Classic Problems of Synchronization

Prof. Naga Kandasamy
ECE Department, Drexel University

We discuss three classic problems of synchronization: producer-consumer, the sleeping barber, and readers-writers. In all cases, we focus on the use of thread synchronization mechanisms to coordinate the execution of the various threads involved.

Producer-Consumer Problem

The producer consumer is a classic design pattern used in many software designs. Figure 1 shows the schematic in which the producer and consumer threads run concurrently. The producer periodically generates "work items" and places them in the bounded shared buffer of size N and the consumer retrieves work items from the buffer for consumption.

The buffer data type is specified as follows:

```
typedef struct queue_s {
    int size;   /* Size of the queue */
    int *buffer; /* Buffer storing items */
    int counter; /* Number of items currently in queue */
    int in, out; /* Indices for the producer and the consumer */
    pthread_mutex_t lock;   /* Lock to protect the shared buffer */
    /* Semaphores to signal full/empty queue conditions */
    sem_t full, empty;
} queue_t;

queue_t *queue;
```

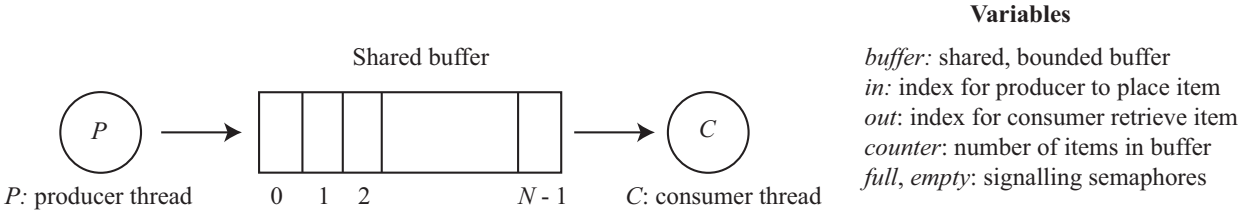


Figure 1: The structure of the producer-consumer problem.

The main thread initializes the data structure and creates the producer and consumer threads.

```
/* Create bounded buffer and initialize variables */
queue = create_queue(size);
/* Create producer and consumer threads */
pthread_create(&producer_id, NULL, producer, (void *)queue);
pthread_create(&consumer_id, NULL, consumer, (void *)queue);
/* Wait for producer and consumer threads to finish */
pthread_join(producer_id, NULL);
pthread_join(consumer_id, NULL);
/* Free memory */
delete_queue(queue);
```

The function that creates the queue is shown below. Pay particular attention to the way the counting semaphores are initialized in lines 9 and 10. The semaphore *full* is initialized to the size of the buffer — indicating the number of free slots available in the buffer. The semaphore *empty* is initialized to zero, indicating that are no items available in the buffer for the consumer yet.

```
1  queue_t *create_queue(int size) {
2      queue_t *queue = (queue_t *)malloc(sizeof(queue_t));
3      queue->size = size;
4      queue->buffer = (int *)malloc(sizeof(int) * size);
5
6      queue->counter = 0;  /* Initialize other members */
7      queue->in = queue->out = 0;
8      pthread_mutex_init(&(queue->lock), NULL);
9      sem_init(&(queue->full), 0, size);
10     sem_init(&(queue->empty), 0, 0);
11     return queue;
12 }
```

The following synchronization constraints must be enforced to ensure proper sequencing between the producer and consumer threads:

- The producer must wait for a free slot to become available if the buffer is full.
- The consumer must wait for the producer to place a new item in the buffer if it is empty.
- When the producer places an item in the buffer, it must signal to the consumer that an item is available for consumption.
- When the consumer retrieves an item, it must signal to the producer that a free slot has opened up in the buffer.

The producer code is shown below in which it produces *num_items* items at some rate and places them in the buffer. The producer blocks on the semaphore *full* in line 5 if the queue is full, waiting for the consumer to unblock it once an item has been retrieved from the buffer. Upon placing an item in the buffer, the producer signals to the consumer via the semaphore *empty* in line 9 in case the consumer is blocked on an empty queue. Finally, since the buffer is shared between the producer and consumer, a lock is acquired to access it (lines 5 and 7).

```
int num_items = UD(10, 20); /* Uniform distribution between [10, 20] */
while (num_items-- > 0) {
    int item = UD(5, 10); /* Produce item simulating processing time */
    sem_wait(&(queue->full)); /* Block here if queue is full */
    pthread_mutex_lock(&(queue->lock));
    add_to_queue(queue, item); /* Add item to queue */
    pthread_mutex_unlock(&(queue->lock));
    sem_post(&(queue->empty)); /* Signal consumer */
    sleep(UD(2, 5)); /* Sleep for some random time between [2, 5] */
}
all_done = 1; /* Producer is all done */
sem_post(&(queue->empty)); /* One last signal to unblock consumer */
```

The following function adds an item to the queue. Note that prior to calling this function, the producer has already acquired the mutex for the buffer. So, the operations performed with this function are thread safe. The buffer is bounded in the sense that if the index reaches the end of the buffer it wraps around to the beginning (line 5).

```
void add_to_queue(queue_t *queue, int item) {
    queue->buffer[queue->in] = item;
    queue->in++;
    queue->counter++;
    if (queue->in == queue->size)
        queue->in = 0; /* Wrap around the circular buffer */
}
```

The code for the consumer thread is shown below.

```
while (1) {
    sem_wait(&(queue->empty)); /* Block if queue is empty */
    if ((all_done == 1) &&
        (queue->counter == 0)) /* Check termination condition */
        pthread_exit(NULL);

    pthread_mutex_lock(&(queue->lock));
    int item = remove_from_queue(queue);
    pthread_mutex_unlock(&(queue->lock));

    sem_post(&(queue->full)); /* Signal producer */
    sleep(item); /* Simulate some processing */
}
```

The consumer blocks on the semaphore *empty* in line 2 if the queue is empty. The termination condition — that is, the queue is empty and the producer thread has finished its quota of items — is checked in line 3. Otherwise, the consumer acquires the lock on the buffer and retrieves an item. It then signals to the producer that there is an empty slot available in the buffer (line 11).

The following function removes an item from the queue. Since the consumer has acquired the lock on the buffer prior to calling this function the operations within are thread safe.

```
int remove_from_queue(queue_t *queue) {
    int item = queue->buffer[queue->out];
    queue->out++;
    queue->counter--;

    if(queue->out == queue->size)
        queue->out = 0; /* Warp around the circular buffer */

    return item;
}
```

The file *producer_consumer.c*, available on BBLearn, contains the complete code listing for the producer-consumer problem.

Sleeping Barber Problem

Consider a barbershop consisting of a waiting room with n chairs and a barber room with one chair. The sleeping barber problem is stated as follows:

- If there are no customers to be served the barber goes to sleep.
- If a customer enters the barbershop and all chairs are occupied, then the customer waits outside the shop. Alternatively, the customer could just go home.
- If the barber is busy but chairs are available in the waiting room, then the customer sits in one of the free chairs and waits for his or her turn.
- If the barber is asleep, the customer wakes up the barber upon entering the barber room.

The various semaphores needed to coordinate between the barber and customer threads are initialized as follows.

```
/* Signal that waiting room can accommodate customers */
sem_t waiting_room;
/* Signal to ensure exclusive access to barber seat */
sem_t barber_seat;
/* Signal customer that barber is done with him/her */
sem_t done_with_customer;
/* Signal to wake up barber */
sem_t barber_bed;

/* Initialize semaphores */
sem_init(&waiting_room, 0, num_waiting_chairs);
sem_init(&barber_seat, 0, 1);
sem_init(&done_with_customer, 0, 0);
sem_init(&barber_bed, 0, 0);
```

The *waiting_room* variable is a counting semaphore that is initialized to indicate the number of available chairs in the waiting room. The other semaphores are binary semaphores.

The main thread creates the barber and customer threads as follows.

```
pthread_create(&btid, 0, barber, 0);

int customerID[MAX_NUM_CUSTOMERS];
int i;
for (i = 0; i < num_customers; i++) {
    customerID[i] = i;
    pthread_create(&tid[i], 0, customer, &customerID[i]);
}

for (i = 0; i < num_customers; i++)
    pthread_join(tid[i], 0);

done_with_all_customers = TRUE;
sem_post(&barber_bed); /* Wake up barber */
```

The barber stays in a while loop until all customers have been serviced. The barber blocks on the *barber_bed* semaphore in line 5 until woken up by a customer sitting in the barber chair. The barber trims hair for some time (line 7) and signals via the *done_with_customer* semaphore to the customer that the job has been completed; and goes back to sleep until woken up by the next customer to repeat the process.

```
1  void *barber(void *arg)
2  {
3      int wait_time;
4      while (!done_with_all_customers) {
5          sem_wait(&barber_bed);
6          if(!done_with_all_customers) {
7            int waitTime = UD(MIN_TIME, MAX_TIME); /* Cut hair */
8            sleep(waitTime);
9            sem_post(&done_with_customer); /* Signal chair is free */
10         }
11         else{
12           break; /* Done with all customers */
13         }
14     }
15 }
```

Code for the customer thread is shown next. The customer attempts to enter the waiting room by probing on the *waiting_room* semaphore in line 10. Since the semaphore is initialized to n, the first n threads will decrement the semaphore and proceed into the waiting room whereas the next thread will block on the semaphore until there is space. Once in the waiting room, the next step is to wait for an available barber chair (line 11). Once the thread moves to the barber chair, it signals on *waiting_room* to indicate that there is now one free spot in the waiting room. Therefore, threads

outside the waiting room, that is blocked on line 10, can now compete to acquire the semaphore. Once in the chair, the thread wakes up the barber by signaling on *barber_bed* and waits for the barber to unblock it from line 15 after hair is trimmed. Finally, the thread signals on *barber_seat*, indicating to other threads in the waiting room (line 11) that the barber chair is now free.

```
1  void *customer(void *customer_number)
2  {
3      int number = *(int *)customer_number;
4      int wait_time = UD(MIN_TIME, MAX_TIME); /* Go to barber */
5      sleep(wait_time);
6
7      sem_wait(&waiting_room); /* Wait to get into the barber shop */
8      sem_wait(&barber_seat); /* Wait for barber to become free */
9      sem_post(&waiting_room); /* Let people waiting outside shop know */
10     sem_post(&barber_bed); /* Wake up barber */
11     sem_wait(&done_with_customer); /* Wait until hair is cut */
12     sem_post(&barber_seat); /* Give up seat */
13 }
```

The file *sleeping_barber.c*, available on BBLearn, contains the complete code listing.

Readers-Writers Locks

Consider large data structures that to be shared among several concurrent threads such as databases and linked lists. Figure 2 shows an example of a singly linked list in which each list element consists of a data value and a pointer to the next element in the list. The *head* pointer always points to the first element in the list while the optional *tail* pointer points to the last element.

```
typedef struct element_s {
    int value;  /* Value of the linked-list element */
    struct element_s *next; /* Pointer to next linked-list element */
} element_t;

typedef struct linked_list_s {
    element_t *head;    /* Head of the linked list */
    element_t *tail;    /* Tail of the linked list */
} linked_list_t;

link_list_t *list;
```

Assume that the linked list will always be maintained in sorted fashion and that duplicate elements are not allowed. During normal operation, some of the threads may want only to read the linked list to determine if an element is in the list whereas other threads may want to update it, that is insert and delete elements from the linked list. Let us now review the insert and delete operations on a singly linked list.

Insert operation: Figure 3 shows the major steps involved in inserting an element *new* between the *prev* and *curr* pointers. It is a two-step process during which the pointers are manipulated as shown in Figs. 3(a) and (b).

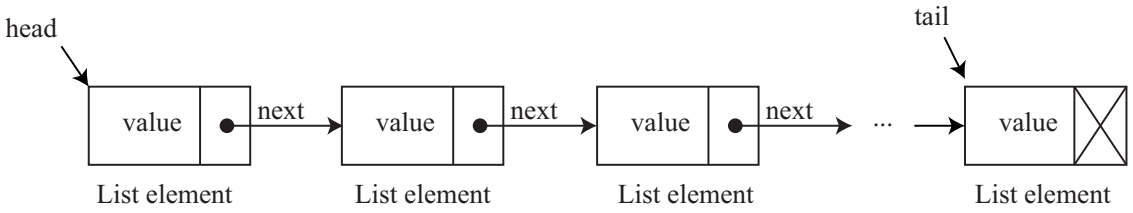
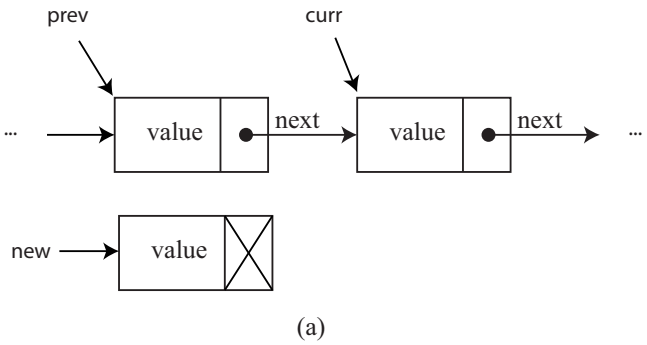
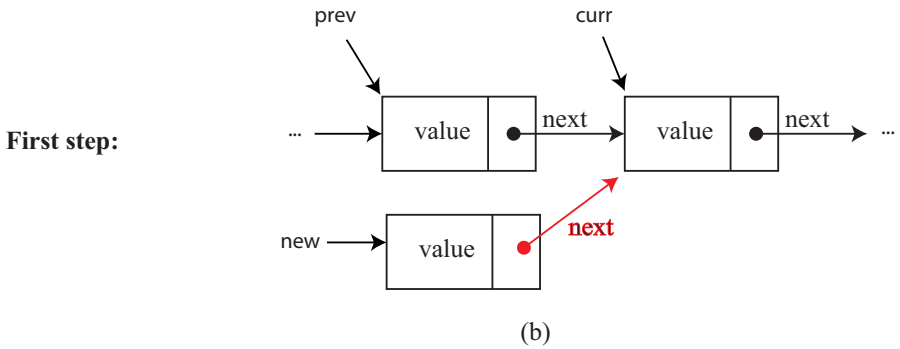


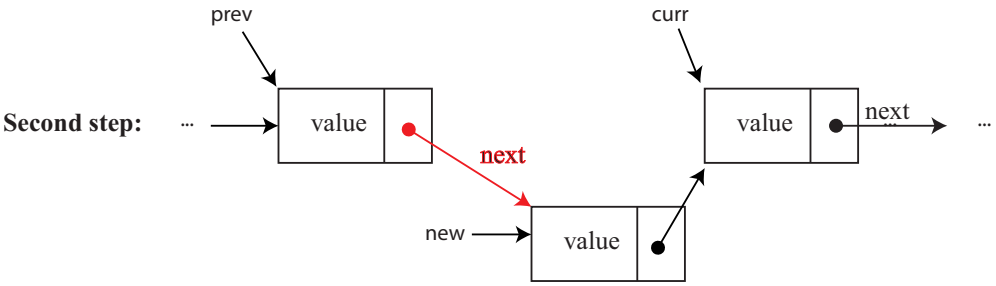
Figure 2: Structure of the singly linked list.



(a)



(b)



(c)

Figure 3: Inserting an element into the linked list.

The following code listing shows the *insert()* function that accounts for all insertion-related cases; for example, if the element is to be inserted at the head (or tail) of the list. The function returns 0 on success or -1 if the value is a duplicate of a list element.

```c
int insert(linked_list_t *list, int value)
{
    element_t *curr, *prev;
    if (list->head == NULL) {    /* List is empty */
        element_t *new = (element_t *)malloc(sizeof(element_t));
        new->value = value;
        new->next = NULL;
        list->head = new;
        return 0;
    }

    curr = list->head;
    if (value < curr->value) {  /* Insert value as first element */
        element_t *new = (element_t *)malloc(sizeof(element_t));
        new->value = value;
        new->next = curr;
        list->head = new;
        return 0;
    }

    prev = curr;
    curr = curr->next;
    while (curr != NULL) { /* Search for location within list */
        if ((value > prev->value) && (value < curr->value) {
            element_t *new = (element_t *)malloc(sizeof(element_t));
            new->value = value;
            prev->next = new;
            new->next = curr;
            return 0;
        }
        prev = curr;
        curr = curr->next;
    }

    if (value > prev->value) {  /* Insert value as last element */
         element_t *new = (element_t *)malloc(sizeof(element_t));
         new->value = value;
         new->next = NULL;
         prev->next = new;
         return 0;
    }

    return -1; /* Value already exists in the list */
}
```

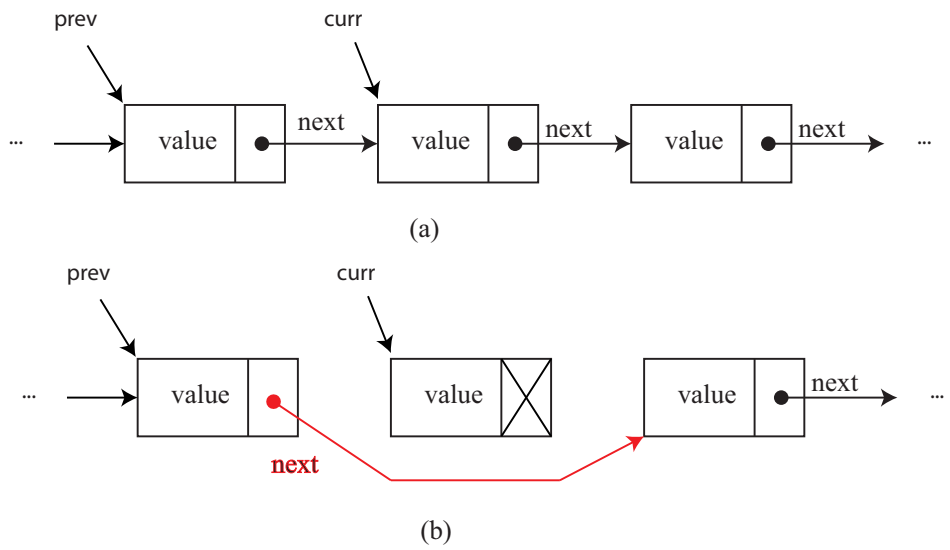


Figure 4: Deleting an element from the linked list.

Delete operation: Figure 4 shows the major steps involved in deleting a list element pointed to by *curr*. The following code listing shows the *delete()* function which returns 0 on success or -1 if the value does not exist in the list.

```
int delete(linked_list_t *list, int value)
{
    element_t *curr = list->head;
    if (curr == NULL)   /* List is empty */
        return -1;

    if (curr->value == value) { /* Delete first element */
        list->head = curr->next;
        free((void *)curr);
        return 0;
    }

    element_t *prev = curr;
    curr = curr->next;
    while (curr != NULL) {
        if (curr->value == value) {
            prev->next = curr->next;
            free((void *)curr);
            return 0;
        }
        prev = curr;
        curr = curr->next;
    }

    return -1; /* Element not in list */
}
```

The other common operation involves searching for an element within the list, as shown by the following *is_present()* function which returns 0 if the value is found within the list, -1 otherwise.

```
int is_present(linked_list_t *list, int value)
{
    element_t *curr = list->head;
    if (curr == NULL) /* Empty list */
        return -1;

    while (curr != NULL) {
        if (curr->value == value)
            return 0;
        curr = curr->next;
    }

    return -1;
}
```

Now that we have reviewed the basic operations that can be performed on a linked list, let us discuss the scenario in which multiple threads can operate within the data structure simultaneously — that is, execute the *insert()*, *delete()*, and *is_present()* functions concurrently. If multiple threads are allowed to perform these functions concurrently without protecting the linked list, many *race conditions* are possible. For example:

- Returning to Fig. 3(c), if one thread is deleting the element pointed to by *prev* (or *curr*) while another is simultaneously trying to insert *new* between *prev* and *curr*.
- Returning to Fig. 4, if two threads are simultaneously trying to delete *curr*.
- If *is_present()* and *delete()* are executed simultaneously.
- If *is_present()* and *insert()* are executed simultaneously.
- If two *insert()* functions are executed simultaneously.

The race conditions can be eliminated by simply locking the entire linked list at a very coarse granularity before any operation can be performed by a thread. A lock is associated with the linked-list structure as follows.

```
typedef struct linked_list_s {
    element_t *head;    /* Head of the linked list */
    element_t *tail;    /* Tail of the linked list */
    pthread_mutex_t lock; /* Lock variable protecting linked list */
} linked_list_t;

link_list_t *list;
```

Then, prior to performing any operation, the mutex must be first acquired as follows:

```
pthread_mutex_lock(&list->lock);
insert(list, value);
pthread_mutex_unlock(&list->lock);

pthread_mutex_lock(&list->lock);
delete(list, value);
pthread_mutex_unlock(&list->lock);

pthread_mutex_lock(&list->lock);
is_present(list, value);
pthread_mutex_unlock(&list->lock);
```

The obvious problem is that we are serializing access to the list no matter the operation; and if the vast majority are calls to the *is_member()* function, this will result in significant performance degradation. Clearly, any number of read operations can be simultaneously performed on the linked list with no adverse effects on the data structure.

Read-write locks, that are supported by the Pthread library, provide a solution to the above problem. A read-write lock is like a mutex except that is provides two lock functions:

- The *pthread_rwlock_rdlock()* function locks the read-write lock for reading.
- The *pthread_rwlock_wrlock()* function locks the read-write lock for writing.
- Multiple threads can simultaneously obtain the read lock for the linked list by calling the *pthread_rwlock_rdlock()* function.
- Only one thread can obtain the lock by calling the *pthread_rwlock_wrlock()* function.
- If any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the *pthread_rwlock_wrlock()* function.
- If any thread owns the lock for writing, any other threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Using the above-described read-write locks, the code can be rewritten as follows.

```
pthread_rwlock_wrlock(&list->rwlock);
insert(list, value);
pthread_rwlock_unlock(&list->lock);

pthread_rwlock_wrlock(&list->lock);
delete(list, value);
pthread_rwlock_unlock(&list->lock);

pthread_rwlock_rdlock(&list->lock);
is_present(list, value);
pthread_rwlock_unlock(&list->lock);
```

Since writers must have exclusive access to the linked list, it is likely that it is much more difficult for a writer to obtain the lock. Many implementations therefore give writers preference.

The files *multi_threaded_linked_list_v1.c* and *multi_threaded_linked_list_v2.c*, available on BBLearn, contain complete code listings for the multi-threaded implementations discussed here.

The pseudo-code for the writer and reader threads is shown below.

```
while (1) {
    sem_wait(write_lock);
    /* Perform write */
    sem_post(write_lock);
}
```

```
while (1) {
    sem_wait(mutex); /* Acquire mutex for read_count */
    read_count++;
    if (read_count == 1)
        sem_wait(write_lock); /* Try to acquire the write lock */
    sem_post(mutex);

    /* Perform read */

    sem_wait(mutex);
    read_count--;
    if (read_count == 0)
        sem_post(write_lock);
    sem_post(mutex);
}
```

The *write_lock* and *mutex* semaphores are both initialized to 1, and the shared variable *read_count* is initialized to 0. The *write_lock* is shared between both reader and writer threads. The *mutex* semaphore is used to ensure mutual exclusion when the *read_count* variable is updated. This variable keeps track of how many threads are currently reading the critical section. The semaphore *write_lock* functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. This is not used by readers that enter or exit while other readers are in their critical sections. If a writer is in the critical section and n readers are waiting, then one reader is queued on *write_lock* and $n - 1$ readers are queued on *mutex*. When the writer releases *write_lock*, we may resume execution of either waiting readers or a single waiting writer. The selection is made by the scheduler.

Reader-writer locks are most useful in applications that have many more readers than writers. This is because reader-writer locks generally require more overhead to establish than simply using semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock. The provided code examples use a mix of 80% *is_present()*, 10% *insert()*, and 10% *delete()* operations to test and time the implementation.