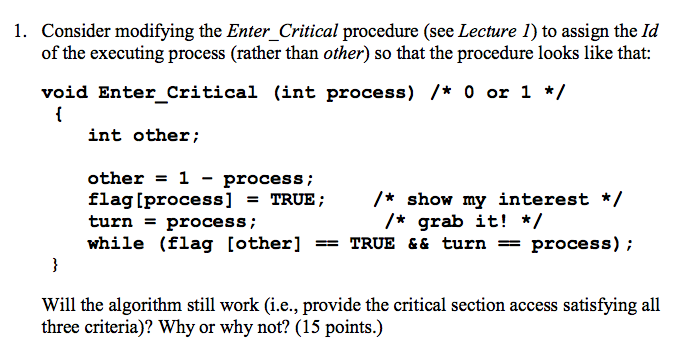
CS 520 Homework 1 | CWID 10430147 | Divyendra Patil | Username: dpatil3  
Date: 09/09/2017



Solution to 1:

The changes made to the algorithm will work just fine. Sample Execution shown below.

Case 1: process = 0;

Other=1-process=1-0=1;

Flag[0]=true;

Turn=process=0;

While(flag[1]&&turn==0){

Wait

}

//Critical Section  
 …  
 //End Critical Section

Flag[0]=false;

Case 2: process = 1;

Other=1-process=1-1=0;

Flag[1]=true;

Turn=process=1;

While(flag[0]&&turn==0){

Wait

}

//Critical Section  
 …  
 //End Critical Section

Flag[1]=false;

**Explanation:**

Critical Section Access Criteria’s:

1] **No more than one process may be inside the critical section**

Let’s the example of **case 1**.

Process 0 has been started.

Firstly, it will set Flag[0] to **true** & turn=0

The while condition for **wait** will be **false** since Flag[1] will be False.  
And hence process 0 will enter the critical condition.

Let’s the example of **case 2**.

Process 1 has been started.

Firstly, it will set Flag[1] to **true** & turn=1

The while loop will be true since Flag[0] is true which was set by process 0 at the beginning. Hence process 1 will enter in the wait state.

When process 0 exits the critical section it will set **Flag[0] = false**; to convey that it has finished executing in the critical section and it is free for use for other processes. Since Flag[0] is true, process 1 can now exit the wait state and enter the critical section with values set as **Flag[1]=True & turn=1**.

Now, if process 0 again wants to enter the critical section, it cannot do that easily since Flag[1]=true (**Set by process 1**) & turn=0 (**Set by process 0 during initialization**) & so process 0 has to enter the wait state.

If Process 0 beings its execution, it sets Flag[0] = true and turn = 0.   
If context switching happens and process 1 starts its execution setting both Flag[1] = true & turn = 1 & If again at this point context switching happens Process 0 resumes to evaluate while condition, the while loop will result in False since even though Flag[1] = true, the value of turn is also 1(set by Process 1). This allows Process 0 to enter in critical section. Even if context switch happens here again i.e. after Process 0 is in critical section, evaluation of while condition in Process 1 will result in true since at this point the values of all variables are as follows:

turn = 1,

Flag[0] = true (set by Process 0) & Flag[1] = true (set by Process 1)

So Process 1 will enter in Wait state till Process 0 exists critical section.

This ensures Mutual Exclusion is preserved by making sure that neither Process cannot enter Critical Section at the same time.

**2] No process outside the critical section may block access to it.**

Progress, between two processes, is defined as if Process 0 doesn’t want to enter in critical section and Process 1 wants to enter in critical section, the Process 1 can enter critical section independently as long as Process 0 doesn’t show interest to enter in critical section.

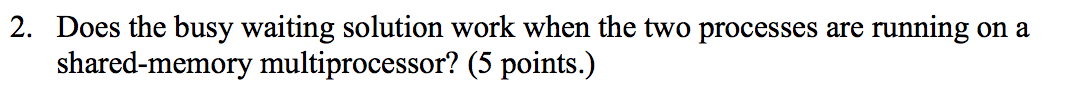
The modified algorithm will still work as Flag[0] will never be set True indicating that Process 0 does not want to enter in critical section. Hence the while condition in Process 1 will always be False and Process 1 can enter critical section any time it desires.

Same logic can be extended for Process 0 if Process 1 has no interest in entering critical section. Thus Progress property is also preserved.

**3] No process should wait forever to enter the critical section. TRUE as long as there are only two processes, and none is stuck.**

Bounded Waiting, between two processes, is defined as number of times another process is allowed to enter critical section when one process shows interests of entering the critical section and is actually allowed to enter critical section.

In above algorithm, any one process waits for maximum one turn after it shows interest to enter critical section and before its allowed in critical section. This is ensured by checking of Flag variable of the other process during execution.



**Solution to 2**:

Yes, the Busy Waiting Solution works when two processes are running on a shared-memory multiprocessor system.  
The Busy Waiting Solution works on two approaches:

1] Disabling Interrupt

2] Lock Mechanism

When these approaches are executed on a single processor system, a global variable is used which decides which process gets into the critical section. Whichever process does not get into the critical section, after checking the global variable enters the wait state (The basic principal of Busy waiting & Critical Section Problem).

Once the process exits the critical section, it changes the flag and the other process gets notified about it and it then enters the critical section.

**In this implementation/approach, the global variable is set on the processor itself.**

There are few differences when working on a shared-memory multiprocessor system. (Assuming there are two processors)

**Firstly, different processes are running on different processors.**

**Secondly, The Global variable is declared on shared memory.**

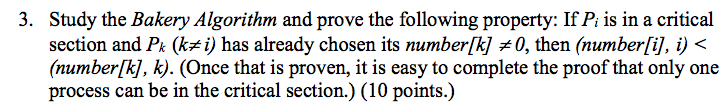
Now assuming there are more than two processors, running more than two or more processes, the scenario is different.

**The Entire System is dependent on global variables defined in CPU & shared memory.**

**The processes which are not using the critical sections on any processors, they will keep wasting CPU cycles to check the global variable.**

**If any processes on any processor does not reset the global variable, the system can crash since it might go into a loop or unlimited waiting for release of the global variable.**

**Note**: In multi-processor system, each CPU/core execute code simultaneously and also It enforces mutual exclusion, so whether the current CPU having disabled interrupts has nothing to avoid other CPU/core from entering the same region & violates the "no process not in its critical section can cause another process to block" constraint.



Solution to 3:

We basically have to prove two important things for this question:

(number[i], i) < (number[k], k)

&

Only one process can be in critical section.

To explain, we will take two cases into consideration.

**Case 1**: Let’s **assume**, (number[i],i) >= (number[k],k)

1. When Pi was entering in critical section, both process Pi and Pk has chosen their respective numbers. If our assumption holds true, then Pk would be currently in critical section instead of Pi indicating that either it has higher priority than Pi or it has chosen number less than Pi. But this contradicts the given condition that Pi is already in critical section.

2. Process Pi has already chosen its number and process Pk is selecting its number, Pi would have been delayed till the selection of number by Pk is completed. In this situation below use-cases are possible.

1. number[k]=number[i] such that k > i
2. number[k]>number[i]

In 1 & 2, our assumption becomes false.

3. number[k]=number[i] such that k < i

4. number[k]<number[i]

In 3 & 4, becomes the case 1, and contradicts given condition with assumption.

Thus, when Pi is in critical section yields to (number[i], i) < (number[k], k) only.

**Case 2**: Only one process can be in critical section.

Let’s assume that while entering in critical section Pi, has set number[i] = 1 (under the

assumption that Pi was the first process to enter critical section) & Pk wants to enter in critical section.

For Pk => number[k] = 2 ∵ Pi is in critical section and number[i] = 1.

The evaluating condition which is “*while(number[i]!= 0 && (number[i],i) < (number[k],k))*”

The second part of the while condition will hold True and process Pk will have to wait for Pi to exist critical section.

Upon exiting critical section, Pi will set number[i] = 0.

In this case, the first part of the evaluation condition becomes false & hence the while condition becomes false. Now process Pk now enters critical section.

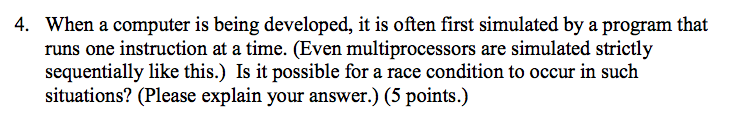
If again Pi again wants to enter critical section:

For Pi => number[i] to 3. ∵ number[k] = 2. And, number[i] > number[k],

∴ at this point Pi can’t enter in critical section & has to wait.

After Pk comes out of critical section, it sets number[k] = 0.

This proves that only one process can enter in critical section at any given time thus proving mutual exclusion property of Bakery algorithm.

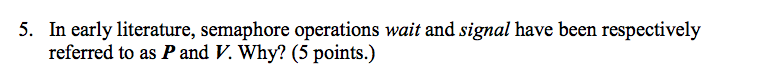


Solution to 4:

When a race condition occurs, it is based on program that is running or computer on which it is running. If the processor/processors is/are enabled for multi- programming or parallel programming, then it’s possible that a race condition can occur even though the developing CPU executes one instruction at a time.

If a process on a CPU starts to execute long loops & during the execution of the process it accesses a shared variable to read its value & at the same time, process scheduler of host environment starts to execute another process on the CPU on **another** thread.   
Suppose this new process tries to access same shared variable, **it will result in a race condition**.

On the other hand, if the parallel computing is disabled on host environment **there won’t be any chance of race condition**. Hence the occurrence of race condition is based on the configuration of host environment.



Solution to 5:

In the book,

The term Wait was indicted by “P”.   
This was derived from Dutch word “proberen” which means “To Test”.

The term Signal was indicated by “V”.   
This was derived from Dutch word “verhogen” which means “To Increment”.

../Screen%20Shot%202017-09-10%20at%202.47.46%20AM.png

../Screen%20Shot%202017-09-10%20at%202.47.55%20AM.png

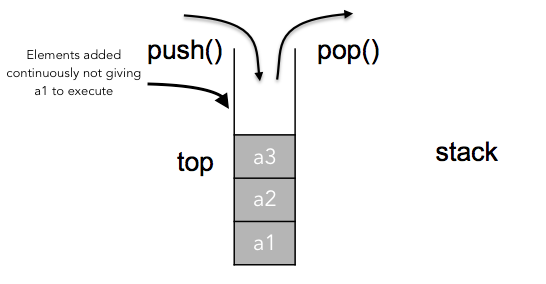
Solution to 6:

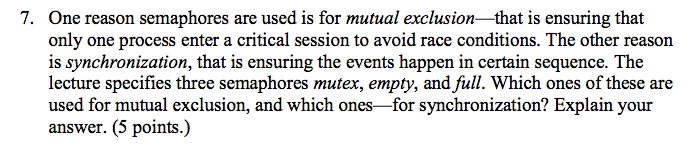
There are many definitions of starvation, a simple explanation might be that

Starvation is simply when a process or service is not being serve, even when there is no deadlock on the system or Starvation indicates that a process fails to progress its work because other processes are denying the resources to the process.

If a queue in semaphore is implemented, it can so happen that the first process which is added in queue will never be able to get out of queue to complete its work if the processes which are added later on might have high priority.

When the queue is been processed for work, and the last element is being served, another process is added into the last queue. So in simple terms, if the queue is never emptied till the first process, it will never be given a chance to utilize the computer resources for its own progress. This will result in starvation of resources for first process in queue.





Solution to 7:

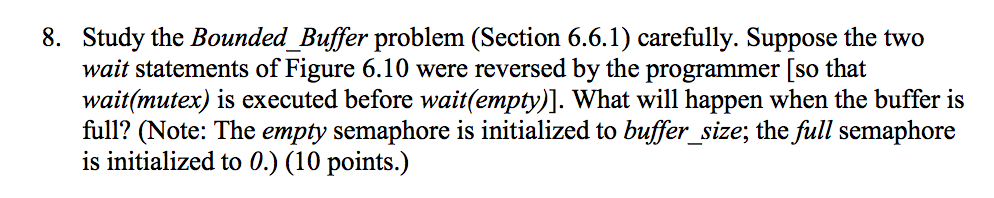
**Mutex**: Mutex makes sure that only one process executes in the critical section i.e. it checks if a process is in the critical section and does not allow any other process to enter the critical section. So it basically provides **Mutual Exclusion** by blocking other process and acting as a locking mechanism between processes.

**Empty & Full**: Empty & Full are **Semaphores** used to synchronize in the producer-consumer problem.

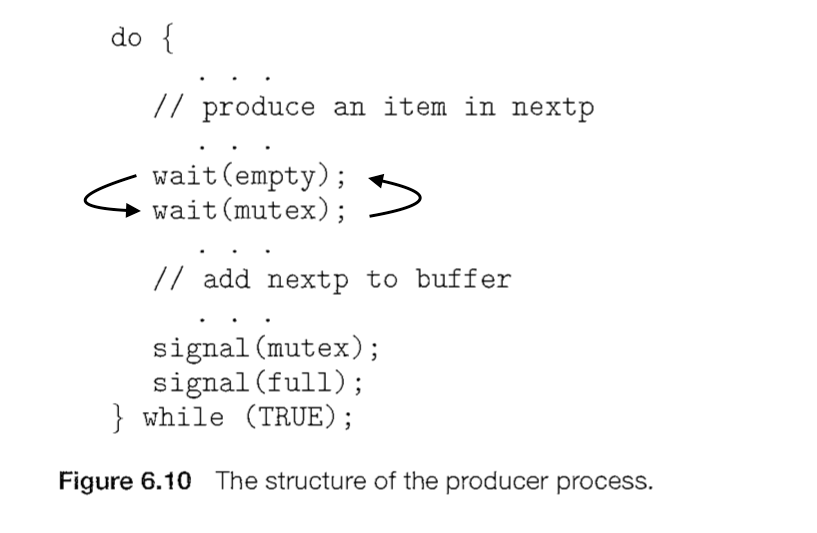
They are used in two conditions:

1. The consumer must wait for the producer to produce something if the queue is empty.
2. The producer must wait for the consumer to consume something if the queue is full.

The semaphores synchronize both producer and consumer by tracking count of filled and empty places in queue & hence symbolizes **Synchronization**.



Solution to 8:



(Representation of Question)

Mutex, Full & Empty are the semaphores used in the Bounded\_Buffer Problem.

Mutex semaphore is used for mutual exclusion between two process to enter in critical section by acquiring a Lock on the Shared section by Process.   
In this problem, it will make sure that either Producer is accessing the Buffer to enter data or Consumer is accessing the Buffer to read the data.

Full and Empty semaphores are used for synchronization between two processes by counting empty and full buffers.

In this problem, Full semaphore will indicate to the Producer that the Buffer is full and so Producer should stop writing Buffer until unless any position in the Buffer is empty.   
It also indicates to the Consumer that it can continue reading data from Buffer.

Similarly, Empty semaphore will indicate to the Consumer that the Buffer is empty and so Consumer should stop reading the Buffer until unless any position in Buffer is filled. It also indicates to the Producer that it can continue writing data in Buffer.

From figure 6.10, we can deduce that:

1] **wait(empty)** will check if any position in the buffer is available to enter the data or not.   
If Buffer is full, Empty semaphore will indicate to Producer that it should stop writing the data in Buffer and it won’t look for mutual exclusion. Whereas if any position is available Empty semaphore will indicate Producer that it can continue writing data in Buffer.

2] If the Producer can write the data in Buffer, it means that Producer can enter in critical section. And hence the wait(mutex) will check and acquire the lock for Producer to write the data in Buffer.

This creates an ideal mutual exclusion and synchronization between Producer, Consumer and Buffer.   
But if the steps are reversed the intended result from both mutual exclusion and synchronization can’t be obtained in case Buffer is full.

In case of situation mentioned in question, steps will be executed as below:

**[a.]** **wait(mutex)** will acquire the lock to the Buffer after evaluating that Consumer is not in the critical section i.e. Consumer is not reading the data from Buffer. Once the Producer has acquired lock it will try to enter in its critical section. Thus prohibiting Consumer to enter in its Critical section for reading data.

**[b.]** **wait(empty)** will check for empty spaces in the Buffer. But if the Buffer is full, Producer will continue to wait till any space in Buffer is emptied.

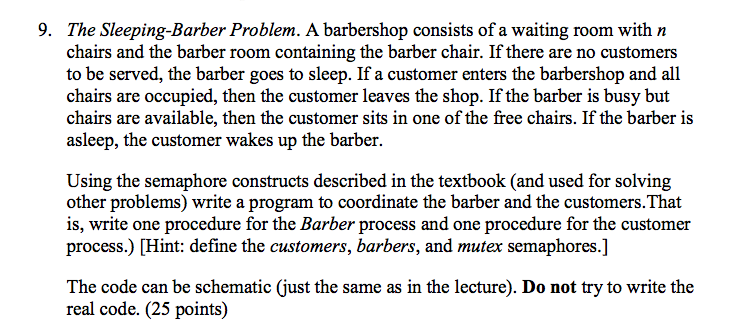
**[c.]** If Producer has acquired the lock on Buffer using **mutex**, Consumer will not be able to enter in its critical section to read the data. And both Producer and Consumer processer are stuck as below:

**i.** Producer in its critical section for any space in Buffer to become available.

**ii.** Consumer in wait state to let Producer release lock on Buffer so that it can read data from the Buffer.

**[d.]** The system goes in DeadLock (A state where both processes are waiting for each other to release lock or in this case complete an operation).

Furthermore, If the Producer tries to write data into the Buffer, the system will be in overflow and data will be lost since there is no space in the buffer yet the Producer will be writing it.



Solution to 9:

Reference taken from Tanenbaum. Page No 80. The Sleeping Barber Problem.

There are some things to be noted before solving the problem(Conditions):

1] If there are no customers, the barber will go to sleep.

2] If all the chairs are take, customer leaves.

3] If barber is busy, but chairs are available, customer sits on the free chairs,

4] If barber is asleep, customer wakes him up.

5] If there is even one customer, barber works.

6] Once the barber has completed his work with a customer, the customer leaves.

**We declare three semaphores**: customers | barbers | mutex

**A shared variable for waiting customers**: waiting

**Values of Barber semaphore.**

Barber = 0 /\*Either if barber is sleeping **OR** is working with no customers waiting\*/

Barber > 0 /\*Barber is working with customers waiting\*/

**Values of Customer semaphore.**

Customer = -1 /\* Barber is sleeping & there are no customers\*/

Customer > 0 /\* If there are customers in the shop waiting in the free chairs\*/

**Initialization values of semaphores.**

**Customer = 0** /\* # of customers waiting for the service | It will be 0 even if there are no customers\*/

**Barber = 0**

**Mutex = 1** /\* To check if customers are waiting **OR** not being cut (Mutual Exclusion)\*/

**waiting = 0** /\* To implement critical section\*/

**Barber Function:**

public void Barber() {

while(true) {

wait(customer); /\* Wait for customers to release the lock OR Sleep if no customers\*/ wait(mutex); /\*try to get lock to check shared variable\*/

waiting = waiting - 1; /\* decrement waiting count on critical section\*/   
signal(mutex); /\* release the lock\*/

signal(customer); /\*ask next customer in barber chair\*/   
 }

}

**Customer Function:**

Public void Customer() {

while(true) {

wait(barber); /\*wait for barber to release the lock.\*/

wait(mutex); /\*try to acquire lock to check shared variable\*/

If (waiting < n) /\*If there are no free chairs, leave. \*/

{

waiting = waiting + 1; /\*increment waiting counter. \*/   
signal(mutex); /\*release the lock\*/

signal(barber); /\*indicate barber of a new customer. \*/

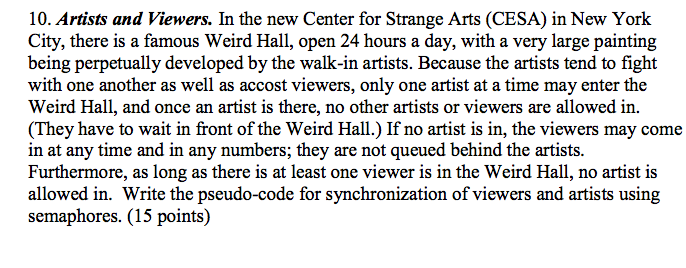
}

else

signal(mutex); /\*release the lock and leave the shop. \*/

}

}



Solution:

**There are some things to be noted before solving the problem (Conditions**):

Two artists can’t be in the hall at the same time.

Artist and a viewer can’t be in the hall at the same time.

Any number of viewers can be in the hall.

**Given condition:**

1. Customer semaphore can be anything from 0 to n

2. Artist semaphore can be either 0 or 1.

Shared variable will be **painting** in hall can have only two values:

1. Developing

2. Viewing

**Semaphore initiation**:

semaphore viewer = n; /\* number of viewers in hall, if no Artist is present.\*/

0; /\* if an Artist is present.\*/

semaphore artist = 0;

semaphore mutex = 1; /\* to have mutual exclusion between viewer and artist. \*/

painting = developed; /\* shared variable between viewer & artist to implement critical section \*/

**//Artist process**

Public void Artist() {

while(true) {

wait(artist); /\* check and wait for another artist to leave \*/

wait(viewer); /\* check and wait for all the viewers to leave and release the lock. \*/ wait(mutex); /\* try to get lock to painting artist = 1\*/

painting = developed; /\* critical section to start painting\*/

signal(mutex); /\* release lock artist = 0\*/

signal(viewer); /\* signal viewer that they can enter in hall\*/

singal(artist); /\* signal another artist. \*/

}

}

**//Viewer Process**

Public void Viewer() {

while(true) {

wait(Artist); /\* wait for artist to release the lock. \*/

wait(mutex); /\* try to acquire lock to check shared variable increment viewer count \*/

Painting = viewed /\* critical section => start viewing\*/

signal(mutex); /\*release the lock viewer = 0. \*/

signal(Artist); /\* indicate artist. \*/

}

}