CS 370: OPERATING SYSTEMS
[PROCESS SYNCHRONIZATION]

Shrideep Pallickara
Computer Science
Colorado State University

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

1

Frequently asked questions from the previous class survey

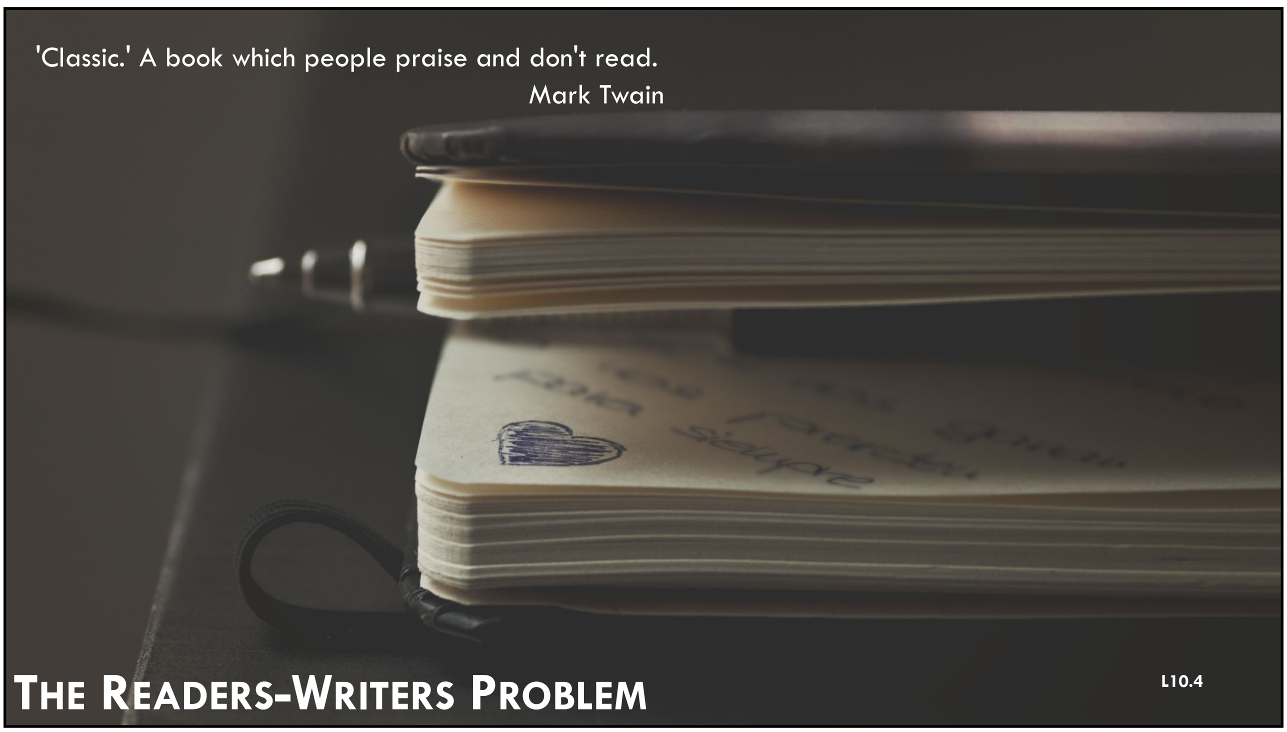
- Why is priority inversion a problem?
- Why use TestAndSet(); why not just check for the state of the lock variable: e.g., while(lock) { ….}
  - The example of N processes had it going clockwise, does it always have to go in that direction?
- Semaphore seems to be increasing and decreasing values. Why not just use variables?

2

Topics covered in the lecture

- Classical process synchronization problems
  - Readers Writers
  - Dining philosopher’s problem
- Monitors
  - Solving dining philosopher's problem using monitors
- Midterm

3



4

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**

9

The Reader process

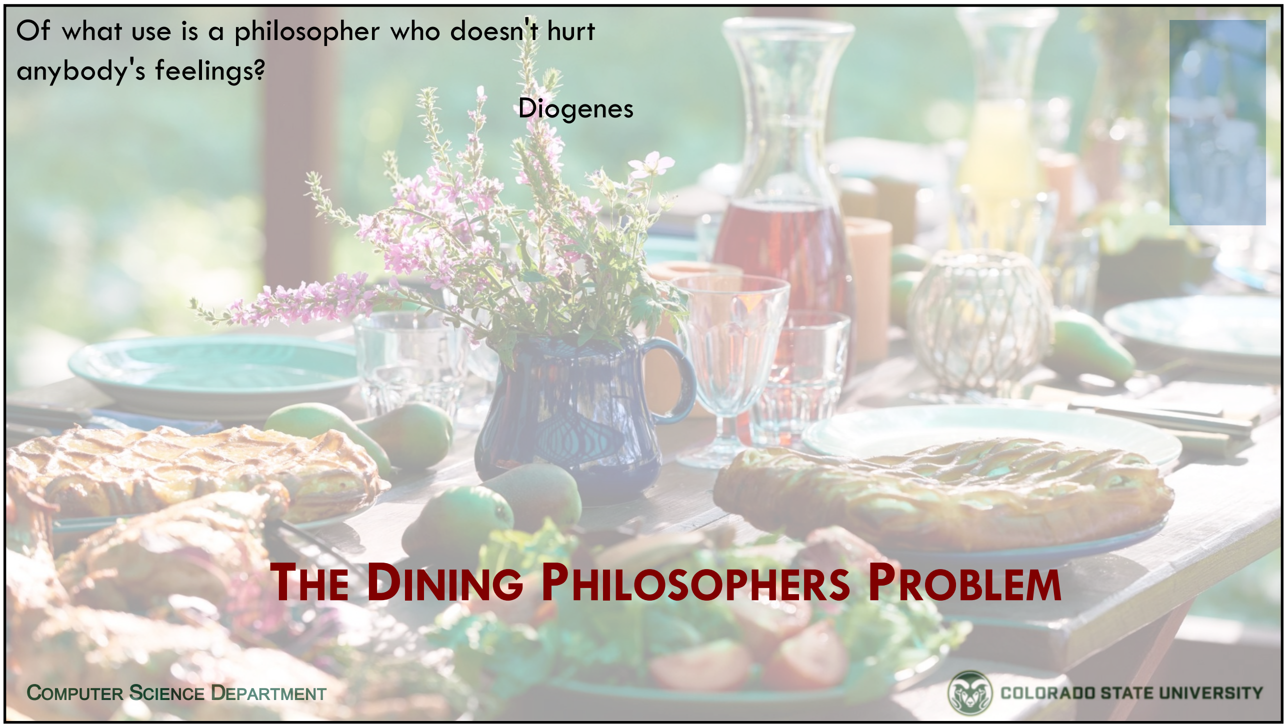
```
do {
        wait(mutex);
        readcount++;
        if (readcount ==1) {
           wait(wrt);
        }
        signal(mutex);

        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

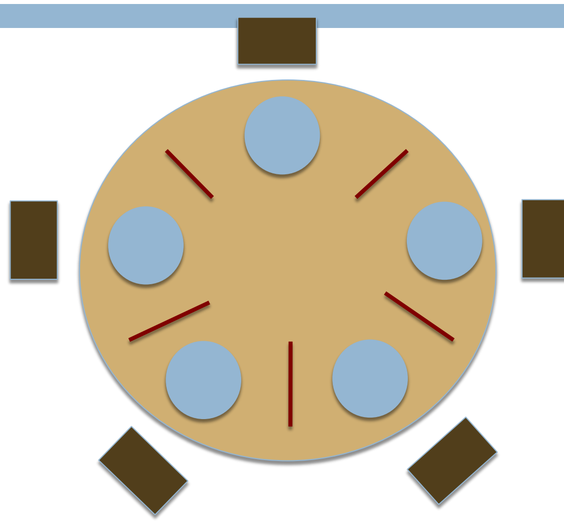
1 is queued on **wrt**
(n-1) queued on **mutex**

10



11

The situation



COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT | INTER-PROCESS SYNCHRONIZATION | L11.12

12

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 [1/2]

- Each chopstick is a semaphore
  - Grab by executing `wait()`
  - Release by executing `signal()`
- Shared data
  - `semaphore chopstick[5];`
  - All elements are initialized to 1

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT INTER-PROCESS SYNCHRONIZATION L11.15

15

Dining philosophers: Simple solution [2/2]

```
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

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT INTER-PROCESS SYNCHRONIZATION L11.16

16

And still they lead me back
To the long winding road
You left me standing here
A long, long time ago
Don't leave me waiting here
Lead me to your door

The Long and Winding Road, John Lennon/Paul McCartney

MONITORS

COMPUTER SCIENCE DEPARTMENT — COLORADO STATE UNIVERSITY

17

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

COLORADO STATE UNIVERSITY — Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT — INTER-PROCESS SYNCHRONIZATION — L11.18

18

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: Interchange order of `wait` and `signal` [1/3]

```
do {

        signal(mutex);
        critical section

        wait(mutex);

        remainder section

} while (TRUE);
```

Problem: Several processes simultaneously active in critical section

? What if?

NB: *Not always* reproducible

Incorrect use of semaphores: Replace `signal` with `wait` [2/3]

```
do {

        wait(mutex);
        critical section

        wait(mutex);

        remainder section

} while (TRUE);
```

What if?

Problem: Deadlock!

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

```
do {

        wait(mutex);
        critical section

        signal(mutex);

        remainder section

} while (TRUE);
```

Omission? (wait) — **Omission:** Mutual exclusion violated

Omission? (signal) — **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

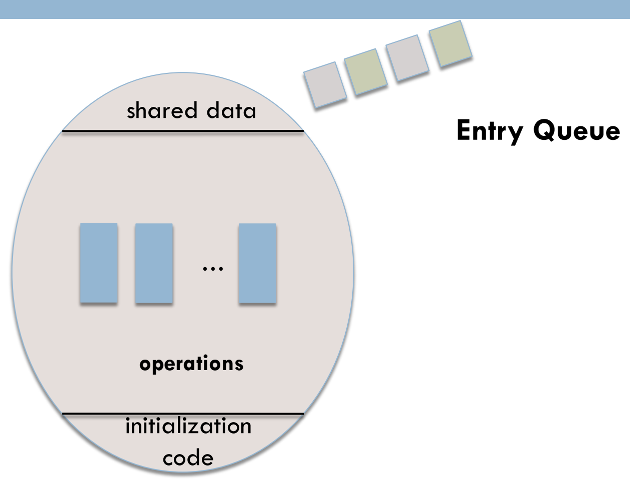
A monitor is an abstract data type

- 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`

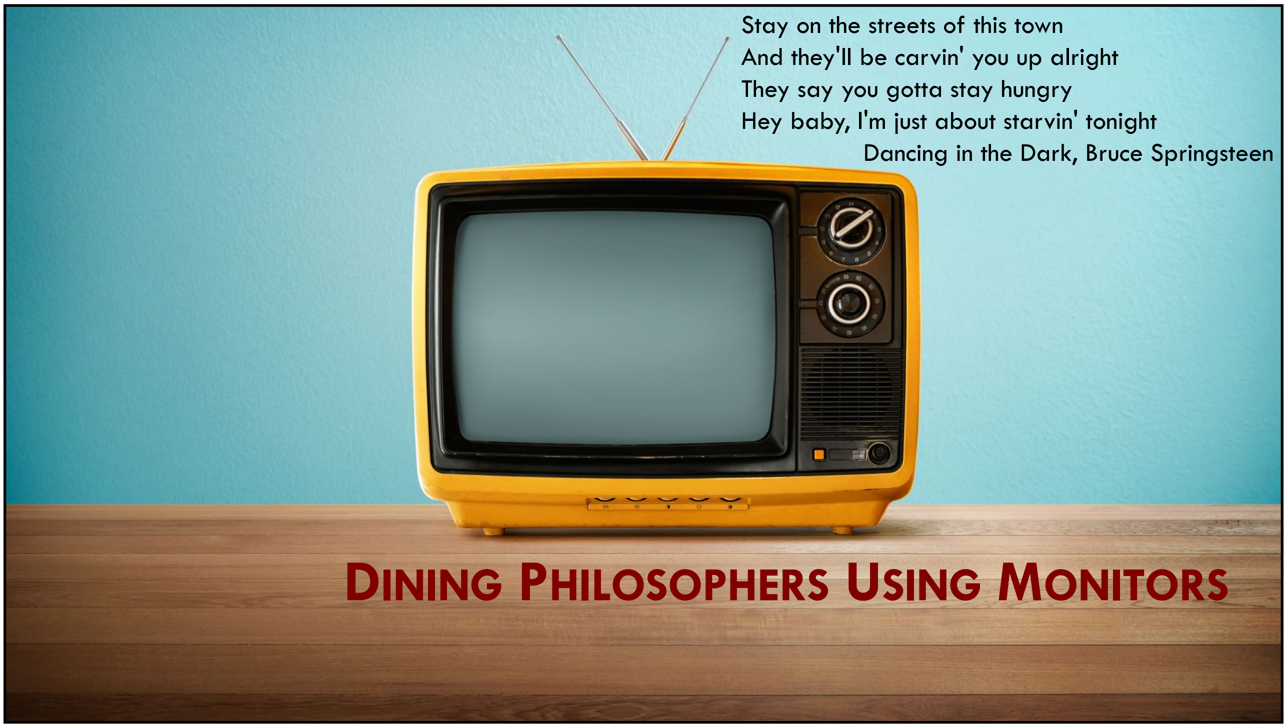
- `x.signal()` invoked by process **P**
- **Q** is the suspended process waiting on `x`

- *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!



31

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

32

Sequence of actions

- Before eating, must invoke `pickup()`
  - May result in suspension of the 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 not 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!

MID-TERM

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

37

Mid-term on Thursday, October 5th @ 2:00 pm

- Held in class
  - Those taking it at the Alternative Testing Center please work with SDC
- Accounts for 20% of your course grade
- Points distribution
  - Processes and Inter-Process Communications: 30 points
  - Threads: 20 points
  - Process Synchronization (including atomic transactions): 30 points

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT INTER-PROCESS SYNCHRONIZATION L11.38

38

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]