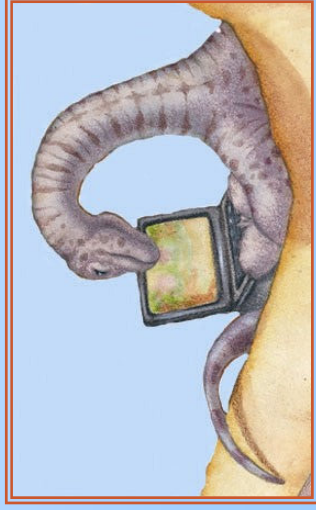


Chapter 6: Process Synchronization





Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.





Producer

```
while (true)
{
    /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ; // do nothing, buffer is full

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```





Consumer

```
while (true)
{
    while (count == 0)
        ; // do nothing - buffer is empty

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume the item in nextConsumed */
}
```





Race Condition

- `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6}
S5: consumer execute count = register2 {count = 4}
```





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 process solution
- Assume that the LOAD and STORE instructions are atomic; that is, 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

```
while (true)
{
    flag[ i ] = TRUE;
    turn = j;
    while ( flag[ j ] && turn == j )
        ; /* do nothing */
    /* CRITICAL SECTION */

    flag[ i ] = FALSE;
    /* REMAINDER SECTION */
}
```





Synchronization Hardware

- Many systems provide hardware support for critical section code
- 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-interruptable**
 - Either test memory word and set value
 - Or swap contents of two memory words





TestAndndSet Instruction

- Definition:

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





Solution using TestAndSet

- Shared boolean variable lock., initialized to false.

- Solution:

```
while (true)
{
    while ( TestAndSet (&lock ))
        ; /* do nothing

/* CRITICAL SECTION */

    lock = FALSE;

/* REMAINDER SECTION */

}
```





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:

```
while (true) {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
}
```





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`
 `; // no-op`
 `S--;`
 `}`
- `signal (S) {`
 `S++;`
 `}`





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
- Provides mutual exclusion
 - Semaphore **S**; // initialized to 1
 - wait (**S**);
Critical Section
 - signal (**S**);





Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
 - Could now have busy waiting in critical section implementation
 - ▶ But implementation code is short
 - ▶ Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore 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 (of type integer)
 - 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 with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){  
    value--;  
    if (value < 0) {  
        add this process to waiting queue  
        block(); }  
}
```

- Implementation of signal:

```
Signal (S){  
    value++;  
    if (value <= 0) {  
        remove a process P from the waiting queue  
        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

wait (S);

wait (Q);

.

.

.

signal (S);

signal (Q);

P_1

wait (Q);

wait (S);

.

.

.

signal (Q);

signal (S);

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





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 **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

```
while (true) {  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
}
```





Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
}
```





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 **mutex** initialized to 1.
 - Semaphore **wrt** initialized to 1.
 - Integer **readcount** initialized to 0.





Readers-Writers Problem (Cont.)

- The structure of a writer process

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





Readers-Writers Problem (Cont.)

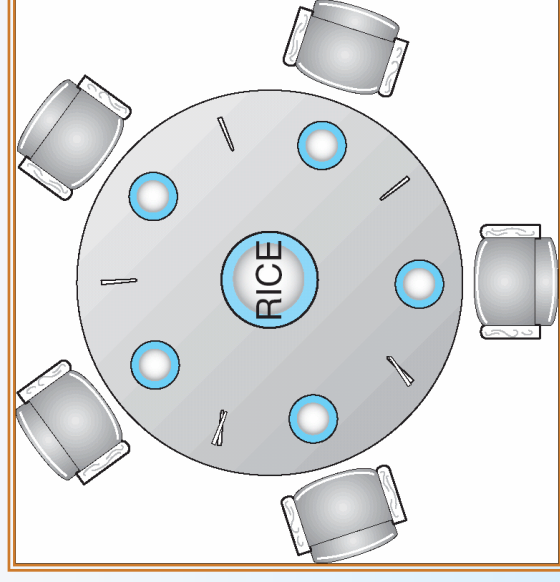
- The structure of a reader process

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





Dining-Philosophers Problem



- Shared data
 - Bowl of rice (data set)
 - Semaphore **chopstick [5]** initialized to 1





Dining-Philosophers Problem (Cont.)

- The structure of Philosopher i :

```
While (true) {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
  
    // eat  
  
    signal ( chopstick[i] );  
    signal ( chopstick[ (i + 1) % 5] );  
  
    // think  
  
}
```





Problems with Semaphores

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





Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

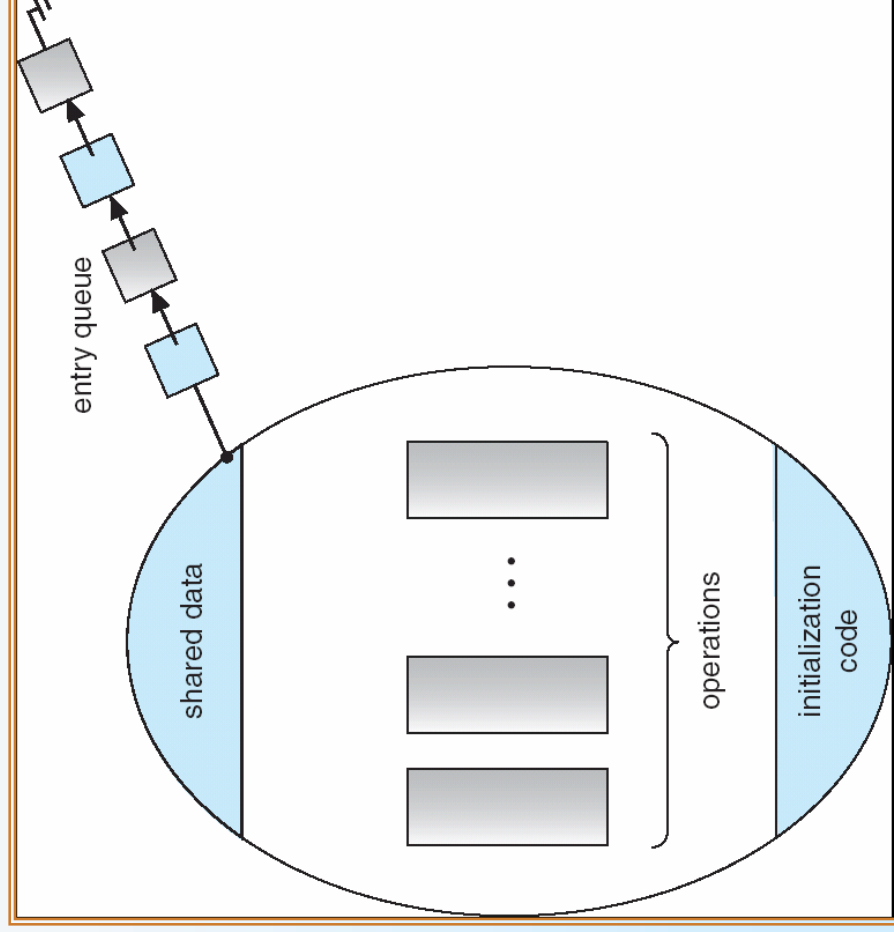
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ..... }
    ...
    procedure Pn (...) {.....}

    Initialization code ( ..... ) { ... }
    ...
}
```





Schematic view of a Monitor





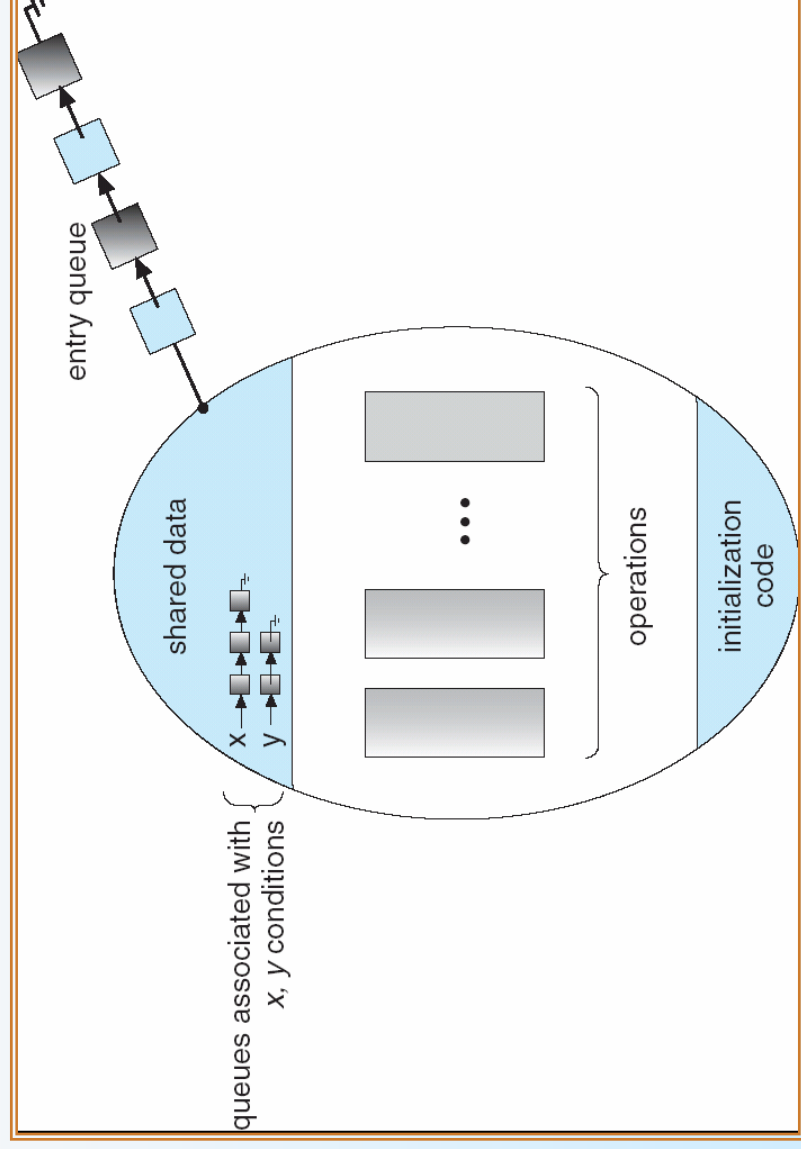
Condition Variables

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





Monitor with Condition Variables





Solution to Dining Philosophers

```
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)

```
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;
}
}
```





Solution to Dining Philosophers (cont)

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

`dp.pickup (i)`

EAT

`dp.putdown (i)`





Monitor Implementation Using Semaphores

- Variables

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

- Each 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.





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--;
```





Monitor Implementation

- The operation `x.signal` can be implemented as:

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



End of Chapter 6

