Lecture - VIII
Process Synchronization - I

Tevfik Koşar

University at Buffalo
September 22nd, 2011

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.

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
  * count++;
  * count--;
  must be performed atomically.
- Atomic operation means an operation without interruption.

Roadmap

- Process Synchronization
- Race Conditions
- Critical-Section Problem
  - Solutions to Critical Section
  - Different Implementations
- Semaphores
- Classic Problems of Synchronization

Shared Variables: count=0, buffer[]

Producer:

```java
while (true) {
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE);
    // do nothing
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

while (1) {
    while (count = 0);
    // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
} /* consume the item in nextConsumed */
```

Consumer:

```java
while (1) {
    while (count == 0);
    // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
} /* consume the item in nextConsumed */
```

Race Condition

- count++ could be implemented as
  ```java
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- count-- could be implemented as
  ```java
  register2 = count
  register2 = register2 - 1
  count = register2
  ```
- Consider this execution interleaving with “count = 5” initially:
  1. S0: producer execute register1 = count [register1 = 5]
  2. S1: producer execute register1 = register1 + 1 [register1 = 6]
  3. S2: consumer execute register2 = count [register2 = 5]
  5. S4: producer execute count = register1 [count = 6]
Race Condition

Significant race conditions in I/O & variable sharing

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

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

Multithreaded echo (lucky)

Significant race conditions in I/O & variable sharing

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

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

Multithreaded echo (unlucky)

Critical Section/Region

- Critical section/region: segment of code in which the process may be changing shared data (eg. 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

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

A

B

> ./echo
Hello world!
Hello world!

Single-threaded echo

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

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

A

B

As 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.

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

Mutual Exclusion

- **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

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

3. **Bounded Waiting** - A bound must exist 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.

   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the \( N \) processes

Mutual Exclusion

- **Implementation 1** — disabling hardware interrupts
  - 1. thread A reaches the gate to the critical region (CR) before B
  - 2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
  - 3. thread A exits CR; thread B can now enter
  - 4. thread B enters CR

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

**Implementation 2 — simple lock variable**

1. thread A reaches CR and finds a lock at 0, which means that A can enter
2. thread A sets the lock to 1 and enters CR, which prevents B from entering
3. thread A exits CR and resets lock to 0; thread B can now enter
4. thread B sets the lock to 1 and enters CR

---

**Implementation 2 — simple lock variable**

- the "lock" is a shared variable
- entering the critical region means testing and then setting the lock
- exiting means resetting the lock

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

---

**Implementation 3 — “indivisible” lock variable**

1. thread A reaches CR and finds the lock at 0 and sets it in one shot, then enters
   1.1 even if B comes right behind A, it will find that the lock is already at 1
   1.2 A sets the lock to 1 and enters CR but cannot prevent the fact that . . .
   1.3 . . . B is going to set the lock to 1 and enter CR, too
2. thread A exits CR, then resets lock to 0
3. thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR

---

**Implementation 3 — “indivisible” lock variable**

- the indivisibility of the “test-lock-and-set-lock” operation can be implemented with the hardware instruction TSL

```c
void echo()
{
  char chin, chout;
do {
    TSL; // test and set lock
    chout = chin;
    putchar(chout);
  }while (...);
}
```

---

**Mutual Exclusion**

- **Implementation 3 — “indivisible” lock ⇔ one key**

1. Thread A reaches CR and finds a key and takes it.
   - Even if B comes right behind A, it will not find a key.
2. Thread A exits CR and puts the key back in place.
3. Thread B finds the key and takes it, just before entering CR.

**Summary**

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

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

**Mutual Exclusion**

- **Implementation 3 — “indivisible” lock ⇔ one key**

  - "Holding" a unique object, like a key, is an equivalent metaphor for "test-and-set".
  - This is similar to the "speaker's baton" in some assemblies: only one person can hold it at a time.
  - Holding is an indivisible action: you see it and grab it in one shot.
  - After you are done, you release the object, so another process can hold on to it.

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

**Acknowledgements**

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