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 $20 = 6!/3!3!$ executions and 2 possible values of $x$ (=2 and =3)

The Shared Data Problem (2)

- The problem exists even if there is only 1 CPU!

The Shared Data Problem (3)

- The problem exists even if there is only 1 CPU!
**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**
  - These "guard" the critical section

---

**Shared Ring Buffer Problem**

![Diagram of shared ring buffer](image)

- **M Consumers**
  - \( M = M' = 1 \)

- **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**

```c
do {
    ... compute newItem ...
    while (count == N) { ... do nothing ... }
    buffer[in] = newItem;
    in = mod (in+1, N);
    count = count + 1;
} do {
    ... use outItem ...
}
```

- **Is this program correct?**
  - Only if \( M=1 \) and 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
- Process must take turns
- Speed is dictated by slowest process
- If a process fails, other processes will be blocked

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) {
    flag[i] = 0;  // backoff
    ... delay ...
    flag[i] = 1;
  }  // 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;
  while (any flag[j != i] == 1) {
    flag[i] = 0;  // backoff
    ... delay ...
    flag[i] = 1;
  }  // critical section ...
  flag[i] = 0;
}

- Potential for livelock (infinite repetitive rollback)

PETEERSON’S ALGORITHM

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

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

- 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
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)
  ```
  while (TestAndSet(Lock) > 0)  do {nothing}; // spin (busy wait)
  Critical Section ...
  Lock = 0;
  ```

Properties Of Hardware Support

- Advantages
  - Applicable to any number of processes and any number of processors
  - Simple and easy to verify
  - Can support multiple critical sections (associate a different variable to each critical section)
- Disadvantages
  - Busy waiting wastes processor cycles (just like software-only solutions)
  - Starvation is possible because of arbitrary selection of waiting process
  - Deadlock is possible
- Need alternative mechanisms

Semaphores

- A synchronization mechanism that doesn’t require much busy waiting
- A semaphore is a non-negative integer count and a queue of threads
  - Count is initialized to the number of free resources.
  - Threads atomically increment the count when resources are added, and atomically decrement the count when resources are removed.
  - When the count becomes 0 (i.e., depleted resources) threads trying to decrement the semaphore will block until count > 0.
- Counting Semaphores
  - Semaphore is typically used to coordinate access to resources (e.g., shared variable).
  - A binary semaphore is a special case (max count = 1)

Counting Semaphores Implementation

```c
wait(S): while (TestAndSet(S.lock) > 0)  { ... do nothing ... };
if (S.count > 0) {
    S.count = S.count - 1;
    S.lock = 0;  // count can be modified now
} else {
    Enter process into S.queue;
    S.lock = 0;
    Yield CPU;
}
signal(S): while (TestAndSet(S.lock) > 0)  { ... do nothing ... }
if (S.queue not empty) {
    Remove first process from S.queue and put on the run queue;
} else {
    S.count = S.count + 1;
}
S.lock = 0;
```

There are other implementations
**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)

```c
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
- **Livelock**
  - 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**

S and Q are semaphores initialized to 1

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

*Note*

» deadlock is NOT guaranteed
» depends on timing
Bank Teller Problem (1)

- N customers and M tellers, N > M
- Customers repeatedly arrive at random times to the bank
- Leave (not enter the lobby) if the lobby is full
- The lobby can hold at most 20 customers
- Return a random time later (after service or full lobby)
- Tellers each serve one customer at a time
- Teller should signal he/she is ready before customer can come to the teller window
- There is exactly one line
- Use semaphores to show the synchronization between the customers and tellers

Bank Teller Problem (2)

- N customer processes and M teller processes, N > M
- Customers repeatedly arrive at random times to the bank
- Leave (not enter the lobby) if the lobby is full
- The lobby can hold at most 20 customers
- Return a random time later (after service or full lobby)
- Tellers each serve one customer at a time
- Teller should signal he/she is ready before customer can come to the teller window
- There is exactly one line
- Use semaphores to show the synchronization between the customers and tellers

Bank Teller Problem (3)

- Limited resources
  - M tellers
  - Lobby capacity = 20
- Simplifications
  - N <= 20
  - No lobby capacity constraint or wait outside of lobby
  - M = 1

Problem 1

- N <= 20 and M = 1

```c
Process customer (int i) {
    do forever {
        ... Random delay ...
        Wait(tRdy);
        Signal(cRdy);
        ... Get service ...
        Wait(tDone);
        ... Leave bank ...
    }
}

Process teller (int i) {
    do forever {
        Signal(tRdy);
        Wait(cRdy);
        ... Serve customer ...
        Wait(tDone);
    }
}
```

Semaphore tRdy = 0;
cRdy = 0;
tDone = 0;
Problem 2

- $M = 1$, $N > 20$ (wait outside if lobby is full)

Semaphore tRdy = 0, cRdy = 0, tDone = 0;
capacity = 20; // forms queue

Process customer (int i) {
    do forever {
        ... Random delay ...
        Wait(capacity);
        Si gnal(cRdy);
        ... Get service ...
        Wait(tDone);
        Si gnal(capacity);
        ... Leave bank ...
    }
}

Process teller (int i) {
    do forever {
        Si gnal(tRdy);
        Wait(cRdy);
        ... Serve customer ...
        Si gnal(tDone);
    }
}

Problem 3

- $M = 1$, go away for random time if lobby is full
  - Don't use 'capacity' semaphore because customer can't get out of capacity queue
  - Replace 'capacity' semaphore with a protected counter

Semaphore tRdy = 0, cRdy = 0, tDone = 0;
int n; // # in lobby
Semaphore nLock = 0; // protect n

Process customer (int i) {
    do forever {
        do {
            ... Random delay ...
        } until \[ n < 20 \&\& \]
        n = n + 1;
        Wait(tRdy);
        Si gnal(cRdy);
        ... Get service ...
        Wait(tDone);
        \[ n = n - 1; \]
        ... Leave bank ...
    }
}

Process teller (int i) {
    do forever {
        Si gnal(tRdy);
        Wait(cRdy[i]);
        ... Serve customer ...
        Si gnal(tDone);
    }
}

[[ ... ]] means:
Wait(nLock): ... Signal(nLock):

Problem 4 (Incomplete)

- $M = 1$, go away for random time if lobby is full
  - Don't use 'capacity' semaphore because customer can't get out of capacity queue
  - Replace 'capacity' semaphore with a protected counter

Semaphore tRdy = 0, cRdy[N] = 0, tDone = 0;
int n; // # in lobby
Semaphore nLock = 0; // protect n
int tFree[M]; // 1 if teller i is free

Process customer (int i) {
    do forever {
        Si gnal(tRdy);
        Wait(cRdy);
        ... Serve customer ...
        Si gnal(tDone);
    }
}

Process teller (int i) {
    do forever {
        do {
            ... Random delay ...
        } until \[ n < 20 \&\& \]
        n = n + 1;
        Wait(tRdy);
        Si gnal(cRdy[i]);
        ... Get service ...
        Wait(tDone);
        \[ n = n - 1; \]
        ... Leave bank ...
    }
}

How will teller know which customer to serve?