MP2: Frame Manager

Dhanraj Murali
UIN: 734003894
CSCE611: Operating System

Assigned Tasks:

Main: Completed.

System Design:

1. Total memory size = 32MB
2. Memory reserved for kernel = 4MB
3. Memory partition to be used by kernel = 2MB to 4MB
4. Memory partition to be used by process = Above 4MB

Code Description:

I. Bitmap is maintained to effectively maintain the state of the frames in the given memory pool. Bitmap is char array with each element comprising of 8 bits. We use 2 bits to maintain the state of each frame, which means status of 4 frames in each byte.

II. We also maintain a linked list of memory pools allocated for each requesting process.

Files modified:

a. cont_frame_pool.H
b. cont_frame_pool.C
c. kernel.C – modified to test different sizes of frames. Reverted to original file.

All changes made as part of MP1 bonus to enable GDB integration in the following files are also included in this repository as well.

a. Makefile
b. linker.ld

Function Description:

a. get_state(unsigned long _frame_no)

Logic Used:

T`he input _frame_no is converted to the index that can be used to access the bitmap array by dividing it by 4. The index is then used to access the byte where the state of the frame is stored. The position of the frame information is calculated by taking the reminder of the above calculation. We use a mask of value 0x3 which is equivalent to 11 in binary. The mask is then shifted to match the position of the state information of the frame in the bitmap byte. The mask is then multiplied with the bitmap to remove the frame state information of the frames that are stored in the byte so we just have the state of the frame in interest. The manipulated byte with only the frame state information of one frame is then right shifted to move the frame state information to the LSB of the byte. It is then compared with the values of the frame states defined earlier – Free, Used and HoS, and the frame state is returned.

b. set_state(unsigned long _frame_no, FrameState _state)

Logic Used:

The input _frame_no is converted to the index that can be used to access the bitmap array by dividing it by 4. The index is then used to access the byte where the state of the frame is stored. We use the mask similar to get_state function and shift it to match the location of the

frame of interest. The input _state is then checked to see the state that is to be set in the frame state information bit. Then the bitmap byte is multiplied with the complement of the mask to change the state information of the frame in interest to 00. Then it is added with the desired frame state value as per the input _state.

c. get_frames(unsigned int _n_frames)

Logic Used:

We use 2 loops to iterate through the bitmap of the frame pool in interest to check if we have contiguous free frames to allocate. The first loop starts from frame 0 and starts to check if it is free, if a free frame is located in the frame pool then control enters the 2nd loop to start checking contiguous free frames of interested size. If no such contiguous locations are available, we just oprint that we don't have enough frames. If such contiguous free frames are present, then we set the new states of the frames, with HoS for the first frame and Used for the rest of the frames. Then we return the frame number of the first free frame of the sequence in its physical memory mapping by adding it to the base frame number of the pool.

d. mark_inaccessible(unsigned long _base_frame_no, unsigned long _n_frames)

Logic used:

We first check if the provided input _base_frame_no belongs to the frame pool of the current instantiation of the class. If it is part of the current frame pool then we mark the input _base_frame_no as HoS and rest of the frames in the mentioned limit _n_frames as Used so that they are inaccessible.

e. release_frames(unsigned long _first_frame_no)

Logic Used:

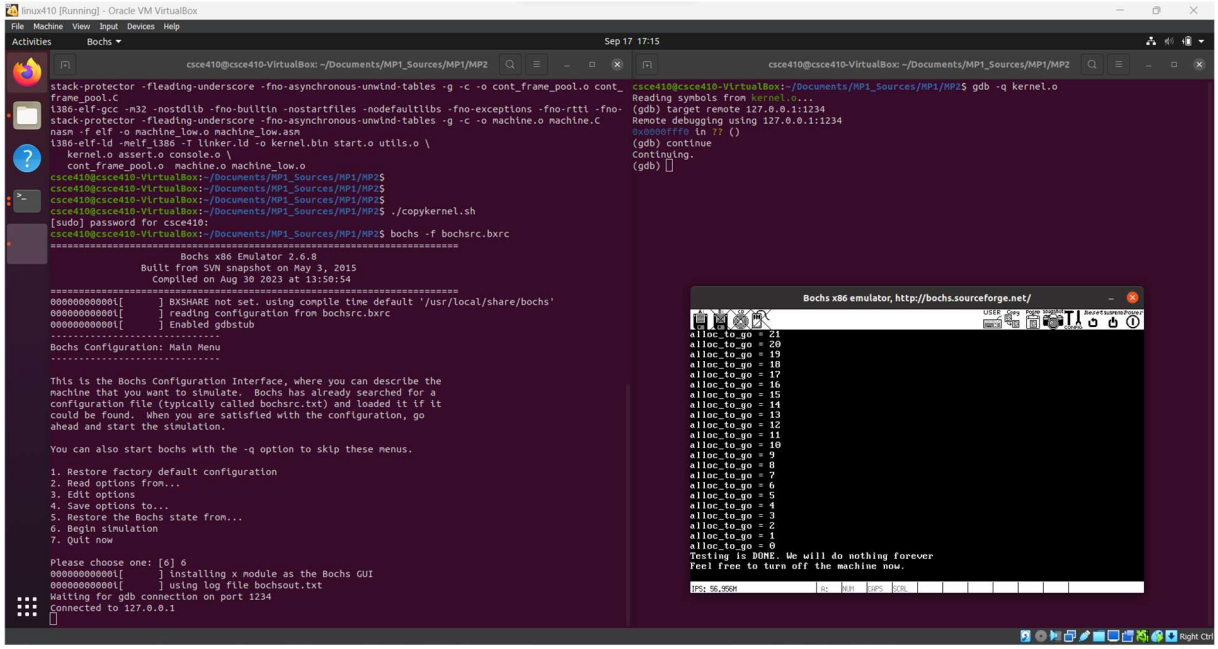
The function is static so we cant call the function using an instantiation i.e a specific memory pool of the class type. So we have to find the memory pool of interest meaning the memory pool in which the _first_frame_no is located. This is done by checking if the frame belongs to the memory pool of the first node in the linked list that is used to maintain the allocated pools. Once the memory pool of interest is found, then we start setting the state of the particular pool until we encounter a frame with a frame state or a frame with HoS state.

f. needed_info_frames(unsigned long _n_frames)

Logic Used:

Here, the info frames are nothing but the number of frames required to store the bitmap of frame states. The frame size is the same as the machine's page size and we also calculated earlier that for each frame we need 2 bytes of information. For example, if we take a frame size of 4 kilobytes which is equal to 4096bytes, we can store the frame state information of 4*4096 frames in 1 frame. So, for any process, the required no of frames is calculated by dividing the memory size by frame size. To find out the no of frames required to store the frame state information, simply divide the number of frames by 4*frame size. Suppose if there any remainder then add one more frame to the required info frames value to accommodate the frame state of the remainder frames.

Testing:


linux410 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Bochs
Sep 17 17:15
csce410@csce410-VirtualBox: ~/Documents/MP1_Sources/MP1/MP2
stack-protector -fleading-underscore -fno-asynchronous-unwind-tables -g -c -o cont_frame_pool.o cont_frame_pool.C
i386-elf-gcc -m32 -nostdlib -fno-builtin -nostartfiles -nodefaultlibs -fno-exceptions -fno-rtti -fno-stack-protector -fleading-underscore -fno-asynchronous-unwind-tables -g -c -o machine.o machine.C
nasm -f elf -o machine_low.o machine_low.asm
i386-elf-ld -melf_i386 -T linker.ld -o kernel.bin start.o utils.o \
kernel.o assert.o console.o \
cont_frame_pool.o machine.o machine_low.o
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$ ./copykernel.sh
[sudo] password for csce410:
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$ bochs -f bochsrc.bxrc
Bochs x86 Emulator 2.6.8
Built from SVN snapshot on May 3, 2015
Compiled on Aug 30 2023 at 13:50:54
00000000000i[ ] BXSHARE not set. using compile time default '/usr/local/share/bochs'
00000000000i[ ] reading configuration from bochsrc.bxrc
00000000000i[ ] Enabled gdbstub
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
Waiting for gdb connection on port 1234
Connected to 127.0.0.1
csce410@csce410-VirtualBox:~/Documents/MP1_Sources/MP1/MP2$ gdb -q kernel.o
Reading symbols from kernel.o...
(gdb) target remote 127.0.0.1:1234
Remote debugging using 127.0.0.1:1234
0x0000fff0 in ?? ()
(gdb) continue
Continuing.
(gdb)
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 DONE. We will do nothing forever
Feel free to turn off the machine now.
IPS: 56.956M
Right Ctrl