Operating Systems (A)
(Honor Track)

Lecture 14: Synchronization (I)

Yao Guo (郭耀)

Peking University

Fall 2024

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

This Lecture

Synchronization Basics

Why do we need synchronizations

Critical sections

Buzz Words

Synchronization

Race condition

Data race

Mutual exclusion

Critical section/region

Atomic operations

This Lecture

Synchronization Basics

Why do we need synchronizations

Critical sections

Synchronization of Multiple Processes/Threads

- The central themes of OS design are all concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing/multithreading
  - Distributed processing
- One big issue is synchronization
  - Managing the interaction of all of these processes/threads
- Real life example
  - Two people talking at the same time, can you hear them clearly?

A Simple Game with Three Volunteers

- Producer: produce 1 chalk per iteration
  - Step1: increment the counter on the board
  - Step2: put one chalk on the table
- Consumer:
  - Step1: read the counter LOUD
  - Step2a: if the counter is zero, go back to step1
  - Step2b: if the counter is nonzero, take a chalk from the table
  - Step3: decrement the counter on the board
- OS:
  - Decide who should operate, who should freeze
  - Rule: only the producer or the consumer could “operate” at any given time
  - Goal: try to get the consumer or the producer into “trouble”

Let’s Have a Try

- **Producer: produce 1 chalk per iteration**
  - **Step1: increment the counter**
  - **Step2: put one chalk on the table**
- **Consumer:**
  - **Step1: check the counter to see if it is zero**
  - **Step2a: if the counter is zero, go back to step1**
  - **Step2b: if the counter is nonzero, take one chalk from the table**
  - **Step3: decrement the counter**

- Stop Producer before step2 and let Consumer go.
- What happens?
- How to fix the code to avoid this problem?

Issues

- Reasons
  - Data are shared
  - Consecutive & related things are broken up

- What are shared in the game?
  - The counter is shared
  - The chalks are shared

Shared Resources

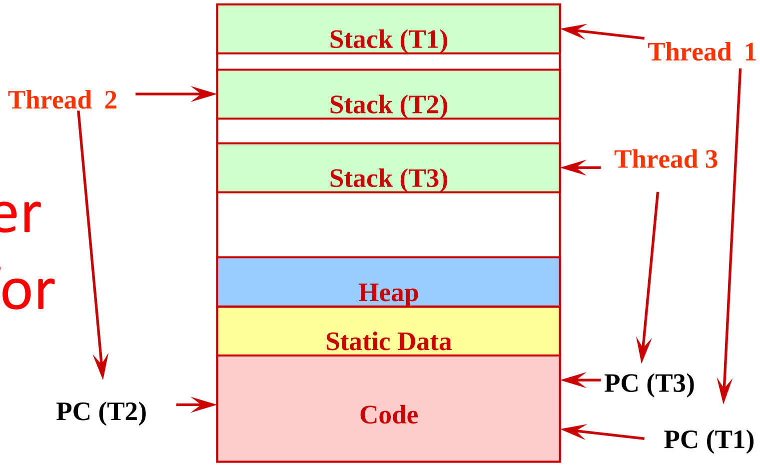
- The problem is that two concurrent threads (or processes, we do not distinguish them here) accessed a shared resource (account) without any synchronization
  - Known as a race condition (remember this buzzword)
- We need mechanisms to control access to these shared resources when facing concurrency
  - So we can reason about how the program should operate
- Shared data structure
  - Buffers, queues, lists, hash tables, etc.

Real World Examples?

- Synchronization is like Traffic Signals
  - Each thread is like a vehicle – it can make progress independently with its own speed and direction
  - What are shared in the road?
  - What are the synchronization methods for those cars?
  - What if we do not have those methods?

When are Resources Shared?

- Local variables are not shared (private)
  - Data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2



- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete

Assumptions for Now

- Suppose we have some *atomic operations*
  - One which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- We'll assume that the only atomic operations are reads and writes of words
  - Some architectures DON’T even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like AND as long as it's not delayed forever

- Will we encounter issues?

Simple Case? Sharing One Set of Data

- Do accesses to shared data need to be synchronized?
  - E.g., considering X = X + 1 or X++;
  - Interleaving by an access from another thread to the same shared data between two subsequent accesses

Read X
.
Manipulate X
.
Write X

Read X
.
Manipulate X
.
Write X

Code Examples

Thread 1

```
void foo ( )
{
    x++;
}
```

Thread 2

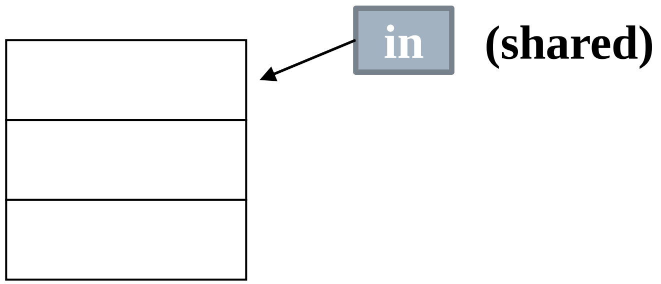
```
void bar ( )
{
    x--;
}
```

- x is a global variable and initially x = 0;
- After thread 1 and 2 execute, what is the value of x?
  - The value could be 0, -1, 1
  - Why?
- Does the above case happen only on a multicore machine? Will it happen on a uniprocessor machine?
- What if x is a local variable declared inside the function foo or bar, respectively?

Even Worse

- Spooling example: two threads

```
int next_free;
next_free = in;
Stores F into next_free;
in=next_free+1;
```



- What if those consecutive & related things…

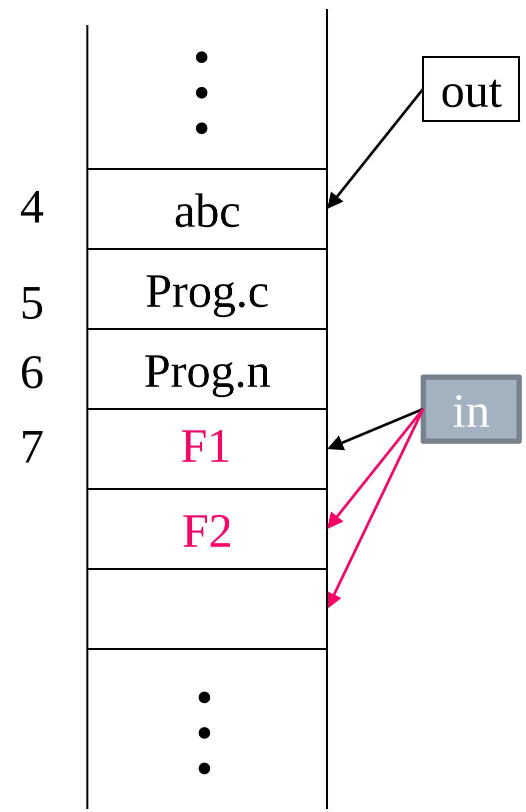
Spooling Example: Problem-Free Interleaving

Thread 1

int next_free;

1. next_free = in;
2. Stores F1 into next_free;
3. in=next_free+1;

Shared memory



Thread 2

int next_free;

4. next_free = in;
5. Stores F2 into next_free;
6. in=next_free+1;

Does this code **always** work?

Spooling Example: Races

Thread 1

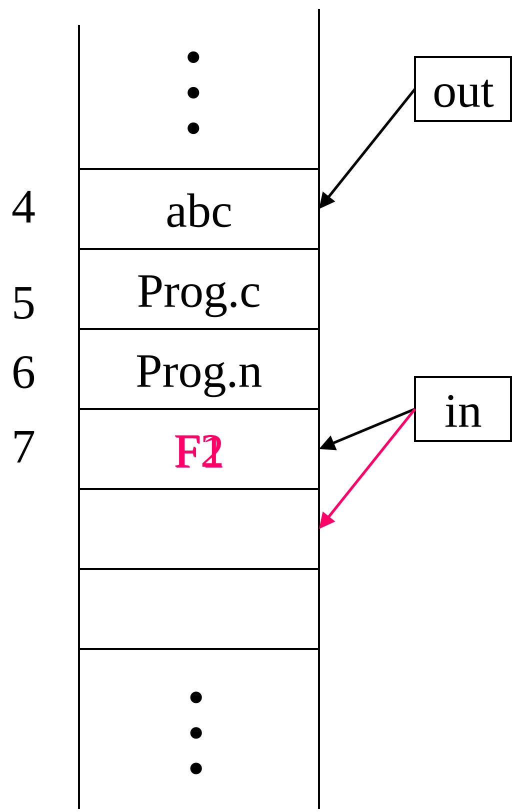
int next_free;

1 next_free = in;

3 Stores F1 into next_free;

4 in=next_free+1;

Shared memory



Thread 2

int next_free;

2 next_free = in;

5 Stores F2 into next_free;

6 in=next_free+1;

Classic Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your significant other share a bank account with a balance of RMB 10000
- Then you each go to separate ATM machines and simultaneously withdraw RMB 1000 from the account

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential interleaving of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;

balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);

put_balance(account, balance);
```

Context switch

- What is the balance of the account now?
- Is the bank happy with our implementation?

What if this is not withdrawal, but deposit?

This Lecture

Synchronization Basics

Why do we need synchronizations

Critical sections

Mutual Exclusion

- We want to use mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
- Code that uses mutual exclusion (and…) to synchronize its execution is called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing your bathroom with roommates
- What requirements would you place on a critical section?

Critical Section (Critical Region) Example

```
Process {
    while (true) {
        ENTER CRITICAL SECTION
        Access shared variables; // Critical Section;
        LEAVE CRITICAL SECTION
        Do other work
      }
}
```

Mutual Exclusion Using Critical Sections

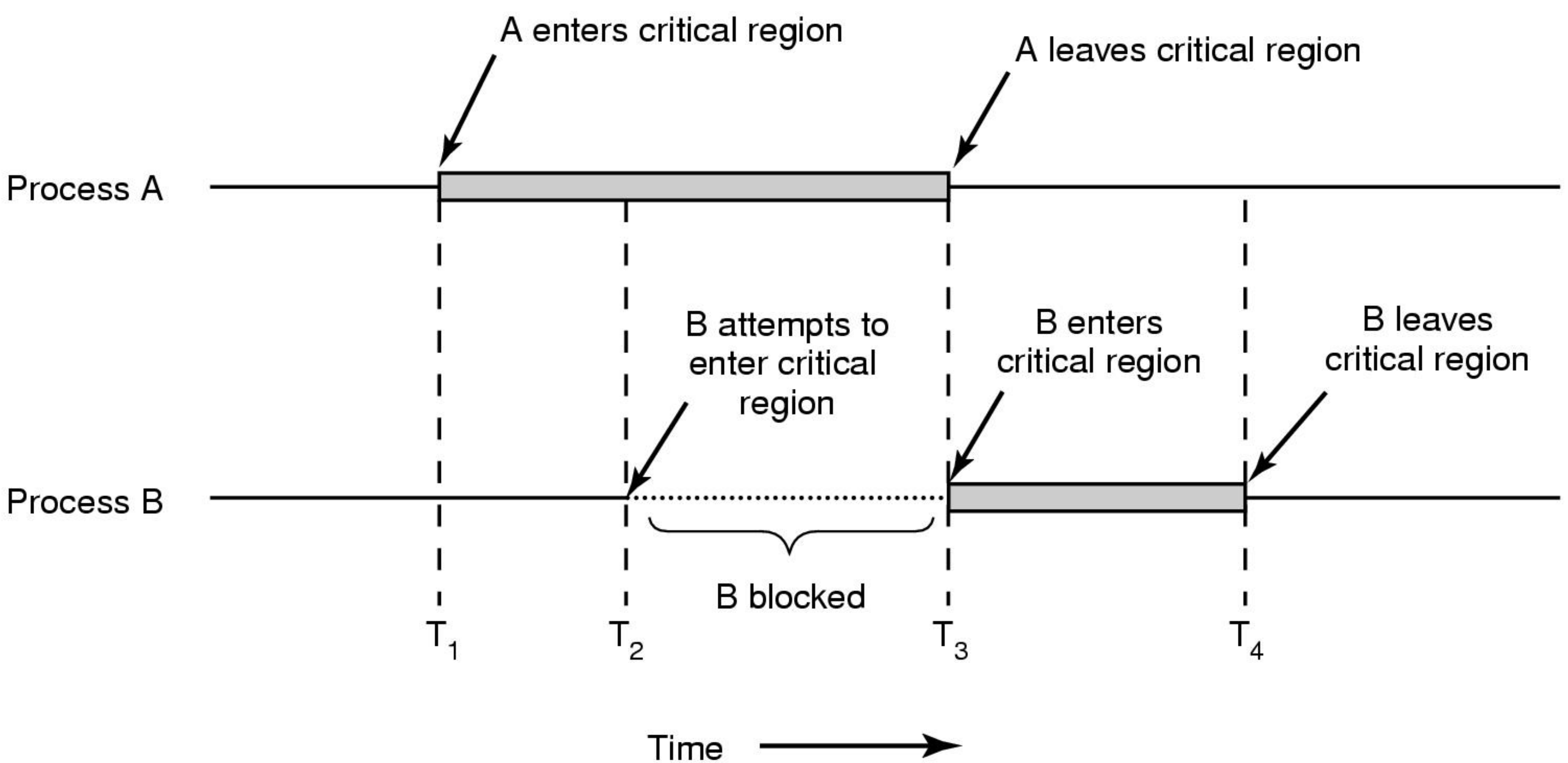


Figure 2-22. Mutual exclusion using critical regions.

Questions to Ponder

- What are the conditions to avoid race condition?
  1. No assumptions may be made about speeds or the number of CPUs
  2. No two processes may be simultaneously inside their critical regions
  3. No process running outside its critical region may block other processes
  4. No process should have to wait forever to enter its critical region
- What will happen otherwise?

Critical Section Requirements

- Mutual exclusion (mutex)
  - If one thread is in the critical section, then no other is
- Progress
  - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
  - A thread in the critical section will eventually leave it
- Bounded waiting (no starvation)
  - If some thread T is waiting on the critical section, then T will eventually enter the critical section
- Performance
  - The overhead of entering and exiting the critical section is small with respect to the work being done within it

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

* Operating System Concept Essentials (Abrahan Silberschatz et al.), P220

About the Requirements

Requirements also expressed as three properties:

- Safety property: nothing bad happens
  - Mutex
- Liveness property : something good happens
  - Progress, Bounded Waiting
- Performance property
  - Performance
- Properties hold for each run, while performance depends on all the runs
  - Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don’t forget liveness)

How to Build Critical Sections

- Mechanisms for building critical sections
  - Atomic read/write
  - Locks
  - Semaphores
  - Monitors
  - Messages
- Let’s look at the atomic read/write first

Mutual Exclusion with Atomic Read/Writes: First Try

```
int turn = 1;
```

```
while (true) {
    while (turn == 2) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn == 1) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation

(1) Does it satisfy mutex?

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2 (why?)
- (turn == 1) ≡ (turn != 2)

(2) Does it work?

- No. It violates progress: e.g. the blue thread could go into an infinite loop outside of the critical section, which will prevent the yellow one from entering

Peterson's Algorithm

```
int turn = 1;
bool ready1 = false, ready2 = false;
```

```
while (true) {
    ready1 = true;
    turn = 2;
    while (turn == 2 && ready2) ;
    critical section
    ready1 = false;
    outside of critical section
}
```

```
while (true) {
    ready2 = true;
    turn = 1;
    while (turn == 1 && ready1) ;
    critical section
    ready2 = false;
    outside of critical section
}
```

- Does it work?
  - Mutual exclusion
  - Progress
  - Bounded waiting

Peterson's Algorithm (Figure 2-24)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Summary

- Why do we need synchronizations?
- Critical sections
  - Concepts
  - Critical sections with atomic read/write
- Next time: Synchronization mechanisms
  - Locks
  - Semaphores
  - Monitors