Betriebssysteme Assignment 3

Design Document

Group 13

Werner Schuster, Andreas Rath, Kerstin Pötsch, Wolfgang Pototschnik

Version 1.1, 24.01.2003

Contents

Chapter 1

Overview

1.1 Overview

The assignment consists of 3 major sets of changes concerning:

- changing the memory protection code from using page tables to using a software-managed TLB
- adding Virtual Memory to Nachos to allow userspace applications to use more memory than physically available
- add syscalls Malloc/Free to Nachos to allow userspace applications to dynamically allocate memory on the heap.

For information about the changes in the API for userspace applications see chapter 2.

Chapter 2

Application Programming Interface

2.1 Libraries

Alway include the `syscall.h` file for necessary definitions of types (SpaceID, OpenFileID) and for the API declarations (functions, constants for errorcodes,...). To Write to and Read from the console, use the STDOUT and STDIN file handles, also defined in `syscall.h`.

2.1.1 stdio

An experimental version of `printf` and `vsprintf`is also available as source code and can be used by including `stdio.h` and linking the applications .o file against the stdio.o file.

2.2 Syscalls

2.2.1 Details for new syscalls

void* Malloc(int size)

Tries to allocate size amount of bytes on the heap of the process. If this cannot be accomplished (not enough memory,...), the returned pointer will be the NULL pointer; otherwise the return value is the pointer to the allocated memory block.

void Free(void* pointer)

This tries to free the memory block that pointer points to. If no memory block has been allocated with that pointer, no memory will be free'd and the syscall returns normally (as opposed to some other implementations of free that would terminate the application).

2.2.2 Overview of new syscalls

syscall	arg1	arg2	arg3	arg4	return value
Malloc	int (number of bytes to allocate)				void * (pointer to the memory block)
Free	void* (pointer to memory block)				

Chapter 3

Implementation

3.1 Software-managed TLB

3.1.1 Changes in existing classes/files

AddrSpace

- TranslationEntry *pageTable
  to store all necessary translations (stored as array, one TranslationEntry per page);

- TranslationEntry *currentTLB
  stores a subset of the pageTable data, which is stored in the CPUs TLB; this is used to quickly restore the CPUs TLB at a context switch;

- numCurrentClockhandPosition
  used for the page replacement algorithm (clock algorithm); it stores the index of the next TLB entry to be examined;

- AddrSpace::RestoreState()
  this must be extende, to restore the CPUs TLB with the stored TLB and backup

- AddrSpace::SaveState()
  this must be extende, to backup state information of the page replacement algorithm.

exception.cc

- void ExceptionHandler(ExceptionType which)
  must be extended to handle the PageFaultException. Two cases must be distinguished:

  - page is invalid
    that means that the page is swapped out and must be read from the page file (more on that in the VM section)

  - page is valid
    the page is in memory, but the translation is not in the TLB; the translation must be taken from the page table and put into the TLB.

  This is also the place to implement the page replacement algorithm (the clock algorithm) to make sure that the most used page translations stay in the TLB and physical memory.

3.2 Virtual Memory

3.2.1 General

Swap file

The swap file name is created in the current working directory and has the name `stdio.h`. Its size is a multiple of the pagesize. The organization is simple: the first page in the swap file has the number $NumPages + 1$, thus no special data structures are needed to organize the swap file. The number of pages is $NumPages + (SizeOfSwapFile/Pagesize)$.

3.3 New subsystem: coreMap

This stores an array of all pages, using structs of type CoreMapEntry. These hold information about the pages owner (SpaceId,). The class CoreMapEntry is defined in the file `threads/CoreMapEntry.h`

3.3.1 Changes in existing classes/files

Initialize in system.cc

- add global coreMap variable (array of CoreEntry structures); these store the SpaceId of the pages owner and whether the pages is in memory or the swap file.
- create and open the swap file
- change the Memory Bitmap to be the size of the virtual memory (instead of just physical memory);

exception.cc

- handle PageFaultException in ExceptionHandler (see section on software managed TLB)
- SwapIn and SwapOut functions that can move pages from physical memory to the swap file and vice versa

addrspace.cc

Update the constructor to initialize/update the coreMap when creating a process.

3.4 Malloc/Free

3.4.1 General

This memory allocator employs a continuous heap that grows from heapBaseAddress upwards. Accessing any address above the heapTopAddress causes a Nachos segfault and the guilty application is immediatly killed. A First-Fit algorithm is used when finding free blocks of memory in the Malloc code (ie. the list is search from the start until a sufficientely sized block is found).

3.4.2 Changes in existing classes/files

addrspace.cc

- int Malloc(int length)
  implements the allocating of memory on the heap for this AddrSpace.
- int Free(int pointer) implements the freeing of memory on the heap for this AddrSpace.
- int AllocateNumPages(int num) Allocates num pages for this AddrSpace.
- Block * FindBlockByPointer(List *list, int pointer) Utility function to find a block in a block list (freeBlocks or allocatedBlocks).
- void FreeBlock(Block * block) Moves a block to the freeBlockList.
- void RemoveItem(List *list, void * item) Utility method for handling List objects.
- int heapBaseAddress
  start address of the heap
- int heapTopAddress
  the top of the heap, which grows upwards (about the same as the breakpoint of heaps in Unix-like OSes).

New data structures to hold information about the managed memory. These are the lists:

- usedPages
  list of Pages that were allocated for this processes heap;
- freeBlocks
  list of Blocks that are free to be allocated; this list should be sorted, to allow for quick first-fit allocation of memory blocks;

- allocatedBlocks
  Blocks that have been allocated;

The following structs are the stored in the lists above:

- Page
  - int pageNumber;
  - int refCount;
- Block
  - int pointer;
  - int size;
  - List pages;

exception.cc

- int doMalloc(int size);
  This walks the freeBlocks list to see if a Block is big enough to fit the desired block size (a first-fit approach is used, actually more of a next-fit approach, because the used List datastructure is not sorted). If no proper Block can be found, allocates enough pages (using MemoryManager) to fit the new memory block in. Information about these pages is added to the usedPages list. A new Block of with the desired size is generated and added to the allocatedBlocks list (the refCount of the used Pages must be incremented by 1). For the remaining space, another Block is generated and put on the freeBlocks list. If it is not possible to find a big enough Block or allocate enough pages to comply with the memory request, the syscall returns with a NULL pointer.
- int doFree(int pointer);
  The allocatedBlocks list is walked and search for a Block with the specified pointer. If that Block is found, it is removed from the allocatedBlocks list and moved to the freeBlocks list, where it stays for future calls of Malloc. This cannot avoid the problem of heap fragmentation, but it is easier to maintain a continous heap than (possibly huge) set of memory areas spread over the address space (one specific problem is checking whether an application tries to access an address which it has not allocated yet; with the continous heap, this is a matter of comparing the address with heapTopAddress, with seperate memory areas this would mean searching them all (or many of them) to do so).

syscall.h

Update the constants and function definitions with the new syscalls.

start.s

Update the code stubs that handle the syscalls with the new syscalls.

Chapter 4

Tests

4.1 Tests

To make sure the implementation has succeded and works correctly, all the new features are tested.

These tests are all userspace programs, that check the correct implementation of the new syscalls and other implemented features, and also check if the stability of the OS by attempting to execute syscalls with invalid arguments.

4.1.1 Software managed TLB

A proper way to test this has not yet been found. Since this is not new functionality, but basically a different form of managing pages, it should be to run the tests of Assignment 2; if those work correctly, this would mean that the implementation is correct (or at least equivalently flawed as the page table solution).

4.1.2 Virtual Memory

This will be tested by working with a program that statically allocates a lot of memory (at least bigger than the amount of core/physical memory). If that program delivers correct results, this means that virtual memory works correctly. This is implemented in the matmult application, which was in the original Nachos distribution, but has been modified to use significantly more memory.

4.1.3 Malloc/Free

This test allocates memory using Malloc in a loop until Malloc returns a NULL pointer (thus signaling that no memory is left to allocate). If the Malloc call is implemented correctly, this should at some point terminate (since memory is bound to run out at some point). If the program/loop terminates, the test is OK. There is another test in this category, called segfaultTest, which tries to access a memory address, which has not been allocated (better: that is outside the heap).