Member:

1. Busheng Lou A53080746
2. Tianqi Chen A53079580
3. Ziqian Xu A53081116
4. Kaiyue Yang A53081576

Part 1:

We created a new method: DemandSpace(); And move most code of the Initialize() to this method. When we initialize a addrspace, we will set the valid bit to false, then the page fault will occur when first load the page, however, we donot set the physicalPage of the pageTable in the Initialize(), we set it in the DemandSpace for each bad vpn. We then calculate the total page numbers of code and data, compare these with badvpn, we will know whether the badvpn is belong to code page or data page or stack page, for different page, we load different part or zero the page for stack.

We also created a new method markPage() in addrspace, we use this method to set valid bit true when necessary. When page fault happens, it will call the PageFault_Handler to fix it, we get badvpn from the register, then use badvpn to decide which page to load, if load failed, it will return false.

Part 2:

For the part 2, we need to consider about the page replacement. First, we create a BackingStore class in the addrspace.h, the input value is addrspace. Every space has a backingstore. The class has two methods, pageOut and pageIn. PageOut is used to store the page which will be replaced and it has been modified, PageIn is used to load page when the page is stored in backing store. The backing store is a file, we can read and write to the position we want with ReadAt and WriteAt. We add a bit in pageTable to show whether the page be stored. If the page is clean, then just free the page, next time, we need the page, we can reload from the executable; if else, we need to load from the store. We need to consider which one to evict, this is due to the replacement algorithm. In this part, we have choose two algorithms: Random and FIFO. (See more in Part 5 about how we implement different algorithm) We use random() in sysdep.cc to get a random number, divide by NumPhysPages, and the remainder will be the phys number of the page we want tot evict. FIFO, we use the list, when need to evict

one page, we always evict the first one of the list, new page will be put in the end of the list. Also, we will set the dirty bit and valid bit to FALSE. Then, when we call the PageFault_Handler, we can find there is a page can be used, we can load the code, data or others to the spare page.

Part 3:
In this part, we write five test programs to test the function of demanding paging and replacement policy based on the locality.

Allref.c: test the case that reference all the pages in memory by referencing all elements in an array sequentially.
Partref.c: test the case that reference only part of the pages in memory by referencing only the first element in an array sequentially.
GoodLocality.c: repeatedly reference a sequence of elements in the array;
PoorLocality.c: repeatedly reference a set of elements that allocated in different pages in the array;
RandLocality.c: randomly choose some elements in the array.

Physical memory size: 32 pages.
Page replacement policy: FIFO.

Program	PageFaults	PageOuts	PageIns
Allref	7	0	6
Partref	6	0	5
GoodLocality	94	55	15
PoorLocality	101	61	22
RandLocality	97	58	18

Physical memory size: 32 pages.
Page replacement policy: Random.

Program	PageFaults	PageOuts	PageIns
Allref	7	0	6
Partref	6	0	5
GoodLocality	90	54	11

PoorLocality	99	60	20
RandLocality	95	58	16

Physical memory size: 32 pages.
Page replacement policy: LRU.

Program	PageFaults	PageOuts	PageIns
Allref	7	0	6
Partref	6	0	5
GoodLocality	94	55	15
PoorLocality	103	61	24
RandLocality	98	58	19

Part 4:
The three numbers: numPageOuts, numPageIns and numPageFaults are global variables. When we start a new process, it will be set to zero. When page fault happens, increase the numPageFaults. When we put the page to the backing store, numPageOuts will increase. The PageIn is a bit different, both we read page from the executable file or the backing store, the numPageIns will increase.

Physical memory size: 32 pages.
Page replacement policy: Random.

Program	PageFaults	PageOuts	PageIns
halt	3	0	2
matmult	99	40	61
sort	956	838	923
releasememory	23	0	17

Physical memory size: 32 pages.
Page replacement policy: FIFO.

Program	PageFaults	PageOuts	PageIns
halt	3	0	
2			
matmult	107	47	69
sort	4832	4361	4799
releasememory	23	0	17

Physical memory size: 32 pages.
Page replacement policy: LRU.

Program	PageFaults	PageOuts	PageIns
halt	3	0	
2			
matmult	108	48	70
sort	1919	1481	1886
releasememory	23	0	17

Part 5;

We implement a class named MemoryStore. Inside MemoryStore, an array named VPN is created and each VPN maps to one page of physical page. Everytime when we allocate a physcial page, the pointer of TranslationEntry and BackingStore will be stored in the mapping VPN. And three different method ReturnPTE_FIFO, ReturnPTE_Rand and ReturnPTE_LRU are created in the MemoryStore. FIFO used a list to implement, Rand used a random method to evict a page to implement, and LRU using the counter inside each VPN to implement.

Also we use a new command line switch to main.cc, with the help of a new argument “a” to invoke nachos with different replacement algorithm as “./nachos -x ../test/.. a”. If a = 1, the replacement is random, if a = 2, the replacement is FIFO, if a = 3, the replacement is LRU, other number will invoke a segment fault to indicate that the replacement instruction is not available for the operating system.

Random	FIFO	LRU
./nachos -x 1 ../test/mytest1 default	./nachos -x 2 ../test/mytest1	./nachos -x 3 ../test/mytest1

The worse case, PoorLocality, when there are small pages, and will always replace the page. For example, PoorLoacality.

Summary()

Everyone in our group contributed to the project. Thank you for grading our project!!!

and MERRY CHRISTMAS! (may too early :))