Operating Systems

Process Synchronization

Slides Courtesy:

Dr. Syed Mansoor Sarwar

Agenda for Today

- Process synchronization
- Recap of lecture

Process Synchronization

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure that cooperating processes access shared data sequentially.

Bounded-Buffer Problem

Shared data

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
    } item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0;
int counter = 0;
```

Bounded-Buffer Problem

Producer process

```
item nextProduced;
…
while (1) {
    while (counter == BUFFER_SIZE)  ;
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Bounded-Buffer Problem

Consumer process

```
item nextConsumed;
while (1) {
    while (counter == 0)  ;
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Bounded-Buffer Problem

- "`counter++`" in assembly language

```
MOV  R1, counter
INC   R1
MOV  counter, R1
```

- "`counter--`" in assembly language

```
MOV  R2, counter
DEC  R2
MOV  counter, R2
```

Bounded-Buffer Problem

- If both the producer and consumer attempt to update the buffer concurrently, the machine language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded-Buffer Problem

- Assume `counter` is initially 5. One interleaving of statements is:

```
producer:   MOV  R1, counter   (R1 = 5)
            INC  R1            (R1 = 6)
consumer:   MOV  R2, counter   (R2 = 5)
            DEC  R2            (R2 = 4)
producer:   MOV  counter, R1   (counter = 6)
consumer:   MOV  counter, R2   (counter = 4)
```

- The value of count may be either 4 or 6, where the correct result should be 5.

Process Synchronization

- **Race Condition**: The situation where several processes access and manipulate shared data concurrently, the final value of the data depends on which process finishes last.

Process Synchronization

- **Critical Section**: A piece of code in a cooperating process in which the process may updates shared data (variable, file, database, etc.).
- **Critical Section Problem**: Serialize executions of critical sections in cooperating processes

Process Synchronization

- **More Examples**
  - Bank transactions
  - Airline reservation

Bank Transactions

Deposit

MOV A, Balance

ADD A, Deposited

MOV Balance, A

Withdrawal

MOV B, Balance

SUB B, Withdrawn

MOV Balance, B

Bank Transactions

- **Bank Transactions**
  - Current balance = Rs. 50,000
  - Check deposited = Rs. 10,000
  - ATM withdrawn = Rs. 5,000

Bank Transactions

Check Deposit:

```
MOV A, Balance          // A = 50,000
ADD  A, Deposit ed      // A = 60,000
```

ATM Withdrawal:

```
MOV B, Balance          // B = 50,000
SUB  B, Withdrawn       // B = 45,000
```

Check Deposit:

```
MOV Balance, A          // Balance = 60,000
```

ATM Withdrawal:

```
MOV Balance, B          // Balance = 45,000
```

Solution of the Critical Problem

- Software based solutions
- Hardware based solutions
- Operating system based solution

Solution of the Critical Problem

- Software based solutions
- Hardware based solutions
- Operating system based solution

Structure of Solution

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (1);
```

Solution to Critical-Section Problem

- **2-Process Critical Section Problem**
- **N-Process Critical Section Problem**
- **Conditions for a good solution**:

1. **Mutual Exclusion**: If a process is executing in its critical section, then no other processes can be executing in their critical sections.

Solution to Critical-Section Problem

2. **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can decide which process will enter its critical section next, and this decision cannot be postponed indefinitely.

Solution to Critical-Section Problem

3. **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.

Solution to Critical-Section Problem

- Assumptions
  - Assume that each process executes at a nonzero speed
  - No assumption can be made regarding the relative speeds of the N processes.

Possible Solutions

- Only 2 processes, P0 and P1
- Processes may share some common variables to synchronize their actions.
- General structure of process Pi

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (1);
```

Algorithm 1

- Shared variables:
  - int turn;
    initially turn = 0
  - turn = i $\Rightarrow$ Pi can enter its critical section

Algorithm 1

- **Process Pi**

```
do {
        while (turn != i)
                ;
            critical section
        turn = j;
            remainder section
} while (1);
```

Algorithm 2

Shared variables

- boolean flag[2]; // Set to false
- flag [i] = true $\Rightarrow$ Pi ready to enter its critical section

Algorithm 2

- **Process Pi**

```
do {
        flag[i] = true;
        while (flag[j])
              ;
              critical section
        flag[i] = false;
              remainder section
} while (1);
```

Peterson's Algorithm

- Combined shared variables of algorithms 1 and 2.
- boolean flag[2]; // Set to false
- int turn=0;

Peterson's Algorithm

- **Process Pi**

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j)  ;

                critical section

        flag[i] = false;

                remainder section

} while (1);
```

Peterson's Algorithm

- Meets all three requirements:
  - **Mutual Exclusion**: 'turn' can have one value at a given time (0 or 1)
  - **Bounded-waiting**: At most one entry by a process and then the second process enters into its CS

Peterson's Algorithm

- **Progress**: Exiting process sets its 'flag' to false … comes back quickly and set it to true again … but sets turn to the number of the other process

n-Process Critical Section Problem

- Consider a system of n processes (P_0, P_1 ... P_{n-1}).
- Each process has a segment of code called a critical section in which the process may change shared data.

n-Process Critical Section Problem

- When one process is executing its critical section, no other process is allowed to execute in its critical section.
- The critical section problem is to design a protocol to serialize executions of critical sections.

Bakery Algorithm

- By Leslie Lamport
- Before entering its critical section, process receives a ticket number. Holder of the smallest ticket number enters the critical section.
- If processes Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first.

Bakery Algorithm

- The ticket numbering scheme always generates numbers in the increasing order of enumeration; i.e., 1, 2, 3, 4, 5 ...

Bakery Algorithm

Notations

- (ticket #, process id #)

  (a,b) < (c,d) if a < c or
  if a == c and b < d

- max (a_0,…, a_{n-1}) is a number, k, such that k ≥ a_i for i = 0, …, n–1

Bakery Algorithm

Data Structures

- boolean choosing[n];
- int number[n];

These data structures are initialized to false and 0, respectively

Bakery Algorithm

Structure of P_i

```
do {
        choosing[i] = true;
        number[i] = max(number[0],
                        number[1], …,
                        number [n – 1]) + 1;
        choosing[i] = false;
```

Bakery Algorithm

```
for (j = 0; j < n; j++) {
   while (choosing[j]) ;
   while ( (number[j] != 0) &&
        ((number[j], j) < (number[i], i)) ) ;
}
```

Critical Section

Bakery Algorithm

```
    number[i] = 0;

        remainder section

} while (1);
```

Bakery Algorithm

Process	Number
P0	3
P1	0
P2	7
P3	4
P4	8

Bakery Algorithm

P0	P2	P3	P4
(3,0) < (3,0)	(3,0) < (7,2)	(3,0) < (4,3)	(3,0) < (8,4)
Number[1] = 0	Number[1] = 0	Number[1] = 0	Number[1] = 0
(7,2) < (3,0)	(7,2) < (7,2)	(7,2) < (4,3)	(7,2) < (8,4)
(4,3) < (3,0)	(4,3) < (7,2)	(4,3) < (4,3)	(4,3) < (8,4)
(8,4) < (3,0)	(8,4) < (7,2)	(8,4) < (4,3)	(8,4) < (8,4)

Bakery Algorithm

- P1 not interested to get into its critical section $\Rightarrow$ number[1] is 0
- P2, P3, and P4 wait for P0
- P0 gets into its CS, get out, and sets its number to 0
- P3 get into its CS and P2 and P4 wait for it to get out of its CS

Bakery Algorithm

- P2 gets into its CS and P4 waits for it to get out
- P4 gets into its CS
- Sequence of execution of processes:

<P0, P3, P2, P4>

Bakery Algorithm

- Meets all three requirements:
  - **Mutual Exclusion**:

    (number[j], j) < (number[i], i) cannot be true for both P_i and P_j
  - **Bounded-waiting**: At most one entry by each process (n-1 processes) and then a requesting process enters its critical section (First-Come-First-Serve)

Bakery Algorithm

- **Progress**:
  - Decision takes complete execution of the ‘for loop’ by one process
  - No process in its ‘Remainder Section’ (with its number set to 0) participates in the decision making

Synchronization Hardware

- Normally, access to a memory location excludes other accesses to that same location.
- Extension: designers have proposed machine instructions that perform two operations atomically (indivisibly) on the same memory location (e.g., reading and writing).

Synchronization Hardware

- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).
- They can be used to provide mutual exclusion but other mechanisms are needed to satisfy the other two requirements of a good solution to the CS problem.

Test-And-Set (TSL) Instruction

- Test and modify a word atomically.

```
boolean TestAndSet(boolean &target)
{
        boolean rv = target;
        target = true;
        return rv;
}
```

Solution with TSL

- Structure for Pi ('lock' is set to false)

```
do {
        while ( TestAndSet(lock) )  ;

            Critical Section

        lock = false;

            Remainder Section
}
```

Solution with `TSL`

- Is the **`TSL`**-based solution good?

  No

- **Mutual Exclusion**: Satisfied
- **Progress**: Satisfied
- **Bounded Waiting**: Not satisfied

Swap Instruction

- Swaps two variables atomically

```
void swap (boolean &a, boolean &b)
{
        boolean temp = a;
        a = b;
        b = temp;
}
```

Solution with Swap

- Structure for Pi
- ‘key’ is local and set to false

```
do {
        key = true;
        while (key == true) swap(lock,key);

                Critical Section

        lock = false;

                Remainder Section
}
```

Solution with `swap`

- Is the **`swap`**-based solution good?

  No

- **Mutual Exclusion**: Satisfied
- **Progress**: Satisfied
- **Bounded Waiting**: Not satisfied

A Good Solution

- ‘key’ local; ‘lock’ and ‘waiting’ global
- All variables set to false

```
do {
        waiting[i] = true;
        key = true;
        while (waiting[i] && key)
            key = TestAndSet(lock);
        waiting[i] = false;

                Critical Section
```

A Good Solution

```
j = (i+1) % n;
while ( (j != i) && !waiting[j] )
        j = (j+1) % n;
if (j == i)
        lock = false;
else
        waiting[j] = false;

    Remainder Section
}
```

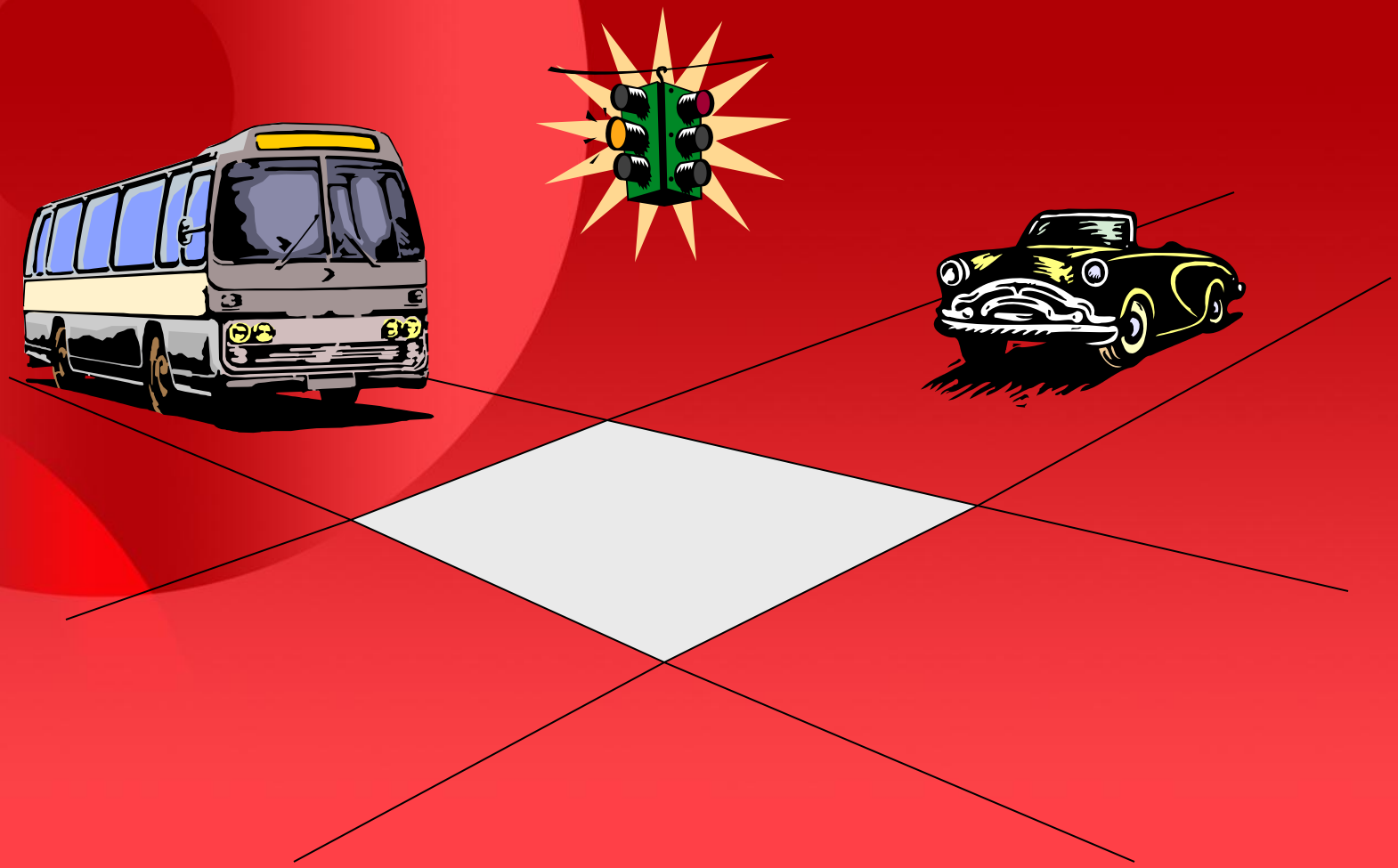
Solution with `Test-And-Set`

- Is the given solution good?

  Yes

- **Mutual Exclusion**: Satisfied
- **Progress**: Satisfied
- **Bounded Waiting**: Satisfied

Semaphores



Semaphores

- Synchronization tool
- Available in operating systems
- Semaphore `S` – integer variable that can only be accessed via two indivisible (atomic) operations, called `wait` and `signal`

Semaphores

```
wait(S){
  while S≤ 0
     ; //no-op
  S--;
}

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

n-Processes CS Problem

- Shared data:

  semaphore mutex = 1;

- Structure of Pi:

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

Is it a Good Solution?

- **Mutual Exclusion**: Yes
- **Progress**: Yes
- **Bounded Waiting**: No

Atomic Execution

- **Uni-Processor Environment**
  - Inhibit interrupts before executing code for `wait()` and `signal()`
- **Bus-based Multi-Processor Environment**
  - Lock the data bus
  - Use a software solution

Busy Waiting

- Processes wait by executing CPU instructions
- **Problem**? Wasted CPU cycles
- **Solution**? Modify the definition of semaphore

Semaphore Implementation

- Define a semaphore as a record

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

Semaphore Implementation

- Assume two simple operations:
  - `block()` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process P.

Semaphore Implementation

- The negative value of `S.value` indicates the number of processes waiting for the semaphore
- A pointer in the PCB needed to maintain a queue of processes waiting for a semaphore

Semaphore `wait()`

- Semaphore operations now defined as

```
void wait(semaphore S){
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block();
    }
}
```

Semaphore signal()

```
void signal(semaphore S){
   S.value++;
   if (S.value <= 0) {
      remove process P from S.L;
      wakeup(P);
   }
}
```

Two Implementations

- Busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Process Synchronization

Execute statement B in Pj only after statement A has been executed in Pi

- Use semaphore `S` initialized to 0

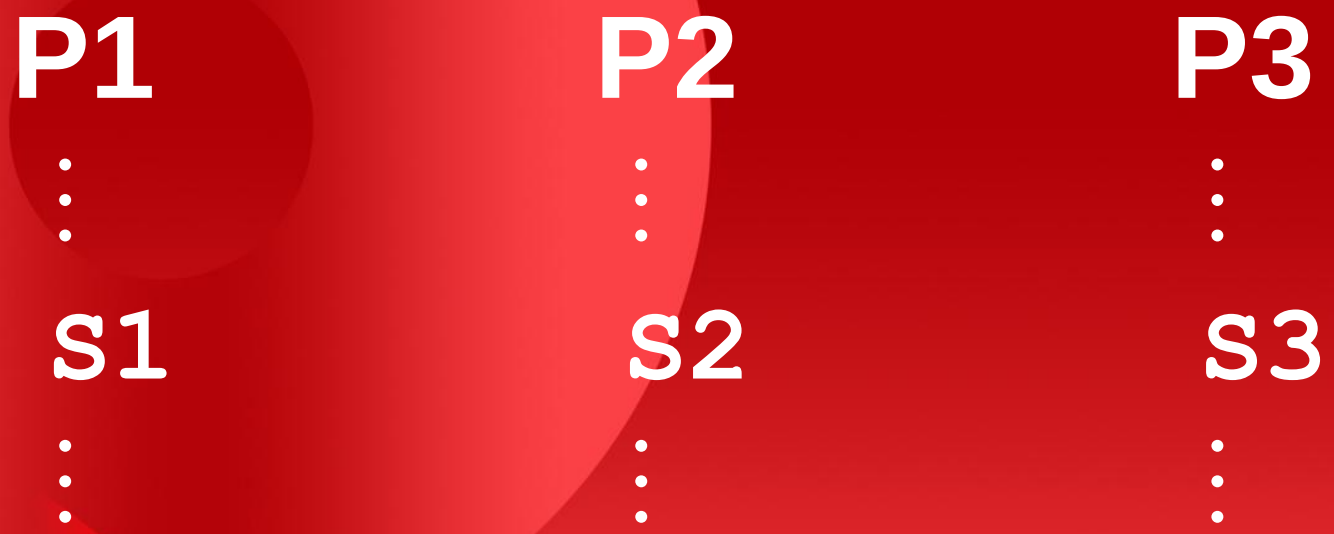
Pi	Pj
⋮	⋮
A	**wait(S)**
signal(S)	**B**

Process Synchronization

- Give a semaphore based solution for the following problem:

  Statement S1 in P1 executes only after statement S2 in P2 has executed, and statement S2 in P2 should execute only after statement S3 in P3 has executed.

Process Synchronization



Solution

Semaphore A=0, B=0;

P1	P2	P3
⋮	⋮	⋮
wait(A)	wait(B)	S3
S1	S2	signal(B)
	signal(A)	
⋮	⋮	⋮

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- But `wait(S)` and `signal(S)` are scattered among several processes. Hence, difficult to understand their effects.

Problems with Semaphores

- Usage must be correct in all the processes.
- One bad (or malicious) process can fail the entire collection of processes.

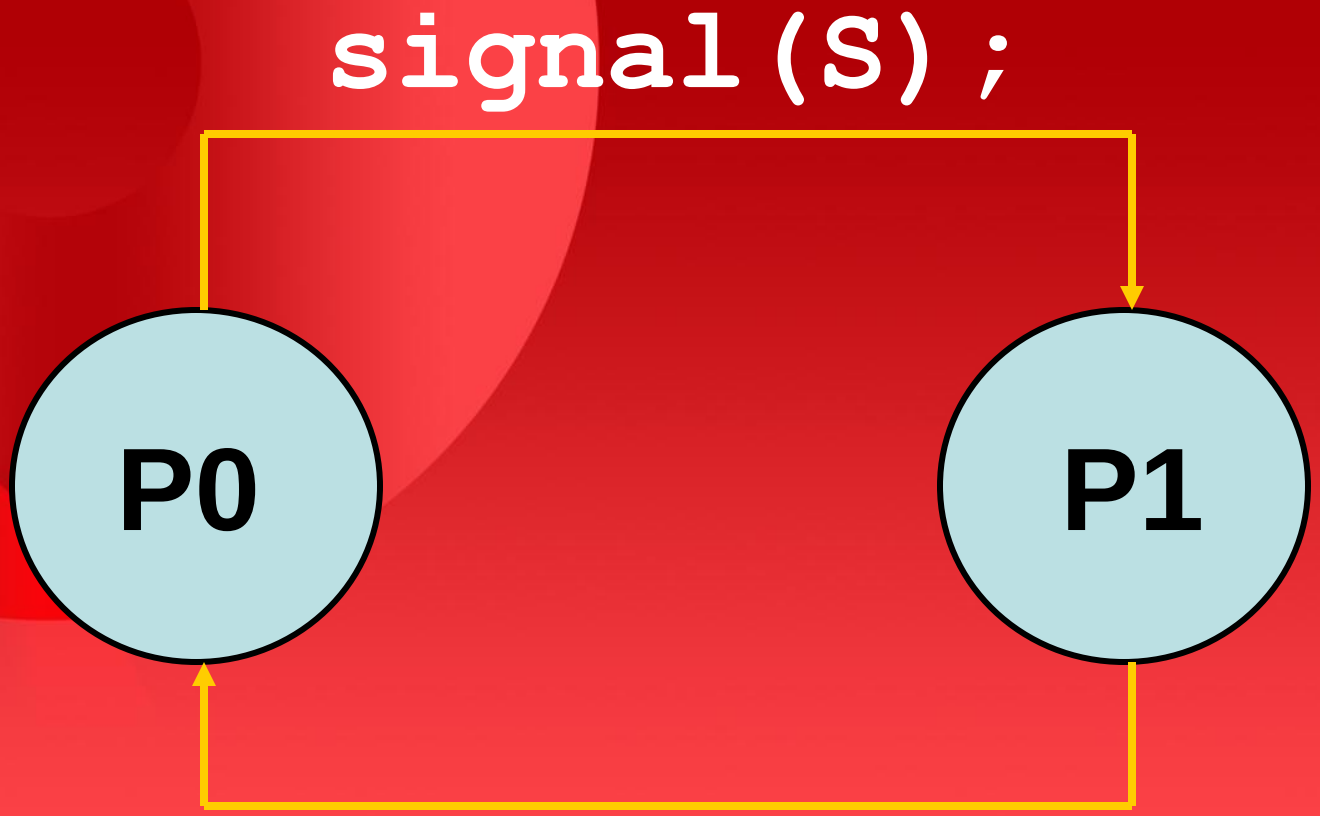
Deadlocks and Starvation

- A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set.
  - Traffic deadlocks
  - One-way bridge-crossing
- Starvation (infinite blocking) due to unavailability of resources

Deadlock

P0	P1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
⋮	⋮
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Deadlock



Starvation (Infinite Blocking)

P0	P1
`wait(S);`	`wait(S);`
⋮	⋮
`wait(S);`	`signal(S);`

Violation of Mutual Exclusion

P0	P1
`signal(S);`	`wait(S);`
⋮	⋮
`wait(S);`	`signal(S);`

Main Cause of Problem and Solution

- **Cause**: Programming errors due to the tandem use of `wait()` and `signal()` operations
- **Solution**: Higher-level language constructs such as critical region (region statement) and monitor.

Types of Semaphores

- **Counting semaphore** – integer value can range over an unrestricted integer domain.
- **Binary semaphore** – integer value cannot be > 1; can be simpler to implement.

Implementing a Counting Semaphore

- **Data structures**

  binary-semaphore `S1, S2`;

  int `C`;

- **Initialization**

  `S1 = 1`

  `S2 = 0`

  `C` = initial value of semaphore S

Implementing a Counting Semaphore

wait(S):

```
wait(S1);
C--;
if (C < 0) {
        signal(S1);
        wait(S2);
}
signal(S1);
```

Implementing a Counting Semaphore

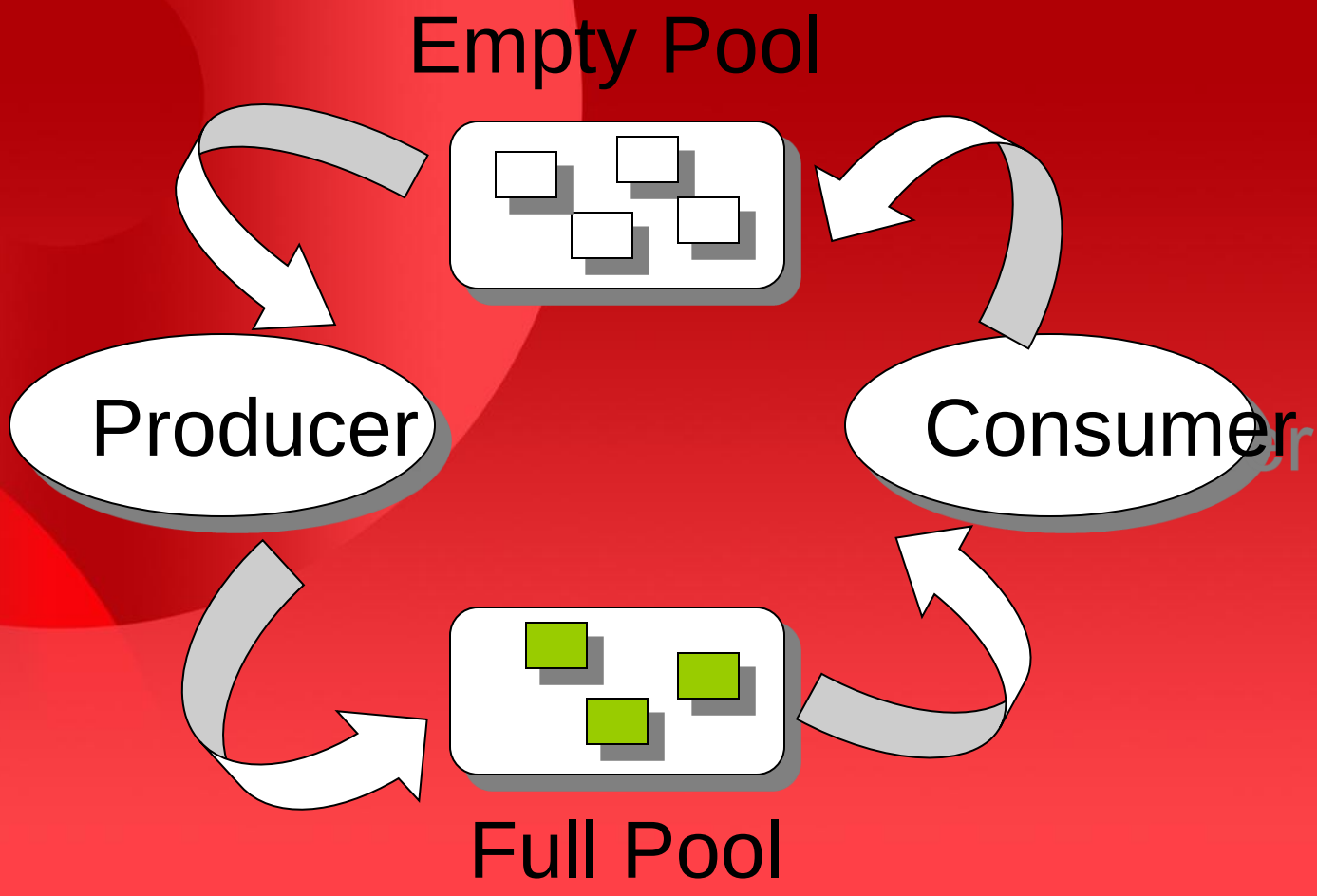
signal(S):

```
wait(S1);
C++;
if (C <= 0)
        signal(S2);
else
        signal(S1);
```

Classical Problems of Synchronization

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

Bounded Buffer Problem



Bounded-Buffer Problem

Shared data

semaphore full, empty, mutex;

Initialization

full = 0, empty = n, mutex = 1;

Producer Process

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

Consumer Process

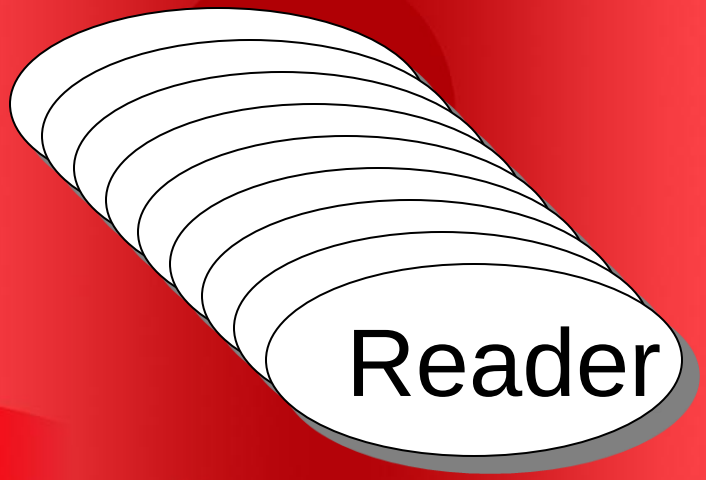
```
do {
    wait(full)
    wait(mutex);
        ...
    remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    consume the item in nextc
        ...
} while (1);
```

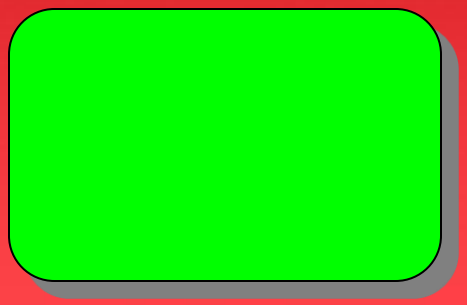
Readers and Writers Problem

Readers

Writers

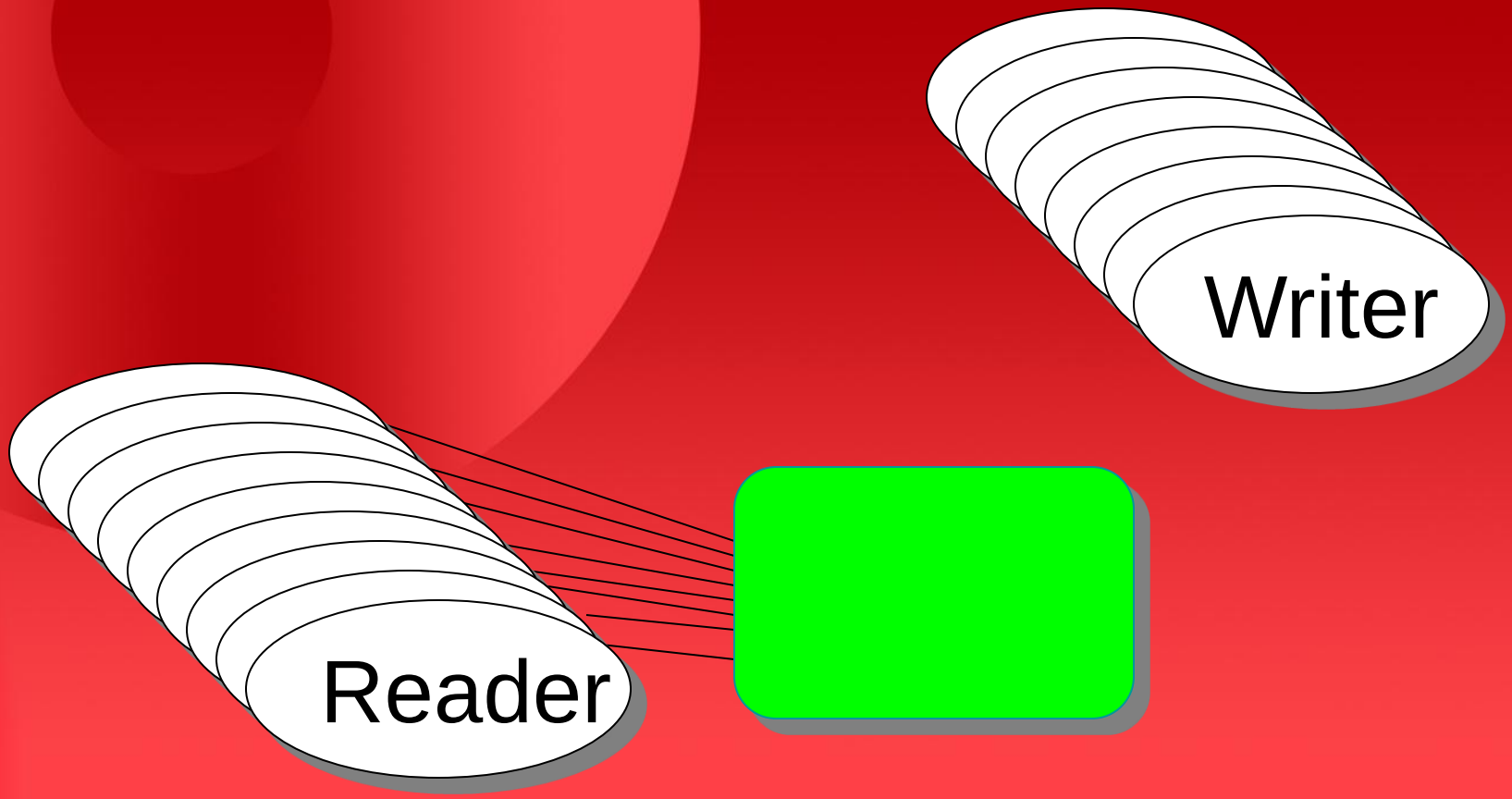
Readers and Writers Problem



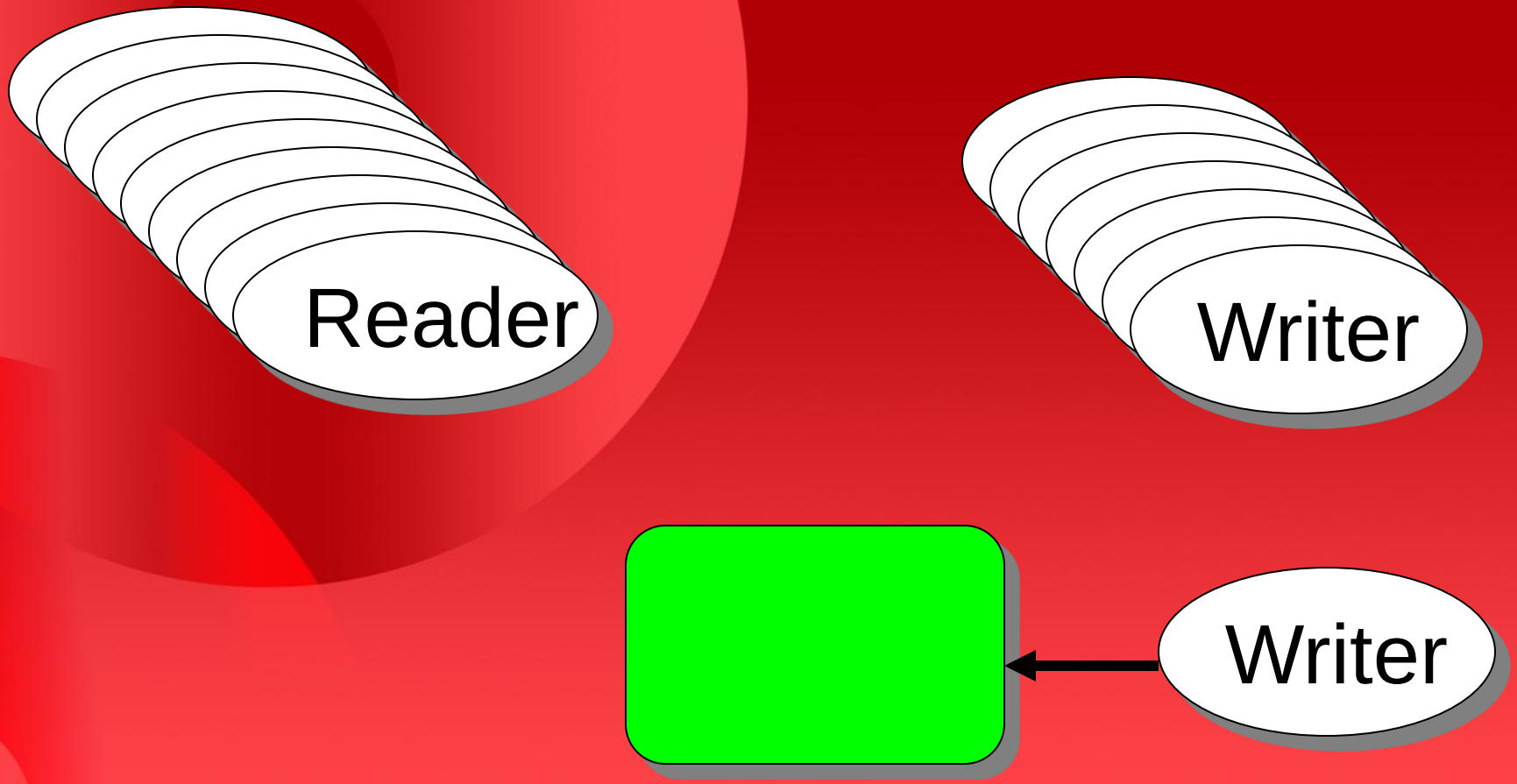


Shared Resource

Readers and Writers Problem



Readers and Writers Problem



First Readers and Writers Problem

No reader will be kept waiting unless a writer has already obtained permission to use the shared object.

Second Readers and Writers Problem

If a writer is ready, it waits for the minimum amount of time.

First Readers and Writers Problem

Shared data

semaphore mutex, wrt;

Initialization

mutex = 1, wrt = 1;
readcount = 0;

Writer Process

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

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);
```

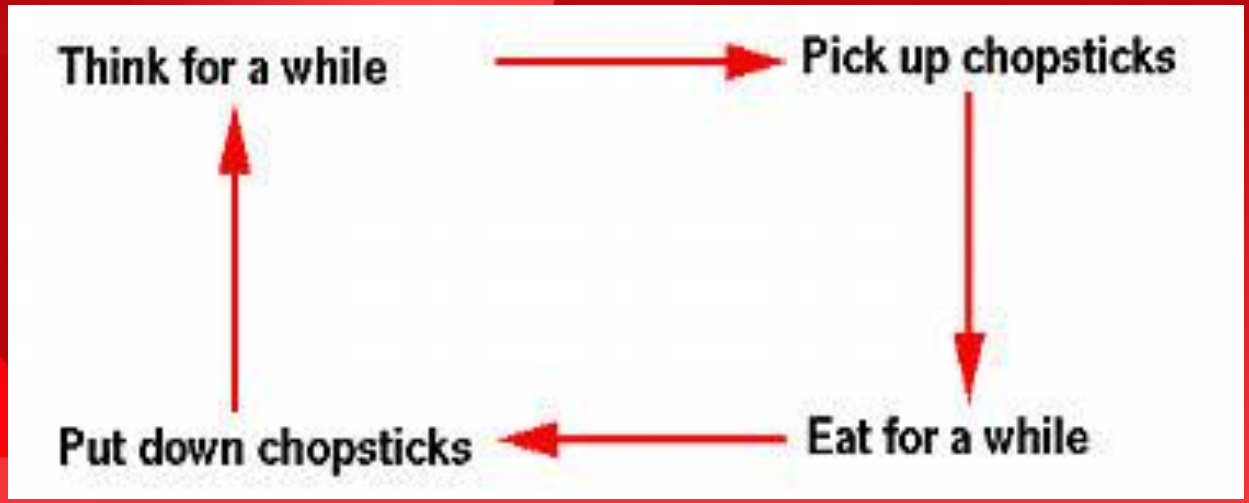
Dining Philosophers Problem

- Five philosophers who spend their lives just thinking and eating.
- Only five chopsticks are available to the philosophers

Dining Philosophers Problem

- Each philosopher thinks. When he becomes hungry, he sits down and picks up the two chopsticks that are closest to him and eats.
- After a philosopher finishes eating, he puts down the chopsticks and starts to think.

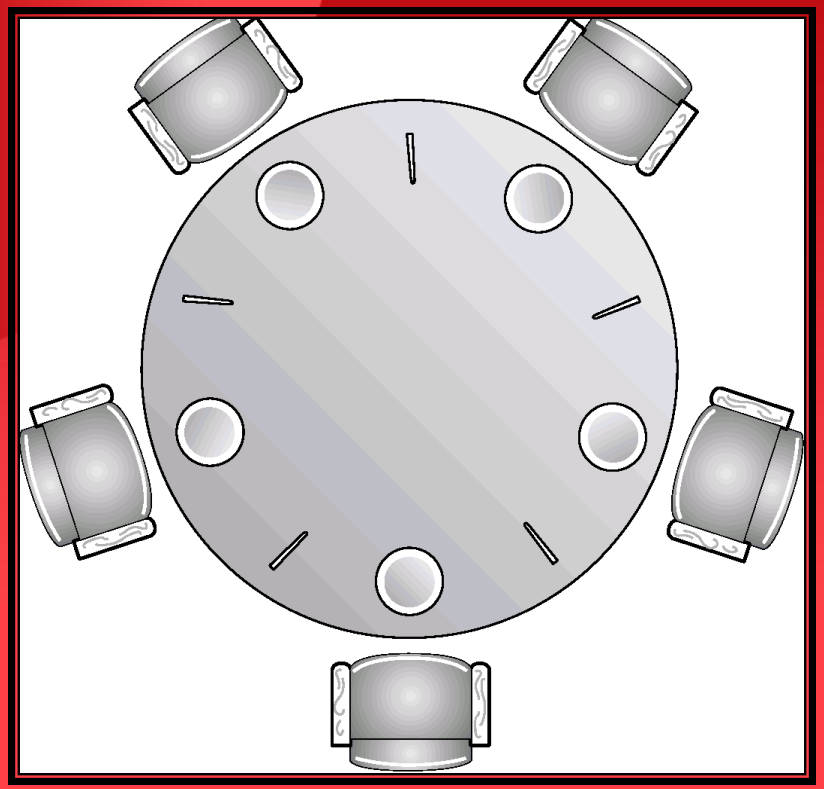
Dining-Philosophers Problem



Dining-Philosophers Problem

Shared data : semaphore chopstick[5];

// Initialized to 1



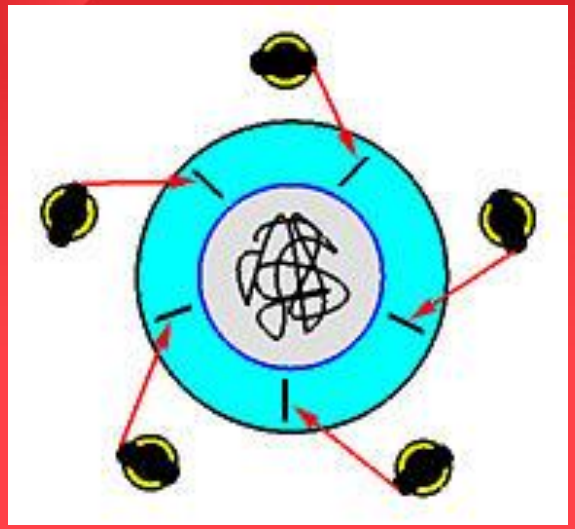
Dining-Philosophers Problem

Philosopher i

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

Possibility of Deadlock

If all philosophers become hungry at the same time and pick up their left chopstick, a deadlock occurs.



Possible Solutions

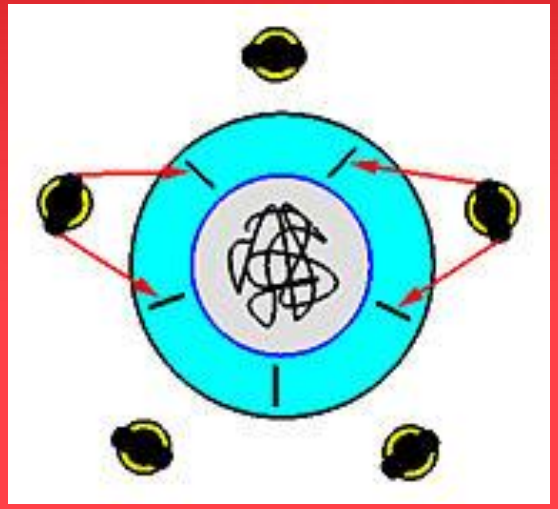
- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his/her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)

Possible Solutions

Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Possibility of Starvation

- Two philosophers who are fast eaters and fast thinkers, and lock the chopsticks before others every time.



Recap of Lecture

- Process synchronization
- Recap of lecture

Operating Systems

Process Synchronization