Synchronization Principles

CS 256/456

Dept. of Computer Science, University of Rochester

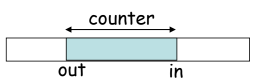
Synchronization Principles

- Background
  - Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- The Critical-Section Problem
  - Pure software solution
  - With help from the hardware
- Synchronization without busy waiting (with the support of process/thread scheduler)
  - Semaphore
  - Mutex lock
  - Condition variables

Bounded Buffer

- Shared data

```
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```



- Producer process

```
item nextProduced;
while (1) {
    while (((in+1)%BUFFER_SIZE==out)
            ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

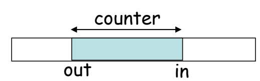
- Consumer process

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


Bounded Buffer

- Shared data

```
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```



- Producer process

```
item nextProduced;
while (1) {
    while (counter==BUFFER_SIZE)
            ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer process

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

Bounded Buffer

- The following statements must be performed *atomically*:
  counter++;
  counter--;
- Atomic operation means an operation that completes in its entirety without interruption.
- The statement "**counter++**" may be compiled into the following instruction sequence:

register1 = counter;
register1 = register1 + 1;
counter = register1;

- The statement "**counter--**" may be compiled into:

register2 = counter;
register2 = register2 - 1;
counter = register2;

10/1/2012 CSC 2/456 5

Race Condition

- **Race condition**:
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data and/or effects on the participating processes depends upon the order of process execution – nondeterminism.
- To prevent race conditions, concurrent processes must be **synchronized**.

10/1/2012 CSC 2/456 6

The Critical-Section Problem

- Problem context:
  - *n* processes all competing to use some shared data
  - Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. **Mutual Exclusion**. No two processes simultaneously in the critical section.
  2. **Progress**. No process running outside its critical section may block other processes.
  3. **Bounded Waiting/Fairness**. Given the set of concurrent processes, 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.

10/1/2012 CSC 2/456 7

Eliminating Concurrency

- First idea: eliminating the chance of context switch when a process runs in the critical section.
  - effective as a complete solution only on a single-processor machine
  - only for short critical sections
- How to eliminate context switch?
  - software exceptions
  - hardware interrupts
  - system calls
- Disabling interrupts?
  - not feasible for user programs since they shouldn't be able to disable interrupts
  - feasible for OS kernel programs

10/1/2012 CSC 2/456 8

Critical Section for Two Processes

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.
- Assumption: instructions are atomic and no re-ordering of instructions.

10/1/2012 CSC 2/456 9

Algorithm 1

- Shared variables:
  - **int turn;**
    initially **turn = 0;**
  - **turn==i** ⇒ P_i can enter its critical section
- Process P_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

10/1/2012 CSC 2/456 10

Algorithm 2

- Shared variables:
  - **boolean flag[2]**;
    initially **flag[0] = flag[1] = false;**
  - **flag[i]==true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
    flag[i] = true;
    while (flag[j]) ;
    critical section
    flag[i] = false;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

10/1/2012 CSC 2/456 11

Algorithm 3

- Combine shared variables of algorithms 1 and 2.
- Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j) ;
        critical section
    flag[i] = false;
        remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes. ⇒ called Peterson's algorithm.

10/1/2012 CSC 2/456 12

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Synchronization Using Special Instruction: TSL (test-and-set)

```
entry_section:
    TSL R1, LOCK            | copy lock to R1 and set lock to 1
    CMP R1, #0              | was lock zero?
    JNE entry_section       | if it wasn't zero, lock was set, so loop
    RET                     | return; critical section entered

exit_section:
    MOV LOCK, #0            | store 0 into lock
    RET                     | return; out of critical section
```

- Does it solve the synchronization problem?
- Does it work for multiple (>2) processes?

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```


Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4      ; mov exchange value
     ll   R2, 0(R1)  ; load linked
     sc R3, 0(R1)    ; store conditional
     beqz R3, try    ; branch if store fails
     mov  R4, R2     ; put load value in R4
```

Solving the Critical Section Problem with Busy Waiting

- In all our solutions, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - Waste of CPU time
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)
- Solution
  - Avoid busy wait as much as possible (yield the processor instead).
  - If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel)

10/1/2012 CSC 2/456 17

Recap

- Concurrent access to shared data may result in data inconsistency – race condition.
- The Critical-Section problem
  - Pure software solution
  - With help from the hardware
- Problems with busy-waiting-based synchronization
  - Waste CPU, particularly when context switch occurs while a process is inside critical section
- Solution
  - Avoid busy wait as much as possible (yield the processor instead).
  - If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel)

10/1/2012 CSC 2/456 18

Scheduler-Based Synchronization

- In all our solutions so far, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - Waste of CPU time
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)
- Solution
  - Avoid busy wait as much as possible (yield the processor instead) – scheduler-based synchronization
  - If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel)

10/1/2012 CSC 2/456 19

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable which can only be accessed via two atomic operations
- Semantics (roughly) of the two operations:

  wait(S) or **P(S):**

  wait until S>0;
  S--;

  signal(S) or **V(S):**

  S++;

- Solving the critical section problem:

Shared data:
semaphore mutex=1;

Process *Pi:*
wait(mutex);
critical section
signal(mutex);
remainder section

10/1/2012 CSC 2/456 20

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int value;
    proc_list *L;
} semaphore;
```

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(P)** resumes the execution of a blocked process **P**.

- Semaphore operations now defined as (both are atomic):

```
wait(S):
    value = (S.value--);
    if (value < 0) {
        add this process to S.L;
        block;
    }
signal(S):
    value = (S.value++);
    if (value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
```

How do we make sure wait(S) and signal(S) are atomic?
So have we truly removed busy waiting?

10/1/2012 CSC 2/456 21

Mutex Lock (Binary Semaphore)

- Mutex lock – a semaphore with only two state: locked/unlocked
- Semantics of the two (atomic) operations:

lock(mutex):

```
wait until mutex==unlocked;
mutex=locked;
```

unlock(mutex):

```
mutex=unlocked;
```

- Can you implement mutex lock using semaphore?
- How about the opposite?

10/1/2012 CSC 2/456 22

Implement Semaphore Using Mutex Lock

- Data structures:

```
mutex_lock L1, L2;
int C;
```

- Initialization:

```
L1 = unlocked;
L2 = locked;
C = initial value of semaphore;
```

- *wait* operation:

```
lock(L1);
C --;
if (C < 0) {
    unlock(L1);
    lock(L2);
}
unlock(L1);
```

- *signal* operation:

```
lock(L1);
C ++;
if (C <= 0)
    unlock(L2);
else
    unlock(L1);
```

10/1/2012 CSC 2/456 23

Classical Problems of Synchronization

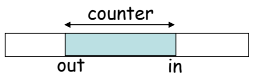
- Bounded-Buffer Problem
- Dining-Philosophers Problem

10/1/2012 CSC 2/456 24

Bounded Buffer

- Shared data

```
typedef struct { ... } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```



- Producer process

```
item nextProduced;
while (1) {
    while (counter==BUFFER_SIZE)
          ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer process

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

10/1/2012 CSC 2/456 25

Bounded Buffer Problem

- Shared data

```
buffer;
```

- Producer process

```
while (1) {
    ...
    produce an item in nextp;
    ...
    add nextp to buffer;
    ...
}
```

- Consumer process

```
while (1) {
    ...
    remove an item from buffer to nextc;
    ...
    consume nextc;
    ...
}
```

- Protecting the critical section for safe concurrent execution.
- Synchronizing producer and consumer when buffer is empty/full.

10/1/2012 CSC 2/456 26

Bounded Buffer Solution

- Shared data

```
buffer;
semaphore full=0;
semaphore empty=n;
semaphore mutex=1;
```

- Producer process

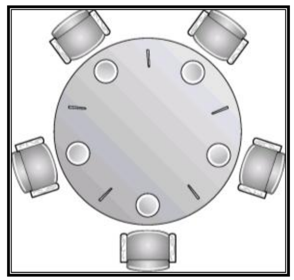
```
while (1) {
    ...
    produce an item in nextp;
    ...
    wait(empty);
    wait(mutex);
    add nextp to buffer;
    signal(mutex);
    signal(full);
    ...
}
```

- Consumer process

```
while (1) {
    ...
    wait(full);
    wait(mutex);
    remove an item from buffer to nextc;
    signal(mutex);
    signal(empty);
    ...
    consume nextc;
    ...
}
```

10/1/2012 CSC 2/456 27

Dining-Philosophers Problem



- Philosopher i (1 ≤ i ≤ 5):

```
while (1) {
    ...
    eat;
    ...
    think;
    ...
}
```

- eating needs both chopsticks (the left and the right one).

10/1/2012 CSC 2/456 28

Dining-Philosophers: Possible Solution

- Shared data:

semaphore chopstick[5];
Initially all values are 1;

- Philosopher i:

```
while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
};
```

Deadlock?

10/1/2012 CSC 2/456 29

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- Native support for mutual exclusion.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        . . .
    }
    procedure body Pn (...) {
        . . .
    }
    {
        initialization code
    }
}
```

10/1/2012 CSC 2/456 30

Condition Variables in Monitors

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

    means that the process invoking this operation is suspended until another process invokes

    x.signal();
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.
- Unlike semaphore, there is no counting in condition variables

10/1/2012 CSC 2/456 31

Two Semantics of Condition Variables

- Hoare semantics:
  - p_0 executes signal while p_1 is waiting $\Rightarrow$ p_0 immediately yields the monitor to p_1
  - The logical condition holds when P1 gets to run

```
if (resourceNotAvailable()) Condition.wait();
/* now available ... continue ... */
. . .
```

- Alternative semantics:
  - p_0 executes signal while p_1 is waiting $\Rightarrow$ p_0 continues to execute, then when p_0 exits the monitor p_1 can receive the signal
  - The logical condition may not hold when P1 gets to run
  - Brinch Hansen ("Mesa") semantics: p_0 must exit the monitor after a signal

10/1/2012 CSC 2/456 32

Dining Philosophers Solution

```
monitor dp {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown (int i) {
        state[i] = THINKING;
        test((i+4)%5);
        test((i+1)%5);
    }

    void test (int i) {
        if (state[(i+4)%5]!=EATING && state[(i+1)%5]!=EATING && state[i] == HUNGRY) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    void init() {
        for (int i=0; i<5; i++)
            state[i] = THINKING;
    }
}
```

10/1/2012 CSC 2/456 33

Dining Philosophers Alternative Solution

```
monitor dp {
    enum {thinking, eating} state[5];
    condition cond[5];
    void pickup(int i) {
        while (state[(i+4)%5]==eating || state[(i+1)%5]==eating)
            cond[i].wait();
        state[i] = eating;
    }
    void putdown(int i) {
        state[i] = thinking;
        cond[(i+4)%5].signal();
        cond[(i+1)%5].signal();
    }
    void init() {
        for (int i=0; i<5; i++)
            state[i] = thinking;
    }
}
```

10/1/2012 CSC 2/456 34

Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, and Kai Shen. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).

10/1/2012 CSC 2/456 93