GROUP : 12
SAYAN MANDAL
SOURAV PAL

1 . How the existing Pintos scheduler works :

The existing scheduler in pintos follows simply Round Robin algorithm in thread_ticks() function when thread_ticks becomes greater than TIME_SLICE it calls intr_yield_on_return() which sets the external interrupt flag true in interrupt.c and henceforth thread_yield() is called. Thread_yield() function first puts the current thread into ready_list after setting the state of the running_thread to READY_STATE and then calls schedule(). schedule() function does the scheduling . At entry, interrupts remains off and the running process's state must have been changed from running to some other state. This function finds another thread to run and switches to it. next_thread_to_run() does this job . It just simply pops off the front element of the ready_list and set its state to RUNNING_STATE, and context_switch happens happens between current_thread and next_thread_to_run .

The function for switch_threads is present in Switch.s which is responsible for context switch between two threads. This function first saves caller's register state .Then it saves current stack pointer to old thread's stack, if any and then restores stack pointer from new stack pointer from new frame's stack and restores caller's register stack.

After this function , thread_schedule_tail() is called . this function sets the current thread's status to RUNNING_STATE . If the previous thread which was just switched is dying then it just frees the page corresponding to that thread if and only if the previous thread is not idle_thread.

2 . Data Structures Added Or Modified :

A new List multi_list is added to maintain the Level 2 queue.

3 . Functions Modified :

Field added in struct thread in thread.h:

1. running time => running time of the thread
2. waiting time => waiting time of the thread
3. Queue => 1 implies the queue from which the thread was scheduled from level 1 , else it was scheduled from level 2

a. thread_create():

Added a static clock and initialised it to 0 to maintain the printing.

B. thread_tick():

Kept a status variable which will denote is the level1 queue is empty or not . If level 1 queue is empty then time quanta will be 2T , else time quanta will be T . Increasing the value of clock which is for printing purpose . Increasing the running time of each thread in level 1 queue and waiting time for each thread of level 2 queue . Transferring those threads from level 2 to level 1 whose waiting time is greater than 6T .

C. thread_unblock():

In this function , when unblocking a thread we are just seeing the queue field of the corresponding thread . If it is set to 1 , that means we are inserting in the level 1 queue , else we are inserting in the level 2 queue .

D. thread_yield():

If the queue field of the thread is set to 1 then the thread is entered into the level 1 queue , else the thread is entered into the level 2 queue. For each thread in the level 1 queue if the running time of the thread is greater than 2T then running time and waiting time is resetting to zero and transferred to the level 2 queue .

E. next_thread_to_run():

If the level 1 queue and level 2 queue is both empty then idle_thread is returned . if level 1 queue is empty then next thread is taken from level 2 queue and queue field of the thread is set to 0 . If level 1 queue is not empty , then next thread is taken from level 1 .

F. init_thread():

In this function , the running_time is set to 0, waiting_time is set to 0 and Queue is set to 1(level 1 queue).

4. Files Changed:

1 . thread.c

2 . thread.h

**************************Buddy memory allocation************************

1 . HOW EXISTING MALLOC WORKS :

Existing malloc works in the following way : It keeps array of descriptors of size 7. Each descriptor denotes the block size corresponding to that descriptor. Each block keeps a freelist of block size corresponding to that block. The size of each request, in bytes, is rounded up to a power of 2 and assigned to the "descriptor" that manages blocks of that size. The descriptor keeps a list of free blocks. If the free list is nonempty, one of its blocks is used to satisfy the request.When we free a block, we add it to its descriptor’s free list. But if the arena that the block was in now has no in-use blocks, we remove all of the arena's blocks from the free list and give the arena back to the page allocator.

If the freelist is empty corresponding to that descriptor then we get a page from page allocator and divide the page into the block_size corresponding to that block and add the blocks to the descriptor.

2 . How Existing Free Works :

Frees the corresponding block and increase the free_count of the corresponding arena of the block and add the free block to the free list of the corresponding descriptor . if the corresponding arena is totally unused then the page is returned to the page allocator .

3. Data Structures Added / Modified :

1. A list named Arenalist is added to maintain the list of arenas that have been allocated till now.
2. A struct called metadata_t is introduced which denotes the block which has a prev pointer denoting the previous block and a next pointer denoting the next block.
3. Each arena contains a magic number to detect the overflow , 8 types of pointer to metadata_t and a list element.
4. A struct address is added for printing purpose which contains the starting address of the block and a list element to insert it in the list.

4. Functions Modified/Added :

malloc():

First it checks for every arena in the arena list if the size needed can be allocated in any arena , if so it just allocates and returns. Else it demands a new page from page allocator and allocates the required size in that new page and returns the starting address of the block allocated .

getblock():

It takes argument as the starting address of the page, the size required and the page number and returns the starting address of the block allocated in that page . It first check the index required for the corresponding size by calling get_index() . If there's a block of memory of that size in the freelist, it gets it and returns it. It separates that block from previous and next block and marks that block as used. If there is a next block for that block then freelist[index] will be the next block. Now just return the starting address of the block .

Now if there is no free block corresponding to that index it just searches for any block of size greater than the required size whose block is free. It then just divide by two and adjusts the freelist accordingly till the block of required size get allocated and then just sets the value in the return block and returns the starting address of the block.

offset_pointer():

Function to offset the pointer. Takes in a metadata_t *and an int offset that determines which direction to go.Returns either ptr + sizeof() if offset is 1 else returns ptr - sizeof() .

free():

It takes an address as its input and return value is nothing . now it just searches for the page for which this address is valid . now it gets the metdata_t pointer by calling offset_pointer with offset equal to 0.It resets the in_use field of the block to 0 .
It then calls find_buddy() function to get the buddy of that block . While buddy exists and it's not in use, merges it and modifies the linked list of the freelist accordingly. Now add that block to the freelist[index] corresponding to the block size . now if the whole page is unused then free that page return it to page allocator by calling palloc_free_page().

find_buddy():

Find the buddy for a given block. This is done fairly simply. Since the address of the buddy should just be size away from the block, either add or subtract that depending on which buddy you have. Alternatively, you could XOR the address by the size, same thing as:

ptr +/- (1 << log2(size))

Then return it.

5 . Files Changed :

1. malloc.c
2. malloc.h