University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 7: Concurrency

University of Nottingham Ningbo China, 2024

Recap
Last Lecture

- Threads vs. processes
- Different thread implementations
- Scheduling of thread
- POSIX Threads (PThreads)

Today Class

- Concurrency Definition
  - Examples of Concurrency Problems
- Race condition
- Critical section
- Critical Section Problem Solutions

Concurrency- context

- Concurrency is the execution of the multiple instruction sequences at the same time.

  - It happens in the operating system when there are **several process /threads** running in **parallel or execute concurrently.**

  - The running process/threads always communicate with each other through shared memory or message passing or can share resources (e.g., devices, variables, memory, data structures, etc.)

Concurrency- context

- There are several motivations for allowing concurrent execution:

  - **Physical resource sharing** : Multiuser environment since hardware resources are limited.

  - **Logical resource sharing:** Shared file(same piece of information).

  - **Computation speedup:** Parallel execution

  - **Modularity:** Divide system functions into separation processes.

Concurrency- context

- A process/thread can be interrupted at any point in time (I/O, timer)

  - The process "state" is saved in the process control block.

  - Sharing data can lead to **inconsistencies** (e.g. when interrupted manipulating data)
    - i.e., the outcome of execution may depend on the order in which instructions are carried out.

- The outcome of programs may become unpredictable. (How?)

How? Example1-Incrementing a counter

- Let see Counter++ statement:
  - It appears to be a single operation in high-level programming languages like C or Java, is actually composed of several steps at the machine level, and these steps are not atomic.
  - In machine language it consists of three separate actions (threads) in practice

```
1.  Read the value of counter and store it in a register
2.  Add one to the value in the register
3.  Store the value of the register in counter
```

  - i.e. they can be interrupted by, e.g., **context switches.**

Atomic operation: A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. [Starlings page 201]

How? Example1-Incrementing a counter

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they do **Not** interact):

- And, assume that the initial value of counter ==1

```
Process 1:                   Process 2:
...                          ...
Read counter -> register     ...
Add 1 to value in register   ...
Store value in counter       ...
...                          Read counter -> register
...                          Add 1 to value in register
...                          Store value in counter
```

- After the P1 and P2 is executed the final result of counter is 3.

How? Example1-Incrementing a counter

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they **DO** interact):

- Assume that the initial value of counter == 1

Process 1:	Process 2:
...	...
Read counter -> register	...
...	Read counter -> register
Add 1 to value in register	...
Store value in counter	...
...	Add 1 to value in register
...	Store value in counter

- After the P1 and P2 is executed the final result of counter is 2.

How? Example 2-Shared procedures

- Consider the following **code shared** between threads/processes
- chin and chout **shared global variables**

```
viod print()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

How? Example 2-Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they do **NOT** interact):

- Assume that the input char is X and Y

```
Process 1:
...
chin = getchar();
chout = chin;
putchar(chout);
...
...
...
```

```
Process 2:
...
...
...
...
chin = getchar();
chout = chin;
putchar(chout);
```

- The output is XY

How? Example 2-Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they do interact):

- Assume that the input char is X and Y

```
Process 1:
...
chin = getchar();
...
chout = chin;
putchar(chout);
...
...
```

```
Process 2:
...
...
chin = getchar();
...
...
chout = chin;
putchar(chout);
```

- The output is YY

How? Example 3- Bounded Buffers

- Consider a **bounded buffer** in which *N* **items** can be stored

- A **counter** is maintained to count the number of items currently in the buffer
  - **Incremented** when an item is **added**
  - **Decremented** when an item is **removed**

- Similar **concurrency problems** as with the calculation of sums happen in the consumer/producer problem

```
while (true) {
    //while buffer is full
    while (counter == BUFFER SIZE) ; /* do nothing */

    // add item when space becomes available
    buffer[in] = new_item;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Producer

```
while (true) {
    // wait until items in buffer
    while (counter == 0); /* do nothing */

    consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next consumed */
}
```

Consumer

Race Conditions-Data Inconsistency (example)

```
//Counter ++
//where register1  is the one of the local
//CPU registers
```

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

```
// Counter --
//where register2 is the one of the local
//CPU registers
```

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

- Suppose we
  - Counter=5 // is a shared variable
  - Two process, Producer and Consumer.
  - Following the execution of counter++ and counter-- statements, The value of variable counter can be 4,5,6!
  - However, the **only correct result** is **counter=5**,
  - One of such interleaving is the following:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- If we reversed the order of the statements at T4 and T5, counter==6
- A situation like this, is known as a **race condition.**

Race Conditions-Data Inconsistency

- A **race condition occurs** when multiple threads/processes **access shared data** and the result is dependent on **the order in which the instructions are interleaved.**
  - I.e., the final result depends on how the instructions are interleaved

- We will discuss **mechanisms** to provide **controlled/synchronized** access to data and **avoid race conditions**

Critical Section-Definition

- **A critical Section:** is a section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
  - Suppose n process {P0, P1…,Pn-1}.
    - Each process has segment of code, called a critical section, in which the process may changing common variables, updating a table, writing file, and so on.
      - When one process is executing in its CS, no other process are executing in its critical section.

```
do
{
    . . .
    // entry to critical section
    critical section, e.g.counter++;
    // exit critical section
    remaining code
    . . .
} while (...);
```

Fig: A general structure of a typical process Pi

A Solution to the Critical Section(CS) Problem

- A **solution to the critical section** problem must satisfy the following three requirements:
  - Mutual exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
  - Progress: The progress condition ensures that if no process is currently in its critical section and some processes want to enter, then only the processes that are not in their remainder sections (i.e., processes that want to enter the critical section) can participate in the decision of which will enter next. This decision cannot be postponed indefinitely.
  - Fairness/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.
    - This ensures fairness and prevents starvation, meaning every process eventually gets to enter the critical section.

Real-World Example: A Shared Kitchen

This Photo by Unknown Author is licensed under CC BY-NC-ND

Imagine a shared kitchen in a college dormitory where multiple students want to cook their meals. This kitchen has limited resources like stovetops, a refrigerator, and cooking utensils.

- **Mutual Exclusion:**
  - Only one student should be allowed to use a particular resource (e.g., a stovetop or the refrigerator) at a time.
  - If two students try to use the same resource simultaneously, there might be accidents, confusion, or resource conflicts.

Real-World Example: A Shared Kitchen

Imagine a shared kitchen in a college dormitory where multiple students want to cook their meals. This kitchen has limited resources like stovetops, a refrigerator, and cooking utensils.

- **Progress:**
  - If the kitchen is empty (no one is cooking) but there are students waiting to cook, they should be able to enter the kitchen and start using the resources.
  - The decision of which student gets access should be based on those who are ready and waiting, not on arbitrary factors or students who are not yet ready to cook.
  - A fair system could be implemented, such as a waiting list or a "first-come, first-served" queue, ensuring that the students who are actively ready to cook are the ones deciding who goes next, without unnecessary delays or indecision.

Real-World Example: A Shared Kitchen

- Fairness (Bounded Waiting): Suppose a student is waiting to use the stovetop. There should be a limit to how many other students can finish cooking before this student gets a chance to cook. This ensures that no student has to wait indefinitely while others keep cooking.

Imagine a shared kitchen in a college dormitory where multiple students want to cook their meals. This kitchen has limited resources like stovetops, a refrigerator, and cooking utensils.

Example of CS

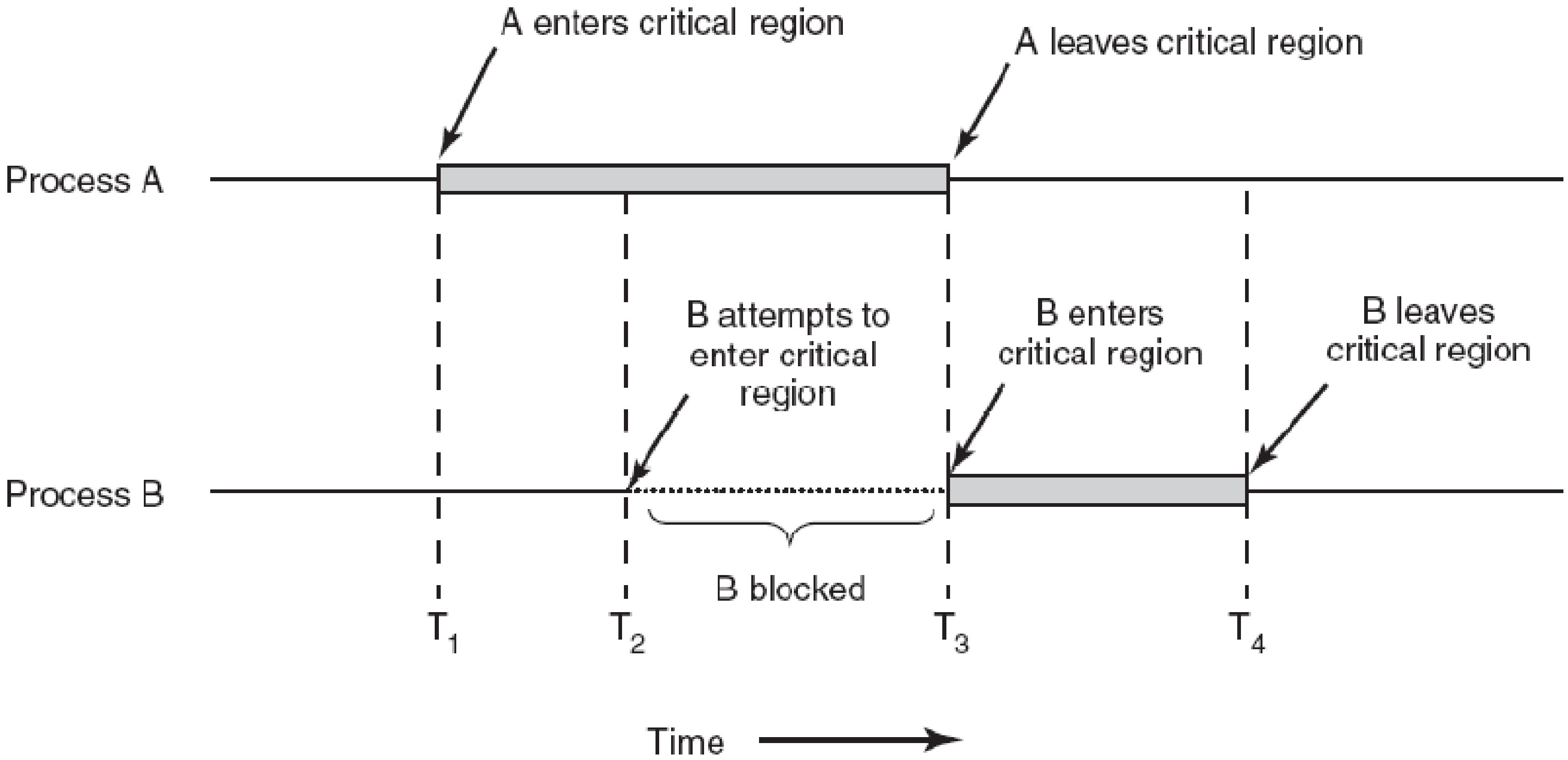

Figure: Tanenbaum, 4th ed., section 2.3.2

Approaches Handling the critical section problem

- The approaches to handle the critical section problem, categorized by:
  - Software based
  - Hardware based
  - Operating System approach

Approaches Handling the critical section problem
Software based

- These involve implementing synchronization mechanisms directly within the application code using variables, flags, and custom algorithms.

- These solutions work at the **user-space** level, where no special hardware support or operating system intervention is needed.

  - Examples:
    - Peterson’s Solution,
    - Lamport’s Bakery Algorithm,
    - Dekker’s Algorithm and Szymanski's Algorithm:

Approaches Handling the critical section problem Hardware based

- These approaches rely on low-level hardware instructions to provide mutual exclusion.

- They ensure that certain operations (like reading and writing shared memory) are atomic, meaning that they cannot be interrupted and will always complete fully.

- Examples:
  - Disabling interrupt,
  - Test_and_set() and
  - Compare_and_swap()

Approaches Handling the critical section problem
Operating System approach

- Provides synchronization primitives to manage access to critical sections.

- These primitives are implemented in the kernel and provide efficient control over shared resources, often relying on hardware mechanisms internally.

  - Examples:
    - Mutexes
    - Semaphores
    - Monitors

Concurrency within the OS-Data Structures

- Two general approaches that operating systems use to handle critical sections, particularly in kernel mode: Preemptive kernels and Non-preemptive kernels

- Preemptive kernels:
  - A process in kernel mode can be interrupted and another process can be scheduled.
  - Multiple processes can be active in the kernel at once, leading to potential race conditions.
  - Requires synchronization:
    - Uses locks, mutexes, and atomic operations to protect shared data.
  - E.g. Windows and Linux

Concurrency within the OS-Data Structures

- Two general approaches that operating systems use to handle critical sections, particularly in kernel mode: Preemptive kernels and Non-preemptive kernels

- Non-preemptive kernels
  - A process runs in kernel mode until it finishes, blocks, or yields.
  - No race conditions: Only one process is active in the kernel at a time, simplifying design.
  - E.g. Older or simpler OS, such as early versions of MS-DOS or embedded systems.

Quiz

University of Nottingham
UK | CHINA | MALAYSIA

Summary-Take-home Message

- **Problems with synchronization** and concurrent/parallel code
- Concurrency from an **OS perspective**
- Concept of **mutual exclusion**
- Requirements and **approaches** for mutual exclusion

Recommended readings

- Modern Operating Systems(Tanenbaum):**Chapter2(2.3)**
- Operating System Concepts(Silberschatz):**Chapter5(5.1-2)**
- Operating Systems: Internals and Design Principles (Starlings):**Chapter5(5.1-2)**