The Shared Data Problem (1)

- Consider the following shared memory code:

  ```
  Shared (Global)
  int x = 1;
  CPU 0
  x = x + 1;
  CPU 1
  x = x + 1;
  ```

- Machine Code has \(20 = \frac{6!}{3!3!}\) executions and 2 possible values of \(x\) (=2 and =3)

  ```
  Shared (Global)
  int x = 1;
  CPU 0
  1) Load R1,x
  2) Inc R1
  3) Store R1,x
  CPU 1
  1) Load R1,x
  2) Inc R1
  3) Store R1,x
  ```

Each CPU has its own registers.

The Shared Data Problem (2)

```
 x = 1
```
Synchronization/Mutual Exclusion

- There is a race condition between the two threads
  - Because the outcome depends on the relative execution times of each thread

- Synchronize threads to provide mutual exclusion
  - Coordinate execution schedules of the threads so that one thread finishes updating the shared variables before another thread accesses the same shared variables

- Critical Section: A code segment requiring synchronization

- Need Entry/Exit Sections

```
[ Entry Section ] ... Critical Section ... [ Exit Section ]
```

“Guards” critical section

Shared Ring Buffer Problem

- Shared Data
  ```
  int buffer[N]; // N integer buffers
  int in, out;
  in = 0; // index to next input buffer
  out = 0; // index to next output buffer
  count = 0; // number of buffers in use
  ```

Ring Buffer Producer/Consumer

```
  do { // Producer
      ... compute newItem ...
      while (count == N) { ... do nothing ... }
      buffer[in] = newItem;
      in = mod (in+1, N);
      count = count + 1;
    }
    do { // Consumer
      while (count == 0) { ... do nothing ... }
      outItem = buffer[out];
      out = mod (out+1, N);
      count = count - 1;
      ... use outItem ...
    }
  }
```

- Is this program correct?
  - Only if arithmetic operations on 'count' are atomic

Requirements For Mutual Exclusion

- Enforcement
  - Enforce mutual exclusion for critical sections sharing same objects

- Isolation
  - A process that halts outside all critical sections should not interfere with other processes

- Bounded Waiting
  - No deadlock or starvation

- Progress
  - When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay

- Delay Insensitive
  - Make no assumptions about relative process speeds

- Finite Blocking
  - A process remains in its critical section for only a finite time
Software Approaches 1 and 2

Can a software approach provide mutual exclusion?

// Algorithm 1
shared int who = 0;
process(i) {
    while (who != i)
    {
        ... do nothing ...
    }
    ... critical section ...
    who = (i+1) mod N;
}

- Process must take turns
- Speed is dictated by slowest process
- If a process fails, other processes will be blocked

// Algorithm 2
shared int flag[N] = {0,...};
process(i) {
    while (any flag[j] == 1)
    {
        ... do nothing ...
    }
    flag[i] = 1;
    ... critical section ...
    flag[i] = 0;
}

- No guarantee of mutual exclusion EVEN if memory operations are atomic

Software Approaches 3 and 4

// Algorithm 3
shared int flag[N] = {0,...};
process(i) {
    flag[i] = 1; // moved earlier
    while (any flag[j] != i) == 1)
    {
        ... do nothing ...
    }
    ... critical section ...
    flag[i] = 0;
}

- Guarantees mutual exclusion
- 2 processes simultaneously set flags ➞ Deadlock

// Algorithm 4
shared int flag[N] = {0,...};
process(i) {
    flag[i] = 1; // backoff
    while (any flag[j] != i) == 1) {
        flag[i] = 0;
        ... delay ...
        flag[i] = 1;
    }
    ... critical section ...
    flag[i] = 0;
}

- Potential for livelock (infinite repetitive rollback)

PETESON’S ALGORITHM

- Peterson’s 2-Process Algorithm
  » Simple, elegant solution
  » As in Algorithm’s 1-4, writes to memory are sequential

shared turn = 0; shared flag[N] = {0,...};
process(i) {
    int other = (i+1) mod 2; // other process # (LOCAL)
    flag[i] = 1; // try to gain entry (GLOBAL)
    turn = other; // GLOBAL, LOCAL
    while (flag[other] and (turn == other))
    {
        ... do nothing ...
    }
    ... critical section ...
    flag[i] = 0;
}

Mutual Exclusion (Peterson)

- Mutual exclusion is preserved
  » Suppose Process 0 has set flag[0] to 1; i.e., it wants to enter the critical section
  » 2 possibilities
    » Process 1 can not enter critical section (i.e., while loop blocks because it executed ‘turn = 0’ AFTER Process 0 executed ‘turn = 1’)
    » Process 1 is already in critical section, flag[1] = 1, and it executed ‘turn = 0’ BEFORE Process 0 executed ‘turn = 1’
  » ‘while ((flag[other]) and (turn == other))’
    » If 1, other process wants to enter but has either:
      » Not reached 3rd statement (‘turn = other’) yet, or
      » Executed the 3rd statement EARLIER
    » Imposes FIFO order on entry to critical section
  » Prevents 1 process from monopolizing critical section

Mutual Exclusion is preserved
Hardware Support

TestAndSet(Lock) Semantics
- The following is executed "atomically" in hardware:
  
  ```
  tmp = Lock
  Lock = 1;
  return tmp;
  ```
- If TestAndSet(Lock) = 1, someone else already has the lock
- If TestAndSet(Lock) = 0, lock is free and is set to 1 by the call

TestAndSet(Lock) Usage (Spin-Lock)

```java
while (TestAndSet(Lock) > 0) do {nothing}; // spin (busy wait)
... Critical Section ...
Lock = 0;
```
**A Critical Section Using Semaphores**

- X is a binary semaphore (count field is 0 or 1)
- Initially, Semaphore X = 1 (i.e., X.count = 1)

```
semaphore X = 1;  // declaration;  X.count = 1

...  
wait(X);  // X.count decremented to 0  
...  
Critical Section  
...  
signal(X);  // X.count incremented to 1
```

**NOTE:** This is NOT C/C++ syntax!
- Only an abstract syntax

**Usage Example**

```c
semaphore X = 1;
semaphore Y = 0;

wait(X);
n = n + 1;
if (n < N) {
    signal(X);
    wait(Y);
} else {
    n = 0;
    signal(X);
    for (i=0; i < N-1; i++) signal(Y);
}
```

**Deadlock And Starvation**

- **Deadlock:** Circular Waiting
- **Blocking**
  - A process is prevented from entering a critical section because another process is already in the critical section
  - If a set of processes are mutually blocked, that set is deadlocked
- **Liveloop**
  - A situation in which a set of processes make no progress even though there is no blocking
- **Starvation**
  - Indefinite blocking while other processes progress
  - Example: Processes 0 and 1 pass a critical section back and forth even though other processes want the critical section

**Deadlock Example**

```
<< P0 >>  << P1 >>
Wait(S);  Wait(Q);
Wait(Q);  Wait(S);
...
Signal(S);  Signal(Q);
Signal(Q);  Signal(S);
```