Operating Systems(Server)

Lecture 5
Process Synchronisation 2

Dr. Kevin Farrell

Chapter 7 Part 2: Process Synchronization 2

- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Semaphores

- Synchronization tool for generalised solutions to more complex problems
- A Semaphore, *S*, is an integer variable
- Semaphores are low-level synchronisation mechanisms; i.e. the operations on semaphores are implemented as **system calls**
- **The operations below are the traditional (Dijkstra) definitions of wait(S) and signal(S)**
- Can only be accessed via two indivisible (atomic) operations:

```
wait (S)
{
    while(S ≤ 0)
        ; // Do nothing
    S - -;
}

signal (S)
{
    S++;
}
```

Semaphore Usage: Critical Section of *n* Processes

- Shared data:
  semaphore mutex; //initially *mutex* = 1
- Process P_i *with mutual exclusion as follows:*

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

Problems with this Semaphore Definition

- Main problem with previous definition of mutual exclusion solutions to critical-section problem, and with our current definition of *wait(S)* on the semaphore is that they all require **busy waiting**.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** because the process “spins” (loops) while waiting for the lock to be released.
- Spinlocks are useful in multiprocessor systems: suppose process, P_0 has the lock and is executing on one processor, then process, P_1 can spin on another processor, waiting for the lock, while P_0 continues to execute.
- Spinlocks are advantageous in situations where context switching to another READY process, P_2 (say), would take considerable time, whereas the spinlock of P_1 would only occupy the processor for a relatively short time
- i.e. if locks are expected to be held for short times, allow spinlocks, and therefore avoid having to make time-consuming context-switches

Semaphore: New Implementation

- **New Definition: d**efine 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**.

New Implementation: Code

- 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);
}
```

New Implementation: Atomicity

- We need to implement wait() and signal() as atomic operations
- This is a critical section problem in its own right!
- On Uniprocessor systems, this can be achieved by:
  - Disabling Interrupts
- On Multiprocessor systems, disabling interrupts is undesirable. Instead, use
  - Hardware synchronisation mechanisms (for eg.: TSL) if they exist, or
  - Software synchronisation mechanisms such as our critical section algorithms of Lecture 5.
- The new implementation of the Semaphore has the following two important consequences:
  - Once a process blocks itself, it sleeps in a wait-queue in a BLOCKED state => it doesn't use CPU cycles for waiting. Therefore, it eliminates busy-waiting from application program code.
  - There *is* some busy-waiting for short periods in the algorithms used to implemented the wait() and signal() operations atomically.

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. This would happen, for example, if we implemented a LIFO policy to remove processes from the waiting queue. (LIFO = Last In First Out)

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.

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**

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);
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem (i.e. Producer-Consumer Problem)

- Shared data:
  - semaphore full, empty, mutex;

- empty = semaphore giving #empty buffer positions
- full = semaphore giving #full buffer positions
- mutex = semaphore for providing mutual exclusion

Initially:

full = 0, empty = n, mutex = 1

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);
```

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);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

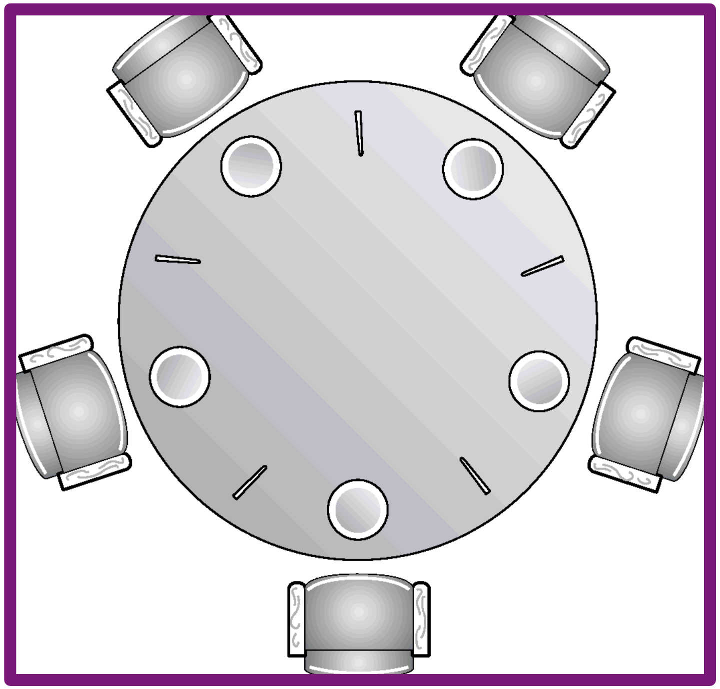
Readers-Writers Problem Writer Process

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

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):
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

Initially all values are 1

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);
```

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***.

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*.

Example – Bounded Buffer

- Shared data:

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

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++;
}
```

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--;
 }
```

Implementation of 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*.

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.

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
    }
}
```

Monitors

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

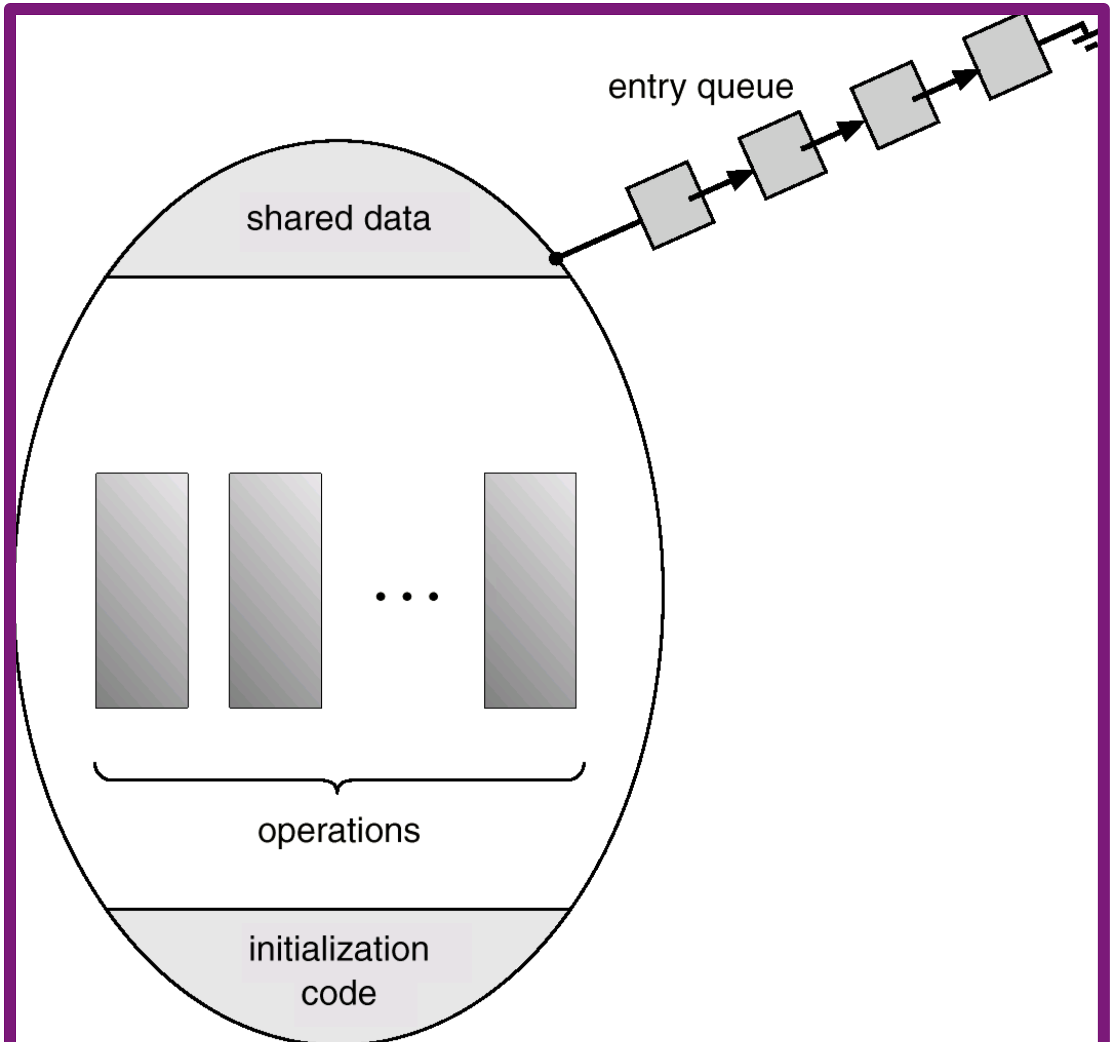
  condition x, y;
- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

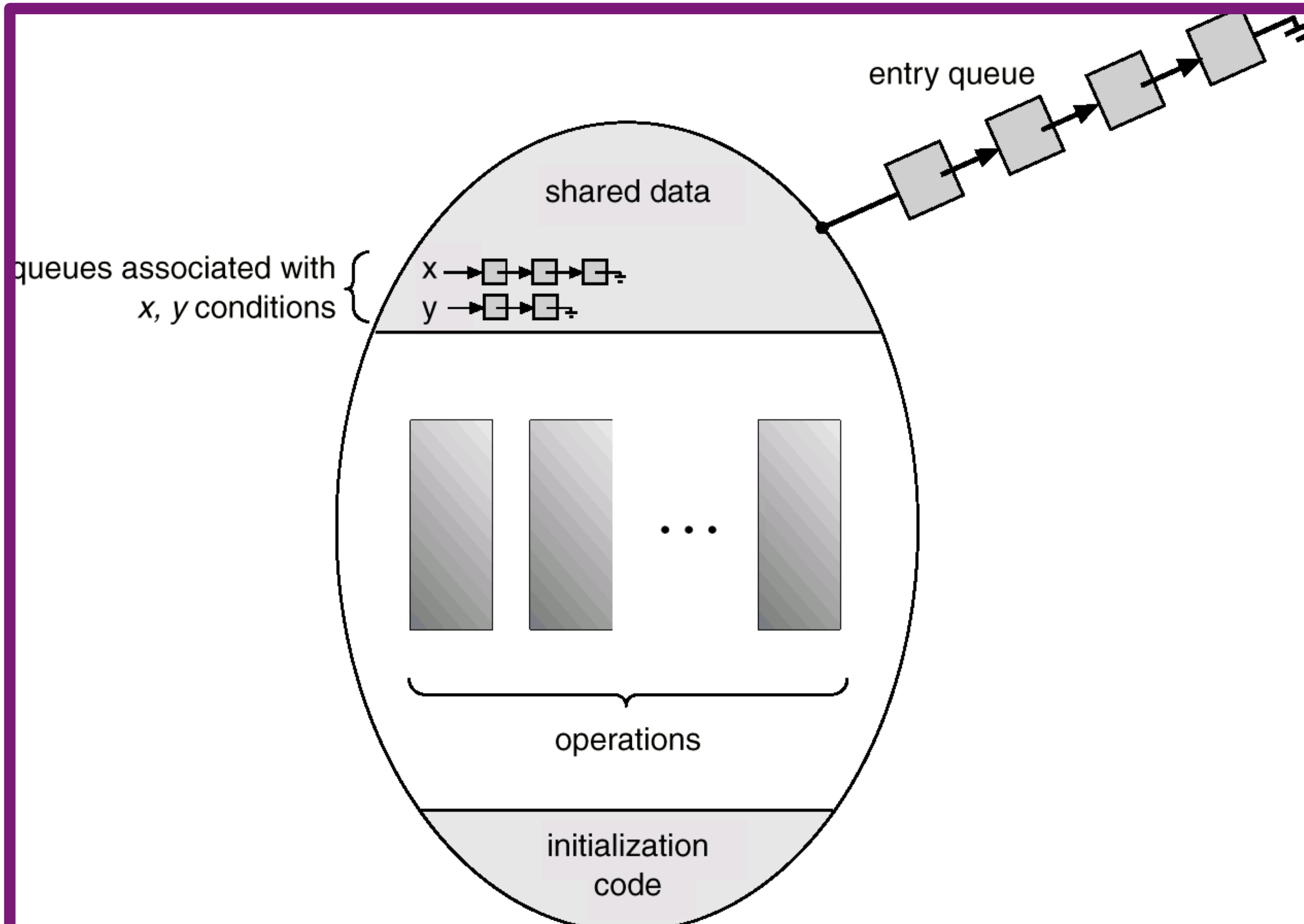
    means that the process invoking this operation is suspended until another process invokes

    x.signal();
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Schematic View of a Monitor



Monitor With Condition Variables



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;
  }
}
```

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);
}
```

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();
        }
}
```

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.

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--;
}
```

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.

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 *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

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.