OS Design Project 3

Alan Luo and Patrick Meng

afl59 and pm708

Functions

SetPhysicalMem:
This function is invoked once in order to initialize all the internal data structures.

First, we calculate the internal parameters based on PGSIZE, MEMSIZE, and MAX_MEMSIZE. The number of entries in the bitmaps and page tables are dynamically calculated based on the 3 parameters defined in the header file.

// Bits for offset within a page
num_of_bits for page offset = $\log_2$(PGSIZE)

// To fit one 2nd-level page table into a single page based on the page size and page entry size
num_of_bits for page index = $\log_2$(PGSIZE / sizeof(pte_t))

// Rest of the bits are reserved for the page dir table index
num_of_bits for dir index = 32 - (num_of_bits for page offset) - (num_of_bits for page index)

We then initialize our various internal data structures:

Physical mem = malloc(MEMSIZE)

Virtual page bitmap (to track usage status of virtual mem space) =
malloc(MAX_MEMSIZE / PGSIZE), we use one byte for each virtual page

Physical page bitmap (to track usage status of physical mem space) =
malloc(MEMSIZE / PGSIZE), we use one byte for each physical page

Directory table (the 1st level page table) =
malloc(sizeof(pde_t * (1 << (num_of_bits for dir index))))

We do NOT allocate the 2nd level page table until it is needed during translation

Translate:
We first check if the virtual page is in the TLB. If so, we retrieve the physical address of the virtual page and combine it with the page offset to get our physical address.

Otherwise, if it isn't in the TLB, we walk through the 2nd-level page tables. We first use bit shift operations on the virtual address to get the index to the dir table (1st-level page table). If the entry of the dir table is NULL, we allocate memory to create a 2nd-level page table.

We then use bit shift and mask operations on the virtual address to get the index for the 2nd-level page table. If the entry on the 2nd-level page table is NULL, we map a free physical page to it. If there is no free physical page, we simply abort, since the expected behavior of this project is that we shouldn't run out of physical memory. The bitmap for the physical pages is then updated to reflect the usage status of that specific page.

Now we have our physical address for the virtual page, and add it into our TLB.

We then combine the physical address of the virtual page and the page offset to return the complete physical address for the requested virtual address.

PageMap:
We add the physical page address into the page table.

check_in_tlb:
We go through the TLB entries to find if there is an entry whose virtual page index matches with the requested virtual address's page index.

We have an internal function that actually returns the physical address of the matching virtual page, rather than the boolean value.

put_in_tlb:
We put the <physical addr, virtual page index> tuple into the TLB using the LRU algorithm, if there is no free TLB entry.

myalloc:
We go through the virtual memory bitmap to look for a contiguous address space large enough for the requested size. The bitmap is then used to track if a virtual page is in use or not. We mark the first allocated page, last allocated page, and the pages in between separately for later validation for the myfree function.

myfree:

We go through the virtual memory bitmap to validate if the virtual address was previously allocated and the size of the block matches. After validation, we free the physical page by clearing its bitmap, NULL out the page table entry, NULL out the TLB entry, then clear the virtual bitmap.

PutVal:
We first calculate the number of bytes we need to copy for the 1st virtual page by performing (PGSIZE - virtual address page offset). Then, we figure out how many full virtual pages we need to copy. Finally, we determine how many bytes we need to copy for the last virtual page. Then, we use the translate function to obtain a physical address for each virtual page and use memcpy to move the data over.

GetVal:
This function operates the same way as PutVal except we swap the src and dest for the memcpy invocation.

print_TLB_missrate():
We keep track of the counts of TLB lookups and misses. Then, we print the miss rate = missed/total

MatMult:
We use a 3-layered loop (i, j, k from 1 to size) to perform matrix multiplication:

answer[i][j] += mat1[i][k] = mat2[k][j], for all k from 1 to size

In code, it utilizes Getval to get mat1[i][k] and mat2[k][j] and calculate the result.

```
    GetVal(mat1+(i*size*sizeof(int))+k*sizeof(int), &x, sizeof(int));
    GetVal(mat2+(k*size*sizeof(int))+j*sizeof(int), &y, sizeof(int));
    temp += x * y;
```

Then, we use PutVal to save the result into answer[i][j]

Benchmark, TLB Miss Rate, and Runtime Improvements

The benchmark used was the given test.c file modified to print the TLB miss rate at certain points.

1. **With** TLB

Command	Matrix Size	Miss Rate	Avg. Exec. Time (nanoseconds)
benchmark/test 5	5x5	0.006	403080
benchmark/test 10	10x10	0.000857	1108359
benchmark/test 25	25x25	0.00006	8080621
benchmark/test 50	50x50	0.000023	35533567
benchmark/test 100	100x100	0.00001	220841000

2. **Without** TLB

Command	Matrix Size	Miss Rate	Avg. Exec. Time (nanoseconds)
benchmark/test 5	5x5	N/A	439046
benchmark/test 10	10x10	N/A	949160
benchmark/test 25	25x25	N/A	7560609
benchmark/test 50	50x50	N/A	32650127
benchmark/test 100	100x100	N/A	169138796

In theory, the TLB can improve the performance by reducing the memory access involved in the walkthrough of the 2nd-level page tables, especially if the TLB is stored in the hardware supported associated cache within the CPU.

In this implementation of TLB, the main memory was used to store TLB entries and the search of TLB is done linearly, since there was no access to hardware supported associated cache for parallel search. We then arrive at the following conclusions:

- When the number of entries in the TLB is small, the cost of linear search is comparable to the cost of the walkthrough of 2nd-level page tables.
- But when the number of entries in the TLB increased, the cost of linear search through entries in TLB increased more than the cost of the walkthrough of the 2nd-level page

tables. TLB started hurting the performance instead of helping. This was also demonstrated in the data above, where after matrix size >= 50. This is likely ONLY because of the simulation of TLB in main memory.

Support for different page sizes

All of the bit shifts and bit masks are calculated based on the page size instead of being hard coded. This was implemented in the initialization function. This means that since the values are calculated dynamically, we can support any defined page size within the header.

Possible Issues

There is no support for swapping out the pages to disks when physical memory is fully used. This means that for high memory load operations, physical memory will run out. We can fix this by implementing a swapping functionality within our assignment and get_next_avail functions.

Most Difficult Part

The most difficult parts of solving this project were the bit shifting and masking, since they were error-prone and were difficult to debug.