Synchronization

Operating Systems
Wenbo Shen

Some slides are from OSC and Yajin Zhou

Background

- Processes 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int counter = 0;
static int loops = 1e6;
/*pthread_mutex_t pmutex = PTHREAD_MUTEX_INITIALIZER; */

void *worker(void *arg) {
    int i;
    printf("%s: begin\n", (char*)arg);
    for(i = 0; i < loops; i++) {
        /*pthread_mutex_lock(&pmutex);*/
        counter++;
        /*pthread_mutex_unlock(&pmutex);*/
    }
    printf("%s: done\n", (char *)arg);
    return NULL;
}

int main() {
    pthread_t p1, p2;

    printf("main: begin (counter =  %d)\n", counter);
    pthread_create(&p1, NULL, worker, "A");
    pthread_create(&p2, NULL, worker, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter : %d)\n", counter);
    return 0;
}
```

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	PC (after instruction)	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

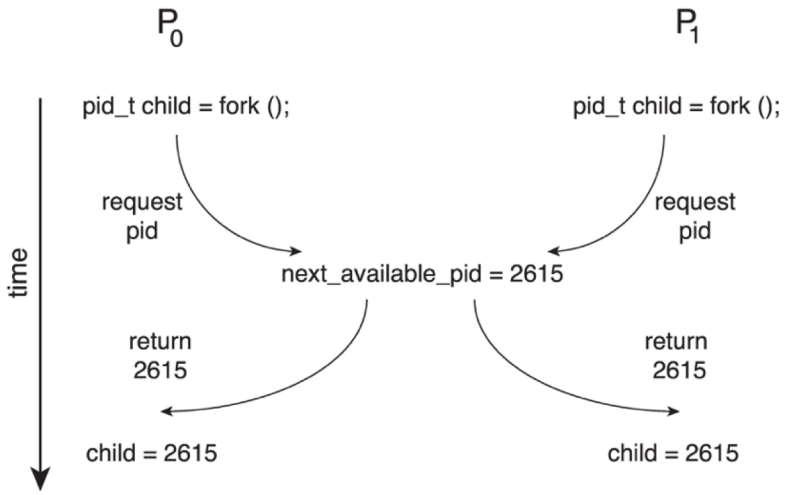
counter: 51 instead of 52!

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, … 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

https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section

Progress

- The purpose of this condition is to make sure that either some process is currently in the CS and doing some work or, if there was at least one process that wants to enter the CS, it will and then do some work. In both cases, some work is getting done and therefore all processes are **making progress** overall.

1. If no process is executing in its critical section

   If there is a process executing in its critical section (even though not stated explicitly, this includes the leave section as well), then this means that some work is getting done. So we are making progress. Otherwise, if this was not the case...

2. and some processes wish to enter their critical sections

   If no process wants to enter their critical sections, then there is no more work to do. Otherwise, if there is at least one process that wishes to enter its critical section...

Progress

3. then only those processes that are not executing in their remainder section

This means we are talking about those processes that are executing in either of the first two sections (remember, no process is executing in its critical section or the leave section)...

4. can participate in deciding which will enter its critical section next,

Since there is at least one process that wishes to enter its CS, somehow we must choose one of them to enter its CS. But who's going to make this decision? Those process who already requested permission to enter their critical sections have the right to participate in making this decision. In addition, those processes that **may** wish to enter their CSs but have not yet requested the permission to do so (this means that they are in executing in the first section) also have the right to participate in making this decision.

5. and this selection cannot be postponed indefinitely.

This states that it will take a limited amount of time to select a process to enter its CS. In particular, no deadlock or livelock will occur. So after this limited amount of time, a process will enter its CS and do some work, thereby making progress.

No process *running outside* the critical section should block the other interested process from entering into it's critical section when in fact the critical section is free.

Bounded waiting

- Bounded waiting: There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.

1. after a process has made request to enter its critical section and before that request is granted.

   In other words, if there is a process that has requested to enter its CS but has not yet entered it. Let's call this process P.

2. There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections

   While P is waiting to enter its CS, other processes may be waiting as well and some process is executing in its CS. When it leaves its CS, some other process has to be selected to enter the CS which may or may not be P. Suppose a process other than P was selected. This situation might happen again and again. That is, other processes are getting the chance to enter their CSs but never P. Note that progress is being made, but by other processes, not by P. The problem is that P is not getting the chance to do any work. To prevent starvation, there must be a guarantee that P will eventually enter its CS. For this to happen, the number of times other processes enter their CSs must be limited. In this case, P will definitely get the chance to enter its CS.

No process should have to wait forever to enter into critical section. there should be boundary on getting chances to enter into critical section. If bounded waiting is not satisfied then there is a possibility of starvation.

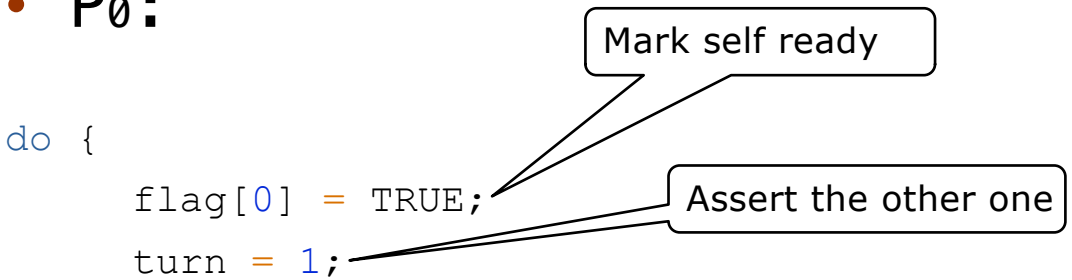
Peterson's Solution

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

Peterson's Solution

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

- P_0:




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

- P_1:

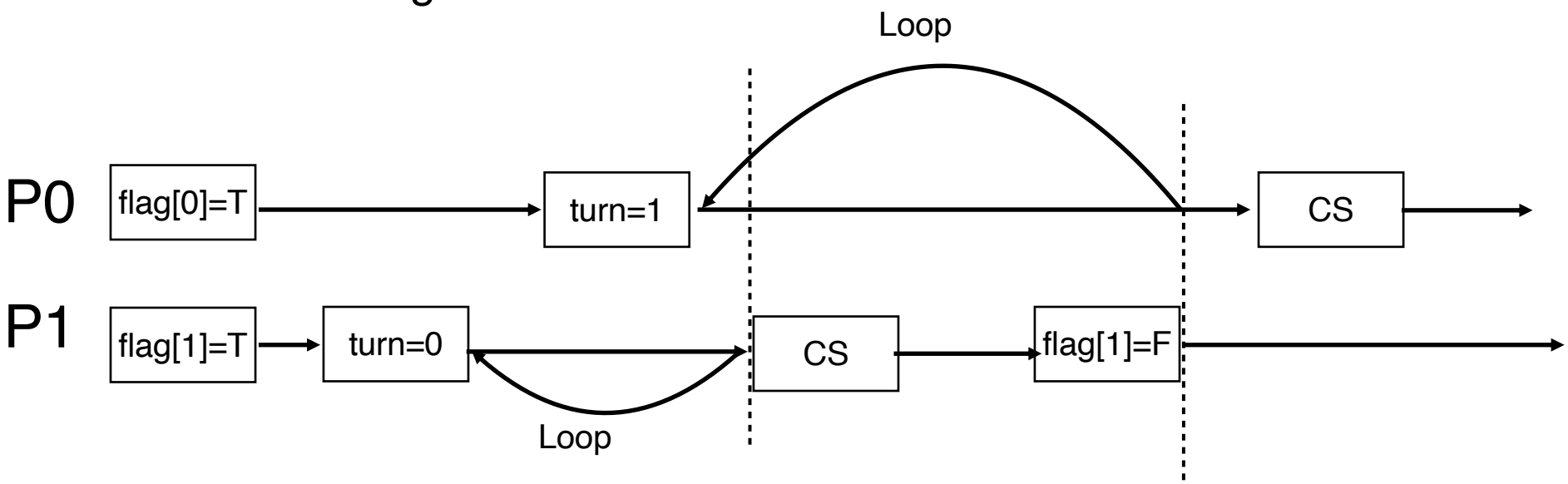
```
do {
      flag[1] = TRUE;
      turn = 0;
      while (flag[0] && turn == 0);
      critical section
      flag[1] = FALSE;
      remainder section
} while (TRUE);
```

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]=false, turn = 1 -> P1 leaves CS
  - Case 3: flag[1]=true, turn=1 -> P0 is looping, contradicts with the fact P0 is in CS
  - Case 4: flag[1]=true, turn = 0 -> P1 is looping

Peterson's Solution

- Progress requirement
- 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.
- 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 **multithreaded the reordering may produce inconsistent** or unexpected results!

Peterson's Solution

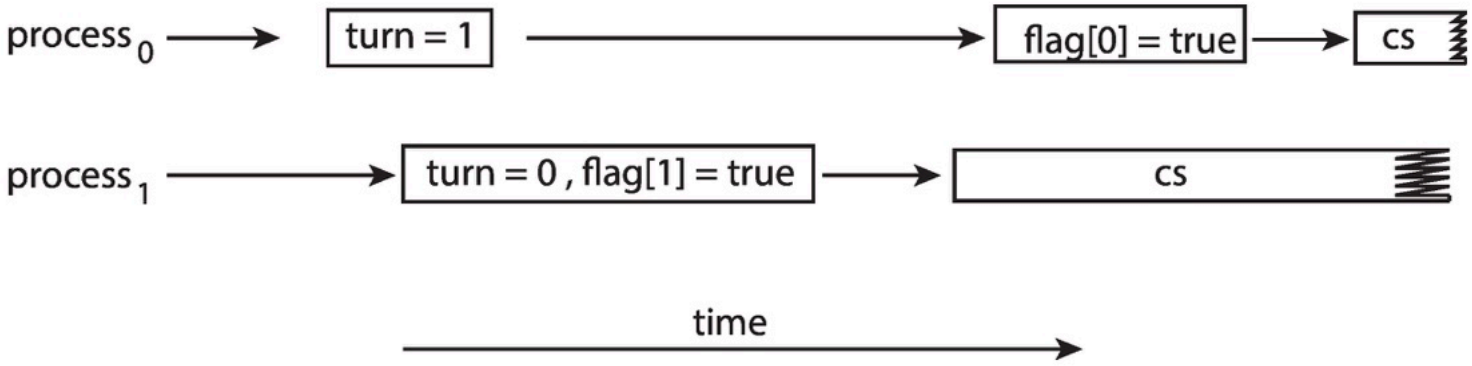
- Two threads share the data:
  - boolean flag = false;
  - int x = 0;
- Thread 1 performs
  - while (!flag)
  - ;
  - print x
- Thread 2 performs
  - x = 100;
  - flag = true
- What is the expected output?

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
    - 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

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**

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

Lock with Test-and-Set

- shared variable: bool **lock** = FALSE

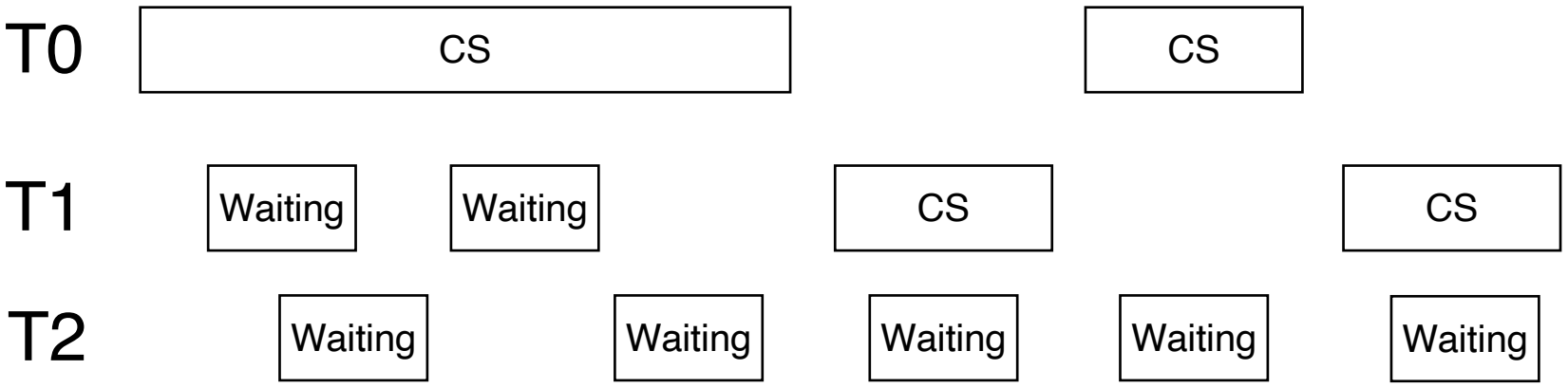
```
do {
    while (test_set(&lock));   // busy wait
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

- 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

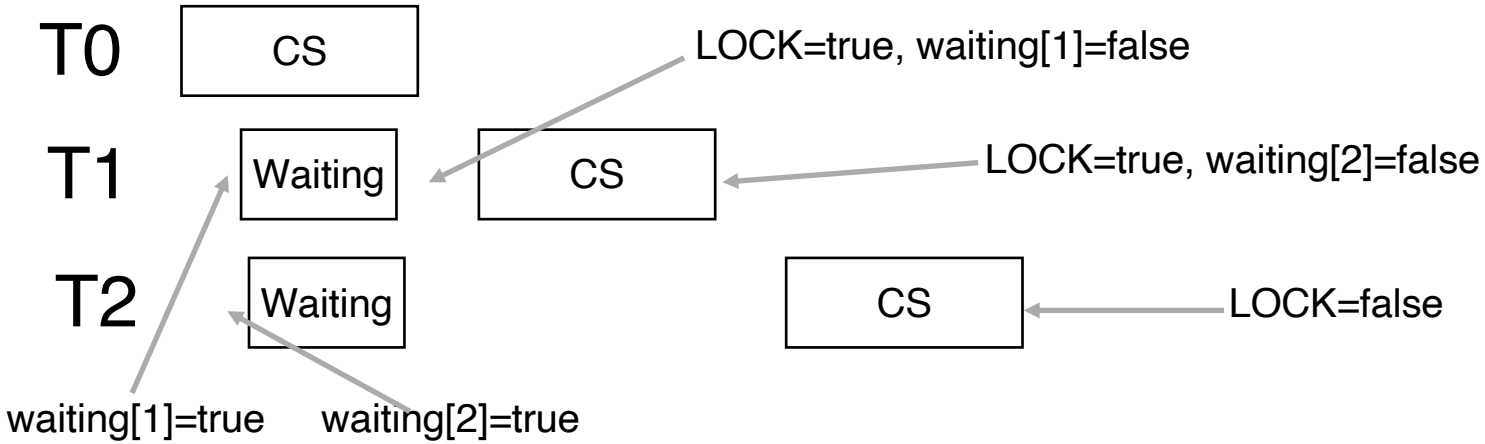
WaitingT1

```
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);
```



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;
}
```

1. Executed atomically
2. Returns the original value of passed parameter **value**
3. 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>
```


source: http://heather.cs.ucdavis.edu/~matloff/50/PLN/lock.pdf

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[a]	Store-Exclusive[a]	Other[a]
Byte	-	LDXRB	STXRB	-
	Load-Acquire/Store-Release	LDAXRB	STLXRB	-
Halfword	-	LDXRH	STXRH	-
	Load-Acquire/Store-Release	LDAXRH	STLXRH	-
Register[b]	-	LDXR	STXR	-
	Load-Acquire/Store-Release	LDAXR	STLXR	-
Pair[b]	-	LDXP	STXP	-
	Load-Acquire/Store-Release	LDAXP	STLXP	-
None	Clear-Exclusive	-	-	CLREX

Compare-and-Swap in Practice: ARM64

- ARM64 does not have cmpxchg

thread 1	thread 2	local monitor的状态
		Open Access state
LDREX		Exclusive Access state
	LDREX	Exclusive Access state
	Modify	Exclusive Access state
	STREX	Open Access state
Modify		Open Access state
STREX		在Open Access state的状态下，执行STREX指令会导致该指令执行失败
		保持Open Access state，直到下一个LDREX指令

http://www.wowotech.net/kernel_synchronization/atomic.html

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

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

- 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
  - What if we have N threads?
    - N-1 threads loops in all their CPU time
    - A huge waste of CPU time and power

Just Yield

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

yield-> moving from running to ready

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 "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore

- **How about busy waiting time**

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

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

```
                    Semaphore mutex;
do {                                    do {
    wait (mutex);                           wait (mutex);
    critical section                        critical section
    signal (mutex);                         signal (mutex);
    remainder section                       remainder section
} while (TRUE);                         } while (TRUE);
```

Semaphore

- **How about busy waiting time**

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

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

```
                    Semaphore mutex;
do {                                    do {
    wait (mutex);                           wait (mutex);
    critical section                        critical section
    signal (mutex);                         signal (mutex);
    remainder section                       remainder section
} while (TRUE);                         } while (TRUE);
```

Lock here

Unlock here

Semaphore

- **How about busy waiting time**

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

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

```
                    Semaphore mutex;
do {                                    do {
    wait (mutex);                           wait (mutex);
    critical section                        critical section
    signal (mutex);                         signal (mutex);
    remainder section                       remainder section
} while (TRUE);                     } while (TRUE);
```

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

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 process P from S->list;
        wakeup(P);
    }
}
```

Semaphore w/ waiting queue

```
Semaphore mutex;     //  initialized to 1
do {
    wait (mutex);
    critical section
    signal (mutex);
    remainder section
} while (TRUE);       //while loop but not busy waiting
```

busy waiting

```
Semaphore mutex;     //  initialized to 1
do {
    wait (mutex);
    critical section
    signal (mutex);
    remainder section
} while (TRUE);       //while loop but not busy waiting
```

busy waiting

busy waiting

Semaphore w/ waiting queue in practice

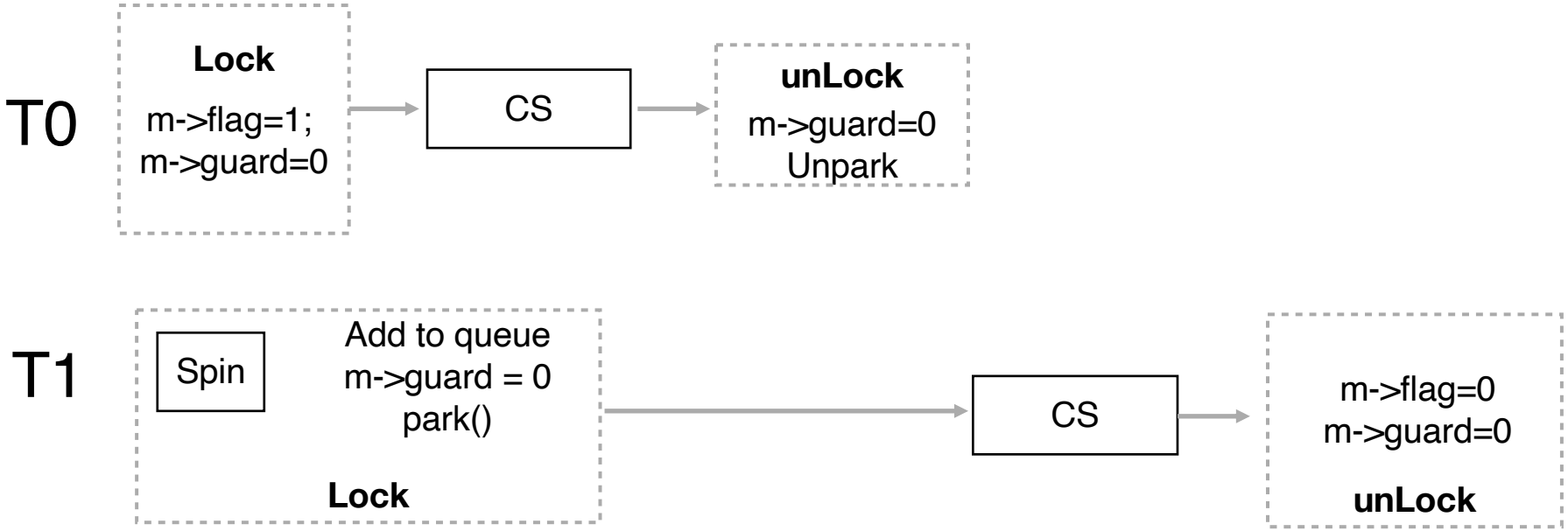
```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //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  }
```

CS to protect m->flag

CS 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?

https://techdifferences.com/difference-between-deadlock-and-starvation-in-os.html

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_L**, **P_M**, and **P_H** with priority $P_L < P_M < P_H$
  - **P_L** holds a lock that was requested by **P_H** ⇨ **P_H** is blocked
  - **P_M** becomes ready and preempted the **P_L**
  - It effectively "inverts" the relative priorities of **P_M** and **P_H**
- Solution: **priority inheritance**
  - temporary assign the highest priority of waiting process (P_H) to the process holding the lock (P_L)

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

Takeaway

- A big wave of homework is coming, including all process management:
  - Process
  - IPC
  - Thread
  - Scheduling
  - Synchronization