /* remainder section */
}
```

# Compare-and-Swap in Practice: x86

---

On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```



# Compare-and-Swap in Practice: ARM64

- ARM64 does not have cmpxchg

**Table B2-3 Synchronization primitives and associated instruction, A64 instruction set**

Transaction size	Additional semantics	Load-Exclusive <sup>a</sup>	Store-Exclusive <sup>a</sup>	Other <sup>a</sup>
Byte	-	LDXRB	STXRB	-
	Load-Acquire/Store-Release	LDAXRB	STLXRB	-
Halfword	-	LDXRH	STXRH	-
	Load-Acquire/Store-Release	LDAXRH	STLXRH	-
Register <sup>b</sup>	-	LDXR	STXR	-
	Load-Acquire/Store-Release	LDAXR	STLXR	-
Pair <sup>b</sup>	-	LDXP	STXP	-
	Load-Acquire/Store-Release	LDAXP	STLXP	-
None	Clear-Exclusive	-	-	CLREX

# Compare-and-Swap in Practice: ARM64

- ARM does not have cmpxchg

thread 1	thread 2	local monitor状态
		Open Access state
LDXR		Exclusive Access state
	LDXR	Exclusive Access state
	Modify	Exclusive Access state
	STXR	Open Access state
Modify		Open Access state
STXR		在Open Access state状态, 执行STXR指令失败
		保持Open Access state, 直到下一个LDXR指令

# Compare-and-Swap in Practice: ARM64

thread 1	thread 2	thread 3	local monitor状态
			Open Access
LDXR			Exclusive Access
	LDXR		Exclusive Access
	Modify		Exclusive Access
	STXR		Open Access
		LDXR	Exclusive Access
		Modify	Exclusive Access
Modify			Exclusive Access
STXR			Open Access (No Failure?)
		STXR	

- The Load-Exclusive instruction reads a value from memory address  $x$ .
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address  $x$  only if no other observer, process, or thread has performed a more recent store to address  $x$ . The Store-Exclusive instruction returns a status bit that indicates whether the memory write succeeded.

# Review

---

- Data inconsistency: orderly execution
- Race condition: outcome depends on the order
- Critical section: change some data ...
- Mutual exclusion, progress, bounded waiting
- Peterson's solution: mutual exclusion, progress, bounded waiting
  - Does not work on modern computer, why?
- Hardware support:
  - test\_and\_set
  - compare\_and\_swap

# Atomic Variables

---

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the `increment()` operation on the atomic variable `sequence` ensures `sequence` is incremented without interruption:
- **`increment(&sequence);`**

# Atomic Variables

---

- The increment() function can be implemented as follows:

```
void increment(atomic_int *v) {  
    int temp;  
  
    do {  
        temp = *v;  
    } while (temp != (compare_and_swap(v, temp, temp+1)));  
}
```

# Mutex Locks

---

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

# Mutex Locks

---

```
while (true) {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
}
```



# Mutex Lock Definitions

---

```
bool locked = false;
```

```
acquire() {  
    while (compare_and_swap(&locked, false, true))  
        ; //busy waiting  
}
```

```
release() {  
    locked = false;  
}
```

- These two functions must be implemented atomically.
  - Both test-and-set and compare-and-swap can be used to implement these functions.

# Too Much Spinning

---

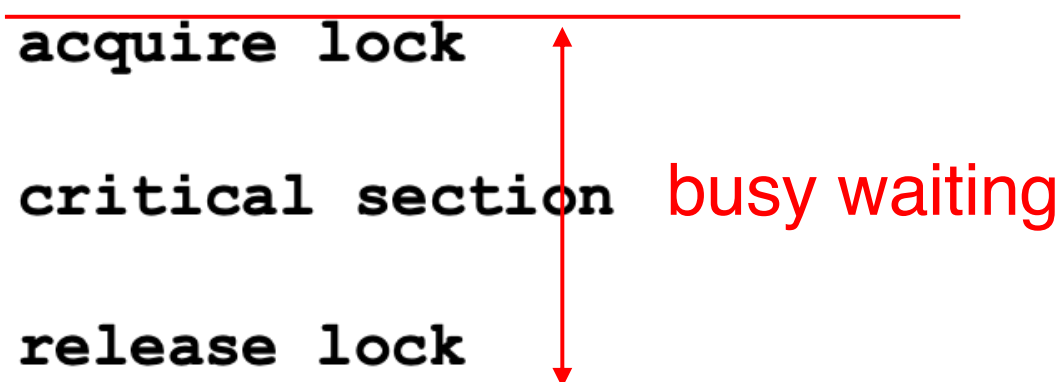
- Two threads on a single processor
  - T0 acquires lock -> INTERRUPT->T1 runs, spin, spin spin ...  
-> INTERRUPT->T0 runs -> INTERRUPT->T1 runs, spin,  
spin spin ...INTERRUPT-> T0 runs, release locks -  
>INTERRUPT->T1 runs, enters CS
  - T1 busy waits all its CPU time until T0 releases lock
  - T0 holds the lock, but does not get CPU
  - What if we have N threads?
    - N-1 threads loops in all their CPU time
    - A huge waste of CPU time

# Busy waiting time

---

- Mutex busy waiting time
  - From acquire to release

```
while (true) {  
    acquire lock  
  
    critical section busy waiting  
  
    release lock  
  
    remainder section  
}
```

A diagram illustrating busy waiting. Two horizontal red lines are drawn. The top line is aligned with the 'acquire lock' statement, and the bottom line is aligned with the 'release lock' statement. A vertical red double-headed arrow connects these two lines, spanning the duration of the 'critical section' and the 'remainder section'. To the right of the arrow, the text 'busy waiting' is written in red.

- What if the critical section is long?
  - A huge waste of CPU time

# Reduce busy waiting – just yield

---

- Can we put the busy waiting thread into suspended?
  - yield-> moving from running to sleeping

```
void init() {  
    flag = 0;  
}
```

```
void lock() {  
    while (test_set(&flag, 1) == 1)  
        yield(); // give up the CPU  
}
```

```
void unlock() {  
    flag = 0;  
}
```

- How to implement?
  - Add a queue
  - When the lock is locked, change process's state to SLEEP, add to the queue, and call schedule()

# Semaphore

---

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore
  - Contain **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()** (Originally called **P()** and **V()** Dutch)
- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0) ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```

# Semaphore

---

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$   
Create a semaphore “**sem**” initialized to 0

**P1:**

```
S1;  
signal(sem);
```

**P2:**

```
wait(sem);  
S2;
```

- Can implement a counting semaphore **S** as a binary semaphore

# Semaphore w/ waiting queue

---

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

```
typedef struct {  
    int value;  
    struct list_head * waiting_queue;  
} semaphore;
```

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

# Implementation with waiting queue

---

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a proc.P from S->list;  
        wakeup(P);  
    }  
}
```



# Semaphore w/ waiting queue

---

```
Semaphore sem;    // initialized to 1
do {
    wait (sem);
    critical section    No busy waiting on critical section
    signal (sem);
    remainder section
} while (TRUE);    //while loop but not busy waiting
```

- No busy waiting on critical section
  - Why?
  - Save lots of CPU time

# Comparison between mutex and semaphore

---

- Mutex or spinlock
  - Pros: no blocking
  - Cons: Waste CPU on looping
  - Good for short critical section
- Semaphore
  - Pros: no looping
  - Cons: context switch is time-consuming
  - Good for long critical section

# Implementation with waiting queue

---

- Should *wait* or *signal* be atomic?
- If so? How to achieve the atomic?
  - Spinlock

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a proc.P from S->list;
        wakeup(P);
    }
}
```

# Semaphore w/ waiting queue

```
Semaphore sem;    // initialized to 1
do {
    wait (sem);    ↕ busy waiting
    critical section  No busy waiting on critical section
    signal (sem);  ↕ busy waiting
    remainder section
} while (TRUE);    //while loop but not busy waiting
```

- No busy waiting on critical section
- Still has the busy waiting on *wait* and *signal*
  - But waiting is much shorter

# Semaphore w/ waiting queue in practice

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

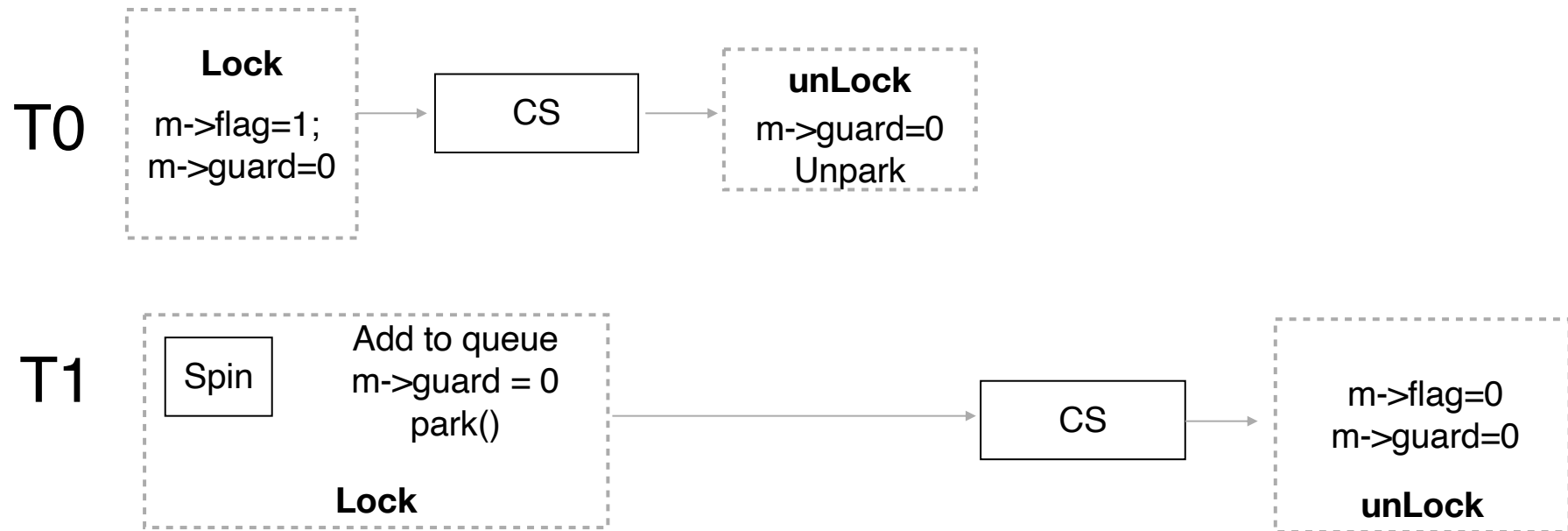
Spinlock to protect m->flag

schedule()

Spinlock to protect m->flag

# Semaphore w/ waiting queue in practice

- A program: lock() -> CS -> unlock



- Note that we have a small cs (previous page) to protect the `m->flag`, and a bigger CS of the program.

# Deadlock and Starvation

---

- **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

let S and Q be two semaphores initialized to 1

P0	P1
wait (S);	wait (Q);
wait (Q);	wait (S);
...	...
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation:** indefinite blocking
  - a process may **never be** removed from the semaphore's waiting queue
  - does starvation indicate deadlock?

# Priority Inversion

---

- **Priority Inversion:** a higher priority process is **indirectly** preempted by a lower priority task
  - Low priority task holds the lock, but never gets CPU due to low priority, therefore can never finish and release lock
  - High priority task waits the lock, forever



# Priority Inversion

---

- **Priority Inversion:** a higher priority process is **indirectly** preempted by a lower priority task
  - e.g., three processes, **P<sub>L</sub>**, **P<sub>M</sub>**, and **P<sub>H</sub>** with priority  $P_L < P_M < P_H$
  - **P<sub>L</sub>** holds a lock that was requested by **P<sub>H</sub>**  $\Rightarrow$  **P<sub>H</sub>** is blocked
  - **P<sub>M</sub>** becomes ready and preempted the **P<sub>L</sub>**
  - It effectively "inverts" the relative priorities of **P<sub>M</sub>** and **P<sub>H</sub>**
- Solution: **priority inheritance**
  - temporary assign the highest priority of waiting process (**P<sub>H</sub>**) to the process holding the lock (**P<sub>L</sub>**)

# Linux Synchronization

---

- Linux:
  - prior to version 2.6, disables interrupts to implement short critical sections
  - version 2.6 and later, fully preemptive
- Linux provides:
  - **atomic integers**
  - **spinlocks**
  - **semaphores**
    - on single-cpu system, spinlocks replaced by enabling/disabling kernel preemption
  - **reader-writer locks**

# Linux Synchronization

---

- Atomic variables
  - [linux/include/linux/atomic.h](#)
  - [linux/include/asm-generic/atomic.h](#)
  - `atomic_t` is the type for atomic integer
- Simple operations
  - `atomic_read()`, `atomic_set()`, ...
- Arithmetic
  - `atomic_{add,sub,inc,dec}()`
  - `atomic_{add,sub,inc,dec}_return{,_relaxed,_acquire,_release}()`
  - `atomic_fetch_{add,sub,inc,dec}{,_relaxed,_acquire,_release}()`

[https://github.com/torvalds/linux/blob/master/Documentation/atomic\\_t.txt](https://github.com/torvalds/linux/blob/master/Documentation/atomic_t.txt)

# Linux Synchronization

---

- Bitwise
  - `atomic_{and,or,xor,andnot}()`
  - `atomic_fetch_{and,or,xor,andnot}{,_relaxed,_acquire,_release}()`
- Swap
  - `atomic_xchg{,_relaxed,_acquire,_release}()`
  - `atomic_cmpxchg{,_relaxed,_acquire,_release}()`
  - `atomic_try_cmpxchg{,_relaxed,_acquire,_release}()`
- Others
  - Reference count, Misc

[https://github.com/torvalds/linux/blob/master/Documentation/atomic\\_t.txt](https://github.com/torvalds/linux/blob/master/Documentation/atomic_t.txt)

# Linux Synchronization

---

- Spinlock
  - [linux/include/linux/spinlock.h](#)
  - Multiple lock/unlock operations supported
- Linux spinlock – x86 implementation

```
ffffffff81a35c00 <_raw_spin_lock>:
ffffffff81a35c00:    31 c0                xor    %eax,%eax
ffffffff81a35c02:    ba 01 00 00 00      mov    $0x1,%edx
ffffffff81a35c07:    f0 0f b1 17          lock cmpxchg %edx,(%rdi)
ffffffff81a35c0b:    75 02                jne    ffffffff81a35c0f <_raw_spin_lock+0xf>
ffffffff81a35c0d:    f3 c3                repz retq
ffffffff81a35c0f:    89 c6                mov    %eax,%esi
ffffffff81a35c11:    e9 5a f8 66 ff       jmpq   ffffffff810a5470 <queued_spin_lock_slowpath>
ffffffff81a35c16:    66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
ffffffff81a35c1d:    00 00 00
```

# Linux Synchronization

- Linux spinlock – ARM64 implementation

```
ffffff8008a98e78 <_raw_spin_lock>:
ffffff8008a98e78:      a9bf7bfd      stp     x29, x30, [sp, #-16]!
ffffff8008a98e7c:      aa0003e3      mov     x3, x0
ffffff8008a98e80:      d5384102      mrs     x2, sp_el0
ffffff8008a98e84:      910003fd      mov     x29, sp
ffffff8008a98e88:      b9401041      ldr     w1, [x2, #16]
ffffff8008a98e8c:      11000421      add     w1, w1, #0x1
ffffff8008a98e90:      b9001041      str     w1, [x2, #16]
ffffff8008a98e94:      d2800001      mov     x1, #0x0                                // #0
ffffff8008a98e98:      d2800022      mov     x2, #0x1                                // #1
ffffff8008a98e9c:      97ff8f53      bl     fffffff8008a7cbe8 <__ll_sc___cmpxchg_case_acq_4>
ffffff8008a98ea0:      d503201f      nop
ffffff8008a98ea4:      d503201f      nop
ffffff8008a98ea8:      35000060      cbnz    w0, fffffff8008a98eb4 <_raw_spin_lock+0x3c>
ffffff8008a98eac:      a8c17bfd      ldp     x29, x30, [sp], #16
ffffff8008a98eb0:      d65f03c0      ret
ffffff8008a98eb4:      aa0003e1      mov     x1, x0
ffffff8008a98eb8:      aa0303e0      mov     x0, x3
ffffff8008a98ebc:      97d98c51      bl     fffffff80080fc000 <queued_spin_lock_slowpath>
ffffff8008a98ec0:      a8c17bfd      ldp     x29, x30, [sp], #16
ffffff8008a98ec4:      d65f03c0      ret
```

```
ffffff8008a7cbe8 <__ll_sc___cmpxchg_case_acq_4>:
ffffff8008a7cbe8:      f9800011      prfm    pstllstrm, [x0]
ffffff8008a7cbec:      885ffc10      ldaxr   w16, [x0]
ffffff8008a7cbf0:      4a010211      eor     w17, w16, w1
ffffff8008a7cbf4:      35000071      cbnz    w17, fffffff8008a7cc00 <__ll_sc___cmpxchg_case_acq_4+0x18>
ffffff8008a7cbf8:      88117c02      stxr    w17, w2, [x0]
ffffff8008a7cbfc:      35ffff91      cbnz    w17, fffffff8008a7cbec <__ll_sc___cmpxchg_case_acq_4+0x4>
ffffff8008a7cc00:      aa1003e0      mov     x0, x16
ffffff8008a7cc04:      d65f03c0      ret
```

Store failure jumps to load again

# Linux Synchronization

---

- Semaphore
  - [linux/include/linux/semaphore.h](#)

```
struct semaphore {  
    raw_spinlock_t      lock;  
    unsigned int         count;  
    struct list_head     wait_list;  
};
```

- [down\(\), up\(\)](#)

# Linux Synchronization

---

- Semaphore
  - `down()`

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

- Sleep when holding spinlock?
  - Check `__down_common()`
  - No, spinlock is released before sleeping



# Linux Synchronization

---

- Semaphore
  - up()

```
void up(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

# Linux Synchronization

---

- reader-writer locks
  - rw\_semaphore
  - RCU (from wiki)
    - **read-copy-update (RCU)** is a synchronization mechanism based on mutual exclusion. It is used when performance of reads is crucial and is an example of space–time tradeoff, enabling fast operations at the cost of more space.
    - Read-copy-update allows multiple threads to efficiently read from shared memory by deferring updates after pre-existing reads to a later time while simultaneously marking the data, ensuring new readers will read the updated data.
    - This makes all readers proceed as if there were no synchronization involved, hence they will be fast, but also making updates more difficult.

# Linux Synchronization

---

- RCU (from wiki): the name comes from the way that RCU is used to update a linked structure in place. A thread wishing to do this uses the following steps:
  - create a new structure,
  - copy the data from the old structure into the new one, and save a pointer to the old structure,
  - modify the new, copied, structure
  - update the global pointer to refer to the new structure, and then
  - sleep until the operating system kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using `synchronize_rcu()`.
  - when the thread that made the copy is awakened by the kernel, it can safely deallocate the old structure.

# POSIX Synchronization

---

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

# POSIX Mutex Locks

---

- Creating and initializing the lock

```
#include <pthread.h>
pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

# POSIX Semaphores

---

- POSIX provides two versions – **unnamed** and **named**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

# POSIX Unnamed Semaphores

---

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

# POSIX Named Semaphores

---

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```



# Condition Variables

---

When should we use condition variables?

- `while` checking is not atomic
- Cannot support multi-threading

```
void* thread_func(void *args) {  
    while (x!=10) {  
        sleep(5);  
    }  
    // 待条件成立后, 执行后续的代码  
}
```

# Condition Variables

---

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs. Condition variables are user-mode objects that cannot be shared across processes.

- `condition x;`
- Operations supported by a condition variable are:
  - `wait(condition, lock)`: release lock, put thread to sleep until condition is signaled; when thread wakes up again, re-acquire lock before returning.
  - `signal(condition, lock)`: if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.
  - `broadcast(condition, lock)`: same as signal, except wake up all waiting threads.
- Note: after signal, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.

# Condition Variables

---

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition);
- Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).
- The idea goes back to Dijkstra's use of "private semaphores";
- A similar idea was later named a "condition variable" by Hoare.

# POSIX Condition Variables

---

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;  
pthread_cond_t cond_var;  
  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cond_var, NULL);
```

- Thread waiting for the condition  $a == b$  to become true:

```
pthread_mutex_lock(&mutex);  
  
while(a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
  
pthread_mutex_unlock(&mutex);
```

release lock when wait,  
acquire lock when being  
signaled

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);  
a = b;  
pthread_cond_signal(&cond_var);  
pthread_mutex_unlock(&mutex);
```

# POSIX Condition Variables

---

- Usage scenarios:

```
pthread_mutex_t mutex;  
pthread_cond_t condition;
```

thread 1 :

```
pthread_mutex_lock(&mutex); //mutex lock  
while(!condition){  
    pthread_cond_wait(&condition, &mutex); //wait for the condition  
}  
  
/* do what you want */  
  
pthread_mutex_unlock(&mutex);
```

thread 2:

```
pthread_mutex_lock(&mutex);  
  
/* do something that may fulfill the condition */  
  
pthread_mutex_unlock(&mutex);  
pthread_cond_signal(&condition); //wake up thread 1
```

# POSIX Condition Variables

---

- Why need mutex?
- What's the difference between condition variable and semaphore?
  - What if we only care if the queue is empty or not, while don't care the queue length?
  - Condition variable can wake up all threads, semaphore can only wake up one by one

# Takeaway

---

- Data race
  - Less than 2M example
  - Reason
- Critical section
  - Three requirements
- Peterson's Solution
- Hardware Support for Synchronization
  - Memory barrier, hardware instruction, atomic variables
- Mutex lock
- Semaphore
- Linux provides:
  - atomic integers
  - spinlocks
  - semaphores
  - reader-writer locks