Assignment - 5

==========

Group-2

Group Members :

1. Ashrujit Ghoshal (14CS10060)
2. Sayan Ghosh (14CS10061)

==========

Part 1 : Scheduler

Existing scheme :

Pintos uses a single ready queue and a round robin scheduling scheme. The time slice for each process is T. If a process is not completed in T time, it is preempted and next process is scheduled.

thread_yield() decides the next process to run by calling the schedule() function. This function yields the CPU.
Schedule() function performs context switch. It takes current thread (using running_thread()) and the next thread (using next_thread_to_run()) . Then it performs a context switch using a assembly code named Switch.S. In case a thread has finished running then the thread_schedule_tail() takes care of that and if the thread is dying, it changes the state and frees the memory allocated to it.
next_thread_to_run() schedules a process from the ready queue by popping from front of ready_list.
To care of round robin scheduling thread_tick () is called at end of every clock tick and takes care of preempting the process if it exceed the time slice alloted

Modification in the data structure :

1. In structure thread : (Done in thread.h)
    a. Variable ‘count1’ is added (type : int)
    b. Variable ‘count2’ is added (type : int)
    c. Variable ‘level’ is added (type : int)
    d. Variable ‘created’ is added (type : int)

e. Variable ‘bstatus’ is added (type : int)
f. Variable ‘blocked’ is added (type : int)
g. Variable ‘unblocked’ is added (type : int)

2. Other variables added/removed :
   a. ready_list is removed
   b. ready_list1 is added (type: list)
   c. ready_list2 is added (type: list)

Note: The variables bstatus, created, blocked and unblocked have been added for printing the transition of states of the thread. For implementation of our scheduler only count1, count2 and level are sufficient.

Functions added : (in thread.c)

1. thread_unblock_create():
   This function is called to make the state of created thread from blocked to ready. The thread is also pushed into the all_list.
2. print_queue():
   Helper function to print queues for debugging
3. print_queue_all ():
   Helper function to print all_list and the status variables of blocked and unblocked.

Changes in functions :

1. thread_tick ():
   Added a check to see if ready_list1 is empty. If empty, then schedule process form ready_list2 (without moving the process to ready_list1).
   Added a for loop to increment count2 of the processes in the ready_list2.
   In this function for every tick we check the blocked and unblocked status of a thread. We traverse the all_list (as when a process is blocked it is not in ready_list) to check for the blocked status.
   To check the unblocked status we traverse the ready_list1 and ready_list2 separately and print if required.
   Added a check to see if any process in the ready_list2 has aged.
2. thread_yield ():
   The count1 for the running process is incremented.
   Added a check to see if the running process has already taken 2T time.
   Added a check for the time elapsed before invoking scheduler for round robin.

If the level of the process is 1, then use default TIME_SLICE.
If level 1 list is empty then round robin is enforced for level 2 with twice TIME_SLICE.

3. next_thread_to_run():
   If both ready_list1 and ready_list2 are empty then we return idle thread. If ready_list1 is empty then we schedule a thread from ready_list2, otherwise a thread from ready_list1 is scheduled. Always the thread is popped from front of the ready_list.
4. thread_unblock ():
   The unblocked flag is set to 1 and bstatus is set to 0.
   If the level of the thread was 1 when blocked it is unblocked and pushed into ready_list1 else pushed into ready_list2.
5. thread_block ():
   In this function we set the blocked variable to 1 and bstatus variable to 1.
6. thread_init ():
   Initializing the count1 and count2 variables of a thread to 0.
7. init_thread ():
   Set level to 1 for a new thread.

For both the situations above, whenever a process is inserted into any list it is always inserted at the end. For this we invoke the pushback method declared in list.h and defined in list.c

Description of implementation:

We keep 2 lists of process named as ready_list1 and ready_list2 corresponding to the 1st and 2nd level of multilevel queue respectively.

A process when created is put in the ready_list1. It is always inserted at the end of the pre-existing list. The count of time a process takes is taken care of by using two variables stated earlier -- count1 and count2.

Count1 stores the time a process spends in ready_list1 and similarly count2 stores the time the process spends in ready_list2. The variable count2 thus is used to take care of aging of a process.

Thread_tick is called at end of every clock tick. In this function the count2 variable for every process in the ready_list2 is incremented by 1. This is done by using a loop and then traversing the doubly linked list, ready_list2. In course of this traversal if a process has spent 6T time in ready_list2 we move it to ready_list1. The processes aged are appended to the end of the ready_list1.

In the thread_yield function the count1 of the running process is incremented by 1. After that a check is performed to see if the process should be moved to ready_list2. In case the process has already spent 2T time in ready_list1 and still has not finished, it is then appended to the end of the ready_list1.
Before pre-empting a thread we check whether the process was originally called from level 1 or level 2. If the process belonged to level 1 then use the default time slice, else we check for double the time slice before preempting it.

Moreover, when a process is shifted from ready_list1 to ready_list2 the value of count2 is reinitialized to 0. Similarly , after aging when the process is moved back again to ready_list1 the count1 is then reinitialized to 0.

For the method thread_unblock () we check the level of the thread. Based on it the process is appended into the suitable list.

We added the print_queue () function to print the ready_lists for debugging. If a list is not empty the function prints it.

Part 2:Memory management system

Existing scheme and implementation :

1. The existing implementation of the memory management system is fairly straighforward. There is a descriptor table that has pointers to free blocks

memory for different sizes(powers of 2 from 16 upto 1024). Pages are called arena which have a variable keeping count of number of free blocks in the arena and a magic variable which checks for corruption in memory.Arena is divided into blocks which are allocated .

2. The following happens in a malloc call:
The OS(in malloc function) rounds up the memory asked for to the nearest power of 2 greater than it , searches for free blocks of the size in the descriptor table.If free block is found,it is removed from the list in descriptor table and allocated to user. If not found, OS calls palloc function to get a new page. It breaks the page into several chunks of required size,allocates 1 to user and adds the rest to the free list of the corresponding entry in the descriptor table.

3. The following happens in a free call:
The page corresponding to block is found and the entry in the descriptor table is found from the page. The block is added to the free list in the corresponding entry of descriptor table.
4. The following happens in a realloc call:
First malloc is called to get the memory of new size. Next the contents of existing memory are copied to the newly allocated memory
(if new_size < old_size then only elements that fit are copied). The newly allocated memory is returned.

Modifications in data structures:

1. Added variable ‘size’ to block.It is block size. We need this since blocks in the same page are of different sizes and we need to keep track of it.
2. Added variable magic to block (to check memory corruption)

Note that we have removed the use of arena (since every page has blocks of different sizes. Also in the original implementation some extra size was allocated to store data of arena like magic, desc and hence it would have reduced the size of the last block in a page. We removed this shortcoming in our implementation.We keep track of a page by keeping track of starting addresses of page.)

How our implementation works (Functions Modified):

The following happens in a malloc call(modification in malloc function):
1.First we add the size asked for to the sizeof block(since each block contain data like size and magic and that part should not be overwritten). If the size is greater than size of a page we allocate and return multiple pages(blocks of 2048B size) similar to the original implementation.

2.We search the desc list to find the smallest free block of size greater than required. There may be 3 possibilities:
a.We find a free block of size power of 2 that is greater than and nearest to required size. In this case we remove the block from the free list and return it to pointing just after the data of b(so that fields are not overwritten)
b.We find a free block of size power of 2 that is greater than and not the nearest to required size. In this case we use the malloc helper function to split the block into 2 equal blocks,add one of the block in into free list of corresponding descriptor,check if the size is just greater than or equal to required size. If yes we return the block ,else we continue the process recursively.Note than while insertion in free list of descriptor we do the insertion in sorted order(sorted by address)
c.There are no free blocks of size greater or equal to required size. We allocate a new page and then follow the same procedure as in b

The following happens in a free call(modifications in free function):

1. We check the block size. If block size> page size we free multiple pages similar to that in the original implementation
2. Otherwise we find the descriptor entry of corresponding size and call free helper function
3. In free helper function we traverse free list of descriptor. We note that since memory is allocated in powers of 2, if 2 blocks are buddies they shall differ in only 1 bit and hence their bitwise XOR will yield the size of the block(only the differing bit will be 1). We check the bitwise XOR of address of the block to be freed with all blocks in this list while traversal.If one of results yields result equal to size of block,it means we have found the buddy. We remove the buddy from this free list, merge the 2 buddies and recursively call the helper function. If buddy isn't found we just add it to the free list such that the list remains sorted.

The following happens in a realloc call(modifications in realloc functions):
First malloc is called to get the memory of new size. Next the contents of existing memory are copied to the newly allocated memory
(if new_size < old_size then only elements that fit are copied). The newly allocated memory is returned.
Only difference from original implementation is that here finding block size is easier.

Functions added:

1. malloc_helper(struct desc *d,struct block *b,size_t required,size_t split):-
   Parameters:
   1.d is the descriptor
   2.b is the block that needs to be split
   3.required is the required size
   4.split is the current size of split block
   Return type:void
   It is a recursive function that helps to split block b into two, add 1 to free list, check if required size is satisfied.If no, then it continues recursively(changes the descriptor and halves split), otherwise returns the block that it didn't add to the free list.

2. free_helper(struct block* b, struct desc* d):-
   Parameters:
   1.b is the block to be freed
   2.d is descriptor of corresponding size.
   Return type:void
   It is a recursive function that traverses the descriptor to find buddy.We note that since memory is allocated in powers of 2, if 2 blocks are buddies they shall differ in only 1 bit and hence their bitwise XOR will yield the size of the block(only the differing bit will be 1). It checks the bitwise XOR of address of the block to be freed with all blocks in this list while traversal.If one of results yields result equal to size of block,it means buddy has been found . The buddy is removed from this free list,2 buddies are merged and recursively it calls itself(with the merged

block and its corresponding descriptor). If buddy isn't found it just adds the block to the free list such that the list remains sorted.

3. list_order_insert(struct list *l,struct list_elem *e):This function inserts element e into list l keeping it sorted. It returns the sorted list

4. getPage(struct list_elem * e[]): This is a helper function used for printing the page allocation.It returns address of page with smallest address where e[i] is the position of the pointer in the free list of descriptor i.It returns start address of the page

5. printMemory():This function is used for printing the page allocation in the format prescribed by the assignment.Initially all pointers in sorted free lists of descriptors are placed at the beginning.The lowest page address is found using getPage() and all blocks in the page is printed.

Files modified:

1. thread.h
2. thread.c
3. malloc.h
4. malloc.c