PROCESS SYNCHRONIZATION

Ken Wong
Washington University, St. Louis

kenw@arl.wustl.edu
www.arl.wustl.edu/~kenw

THE SHARED DATA PROBLEM (1)

<table>
<thead>
<tr>
<th>Shared (Global)</th>
<th>PC 0</th>
<th>PC 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 1;</td>
<td>(1) x = x+1;</td>
<td>(3) x = 2^x;</td>
</tr>
<tr>
<td>int y = 1;</td>
<td>(2) y = y+1;</td>
<td>(4) y = 2^y;</td>
</tr>
</tbody>
</table>

• Result \(6 = 4!/(2!2!))\) executions

<table>
<thead>
<tr>
<th>Execution Order</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,3,2,4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,3,4,2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3,1,2,4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3,1,4,2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3,4,1,2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

THE SHARED DATA PROBLEM (2)

• How do we implement a program that uses 2 program counters?

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP → PC</td>
<td>SP → PC</td>
</tr>
<tr>
<td>Registers</td>
<td>Registers</td>
</tr>
</tbody>
</table>

Main Memory

x = 1
y = 1

SHARED BUFFER PROBLEM

<table>
<thead>
<tr>
<th>0</th>
<th>N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>in</td>
</tr>
</tbody>
</table>

M Consumers M Producers

M = 1

• Shared Data

```c
int buffer[N]; // N integer buffers
int in, out;  // index to next input buffer
in = 0;       // index to next output buffer
out = 0;      // number of buffers in use
```
**PRODUCER AND 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 ...
}

**CRITICAL SECTION**

- **Critical Section**: A code segment which accesses shared data that should only be accessible to one process at a time.

- **Mutual Exclusion**
  
  If a process is executing a critical section C which accesses a set of shared variables S, then no other critical section accessing any variables in S should be executing at the same time.

- **Need Entry/Exit Sections**
  
  [ Entry Section ]
  ...
  Critical Section ...
  [ Exit Section ]

**REQUIREMENTS FOR MUTUAL EXCLUSION**

- ** Enforcement**: Mutual exclusion must be enforced for critical sections sharing the same object.

- **Isolation**: A process that halts in its noncritical section 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**: No assumptions are made about the relative process speeds or number of processors.

- **Finite Blocking**: A process remains inside its critical section for only a finite time.

**ALGORITHM 1 (RELAY)**
**ALGORITHM 1 (RELAY)**

- Based on *busy waiting*
  - The shared variable $\text{who}$ indicates the process that should be allowed to enter the critical section
- Process i's Algorithm
  ```c
  while (who != i) { ... do nothing (busy wait) ... }
  ... critical section ...
  who = i+1 mod N; // pass control to next process
  ```
- **Drawbacks**
  - Processes must take turns
  - Speed is dictated by the slowest process
  - If a process fails, the other process is permanently blocked
- **Coroutine control structure**

**ALGORITHM 2 (CONTENTION)**

- The N *global* variables $\text{flag}[i], i=0..(N-1)$ indicate by the value TRUE that process $i$ is allowed into the critical section
- Process i's Algorithm
  ```c
  while (any $\text{flag}[j]$ is TRUE) { ... busy wait ... }
  $\text{flag}[i] = \text{TRUE};$
  ... critical section ...
  $\text{flag}[i] = \text{FALSE};$
  ```
- **Fixes to Algorithm 1**
  - Processes no longer must take turns: Fight (contend) for entry to critical section
  - If a process fails OUTSIDE its critical section, the other process is NOT permanently blocked

---

**ALGORITHM 3 and 4**

- **Algorithm 3 (Process i):** Same as Alg. 2, but set $\text{flag}[i]$ sooner.
  ```c
  $\text{flag}[i] = \text{TRUE};$ // moved in front of while loop
  while (any $\text{flag}[j]$ != i is TRUE) { ... busy wait ... }
  ... critical section ...
  $\text{flag}[i] = \text{FALSE};$
  ```
- **Algorithm 4 (Process i):** Same as Alg. 3, but allow process to backoff and defer to another process ("mutual courtesy").
  ```c
  $\text{flag}[i] = \text{TRUE};$ // try to gain entry
  while (any $\text{flag}[j]$ is TRUE) {
      $\text{flag}[i] = \text{FALSE};$ // backoff
      ... delay ...
      $\text{flag}[i] = \text{TRUE};$
  }
  ... critical section ...
  $\text{flag}[i] = \text{FALSE};$
  ```
DEKKER’S ALGORITHM

- **Idea 1**: Allow each process to examine state of other processes (e.g., Alg. 2-4)
- **Idea 2**: Avoid mutual courtesy (impose an order on activities (e.g., Alg. 1))

**Process i’s Algorithm (2 Process Version)**

```c
int other = (i+1) mod 2;  // other process # (LOCAL)
flag[i] = TRUE;           // try to gain entry
while (flag[other]) {
    if (turn == other) {  // turn is GLOBAL VARIABLE
        flag[i] = FALSE;  // backoff
        while (turn == other) { ... do nothing ... }  
        flag[i] = TRUE;
    }
}
... critical section ...
turn = other;
flag[i] = FALSE;
```

PETTERSON’S ALGORITHM

- **Drawback of Dekker’s Algorithm**
  
  - Complex program that is difficult to follow and whose correctness is tricky to prove.

- **Peterson’s Algorithm**: A simple and elegant solution

  - **Global Variables**: flag[i], turn

  **Process i’s Algorithm (2 Process Version)**

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

LAMPORT’S BAKERY ALGORITHM

- Each process receives a number before entering its critical section
- Smallest number has precedence
- If two processes i and j have the same number and i < j, then i is served first; else j is served.
- Numbers are generated in non-decreasing order (e.g., 1, 2, 3, 3, 4, 5)
- **Semantics of (a, b) < (c, d)**
  
  - True if (a < c) or if (a = c) and (b < d)

LAMPORT’S BAKERY ALGORITHM

- **Global Variables**
  
  - choosing[0..N-1], boolean array, initially all FALSE
  - num[0..N-1], integer array, initially all 0

- **Process i’s Algorithm**

  ```c
  choosing[i] = TRUE;
  num[i] = 1 + max(num[0], ..., num[N-1]);
  choosing[i] = FALSE;
  for j=0 to N-1 {
    while (choosing[j]) { ... do nothing ... }
    while ((num[j] != 0) and ((num[j], j) < (num[i], i))
      { ... do nothing ... }
  }
  ... critical section ...
  num[i] = 0;
  ```
**HARDWARE SUPPORT**

- **TestAndSet(Lock) Semantics**
  - The following is executed "atomically" in hardware:
    ```c
    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 region ...
  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.

- 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).

---

**COUNTING SEMAPHORES IMPLEMENTATION**

- **wait(S):**
  ```
  while (TestAndSet(S.lock) > 0) { ... 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;
    Suspend; // no matching signal
  }
  ```

- **signal(S):**
  ```
  while (TestAndSet(S.lock) > 0) { ... nothing ... };
  if S.queue not empty then {
    Remove first process from S.queue and put on the run queue;
  } else {
    S.count = S.count + 1; }
  S.lock = 0;
  ```
A CRITICAL SECTION USING SEMAPHORES

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

```java
wait(X);  // X.count decremented to 0
... critical section ...
signal(X);  // X.count incremented to 1
```

USAGE EXAMPLE

- Initially, n = 0; X = 1; Y = 0;

```java
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.

- **Deadlock Example**
  - S and Q are semaphores initialized to 1

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

- **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