NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 3

Memory Management

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman

Handout: 06.02.2023

Introduction

This week's exercise is all about memory management. We are going to focus on different aspects, such as the methods the OS has to ensure security through isolation, the illusion the OS creates and how this all looks from the perspective of a user process.

1 Address Space of a Process

1. Memry Organization in a User-Space Process:

   (a) How is memory usually organized in a user-space process?

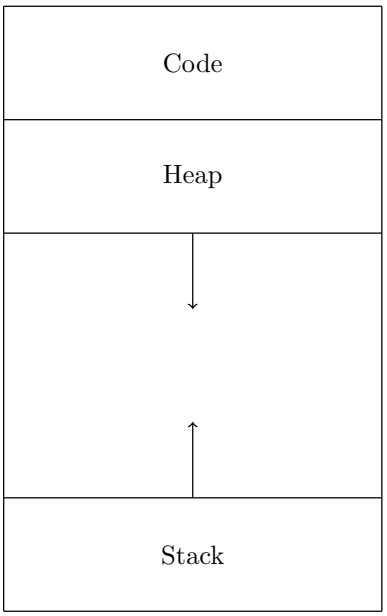
   Answer: *We can distinguish three different parts:*

   i. The static region, where the OS instantiates code, and other static data, such as constants.

   ii. The heap, where the process can allocate memory more or less freely by allocating memory through the OS. The management of that memory purely resides with the process, so unless the process frees a certain memory region again, it will stay allocated. The heap grows from the bottom towards the top of the address space.

   iii. The stack, where often the compiler takes on the management of the available space. Every function can allocate as much data on the stack as it wants, but after returning from a function, whatever was stored on the stack for this function is removed. The space on the stack that is specific to one function is also called a stack-frame. The stack grows from top to bottom (starts at the highest address).

   Those parts are usually organized in the following way:



   (b) Does memory organisation within a process change if we run on a uniprogramming or a multiprogramming OS?

   Answer: *No, the logic distribution of Code, Heap, and Stack section stays the same. The absolute addresses and the perceived size of the available memory region might change (if we are working on physical addresses), but the layout would stay the same.*

   (c) Why do we organize memory within processes like that? What are the advantages or disadvantages of organizing memory and address spaces in that particular way?

   Answer: *We need to place code somewhere, and while it does not matter where the code region is placed, placing it either close to the top or the bottom of the space not to fragment the available address space. Further, placing the Heap and the Stack on the*

opposite ends of the address region and letting them grow towards each other allows us to dynamically allocate as much area to either of those regions without having a fixed limit. Certain programs might make heavy use of the stack, while others rely more on the heap. This flexibility of not having to decide the heap and stack size ahead of time makes this layout very useful for most environments. In certain cases (e.g. on very constrained systems), we might want to develop a different memory layout to reduce management overhead. However, for such systems we usually have more information about the workload it is running and thus can fix the memory regions ahead of time.

2. Given the following virtual addresses, select to which of the three parts of the process memory they most likely belong to. **Note:** Knowing that can be very useful when debugging, as we know that we might need to look at a local variable going out of scope or checking variables that were allocated or if we are trying to write to a constant value.

   The solution below crosses every option that is possible. When you crossed one of them the solution would count as correct. There is no exact solution to that question, so don't expect such a question in the exam.

   - 0x0000000100003f98
     - ☒ Code
     - ☒ Heap
     - ☐ Stack
     - ☐ Stack or Code
     - ☒ Heap or Code
     - ☐ Stack or Heap
   - 0x7fffc7e18c2bcf98
     - ☐ Code
     - ☐ Heap
     - ☒ Stack
     - ☐ Stack or Code
     - ☐ Heap or Code
     - ☐ Stack or Heap
   - 0x000055e66a841004
     - ☒ Code
     - ☒ Heap
     - ☐ Stack
     - ☐ Stack or Code
     - ☒ Heap or Code
     - ☐ Stack or Heap

3. Let's see the stack at work: Write a small program that calls a function recursively, allocates a variable (remember: this variable will be put on the stack) and prints out the address of the variable. How do you expect the addresses that you print out to change, the deeper in the callstack we go?

   Answer: *See recursion.c for a sample C program that prints out the addresses. We would expect the addresses to become smaller and smaller, as we add stack frame after stack frame, since the stack should grow downwards.*

4. Which type of addresses are seen by the user-space program? Do the addresses that are seen by the kernel differ from the addresses by the user-space program?

   Answer: *The user-space program sees purely virtual addresses. That means that they do not directly correspond to a location in physical memory but can be arbitrarily mapped. E.g. an address could be mapped to a device, a file, physical memory or to nothing at all. The addresses that the kernel sees can be either physical addresses or whatever virtual address the kernel wants it to be. The kernel just needs to know in which memory space it is currently operating to ensure accessing a location in the right way.*

2 Address Space Organization by the OS

For the following questions, take the position of the OS. So you might have to handle multiple processes, with multiple address spaces, and the physical addresses. Also you are responsible to handle the physical address space and ensure all the guarantees that we expect from the OS with respect to process isolation and similar.

1. We discussed that the addresses within a single process follow a certain inner logic. When we look at the corresponding physical addresses, do they also have to follow the same logic? Please explain your solution

   Answer: *It does not necessarily follow the same rules as the virtual addresses of a process: The OS usually maps virtual addresses to physical addresses through the use of the MMU, a hardware structure that supports address translation. Thus, the OS does not need to follow the same structure, as arbitrary translation would be possible. Often however, the OS allocates a larger page in physical memory to make address translation more efficient, thus some of the structures might follow similar behaviours.*

2. When we start a process, we need to load the code and other static content into memory. How does the compiler know which addresses it needs to put down for the instructions, e.g. at which address a function starts?

   Answer: *The compiler does not know which absolute address to use. It assumes that the code starts at a specific address (e.g. 0) and writes all accesses relative to that address. When executing, these addresses become the virtual addresses for the code, and the OS is responsible to translate the addresses "on the fly". For this translation we can use simple mechanisms that don't require hardware support (e.g. such as constant offset for the entire program) or more sophisticated techniques, such as the MMU.*

3. Given the different different methods (fixed and variable partition) to allocate memory for a process, which problems do arise? How can they be circumvented?

   Answer: *In both cases we might end up fragmenting the physcial memory space to very small chunks that are not necessarily usable again. The mechanic however is different for the two approaches:*

Fixed partition: *Leads to internal fragmentation, since a process might not need all the allocated space, thus only ever using a small part of it, wasting a large part of the memory available to it. One way of making the problem less pronounced is by allocating smaller blocks, which on the other hand increases the work required to keep track of all the small blocks.*

Variable partition: *Leads to external fragmentation, since the allocated blocks are not of the same size, we might end up with holes that are too small for the next allocation to show up. External fragmentation can be counteracted by compacting the physical space from time to time. That however carries a large performance overhead, as lots of memory has to be moved around.*

4. Since modern hardware offers memory management units, capable of checking memory accesses, and fast address translation, why don't we just stitch together larger virtual blocks out of smaller physical blocks? Give at least three reasons why this might not be desireable.

   Answer:

   (a) *Mapping one large block can be done in a single entry, specifying its base address and its size. Building up the same block out of multiple small zones requires at least the same amount of entries.*

   (b) *We can cache some of the address translation calculations if we are currently operating on a very specific part of the dataset or the program code. When we stitch together larger blocks from smaller blocks, we would need a lot larger caches (called translation lookaside buffers, TLBs) to keep the same amount of memory in the buffer.*

(c) *If we have a larger block, we only need to translate part of the address, as the rest of the address can be read as the local offset into the larger block. The smaller the blocks, the larger the part of the address that actually needs to be translated.*

5. Does Segmentation help with external fragmentation?

   Answer: *No, it makes the system more flexible, but external fragmentation is still a problem, if not worse, since we now need to find space for three blocks for every process in the system.*

6. Can you suggest a solution in between variable and fixed partition? E.g. trying to combine the flexibility of the variable partitioning approach with the predictability of the fixed approach?

   Answer: *We could only allow a certain number of sizes, e.g. powers of 2. The nice thing about powers of 2 sized blocks, is that we can have multiple levels of translations (addresses stop at different offsets) and the smaller blocks make up one larger block entirely. That gives us some flexibility in the size while making external fragmentation less of a problem, as the resulting smaller block is known in size and if designed properly can house one of the smaller blocks perfectly.*