Lecture - VIII
Process Synchronization - I

Tevfik Koşar
Louisiana State University
September 17th, 2009

Roadmap
- Process Synchronization
- Race Conditions
- Critical-Section Problem
  - Solutions to Critical Section
  - Different Implementations

Background
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Consider consumer-producer problem:
  - Initially, count is set to 0
  - It is incremented by the producer after it produces a new buffer
  - And is decremented by the consumer after it consumes a buffer.

Shared Variables: count=0, buffer[]

Producer:
```c
while (true){
  /* produce an item and put in nextProduced */
  while (count == BUFFER_SIZE);
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
}
```

Consumer:
```c
while (1) {
  while (count == 0);
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;
} /* consume the item in nextConsumed */
```

Race Condition
- Race condition: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be synchronized.
  - Ensure that only one process at a time is manipulating the variable counter.
- The statements
  - `counter++;
  - `counter--;
must be performed atomically.
- Atomic operation means an operation without interruption.

Race Condition
- `count++` could be implemented as
  ```c
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- `count--` could be implemented as
  ```c
  register2 = count
  register2 = register2 - 1
  count = register2
  ```
- Consider this execution interleaving with “count = 5” initially:
  ```c
  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}
  ```
Race Condition

Significant race conditions in I/O & variable sharing

```c
char chin, chout; //shared
void echo() {
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
        while (...);
    } while (...);
}
```

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

Multithreaded echo (lucky)

---

Race Condition

Significant race conditions in I/O & variable sharing

```c
char chin, chout; //shared
void echo() {
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
        while (...);
    } while (...);
}
```

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

Multithreaded echo (unlucky)

---

Race Condition

Significant race conditions in I/O & variable sharing

```c
void echo() {
    char chin, chout;
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
        while (...);
    } while (...);
}
```

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

Multithreaded echo (unlucky)

---

Race Condition

Significant race conditions in I/O & variable sharing

- in this case, replacing the global variables with local variables did not solve the problem
- we actually had two race conditions here:
  - one race condition in the shared variables and the order of value assignment
  - another race condition in the shared output stream; which thread is going to write to output first (this race persisted even after making the variables local to each thread)

=> generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)

---

Critical Section/Region

- Critical section/region: segment of code in which the process may be changing shared data (e.g. common variables)
- No two processes should be executing in their critical sections at the same time --> prevents race conditions
- Critical section problem: design a protocol that the processes use to cooperate

---

Critical Section

The “indivisible” execution blocks are critical regions

- a critical region is a section of code that may be executed by only one process or thread at a time

A

B

common critical region

- although it is not necessarily the same region of memory or section of program in both processes

A

B

A's critical region

B's critical region

=> but physically different or not, what matters is that these regions cannot be interleaved or executed in parallel (pseudo or real)
Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following requirements:

1. **Mutual Exclusion** - If process \( P_i \) is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.


Critical Section

- We need mutual exclusion from critical regions.
  - Critical regions can be protected from concurrent access by padding them with entrance and exit gates (we’ll see how later): a thread must try to check in, then it must check out.

```c
void echo()
{
    char chin, chout;
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    } while (...);
}
```

Mutual Exclusion

- Desired effect: mutual exclusion from the critical region.

**Implementation 1 — disabling hardware interrupts**

1. Thread A reaches the gate to the critical region (CR) before B.
2. As soon as A enters CR, it disables all interrupts, thus B cannot be scheduled.
3. As soon as A exits CR, it enables interrupts; B can be scheduled again.
4. Thread B enters CR.

**Implementation 1 — disabling hardware interrupts**

- It works, but not reasonable!
- What guarantees that the user process is going to ever exit the critical region?
- Meanwhile, the CPU cannot interleave any other task, even unrelated to this race condition.
- The critical region becomes one physically indivisible block, not logically.
- Also, this is not working in multi-processors.

```c
void echo()
{
    char chin, chout;
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    } while (...);
}
```
Summary

- Process Synchronization
- Race Conditions
- Critical-Section Problem
  - Solutions to Critical Section
  - Different Implementations

- Next Lecture: Synchronization - II
- Reading Assignment: Chapter 6 from Silberschatz.

Acknowledgements

- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from UNR