# *Process Synchronization*

**CS 502**
**Spring 99**
**WPI MetroWest/Southboro Campus**

## *Process Synchronization*

- Background
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in NT
- Atomic Transactions

## *Background*

- Concurrent access to shared data may result in data inconsistency.
  - Examples?
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Recall the shared-memory solution to the bounded-buffer problem, which allowed at most (*n* - 1) items in the shared 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 *counter*, initialized to 0 and incremented each time a new item is added to the buffer.



## *Bounded-Buffer*

- Shared Data

```
const int n = MAX_BUF;
typedef struct {…} item_t;
int in, out;
int counter;
item_t * buffer[n];
in, out, counter = 0;
```

- Producer process

```
item_t * nextp;

do {
  …
  // produce an item in nextp
  …
  while (counter == n) ;
  buffer[in] = nextp;
  in = (in + 1) % n;
  counter = counter + 1;
} while (TRUE);
```

## *Bounded-Buffer (Cont.)*

- Consumer process

```
item_t * nextc;

do {
  while (counter == 0) ;
  nextc = buffer[out];
  out = (out + 1) % n;
  counter = counter - 1;
  …
  // consume the item in nextc
  …
} while (TRUE);
```

- The statements:

```
counter = counter + 1;
counter = counter - 1;
```

must be executed atomically.

4

## *The Critical-Section Problem*

- *n* processes all competing to use some shared data.
- Each process has a code segment, called a 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.
- General structure of a process $P_i$:

**do**

*entry section*

critical section

*exit section*

remainder section

**while** (**true**)

5

## *Properties of a solution*

1. **Mutual Exclusion**. If process Pi 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.

- Assumptions:
  - Each process executes at a nonzero speed.
  - No assumption concerning the relative speed of the n processes.

6

## *Initial Attempts to Solve Problem*

- Only two processes, $P_0$ and $P_1$
- Processes may share some common variables to synchronize their actions.
- Recall general structure of process $P_i$ (others are $P_j$)

**do**

*entry section*

critical section

*exit section*

remainder section

**while** (**true**)

7

## *Algorithm 1*

- Shared variables
  - `int turn; /* 0..1 */`
    initially `turn = 0.`
  - `turn = I` ⇒ $P_i$ can enter its critical section
- Process $P_i$:

**do**

```
while (turn != i) ;
```

critical section

```
turn = j
```

remainder section

**while** (**true**)

- What properties does this solution satisfy?
  - Satisfies mutual exclusion
  - Does not satisfy progress

8

## *Algorithm 2a*

- Shared variables
  - `boolean flag[2];`
    initially `flag[0] = flag[1] = false.`
  - `flag[i] = true` ⇒ $P_i$ *ready* to enter its critical section
- Process $P_i$:

**do**

```
flag[i] = true;
while (flag[j] == true) ;
```

critical section

```
flag[i] = false
```

remainder section

**while** (**true**)

- What properties does this solution satisfy?
  - Satisfies mutual exclusion
  - Does not satisfy progress

9

## *Algorithm 2b*

- Process $P_i$:

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

- What properties does this solution satisfy?
  - Satisfies progress
  - Does not satisfy mutual exclusion

10

## *Algorithm 3*

- Combine shared variables of algorithms 1 and 2
- Process $P_i$:

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

- Meets all three requirements; solves the critical-section problem for two processes.

11

## *Bakery Algorithm (Lamport)*

- 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, then if i < j, then $P_i$ is served first; else $P_j$ is served first.
  - The number selection algorithm guarantees that numbers are generated in increasing order of enumeration; I.e.

    1, 2, 3, 3, 3, 4, 4, 5, 6
  - Notation: < is defined as lexicographical order (ticket #, pid)
    - (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 ≥ aid for i = 0, …, n-1
  - Shared data
    - `boolean choosing[n];`
    - `int number[n];`
    - initialized to **`false`** and 0, respectively

12

## *Bakery Algorithm (Cont.)*

**do**

```
choosing[i] = true;
number[i] = max(number[0], …, number[n-1]) + 1;
choosing[i] = false;
for (j = 0; j < n; j++) {
  while (choosing[j] == true) ;
  while ((number[j] != 0) &&
         ((number[j], j) < (number[i], i))) ;
}
```

critical section

```
number[i] = 0;
```

remainder section

**while** (**true**)

13

## *Synchronization Hardware*

- Test and modify the content of a word atomically.

```
boolean TestAndSet ( boolean * target )
{
  boolean oldval;

  oldval = *target;
  *target = true;

  return oldval;
}
```

14

## *Mutual Exclusion with TestAndSet*

- Shared data
  - `boolean lock;`
  - initially **`false`**
- Process $P_i$:

**do**

```
while (TestAndSet(&lock) == true) ;
```

critical section

```
lock = false;
```

remainder section

**while** (**true**)

15

## *Semaphore*

- Synchronization tool that does not require busy waiting
- Semaphore S - integer variable with additional semantics
- Can only be accessed via two atomic operations:
  - wait(S):
    - while (S <= 0) ;
    - S = S - 1;
  - signal(S);
    - S = S + 1;

16

## *Example: Critical Section for* n *Processes*

Shared data

```
semaphore mutex;
```

initially `mutex = 1;`

Process $P_i$:

```
do
    wait ( mutex ) ;
        critical section
    signal ( mutex );
        remainder section
while (true)
```

17

## *Semaphore Implementation*

- Define a semaphore as a record

```
typedef struct {
  int value;
  processList_t L;
} semaphore, *semptr;
```

- Assume help from the OS with two simple operations:
  - *block* suspends the process that invokes it.
  - *wakeup*(*P*) resumes the execution of a blocked process *P*.

18

## *Implementation (Cont.)*

- Semaphore operations now defined as:
  - wait(semptr S):

```
S->value = S->value - 1;
if (S->value < 0) {
        add this process to S->L;
        block;
}
```

  - signal(semptr S):

```
S->value = S->value + 1;
if (S->value <= 0) {
        remove a process P from S->L;
        wakeup(P)
}
```

19

## *Semaphore for General Synchronization*

- Desire to execute B in $P_j$ only after A executed in Pi
- Use semaphore flag initialized to 0
- Code:

| $P_i$ | $P_j$ |
|---|---|
| . | . |
| . | . |
| A | . |
| *signal*(*flag*) | *wait*(*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

| $P_0$ | $P_1$ |
|---|---|
| *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.

## *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
- Data Structures:
  - binarySemaphore S1;
  - binarySemaphore S2;
  - binarySemaphore S3;
  - int C;
- Initialization:
  - S1->value = S3->value = 1
  - S2->value = 0
  - C = initial value of semaphore

22

## *Implementing a Binary Semaphore*

- *wait* operation

```
wait(S3);
wait(S1);
C = C-1;
if (C < 0) {
  signal(S1);
  wait(S2);
} else {
  signal(S1);
}
signal(S3);
```

- *signal* operation

```
wait(S1);
C = C + 1;
if (C <= 0) {
  signal(S2);
}
signal(S1);
```

23

## *Classical Problems of Synchronization*

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

24

## *Bounded-Buffer Problem*

- Shared Data:
  - typedef struct { … } item_t;
  - item_t * buffer[N];
  - sem full, empty, mutex;
  - item_t * nextp, * nextc;
  - full->value = 0; empty->value = N; mutex->value = 1;

25

## Producer and Consumer

**Producer**

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

**Consumer**

```
do {
    wait(full);
    wait(mutex);
    …
    remove an item from buffer to
    nextc
    …
    signal(mutex);
    signal(empty);
    …
    produce an item in nextp;
    …
} while (true);
```

26

## Readers-Writers Problem

- Shared Data
  - sem mutex (= 1); sem wrt (= 1);
  - int readcount (=0);

**Writer Process**

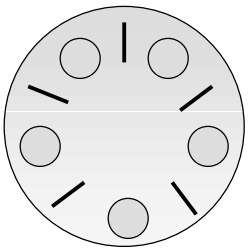
```
wait(wrt);
…
writing is performed
…
signal(wrt);
```

**Reader Process**

```
wait(mutex);
        readcount = readcount + 1;
        if (readcount == 1) {
                wait(wrt);
        }
signal(mutex);
…
reading is performed
…
wait(mutex);
        readcount = readcount - 1;
        if (readcount == 0) {
                signal(wrt);
        }
signal(mutex);
```

27

## *Dining-Philosophers Problem*



**sem chopstick[5]; (all = 1)**

28

## *Dining-Philosophers (Cont.)*

```
do {
        wait(chopstick[i]);
        wait(chopstick[(i + 1) % 5]);
        …
        eat
        …
        signal(chopstick[i]);
signal(chopstick[(i + 1) % 5]);
        …
        think
        …
} while (true);
```

29

## *Monitors*

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
class MonitorADT {
   private int sharedInt;
   private boolean writeable = true;

   public synchronized void setSharedInt ( int val )
   {
      while ( !writeable ) {
         wait();
      }

      sharedInt = val;
      writeable = false;
      notify();
   }

   public synchronized int getSharedInt ( void )
   {
      while ( writeable ) {
         wait();
      }

      writeable = true;
      notify();
      return sharedInt;
   }
```

30

## *Monitors (Cont.)*

- Generalization of wait/notify within a monitor — condition variables.
  - **condition** x, y;
- Condition variables can only be used with the operations wait and signal.
  - The operation **x.wait** means that the thread invoking this operation is suspended until another process invokes **x.signal**
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

31

## *Dining Philosophers Example*

```
class DiningPhilosophers {
    typedef enum {Thinking, Hungry, Eating} PhilosoperState;
    private PhilosopherState state[5];
    private condition self[5];

    public synchronized void pickup ( int i )
    {
        state[i] = Hungry;
        test (i);
        if (state[i] != Eating) {
            self[i].wait();
        }
    }

    public synchronized void putdown ( int i )
    {
        state[i] = Thinking;
        test ((i+4) % 5);
        test ((i+1) % 5);
    }

    private void test ( int k )
    {
        if ((state[(k+4) % 5] != Eating) &&
            (state[k] == Hungry) &&
            (state[(k+1) % 5) != Eating)) {
            state[k] = Eating;
            self[k].signal();
        }
    }
}
```

32

## *Monitor Implementation Using Semaphores*

- Variables

```
semaphore mutex; (init = 1)
semaphore next; (init = 0)
int nextCount; (init = 0)
```

- Replace external procedure *F* with:

```
wait(mutex);
    …
    body of F
    …
if (nextCount > 0) {
    signal(next);
} else {
    signal(mutex);
}
```

- Mutual Exclusion within a monitor is ensured.

33

## *Implementing Condition Variables*

- For each condition variable x, define:

```
semaphore xSemaphore; (init = 0)
int xCount; (init = 0)
```

x.wait

```
xCount = xCount + 1;
if (nextCount > 0) {
    signal(next);
} else {
    signal(mutex);
}
wait(xSemaphore);
xCount = xCount - 1;
```

x.signal

```
if (xCount > 0) {
    nextCount = nextCount + 1;
    signal(xSemaphore);
    wait(next);
    nextCount = nextCount - 1;
}
```

34

## *Monitor Usage*

- User processes must always make their calls on the monitor in a correct sequence.
- Still must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

35

## *Windows NT Operating System*

- Implements a variety of synchronization primitives to support multitasking, multithreading, and multiprocessing.
- Different set of mechanisms within the Operating System and for User Processes.
- Objects of different scope are appropriate for different levels of sharing.

36