Operating Systems
CS2006

Lecture 11

Process Synchronization-II

(Semaphores)

22nd March 2023

Dr. Rana Asif Rehman

Semaphores

- A Semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
- wait(S)
  - sometimes called P()
    - Dutch proberen: "to test"
- signal(S)
  - sometimes called V()
    - Dutch verhogen: "to increment"

- The classical definition of **wait** and **signal** is as shown in the following figures
- Useful when critical sections last for a short time, or we have lots of CPUs
- S initialized to positive value (to allow someone in at the beginning)

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

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

Semaphores in Action

```
Initialize mutex to 1

Process Pi:                    Process Pj:
   repeat                      repeat
 wait(mutex);                     wait(mutex);
        CS                         CS
 signal(mutex);                   signal(mutex);
        RS                         RS
  forever                      forever
```

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

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

Semaphore

Processes: P1, P2… Pn   Initiate S=1;

Processes: P1

```
do {
    wait(S);

    Critical
    section

    Signal(S);


  }while;
```

Processes: P2

```
do {
    wait(S);

    Critical
    section

    Signal(S);


  }while;
```

Processes: P3

```
do {
    wait(S);

    Critical
    section

    Signal(S);


  }while;
```

Synchronizing Processes using Semaphores

- Two processes:
  - P_1 and P_2
- Statement S_1 in P_1 needs to be performed **before** statement S_2 in P_2
- We want a way to make P_2 wait
  - Until P_1 tells it is OK to proceed

Define a semaphore "synch"

Initialize synch to 0

Put this in P_2:

```
wait(synch);
S2;
```

And this in P_1:

```
S1;
signal(synch);
```

Semaphore - Busy Waiting

- Implementation of **Mutex locks** suffers from busy waiting – **spinlock.**
- **Similar issue in Semaphores:** When a process executes the **wait()** operation and finds that the semaphore value is not positive, **it must wait.**
- **Remedy:** Rather than engaging in busy waiting, the process can block itself.

⇒ The **block operation** places a process into a **waiting queue** *associated with the semaphore*, and the state of the process is switched to the **waiting state.**

⇒ Then control is transferred to the **CPU scheduler**, which selects another process to execute.

Semaphore - Busy Waiting (2)

- **wakeup ()** operation
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation, i.e. value of S changes to 1.
- The process is restarted by a wakeup() operation, which changes the process from the **waiting state** to the **ready state**.
- The process is then placed in the **ready queue**.
- The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.

Semaphore - Implementation

- Two operations:
  - **block** – place the process invoking the wait() operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- We can define a semaphore as follows.

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

- When a process must wait on a semaphore, it is added to the **`list`** of processes.
- A **`signal()`** operation removes one process from the list of waiting processes

and awakens that process.

- **`value`** shows how many processes are currently in blocked state.

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

The block() operation suspends the process that invokes it.

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

The wakeup(P) operation resumes the execution of a blocked process P.

- Semaphore values may be negative, whereas they are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlock and Starvation

- An implementation of a semaphore with a waiting queue may result in:
  - **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
    - Let S and Q be two semaphores initialized to 1

P0	**P1**
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
- If we add or remove processes from the list associated with a semaphore in LIFO manner

Classical Problems of Synchronization

```
wait(S) {
  while S<=0 do ;
  S- -;  }
```

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

Advantage: Signaling

- Simplest use for a semaphore is signaling.
- One thread sends signal to other thread that something has happened
- Assume that we have a semaphore named **sem** with initial value 0, and that Threads A and B have shared access to it.

Thread A

```
statement a1
sem.signal()
```

Thread B

```
sem.wait()
statement b1
```

- **statement** represents an arbitrary program statement
- Imagine that **a1** reads a line from a file, and **b1** displays the line on the screen.
- Semaphore guarantees that Thread A has completed **a1** before Thread B begins **b1**.

Rendezvous Problem

- Generalize the signal pattern so that it works both ways.
- Thread A has to wait for Thread B and vice versa.
- Given this code, we want to guarantee a1 happens before b2 and b1 happens before a2.

Thread A

```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

- We don't care about the order of a1 and b1.
- In writing your solution, be sure to specify the names and initial values of your semaphores.

Possible Solutions

- Initialize Semaphore aArrived, bArrived to 0
- Solution #1

Thread A

```
statement a1
bArrived.wait()
aArrived.signal()
statement a2
```

Thread B

```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

- Solution #2

Thread A

```
statement a1
aArrived.signal()
bArrived.wait()
statement a2
```

Thread B

```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

Mutex problem

- Solve this critical section problem
- Add semaphores to the following example to enforce mutual exclusion to the shared variable `count`.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Solution

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

- Such solutions are called symmetric solutions
- If threads had different code, it would be called asymmetric solution
- Question: What is mutex initialized to?

Multiplex

- Puzzle: Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads. In other words, no more than n threads can run in the critical section at the same time.

Multiplex: solution

- Initialize semaphore (multiplex) to n – the number of threads you wish to run concurrently.

Multiplex solution

```
multiplex.wait()
    critical section
multiplex.signal()
```

Applications of Semaphores

- Binary Semaphores
- Counting Semaphores
- Applications
  - Critical Section Problem
  - Deciding order of execution
  - For Managing Resources
    - e.g. 5 printers

Classical Problems of Synchronization: Bounded-Buffer Problem

- Shared data: **semaphore full, empty, mutex;**
- Initially: **full = 0, empty = n, mutex = 1**
- We have n buffers. Each buffer is capable of holding ONE item

Consumer

```
do {
    wait(full)
    wait(mutex);
    …
    remove an item from
    buffer to nextc
    …
    signal(mutex);
    signal(empty);
    …
  consume the item in nextc
  …
} while (1);
```

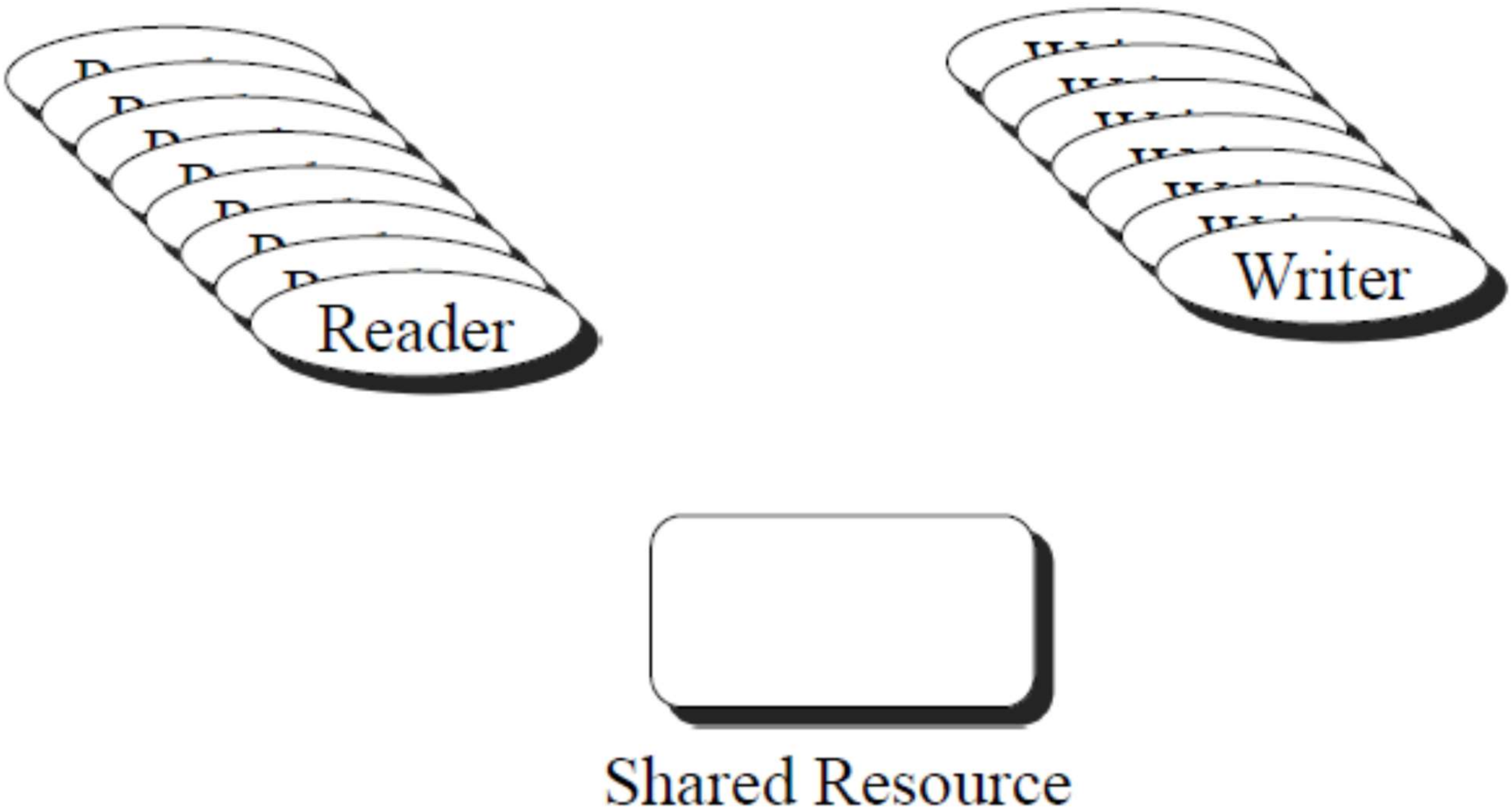
Producer

```
do {
    …
   produce an item in
    nextp
    …
    wait(empty);
    wait(mutex);
    …
   add nextp to buffer
    …
    signal(mutex);
    signal(full);
} while (1);
```

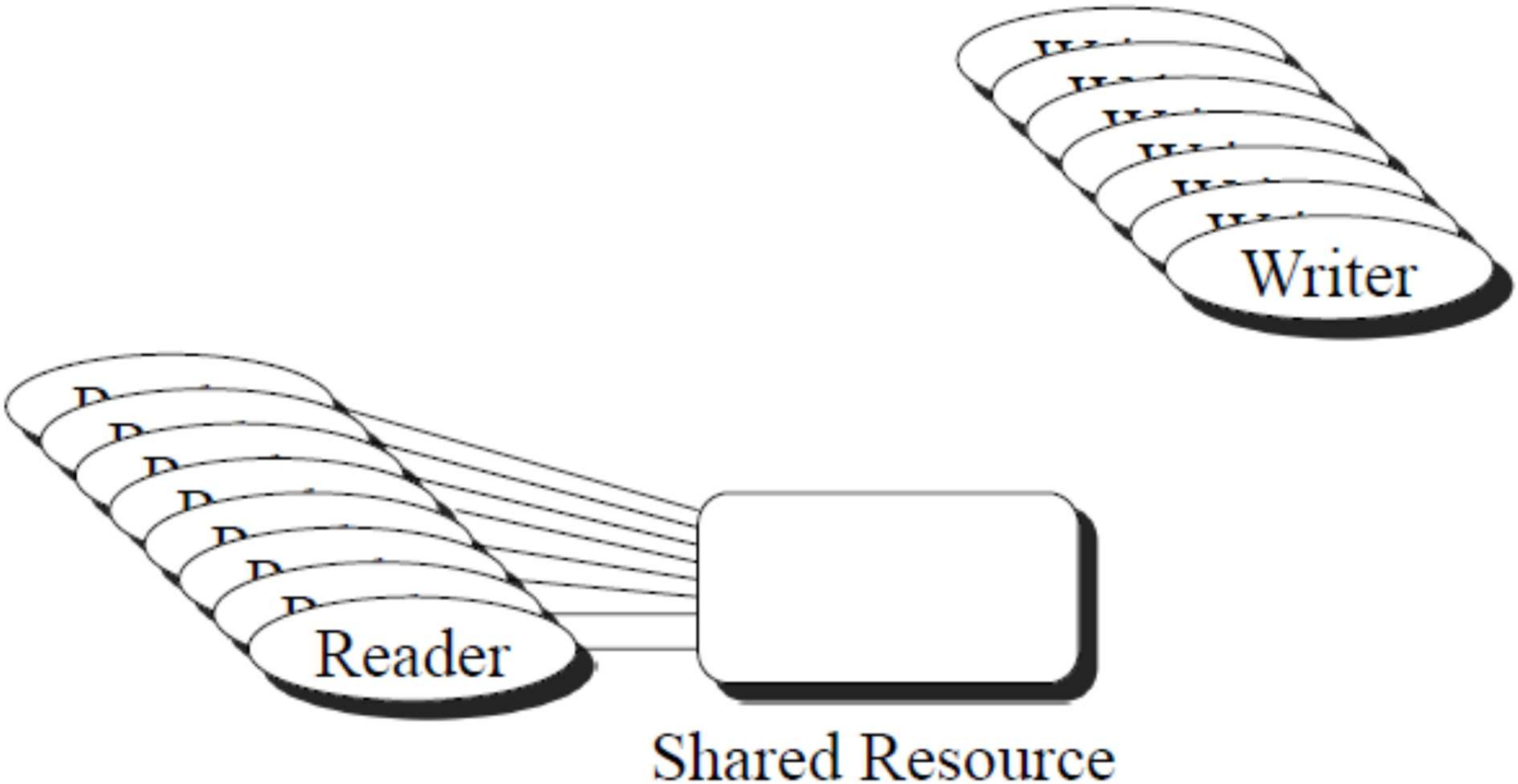
Classical Problems of Synchronization: Readers-Writers Problem

- There is one writer and multiple readers
- The writer wants to write to the database
- The readers wants to read from the database
- We can not allow a writer and a reader writing and reading the database at the same time
- We can allow one or more readers reading from the database at the same time
- Two different versions:
  - **First reader-writers problem**
  - **Second readers-writers problem**

Reader-writers problem (Cont.)

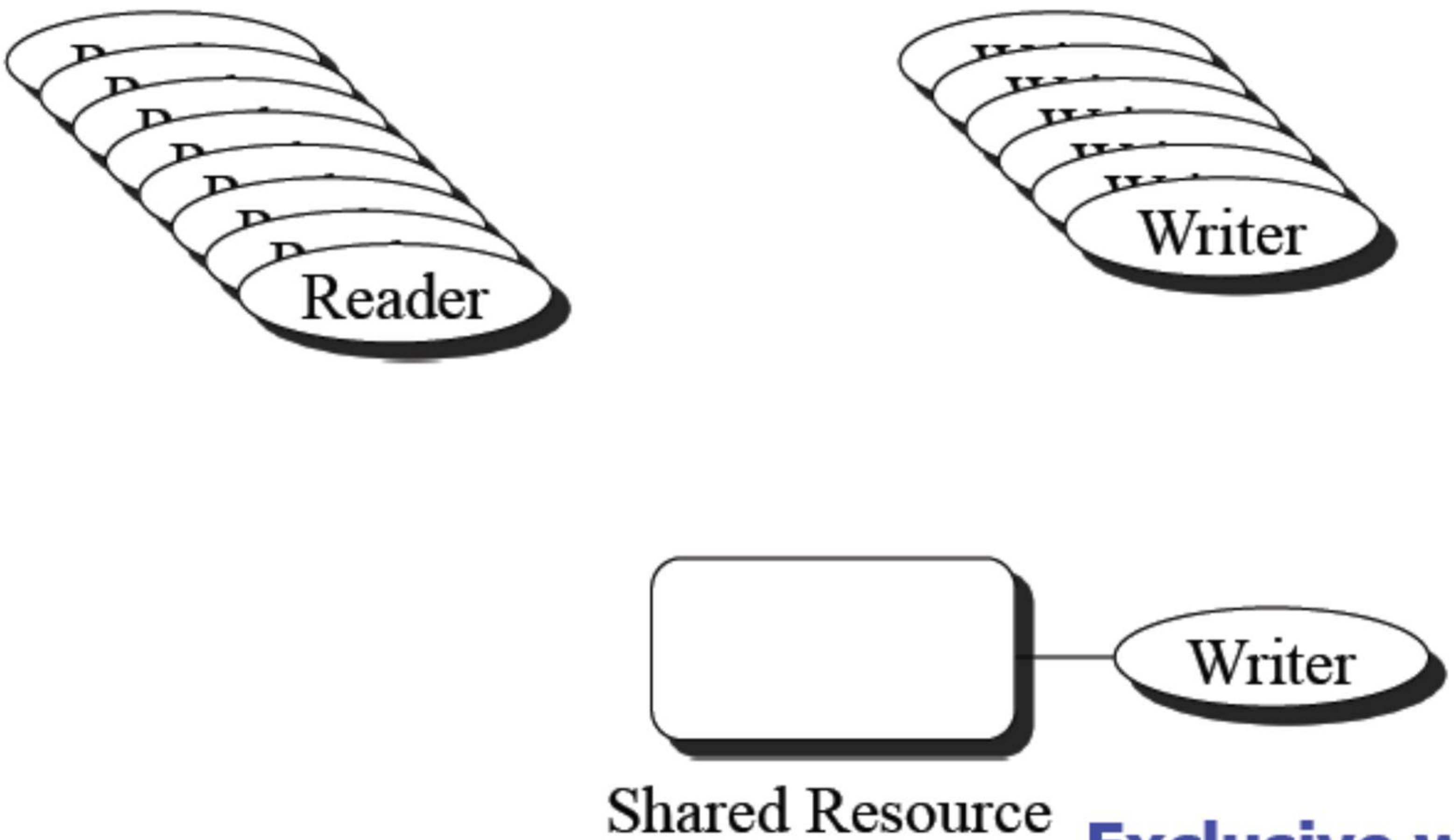


Reader-writers problem (Cont.)



Concurrent readers

Reader-writers problem (Cont.)



Exclusive writer

First Solution: Reader's precedence

```
Reader() {
    while(TRUE) {
        wait(mutex);
          readCount++;
          if(readCount==1)
              wait(wrt);
        signal(mutex);

        read(resource);

        wait(mutex);
          readCount--;
          if(readCount == 0)
              signal(wrt);
        signal(mutex);
}}
```

```
Writer() {
    while(TRUE) {
        wait(wrt);
          write(resource);
        signal(wrt);
    }
}
```

```
resourceType *resource;
int readCount = 0;
semaphore mutex = 1;
semaphore wrt = 1;
```

- First reader competes with writers
- Last reader signals writers

First Solution: reader's precedence

```
Reader() {
    while(TRUE) {
        wait(mutex);
          readCount++;
          if(readCount==1)
              wait(wrt);
        signal(mutex);
        read(resource);
        wait(mutex);
          readCount--;
          if(readCount == 0)
              signal(wrt);
        signal(mutex);
}}
```

```
Writer() {
    while(TRUE) {
        wait(wrt);
          write(resource);
        signal(wrt);
    }
}
```

- First reader competes with writers
- Last reader signals writers
- Any writer must wait for all readers
- Readers can starve writers
- "Updates" can be delayed forever

Second Solution: Writer's precedence

```
Reader() {
    while(TRUE) {
        wait(rd);
            wait(mutex1);
                readCount++;
                if(readCount == 1)
                    wait(wrt);
            signal(mutex1);
        signal(rd);
        read(resource);
      wait(mutex1);
        readCount--;
        if(readCount == 0)
            signal(wrt);
       signal(mutex1);
    }
}
```

```
writer() {
    while(TRUE) {
        wait(mutex2);
            writeCount++;
            if(writeCount == 1)
                wait(rd);
        signal(mutex2);
        wait(wrt);
            write(resource);
        signal(wrt);
        wait(mutex2)
            writeCount--;
            if(writeCount == 0)
                signal(rd);
        signal(mutex2);
```

int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore rd = 1, wrt = 1;

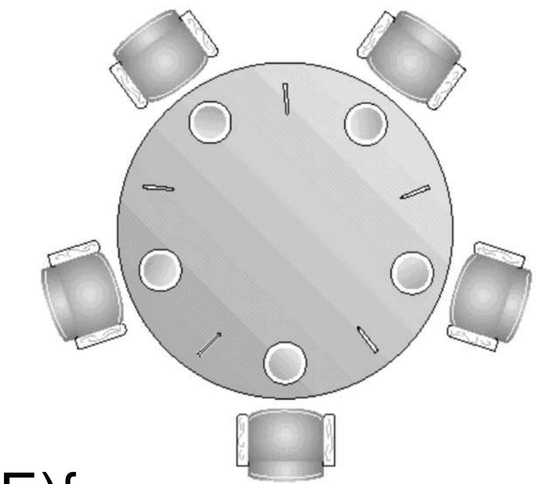
The Dining Philosophers Problem

- A classical synchronization problem
- 5 philosophers who only eat and think
- Each need to use 2 forks for eating
- There are only 5 forks
- Illustrates the difficulty of allocating resources among process without deadlock and starvation

The Dining Philosopher Problem

- Each philosopher is a process
- One semaphore per fork:
  - Fork: array[0..4] of semaphores
  - Initialization:

    fork[i].count:=1 for i:=0..4
- A first attempt:
  - Deadlock if each philosopher starts by picking his left fork!

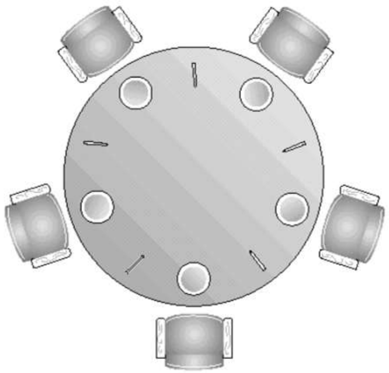


```
Pi() {
  while(TRUE){
    think;
    wait(fork[i]);
      wait(fork[i+1 mod 5]);
        eat;
      signal(fork[i+1 mod 5]);
    signal(fork[i]);
  }
}
```

The Dining Philosophers Problem

- Idea: admit only 4
- philosophers at a time who try to eat
- Then, one philosopher can always eat when the other 3 are holding one fork
- Solution: use another semaphore T to limit at 4 the number of philosophers "sitting at the table"
- Initialize: T.count:=4

```
Pi(){
  while(TRUE){
    think;
    wait(T);
      wait(fork[i]);
        wait(fork[i+1 mod 5]);
          eat;
        signal(fork[i+1 mod 5]);
      signal(fork[i]);
    signal(T);
  }
}
```



Recall: Problems with Semaphores

- Semaphores are a powerful tool for enforcing mutual exclusion and coordinate processes
- Problem: wait(S) and signal(S) are scattered among several processes
  - It is difficult to understand their effects
  - Usage must be correct in all processes
  - One bad (or malicious) process can fail the entire collection of processes

References

- Operating System Concepts (Silberschatz, 8th edition) Chapter 6