Advance C Programming

Weekly Assessment - Module 3 Assessment

VIVEK MUNNAA D

1. Which signals are triggered, when the following actions are performed

1. **user press Ctrl+C**

Ans: SIGINT(2)

2. **kill() system call is invoked**

Ans: SIGKILL(9), SIGTERM(15)

3. **CPU tried to execute an illegal instruction**

Ans: SIGILL(4)

4. **When the program access the unassigned memory**

Ans: SIGSEGV(11)

---

2. List the gdb command for the following operations

1. **To run the current executable file**

Ans: run or r

2. **To create breakpoints at**

Ans: break or b

Syntax: break <line_number> or b <line_number> or

break <function_name> or break <file_name>:<line_number>

3. **To resume execution once after breakpoint**

Ans: continue

4. **To clear break point created for a function**

Ans: clear <function_name>

5. **Print the parameters of the function in the backtrace**

Ans: backtrace

frame <frame_number>

info args

---

3. Guess the output for the following program.

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

int main()
{
   if (fork() && (!fork())) {
      if (fork() || fork()) {
         fork();
      }
   }
   printf("2 ");
   return 0;
}
```

Ans: 2 2 2 2 2 2

---

4. Guess the output for the following program.

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

int main()
{
   if (fork()) {
      if (!fork()) {
         fork();
         printf("1 ");
      }
      else {
         printf("2 ");
      }
   }
   else {
      printf("3 ");
   }
   printf("4 ");
   return 0;
}
```

Ans: 2 4 1 4 1 4 3 4

(or) 2 4 3 4 1 4 1 4

(or) 3 4 2 4 1 4 1 4

Explanation: This program may provide different outputs on execution because the order in which the numbers get printed depends upon the parent and child processes whose orderly execution are uncertain. Hence, we may get 2 4 1 4 1 4 3 4 on the first time of execution and the same or one of the different outputs listed above on the subsequent executions.

---

5. Create two thread functions to print hello and world separately and create threads for each and execute them one after other in C

main.c

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

void *hello(void *arg){
    printf("Hello ");
    pthread_exit(NULL);
}

void *world(void *arg){
    printf("World ");
    pthread_exit(NULL);
}

int main(){
    pthread_t t1,t2;
    pthread_create(&t1,NULL,hello,NULL);
    pthread_join(t1,NULL);
    pthread_create(&t2,NULL,world,NULL);
    pthread_join(t2,NULL);
    return 0;
}
```

```
Hello World

...Program finished with exit code 0
Press ENTER to exit console.
```

6. How to avoid Race conditions and deadlocks?

Ans: To Avoid Race Conditions:

- Race conditions mainly occur due to the shared access of code blocks and resources, we can prevent it by synchronization elements such as mutexes, semaphores and condition variables.
- The Critical section must be protected. The critical section is the part where the shared resources are identified we can protect them using mutex locks and this works by acquiring the lock before entering the critical section and releasing the lock after completion of execution to ensure mutual exclusion.
- We can also avoid race conditions by employing thread synchronizations, in which a part of a program can execute only one thread at a time.

To Avoid Deadlocks:

- Resource Allocation ordering can be done by defining a consistent order for allocating resources and for acquiring and releasing locks.
- A lock hierarchy is established if multiple locks are needed and defining a parent-child relationship between locks and will ensure that threads always acquire locks in a top-down manner to avoid potential deadlocks.
- We can also implement deadlock detection mechanisms to identify and mitigate or to recover from potential deadlocks. (Some include Wait-for Graphs, Banker's Algorithm, Resource allocation Graphs and timeouts)

---

7. What is the difference between exec and fork?

Ans: exec():

- When the exec() system call is invoked, the program given as parameter will replace the entire process. All threads involved in the process are replaced with the parameter given in exec().
- However, the pid (process ID) remains the same but the memory, data, and state of the current process are replaced by those of the new program being executed.
- Exec() is commonly used after fork() where the new child process created needs to run a different program.
- It retains the process attributes such as PID and file descriptors.

Fork():

- fork() is a process creation system call.
- It creates a copy of the calling process, known as the child process. The child process has its own address space and it also contains a copy of the parent's address space including the code, data and stack.
- In parent process the return value is the PID of the child process, whereas in the child process it returns 0 and a negative value in case of a unsuccessful process creation.
- The total number of processes created in calling multiple forks can be calculated using the formula 2^n-1 where n is the number of fork calls.

---

8. What is the difference between process and threads.

Ans: Process:

- A process is the instance of a program in execution. Each processes have their own memory space, resources and state.
- Processes are used for parallelism and isolation when multiple processes execute simultaneously, each can run independently of the other process.
- Process has its own independent memory spaces, so they won't share data. To communicate between different processes, Inter-process communication mechanisms such as sockets and message queues are used.
- It takes more time for process creation, termination and context switching. Processes are also known as heavyweight process.

Threads:

- Threads are the smallest unit of execution within a process. Multiple threads share the same memory space and resources. Also known as lightweight process.
- The overhead of creating a new thread is lower than that of creating a new process as threads share resources and memory space. For consistency and avoiding deadlocks and race conditions in multi-threaded execution, synchronization mechanisms such as locks, mutexes and semaphores are used.

---

9. Write a C program to demonstrate the use of Mutexes in threads synchronization.

main.c

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

#define NUM 5

pthread_mutex_t mutex;

int shared_variable=0;
void *threadfun(void *threadid){
    int id=*(int *)threadid;
    pthread_mutex_lock(&mutex);
    printf("\nThread %d is accessing the shared variable.!",id);
    shared_variable++;
    printf("\nThread %d : shared_variable = %d",id,shared_variable);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main(){
    pthread_t threads[NUM];
    int threadid[NUM];
    int i;

    for(i=0;i<NUM;i++){
        threadid[i]=i;
        pthread_create(&threads[i],NULL,threadfun,(void *)&threadid[i]);
    }

    for(i=0;i<NUM;i++){
        pthread_join(threads[i],NULL);
    }

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

```
Thread 0 is accessing the shared variable.!
Thread 0 : shared_variable = 1
Thread 1 is accessing the shared variable.!
Thread 1 : shared_variable = 2
Thread 2 is accessing the shared variable.!
Thread 2 : shared_variable = 3
Thread 3 is accessing the shared variable.!
Thread 3 : shared_variable = 4
Thread 4 is accessing the shared variable.!
Thread 4 : shared_variable = 5

...Program finished with exit code 0
Press ENTER to exit console.
```