CSE 3211 : Operating System Assignment 1

Saif Mahmud
2015-116-815 & SH-54

Tauhid Tanjim
2015-716-819 & SH-58

October 22, 2018

1 Concurrent Mathematics Problem

A number of threads are invoked in order to run the *adder* function. The global variable *counter* is loaded on the local variable *a* and the problem occurs with the event of context switching. In the case where a thread switch occurs after loading *counter* to *a*, another thread will increase the value of *counter*. Therefore, the comparison followed by this variable assignment with *b* will result in false. The same problem will occur with the context switch after loading counter to *b*. Therefore, we have defined *counter* as the critical region of this program.

In order to protect this critical area, binary semaphore *lock_cnt* is created before forking multiple threads. Each time a thread needs to acquire the lock before it can get access to the critical area *counter*, and after executing the tasks, it releases the semaphore *lock_cnt*.

2 Paint Shop Synchronization Problem

According to definition of the problem, we have maintained two buffers. The first one is *order_buffer* which maintains the ordered and yet to be mixed paint cans. Another is *delivery_buffer* which represents the mixed and yet to be shipped paint cans. Here, these two buffers are defined as critical regions.

2.1 void paintshop_open(void)

Initialization of required binary and counting semaphores as well as two buffers using functions *init_semaphore()* and *init_buffer()* respectively. We have also initialized the variable *remaining_customers* with the number of customers *(NCUSTOMERS)* here.

2.2 void order_paint(paint_can *can)

In this function, we have first placed the *paint_can* in the *order_buffer*. Before doing this, we have to put a wait on the counting semaphore *order_buffer_empty* which demonstrates the number of empty slots in the *order_buffer*. It prevents the threads accessing *order_buffer* when it is full which results in eliminating

the problem of busy waiting for deadlock resolution. We have used the binary semaphore *order_mutex* in order to control access to the critical region *order_buffer*.

Afterwards, we have searched the *delivery_buffer* looking for if the parameter *paint_can* is ready. We have used a binary semaphore *delivery_mutex* in order to prevent simultaneous access to *delivery_buffer*. However, searching the *delivery_buffer* creates a problem of busy waiting and so, we have put a wait on the semaphore *ready_cans*. If the ordered paint can is found, then the function removes the can from the *delivery_buffer*. Otherwise, it signals the semaphore to wake up another thread waiting on it.

2.3 void go_home(void)

After getting delivery of the desired *paint_can* the customer would be able to call this function. We have decreased the value of *remaining_customer* by 1 in order that the variable always reflects the number of present customers. However, it is possible for two *customer* threads to use this function simultaneously and therefore, we have used a binary semaphore *remaining_customers_mutex* in order to prevent access to the critical region at the same time.

2.4 void * take_order(void)

In this function, the loop checks whether *remaining_customer* is 0. If it is the case, the function returns NULL which results in *staff thread* to be terminated. Otherwise, it iterates through the *order_buffer* and pick an order for shipment. In this regard, we have avoided busy waiting using the semaphore *order_buffer_full*. This is initialized as zero and only the *order_paint(paint_can *can)* function signals it while placing an order. Hence, when there is no can in the *order_buffer* the *staff thread* will sleep on the counting semaphore *order_buffer_full*.

2.5 void fill_order(void *v)

In order to ensure parallel access to tints from different *staff* threads in the case where same tints are not required for the specific paint can, we have created an array of binary semaphore *access_specific_tints* of size *NCOLORS* which approves access to all the tints which are not in use at the specific moment. We have put a wait on the binary semaphore *tints_mutex* before locking specific tints by using the semaphore array. After acquiring the lock for specific tints in use, the semaphore *tints_mutex* is signaled to release and then the function *mix()* is called. Semaphore on wait for requested tints of the parameter paint can is released after mixing.

2.6 void serve_order(void *v)

This function puts the mixed can on the *delivery_buffer* and signals the semaphore *ready_cans* on which the customer thread is waiting. Moreover, the binary semaphore *delivery_mutex* is used to control access to the critical region *delivery_buffer*.

2.7 void paintshop_close(void)

All the semaphores created before has been destroyed here.