

6CS005

High Performance Computing

Week 4 Workshop

Revision on Multithreading

Student Number: 2408869

Name: Narayan Lohani

Group: L6CG15

Table of Contents

Explanation	2
Output of Program 1	4
Output of Program 2	4
Source Code of Program 1	5
Source Code of Program 2	6

Table of Figures

Figure 1: Output of Program 1	4
Figure 2: Output of Program 2.....	4
Figure 3: Source Code of Program 1: Thread Routine	5
Figure 4: Source Code of Program 1: Main function	5
Figure 5: Source Code of Program 2: Main Function.....	6
Figure 6: Source Code of Program 2: Thread Routine	6

Explanation

1. In a banking system, multiple users can access and modify their accounts concurrently. Each user has a balance, and they can deposit or withdraw money. Modify the code below to use a mutex to prevent race conditions during balance updates and ensure that multiple users can't simultaneously update the balance of the same account.

When multiple users try to make changes to their money either by withdrawing or depositing on the same account, critical section is formed and race condition may arise. To prevent this, whenever any thread try to make any change to the account balance, first we lock the thread using mutex, perform our desired action (withdraw or deposit), and unlock the thread. This way ensures that no balance is lost at any point.

To accomplish this task, we added a mutex variable namely locker inside the structure of account so that each account has its own locker. In the main function, along with initializing the account Ids we also initialize the locker of each account. Because the accounts are structure type global variables, inside thread routine we directly lock the thread using locker in the structure of that account. The account reference or account Id is passed using thread arguments which is then type casted before using. After locking the thread using mutex, the desired operation (withdraw/deposit) is performed and thread is unlocked.

```
qlc: ~ % 
4 typedef struct {
5 } Account;
6
7 Account accounts[10];
8
9 pthread_mutex_t withdraw_locker, deposit_locker;
10
11 void* withdraw(void* a) {
12     int accountid = *(int*)a;
13     double amount = 100; // Withdraw amount
14
15     pthread_mutex_lock(&(withdraw_locker));
16     accounts[accountid].balance -= amount;
17     pthread_mutex_unlock(&(withdraw_locker));
18 }
19
20 void* deposit(void* a) {
21     int accountid = *(int*)a;
22     double amount = 100; // Deposit amount
23
24     pthread_mutex_lock(&(deposit_locker));
25     accounts[accountid].balance += amount;
26     pthread_mutex_unlock(&(deposit_locker));
27 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE

zah-workshop

```
% narayan@narayan workshop % gcc ql.c -o ql
narayan@narayan workshop % ./ql
Account 0 balance = 0.00
Account 1 balance = 0.00
Account 2 balance = 0.00
Account 3 balance = 0.00
Account 4 balance = 0.00
Account 5 balance = 0.00
Account 6 balance = 0.00
Account 7 balance = 0.00
Account 8 balance = 0.00
Account 9 balance = 0.00
%
% narayan@narayan workshop %
```

Source Code of Program 1

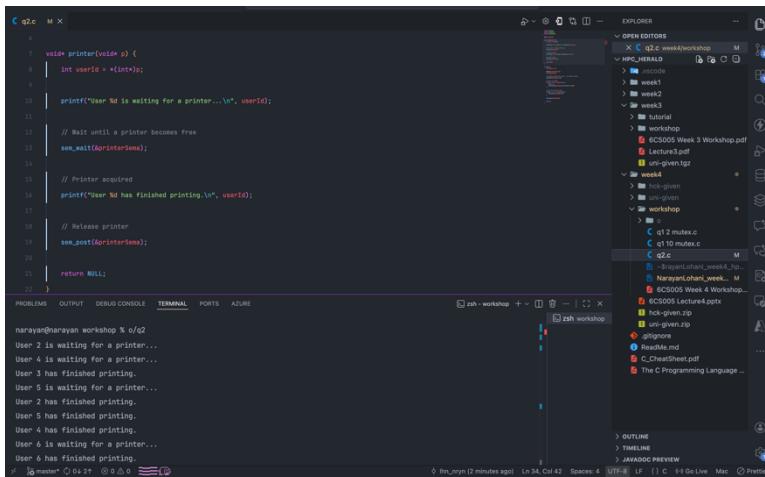
Output of Program 1

- 2. A printer is shared among multiple users. Each user can either print a document or wait if the printer is in use. There are only 2 printers in the office. Use semaphores to manage the printer access, ensuring that no more than 2 users can print at the same time.**

When many users try to use the office printers at the same time, a critical section is formed because only a limited number of printers are available. If all users try to print together, the system needs a way to make some users wait until a printer becomes free. To handle this, we use a semaphore that keeps track of how many printers are available. Since there are two printers, the semaphore is initialized with the value 2.

Whenever a thread wants to print a document, it first performs a `sem_wait`, which decreases the semaphore value if a printer is free. If both printers are busy, the thread automatically waits. After finishing the printing task, the thread performs `sem_post`, which increases the semaphore value and lets another waiting user print.

This approach ensures that at most two users can print at the same time, preventing printer conflicts while still allowing multiple users to work efficiently.



```

1 // q2.c
2
3 void* printer(void* p) {
4     int userId = *(int*)p;
5
6     printf("User %d is waiting for a printer...\n", userId);
7
8     // Wait until a printer becomes free
9     sem_wait(&printerSemaphore);
10
11    // Printer acquired
12    printf("User %d has finished printing.\n", userId);
13
14    // Release printer
15    sem_post(&printerSemaphore);
16
17    return NULL;
18}

```

The terminal output shows the following sequence of events:

```

narayan@narayan-workshop ~ % o/q2
User 1 is waiting for a printer...
User 2 is waiting for a printer...
User 3 has finished printing.
User 5 is waiting for a printer...
User 2 has finished printing.
User 5 has finished printing.
User 4 has finished printing.
User 6 is waiting for a printer...
User 5 has finished printing.

```

Source Code of Program 2

Output of Program 2

Output of Program 1

```
[narayan@narayan workshop % gcc q1.c -o o/q1
[narayan@narayan workshop % o/q1
Account 0 balance = 0.00
Account 1 balance = 0.00
Account 2 balance = 0.00
Account 3 balance = 0.00
Account 4 balance = 0.00
Account 5 balance = 0.00
Account 6 balance = 0.00
Account 7 balance = 0.00
Account 8 balance = 0.00
Account 9 balance = 0.00
narayan@narayan workshop % ]
```

Figure 1: Output of Program 1

Output of Program 2

```
[narayan@narayan workshop % gcc q2.c -o o/q2
q2.c:32:5: warning: 'sem_init' is deprecated [-Wdeprecated-declarations]
  32 |     sem_init(&printerSema, 0, 2);
      |     ^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
  55 | int sem_init(sem_t *, int, unsigned int) __attribute__((__deprecate
      |
//Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
  223 | #define __deprecated    __attribute__((__deprecate
      |     ^
q2.c:45:5: warning: 'sem_destroy' is deprecated [-Wdeprecated-declarations]
  45 |     sem_destroy(&printerSema);
      |     ^
//Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
  53 | int sem_destroy(sem_t *) __attribute__((__deprecate
      |
//Library/Developer/CommandLineTools/SDKs/MacOSX.sdk
  223 | #define __deprecated    __attribute__((__deprecate
      |
2 warnings generated.
[narayan@narayan workshop % o/q2
User 0 is waiting for a printer...
User 0 has finished printing.
User 1 is waiting for a printer...
User 1 has finished printing.
User 2 is waiting for a printer...
User 2 has finished printing.
User 3 is waiting for a printer...
User 3 has finished printing.
User 4 is waiting for a printer...
User 4 has finished printing.
User 5 is waiting for a printer...
User 5 has finished printing.
User 6 is waiting for a printer...
User 6 has finished printing.
User 7 is waiting for a printer...
User 7 has finished printing.
User 8 is waiting for a printer...
User 8 has finished printing.
User 9 is waiting for a printer...
User 9 has finished printing.
narayan@narayan workshop % ]
```

Figure 2: Output of Program 2

Source Code of Program 1

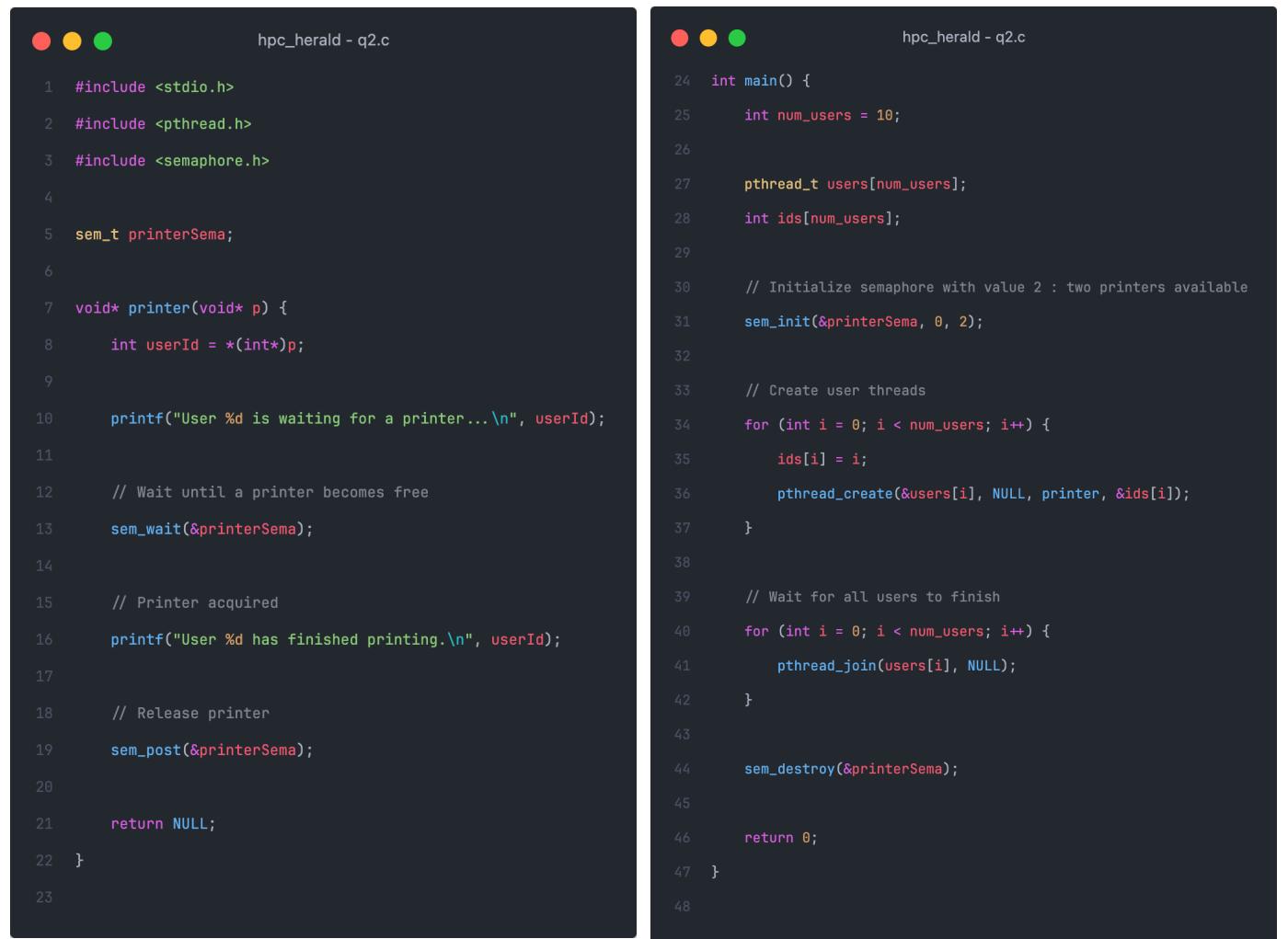
```
hpc_herald - q1.c
1 #include <stdio.h>
2 #include <pthread.h>
3
4 typedef struct {
5     int accountNumber;
6     double balance;
7     pthread_mutex_t locker;
8 } Account;
9
10 Account accounts[10];
11
12 void* withdraw(void* p) {
13     int accountId = *(int*)p;
14     double amount = 100; // Withdraw amount
15
16     pthread_mutex_lock(&accounts[accountId].locker);
17     accounts[accountId].balance -= amount;
18     pthread_mutex_unlock(&accounts[accountId].locker);
19
20     return NULL;
21 }
22
23 void* deposit(void* p) {
24     int accountId = *(int*)p;
25     double amount = 100; // Deposit amount
26
27     pthread_mutex_lock(&accounts[accountId].locker);
28     accounts[accountId].balance += amount;
29     pthread_mutex_unlock(&accounts[accountId].locker);
30
31     return NULL;
32 }
```

```
hpc_herald - q1.c
34 int main() {
35     pthread_t threads[20];
36     int ids[10];
37
38     // Create multiple threads to simulate transactions on the same account
39     for (int i = 0; i < 10; i++) {
40         ids[i] = i;
41         pthread_mutex_init(&accounts[i].locker, NULL);
42         pthread_create(&threads[i], NULL, withdraw, &ids[i]);
43         pthread_create(&threads[i + 10], NULL, deposit, &ids[i]);
44     }
45
46     for (int i = 0; i < 20; i++) {
47         pthread_join(threads[i], NULL);
48     }
49
50     // Print final balance
51     for (int i = 0; i < 10; i++) {
52         pthread_mutex_destroy(&accounts[i].locker);
53         printf("Account %d balance = %.2f\n", i, accounts[i].balance);
54     }
55
56     return 0;
57 }
```

Figure 4: Source Code of Program 1: Main function

Figure 3: Source Code of Program 1: Thread Routine

Source Code of Program 2



```
hpc_herald - q2.c

1 #include <stdio.h>
2 #include <pthread.h>
3 #include <semaphore.h>
4
5 sem_t *printerSema;
6
7 void* printer(void* p) {
8     int userId = *(int*)p;
9
10    printf("User %d is waiting for a printer...\n", userId);
11
12    // Wait until a printer becomes free
13    sem_wait(&printerSema);
14
15    // Printer acquired
16    printf("User %d has finished printing.\n", userId);
17
18    // Release printer
19    sem_post(&printerSema);
20
21    return NULL;
22 }

hpc_herald - q2.c

24 int main() {
25     int num_users = 10;
26
27     pthread_t users[num_users];
28     int ids[num_users];
29
30     // Initialize semaphore with value 2 : two printers available
31     sem_init(&printerSema, 0, 2);
32
33     // Create user threads
34     for (int i = 0; i < num_users; i++) {
35         ids[i] = i;
36         pthread_create(&users[i], NULL, printer, &ids[i]);
37     }
38
39     // Wait for all users to finish
40     for (int i = 0; i < num_users; i++) {
41         pthread_join(users[i], NULL);
42     }
43
44     sem_destroy(&printerSema);
45
46     return 0;
47 }
```

Figure 6: Source Code of Program 2: Thread Routine

Figure 5: Source Code of Program 2: Main Function

