Process Synchronization & Race Condition

- As we understand in a multiprogramming environment a good number of processes compete for limited number of resources. Concurrent access to shared data at some time may result in data inconsistency for e.g.

```
P ()
{
        read ( i );
        i = i + 1;
        write( i );
}
```

- Race condition is a situation in which the output of a process depends on the execution sequence of process. i.e. if we change the order of execution of different process with respect to other process the output may change.

General Structure of a process

- **Initial Section:** Where process is accessing private resources.
- **Entry Section:** Entry Section is that part of code where, each process request for permission to enter its critical section.
- **Critical Section:** Where process is access shared resources.
- **Exit Section:** It is the section where a process will exit from its critical section.
- **Remainder Section:** Remaining Code.

```
P()
{
    While(T)
    {
        Initial Section
        Entry Section
        Critical Section
        Exit Section
        Remainder Section
    }
}
```

Criterion to Solve Critical Section Problem

- **Mutual Exclusion:** No two processes should be present inside the critical section at the same time, i.e. only one process is allowed in the critical section at an instant of time.

- **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(means other process will participate which actually wish to enter). there should be no deadlock.

- **Bounded Waiting:** There exists a bound or a limit on the number of times a process is allowed to enter its critical section and no process should wait indefinitely to enter the CS.

Some Points to Remember

- Mutual Exclusion and Progress are mandatory requirements that needs to be followed in order to write a valid solution for critical section problem.

- Bounded waiting is optional criteria, if not satisfied then it may lead to starvation.

Solutions to Critical Section Problem

We generally have the following solutions to a Critical Section Problems:

1. Two Process Solution
    1. Using Boolean variable turn
    2. Using Boolean array flag
    3. Peterson's Solution

2. Operating System Solution
    1. Counting Semaphore
    2. Binary Semaphore

3. Hardware Solution
    1. Test and Set Lock
    2. Disable interrupt

Two Process Solution

- In general it will be difficult to write a valid solution in the first go to solve critical section problem among multiple processes, so it will be better to first attempt two process solution and then generalize it to N-Process solution.

- There are 3 Different idea to achieve valid solution, in which some are invalid while some are valid.

- **1- Using Boolean variable turn**

- **2- Using Boolean array flag**

- **3- Peterson's Solution**

- Here we will use a Boolean variable turn, which is initialize randomly(0/1).

P_0	P_1

while (1)
{
while (turn! = 0);
Critical Section
turn = 1;
Remainder section
}
	while (1)
	{
	while (turn! = 1);
	Critical Section
	turn = 0;
	Remainder Section
	}

- The solution follows Mutual Exclusion as the two processes cannot enter the CS at the same time.

- The solution does not follow the Progress, as it is suffering from the strict alternation. Because we never asked the process whether it wants to enter the CS or not?

- Here we will use a Boolean array flag with two cells, where each cell is initialized to F

P_0	P_1

while (1)
{
flag [0] = T;
while (flag [1]);
Critical Section
flag [0] = F;
Remainder section
}
	while (1)
	{
	flag [1] = T;
	while (flag [0]);
	Critical Section
	flag [1] = F;
	Remainder Section
	}

- This solution follows the Mutual Exclusion Criteria.
- But in order to achieve the progress the system ended up being in a deadlock state.

Dekker's algorithm

P_i	P_j

do
{
flag[i] = true;
while (flag[j])
{
if (turn == j)
{
flag[i] = false;
while (turn == j) ;
flag[i] = true;
}
}
/* critical section */
turn = j;
flag[i] = false;
/* remainder section */
}
while (true);
	do
	{
	flag[j] = true;
	while (flag[i])
	{
	if (turn == i)
	{
	flag[j] = false;
	while (turn == i) ;
	flag[j] = true;
	}
	}
	/* critical section */
	turn = i;
	flag[j] = false;
	/* remainder section */
	}
	while (true);

- Peterson's solution is a classic Software-based solution to the critical-section problem for two process. We will be using both: **turn and Boolean flag**.

P_0	P_1

while (1)
{
flag [0] = T;
turn = 1;
while (turn = = 1 && flag [1] = = T);
Critical Section
flag [0] = F;
Remainder section
}
	while (1)
	{
	flag [1] = T;
	turn = 0;
	while (turn = = 0 && flag [0] = =T);
	Critical Section
	flag [1] = F;
	Remainder Section
	}

- This solution ensures **Mutual Exclusion, Progress and Bounded Wait.**

Operation System Solution (Semaphores)

1. Semaphores are synchronization tools using which we will attempt n-process solution.
2. A semaphore S is a simple integer variable that, but apart from initialization it can be accessed only through two standard atomic operations: wait(S) and signal(S).
3. The wait(S) operation was originally termed as P(S) and signal(S) was originally called V(S).

```
Wait(S)
{
    while(s<=0);
    s--;
}
```

```
Signal(S)
{
    s++;
}
```

- Peterson's Solution was confined to just two processes, and since in a general system can have n processes, Semaphores provides n-processes solution.
- While solving Critical Section Problem only we initialize semaphore S = 1.
- Semaphores are going to ensure Mutual Exclusion and Progress but does not ensures bounded waiting.

Wait(S)
{
while(s<=0);
s--;
}

Signal(S)
{
s++;
}

$P_i()$
{
While(T)
{
Initial Section
wait(s)
Critical Section
signal(s)
Remainder Section
}
}

Classical Problems on Synchronization

- There are number of actual industrial problem we try to solve in order to improve our understand of Semaphores and their power of solving problems.
- Here in this section we will discuss a number of problems like
  - Producer consumer problem/ Bounder Buffer Problem
  - Reader-Writer problem
  - Dining Philosopher problem
  - The Sleeping Barber problem

Producer-Consumer Problem

- **Problem Definition** – There are two process Producer and Consumers, producer produces information and put it into a buffer which have n cell, that is consumed by a consumer. Both Producer and Consumer can produce and consume only one article at a time.
- A producer needs to check whether the buffer is overflowed or not after producing an item, before accessing the buffer.
- Similarly, a consumer needs to check for an underflow before accessing the buffer and then consume an item.
- Also, *the producer and consumer must be synchronized, so that once a producer and consumer it accessing the buffer the other must wait.*

Solution Using Semaphores

- Now to solve the problem we will be using three semaphores:
  - Semaphore S = 1 // CS
  - Semaphore E = n // Count Empty cells
  - Semaphore F = 0 // Count Filled cells

Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()

Knowledge Gate Website

Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
}	}

Knowledge Gate Website

Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
// Produce an item	
	// Pick item from buffer
// Add item to buffer	
	// Consume item
}	}

Semaphore S =

Semaphore E =

Semaphore F =

Producer()	Consumer()
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
// Produce an item	
	wait(S)
wait(S)	// Pick item from buffer
// Add item to buffer	signal(S)
signal(S)	
	// Consume item
}	}

Total three resources are used

- semaphore E take count of empty cells and over flow
- semaphore F take count of filled cells and under flow
- Semaphore S take care of buffer

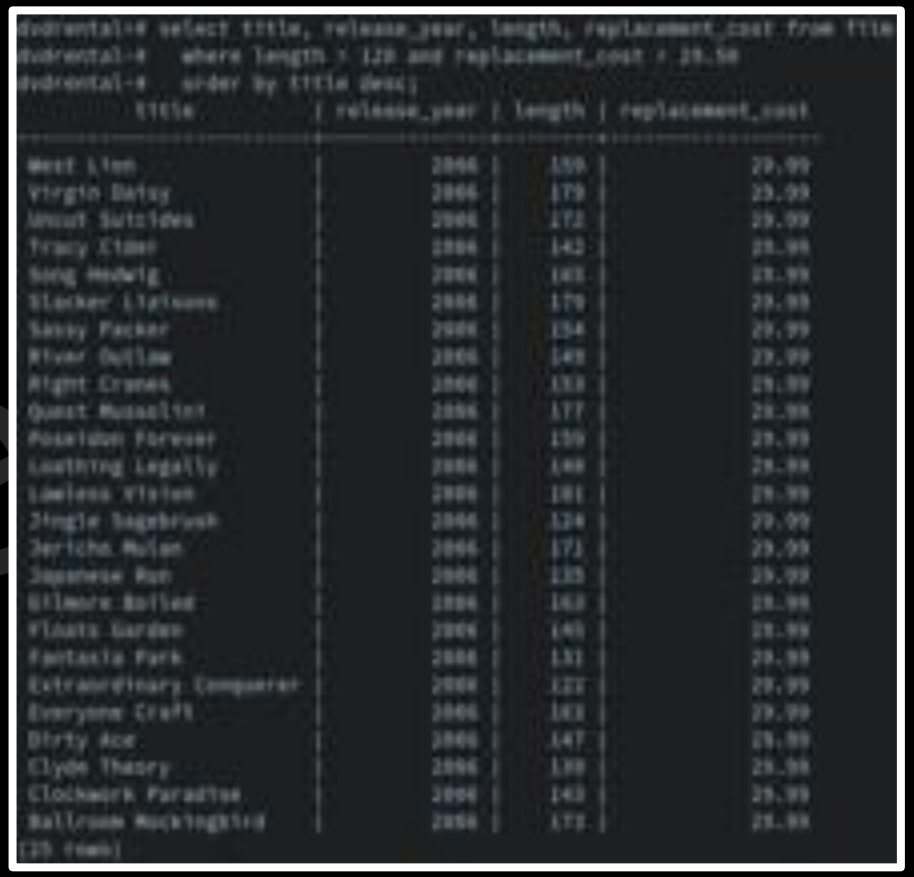
Producer()	Consumer()
{	{
while(T)	while(T)
{	{
// Produce an item	wait(F)//UnderFlow
wait(E)//OverFlow	wait(S)
wait(S)	// Pick item from buffer
// Add item to buffer	signal(S)
signal(S)	wait(E)
wait(F)	Consume item
}	}
}	}

Reader-Writer Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers), whereas others may want to update (that is, to read and write) the database(writers).

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

- 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.



Knowledge Gate Website

- **Points that needs to be taken care for generating a Solutions:**
  - The solution may allow more than one reader at a time, but should not allow any writer.
  - The solution should strictly not allow any reader or writer, while a writer is performing a write operation.

- **Solution using Semaphores**
  - The reader processes share the following data structures:
  - semaphore mutex = 1, wrt =1; // Two semaphores
  - int readcount = 0; // Variable

Three resources are used

- Semaphore Wrt is used for synchronization between WW, WR, RW
- Semaphore reader is used to synchronize between RR
- Readcount is simple int variable which keep counts of number of readers

Mutex =

Wrt =

Readcount =

Writer()	Reader()
CS //Write	CS //Read

Mutex =

Wrt =

Readcount =

Writer()	Reader()
Wait(wrt)	
CS //Write	CS //Read
Signal(wrt)	

Mutex =

Wrt =

Readcount =

	Readcount++
Wait(wrt)	
CS //Write	CS //Read
Signal(wrt)	
	Readcount--

[Knowledge Gate Website]

Mutex =

Wrt =

Readcount =

Writer()	Reader()
	Wait(mutex)
	Readcount++
Wait(wrt)	**signal(mutex)**
CS //Write	CS //Read
Signal(wrt)	**Wait(mutex)**
	Readcount--
	signal(mutex)

Knowledge Gate Website

Mutex =

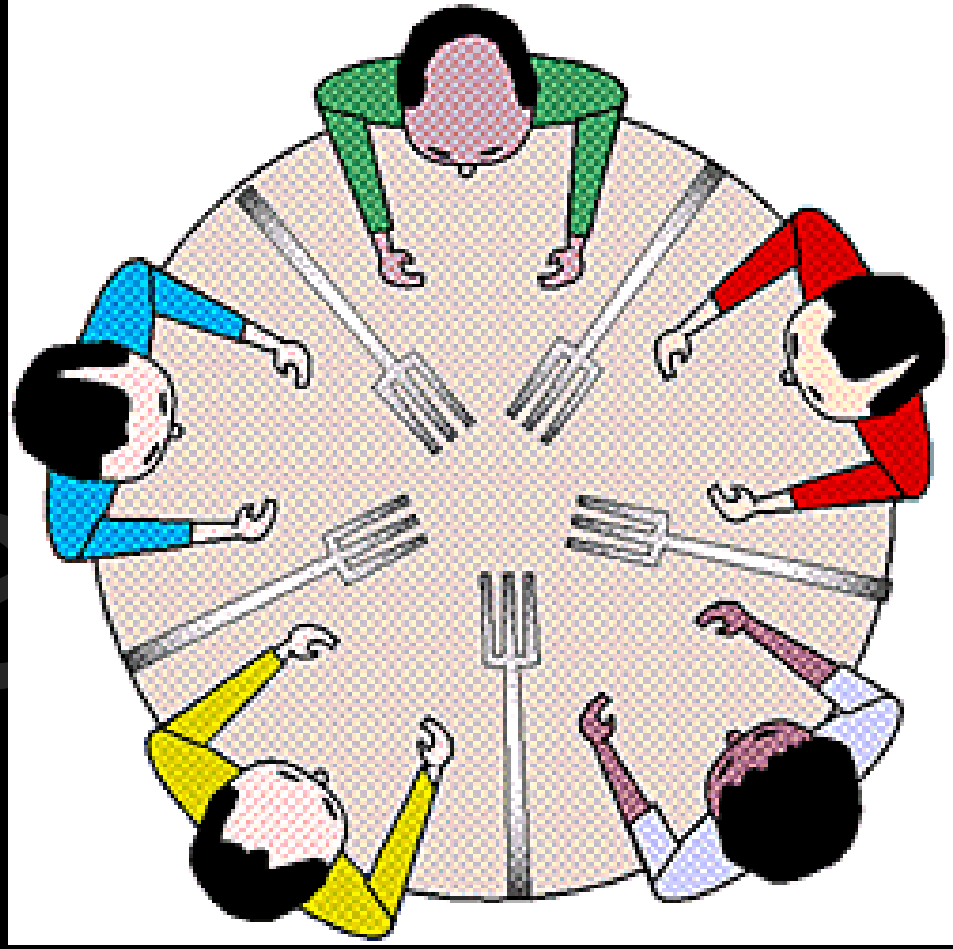
Wrt =

Readcount =

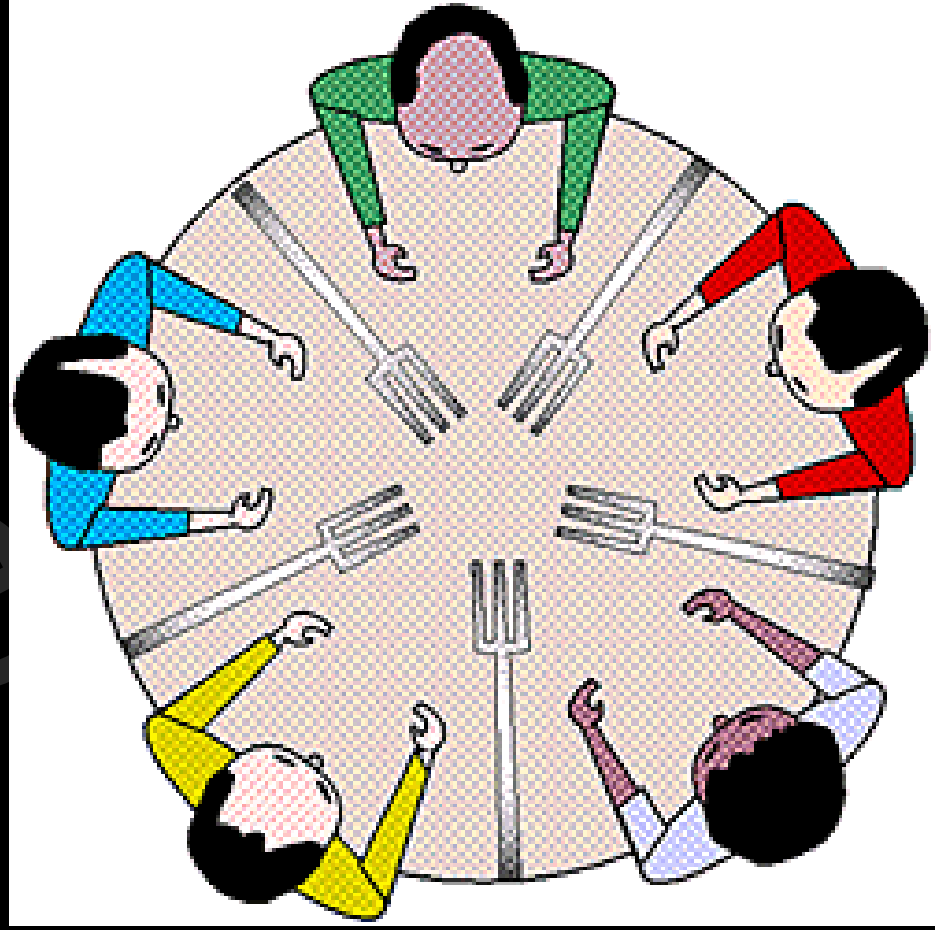
Writer()	Reader()
	Wait(mutex)
	Readcount++
	If(readcount ==1)
	wait(wrt) // first
Wait(wrt)	signal(mutex)
CS //Write	CS //Read
Signal(wrt)	Wait(mutex)
	Readcount--
	If(readcount ==0)
	signal(wrt) // last
	signal(mutex)

Dining Philosopher Problem

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.



- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she can’t pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.



Knowledge Gate Website

Indian Chopsticks

Knowledge Gate Website

Solution for Dining Philosophers

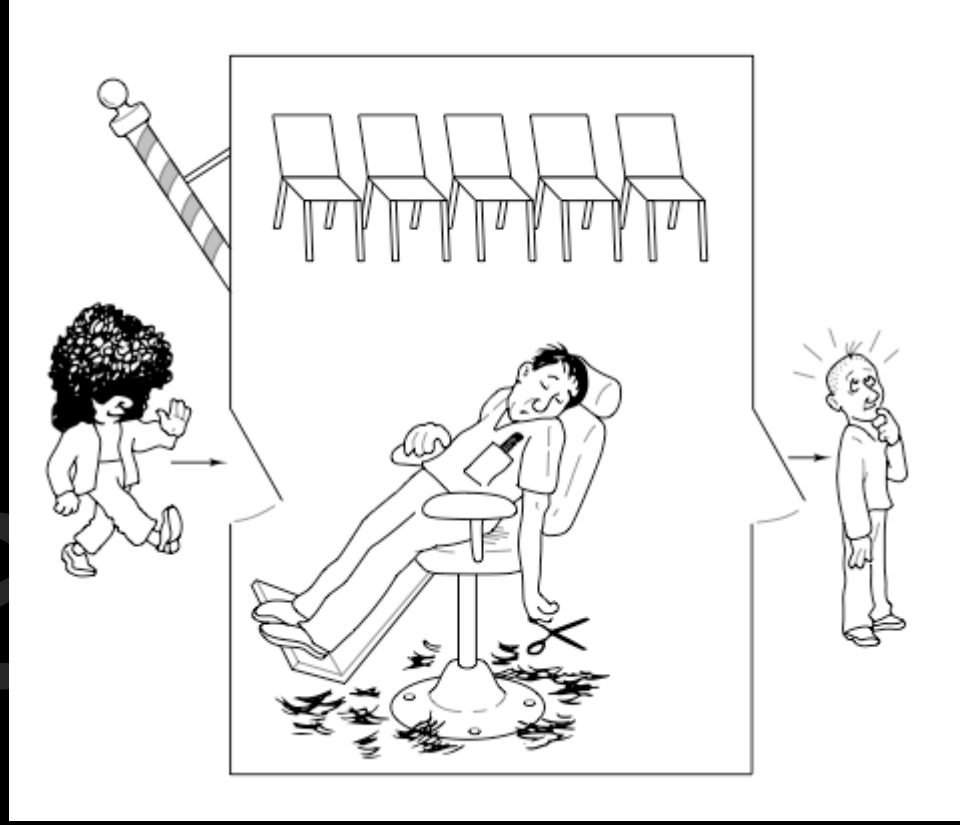
```
Void Philosopher (void)
{
        while ( T )
        {
                Thinking ( ) ;
                wait(chopstick [i]);
                wait(chopstick[(i+1)%5]);
                Eat( );
                signal(chopstick [i]);
                signal(chopstick[(i+1)%5]);
        }
}
```

- Here we have used an array of semaphores called chopstick[]
- Solution is not valid because there is a possibility of deadlock.

- The proposed solution for deadlock problem is
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow six chopstick to be used 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).
- One philosopher picks up her right chopstick first and then left chop stick, i.e. reverse the sequence of any philosopher.
- Odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

The Sleeping Barber problem

- **Barbershop**: A barbershop consists of a waiting room with n chairs and a barber room with one barber chair.
- **Customers**: Customers arrive at random intervals. If there is an available chair in the waiting room, they sit and wait. If all chairs are taken, they leave.
- **Barber**: The barber sleeps if there are no customers. If a customer arrives and the barber is asleep, they wake the barber up.
- **Synchronization**: The challenge is to coordinate the interaction between the barber and the customers using concurrent programming mechanisms.



```
semaphore barber = 0; // Indicates if the barber is available
semaphore customer = 0; // Counts the waiting customers
semaphore mutex = 1; // Mutex for critical section int
waiting = 0; // Number of waiting customers
```

Barber	Customer

while(true)
{
wait(customer);
wait(mutex);
waiting = waiting - 1;
signal(barber);
signal(mutex);
// Cut hair
}
	wait(mutex);
	if(waiting < n)
	{
	waiting = waiting + 1;
	signal(customer);
	signal(mutex);
	wait(barber);
	// Get hair cut
	}
	else
	{
	signal(mutex);
	}

Hardware Type Solution Test and Set

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software, all these solutions are based on the premise of locking —that is, protecting critical regions through the use of locks.

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

```
Boolean test and set (Boolean *target)
{
        Boolean rv = *target;
        *target  = true;
        return rv;
}
```

```
While(1)
{
        while (test and set(&lock));
        /* critical section */
        lock = false;
        /* remainder section */
}
```

- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word atomically —that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner.

- The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Basics of Dead-Lock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

P_1 P_2

R_1 R_2