CS 370: OPERATING SYSTEMS
[PROCESS SYNCHRONIZATION]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in the lecture

- Classical process synchronization problems
  - Bounded Buffer – Producer/Consumer problem
  - Readers Writers
  - Dining philosopher's problem

Classic Problems of Synchronization

The bounded buffer problem

- Binary semaphore (`mutex`)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - `empty`: Number of empty slots available to produce
    - Initialized to ***n***
  - `full`: Number of filled slots available to consume
    - Initialized to **0**

Some other things to bear in mind

- Producer and consumer must be **ready** before they **attempt to enter** critical section
- Producer readiness?
  - When a slot is available **to add** produced item
    - `wait(empty)`: `empty` is initialized to **n**
- Consumer readiness?
  - When a **producer has added** new item to the buffer
    - `wait(full)` : `full` initialized to **0**

The Producer

```
do {
        produce item nextp
        wait(empty);
        wait(mutex);

        add nextp to buffer

        signal(mutex);
        signal(full);

        remainder section

} while (TRUE);
```

`wait` till slot available

Only producer OR consumer can be in critical section

Allow producer OR consumer to (re)enter critical section

`signal` consumer that a slot is available

The Consumer

```
do {

    wait(full);
    wait(mutex);

    remove item from buffer
        (nextc)

    signal(mutex);
    signal(empty);

    consume nextc

} while (TRUE);
```

`wait` till slot available for consumption

Only producer OR consumer can be in critical section

Allow producer OR consumer to (re)enter critical section

`signal` producer that a slot is available to add

The Readers-Writers Problem

The Readers-Writers problem

- A database is **shared** among several concurrent processes

- Two types of processes
  - Readers
  - Writers

Readers-Writers: Potential for adverse effects

- If *two readers* access shared data simultaneously?
  - No problems

- If a *writer and some other reader* (or writer) access shared data simultaneously?
  - Chaos

Writers must have exclusive access to shared database while writing

- FIRST readers-writers problem:
  - No reader should wait for other readers to finish; simply because a writer is waiting
    - Writers may starve

- SECOND readers-writers problem:
  - If a writer is ready it performs its write ASAP
    - Readers may starve

Solution to the FIRST readers-writers problem

- Variable `int readcount`
  - Tracks how many readers are reading object

- Semaphore **`mutex`** **{1}**
  - Ensure mutual exclusion when `readcount` is accessed

- Semaphore **`wrt`** **{1}**
  1. Mutual exclusion for the writers
  2. First (last) reader that enters (exits) critical section
     - Not used by readers, when **other** readers **are in** their critical section

The Writer: When a writer signals either a waiting writer or the readers resume

```
do {

        wait(wrt);

        writing is performed

        signal(wrt);

} while (TRUE);
```

When:
writer in critical section
and if n readers waiting

1 reader is queued on **wrt**
(n-1) readers queued on **mutex**

The 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);
```

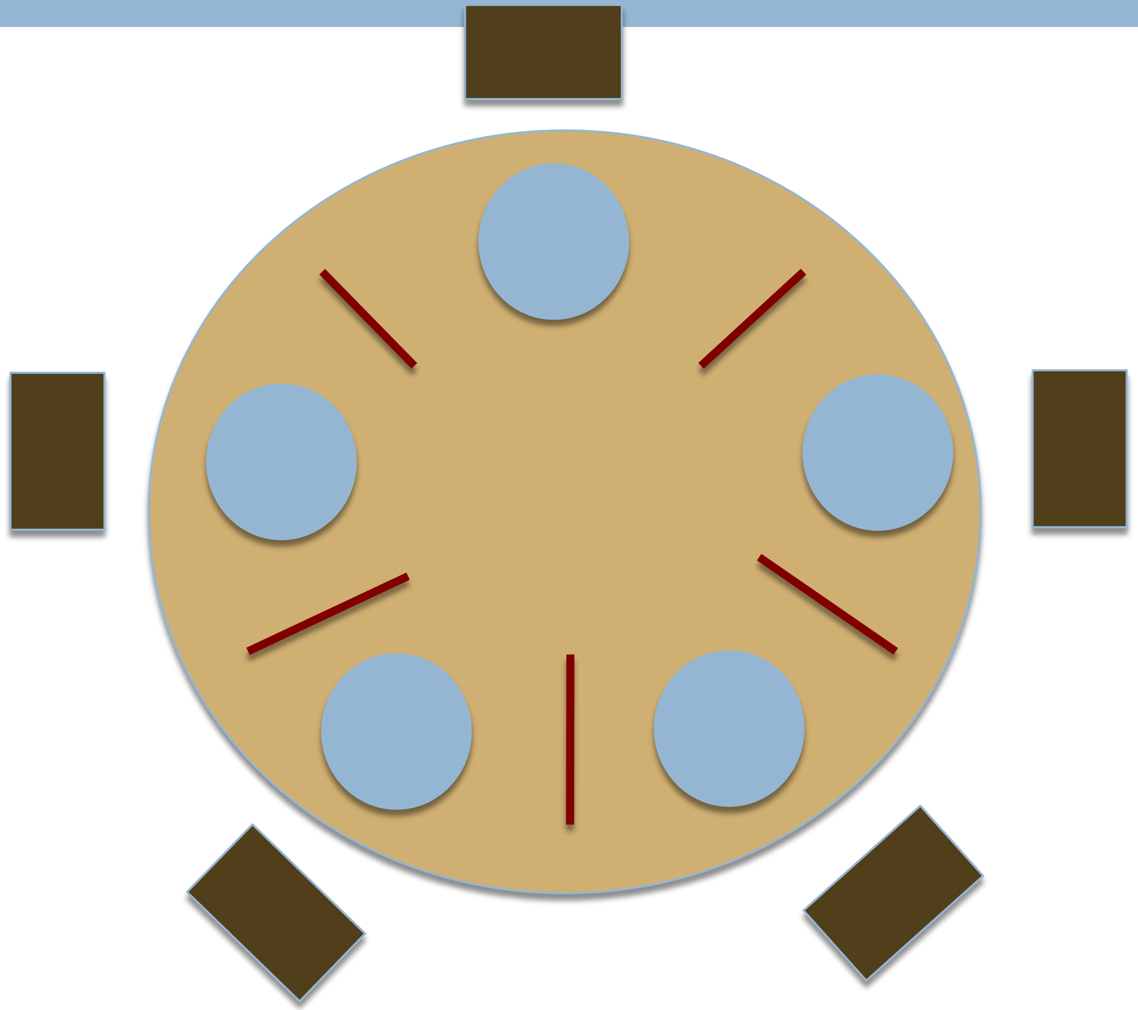
mutex for mutual exclusion to `readcount`

When:
writer in critical section and if n readers waiting

1 is queued on **wrt**
(n-1) queued on **mutex**

THE DINING PHILOSOPHERS PROBLEM

The situation



The Problem

① Philosopher tries to *pick up two closest* **{LR}** chopsticks

② Pick up only **1 chopstick at a time**

- Cannot pick up a chopstick being used

③ Eat only when you have *both* chopsticks

④ When done; *put down both* the chopsticks

Why is the problem important?

- Represents allocation of **several resources**
  - AMONG **several processes**

- Can this be done so that it is:
  - Deadlock free
  - Starvation free

Dining philosophers: Simple solution

- Each chopstick is a semaphore
  - Grab by executing `wait()`
  - Release by executing `signal()`

- Shared data
  - `semaphore chopstick[5];`
  - All elements are initialized to 1

What if all philosophers get hungry and grab the same {L/R} chopstick?

```
do {

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

            //eat

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

            //think

} while (TRUE);
```

Deadlock:
If all processes access chopstick with same hand

We will look at solution with monitors

MONITORS

Overview of the semaphore solution

- Processes share a semaphore **`mutex`**
  - Initialized to 1

- Each process MUST execute
  - **`wait`** *before entering* critical section
  - **`signal`** *after exiting* critical section

Incorrect use of semaphores can lead to timing errors

- Hard to detect
  - Reveal themselves only during specific execution sequences
- If correct sequence is not observed
  - 2 processes may be in critical section simultaneously
- Problems even if *only one* process is not well behaved

Incorrect use of semaphores: [1]
Interchange order of `wait` and `signal`

```
do {

        signal(mutex);
        critical section

        wait(mutex);

        remainder section

} while (TRUE);
```

Problem:
Several processes simultaneously active in critical section

NB: *Not always* reproducible

Incorrect use of semaphores: Replace `signal` with `wait` [2]

```
do {

        wait(mutex);
        critical section

        wait(mutex);

        remainder section

} while (TRUE);
```

Problem:
Deadlock!

Incorrect use of semaphores: [3]
What if you omit `signal` AND/OR `wait`?

```
do {

        wait(mutex);
        critical section

        signal(mutex);

        remainder section

} while (TRUE);
```

Omission:
Mutual exclusion violated

Omission:
Deadlock!

When programmers use semaphores incorrectly problems arise

- We need a higher-level synchronization construct
  - **Monitor**

- Before we move ahead: Abstract Data Types
  - Encapsulates *private data* with
    - *Public methods* to operate on them

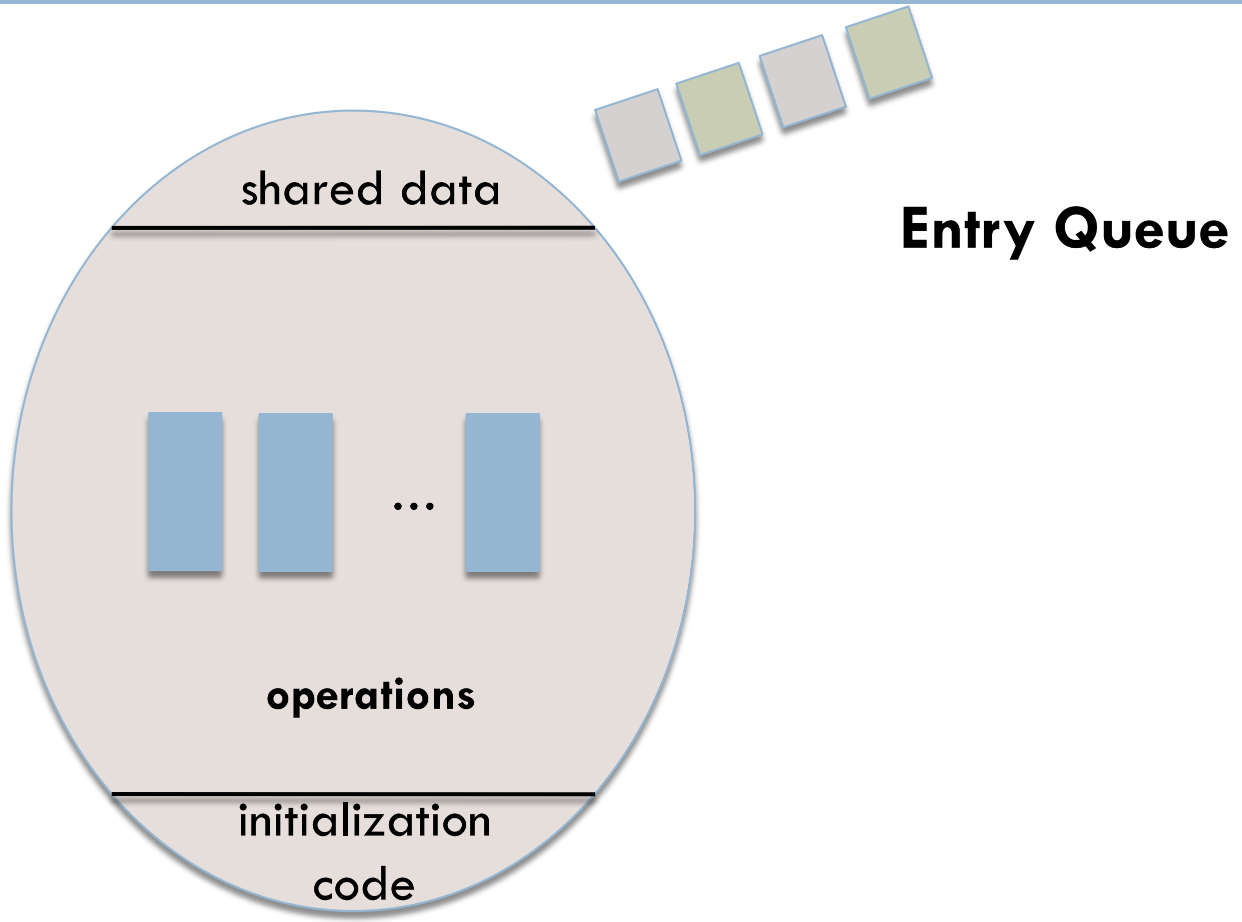
A monitor is an abstract data type

- Mutual exclusion provided **within** the monitor
- Contains:
  - Declaration of variables
    - Defining the instance's state
  - Functions that operate on these variables

Monitor construct ensures that only one process at a time is active within monitor

```
monitor monitor name {

    //shared variable declarations

    function F1(..) {.. .}

    function F2(..) {.. .}

    function Fn(..) {.. .}

    initialization code(..) {.. .}


}
```

Programmer does not code synchronization constraint explicitly



Basic monitor scheme not sufficiently powerful

- Provides an easy way to achieve mutual exclusion
- But … we also need a way for processes to **block** when they cannot proceed

This blocking capability is provided by the condition construct

- The **`condition`** construct
  - `condition x, y;`
- Operations on a **`condition`** variable
  - `wait`: e.g. **`x`**`.wait()`
    - Process invoking this is suspended UNTIL
  - `signal`: e.g. **`x`**`.signal()`
    - Resumes exactly-one suspended process
    - If no process waiting; NO EFFECT on state of **`x`**

Semantics of `wait` and `signal`

- `x.signal()` invoked by process **P**
- **Q** is the suspended process waiting on `x`

- *Signal and wait*: **P** waits for **Q** to leave monitor
- *Signal and continue*: **Q** waits till **P** leaves monitor

- PASCAL: When thread **P** calls signal
  - **P** leaves immediately
  - **Q** immediately resumed

Difference between the `signal()` in semaphores and monitors

- Monitors {condition variables}: Not persistent
  - If a signal is performed and no waiting threads?
    - Signal is simply ignored
  - During subsequent `wait` operations
    - Thread blocks
- Semaphores
  - Signal **increments** semaphore value *even if* there are no waiting threads
    - Future `wait` operations would immediately succeed!

Dining Philosophers Using Monitors

Dining-Philosophers Using Monitors Deadlock-free

```
enum {THINKING,HUNGRY,EATING} state[5];
```

- `state[i] = EATING` **only if**
  - `state[(i+4)%5] != EATING && state[(i+1)%5] != EATING`
- `condition self[5]`
  - **Delay** self when *HUNGRY but unable* to get chopsticks

Sequence of actions

- Before eating, must invoke `pickup()`
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

```
DiningPhilosophers.pickup(i);
         ...

         eat
         ...
DiningPhilosophers.putdown(i);
```

The `pickup()` and `putdown()` operations

```
pickup(int i) {
   state[i] = HUNGRY;
   test(i);
   if (state[i] != EATING) {
      self[i].wait();
   }
}

putdown(int i) {
   state[i] = THINKING;
   test( (i+4)%5 );
   test( (i+1)%5 );
}
```

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick

`test()` to see if philosopher can eat

Eat only if HUNGRY and Person on **Left AND Right** are <u>not</u> eating

```
test(int i) {
  if (state[(i+4)%5] != EATING &&
      state[i] == HUNGRY &&
      state[(i+1)%5 != EATING] ) {

   state[i] = EATING;
   self[i].signal();
  }
}
```

Signal a process that was suspended while trying to eat

Possibility of starvation

- Philosopher `i` can **starve** if eating periods of philosophers on left and right overlap

- Possible solution
  - Introduce new state: STARVING
  - Chopsticks can be picked up if **no** neighbor is starving
    - Effectively wait for neighbor’s neighbor to stop eating
    - REDUCES concurrency!

Implementing a Monitor Using Semaphores

Implementing a monitor using semaphores

- For each monitor
  - Semaphore `mutex` initialized to 1

- Process must execute
  - `wait(mutex)` : Before entering the monitor
  - `signal(mutex)`: Before leaving the monitor

Semantics of the signaling process

- Signaling process must ***wait*** until the <u>resumed process</u> leaves or waits
  - Additional semaphore **`next`** is introduced

- So signaling process needs to **suspend itself**
  - Semaphore **`next`** initialized to 0
    - Signaling processes use **`next`** to suspend themselves
  - Integer variable `next_count`
    - Counts number of processes suspended on **`next`**

Implementing a function `F` in the monitor

```
wait(mutex);
    ...
  body of function F
    ...

if (next_count > 0) {
   signal(next);
} else {
   signal(mutex);
}
```

Implementing condition variables:

```
x_count++;
if (next_count > 0) {
   signal(next);
} else {
   signal(mutex);
}
wait(x_sem);
x_count--;
```

`x.wait()` Operation

For each condition **x** we have:
semaphore `xsem` and
integer variable `x_count`
Both initialized to 0

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

`x.signal()` Operation

Resuming processes within a monitor

- **{C1}** Several processes suspended on condition `x`
- **{C2}** `x.signal()` executed by some process

- Which suspended process should be resumed next?
  - Simple solution: FCFS ordering
    - Process waiting the longest is resumed first

Process resumption: conditional `wait`

- `x.wait(`**`c`**`)`
- **`c`** is an *integer expression*; evaluated when `wait()` is executed
- Value of **`c`** is the priority number
  - Stored with the name of process that is suspended
- When `x.signal()` is executed
  - Process with smallest priority number resumed next

Monitor to allocate a single resource

```
Monitor ResourceAllocator  {
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy) {
         x.wait(time);
      }
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }
   initialization() {busy = FALSE;}
}
```

An example of conditional waits

Specify maximum time resource will be used

```
R.acquire(t);
   ...
  access the resource;
   ...
R.release();
```

Monitor allocates resource based on shortest duration

Monitor *cannot* guarantee that the access sequence will be observed

Avoiding time dependent errors and ensuring that scheduling algorithm is not defeated

- User processes must make their calls on the monitor in **correct sequence**
- Ensure that uncooperative processes do not ignore the mutual exclusion gateway
  - Should not access resource directly!

The contents of this slide set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 5]

- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2]