Race Conditions
Critical Sections
Dekker's Algorithm

Yi Shi

Fall 2018

Xi'an Jiaotong University

Review: CPU Scheduling

- Scheduling problem
  - Given a set of processes that are ready to run
  - Which one to select *next*
- Scheduling criteria
  - CPU utilization, Throughput, Turnaround, Waiting, Response
- Scheduling algorithms
  - FCFS, SJF, SRTF, RR
  - Multilevel (Feedback-)Queue Scheduling

Goals to Today

- Introduction to Synchronization
  - ..or: the trickiest bit of this course

- Background
- Race Conditions
- The Critical-Section Problem

- Dekker's Solution

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers.
  - Assume an integer count keeps track of the number of full buffers.
  - Initially, count is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - It is decremented by the consumer after it consumes a buffer.

Producer-Consumer

- Producer

```
while (true) {

  /*  produce an item and */
  /* put in nextProduced  */

  while (count == BUFFER_SIZE)
          ; // do nothing b/c full

  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
}
```

- Consumer

```
while (true)  {

  /*  consume the item */
  /*  in nextConsumed  */

while (count == 0)
          ; // do nothing b/c empty

  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;
}
```

Race Condition

- count++ *not* atomic operation. Could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- *not* atomic operation. Could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

What just happened?

- Threads share global memory
- When a process contains multiple threads, they have
  - Private registers and stack memory (the *context switching* mechanism needs to save and restore registers when switching from thread to thread)
  - Shared access to the remainder of the process “state”
- This can result in *race conditions*

Two threads, one counter

Popular web server

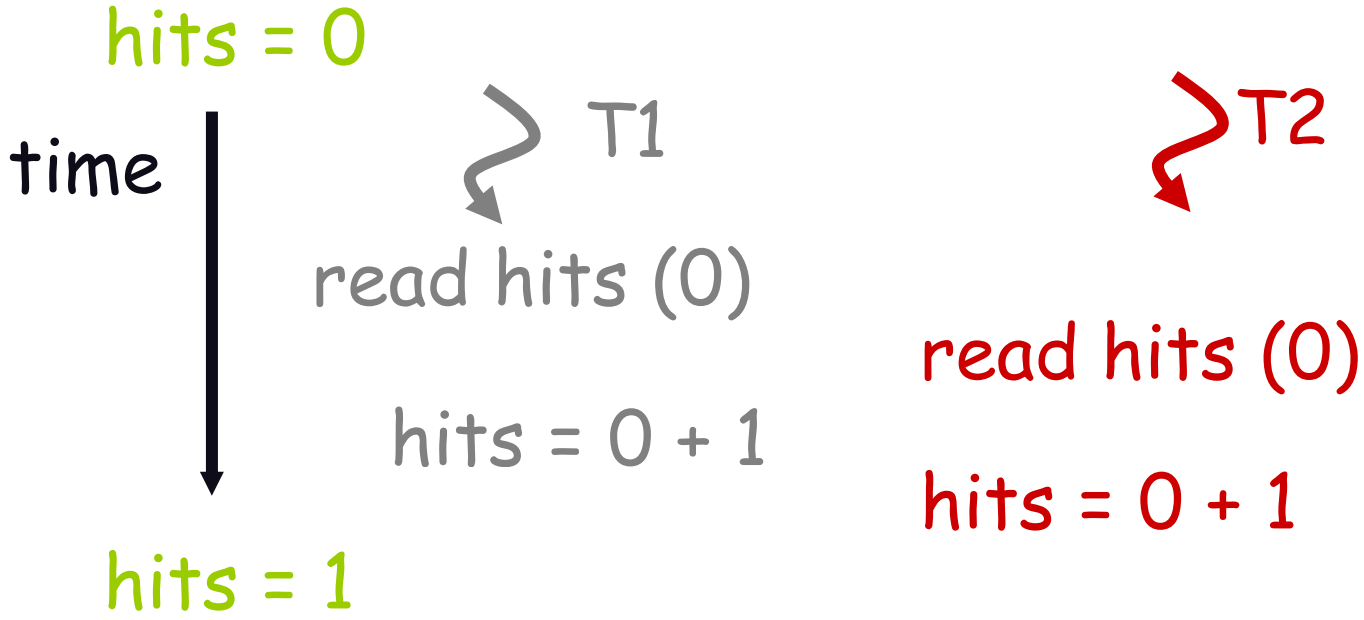
- Uses multiple threads to speed things up.
- Simple shared state error:
  - each thread increments a shared counter to track number of hits

```
...
hits = hits + 1;
...
```

- What happens when two threads execute concurrently?

Shared counters

- Possible result: lost update!



- One other possible result: everything works.
  - ⇒ Difficult to debug
- Called a “race condition”

Race conditions

- Def: *a timing dependent error involving shared state*
  - Whether it happens depends on how threads scheduled
  - In effect, once thread A starts doing something, it needs to "race" to finish it because if thread B looks at the shared memory region before A is done, it may see something inconsistent
- Hard to detect:
  - All possible schedules have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - they are intermittent
    - Timing dependent = small changes can hide bug
  - Celebrate if bug is deterministic and repeatable!

Scheduler assumptions

```
Process a:
    while(i < 10)
        i = i +1;
    print "A won!";
```

```
Process b:
    while(i > -10)
        i = i - 1;
    print "B won!";
```

If *i* is shared, and initialized to 0

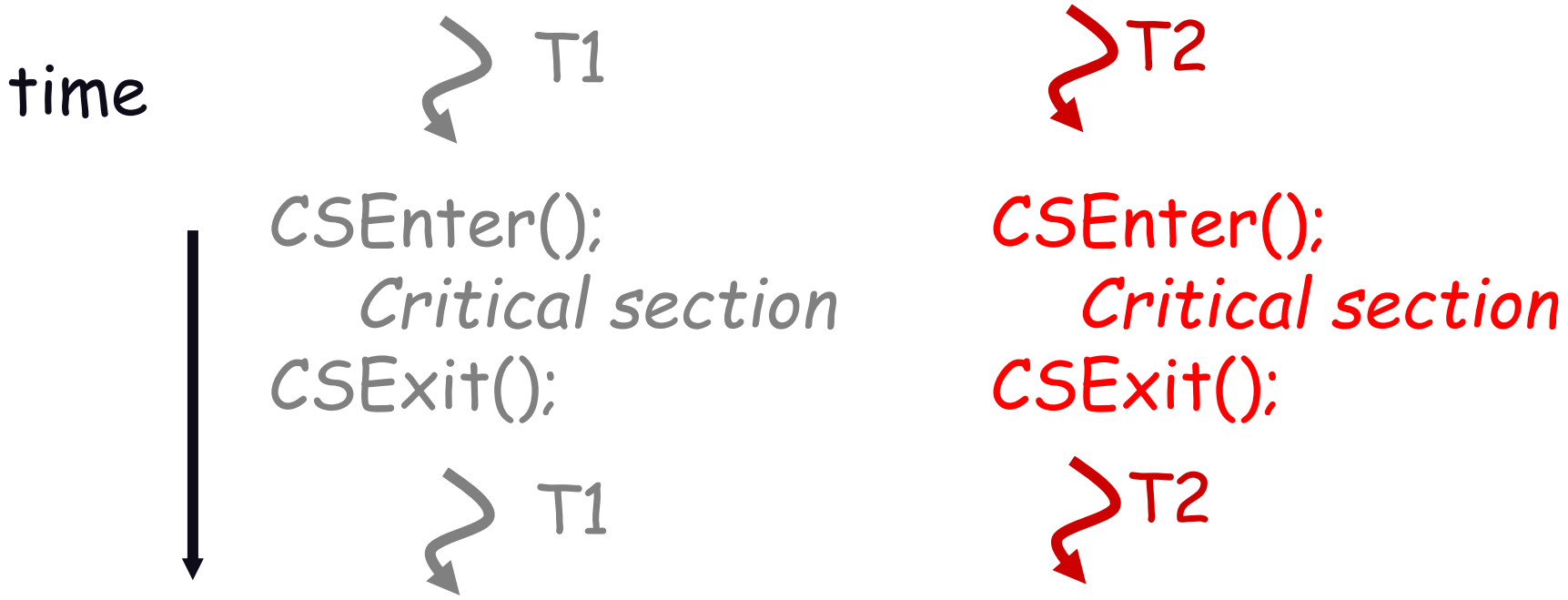
- Who wins?
- Is it guaranteed that someone wins?
- What if both threads run on identical speed CPU
  - executing in parallel

Scheduler Assumptions

- Normally we assume that
  - A scheduler always gives every executable thread opportunities to run
    - In effect, each thread makes *finite progress*
  - But schedulers aren't always fair
    - Some threads may get more chances than others
  - To reason about worst case behavior we sometimes think of the scheduler as an adversary trying to "mess up" the algorithm

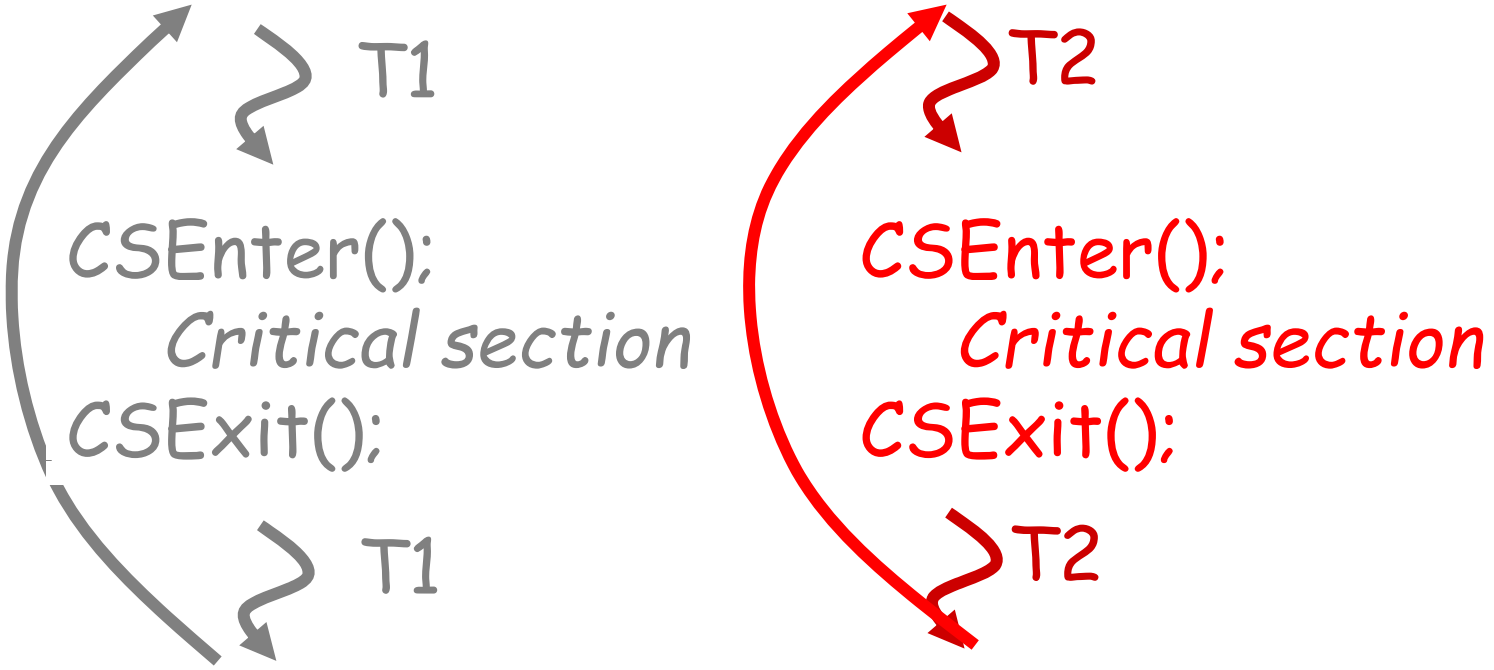
Critical Section Goals

- Threads do some stuff but eventually *might* try to access shared data



Critical Section Goals

- Perhaps they loop (perhaps not!)



Critical Section Goals

- We would like
  - Safety (aka mutual exclusion)
    - No more than one thread can be in a critical section at any time.
  - Liveness (aka progress)
    - A thread that is seeking to enter the critical section will eventually succeed once the previous one exit its critical section.
  - Bounded waiting
    - A bound must exist on the number of times that other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
- Ideally we would like fairness as well
  - If two threads are both trying to enter a critical section, they have equal chances of success
  - … in practice, fairness is rarely guaranteed

Solving the problem

- A first idea:
  - Have a boolean flag, *inside*. Initially false.

```
CSEnter()
{
    while(inside) continue;
    inside = true;
}
```

```
inside = false;
}
```

Code is unsafe: thread 0 could finish the while test when inside is false, but then 1 might call CSEnter() before 0 can set inside to true!

- Now ask:
  - Is this Safe? Live? Bounded waiting?

Solving the problem: Take 2

- A different idea (assumes just two threads):
  - Have a boolean flag, *inside[i]*. Initially false.

```
CSEnter(int i)
{
    inside[i] = true;
    while(inside[i^1]) continue;
}
```

```
                    {
    Inside[i] = false;
}
```

Code isn't live: with bad luck, both threads could be looping, with 0 looking at 1, and 1 looking at 0

- Now ask:
  - Is this Safe? Live? Bounded waiting?

Solving the problem: Take 3

- Another broken solution, for two threads
  - Have a turn variable, turn, initially 1

```
CSEnter(int i)
{
    while(turn != i) continue;
}
```

```
turn = i ^ 1;
}
```

Code isn't live: thread 0 can't enter unless thread 1 did first, and vice-versa. But perhaps one thread needs to enter many times and the other fewer times, or not at all

- Now ask:
  - Is this Safe? Live? Bounded waiting?

A solution that works

- Dekker's Algorithm (1965)
  - (book: Exercise 6.9 in 8th Edition, and 6.1 in 7th Edition)

```
CSEnter(int i)
{
  inside[i] = true;
  while(inside[j])
  {
    if (turn == j)
    {
       inside[i] = false;
       while(turn == j) continue;
       inside[i] = true;
     }
  }}
```

```
CSExit(int i)
{
  turn = j;
  inside[i] = false;
  }
```

Dekker's Algorithm

1. Mutual exclusion:

Initial: inside[*i*] = false; inside[*j*] = false; turn = *i*

```
do{
    inside[i] = true;
    while(inside[j])
    {
     if (turn == j)
      {  inside[i] = false;
         while(turn == j) continue;
         inside[i] = true;
       }
    }
    Critical section
    turn = j;
    inside[i] = false;
    Remainder section
} while(TRUE)
```

Process i

```
do{
    inside[j] = true;
    while(inside[i])
    {
     if (turn == i)
      {  inside[j] = false;
         while(turn == i) continue;
         inside[j] = true;
       }
    }
    Critical section
    turn = i;
    inside[j] = false;
    Remainder section
} while(TRUE)
```

Process j

Dekker's Algorithm

2. Liveness:

Initial: inside[*i*] = false; inside[*j*] = false; turn = *i*

```
do{
    inside[i] = true;
    while(inside[j])
    { if (turn == j)
      {  inside[i] = false;
         while(turn == j) continue;
         inside[i] = true;
      }
    }
    Critical section
    turn = j;
    inside[i] = false;
    Remainder section
} while(TRUE)
```

Process i

```
do{
    inside[j] = true;
    while(inside[i])
    { if (turn == i)
      {  inside[j] = false;
         while(turn == i) continue;
         inside[j] = true;
      }
    }
    Critical section
    turn = i;
    inside[j] = false;
    Remainder section
} while(TRUE)
```

Process i

Dekker's Algorithm

Bounded waiting:

Initial: inside[*i*] = false; inside[*j*] = false; turn = *i*

```
do{
    inside[i] = true;
    while(inside[j])
    { if (turn == j)
      {  inside[i] = false;
         while(turn == j) continue;
         inside[i] = true;
      }
    }
    Critical section
    turn = j;
    inside[i] = false;
    Remainder section
} while(TRUE)
```

Process i

```
do{
    inside[j] = true;
    while(inside[i])
    { if (turn == i)
      {  inside[j] = false;
         while(turn == i) continue;
         inside[j] = true;
      }
    }
    Critical section
    turn = i;
    inside[j] = false;
    Remainder section
} while(TRUE)
```

Process i

Analysis of Dekker's algorithm:

- Safety: No process will enter its CS without setting its *inside* flag. Every process checks the other process *inside* flag after setting its own. If both are set, the *turn* variable is used to allow only one process to proceed.
- Liveness: The *turn* variable is only considered when both processes are using, or trying to use, the resource
- Bounded waiting: The *turn* variable ensures alternate access to the resource when both are competing for access

Why does it work?

- Safety: Suppose thread 0 is in the CS.
  - Then inside[0] is true.
  - If thread 1 was simultaneously trying to enter, then turn must equal 0 and thread 1 waits
- Liveness&Bounded waiting: Suppose thread 1 wants to enter and can't (stuck in while loop)
  - Thread 0 will eventually exit the CS
  - When inside[0] becomes false, thread 1 can enter
  - If thread 0 tries to reenter immediately, it sets turn=1 and hence will wait politely for thread 1 to go first!

Summary

- Race Condition
- Critical Section
- Dekker’s Solution