**CISC2005 Lab 07**

1. **Prerequisites**

* Successful installation of \*nix terminal (e.g., WSL, cygwin) or a \*nix operating system (e.g., Linux, Mac).
* Successful installation of a C compiler, GCC is recommended.

You can also use an online c compiler: <https://www.onlinegdb.com/online_c_compiler>

* Successful installation of a text editor, VIM or Emacs is recommended.

1. **Exercises**

Semaphore is a synchronization tool that is used to control access to a shared resource in a concurrent system. There are two types of semaphores: mutex semaphore and counting semaphore. A binary semaphore can only take on two values, 0 and 1, and is typically used for mutual exclusion. A counting semaphore, on the other hand, can take on a range of non-negative values and is typically used to control access to a finite pool of resources.

In a \*nix operating system (e.g., Linux, Mac), the <semaphore.h> header defines the ***sem\_t*** type, used in performing semaphore operations. Here are some important functions about semaphore operations.

1. ***sem\_init***

Function:

int ***sem\_init***(sem\_t \*sem, int pshared, unsigned value);

Description:

The sem\_init() function initializes an unnamed semaphore and sets its initial value. The maximum value of the semaphore is set to SEM\_VALUE\_MAX. The title for the semaphore is set to the character representation of the address of the semaphore. If an unnamed semaphore already exists at ***sem***, then it will be destroyed, and a new semaphore will be initialized.

Parameters:

1. ***sem***: a pointer to the storage of an uninitialized unnamed semaphore. The pointer must be aligned on a 16-byte boundary. This semaphore is initialized.
2. ***pshared***: An indication to the system of how the semaphore is going to be used. A value of zero indicates that the semaphore will be used only by threads within the current process. A nonzero value indicates that the semaphore may be used by threads from other processes.
3. ***value***: The value used to initialize the value of the semaphore.

Return:

***sem\_init*** returns 0 on success; on error, -1 is returned, and ***errno*** is set to indicate the error.

1. ***sem\_post //*** V(S) release lock

Function:

int ***sem\_post***(sem\_t \*sem);

Description:

***sem\_post*** increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a ***sem\_wait*** call will be woken up and proceed to lock the semaphore.

Parameters:

1. ***sem***: a pointer to the storage of an uninitialized unnamed semaphore. The pointer must be aligned on a 16-byte boundary. This semaphore is initialized.

Return:

***sem\_post*** returns 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned,, and ***errno*** is set to indicate the error.

1. ***sem\_trywait*** and ***sem\_wait //*** P(S) hold/add lock

Function:

int ***sem\_trywait***(sem\_t \*sem);

int ***sem\_wait***(sem\_t \*sem);

Description:

***sem\_wait*** decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

***sem\_trywait*** is the same as ***sem\_wait***, except that if the decrement cannot be immediately performed, then call returns an error (errno set to EAGAIN) instead of blocking.

Parameters:

1. ***sem***: a pointer to the storage of an uninitialized unnamed semaphore. The pointer must be aligned on a 16-byte boundary. This semaphore is initialized.

Return:

Both these functions return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and ***errno*** is set to indicate the error.

In this exercise, students are required to complete missing codes ( ??? ) to implement several basic synchronization algorithms using **semaphore**. For each program, students should capture **screenshots** of the successful execution, and answer the **questions** for the program. In the submission file, please attach the added code and execution screenshot in sequence.

**Example 1**: **Mutex semaphore**

In this example, we use a global variable “counter” and a semaphore variable “s” to count the number of accesses to a shared resource. Please use ***sem\_init*, *sem\_wait***, and ***sem\_post*** tocomplete the following code tocorrectly count accesses.

1. // Global variables
2. **static** **volatile** **long** counter = 0;
3. sem\_t s; // Semaphore variable
5. **void** \*increment(**void** \*vargp) {
6. // Enter critical region
7. **for** (**long** i = 0; i < N\_ITERATIONS; i++) {
8. // Block on a semaphore count
9. ??? ; // P(S)
10. // Shared resource
11. counter++;
12. // Increment a semaphore
13. ??? ; // V(S)
14. }
15. // Leave critical region
16. **return** NULL;
17. }
19. **int** main(**int** argc, **char** \*\*argv) {
20. // Initialize the semaphore
21. sem\_init(&s, 0,  ??? );
22. // Create threads
23. pthread\_t threads[NUM\_THREADS];
24. // Create threads
25. **for** (**int** i = 0; i < NUM\_THREADS; i++) {
26. pthread\_create(&threads[i], NULL, increment, NULL);
27. }
28. printf("Create %d threads.\n", NUM\_THREADS);
29. // Wait for threads to finish
30. **for** (**int** i = 0; i < NUM\_THREADS; i++) {
31. pthread\_join(threads[i], NULL);
32. }
33. printf("counter=%ld.\n", counter);
34. // Destroy the semaphore
35. sem\_destroy(&s);
36. **return** 0;
37. }

**Code:**

1. Download the sample code from UMMoodle ([example1.c](https://ummoodle.um.edu.mo/mod/resource/view.php?id=2998485)).
2. Compile the above C program and execute. (Tips: use the following compile command.)

gcc example1.c -o example1 -l pthread

**Question 1**:

1.1 Please complete the code in example1.c.

1.2 How does the semaphore “s” change as the program runs?

Requirements:

1. Capture 1 **screenshots** of the successful execution.
2. Answer Question 1.

**Example 2**: **Counting semaphore**

Imagine the following scenario: there are **two** workers serving 5 customers, and each worker can only serve one customer at a time. Complete the code in example 2 to simulate this scenario.

1. // Number of customers
2. **int** NUM\_CUSTOMERS = 5;
4. sem\_t s; // Semaphore variable
6. // Simulate the process of service
7. **void** \*get\_service(**void** \*arg)
8. {
9. **int** id = \*((**int**\*)arg);
10. // Block on a semaphore count
11. ??? ; // P(S)
12. // Enter critical region
13. printf("Customer [%d] is getting service.\n", id);
14. sleep(2);
15. printf("Customer [%d] has left.\n", id);
16. // Leave critical region
17. // Increment a semaphore
18. ??? ; // V(S)
19. **return** 0;
20. }
22. **int** main()
23. {
24. // Initialize the semaphore
25. sem\_init(&s, 0,  ??? );
26. // Create threads
27. pthread\_t threads[NUM\_CUSTOMERS];
28. **int** thread\_ids[NUM\_CUSTOMERS];
29. printf("Create %d threads.\n", NUM\_CUSTOMERS);
30. **for** (**int** i = 0; i < NUM\_CUSTOMERS; i++) {
31. thread\_ids[i] = i + 1;
32. pthread\_create(&threads[i], NULL, get\_service, &thread\_ids[i]);
33. }
34. // Wait for threads to finish
35. **for** (**int** i = 0; i < NUM\_CUSTOMERS; i++) {
36. pthread\_join(threads[i], NULL);
37. }
38. // Destroy the semaphore
39. sem\_destroy(&s);
40. **return** 0;
41. }

**Code:**

1. Download the sample code from UMMoodle ([example2.c](https://ummoodle.um.edu.mo/mod/resource/view.php?id=2998488)).
2. Compile the above C program and execute. (Tips: use the following compile command.)

gcc example2.c -o example2 -l pthread

**Question 2**:

2.1 Please complete the code in example2.c.

2.2 How does the semaphore “s” change as the program runs?

Requirements:

1. Capture 1 **screenshots** of the successful execution.
2. Answer Question 2.

**Example 3**: **Possible deadlocks with semaphores**

Deadlock is the permanent and unresolvable blocking of two or more threads that results from each waiting on the other. The following code in example3.c provides a simple illustration of how deadlock can happen. There are two threads, running the ***first*** and ***second*** functions. The code for ***first*** tries to acquire semaphore “A” before “B”, while ***second*** acquires them in the opposite order. Figure 1 shows the state model for this example. Please execute example3 and explain why the deadlock occurs.

手机屏幕的截图

描述已自动生成

Figure 1. State model for example 3.

Example3 code:

1. /\* struct contains shared semaphores \*/
2. **typedef** **struct** {
3. sem\_t A;
4. sem\_t B;
5. } SEMAPHORE;
7. **void** \*first (**void** \* args)
8. {
9. SEMAPHORE \*sem = (SEMAPHORE \*) args;
10. /\* Acquire "A" before "B" \*/
11. printf("Job started in the first thread.\n");
12. sem\_wait(&sem->A);
13. sleep(1);
14. **while** (-1 == sem\_trywait(&sem->B))
15. {
16. printf("The first thread has acquired 'A' and waits for 'B'.\n");
17. sleep(1);
18. }
19. /\* other code here \*/
20. **return** NULL;
21. }
23. **void** \*second (**void** \* args)
24. {
25. SEMAPHORE \*sem = (SEMAPHORE \*) args;
26. /\* Acquire "B" before "A" \*/
27. printf("Job started in the second thread.\n");
28. sem\_wait(&sem->B);
29. sleep(1);
30. **while** (-1 == sem\_trywait(&sem->A))
31. {
32. printf("The second thread has acquired 'B' and waits for 'A'.\n");
33. sleep(1);
34. }
35. /\* other code here \*/
36. **return** NULL;
37. }

**Code:**

1. Download the sample code from UMMoodle ([example3.c](https://ummoodle.um.edu.mo/mod/resource/view.php?id=2998491)).
2. Compile the above C program and execute. (Tips: use the following compile command.)

gcc example3.c -o example3 -l pthread

**Question 3**:

Why does the deadlock occur in example 3?

Requirements:

1. Capture 1 **screenshots** of the successful execution.
2. Answer Question 3.

**Example 4**: **Bounded Buffer with semaphores**

This example provides an implementation of Bounded Buffer with semaphores.

1. // Global variables
2. buffer\_item START\_NUMBER = 0;
3. buffer\_item buffer[BUFFER\_SIZE];
4. sem\_t mutex;
5. sem\_t empty;
6. sem\_t full;
8. **void** initialization(){
9. sem\_init(&mutex, 0, 1);
10. sem\_init(&full, 0, BUFFER\_SIZE);
11. sem\_init(&empty, 0, 0);
12. }
14. **void** \*producer(**void** \*arg)
15. {
16. **int** thread\_id = \*(**int**\*)arg;
17. buffer\_item item;
19. **while**(TRUE) {
20. sleep(1);
21. sem\_wait(&full);
22. sem\_wait(&mutex);
24. item = START\_NUMBER++;
25. insert\_item(item);
27. printf("Producer %d produced %d \n", thread\_id, item);
29. sem\_post(&mutex);
30. sem\_post(&empty);
31. }
33. **return** NULL;
34. }
36. **void** \*consumer(**void** \*arg)
37. {
38. **int** thread\_id = \*(**int**\*)arg;
39. buffer\_item item;
41. **while**(TRUE){
42. sleep(1);
43. sem\_wait(&empty);
44. sem\_wait(&mutex);
46. remove\_item(&item);
47. printf("Consumer %d consumed %d \n", thread\_id, item);
49. sem\_post(&mutex);
50. sem\_post(&full);
51. }
53. **return** NULL;
54. }

**Code:**

1. Download the sample code from UMMoodle ([example4.c](https://ummoodle.um.edu.mo/mod/resource/view.php?id=2998803)).
2. Compile the above C program and execute. (Tips: use the following compile command.)

gcc example4.c -o example4 -l pthread

Requirements:

1. Capture 1 **screenshots** of the successful execution.