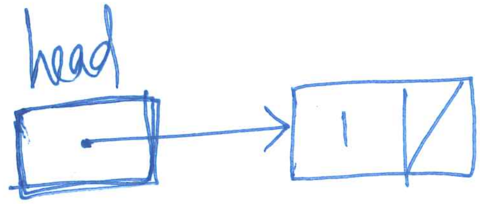
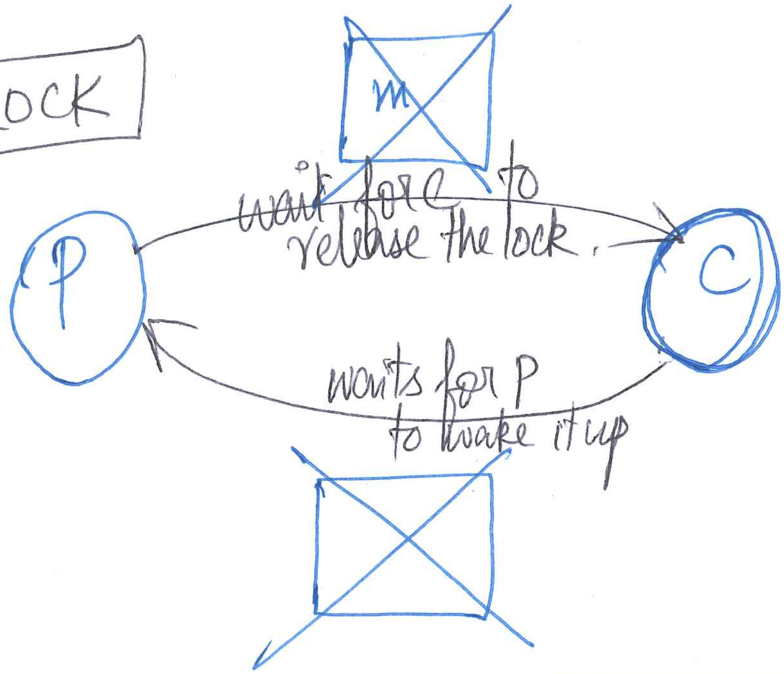
1. Lock around malloc?
2. Lookup code - how to lock?

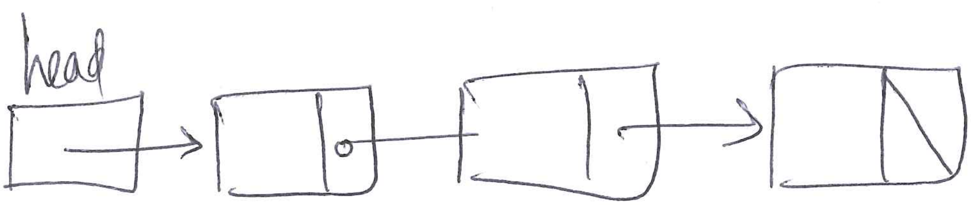
```
// Make this Linked List implementation to be thread-safe!

#include <stdio.h>
#include <stdlib.h>

typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

typedef struct __list_t {
  node_t *head;
} list_t;

void List_Init(list_t *L) { L->head = NULL; }

void List_Insert(list_t *L, int key) {            lock
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");                             unlock
    return;
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
}                                                 unlock

int List_Lookup(list_t *L, int key) {             lock
  node_t *tmp = L->head;
  while (tmp) {
    if (tmp->key == key) return 1;                unlock
    tmp = tmp->next;
  }
  return 0;                                       unlock.
}

void List_Print(list_t *L) {
  node_t *tmp = L->head;
  while (tmp) {
    printf("%d ", tmp->key);
    tmp = tmp->next;
  }
  printf("\n");
}

int main(int argc, char *argv[]) {
  list_t mylist;
  List_Init(&mylist);
  List_Insert(&mylist, 10);
  List_Insert(&mylist, 30);
  List_Insert(&mylist, 5);
  List_Print(&mylist);
  printf("In List: 10? %d 20? %d\n", List_Lookup(&mylist, 10),
         List_Lookup(&mylist, 20));
  return 0;
}
```



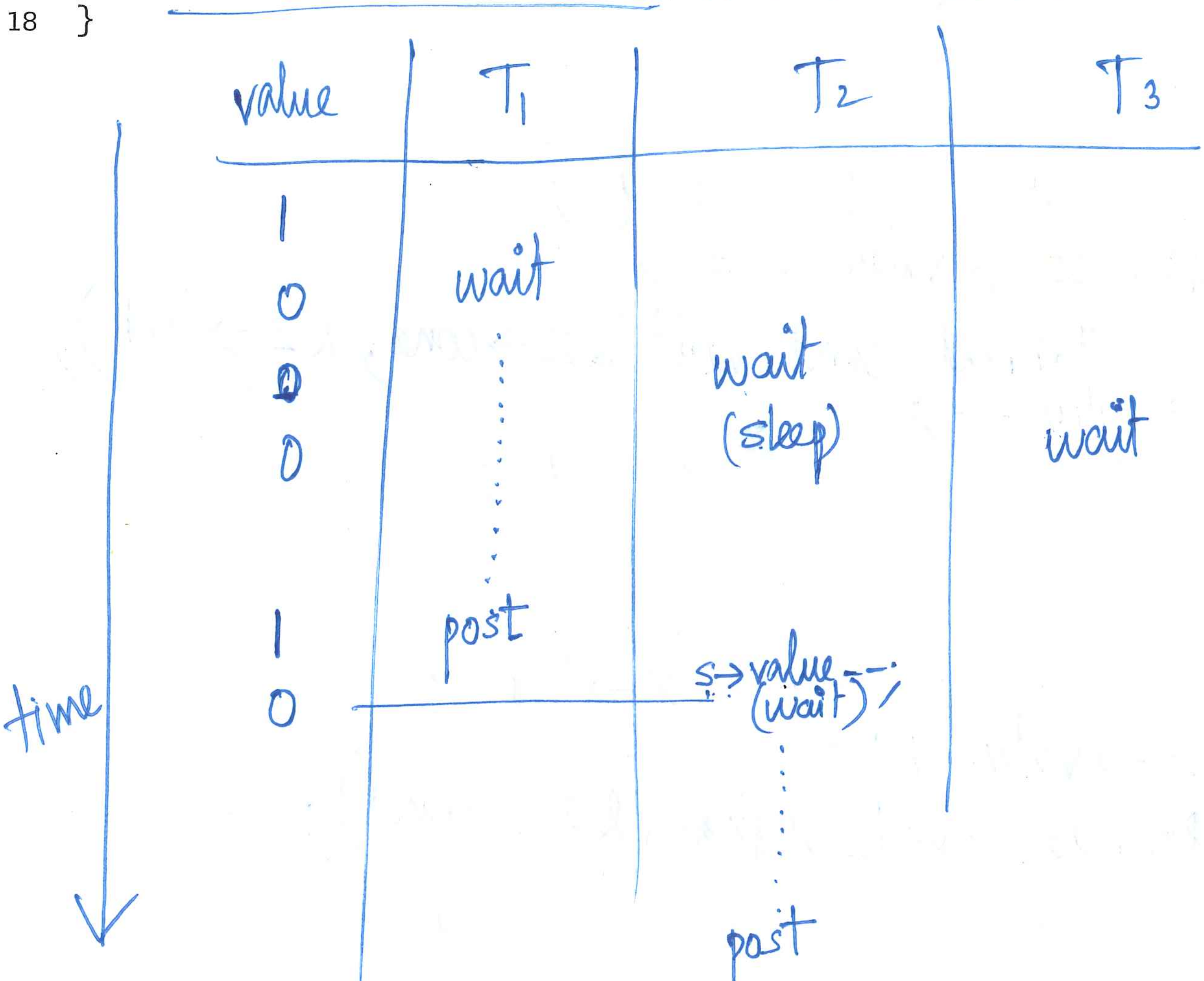
T1 T2

DEADLOCK





```
//
// ZEMAPHORE: PSEUDO-CODE
//
Zem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}

// There is a subtle difference in Zem_wait (when compared to sem_wait)
Zem_wait(Zem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}

Zem_post(Zem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}
```



```c
// Implement your own Semaphore (with the name Zemaphore!)
#ifndef __ZEMAPHORE_h__
#define __ZEMAPHORE_h__

typedef struct __Zem_t {
  int value;
  pthread_cond_t cond;    // cond_signal(c), cond_wait(c, m)
  pthread_mutex_t lock;   // mutex_lock(m),  mutex_unlock(m)
} Zem_t;

// can assume only called by one thread
void Zem_init(Zem_t *z, int value) {
  z->value = value;
  // init lock and CV
  pthread_cond_init(&z->cond, NULL);
    "   _mutex_init(&z->lock, NULL);
}

void Zem_wait(Zem_t *z) {
  // use semaphore definition as your guide
     pthread_mutex_lock(&z->lock);
        while(z->value <= 0)
            Pthread_cond_wait(&z->cond, &z->lock);
        z->value --;
    Pthread_mutex_unlock(&z->lock);
}

void Zem_post(Zem_t *z) {
  // use semaphore definition as your guide
     Pthread_mutex_lock(&z->lock);
        z->value ++;
        Pthread_cond_signal(&z->cond);
     Pthread_mutex_unlock(&z->lock);
}

#endif  // __ZEMAPHORE_h__
```

```
//Reader-Writer Locks

typedef struct _rwlock_t {
  sem_t writelock;   // to prevent multiple writers.
  sem_t lock;   //mutex.
  int readers;   // count of readers.
} rwlock_t;

void rwlock_init(rwlock_t *L) {
  L->readers = 0;
  sem_init(&L->lock, 1);
  sem_init(&L->writelock, 1);
}

void rwlock_acquire_readlock(rwlock_t *L) {
  sem_wait(&L->lock);            // a1
  L->readers++;                  // a2
  if (L->readers == 1)           // a3
    sem_wait(&L->writelock);     // a4
  sem_post(&L->lock);            // a5
}

void rwlock_release_readlock(rwlock_t *L) {
  sem_wait(&L->lock);            // r1
  L->readers--;                  // r2
  if (L->readers == 0)           // r3
    sem_post(&L->writelock);     // r4
  sem_post(&L->lock);            // r5
}

void rwlock_acquire_writelock(rwlock_t *L) {
  sem_wait(&L->writelock);
}

void rwlock_release_writelock(rwlock_t *L) {
  sem_post(&L->writelock);
}
```

R_1 : a_1 a_2 a_3 a_4 a_5 $\boxed{CS}$

R_2 :

a_1 a_2

a_3

$\cancel{a_4}$

a_5

$\boxed{CS}$

R_1 start, R_2 start, W_1 start (wait), $R_{1\,end}$, $R_{3\,start}$, $R_{4\,start}$, $R_{2\,end}$, $R_{3\,end}$.

```
// Dining Philosopers Problem
// The basic setup for the problem is this.
// Assume there are five “philosophers” sitting around a table.
// Between each pair of philosophers is a single fork (and thus,
// five total). The philosophers each have times where they think,
// and don’t need any forks, and times where they eat.
// In order to eat, a philosopher needs two forks, both the one
// on their left and the one on their right.

// Basic Loop for each philosopher
while (1) {
  think();
  getforks();
  eat();
  putforks();
}
                                        sem_t forks[5];

// Helper Functions
int left(int p) {
  return p;
}

int right(int p) {
  return (p + 1) % 5;
}

// getforks() routine
void getforks() {
    sem_wait(forks[left(p)]);
    sem_wait(forks[right(p)]);
}

// putforks() routine
void putforks() {
    sem_post(forks[left(p)]);
    sem_post(forks[right(p)]);
}
```

Dining Philosophers