

Operating Systems

LECTURE-10 The Critical-Section Problem



Cooperating Processes

- Introduction to Cooperating Processes
- Producer/Consumer Problem
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores

The Critical-Section Problem

- n processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.



CS Problem Dynamics (1)

- When a process executes code that manipulates shared data (or resource), we say that the process is in its Critical Section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.

CS Problem Dynamics (2)

- The section of code implementing this request is called the Entry Section (ES).
- The critical section (CS) might be followed by a Leave/Exit Section (LS).
- The remaining code is the Remainder Section (RS).
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
 1. **Mutual Exclusion**
 2. **Progress**
 3. **Bounded Waiting**
- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.



Solution to CS Problem – Mutual Exclusion

- 1. Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
 - Implications:
 - Critical sections better be focused and short.
 - Better not get into an infinite loop in there.
 - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

Solution to CS Problem – Progress

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 process that will enter the critical section next cannot be postponed indefinitely:

- If only one process wants to enter, it should be able to.
- If two or more want to enter, one of them should succeed.

Solution to CS Problem – Bounded Waiting

- 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.

Types of solutions to CS problem

- Software solutions –
 - algorithms whose correctness does not rely on any other assumptions.
- Hardware solutions –
 - rely on some special machine instructions.
- Operating System solutions –
 - provide some functions and data structures to the programmer through system/library calls.
- Programming Language solutions –
 - Linguistic constructs provided as part of a language.

Semaphores

- ◆ **Semaphore** is a type of generalized lock
 - Defined by Dijkstra in the last 60s
 - Main synchronization primitives used in UNIX
 - Consist of a positive integer value
 - Two operations
 - ◆ **P()**: an atomic operation that waits for semaphore to become positive, then decrement it by 1
 - ◆ **V()**: an atomic operation that increments semaphore by 1 and wakes up a waiting thread at P(), if any.

Semaphores vs. Integers

- ◆ No negative values
- ◆ Only operations are P() and V()
 - Cannot read or write semaphore values
 - Except at the initialization times
- ◆ Operations are atomic
 - Two P() calls cannot decrement the value below zero
 - A sleeping thread at P() cannot miss a wakeup from V()

Binary Semaphores

- ◆ A **binary semaphore** is initialized to 1
- ◆ P() waits until the value is 1
 - Then set it to 0
- ◆ V() sets the value to 1
 - Wakes up a thread waiting at P(), if any

Two Uses of Semaphores

1. Mutual exclusion

- Lock was designed to do this

```
lock->acquire();  
// critical section  
lock->release();
```

Two Uses of Semaphores

1. Mutual exclusion

1. The lock function can be realized with a binary semaphore: semaphore subsumes lock.

- ◆ Semaphore has an initial value of 1
- ◆ P() is called before a critical section
- ◆ V() is called after the critical section

```
semaphore litter_box = 1;  
P(litter_box);  
// critical section  
V(litter_box);
```

Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- P() is called before a critical section
- V() is called after the critical section

```
semaphore litter_box = 1;
```

```
P(litter_box);
```

```
// critical section
```

```
V(litter_box);
```



litter_box = 1

Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- P() is called before a critical section
- V() is called after the critical section



```
semaphore litter_box = 1;
```

```
P(litter_box); // purrr...
```

```
// critical section
```

```
V(litter_box);
```

litter_box = 1 → 0

Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- P() is called before a critical section
- V() is called after the critical section

```
semaphore litter_box = 1;  
P(litter_box);  
// critical section  
V(litter_box);
```

litter_box = 0



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- P() is called before a critical section
- V() is called after the critical section

```
semaphore litter_box = 1;
```

```
P(litter_box); // meow...
```

```
// critical section
```

```
V(litter_box);
```



litter_box = 0



Two Uses of Semaphores

1. Mutual exclusion

- Semaphore has an initial value of 1
- P() is called before a critical section
- V() is called after the critical section

```
semaphore litter_box = 1;
```

```
P(litter_box);
```

```
// critical section
```

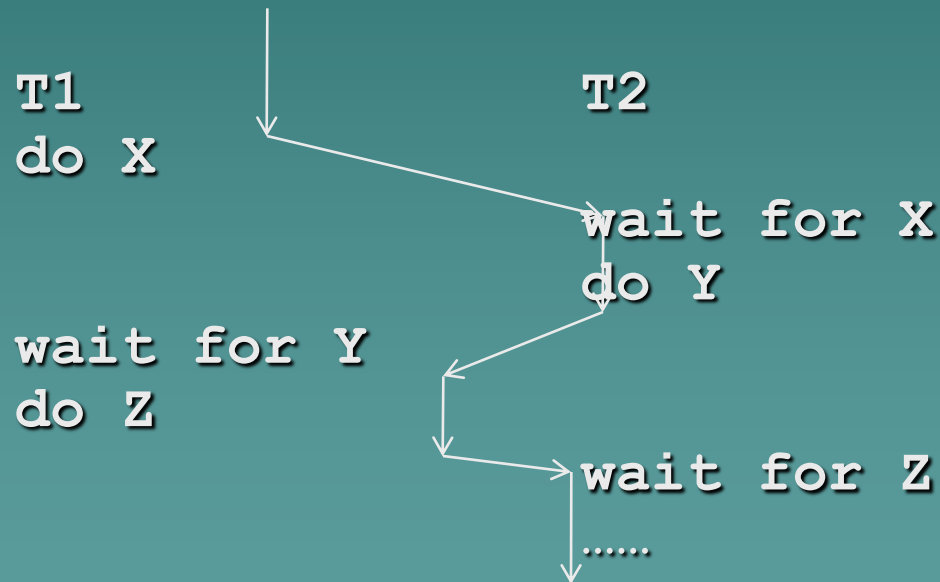
```
V(litter_box);
```



litter_box = 0 → 1

Two Uses of Semaphores

2. Synchronization: Enforcing some order between threads



Two Uses of Semaphores

2. Synchronization

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```

Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```



```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```

wait



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0 → 1  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 1 → 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

wait



```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```

Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0 → 1
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 1 → 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```



Two Uses of Semaphores

2. Scheduling

- Semaphore usually has an initial value of 0

```
semaphore wait_left = 0;  
semaphore wait_right = 0;
```

```
wait_left = 0  
wait_right = 0
```

```
Left_Paw() {  
    slide_left();  
    V(wait_left);  
    P(wait_right);  
    slide_right();  
}
```

```
Right_Paw() {  
    P(wait_left);  
    slide_left();  
    slide_right();  
    V(wait_right);  
}
```




Two Uses of Semaphores

2. Synchronization

- Semaphore usually has an initial value of 0

```
semaphore s1 = 0;  
semaphore s2 = 0;
```

```
A() {  
    write(x);  
    V(s1);  
    P(s2);  
    read(y);  
}  
  
B() {  
    P(s1);  
    read(x);  
    write(y);  
    V(s2);  
}
```



Algorithm for Process P_i

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    critical section  
    flag[i] = FALSE;  
    remainder section  
} while (TRUE);
```

- ◆ Provable that
- 1 Mutual exclusion is preserved

Synchronization

Hardware(Language mechanism for Synchronization)

- ◆ 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
 - ◆ Operating systems using this not broadly scalable
- ◆ Modern machines provide special atomic hardware instructions
 - ◆ Atomic = non-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {  
    _____  
    _____ acquire lock  
        critical section  
    release lock  
        remainder section  
} while (TRUE);
```

TestAndSet Instruction

- ◆ Definition:

```
boolean TestAndSet
(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Solution using TestAndSet

- ◆ Shared boolean variable lock, initialized to FALSE
- ◆ Solution:

```
do {  
    while ( TestAndSet (&lock ))  
        ; // do nothing  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```


Swap Instruction

- ◆ Definition:

```
void Swap (boolean *a,  
boolean *b)  
{  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Solution using Swap

- ◆ Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

- ◆ Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```

ASSIGNMENT

- Q: What are the necessary conditions required for critical section problem solution?