Lecture - IX
Process Synchronization - II

Tevfik Koşar
Louisiana State University
September 22nd, 2009

Roadmap
- Solutions for Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber

Mutual Exclusion
- Desired effect: mutual exclusion from the critical region

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

HOW is this achieved??

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

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

Implementation 2 — simple lock variable

- thread A reaches the gate to the critical region (CR) before B
- as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
- as soon as A exits CR, it enables interrupts; B can be scheduled again
- thread B enters CR
Mutual Exclusion

**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

```cpp
bool lock = FALSE;
void echo()
{
    char chin, chout;
do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    }while (lock);
}/* do nothing: loop */
lock = TRUE;
lock = FALSE;
```

1. thread A reaches CR and finds a lock at 0, which means that A can enter
   - before A can set the lock to 1, B reaches CR and finds the lock is 0, too
2. A sets the lock to 1 and enters CR but cannot prevent the fact that . . .
3. . . . B is going to set the lock to 1 and enter CR, too

**Implementation 3 — "indivisible" lock variable**
- the indivisibility of the "test-lock-and-set-lock" operation can be implemented with the hardware instruction TSL

```cpp
void echo()
{
    char chin, chout;
do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    }while (0);
}/* do nothing: loop */
TSL  // copy lock to register
B
A
```

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 (...);
}
```

Mutual Exclusion

- **Implementation 4** — no-TSL toggle for two threads
  - the “toggle lock” is a shared variable used for strict alternation
  - here, entering the critical region means only testing the toggle: it must be at 0 for A, and 1 for B
  - exiting means switching the toggle: A sets it to 1, and B to 0

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

Mutual Exclusion

- **Implementation 5** — Peterson’s no-TSL, no-alternation
  - A and B each have their own lock: an extra toggle is also masking either lock
  - A arrives first, sets its lock, pushes the mask to the other lock and may enter
  - then, B also sets its lock & pushes the mask, but must wait until A’s lock is reset
  - A exits the CR and resets its lock; B may now enter

```c
lock[A] = FALSE;
latch[B] = FALSE;
while (lock[B] && mask == B); /* loop */
lock[A] = TRUE;
latch[B] = TRUE;
while (lock[A] && mask == A); /* loop */
```
Mutual Exclusion

- **Implementation 5 — Peterson's no-TSL, no-alternation**
  1. A and B each have their own lock; an extra toggle is also masking either lock
  2.1 A is interrupted between setting the lock & pushing the mask; B sets its lock
  2.2 now, both A and B race to push the mask: whoever does it last will allow the other one inside CR
  ➡️ mutual exclusion holds!! (no bad race condition)

---

Summary of these implementations of mutual exclusion

- **Impl. 1 — disabling hardware interrupts**
  ➡️ NO: race condition avoided, but can crash the system!
- **Impl. 2 — simple lock variable (unprotected)**
  ➡️ NO: still suffers from race condition
- **Impl. 3 — indivisible lock variable (TSL)**
  ➡️ YES: works, but requires hardware
- **Impl. 4 — no-TSL toggle for two threads**
  ➡️ NO: race condition avoided inside, but lockup outside
- **Impl. 5 — Peterson's no-TSL, no-alternation**
  ➡️ YES: works in software, but processing overhead

---

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
    - Either test memory word and set value
    - Or swap contents of two memory words

---

Semaphore

- Semaphore $S$ - integer variable
- Two standard operations modify `wait()` and `signal()`
  - Originally called `P()` and `V()`
    - `wait ($S$)`
      - `while $S <= 0
       $S--;
      ; // no-op`
    - `signal ($S$)`
      - `$S++;`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

---

Semaphores as Synchronization Tool

- **Counting** semaphore - integer value can range over an unrestricted domain
- **Binary** semaphore - integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Provides mutual exclusion
  - `Semaphore S;  // initialized to 1`
  - `wait ($S$);`
  - `Critical Section`
  - `signal ($S$);`

---

Problem: all implementations (2-5) rely on busy waiting

- “busy waiting” means that the process/thread continuously executes a tight loop until some condition changes
- busy waiting is bad:
  - waste of CPU time — the busy process is not doing anything useful, yet remains “Ready” instead of “Blocked”
  - paradox of inverted priority — by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and . . .
  - liberating B! (B is working against its own interest)

- we need for the waiting process to **block**, not keep idling
Deadlock and Starvation

• **Deadlock** - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - Let \( S \) and \( Q \) be two semaphores initialized to 1
    
    \[
    P_0 \\
    \text{wait}(S); \\
    \text{wait}(Q); \\
    \text{wait}(Q); \\
    \text{wait}(S); \\
    \text{signal}(S); \\
    \text{signal}(Q); \\
    \text{signal}(Q); \\
    \text{signal}(S);
    \]
  - Starvation - indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

• Bounded-Buffer Problem
• Readers and Writers Problem
• Dining-Philosophers Problem
• Sleeping Barber Problem

Bounded-Buffer Problem

- \( N \) buffers, each can hold one item
- Semaphore **mutex** for access to the buffer, initialized to 1
- Semaphore **full** (number of full buffers) initialized to 0
- Semaphore **empty** (number of empty buffers) initialized to \( N \)

Bounded Buffer Problem (Cont.)

- The structure of the **producer process**
  
  ```
  do {
    // produce an item
    wait (empty);
    wait (mutex);
    // add the item to the buffer
    signal (mutex);
    signal (full);
  }
  ```

Bounded Buffer Problem (Cont.)

- The structure of the **consumer process**
  
  ```
  do {
    wait (full);
    wait (mutex);
    // remove an item from buffer
    signal (mutex);
    signal (empty);
    // consume the removed item
  }
  ```

Summary

• Solutions for Critical-Section Problem
  - Semaphores
  - Classic Problems of Synchronization
    - Bounded Buffer
    - Readers-Writers
    - Dining Philosophers
    - Sleeping Barber
  
  • **Next Lecture**: Deadlocks - I
  • **Reading Assignment**: Chapter 6 from Silberschatz.
Acknowledgements

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