Memory Management

Summer 2021

Memory: an Historical Perspective

In the past, whether one was considering computers like that Sinclair ZX81 or the original IBM PC, programs accessed physical memory directly. These computers were intended to run a single program at once, but the result was still chaos. A program could access, or modify, any physical address, including those used by the operating system, or, in most computers, the video memory too.

Debugging was horrible, for a program could accidentally over-write part of the OS, exit successfully, and perhaps the next program would use the part of the OS which had been corrupted, and fail. Ideal for the intentionally mallicious, so viruses abounded.

On a modern multiuser system, this would be completely unacceptable.

As Good as Possible

Early CPUs, such as the Z80A (ZX81), 8086 (IBM PC), 68000 (Mac Classic), did not support any further sophistication. But since the mid 1980s most new CPU designs have supported the separation of the physical addresses from the 'virtual' addresses seen by programs. In a modern OS, each process ends up with its own virtual address space, and cannot access another process's address space.

Physical memory has no idea what type of data it stores: integer, floating point, program code, text, it's all just bytes. But a virtual address may have one of several attributes:

Invalid	not allocated
Read only	for constants and program code
Executable	for program code, not data
Shared	for inter-process communication
On disk	paged to disk to free up real RAM

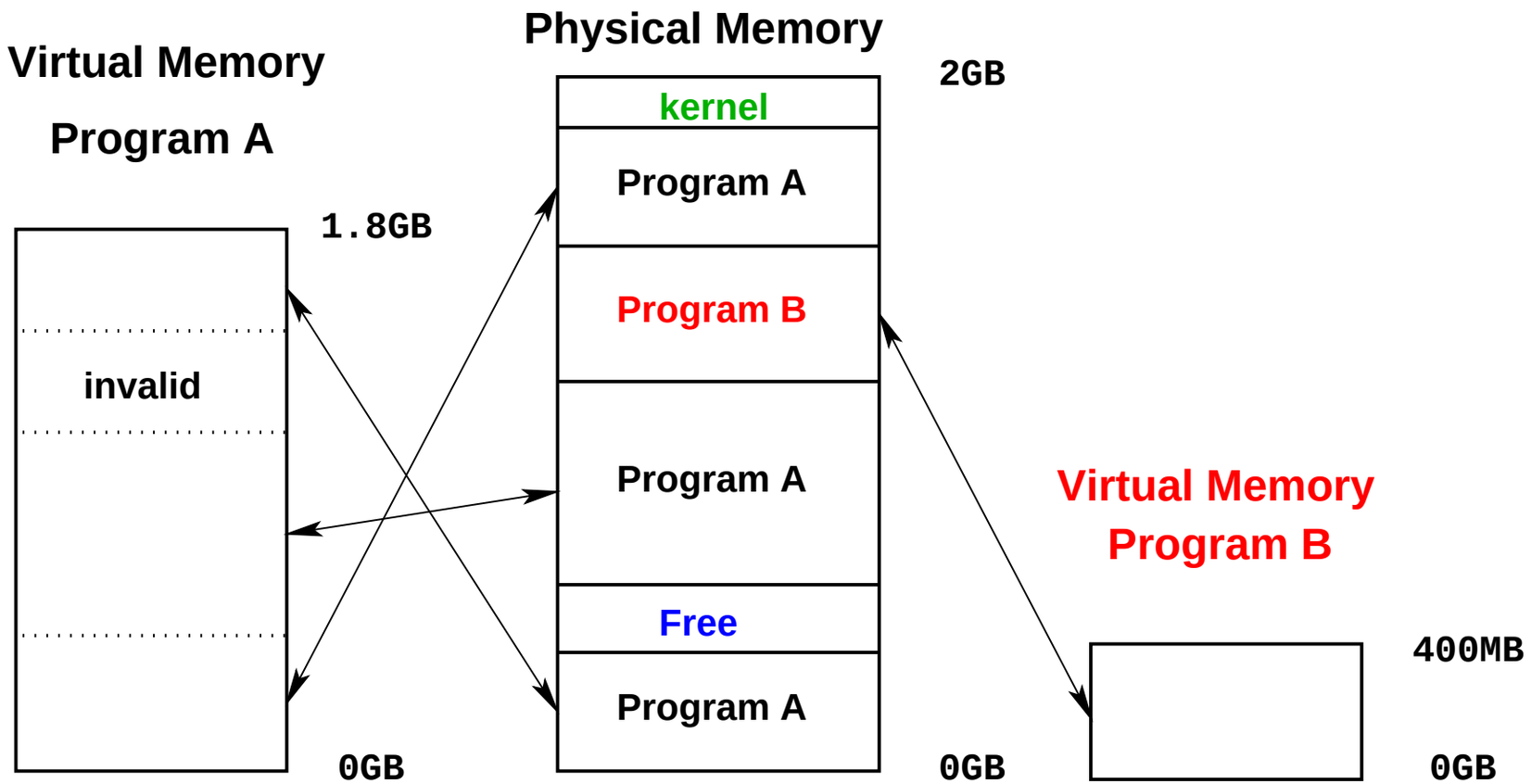
Pages

In practice book-keeping is simplified by dividing the virtual addresses into *pages*, contiguous regions, often of 4KB, which are described by just a single set of the above attributes. When the operating system allocates memory to a program, the allocation must be an integer number of pages. If this results in some extra space, `malloc()` or `allocate()` will notice, and may use that space in a future allocation without troubling the operating system.

Modern programs, especially those written in C or, worse, C++, do a lot of allocating and deallocating of small amounts of memory. Some remarkably efficient procedures have been developed for dealing with this. Ancient programs, such as those written in Fortran 77, do no run-time allocation of memory. All memory is fixed when the program starts.

Pages also allow for a mapping to exist between *virtual* addresses as seen by a process, and *physical* addresses in hardware.

No Fragmentation



Pages have an associated location in real, physical memory. In this example, program A believes that it has an address space extending from 0MB to 1800MB, and program B believes it has a distinct space extending from 0MB to 400MB. Neither is aware of the mapping of its own virtual address space into physical memory, or whether that mapping is contiguous. The OS can add (some of) its free memory to either process's address space.

Splendid Isolation

This scheme gives many levels of isolation.

Each process is able to have a contiguous address space, starting at zero, regardless of what other processes are doing.

No process can accidentally access another process's memory, for no process is able to use physical addresses. They have to use virtual addresses, and the operating system will not allow two virtual addresses to map to the same physical address (except when this is really wanted).

If a process attempts to access a virtual address which it has not been granted by the operating system, no mapping to a physical address will exist, and the access must fail. A segmentation fault. UNIX will respond to the invalid memory request by sending the signal SIGSEGV to the process. (Windows will complain of a 'General Protect Fault'. It makes the same complaint for illegal instructions (SIGILL on UNIXes) and some other errors too.)

A virtual address is unique only when combined with a process ID (deliberate sharing excepted).

Fast, and Slow

This scheme might appear to be very slow. Every memory access involves a translation from a virtual address to a physical address. Large translation tables (page tables) are stored in memory to assist. These are stored at known locations in physical memory, and the kernel, unlike user processes, can access physical memory directly to avoid a nasty catch-22.

Every CPU has a cache dedicated to storing the results of recently-used page table look-ups, called the TLB. This eliminates most of the speed penalty, except for random memory access patterns.

A TLB is so essential for performance with virtual addressing that the 80386, the first Intel processor to support virtual addressing, had a small (32 entry) TLB, but no other cache.

Page Tables

A 32 bit machine will need a four-byte entry in a page table per page. With 4KB pages, this could be done with a 4MB page table per process covering the whole of its 4GB virtual address space. However, for processes which make modest use of virtual address space, this would be rather inefficient. It would also be horrific in a 64 bit world.

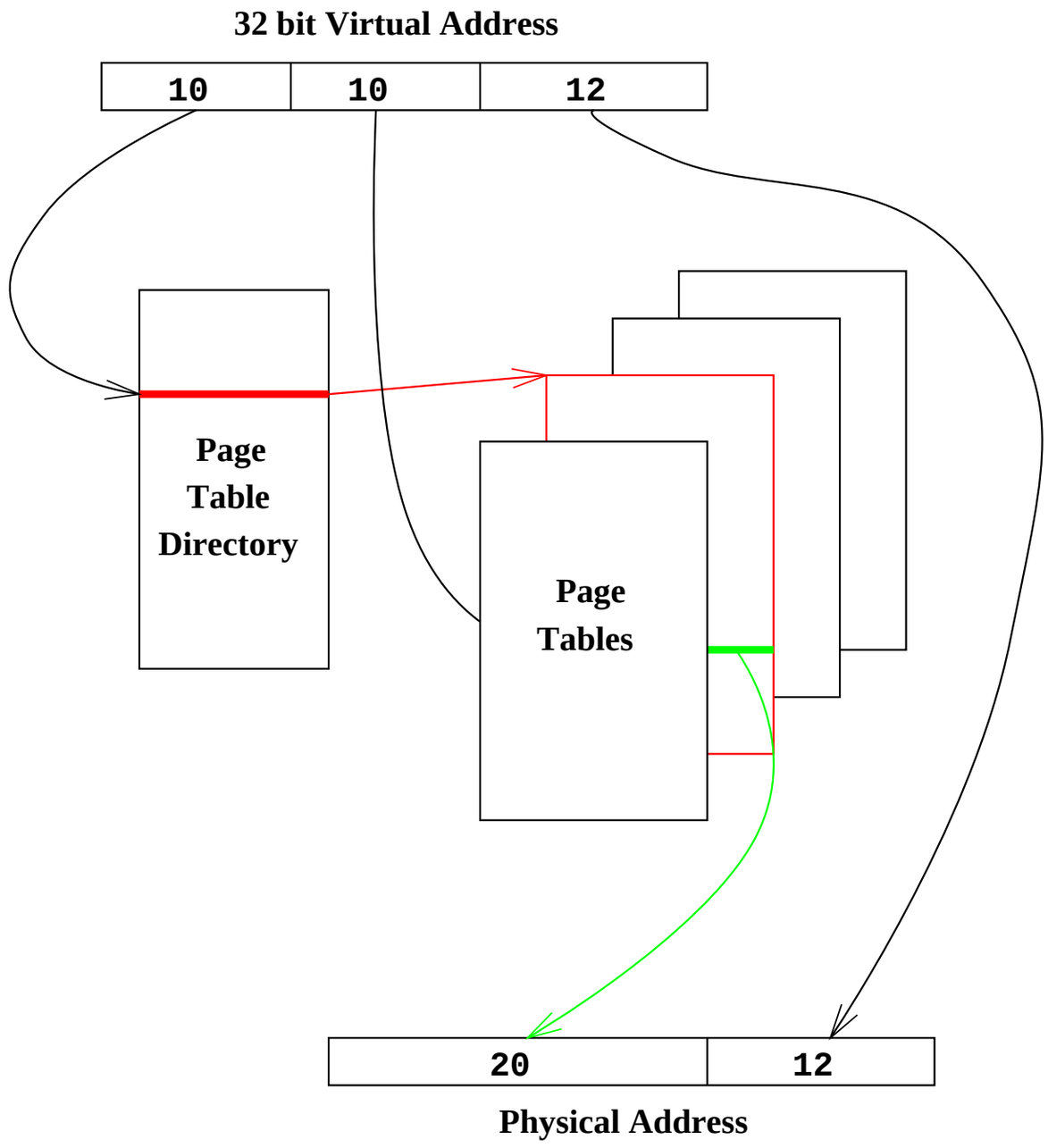
So the page table is split into two. The top level describes blocks of 1024 pages (4MB). If no address in that range is valid, the top level table simply records this invalidity. If any address is valid, the top level table then points to a second level page table which contains the 1024 entries for that 4MB region. Some of those entries may be invalid, and some valid.

The logic is simple. For a 32 bit address, the top ten bits index the top level page table, the next ten index the second level page table, and the final 12 an address within the 4KB page pointed to by the second level page table.

This sort of zero effort decoding also applies to cache lines, tags, and offsets within lines.

(Four bytes entries? The address of the page in physical memory will be exactly four bytes (32 bits), but the bottom 12 bits will be zero as pages start at a multiple of 4KB in physical memory. So these can be used for flags such as read-only, execute allowed, etc.)

Page Tables in Action



More paging

Having suffering one level of translation from virtual to physical addresses, it is conceptually easy to extend the scheme slightly further. Suppose that the OS, when asked to find a page, can go away, read it in from disk to physical memory, and then tell the CPU where it has put it. This is what all modern OSes do (UNIX, OS/2, Win9x / NT, MacOS X), and it merely involves putting a little extra information in the page table entry for that page.

If a piece of real memory has not been accessed recently, and memory is in demand, that piece will be paged out to disk, and reclaimed automatically (if slowly) if it is needed again. Such a reclaiming is also called a page fault, although in this case it is not fatal to the program.

Rescuing a page from a spinning disk will take about 10ms, compared with under 100ns for hitting main memory. If just one in 10^5 memory accesses involve a page-in, the code will run at half speed, and the disk will be audibly 'thrashing'.

The union of physical memory and the page area on disk is called *virtual memory*. Virtual addressing is a prerequisite for virtual memory, but the terms are not identical.

Blatant Lies

Paging to disk as above enables a computer to pretend that it has more RAM than it really does. This trick can be taken one stage further. Many OSes are quite happy to allocate virtual address space, leaving a page table entry which says that the address is valid, not yet ever been used, and has no physical storage associated with it. Physical storage will be allocated on first use. This means that a program will happily pass all its `malloc()` / `allocate` statements, and only run into trouble when it starts trying to use the memory.

The `ps` command reports both the virtual and physical memory used:

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
spqr1 20241  100 12.7 711764 515656 pts/9  Rl+  13:36  3:47 castep si64
```

RSS – Resident Set Size (i.e. physical memory use). Will be less than the physical memory in the machine. `%MEM` is the ratio of this to the physical memory of the machine, and thus can never exceed 100.

VSZ – Virtual SiZe, i.e. total virtual address space allocated. Cannot be smaller than RSS.

The Problem with Lying

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS     TTY  STAT START  TIME COMMAND
spqr1 25175 98.7 25.9 4207744 1049228 pts/3 R+   14:02  0:15 ./a.out
```

Currently this is fine – the process is using just under 26% of the memory. However, the `VSZ` field suggests that it has been promised 104% of the physical memory. This could be awkward.

```
$ ps aux
USER    PID %CPU %MEM    VSZ   RSS     TTY  STAT START  TIME COMMAND
spqr1 25175 39.0 90.3 4207744 3658252 pts/0 D+   14:02  0:25 ./a.out
```

Awkward. Although the process does no I/O its status is ‘D’ (waiting for ‘disk’), its share of CPU time has dropped (though no other process is active), and inactive processes have been badly squeezed. At this point Firefox had an RSS of under 2MB and was extremely slow to respond. It had over 50MB before it was squeezed.

Interactive users will now be very unhappy, and if the computer had another GB that program would run almost three times faster.

One can experiment with `ulimit -v` to limit a process’s virtual address space.

Grey Areas – How Big is Too Big?

It is hard to say precisely. If a program allocates one huge array, and then jumps randomly all over it, then the entirety of that array must fit into physical memory, or there will be a huge penalty. If a program allocates two large arrays, spends several hours with the first, then moves it attention to the second, the penalty if only one fits into physical memory at a time is slight. Total usage of physical memory is reported by `free` under Linux. Precise interpretation of the fields is still hard.

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4050700     411744    3638956          0       8348     142724
-/+ buffers/cache:     260672    3790028
Swap:      6072564      52980    6019584
```

The above is fine. The below isn't. Don't wait for `free` to hit zero – it won't.

```
$ free
             total       used       free     shared    buffers     cached
Mem:       4050700    4021984      28716          0        184     145536
-/+ buffers/cache:    3876264     174436
Swap:      6072564     509192    5563372
```

Page sizes

A page is the smallest unit of memory allocation from OS to process, and the smallest unit which can be paged to disk. Large page sizes result in wasted memory from allocations being rounded up, longer disk page in and out times, and a coarser granularity on which unused areas of memory can be detected and paged out to disk. Small page sizes lead to more TLB misses, as the virtual address space 'covered' by the TLB is the number of TLB entries multiplied by the page size.

Large-scale scientific codes which allocate GB of memory benefit from much larger page sizes than a mere 4KB. However, a typical UNIX system has several dozen small processes running on it which would not benefit from a page size of a few MB.

Intel's 80386 CPU was the first of the line including today's Core range to support virtual memory. It used a 4KB page size, and it was introduced in late 1985 when most computers had under 1MB of memory as memory cost well over £100/MB.

Choice

In Intel's 64 bit world, pages are still 4KB. A page used to store page table entries, each of which will be 64 bits (8 bytes), will contain $2^9 = 512$ entries. Four levels of page table are used, so the bits of an address are broken down as follows

16–24	25–33	34–42	43–51	52–63
index into directory	index into directory	index into directory	index into page table	offset with page

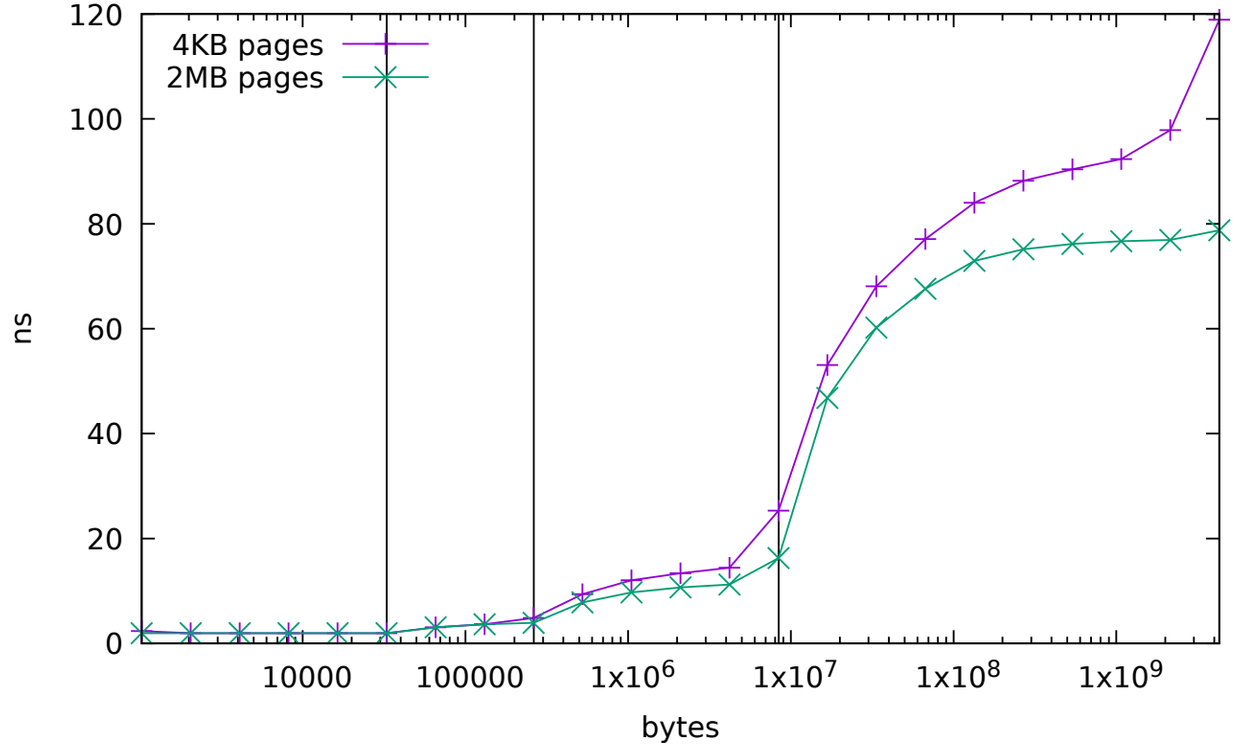
Bits 0 to 15 are not used; the virtual address space is just 48 bits (extended to 57 bits by Ice Lake, the 10th generation Intel Core CPU).

But, if the page table covering bits 43–51 actually describes 512 contiguous pages, a 2MB region, and this region starts at an address which is a multiple of 2MB, then it can be omitted and the directory will note that it is pointing to the real address, not another level of page table.

Randomness

If one's memory accesses are sequential, one will suffer at most one TLB miss every 512 accesses, assuming one's data are double precision variables, and one's page-size is 4KB.

If one's access pattern is random, it could be one miss per access. If it is random over an array size of 4MB, and one has 1024 TLB entries and 4KB pages, all is fine. But if those TLB entries were used with 2MB pages, all would be fine for random access to a 2GB array.



Expectations

The above graph used a 3.1GHz Haswell, and the vertical lines show the sizes of its three levels of cache.

Its TLB structure also has two levels. The first has 64 entries for 4KB pages (covering 256KB) and 32 entries for 2MB pages (covering 64MB). The last has 1024 entries, usable for either page size, so covering 4MB or 2GB.

The two lines for 4KB and 2GB page sizes start to separate for random access to a 512KB array. With 2MB pages, only the first level TLB is being used, but for 4KB pages both levels are needed. This is costing around 3ns.

The next problem comes at 4MB, when with 4KB pages all levels of TLB start to be missed. This seems to cost around 18ns.

But that is not the end of the story. When the TLB is missed, and page tables need to be referenced, page table entries are just like any other memory access, and are cached in the usual way. When the array size is 1GB, with a 4KB page size one needs a little over 2MB of page table to describe it. With a 2MB page size this reduces to a little over 4KB. So in one case the page tables are in L1 memory cache, in the other they are in L3. Increase the array size to 4GB, and with 4KB pages the page tables themselves would fill the L3 cache, leaving no room for data.

Big is better?

There are disadvantages to large pages. Sometimes fragmentation in physical memory will prevent Linux from using (as many) large pages. This will make code run slower, and the poor programmer will have no idea what has happened. Other disadvantages are that the kernel needs to avoid fragmentation in physical memory, and that paging to disk when the physical memory is full is more awkward, and slower, if one is faced with 2MB pages.

A rather artificial example (which is not too dissimilar from some finite element or sparse matrix problems) has shown speed increases of up to 50% for large pages. But the slowdowns when they go wrong can be much greater than this. Sufficient commercial applications (especially databases) run slower with huge pages that Linux tends to have support turned off by default, having been through a period when support was on and 'transparent', in that large allocations received large pages automatically with no code rewriting.

For sensible (i.e. unit stride) access patterns, the potential gain is likely to be a percent or two at most.

(Google libhugetlbfs for more on Linux large page support.)

Segments

A program uses memory for many different things. For instance:

- The code itself
- Shared libraries
- Statically allocated uninitialised data
- Statically allocated initialised data
- Dynamically allocated data
- Temporary storage of arguments to function calls and of local variables

These areas have different requirements.

Segments

Text

Executable program code, including code from statically-linked libraries. Sometimes constant data ends up here, for this segment is read-only.

Data

Initialised data (numeric and string), from program and statically-linked libraries.

BSS

Uninitialised data of fixed size. Unlike the data segment, this will not form part of the executable file. Unlike the heap, the segment is of fixed size. (BSS = Block Started by Symbol, from IBM assembler syntax of the mid 1950s!)

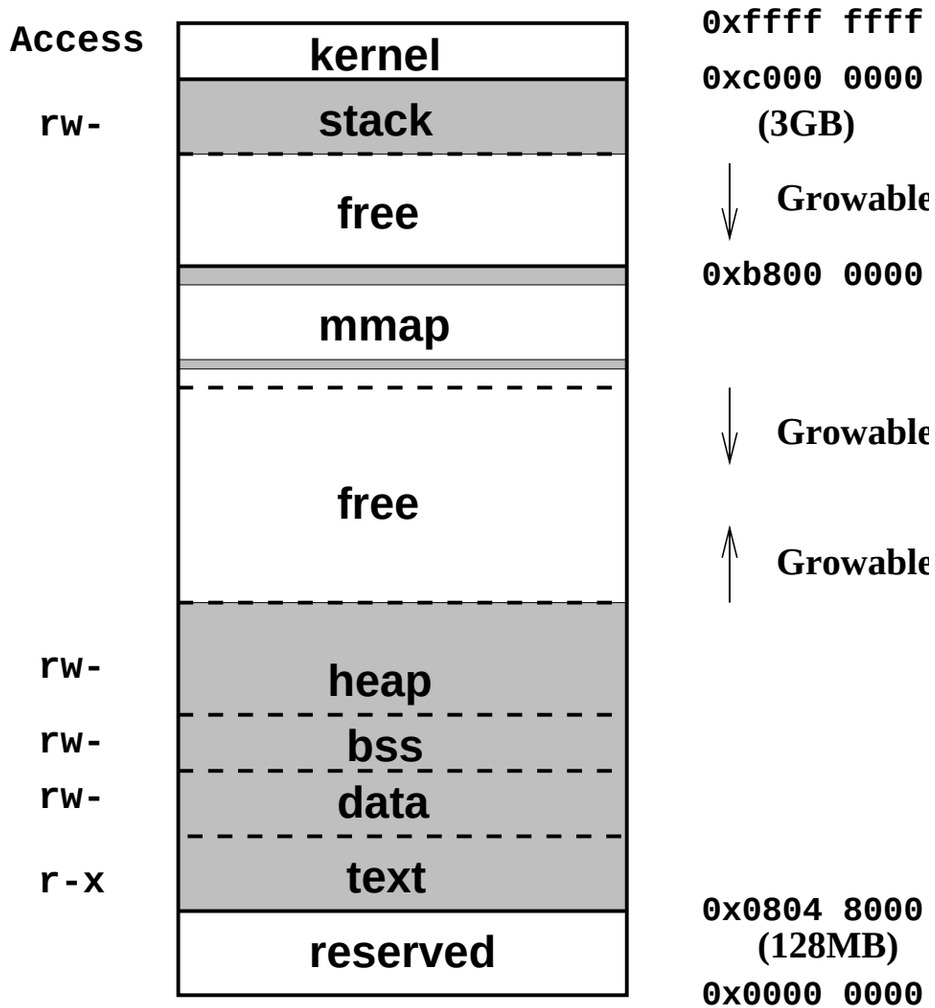
Heap

Area from which `malloc()` / `allocate()` traditionally gain memory.

Stack

Area for local temporary variables in (recursive) functions, function return addresses, and arguments passed to functions.

A Linux Virtual Address Space Map



This is roughly the layout used by Linux 2.6 on 32 bit machines, and *not to scale*.

Note the area around zero is reserved. This is so that null pointer dereferencing will fail: important in C/C++.

What Went Where?

Determining to which of the above data segments a piece of data has been assigned can be difficult. One would strongly expect C's `malloc` and F90's `allocate` to reserve space on the heap. Likewise small local variables tend to end up on the stack. The `mmap` region deals with shared libraries and part of the heap (see later).

Large local variables really ought not go on the stack: it is optimised for the low-overhead allocation and deletion needed for dealing with lots of small things, but performs badly when a large object lands on it. However compilers sometimes get it wrong.

UNIX limits the size of the stack segment and the heap, which it 'helpfully' calls 'data' at this point. See the '`ulimit`' command (`[ba]sh`).

Some UNIXes can also limit the total virtual address space used (`ulimit -v`). Some claim to be able to limit the resident set size (`ulimit -m`), but few actually do (Linux does not).

Because `ulimit` is an internal shell command, it is documented in the shell man pages (e.g. '`man bash`'), and does not have its own man page.

Sharing

If multiple copies of the same program or library are required in memory, it would be wasteful to store multiple identical copies of their unmodifiable read-only pages. Hence many OSes, including UNIX, keep just one copy in memory, and have many virtual addresses refering to the same physical address. A count is kept, to avoid freeing the physical memory until no process is using it any more!

UNIX does this for shared libraries and for executables. Thus the memory required to run three copies of Firefox is less than three times the memory required to run one, even if the three are being run by different users. It also greatly reduces program start-up times if their shared libraries are already in memory and being used by another process.

Two programs or libraries are considered identical by UNIX if they are on the same device and have the same inode. See elsewhere for a definition of an inode.

If an area of memory is shared, the `ps` command apportions it appropriately when reporting the RSS size. If the whole libc is being shared by ten processes, each gets merely 10% accounted to it.

mmap

It has been shown that the OS can move data from physical memory to disk, and transparently move it back as needed. However, there is also an interface for doing this explicitly. The `mmap` system call requests that the kernel set up some page tables so that a region of virtual address space is mapped onto a particular file. Thereafter reads and writes to that area of 'memory' actually go through to the underlying file.

The reason this is of interest, even to Fortran programmers, is that it is how all executable files and shared libraries are loaded. It is also how large dynamic objects, such as the result of large `allocate` / `malloc` calls, get allocated. They get a special form of `mmap` which has no physical file associated with it.

Heap vs `mmap`

Consider the following code:

```
a=malloc(1024*1024*1024); b=malloc(1); free(a)
```

(in the real world one assumes that something else would occur before the final `free`).

With a single heap, the heap now has 1GB of free space, followed by a single byte which is in use. Because the heap is a single contiguous object with just one moveable end, there is no way of telling the OS that is can reclaim the unused 1GB. That memory will remain with the program and be available for its future allocations. The OS does not know that its current contents are no longer required, so its contents must be preserved, either in physical memory or in a page file. If the program (erroneously) tries accessing that freed area, it will succeed.

Had the larger request resulted in a separate object via `mmap`, then the `free` would have told the kernel to discard the memory, and to ensure that any future erroneous accesses to it result in segfaults.

Automatically done

Currently by default objects larger than 128KB allocated via `malloc` are allocated using `mmap`, rather than via the heap. The size of allocation resulting will be rounded up to the next multiple of the page size (4KB). Most Fortran runtime libraries end up calling `malloc` in response to `allocate`. A few do their own heap management, and only call `brk`, which is the basic call to change the size of the heap with no concept of separate objects existing within the heap.

Fortran 90 has an unpleasant habit of placing large temporary and local objects on the stack. This can cause problems, and can be tuned with options such as `-heap-arrays` (ifort) and `-static-data` (Open64).

Objects allocated via `mmap` get placed in a region which lies between the heap and the stack. On 32 bit machines this can lead to the heap (or stack) colliding with this region. On 64 bit machines, there is no problem.

Heap layout

```
double precision, allocatable :: a(:),b(:),c(:)
allocate (a(300),b(300),c(20000))
```

In the absence of other allocations, one would expect the heap to contain `a` followed by `b`. This is 600 doubles, 4,800 bytes, so the heap will be rounded to 8KB (1024 doubles), the next multiple of 4KB. The array `c`, being over 128KB, will go into a separate object via `mmap`, and this will be 160KB, holding 20,480 doubles.

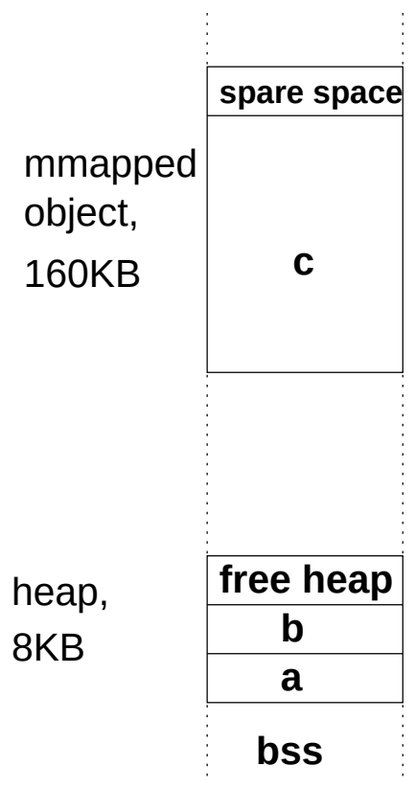


Diagram not to scale!

More segfaults

So attempts to access elements of `c` between one and 20,480 will work, and for `a` indices between one and 300 will find `a`, between 301 and 600 will find `b`, and 601 and 1024 will find free space. Only `a(1025)` will cause a segfault. For indices less than one, `c(0)` would be expected to fail, but `b(-100)` would succeed, and probably hit `a(200)`. And `a(-100)` is probably somewhere in the static data section preceeding the heap, and fine.

Array overwriting can go on for a long while before segfaults occur, unless a pointer gets overwritten, and then dereferenced, in which case the resulting address is usually invalid, particularly in a 64 bit world where the proportion of 64 bit numbers which are valid addresses is low.

Fortran compilers almost always support a `-C` option for checking array bounds. It very significantly slows down array accesses – use it for debugging, not real work! The `-g` option increases the chance that line numbers get reported, but compilers differ in how much information does get reported.

Theory in Practice

```
$ cat test.f90
double precision, allocatable :: a(:),b(:),c(:)

allocate (a(300),b(300),c(20000))
a=0
b(-100)=5

write(*,*)'Maximum value in a is ',maxval(a), &
          ' at location ',maxloc(a)
end

$ ifort test.f90 ; ./a.out
 Maximum value in a is   5.00000000000000   at location   208
$ gfortran test.f90 ; ./a.out
 Maximum value in a is   5.00000000000000000   at location   202
$ flang-amd test.f90 ; ./a.out
 Maximum value in a is   5.000000000000000 at location 206
$ nagfor test.f90 ; ./a.out
 Segmentation fault
```

-C

```
$ ifort -C -g test.f90 ; ./a.out
forrtl: severe (408): fort: (3): Subscript #1 of the array B
has value -100 which is less than the lower bound of 1

$ gfortran -C -g test.f90 ; ./a.out
 Maximum value in a is  5.0000000000000000   at location   202
$ gfortran -fcheck=bounds -g test.f90 ; ./a.out
At line 5 of file test.f90
Fortran runtime error: Index '-100' of dimension 1 of array 'b'
below lower bound of 1

$ nagfor -C -g test.f90 ; ./a.out
Runtime Error: test.f90, line 5: Subscript 1 of B (value -100)
 is out of range (1:300)
```

(`flang-amd` has no bounds checking option yet.)

Disclaimer

By the time you see this, it is unlikely that any of the above examples is with the current version of the compiler used. These examples are intended to demonstrate that different compilers are different. That is why I have quite a collection of them!

`ifort`: Intel's compiler, v 19.1
`gfortran`: Gnu's compiler, v 9.3
`flang-amd`: AMD/flang compiler, v 3.0 (LLVM 11.0.0) `nagfor`: NAG compiler, v 7.0

Four compilers. Only two managed to report line number, and which array bound was exceeded, and the value of the errant index.

Note too that if different compilers result in the same code behaving differently, this can mean that the code is buggy.

Generic Heap Debugging: Electric Fence

If your compiler won't help with heap debugging, there are alternatives. One is a libary which replaces `malloc()` with a function which guarantees an inaccessible page at each end of every region allocated on the heap. This is to make it more likely that wandering past the end of an array produces a segfault, rather than corrupting some other data. With addresses relative to the start of the heap, conceptually the below is produced.

```
180KB - 184KB   Invalid page
 20KB - 180KB   Array c (160,000 bytes) and 1,920 bytes of accessible padding
 16KB -  20KB   Invalid Page
 12KB -  16KB   Array b (2,400 bytes) and 1,696 of accessible padding
  8KB -  12KB   Invalid Page
  4KB -   8KB   Array a (2,400 bytes) and 1,696 of accessible padding
  0KB -   4KB   Invalid Page
```

The padding is a problem. Minor excursions do not get noticed. Electric fence's default is to put the padding first, which has implications for the alignment of the memory returned by `malloc()`.

Electric Fence in Action

```
$ gfortran -O0 -g test.f90
$ ./a.out
 Maximum value in a is    5.0000000000000000       at location          202

$ LD_PRELOAD=libefence.so.0.0 ./a.out
  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens
 Maximum value in a is    0.0000000000000000       at location            1

$ LD_PRELOAD=libefence.so.0.0 EF_PROTECT_BELOW=1 ./a.out
  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens
Segmentation fault - invalid memory reference.

[...]
#3  0x55a143ccd4aa in MAIN__
at test.f90:5
#4  0x55a143ccd6d2 in main
at test.f90:9
```

Generic Heap Debugging: Valgrind

Another commonly-used heap debugger is valgrind. It executes code in a virtual machine through a CPU emulator with all memory allocations and frees carefully recorded, and all memory accesses checked for validity. It does not use coarse pages for this, but places a 16 byte ‘red zone’ before and after each allocation in order to detect errors.

```
$ ./a.out
 Maximum value in a is    5.0000000000000000       at location          202
```

```
$ valgrind ./a.out
[...]
 Maximum value in a is    5.0000000000000000       at location          208
[...]

$ valgrind --redzone-size=1000 ./a.out
==2364== Invalid write of size 8
==2364==    at 0x1094AA: MAIN__ (test.f90:5)
==2364==    by 0x1096D2: main (test.f90:9)
==2364==  Address 0x4eedcc8 is 808 bytes before a block of size 2,400 alloc’d
==2364==    by 0x10933F: MAIN__ (test.f90:3)
==2364==    by 0x1096D2: main (test.f90:9)
[...]
 Maximum value in a is    0.0000000000000000       at location            1
```

Electric Fence vs Valgrind

Valgrind will detect single byte overflows at either end of arrays, whilst keeping correct alignments. Electric fence will detect at just one end, and may upset alignments.

Both will ensure that accessing memory which has been freed / deallocated will fail.

If your code does lots of very small memory allocations, Electric fence can increase memory usage dramatically. Each allocation is rounded up to a multiple of 4KB, and has 4KB added. Valgrind just adds a 16 byte red zone per allocation.

Valgrind will report where the nearest allocation to a failing access occured, as well as the location of the access. Electric fence reports only the access.

Valgrind will report memory not freed, and possibly leaked, at program termination.

Valgrind will report if the value of an uninitialised variable is used. (This can be very useful.)

Valgrind is very slow. Its execution time overhead can be a factor of twenty or more, whereas Electric fence is generally negligible.

Valgrind can continue past (some) errors if desired.

Both rely on memory allocations being via `malloc()`. This is usually, but not always, so.

The Stack

The stack is the area of memory used by a program in which bugs are most likely to occur, and hardest to trace. It mixes local function variables, and also function arguments and return addresses, in an unhelpful (and arguably unsafe) manner.

On almost all systems the stack grows downwards (whereas the heap grows upwards), and there is usually no explicit OS call to grow the stack. The OS ensures that an attempt to access memory 'just' beyond the end of the stack is interpretted as a request to grow the stack, rather than a segfault. On most OSes it never shrinks, so is inappropriate for very large objects.

Processors have instructions for dealing efficiently with the single stack. The instruction `push reg` decrements the stack pointer by the size of the register reg, and then saves its value at the location pointed to by the stack pointer. The instruction `pop reg` restores the value, and increments the stack pointer.

Calling a Function, the Traditional Approach

Address	Contents	Frame Owner
	space for return value	calling function
	. . .	
	2nd argument	
`%ebp+8`	1st argument	
`%ebp+4`	return address	current function
`%ebp`	previous `%ebp`	
	local variables etc.	
`%esp`	end of stack	

The stack grows downwards, and is divided into frames, each frame belonging to a function which is part of the current call tree. Two registers are devoted to keeping it in order, the Stack Pointer and Base Pointer. This example uses 4 byte addresses.

A Function Call

Caller

Increment stack pointer to create space for function's return value.

Push arguments onto the stack, starting with the last.

Push the return address onto the stack.

Call function.

Called function

Push base pointer onto stack.

Copy stack pointer to base pointer.

Push to stack any other registers which need to be preserved but would otherwise be overwritten.

Decrement stack pointer as needed for local variables.

Access arguments at any point in the function as offsets from base pointer.

Ending a Function Call

Restore any registers pushed.

Set stack pointer to base pointer (if not already).

Restore base pointer.

Execute the `return` instruction, which pops an address from the stack, and jumps to it.

Registers

There will be rules about which registers a function may use without needing to restore their original values. In other words, a caller will expect the contents of some registers to be lost after a function call, but perhaps not all of them.

The Linux convention is that very few registers are preserved. None of the floating point or vector registers, and just five of the general-purpose registers, `%rbx` and `%r12` to `%r15` (which the called function must save and restore if it modifies them). Windows adds to this list `%rsi`, `%rdi` and `%xmm6` to `%xmm15`.

Optimising Calling

The above explains why calling functions is quite expensive. For trivial operations (square roots, most operations on vectors of length two or three), the overhead of the call can be greater than the cost of the body of the function. Hence compilers try to inline functions to increase speed.

Optimisations on 64 bit Linux are that the return value is returned in a register if it is integer (up to 128 bits), floating point, or complex (a pair of floats). The first six integer arguments are passed in registers, as are the first eight floating-point (or complex) arguments. A ‘leaf’ function (a function which calls no other function) need not adjust the base pointer, and may use up to 128 bytes below the stack pointer for temporary storage without modifying the stack pointer. The base pointer is optional too, at the expense of less reliably debugging.

That the caller must assume that any function might trash any of the vector registers is a potential problem. If AVX-512 is in heavy use, there could be 32 registers, each of 64 bytes, to save before the function call and to restore afterwards!

Windows is different. Only the first four integer arguments are passed in registers, and only the first four floating point.

Debugging

With the traditional approach, a debugger can ‘walk the stack’ recursively, using the value of the base pointer to give the deepest frame boundary, and there finding the position of the next deepest. All the arguments to each function are on the stack, and, as long as the debugger knows what type of argument to expect, their values can be displayed.

With the optimisations on 64-bit Linux (or Windows), most of the function arguments disappear from the stack. As functions will modify their registers whilst they execute, they are completely lost. Compile with debugging turned on *and* optimisation off, and a separate copy of function arguments will be saved for the benefit of the debugger (and your code will run slower). Otherwise debuggers are likely to show arguments as having the value ‘optimised out.’

When in 32 bit mode, only eight integer registers are available, two of which are the stack pointer and base pointer, leaving six. The base pointer is only essential for walking the stack, and, with so few registers, gaining an extra one by using the base pointer as a general purpose register, rather than having it continuously hold the value of the start of the current stack frame, can result in faster code. Hence GCC’s `--f[no]omit-frame-pointer` option.

In 64 bit mode, there are an extra eight integer registers, and the benefit of moving from 14 to 15 is much less than the benefit of moving from six to seven.

Stack Theory in Practice

```
#include <stdio.h>

void foo(){
  char b[50];
  b[90]='O';
}

int main(){
  char a[]="XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  printf("%s\n",a);
  foo();
  printf("%s\n",a);
  return 0;
}

$ gcc -O0 stack.c
$ ./a.out
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXOXXXXXXXXXXXXXXXXXXXXXXXXX
```

More Stack Theory in Practice

```
#include <stdio.h>
void foo();

int main(){
  printf("Hello\n"); foo(); printf("Goodbye\n");return 0;
}

int bar(){
  asm volatile(".rept 200\n nop\n .endr");
  printf("Only wizards see this\n"); return 0;
}

void foo(){
  void **ptr;
  int i;
  ptr=&ptr;

  for(i=0;i<20;i++)
    if ((ptr[i]>main)&&(ptr[i]<bar)){
      printf("Found it!\n"); ptr[i]+=24; break;
    }
}
```

More Stack Theory in Practice (2)

The function `main()` prints 'Hello', calls `foo()`, prints 'Goodbye', and exits.

The function `foo()` uses the address of a local variable to get the location of something on the stack (`ptr=&ptr`). It then works up the stack until it finds a value which is within the address range of `main()`. This is probably the value stored on the stack as the return address of this function. It adds `24*sizeof(void*)` to it, so it will now return 192 bytes further along in the program than expected.

The new return address will probably hit the large block of `nop`s in the function bar. (`nop` = NO Operation, a single byte instruction meaning do nothing, move to next byte). So rather than executing the end of `main()`, it executes the end of the function `bar()`. Its output is

```
Hello
Found it!
Only wizards see this
```

with 'Goodbye' not printed. Note we have assumed that the compiler and linker choose to keep the ordering main, bar, foo in memory, that the alignment of `ptr` on the stack is compatible with the alignment of the return address, and that adding 192 bytes to the return address is sufficient to skip the rest of `main()` and the start of `bar()`. Thus we demonstrate that our local variables and function return addresses sit alongside each other on the stack.

Debugging Stack Problems

A nightmare.

The lack of a formal process for allocation and deallocation means that segfaults almost never happen immediately, and Electric Fence and Valgrind are of no use. Worse, one can readily overwrite the important markers around the boundary of each frame (return addresses, saved value of the Base Pointer, and function arguments). This means that when the corruption finally triggers a segfault as a secondary effect, it is likely that the debugger cannot correctly walk back along the stack to show the current call tree, as necessary data will have already been destroyed.

Changing the number of local variables allocated to the stack (which includes doing something which changes whether a variable is held entirely within a register and has no associated memory location) will alter the position of things on the stack, and may make a program which is erroneously overwriting things on the stack behave differently. Changing the compiler optimisation level is usually sufficient.

Some compilers can place canaries on the stack. These are recognisable values placed immediately after the return address and saved base pointer. When the function exits, it first checks that the canary still holds the expected value. If not, then the canary has been overwritten, and probably the function return address too, so it is time to panic. But, from the point of view of debugging, it is already too late, for the stack frame is probably already damaged beyond repair.

The Nightmare Continues

One might wonder whether the heap test program on slide 28 can be provoked into giving a segfault. It can, but surprisingly the reference to `b` had to be decreased to about `b(-2200)` to achieve this. What of

```
double precision :: a(300),b(300),c(20000)

a=0
b(-100)=5
write(*,*)'Maximum value in a is ',maxval(a), &
          ' at location ',maxloc(a)
end
```

The arrays are now on the stack, being local variables. An index to `b` of –1,000,000 does not (quite) produce a segfault. The point at which the segfault occurs is when the stack size exceeds

```
$ ulimit -a | grep stack
stack size              (kbytes, -s) 8192
```

The arrays are stored, at least with gfortran 9.3, in the opposite order, so `b(400)=5` will modify a value in `a`. The compiler regards `c` as too big for the stack, unless `-fstack-arrays` is specified. Valgrind does detect the error of accessing addresses below the stack pointer.

Memory Maps in Action

Under Linux, one simply needs to examine `/proc/[pid]/maps` using `less` to see a snapshot of the memory map for any process one owns. It also clearly lists shared libraries in use, and some of the open files. Unfortunately it lists things upside-down compared to our pictures above.

The example on the next page clearly shows a program with the bottom four segments being text, data, bss and heap, of which text and bss are read-only. In this case mmaped objects are growing downwards from `f776 c000`, starting with shared libraries, and then including large malloced objects.

The example was from a 32 bit program running on 64 bit hardware and OS. In this case the kernel does not need to reserve such a large amount of space for itself, hence the stack is able to start at `0xfffb 9000` not `0xc000 0000`, and the start of the `mmap` region also moves up by almost 1GB.

Files in `/proc` are not real files, in that they are not physically present on any disk drive. Rather attempts to read from these 'files' are interpretted by the OS as requests for information about processes or other aspects of the system.

The machine used here does not set read and execute attributes separately – any readable page is executable.

The Small Print

```
$ tac /proc/20777/maps
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
fff6e000-fffb9000 rwxp 00000000 00:00 0          [stack]
f776b000-f776c000 rwxp 0001f000 08:01 435109     /lib/ld-2.11.2.so
f776a000-f776b000 r-xp 0001e000 08:01 435109     /lib/ld-2.11.2.so
f7769000-f776a000 rwxp 00000000 00:00 0
f774b000-f7769000 r-xp 00000000 08:01 435109     /lib/ld-2.11.2.so
f7744000-f774b000 rwxp 00000000 00:00 0
f773e000-f7744000 rwxp 00075000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76c8000-f773e000 r-xp 00000000 00:13 26596314   /opt/intel/11.1-059/lib/ia32/libguide.so
f76a7000-f76a9000 rwxp 00000000 00:00 0
f76a6000-f76a7000 rwxp 00017000 08:01 435034     /lib/libpthread-2.11.2.so
f76a5000-f76a6000 r-xp 00016000 08:01 435034     /lib/libpthread-2.11.2.so
f768e000-f76a5000 r-xp 00000000 08:01 435034     /lib/libpthread-2.11.2.so
f768d000-f768e000 rwxp 00028000 08:01 435136     /lib/libm-2.11.2.so
f768c000-f768d000 r-xp 00027000 08:01 435136     /lib/libm-2.11.2.so
f7664000-f768c000 r-xp 00000000 08:01 435136     /lib/libm-2.11.2.so
f7661000-f7664000 rwxp 00000000 00:00 0
f7660000-f7661000 rwxp 00166000 08:01 435035     /lib/libc-2.11.2.so
f765e000-f7660000 r-xp 00164000 08:01 435035     /lib/libc-2.11.2.so
f765d000-f765e000 ---p 00164000 08:01 435035     /lib/libc-2.11.2.so
f74f9000-f765d000 r-xp 00000000 08:01 435035     /lib/libc-2.11.2.so
f74d4000-f74d5000 rwxp 00000000 00:00 0
f6fac000-f728a000 rwxp 00000000 00:00 0
f6cec000-f6df4000 rwxp 00000000 00:00 0
f6c6b000-f6c7b000 rwxp 00000000 00:00 0
f6c6a000-f6c6b000 ---p 00000000 00:00 0
f6913000-f6b13000 rwxp 00000000 00:00 0
f6912000-f6913000 ---p 00000000 00:00 0
f6775000-f6912000 rwxp 00000000 00:00 0
097ea000-0ab03000 rwxp 00000000 00:00 0          [heap]
0975c000-097ea000 rwxp 01713000 08:06 9319119    /scratch/castep
0975b000-0975c000 r-xp 01712000 08:06 9319119    /scratch/castep
08048000-0975b000 r-xp 00000000 08:06 9319119    /scratch/castep
```