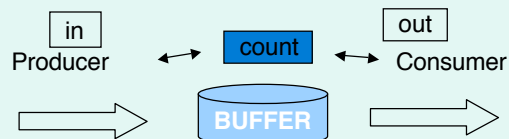


Synchronization Principles

Gordon College
Stephen Brinton

The Problem with Concurrency

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- **CONSUMER-PRODUCER problem**



Producer-Consumer

PRODUCER

```
while (true)
{
    /* produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

CONSUMER

```
while (true)
{
    while (count == 0)
        ; // do nothing
    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

1. Mutual Exclusion – exclusive access to the critical section of the cooperating group.

```
do {  
    Entry section  
    critical section  
    Exit section  
    remainder section  
} while (TRUE);
```

Solution to Critical Section (CS)

1. Mutual Exclusion – exclusive access to the critical section of the cooperating group.
2. Progress – no process in CS – then selection of process to enter CS cannot be postponed indefinitely

Solution to Critical Section

1. Mutual Exclusion – exclusive access to the critical section of the cooperating group.
2. Progress – no process in CS – then selection of process to enter CS cannot be postponed indefinitely
3. Bounded Waiting - There exists a bound (or limit) on the number of times other processes can enter CS after a process has made a request to enter and before it enters.

Peterson's Solution: Algorithmic Model

- 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: process is ready to enter the critical section. If (**flag[i]** == true) implies that process P_i is ready!

Peterson's Solution: Process P1

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j);  
    CRITICAL SECTION  
    flag[j] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

Acquire Lock

Release Lock

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 (must tell all CPUs)
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions: Atomic = non-interruptable
 - Two types:
 - test memory word and set value
 - swap contents of two memory words

TestAndndSet Instruction

Definition:

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

Solution Demo: TestAndndSet Instruction

- Shared boolean variable lock., initialized to false.
- Solution:

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

    // critical section

    lock = FALSE;                    Release Lock

    // remainder section

} while ( TRUE);
```

Solution Demo: TestAndndSet Instruction

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
```

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

Acquire Lock

```
    // critical section
```

```
    lock = FALSE;
```

Release Lock

```
    // remainder section
```

```
} while ( TRUE);
```

```
boolean TestAndSet (boolean *target)
```

```
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Swap Instruction

Definition:

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

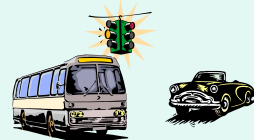
Solution Demo: Swap Instruction

- 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);
```

Semaphore



- Does this 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++;  
}
```


The Basic Semaphore

- 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**
- Provides mutual exclusion

Semaphore S; // initialized to 1

wait (S);

Critical Section

signal (S);

Another Semaphore Use

Process 1

Process 2

S ₁ ; signal(synch);	wait(synch); S ₂ ;
------------------------------------	----------------------------------

Both processes are running concurrently – statement S2 must be executed only after executing statement S1

Semaphore Implementation

- Requires Busy Waiting (waste of CPU cycles)
 - Called a “Spin Lock”
- Can modify the definition of wait() and signal():
 - No busy waiting
 - Uses a queue, block, and wakeup

```
typedef struct {  
    int value;  
    struct process *list  
} semaphore;
```

Semaphore Implementation: no Busy waiting

■ Implementation of wait:

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

■ Implementation of signal:

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

Semaphore Implementation: 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

- Must be executed atomically: no 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

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.

Deadlock and Starvation Solution?



What is a transaction?

- A transaction is a list of operations
 - When the system begins to execute the list, it must execute all of them without interruption, or
 - It must not execute any at all
- Example: List manipulator
 - Add or delete an element from a list
 - Adjust the list descriptor, e.g., length
- Too heavyweight – need something simpler

Well-known Problems of Synchronization

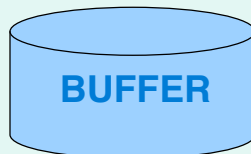
Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

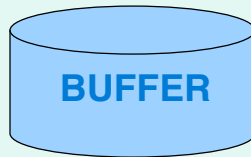
Bounded-Buffer Problem

- 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

- 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  
    wait (empty);  
    wait (mutex);  
    // add the item 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  
    signal (mutex);  
    signal (empty);  
    // consume the removed item  
} 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 **mutex** initialized to 1.
 - Semaphore **wrt** initialized to 1.
 - Integer **readcount** 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) ;  
    readcount ++ ;  
    if (readcount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount -- ;  
    if readcount == 0) signal (wrt) ;  
    signal (mutex) ;  
} while (true)
```


Readers-Writers Locks

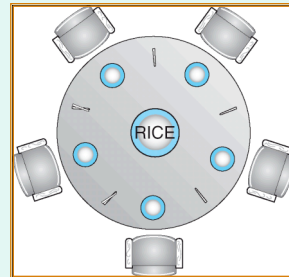
Generalized to provide reader-writer locks on some systems.

Most useful in following situations:

1. In apps where it is easy to identify which processes only read shared data and which only write shared data.
2. In apps with more readers than writers.
More overhead to create reader-writer lock than plain semaphores.

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 :

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

Dining-Philosophers Problem (Cont.)

- 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) ;
```

DEADLOCK POSSIBLE

Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
No mutual exclusion
 - wait (mutex) ... wait (mutex)
Deadlock
 - Omitting of wait (mutex) or signal (mutex) (or both)
Either no mutual exclusion or deadlock

Monitors

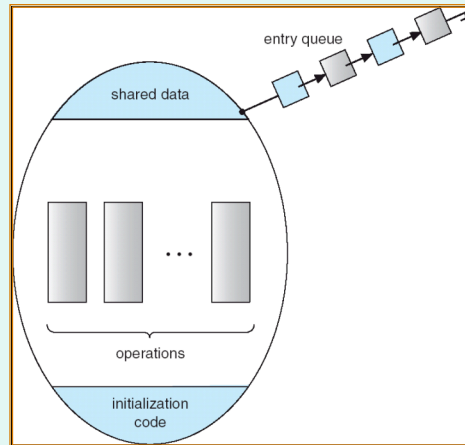
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

    procedure Pn (...) {.....}

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

Schematic view of a Monitor



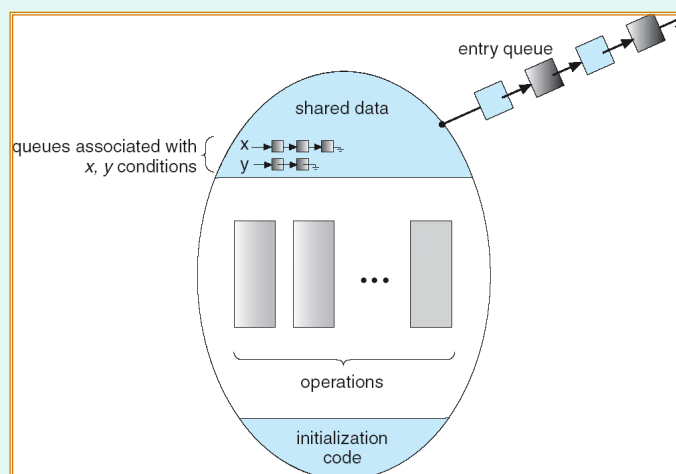
Condition Variables

- `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 ()`

Condition Variables

- If Q is signaled to continue then P must wait:
 - Note: remember only one process in monitor at a time
- Possible scenarios:
 - Signal and wait: P waits for Q to leave or suspend
 - Signal and continue: Q waits for P to leave or suspend

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);
}
```

```
dp.pickup(i);
...
Eat
...
dp.putdown(i);
```

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;
}
}
```

Java Monitors

- Every object in Java has associate with it a single lock
- A method declared synchronized means - calling the method means capturing the lock for the object.

```
public class SimpleClass {  
    ...  
    public synchronized void safeMethod() {  
        ...  
    }  
}
```

Java Monitors

- Every object in Java has associate with it a single lock
- A method declared synchronized means - calling the method means capturing the lock for the object.

```
public class SimpleClass {  
    ...  
    public synchronized void safeMethod() {  
        ...  
    }  
}
```

```
SimpleClass sc = new SimpleClass();
```

Synchronization Examples

- Windows XP
- Linux
- Pthreads

Windows XP 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 either mutexes and semaphores
- Dispatcher objects may also provide events
 - An event acts much like a condition variable

Linux Synchronization

- Linux:
 - disables interrupts to implement short critical sections
- Linux provides:
 - semaphores
 - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
 - mutex locks
 - condition variables
- Non-portable extensions include:
 - read-write locks
 - spin locks