CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

University Institute of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Bachelor of Engineering (Computer Science & Engineering)

Subject Name : Operating System

Chapter : Process Synchronization

By: Er. Inderjeet Singh(e8822)

DISCOVER . LEARN . EMPOWER

Outline

- Process Synchronization
- Race Condition
- Critical Section Problem
- Peterson’s solution
- Mutex Lock
- Semaphores
- Classical Problem of Synchronization
- Monitors

Process Synchronization

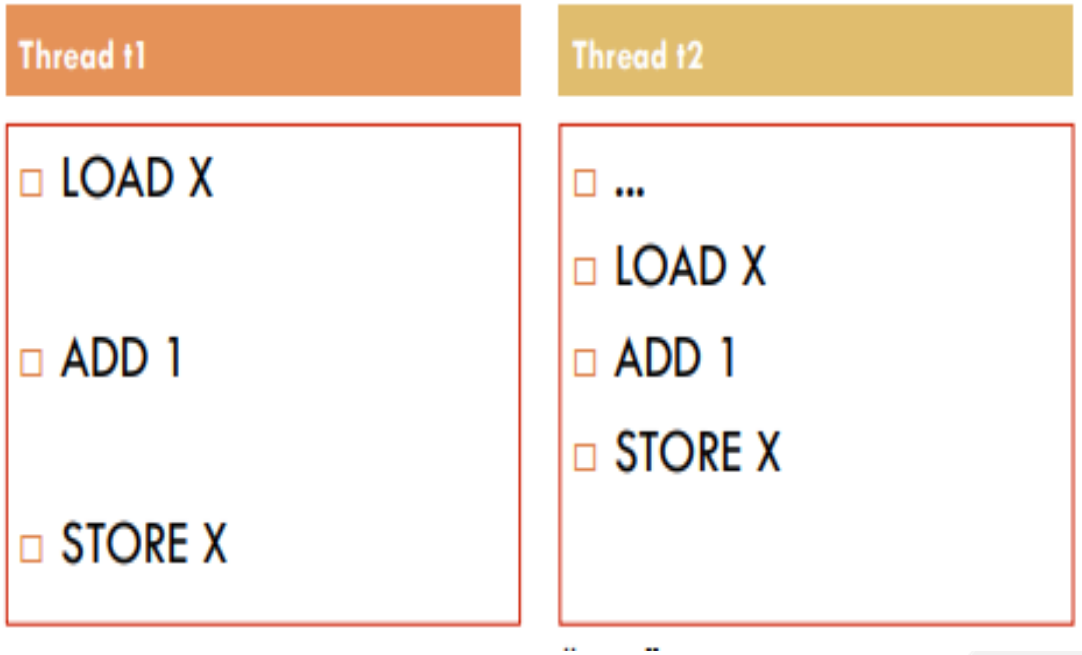
- When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.
- The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization.
- Sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Race Condition

- When more than one processes are executing the same code or accessing the same memory or any shared variable
- In that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing race to say that my output is correct. This condition known as **race condition**.
- Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.
- A Race Condition typically occurs when two or more threads try to read, write and possibly make the decisions based on the memory that they are accessing concurrently.
- For example, suppose t1 and t2 simulatenously execute the statement x = x + 1; for some static global x.
  - Internally, this involves loading x, adding 1, storing x .
  - If t1 and t2 do this concurrently, we execute the statement twice, but x may only be incremented once.
  - Here t1 and t2 “race” to do the update.

Race Condition-Example

- Suppose X is initially 5



- After finishing, X=6! We “lost” an update
- t1 and t2 “race” to do the update

Race Condition- Producer Consumer Example

- Suppose Counter is initially 5
- The producer and consumer processes concurrently execute the statements "counter++ " and "counter-- ".
- Following the execution of these two statements, the value of the variable counter maybe 4,5, or 6!

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

$$
\begin{array}{lllll}
T_0: & producer & \text{execute} & register_1 = \texttt{counter} & \{register_1 = 5\} \\
T_1: & producer & \text{execute} & register_1 = register_1 + 1 & \{register_1 = 6\} \\
T_2: & consumer & \text{execute} & register_2 = \texttt{counter} & \{register_2 = 5\} \\
T_3: & consumer & \text{execute} & register_2 = register_2 - 1 & \{register_2 = 4\} \\
T_4: & producer & \text{execute} & \texttt{counter} = register_1 & \{counter = 6\} \\
T_5: & consumer & \text{execute} & \texttt{counter} = register_2 & \{counter = 4\}
\end{array}
$$

Critical Section Problem

- Consider a system consisting of n processes {P0, P1,..., Pn−1}.
- Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The critical-section problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its **critical section**.
  - The section of code implementing this request is the **entry section.**
  - The critical section may be followed by an **exit section**.
  - The remaining code is the **remainder section**

Structure Critical Section Problem

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

Figure 5.1 General structure of a typical process P_i.

Critical Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

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

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 participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting**: There exists a bound, or limit, 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.

Exercise-1: Critical Section

Question: Consider the methods used by processes P_1 and P_2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S_1 and S_2 are randomly assigned.

Method used by P_1	Method used by P_2
while (S_1 == S_2); Critical Section S_1 = S_2	while (S_1 != S_2); Critical Section S_2 = !S_1

Which one of the following statements describes the properties achieved?

1. Mutual exclusion but not progress
2. Progress but not mutual exclusion
3. Neither mutual exclusion nor progress
4. Both mutual exclusion and progress

Solution-1

- The initial values of shared Boolean variables S_1 and S_2 are randomly assigned. The assigned values may be-

1. $S_1 = 0$ and $S_2 = 0$
2. $S_1 = 0$ and $S_2 = 1$
3. $S_1 = 1$ and $S_2 = 0$
4. $S_1 = 1$ and $S_2 = 1$

Solution-1-case-1

Method used by P_1	Method used by P_2
while (S_1 == S_2); Critical Section S_1 = S_2	while (S_1 != S_2); Critical Section S_2 = !S_1

- **Case-01: If S_1 = 0 and S_2 = 0-**
- Process P_1 will be trapped inside an infinite while loop.
- However, process P_2 gets the chance to execute.
- Process P_2 breaks the while loop condition, executes the critical section and then sets S_2 = 1.
- Now, S_1 = 0 and S_2 = 1.
- Now, process P_2 can not enter the critical section again but process P_1 can enter the critical section.
- Process P_1 breaks the while loop condition, executes the critical section and then sets S_1 = 1.
- Now, S_1 = 1 and S_2 = 1.
- Now, process P_1 can not enter the critical section again but process P_2 can enter the critical section.

Solution-1-case-1

Thus,

- Processes P1 and P2 executes the critical section alternately starting with process P_2.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.

Solution-1-case-2

- **Case-02: If $S_1 = 0$ and $S_2 = 1$-**
- Process P_2 will be trapped inside an infinite while loop.
- However, process P_1 gets the chance to execute.
- Process P_1 breaks the while loop condition, executes the critical section and then sets $S_1 = 1$.
- Now, $S_1 = 1$ and $S_2 = 1$.
- Now, process P_1 can not enter the critical section again but process P_2 can enter the critical section.
- Process P_2 breaks the while loop condition, executes the critical section and then sets $S_2 = 0$.
- Now, $S_1 = 1$ and $S_2 = 0$.
- Now, process P_2 can not enter the critical section again but process P_1 can enter the critical section.

Solution-1-case-2

Thus,

- Processes P1 and P2 executes the critical section alternately starting with process P_1.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.

Solution-1-case-3 and 4

- **Case-03: If $S_1 = 1$ and $S_2 = 0$-**
- This case is same as case-02.
- **Case-04: If $S_1 = 1$ and $S_2 = 1$-**
- This case is same as case-01.

Solution-1

- Thus, Overall we can conclude-
- Processes P1 and P2 executes the critical section alternatively.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.

Peterson's Solution

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process; that is, j equals 1−i.
- Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

- The variable **turn** indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
- The **flag array** is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

Peterson's Solution

A classic software-based solution to the critical-section problem known as Peterson's solution.

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

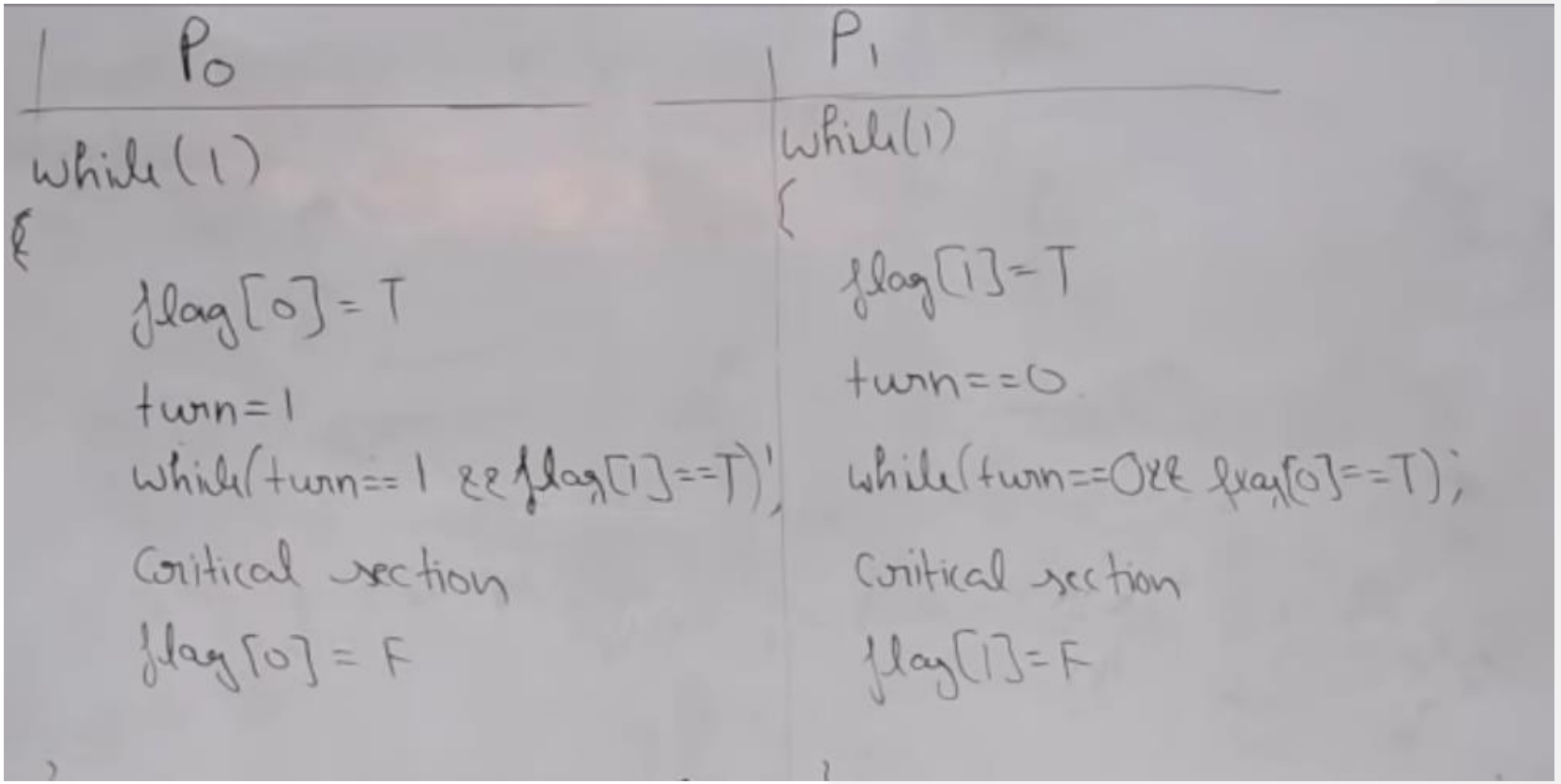
Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson's Solution

- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, there by asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

Peterson's Solution

Mutual Exclusion is there Because only one process execute in Critical Section



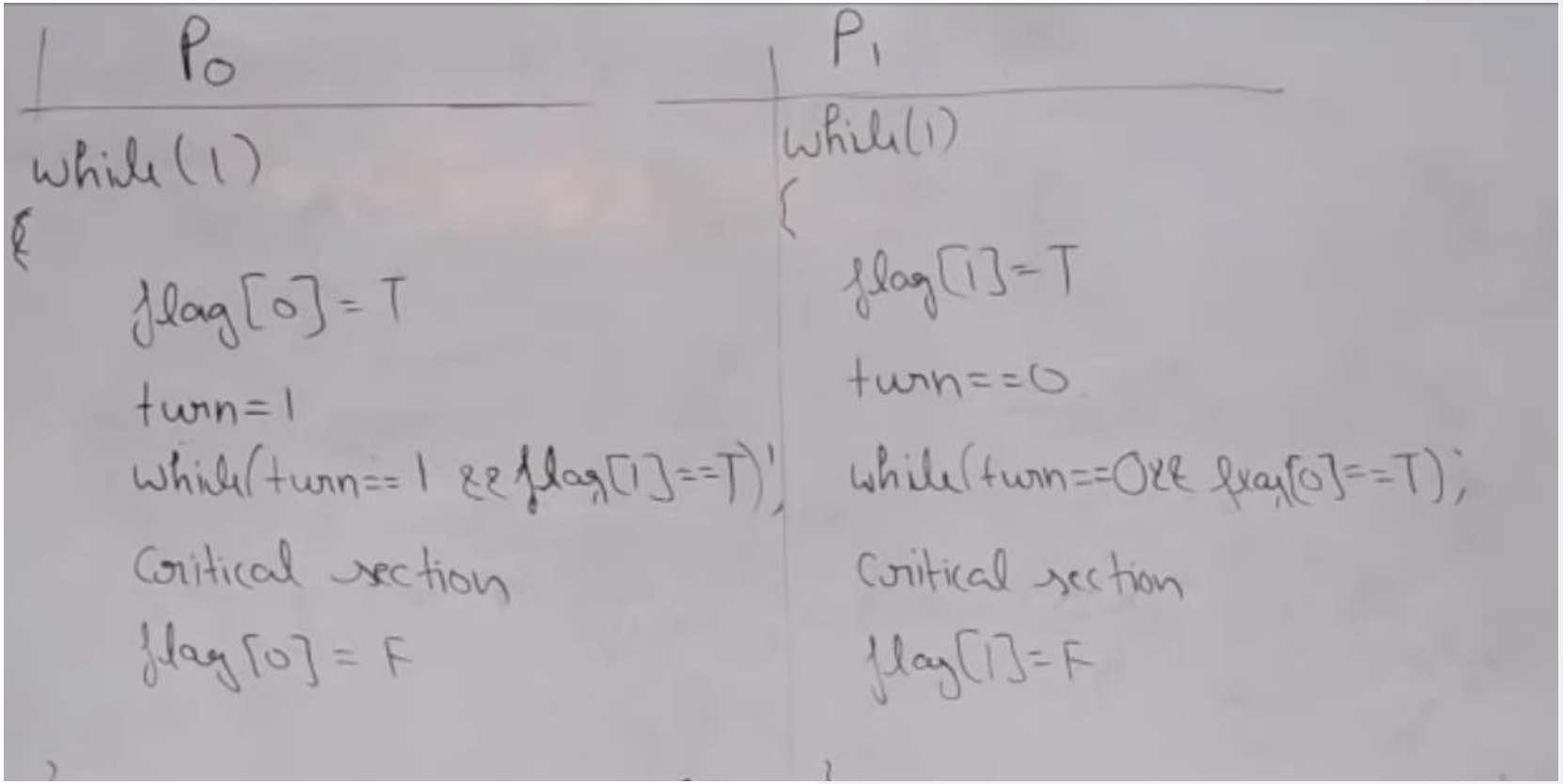
Turn= 0 or 1

Flag=

False	**False**

Peterson's Solution

Progress is there Because P0 can enter critical section as many number of times



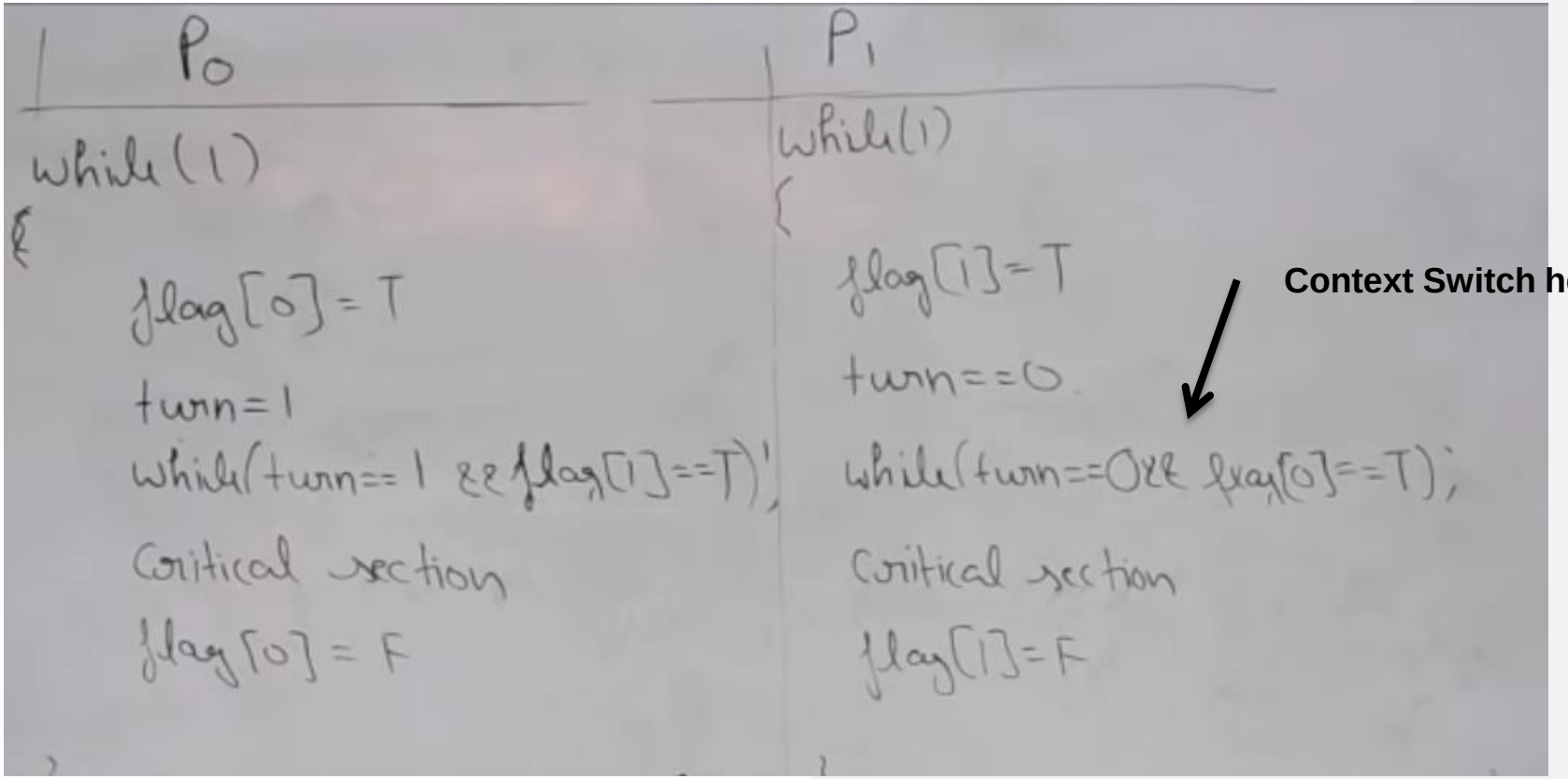
Turn= 1

Flag=

True	False

Peterson's Solution

If both process wishes to enter critical section at same time , no deadlock occurs



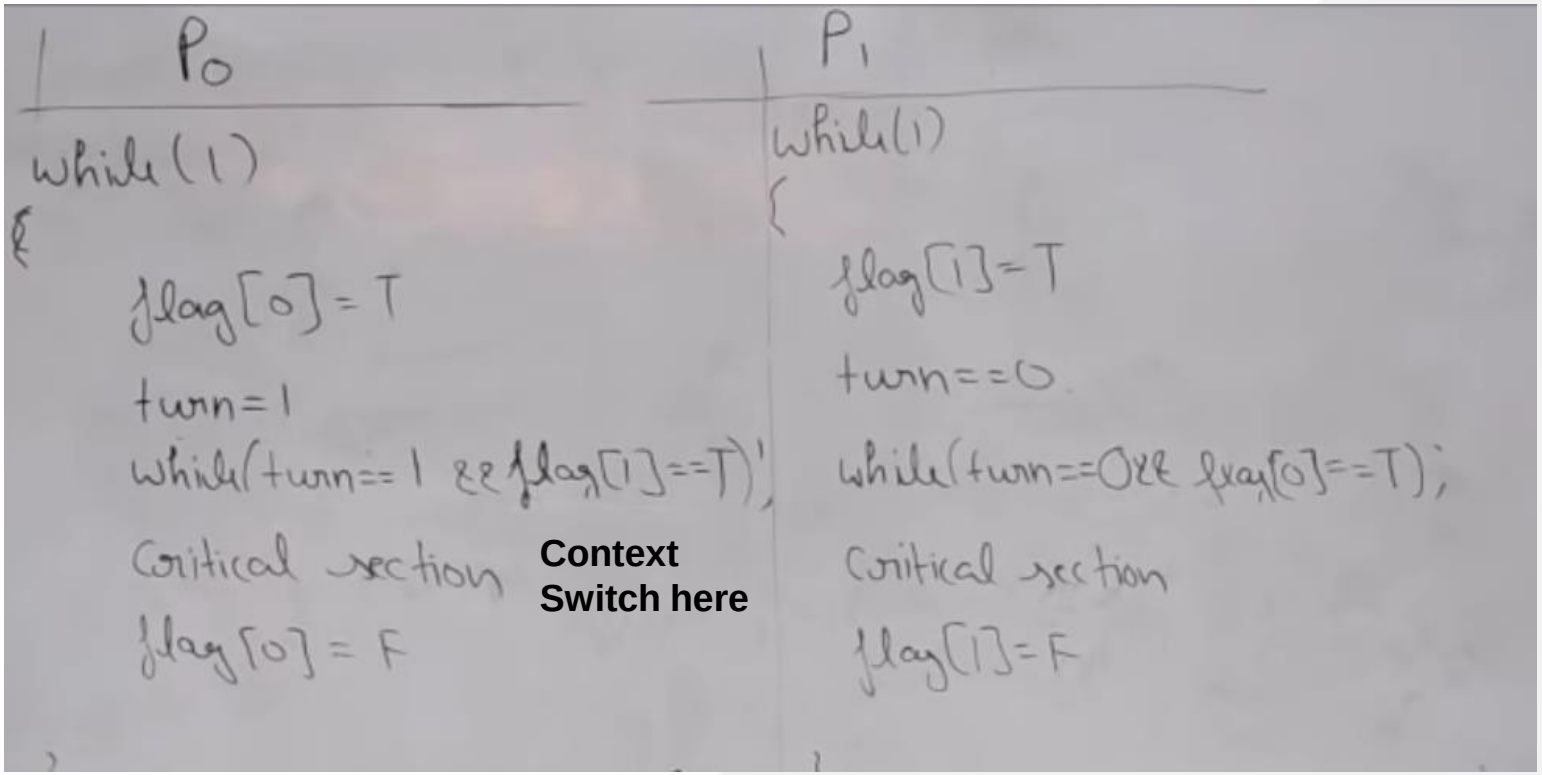
Turn= 0

Flag=

False	True

Peterson's Solution

Bounding waiting condition satisfied because Once P0 enter Critical Section and then P1 interested to go to critical section but p1 will not be able to enter, because Already p0 is in critical section. After completion, P0 will not enter critical section because of P1 i.e. bounded to wait



Turn= 1

Flag=

True	False

Mutex locks

- The simplest of software tools to solve the critical-section problem is the mutex lock. (the term mutex is short for mutual exclusion).
- A process acquires the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire() function acquires the lock, and the release() function releases the lock.

The definition of `release()` is as follows:

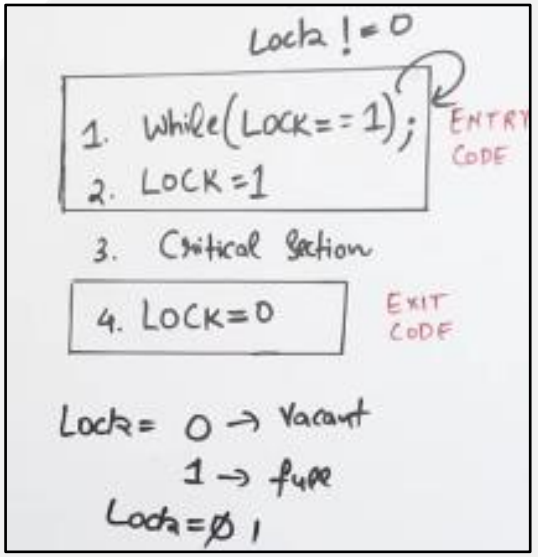
```
release() {
   available = true;
}
```

Mutex locks

- The main disadvantage of the implementation given here is that it requires **busy waiting.** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire() .
- In fact, this type of mutex lock is also called a spinlock because the process "spins" while waiting for the lock to become available.

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (true);
```

Figure 5.8 Solution to the critical-section problem using mutex locks.



Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()and signal().**
- All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- The **wait()** operation was originally termed **P** (from the Dutch proberen, "to test"); **signal()** was originally called **V** (from verhogen,"to increment "). The definition of wait()is as follows:

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

The definition of `signal()` is as follows:

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

Semaphore Usage

Semaphore to solve Synchronization problem

- For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1and P2 share a common semaphore synch, initialized to 0.

  In process P1, we insert the statements
  S1;
  signal(synch);
  In process P2, we insert the statements
  wait(synch);
  S2;

- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Semaphore Implementation

- Semaphores are defined as follows:

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

- Each semaphore has an integer value and a list of processes list.
- 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.

Semaphore Implementation

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows:

- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait.
- However, 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.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state and then placed in the ready queue.

Semaphore Implementation

- The wait() semaphore operation and the signal() signal operation can be defined as:

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

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

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- These two operations are provided by the operating system as basic system calls.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlock and Starvation in Semaphores

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked.**
- To illustrate this, consider a system consisting of two processes, P0 and P1,each accessing two semaphores, S and Q, set to the value 1:

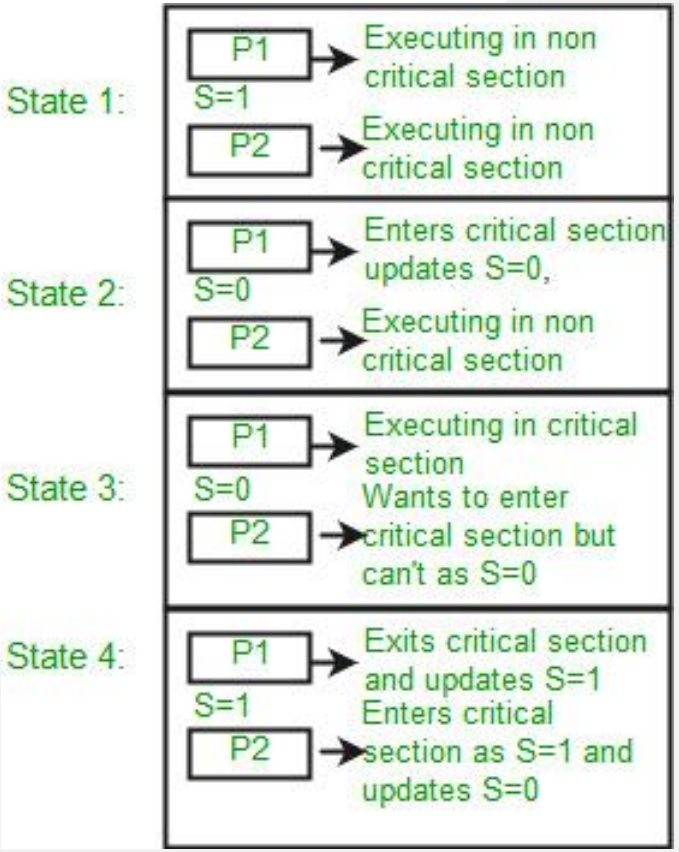
P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- Suppose that P0 executes wait(S) and then P1 executes wait(Q).When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, **P0 and P1 are deadlocked.**

Semaphores Usage

1. **Binary Semaphore:** The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved.



Semaphores Usage

2. **Counting Semaphores**

- The value of a **counting semaphore** can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

- Now suppose there is a resource whose number of instances is 4.
- Now we initialize S = 4 and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function.
- If the value of S becomes zero then a process has to wait until S becomes positive.
- For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.

Exercises-1 with Semaphores

Ques. A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution:

We know-

P operation also called as wait operation decrements the value of semaphore variable by 1.

V operation also called as signal operation increments the value of semaphore variable by 1.

Exercises-1 with Semaphores

Ques. A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution:

We know-

P operation also called as wait operation decrements the value of semaphore variable by 1.

V operation also called as signal operation increments the value of semaphore variable by 1.

Final value of semaphore S

= 10-(6*1)+(4*1)

= 10-6+4

=8

Exercise 2: Semaphores

Q: Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12V(S)operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _____

(A) 7

(B) 8

(C) 9

(D) 10

Exercise 2: Semaphores

Q: Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12V(S)operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _____

(A) 7(Correct)
(B) 8
(C) 9
(D) 10

Solution:

20-7 -> 13 will be in blocked state, when we perform 12 V(S) operation makes 12 more process to get chance for execution from blocked state. So one process will be left in the queue (blocked state) here i have considered that if a process is in under CS then it not get blocked by other process.

S-20+12= -1

Classical Problems of Synchronization

- Here, a number of classical problems of synchronization as examples of a large class of concurrency-control problems are discussed.
- In solutions to the problems, semaphores are used for synchronization, since that is the traditional way to present such solutions.
- However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

- **Problems are:**
  1. Bounded-buffer (or Producer-Consumer) Problem,
  2. Readers and Writers Problem,
  3. Dining-Philosophers Problem,

Bounded-buffer (or Producer-Consumer) Problem

- In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.
- The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The **producer**'s job is to generate data, put it into the buffer, and start again.
- At the same time, the **consumer** is consuming the data (i.e. removing it from the buffer), one piece at a time.

- **Problem :** To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Producer-Consumer Problem Solution using Semaphores

Problem : To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

- To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

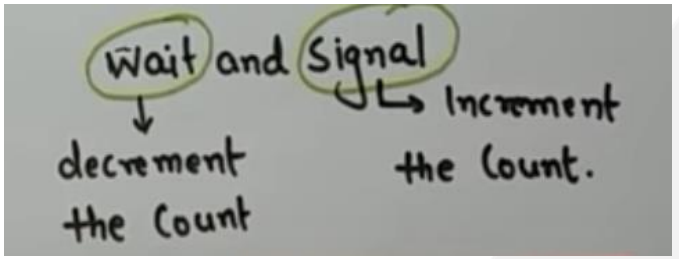
- **Initialization of semaphores –**
  mutex = 1
  Full = 0 // Initially, all slots are empty. Thus full slots are 0
  Empty = n // All slots are empty initially

- We assume that the pool consists of **n buffers**, each capable of holding one item.

Producer-Consumer Problem Solution using Semaphores

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```



- When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now.
- The value of mutex is also reduced to prevent consumer to access the buffer.
- Now, the producer has placed the item and thus the value of "full" is increased by 1.
- The value of mutex is also increased by 1 because the task of producer has been completed and consumer can access the buffer.

Producer-Consumer Problem Solution using Semaphores

```
do{

wait(full);
wait(mutex);

// remove item from buffer

signal(mutex);
signal(empty);

// consumes item

}while(true)
```

Solution for Consumer –

- As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment.
- Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Readers and Writers Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.

- Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

- **Problem:** To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem.

Readers and Writers Solution

- The readers–writers problem and its solutions have been generalized to provide reader–writer locks on some systems.
- Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access.
- When a process wishes only to read shared data, it requests the reader–writer lock in read mode.
- A process wishing to modify the shared data must request the lock in write mode.

- Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Readers and Writers Solution

- **The reader processes share the following data structures:**
  - semaphore rw mutex = 1;
  - semaphore mutex = 1;
  - int read count = 0;
- The semaphores **mutex** and **rw mutex** are initialized to 1; read count is initialized to 0.
- The semaphore **rw mutex** is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
- The read count variable keeps track of how many processes are currently reading the object.
- The semaphore rw mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

Readers and Writers Solution

- The code for the **writer** process:

```
do
{
wait(rw mutex);
...
/* writing is performed */
...
 signal(rw mutex);
} while (true);
```

Writer process:
Writer requests the entry to critical section.
If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
It exits the critical section.

Readers and Writers Solution

- The code for the **reader** process:

```
do
{
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex);
signal(mutex);
...
/* reading is performed */
...
wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```

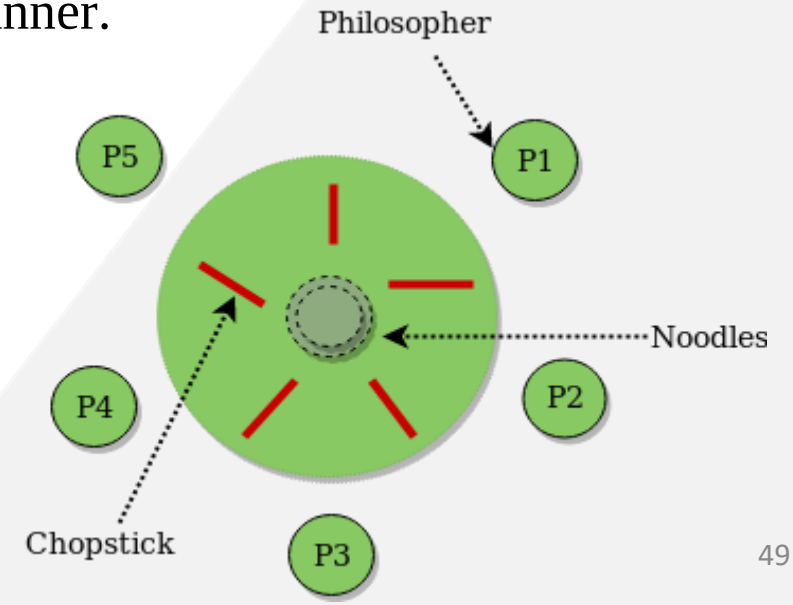
Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
   - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **rw mutex** semaphore to restrict the entry of writers if any reader is inside.
   - It then, signals mutex as any other reader is allowed to enter while others are already reading.
   - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “rw mutex” as now, writer can enter the critical section.

If not allowed, it keeps on waiting.

Dining Philosophers Problem

- The Dining Philosopher Problem states that K (here 5) philosophers seated around a circular table with one chopstick between each pair of philosophers.
- There is one chopstick between each philosopher.
- A philosopher may eat if he can pickup the two chopsticks adjacent to him.
- One chopstick may be picked up by any one of its adjacent followers but not both.
- This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



Dining Philosophers Solution

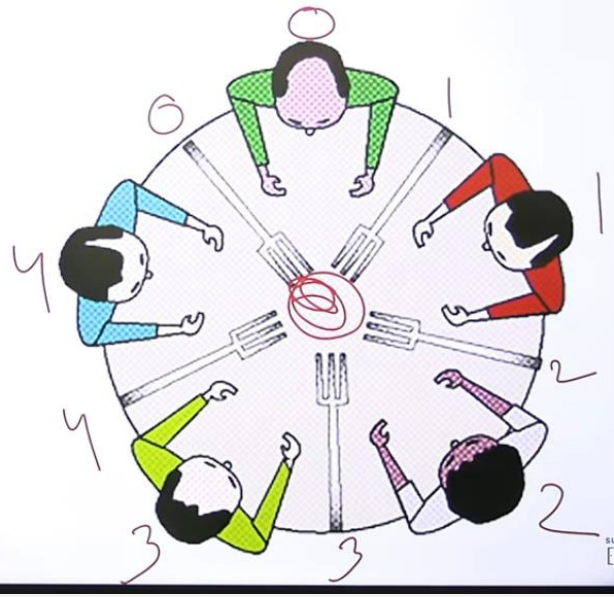
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- Thus, the shared data are: semaphore chopstick[5]; where all the elements of chopstick are initialized to 1.

Solution for Dining Philosophers

```
Void Philosopher (void)
{
        while ( T )
        {
                Thinking ( ) ;
                wait(chopstick [i]);
                wait(chopstick[(i+1)%5]);
                Eat( );
                signal(chopstick [i]);
                signal(chopstick[(i+1)%5]);
        }
}
```

Chopstick[5] Semaphore

1	1	1	1	1



Note: Deadlock might occur as every philosopher holds one chopstick at time

Monitors

Semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur.

For example:

Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

Monitors

Case 2: Suppose that a process replaces signal(mutex) with wait(mutex) .That is, it executes

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

Definition: Monitors

- Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.
- Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.
- Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Figure 5.15 Syntax of a monitor.

Definition: Monitors

- **Condition Variables**

There are two sorts of operations we can perform on the monitor's condition variables:

- Wait
- Signal

Consider a condition variable (y) is declared in the monitor:

- **y.wait():** The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.
- **y.signal():** If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

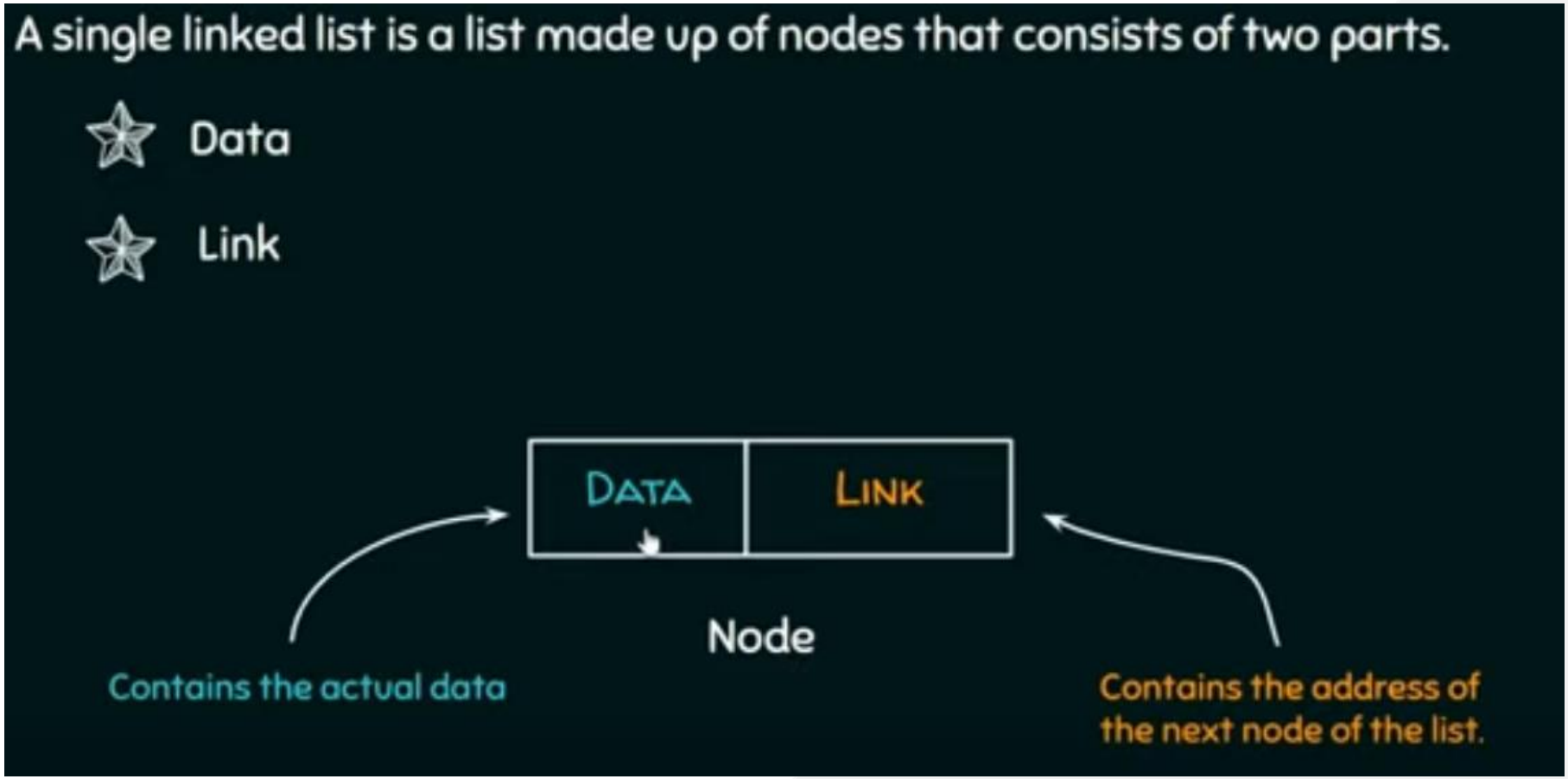
Monitors v/s Semaphores

Features	Semaphore	Monitor
Definition	A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS.	It is a synchronization process that enables threads to have mutual exclusion and the wait() for a given condition to become true.

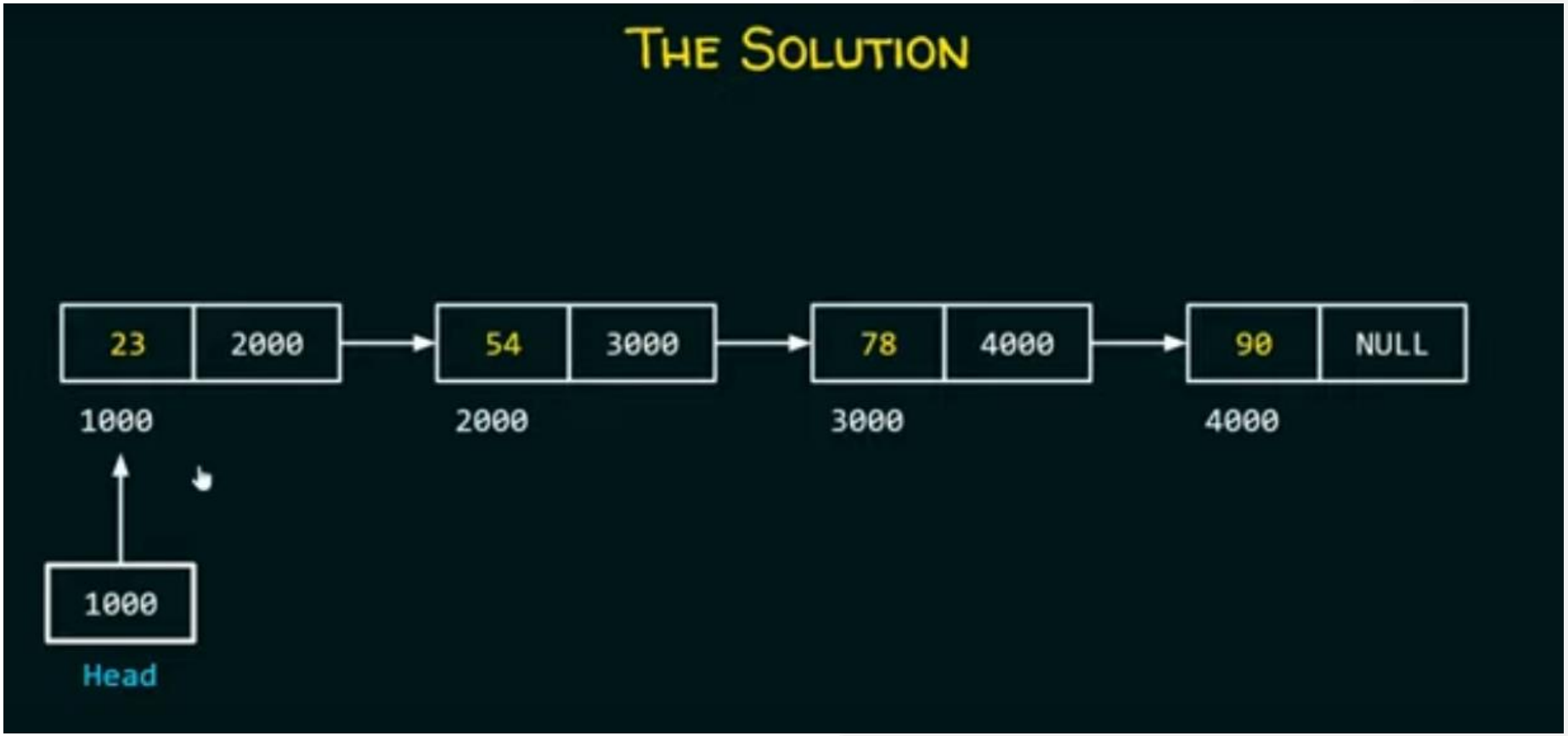
Syntax	// Wait Operation
	wait(Semaphore S) {
	while (S<=0);
	S--;
	}
	// Signal Operation
	signal(Semaphore S) {
	S++;
	}
		monitor {
		//shared variable declarations
		data variables;
		Procedure P1() { ... }
		Procedure P2() { ... }
		.
		.
		.
		Procedure Pn() { ... }
		}

Basic	Integer variable	Abstract data type
Access	When a process uses shared resources, it calls the wait() method on S, and when it releases them, it uses the signal() method on S.	When a process uses shared resources in the monitor, it has to access them via procedures.
Action	The semaphore's value shows the number of shared resources available in the system.	The Monitor type includes shared variables as well as a set of procedures that operate on them.
Condition Variable	No condition variables.	It has condition variables.

Single Linked List

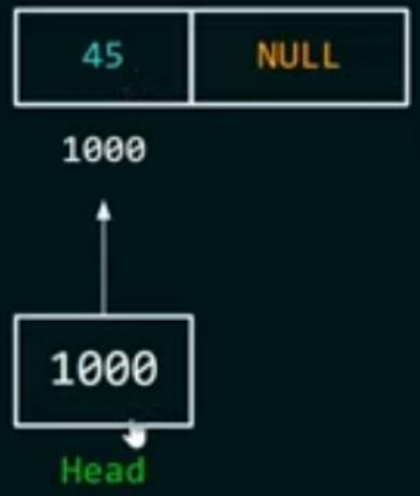


Single Linked List



Creating a Single Node of a List

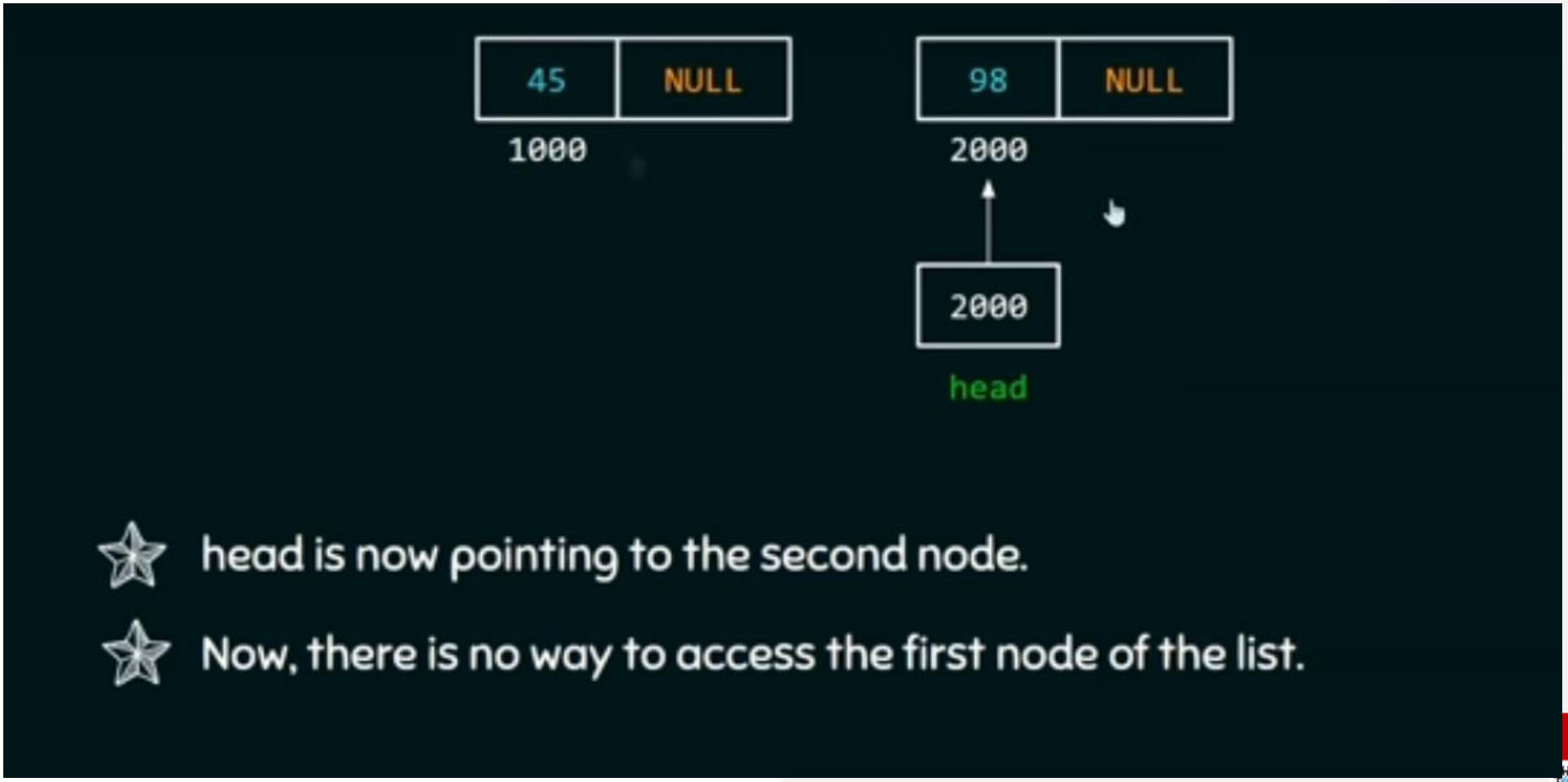
```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = NULL;
    head = (struct node *)malloc(sizeof(struct node));

    head->data = 45;
    head->link = NULL;

    printf("%d", head->data);
    return 0;
}
```



Creating a Linked List

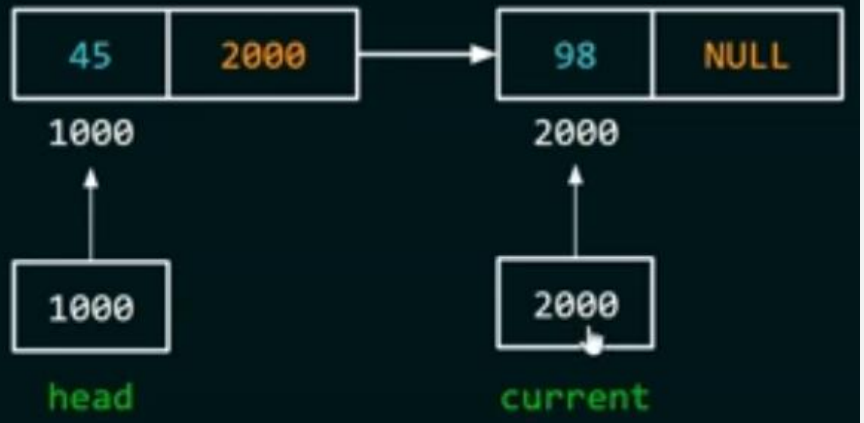
```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    head = malloc(sizeof(struct node));
    head->data = 98;
    head->link = NULL;

    return 0;
```

Creating a Linked List



Creating a Linked List

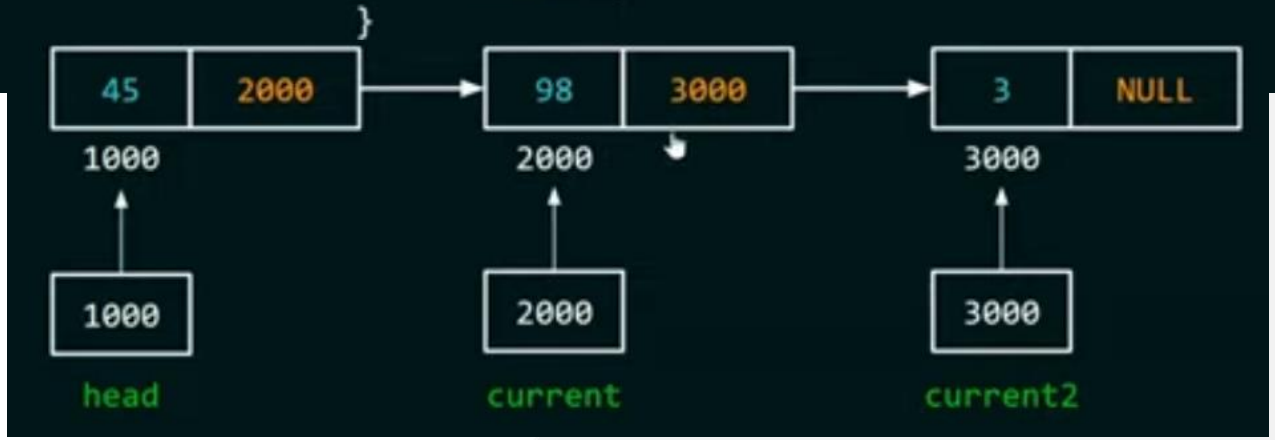
```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;
    return 0;
```



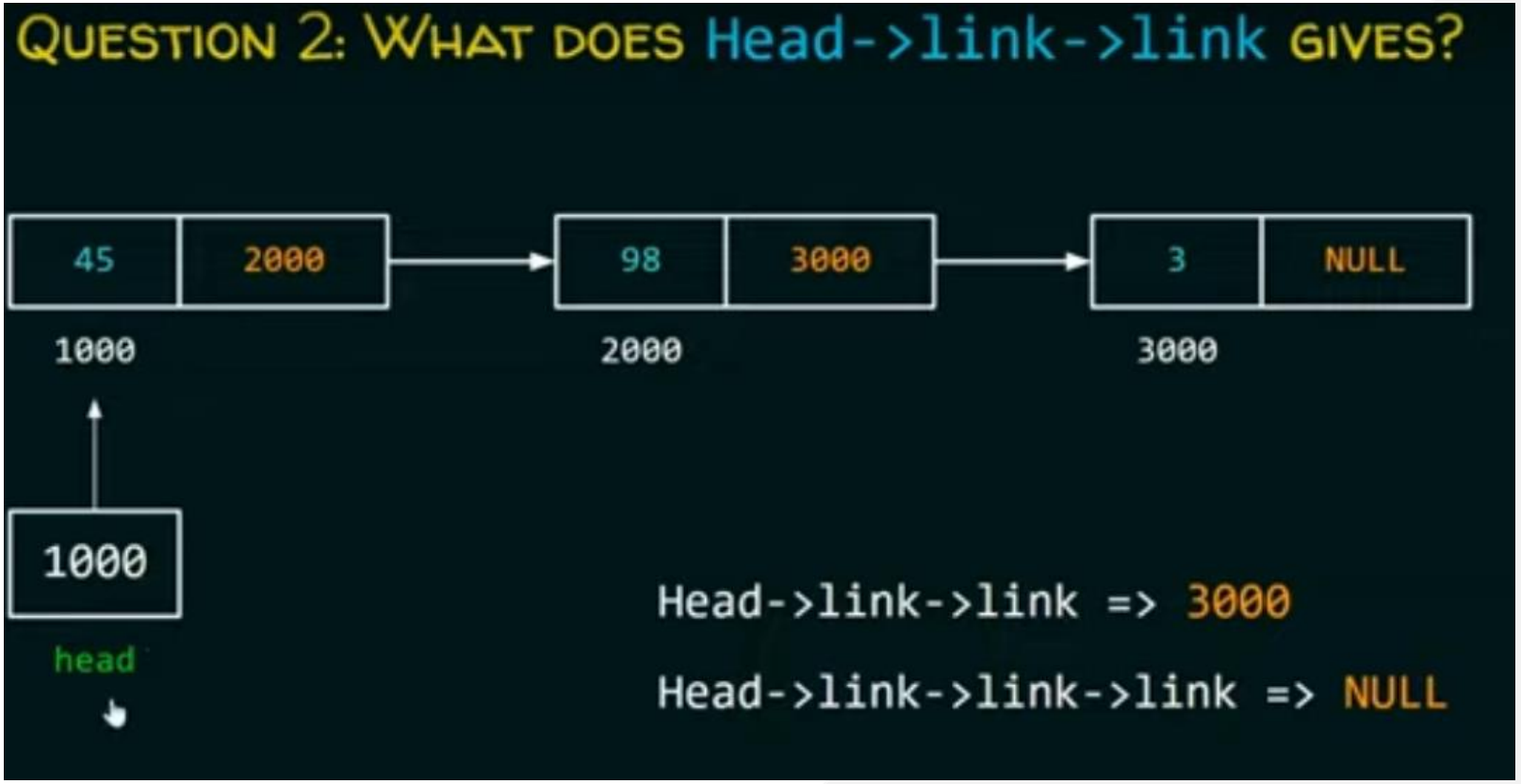
Linked List with 3 elements

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    struct node *current2 = malloc(sizeof(struct node));
    current2->data = 3;
    current2->link = NULL;
    current->link = current2;

    return 0;
}
```

Wastage of Memory



Accessing Linked List with Head pointer



Accessing Linked List with Head pointer

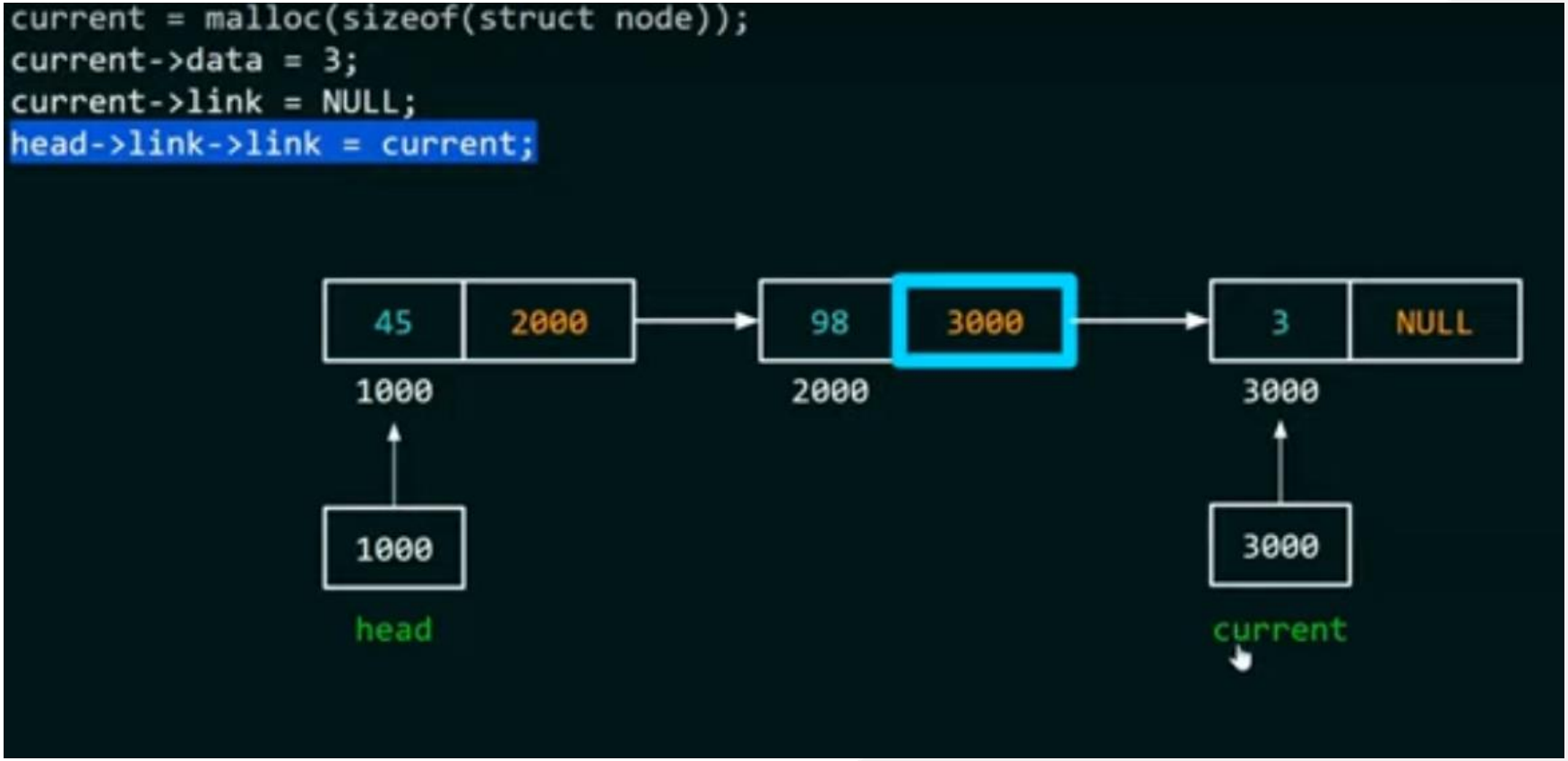
```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 98;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 3;
    current->link = NULL;



    return 0;
}
```

Accessing Linked List with Head pointer

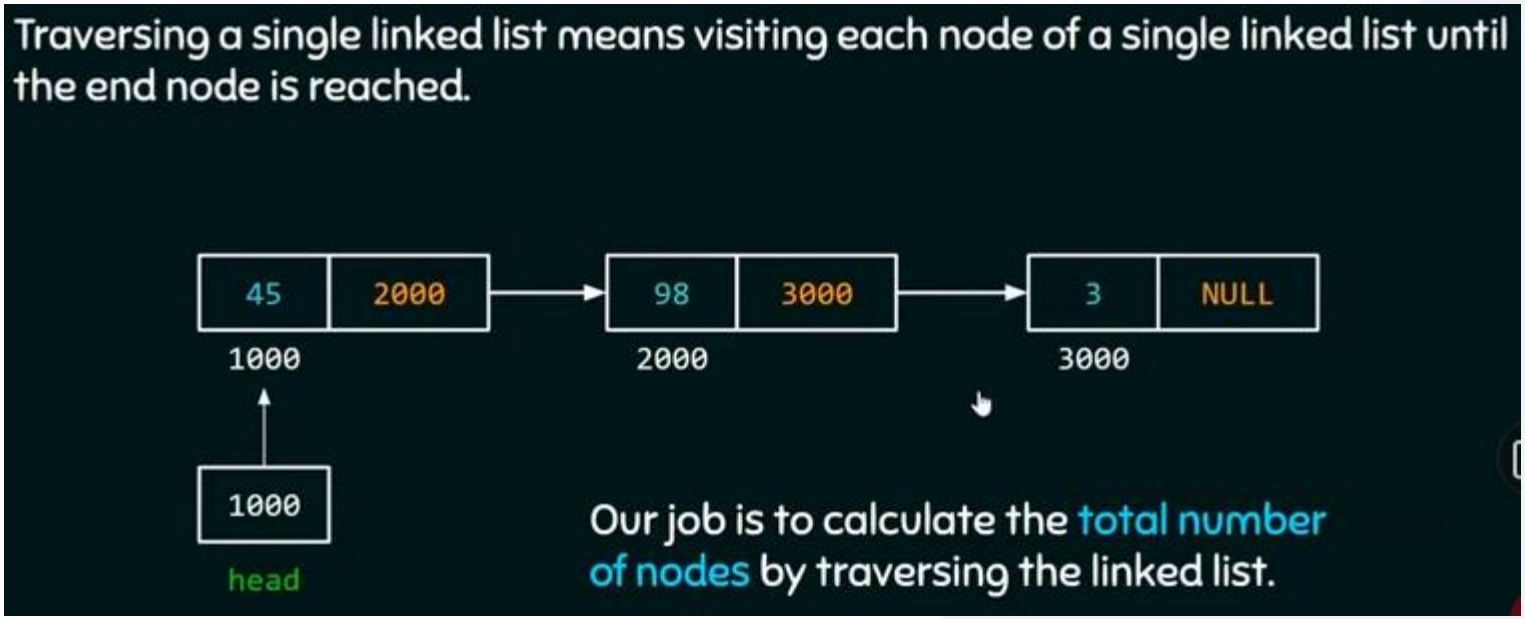
```
current = malloc(sizeof(struct node));
current->data = 3;
current->link = NULL;
```

Accessing Linked List with Head pointer

```
current = malloc(sizeof(struct node));
current->data = 3;
current->link = NULL;
head->link->link = current;
```

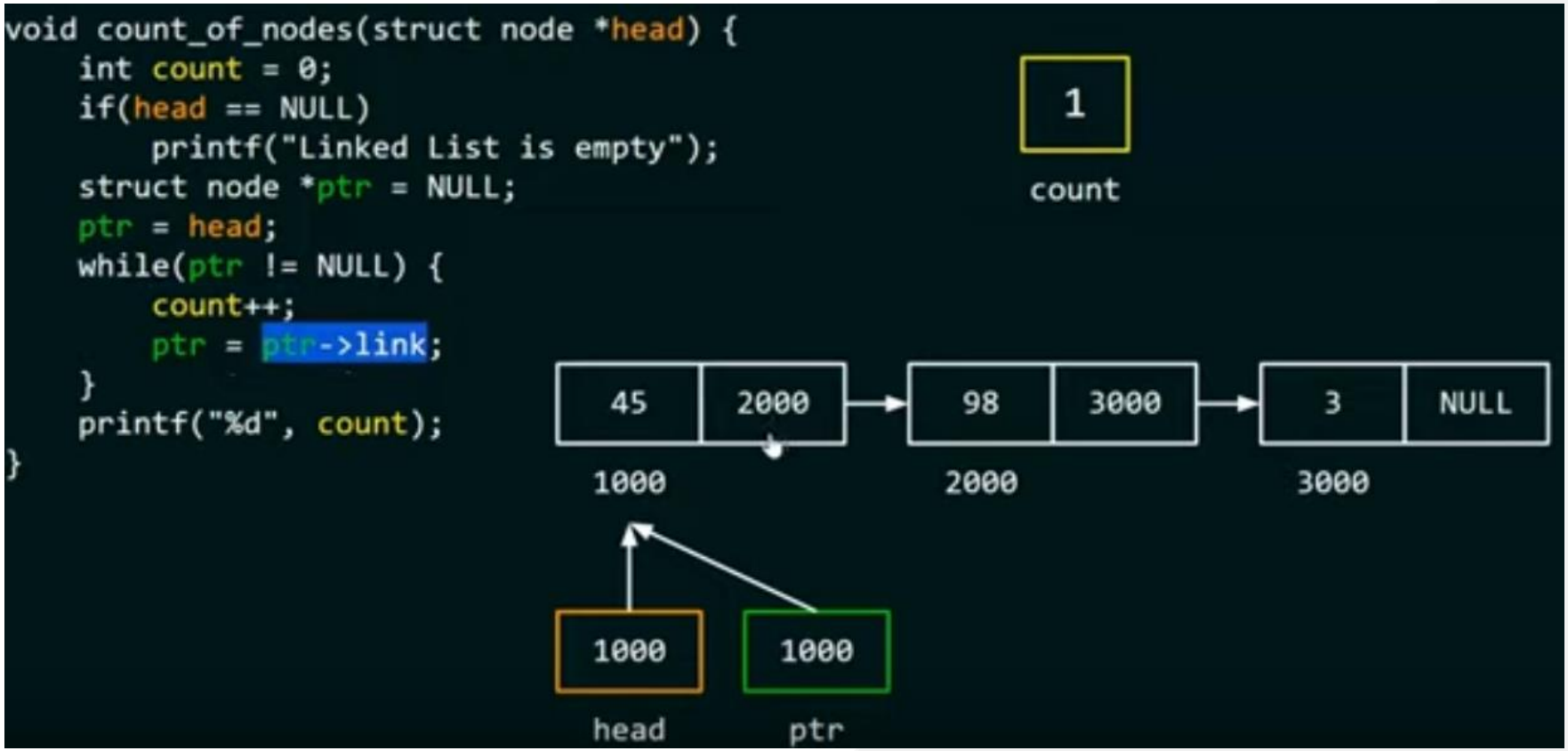


Traversing a Linked List



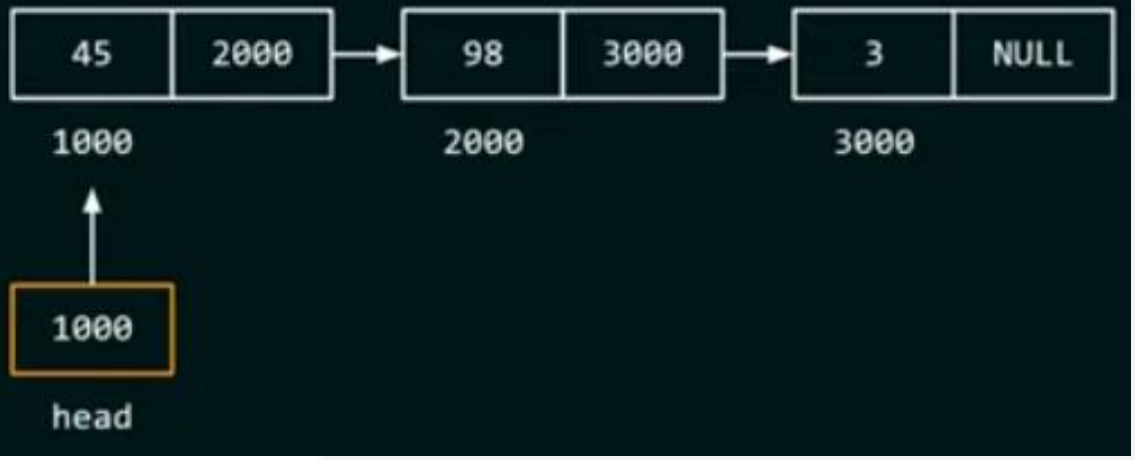
Traversing a Linked List

```
void count_of_nodes(struct node *head) {
    int count = 0;
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        count++;
        ptr = ptr->link;
    }
    printf("%d", count);
}
```



Traversing a Linked List(Printing Data)

```
void print_data(struct node *head) {
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }
}
```



References

- *Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. 2012. Operating System Concepts (9th. ed.). Wiley Publishing.*

- https://www.gatevidyalay.com/process-synchronization-practice-problems/

- https://www.studytonight.com/operating-system/process-synchronization

- https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/

THANK YOU