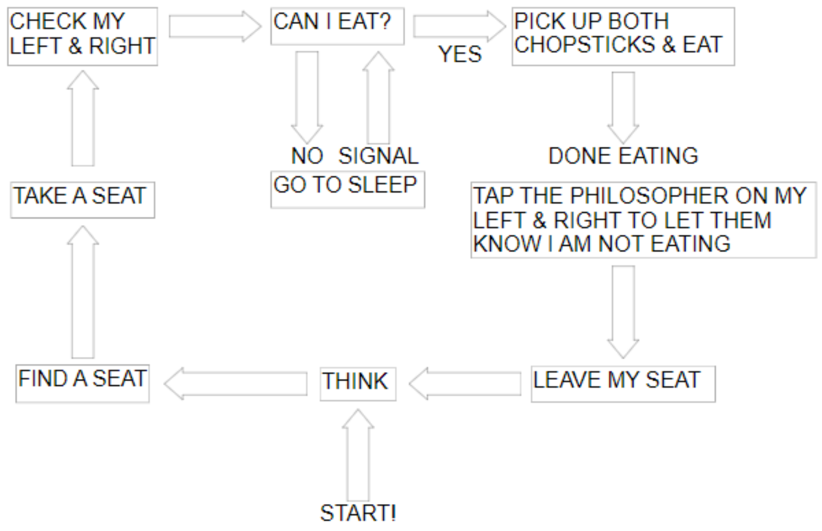
WALKING PHILOSOPHERS USING MONITORS

BY MUHAMMAD SARIM MEHDI

REQUIREMENT:

You have 10 philosophers in a room but only 5 seats. So, at a time, only 5 philosophers can take a seat and **eat**. On the other hand, the other 5 philosophers will simply have to wait for their turn. Whenever a philosopher decides to **think**, he will vacate the seat and do the whole thinking process while walking around the room. While seated, the philosophers will follow the classic mode of picking up chopsticks i.e. they will eat only when the philosophers seated to the left and right of them on the round table are not eating. Hence, the Tanenbaum solution is also applicable here.

EXECUTION FLOW FOR A SINGLE PHILOSOPHER:



MY DESIGN METHODOLOGY:

I have decided to summarize my design in the following steps:

-> ***There are 10 philosophers in the room***. My requirement is concurrency and no starvation. Concurrency is easily applied because I use Tanenbaum's solution for the philosophers who are seated. But how do I make sure there is no starvation?

-> ***No starvation means each philosopher waits an equal amount of time before eating***. However, in the beginning, my major obstacle was overcoming deadlock. I decided to introduce another independent entity to take care of deadlock and starvation at the same time. I call this independent entity my waiter.

-> ***<u>Now, let's say my 10 philosophers are thinking</u>***. One of them stops thinking and gets hungry. The first step is to take a seat. For this, I have introduced a new function called assignSeat which assigns a seat to a philosopher and returns the seat number that was assigned to the philosopher. Assigning a seat is very simple. The philosopher checks the availability of each seat. For convenience I assign states to a single seat. A seat can be THINKING, HUNGRY OR EATING. A THINKING seat means it is vacant (I use different states for philosophers now which I will explain later). So, this is the code that allows a philosopher to find an empty seat:

```
while (state[i]==NOT_ASSIGNED) {
      for (j=0;j<N_SEATS;j++) {
         if (seats[j]==THINKING) {
            if (!IHaveEaten[i]) {
               leaving++;
               seats[j]=HUNGRY;
               state[i]=ASSIGNED;
               theSeatAssigned[i] = j;
               thePhilosopherAssigned[j] = i;
               j=N_SEATS;
            }
         }
      }
      if (state[i]==NOT_ASSIGNED) {
         IHaveEaten[i]=false;
         arrival++;
         //printf("%d: I didn't get a seat, so going to sleep\n",i);
         pthread_cond_wait(&self[i], &om);
      }
   }
   pthread_mutex_unlock(&om);
   return theSeatAssigned[i];
}
```

So, a philosopher enters this loop because in the beginning each philosopher has the state NOT_ASSIGNED. The philosopher goes through each seat. Whenever it finds a seat, it will immediately leave the loop and proceed with dining. If you look at this code now, a lot of the variables won't make sense. So, for your own clarity, just interpret it as a simple for loop going through each seat while paying no attention to other variables like IHaveEaten and so on.

-> ***<u>Now, in the code, the seat selection is guarded by a mutex called om</u>***. So, only one philosopher can choose a seat at the time and this process can't happen concurrently. Now, let's say the first 5 philosophers find a seat (you can also notice that philosophers are assigned seat starting from beginning to end. Meaning, the first philosopher will always be assigned seat 0 and the 5th philosopher will always be assigned seat 4th) and the next 5 want a seat. There is no seat available, so what will they do now. To deal with this, I simply put them to sleep on a condition variable:

```
if (state[i]==NOT_ASSIGNED) {
      IHaveEaten[i]=false;
```

```
        arrival++;
        //printf("%d: I didn't get a seat, so going to sleep\n",i);
        pthread_cond_wait(&self[i], &om);
    }
```

Now, you must be wondering what is arrival? Don't worry, I will explain this later on. For now, all you need to consider is that 5 philosophers have taken a seat and the other 5 are just sleeping. Also, notice that I didn't use while and used if. This is because, I want my philosopher to go through the previous while loop for seat selection when it wakes up. Don't worry, everything will make sense as I explain further.

-> ***<u>Now, back to the philosophers who are seated at the table</u>***. You are wondering what they are doing right now? Are they eating? Well, this is what they are doing:

```
pthread_mutex_lock(&mutex);
  //printf("%d: I am in my seat, going to sleep\n",thePhilosopherAssigned[i]);
  if (!nowStop) {
     pthread_cond_wait(&notAtTable[i], &mutex);
  }
```

That's right! They are sleeping. They are sleeping while seated at the table and also notice that they are sleeping on another condition variable. For now, ignore the nowStop boolean and the if condition (I will explain it later on). Just imagine the code as a mutex lock and then a conditional wait.

-> ***<u>Now, you must be wondering what is going on?</u>*** Here, I will introduce my waiter which I run on another thread in the main.c file. This is what the waiter will do:

```
  if (arrival >= N_THREADS - seatsOccupied) {
     //printf("5 philosophers who didn't get a seat are now asleep\n");
     pthread_mutex_lock(&om);
     int i;
     for(i=0;i<N_THREADS;i++) {
        pthread_cond_signal(&notAtTable[i]);
     }
     arrival = 0;
     pthread_mutex_unlock(&om);
  }
```

Now, I will explain to you what is the purpose of arrival. What I do is that I don't want the 5 philosophers seated to immediately start eating. Instead, I want them to wait until the other 5 (N_THREADS – seatsOccupied, I am using this because **<u>my code is generic for any number of seats and philosophers</u>** and also I will later on explain what is seatsOccupied and why I am using it instead of simply N_SEATS) have gone to sleep on their condition variable. Why am I doing this? To make sure there is no starvation? How? You will soon see.

-> ***<u>So, 5 philosophers take a seat at the table and fall asleep</u>***. The other 5 check for available seats and fall asleep while standing because none are available. Once the waiter sees that the remaining philosophers have fallen asleep, he wakes up the philosophers seating at the table to let them know they can start eating.

-> ***<u>Now, for the seated philosopher</u>***, I change the state of the seats to THINKING, HUNGRY or EATING and not that of philosophers which is odd because a seat cannot realistically eat! But for the sake of design I decided to go with this. You can think of it like this: If the state of a seat is THINKING, it means that it is vacant. You must have noticed in the assignSeat function I check for available seats by checking whether the state of a seat is THINKING or not. If the state of a seat is HUNGRY, it means the philosopher on that seat is HUNGRY (however, the philosopher on that seat will still have the same state i.e. ASSIGNED). If the state of a seat is EATING, it means the philosopher on that seat is EATING. It feels confusing at first but once I start explaining the rest of the code, everything will fall into place. So, in the test function:

```
void test(int i) {
  int philo = thePhilosopherAssigned[i];
  if(seats[i]==HUNGRY && seats[LEFT]!=EATING && seats[RIGHT]!=EATING) {
    if (philo!=-1) {
      seats[i]=EATING;
      pthread_cond_signal(&mySeat[i]);
    }
  }
}
```

And in the void *philo() thread, I get my seat number by i=assignSeat(philo) and then I call pickup(i) and putdown(i) later on instead of pickup(philo) and putdown(philo) as is expected when you just have dining philosophers and not walking philosophers: (just ignore everything else and focus on the bold sentences)

```
void *philo(void *arg) {
  int philo=atoi((char*)arg);
  int n=EXECUTION_LENGTH, i;
  while(n--) {
    if (FOREVER) { n++; }
  do_something(philo,"THINKING", SHORT_TIME);
  pthread_mutex_lock(&myMutex);
    tabs(philo);
    printf("%d: HUNGRY!!\n", philo);
  pthread_mutex_unlock(&myMutex);
  i = assignSeat(philo);
  pickup(i);
  do_something(philo,"EATING", LONG_TIME);
  putdown(i);
  if (n!=0) {
    GoToSleep(philo);
  }
  else {
    WakeupBeforeLeaving();
```

```
   }
}
   return NULL;
}
```

-> ***<u>So, the 5 philosophers are finally done eating</u>***. Now what? Now, I put them to sleep AGAIN! Why am I doing this? Because I don't want deadlock and also because I don't want the philosophers who are leaving the seat now to have a chance to take a seat again. In this way, only the philosophers who didn't get a seat previously (who were sleeping while the rest were dining) will get a chance to take a seat. **<u>This is my idea of preventing starvation</u>**. I let 5 eat first and then I put them to sleep once they are done eating and I let the other 5 eat first and then they are put to sleep as well and the previous 5 wake up and are allowed to dine and this cycle repeats and you never have starvation because there is always fairness and each philosopher waits for his turn at the table an equal amount of time.

-> ***<u>But how do I do this?</u>*** Simple:

```
void GoToSleep(int i) {
   pthread_mutex_lock(&waiterMut);
   printf("%d: I have vacated my seat and now going to sleep\n",i);
   IHaveEaten[i] = true;
   nextOne++;
   if (!nowStop) {
      pthread_cond_wait(&leavingTable[i], &waiterMut);
   }
   pthread_mutex_unlock(&waiterMut);
}
```

Once again, ignore nowStop and just imagine this code goes like this: mutex lock and go to sleep (releasing the lock). Also, notice that now I am using a different condition to put the philosophers to sleep than before (before it was notAtTable). I am doing this because I want to differentiate between philosophers going to sleep while SEATED at the table and when they leave the table right after EATING. Also, notice that I am setting a boolean IHaveEaten to true for each philosopher.

This is very important as you will later on see when I will try to explain how my code works even for non-symmetric design (meaning that the number of seats is not equal to half the number of philosophers). For this design of 10 philosophers and 5 seats, if I remove IHaveEaten then my code would go on as expected but I want to make my code generic for any number of philosophers and seats (except 0 of course or negative values). Basically, if you recall the previous while loop where a philosopher looks for a seat, you must've noticed this boolean being check before a seat is assigned. If IHaveEaten is true then the seat will not be assigned to the philosopher even though it is available! Why am I doing this? Because in my design the next group of philosophers will not eat until the previous group of philosophers is done eating regardless of the availability of seats.

I didn't want to get into so much detail of this but I will give you a basic idea of why I am using this (and also why I am using seatsOccupied). Let's say I have 10 philosophers and 7 seats. This is a non-symmetric design as I explained previously. It is a non-symmetric design because after every iteration the number of philosophers seated at the table at one time won't be the same as were before. So, with 10 philosophers and 5 seats, every time you will have 5 philosophers seated at the table at once. But, with 10 philosophers and 7 seats, I will have 7 philosophers take a seat at one time but then the next time I will only let 3 philosophers (the ones who were unable to take a seat before) take a seat and I will put the other 7 (who already got their turn before) to sleep because otherwise they will get more than their fair share and this might lead to starvation.

SeatsOccupied is initialized to N_SEATS and is very important to tell the waiter when to wake up philosophers from their sleep. How does it work? Let's say I have 10 philosophers. 5 of them take a seat and the other 5 can't take a seat. So, they go to sleep as explained here:

```
if (state[i]==NOT_ASSIGNED) {
        IHaveEaten[i]=false;
        arrival++;
        //printf("%d: I didn't get a seat, so going to sleep\n",i);
        pthread_cond_wait(&self[i], &om);
    }
```

Now, the ones who took a seat don't immediately start eating but they go to sleep and are awoken by the waiter only when the remaining 5 (who didn't get a seat) all go to sleep. This is sort of like using barriers where you wait for all threads to arrive before proceeding. So, imagine this in a similar context. 5 philosophers who take a seat will wait at a barrier for the remaining 5 philosophers to go to sleep (meanwhile, they themselves are sleeping at the barrier too). How will the waiter know that the remaining 5 philosophers who didn't get a seat are all asleep? Easy, like this:

```
if (arrival >= N_THREADS - seatsOccupied) {
    //printf("5 philosophers who didn't get a seat are now asleep\n");
    pthread_mutex_lock(&om);
    int i;
    for(i=0;i<N_THREADS;i++) {
      pthread_cond_signal(&notAtTable[i]);
    }
    arrival = 0;
    pthread_mutex_unlock(&om);
  }
```

Now, if seatsOccupied is initialized to the number of seats then that means it's value in the beginning is 5. N_THREADS (which is 5) when subtracted to seatsOccupied gives 5 which is the exact number my waiter is looking for before deciding to wakeup the philosophers who are sleeping on their seats. Just think of the waiter as an observer who is standing there waiting for the remaining 5 philosophers to go to sleep and once that happens he goes around the dining table and wakes up all the sleeping philosopher. Also, notice that when the philosophers wake up they won't be able to start eating immediately because the mutex they want to regain is still held by the waiter who will only release it once he has woken up all the seated philosophers.

This is one important design choice I made (I could've used any other mutex there) to prevent deadlock (every little detail, especially the mutex, counts. In my previous attempts, I always ran into problems where a certain mutex would be held by one philosopher and I would start splitting my hairs trying to find which one was it. So, I just decided to use appropriately common mutexes to minimize deadlock as much as possible and, in this case, it is non-existent!).

-> ***Now, in the previous section***, you must be thinking what is nextOne? Well, nextOne has the same purpose as arrival. As in, it tells the waiter that 5 philosophers have gone to sleep after dining. So, what does the waiter do now? Well:

```
 if (nextOne >= seatsOccupied) {
      //printf("5 philosophers who have vacated their seat are now asleep\n");
      pthread_mutex_lock(&waiterMut);
      seatsOccupied = N_THREADS - seatsOccupied;
      int k;
      for(k=0;k<N_THREADS;k++) {
         pthread_cond_signal(&self[k]);
      }
      nextOne = 0;
      pthread_mutex_unlock(&waiterMut);
   }
```

So, as you can clearly see, our waiter is waking up the philosophers who were unable to find a seat but why am I changing the value of seatsOccupied? Again, you won't understand it's importance here where we are dealing with a symmetric situation. In a non-symmetric situation this is very important. So, I will try my best to explain to you why I am doing this. Again, recall the example of 10 philosophers and 7 seats. Think of it like this. When I initialize my seatsOccupied to 7 I am basically telling the philosophers that only 7 will be allowed to take a seat which makes sense because you have 7 seats. Then after the first group of 7 philosophers have eaten and they get up and want to have another go, I change my seatsOccupied to 3 (N_THREADS, which is 10, when subtracted to seatsOccupied, which was 7, gives me 3) and tell the philosophers that now only 3 an take a seat which is absurd but, once again, this is my design methodology (or interpretation to be exact) of preventing starvation. So, when I wake up the 3 philosophers who didn't get a seat before, they will take a seat and this whole cycle will repeat but wait a second! Something is not right!

-> ***The philosophers who ate before vacated their seats and are still sleeping!*** So, how do I wake them up? Now, I will revisit the very first segment of code I showed here and explain to you the importance of another variable I was using there:

```
int assignSeat(int i) {
   if (N_SEATS >= N_THREADS) { nowStop = true; }
   pthread_mutex_lock(&om);
   int j;
   while (state[i]==NOT_ASSIGNED) {
      for (j=0;j<N_SEATS;j++) {
         if (seats[j]==THINKING) {
            if (!IHaveEaten[i]) {
               leaving++;
               seats[j]=HUNGRY;
               state[i]=ASSIGNED;
```

```
                theSeatAssigned[i] = j;
                thePhilosopherAssigned[j] = i;
                j=N_SEATS;
              }
            }
          }
          if (state[i]==NOT_ASSIGNED) {
            IHaveEaten[i]=false;
            arrival++;
            //printf("%d: I didn't get a seat, so going to sleep\n",i);
            pthread_cond_wait(&self[i], &om);
          }
        }
        pthread_mutex_unlock(&om);
        return theSeatAssigned[i];
      }
```

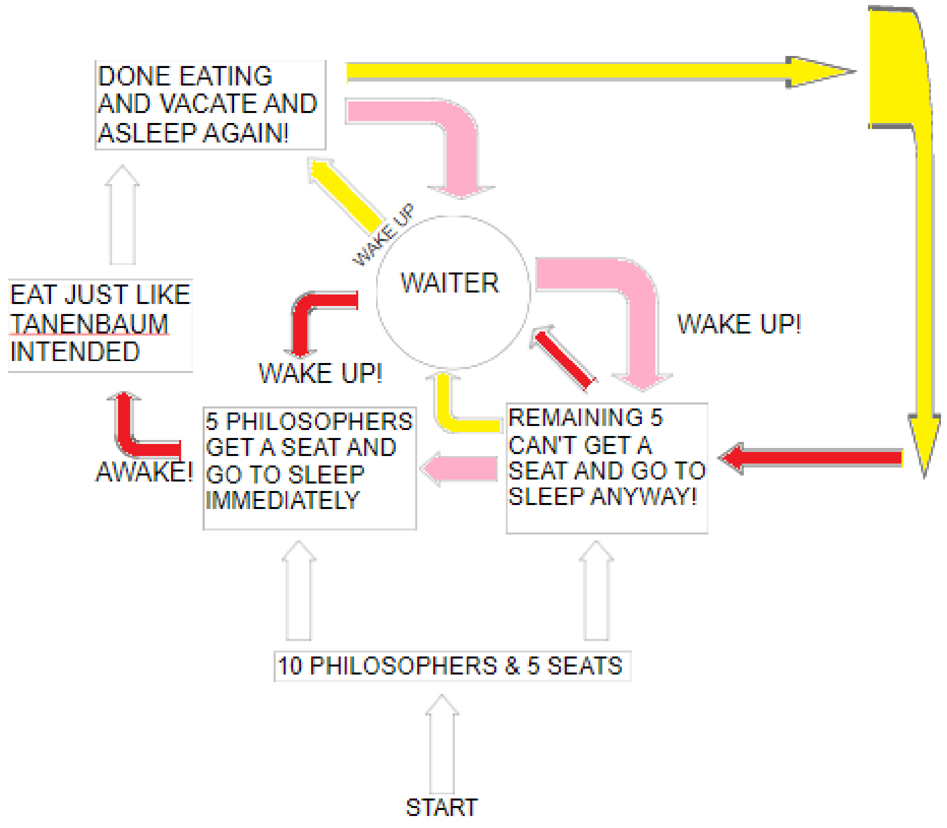
Notice the variable in bold letters. It is called leaving. I previously told you to ignore it but now, with so many other things explained, I believe now will be the right time to explain its importance to you. Basically, our waiter will check this value in its own function before deciding to wake up the philosophers who went to sleep after eating. But you must be wondering what happens when this value is updated at the very beginning of the code when you don't have philosophers who go to sleep after eating (because no one has eaten yet!). So, what happens then? Nothing! Because, as we already know, if you signal a condition on which no one is waiting then absolutely nothing happens. This is the convenience of using condition variables. So, here is how our waiter wakes up the philosophers who, after eating and vacating their seats, went to sleep:

```
  if (leaving >= seatsOccupied) {
    //printf("5 philosophers who have vacated their seat are now asleep\n");
    pthread_mutex_lock(&waiterMut);
    int j;
    for(j=0;j<N_THREADS;j++) {
      pthread_cond_signal(&leavingTable[j]);
    }
    leaving = 0;
    pthread_mutex_unlock(&waiterMut);
  }
```

Now you will notice that I am comparing my value of leaving with the value of seatsOccupied. In the beginning, seatsOccupied is equal to the number of seats I want to assign (which is 5 or 7 depending on which case you consider. In fact, you know what, I will say it was 7 because, in my opinion, explaining my code under the context of a non-symmetric case will shed light on the importance of several variables I am using that would otherwise seem redundant if I were to explain using a symmetric case). So, 7 philosophers chose a seat and increment the value of leaving while they are at it. When leaving becomes 7, the waiter signals all those who are waiting on the leavingTable condition but no one is waiting so absolutely nothing happens.

Now, when the second group of philosophers decides to take a seat, the value of leaving is only incremented to 3 (this is because the other 7 won't be able to take a seat as I set their IHaveEaten to true to prevent them from taking a seat even if it is available, I hope I don't explain again why I am doing this! Also, before the philosophers, who don't get a seat, go to sleep they set their IHaveEaten to false so next time they are bound to get their turn) and, as I explained previously, I changed my value for seatsOccupied (again, remember that seatsOccupied is basically the number of seats I want to have occupied) to 3 by using **seatsOccupied = N_THREADS – seatsOccupied**. So, with the next batch of philosophers having successfully taken their seat, the waiter will now signal to the previous philosophers who went to sleep after eating and they will, unsuccessfully, try to take a seat and then go to sleep because they were unable to get a seat and you have the same cycle repeating.

<u>UPDATED EXECUTION FLOW: (JUST REMEMBER: RED TO PINK TO YELLOW AND REPEAT!)</u>


DONE EATING AND VACATE AND ASLEEP AGAIN!
WAKE UP
WAITER
EAT JUST LIKE TANENBAUM INTENDED
WAKE UP!
WAKE UP!
AWAKE!
5 PHILOSOPHERS GET A SEAT AND GO TO SLEEP IMMEDIATELY
REMAINING 5 CAN'T GET A SEAT AND GO TO SLEEP ANYWAY!
10 PHILOSOPHERS & 5 SEATS
START

<u>SOME FINAL REMARKS:</u>

-> ***<u>What if I have more seats than philosophers?</u>*** Well, this is why I introduced nowStop. Basically, if you have more seats than philosophers then you don't need a waiter! As a result, they won't go to sleep after taking a seat or after eating because no other philosophers are waiting to take their seat. So, for example:

```
int assignSeat(int i) {
  if (N_SEATS >= N_THREADS) { nowStop = true; }
  pthread_mutex_lock(&om);
  int j;
  while (state[i]==NOT_ASSIGNED) {
    for (j=0;j<N_SEATS;j++) {
            if (seats[j]==THINKING) {
                if (!IHaveEaten[i]) {
                    leaving++;
                    seats[j]=HUNGRY;
                    state[i]=ASSIGNED;
                    theSeatAssigned[i] = j;
                    thePhilosopherAssigned[j] = i;
                    j=N_SEATS;
                }
            }
        }
        if (state[i]==NOT_ASSIGNED) {
            IHaveEaten[i]=false;
            arrival++;
            //printf("%d: I didn't get a seat, so going to sleep\n",i);
            pthread_cond_wait(&self[i], &om);
        }
    }
    pthread_mutex_unlock(&om);
    return theSeatAssigned[i];
}
```

The bold section of the code is where I set nowStop after checking whether I have more seats than philosophers. Then:

```
bool waiter() {
    if (arrival >= N_THREADS - seatsOccupied) {
        //printf("5 philosophers who didn't get a seat are now asleep\n");
        pthread_mutex_lock(&om);
        int i;
        for(i=0;i<N_THREADS;i++) {
            pthread_cond_signal(&notAtTable[i]);
        }
        arrival = 0;
        pthread_mutex_unlock(&om);
    }

    if (nextOne >= seatsOccupied) {
        //printf("5 philosophers who have vacated their seat are now asleep\n");
        pthread_mutex_lock(&waiterMut);
        seatsOccupied = N_THREADS - seatsOccupied;
        int k;
        for(k=0;k<N_THREADS;k++) {
            pthread_cond_signal(&self[k]);
        }
        nextOne = 0;
        pthread_mutex_unlock(&waiterMut);
    }
    if (leaving >= seatsOccupied) {
        //printf("5 philosophers who have vacated their seat are now asleep\n");
        pthread_mutex_lock(&waiterMut);
        int j;
        for(j=0;j<N_THREADS;j++) {
            pthread_cond_signal(&leavingTable[j]);
        }
        leaving = 0;
        pthread_mutex_unlock(&waiterMut);
    }

    if (nowStop) {
        return true;
    }
    else {
        return false;
    }
}
```

Again, only notice the bold section. If nowStop is true then my waiter function will return true signaling to the waiter thread in main.c to come out of its infinite loop:

```
void *wait(void *arg) {
    while (!x) {
        x = waiter();
    }
    return NULL;
}
```

-> ***<u>If my philosophers are set to eat only 10 times then how will the philosophers who have already eaten 10 times signal the philosophers who are about to eat their last meal?</u>*** To deal with a finite loop (no such exception handling needs to be done in an infinite loop) I just tell the philosophers to signal the ones sleeping before them:

```
void WakeupBeforeLeaving() {
    pthread_mutex_lock(&waiterMut);
    nextOne++;
    if (nextOne >= N_SEATS) { nowStop = true; }
    pthread_mutex_unlock(&waiterMut);
}
```

Once again notice the bold section. So now, instead of falling asleep after eating (which makes no sense because it was their last meal and they will not be eating again. So, they should just leave!), they just update nextOne and the waiter wakes up the previous philosophers. Also, notice that I am making nowStop to true. This is because, once the waiter wakes up the last remaining philosophers for their last meal, he will have no use and will simply exit from the thread on which he was executing. But our remaining philosophers will go to sleep when they get a seat at the table, so who will wake them up now. That's easy, they won't go to sleep now! Look at this: (You probably saw this before and it didn't make sense, but now, hopefully, it does)

```
void pickup(int i) {
   pthread_mutex_lock(&mutex);
   //printf("%d: I am in my seat, going to sleep\n",thePhilosopherAssigned[i]);
   if (!nowStop) {
      pthread_cond_wait(&notAtTable[i], &mutex);
   }
   test(i);
   while(seats[i]==HUNGRY) {
      pthread_cond_wait(&mySeat[i], &mutex);
   }
   pthread_mutex_unlock(&mutex);
}
```

Look at the bold section. The philosopher who gets a seat will only go to sleep if nowStop is false but we set nowStop to true so the philosopher won't go there and will, instead, go right ahead eating.

-> ***What the hell is this?*** (In bold)

```
void test(int i) {
   int philo = thePhilosopherAssigned[i];
   if(seats[i]==HUNGRY && seats[LEFT]!=EATING && seats[RIGHT]!=EATING) {
      if (philo!=-1) {
         seats[i]=EATING;
         pthread_cond_signal(&mySeat[i]);
      }
   }
}
```

You must be thinking why I am checking whether there is a philosopher on a seat before assigning the state of a seat to EATING. Well, the answer is right there! If I set an empty seat to EATING then that is absurd! So, this is just a precaution!

HOW TO USE THIS CODE:
You can play around with the values I have defined in the beginning of my header file and main.c. I have mentioned in the comments what you can change. Have fun!