Process Synchronization
- Background
- The Critical-Section Problem
- Software Solutions
- Semaphores
- Hardware Support
- Classical Problems of Synchronization

Motivation
- Synchronization is needed to coordinate 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:
  - atomic machine instructions
  - test-and-set primitive
  - spinlocks
- OPERATING SYSTEM
  - software solutions
  - semaphores
  - message-passing
  - logical clocks
  - event counts and sequencers
- PROGRAMMING LANGUAGES
  - a long history of language constructs
  - monitor approach and Java threads

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: intuitive approach
- 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

CISC 415

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

- **This solution has problems**
  - Fast producer over-writes data that has not yet been consumed
  - Fast consumer re-reads old data

### Producer Consumer Race Condition

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

### Producer Consumer Race Condition

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

### Producer-Consumer Solution

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

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

### Producer Consumer Solution

- **CONSUMER CODE:**
  - 1. Initially, OUT = 0, COUNT = 0
  - 2. While (COUNT = 0) loop /* buffer empty */
  - 3. Copy item out of BUFFER[OUT]
  - 4. OUT = OUT + 1 (mod N)
  - 5. COUNTER = COUNTER - 1
  - 6. Go to 2

### Producer-Consumer Closer Look

- **ASSEMBLY CODE:**
  - **PRODUCER CODE for Line 5:** \( \text{COUNT} = \text{COUNT} + 1 \)
    - P1: LOAD ACC,COUNT
    - P2: ADD ACC,1
    - P3: STORE ACC,COUNT
    - 3 assembly code instructions
  - **CONSUMER CODE for Line 5:** \( \text{COUNT} = \text{COUNT} - 1 \)
    - C1: LOAD ACC,COUNT
    - C2: SUB ACC,1
    - C3: STORE ACC,COUNT
    - 3 assembly code instructions
Producer-Consumer Race Condition

- **Assembly Code:**
  - **Producer Code** for lines 5 & 8: `COUNT = COUNT + 1`
    - P1. LOAD ACC, COUNT
    - P2. ADD ACC, 1
    - P3. STORE ACC, COUNT
  - **Consumer Code** for lines 5 & 8: `COUNT = COUNT - 1`
    - C1. LOAD ACC, COUNT
    - C2. SUB ACC, 1
    - C3. STORE ACC, COUNT

- Assume COUNT ≠ 0. What is the value after these interleavings?
  - P1, P2, P3, C1, C2, C3: COUNT = 4
  - P1, P3, C1, C2, C3: COUNT = 3
  - P1, C1, P3, C2, C3: COUNT = 5

Producer Consumer 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... unpredictable!

- 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.

Critical-Section Problem Requirements

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.

Critical Section Problem Software Solution Structure

- General structure of process \( P_i \) (other process \( P_j \))
  ```
  do {
    entry section
    critical section
    exit section
    remainder section
  } while (1);
  ```
- Processes may share some common variables to synchronize their actions.

Algorithm 1 (for only 2 processes)

- **Shared variables:**
  - int turn;
    - initially \( turn = 0 \)
    - \( turn -1 \) \( \rightarrow P_i \) can enter its critical section
- **Process** \( P_i \)
  ```
  do {
    while (turn != i) ;
    critical section
    turn = j;
    remainder section
  } while (1);
  ```
- Satisfies mutual exclusion, but not progress
Algorithm 2 (only 2 processes)

- Shared variables
  - flag[i] = true ⇒ P_i ready to enter its critical section
- Process P_i
  do {
    flag[i] := true;
    while (flag[j]) ;
    critical section
    flag[i] = false;
  } while (1);
- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3 (2 processes only)

- 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;
  } while (1);
- Meets all three requirements; solves the critical-section problem for two processes.
- Known as Peterson’s Algorithm

Lamport’s Bakery Algorithm (for n processes)

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...

Lamport’s Bakery Algorithm

- Notation ≺ ≡ lexicographical order (ticket #, process id #)
  - (a,b)≺(c,d) if a < c or if a = c and b < d
  - max (a_0,...,a_{n-1}) is a number, k, such that k ≥ a_i for i = 0,...,n - 1
- Shared data
  boolean choosing[n];
  int number[n];
  Data structures are initialized to false and 0 respectively

Lamport’s Bakery Algorithm

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,j] < number[i,i])) ;
  }
  critical section
  number[i] = 0;
  remainder section
} while (1);

Flaws with Software Solutions

- Busy-waiting (spinning on some variable, wastes CPU cycles)
- Error-prone
- Nuisance for the application programmer
**Semaphores (Edsger Dijkstra)**

Important computer scientist in a wide range of areas from algorithms to OS to distributed computing to programming languages and verification

- Dijkstra's shortest path algorithm
- Semaphores
- And Perfect handwriting

**Semaphores (Dijkstra)**

- 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);`

Moves busy-waiting into the semaphore code (not in application programmer’s code)

**Critical Section of $n$ Processes**

- Shared data: `semaphore mutex; // Initially mutex = 1`
- Process $P_i$:
  - `do {
      wait(mutex);
      critical section
      signal(mutex);
    } while (true);`

**Semaphore with blocking (no busywait)**

- Define a semaphore as a record
  - `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 $P$.

**Semaphore with blocking**

- Semaphore operations now defined as
  - `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($P$);
    }`

**Blocking Semaphore Implementation (Semaphores and Scheduling)**

- See handout.
Semaphore Implementation: Atomicity

- 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!

Implementing Semaphores with HW Support

- 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)

- We allow busy-waiting with test-and-set, but wish to eliminate it for P() and V().
- The critical section protected by test-and-set is a small limited amount of code within the OS; thus busy-waiting is acceptable.
- 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.

Synchronization Hardware

- Test and modify the content of a word atomically.
  boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
  }

HW Mutual Exclusion with Test-and-Set

- Shared data:
  boolean lock = false;
- Process P_i
  do {
    while (TestAndSet(lock)) ;
    critical section
    lock = false;
    remainder section
  }

Implementing Semaphores with Test-and-Set: wait/P

- Shared data for semaphore sema
  boolean sema.lock = false;
  integer sema.S = 1;
- Code for wait(sema): {
  do {
    while (TestAndSet(sema.lock)) ;
    while sema.S <= 0 do no-op;
    sema.S--;
    sema.lock = false;
  }
}
Implementing Semaphores with Test-and-Set: `signal/V`

- Shared data for semaphore `sema`
  ```java
  boolean sema.lock = false;
  Integer sema.S = 1;
  ```
- Code for `signal(sema)`
  ```java
  do {
    while (TestAndSet(sema.lock)) {
      sema.S++;
      sema.lock = false;
    }
  }
  ```

Two Types of Semaphores

- **Counting** semaphore – arbitrary positive integer value
- **Binary** semaphore – integer value is only 0 or 1
- Can implement a counting semaphore `S` as a binary semaphore.

Implementing a Counting Semaphore Using a Binary Semaphore

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

Implementing a Counting Semaphore

- **wait** operation
  ```java
  wait(S1):
  C--;
  if (C < 0) {
    signal(S1);
    wait(S2);
  }
  ```
- **signal** operation
  ```java
  wait(S1):
  C++;
  if (C <= 0)
    signal(S2); /* pass mutex to waiting processes */
  else
    signal(S1);
  ```

Classical Synchronization Problems

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

Bounded-Buffer Problem (Producer Consumer Problem)

- Shared data
  ```java
  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:
```java
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);
```

Dining Philosophers Problem

Five philosophers are seated around a circular table. The center of the table is a plate.

Philosopher code:
repeat forever {
    think
    pick up 2 chopsticks
    eat
    put down 2 chopsticks
}

Philosophers Solution #1

```c
SOLUTION 1: non-synchronized solution
chopsticks[n] array of boolean, all initialized to 1;
repeat forever {
    THINK;
    think;
    if (chopsticks[n-1] && chopsticks[n+1]) {
        pick up left;
        pick up right;
        eat;
        put down right;
        put down left;
    }
    else {
        put down right;
        put down left;
        eat;
    }
}
```

Philosophers Solution #2

```c
SOLUTION 2: put each chopstick adjacent to a semaphore
chopsticks[n] array of binary semaphores, all initialized to 1;
repeat forever {
    THINK;
    think;
    if (chopsticks[n-1] && chopsticks[n+1]) {
        pick up left;
        pick up right;
        eat;
        put down right;
        put down left;
    }
    else {
        put down right;
        put down left;
        eat;
    }
}
```

Philosophers Solution #3

```c
SOLUTION 3: check to see if both chopsticks are free, then pick both
chopsticks[n] array of binary semaphores, all initialized to 1;
repeat forever {
    THINK;
    think;
    if (chopsticks[n-1] && chopsticks[n+1]) {
        pick both chopsticks
        eat
        put both chopsticks
    }
    else {
        for (i = n-1; i < n+2; i++)
            if (chopsticks[i] == 1) {
                free one chopstick
                eat
            }
    }
}
```
Philosophers Solution #4

SOLUTION: 1. only allow one philosopher to eat at the table at a time.
2. table holding semaphores, initialized to 0.
3. chopsticks(): 4 binary semaphores, all initialized to 1.

```c
repeat forever
   THINK
   check if space at the table
   all down
   \{
      \{ philosophers() and EAT
         \{ pick up left
            \{ check if left is free
            \{ EAT
            \} put down left
         \} put down right
         \} left need to eat is free
      \} right need to eat is free
      \} up
   \}
```

Readers-Writers Problem

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

```c
semaphore mutex, wrt;
Initially
mutex = 1, wrt = 1, readcount = 0
```

Readers-Writers Problem Writer Process

```c
wait(wrt);
...
writing is performed
...
signal(wrt);
```

Readers-Writers Problem Reader Process

```c
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);
```

Using Semaphores: Precedence

- Execute B in Pj only after A executed in Pi
- Use semaphore flag initialized to 0
- Code:
  ```c
  P_i  P_j
  \vdots \vdots
  A wait(flag)
  signal(flag)  B
  ```

Caution with Semaphores: 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
  ```c
  P_0
  wait(S); wait(Q);
  wait(S); wait(Q);
  signal(Q); signal(S);
  signal(Q); signal(S);
  ```
- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.