**Slip 1 or slip5/13**  
Q1. write prg that demonstrated use of nice()system call after child process is started using fork(). assign high priority to child using nice()

#include <stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
int main()  
{  
 pid\_t pid;  
 int nice\_value=-10; // lower nice value means higher priority  
 pid= fork();  
 if(pid==0)  
 {  
 //this code is executed by child process  
 printf("child process(PID %d)is running with normal priority.\n",getpid());  
 }  
 else if(pid>0)  
 {  
 //this code executed by parent process  
 printf("parent process(PID %d) is running\n",getpid());  
 //adjust priority of child process using nice() call  
 if(nice(nice\_value)==-1)  
 {  
 perror("nice");  
 exit(EXIT\_FAILURE);  
 }  
 printf("parent process has adjusted the child process priority to be higher\n");  
 }  
 else  
 {  
 perror("fork");  
 exit(EXIT\_FAILURE)  
 }  
 //add sleep to keep process running for a while  
 sleep(5);  
 return 0;  
}

Output: parent process(PID 37713) is running. child process(PID 37714)is running with normal priority.

**SLIP2/SLIP11 Q1**

**HII /HELLO WORLS MSG**

//create a child process using fork() display parent and child process id child process display the msg hello world and parent display hii msg  
#include <stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
int main()  
{  
 pid\_t pid;  
 pid=fork();  
 if(pid==0)  
 {  
 printf("child process ID:%d\n",getpid());  
 printf("hello world");  
 }  
 else if(pid>0)  
 {  
 printf("parent process id:%d\n",getpid());  
 printf("hii\n");  
 }  
 else  
 {  
 perror("fork");  
 return 0;  
  
 }  
}

**parent process id:46472**

**hii**

**child process ID:46473**

**hello world**

**SLIP3 exec system call- process termination**

# include <stdio.h>  
# include <stdlib.h>  
# include <unistd.h>  
# include <sys/wait.h>  
int main()  
{  
 pid\_t pid;  
pid = fork();  
if (pid < 0)  
{  
perror("Fork failed");  
exit(1);  
}  
else if (pid == 0) {  
printf("Child Process ID: %d\n", getpid());  
printf("Parent of Child Process ID: %d\n", getppid());  
  
// Replace child process with a new process using execl()  
execl("/bin/ls", "ls", "-l", NULL);  
  
// If execl() fails, print error and exit  
perror("execl failed");  
exit(1);  
} else {  
// Parent process  
printf("Parent Process ID: %d\n", getpid());  
  
// Wait for child process to finish  
wait(NULL);  
printf("Child process terminated. Control returned to parent.\n");  
}  
return 0;  
}

**SLIP 4/slip10/12 Q1**

**ORPHAN**

//illustrate orphan prg  
#include <stdio.h>  
#include<sys/types.h>  
#include<unistd.h>  
int main()  
{  
 int pid=fork();  
 if(pid>0)  
 {  
 printf("parent process ");  
 printf("ID:%d\n",getpid());  
 }  
 else if(pid==0)  
 {  
 printf("child process ");  
 printf("id:%d\n",getpid());  
 printf("parent ID:%d\n",getppid());  
   
 sleep(10);  
 printf("child process ");  
 printf("id:%d\n",getpid());  
 printf("parent ID:%d\n",getppid());  
 }  
 else  
 {  
 printf("failed to create child process:");  
 }  
 return 0;  
}

**parent process ID:21347**

**child process id:21348, parent ID:21347**

**SLIP6, 14 ,16 clock() system call**

#include<stdio.h>  
#include<time.h>  
int main()  
{  
clock\_t start,end;  
double cpu\_time\_used;  
int i;  
start=clock();  
for(i=0;i< 1000000;i++)  
end = clock();  
cpu\_time\_used = ((double) (end-start));  
printf("Execution time: %f seconds\n",cpu\_time\_used);  
return 0;  
}

**O/p:Execution time: 810348.000000 seconds**

**SLIP7 ,9,15,19,20 Q1 execl()**

#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
int main()   
{  
 pid\_t pid;  
 # Fork a child process  
 pid = fork();  
  
 if (pid < 0)   
 { // Fork failed  
 perror("Fork failed");  
 exit(EXIT\_FAILURE);  
 }   
 else if (pid == 0)   
 { # Child process  
 printf("Child process (PID: %d) is executing 'ls' command using execl().\n", getpid());  
  
 # Execute the "ls" command  
 execl("/bin/ls", "ls", "-l", (char \*)NULL);  
# If execl() fails  
 perror("execl() failed");  
 exit(EXIT\_FAILURE);  
 }   
 else { # Parent process  
 printf("Parent process (PID: %d) is going to sleep.\n", getpid());  
# Parent process goes to sleep  
 sleep(5);  
## Wait for the child process to complete  
 wait(NULL);  
 printf("Parent process (PID: %d) has woken up and child process completed.\n", getpid());  
 }  
 return 0;  
}

**Parent process pid655 is going to sleep**

**Child process pid 654 is executing ls command using execl()**

**Total=0 parent process 655 is woken up and child process completed**

**Slip 8,18 Q1 need matrix content display**

# include <stdio.h>  
int main()  
{  
int num\_processes, num\_resources;  
# Accept the number of processes and resources  
printf("Enter the number of processes: ");  
scanf("%d", & num\_processes);  
  
printf("Enter the number of resources: ");  
scanf("%d", & num\_resources);  
  
int max[num\_processes][num\_resources];  
int allocation[num\_processes][num\_resources];  
int need[num\_processes][num\_resources];  
  
# Accept the Max matrix  
printf("Enter the Max matrix (%d x %d):\n", num\_processes, num\_resources);  
for (int i = 0; i < num\_processes; i++)  
{  
for (int j = 0; j < num\_resources; j++)  
{  
 scanf("%d", & max[i][j]);  
}  
}  
  
#Accept the Allocation matrix  
printf("Enter the Allocation matrix (%d x %d):\n", num\_processes, num\_resources);  
  
for (int i = 0; i < num\_processes; i++)  
{  
for (int j = 0; j < num\_resources; j++)  
{  
scanf("%d", & allocation[i][j]);  
}  
}  
  
# Calculate the Need matrix  
for (int i = 0; i < num\_processes; i++)  
{  
for (int j = 0; j < num\_resources; j++)  
{  
need[i][j] = max[i][j] - allocation[i][j];  
}  
}  
  
# Display the Need matrix  
printf("The Need matrix is:\n");  
for (int i = 0; i < num\_processes; i++)  
{  
for (int j = 0; j < num\_resources; j++)  
{  
printf("%d ", need[i][j]);  
}  
printf("\n");  
}  
return 0;  
}

**O/P:**

Enter the number of processes: 2

Enter the number of resources: 2

Enter the Max matrix (2 x 2):

7

8

6

5

Enter the Allocation matrix (2 x 2):

0

0

1

2

The Need matrix is:

7 8

5 3

**Slip 17 Q1 deadlock resources**

# include <stdio.h>  
int main()  
{  
 int processes, max\_resources, available, i, j;  
#Input the number of processes and maximum resources  
printf("Enter number of processes: ");  
scanf("%d", & processes);  
printf("Enter the number of resource instances: ");  
scanf("%d", & max\_resources);  
  
int max[processes], allocated[processes];  
  
#Input maximum and allocated resources for each process  
printf("Enter the maximum resources required by each process:\n");  
for (i = 0; i < processes; i++) {  
printf("Process %d: ", i);  
scanf("%d", & max[i]);  
}  
  
printf("Enter the currently allocated resources for each process:\n");  
for (i = 0; i < processes; i++) {  
printf("Process %d: ", i);  
scanf("%d", & allocated[i]);  
}  
# Calculate the minimum number of resources needed to avoid deadlock  
int total\_max = 0, total\_allocated = 0;  
for (i = 0; i < processes; i++) {  
total\_max += max[i];  
total\_allocated += allocated[i];  
}  
int min\_resources\_needed = total\_max - total\_allocated;  
printf("Minimum additional resources needed to avoid deadlock: %d\n", min\_resources\_needed);  
return 0;  
}

**O/P: Enter number of processes: 2**

**Enter the number of resource instances: 3**

**Enter the maximum resources required by each process:**

**Process 0: 5**

**Process 1: 6**

**Enter the currently allocated resources for each process:**

**Process 0: 1**

**Process 1: 2**

**Minimum additional resources needed to avoid deadlock: 8**