Design Document - Frame Manager

Operating Systems - MP2 - Rajendra Thottempudi

Design :

The idea of this machine problem is to make us design and build a frame manager which has the ability to allocate and manage a set of frame pools which contain continuous frames allocated to different processes.

The total size of the system memory available is 32MB out of which 4 MB is used by the kernel and a 1MB (which is present at 15 MB) which is inaccessible. The rest of the memory can be freely mapped to any of the pages as and when required. I have used a frame size of 4KB which makes the maximum number of frames possible = 8k. Also, since we are trying to build a bitmap which stores the states of all the frames - we want to make the size of a bitmap such that it can fit in a single frame.

The number of bits that can be stored in one frame = 4096 * 8 ~ 32k
Since we are using two bits to store the state of a single frame, the number of frames whose states can be stored in a single frame = 32k/2 = 16k

Now, to implement this frame manager, several modules have been created in the Cont_frame_pool class. The following api's in this class are written :

- The Constructor - ContFramePool
- get_frames method - to return the frames as asked by the process whenever possible
- mark_inaccessible method - to ensure that the frames present in the place in the memory which is inaccessible is not returned to any process
- release_frames method - to release the frames and to make them free/available for other processes
- needed_info_frames

Also, assuming that there can be any number of frame pools, we need to maintain the release frames as a static method and we need to call the release frames method of a particular frame pool after ensuring that the frames in question belong to that particular pool.

Now, each frame can possess one of the following states : HeadOfSequence, Used, Free, Inaccessible. We can manage these 4 frames by using only 2 bits as follows : 00 refers to a free frame, 11 refers to a frame being used, 10 refers to a frame that is inaccessible and 01 refers to a frame which is the head of a sequence. Also, we now add the variables required to keep a track of the frames in Cont_Frame_Pool.H. Apart from the regular variables, I have used a static list to keep a track of the frame pools.

Implementations of each of the above functions :

Constructor : This is the constructor which will be called when we are trying to initiate a frame pool. It takes the base frame number, the number of frames in that pool and the frame number in which

management information related to this frame should be stored and assigns them to the corresponding private variables. We also need to check if the number of frames that are being asked could be stored in a single bitmap frame. Along with this, we also need to add the frame pool to the static pool list and store its information for future reference.

Get_frames method : In this method, we take the input as the number of frames that is being asked by the process and return the frame number of the head if that particular number of frames could be found in our bitmap which are free and are continuous. We first check if the number of frames that is being asked is less than or equal to the number of free frames available with the memory. Then we search all the current frames in our bitmap to check if we can find a continuous group of frames which are free. If found, we change the state of the first one to head and change the state of all the other frames as used and return the head to the process. Certain bit manipulations were needed to find the state of a particular frame within the bitmap. As we traverse throughout the frames it takes a time of the order of O(n) where n is the number of frames that we have. I.e. we take a linear search time to find if we and return the head.

Bit manipulation example :

If we have a byte with the value byte_a = xxxx01xx and we want to know the state of the third frame which has 01 in its bits we can do the following :

Let i = 11000000 (this represents a byte in binary system)
If we right shift it by 4 we get - i = 00001100 (i.e. aligning the position of 1s with that of the frame position in the byte)
Now we perform the operation -> byte_a & i
I.e. xxxx01xx & 00001100
If the output of this operation = 00000100 which when right shifted by 2 will be 00000001 which is equal to int 1. Thus, if the output of the above operation is int 1 then we can confidently say that it is a head frame. If the output is int 0 then the frame is a free frame if the output is int 3 then it is a used frame and if it is 2 then it is an inaccessible frame. This way we can get the state of all the frames within a byte by bit manipulation.

Mark_inaccessible method : In this method, we mark the frames which are inaccessible as 01 by traversing through the bitmap initially for that part of the memory which cannot be accessed by any process and is used by the kernel.

Release_frames method : In this method, we are supposed to mark a set of frames that are being used to free i.e. the process no longer needs them and they must be made available to the other processes. To do this, we first need to check the frame pool to which those frames are a part of. Once we find that, we get the bitmap for that particular frame pool. Then we check if the frame that we have in the argument is a head frame or a used frame - If it is a head frame, we have to change the state of this frame and all the following frames until we reach another free or head frame. If it is a used frame, we just change the state of the frame to free and return. Bit manipulations were needed to achieve this and the tricky part is to ensure that only the 2 bits of the current frame are changed and none of the other bits in that particular byte are changed.

Bit Manipulation example :

Let the byte containing the state of frames byte_a = xxxx11xx (in binary form) and we want to change the state of the third frame (second frame from right) from used 11 to free 00 without changing any of the other frame states in the byte.
Now, if we take a masked 8 bit number primary_mask = 11 (= 3 in decimal)
Left shifting the above mask to align the position of 1's with those of the above frame we get the primary mask to be -> primary_mask = 00001100;
On inverting this primary mask bit wise we get -> primary mask = 11110011
Now if we perform an and operation with byte_a and primary mask :

Instance 1 : if byte_a = 01001100

byte _a & primary_mask = (01001100) & (11110011) = 01000000

If we observe the output, we see that the state of frame 1, 2 and 4 in the byte remained the same as they were before the bit manipulation but the state of only the third frame has changed from 11 to 00 as desired.

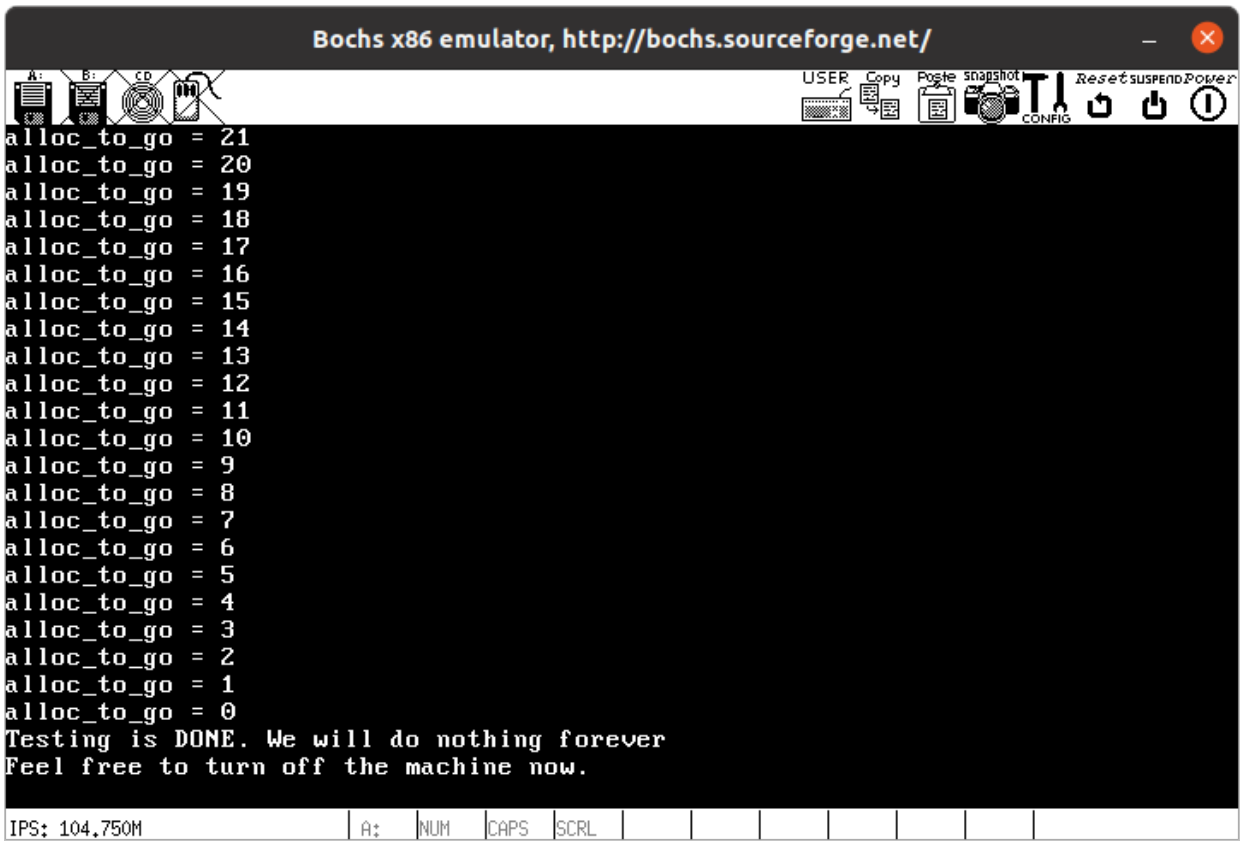
Instance 2 : if byte_a = 11001101

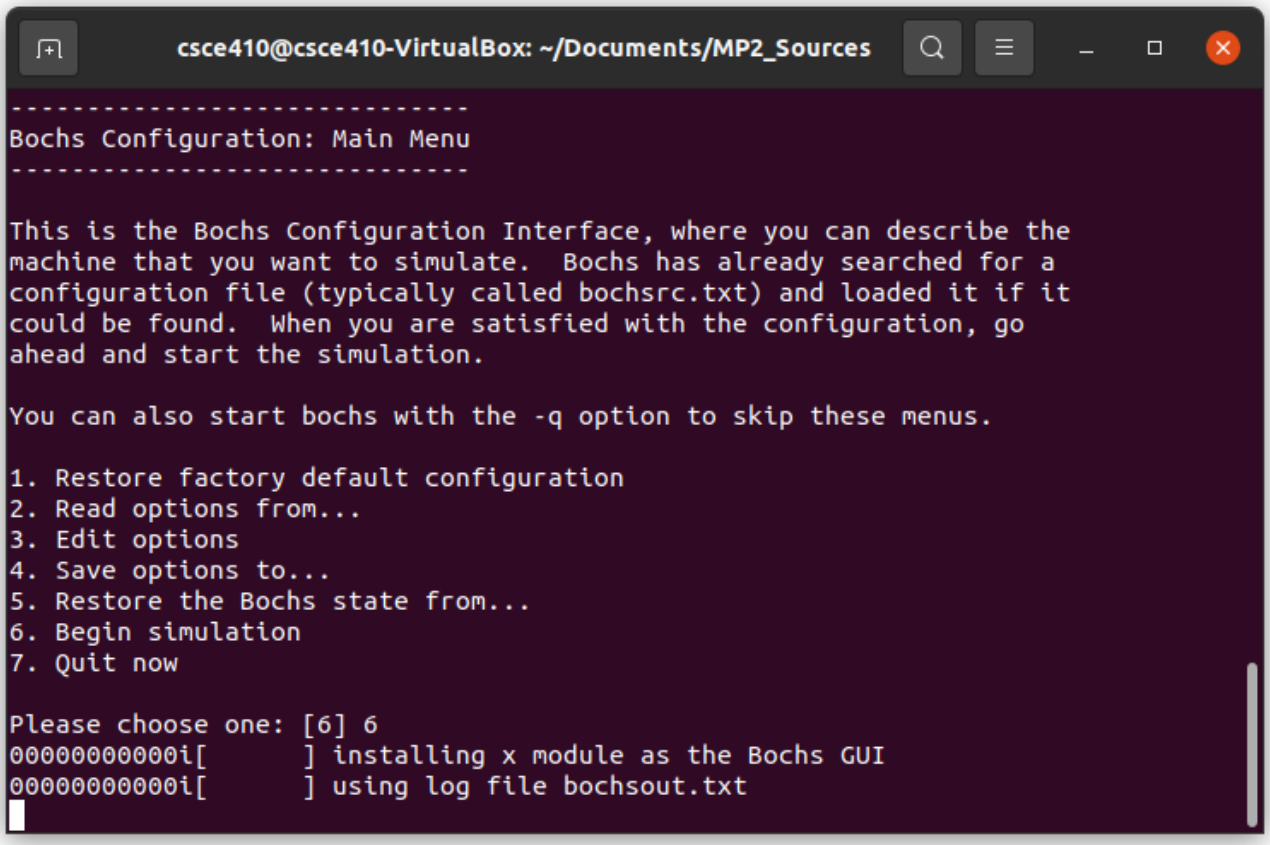
Byte_a & primary_mask = (11001101) & (11110011) = 11000001

Similar to the above instance, the state of all the frames (except 3) remained the same as it was before. Thus, this kind of bit manipulation can be performed to retain the states of all the other frames in the byte except for the one that we want to change.

Needed_info_frames method : In this method, we calculate the number of management frames that are needed for a certain memory. An example of these calculations can be seen in the beginning of this document.

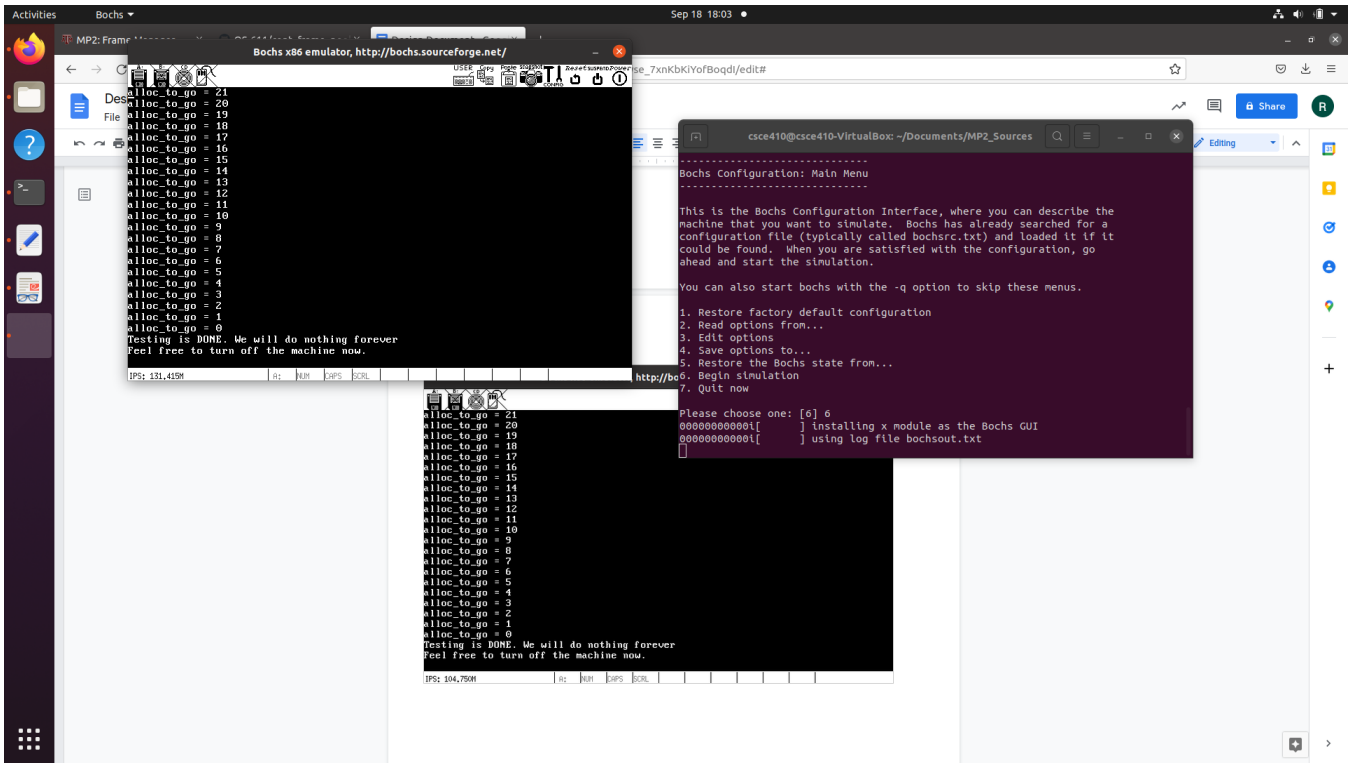
Attaching the screenshots of the output of a successful run :


Bochs x86 emulator, http://bochs.sourceforge.net/
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
SUSPEND
Power
alloc_to_go = 21
alloc_to_go = 20
alloc_to_go = 19
alloc_to_go = 18
alloc_to_go = 17
alloc_to_go = 16
alloc_to_go = 15
alloc_to_go = 14
alloc_to_go = 13
alloc_to_go = 12
alloc_to_go = 11
alloc_to_go = 10
alloc_to_go = 9
alloc_to_go = 8
alloc_to_go = 7
alloc_to_go = 6
alloc_to_go = 5
alloc_to_go = 4
alloc_to_go = 3
alloc_to_go = 2
alloc_to_go = 1
alloc_to_go = 0
Testing is DONE. We will do nothing forever
Feel free to turn off the machine now.
IPS: 104.750M
A:
NUM
CAPS
SCRL


csce410@csce410-VirtualBox: ~/Documents/MP2_Sources

Bochs Configuration: Main Menu

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate. Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found. When you are satisfied with the configuration, go
ahead and start the simulation.
You can also start bochs with the -q option to skip these menus.
1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now
Please choose one: [6] 6
00000000000i[ ] installing x module as the Bochs GUI
00000000000i[ ] using log file bochsout.txt


Activities
Bochs
Sep 18 18:03
Bochs x86 emulator, http://bochs.sourceforge.net/
alloc_to_go = 21
alloc_to_go = 20
alloc_to_go = 19
alloc_to_go = 18
alloc_to_go = 17
alloc_to_go = 16
alloc_to_go = 15
alloc_to_go = 14
alloc_to_go = 13
alloc_to_go = 12
alloc_to_go = 11
alloc_to_go = 10
alloc_to_go = 9
alloc_to_go = 8
alloc_to_go = 7
alloc_to_go = 6
alloc_to_go = 5
alloc_to_go = 4
alloc_to_go = 3
alloc_to_go = 2
alloc_to_go = 1
alloc_to_go = 0
Testing is DOME. We will do nothing forever
Feel free to turn off the machine now.
IPS: 131,415M
csce410@csce410-VirtualBox: ~/Documents/MP2_Sources
Bochs Configuration: Main Menu
This is the Bochs Configuration Interface, where you can describe the machine that you want to simulate. Bochs has already searched for a configuration file (typically called bochsrc.txt) and loaded it if it could be found. When you are satisfied with the configuration, go ahead and start the simulation.
You can also start bochs with the -q option to skip these menus.
1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now
Please choose one: [6] 6
00000000000i[ ] installing x module as the Bochs GUI
00000000000i[ ] using log file bochsout.txt
IPS: 104,750M
Share
Editing