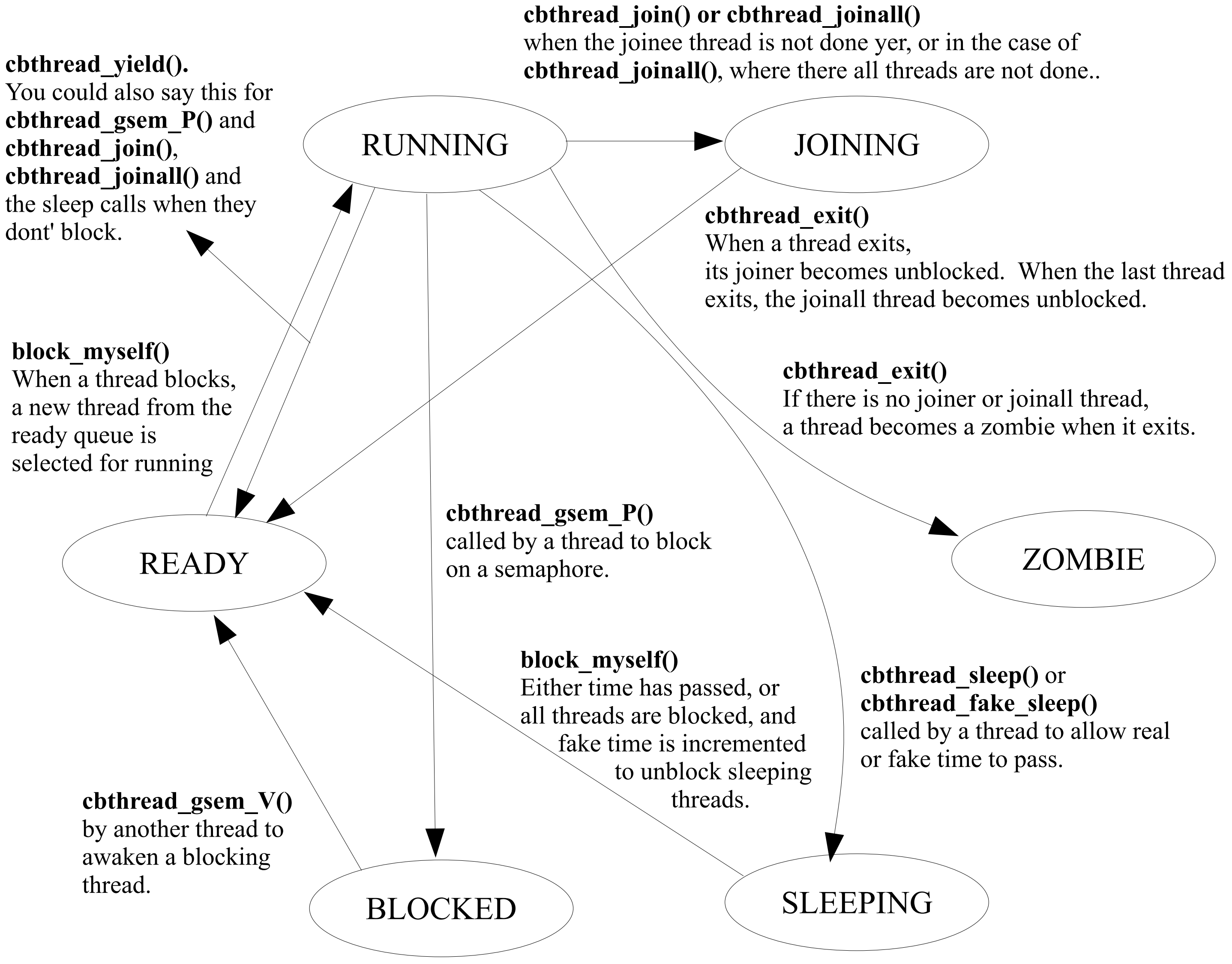
Answer for Question 1


RUNNING
JOINING
READY
ZOMBIE
BLOCKED
SLEEPING
cbthread_join() or cbthread_joinall() when the joinee thread is not done yer, or in the case of **cbthread_joinall()**, where there all threads are not done..
cbthread_yield(). You could also say this for **cbthread_gsem_P()** and **cbthread_join(), cbthread_joinall()** and the sleep calls when they dont' block.
cbthread_exit() When a thread exits, its joiner becomes unblocked. When the last thread exits, the joinall thread becomes unblocked.
block_myself() When a thread blocks, a new thread from the ready queue is selected for running
cbthread_exit() If there is no joiner or joinall thread, a thread becomes a zombie when it exits.
cbthread_gsem_P() called by a thread to block on a semaphore.
block_myself() Either time has passed, or all threads are blocked, and fake time is incremented to unblock sleeping threads.
cbthread_sleep() or **cbthread_fake_sleep()** called by a thread to allow real or fake time to pass.
cbthread_gsem_V() by another thread to awaken a blocking thread.

Grading

- READY->RUNNING: 1.5 points for arrow & explanation
- RUNNING->READY: 1 point for arrow & explanation
- RUNNING->JOINING: 2.5 points for array & explanation that includes both join and joinall
- RUNNING->ZOMBIE: 1.5 points for arrow & explanation
- RUNNING->SLEEPING: 2 points for arrow and explanation that includes both sleep and fakesleep
- JOINING->READY: 2.5 points for arrow and explanation that includes both join and joinall
- SLEEPING->READY: 2 points for arrow and explanation that includes both sleep and fakesleep

That's a total of 13 points.

Question 2

i_am_hungry_1(): This is the implementation that uses a global queue to prevent starvation. So, it's a good thing that it prevents starvation. However, it has two big problems. First, the global queue is inefficient, especially when the table is big, because philosophers that are far away from you can keep you from eating. This leads to large block times. Second, since we only check adjacent philosophers in i_am_sated(), our system can stall. See "Solution 6" from the dining philosophers lecture notes – this is the same solution.

i_am_hungry_2(): This is a solution that also prevents starvation. It works by checking adjacent philosophers, and if either adjacent philosopher has been hungry longer than you have, you can't eat. It is good in that it prevents starvation, but it blocks way too much because like i_am_hungry_1(), philosophers that are relatively far away from you can prevent you from eating. This effect is more pronounced for smaller tables. Basically, this solution is too aggressive at preventing starvation, and it leads to larger blocked times. To solve it, we loosen the restriction on adjacent philosophers to say that as long as an adjacent philosopher has not been waiting for some threshold time, you can eat.

i_am_hungry_3(): This is the book's solution – if both chopsticks are available, then grab them. It has good blocking time, but it can exhibit starvation in some cases. It also suffers from the fact that you have to allocate both chopsticks before you start picking one up, and if it takes time to pick up the chopsticks, then this solution can be improved by picking up one chopstick before the other chopstick has been put on the table.

Since the usage of the lock wasn't specified clearly, if you said that we could have race conditions or deadlock in these solutions, you received credit for it, as long as you gave a cogent explanation.

Grading

11 points, allocated as follows:

i_am_hungry_1(): uses central queue, prevents starvation, high block times/poor performance, system stalls.
i_am_hungry_2(): checks adjacent philosophers' hunger times, prevents starvation, poor performance.
i_am_hungry_3(): book's solution, good performance, can starve, anything else reasonable.

Question 3

This is a nuts-and-bolts fork/exec/pipe/dup program:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
  int p[2];
  int ifd;
  int ofd;
  int efd;
  int pid1, pid2;
  int dummy;

  pipe(p);
  pid1 = fork();

  if (pid1 == 0) {
    ifd = open("cin.txt", O_RDONLY);
    if (ifd < 0) { perror("cin.txt"); exit(1); }
    dup2(ifd, 0);
    close(ifd);

    efd = open("cerr.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (efd < 0) { perror("cerr.txt"); exit(1); }
    dup2(efd, 2);
    close(efd);

    dup2(p[1], 1);
    close(p[1]);
    close(p[0]);
    execlp("cat", "cat", NULL);
    perror("cat");
    exit(1);
  }

  pid2 = fork();

  if (pid2 == 0) {
    ofd = open("gout.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (ofd < 0) { perror("gout.txt"); exit(1); }
    dup2(ofd, 1);
    close(ofd);

    dup2(p[0], 0);
    close(p[1]);
    close(p[0]);
    execlp("grep", "grep", "xxx", NULL);
    perror("grep");
    exit(1);
  }

  close(p[0]);
  close(p[1]);
  wait(&dummy);
  wait(&dummy);
}
```

Grading: 13 points. Basically, you start with 13 points if your program has the correct overall structure.
If you don't have the right overall structure, you start with less – any solution with just one fork() call started with 9 points. After that, I deduct for mistakes, such as not calling pipe() in the right place, not closing file descriptors, stray wait() calls, etc.

Question 4

You need a red-black tree indexed on i. Each val field of these will be a struct that contains a condition variable and a number of blocked threads. If the number of blocked threads equals nthreads, then it's time to wake everyone up. Otherwise, block the thread. It's straightforward code, although you have to maintain a lock as well, to protect access to the data structures:

```
typedef struct {
  int nthreads;
  JRB t;
  pthread_mutex_t *lock;
} Barrier_Tree;

typedef struct {
  pthread_cond_t *cond;
  int i;
  int nblocked;
} Barrier_Main;

barrier barrier_create(int nthreads)
{
  Barrier_Tree *bt;

  bt = (Barrier_Tree *) malloc(sizeof(Barrier_Tree));
  bt->nthreads = nthreads;
  bt->lock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
  pthread_mutex_init(bt->lock, NULL);
  bt->t = make_jrb();
  return (void *) bt;
}

void barrier_block(barrier bar, int i)
{
  Barrier_Tree *bt;
  Barrier_Main *b;
  JRB tmp;

  bt = (Barrier_Tree *) bar;
  pthread_mutex_lock(bt->lock);
  tmp = jrb_find_int(bt->t, i);
  if (tmp == NULL) {
    b = (Barrier_Main *) malloc(sizeof(Barrier_Main));
    b->i = i;
    b->nblocked = 0;
    b->cond = (pthread_cond_t *) malloc(sizeof(pthread_cond_t));
    pthread_cond_init(b->cond, NULL);
    tmp = jrb_insert_int(bt->t, i, new_jval_v((void *) b));
  } else {
    b = (Barrier_Main *) tmp->val.v;
  }
  b->nblocked++;
  if (b->nblocked == bt->nthreads) {
    for (i = 0; i < bt->nthreads-1; i++) pthread_cond_signal(b->cond);
    free(b->cond);
    free(b);
    jrb_delete_node(tmp);
  } else {
    pthread_cond_wait(b->cond, bt->lock);
  }
  pthread_mutex_unlock(bt->lock);
}
```

Grading

14 points. I was lenient on syntax and heavy on structure. If you didn't deal with i well, your maximum was 10 points.