OPERATING SYSTEMS

Process Synchronization

FBA

BACKGROUND

- Processes can execute concurrently or in parallel
- CPU scheduler switches rapidly between processes to provide concurrent execution
- A process may be interrupted at any point in its instruction stream
- Parallel execution, in which two instruction streams execute simultaneously on separate processing cores
- We will explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes

PRODUCER-CONSUMER PROBLEM

- Modify the algorithm to remedy this deficiency - add an integer variable *counter*, initialized to 0
- *counter* is incremented every time we add a new item to the buffer
- decremented every time we remove one item from the buffer

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

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

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

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

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

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

DATA INTEGRITY PROBLEM

$register_1$ = `counter`
$register_1 = register_1 + 1$
`counter` = $register_1$

$register_2$ = `counter`
$register_2 = register_2 - 1$
`counter` = $register_2$

- "*counter++*" and "*counter--* " in machine language is like in the above.
- *register1* and *register2* is local CPU registers.
- Concurrent execution of "*counter++*" and "*counter--*" and allowing them to manipulate the counter variable create incorrect state.

T_0:	*producer*	execute	$register_1$ = `counter`	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = `counter`	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	`counter` = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	`counter` = $register_2$	$\{counter = 4\}$

RACE CONDITION

- Several process access and manipulate the same data concurrently
- Outcome of the execution depends on the particular order in which the access takes place
- To guard against this condition –
  - Ensure that only one process at a time can manipulate the counter variable (shared data)
  - The processes should be synchronized

OPERATING SYSTEMS

Critical Section

CRITICAL SECTION

- Consider a system consisting of ***n*** processes $\{P_0, P_1, \ldots, P_{n-1}\}$.
- ***Critical Section:*** segment of code of each process, which may change common variables, update a table, write a file and so on.
- While one process execute its critical section, no other process can execute their own critical section.
- ***Entry Section:*** section of code implementing critical section execution request
- ***Exit Section:*** section of code exiting from critical section
- ***Remainder section:*** Remaining code of the program.

REQUIREMENTS OF SOLUTION TO CRITICAL SECTION PROBLEM

1. ***Mutual exclusion:***
   - If a process is executing its critical section, no other process can be executing in their critical sections.
2. ***Progress:***
   - No process is executing in its critical section
   - Some process wish to enter their critical sections
   - Only those, who are not executing in their remainder section can participate in deciding which will enter the CS.
   - This selection cannot be postponed indefinitely.
3. ***Bounded waiting:***
   - Bound or Limit on number of times other process can enter their CS after a process has made request to enter its CS and the request is granted

OPERATING SYSTEMS

Hardware based solution for Critical Section Problem

HARDWARE-BASED SOLUTION TO THE CRITICAL SECTION PROBLEM

- More solutions to the critical-section problem using techniques ranging from hardware to software-based APIs
- These solutions are based on the premise of **locking** — protecting critical regions through the use of locks.
- In a single-processor environment CS problem can be solved by preventing interrupts from occurring while a shared variable is being modified.
- For multiprocessor environment, we need different measures.

- Modern computer systems allow to test and modify the content of a word or to swap the contents of two words atomically – which is uninterruptable unit. We can use *test_and_set()* and *compare_and_swap()* instructions.

TEST_AND_SET( )

- Executed atomically
- Mutual exclusion can be implemented by initializing a Boolean variable lock to false

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

COMPARE_AND_SWAP()

- Mutual exclusion can be achieved by declaring a global variable *lock* and initializing it to *0*
- First process that invokes this instruction will set *lock* to *1* and no other process can execute CS until this process updates it to *0* after CS execution.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

MUTEX LOCKS

- Operating-systems designers build software tools to solve CS problem.
- Simplest of these tools is “Mutex Lock” (Mutex = Mutual Exclusion)
- A process must acquire the lock before entering CS [*acquire()* function]
- A process must release the lock after exiting the CS [*release()* function]
- Mutex lock has a variable, *available* which indicates if the lock is available.

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}

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

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

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

MUTEX IMPLEMENTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int t_id[2]={1,2};
void *t_func(int *id);
int sum=0;
pthread_mutex_t mutex;
int main(){
        pthread_t t[2];
        pthread_mutex_init(&mutex,NULL);
        for(int i=0;i<2;i++){
            pthread_create(&t[i],NULL,(void *)t_func,&t_id[i]);
        }
        for(int i=0;i<2;i++){
                pthread_join(t[i],NULL);
        }
        pthread_mutex_destroy(&mutex);
        printf("Total count: %d\n",sum);
        return 0;
}
void *t_func(int *id){
        printf("Entered in Thread %d...\n",*id);
        for(int i=0;i<3;i++){
                pthread_mutex_lock(&mutex);
                sum++;
                pthread_mutex_unlock(&mutex);
        }
}
```

Declaring mutex variable

Initializing the mutex

Destroying the mutex after usage

Calling acquire func.

Calling release func.

Output:

```
Entered in Thread 2...
Entered in Thread 1...
Total count: 6
```

SEMAPHORE

- A semaphore S is an integer variable
- Is accessed only through two standard atomic operations: **wait()** and **signal()**.
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In case of wait(S), the testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption, i.e., **this operations are atomic**.

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

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

Types of Semaphores

- **Counting Semaphore**: The value can range over an unrestricted domain.
  - Used to control access to a given resource consisting of finite number of instances
  - Solves various synchronization problems.
- **Binary Semaphore:** The value can range only between 0 and 1. This behaves similar to Mutex Lock.

Counting Semaphore

- initialized to the number of resources available, **S** = **n**
- Each process that wishes to use a resource performs a wait() operation

  S = **S -1**

- When a process releases a resource, it performs a signal() operation

  S = **S** +**1**

- When **S** becomes **0**, all resources are being used
- processes that wish to use a resource will block until **S**>**0**

Binary Semaphore - Synchronization

- P_1 has statement S_1
- P_2 has statement S_2

- We want to make sure that S_1 executes before S_2
- We can use a semaphore variable **sync** and initialize it to **0**

P1:

```
S1;
signal(sync);
```

P2:

```
wait(sync);
S2;
```

Mutual Exclusion With Semaphores

- Binary Semaphores (mutex) can be used to solve CS problem.
- A semaphore variable (say mutex) can be shared by n processes and initialized to 1.
- Each process is structured as follows :

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

OPERATING SYSTEMS

Semaphore Implementation

SEMAPHORE IMPLEMENTATION

- Must guarantee that no two processes can execute the **wait**() and **signal**() on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

SEMAPHORE IMPLEMENTATION

- When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait
- Rather than this busy waiting, the process can block itself which places it into a waiting queue associated with the semaphore
- State of the process is switched to the waiting state and control is transferred to CPU scheduler which selects another process to execute.
- It will be restarted when some other process executes a signal() operation
- Restarted by a **wakeup**() operation that changes it from waiting state to ready state.

SEMAPHORE IMPLEMENTATION

Definition of a semaphore:

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

Definition of wait():

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

Definition of signal():

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

SEMAPHORE IMPLEMENTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
int t_id[2]={1,2};
void *t_func(int *id);
int sum=0;
sem_t s;
int main(){
        pthread_t t[2];
        sem_init(&s,0,1);
        for(int i=0;i<2;i++){
            pthread_create(&t[i],NULL,(void *)t_func,&t_id[i]);
        }
        for(int i=0;i<2;i++){
                pthread_join(t[i],NULL);
        }
        sem_destroy(&s);
        printf("Total count: %d\n",sum);
        return 0;
}
void *t_func(int *id){
        printf("Entered in Thread %d...\n",*id);
        for(int i=0;i<3;i++){
                sem_wait(&s);
                sum++;
                sem_post(&s);
        }
}
```

Declaring semaphore variable

Initializing the semaphore

Destroying the semaphore after usage

Calling wait func.

Calling signal func.

Output:

```
Entered in Thread 2...
Entered in Thread 1...
Total count: 6
```

OPERATING SYSTEMS

(Application of Semaphore) Bounded Buffer Problem

Bounded Buffer Problem

- Producer and Consumer share a fixed size buffer used as a queue
- Producer generates piece of data and put it into the buffer and start again
- Consumer removes data from the buffer
- **Concern 1:** Producer won’t produce data when the buffer is full
- **Concern 2:** Consumer won’t remove data when the buffer is empty

Inadequate Solution

```
procedure producer(){
    while(true){
        item = produceItem();
        if(itemCount == BufferSize)
            Sleep();
        putIntoBuffer(item);
        itemCount ++;
        if(itemCount == 1)
            wakeup(consumer);
        }
}
```

```
procedure consumer(){
    while(true){
    if(itemCount == 0)
        Sleep();
    item = removeFromBuffer();
    itemCount --;
    if(itemCount == BufferSize - 1)
            wakeup(producer);
    }
}
```

DEADLOCK !!!

If, consumer is interrupted just after checking the itemCount value and before sleep() call, and producer produces an item it will call wakeup(). But, this call will be lost as no consumer in sleeping yet. And it will continue to produce items. Eventually, producer will sleep when the buffer is full. On the other hand, consumer will also stay in sleep as producer won't call wakeup().

Deadlock & Starvation

- Two or more process can wait indefinitely for an event - **DEADLOCK** !!!
- It occurs because - two process depends on each other for causing an event in a specific manner

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

- **Starvation**: Processes wait indefinitely within the semaphore
- Occurs if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order

Use of Semaphore for Solution

- Solves the lost wakeup call by using two semaphore - countFill, countEmpty
- **countFill**: number of available item in the buffer to be read
- **countEmpty**: number of available space in the buffer to write
- If producer tries to decrement *countEmpty* when it is *0*, will be put to sleep
- When an item is consumed, *countEmpty* will be incremented and producer wakes up.
- Consumer works analogously.

Semaphore for multiple producer consumer

semaphore **mutex** = 1;

semaphore **countFull** = 0;

semaphore **countEmpty** = BUFFER_SIZE;

```
procedure producer(){
    while(true){
        item = producedItem();
        wait(countEmpty);
        wait(mutex);
        putIntoBuffer(item);
        signal(mutex);
        signal(countFull);
    }
}
```

```
procedure consumer(){
    while(true){
        wait(countFull);
        wait(mutex);
        item = removeFromBuffer();
        signal(mutex);
        signal(countEmpty);
        consumeItem(item);
    }
}
```

OPERATING SYSTEMS

(Application of Semaphore) Reader-Writer Problem

Reader - Writer Problem

- Data is shared among number of processes
- Any number of readers may simultaneously read the data
- Only one writer can write to the data
- If a writer is writing, no reader can read
- If at least one reader is reading, no writer can write to it
- Readers only read, Writers only write

Variation of the problem:

- **First Variation:** No reader will be kept waiting, unless a writer has got the permission to use the data, i.e., no reader should wait for other reader to finish
- **Second Variation:** Once a writer is ready, the writer performs as soon as possible, i.e., no new readers can start reading.

Solution to either problem may create starvation -

  - First case: writer may starve
  - Second case: reader may starve

Solution to First Variation

Reader process has the following data structures -

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

- **rw_mutex**: common to both reader and writer, functions as mutual exclusion semaphore for writer
- **mutex**: ensures mutual exclusion when read_count is updated
- **read_count**: keeps track of how many processes are reading the object

Solution to First Variation

- If a writer is in the critical section and ***n*** readers are waiting
  - One reader is queued on **rw_mutex**
  - Other n-1 readers are queued on **mutex**
- When a writer executes signal(rw_mutex)
  - Resume the execution of either waiting readers or single waiting writer
- Multiple process can acquire rw_mutex in read mode
- Only one process can acquire rw_mutex in write mode

Writers

```
do {
    wait( rw_mutex );

    /* writing is performed */

    signal( rw_mutex );
} while(true);
```

Readers

```
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)
```

Writer

```
do {
    wait( rw_mutex );

    /* writing is performed */

    signal( rw_mutex );
} while(true);
```

Readers

```
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)
```

Shared File

rw_mutex:

mutex:

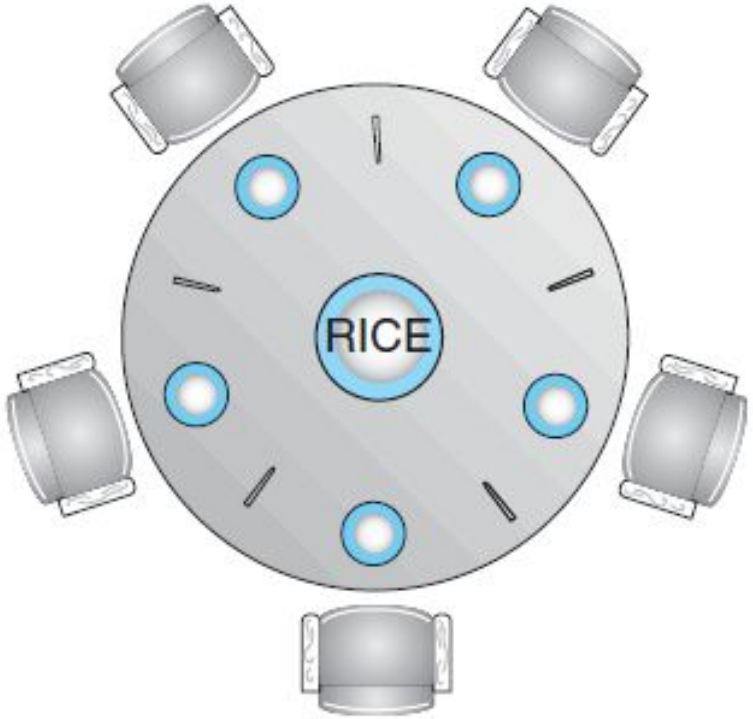
read_count:

OPERATING SYSTEMS

(Application of Semaphore) Dining Philosophers Problem

Dining-Philosophers Problem

- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks
- When she is finished eating, she puts down both chopsticks and starts thinking again



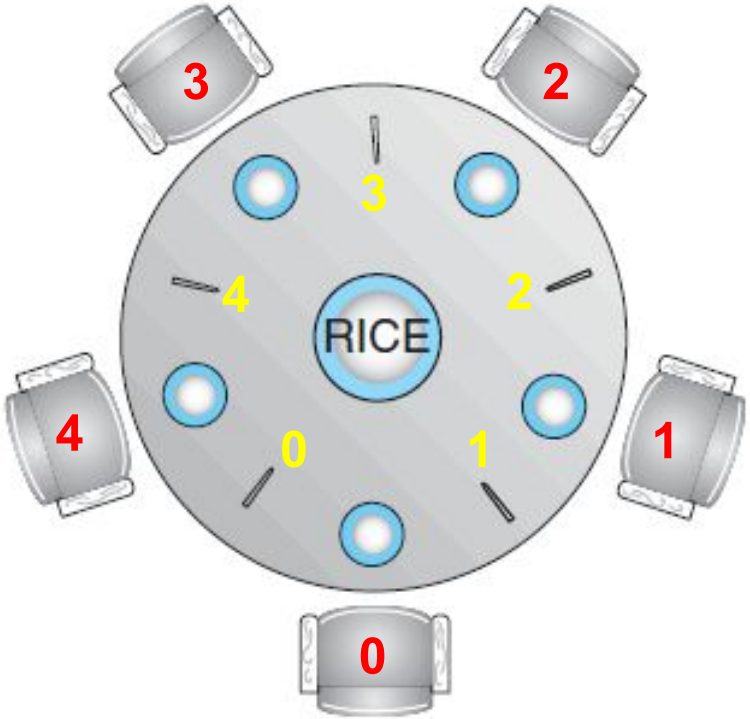
Solution to the problem

One of the simple solution is to use 5 semaphores for 5 chopsticks.

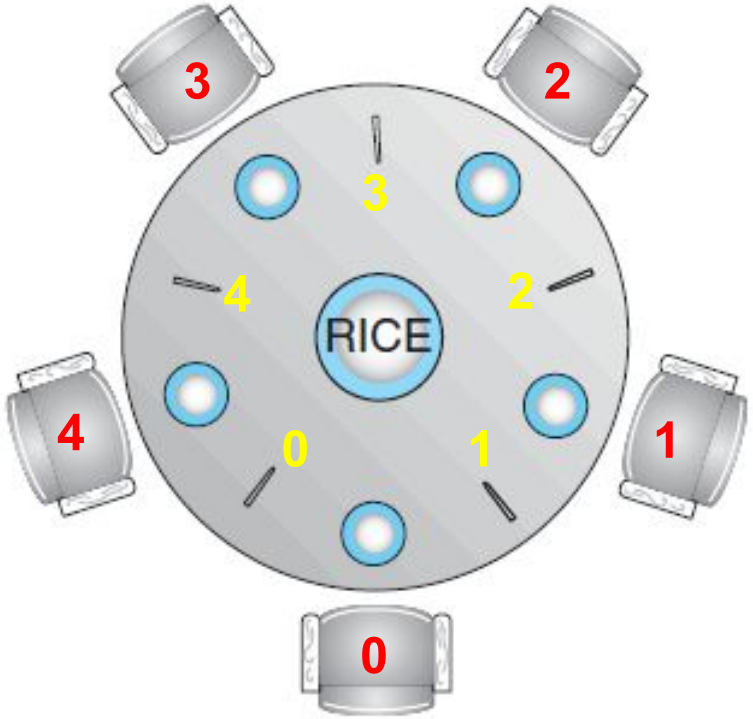
```
semaphore chopstick[5];
```

Structure of i^{th} philosopher's process can be -

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

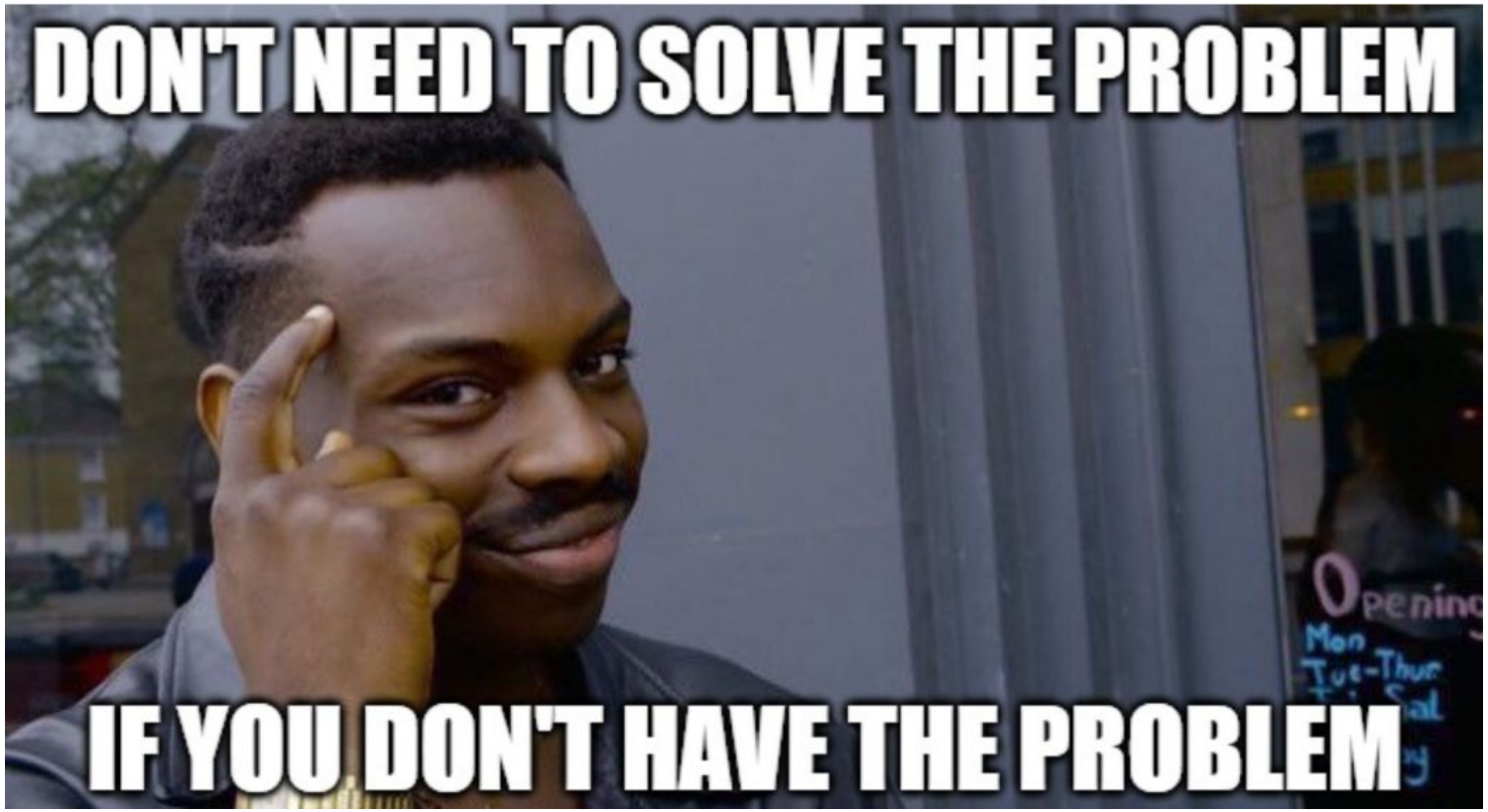


BUT GUESS WHAT'S COMING????



If all the philosophers are hungry at the same time and grab their left chopstick by calling wait(chopstick[i]), No one will ever find a second chopstick for eating. Consequently, they will die from starvation :P

How to prevent deadlock in this solution?



How to prevent deadlock in this solution?

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution — that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

Necessary Conditions for Deadlock

Necessary conditions for deadlocks are -

- Mutual Exclusion
- Hold and Wait
- No Preemption (cannot force to release the resource)
- Circular wait

If one of these are not present in a system, then the system is “**Deadlock Free**”.