Operating Systems

BITS Pilani
K K Birla Goa Campus

Dr. Lucy J. Gudino
Dept. of CS and IS

innovate achieve lead

BITS Pilani
K K Birla Goa Campus

Process Synchronization

Last class

- Producer - Consumer
- Implementation of counter
- Race condition
- Critical Section Problem
  - Structure
- 3 – requirements
  – Mutual Exclusion
  – Progress
  – Bounded Waiting
- 2 solutions
  - Solution 1 : Turn variable
  - Solution 2: Turn and flag[i]

Hardware approach

- **Test_and_Set** instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
- **Definition of the TestAndSet () instruction:**

```
boolean TestAndSet(boolean *target) {
        boolean rv = *target;
        *target = true;

        return rv;
}
```

Mutual Exclusion with Test_and_Set instruction

Shared data: Boolean lock = false;

Process P_0	Process P_1
do {	do {
while (Test_And_Set(&lock))	while (Test_And_Set(&lock))
; //wait	; //wait
critical section	critical section
lock = false;	lock = false;
remainder section	remainder section
} while (TRUE);	} while (TRUE);

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

swap () instruction

- Atomically swap two variables.

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

Mutual Exclusion with Swap

Shared data (initialized to **false**):

boolean lock;

Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
        critical section
    lock = false;
        remainder section
} while (TRUE);
```

Semaphores

- is a variable which is treated in a special way
- allow processes to make use of critical section in exclusion to other processes
- is a synchronization tool that does not require busy waiting.
- process wanting to access critical section locks semaphore and releases lock on exit.

Contd…

- Basic properties of semaphore:
  - Semaphore *S* – integer variable
  - can only be accessed via two operations

    ***wait* (*S*):**

    while *S* $\leq$ 0 do *no-op*;

    S--;

    ***signal* (*S*):**

    S++;
  - Semaphore operation is atomic and indivisible
    - wait and signal operations are carried out without interruption
- Binary semaphore : can have two values 0 and 1
- On some systems, binary semaphores are known as mutex locks, as they are locks that provide *mutual exclusion.*

Critical Section of *n* Processes

Shared data:

semaphore mutex; *//*initially *mutex* = 1

Process *Pi:*

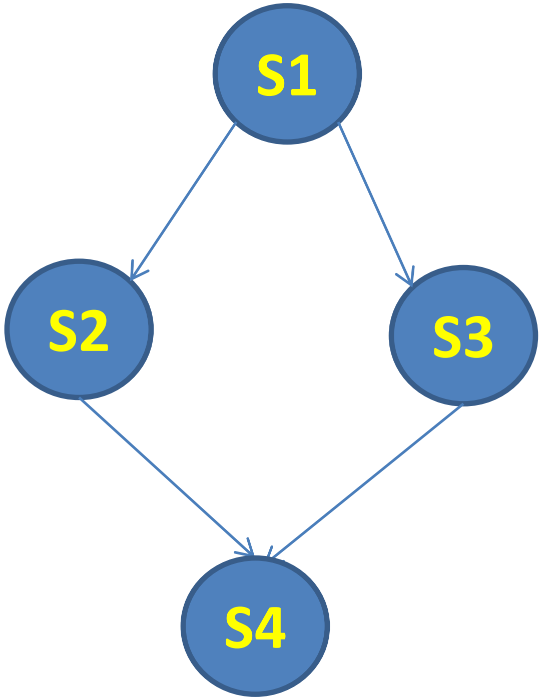
```
do {
        remainder section
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

Contd…

- semaphore can be used to solve various synchronization problems
- Example : two concurrent processes : P1 (with S1) and P2 (with S2) and semaphore variable : synch initialized to zero.
    - requirement : **s2 should be executed only after s1 is executed**

```
P1:
  S1;
  signal (synch);

P2:
  wait (synch);
  S2;
```




```
semaphore a, b, c, d // all are initialized
    to zero
begin
  cobegin
    begin S1; signal(a), signal (b) ; end
    begin  wait (a); S2; signal (c) ; end
    begin  wait (b); S3; signal (d) ; end
    begin  wait (c ); wait (d); S4; end
  coend
end
```

Contd…

- Main disadvantage : Busy Waiting
- semaphore is also called a **spinlock** because the process "spins" while waiting for the lock.
- To overcome busy waiting, we need to modify the definition of the wait () and signal () semaphore operations.

Semaphore Implementation

Define a semaphore as a record

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

Assume two simple operations:

- **block** suspends the process that invokes it.
- **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

Semaphore operations now defined as

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this process to S.L;
        block;
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
}
```

P1:

wait(S)
CS
signal(S)
RS

P2:

wait(S)
CS
signal(S)
RS

Contd…

- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- CPU scheduler selects another process to execute.
- wakeup () operation changes the process from the waiting state to the ready state

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.

Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

- *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.

Classical Problems of Synchronization

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem

- Also known as producer consumer problem
- Describes two processes Producer and Consumer share a common fixed size buffer (queue)
- producer is to either go to sleep or discard data if the buffer is full
  - when consumer consumes a data item, notifies producer
- consumer can go to sleep if it finds the buffer to be empty.
  - when producer puts data into the buffer, it wakes up the sleeping consumer.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.
- Best solution to use semaphore

contd…

Contains buffer of size n

3 semaphores:

- full : the number of items to be read from the buffer
- empty: number of empty spaces that are available
- mutex : provides mutual exclusion for the accesses to the buffer

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

wait **(*S*): while *S*≤ 0 do *no-op*;**

***S*--;**

signal **(*S*): *S++;***

Producer Process:

do {

…

produce an item in **nextp**

…

wait(empty);

wait(mutex);

…

add **nextp** to buffer

…

signal(mutex);

signal(full);

} while (1);

Consumer Process:

do {

wait(full)

wait(mutex);

…

remove an item from buffer to **nextc**

…

signal(mutex);

signal(empty);

…

consume the item in **nextc**

…

} while (1);