Operating System

Tutorial 9

Semaphores

&

Classical Problems of Synchronization

Critical Section Problem

- Critical section is a code segment that can be accessed by only one process at a time.
- Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.
- i.e. **count** variable

Critical Section

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

- In the entry section, the process requests for entry in the Critical Section.
- In the exit section, the process inform other processes about it’s completion.

Solutions to Critical Section Problem

1. Software Solution

   (Peterson’s Algorithm)

2. Hardware Solution

   (using TestAndSet instructions)

3. Semaphore Solution [Operating system]

   (using wait() and signal() operations)

4. Monitor

Solutions to Critical Section Problem

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion** : If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress** : If no process is in the critical section, then no other process from outside can block it from entering the critical section.
- **Bounded Waiting** : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Semaphores

- ***Semaphore*** is a type of generalized lock
  - Defined by Dijkstra in the last 60s
  - Main synchronization primitives used in UNIX
  - Consist of a positive integer value
  - Two operations
    - ***P()***: an atomic operation that waits for semaphore to become positive, then decrement it by 1
    - ***V()***: an atomic operation that increments semaphore by 1 and wakes up a waiting thread at P(), if any.

Wait() and Signal Operations

```
wait(S):

    while S ≤ 0 do

        no-op;

    S.value := S.value - 1;
```

```
signal(S):

    S := S + 1;
```

Why to use semaphores?

- Semaphores impose deliberate constraints that help programmers avoid errors (i.e. atomicity).
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

Two types of semaphores

- Binary semaphore (aka mutex semaphore)
  - sem is initialized to 1
  - guarantees mutually exclusive access to resource (e.g., a critical section of code)
  - only one thread/process allowed entry at a time

- Counting semaphore
  - sem is initialized to N
    - N = number of units available
  - represents resources with many (identical) units available
  - allows threads to enter as long as more units are available

Create a semaphore

int sem_init (sem_t* sem, int pshared, unsigned int value);

- *sem - the semaphore to be initialized*
- *pshared - indicates whether this semaphore is to be shared between the threads of a process [0], or between processes [non zero].*
- *value -* the initial value of the semaphore

- All semaphore functions return zero on success and non-zero on failure.

Semaphore Operations

- int **sem_post**(sem_t * *sem*);
  - This will increase the value of the semaphore by one.
  - (Implementation of Signal Operation)

- int **sem_wait** (sem_t* sem);
  - This will return immediately if the value of the semaphore is greater than zero and block the thread otherwise. It decreases the semaphore by one.

Semaphore Operations

- int **sem_destroy**(sem_t * *sem*);
  - It releases the resources that semaphore has and destroys it.
- int **sem_getvalue**(sem_t * *sem*, int * *semdeg*);
  - The current semaphore value is stored in semdeg variable.

Classical Problems

1. Producer-Consumer Problem
2. Reader-Writer Problem
3. Dining-Philosopher Problem

Readers-Writers Problem

- Consider a situation where we have a file shared between many people.
- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.
- Precisely in OS we call this situation as the **readers-writers problem** (often occurs in Databases)**.**

Solution for R-W problem

- Two semaphore variables **mutex, wrt,** and one integer variable **readcnt** are used to implement the solution.
- **readcnt** tells the number of processes performing read in the critical section, initially 0.
- semaphore **mutex** is used to ensure mutual exclusion when readcnt is updated.
- semaphore **wrt** is used to ensure mutual exclusion for writers and used by both readers and writers.

Solution for R-W problem

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure 6.12 The structure of a writer process.

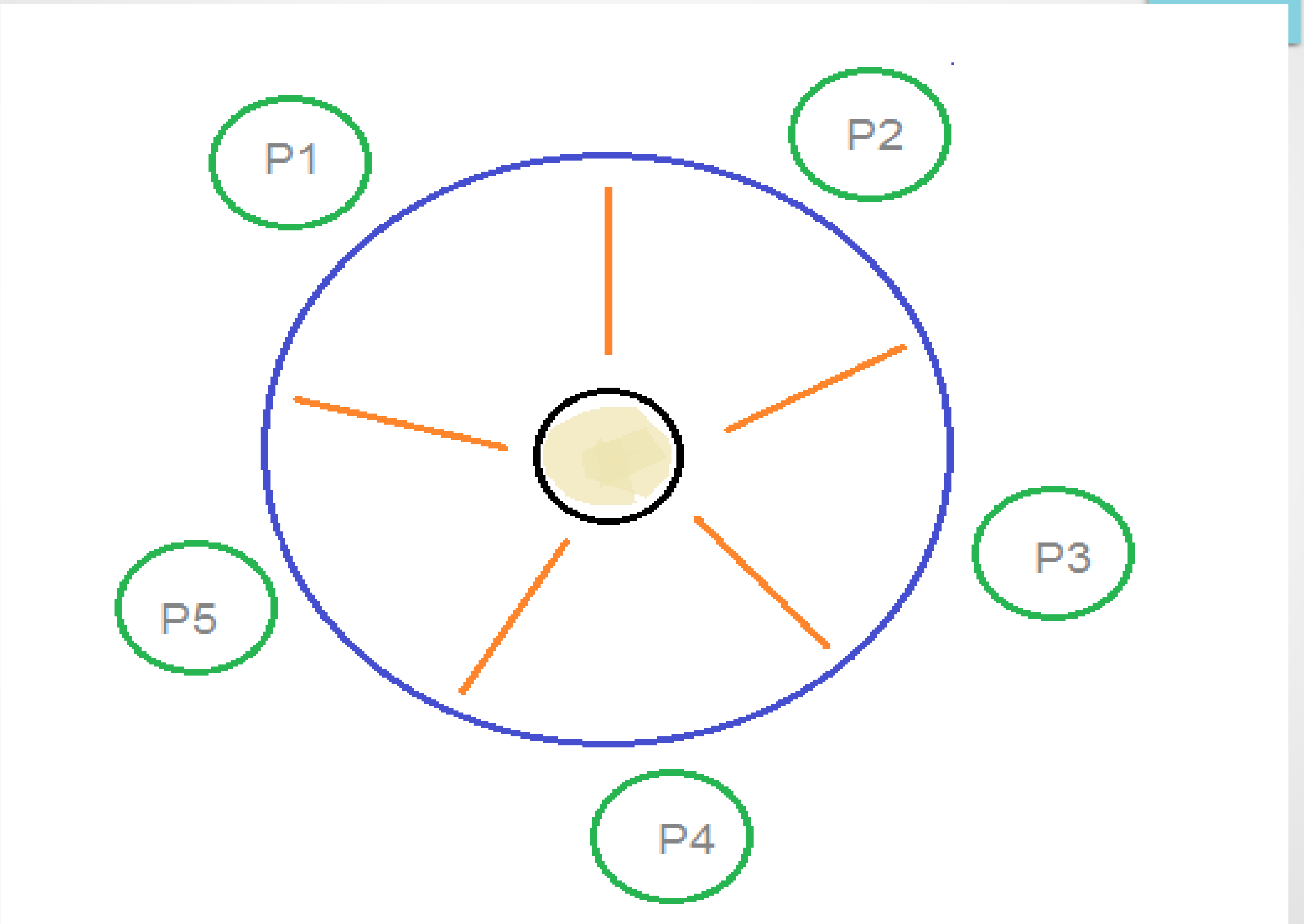
```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

gure 6.13 The structure of a reader process.

Dining-Philosopher Problem

- The Dining Philosopher Problem states that K philosophers seated around a circular table.
- There is one chopstick between each pair of philosopher.
- A philosopher may eat if he can pickup the two chopsticks adjacent to him (can pick only one at a time).
- One chopstick may be picked up by any one of its adjacent followers but not both.
- There are three states of philosopher : **THINKING, HUNGRY and EATING**.

Dining-Philosopher Problem



Solution for D-P Problem

- Here there are two semaphores : Mutex and a semaphore array for the philosophers.
- Mutex is used such that no two philosophers may access the pickup or putdown at the same time.
- The semaphore array chopstick[] is used to represent chopsticks.
- But, semaphores can result in deadlock due to programming errors.

Solution for D-P Problem

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

Figure 6.15 The structure of philosopher *i*.

Some Definitions

- **Deadlock**: A situation in which two or more processes are unable to proceed because each is waiting for one the others to do something.
- **Livelock**: A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work:
- **Starvation**: A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

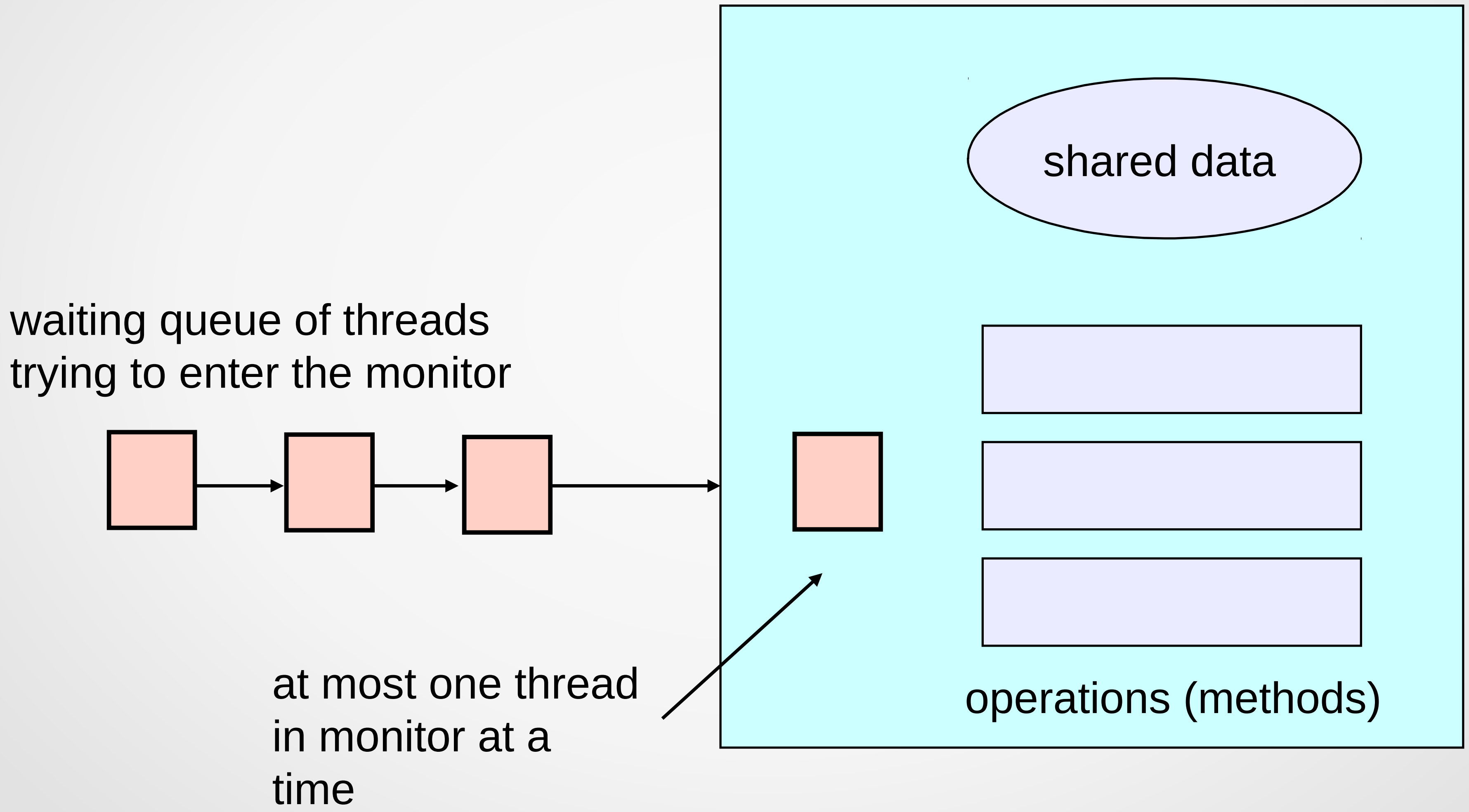
Problems with semaphores (and locks)

- They can be used to solve any of the traditional synchronization problems, but:
  - semaphores are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the semaphore and the data being controlled by it
  - used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - no control over their use, no guarantee of proper usage
- Thus, they are prone to bugs

One More Approach: Monitors

- A *monitor* is a programming language construct that supports controlled access to shared data
  - synchronization code is added by the compiler
    - why does this help?
- A monitor encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent threads that invoke those procedures
- Data can only be accessed from within the monitor, using the provided procedures
  - protects the data from unstructured access
- Addresses the key usability issues that arise with semaphores

A monitor


shared data
waiting queue of threads
trying to enter the monitor
at most one thread
in monitor at a
time
operations (methods)

Monitor facilities

- “Automatic” mutual exclusion
  - only one thread can be executing inside at any time
    - thus, synchronization is implicitly associated with the monitor – it “comes for free”
  - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
    - more restrictive than semaphores
    - but easier to use (most of the time)

- But, there’s a problem…

Task 1

Write a program using two threads modifying a shared variable in different way. Use semaphore to synchronize the threads.

Task 2

Implement Bounded-Buffer Producer-Consumer Problem and synchronize using semaphores.

(With one consumer and one producer)

Task 3

Suppose you have three threads A, B, and C. Thread A will execute **study_os()**, thread B will execute **drink_coffee()**, and thread C will execute **take_exam()**. Using a single semaphore, sketch code for each thread that will ensure that both **drink_coffee()** and **study_os()** happen before **take_exam()**. Remember to specify the starting value of the semaphore!

Task 4

Consider the two following threads:

T_1 = while true do print A

T_2 = while true do print B

a) Add semaphores such that the string printed is:

ABABABA...

b) Is it possible to have the same result using only instructions “sleep(n)” that interrupts a thread for “n” seconds?

Task 5

Implement sleeping barber problem.

(Exercise 6.63 in textbook)

Task 6

Implement Dining-Philosopher problem.