



## Chapter 7: Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000




Operating System Concepts 7.1 Silberschatz, Galvin and Gagne ©2002




## 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 *counter*, initialized to 0 and incremented each time a new item is added to the buffer



Operating System Concepts 7.2 Silberschatz, Galvin and Gagne ©2002




## Bounded-Buffer

Shared data


- ```

#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;

```



Operating System Concepts 7.3 Silberschatz, Galvin and Gagne ©2002



## Bounded-Buffer


Producer process

- ```


item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

```



Operating System Concepts 7.4 Silberschatz, Galvin and Gagne ©2002



## Bounded-Buffer


Consumer process

- ```


item nextConsumed;

while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}

```



Operating System Concepts 7.5 Silberschatz, Galvin and Gagne ©2002




## Bounded Buffer

The statements


- ```

counter++;
counter--;

```
- must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption.



Operating System Concepts 7.6 Silberschatz, Galvin and Gagne ©2002



## Bounded Buffer


- • The statement `count++` may be implemented in machine language as:
 

```


register1 = counter
register1 = register1 + 1
counter = register1
      
```
- • The statement `count--` may be implemented as:
 

```

register2 = counter
register2 = register2 - 1
counter = register2
      
```




Operating System Concepts 7.7 Silberschatz, Galvin and Gagne ©2002




## Bounded Buffer

- • 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.



Operating System Concepts 7.8 Silberschatz, Galvin and Gagne ©2002




## Bounded Buffer


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




Operating System Concepts 7.9 Silberschatz, Galvin and Gagne ©2002




## 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.
- • To prevent race conditions, concurrent processes must be **synchronized**.




Operating System Concepts 7.10 Silberschatz, Galvin and Gagne ©2002




## 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.




Operating System Concepts 7.11 Silberschatz, Galvin and Gagne ©2002



## 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.



Operating System Concepts 7.12 Silberschatz, Galvin and Gagne ©2002

## Initial Attempts to Solve Problem

- Only 2 processes,  $P_0$  and  $P_1$
- 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.

Operating System Concepts 7.13 Silberschatz, Galvin and Gagne ©2002

## Algorithm 1

- Shared variables:
 

```
int turn;
initially turn = 0
turn - i ⇒ Pi 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

Operating System Concepts 7.14 Silberschatz, Galvin and Gagne ©2002

## Algorithm 2

- Shared variables
 

```
boolean flag[2];
initially flag [0] = flag [1] = false.
flag [i] = true ⇒ Pi 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.

Operating System Concepts 7.15 Silberschatz, Galvin and Gagne ©2002

## Algorithm 3

- 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;
  remainder section
} while (1);
```

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

Operating System Concepts 7.16 Silberschatz, Galvin and Gagne ©2002

## Bakery Algorithm

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., 1,2,3,3,3,4,5...

Operating System Concepts 7.17 Silberschatz, Galvin and Gagne ©2002

## Bakery Algorithm

- Notation  $\leq$  lexicographical order (ticket #, process id #)
 

```
(a,b) <= (c,d) if a < c or if a = c and b < d
max (a0, ..., an-1) is a number, k, such that k ≥ ai for i = 0, ..., n - 1
```
- Shared data
 

```
boolean choosing[n];
int number[n];
```

Data structures are initialized to false and 0 respectively

Operating System Concepts 7.18 Silberschatz, Galvin and Gagne ©2002

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

```

Operating System Concepts 7.19 Silberschatz, Galvin and Gagne ©2002

## Synchronization Hardware

Test and modify the content of a word atomically

```

boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target = true;

    return rv;
}

```

Operating System Concepts 7.20 Silberschatz, Galvin and Gagne ©2002

## Mutual Exclusion with Test-and-Set

Shared data:

```

boolean lock = false;

```

Process  $P_i$

```

do {
    while (TestAndSet(lock));
    critical section
    lock = false;
    remainder section
}

```

Operating System Concepts 7.21 Silberschatz, Galvin and Gagne ©2002

## Synchronization Hardware

Atomically swap two variables.

```

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

```

Operating System Concepts 7.22 Silberschatz, Galvin and Gagne ©2002

## 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
}

```

Operating System Concepts 7.23 Silberschatz, Galvin and Gagne ©2002

## Semaphores

Synchronization tool that does not require busy waiting.

Semaphore  $S$  integer variable  
can only be accessed via two indivisible (atomic) operations

```

wait (S):
    while S ≤ 0 do no-op;
    S--;

signal (S):
    S++;

```

Operating System Concepts 7.24 Silberschatz, Galvin and Gagne ©2002

## Critical Section of $n$ Processes

- Shared data:
  - semaphore mutex;** //initially  $mutex = 1$
- Process  $P_i$ :
 

```
do {
  wait(mutex);
  critical section
  signal(mutex);
  remainder section
} while (1);
```

Operating System Concepts 7.25 Silberschatz, Galvin and Gagne ©2002

## Semaphore Implementation

- 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  $P$ .

Operating System Concepts 7.26 Silberschatz, Galvin and Gagne ©2002

## Implementation

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);
}
```

Operating System Concepts 7.27 Silberschatz, Galvin and Gagne ©2002

## Semaphore as a General Synchronization Tool

- Execute  $B$  in  $P_j$  only after  $A$  executed in  $P_i$
- Use semaphore  $flag$  initialized to 0
- Code:
 

$P_i$	$P_j$
⋮	⋮
$A$	$wait(flag)$
$signal(flag)$	$B$

Operating System Concepts 7.28 Silberschatz, Galvin and Gagne ©2002

## 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.

Operating System Concepts 7.29 Silberschatz, Galvin and Gagne ©2002

## Two 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.

Operating System Concepts 7.30 Silberschatz, Galvin and Gagne ©2002

## Implementing S as a Binary Semaphore

Data structures:

```

- .
    binary-semaphore S1, S2;
    int C;

```

Initialization:

```

    S1 = 1
    S2 = 0
    C = initial value of semaphore S

```

Operating System Concepts 7.31 Silberschatz, Galvin and Gagne ©2002

## Implementing S

wait operation

```

wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);

```

signal operation

```

wait(S1);
C++;
if (C <= 0)
    signal(S2);
else
    signal(S1);

```

Operating System Concepts 7.32 Silberschatz, Galvin and Gagne ©2002

## Classical Problems of Synchronization

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Operating System Concepts 7.33 Silberschatz, Galvin and Gagne ©2002

## Bounded-Buffer Problem

Shared data

```

semaphore full, empty, mutex;

```

Initially:

```

full = 0, empty = n, mutex = 1

```

Operating System Concepts 7.34 Silberschatz, Galvin and Gagne ©2002

## Bounded-Buffer Problem Producer Process

```

. .
do {
    produce an item in nextp

    wait(empty);
    wait(mutex);

    add nextp to buffer

    signal(mutex);
    signal(full);
} while (1);

```

Operating System Concepts 7.35 Silberschatz, Galvin and Gagne ©2002

## Bounded-Buffer Problem Consumer Process

```

. .
do {
    wait(full)
    wait(mutex);

    remove an item from buffer to nextc

    signal(mutex);
    signal(empty);

    consume the item in nextc
} while (1);

```

Operating System Concepts 7.36 Silberschatz, Galvin and Gagne ©2002

## Readers-Writers Problem

Shared data

- •

```

semaphore mutex, wrt;

Initially
mutex = 1, wrt = 1, readcount = 0
  
```

Operating System Concepts 7.37 Silberschatz, Galvin and Gagne ©2002

## Readers-Writers Problem Writer Process

- •

```

wait(wrt);

writing is performed

signal(wrt);
  
```

Operating System Concepts 7.38 Silberschatz, Galvin and Gagne ©2002

## Readers-Writers Problem Reader Process

- •

```

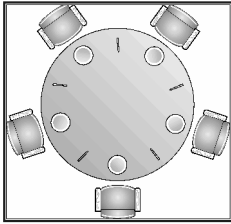
wait(mutex);
readcount++;
if (readcount == 1)
  wait(rt);
signal(mutex);

reading is performed

wait(mutex);
readcount--;
if (readcount == 0)
  signal(wrt);
signal(mutex);
  
```

Operating System Concepts 7.39 Silberschatz, Galvin and Gagne ©2002

## Dining-Philosophers Problem



Shared data

```

semaphore chopstick[5];
Initially all values are 1
  
```

Operating System Concepts 7.40 Silberschatz, Galvin and Gagne ©2002

## Dining-Philosophers Problem

Philosopher *i*:

- •

```

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

  eat

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

  think
 } while (1);
  
```

Operating System Concepts 7.41 Silberschatz, Galvin and Gagne ©2002

## Critical Regions

- •

High-level synchronization construct

A shared variable *v* of type *T*, is declared as:

```

v: shared T
  
```

Variable *v* accessed only inside statement

```

region v when B do S
  
```

where *B* is a boolean expression.



While statement *S* is being executed, no other process can access variable *v*.

Operating System Concepts 7.42 Silberschatz, Galvin and Gagne ©2002

## Critical Regions

- Regions referring to the same shared variable exclude each other in time.

When a process tries to execute the region statement, the Boolean expression  $B$  is evaluated. If  $B$  is true, statement  $S$  is executed. If it is false, the process is delayed until  $B$  becomes true and no other process is in the region associated with  $v$ .

Operating System Concepts 7.43 Silberschatz, Galvin and Gagne ©2002



## Example Bounded Buffer

Shared data:

```

struct buffer {
    int pool[n];
    int count, in, out;
}

```

Operating System Concepts 7.44 Silberschatz, Galvin and Gagne ©2002



## Bounded Buffer Producer Process

Producer process inserts **nextp** into the shared buffer

```

region buffer when( count < n) {
    pool[in] = nextp;
    in := (in+1) % n;
    count++;
}

```

Operating System Concepts 7.45 Silberschatz, Galvin and Gagne ©2002



## Bounded Buffer Consumer Process

Consumer process removes an item from the shared buffer and puts it in **nextc**

```

region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}

```



Operating System Concepts 7.46 Silberschatz, Galvin and Gagne ©2002

## Implementation region $x$ when $B$ do $S$

- Associate with the shared variable  $x$ , the following variables:
  - semaphore mutex, first-delay, second-delay;**
  - int first-count, second-count;**

Mutually exclusive access to the critical section is provided by **mutex**.

If a process cannot enter the critical section because the Boolean expression  $B$  is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate  $B$ .



Operating System Concepts 7.47 Silberschatz, Galvin and Gagne ©2002

## Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.

The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.

For an arbitrary queuing discipline, a more complicated implementation is required.

Operating System Concepts 7.48 Silberschatz, Galvin and Gagne ©2002



## Monitors

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

```

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

```

Operating System Concepts 7.49 Silberschatz, Galvin and Gagne ©2002

## Monitors

To allow a process to wait within the monitor, a **condition** variable must be declared, as

```

condition x, y;

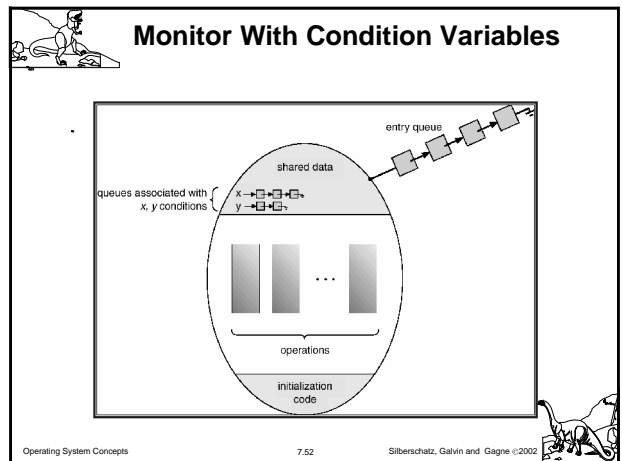
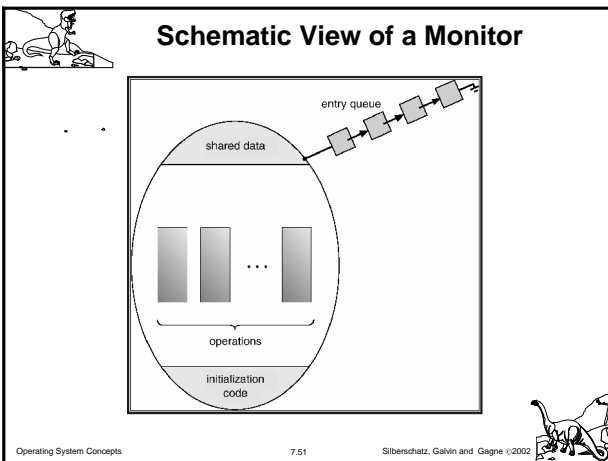
```

Condition variable can only be used with the operations **wait** and **signal**.

The operation **x.wait()** means that the process 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.

Operating System Concepts 7.50 Silberschatz, Galvin and Gagne ©2002



## Dining Philosophers Example

```

monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i) // following slides
    void putdown(int i) // following slides
    void test(int i) // following slides
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}

```

Operating System Concepts 7.53 Silberschatz, Galvin and Gagne ©2002

## Dining Philosophers


```

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);
}

```

Operating System Concepts 7.54 Silberschatz, Galvin and Gagne ©2002





## Dining Philosophers

```

void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}

```

Operating System Concepts 7.55 Silberschatz, Galvin and Gagne ©2002

## Monitor Implementation Using Semaphores

Variables

```

semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;

```

Each external procedure *F* will be replaced by

```

wait(mutex);



body of F;

if (next-count > 0)
    signal(next)
else
    signal(mutex);

```

Mutual exclusion within a monitor is ensured.

Operating System Concepts 7.56 Silberschatz, Galvin and Gagne ©2002

## Monitor Implementation

For each condition variable *x*, we have:

```

semaphore x-sem; // (initially = 0)
int x-count = 0;

```



The operation *x.wait* can be implemented as:

```

x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;

```

Operating System Concepts 7.57 Silberschatz, Galvin and Gagne ©2002

## Monitor Implementation



The operation *x.signal* can be implemented as:

```

if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}

```

Operating System Concepts 7.58 Silberschatz, Galvin and Gagne ©2002

## Monitor Implementation



**Conditional-wait construct: *x.wait(c)*;**

- c* integer expression evaluated when the *wait* operation is executed.
- value of *c* (a *priority number*) stored with the name of the process that is suspended.
- when *x.signal* is executed, process with smallest associated priority number is resumed next.

Check two conditions to establish correctness of system:

- User processes must always make their calls on the monitor in a correct sequence.
- 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.


Operating System Concepts 7.59 Silberschatz, Galvin and Gagne ©2002

## Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstile*s to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Operating System Concepts 7.60 Silberschatz, Galvin and Gagne ©2002





## Windows 2000 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.

Uses *spinlocks* on multiprocessor systems.

Also provides *dispatcher objects* which may act as wither mutexes and semaphores.

Dispatcher objects may also provide *events*. An event acts much like a condition variable.



This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.