CS416 Project 3

Christopher Yong (cy287), Joshua Ross (jjr276)

1 Logic

1.1 2-level implementation

set_physical_mem()

Step 1) Calloc the MEMSIZE and set the return pointer to the a global variable called physicalMemoryBase

Step 2) Check if the physicalMemoryBase is still NULL

- Step 2a If physical Memory Base is NULL, exit(-1) because that means we could not even calloc our "physical memory"
- Step 2b If physicalMemoryBase is not null, calloc the physical bit map via $(unsignedlong)(ceil(\frac{MEMSIZE}{(PGSIZE)*8.0})$ (also check if it not NULL) which is the number of chars to represent all pages and set the returning pointer to a global variable called physicalBitMap. If the number of pages is not a multiple of 8, set the physical pages that should not exist in the last char to 1 to indicate it should not be used to retrieve free pages.

Step 3) Calloc the virtual memory bitmap via $(unsignedlong)(ceil(\frac{MAX\_MEMSIZE}{(PGSIZE)*8.0})$ and set the return pointer to a global variable called virtualBitMap.

- Step 3a) if the virtualBitMap is NULL, exit(-1) because we could not calloc our virtual bit map.
- Step 3b) if the virtualBitMap is not null, If the number of pages is not a multiple of 8, set the virtual pages that should not exist in the last char to 1 to indicate it should not be used to retrieve free pages.

Step 4) set the first virtual page to 1 to indicate it is used and cannot be allocated . (we do this because we want the user to be able to tell if a_malloc returns NULL or 0x0, it means it could not find a free page and the malloc failed otherwise if we allowed the 1st page to be allocated or 0x0, then the user cannot tell when a_malloc returns, if failed or if 0x0 or NULL is the starting address and it starts at the first page.)

translate(pde_t* pgdir, void* va)

Step 1) Increment the global TLB miss variable by one

Step 2) Calculate the number of bits for the inner page tables and the page directory. This done by

- Step 1a) calculate virtualPageBits via (sizeof(void*) * 8) - log2(PGSIZE)
- Step 1b) calculate pageTableBits via log2(PGSIZE / sizeof(void*))
- Step 1c) calculate pageTableLevels via (sizeof(void*) * 8) / 16.0
- Step 1d) check if pageTableLevels * pageTableBits > virtualPageBits
  - a) if virtualPageBits $\leq$ pageTableBits or (pageTableBits * (pageTableLevels - 1) $\geq$ virtualPageBits) then set pageTableBits to ceil(virtualPageBits / ((double)pageTableLevels)))
  - b) if the two conditions is false, continue to next step
- Step 1e) while pageTableLevels > 1, decrease virtualPageBits by pageTableBits and decrease pageTableLevels by 1 every iteration (when the loop exits, the virtualPageBits will be the bits for the page directory)
- Step 1f) reset pageTableLevels back to (sizeof(void*) * 8) / 16.0

Step 3) Go through each level of multilevel page table and access the index in each level till get the physical address and if one of the index is NULL, return NULL because it means it has not been mapped and it is an error. This is done by

a) set nextAddress to pgdir

b) set usedTopBits to virtualPageBits

c) set the pageTableMask to (ULONG_MAX >> ((sizeof(void*) * 8) - usedTopBits)) << ((sizeof(void*) * 8) - usedTopBits)

d) while pageTableLevels not equal to 0

- Step 1) Find the index to access in the current page table by masking the page table bits with the given virtual address
  - A. pageTableIndex = ((unsigned long)va) & pageTableMask
  - B. pageTableIndex >>= (sizeof(void*) * 8) - usedTopBits
  - C. nextAddress = (unsigned long*) *(nextAddress + pageTableIndex)
  - D. if nextAddress is NULL, return NULL (This is an invalid translate)
  - E. decrease pageTableLevel by one
- Step 2) Create the new page table mask for the next page table level
  - A. create an inverted mask to zero out the top used bits via inverseMask ~= pageTableMask

B. usedTopBits += pageTableBits

C. pageTableMask = (ULONG_MAX >> ((sizeof(void*) * 8) - usedTopBits) << ((sizeof(void*) * 8) - usedTopBits)

D. pageTableMask &= inverseMask

Step 4) Retrieve the offset from the VA and add it to the nextAddress and return the value (at this point, nextAddress should be the physical address associated with the virtual address given)

page_map(pde_t* pgdir, void* va, void* pa)

This is implemented like translate except instead of returning NULL if nextAddress is NULL, it allocates another page table and when page table level == 0 or if it's the last page table that should contain the physical address of the virtual address, then it will overwrite the entry in that page table with the given pa.

Step 1) Calculate the number of bits for the inner page tables and the page directory. This done by

Step 1a) calculate virtualPageBits via (sizeof(void*) * 8) - log2(PGSIZE)

Step 1b) calculate pageTableBits via log2(PGSIZE / sizeof(void*))

Step 1c) calculate pageTableLevels via (sizeof(void*) * 8) / 16.0

Step 1d) check if pageTableLevels * pageTableBits > virtualPageBits

a) if virtualPageBits $\leq$ pageTableBits or (pageTableBits * (pageTableLevels - 1) $\geq$ virtualPageBits) then set pageTableBits to ceil(virtualPageBits / ((double)pageTableLevels)))

b) if the two conditions is false, continue to next step

Step 1e) while pageTableLevels > 1, decrease virtualPageBits by pageTableBits and decrease pageTableLevels by 1 every iteration (when the loop exits, the virtualPageBits will be the bits for the page directory)

Step 1f) reset pageTableLevels back to (sizeof(void*) * 8) / 16.0

Step 2) Go through each level of multilevel page table and access the index in each level till get the physical address and if one of the index is NULL, return NULL because it means it has not been mapped. This is done by

a) set a nextAddress to pgdir

b) set usedTopBits to virtualPageBits

c) set the pageTableMask to (ULONG_MAX >> ((sizeof(void*) * 8) - usedTopBits)) << ((sizeof(void*) * 8) - usedTopBits)

d) while pageTableLevels not equal to 0

Step 1) Find the index to access in the current page table by masking the page table bits with the given virtual address

Step 1a) decrease pageTableLevel by one

Step 1b) pageTableIndex = ((unsigned long)va) & pageTableMask

Step 1c) pageTableIndex >>= (sizeof(void*) * 8) - usedTopBits

Step 1d) holdNextAddress = (nextAddress + pageTableIndex);

Step 1e) nextAddress = (unsigned long*) *(nextAddress + pageTableIndex)

Step 1f) if nextAddress is NULL or pageTableLevels == 0

- if page table level is 0, set the *(holdNextAddress) to the given PA and return 1
- if page table is not 0, allocate a new page table via allocating a new physical page and setting new address to equal the new physical page address via *((unsigned long*) holdNextAddress) = ((unsigned long) physicalPageAddress) where physicalPageAddress is the new physical page address and set nextAddress = physicalPageAddress and also zero out the new physical page and set the physical bit map to say the physical page is allocated. (Also if physicalPageAddress is NULL, return -1 and unallocate all the physical pages allocated for this mapping)

Step 2) Create the new page table mask for the next page table level

A. create an inverted mask to zero out the top used bits via inverseMask ∼= pageTableMask

B. usedTopBits += pageTableBits

C. pageTableMask = (ULONG_MAX >> ((sizeof(void*) * 8) - usedTopBits) << ((sizeof(void*) * 8) - usedTopBits)

D. pageTableMask &= inverseMask

get_next_avail(int num_pages)

Find a spot in the virtual bit map where there is num_pages continous free pages.

1. for each char, check if the char has a free page in it via 0xFF mask and see if does not equal 0xFF. (that means one of the bits in the char is 0 or there is a free page)

2. for each bit index of a char, check if it is free and if it is free, increment the number of continuous pages found and store the beginning of the free continuous pages.

3. if a bit index is 1 or allocated and the number of continuous pages found is not equal to num_pages, reset the variable that holds the beginning of the free continuous pages and reset the number of continuous pages found.

4. once the number of continuous pages found is equal to num_pages then return the starting address of the beginning of the free continous pages.

5. if could not find a spot in the virtual page map for the number of pages, return NULL.

get_next_physicalavail()

Find a free page in the physical bit map.

1. for each char in the physical bit map, check if the char has a free page in it via 0xFF mask and see if it does not equal 0xFF. (that means one of the bits in the char is 0 or there is a free page)

2. for each bit index of the char, check if it is free and if it is freed, return the physical address of that page via physicalMemoryBase + ((bitIndex + (charIndex * 8)) * PGSIZE). Note the physicalMemoryBase is a char* so we dont need to explicitly cast it.

add_TLB(void* va, void* pa)

Our TLB struct contains a void* physicalPageAddress, unsigned long virtualPageNumber, and a char that represents the metadata where the first bit in the char is the valid bit.

Step 1) extract the virtual page number from the va

Step 2) get the tlb index via virtual page number % TLB_ENTRIES

Step 3) retrieve the tlb entry via (tlbBaseAddress + tlbIndex) (Note tlbBaseAddress is a global variable that represents the base address of the TLB node array)

Step 4) set the tlb entry's virtual page number = virtual page number

Step 5) remove the pa offset by converting it to the physical page number and then reconverting it to the physical address (so the pa will always be in the start of the physical page)

Step 6) set the tlb entry's physicalPageAddress = pa

Step 7) set the tlb entry's valid bit to 1

check_TLB(void* va)

Our TLB struct contains a void* physicalPageAddress, unsigned long virtualPageNumber, and a char that represents the metadata where the first bit in the char is the valid bit.

Step 1) extract the virtual page number from the VA

Step 2) get the tlb index via virtual page number % TLB_ENTRIES

Step 3) retrieve the tlb entry via (tlbBaseAddress + tlbIndex) (Note tlbBaseAddress is a global variable that represents the base address of the TLB node array)

Step 4) check if the tlb entry's valid bit is 1 and the tlb entry's virtual page number == virtual page number

a) if both conditions are true, increment global TLB HIT variable by one and extract the offset from the passed in va and return the tlb entry's physical address + the offset.

b) if one of the conditions fail, continue to next step.

Step 5) return NULL

remove_TLB(void* va)

Our TLB struct contains a void* physicalPageAddress, unsigned long virtualPageNumber, and a char that represents the metadata where the first bit in the char is the valid bit.

Step 1) extract the virtual page number from the VA

Step 2) get the tlb index via virtual page number % TLB_ENTRIES

Step 3) retrieve the tlb entry via (tlbBaseAddress + tlbIndex) (Note tlbBaseAddress is a global variable that represents the base address of the TLB node array)

Step 4) check if the tlb entry's valid bit is 1 and the tlb entry's virtual page number == virtual page number

a) if both conditions are true, set the tlb entry's valid bit to 0.

print_TLB_missrate()

Step 1) initialize a double variable called miss_rate to 0

Step 2) if global tlbHit and tlbMiss != 0 then, miss_rate = ((double)tlbMiss) / (tlbHit + tlbMiss)

Step 3) fprintf the number of misses and hits

Step 4) fprintf the miss rate to .17 decimal for the most precision.

a_malloc(unsigned int num_bytes)

Step 1) lock the global mutex

Step 2) if physicalMemoryBase is NULL, call set_physical_mem() and createTLB to initialize the virtual and physical bit maps and the physical memory and the TLB.

Step 3) if the pageDirectoryBase is NULL, calculate the number of bits to represent the pageDirectory (see translate or page map) and calculate the number of continuous Physical Pages to allocate the page directory via (unsigned long) ceil(((1 << page directory bits) * sizeof(void*)) / ((double)PGSIZE)) or (# of entries in page directory * size of entry in bytes) / page size in bytes and set the pageDirectoryBase to the starting physical pages that are continuous and set the physical bit map to reflect the allocated physical pages for the page directory (note at this point, there should not be any physical pages allocated therefore we can do a continous physical page allocation easily by calling get_next_physicalavail() and updating the physical bit map and calling it again till we reach the N continuous physical pages needed)

Step 4) afterwards, calculate the number of pages required to allocate for the num_bytes via ceil((double)num_bytes)/ PGSIZE)

Step 5) call get_next_avail with the number of pages required for the allocation and set the return pointer to startingVirtualPageAddress

Step 6) if startingVirtualPageAddress is NULL, unlock the global mutex and return NULL otherwise continue to step 7

Step 7) for the number of virtual pages required, find if there are enough physical pages and if there is not enough physical pages, unlock the global mutex and return NULL and update the virtual and physical bit maps to reflect the pages are free still and not allocated.

Step 8) If we found enough physical pages and a place for the continuous virtual pages, map each one and ensure the physical and virtual bit maps reflect these pages are allocated and also zero out the physical pages.

Step 9) unlock the global mutex and return the startingVirtualPageAddress

a_free(void* va, int size)

In our implementation, we are doing a lazy free where we only set the virtual and physical bit maps to reflect they are no longer allocated

Step 1) if va is NULL or if the va + size is greater than maximum virtual address or ULONG_MAX, then return

Step 2) calculate the number of pages that need to be freed from the size via ceil((double) size / PGSIZE)

Step 3) extract the virtual page number from the va

Step 4) lock the global mutex

Step 5) check if the virtual pages are actually allocated in the virtual bit map and if there are not, unlock the global mutex and return

Step 6) for each page

(a) get the physical address via the check_TLB (and remove from the TLB) and if the TLB does not have it, retrieve it via translate.

(b) update the physical and virtual bit maps to reflect the physical page and the virtual page is now free or 0 in the bit map.

Step 7) unlock the global mutex

put_value(void* va, void* val, int size)

Step 1) Retrieve the offset

Step 2) check if the size > (PGSIZE - offset) or the number of pages we have to access more than the given va

Step 3) set the number of pages to access beyond this first page via remaining-Size = size - (PGSIZE - offset) and then numberOfPages = ceil(((double)remainingSize) / PGSIZE)

Step 4) extract the virtual page number from the given va

Step 5) set copied to 0

Step 6) set copy to, if size > (PGSIZE - offset) then PGSIZE - offset otherwise size (basically if the size is greater than the available room in the 1st page, set the size equal to the available room in the 1st page otherwise set it to the size)

Step 7) lock the global mutex

Step 8) check if the pages that we have to access (including the given va), are allocated and if there are not, unlock the global mutex and return

Step 9) grab the physical address of the given va from the TLB and if not from the TLB, get it from translate and store the physical address and the given va to the TLB

Step 10) memcpy where the destination is the physical address and the source is the)((char*)val) + copied and and the number of bytes to copy is sizeof(char) * copy

Step 11) update the size to size -= copy, and the copied to sizeof(char) * copy and copy to if size < PGSIZE then copy = size otherwise copy = PGSIZE.

Step 12) get the virtual Page number of the VA and add 1 (because we are done filling the 1st page with all the bytes it could store and now we have to fill the next N pages with the size remaining)

Step 13) for each page that is less than number of pages to allocate and size > 0

(a) get the starting virtual address of the given virtual page number

(b) get the physical address from the TLB and if it is not in the TLB, use translate and store the retrieved physical address and the starting virtual address in the TLB

(c) memcpy where the destination is the physical address and the source is ((char*) val) + copied, and the number of bytes is sizeof(char) * copy

(d) update the size to be size -= copy and copied to be copied += sizeof(char) * copy and the copy to be if size < PGSIZE, size otherwise PGSIZE.

(e) virtual page number increment by 1

Step 14) unlock the global mutex

get_value(void* va, void* val, int size)

Step 1) Retrieve the offset

Step 2) check if the size > (PGSIZE - offset) or the number of pages we have to access more than the given va

Step 3) set the number of pages to access beyond this first page via remaining-Size = size - (PGSIZE - offset) and then numberOfPages = ceil(((double)remainingSize) / PGSIZE)

Step 4) extract the virtual page number from the given va

Step 5) set copied to 0

Step 6) set copy to, if size > (PGSIZE - offset) then PGSIZE - offset otherwise size (basically if the size is greater than the available room in the 1st page, set the size equal to the available room in the 1st page otherwise set it to the size)

Step 7) lock the global mutex

Step 8) check if the pages that we have to access (including the given va), are allocated and if there are not, unlock the global mutex and return

Step 9) grab the physical address of the given va from the TLB and if not from the TLB, get it from translate and store the physical address and the given va to the TLB

Step 10) memcpy where the destination is ((char*) val) + copied and the source is physical address, and the number of bytes is sizeof(char) * copy

Step 11) update the size to size -= copy, and the copied to sizeof(char) * copy and copy to if size < PGSIZE then copy = size otherwise copy = PGSIZE.

Step 12) get the virtual Page number of the VA and add 1 (because we are done filling the 1st page with all the bytes it could store and now we have to fill the next N pages with the size remaining)

Step 13) for each page that is less than number of pages to allocate and size > 0

(a) get the starting virtual address of the given virtual page number

(b) get the physical address from the TLB and if it is not in the TLB, use translate and store the retrieved physical address and the starting virtual address in the TLB

(c) memcpy where the destination is ((char*) val) + copied and the source is physical address, and the number of bytes is sizeof(char) * copy

(d) update the size to be size -= copy and copied to be copied += sizeof(char) * copy and the copy to be if size < PGSIZE, size otherwise PGSIZE.

(e) virtual page number increment by 1

Step 14) unlock the global mutex

mat_mult(void* mat1, void* mat2, int size, void* answer)

Step 1) Create a for loop called i where i = 0 and loops till SIZE

Step 2) create a for loop within i loop called j where j = 0 and loops till SIZE

Step 3) within the j for loop, create a temp variable called z that and set to 0

Step 4) create a for loop within the j loop called temp that starts at 0 and loops till SIZE

Step 5) within the temp loop, calculate the address_a and address_b via ((unsigned long) mat1) + ((i * size * sizeof(int)) + ((temp * sizeof(int)) and ((unsigned long)mat2) + (temp * size * sizeof(int)) + (j * sizeof(int))

Step 6) get_value for address a and address b and store it into some variable called x and y (note x and y will be initialized to 0 beforehand)

Step 7) set z = z + (x * y)

Step 8) once it is out of the temp loop, calculate address_c via ((unsigned long) answer) + ((i * size * sizeof(int)) + (j * sizeof(int))

Step 9) put the z value into address_c

Step 10) Repeat Steps 4 to 9 for j * i times

2 Benchmarks

2.1 32-bit

test.c:

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
Storing integers to generate a SIZExSIZE matrix
Fetching matrix elements stored in the arrays
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplication with itself!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Freeing the allocations!
Checking if allocations were freed!
free function works
```

TLB Misses: 3
TLB Hits: 400
TLB miss rate: 0.74441687344913154%

multitest.c:

```
Allocated Pointers:
1000 3000 19000 7000 17000 5000 d000 1d000 11000 15000 9000 b000 1b000 f000 13000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplications in multiple threads threads!
Randomly checking a thread allocation to see if everything worked correctly!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Gonna free everything in multiple threads!
Free Worked!
```

TLB Misses: 30

TLB Hits: 1800
TLB miss rate: 1.639344262295082%

2.2 64-bit

test.c:

```
Allocating three arrays of 400 bytes
Addresses of the allocations: 1000, 2000, 3000
Storing integers to generate a SIZExSIZE matrix
Fetching matrix elements stored in the arrays
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplication with itself!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Freeing the allocations!
Checking if allocations were freed!
free function works
```

TLB Misses: 3
TLB Hits: 400
TLB miss rate: 0.74441687344913154%

multitest.c:

```
Allocated Pointers:
1000 3000 19000 7000 17000 5000 d000 1d000 11000 15000 9000 b000 1b000 f000 13000
initializing some of the memory by in multiple threads
Randomly checking a thread allocation to see if everything worked correctly!
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
Performing matrix multiplications in multiple threads threads!
Randomly checking a thread allocation to see if everything worked correctly!
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
5 5 5 5 5
Gonna free everything in multiple threads!
Free Worked!
```

TLB Misses: 30
TLB Hits: 1800
TLB miss rate: 1.639344262295082%

TLB miss rate aligns with the calculation using the following formula:
ceiling(Allocated bytes / Page size) * number of threads

Since the actual used data in the tests will always take up less pages than can fit in the TLB, there will be no other misses (since no TLB entry will get kicked out). Each table will have to be accessed at least once to be freed, so the misses should equal our calculated misses

3 Support for Page Sizes

Multiples of 4K (up to 128K) work and provide similar output in the benchmarks to what is shown

4 Possible Issues

N/A

5 Extra Credit

For our 4 level page table, we design it so the inner pages will always fit within a page and the page directory or the 1st level page table will take up all the extra bits. To do this, we calculate the number of bits that we have for all our page tables via address space bits - offset bits and called this our virtual page table bits. Then we calculated the number of entries a page can hold via PGSIZE / sizeof(void*) or PGSIZE / size of entries and then took the log2 of that to find the number of bits to represent the number of entries we could hold and called this page table bits. We then checked if page table bits * number of page table levels (calculated by address space bits / 16) $\geq$ virtual page table bits and if so, then we checked if the virtual page table bits was $\leq$ page table bits or page table bits * (page table levels - 1) $\geq$ virtual page table bits and if any of those two conditions were true, we set the page table bits equal to ceil(virtual page table bits / page table levels). Afterwards, we created a while loop that looped till the page table level was 1 and subtracted the virtual page bits by page table bits and decremented the virtual page tables to find the number of bits required for the 1st or page directory page table. (This is shown in translate, map, and a_malloc (for our page directory allocation)). Therefore our design is dynamic and supports any page sizes that are a multiple of power of 2.