Andrzej Bednarski, IDA
Linköpings universitet, 2005

Process Synchronization

- Introduce the critical-section problem
- Provide software and hardware solutions
- Introduce the concept of atomic transaction

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Chapter 4) allows at most \( n - 1 \) items in buffer at the same time. A solution, where all \( N \) buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable count, initialized to 0 and incremented each time a new item is added to the buffer

Race Condition:
- outcome of execution depends on a particular order

Bounded-Buffer: Producer

- Additional shared variable:
  - int count = 0;
- Producer process:
  ```c
  while (1) {
    /* produce an item in nextProduced */
    while (count == N) ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % N;
    count++;
  }
  ```

Bounded-Buffer: Consumer

- Consumer process:
  ```c
  while (1) {
    while (count == 0) ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % N;
    count--;
  }
  ```
- The statements:
  + count = count + 1;
  + count = count - 1;
  must be executed atomically.

Race Condition

- count++ could be implemented as
  ```c
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- count-- could be implemented as
  ```c
  register2 = count
  register2 = register2 - 1
  count = register2
  ```
- Consider this execution interleaving with "count = 5" initially:
  ```c
  S0: producer executes register1 = count (register1 = 5)
  S1: producer executes register1 = register1 + 1 (register1 = 6)
  S2: consumer executes register2 = count (register2 = 6)
  S3: consumer executes register2 = register2 - 1 (register2 = 5)
  S4: producer executes count = register1 (count = 6)
  S5: consumer executes count = register2 (count = 5)
  ```
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 (protocol).
- Structure of process $P_i$
  
  ```
  do {
    entry section
    critical section
    exit section
    reminder section
  } while (1);
  ```

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.

Peterson's Solution

- Two processes solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn = 0;
  - boolean flag[2] = {false, false};
- The variable turn indicates whose turn it is to enter the critical section (initialized to 0 or 1).
- The flag array is used to indicate if a process is ready to enter the critical section. flag[$i$] = true implies that process $P_i$ is ready. (Flag vector is initialized to false)

Algorithm for Process $P_i$

```
  do {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);  
    CRITICAL SECTION
    flag[i] = false;
    REMAINDER SECTION
  } while(1);
```

1) Preserves mutual exclusion
2) Satisfies progress requirement
3) Meets bounded-waiting

Synchronization Hardware

- Idea: avoid/disable interrupts while common variables are changed
- Not applicable in multi-processor environments
- Modern machines provide special atomic hardware instructions (Atomic = non-interruptable)
- Either test memory word and set value
  ```
  boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;
    return rv;
  }
  ```
- Or swap contents of two memory words
  ```
  void Swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
  }
  ```

Mutual Exclusion with TestAndSet

- Shared data:
  ```
  boolean lock = false;
  ```
- Process $P_i$
  ```
  do { 
    while (TestAndSet(lock)); /* busy wait */
  } while(1);
  ```
- critical section
  ```
  lock = false; /* release lock */
  remainder section
  ```


Mutual Exclusion with Swap

- Shared boolean variable lock initialized to FALSE;
  Each process has a local boolean variable key.
- Solution:
  
  ```
  do {
    key = true;
    while(key == true)
      Swap(&lock, &key);
    /* critical section */
    lock = false;
    /* remainder section */
  } while(1);
  ```

TestAndSet/Swap: Pros and Cons

+ Applicable to any number of processes
  on either uni- or multi-processor machine
+ Simple and therefore easy to verify
+ Can support multiple critical sections
  (each critical section can be defined by its own variable)
- Busy waiting is employed (spinning)
- Starvation is possible: when a process leaves a critical section and
  more than one process is waiting, the selection of a waiting process
  is arbitrary; thus, some process could indefinitely be denied access.
- Deadlock is possible:
  i) P1 executes and enters its critical section
  ii) P2 interrupts P1 due to high priority, but cannot enter its critical section
  iii) P1 never dispatches again, since it is low priority job

Semaphores

- Synchronization tool that does not require busy waiting.
- Synchronization operations are atomic, i.e. indivisible
  + Operation P = wait (Dutch for proberen (test))
  + Operation V = signal (Dutch for verhogen (increment))
- Applicability:
  + N-process critical section problems
  + Synchronization problems, e.g., precedence constraints
- Semaphore S – integer variable

  ```
  wait (S) {
    while (S <= 0)
      ; /* no-op */
    S –– ;
  }

  signal (S) {
    S ++;
  }
  ```

Example: Critical Section of n Processes

- Shared variable:
  semaphore mutex = 1;
- Process Pi
  ```
  do {
    wait(mutex);
    critical section
    signal(mutex);
    reminder section
  } while (1);
  ```

Semaphore as General Synchronization Tool

- 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
    + Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

Implementing S using Binary Semaphore

- Data structures:
  binary-semaphore S1, S2;
  int C;
- Initialization:
  ```
  S1 = 1;
  S2 = 0;
  C = initial value of semaphore S
  ```
Implementing S (Cont.)

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

- **signal operation**
  ```
  wait(S1);
  C++;
  if (C <= 0)
    signal(S2);
  else
    signal(S1);
  ```

Semaphore: Pros and Cons

- **Cons:**
  - "Spin-lock" method -> requires busy waiting

- **Pros:**
  - Spin-lock has no context switching when waiting
  - Applicable in multi-processor environments
  - Solution to busy-waiting: block yourself

Semaphore Implementation (without busy waiting)

- Define a semaphore as a struct
  ```
  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 Implementation (without busy waiting) (Cont.)

- Semaphore operations now defined as:
  ```
  void wait(semaphore S) {
    S.value--;
    if (S.value < 0) {
      add this process to S.L;
      block();
    }
  }

  void signal(semaphore 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 \( P_1 \) only after A executed in \( P_0 \)
- Use semaphore flag initialized to 0
- Code:
  ```
  \( P_0 \) \( P_1 \)
  \[ \]
  A \( \text{wait(flag)} \)
  \( \text{signal(flag)} \) \( B \)
  ```

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 \hspace{1cm} P_1
    \text{wait(S)}; \hspace{1cm} \text{wait(Q)};
    \text{wait(Q)}; \hspace{1cm} \text{wait(S)};
    \text{signal(S)}; \hspace{1cm} \text{signal(Q)};
    \text{signal(Q)}; \hspace{1cm} \text{signal(S)};
    ```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
  (e.g., LIFO order)
Classical Problems of Synchronization

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

Bounded-Buffer Problem

- \( N \) buffers, each can hold one item
- Semaphore \( \text{mutex} \) initialized to the value 1
- Semaphore \( \text{full} \) initialized to the value 0
- Semaphore \( \text{empty} \) initialized to the value \( N \)
- Shared data

\[
\text{typedef struct} \\
\quad \text{item} \\
\quad \text{buffer}[N]; \\
\quad \text{semaphore mutex = 1, full = 0, empty = N;} \\
\]

Bounded-Buffer Problem (Cont.)

Producer process

\[
\text{do} \\
\quad \text{produce an item in nextp} \\
\quad \text{wait (empty);} \\
\quad \text{wait (mutex);} \\
\quad \text{add nextp to buffer} \\
\quad \text{signal (mutex);} \\
\quad \text{signal (full);} \\
\quad \text{while(1);} \\
\]

Consumer process

\[
\text{do} \\
\quad \text{wait (full);} \\
\quad \text{wait (mutex);} \\
\quad \text{remove an item from buffer to nextc} \\
\quad \text{signal (mutex);} \\
\quad \text{signal (empty);} \\
\quad \text{consume the item in nextc} \\
\quad \text{while(1);} \\
\]

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore \( \text{mutex} \) initialized to 1
  - Semaphore \( \text{wrt} \) initialized to 1
  - Integer \( \text{readcount} \) initialized to 0

Readers-Writers Problem (Cont.)

Reader process

\[
\text{do} \\
\quad \text{wait (mutex);} \\
\quad \text{readcount++;} \\
\quad \text{if (readcount == 1) \text{wait (wrt);}} \\
\quad \text{signal (wrt);} \\
\quad \text{reading is performed} \\
\quad \text{wait (mutex);} \\
\quad \text{readcount--;} \\
\quad \text{if (readcount == 0) \text{signal (wrt);}} \\
\quad \text{signal (mutex);} \\
\quad \text{while(1);} \\
\]

Writer process

\[
\text{do} \\
\quad \text{wait (wrt);} \\
\quad \text{writing is performed} \\
\quad \text{signal (wrt);} \\
\quad \text{while(1);} \\
\]

Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore \( \text{chopstick[5]} \) initialized to 1
Dining-Philosophers Problem (Cont.)

- Philosopher i:
  - do {
    - wait(chopstick[i]);
    - wait(chopstick[(i+1) % 5]);
    - eat
    - signal(chopstick[i]);
    - signal(chopstick[(i+1) % 5]);
    - think
  - } while (1);

- Guarantees that neighbors are not eating simultaneously
- Does not guarantee avoidance of deadlocks

Dining-Philosophers Problem
Three Deadlock-Free Solutions

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her/his chopsticks only if both chopsticks are available (both in the same critical section)
- Odd philosophers pick up left chopstick first;
  even philosophers pick up right chopstick first
  (asymmetric solution)

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) .... wait (mutex)
  - wait (mutex) .... wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- Only one process may be active within the monitor at a time

```
monitor monitor-name {
  shared variable declarations
  procedure body P1 (...){
    ...
  }
  procedure body P2 (...){
    ...
  }
  procedure body Pn (...){
    ...
  }
  {
    initialization code
  }
}
```

Condition Variables

- condition x, y
- Two operations on a condition variable:
  - x.wait(): a process that invokes the operation is suspended.
  - x.signal(): resumes one of processes (if any) that invoked x.wait()
Solution to Dining Philosophers

```c
monitor DP {
    enum { THINKING, HUNGRY, EATING) state [5];
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }
    void putdown (int i) {
        state[i] = THINKING;
        /* test left and right neighbors */
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    ...
}
```

Solution to Dining Philosophers (Cont.)

```c
... void test (int i) {
    if (state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
    initialization_code() {
        for(int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Atomic Transactions

- **Transactions** - several operations grouped as one logical atomic/indivisible step of execution
  - Commit or abort (roll back)
- **Database technique**
  - Could provide more flexible and powerful techniques at OS level
  - "Transaction Processing Systems"
- **Log-based Recovery**
- **Checkpoints**
  - Log-recovery is performed since the last checkpoint (i.e., not the entire log)

Concurrent Atomic Transactions

- **Serializability**
- **Locking protocol** - ensures serializability
  - 2PL - Two phase locking
    - Growing phase: a transaction only obtains locks
    - Shrinking phase: a transaction only releases locks
- **Timestamp-based protocols**

Recommended Reading and Exercises

- **Reading**
  - Chapter 6 [SGG7]
    - Section 6.8 (optional)
  - Chapter 7 (6th edition)
- **Exercises**
  - All
  - Project (Producer-Consumer Problem)