1. (1 pt.)

- Running state – a process is using CPU for ALU operation
- Ready state – a process waiting for CPU (shot term scheduler will assign CPU for it)
- Blocked state – a process waiting for I/O finish
- Transaction 1 – a process need I/O
- Transaction 2 – a process used up it's time quantum (time out)
- Transaction 3 – a scheduler select a process from ready queue and let it use CPU
- Transaction 4 – a process get input, a process need wait for scheduler

2. (1 pt.)

a. Race condition –A situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.

b. Mutual exclusion of critical section – only one process can access shared resources at any moment.

c. Zombie process – when a child process terminate if parent does not call wait() or waitpid() to get child process's termination status, child will be remain as a zombie.

d. A process can create a child process by using fork() or vfork() system call. Discuss two main differences between two child created by fork() and vfork().

A child with fork(): has it's own address space. Only share text segment with its parent.
A child with vfork(): memory space is shared with its parent. A parent always wait for the child.

3. (1 pt.)

```
#include <stdio.h>
#include <unistd.h>

extern char **environ;
int main(int argc, char *argv[])
{
      char **p = environ;
      while (*p != NULL)
     {
         printf("%s\n", *p);
         *p++;
     }
      return 0;
}
```

```
#include <stdio.h>

int main(int argc, char *argv[],
char *envp[])
{
     char **p = envp;

    while (*p != NULL)
   {
       printf("%s \n", *p);
      *p++;
   }

     return 0;
}
```

4. (1 pt.)

```
#include  <stdio.h>
#include  <string.h>
#include <unistd.h>
#include  <sys/types.h>

int  main(void)
{
     int pid;
     pid=fork();
     if(pid==0)
     {
           while (1)
            ;
     }
     else
     {
       exit(0);
     }
}
```

5. (3.5 pt.)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    pid_t child_pid, grandchild_pid;

    child_pid = fork(); // create a child
    if (child_pid == 0) // Child Process
    {
       grandchild_pid = fork(); // create grandchild

        if (grandchild_pid == 0)  //Grandchild Process Code
        {
            while (1)
            {
                printf("grandchild is running\n");
                sleep(1);
                if (getppid() == 1) //means parent process is terminated
                {
                    printf("Grandchild: my parent is gone. I will exit.\n");
                    exit(0);
                }
            }
        }
        else //Child process Code
        {
             for (int i = 0; i < 20; i++)
             {
                printf("child is running\n");
                sleep(1);
             }
            printf("Child: done with my work, exiting now.\n");
            exit(0);
        }
    }
    else //Parent process code
    {
        while (1)
        {
            int status;
            pid_t done = waitpid(child_pid, &status, WNOHANG); //waiting for child termination

            if (done == 0) //means child is still alive
            {
                printf("parent is running\n");
                sleep(1);
            }
           else
            {
                printf("Parent:I finish my job since all of my descendant terminated \n");
                break;
            }
        }
    }

    return 0;
}
```

6. (2.5 pt.)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
      int i,n,m ;
      char * buffer, ch[20];

      i=atoi(argv[1]);

      buffer = (char*) malloc (i+1);
      if (buffer==NULL)
            exit (1);

      for (n=0; n<i; n++)
            buffer[n]=rand()%26+'a';
      buffer[i]='\0';
      printf ("Random string: %s\n",buffer);

      write (1,"integer for extension?", 22);
      read(0, ch, 20);
      m=atoi(ch);
      buffer = (char*) realloc(buffer, (i+m)*sizeof (char));

      for (n = i;  n< i+m; n++)
            buffer[n]=rand()%26+'a';
       buffer[i+m]='\0';
        printf ("Extended Random string: %s\n", buffer);

      free (buffer);

      return 0;
}
```