Deadlocks In Linux

-> Implementing functions to simulate the custom Dining Philosophers Problem.
-> Launching five threads, each of which relies on three different functions, Thr_A(),Thr_B(),Thr_C(),Thr_D() and Thr_E().
-> Launching Varied versions of the problem and their solutions:

- launchThreads_1(Simulate the Problem)
- launchThreads_2(Avoid deadlock using Serialization)
- launchThreads_3(Avoid deadlock using Semaphores)
- launchThreads_4(Two sauce Bowls.. Avoid deadlock using Semaphores)

Note

The STDOUT in the Question is facing Race Conditions, hence the output observed may/may not be in the order in which the processes are running.

Q1) Modified Dining Philosophers Problem

The dining philosophers problem contains five philosophers sitting on a round table can perform only one among two actions – eat and think. For eating, each of them requires two forks, one kept beside each person. Typically, allowing unrestricted access to the forks may result in a deadlock.

`Installation`

```
 # Navigate to src/1
$> cd src/1

# run the makefile
$> make -f Makefile
       OR
$> make

# clean
$> make clean
```

`Technologies Used`

- Semaphores
- Multi-Threading
- Mutex Locks

InterProcess Communication in Linux

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes.

Q2) InterProcess Communication

Two programs P1 and P2. The first program P1 needs to generate an array of 50 random strings (of characters) of fixed length each. P1 then sends a group of five consecutive elements of the array of strings to P2 along with the ID's of the strings, where the ID is the index of the array corresponding to the string. The second program P2 needs to accept the received strings, and send back the highest ID received back to P1 to acknowledge the strings received. The program P2 simply prints the ID's and the strings on the console. On receiving the acknowledged packet, P1 sends the next five strings, with the string elements starting from the successor of the acknowledged ID.

`Directory Structure`

```
  src/2
       /FIFO
            /P1.c
            /P2.c
            /Makefile
       /SHM
            /P1.c
            /P2.c
            /Makefile
       /SOCKET
            /P1.c
            /P2.c
            /Makefile
```

`Running the Programs`

```
 # run the Makefile
$> make -f Makefile
       OR
$> make

# run the Programs

! In Two Separate Terminals

$> ./p2 (Terminal 1)
       AND
$> ./p1 (Terminal 2)

! In this order
```

Q2)(A) Using FIFO

For this Implementation two FIFOs have been created in the following fashion

```
P1 writes into FIFO_1
P2 reads from FIFO_1
P2 writes into FIFO_2
P1 reads from FIFO_2
```

This is done to prevent unnecessary read/writes from FIFOs while the other is trying to communicate.

Firstly `mkfifo(char *, int)` is used to create the FIFOs. Then For every read or write command firstly we open the required FIFO by using `open(char *, int)` with necessary specifiers like `O_WRONLY` or `RD_ONLY` to open the FIFO which returns the file descriptor. Then to read or write we use `read(int fd, char *, sizeof(char *))` and `write(int fd, char *, sizeof(char *))`

Q2)(B) Using SHM

For this Implementation Shared Memory is utilized in the following fashion

```
P1 writes into MEM
P2 reads from MEM
P2 writes into MEM
P1 reads from MEM
```

This is done to prevent unnecessary read/writes from processes while the other is trying to communicate.

Firstly `key_t ftok(const char *pathname, int proj_id);` is used to create the file identied by the given pathname. Then `int shmget(key_t key, size_t size, int shmflg);` is used to link to that specific memory location. Then we use `void *shmat(int shmid, const void *shmaddr, int shmflg);` to access the memory block and use the basic `read()` and `sprintf` to read and write to the Shared memory.

Q2)(C) Using SOCKETS

For this Implementation one Socket has been created in the following fashion

```
P1 writes into ProgSocket
P2 reads from ProgSocket
P2 writes into ProgSocket
P1 reads from ProgSocket
```

This is done to prevent unnecessary read/writes from programs while the other is trying to communicate.

An `AF_UNIX` socket has been used as the interprocess communications are local.
`sockaddr_un structure` has been used to create and use the Socket which is then initialised by `AF_UNIX` by `addr.sun_family = AF_UNIX` where addr is the name of the struct. We then create a data socket and connect it to adde using `connect(data_socket, (const struct sockaddr *) &addr, sizeof(addr))`
The above two steps are done in both P1 and P2
Then we use `write(data_socket, char *, sizeof(char *))` and `read(data_socket, char *, sizeof(char *))` to read and write using the data socket we made.

Kernel Module

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. A module can be configured as built-in or loadable.

Q3) Kernel Module

Implementing a kernel system call as a module. The task of the system call would be to read the entries of the process task_struct corresponding to any given process (supplied as input via command line argument) and prints the values of the following field: pid, user id, process group id (pgid) and command path. The system call should be implemented in the kernel and not directly as a mainline kernel function.

Implementation

For this Implementation the `task_struct` of the given `PID` is fetched using the `pid_task(find_pid_ns(id, &init_pid_ns), PIDTYPE_PID);` then the `task_struct` is read for the specific `PID` and the output is dumped to the `SYSLOG` using the `printk` and read to the user space using the `dmesg` call.

`Installation`

```
 # run the Makefile
$> make -f Makefile
       OR
$> make
```

`Load the Module with PID as parameter`

```
 $> sudo insmod main.ko pid=(PID)
```

`Check Output`

```
 $> sudo tail -n 50 /var/log/syslog
```

`Unload the Module`

```
 $> sudo rmmod main
```

License

--- EOF ---