Synchronization Tools

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

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

```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Example

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Why?

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Uncontrolled Scheduling

- Counter = counter + 1

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

OS	Thread 1	Thread 2	(after instruction) PC	%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!

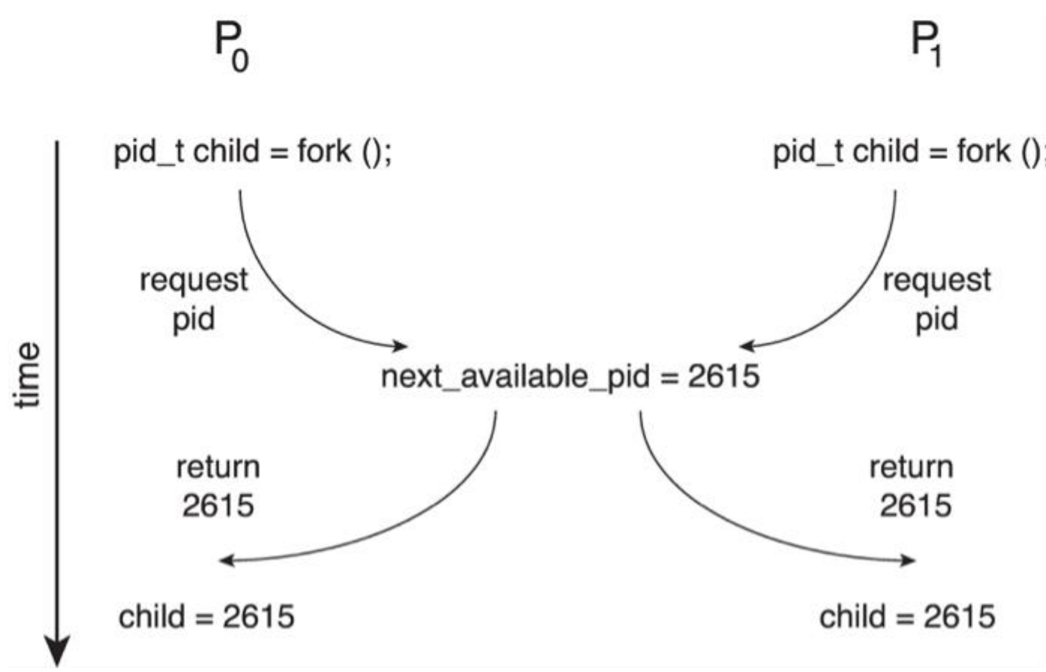
Race Condition

- 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 $\{p_0, p_1, \ldots p_{n-1}\}$
- 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

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

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 retainer 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**

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.

```
do {
    initialization          1
    entry section           2
        critical section    3
    exit section            4
        remainder section   5
} while (true);
```

- 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...

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

> 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.

> 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** synchronization
- It assumes that LOAD and STORE are **atomic**
  - **atomic**: execution cannot be interrupted
- 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

Peterson's Solution

- P_0:

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

- P_1:

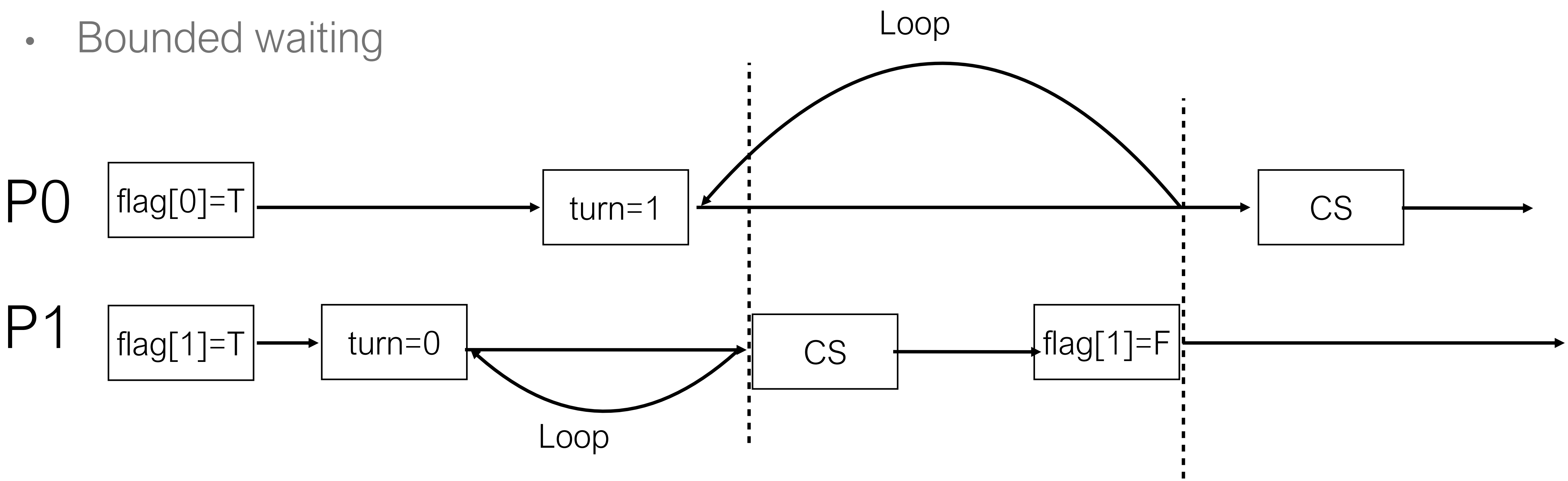
```
do {
        flag[1] = TRUE;
        turn = 0;
        while (flag[0] && (turn == 0));
        critical section
        flag[1] = FALSE;
        remainder section
} while (TRUE);
```

Peterson’s Solution

- 1. **Mutual exclusion** is preserved
- Suppose P0/P1 wants to enter the CS at the same time
- Flag[0]/flag[1] = true
- However, the value of turn will be eventually 0 or 1. In this case, one process will be waiting.
- Flag[0] = true, flag[1] = true
    - Case 1: turn = 0: P0 enter CS
    - Case 2: turn =1, P1 enter CS

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

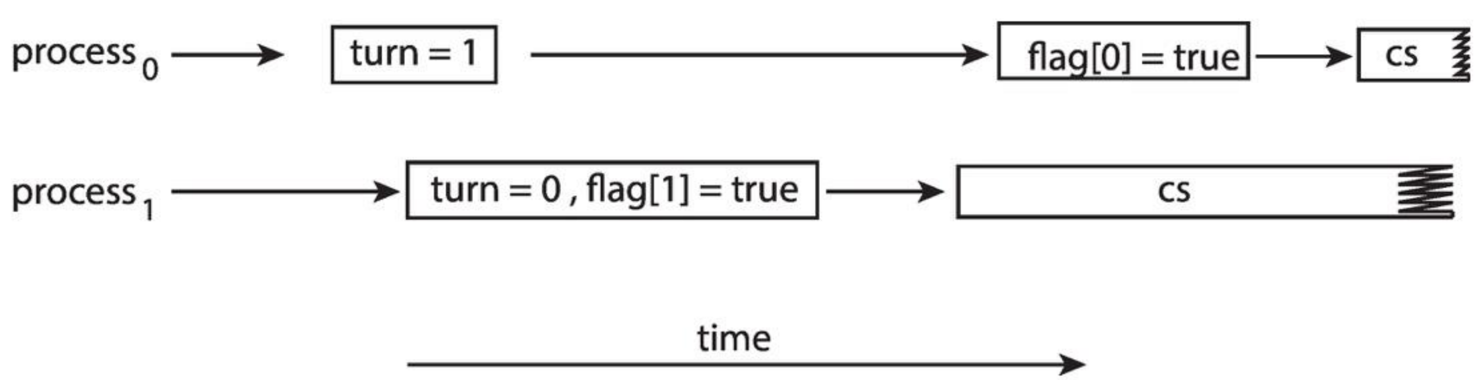
- Two threads share the data:
  - boolean flag = false;
  - int x = 0;
- Thread 1 performs
  - while (!flag);
  - print x
- Thread 2 performs
  - x = 100;
  - flag = true
- What is the expected output?

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
    - **swap**: 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 (uninterruptibly.)
- **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

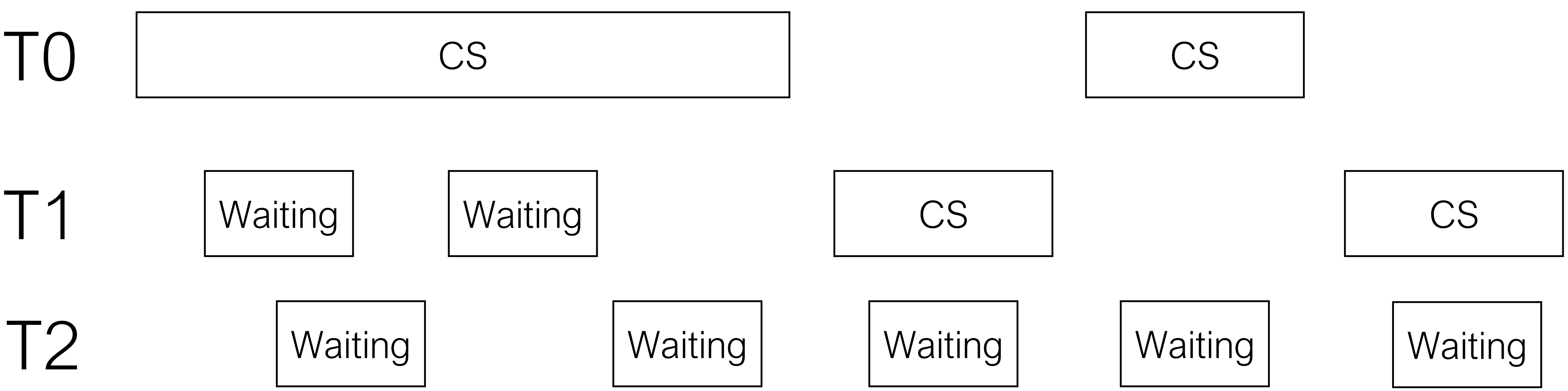
```
do {
   while (test_set(&lock));   // busy wait
   critical section
   lock = FALSE;
   remainder section
} while (TRUE);
```

- Mutual exclusion?
- progress?
- bounded-waiting? Why?

Bounded Waiting for Test-and-Set Lock

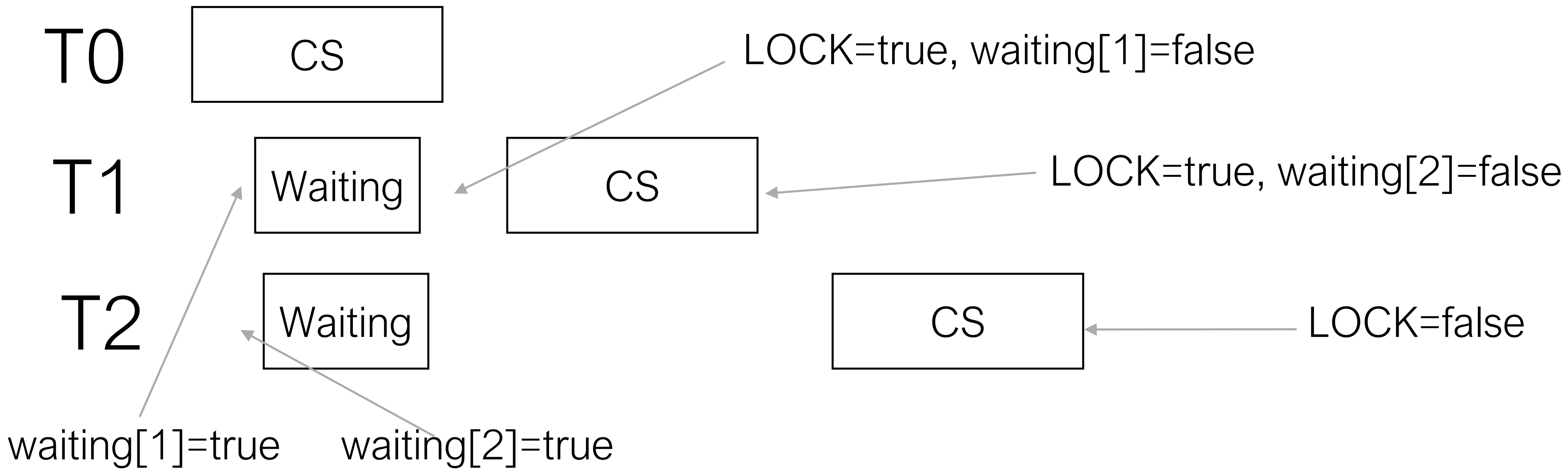
Suppose we have three threads

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



Bounded Waiting for Test-and-Set Lock

```
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 */
}
```

Review

- Data inconsistency: orderly execution
- Race condition: outcome depends on the order
- Critical section: change some data …
- Program structure: entry, critical section, exit section, remainder section
- Mutual exclusion, progress, bounded waiting
- Peterson’s solution: ME, progress, bounded waiting
  - Does not work on modern computer, why?
- Hardware support:
  - test_and_set
  - compare_and_Swap
  - Bounded waiting?

Compare-and-Swap in Practice: ‘cmpxchg’

An instruction-instance

- In our recent disassembly of Linux’s kernel function ‘rtc_cmos_read()’, this ‘cmpxchg’ instruction-instance was used:

```
lock cmpxchg   %edx, cmos_lock
```

prefix *opcode* *source-operand* *destination-operand*

Note: Keep in mind that the accumulator %eax will affect what happens!
So we need to consider this instruction within it’s surrounding context

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

cmpxchg

‘effects’ and ‘affects’

- According to Intel’s manual, the ‘cmpxchg’ instruction also uses two ‘**implicit**’ operands (i.e., operands not mentioned in the instruction)
  - The CPU’s accumulator register
  - The CPU’s EFLAGS register
- The accumulator-register (EAX) is both a source-operand and a destination-operand
- The six status-bits in the EFLAGS register will get modified, as a ‘side-effect’ this instruction

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

cmpxchg

‘cmpxchg’ description

- This instruction compares the accumulator with the destination-operand (so the ZF-bit in EFLAGS gets assigned accordingly)
- Then:
  - If (accumulator == destination)
    { ZF ← 1; destination ← source; }
  - If (accumulator != destination)
    { ZF ← 0; accumulator ← destination; }

cmos_lock: rvalue
eax: expected_Value
edx: new value

and use eax
as temp return value

```
lock cmpxchg   %edx, cmos_lock
```

prefix *opcode* *source-operand* *destination-operand*

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

First Version

The 'busy-wait' loop

```
# Here is a 'busy-wait' loop, used to wait for the CMOS access to be 'unlocked'
spin:   mov     cmos_lock, %eax         # copy lock-variable to accumulator
        test    %eax, %eax              # was CMOS access 'unlocked'?
        jnz     spin                    # if it wasn't, then check it again

        # A CPU will fall through to here if 'unlocked' access was detected,
        # and that CPU will now attempt to set the 'lock' – in other words, it
        # will try to assign a non-zero value to the 'cmos_lock' variable.

        # But there's a potential 'race' here – the 'cmos_lock' might have been
        # zero when it was copied, but it could have been changed by now…
        # … and that's why we need to execute 'lock cmpxchg' at this point
```

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

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

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

With cmpxchg

Busy-waiting will be brief

```
spin:    # see if the lock-variable is clear
         mov     cmos_lock, %eax
         test    %eax, %eax
         jnz     spin

         # ok, now we try to grab the lock
         lock cmpxchg  %edx, cmos_lock

         # did another CPU grab it first?
         test    %eax, %eax
         jnz     spin
```

cmos_lock: rvalue
eax: expected_Value
edx: new value

and use eax
as temp return value

If our CPU wins the 'race', the (non-zero) value from source-operand EDX will have been stored into the (previously zero) 'cmos_lock' memory-location, but the (previously zero) accumulator EAX will not have been modified; hence our CPU will not jump back, but will fall through and execute the 'critical section' of code (just a few instructions), then will promptly clear the 'cmos_lock' variable.

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

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

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

With cmpxchg

The ‘less likely’ case

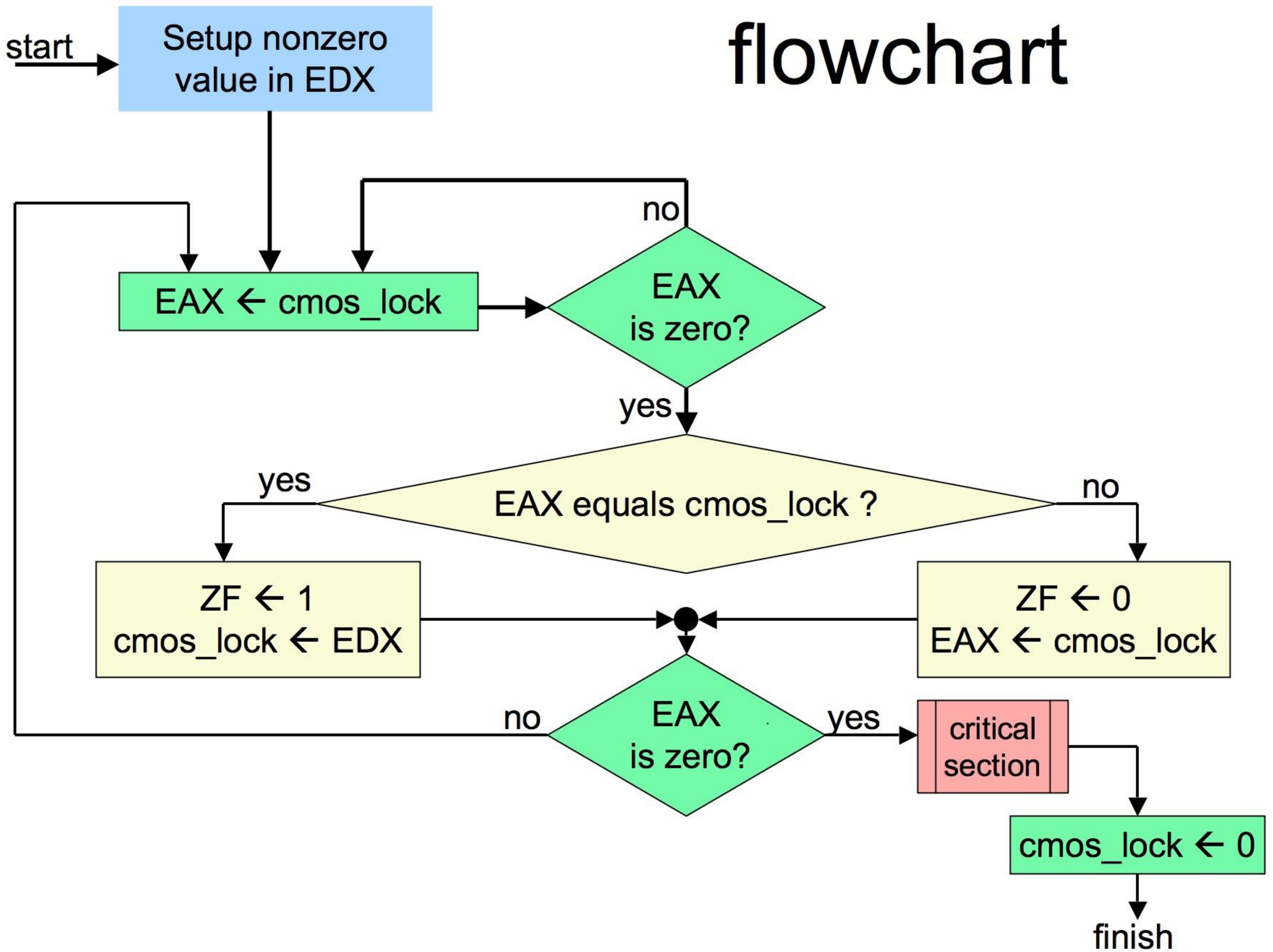
```
spin:   # see if the lock-variable is clear
        mov        cmos_lock, %eax
        test       %eax, %eax
        jnz        spin

        # ok, now we try to grab the lock
        lock cmpxchg  %edx, cmos_lock

        # did another CPU grab it first?
        test       %eax, %eax
        jnz        spin
```

cmos_lock: rvalue
eax: expected_Value
edx: new value

If our CPU loses the ‘race’, because another CPU changed ‘cmos_lock’ to some non-zero value after we had fetched our copy of it, then the (now non-zero) value from the ‘cmos_lock’ destination-operand will have been copied into EAX, and so the final conditional-jump shown above will take our CPU back into the spin-loop, where it will resume busy-waiting until the ‘winner’ of the race clears ‘cmos_lock’.

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

Flow Chart



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

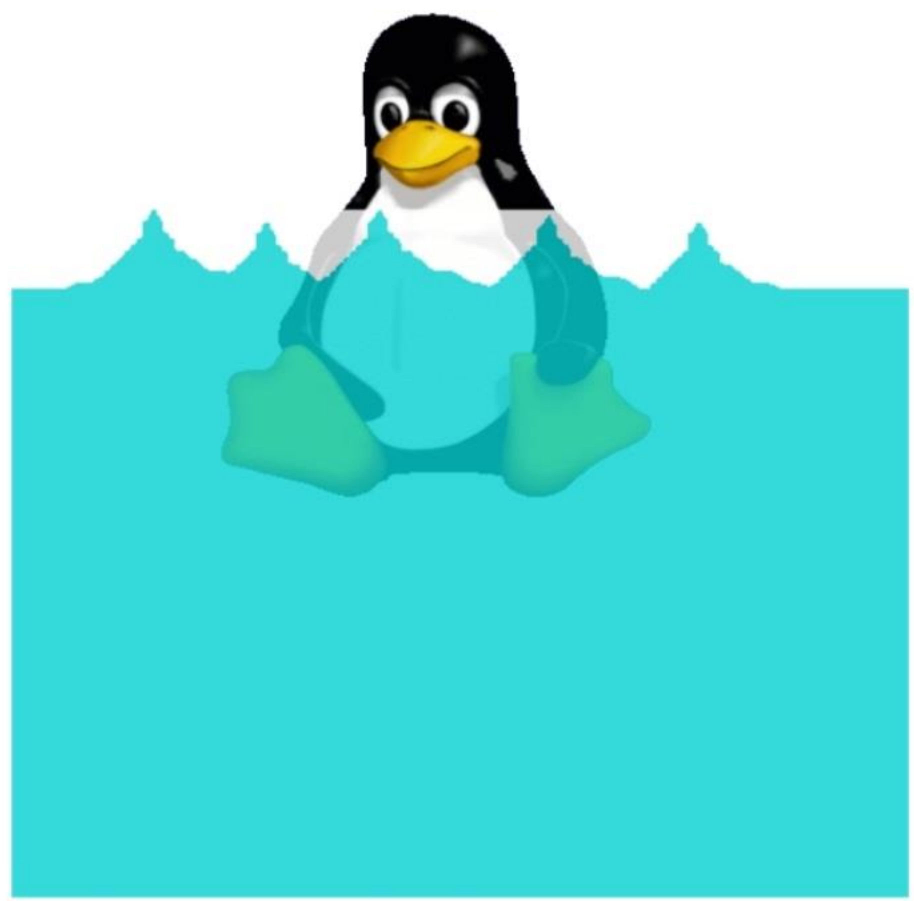
cmos_lock

- This global variable is initialized to zero, meaning that access to CMOS memory locations is not currently 'locked'
- If some CPU stores a non-zero value in this variable's memory-location, it means that access to CMOS memory is 'locked'
- The kernel needs to insure that only one CPU at a time can set this 'lock'

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

Below C Level: An Introduction to Computer Systems

Norm Matloff
University of California, Davis



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

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
    - What if we have N threads?

Just Yield

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

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

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

yield-> moving from running to ready

Still Not Efficient Enough

- 100 threads for the lock
  - T0 gets lock
  - T1, T99 will call lock and then yield, still not efficient
    - Of course, more efficient than spinning

Semaphore

- **Semaphore** S is an integer variable
  - e.g., to represent *how many units of a particular resource is available*
- It can only be **updated with two atomic operations: wait** and **signal**
  - **spin lock** can be used to guarantee atomicity of wait and signal
  - originally called P and V (Dutch)
  - a simple implementation with busy wait can be:

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

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

Semaphore

- **Counting semaphore**: allowing arbitrary resource count
- **Binary semaphore**: integer value can be only 0 or 1
  - also known as **mutex lock** to provide mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
        wait (mutex);
        critical section
        signal (mutex);
        remainder section
} while (TRUE);
```

Semaphore w/ Waiting Queue

- Associate a waiting queue with each semaphore
  - place the process on the waiting queue if **wait** cannot return immediately
  - wake up a process in the waiting queue in **signal**
- There is no need to **busy wait**
- Note: wait and signal must still be atomic

Semaphore w/ 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);
    }
}
```

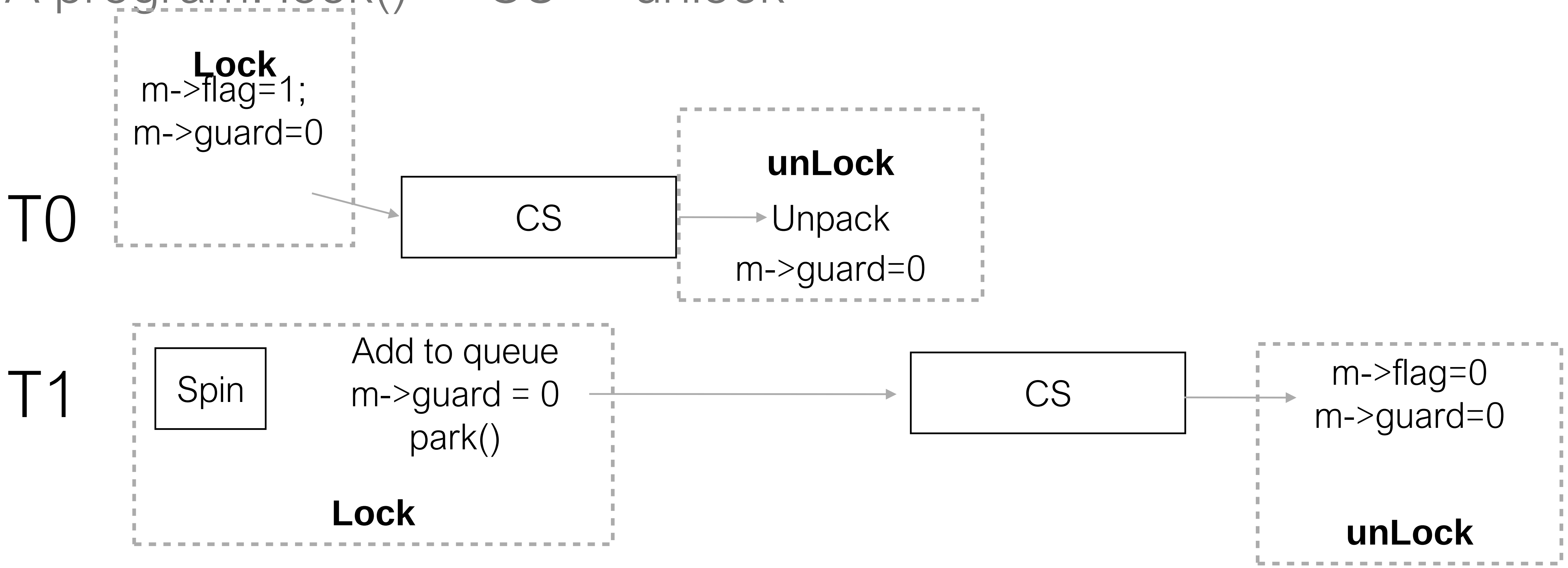
```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

CS to protect m->flag

```c
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

CS to protect m->flag

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

HW6 is out