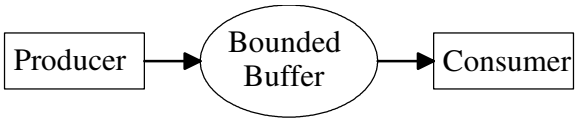
1) Complete the `bufferRemove` code called by the consumer threads.



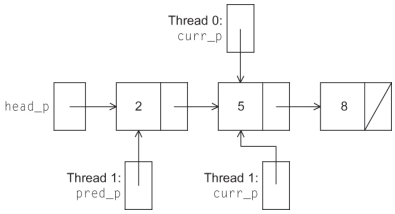

```
/* Description: Bounded buffer using pthread condition variables to avoid over or underflowing the bounded buffer. */
// Global Bounded Buffer
#define SIZE 5
int buffer[SIZE];
int count = 0;
int front = -1;
int rear = 0;

pthread_mutex_t lock;
pthread_cond_t nonFull;
pthread_cond_t nonEmpty;

void  *producerWork(void *);
void  *consumerWork(void *);
void bufferAdd(int);
int bufferRemove();

int main(int argc, char * argv[]) {
  pthread_mutex_init(&lock, NULL);
  pthread_cond_init(&nonFull, NULL);
  pthread_cond_init(&nonEmpty, NULL);
...
 } // end main

void  *producerWork(void * args) {
  int threadId;
  int counter = 0;
  unsigned int sleepAmt;

  threadId = *((int *) args);
  while (counter < MAX_DURATION) {
    sleepAmt = rand() % MAX_PRODUCER_SLEEP;
    sleep(sleepAmt);
    printf("Producer try to add when count = %d\n", count);
    bufferAdd(threadId);
    printf("Producer added\n");
    counter++;
  } // end while
} // end producerWork

void bufferAdd(int item) {
  pthread_mutex_lock(&lock);
  while(count == SIZE) {
    while(pthread_cond_wait(&nonFull, &lock) != 0);
  } // end while
  if (count == 0) {
    front = 0;
    rear = 0;
  } else {
    rear = (rear + 1) % SIZE;
  } // end if
  buffer[rear] = item;
  count++;
  pthread_cond_signal(&nonEmpty);
  pthread_mutex_unlock(&lock);

} // end bufferAdd
```

2) Shared sorted linked list textbook example with three operations Member (search), Delete, and Insert.

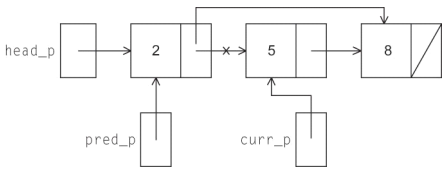
a) Which operation(s) need mutex exclusion?

b) Which operation(s) can occur simultaneously?

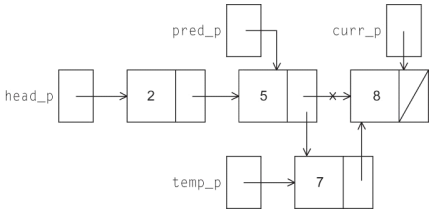
Member



Delete



Insert



Textbooks discusses three solutions:

- Threads put mutex lock and unlock around their calls to list operations
- Lock individual nodes by putting mutex in each node. ("fine-grained" approach)

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

- Read-Write Locks to allow multiple readers/Member operations

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete

Implementation	Number of Threads 1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread
80% Member
10% Insert
10% Delete

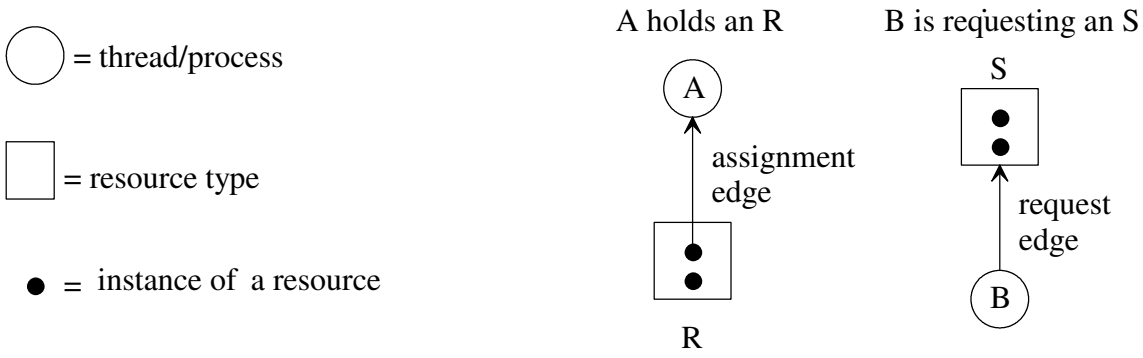
c) Explain the results?

3. **Necessary Conditions for Deadlock** (Coffman, et al 1971): ALL MUST HOLD!

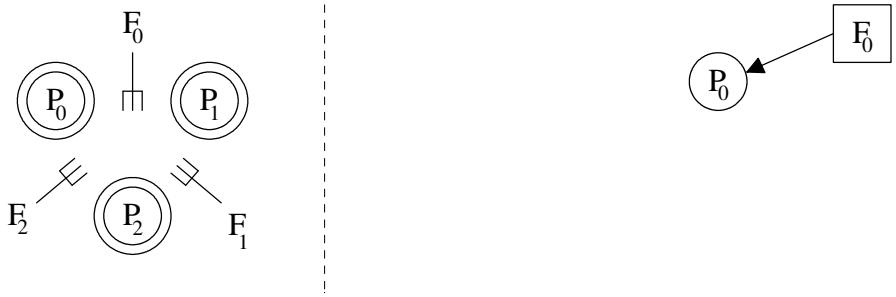
- mutual exclusion: resource(s) used in mutually exclusive fashion
- hold and wait: threads/processes currently holding granted resources can request new resources
- no preemption: resources are released voluntarily by the thread/process holding it after it has completed its task
- circular wait: a circular chain of two or more threads/processes that are each waiting for a resource held by the next member of the chain.

A mutex is a single-instance resource drawn as just a box in the text.

Resource-Allocation Graph (Holt '72) - (a directed graph model)



a) Consider the Dining Philosophers problem below. Complete the resource-allocation graph showing deadlock if all the philosophers are holding their left forks.



Strategies used for dealing with deadlock:

1) Ignore the problem - "Ostrich algorithm" - Is it a rare occurrence with minor consequences?
2) Detection and recovery - Periodically run an algorithm that checks for cycles in a resource-allocation graph. Rollback or kill a thread/process to break the cycle.
3) Deadlock Prevention - OS rules that prevent one of the 4 necessary conditions for deadlock
4) Deadlock Avoidance - dynamically avoid deadlock by careful resource allocation - make sure that there is some order that threads/processes can finish execution without deadlock.

4. One appoach to *deadlock prevention is through resource ordering* - resources are ordered (e.g., numbered artificially) and we require threads/processes to request needed resources in ascending order.

a) Consider your resource-allocation graph for the above Dining Philosophers problem. If forks must be acquired in ascending order, how is the above deadlock prevented?

```
/* File:  PhilosopherB.java
   Description:  Solve the Dining Philosopher problem by preventing
   deadlock from occurring.  Philosopher 0 acquires his right fork before
   his left fork, so the deadlock cycle is prevented.
 */
public class PhilosopherB extends Thread{

      private Object leftFork, rightFork;
      private int myNumber;

      public PhilosopherB(Object left, Object right, int number){
            leftFork = left;
            rightFork = right;
            myNumber = number;
      }

      public void run(){
            int timesDined = 0;
            while(true){
                  synchronized(leftFork){
                        synchronized(rightFork){
                              timesDined++;
                        }
                  }
                  if(timesDined % 100000 == 0)
                        System.err.println("Thread " + myNumber + " is running.");
            }
      }

      public static void main(String[] args){
            final int PHILOSOPHERS = 5;
            Object[] forks = new Object[PHILOSOPHERS];
            for(int i = 0; i < PHILOSOPHERS; i++){
                  forks[i] = new Object();
            }

            PhilosopherB p = new PhilosopherB(forks[1], forks[0], 0);
            p.start();

            for(int i = 1; i < PHILOSOPHERS; i++){
                  int next = (i+1) % PHILOSOPHERS;
                  p = new PhilosopherB(forks[i], forks[next], i);
                  p.start();
            }
      }
}
```