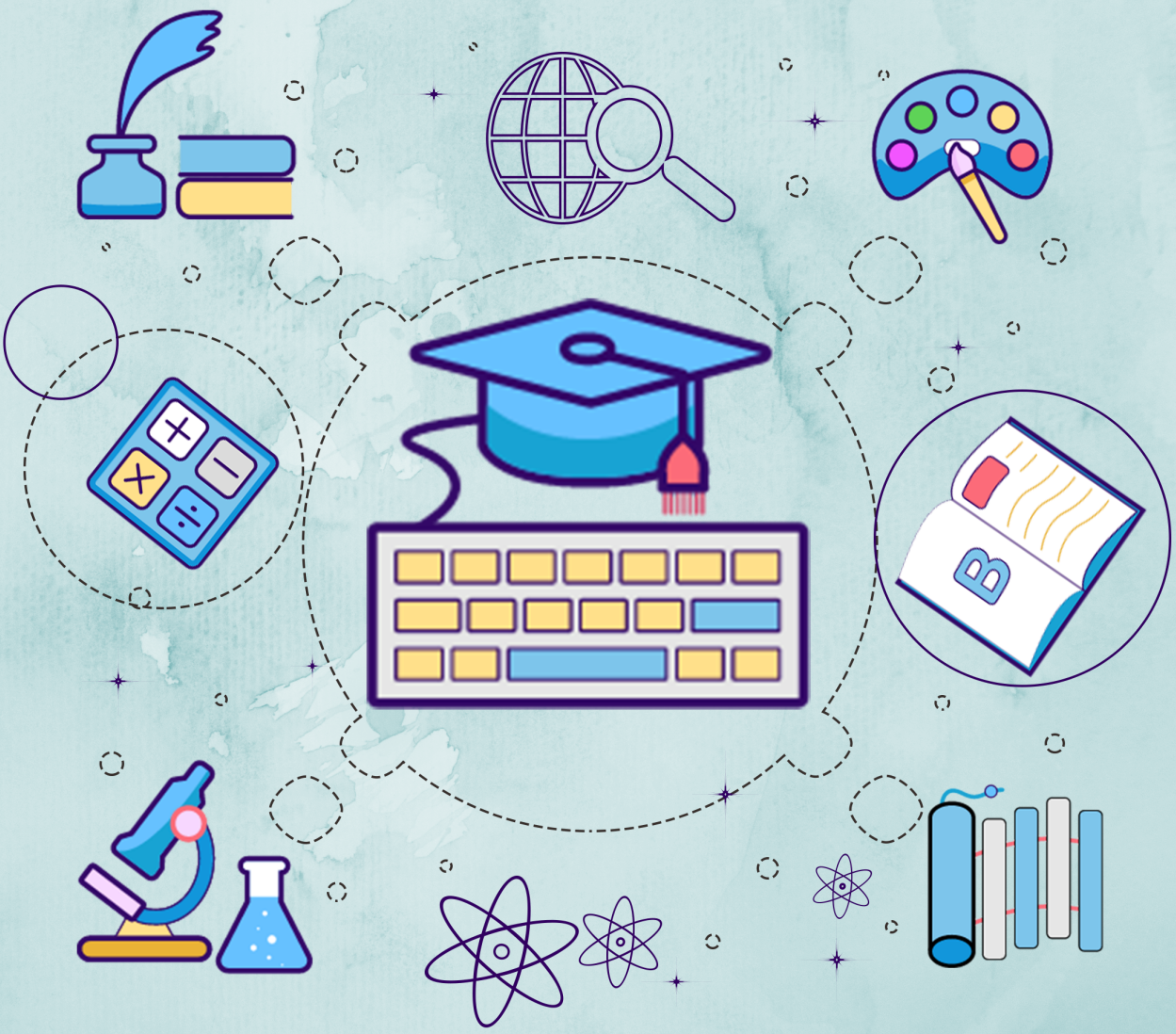
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU S4 CSE PDF NOTES

OPERATING SYSTEM (CST200)

Module 3

Related Link :

- KTU S4 CSE NOTES 2019 SCHEME
- KTU S4 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S4 CSE SOLVED
- KTU CSE TEXTBOOKS S4 B.TECH PDF DOWNLOAD
- KTU S4 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

MODULE III

PROCESS SYNCHRONISATION

CRITICAL SECTION

- 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 to be allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.

Critical Section Problem

- 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.
- The general structure of a typical process *Pi* is shown below

do
{
entry section

critical section

exit section

remainder section

} while (TRUE);

- 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.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes. At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system *(kernel code)* is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

- Two general approaches are used to handle critical sections in operating systems:
  (1) **pre-emptive kernels**
  (2) **nonpreemptive kernels.**
- A pre-emptive kernel allows a process to be pre-empted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be pre-empted.A kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

PETERSON'S SOLUTION

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Modern computer architectures perform basic machine-language instructions, such as load and store.
- There are no guarantees that Peterson's solution will work correctly on such architectures.
- We present the solution because it provides a good algorithmic description of solving the critical-section problem
- It illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered *Po* 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. With an explanation of these data structures complete. The algorithm explains below.

To enter the critical section, process *Pi* first sets flag [i] to be true and then sets turn to the value j, thereby 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.

```
Do
{

        flag [i] = TRUE;
        turn= j;
        while (flag[j] && turn j);


         critical section

        flag [i] = FALSE;

         remainder section
}
While (TRUE);
```

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

To prove that this solution is correct, it is required to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each *P;* enters its critical section only if either flag [j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true. These two observations imply that *Po* and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes -say, *Pi* - must have successfully executed the while statement , whereas *P;* had *to* execute at least one additional statement ("turn== j"). However, at that time, flag [j] == true and turn == j, and this condition will persist as long as *Pi* is in its critical section; as a result, mutual exclusion is preserved. To prove properties 2 and 3, we note that a process *P;* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn=== j; this loop is the only one possible. If *Pi* is not ready to enter the critical section, then flag [j] ==false, and *P;* can enter its critical section. If *Pj* has set flag [j] to true and is also executing in its while statement, then either turn === i or turn === j. If turn == i, then *P;* will enter the critical section. If turn== j, then *Pi* will enter the critical section. However, once *Pi* exits its critical section, it will reset flag [j] to false, allowing *P;* to enter its critical section. If *Pi* resets flag [j] to true, it must also set turn to i. Thus, since *P;* does not change the value of the variable turn while executing the while statement, *P;* will enter the critical section (progress) after at most one entry by *P1* (bounded waiting).

SYNCHRONIZATION

SYNCHRONIZATION HARDWARE

It can generally state that any solution to the critical-section problem requires a simple tool-a **lock.** Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

The critical-section problem could be solved simply in a single-processor environment if it could prevent interrupts from occurring while a shared variable was being modified. In this way, it could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically.

- Definition:

```
boolean test_and_set (boolean *target)
  {
     boolean rv = *target;
     *target = TRUE;
     return rv:
  }
```

The definition of the test and set() instruction

- Solution:

```
do{
while (test_and_set(&lock))
       ; /* do nothing */
         /* critical section */
    lock = false;
         /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test and set().

The compare and swap() instruction, operates on three operands.

- Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
 return temp;
}
```

The definition of the compare and swap() instruction.

- Solution:

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

```
    lock = 0;
      /* remainder section */
    } while (true);
```

Mutual-exclusion implementation with the compare and swap() instruction

Mutex Locks

- Simplest tool s to solve the critical-section problem is **mutex lock.**
- A process must acquire 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
- A mutex lock has a boolean variable **available** whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

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

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

```
do
{
        acquire lock

critical section

        release lock

remainder section

} while (TRUE);
```

- The definition of release() is

```
release() {
    available = true;

}
```

- Calls to either acquire() or release() must be performed atomically.
- 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.
-

SEMAPHORE

The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use. To overcome this difficulty a synchronization tool called a semaphore is used. **A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().** 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
II busy waiting
s--;

}
```

The definition of signal() is as follows:

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

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. In addition, in the 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.

Semaphore Usage

Operating systems often distinguish between counting and binary semaphores.

- The value of **a counting semaphore** can range over an unrestricted domain.
- The value of **a binary semaphore** can range only between 0 and 1.

Counting semaphores can be used to control access to a given resource consisting of a finite number of 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.

- Consider P_1 and P_2 that require S_1 to happen before S_2
  Create a semaphore "synch" initialized to 0

P1:
  S_1;
  signal(synch);
P2:
  wait(synch);
  S_2;

Semaphore Implementation

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. The process is then placed in the ready queue.

- a semaphore can be defined as follows:

```
typedef struct{

 int value;

 struct process *list;

 } semaphore;
```

- The wait() semaphore operation can be defined as

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

- the signal() semaphore operation can be defined as

```
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.

If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

Deadlocks and Starvation

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. When such a state is reached, these processes are said to be deadlocked.

- **Let *S* and *Q* be two semaphores initialized to 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.

Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore.

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process is called priority inversion.
- Solution
    - to have only two priorities
    - implement a priority-inheritance protocol- According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor type is an ADT(*Abstract data type)* that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

```
monitor monitor name
 {
 /* shared variable declarations */
 function P1(...) {
 ...
          }
function P2(...) {
 ...
          }
 . . .
function Pn(...) {
 ...
           }
 initialization code (...) {
...
          }
}
```

Syntax of a monitor.

- A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local functions.
- The monitor construct ensures that only one process at a time is active within the monitor.

Schematic view of a monitor

- A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
- The only operations that can be invoked on a condition variable are wait() and signal(). The operation **x.wait();** means that the process invoking this operation is suspended until another process invokes.The **x.signal();** operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect.

Monitor with condition variables.

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**,
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- The producer and consumer processes share the following data structures:

  int n;

  semaphore mutex = 1;

  semaphore empty = n;

  semaphore full = 0

- The pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- The producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.
- The structure of the producer process.

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

- The structure of the consumer process

```
do {
  wait(full);
  wait(mutex);
    ...
  /* remove an item from buffer to next_consumed */
    ...
  signal(mutex);
  signal(empty);
    ...
  /* consume the item in next consumed */
    ...
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do *not* perform any updates
  - Writers – can both read and write
- Problem –
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- 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
- The readers–writers problem has several variations,

The first readers–writers problem requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting

- **The second readers –writers problem** requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

- A solution to either problem may result in starvation.

- Shared Data

  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.

- The structure of a writer process

```
        do {
  wait(rw_mutex);

            ...
      /* writing is performed */

            ...

        signal(rw_mutex);

      } while (true);
```

- The structure of a 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);
```

- 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.

The Dining-Philosophers Problem

The situation of the dining philosophers

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - Need both to eat, then release both when done
- The shared data are

  semaphore chopstick[5];
- Where all the elements of chopstick are initialized to 1.
- The structure of Philosopher *i*:

```
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);
```

DEADLOCK

Process require resource, if not available then moves to waiting state. The resources that are requested are held by waiting process.

System Model

Set of resources → Logical resources [files, semaphore]
→ Physical resources [CPU, memory, printer etc]

A process must request a resource before using it must release the resource after using it.

The no. of resources requested may not exceed the total no. of resources available in the system.

Normal mode of operation :

1) Request : If the request cannot be granted immediately ie, the resource is being used by another process, then the requesting process must wait until it can acquire the resource.

2) Use : The process can operate on the resource ie, if the resource is a printer, the process can print on the printer.

3) Release : The process releases the resources.

Deadlock Characteristics

1) Necessary Conditions.

Deadlock condition hold if all the four conditions hold simultaneously :

(i) Mutual Exclusion.

- Atleast one resource must be held in non-sharable mode. ie, if only one process at a time can use the resource.
- If another process requests that resource, the requesting process must be delayed until the resource has been released.

(ii) Hold and wait.

- A process must be holding atleast one resource & waiting to acquire additional resources that are currently being held by other processes

(iii) No Preemption.

- Resources cannot be preempted ie, a resource ~~can~~ can be released only voluntarily by the process holding it, after that process has completed its task.

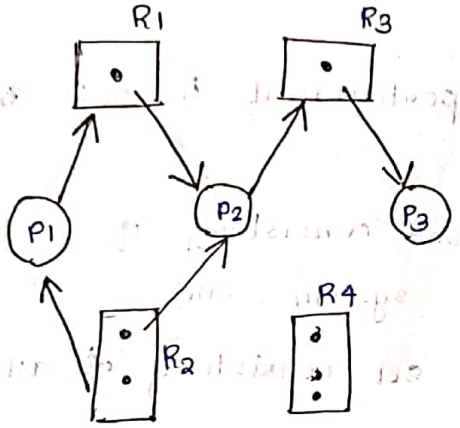
(iv) Circular wait.

- A set $\{P_0, P_1, P_2, \ldots P_n\}$ of waiting process must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots P_{n-1}$ is waiting for a resource that is held by P_n and P_n is waiting for a resource that is held by P_0.

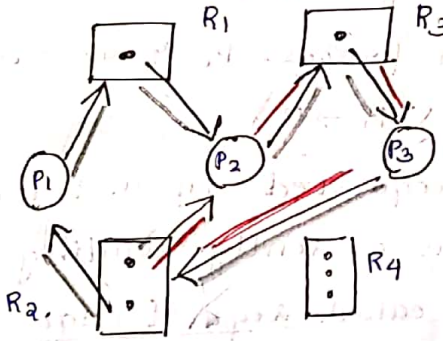
2) Resource Allocation Graph.

- Deadlocks can be described in terms of directed graph called a system resource-allocation graph.
- The graph consists of a set of vertices V & a set of edges E.
- The set of vertices V is partitioned into 2 different types of nodes:
  - P = {P1, P2, ... Pn}, the set consisting of all the active process in the system. and
  - R = {R1, R2, ... Rm}, the set consisting of all resource types in the system.
- A directed edge from process Pi to resource type Rj is denoted by Pi → Rj.
  ie, The Process Pi requested an instance of resource type Rj & is currently waiting for that resource. It is called request edge.
- A directed edge from resource type Rj to process Pi is denoted by Rj → Pi.
  ie, An instance of resource type Rj has been allocated to process Pi. It is called Assignment edge.
- Process as circle, resource as square.
- Since request type Rj has more than one instance, we represent each such instance as a dot within square.
- When Process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled the request edge is transformed to assignment edge. When the process no longer needs access to the resource it releases the resource, as a result assignment edge is deleted.

- If the graph contains no cycle, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies a deadlock has occurred



[(a) Resource Allocation graph]



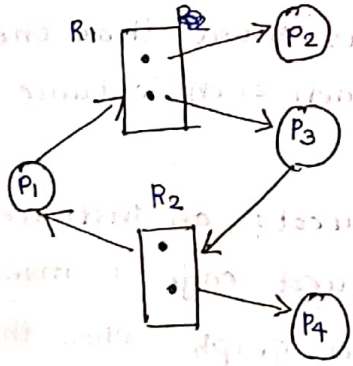
[(b) Resource Allocation graph with deadlock]

- Suppose that P_3 requests an instance of R_2. Since no resource instance is currently available, a req-edge $P_3 \rightarrow R_2$ is added to graph.
- Two minimal cycles exists,

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$.

- ✓ P_1, P_2, P_3 are deadlocked.
- ✓ P_2 is waiting for R_3 which is held by P_3.
- ✓ P_3 is waiting for either P_1 or P_2 to release R_2.
- ✓ P_1 is waiting for P_2 to release R_1.



[(c) Resource allocation graph with a cycle but no deadlock]

- A cycle has occured,

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$.

- There is no deadlock.
- P_4 may release its instance of R_2.
- That resource can then be allocated to P_3, breaking the cycle.

Methods of Handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1. Can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. Can allow the system to enter a deadlock state, detect it & recover.
3. Can ignore the problem altogether, & pretend the deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock avoidance scheme.

- Deadlock Prevention is a set of methods for ensuring that atleast one of the necessary conditions cannot hold.
- Deadlock Avoidance requires that the OS to give additional information about which resources a process will request & use.

Deadlock Prevention.

Ensure that atleast one condition among 4 doesn't hold:

1. Mutual Exclusion.

- Mutual exclusion condition must hold for non-sharable resources.
- Eg: A printer cannot be simultaneously shared by several processes.
- Sharable resources do not require mutually exclusive access & thus cannot be involved in a dead lock.

- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.

2. Hold and wait.

- Whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request & be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resource only when the process has none. A process may request some resources & use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated

We consider a process that copies data from tape drive to a disk files, sort the disk files & then prints the results to a printer. If all the resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file & printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive & disk file. It copies from the tape drive to disk, then releases both the tape drive & disk file. The process must then again request the disk file & printer. After copying the disk file to the printer, it releases these 2 resources & terminates.

3. No Preemption.

$P_i \rightarrow R_j \rightarrow P_j$

- P_i wait until P_j releases R_j

- If a process is holding some resources & requests another resources that cannot be immediately allocated it, ie, the process must wait, then all resources currently being held are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that is requesting.
- Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. $P_i \rightarrow R_j \not\rightarrow P_j \rightarrow R_K$.
  $P_i \rightarrow R_j$ ✓ then $R_j \rightarrow P_j$.
- If they are not available, we check whether they are allocated to some other process that is waiting for additional resource. If so, we preempt the desired resources from the waiting process & allocate them to the requesting process.
- If the resources are not either available or held by a waiting process, the requesting process must wait
- while it is waiting, some of its resources may be preempted but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting & recovers any resource that were preempted while it was waiting.

4. Circular Wait.

One way to ensure that this condition never holds is to impose a total ordering of all resource types & to require that each process requests resources in an increasing order of enumeration.

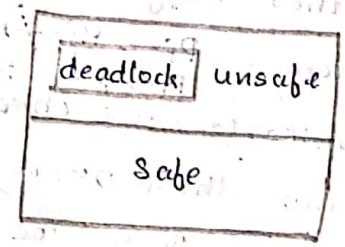
A Function $F: R \to N$, where N is the set of natural numbers.

$F(R_0) < F(R_1) < \ldots F(R_n) < F(R_0)$

Deadlock Avoidance.

- For Avoiding deadlock it is to require additional information about how resources are to be requested.
- The simplest and most useful model requires that each process declare the maximum no. of resources of each type that it may need.

1) Safe state.



- A state is safe, if the system can allocate resources to each process in some order & still avoid a deadlock.
- A system is in a safe state only if there is a safe sequence.
- A sequence of processes $< P_1, P_2, \ldots P_n >$ is a safe sequence, if for each P_i, the resources that P_i ~~can~~ ~~still~~ requests can be satisfied by the currently available resources plus the resources held by all P_j.
- If the resource that Process P_i needs are not immediately available, then P_i can wait until all P_j has finished. When they have finished, P_i can obtain all of its needed resources, complete its task, return its allocated resources & terminate. When P_i terminates, P_{i+1} can obtain its needed resource & so on.
- If no such sequence exists, then system is in unsafe.

- A safe state is not a dead lock state.
- A deadlock state is an unsafe state
- Not all unsafe states are dead locks.

Problem.

Total : 12 magnetic tape drives.

	Max. Need	Current Need (Allocated)	Needs.
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7.
		9	

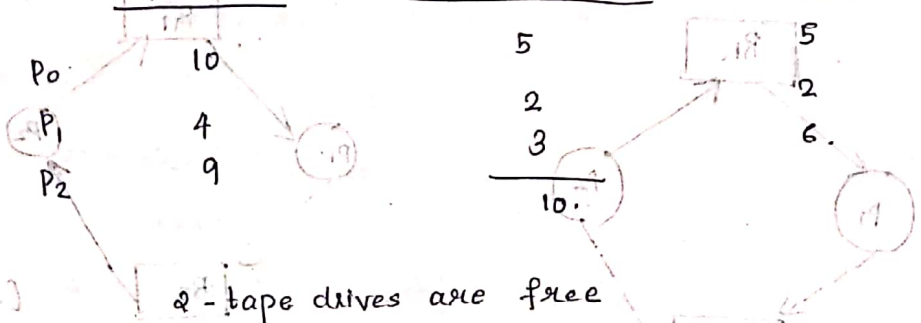
3-tape drives are free.

$\langle P_1, P_0, P_2 \rangle$ - safe state.

First P_1 is allocated, then when it releases is have 5 availdl. Then P_0 can get all its tape drives & return them, now it may have 10 available. & finally P_2 could get all 10 & it may return as 12.

Suppose if P_2 requests & is allocate 1 tape drive.

	Max. Need	Current Need	Needs
P_0	10	5	5
P_1	4	2	2
P_2	9	3	6.
		10.	



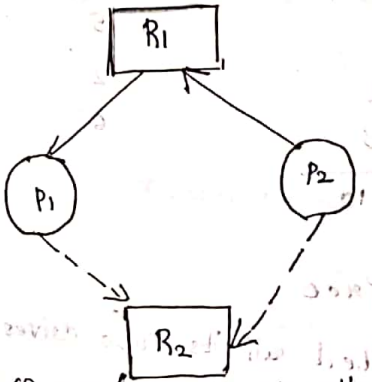
2 - tape drives are free

Here, P_1 can ~~only~~ be allocated all its tape drives. when it returns, it has 4. Since P_0 needs 5 it cannot satisfy the need of P_0 or P_2. Thus a dead lock arises.

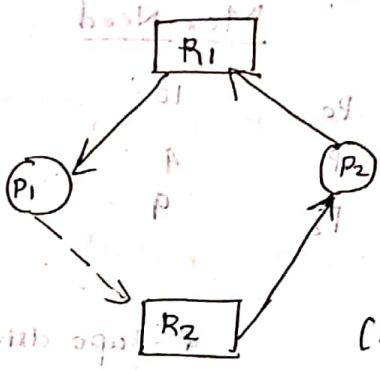
2) Resource Allocation Graph.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in future. This edge resembles a request edge in direction, but is represented by dashed lines. When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to request edge. Similarly, when a resource R_j is released by P_j, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. Suppose that process P_i requests resource R_j. The resource can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in resource allocation graph.

If no cycle exists, then the allocation of resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.



(Fig-1). (Resource allocation graph for deadlock avoidance)



(Fig-2)
(Unsafe state in resource allocation graph)

[Fig-1] - Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph.

[Fig-2] - A cycle indicates that the system is unsafe. If P_1 request R_2 & P_2 requests R_1, then a deadlock will occur.

Banker's Algorithm.

↳ Available [j] = k

- k instance of resource type j available.

↳ Max [i,j] ~~= Pi~~.

- Pi require atmost k instance of resource type R

↳ Allocation [i,j] = k.

- Process Pi is currently allocated k instance of resource type Rj.

↳ Need [i,j]

- Process Pi need k more instance of resource type Rj to complete its task.

Need [i,j] = Max [i,j] − Allocation [i,j].

Safety Algorithm.

1) Let work & finish be vectors of length m & n resp.

Initialize work := Available &

Finish [i] := false, for i = 1, 2, ... n.

2) Find an i such that both.

(a) Finish [i] = false

(b) Need i ≤ work.

If no such i exists then go to step 4.

3) work := work + Allocation i.

Finish [i] := true

Go to step 2.

4) If Finish [i] = true for all i, then the system is in a safe state.

Resource Request Algorithm.

1.) If Request ≤ Needi, go to step 2.
Otherwise raise an error condition, since the process has executed its maximum claim.

2.) If Requesti ≤ Available, go to step 3.
Otherwise Pi must wait, since the resource are not available.

3.) Have the system pretend to have allocated the requested resource to process Pi by modifying the state as follows:

- Available := Available − Request;
- Allocation := Allocation + Request;
- Needi := Needi − Request;

Problem

4)

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Soln:

Need [Max - Allocation]

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Safety Algorithm

Need ≤ Avail

① P_0 743 ≤ 332 ✗

② P_1 122 ≤ 332 ✓

Avail = 332 + 200 = 532

③ P_2 600 ≤ 532 ✗

④ P_3 011 ≤ 532 ✓

Avail = 532 + 211 = 743

⑤ P_4 431 ≤ 743 ✓

Avail = 743 + 002 = 745.

⑥ P_0 743 ≤ 745 ✓

Avail = 745 + 010 = 755.

⑦ P_2 600 ≤ 755 ✓

∴ Avail = 755 + 302 = 1057

∴

A	B	C
10	5	7

∴ < P_1, P_3, P_4, P_0, P_2 > is safe.

b) Request P_1 comes for 102.

Request Algorithm.

① 102 ≤ 122 [Req ≤ need]

② 102 ≤ 332 [Req ≤ Avail]

③ Avail := 332 − 102 = 230 [Avail := Avail − Req]

④ Allocat = 200 + 102 = 302 [Allo := Alloc + Req]

⑤ Need = 122 − 102 = 020 [Need := Need − Req]

	Alloc	Max	Avail	Need
P_1	302	322	230	020.

Part [1]

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	2	3	0	7	4	3
P_1	3	0	2	8	2	2				0	2	0
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1
	8	2	7									

① P_0 . $743 \leq 230$ ✗ .

② P_1 $020 \leq 230$ ✓

$230 + 302 = 532$.

③ P_2 ~~020~~ $600 \leq 532$ ✗ .

④ P_3 $011 \leq 532$ ✓

$532 + 211 = 743$.

⑤ P_4 $431 \leq 743$ ✓

$743 + 002 = 745$.

⑥ P_0 . $743 \leq 745$ ✓

$745 + 010 = 755$.

⑦ P_2 $600 \leq 755$ ✓

$755 + 302 = 1057$ (A B C)

$\therefore \langle P_1, P_3, P_4, P_0, P_2 \rangle$ is safe

Ob Req. P_4 (3 3 0) & Req . P_0 (0 2 0) comes .

Req. Algorithm .

① $330 \leq 431$ ✓ $330 \leq 230$ ✗ .

② $020 \leq 743$ ✓ $020 \leq 230$ ✓ .

Avail = 230 − 020 = 210

All = 010 + 020 = 030 .

Need = 743 − 020 = 723 .

	Allocation			Need			Avail		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	2	3	2	1	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

① P_0 $723 \leq 210$ ✗

② P_1 $020 \leq 210$ ✗

③ P_2 $600 \leq 210$ ✗

④ P_3 $011 \leq 210$ ✗

⑤ P_4 $431 \leq 210$ ✗

Undo the changes, since it is unsafe

So change as in Part [1]

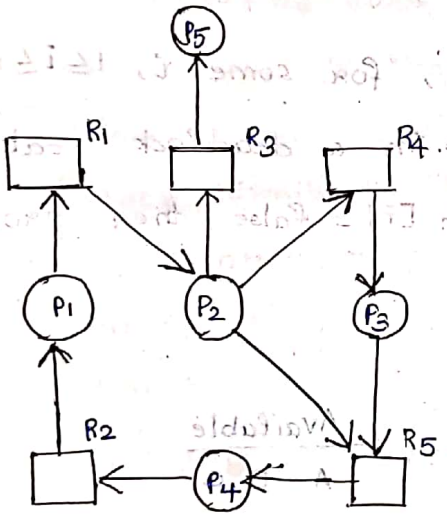
Deadlock Detection.

Determine whether deadlock has occured.

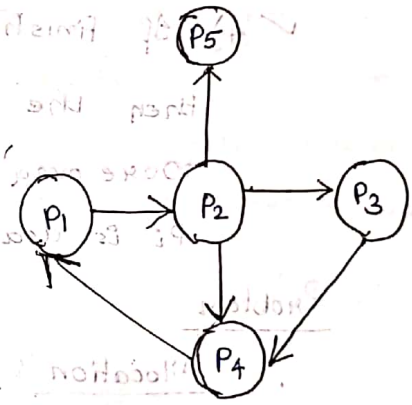
1.) Single Instance of Each Resource Type.

If all the resources have only a single instance, then we can define a deadlock detection algorithm. called a wait-for graph.

An Edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if & only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ & $R_q \rightarrow P_j$ for some resoura R_q.



(a) Resource-allocation graph



(b) Corresponding wait-for graph.

2.) Several Instance of a Resource Type.

The wait-for graph scheme is not applicable to a resource allocation system with multiple instance of each resource type.

Deadlock detection Algorithm

1.) Let work & finish be vectors of length m & n respectively

Initialize work := Available, for i = 0, 1, 2, ...,

If Allocation ≠ 0, then finish [i] = false

otherwise

finish [i] = true.

2.) Find an index i such that both:

(a) finish [i] = false.

(b) Request$_i$ ≤ work.

If no such i exists, go to step 4

3.) work := work + Allocation.

finish [i] := true, go to step 2.

4.) If finish [i] = false, for some i, $1 \le i \le n$, then the system is in a deadlock state. Moreover, if finish [i] = false, then process P_i is deadlocked.

Problem:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

P_0. 000 ≤ 000 ✓

000 + 010 = 010.

P_1 202 ≤ 010 ×.

P_2 000 ≤ 010 ✓

010 + 303 = 313.

P_3 100 ≤ 313 ✓

313 + 211 = 524

P_4. 002 ≤ 524 ✓

524 + 002 = 526

526 + 200 = 726

< P_0, P_2, P_3, P_4, P_1 > ∴ finish [i] = true

Change P_2 < 001 >.

P_0 000 ≤ 000 ✓

000 + 010 = 010.

P_1 202 ≤ 010 ×.

P_2 001 ≤ 010 ×.

P_3 100 ≤ 010 ×.

P_4 002 ≤ 010 ×

Recovery from Deadlock

1) Process Termination

✓ Abort all deadlocked processes.

- This method clearly will break the deadlock cycle, but at a great expense.

✓ Abort one process at a time until deadlock cycle is eliminated.

- This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job.

Criteria for process selection :

✓ what the priority of the process is.

✓ How long process was computed.

✓ How many & what type of resource used.

✓ How many more resources needed.

✓ How many process will need to be terminated.

2) Resource Preemption

Issues regarding resource preemption:

✓ Selecting a victim.

- Which resources & which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost

✓ Roll back.

- If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state & restart it from that state.

✓ Starvation.

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation arises.