Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions - skip
- Monitors -skip
- Synchronization in Solaris 2 & Windows 2000 - skim

Motivation

- Synchronization is needed to coordinate \textit{CONCURRENT PROCESSES} when they share resources.
  - multiprogrammed uniprocessor
  - uniprocessor and I/O processor
  - shared memory multiprocessor
  - distributed memory multiprocessor
  - network of workstations
  - world wide web computing
Support for Synchronization

- **HARDWARE:**
  - interrupts
  - atomic machine instructions
  - test-and-set primitive
  - spinlocks

- **OPERATING SYSTEM**
  - software solutions
  - semaphores
  - message-passing
  - logical clocks
  - event counts and sequencers

Support (cont’d)

- **LIBRARY ROUTINES**
  - monitors
  - message-passing
  - rendezvous
  - remote procedure call
  - path expressions
  - conditional critical regions
Support (cont’d)

- PROGRAMMING LANGUAGES
  - Concurrent Pascal (monitors/Hansen)
  - Path Pascal (path expressions/Campbell and Habermann)
  - CSP (messages/Hoare)
  - Modula (monitors/Wirth)
  - Mesa (monitors/Xerox)
  - Ada (rendezvous/DOD)
  - Concurrent Euclid (monitors, Holt)
  - Actors (messages/Agha)

Producer-Consumer Problem

- Producer-Consumer with Bounded Buffer
  - Circular buffer of size N.
  - Producer puts items into buffer.
  - Consumer takes items out of buffer.
  - No guarantee on the relative order or speed of the producer and the consumer.
- Problem: how to guarantee that items are consumed in the order that they are produced?
Producer-Consumer Solution

SOLUTION 1:
- PRODUCER CODE: (IN = a counter to keep track of produced items)
  - 1. Initially IN = 0
  - 2. Put item in BUFFER[IN]
  - 3. IN = IN + 1 (mod N)
  - 4. Go to 2

Producer-Consumer Solution

- CONSUMER CODE: (OUT = a counter to keep track of produced items)
  - 1. Initially OUT = 0
  - 2. Copy item out of BUFFER[OUT]
  - 3. OUT = OUT + 1 (mod N)
  - 4. Go to 2
Producer-Consumer Solution

**SOLUTION 2**: (COUNT is a global variable that keeps track of number of items in the buffer)

- **PRODUCER CODE**:
  - 1. Initially, \( \text{IN} = 0 \), \( \text{COUNT} = 0 \)
  - 2. While (\( \text{COUNT} = N \)) loop
  - 3. Put item in \( \text{BUFFER}[\text{IN}] \)
  - 4. \( \text{IN} = \text{IN} + 1 \) (mod \( N \))
  - 5. \( \text{COUNTER} = \text{COUNTER} + 1 \)
  - 6. Go to 2

Producer-Consumer Solution

- **CONSUMER CODE**:
  - 1. Initially, \( \text{OUT} = 0 \), \( \text{COUNT} = 0 \)
  - 2. While (\( \text{COUNT} = 0 \)) loop
  - 3. Copy item out of \( \text{BUFFER}[\text{OUT}] \)
  - 4. \( \text{OUT} = \text{OUT} + 1 \) (mod \( N \))
  - 5. \( \text{COUNTER} = \text{COUNTER} - 1 \)
  - 6. Go to 2
Producer-Consumer Closer Look

- **ASSEMBLY CODE:**
- **PRODUCER CODE** for Line 5: \( \text{COUNT} = \text{COUNT} + 1 \)
  - 1. LOAD ACC,COUNT
  - 2. ADD ACC,1
  - 3. STORE ACC,COUNT
- **CONSUMER CODE** for Line 5: \( \text{COUNT} = \text{COUNT} - 1 \)
  - 1. LOAD ACC,COUNT
  - 2. SUB ACC,1
  - 3. STORE ACC,COUNT

Bounded Buffer

- The statements
  
  \[
  \text{COUNT} = \text{COUNT} + 1;  \\
  \text{COUNT} = \text{COUNT} - 1;  \\
  \]

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.
Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

Assume `counter` is initially 5. One interleaving of statements is:

producer: `register1 = counter` \((register1 = 5)\)
producer: `register1 = register1 + 1` \((register1 = 6)\)
consumer: `register2 = counter` \((register2 = 5)\)
consumer: `register2 = register2 - 1` \((register2 = 4)\)
producer: `counter = register1` \((counter = 6)\)
consumer: `counter = register2` \((counter = 4)\)

The value of `count` may be either 4 or 6, where the correct result should be 5.
Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be synchronized.

The Critical-Section Problem

- $n$ processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
Solution to Critical-Section Problem

1. **Mutual Exclusion.** If process \( P_i \) is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting.** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   
   Assume that each process executes at a nonzero speed
   
   No assumption concerning relative speed of the \( n \) processes.

Initial Attempts to Solve Problem

- Only 2 processes, \( P_0 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P_j \))

```plaintext
do {
  entry section
  critical section
  exit section
  reminder section
} while (1);
```

Processes may share some common variables to synchronize their actions.
Algorithm 1

- Shared variables:
  - `int turn;`
  - Initially `turn = 0`
  - `turn - i` if `P_i` can enter its critical section
- Process `P_i`
  ```
  do {
    while (turn != i);
    critical section
    turn = j;
    reminder section
  } while (1);
  ```
- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - `boolean flag[2];`
  - Initially `flag[0] = flag[1] = false`
  - `flag[i] = true` if `P_i` ready to enter its critical section
- Process `P_i`
  ```
  do {
    flag[i] := true;
    while (flag[i]);
    critical section
    flag[i] = false;
    remainder section
  } while (1);
  ```
- Satisfies mutual exclusion, but not progress requirement.
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process $P_i$
  
  ```
  do {
    flag [i] := true;
    turn = j;
    while (flag [j] and turn = j) ;
    critical section
    flag [i] = false;
    remainder section
  } while (1);
  ```
- Meets all three requirements; solves the critical-section problem for two processes.

Bakery Algorithm

Critical section for $n$ processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
Bakery Algorithm

- Notation <= lexicographical order (ticket #, process id #)
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max (a₀, ..., aₙ₋₁) is a number, k, such that k ≥ aᵢ for i = 0, ..., n - 1

- Shared data
  
  boolean choosing[n];
  int number[n];

  Data structures are initialized to false and 0 respectively

---

Bakery Algorithm

```c
do {
  choosing[i] = true;
  number[i] = max(number[0], number[1], ..., number[n - 1]) + 1;
  choosing[i] = false;
  for (j = 0; j < n; j++) {
    while (choosing[j]) ;
    while ((number[j] != 0) && (number[j] < number[i])) ;
  }
  critical section
  number[i] = 0;
  remainder section
} while (1);
```
Synchronization Hardware

- Test and modify the content of a word atomically

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

Mutual Exclusion with Test-and-Set

- Shared data:
  ```java
  boolean lock = false;
  ```

- Process \( P_i \)
  ```java
  do {
      while (TestAndSet(lock)) ;
      critical section
      lock = false;
      remainder section
  }
  ```
Synchronization Hardware

- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to false):
  ```
  boolean lock;
  boolean waiting[n];
  ```

- Process $P_i$
  ```
  do {
    key = true;
    while (key == true)
      Swap(lock,key);
    lock = false;
    critical section
    remainder section
  }
  ```
Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations
  
  **wait** ($S$):
  
  ```
  while $S \leq 0$ do no-op;
  S--;
  ```

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

Critical Section of $n$ Processes

- Shared data:
  
  ```
  semaphore mutex; //Initially mutex = 1
  ```

- Process $P_i$:
  
  ```
  do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
  } while (1);
  ```
Semaphore Implementation

- Define a semaphore as a record
  ```c
  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
  ```c
  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);
      }
  ```
Semaphore as a General Synchronization Tool

- Execute B in \( P_j \) only after A executed in \( P_i \)
- Use semaphore flag initialized to 0
- Code:

\[
\begin{array}{c}
P_i \quad P_j \\
\text{\quad } \text{\quad }
\hline
\text{\quad } \text{\quad }
\text{\quad } \text{\quad }
\text{\quad } \text{\quad }
\end{array}
\]

\[
\begin{array}{cccc}
P_i & P_j \\
A & \text{wait(flag)} \\
\text{signal(flag)} & B
\end{array}
\]

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.
- Let \( S \) and \( Q \) be two semaphores initialized to 1

\[
\begin{array}{c}
P_0 \quad P_1 \\
\text{\quad } \text{\quad }
\hline
\text{\quad } \text{\quad }
\text{\quad } \text{\quad }
\text{\quad } \text{\quad }
\end{array}
\]

\[
\begin{array}{cccc}
P_0 & P_1 \\
\text{wait}(S); & \text{wait}(Q); \\
\text{wait}(Q); & \text{wait}(S); \\
\text{signal}(S); & \text{signal}(Q); \\
\text{signal}(Q) & \text{signal}(S)
\end{array}
\]

- **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.
- Can implement a counting semaphore $S$ as a binary semaphore.

Implementing $S$ as a Binary Semaphore

- Data structures:
  - binary-semaphore $S1$, $S2$;
  - int $C$;
- Initialization:
  - $S1 = 1$  /* use this for mutex
  - $S2 = 0$  /* use this for blocking
  - $C =$ initial value of semaphore $S$
  - /* use this for the count
Implementing $S$

- **wait operation**
  ```
  wait(S1);
  C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  signal(S1);
  ```

- **signal operation**
  ```
  wait(S1);
  C ++;
  if (C <= 0)
    signal(S2); /* pass mutex to waiting process
  else
    signal(S1);
  ```

Semaphore Implementation

- **Problem:**
  - $P(S)$ and $V(S)$ must be atomic. The result of concurrent execution of combinations of many $P()$ and $V()$ must execute as if each were a single indivisible instruction.

- **Key Issue:**
  - The $P$ and $V$ code are themselves critical sections!
Semaphore Implementation

- Solution:
  - Protect P and V code with test-and-set.
    P(S): while (test-and-set(lock)) do no-op;
      --- code for P(S) ---
      lock = false;
    V(S): while (test-and-set(lock)) do no-op;
      --- code for V(S) ---
      lock = false;

Semaphore Implementation (cont’d)

The reason why we allow busy-waiting with test-and-set, but wish to eliminate it for P() and V() is because the critical section protected by test-and-set is a set limited amount of code within the OS. The critical sections protected by P and V are an undefined amount of code at the user level; busy-waiting for a user to execute a large amount of code is undesirable.
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

(Producer Consumer Problem)

- Shared data

  semaphore full, empty, mutex;
  /* mutex to protect the buffer slot
  /* full for keep track of filled slots and block
  consumer when 0 items in buffer
  /* empty to keep track of empty slots and block
  producer when buffer is full (0 empty slots)

Initially:

  full = 0, empty = n, mutex = 1
Bounded-Buffer Problem Producer Process

```c
    do {
        ...
        produce an item in nextp
        ...
        wait(empty);
        wait(mutex);
        ...
        add nextp to buffer
        ...
        signal(mutex);
        signal(full);
    } while (1);
```

Bounded-Buffer Problem Consumer Process

```c
    do {
        wait(full)
        wait(mutex);
        ...
        remove an item from buffer to nextc
        ...
        signal(mutex);
        signal(empty);
        ...
        consume the item in nextc
        ...
    } while (1);
```
Readers-Writers Problem

- Shared data with many readers, only one writer allowed
- While writing occurs, no reading should occur

```
semaphore mutex, wrt;
```

Initialy

```
mutex = 1, wrt = 1, readcount = 0
```

---

Readers-Writers Problem Writer Process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```
Readers-Writers Problem Reader Process

wait(mutex);
readcount++;
if (readcount == 1) /* the first reader
    wait(wrt);
signal(mutex);
... reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0) /* the last reader
    signal(wrt);
signal(mutex);

Dining Philosophers Problem

Five philosophers are seated around a circular table. In the center is a bowl of rice and the table is laid with five single chopsticks. Each chopstick is shared by two philosophers. The challenge is to write code to synchronize the five philosophers so that each can execute the following basic code:

**Philosophers' Algorithm**
- Take right fork
- Take left fork
- EAT
- Replace both forks

Problem: Each philosopher may pick up right fork, and
Philosophers Solution #1

SOLUTION 1: non-semaphore solution
chopstick[0..4]: array of semaphore, initialized to 1;
repeat forever {  THINK
  repeat (  THINK
    if (chopstick[0] == 1) {  THINK
      if left free, pick up left;
      if (chopstick[1] == 1) {  THINK
        if right free, pick up right;
      } else {  THINK
        put down left;
      }
    }
  }
  until (chopstick[0] == 0 and chopstick[1] == 0); until I've got both.
EAT
  put down left;
  chopstick[0] = 1;
  put down right;
  chopstick[1] = 1;
}

Philosophers Solution #2

SOLUTION 2: protect each chopstick separately
chopstick[0..4]: array of binary semaphore, initialized to 1;
repeat forever {  THINK
  P(chopstick[0]);  THINK
  pick up left;  THINK
  P(chopstick[1] mod 5));  THINK
  pick up right;  THINK
  EAT
  put down left;  THINK
  V(chopstick[0]);  THINK
  tell next in line left is free
  put down right;  THINK
  V(chopstick[1] mod 5));  THINK
  tell next in line right is free
}
Philosophers Solution #3

SOLUTION 3: check to see if both chopsticks are free, if so, pick up both.

`chopstick[0..4]:` array of binary semaphore; all initialized to 1.

```
repeat forever {
    THINK:  
P(chopstick[0] mod 5));
    pick up both chopsticks
    EAT:     
    put down both chopsticks
    V(chopstick[0]);
    V(chopstick[1] mod 5));
}
```

Philosophers Solution #4

SOLUTION 4: only allow four philosophers to sit at the table at a time.

`table: ` counting semaphore, initialized to 0;
`chopstick[0..4]: ` array of binary semaphore, all initialized to 1;

```
repeat forever {
    THINK:  
P(table);
    sit down
    P(chopstick[0]);
    pick up left;
    P(chopstick[0+1 mod 5]);
    pick up right;
    EAT:     
    put down left
    V(chopstick[0]);
    put down right
    V(chopstick[0+1 mod 5]);
    V(table);
}
```