WEEK – IV
Process & Threads (II)

1. Replace the current process in memory and execute a new process

exec() set of functions

The exec subroutine, in all its forms, executes a new program in the calling process. The exec subroutine does not create a new process, but overlays the current program with a new one, which is called the new-process image. The new-process image file can be one of three file types:

- An executable binary file.
- An executable text file that contains a shell procedure (only the execlp and execvp)
- A file that names an executable binary file or shell procedure to be run.

There are various forms of the exec functions:

- The **l** form of the exec functions (**execl...()**) contain an argument list that's terminated by a NULL pointer. The argument **arg0** should point to a filename that's associated with the program being loaded.
- The **v** form of the exec functions (**execv...()**) contain a pointer to an argument vector. The value in **argv[0]** should point to a filename that's associated with the program being loaded. The last member of **argv** must be a NULL pointer. The value of **argv** cannot be NULL, but **argv[0]** can be a NULL pointer if no argument strings are passed.
- The **p** form of the exec functions (**execlp...(), execvp...()**) use paths listed in the PATH environment variable to locate the program to be loaded, provided that the following conditions are met:
    - The argument file identifies the name of program to be loaded.
    - If no path character (/) is included in the name, an attempt is made to load the program from one of the paths in the PATH environment variable.
    - If PATH isn't defined, the current working directory is used.
    - If a path character (/) is included in the name, the program is loaded as in the following point.

The difference between **execl*** and **execv*** is the argument passing. **execl*** require a list of arguments while **execv*** require a vector of arguments.

A list of arguments is useful if you know all the arguments at compile time and so, **execl*** functions can be used.

In case the arguments are to be entered by the user, you must construct a vector of arguments at run time, so **execv*** functions should be used.

The functions with suffix **p** use the PATH environment variable to find the program (e.g.: "ls"), without this you must specify the full path (either absolute or relative to the current directory, e.g.: "/bin/ls").

Below are two examples, first program uses the p option, second program does not. In both programs the child process executes the **execl*** functions, and the parent process executes the **execv*** functions.

exec1.c

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

int main(){
    pid_t pid;
    char *args[3], a[3];
    int l, i;
    char *x;

    if ((pid = fork()) == 0) { // child process
      system ("echo CHILD executing LS:");
      printf("\nI am Child, my ID %d, MY PARENT ID =
%d",getpid(),getppid());
      printf("\n");
      execlp("ls", "ls", "-l", (char*)NULL);
      exit(0);
    }

    else if (pid > 0) { // parent process
      wait(NULL);
      system ("echo PARENT executing User Input Command:");
      printf("\n\nI am Parent, my ID %d, I have child with ID: %d, My
ParentID = %d", getpid(), pid, getppid());

      printf ("\nEnter the command: ");
      scanf("%s", a);
      l = strlen(a);
      x = (char *)malloc(l + 1);
      strcpy(x, a);
      args[0] = x;
```

```
      printf ("\nEnter the argument for %s: ", args[0]);
      scanf("%s", a);
      l = strlen(a);
      x = (char *)malloc(l + 1);
      strcpy(x, a);
      args[1] = x;

      args[2] = NULL;

      execvp(args[0], args);
      exit(0);
    }

   else printf("Problem in child creation....\n");
   return 0;
}
```

exec2.c

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

int main(){
    pid_t pid;
    char *args[4], a[3];
    int l, i;
    char *x;

    if ((pid = fork()) == 0) { // child process
      system ("echo CHILD executing LS in current directory:");
```

```
      printf("\nI am Child, my ID %d, MY PARENT ID =
%d",getpid(),getppid());
      printf("\n");
      execl("ls", "ls", "5", "10", (char*)NULL);
      exit(0);
    }

    else if (pid > 0) { // parent process
      wait(NULL);
      system ("echo PARENT executing User Input Command in current
directory:");
      printf("\n\nI am Parent, my ID %d, I have child with ID: %d, My
ParentID = %d", getpid(), pid, getppid());

      printf ("\nEnter the command: ");
      scanf("%s", a);
      l = strlen(a);
      x = (char *)malloc(l + 1);
      strcpy(x, a);
      args[0] = x;

      printf ("\nEnter the 1st argument for %s: ", args[0]);
      scanf("%s", a);
      l = strlen(a);
      x = (char *)malloc(l + 1);
      strcpy(x, a);
      args[1] = x;

      printf ("\nEnter the 2nd argument for %s: ", args[0]);
      scanf("%s", a);
      l = strlen(a);
      x = (char *)malloc(l + 1);
      strcpy(x, a);
      args[2] = x;
```

```
args[3] = NULL;

      execv(args[0], args);
      exit(0);
    }

   else printf("Problem in child creation....\n");
   return 0;
}
```

2. Thread Mutex

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depend on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

mut1.c

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void *func(void *arg){
   pthread_t *thd_id = (pthread_t*)arg;
   pthread_t sid;

   pthread_mutex_lock( &mutex1 );
    sid = pthread_self();
    printf("Hello from new thread %lu got %lu!\n",sid,*thd_id);
    counter++;
    printf("Counter value: %d\n",counter);
```

```
pthread_mutex_unlock( &mutex1 );

   pthread_exit(NULL);
}

int main(){
   int rc1, rc2;
   pthread_t tid1, tid2, sid;

   /* Create independent threads each of which will execute func*/

   if( (rc1=pthread_create( &tid1, NULL, &func, (void *)&tid1)))
   {
      printf("Thread creation failed: %d\n", rc1);
      exit (1);
   }

   if( (rc2=pthread_create( &tid2, NULL, &func, (void *)&tid2)))
   {
      printf("Thread creation failed: %d\n", rc2);
      exit (1);
   }

   /* Wait till threads are complete before main continues. Unless we
*/
   /* wait we run the risk of executing an exit which will terminate
*/
   /* the process and all threads before the threads have completed.
*/

   pthread_join( tid1, NULL);
   pthread_join( tid2, NULL);

   return 0;
}
```

which has the following desired output

```
Hello from new thread 140511271483136 got 140511271483136!
Counter value: 1
Hello from new thread 140511263090432 got 140511263090432!
Counter value: 2
```

Compare the above result with the situation when the **`pthread_mutex_lock( &mutex1 );`** and **`pthread_mutex_unlock( &mutex1 );`** lines are commented out.

mut2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

void *func(void *arg){
   pthread_t *thd_id = (pthread_t*)arg;
   pthread_t sid;

   //pthread_mutex_lock( &mutex1 );
    sid = pthread_self();
    printf("Hello from new thread %lu got %lu!\n",sid,*thd_id);
    counter++;
    printf("Counter value: %d\n",counter);
   //pthread_mutex_unlock( &mutex1 );

   pthread_exit(NULL);
}

int main(){
   int rc1, rc2;
   pthread_t tid1, tid2, sid;
```

```
   /* Create independent threads each of which will execute func*/

   if( (rc1=pthread_create( &tid1, NULL, &func, (void *)&tid1)))
   {
      printf("Thread creation failed: %d\n", rc1);
      exit (1);
   }

   if( (rc2=pthread_create( &tid2, NULL, &func, (void *)&tid2)))
   {
      printf("Thread creation failed: %d\n", rc2);
      exit (1);
   }

   /* Wait till threads are complete before main continues. Unless we
*/
   /* wait we run the risk of executing an exit which will terminate
*/
   /* the process and all threads before the threads have completed.
*/

   pthread_join( tid1, NULL);
   pthread_join( tid2, NULL);

   return 0;
}
```

Which now may give the output as shown

```
❯ ./mut2
Hello from new thread 140313201211136 got 140313201211136!
Counter value: 1
Hello from new thread 140313192818432 got 140313192818432!
Counter value: 2
❯ ./mut2
Hello from new thread 139835634665216 got 139835634665216!
Hello from new thread 139835626272512 got 139835626272512!
Counter value: 2
Counter value: 1
```

The second situation above is of interest, which shows an out of order inconsistent behaviour.

3. Classwork

a) Write a program [named **demo.c**] that accepts two integers (**low, high**) as command line argument. **demo.c** in turn should call two other programs **pro1** and **pro2**.

pro1 should calculate the summation of all integers between (**low, high**) as **sum_res**.

pro2 should evaluate whether **sum_res** is prime or not.

b) Write a program to illustrate the **Bounded Buffer** or **Producer/Consumer** problem.

4. Home Assignment

a) Write a multi-threaded program such that it creates 10 more threads. Each thread should print ***Hello from nth thread*** along with the argument received from the main thread. Each thread should return its own id and a unique value to main using **pthread_exit()**. Main thread should be able to print this returned message identifying which thread ended within its own thread. Use mutex for synchronization and compare the results with the previous week's assignment.

i) Synchronize it such that all 10 threads are created first and then all of them are terminated.

Eg:
```
0 MAIN
        1 created
        2 created
        3 created
        .
        .
        .
        .
        10 created
        1 ended
        2 ended
        3 ended
        .
        .
        .
        .
        10 ended
    MAIN ended
```

ii) Synchronize it such that one is created and ends before the next one is created (till all 10 are created and terminated)

Eg:
```
0 MAIN
    1 created
    1 ended
    2 created
    2 ended
    3 created
    3 ended
    .
    .
    .
    .
    10 created
    10 ended
MAIN ended
```

iii) Synchronize it such that a pair is created at a time and terminated in the reverse order of how they were created.

Eg:
```
0 MAIN
    1 created
    2 created
    2 ended
    1 ended
    3 created
    4 created
    4 ended
    3 ended
    .
    .
    .
    .
    9 created
    10 created
    10 ended
    9 ended
MAIN ended
```