COP 4610

Operating System Principles

Synchronization

1

Overview

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

2

Objectives

- To introduce the **critical-section problem**, whose solutions can be used to ensure the **consistency** of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in **data inconsistency**
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

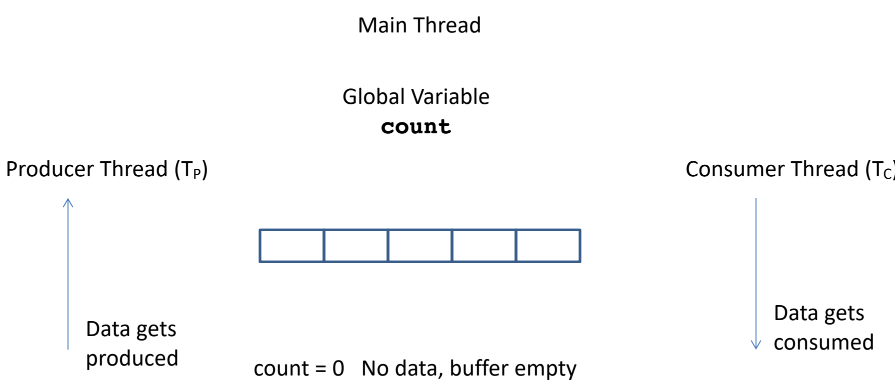
Example

Suppose that we wanted to provide a solution to the consumer-producer problem.

We can do so by having an integer **`count`** that keeps track of the slots taken. As we add things, **count** grows. As we consume things, **count** shrinks.

5

Illustration



count = 0 No data, buffer empty

count = 5 Full of data, no room

6

Code

```
int count = 0;
int in = 0;
int out = 0;

int main (int argc, char * argv[])
{
  pthread_t tC, tP;

  pthread_create(&tP, NULL, thread_Producer, NULL, NULL);
  pthread_create(&tC, NULL, thread_Consumer, NULL, NULL);

  /* Hang around for them to be done (never) */
  pthread_join(tP);
  pthread_join(tC);

  return 1;
}
```

Address of functions

We get two threads that will be executing in addition to our main thread

Producer

```
void thread_Producer (void * pData)
{
   while (1)
   {
       /* produce an item in next produced */

       while (count == BUFFER SIZE) ;
              /* do nothing */

       /* Space in the buffer! */
       buffer[in] = next_produced;
       in = (in+1)%BUFFER_SIZE;
       count++;
   }
}
```

If the buffer is full, hold up

Consumer

```
void thread_Consumer (void * pData)
{
   while (1)
   {
      while (count == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out+1)%BUFFER_SIZE;
      count--;
      /* consume the item in next consumed */
    }
}
```

If the buffer is empty, hold up

Race Condition

- count++:
  register1 = count
  register1 = register1 + 1
  count = register1
- count--:
  register1 = count
  register1 = register1 - 1
  count = register1

Race Condition

- Assume count=5

  Step 1: Producer: register1 = count (register1 = ?)

  Step 2: Producer: register1 = register1 + 1 (?)

  Step 3: Consumer: register2 = count (register2 = ?)

  Step 4: Consumer: register2 = register2 – 1 (?)

  Step 5: Producer: count = register1 (count = ?)

  Step 6: Consumer: count = register2 (count = ?)

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section

- **Critical section: section in code where race conditions can occur!**
- **Critical section problem** is to design protocol to solve this

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process p_i is

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```


13

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

14

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** instructions are **atomic**; that is, they cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- 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!

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- **Uniprocessors** – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value (TestAndSet())
  - Or swap contents of two memory words (Swap())

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
          critical section
    release lock
          remainder section
} while (TRUE);
```

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
     {
          boolean rv = *target;
          *target = TRUE;
          return rv:
     }
```


Solution using test_and_set()

- Shared boolean variable **lock**, initialized to FALSE
- Solution:

```
do {
   while (TestAndSet(&lock))
      ; /* do nothing */
   /* critical section */
   lock = FALSE;
   /* remainder section */
} while (TRUE);
```

Swap Instruction

- Definition:

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


Solution using 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 (TRUE);
```

Bounded-waiting Mutual Exclusion with TestAndSet

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```


Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers!
- OS designers build software tools to solve critical section problem
- Simplest is **mutex lock**
- Protect critical regions with it by first `acquire()` a lock then `release()` it
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}

do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```


Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
     while (S <= 0)
         ; // busy wait
     S--;
}
signal (S) {
     S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```


Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time
  - Internally implemented using atomic instructions, disabled interrupts, …
- Implementation uses **busy waiting**:
  - Ok if waiting time is rare or short (e.g., critical section is rarely occupied and/or short)
  - If applications spend a lot of time in critical sections, this is not a good solution

Semaphore Implementation with no Busy Waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (identifying waiting process)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process from S->list;
                wakeup(P);
        }
}
```

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	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`.`	`.`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
   /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
   /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
   wait(full);
   wait(mutex);
      ...
   /* remove an item from buffer to next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
   /* consume the item in next consumed */
      ...
} while (true);
```


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 **`wrt`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {

   wait(wrt);
      ...
   /* writing is performed */
      ...
   signal(wrt);
} while (true);
```

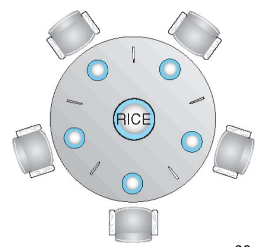

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
        wait(wrt);
    signal(mutex);
      ...
   /* reading is performed */
      ...
   wait(mutex);
   read_count--;
   if (read_count == 0)
       signal(wrt);
    signal(mutex);
} while (true);
```

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



39

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              // eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

             // think

} while (TRUE);
```

- What is the problem with this algorithm?

40

Monitors

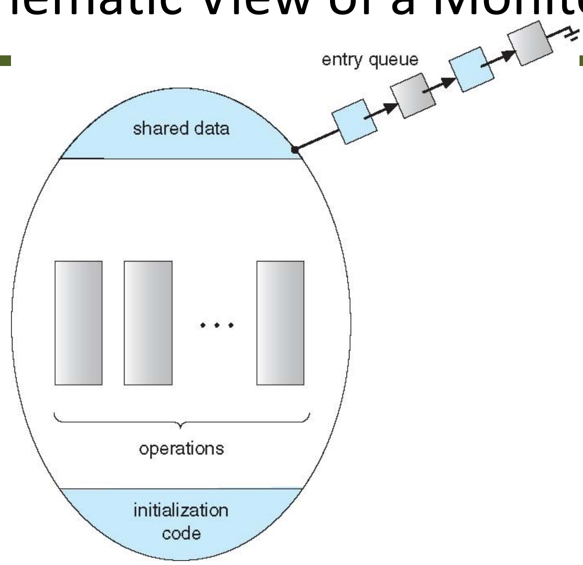
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }

   procedure Pn (…) {……}

   Initialization code (…) { … }
   }
}
```

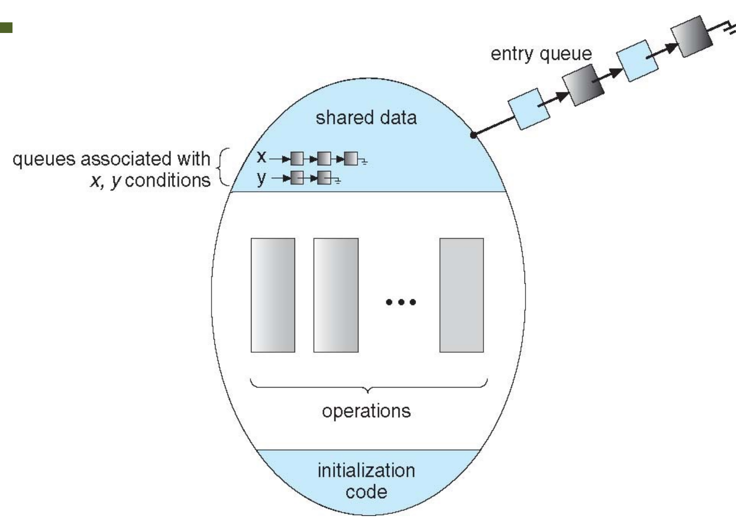

Schematic View of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DiningPhilosophers
 {
    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.)

```
    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;
    }
}
```

Dining Philosophers

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

  DiningPhilosophers.pickup (i);

  EAT

  DiningPhilosophers.putdown (i);

- No deadlock, but starvation is possible!