OS Assignment 5
Report

Group number : 15
Group members : Apurv Kumar 14CS10006
Shubham Sharma 14CS30034

Scheduler :

The files changed are thread.c and thread.h

For the default scheduler, there is only one ready queue, firstly, in thread_init, the current running program is converted to thread by initialising the thread structure for the current thread, this is the initial thread and is allocated a unique tid by which it is identified.
Now for new threads, the thread_create function is called, which first waits for the initialisation of the initial thread using the idle_thread semaphore. Now after the initialisation of the new thread, it is added for the ready queue.
Each running thread is given a time slice to run, on completion of their allotted time, an external clock interrupt is fired from the kernel which calls the function thread_ticks which decides whether the thread should be blocked or not, if the no of it running ticks is more than the set number of ticks, it is made to stop and a scheduler is called, but first threas_yield function is called to convert the thread from running state to ready state and the thread data structure is added to the ready queue.
The scheduler calls the next thread_to_run function which select the next thread depending on their time of addition to the ready queue and the priority.
It then calls the scheduler_tail function which destroys the thread if its job is finished and frees the data structure.

Multilevel feedback queue works almost in the same way, except for the fact that it has 2 levels of feedback queues, and the threads jump from one ready queue to another to ensure that each thread gets sufficient CPU time to run and they don't have to wait for long amount of clock ticks to run.

The specifications are as follows :
i. Threads are added on creation to Level 1 queue.
ii. Level 2 queue threads are scheduled only if Level 1 queue is empty.
iii. Level 1 scheduling policy is round-robin with time quanta equal to T. However, if a thread runs for two time quanta and still does not finish, it is pushed down to the end of the Level 2 queue.
iv. Level 2 scheduling is round-robin with time quanta equal to 2T. However, if a thread in Level 2 waits for 6T and still does not get the CPU (because Level 1 queue is not empty), it is pushed up to the end of the Level 1 queue.

The 6T count in Level 2 queue is the waiting time in ready queue WHEN THE LEVEL 1 QUEUE IS NOT EMPTY. If the Level 1 queue is empty, waiting time does not count because that is between Level 2 processes only. So think of it this way: When a process goes to Level 2 queue for the first time, its time count of 6T starts. If Level 1 queue becomes empty, the time count stops where it is. It starts again when there is another process in Level 1 queue. If the process gets scheduled in Level 2 queue, the timer gets reset to 0 (the 6T period starts again, after it finishes its time quanta) and the same above rule applies.
Note that a running process, be it in Level 1 or Level 2 queues, can block while running. In that case, you should remember which queue it was in, and when it is unblocked, it should be put back in the same queue.

Now what we have changed to make it a multilevel feedback scheduler is as follows : Basically we have changed thread_ticks and thread_yield according to the question in the problem statement. We have prepared two ready queues, instead of one, and modified some of the structures however all changes are in thread.c and thread.h files only.

For scheduler following things have been changed :
1.

```
int level;
 int ticks;
 int level_2_wait;
```

Added to the thread structure

2.instead of the default static ready_list we now have two ready_lists as follows :

```
static struct list ready_list_1;
static struct list ready_list_2;
```

also , the following flag variable :

```
static bool lvl1_fin=false;
static bool lvl1_com=false;
static bool lvl2_fin=false;
Static int x=0;
```

3.default ready list is renamed to ready_list_1

4.thread_ticks changed :

```
 thread_ticks++;
 (t->ticks)++;


 if(t->level == 1)
 {
  if(t->ticks >= 2 * TIME_SLICE)
    {
      lvl1_fin=false;
      lvl1_com=true;
      lvl2_fin=false;
      x=1;
      intr_yield_on_return ();
    }
  else if(thread_ticks >= TIME_SLICE)
    {
      lvl1_fin=true;
      lvl1_com=false;
      lvl2_fin=false;
      x=2;
      intr_yield_on_return ();
    }
 }
 else
 {
  if(thread_ticks >= 2 * TIME_SLICE)
    {
      lvl2_fin=true;
      lvl1_fin=false;
      lvl1_com=false;
      x=3;
      intr_yield_on_return ();
```

```
    }
  }
```

4.thread init changed, following lines added :

```
  initial_thread->level = 1;
  initial_thread->ticks = 0;
  initial_thread->level_2_wait = 0;
```

5.thread_yield changed :

```
 struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
if(!list_empty(&ready_list_1))
  {
          if(!list_empty(&ready_list_2))
          {
          struct list_elem *e,*tmp=NULL;
          for (e = list_begin (&ready_list_2); e != list_end (&ready_list_2);
                  e = list_next (e))
                  {
                  struct thread *t=list_entry(e,struct thread,elem);
                  // if(t->status == THREAD_READY){
                          if(tmp!=NULL)
                                  {
                                          struct thread *f=list_entry(tmp,struct thread,elem);
                                          list_remove(tmp);
                                          list_push_back(&ready_list_1,tmp);
                                          f->level_2_wait = 0;
                                          printf("%d: thread %d goes from L2 queue to L1
queue\n",clk,f->tid);
                                  }
                           ++(t->level_2_wait);
                           if(t->level_2_wait >=6)
                          {
```

```
                                tmp=e;
                                t->level=1;
                                t->ticks=0;
                        }
                         else
                                        tmp=NULL;
                        //          }


                }
   if(tmp!=NULL)
                                {
                                        struct thread *f=list_entry(tmp,struct thread,elem);
                                        list_remove(tmp);
                                        list_push_back(&ready_list_1,tmp);
                                        f->level_2_wait = 0;
                                        printf("%d: thread %d goes from L2 queue to L1
queue\n",clk,f->tid);
                                }
          }

 }


 if(cur!=idle_thread){
  if(x==1)
  {


          printf("%d: thread %d goes from L1 queue to L2 queue \n",clk,cur->tid);
          cur->status = THREAD_READY;
          cur->level=2;
          cur->ticks=0;
          cur->level_2_wait=0;
          list_push_back (&ready_list_2, &cur->elem);
    schedule ();
   /* struct thread* f= thread_current();
          printf("\n%d: thread %d goes from L%d to running state",clk,f->tid,f->level);*/
  }
   else if(x==2)
  {

          printf("%d: thread %d goes from running state to L1 queue\n",clk,cur->tid);
```

```
          cur->status = THREAD_READY;
          list_push_back (&ready_list_1, &cur->elem);
     schedule ();
          /*struct thread* f= thread_current();
          printf("\n%d: thread %d goes from L%d to running state",clk,f->tid,f->level);*/
  }
  else if(x==3)
  {


    printf("%d: thread %d goes from running state to L2 queue\n",clk,cur->tid);
          cur->status = THREAD_READY;
          cur->ticks=0;
    list_push_back (&ready_list_2, &cur->elem);
     schedule ();
 /*   struct thread* f= thread_current();
          printf("\n%d: thread %d goes from L%d to running state",clk,f->tid,f->level);*/
  }
  }

  intr_set_level (old_level);
}
```

6. Thread_unbolck changed:

```
  if (t->level==1) {
    t->status=THREAD_READY;
    list_push_back (&ready_list_1, &t->elem);
  }
  else {
    t->status = THREAD_READY;
    list_push_back (&ready_list_2, &t->elem);
  }
```

7.next_thread_to_run changed :

```
if (list_empty (&ready_list_1)) {
    if (list_empty (&ready_list_2)  return idle_thread;
    else return list_entry (list_pop_front (&ready_list_2), struct thread, elem);
  }
  else
    return list_entry (list_pop_front (&ready_list_1), struct thread, elem);
```

8. Create_thread modified :

These lines added after init_thread call

```
  t->level = 1;
  t->ticks = 0;
  t->level_2_wait = 0;
```

Memory Management :

For this part the files modified are : malloc.c and malloc.h only.

The main functions are malloc and free, in the default memory allocation algorithm of the pintos,
First for malloc, size of each requests which is in bytes is rounded to the next power of two, we have a list of descriptors one for each size of possible blocks. They manage the blocks of a particular size like for example they contain a list of free blocks, size of the blocks which they manage etc. Now in malloc, we check the list of appropriate descriptor, if the list is non empty we pop the list and return the block.

Otherwise, a new page of memory, called an "arena", is obtained from the page allocator (if none is available, malloc() returns a null pointer). The new arena is divided into blocks, all of which are added to the descriptor free list. Then we return one of the new blocks.

Now for the free function, we add the block to the respective descriptor list, and now we check the arena, if the arena has all of its blocks free, then we remove the arena and free the page.

We can't handle blocks bigger than 2 kB using this scheme, because they're too big to fit in a single page with a descriptor. We handle those by allocating contiguous pages with the page allocator and sticking the allocation size at the beginning of the allocated block's arena header.

Now for buddy algorithm, on malloc, we see if the descriptor list is non empty, if yes then we return the popped block, if no, we first search the preceding descriptor list for block sizes twice the size of block requested, if such a block is free we divide the block into two parts, return one block and add the other block to the respective descriptor list, we follow this step recursively until we find a non empty list and then keep on dividing one popped block until we get the block of requested size. If we still cant find a block we request for a new page.

Now, for free, we first see if the buddy of the block is free or not, buddy of a block is the consecutive block which together combine to form a larger block, if the buddy is free, we merge them into a bigger block which is free, add it to the respective list and then repeat this step recursively for larger blocks, finding their buddies, seeing if they are free and merging them together.

For memory management, the following have been changed :

1. The block structure is modified

```
/* Free block. */
struct block
 {
        struct list_elem free_elem; /* Free list element. */
        size_t size;
 };
```

2. Following three functions are added :

```
static void malloc_breakdown (struct desc *,struct desc *);
static void free_buildup (struct block *, struct desc *);
static void printMemory(void);
```

3. Malloc is modified and it also uses malloc_breakdown as the helper function

```
if (list_empty (&d->free_list))
        {
        int result=0;
        t=d;
        for (t = d; t < descs + desc_cnt; t++) {
        if (!list_empty(&t->free_list)) {
        result=1;
        break;
        }
        }
```

*****We search for parents blocks which are free in case the list for current block size is not free, then we call the helper function for further processing, which breaks the parent block into children buddy blocks until one block of the requested size is found.

4. Free is modified and it also uses free_buildup as the helper function

```
 for (e = list_begin (&d->free_list); e != list_end (&d->free_list); e = list_next (e))
```

```
        if ((uint8_t *)&b->free_elem < (uint8_t *)e)
        break;
        list_insert (e, &b->free_elem);
        a->free_cnt++;
```

*****We insert the current block into the appropriate list and then call the free helper function which merges the buddy blocks and then check the parent's list for possibility of merging

**********All insertions are in ascending order.

5. Since we already added the size attribute in the block structure, the block_size function is changed

```
/* Returns the number of bytes allocated for BLOCK. */
static size_t
block_size (void *block)
{
  struct block *b = block;
  //struct arena *a = block_to_arena (b);
  //struct desc *d = a->desc;
  return b->size;
  //return d != NULL ? d->block_size : PGSIZE * a->free_cnt - pg_ofs (block);
}
```