Mutex_t and cond_t are struct type. The type of waiting queue is circular queue so that both of them have start and end point to check which position each process/thread should be allocated in. Queue size(qsize) variable is also used to check the size of queue.

1. **Mutex init**

   Mutex_init is used to initialize mutex. All the variables being contained in mutex_t structure is initialized in this function.

```
11
12 //  Initialize mutex
13 int mutex_init(struct mutex_t *mutex){
14
15   if(mutex == 0) return -1; // return-1 : invalid
16   if(mutex->valid !=0) return -2; // return-2 : attempting to reinitialize
17
18   //mutex->lock = malloc(sizeof(*mutex->lock));
19   initlock(&mutex->lock,"mutex");
20   acquire(&mutex->lock);
21
22   mutex->current = 0;
23   mutex->valid=1;
24   mutex->qsize=0;
25   mutex->start =0;
26   mutex->end=0;
27
28   release(&mutex->lock);
29   return 0;  //return 0 : success
30 }
31
```

2. **Mutex_lock**

   First of all, we have to check some awkward situation before we get the mutex_lock. Those exceptions are written in the guide line. The next step is, when you finally pass all the exception, we should put the process into the queue to make it sleep. It is because another process already has mutex lock so that curproc has to wait(sleep) while mutex lock is released.

```
33
34 //  assign lock
35 int mutex_lock(struct mutex_t *mutex){
36   struct proc *curproc = myproc();
37
38   //return-1 : mutex invalid
39   if(mutex == 0){
40     release(&mutex->lock);
41     return -1;
42   }
43
44   acquire(&mutex->lock);
45
46   //return-2 : mutex not initialized
47   if(mutex->valid ==0){
48     release(&mutex->lock);
49     return -2;
50   }
51
52   //return-3 : already has the mutex
53   if(mutex->current ==curproc){
54     release(&mutex->lock);
55     return -3;
56   }
57
58   //no one has the mutex
59   if(mutex->current==0 && mutex->qsize==0){
60     mutex->current = curproc;
61     release(&mutex->lock);
62     return 0;
63   }
64
65   //someone has the mutex
66   if(mutex->qsize<NTHREAD-1){
67     mutex->queue[mutex->end]=curproc; //enqueue
68     mutex->qsize++; //increase q size
69     mutex->end=(mutex->end+1) % (NTHREAD-1);
70
71     while(mutex->current!= curproc){ //내가 lock을 가지고 있지 않을 때 sleep
72        sleep(curproc, &mutex->lock);
73     }
74   }
75   else {
76     cprintf("mutex queue full(%d).\n", curproc->tid);
77   }
                                                                    52,1
```

3. Mutex_unlock

When the mutex lock is released, since we have to execute in order according to the queue position, we dequeue the first one from the mutex queue and change its state to runnable(wakeup).

```
//preassumption: someone already has the lock
int mutex_unlock(struct mutex_t *mutex){
  struct proc *curproc = myproc();
  //return-1 : mutex invalid
  if(mutex == 0){
    release(&mutex->lock);
    return -1;
  }


  //return-2 : mutex not initialized
  if(mutex->valid ==0) return -2;


  acquire(&mutex->lock);
  //return-3 : doesn't have the mutex
  if(mutex->current !=curproc){
    release(&mutex->lock);
    return -3;
  }

  //how to unlock?!
  if(mutex->qsize ==0){
    mutex->current=0;
    release(&mutex->lock);
    return 0;
  }

  mutex->current = mutex->queue[mutex->start]; //current(at the start point) ha
ock
  mutex->queue[mutex->start] =0; //dequeue
  mutex->start = (mutex->start+1) % (NTHREAD-1);
  mutex->qsize--; //decrease qsize

    //cprintf("pp22\n");
  wakeup(mutex->current);

  release(&mutex->lock);
  return 0;
}
```

4. Cond_init

Conditional variable is initialized in this function. This function is very similar with the mutex_init.

```c
//  Initialize CV
int cond_init(struct cond_t *cond){
  if(cond == 0) return -1; // return-1 : invalid
  if(cond->active !=0) return -2; // return-2 : attempting to reinitialize

 // cond->lock = malloc(sizeof(*cond->lock));
  initlock(&cond->lock,"Condition Variable");
  acquire(&cond->lock);

  cond->active=1; //initialized
  cond->qsize=0;
  cond->start=0;
  cond->end=0;

  release(&cond->lock);
  return 0;  // return 0 : success
}
```

5. **Cond_wait**

This function is used when a specific condition is not qualified. Process has to wait in the condition queue and goes to sleep state. When the state is being changed, the process has to release mutex lock first not to face the deadlock. If it receives signal to wake up, it takes mutex lock again.

```
145 //Lock CV
146 int cond_wait(struct cond_t *cond, struct mutex_t *mutex){
147   struct proc *curproc = myproc();
148   int count=0;
149   //return-1 : mutex invalid
150   if(mutex == 0) return -1;
151
152   //return-2 : mutex initialized
153   if(mutex->valid ==0) return -2;
154
155   //return-3 : no mutex
156   if(mutex->current !=curproc) return -3;
157
158
159   //return-1 : cv invalid
160   if(cond == 0) return -1;
161
162   //return-2 : cv initialized
163   if(cond->active ==0) return -2;
164
165
166   acquire(&cond->lock);
167   if(cond->qsize<NTHREAD-1){
168     cond->queue[cond->end]=curproc; //enqueue
169     cond->end=(cond->end+1) % (NTHREAD-1); //move endpoint
170     cond->qsize++; //increase q size
171
172     mutex_unlock(mutex);
173     while(1){
174
175      sleep(curproc, &cond->lock);
176
177      count=0;
178        for(int i=0;i<NTHREAD-1;i++){
179          if(cond->queue[i] == curproc) count++;
180        }
181      if(count==0) break;
182
183     }
184     release(&cond->lock);
185     mutex_lock(mutex);
186   }
187
188   else {
189     cprintf("cond queue full(%d).\n", curproc->tid);
190     release(&cond->lock);
191   }
                                                          157,2
```

6. Cond_signal

When the condition is qualified, this function sends signal to the sleeping process, which is in the cond waiting queue.

HW3 20143144433 이영숙 운영체제(synch)

```
//unblock a thread
int cond_signal(struct cond_t *cond){
  struct proc *next;

  //return-1 : cv invalid
  if(cond == 0){
    release(&cond->lock);
    return -1;
  }

  acquire(&cond->lock);
  //return-2 : cv initialized
  if(cond->active ==0){
    release(&cond->lock);
    return -2;
  }


  if(cond->qsize ==0){
    release(&cond->lock);
    return 0;
  }

  if(cond->qsize!=0){
    next = cond->queue[cond->start];
    cond->queue[cond->start] =0; //dequeue
    cond->qsize--; //decrease qsize
    cond->start = (cond->start+1) % (NTHREAD-1);

    wakeup(next);
  }


  release(&cond->lock);
  return 0;
}
```