USN | | | | | | | | | | |

Internal Assessment Test 2 – APR 2019

Sub:	OPERATING SYSTEMS					Sub Code:	15CS64	Branch:	CSE
Date:	20/04/2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	VI(A,B,C)		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Mention the requirements of Critical Section problem? Explain Peterson's solution to Critical Section problem.	[05]	CO4	L1
(b)	Summarize the Scheduling Criteria kept in mind while choosing the different scheduling algorithm.	[05]	CO2	L2
2 (a)	Discuss Readers-Writers problem using semaphore.	[05]	CO4	L2
(b)	Describe Dining Philosopher's problem.	[05]	CO4	L2
3 (a)	Define Semaphore. Summarize its usage and implementation.	[10]	CO4	L2
4 (a)	Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:	[10]	CO2	L3

Process	Burst Time	Arrival time	Priority
P1	10	0	3
P2	1	1	1
P3	2	2	3
P4	1	3	4
P5	5	4	2

USN | | | | | | | | | | |

Internal Assessment Test 2 – APR 2019

Sub:	OPERATING SYSTEMS					Sub Code:	15CS64	Branch:	CSE
Date:	20/04/2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	VI(A,B,C)		OBE

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1 (a)	Mention the requirements of Critical Section problem? Explain Peterson's solution to Critical Section problem.	[05]	CO4	L1
(b)	Summarize the Scheduling Criteria kept in mind while choosing the different scheduling algorithm.	[05]	CO2	L2
2 (a)	Discuss Readers-Writers problem using semaphore.	[05]	CO4	L2
(b)	Describe Dining Philosopher's problem.	[05]	CO4	L2
3 (a)	Define Semaphore. Summarize its usage and implementation.	[10]	CO4	L2
4 (a)	Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:	[10]	CO2	L3

Process	Burst Time	Arrival time	Priority
P1	10	0	3
P2	1	1	1
P3	2	2	3
P4	1	3	4
P5	5	4	2

I) Draw the Gantt chat for SRTF (SJF-Preemptive), Preemptive Priority and Round Robin algorithms (a smaller priority number implies a higher priority & RR (quantum = 2)).
ii) Calculate Turnaround time and waiting time for each process.
iii) Find the average waiting time and turnaround time.

5 (a)	Consider the following snapshot of a system									[10]	CO4	L3

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

i) Find the need matrix and safe sequence using Banker's algorithm.
ii) Mention above system is safe or not.

6 (a)	Discuss how Deadlocks are detected when "Resource type has several instances", along with an example.	[10]	CO4	L2
7 (a)	Define Deadlock. What are the necessary conditions for deadlock? How deadlocks are recovered.	[10]	CO4	L1
8 (a)	What is paging? Explain paging hardware with Translation look-aside buffer.	[06]	CO3	L1
(b)	Write short note on Logical and Physical address space.	[04]	CO3	L1

---

i) Draw the Gantt chat for SRTF (SJF-Preemptive), Preemptive Priority and Round Robin algorithms (a smaller priority number implies a higher priority & RR (quantum = 2)).
ii) Calculate Turnaround time and waiting time for each process.
iii) Find the average waiting time and turnaround time.

5 (a)	Consider the following snapshot of a system									[10]	CO4	L3

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

i) Find the need matrix and safe sequence using Banker's algorithm.
ii) Mention above system is safe or not.

6 (a)	Discuss how Deadlocks are detected when "Resource type has several instances", along with an example.	[10]	CO4	L2
7 (a)	Define Deadlock. What are the necessary conditions for deadlock? How deadlocks are recovered.	[10]	CO4	L1
8 (a)	What is paging? Explain paging hardware with Translation look-aside buffer.	[06]	CO3	L1
(b)	Write short note on Logical and Physical address space.	[04]	CO3	L1

1 (a) Mention the requirements of Critical Section problem? Explain Peterson's solution to Critical Section problem.

Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section".

- A solution to the problem must satisfy the following 3 requirements:

1. Mutual Exclusion

Only one process can be in its critical-section.

2. Progress

Only processes that are not in their remainder-section can enter their critical-section, and the selection of a process cannot be postponed indefinitely.

3. Bounded Waiting

There must be a bound 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 the request is granted.

Peterson's solution

This is a classic **software-based solution** to the critical-section problem.

This is limited to 2 processes.

The 2 processes alternate execution between critical-sections and remainder-sections.

The 2 processes share 2 variables (Figure 3.2):

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

where *turn* = indicates whose turn it is to enter its critical-section.
(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).

flag = used to indicate if a process is ready to enter its critical-section.
(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- To enter the critical-section,
  → firstly process *Pi* sets *flag[i]* to be true and
  → then sets *turn* to the value j.
- If both processes try to enter at the same time, *turn* will be set to both i and j at roughly the same time.
- The final value of *turn* determines which of the 2 processes is allowed to enter its critical-section first.
- To prove that this solution is correct, we show that:
  - Mutual-exclusion is preserved.
  - The progress requirement is satisfied.
  - The bounded-waiting requirement is met.

(b) Summarize the Scheduling Criteria kept in mind while choosing the different scheduling algorithm.

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

• **CPU utilization** - The CPU must be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent .

• **Throughput** - If the CPU is busy executing processes, then work is done fast. One measure of work is the number of processes that are completed per time unit, called throughput.

• **Turnaround time -** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Time spent waiting (to get into memory + ready queue + execution + I/O)

• **Waiting time -** The total amount of time the process spends waiting in the ready queue.

• **Response time -** The time taken from the submission of a request until the first response is produced is called the response time. It is the time taken to start responding. In interactive system, response time is given criterion.

2 a) Discuss Readers-Writers problem using semaphore.

Readers-Writers Problem

- A data set is shared among a number of concurrent processes.
- **Readers** are processes which want to only read the database (DB).

1. Problem:

Obviously, if 2 readers can access the shared-DB simultaneously without any problems.

However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.

Solution:

The writers must have exclusive access to the shared-DB while writing to the DB.

- **Shared-data**

```
semaphore mutex, wrt;
int readcount;
```

where,

mutex is used to ensure mutual-exclusion when the variable *readcount* is updated.

wrt is common to both reader and writer processes.

wrt is used as a mutual-exclusion semaphore for the writers.

wrt is also used by the first/last reader that enters/exits the critical-section.

readcount counts no. of processes currently reading the object.

Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process: **Reader Process:**

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

```
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);
```

- The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
- The mode of lock needs to be specified:
  - **1. read mode**
    - When a process wishes to read shared-data, it requests the lock in *read mode*.
  - **2. write mode**
    - When a process wishes to modify shared-data, it requests the lock in *write mode*.

Multiple processes are permitted to concurrently acquire a lock in read mode, but only one process may acquire the lock for writing.

1. These locks are most useful in the following situations:
   - In applications where it is easy to identify which processes only read shared-data and which threads only write shared-data.
   - In applications that have more readers than writers.

The Dining-Philosophers Problem

Problem statement:

There are 5 philosophers with 5 chopsticks (semaphores).
A philosopher is either eating (with two chopsticks) or thinking. The philosophers share a circular table (Figure 3.10).
The table has a bowl of rice in the center and 5 single chopsticks.
From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time.
Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
When 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.

- Problem objective:

To allocate several resources among several processes in a deadlock-free & starvation-free manner.

Solution:

Represent each chopstick with a semaphore (Figure 3.11).
A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
The philosopher releases her chopsticks by executing the signal() on the semaphores. This solution guarantees that no two neighbors are eating simultaneously.

Shared-data

semaphore chopstick[5];

Initialization

chopstick[5]={1,1,1,1,1}.


```
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);
```

Disadvantage:

Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Three possible remedies to the deadlock problem:

1. Allow **at most 4** philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
3. Use an **asymmetric solution**; i.e. an 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.

3 (a) Define Semaphore. Summarize its usage and implementation.

Semaphores

- A *semaphore* is a synchronization-tool.
- It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.
- A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:
  1. wait() and
  2. signal().
- wait() is termed P ("to test").
  signal() is termed V ("to increment").
- **Definition of wait():**

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

Definition of signal():

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

- When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.
- Also, in the case of wait(S), following 2 operations must be executed without interruption:
  → Testing of S(S<=0) and
  → Modification of S (S--)

Semaphore Usage

Counting Semaphore

- The value of a semaphore can range over an unrestricted domain

Binary Semaphore

- The value of a semaphore can range only between 0 and 1.
- On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual-exclusion.

1) Solution for Critical-section Problem using Binary Semaphores

- Binary semaphores can be used to solve the critical-section problem for multiple processes.
- The „n" processes share a semaphore *mutex* initialized to 1 (Figure 3.9).

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Figure 3.9 Mutual-exclusion implementation

with semaphores **2) Use of counting semaphores**

- Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving synchronization problems

- Semaphores can also be used to solve synchronization problems.
- For example, consider 2 concurrently running-processes:

1. Suppose we require that S2 be executed only after S1 has completed.
2. We can implement this scheme readily
   by letting P1 and P2 share a common semaphore *synch* initialized to 0, and
   by inserting the following statements in process P1

```
S1;
signal(synch);
```

   and the following statements in process P2

```
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

- Main disadvantage of semaphore:
  → Busy waiting.
- **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 entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock).
- To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:

→ When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself.

→ A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup().

- We assume 2 simple operations:
  → **block()** suspends the process that invokes it.
  → **wakeup(P)** resumes the execution of a blocked process P.
- We define a semaphore as follows:
-

```
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);
        }
}
```

- This (critical-section) problem can be solved in two ways:
  1. In a **uni-processor** environment
     Inhibit interrupts when the wait and signal operations execute.
     Only current process executes, until interrupts are re-enabled & the scheduler regains control.
  2. In a **multiprocessor** environment
     Inhibiting interrupts doesn't work.
     Use the hardware / software solutions described above.

6 a. Discuss how Deadlocks are detected when "Resource type has several instances", along with an example.

Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:
  Let 'n' be the number of processes in the system Let 'm' be the number of resources types.
- Following data structures are used to implement this algorithm.
  1) Available [m]
  - This vector indicates the no. of available resources of each type.
  - If Available[j]=k, then k instances of resource type Rj is available.

  2) Allocation [n][m]
  - This matrix indicates no. of resources currently allocated to each process.
  - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.

  3) Request [n][m]
  - This matrix indicates the current request of each process.
  - If Request [i, j] = k, then process Pi is requesting k more instances of resource type Rj.

Step 1:

Let Work and Finish be vectors of length m and n respectively.

a) Initialize Work = Available

b) For i=0,1,2…….n

if Allocation(i) != 0

then

Finish[i] = false;

else

Finish[i] = true;

Step 2:

Find an index(i) such that both

a) Finish[i] = false

b) Request(i) <= Work.

If no such i exist, goto step 4.

Step 3:

Set:

Work = Work + Allocation(i)

Finish[i] = true

Go to step 2.

Step 4:

If Finish[i] = false for some i where 0 < i < n, then the system is in a deadlock state.

7 (a) Define Deadlock. What are the necessary conditions for deadlock? How deadlocks are recovered.

Deadlocks

- Deadlock is a situation where a set of processes are blocked because each process is
  → holding a resource and
  → waiting for another resource held by some other process.

Necessary Conditions

- There are four conditions that are necessary to achieve deadlock:

1) Mutual Exclusion

- At least one resource must be held in a non-sharable mode.
- If any other process requests this resource, then the requesting-process must wait for the resource to be released.

2) Hold and Wait

- A process must be simultaneously
  → holding at least one resource and
  → waiting to acquire additional resources held by the other process.

3) No Preemption

- Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4) Circular Wait

- A set of processes { P0, P1, P2, . . ., PN } must exist
  such that P0 is waiting for a resource that is held by
  P1
  P1 is waiting for a resource that is held by P2, and so on

Recovery from deadlock

- Three approaches to recovery from deadlock:
  1) Inform the system-operator for manual intervention.
  2) Terminate one or more deadlocked-processes.
  3) Preempt(or Block) some resources.

Process Termination

- Two methods to remove deadlocks:

1) Terminate all deadlocked-processes.

- This method will definitely break the deadlock-cycle.
- However, this method incurs great expense. This is because
  → Deadlocked-processes might have computed for a long time.
  → Results of these partial computations must be discarded.

→ Probably, the results must be re-computed later.

2) Terminate one process at a time until the deadlock-cycle is eliminated.

- This method incurs large overhead. This is because after each process is aborted, deadlock-algorithm must be executed to determine if any other process is still deadlocked

- For process termination, following factors need to be considered:
  1) The priority of process.
  2) The time taken by the process for computation & the required time for complete execution.
  3) The no. of resources used by the process.
  4) The no. of extra resources required by the process for complete execution.
  5) The no. of processes that need to be terminated for deadlock-free execution.
  6) The process is interactive or batch.

Resource Preemption

- Some resources are taken from one or more deadlocked-processes.
- These resources are given to other processes until the deadlock-cycle is broken.
- Three issues need to be considered:

1) Selecting a victim

- Which resources/processes are to be pre-empted (or blocked)?
- The order of pre-emption must be determined to minimize cost.
- Cost factors includes
  1. The time taken by deadlocked-process for computation.
  2. The no. of resources used by deadlocked-process.

2) Rollback

- If a resource is taken from a process, the process cannot continue its normal execution.
- In this case, the process must be rolled-back to break the deadlock.
- This method requires the system to keep more info. about the state of all running processes.

3) Starvation

- Problem: In a system where victim-selection is based on cost-factors, the same process may be always picked as a victim.
- As a result, this process never completes its designated task.
- Solution: Ensure a process is picked as a victim only a (small) finite number of times.

8 (a) What is paging? Explain paging hardware with Translation look-aside buffer.

Paging

- Paging is a memory-management scheme.
- This permits the physical-address space of a process to be non-contiguous.

This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.

Hardware Support for Paging

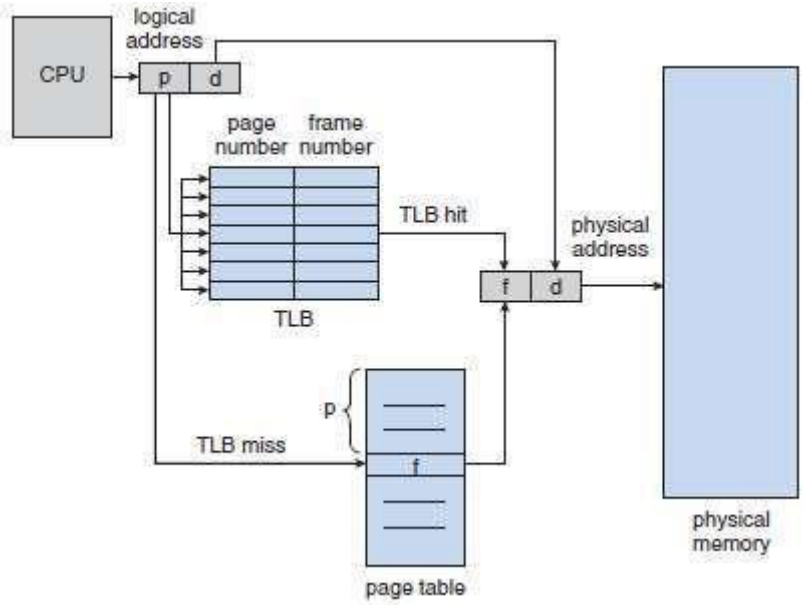
- Most OS's store a page-table for each process.
- A pointer to the page-table is stored in the PCB.

Translation Lookaside Buffer

- The TLB is associative, high-speed memory.
- The TLB contains only a few of the page-table entries.
- Working:
  - When a logical-address is generated by the CPU, its page-number is presented to the TLB.
  - If the page-number is found (**TLB hit**), its frame-number is
    → immediately available and
    → used to access memory.
  - If page-number is not in TLB (**TLB miss**), a memory-reference to page table must be made.
  - The obtained frame-number can be used to access memory (Figure 3.19).
  - In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called **hit ratio**.
- Advantage: Search operation is fast.

Disadvantage: Hardware is expensive.

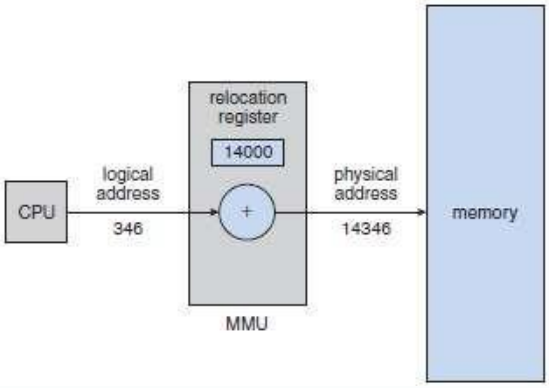
- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely
  → identify each process and
  → provide address space protection for that process.



(b) Write short note on Logical and Physical address space.

Logical versus Physical Address Space

- **Logical-address** is generated by the CPU (also referred to as virtual-address).
  Physical-address is the address seen by the memory-unit.
- Logical & physical-addresses are the same in compile-time & load-time address-binding methods. Logical and physical-addresses differ in execution-time address-binding method.
- MMU (Memory-Management Unit)
  - Hardware device that maps virtual-address to physical-address (Figure 3.11).
  - The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
  - The user-program deals with logical-addresses; it never sees the real physical-addresses.



5 a)

Banker's algorithm:

	Allocation A	B	C	Max			Available		
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

(i) Need = Max - allocation

	A	B	C
P_0	0	0	2
P_1	1	0	1
P_2	0	0	2
P_3	2	1	0
P_4	0	1	4

Work = available

ie) Work

A	B	C
1	0	2
+ 0	0	2
1	0	4
1	0	0
2	0	4

	Finish
P_0	T
P_1	T
P_2	T
P_3	T
P_4	T

Safe sequence $< P_0, P_1, P_2, P_3, P_4 >$

i) P_0 Need $\leq$ work

work = work + allocation (P_0), Finish (P_0) = T

ii) P_1 Need $\leq$ work

work = work + allocation (P_1), Finish (P_1) = T

iii) P_2 Need $\leq$ work

work = 2 0 4
+ 1 3 5
3 3 9 , Finish (P_2) = T

iv) P_3 Need $\leq$ work

work = 3 3 9
+ 6 3 2 , Finish (P_3) = T
9 6 11