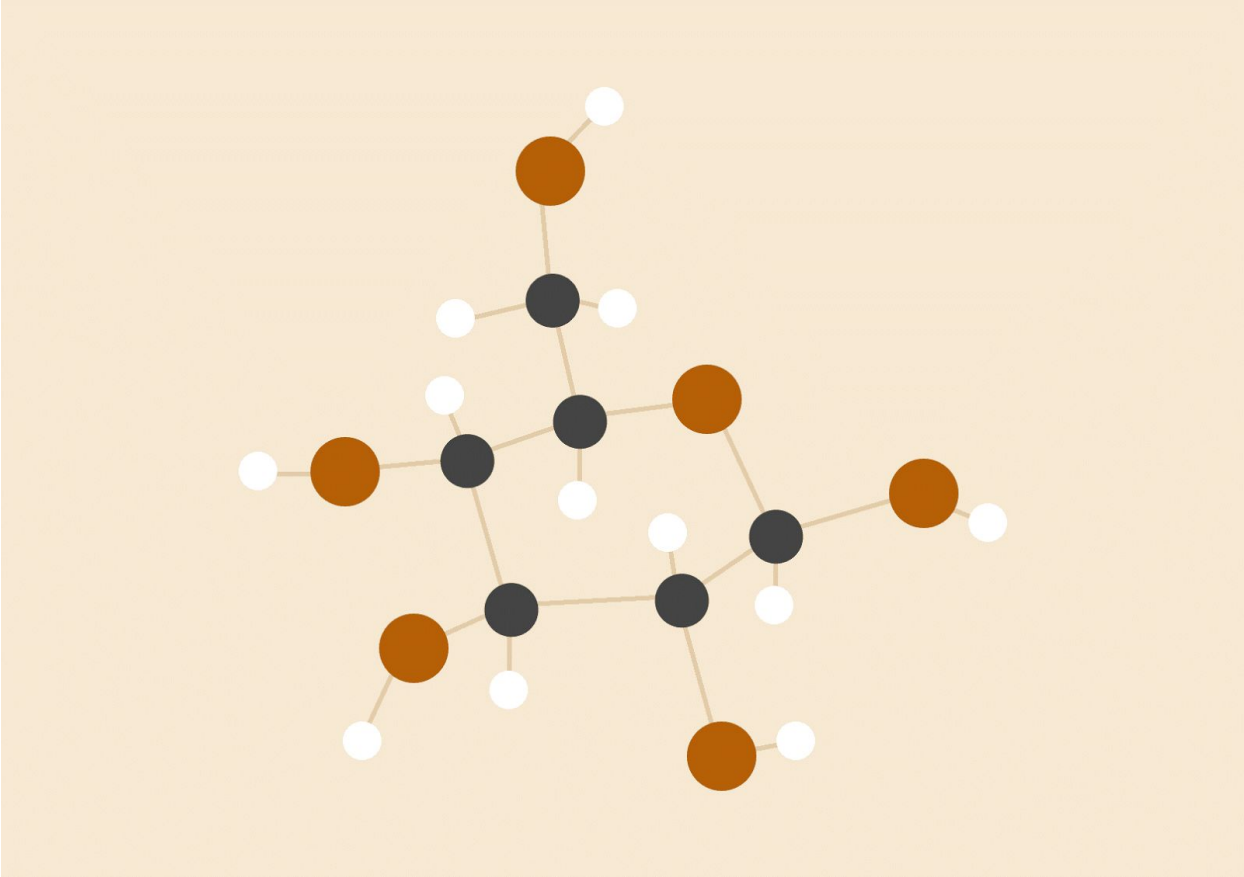
OPERATING SYSTEMS

Implementing Multilevel Feedback Queue in Pint OS



Arijit Panigrahy | 14CS30005

Aniket Choudhary | 14CS30004

26.03.2017

Operating Systems Laboratory

GROUP:- 1

OBJECTIVE

The existing PINT OS used a round-robin scheduling policy with time quanta=4. The objective is to modify the required files to use a multilevel feedback queue with 2 levels.

WORKING OF THE EXISTING CODE

There are two files (main ones) for creating, scheduling and other functions of threads.

1. Thread.h
   The thread has the following members for :-
   - Thread Id
   - Thread Status
   - Thread's name
   - Stack :- It is needed to keep track of its state.
   - Thread priority (not needed for default one)
   - allelem :- in order to keep track of all the threads that are created but not destroyed.
   - Elem :- The element that is either put in the ready list or in the list of waiting threads.
   - Magic :- To check the overflow of the stack.
2. Thread.c
   - thread_init()
     Main calls the thread_init() function to initialize the thread system. It is called in the init.c. It calls init_thread and creates a thread named main. It allocates the tid=1 for the main thread.
   - init_thread()
     It initializes the thread parameters after adding to the blocklist and adds the elem to the all_list.
   - thread_start()
     It is called by main() to start the scheduler. It creates the idle thread with a tid=2 and enables the interrupt (thereby calling the scheduler).
   - thread_create()
     It creates and starts a new thread with some given parameters such as name, priority, function to execute etc. It allocates a page for the thread structure, sets up the stack and calls thread_unblock() {it was in block state

earlier}.

- thread_unblock()
  It changes the status of the thread to THREAD_READY and adds it to the ready queue.
- thread_block()
  Changes the status of the thread to THREAD->BLOCKED and call the scheduler.
- thread_tick()
  Called by the timer interrupt and keeps tracks of various time counters. Also, it triggers the scheduler when the time slice expires.
- thread_print_stats()
  Prints the thread statistics.
- thread_current()
  Returns the running thread.
- thread_exit()
  Removes the thread from the all_list. Changes the status to THREAD_DYING and calls the scheduler.
- thread_yield()
  Yields the CPU to the scheduler by putting the current thread on the list and changing its status and calling the scheduler.
- idle()
  It blocks as soon as it is put to run. It never returns to the ready list and remains blocked.
- is_thread()
  Checks if the thread is not corrupted or exists.
- aloc_frame()
  Allocates a frame for the thread by creating space.
- next_thread_to_run()
  Chooses a thread from the ready list and returns (here, the first thread).
- schedule()
  It records the current thread in local variable cur, determines the next thread to run as local variable next (by calling next_thread_to_run()), and then calls switch_threads() to do the actual thread switch. The thread we switched to was also running inside switch_threads(), as are all the threads not currently running, so the new thread now returns out of switch_threads(), returning the previously running thread.
- Switch_thread()

It saves registers on the stack, saves the CPU's current stack pointer in the current struct thread's stack member, restores the new thread's stack into the CPU's stack pointer, restores registers from the stack, and returns.

- thread_schedule_tail()
  It marks the new thread as running. If the thread we just switched from is in the dying state, then it also frees the page that contained the dying thread's struct thread and stack.

DATA STRUCTURES CHANGED

Following things were added in thread.c.
unsigned l2_count; //For running time of the current thread in L1.
unsigned l1_count; //For waiting time of the threads in L2.
bool is_l1; //To indicate if a thread is present in L1 or not.

Two new lists were created instead of ready_list. One for level 1 and other for level 2.
static struct list l1_list;
static struct list l2_list;

FUNCTIONS MODIFIED

1. Thread_init():
   Initialized l1_list and l2_list.
2. Thread_unblock():
   When a thread is unblocked, we check if it was earlier present in Level 1 or Level 2 and push_back accordingly to the list.
3. thread_yield():
   This function actually moves the thread from L1->L2 and initializes the L2 count. If the time_slice has expired (Known from thread_tick()) or the thread was present in L2 earlier, we move the thread to l2_list, else put it in l1_list.
4. init_thread():
   This function initializes l1_count to 0 and turns is_l1 to true initially and puts the thread in all_list.
5. next_thread_to_run():
   If l1 is not empty, return the front element of l1, else if l2 is not empty, return the front element of l2, else return idle_thread.

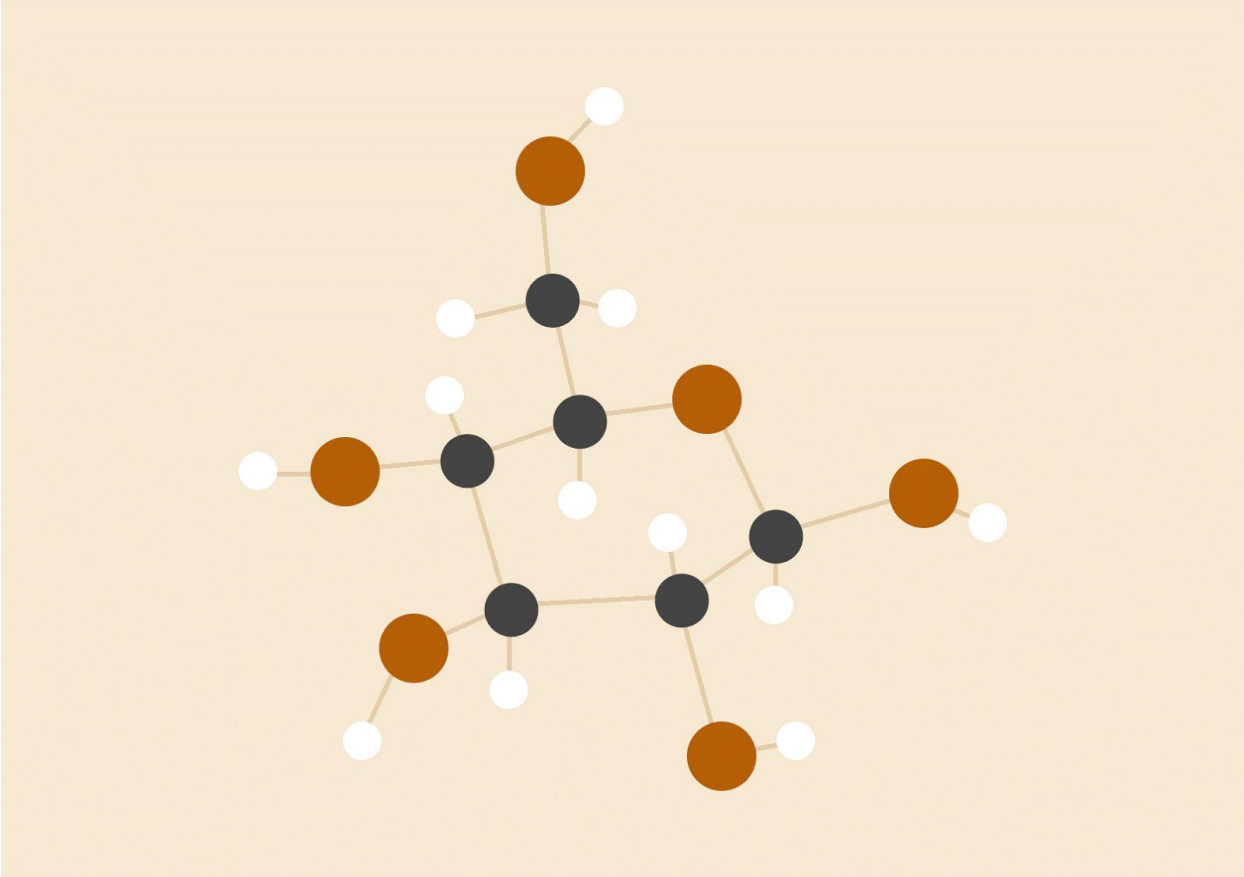
6. thread_tick():
   If the thread that is running is from l1, increment the l2_count of all the threads and also increment the l1_count of the running thread.
   If the l2_counts of the threads in level 2 has exceeded 6*T, move them from l2 to l1 and initialize the counters.
   If the l1_count of the running thread has exceeded the time slice, yield the CPU. Also, if the l1_count has exceeded 2*T, turn is_l1 false so as to be moved to level2 by the thread_yield() function.

FILES MODIFIED

Thread.h :- To modify the data structure.
Thread.c :- To modify the functions associated with thread_creation and scheduling so as to use MLFQ scheduler.

OPERATING SYSTEMS

Implementing Buddy Memory Allocation in PINT OS



Arijit Panigrahy | 14CS30005

Aniket Choudhary | 14CS30004

26.03.2017

Operating Systems Laboratory

OBJECTIVE

The existing PINT OS used a simple memory allocation algorithm. The objective of the assignment is to design a buddy memory allocation system.

WORKING OF THE EXISTING CODE

There are two files (main ones) for creating, scheduling and other functions of threads.

1. Malloc.h
   The malloc.h has the declarations of some function prototypes implemented in malloc.c.

2. Malloc.c

   It has the declarations of structures of descriptor, arena, block etc.

   - malloc_init()
     Main calls the malloc_init() function to initialize the memory allocation system. It initializes the descriptors and the list (as well as locks) associated with it. Now, malloc() can be called.
   - malloc(size)
     It obtains and returns a new block of at least SIZE bytes. It finds the nearest power of 2 >= SIZE say d. It checks if d>=2KB (1KB in code), and if it is it allocates multiple pages and initializes the arena. If it is less than 2KB (1KB), then it checks the corresponding descriptor has any free descriptor. If no, it gets a page, breaks it into a number of blocks and puts it in the free list of the descriptor. After that, it pops a block from the free_list and returns it.
   - calloc(size a, size b)
     It returns a block of size a*b after initializing it to 0. It calls malloc for getting the block.
   - realloc(void * old, size new)
     It copies the old block in a new block of given size. It gets the new block by calling malloc and frees the old block by calling free.

- free(void * block)
  It frees the block which has been allocated before. It gets the block size using the block_size() function. If the block_size>=2KB, it frees using palloc_free_multiple. Else, it adds the block address to the free list of the descriptors. It then checks if all the blocks of a page is in the free_list, in which case, it frees the page.
- block_to_arena(void* block)
  Returns the corresponding arena by pg_round_down.
- arena_to_block(arena,index)
  Returns the block address by adding the address of arena and index*size_of_block.

DATA STRUCTURES CHANGED

Arena structure was removed completely.

In the block structure, a field for block_size was added.

FUNCTIONS MODIFIED

Anything related to arena has to be removed. Since, I have deleted the arena data_strucuture.

1. Malloc_init():
   Initialized the descriptors,
2. Malloc():
   The modification is for the case when the size<=2KB and we have to get a page. We have to fetch the page, divide it into two blocks, put one block in its corresponding descriptor, and check if the size of other block == required size (nearest power of two). If it is true, put this block in the descriptor list as well else keep on dividing by two and checking.
3. Free():
   The modification is for the case when we want to free a block of size <=2KB. We add the block to the free list. We check if the buddy of this block is free. If yes, we put the lower of the block and its buddy in the higher descriptor. (For ex. If blocks 1000 and 1032 are free and present in 32B desc and are buddies, we remove 1000 and 1032 from 32 byte desc and put 1000 in 64 B desc.). We continue this process until the buddy is not found or the size of desc reaches 2KB. In this case, we free

the page.

4. Realloc(void* b):
   It copies the old block in a new block of given size. It gets the new block by calling malloc and frees the old block by calling free. We have to get the old_block size by accessing block->block_size. In order to do it, we have to access the starting address of the block by b-1.
5. PrintMemory():
   It prints the descriptor free_lists sorted by page numbers. The starting address of the page are stored in the array. It just round_down the free_list element and if it is equal to the page_header print it. In the free_list, the blocks are inserted after sorting.

FILES MODIFIED

Malloc.c :-To implement the buddy memory allocation.