Report5

1. What is deadlock?
    - 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.
2. What are the requirements of deadlock?
    - Mutual exclusion.
        - At least one resource must be held in a nonsharable mode; that is, 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.
    - Hold and wait.
        - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
    - No preemption.
        - Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
    - Circular wait.
        - A set { P0 , P1 , ..., Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
3. What's different between deadlock prevention and deadlock avoidance?
    - deadlock prevention
        - Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.
        - Non-blocking synchronization algorithms and serializing tokens are some deadlock prevention algorithms.
        - In deadlock prevention, all resources are requested at once.
    - deadlock avoidance
        - Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait.
        - Banker's algorithm is the most common deadlock avoidance algorithm.
        - In deadlock avoidance, the requests for resources are manipulated until at least one safe path is found.
4. How to prevent deadlock? Give at least two examples.

- Eliminate Hold and wait
  - To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
    - Either of the methods described above can lead to starvation if a process requires one or more popular resources.
  - For example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.
- Eliminate No Preemption
  - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
  - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
  - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.
- Eliminate Circular Wait
  - One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
  - In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
  - One big challenge in this scheme is determining the relative ordering of the different resources
  - For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

5. Which way does recent UNIX OS choose to deal with deadlock problem, why?

- Ignore the problem and pretend that deadlocks never occur in the system.
- This is also an application of the Ostrich algorithm. In computer science, the ostrich algorithm is a strategy of ignoring potential problems on the basis that they may be exceedingly rare. It is named for the ostrich effect which is defined as "to stick one's head in the sand and pretend there is no problem". It is used when it is more cost-effective to allow the problem to occur than to attempt its prevention.

- Because the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

6. What data structures you use in your implementation（of Banker's algorithm）? Where and why you use them? Are they optimal for your purpose?

    - Data structures: map, vector
    - Where?
        - vector<int> resource;
            - the maximum quantity of each resource
        - vector<int> available;
            - the available quantity of each resource
        - map<int, vector<int> > need;
            - the need of each resource about the process
        - map<int, vector<int> > allocation;
            - allocated resource about the process
        - map<int, vector<int> > maxNeed;
            - the max need of each resource about the process
        - vector<int> request;
            - each command's parameters
    - Why?
        - vector
            - Not sure about the number of commands and resource
        - map
            - The pid is not a continuous space, list is not suitable.
            - Map will quickly query whether the key exists and the corresponding value.
    - Optimal?
        - No