Understanding Memory

Our context for this discussion is the AICT Linux Cluster, which runs 64-bit GNU/Linux on AMD Opteron hardware. .

Contents

Introduction

Under Linux, all programs run in a *virtual memory* environment. If a C programmer prints the value of a pointer (never necessary in practice), the result will be a virtual memory address. In Fortran, although pointers are not a standard feature, virtual memory addressing is implicit in every variable reference and subroutine call.

Of course, a program's code and data actually reside in *real physical memory*. Therefore, each virtual memory address is mapped by the operating system to a physical memory address in a structure known as the *page table* (see Figure 1). So, for example, to retrieve or update the value of a variable, a program must first acquire the variable's real memory address by looking up the associated virtual memory address in the page table. Fortunately, this step is handled transparently by Linux. Optimized code in the Linux kernel and specialized circuitry in the CPU combine to make this a reasonably efficient operation. Nevertheless, you may wonder why do it at all.

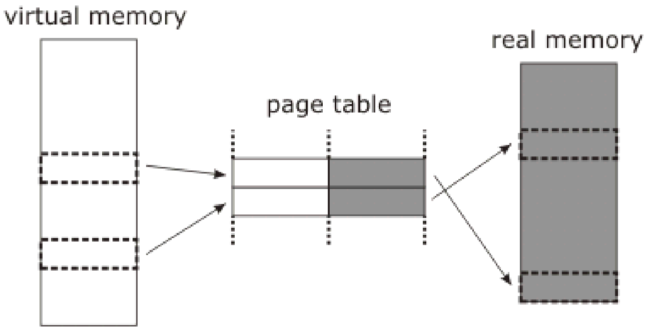


Fig. 1 Mapping virtual memory to real memory through the page table.

In a general purpose multi-user computing environment such as Linux, all programs must share the finite physical memory that is available. In the absence of virtual memory, each program would have to be aware of the activities of its neighbours. For example, consider two independent programs with direct memory access trying to claim the same area of free memory at the same time. To avoid a conflict, the programs would have to participate in a synchronization scheme, leading to overly complex code. Instead, with virtual memory, all low-level memory management functions are delegated to the operating system. Accordingly, the Linux kernel maintains a private page table for each program giving it the illusion that it is alone on the computer. When two concurrent programs reference the same virtual memory address, the kernel ensures that each one resolves to a different real memory address as shown in Figure 2.

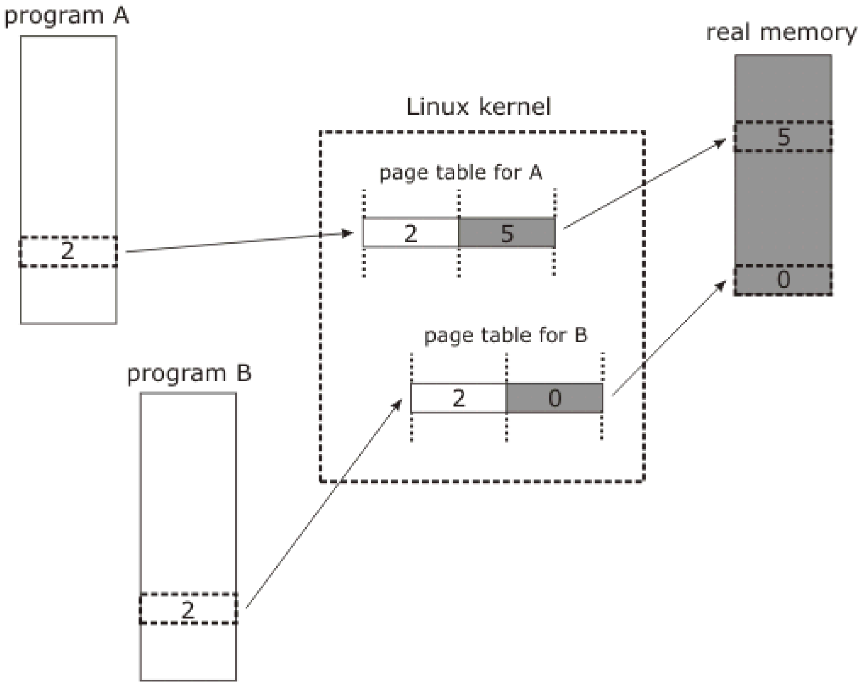


Fig. 2 Concurrent programs.

It may also help to think of virtual memory as an abstraction layer that the operating system uses to insulate the memory hardware. Programs simply run on top of this layer without having to be aware of the details of any specific memory implementation.

We will become acquainted with the concepts and terminology associated with virtual memory in the sections that follow. Although our focus will be the AICT Linux cluster, the principles described apply to most other computing environments.

goto top

Programs and Processes

A program's page table is one component of its execution context. Other components include its current working directory, list of open files, environment (list of environment variables), and so on. Together, they constitute what is known as a *process* or *task*.

Often, "program" and "process" are used interchangeably. There is a difference however. The term program is usually associated with source code written in a particular programming language. We speak of a Fortran program or a C program for example. It also refers to the compiled source code or executable file on disk. A process, on the other hand, is the operating system's concept for a running program.

The kernel assigns each process a unique identification number, a *process ID* (pid), and uses this to index the various data structures that store information about the process. A program can retrieve its pid and many other process-related attributes through a programming interface. This interface is standard in C and is supported as an extension in Fortran. For example, `getuid()` returns the user ID (uid) of the calling process.

Executing a program interactively at the shell's command prompt is a common way to create a new process. The new process is literally spawned, or forked (after the `fork()` system call that is involved), from the shell process. In this way, a process hierarchy is established, with the shell as parent and the new process as child. Naturally, the child inherits many of the attributes of its parent, like the current working directory and environment. Significantly, memory resource limits are also passed down from parent to child. More on this later.

goto top

Storage Class and Scope

Programs comprise executable statements and data declarations. Each datum has a property known as *storage class* that reflects its lifespan during program execution. A related property called *scope* characterizes the extent of a datum's visibility. Storage class and scope are assumed from the location of a datum's declaration, and determine its placement within virtual memory.

In C, data declared outside the body of any function have global scope and static (permanent) duration. Although initial values may be assigned, global data are usually uninitialized. Meanwhile, data declared inside the body of a function—including `main()`—have local scope and automatic (temporary) duration. A local datum can be made permanent by qualifying its declaration with the `static` keyword so that it retains its value between function invocations.

In Fortran, all data are local in scope unless they are declared in a module (Fortran 90/95) or appear in a named common block. Furthermore, by assigning the `SAVE` attribute to module variables and by referencing common blocks in a `SAVE` statement (or carefully locating common block specifications), they effectively acquire global scope and static duration. In the case of the Portland Group (PGI) Fortran compilers pgf77 and pgf95, local explicit-shaped arrays (those for which fixed bounds are explicitly specified for each dimension) have the static, rather than automatic, storage class. However, the contents of such arrays are invalidated between subroutine invocations, unless they are declared with the `SAVE` attribute or they appear in a `SAVE` statement.

> *Be aware that the treatment of explicit-shaped arrays differs among compilers. In contrast to pgf95, the IBM compiler xlf95, for example, considers them automatic. If necessary, these semantics can be altered with compiler options.*

goto top

Program Size

Compilers translate a program's executable statements into CPU instructions, and declarations of *static* data are translated into machine-specific data specifications. To create an executable file, the system linker aggregates the instructions and the data into distinct segments. All of the instructions go into one segment traditionally called *text*. Unfortunately, that name conveys the impression that the segment contains source code, which it does not. Meanwhile, the data are arranged in two segments. One is called *data*, for the initialized static data and literal constants, and the other, *bss*, for the uninitialized static data. Bss once stood for "block started from symbol," which was a mainframe assembly language instruction, but the term carries no meaning today.

Consider the following simple C program, and the equivalent Fortran 90/95 version, in which the major data component is an uninitialized static array of 200 million bytes.

```
/**
  * simple.c
  */
#include <stdio.h>
#include <stdlib.h>

#define NSIZE 200000000

char x[NSIZE];

int
main (void)
{
  for (int i=0; i<NSIZE; i++)
    x[i] = 'x';

  printf ("done\n");

  exit (EXIT_SUCCESS);
}
```

```
$ pgcc -c9x -o simple simple.c
$ size simple
   text    data    bss         dec         hex        filename
   1226    560     200000032   200001818   bebc91a    simple

$ ls -l simple
-rwxr-xr-x  1 esumbar uofa 7114 Nov 15 14:12 simple
```

```
!
! simple.f90
!
module globals
```

```
  implicit none
  integer, parameter :: NMAX = 200000000
  character(1), save :: x(NMAX)
end module globals

program simple
  use globals
  implicit none
  integer :: i

  do i = 1, NMAX
     x(i) = 'x'
  end do
  print*, "done"
  stop
end program simple
```

```
$ pgf95 -o simple simple.f90
$ size simple
   text    data    bss         dec         hex       filename
   77727   1088772 200003752   201170251   bfd9d4b   simple

$ ls -l simple
-rwxr-xr-x  1 esumbar uofa 1201694 Nov 15 14:12 simple

$ file simple
simple: ELF 64-bit LSB executable, AMD x86-64, ...
```

Compiling (and implicitly linking) as illustrated above produces an executable program file in the ELF (Executable and Linking Format) format. Running the size command extracts the magnitude of the text, data, and bss segments from the ELF file.

In both cases, the bss segment is indeed 200 million (plus some administrative overhead). The program's contribution to the data segment includes only two string literals and one numeric constant, which does not account for the sizes reported. Apparently, the compilers are responsible for the discrepancy.

Furthermore, because the ELF file contains all of the program instructions and all of the initialized data, the sum of the text and data segments should approach, but never exceed, the size of the file on disk. Reserving space for the bss segment in the file is unnecessary since there are no values to store. This is confirmed by the examples.

> *Be aware that the data and bss segments are frequently referred to collectively as just data, occassionally leading to confusion.*

goto top

Memory Map

When the ELF file is executed, the text and the two data segments are loaded into separate areas of virtual memory. By convention, text occupies the lowest addresses, with data above. Appropriate permissions are assigned to each. In general, the text segment is read-execute, while the data segment is read-write. A typical process *memory map* is illustrated in Figure 3.

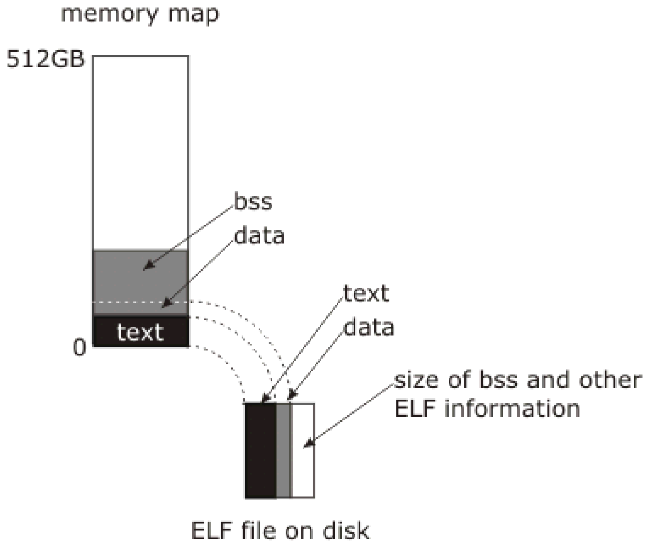


Fig. 3 Process memory map showing text, data, and bss segments.

Virtual memory addresses start with zero at the bottom of the figure and increase to a maximum of 512GB at the top. Addresses above 512GB are reserved for use by the Linux kernel. This is specific to AMD64 hardware. Other architectures may have different limits.

Although the *size* (text+data+bss) of a process is established at compile time and remains constant during execution, a process can expand into the unoccupied portion of virtual memory at runtime using the `malloc()` function in C or `ALLOCATABLE` arrays in Fortran 90/95. A similar feature is also available in Fortran 77 through non-standard extensions. This dynamically allocated memory is located above data in the *heap segment*. See Figure 4.

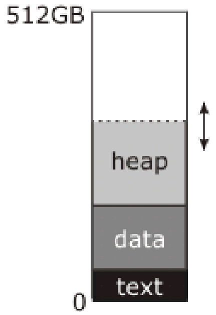


Fig. 4 Memory map with heap segment included.
Data and bss segments are shown as one.

All three segments, text, data (data+bss), and heap, are mapped to real memory through the page table. The figure shows that the heap segment expands and contracts as memory is allocated and deallocated. Consequently, page table entries are added or deleted as necessary.

goto top

Call Stack

Programs written in the procedural style (as opposed to object-oriented style) are organized as a logical hierarchy of subroutine calls. In general, each subroutine call involves passing arguments from the caller to the callee. In addition, the callee may declare temporary local variables. Subroutine arguments and *automatic* local variables are accommodated at the top of virtual memory in an area known as the *stack segment* or simply as the *stack*. See Figure 5.

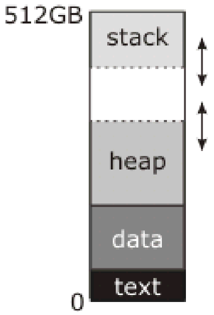
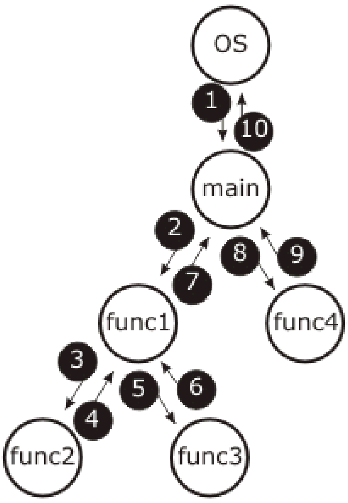


Fig. 5 Memory map showing the stack segment.

The hierarchy of subroutine calls begins when the operating system invokes the program's `main()` function in C or the `MAIN` program in Fortran. Under normal circumstances, it ends when "main" returns to the operating system. The entire sequence can be represented as a call graph like that in Figure 6.



1. OS calls main
2. main calls func1
3. func1 calls func2
4. func2 returns to func1
5. func1 calls func3
6. func3 returns to func1
7. func1 returns to main
8. main calls func4
9. func4 returns to main
10. main returns (exit status) to OS

Fig. 6 Typical subroutine call graph.

Before calling main, the operating system pushes the elements of the command line that was used to invoke the program on "top" of the initially empty stack. In C, the `main()` function has access to these arguments through the `argc` and `argv` parameters, while Fortran `MAIN` programs can use the `IARGC` and `GETARG` subroutines, which are non-standard extensions.

As execution commences, main pushes its automatic variables on top of the stack. This makes the stack "grow" towards lower addresses. Then, just prior to calling func1, main pushes the arguments to func1. Together, main's automatic variables and the arguments to func1 constitute a *stack frame*. Stack frames accumulate on the stack as the program descends the call graph, and are dismantled as it ascends. The procedure is outlined in Figure 7 below.

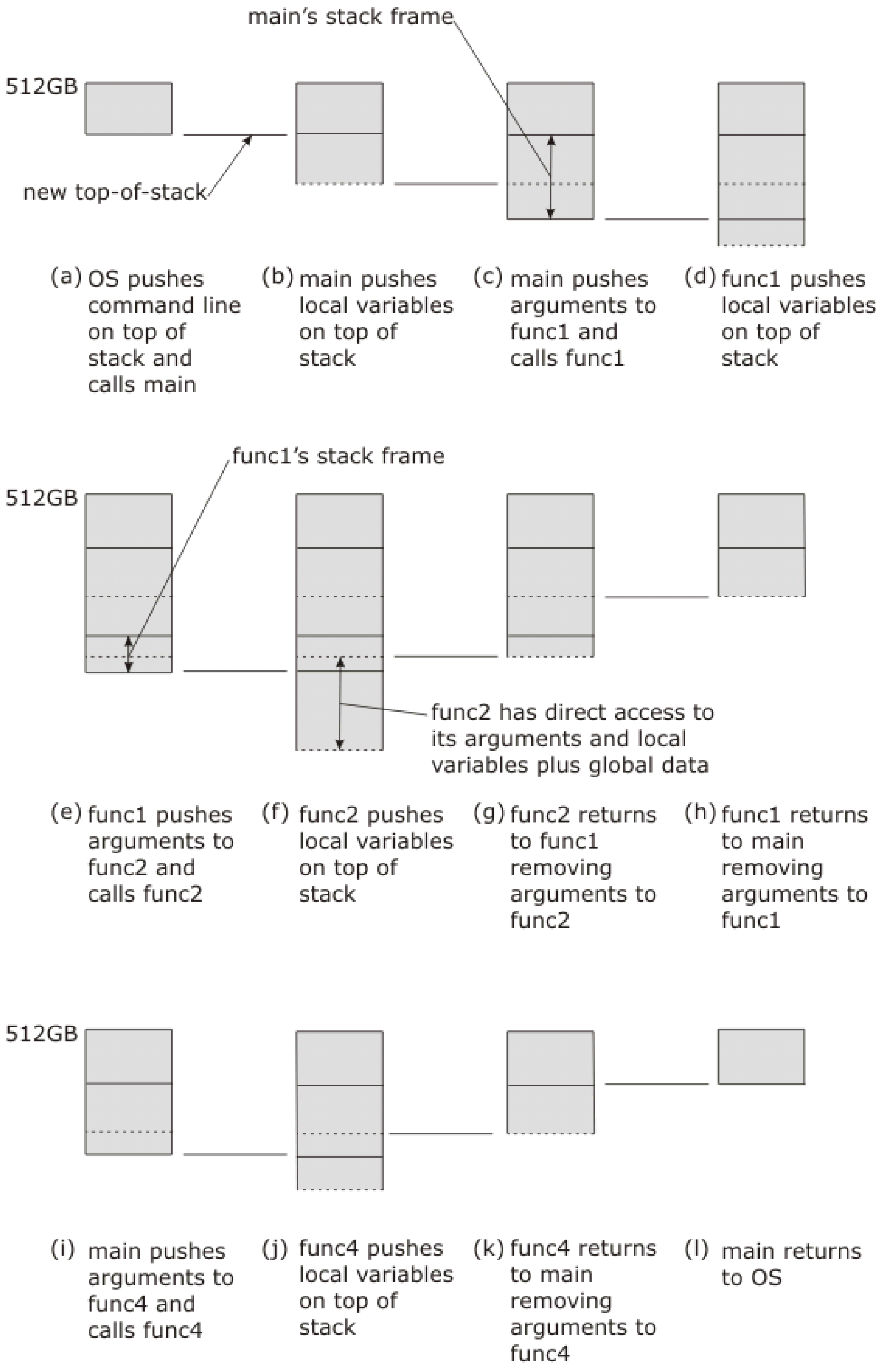

main's stack frame
512GB
new top-of-stack
(a) OS pushes command line on top of stack and calls main
(b) main pushes local variables on top of stack
(c) main pushes arguments to func1 and calls func1
(d) func1 pushes local variables on top of stack
func1's stack frame
512GB
func2 has direct access to its arguments and local variables plus global data
(e) func1 pushes arguments to func2 and calls func2
(f) func2 pushes local variables on top of stack
(g) func2 returns to func1 removing arguments to func2
(h) func1 returns to main removing arguments to func1
512GB
(i) main pushes arguments to func4 and calls func4
(j) func4 pushes local variables on top of stack
(k) func4 returns to main removing arguments to func4
(l) main returns to OS

Fig. 7 Evolution of the stack segment corresponding to the call graph shown in Figure 6.

By convention, the currently active subroutine can only reference the arguments it was passed and its own local automatic and static variables (plus any globally accessible data). For example, while func2 is executing, it can not access func1's local variables, unless of course func1 passes references to its local variables in the argument list to func2.

goto top

Page Table

Figure 8 illustrates the relationship between the memory map, page table, and real memory. The page table expands or contracts, mapping more or less real memory, as both the stack and heap segments change size.

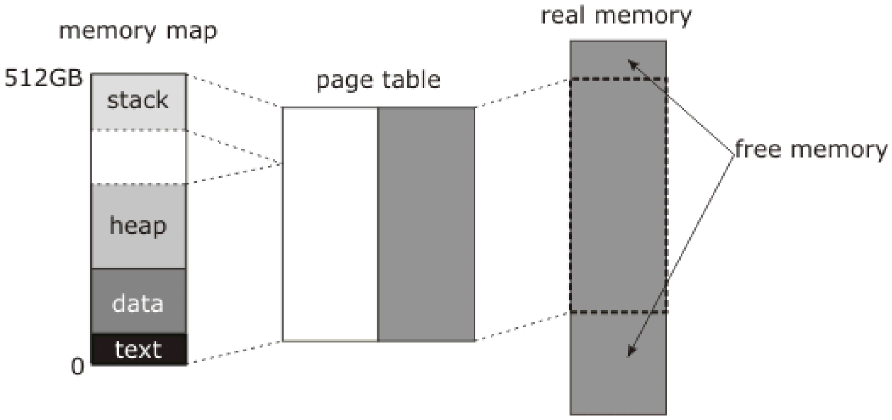


Fig. 8 Memory map, page table, and real memory.

Suppose each page table entry comprises one 64-bit number representing a virtual memory address and another 64-bit number representing a real memory address, for a total of 16 bytes per entry. To map each byte of a 200MB process, for instance, would require a 3200MB page table. Obviously impractical. Instead of mapping individual bytes, the page table maps larger chunks of virtual memory called *pages* (hence the name). The corresponding increment of real memory is called a *page frame*.

Page size is architecture-specific and often configurable. It is always some power of two. The AMD Opteron processors in the AICT Linux cluster use 4KB. Accordingly, the page table for a 200MB process would be only 800KB.

Over 128 million pages span the 512GB virtual memory address range. They are numbered consecutively with a virtual page number (VPN). Each page table entry maps a virtual page from the process memory map to a page frame in real memory, that is, from a VPN to a page frame number (PFN). The VPN is calculated from the virtual memory address by dividing it by the page size (bit-shifting the address to the right). The specific byte is located as an offset from the start of the page.

goto top

Libraries

Linking a *static* library into a program integrates the library's text and data segments into the ELF program file. As a result, two programs linked with the same static libraray both map the library into real memory. See Figure 9.

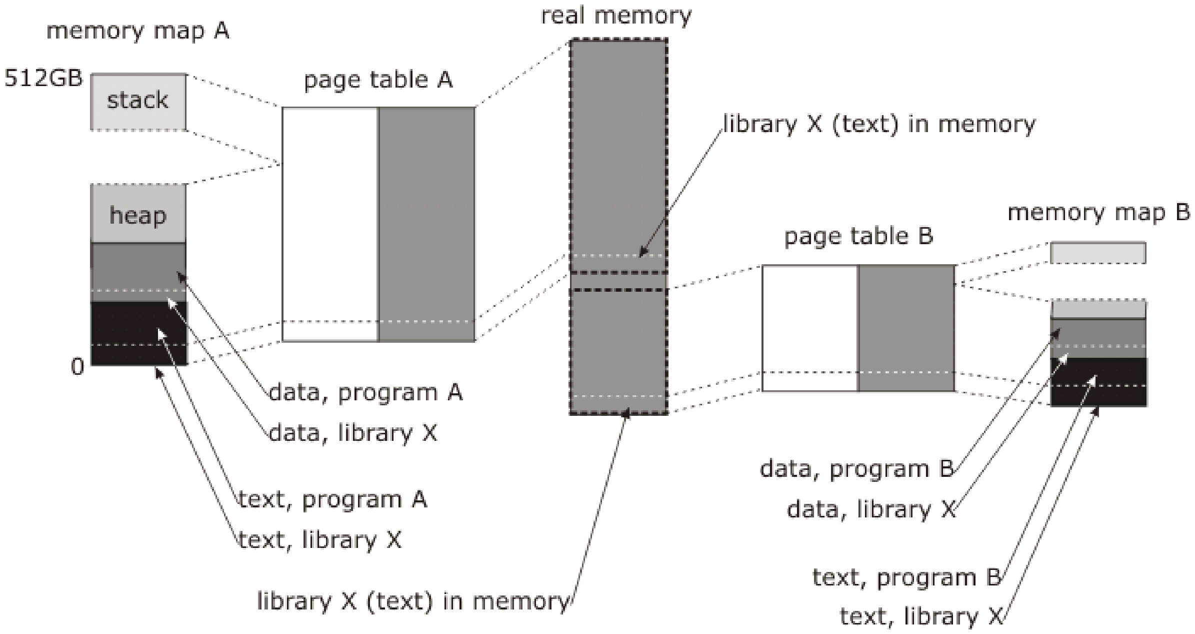


Fig. 9 Two programs linked with the same static library.

Static library files in the ELF format have the `.a` (for archive) name extension. The ELF format also supports *dynamic shared* libraries. These have the `.so` (for shared object) name extension.

Two aspects of a dynamic shared library distinguish it from a static library. First, only the name of the library is recorded in the ELF program file, no text or data, resulting in a smaller executable. Second, only one copy of the library is loaded into real memory. This conserves real memory and also speeds up the loading of programs that are linked with the same dynamic shared library. Two such programs are illustrated in Figure 10.

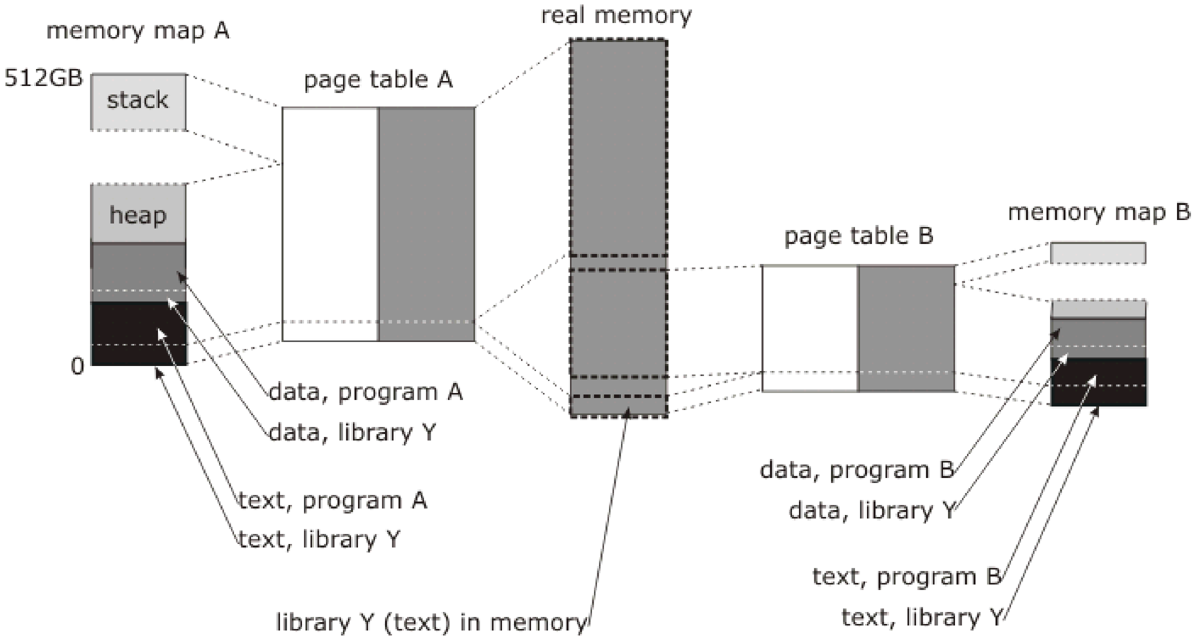


Fig. 10 Two programs linked with the same dynamic shared library.

When the first of the two programs is executed, the system finds the library and updates the page table by mapping the library's text and data into real memory. When the second program is executed, the page table entries referring to the library text are mapped to existing real memory. The library text segment can be shared like this because its permissions, like all text segments, are read-execute.

Initially, the library's read-only data segment is also shared. However, when a program attempts to update the segment, a private copy is made and the program's page table is mapped to the copy, a strategy known as COW (copy on write). Figure 10 shows each program with its own copy of the library data segment.

goto top

Memory Limits

In the section Programs and Processes, we mentioned that a process inherits certain memory limits from its parent, which is usually a shell. In the case of the AICT cluster, the bash and tcsh shells are configured to impose a *soft limit* on the size of the stack segment of 10MB. The other supported memory parameters, data segment, total virtual memory, and total real memory, are unrestricted.

Although the text, data, heap, and stack segments have been depicted with roughly equal size, in reality, the memory map of a typical high performance computing application that features one or more large arrays will likely be dominated by either the bss, stack, or heap segment. If the large arrays are stack-based, the potential exists for exceeding the inherited stack size limit. When this happens, the process is terminated abnormally with a "Segmentation violation" signal.

To avoid this outcome, the soft limit can be increased using a builtin shell command. For example, `ulimit -s 20480` under bash (see `help ulimit`) or `limit stacksize 20480` under tcsh (see `man tcsh`) sets the stack size limit to 20MB. Soft limits can be increased up to the configured *hard limit*. On the AICT cluster, the hard limit for the size of the stack segment is "unlimited," meaning unrestricted. It is important to note that the new limit takes effect only for processes launched from the particular shell in which the limit was increased.

> *Be aware that the soft and hard limits on shell resources vary greatly among systems.*

Many AICT cluster nodes have 10GB of RAM installed. Of this, approximately 1000MB is reserved by the operating system to maintain process information, including the page tables. An additional 1GB of slow secondary storage on disk is configured as *swap space* (or, more accurately, paging space). The sum of the available RAM plus the configured swap space amounts to about 9800MB (determined empirically), and represents the total memory, real and virtual, that can be mapped by all the processes on such a node. The term for this quantity is *SWAP* (also known as logical swap space).

To clarify this with a diagram, the stack size soft limit for a process running interactively (on the AICT cluster head node) is illustrated in Figure 11.

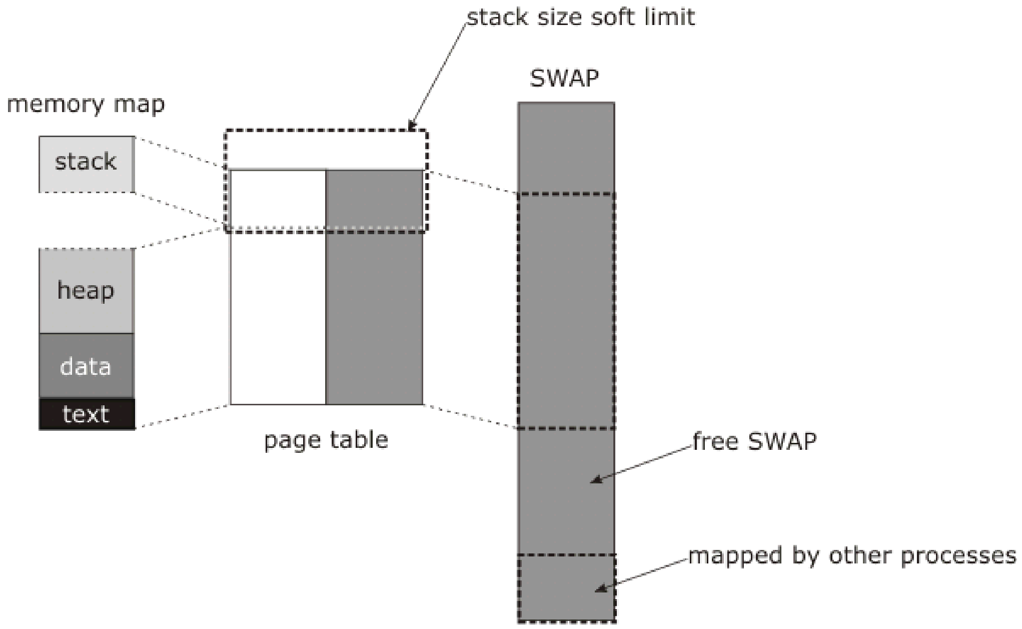


Fig. 11 Stack size soft limit imposed on an interactive process.

The soft limit is represented as a bounding box surrounding that part of the page table mapping the stack segment. Making the soft limit "unlimited" effectively removes the box. Meanwhile, expansion of the heap segment is limited only by the available freee SWAP.

Since a batch job is a shell script, it may need to include an appropriate limit command to increase the stack size limit before invoking the computational program. Moreover, batch jobs are assigned to nodes that have enough free SWAP to satisfy the requested process virtual memory (pvmem). To ensure that a job does not exceed its pvmem specification, the batch system reduces the virtual memory soft limit of the associated shell, which is initially set to "unlimited," to match the pvmem value. Whenever a soft limit is modified, the hard limit is automatically reset to be identical. Thus, a job is prevented from subsequently increasing its virtual memory soft limit once it has been adjusted by the batch system.

Accordingly, a job whose size (text+data+bss) exceeds the virtual memory limit is not allowed to start running. While a job trying to expand at runtime beyond its virtual memory limit is terminated abnormally with a "Segmentation violation" signal. To exit gracefully, C programs can test the return value from the `malloc()` function for a `NULL` pointer and Fortran 90/95 programs can test the `STAT` parameter of the `ALLOCATE` statement for a nonzero value.

For a process running in batch mode on the AICT cluster, the memory limits are shown in Figure 12.

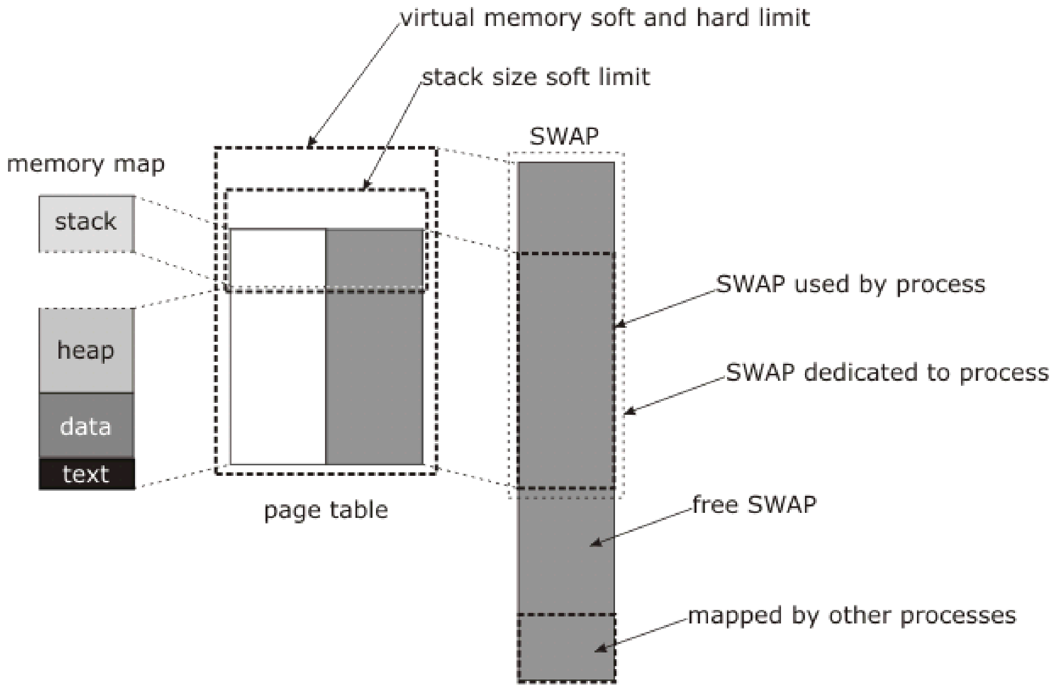


Fig. 12 Stack size and virtual memory limits imposed on a batch job process.

Because the page table represents the virtual memory footprint of a process, the virtual memory soft limit is depicted as a bounding box surrounding the page table. *Note that under certain conditions, the virtual memory limit can be violated by stack-based data even if the stack does not exceed the stack size limit*.

A job's pvmem specification translates as the amount of SWAP that is dedicated to the process by the batch job system. The process may actually use less. Nevertheless, when considering a node for scheduling another job, the batch system accounts for the total SWAP dedicated to—not used by—all the other jobs currently running on the node. This could lead to waste through unused resources unless the pvmem estimates are accurate.

goto top

Memory Allocation

Thus far, our graphical representations of the page table have implied that virtual memory pages are always mapped to real page frames. However, because some pages may never be used, it is more efficient for the operating system to defer the mapping until the page is actually referenced, a technique known as *demand paging*.

For example, a common practice in Fortran 77 is to compile a program with the largest anticipated static arrays and use the same executable to perform calculations with varied array extents. In this case, a sufficient number of virtual pages are always *reserved* in the page table to completely accommodate the large arrays. However, only those pages containing array elements actually referenced by the program are *allocated* page frames in real memory. Those pages that are never referenced are not allocated page frames. The sum of all the allocated page frames is called the *resident set size* (RSS). As shown in Figure 13, RSS will be less than or equal to the process's virtual memory footprint.

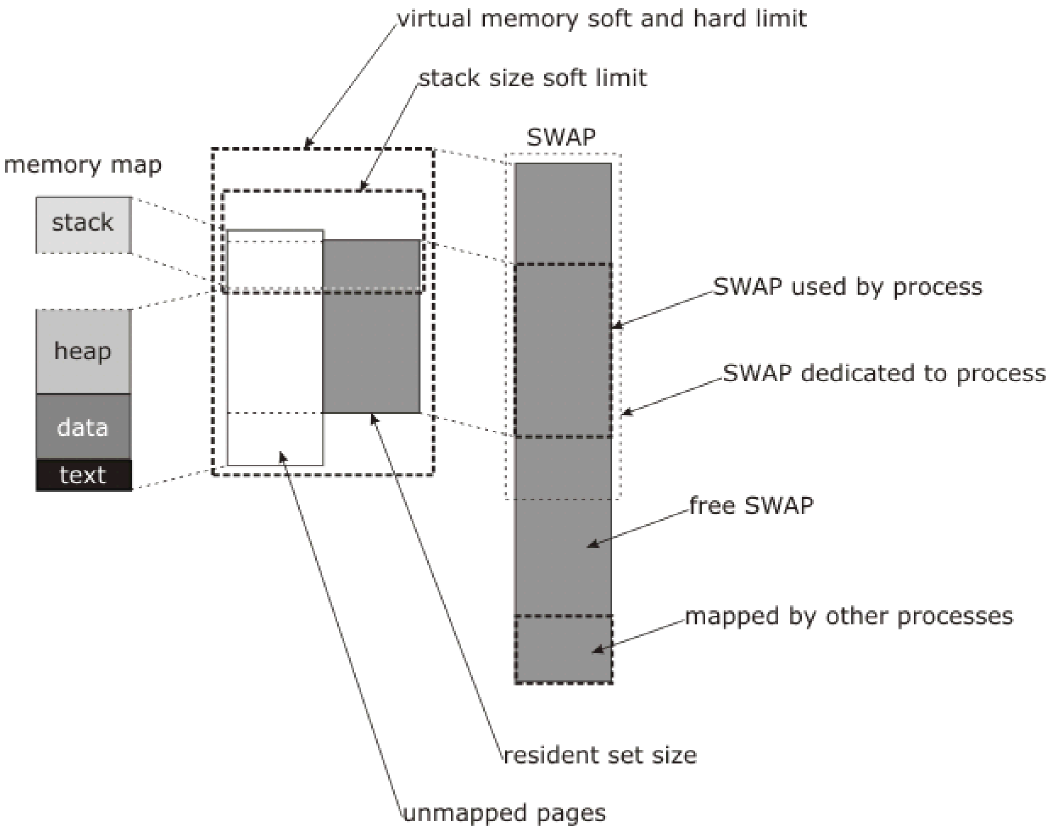


Fig. 13 A more realistic version of Figure 12 depicting resident set size.

When a process references a virtual page for the first time, a *page fault* occurs. Linux responds by acquiring a real page frame from the pool of free pages and allocating it to the virtual page in the process's page table. In addition, the 512 most recent page table entries are cached in the Opteron CPU's *translation lookaside buffer* (TLB). Referencing a page that has a TLB entry is faster than going through the page table in the kernel's memory.

For high performance applications, the ideal is to have every memory allocation satisfied from the available RAM. However, if so much memory is being used that RAM is depleted, a frame is "stolen" from another process. The contents of the stolen frame are transferred to the swap space on disk and the owner's page table updated. In other words, the page is *swapped out* or, more accurately, *paged out*. Frames in swap space can not be utilized directly. Therefore, if a process references such a page (called a major page fault), it has to be *swapped in* first, causing a page from some other process to be swapped out. As more memory is demanded, more swap space is utilized, which causes more swapping. Because disk access is slow, the result is significantly reduced performance for all processes. Eventually, if even swap space is exhausted, processes are killed in order to harvest their page frames. At this point, the node quickly becomes unusable. To prevent this, the batch system implements the pvmem specification to ensure that the total SWAP on a node is never exceeded.

goto top

Implementation Details

Finally, let us analyze a real memory map to discover how some of the preceding concepts are actually implemented. Under Linux, the memory map for a process with process id *pid* is available in the read-only file `/proc/pid/maps` (see `man proc`). Consider the following C program that incorporates data of various storage class and scope.

```
/**
 * map.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define NSIZE   (5*(1<<20))
#define SLEEPT  10

long  gx[NSIZE];

int
main (int argc, char *argv[])
{
  char  c[NSIZE];
  int  *px = malloc (NSIZE*sizeof(int));

  for (int i=0; i<NSIZE; i++) {
    gx[i] = (long)i;
    px[i] = i;
    c[i]  = 'c';
  }
  printf ("address of gx[0] = %012p\n", &gx[0]);
  printf ("address of px[0] = %012p\n", &px[0]);
  printf ("address of  c[0] = %012p\n", &c[0]);

  printf ("memory map file: /proc/%d/maps\n", getpid());
  printf ("sleeping %d...", SLEEPT);
  fflush (NULL);
  sleep (SLEEPT);

  free (px);

  printf ("\ndone\n");
  exit (EXIT_SUCCESS);
}
```

A label has been added to each line of the maps file output in the terminal display below to make subsequent references more convenient.

```
$ pgcc -c9x -o map map.c
$ ls -l map
-rwxr-xr-x  1 esumbar uofa 8176 Dec  1 09:24 map
$ size map
   text    data    bss         dec         hex        filename
   1768    724     41943072    41945564    28009dc   map
$ ./map
address of gx[0] = 0x0000500bc0
address of px[0] = 0x2a9557a010
address of  c[0] = 0x7fbfaff460
memory map file: /proc/23114/maps
sleeping 10...^Z
[2]+  Stopped                 ./map
$ cat /proc/23114/maps
(a) 00400000-00401000 r-xp ... /scratch/esumbar/map
(b) 00500000-00501000 rw-p ... /scratch/esumbar/map
(c) 00501000-02d01000 rwxp ...
(d) 2a95556000-2a95557000 rw-p ...
(e) 2a95578000-2a9697c000 rw-p ...
(f) 330c300000-330c315000 r-xp ... /lib64/ld-2.3.4.so
(g) 330c414000-330c416000 rw-p ... /lib64/ld-2.3.4.so
(h) 330c500000-330c62a000 r-xp ... /lib64/tls/libc-2.3.4.so
(i) 330c62a000-330c729000 ---p ... /lib64/tls/libc-2.3.4.so
(j) 330c729000-330c72c000 r--p ... /lib64/tls/libc-2.3.4.so
(k) 330c72c000-330c72f000 rw-p ... /lib64/tls/libc-2.3.4.so
(l) 330c72f000-330c733000 rw-p ...
(m) 330c800000-330c885000 r-xp ... /lib64/tls/libm-2.3.4.so
(n) 330c885000-330c984000 ---p ... /lib64/tls/libm-2.3.4.so
(o) 330c984000-330c986000 rw-p ... /lib64/tls/libm-2.3.4.so
(p) 7fbfafe000-7fc0000000 rwxp ...
(q) ffffffffff600000-ffffffffffe00000 ---p ...
$ fg
./map

done
```

Memory addresses are customarily expressed as hex (hexidecimal) numbers, using a base of 16 rather than 10, with digits 0 through 9 and A (10) through F (15). Accordingly, a page size of 4KB is expressed as 1000_{16} or 0x1000. Notice how each of the addresses on the left of the maps display conclude with three zeroes, indicating that they are multiples of the page size.

Unfortunately, the maps file does not identify the memory regions as "text," "data," and so on. However, in many cases, these labels can be inferred. For example, region (a) is a read-execute segment associated with the ELF program file. This must be the text segment. Notice that one whole page is reserved even though the actual text occupies only 1768 bytes. Also note that the text segment starts at address 0x400000 (4MB) not zero as we have been assuming. This will be the case for all processes and is intended to ensure that zero is treated as an invalid memory address if it is ever used inadvertently.

Incidentally, the "p" in the permission bits `r-xp` means that the region is private. Processes that manage shared memory with the `shmat()` or `mmap()` functions will have one or more regions displaying an "s."

Region (b) is the read-write data segment. Whereas the text segment ends at 0x401000 (4MB+4KB), the data segment does not start until 0x500000 (5MB). Hence, adjacent segments are not necessarily contiguous. (Likewise, the mapped real page frames are also non-contiguous.) Judging by the address of the static array *gx*, the bss segment starts at or near 0x500bc0 within region (b) and continues into region (c), which has a span of exactly 0x2800000 (40MB).

Next come two non-contiguous regions, (d) and (e). The extent of region (e) is 0x1404000 (20MB+16KB), enough to accommodate the dynamically allocated memory. Indeed, the value of the pointer *px* falls near the beginning of this region. Region (d), on the other hand, is a mystery. Based on its proximity to region (e), it may have something to do with the management of dynamic memory for this process.

Regions (f) and (g) map the text and data segments, respectively, of the runtime dynamic loader. This is the code that the operating system invokes first when the program is exeucted. It finds and loads the dynamic shared libraries linked to the program before calling main. Although no libraries were specified during compilation, the program implicitly references the standard C (libc.so) and math (libm.so) libraries as the output from the ldd command confirms.

```
$ ldd map
        libc.so.6 => /lib64/tls/libc.so.6 (0x000000330c500000)
        libm.so.6 => /lib64/tls/libm.so.6 (0x000000330c800000)
```

```
            /lib64/ld-linux-x86-64.so.2 (0x000000330c300000)
```

Practically all C programs need the functionality provided by these libraries to implement `printf()` and so on. Consequently, the Portland compiler links them by default. Moreover, because of their ubiquity, they are implemented as dynamic shared libraries.

As such, regions (h) through (k) represent libc's text, internal data, read-only data, and read-write data, respectively, while regions (m) through (o), libm's text, internal data, and read-write data. (l) is another mytery region.

Finally, the stack segment is mapped in region (p) whose size is 0x502000 (5MB+8KB). Observe that the upper limit of virtual memory 0x7fffffffff does not exactly coincide with the top of the stack. Also, the starting address of variable *c* is close to the "bottom" of the stack. Region (q) (8MB) is not directly accessible to the process. However, it is used indirectly when requesting services from the kernel, for example, to open a file.

goto top

References

1. *Understanding the Linux Kernel*, by Daniel P. Bovet and Marco Cesati, ©2001 O'Reilly
2. *Linkers and Loaders*, by John R. Levine, ©2000 Morgan Kaufmann
3. *System V Application Binary Interface: AMD64 Architecture Processor Supplement, Draft Version 0.96*, by Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell

goto top

$Id: mem.html 299 2010-05-12 16:16:29Z esumbar $