**Synchronization (CS422S)**

Ken Wong  
Washington University  
kenw@wustl.edu  
www.arl.wustl.edu/~kenw

---

**The Shared Data Problem (1)**

- Consider the following shared memory code:
  ```
  CPU 0  
  int x = 1;  
  x = x + 1;  
  
  CPU 1  
  int x = 1;  
  x = x + 1;
  ```
- Machine Code has $20 = 6/(3!3!)$ executions and 2 possible values of $x (\pm 2$ and $\pm 3$)

**The Shared Data Problem (2)**

![Diagram showing CPU 0 and CPU 1 with shared memory](image)

**The Shared Data Problem (3)**

- The problem exists even if there is only 1 CPU!
  ```
  CPU 0  
  Shared (Global)  
  int x = 1;  
  1) Load R1.x  
  2) Inc R1  
  3) Store R1.x  
  
  CPU 1  
  Shared (Global)  
  int x = 1;  
  1) Load R1.x  
  2) Inc R1  
  3) Store R1.x  
  ```
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]

Ring Buffer Producer/Consumer

```java
do {
    // Producer
    ... compute newItem ...
    while (count == N) { ... do nothing ... }
    buffer[n] = newItem;
    in = mod (in+1, N);
    count = count + 1;
}
```

```java
... 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

Shared Ring Buffer Problem

```
\[
\begin{array}{c|c|c|c}
\text{M Consumers} & M = 1 & \text{M Producers} \\
\hline
\text{out} \downarrow & \text{in} \uparrow \\
\end{array}
\]
```

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

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

Software Approaches 3 and 4

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

Peterson’s Algorithm

- Peterson’s 2-Process Algorithm
  » Simple, elegant solution

  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
    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[0] = 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 {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.
  - Starvation is possible because of arbitrary selection of waiting process.
  - Deadlock is possible.

- **Need alternative mechanisms**

Semaphores

- A synchronization mechanism that doesn't require busy waiting.

- **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).

Counting Semaphores Implementation

- `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;`
    - `S.count = 0;`
    - `S.queue = 0;`
    - Suspend: // no matching signal
  }

- `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;`