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:
  - 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/DOE)
  - 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

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 = 0) loop
  3. Put item in BUFFER[IN]
  4. IN = IN + 1 (mod N)
  5. COUNTER = COUNTER + 1
  6. Go to 2

Producer-Consumer Closer Look

ASSEMBLY CODE:
- PRODUCER CODE for Line 5: COUNT = COUNT + 1
  1. LOAD ACC,COUNT
  2. ADD ACC,1
  3. STORE ACC,COUNT
- CONSUMER CODE for Line 5: COUNT = COUNT - 1
  1. LOAD ACC,COUNT
  2. SUB ACC,1
  3. STORE ACC,COUNT

Producer Consumer Race Condition

The statements
COUNT = COUNT + 1;
COUNT = COUNT - 1;

must be performed atomically.

Atomic operation means an operation that completes in its entirety without interruption.
Producer Consumer Race Condition

- 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.

Producer Consumer Race Condition

- 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 `counter` may be either 4 or 6, where the correct result should be 5.

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.

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 \( P \) (other process \( P \))
  ```
  do {
    entry-section
    critical-section
    exit-section
  } while (1);
  ```
- Processes may share some common variables to synchronize their actions.
Algorithm 1 (for only 2 processes)

- Shared variables:
  - int turn
  - turn - i, P 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;
    - remainder section
  - ) 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;
    - critical section
    - turn = j;
    - remainder section
  - ) while (1);
- Meets all three requirements; solves the critical-section problem for two processes.
- Dekker’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, ...

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) 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

Bakery Algorithm

- do
  - choosing[] = true;
  - number[] = max(number[0], number[1], ..., number [n - 1]) + 1;
  - choosing[] = false;
  - for (i = 0; j < n; j++) {
    - while (choosing[j]) ;
    - while ((number[i] ≺ number[j]) & (number[i] < number[j])) ;
  - )
  - critical section
  - number[] = 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 (Dijkstra)

- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations
  
  ```
  war(S):    while (S>0) do no-op;
  S--;
  signal(S):  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);
  }
  ```

Using Semaphores: Precedence

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore flag initialized to 0
- Code:
  
  ```
  $P_i$  $P_j$
  [ ] [ ]
  A  war(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

\[
P_0 : \text{wait}(S); \text{wait}(Q); \\
\text{signal}(S) \quad \text{signal}(Q); \\
\]

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

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 \( S \) as a Binary Semaphore

- **Data structures:**
  
  binary-semaphore S1, S2;
  int C;

- **Initialization:**
  
  \( S1 = 1 \quad \text{/* use this for mutex} \)
  \( S2 = 0 \quad \text{/* use this for blocking} \)
  \( C = \text{initial value of semaphore} \ S \quad \text{/* use this for the count} \)

Implementing \( S \)

- **wait operation**

\[
\text{wait}(S1);
\text{C}--;
\text{if (C < 0)} \{
\text{signal}(S1);
\text{wait}(S2);
\}
\text{signal}(S1);
\]

- **signal operation**

\[
\text{wait}(S1);
\text{C}++;
\text{if (C <= 0)} \{
\text{signal}(S1);
\text{signal}(S2); \quad \text{/* pass mutex to waiting process} \)
\text{else}
\text{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:**

\[
\text{Protect P and V code with test-and-set.} \\
P(S): \text{while (test-and-set(lock)) do no-op;} \\
\quad \text{code for P(S) ---} \\
\quad \text{lock = false}; \\
V(S): \text{while (test-and-set(lock)) do no-op;} \\
\quad \text{code for V(S) ---} \\
\quad \text{lock = false};
\]
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.

Mutual Exclusion with Test-and-Set

- Shared data:
  
  boolean lock = false;

- Process $P_i$
  
  do {
    while (TestAndSet(lock)) ;
    critical section
    lock = false;
    remainder section
  }

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);
    critical section
    lock = false;
    remainder section
  }

Synchronization Hardware

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

- Atomically swap two variables.
  
  void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
  }

Classical Problems of Synchronization

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

7.43

CIS 415

Bounded-Buffer Problem
(Producer Consumer Problem)

- Shared data
  - semaphore full, empty, mutex;
    - full mutex to protect the buffer slot
    - full for keep track of filled slots and block
    - consumer when 0 items in buffer
    - empty mutex 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

```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);
if (readcount == 0) /* the last reader */
  wait(mutex);
  readcount--;
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 available to any two neighboring 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: philosophers take turns to eat, all initiated to 1.

repeat forever: THINK

check left is free (chopstick 1 mod 5)
check right is free (chopstick 2 mod 5)

pick up left
pick up right

Philosophers Solution #2

SOLUTION 2: philosophers eat separately

check left is free (chopstick 1 mod 5)
check right is free (chopstick 2 mod 5)

pick up left
pick up right

Philosophers Solution #3

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

check left is free (chopstick 1 mod 5)
check right is free (chopstick 2 mod 5)

Philosophers Solution #4

SOLUTION 4: all philosophers to eat at the table at a time.

check left is free (chopstick 1 mod 5)
check right is free (chopstick 2 mod 5)

pick up left
pick up right

Eat

Replace both forks