

PROCESS & THREAD SYNCHRONIZATION

CS 475

1

Background

- ❑ Concurrent access to shared data may result in data inconsistency.
- ❑ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- ❑ Bounded Buffer problem (also called producer consumer problem)

2

Bounded-Buffer

❑ Shared data

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

3

Bounded-Buffer

❑ Producer process

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

4

Bounded-Buffer

- ❑ Consumer process

`item nextConsumed;`

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

5

Bounded Buffer

- ❑ The statements

`counter++;`
`counter--;`

must be performed *atomically*.

- ❑ Atomic operation means an operation that completes in its entirety without interruption.

6

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
```

7

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.

8

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.

9

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**.

10

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.

11

Mutual Exclusion: Conditions for Solution

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

12

Solutions to the Problem

- General structure of process P_i

```
do {  
    entry section  
    critical section  
    exit section  
    reminder section  
} while (1);
```

- Processes may share some common variables to synchronize their actions.

13

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```

14

Mutual Exclusion with Test-and-Set

- ❑ Shared data:
 boolean lock = false;
- ❑ Process P_i
 do {
 while (TestAndSet(lock)) ;
 critical section
 lock = false;
 remainder section
 }

15

Semaphores

- ❑ The solution we have looked at (TSL instruction) involves ***busy waiting***
 - ***Potential waste of CPU cycles***
- ❑ Semaphores are synchronization mechanism *that does not require busy waiting.*
 - Uses ***blocking synchronization***
- ❑ can only be accessed via two indivisible (atomic) operations: wait() and signal()
- ❑ Each semaphore has an integer value and a queue associated with it

16

Semaphore Implementation

- ❑ Define a semaphore as a record

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

- ❑ Assume two simple operations:

- o **block** suspends the process that invokes it.
- o **wakeup(P)** resumes the execution of a blocked process **P**.

17

Implementation

- ❑ Semaphore operations 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);  
    }
```

18

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);

19

Implementation cont'd

- ❑ Critical aspect of semaphore implementation is that the **wait()** and **signal()** operations must be executed atomically
 - need to guarantee that no two processes can execute **wait()** or **signal()** at the same time
 - **Wait()** and **signal()** have to be executed as critical sections!!
- ❑ Uniprocessors – disable interrupts while executing **wait()** and **signal()**
- ❑ Multiprocessors – disabling interrupts will not work because there are multiple processors
 - Most current CPUs have hardware support available (TSL), use for implementing critical section

20

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:

$\underline{P_i}$	$\underline{P_j}$
code	code
A	$wait(flag)$
$signal(flag)$	B

21

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
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
\vdots	\vdots
$signal(S);$	$signal(Q);$
$signal(Q)$	$signal(S);$

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

22

Classical Problems of Synchronization

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

23

Bounded-Buffer Problem

- ❑ Shared data

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

24

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);
```

25

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);
```

26

Readers-Writers Problem

❑ Shared data

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0

27

Readers-Writers Problem Writer Process

wait(wrt);

...

writing is performed

...

signal(wrt);

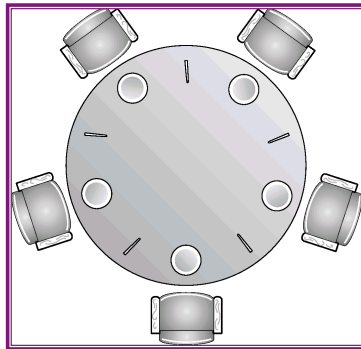
28

Readers-Writers Problem Reader Process

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

29

Dining-Philosophers Problem



❑ Shared data

```
semaphore chopstick[5];
```

Initially all values are 1

30

Dining-Philosophers Problem: A non-solution

Philosopher i :

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

31

High-level synchronization mechanisms

- ❑ Semaphores are a very powerful mechanism for process synchronization, but they are a *low-level* mechanism
- ❑ Several high-level mechanisms that are easier to use have been proposed
 - Monitors
 - Critical Regions
 - Read/Write Locks
- ❑ We will study monitors (Java and Pthreads provide synchronization mechanisms based on monitors)
- ❑ NOTE: high-level mechanisms easier to use but equivalent to semaphores in power

32

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 Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

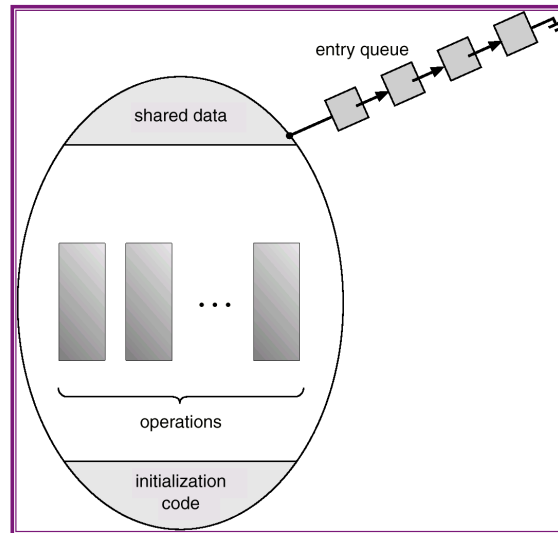
33

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**.
 - o The operation
x.wait();
means that the process invoking this operation is suspended until another process invokes
x.signal();
 - o The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

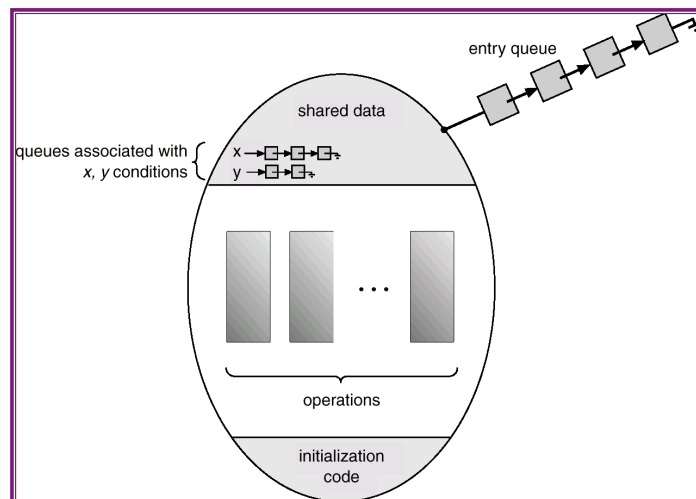
34

Schematic View of a Monitor



35

Monitor With Condition Variables



36

Producer-Consumer using monitors

```
monitor ProducerConsumer
condition full, empty;
integer count;
procedure insert(item: integer);
begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
end;
function remove: integer;
begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
end;
count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

37

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;
    }
}
```

38

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);
}
```

39

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();
    }
}
```

40

Synchronization Mechanisms

❑ Pthreads

- Semaphores
- Mutex locks
- Condition Variables
- Reader/Writer Locks

❑ Java

- Each object has an (implicitly) associated lock and condition variable

41

Java thread synchronization calls

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), *object.notifyAll()*

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

42

Mutual exclusion in Java

```
class Interfere {  
    private int data = 0;  
    public synchronized void update() {  
        data++;  
    }  
}
```

```
class Interfere {  
    private int data = 0;  
    public void update() {  
        synchronized(this) {  
            data++;  
        }  
    }  
}
```

43

Producer consumer using Java

```
public class ProducerConsumer {  
    static final int N = 100; // constant giving the buffer size  
    static producer p = new producer(); // instantiate a new producer thread  
    static consumer c = new consumer(); // instantiate a new consumer thread  
    static our_monitor mon = new our_monitor(); // instantiate a new monitor  
    public static void main(String args[]) {  
        p.start(); // start the producer thread  
        c.start(); // start the consumer thread  
    }  
    static class producer extends Thread {  
        public void run() { // run method contains the thread code  
            int item;  
            while (true) { // producer loop  
                item = produce_item();  
                mon.insert(item);  
            }  
        }  
        private int produce_item() { ... } // actually produce  
    }  
    static class consumer extends Thread {  
        public void run() { // run method contains the thread code  
            int item;  
            while (true) { // consumer loop  
                item = mon.remove();  
                consume_item(item);  
            }  
        }  
        private void consume_item(int item) { ... } // actually consume  
    }  
}
```

44

Producer consumer using Java cont'd

```
static class our_monitor {           // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer [hi] = val;             // insert an item into the buffer
        hi = (hi + 1) % N;              // slot to place next item in
        count = count + 1;              // one more item in the buffer now
        if (count == 1) notify();       // if consumer was sleeping, wake it up
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer [lo];             // fetch an item from the buffer
        lo = (lo + 1) % N;             // slot to fetch next item from
        count = count - 1;             // one less items in the buffer
        if (count == N - 1) notify();  // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```

45

Pthreads Synchronization Mechanisms

- ❑ Mutex Locks
- ❑ Condition Variables
- ❑ Semaphores
- ❑ Read Write Locks

46

Mutex Locks

- Mutual Exclusion Locks
- Example:

```
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
int count;

increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

get_count()
{
    int c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return(c);
}
```

Condition Variables

- based on **monitor** condition variables
- Easier to understand and use than semaphores
- cond_wait and cond_signal operations
 - different semantics from semaphore wait and signal operations
- always use in conjunction with a mutex lock

Producer-Consumer using condition variables

```
char buf[BSIZE];
int occupied;
int nextin;
int nextout;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

producer(char item)
{
    pthread_mutex_lock(&mutex);
    while (occupied == BSIZE)
        pthread_cond_wait(&not_full, &mutex);

    /* insert item */
    buf[nextin++] = item;
    nextin = nextin % BSIZE;
    occupied++;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}
```

```
consumer()
{
    pthread_mutex_lock(&mutex);
    while (occupied == 0)
        pthread_cond_wait(&not_empty, &mutex);

    /* consume item */
    item = buf[nextout++];
    nextout = nextout % BSIZE;
    occupied--;
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);
}
```