CS 332/532 Systems Programming

Lecture 31

Semaphores / 2

Professor : Mahmut Unan – UAB CS

Agenda

- Semaphores
- Thread synchronization using semaphores

Producer – Consumer Problem

- The **Producer-Consumer Problem** is a classic synchronization problem in computer science. It involves two processes/threads:
  1. **Producer**: Generates data and adds it to a shared buffer.
  2. **Consumer**: Consumes data from the shared buffer.
- The challenge is to ensure:
- The producer doesn't add data to a full buffer.
- The consumer doesn't remove data from an empty buffer.
- Proper synchronization is required to avoid issues like **race conditions** or **deadlocks**.

Where We Observe the Problem?

- **Multithreaded applications**: Where threads share a resource like a job queue.
- **Operating Systems**: In processes communicating through shared memory or pipes.
- **Database Systems**: When managing a pool of connections.
- **IoT**: When a sensor (producer) sends data to a server (consumer) at varying speeds.

Role of Semaphores

- **Semaphores** are synchronization primitives that help coordinate access to the shared buffer.

1. **Counting Semaphore**:
   1. Used to count available buffer slots.
   2. Two semaphores are commonly used:
      1. **full**: Tracks the number of filled slots (used by the consumer).
      2. **empty**: Tracks the number of empty slots (used by the producer).
2. **Binary Semaphore (Mutex)**:
   1. Ensures mutual exclusion to prevent simultaneous access to the buffer.

sem_init

- **#include <semaphore.h>**
- **int sem_init(sem_t ****sem*, **int** *pshared*, **unsigned int** *value***);**
- Link with *-pthread*.

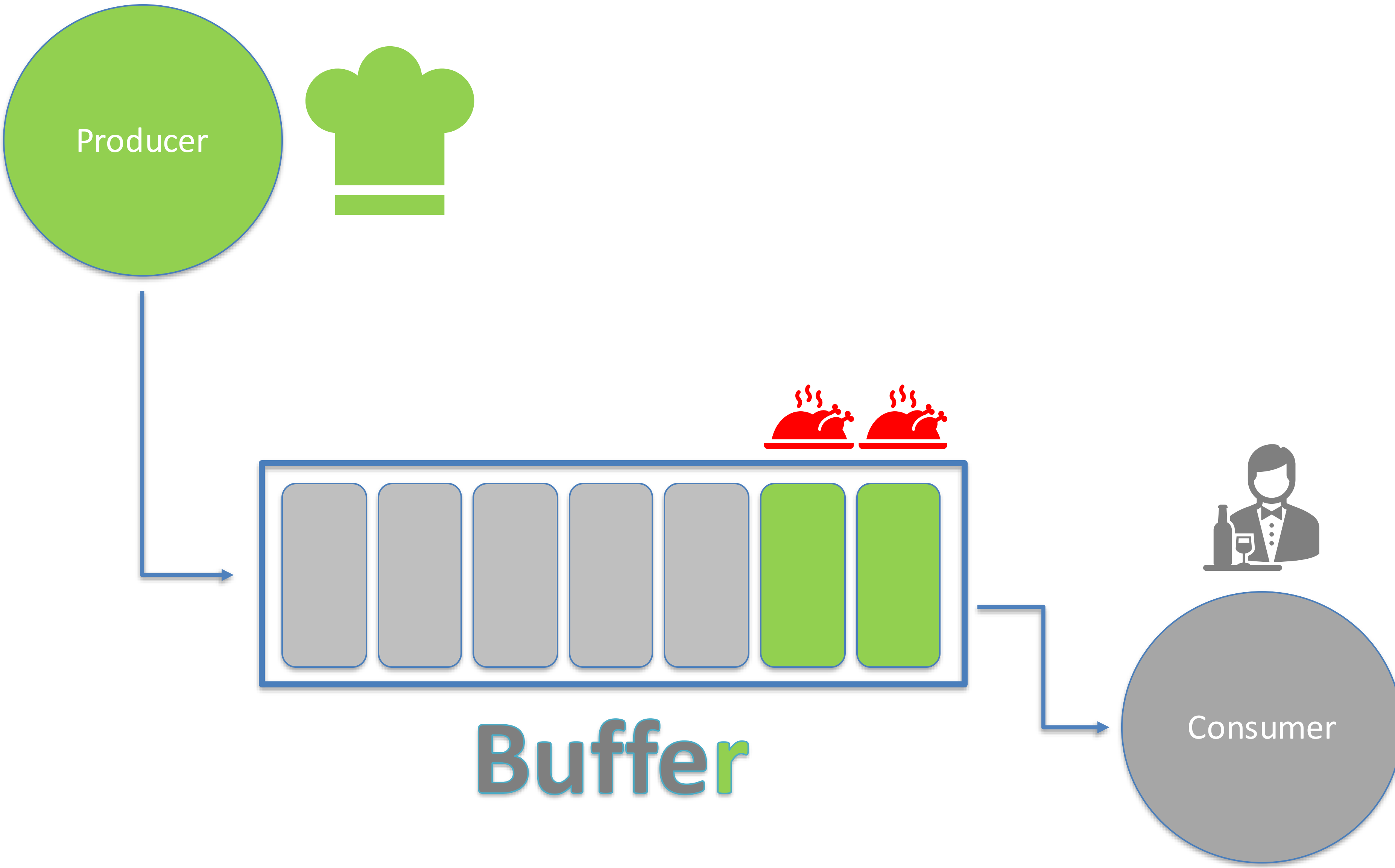
- **sem_init**() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore. The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

https://www.man7.org/linux/man-pages/man3/sem_init.3.html

Thread 1	Thread 2	data
`sem_wait (&mutex);`	---	0
---	`sem_wait (&mutex);`	0
`a = data;`	`/* blocked */`	0
`a = a+1;`	`/* blocked */`	0
`data = a;`	`/* blocked */`	1
`sem_post (&mutex);`	`/* blocked */`	1
`/* blocked */`	`b = data;`	1
`/* blocked */`	`b = b - 1;`	1
`/* blocked */`	`data = b;`	2
`/* blocked */`	`sem_post (&mutex);`	2

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem_wait* and *sem_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens



Producer
Buffer
Consumer

prodcons1.c

```c
/* Solution to the single Producer/Consumer problem using semaphores.
   This example uses a circular buffer to put and get the data
   (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
   To run: ./<filename> <#items>

   To enable printing add -DDEBUG to compile:
   gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

```

```c

#define  NBUFF           10

int      nitems;                                /* read-only */

struct {                  /* data shared by producer and consumer */
  int       buff[NBUFF];
  sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  pthread_t       tid_producer, tid_consumer;

  if (argc != 2) {
    printf("Usage: %s <#items>\n", argv[0]);
    exit(-1);
  }

```

```c

  nitems = atoi(argv[1]);

  /* initialize three semaphores */
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.nempty, 0, NBUFF);
  sem_init(&shared.nstored, 0, 0);

  /* create one producer thread and one consumer thread */
  pthread_create(&tid_producer, NULL, producer, NULL);
  pthread_create(&tid_consumer, NULL, consumer, NULL);

  /* wait for producer and consumer threads */
  pthread_join(tid_producer, NULL);
  pthread_join(tid_consumer, NULL);

  /* remove the semaphores */
  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
  }
  return (NULL);
}
/* end producer */
```

```c
/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);        /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
            shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);         /* 1 more empty slot */
  }
  return (NULL);
}
/* end consumer */
```

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```
int       nitems;              /* read-only */
struct {
/* data shared by producer and consumer */
   int        buff[NBUFF];
    sem_t      mutex, nempty, nstored;
      /* semaphores */
} shared;
```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem_init* as per the pseudocode from the textbook.

```
nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);
```

- We create two separate threads, one for the producer and one for the consumer, and wait for the two threads to complete.

```
/* create one producer thread and one
consumer thread */
pthread_create(&tid_producer, NULL,
producer, NULL);
pthread_create(&tid_consumer, NULL,
consumer, NULL);

/* wait for producer and consumer threads */
pthread_join(tid_producer, NULL);
pthread_join(tid_consumer, NULL);
```

- Now let us look at the producer thread.
- It executes a loop equal to the number of items specified (*nitems*) and during each iteration of the loop, waits on the semaphore *nempty*.
- Initially *nempty* is set to *NBUFF*, so *sem_wait* returns immediately and waits on the semaphore *mutex*.
  Since *mutex* is initially set to 1, the producer thread enters the critical section and assigns the value *i* to the buffer location *i % NBUFF* and then release the *mutex* semaphore (calls *sem_post* on the semaphore *mutex*).
- Now that there is at least one element in the buffer, it also posts *sem_post* on the semaphore *nstored* to indicate to the consumer that there is an element in the buffer and continues with the loop.
- The producer thread terminates when the loop completes (i.e., after *nitems* iterations).
-

- ```
  /* producer function */
  void *producer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
      sem_wait(&shared.nempty);        /* wait for at least
  1 empty slot */
      sem_wait(&shared.mutex);

      shared.buff[i % NBUFF] = i;     /* store i into
  circular buffer */
  #ifdef DEBUG
      printf("wrote %d to buffer at location %d\n", i, i %
  NBUFF);
  #endif

      sem_post(&shared.mutex);
      sem_post(&shared.nstored);      /* 1 more stored
  item */
    }

    return (NULL);
  }
  /* end producer */
  ```
  ```

- Meanwhile, the consumer thread will enter the loop and wait on the semaphore *nstored*.
- Since initially *nstored* is set to 0, this call will block and the consumer will wait until the producer posts on the semaphore *nstored*.
- When the producer posts on the semaphore *nstored*, the consumer will return from *sem_wait* on *nstored* and invoke the *sem_wait* on the semaphore *mutex*.
- If the producer is not in the critical section, the consumer will obtain the *mutex* semaphore, consume the buffer (we simply check if the value in the buffer match the corresponding *(loop index mod NBUFF)* and print an error message in case they don't match), and release the mutex by calling *sem_post* on the *mutex* semaphore.
- Then the consumer thread will post the *sem_post* on the semaphore *nempty* to indicate to the producer that now there is an empty slot. The consumer thread terminates when the loop completes (i.e., after *nitems* iterations).

- ```
  /* consumer function */
  void *consumer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
      sem_wait(&shared.nstored);       /* wait for at
  least 1 stored item */
      sem_wait(&shared.mutex);

      if (shared.buff[i % NBUFF] != i)
        printf("error: buff[%d] = %d\n", i, shared.buff[i
  % NBUFF]);
  #ifdef DEBUG
      printf("read %d from buffer at location %d\n",
              shared.buff[i % NBUFF], i % NBUFF);
  #endif

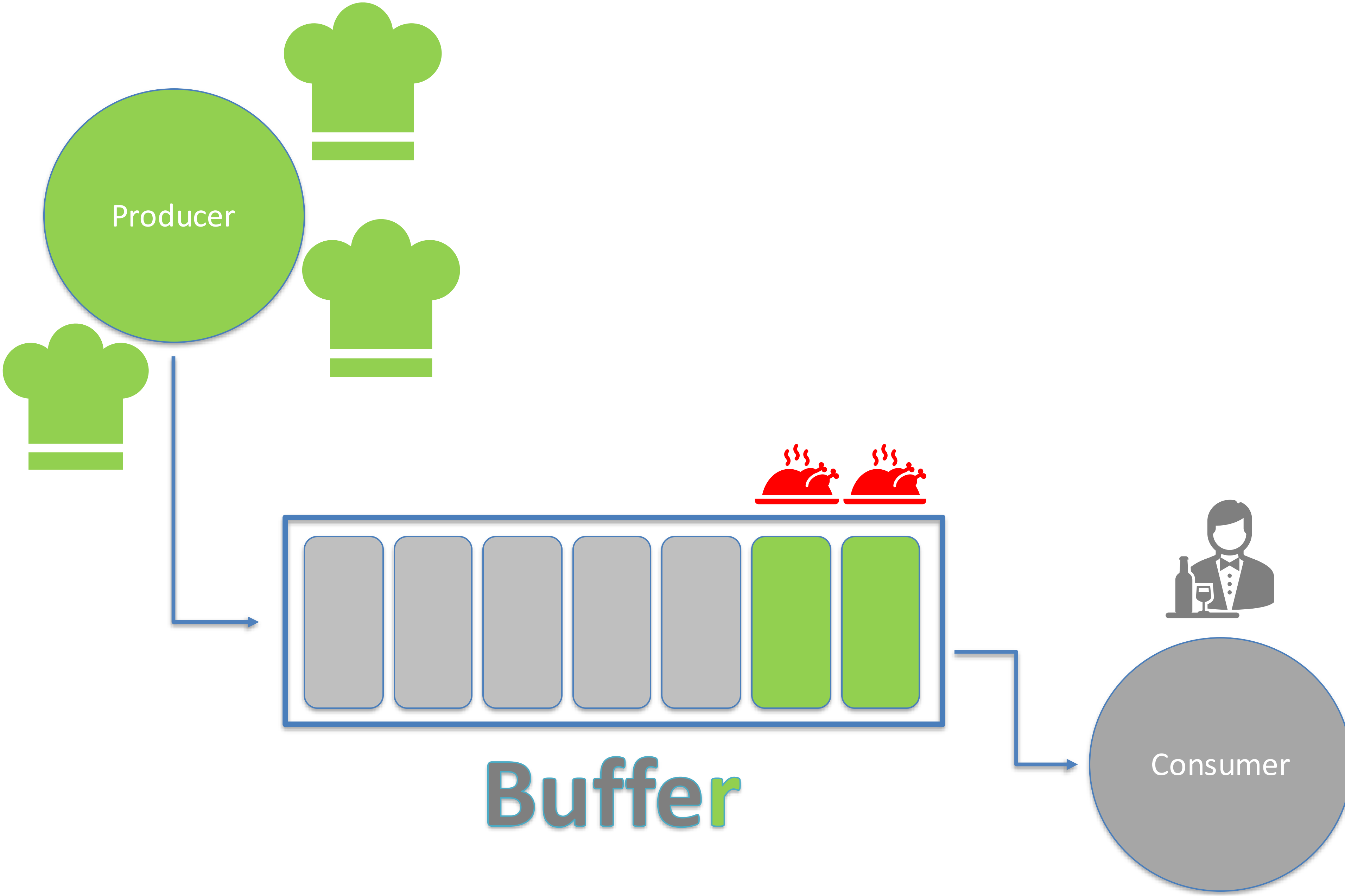
      sem_post(&shared.mutex);
      sem_post(&shared.nempty);        /* 1 more empty
  slot */
    }

    return (NULL);
  }
  /* end consumer *
  ```
  ```

- You can compile the program with the DEBUG variable defined using -DDEBUG during compilation and see how the two threads progress. Here is a sample output when we execute the program with 20 items. You will notice that the output would be different every time you execute the program even with the same number of elements.

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**

```
$ ./a.out 20
wrote 0 to buffer at location 0
wrote 1 to buffer at location 1
wrote 2 to buffer at location 2
wrote 3 to buffer at location 3
wrote 4 to buffer at location 4
wrote 5 to buffer at location 5
wrote 6 to buffer at location 6
wrote 7 to buffer at location 7
wrote 8 to buffer at location 8
read 0 from buffer at location 0
read 1 from buffer at location 1
read 2 from buffer at location 2
read 3 from buffer at location 3
read 4 from buffer at location 4
read 5 from buffer at location 5
read 6 from buffer at location 6
read 7 from buffer at location 7
read 8 from buffer at location 8
wrote 9 to buffer at location 9
wrote 10 to buffer at location 0
wrote 11 to buffer at location 1
wrote 12 to buffer at location 2
wrote 13 to buffer at location 3
wrote 14 to buffer at location 4
wrote 15 to buffer at location 5
wrote 16 to buffer at location 6
wrote 17 to buffer at location 7
wrote 18 to buffer at location 8
read 9 from buffer at location 9
read 10 from buffer at location 0
read 11 from buffer at location 1
read 12 from buffer at location 2
read 13 from buffer at location 3
read 14 from buffer at location 4
read 15 from buffer at location 5
read 16 from buffer at location 6
read 17 from buffer at location 7
read 18 from buffer at location 8
wrote 19 to buffer at location 9
read 19 from buffer at location 9
```


Producer
Buffer
Consumer

prodcons2.c

- Solution to multiple producer and single consumer problem:

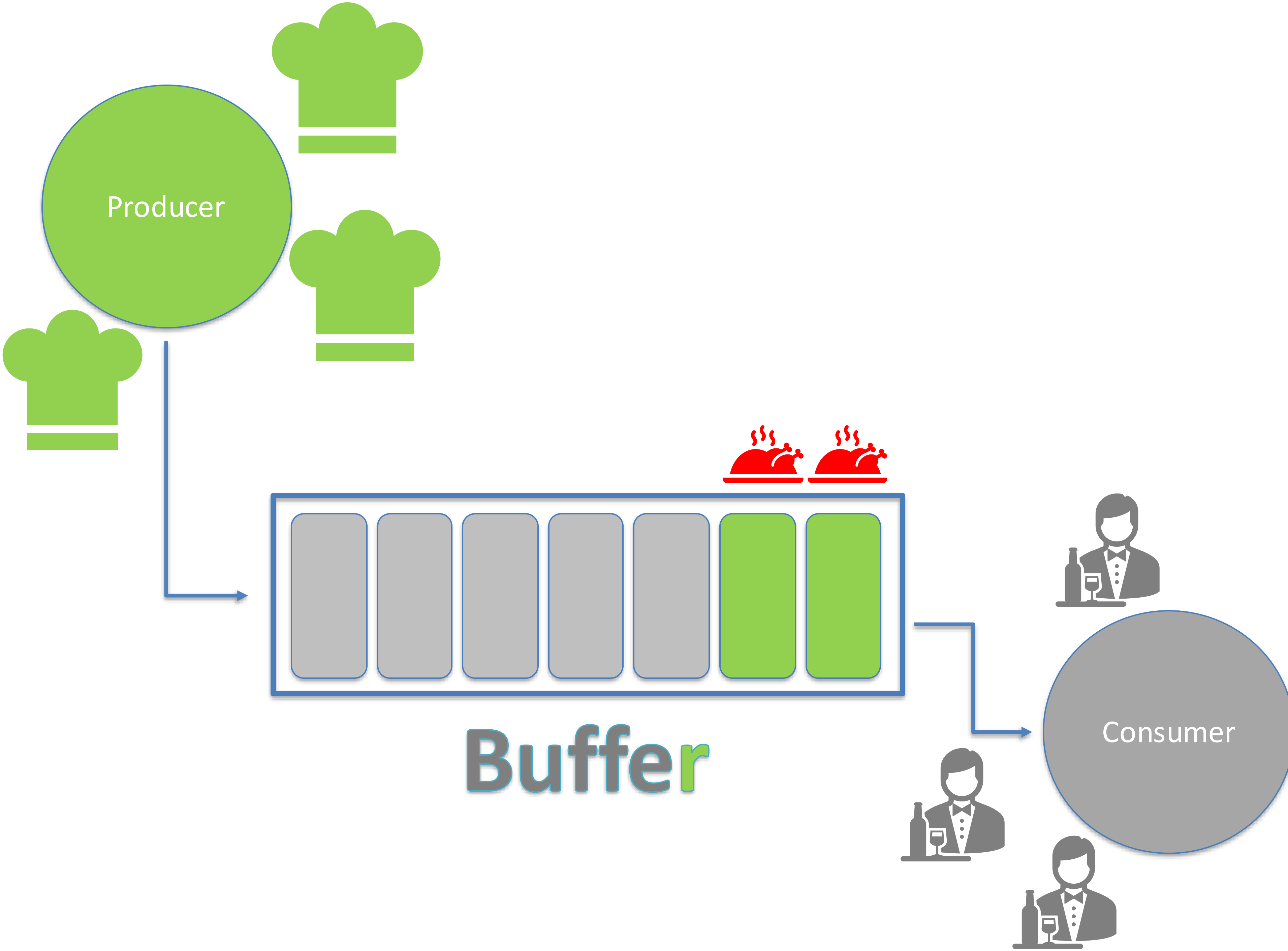
```c
/* Solution to the Multiple Producer/Single Consumer problem using
   semaphores. This example uses a circular buffer to put and get the
   data (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

#define  min(a, b)  (((a) < (b)) ? (a) : (b))

#define  NBUFF         10
#define  MAXNTHREADS   100
```

```c

int      nitems, nproducers;                       /* read-only */

struct {                    /* data shared by producers and consumers */
  int        buff[NBUFF];
  int        nput;                         /* item number: 0, 1, 2, ... */
  int        nputval;                      /* value to store in buff[] */
  sem_t      mutex, nempty, nstored;    /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  int              i, prodcount[MAXNTHREADS];
  pthread_t        tid_producer[MAXNTHREADS], tid_consumer;

  if (argc != 3) {
    printf("Usage: %s <#items> <#producers>\n", argv[0]);
    exit(-1);
  }

```

```c
  nitems = atoi(argv[1]);
  nproducers = min(atoi(argv[2]), MAXNTHREADS);

  /* initialize three semaphores */
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.nempty, 0, NBUFF);
  sem_init(&shared.nstored, 0, 0);

  /* create all producers and one consumer */
  for (i = 0; i < nproducers; i++) {
    prodcount[i] = 0;
    pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
  }
  pthread_create(&tid_consumer, NULL, consumer, NULL);

  /* wait for all producers and the consumer */
  for (i = 0; i < nproducers; i++) {
    pthread_join(tid_producer[i], NULL);
    printf("producer count[%d] = %d\n", i, prodcount[i]);
  }
  pthread_join(tid_consumer, NULL);

  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  for ( ; ; ) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    if (shared.nput >= nitems) {
      sem_post(&shared.nempty);
      sem_post(&shared.mutex);
      return(NULL);                 /* all done */
    }

    shared.buff[shared.nput % NBUFF] = shared.nputval;
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n",
           shared.nputval, shared.nput % NBUFF);
#endif
    shared.nput++;
    shared.nputval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
    *((int *) arg) += 1;
  }
}
/* end producer */
```

```c

/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
           shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);        /* 1 more empty slot */
  }

  return (NULL);
}
/* end consumer */
```


Producer
Buffer
Consumer

prodcons3.c

- Solution to multiple producer and multiple consumer problem:

```c
/* Solution to the Multiple Producer/Multiple Consumer problem using
   semaphores. This example uses a circular buffer to put and get the
   data (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

#define  min(a, b)  (((a) < (b)) ? (a) : (b))
```

```c

#define  NBUFF          10
#define  MAXNTHREADS    100

int      nitems, nproducers, nconsumers;              /* read-only */

struct {                  /* data shared by producers and consumers */
  int       buff[NBUFF];
  int       nput;                      /* item number: 0, 1, 2, ... */
  int       nputval;                   /* value to store in buff[] */
  int       nget;                      /* item number: 0, 1, 2, ... */
  int       ngetval;                   /* value fetched from buff[] */
  sem_t     mutex, nempty, nstored;    /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  int             i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
  pthread_t       tid_producer[MAXNTHREADS], tid_consumer[MAXNTHREADS];

  if (argc != 4) {
    printf("Usage: %s <#items> <#producers> <#consumers>\n", argv[0]);
    exit(-1);
  }
```

```c

    nitems = atoi(argv[1]);
    nproducers = min(atoi(argv[2]), MAXNTHREADS);
    nconsumers = min(atoi(argv[3]), MAXNTHREADS);

    /* initialize three semaphores */
    sem_init(&shared.mutex, 0, 1);
    sem_init(&shared.nempty, 0, NBUFF);
    sem_init(&shared.nstored, 0, 0);

    /* create all producers and all consumers */
    for (i = 0; i < nproducers; i++) {
      prodcount[i] = 0;
      pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
      conscount[i] = 0;
      pthread_create(&tid_consumer[i], NULL, consumer, &conscount[i]);
    }

    /* wait for all producers and all consumers */
    for (i = 0; i < nproducers; i++) {
      pthread_join(tid_producer[i], NULL);
      printf("producer count[%d] = %d\n", i, prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
      pthread_join(tid_consumer[i], NULL);
      printf("consumer count[%d] = %d\n", i, conscount[i]);
    }
```

```c

  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */

/* producer function */
void *producer(void *arg)
{
  for ( ; ; ) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    if (shared.nput >= nitems) {
      sem_post(&shared.nstored);    /* let consumers terminate */
      sem_post(&shared.nempty);
      sem_post(&shared.mutex);
      return(NULL);                 /* all done */
    }

    shared.buff[shared.nput % NBUFF] = shared.nputval;
    shared.nput++;
    shared.nputval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
    *((int *) arg) += 1;
  }
}
/* end producer */
```

```c

/* consumer function */
void *consumer(void *arg)
{
  int i;

  for ( ; ; ) {
    sem_wait(&shared.nstored);        /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.nget >= nitems) {
      sem_post(&shared.nstored);
      sem_post(&shared.mutex);
      return(NULL);                   /* all done */
    }

    i = shared.nget % NBUFF;
    if (shared.buff[i] != shared.ngetval)
      printf("error: buff[%d] = %d\n", i, shared.buff[i]);
    shared.nget++;
    shared.ngetval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);         /* 1 more empty slot */
    *((int *) arg) += 1;
  }
}
/* end consumer */

```

Semaphore for Resource Allocation

- Pool of N problems
- Resource sharing among multiple processes / uninterrupted period
- Limit the highest number of resources in use at any time
- ………