Operating Systems & Systems Programming I

Memory Management

Keith Bugeja / Joshua Ellul / Alessio Magro / Kevin Vella

April 30, 2018

Department of Computer Science
Faculty of ICT, University of Malta

Overview

Overview

Introduction

- Process address space
- Pages and paging
- Memory regions

Memory allocation

- Primitives
- Manipulation
- Common problems

Advanced memory allocation

- Anonymous memory mappings
- Shared memory

Memory Management

Process address space

Linux *virtualises* physical memory

- Physical memory not directly accessible by processes
- Processes are presented with a logical view of memory
- Kernel associates a virtual address space with each process:

  Linear starts at zero and contiguously increases, and

  Flat directly accessible without the need for segmentation

In memory management, the page is the smallest addressable unit of memory that the memory management unit (MMU) can manage:

In memory management, the page is the smallest addressable unit of memory that the memory management unit (MMU) can manage:

- The virtual address space is divided into pages, the size of which is determined by the machine architecture (e.g. 4 KB for 32-bit and 8 KB for 64-bit systems)

Pages and paging

In memory management, the page is the smallest addressable unit of memory that the memory management unit (MMU) can manage:

- The virtual address space is divided into pages, the size of which is determined by the machine architecture (e.g. 4 KB for 32-bit and 8 KB for 64-bit systems)
- A 32-bit address space contains around 1 million 4 KB pages
    - Not all pages necessarily correspond to anything and thus are either *valid* or *invalid*
    - A valid page is associated with an actual page of data, either in physical memory (RAM) or on secondary storage
    - Access to an invalid (unmapped) page results in a segmentation violation

Pages and paging

When a page is on secondary storage, attempts to access it causes the memory management unit to generate a *page fault*

- Kernel intervenes and transparently *pages in* the data from secondary storage to physical memory
- The kernel may have to move data out of memory to make space for data paged in - a process called *paging out*
    - To minimise subsequent page ins, the kernel attempts to page out (evict) data that is least likely to be used in the near future

Page sharing

Multiple pages of virtual memory, even in different virtual address spaces owned by different processes, may map to a single physical page

- virtual address spaces may share data in physical memory (e.g. many processes using the standard C library map it into their virtual address space, even though only one copy exists in physical memory).
- shared data may be read-only, writable or both read and writable
- basis for copy-on-write (COW); MMU intercepts write operation and raises exception - kernel in response creates a new copy of the page for the writing process

Memory regions

Kernel arranges pages into blocks that share certain properties (e.g. access permissions); these blocks are called *mappings* or *memory regions*.

text segment contains process program code, string literals, constant variables and other read-only data. Mapped directly from object file

stack contains process execution stack, which grows and shrinks dynamically

data segment/heap contains the process dynamic memory

bss segment contains uninitialised global variables (all zeros, according to C standard)

Hint

Examine `/proc/self/maps`

Memory allocation

Allocating dynamic memory

Foundation of memory management is the allocation, use and eventual return of *dynamic memory*

- Allocated at runtime (not compile time)
- Sizes may be unknown until the moment of allocation

The C interface for allocating dynamic memory is `malloc`:

```c
#include <stdlib.h>

void * malloc (size_t size);
```

The C interface for allocating dynamic memory is `malloc`:

```c
#include <stdlib.h>

void * malloc (size_t size);
```

Allocates `size` bytes of memory and returns a pointer to the allocated region

- contents of memory are undefined!

Allocating dynamic memory

The C interface for allocating dynamic memory is `malloc`:

```c
#include <stdlib.h>

void * malloc (size_t size);
```

Allocates `size` bytes of memory and returns a pointer to the allocated region

- contents of memory are undefined!

On failure, `malloc` returns `NULL` and sets `errno` to `ENOMEM`

For instance, to allocate an 8 KB chunk of memory:

For instance, to allocate an 8 KB chunk of memory:

```c
char *buffer = malloc(8192);

if (!buffer)
  perror("malloc");
```

For instance, to allocate an 8 KB chunk of memory:

```c
char *buffer = malloc(8192);

if (!buffer)
  perror("malloc");
```

Note

C automatically promotes pointers to `void` to any other pointer type on assignment.

Allocating dynamic memory

It is important to check the return value from a call to `malloc` and handle error conditions

- many programs employ a `malloc` wrapper that prints an error message and terminates the program if `malloc` returns `NULL`

It is important to check the return value from a call to `malloc` and handle error conditions

- many programs employ a `malloc` wrapper that prints an error message and terminates the program if `malloc` returns `NULL`

```c
void * xmalloc(size_t size)
{
  void *buffer = malloc(size);

  if (!buffer) {
    perror("xmalloc");
    exit(EXIT_FAILURE);
  }

  return buffer;
}
```

Dynamic allocation of arrays, where the size of an element may be fixed but the number of elements is dynamic, is accomplished through the `calloc` function:

```c
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

Allocating arrays

Dynamic allocation of arrays, where the size of an element may be fixed but the number of elements is dynamic, is accomplished through the `calloc` function:

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

Allocates `size * nr` bytes of memory and returns a pointer to the allocated region

- contents of memory are zeroed!

Dynamic allocation of arrays, where the size of an element may be fixed but the number of elements is dynamic, is accomplished through the `calloc` function:

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

Allocates `size * nr` bytes of memory and returns a pointer to the allocated region

- contents of memory are zeroed!

On failure, `calloc` returns `NULL` and sets `errno` to `ENOMEM`

In the same vein as `xmalloc`, it is commonplace to define wrapper methods that allocate and zero memory:

```c
#include <stdlib.h>

void * malloc0 (size_t size) {
  return calloc(1, size);
}
```

In the same vein as `xmalloc`, it is commonplace to define wrapper methods that allocate and zero memory:

```
#include <stdlib.h>

void * malloc0 (size_t size) {
  return calloc(1, size);
}
```

Exercise

Implement `xmalloc0`, a method that allocates and zeroes memory, and terminates the program with an error if the allocation fails.

Resizing allocations

The size of an existing allocation may need to change (shrink or grow), e.g. dynamic arrays or contiguous list of items, strings, etc.

Resizing allocations

The size of an existing allocation may need to change (shrink or grow), e.g. dynamic arrays or contiguous list of items, strings, etc.

C language provides `realloc` for resizing allocations:

```c
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

Resizing allocations

The size of an existing allocation may need to change (shrink or grow), e.g. dynamic arrays or contiguous list of items, strings, etc.

C language provides `realloc` for resizing allocations:

```c
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

Resizes the region pointed by `ptr` to `size` bytes

- returned pointer may or may not be the same as `ptr`
- operation may be costly due to possible copy
- if `size` is 0, effect is the same as freeing memory
- if `ptr` is `NULL`, result is the same as `malloc`

Resizing allocations

The size of an existing allocation may need to change (shrink or grow), e.g. dynamic arrays or contiguous list of items, strings, etc.

C language provides `realloc` for resizing allocations:

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

Resizes the region pointed by `ptr` to `size` bytes

- returned pointer may or may not be the same as `ptr`
- operation may be costly due to possible copy
- if `size` is 0, effect is the same as freeing memory
- if `ptr` is `NULL`, result is the same as `malloc`

On failure, `realloc` returns `NULL` and sets `errno` to `ENOMEM`; the state of the memory pointed by `ptr` is unchanged.

Freeing dynamic memory

Dynamic allocations are permanent parts of the process address space until they are manually freed.

Freeing dynamic memory

Dynamic allocations are permanent parts of the process address space until they are manually freed.

Memory allocated with `malloc`, `calloc` and `realloc` must be returned to the system when no longer in use via `free`:

```c
#include <stdlib.h>

void free (void *ptr);
```

Freeing dynamic memory

Dynamic allocations are permanent parts of the process address space until they are manually freed.

Memory allocated with `malloc`, `calloc` and `realloc` must be returned to the system when no longer in use via `free`:

```c
#include <stdlib.h>

void free (void *ptr);
```

A call to `free` releases the memory at `ptr`

- previously allocated using one of the functions above
- releases entire block of allocated memory
- if `ptr` is `NULL`, `free` does nothing

Example

Time for an example: a function that concatenates two strings into a third

```c
char *concatenate(char *s1, char *s2)
{
  // make sure both are valid strings
  assert(s1 != NULL && s2 != NULL);

  // assume strings are null-terminated
  // get respective lengths
  size_t l1 = strlen(s1),
    l2 = strlen(s2);

  // allocate a zeroed memory chunk the cumulative size of the
  // two strings plus an additional character for null termination
  char *s = calloc(1, l1 + l2 + 1);
  if (!s) {
    perror ("calloc");
    exit(EXIT_FAILURE);
  }

  // copy strings into newly allocated buffer
  strcpy(s, s1);
  strcpy(s + l1, s2);

  return s;
}
```

Example

Next we use the function to concatenate two strings:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

char *concatenate(char *s1, char *s2); // concatenate goes here

int main(int argc, char *argv[]) {
  // define strings s1 and s2
  char *s1 = "abra", *s2 = "cadabra";

  // call concatenate; returns a pointer to the newly concatenated string
  char *s = concatenate(s1, s2);

  // print output
  printf("%s + %s = %s\n", s1, s2, s);

  // free memory allocated by concatenate function
  free(s);
}
```

Running the example program displays the newly concatenated string:

```
$ gcc -o concat concat.c
$ ./concat
abra + cadabra = abracadabra
$
```

Running the example program displays the newly concatenated string:

```
$ gcc -o concat concat.c
$ ./concat
abra + cadabra = abracadabra
$
```

Exercise

Change the program to get the source strings `s1` and `s2` from the command line arguments.

There are a number of common pitfalls programmers fall into when manually managing memory:

- memory leak
- use-after-free
- dangling pointer

Memory leaks are among the most common and detrimental mishaps in C programming:

- typically triggered by a missing invocation to `free`
- memory is never returned to the system (during the lifetime of the process)
- process loses its only reference to the memory and becomes unable to access it again

Example

This program exhibits memory leaks:

```
...

// concatenate prototype
char *concatenate(char *s1, char *s2);

char *concatenate4(char *s1, char *s2, char *s3, char *s4) {
  char *s12 = concatenate(s1, s2); // s12 = s1s2
  char *s34 = concatenate(s3, s4); // s34 = s3s4
  char *s = concatenate(s12, s34); // s = s12s34

  return s;
}

int main(int argc, char *argv[]) {
  char *s1 = "You ", *s2 = "shall ",
       *s3 = "not ", *s4 = "pass.";

  // concatenate the four strings
  char *s = concatenate4(s1, s2, s3, s4);

  // print output
  printf("%s + %s + %s + %s = %s\n", s1, s2, s3, s4, s);

  // free memory block returned by concatenate4
  free(s);
}
```

Memory leaks

The program leaks memory in the `concatenate4` function:

- `concatenate` returns allocated memory for `s12`, `s34` and `s`
- memory pointed to by `s12` and `s34` is never freed
- being locals, `s12` and `s34` are lost once `concatenate4` returns
- memory cannot be freed once `s12` and `s34` go out of scope

Use-after-free

Use-after-free occurs when a block of memory is freed and then subsequently accessed:

- once `free` is called on a block of memory, a program should never access its contents again

Example

Use-after-free is shown in the example below, where the order of `printf` and `free` is switched:

```c
...
char *concatenate(char *s1, char *s2); // concatenate goes here

int main(int argc, char *argv[]) {
  // define strings s1 and s2
  char *s1 = "abra", *s2 = "cadabra";

  // call concatenate; returns a pointer to the newly concatenated string
  char *s = concatenate(s1, s2);

  // free memory allocated by concatenate function
  free(s);

  // print output
  printf("%s + %s = %s\n", s1, s2, s);
}
```

Dangling pointers

Dangling pointers are non-NULL pointers that point to invalid (or deallocated) memory

- become invalid due to deallocation (e.g. calls to `free`, references going out of scope, etc.)
- pointer is not set to `NULL`

Example

Example snippet demonstrating use-after-free of a dangling pointer:

```c
...
// print s if not a NULL pointer
void print_string(char *s) {
  if (s) printf("%s\n", s);
}

int main(int argc, char *argv[]) {
  char *s1 = "abra", *s2 = "cadabra";
  char *s = concatenate(s1, s2);
  printf("%s + %s = %s\n", s1, s2, s);

  // show pointer before call to free
  printf("Pointer before free is %p\n", s);

  // free memory allocated by concatenate function
  free(s);

  // show pointer after call to free
  printf("Pointer after free is %p\n", s);

  // pass the dangling pointer to function
  print_string(s);
}
```

Running the example shows that the value of the pointer before and after the call to `free` remains the same:

```
$ ./dangling
abra + cadabra = abracadabra
Pointer before free is 0x56446c8aa260
Pointer after free is 0x56446c8aa260

$
```

Running the example shows that the value of the pointer before and after the call to `free` remains the same:

```
$ ./dangling
abra + cadabra = abracadabra
Pointer before free is 0x56446c8aa260
Pointer after free is 0x56446c8aa260

$
```

Thus, `print_string` has no way of determining whether the pointer is valid or dangling.

- calling `printf` on the freed memory block results in undefined behaviour

Detecting memory errors

On Linux, a number of tools, such as *Valgrind*, are available to help us detect and debug memory errors in our programs

- the *Valgrind* tool suite can automatically detect many memory management and threading bugs
- can perform detailed profiling to help spot bottlenecks in your programs
- to install, open a terminal session and type:

```
$ sudo apt install valgrind
```

Valgrind Examples

Let's run `valgrind` on the concatenation program that suffered from memory leaks:

```
$ valgrind ./concat4
...
==39702== Command: ./concat4
==39702==
You  + shall  + not  + pass. = You shall not pass.
==39702==
==39702== HEAP SUMMARY:
==39702==     in use at exit: 21 bytes in 2 blocks
==39702==   total heap usage: 4 allocs, 2 frees, 1,065 bytes allocated
==39702==
==39702== LEAK SUMMARY:
==39702==    definitely lost: 21 bytes in 2 blocks
==39702==    indirectly lost: 0 bytes in 0 blocks
==39702==      possibly lost: 0 bytes in 0 blocks
==39702==    still reachable: 0 bytes in 0 blocks
==39702==         suppressed: 0 bytes in 0 blocks
```

Valgrind Examples

And again on the program with the dangling pointer:

```
$ valgrind ./dangling
...
==40010== Command: ./dangling
==40010==
abra + cadabra = abracadabra
Pointer before free is 0x521c040
Pointer after free is 0x521c040
...
==40010==    by 0x1088A8: print_string
==40010==    by 0x10894F: main
==40010==  Address 0x521c040 is 0 bytes inside a block of size 12
  ↪  free'd
==40010==    at 0x4C30D3B: free
==40010==    by 0x10892B: main
==40010==  Block was alloc'd at
==40010==    at 0x4C31B25: calloc
==40010==    by 0x1089CF: concatenate
==40010==    by 0x1088E3: main
...
```

Managing the Data Segment

Unix systems have traditionally provided interfaces to allow direct management of the data segment:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

Managing the data segment

Unix systems have traditionally provided interfaces to allow direct management of the data segment:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

Inheritance from legacy systems where the data segment and the heap lived in the same segment:

- heap grows upward from the bottom of segment
- stack grows downward from the top of segment
- like of demarcation between them is called the *break* or *break point*

Unix systems have traditionally provided interfaces to allow direct management of the data segment:

```c
#include <unistd.h>

int brk (void *end);
```

A call to `brk` sets the break point address to the value specified by `end`

- if successful, the call returns 0
- on failure, it returns -1 and sets `errno` to `ENOMEM`

Unix systems have traditionally provided interfaces to allow direct management of the data segment:

```c
#include <unistd.h>

void * sbrk (intptr_t increment);
```

A call to `sbrk` changes the break point address by the offset specified in `increment`:

- an increment of 0 returns the current break point

Example

The following example shows how calls to `malloc` affect the current break address:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  const int blocks = 50;
  const int block_size = 32 * 1024;
  char *block_list[blocks];

  for (int j = 0; j < blocks; ++j)
  {
    block_list[j] = malloc(block_size);
    printf("Current break is at %p\n", sbrk(0));
  }

  for (int j = 0; j < blocks; ++j)
    free (block_list[j]);
}
```

Example

The program outputs the current break address after each 32 KB allocation:

```
$ ./sbrk
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec866000
...
Current break is at 0x5588ec9a6000
Current break is at 0x5588ec9a6000
Current break is at 0x5588ec9ce000
$
```

Example

The program outputs the current break address after each 32 KB allocation:

```
$ ./sbrk
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec83e000
Current break is at 0x5588ec866000
...
Current break is at 0x5588ec9a6000
Current break is at 0x5588ec9a6000
Current break is at 0x5588ec9ce000
$
```

Note

The use of `malloc` should be preferred over `brk` and `sbrk` for its convenient and portable memory allocation interface.

`malloc` is classically implemented using an algorithm called *buddy memory allocation scheme*

- data segment is divided into a series of power-of-2 partitions
- allocations are satisfied by returning the partition that is the closest fit to the requested size
- memory is freed by marking the partition as *free*
- adjacent free partitions can be coalesced into a single larger partition
- if the top of the heap is free, the system can use `brk` to lower the break point, shrinking the heap and returning memory to the kernel

The buddy memory allocation scheme has the advantage of being fast and simple; however:

- introduces internal and external fragmentation
- results in inefficient use of memory or failed memory allocations
- allows memory allocations to *pin* one another, preventing the heap from shrinking after large memory sections are freed

Anonymous Memory Mappings

To avoid the problems with the buddy memory allocation scheme, `glibc` does not use the heap for large allocations:

- creates an anonymous memory mapping to satisfy a large allocation request
- large zero-filled blocks of memory, ready for use

Anonymous memory mapping

Allocating memory via anonymous mappings has several benefits:

- no fragmentation concerns; memory is returned to the system as soon as mapping is unmapped
- mappings are resizable;
- exist in a separate memory mapping, removing any need to manage the global heap

Anonymous memory mapping

Allocating memory via anonymous mappings has several benefits:

- no fragmentation concerns; memory is returned to the system as soon as mapping is unmapped
- mappings are resizable;
- exist in a separate memory mapping, removing any need to manage the global heap

There are also two downsides to using anonymous memory mappings rather than the heap:

- each mapping is an integer multiple of the system page size, possibly resulting in wasted space
- creating a new memory mapping incurs more overhead than satisfying an allocation from the heap

The following example demonstrates the behaviour of `malloc` vis-a-vis small and large allocations:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  const int blocks = 4;
  const int block_size_heap = 64 * 1024;
  const int block_size_amm = block_size_heap * 1024;

  char *block_list[blocks];

  printf("Allocation size: %d\n", block_size_heap);
  for (int j = 0; j < blocks; ++j) {
    block_list[j] = malloc(block_size_heap);
    printf("Address : %p, Break : %p\n", block_list[j], sbrk(0));
  }

  printf("Allocation size: %d\n", block_size_amm);
  for (int j = 0; j < blocks; ++j) {
    free(block_list[j]);
    block_list[j] = malloc(block_size_amm);
    printf("Address : %p, Break : %p\n", block_list[j], sbrk(0));
  }

  for (int j = 0; j < blocks; ++j)
    free (block_list[j]);
}
```

Example

The program outputs the address of the newly allocated block together with the current program break address:

```
$ ./malloc
Allocation size: 65536
Address : 0x561a835a8670, Break : 0x561a835c9000
Address : 0x561a835b8680, Break : 0x561a835c9000
Address : 0x561a835c8690, Break : 0x561a835f9000
Address : 0x561a835d86a0, Break : 0x561a835f9000
Allocation size: 67108864
Address : 0x7f882981f010, Break : 0x561a835f9000
Address : 0x7f882581e010, Break : 0x561a835f9000
Address : 0x7f882181d010, Break : 0x561a835f9000
Address : 0x7f881d81c010, Break : 0x561a835c9000
$
```

Example

The program outputs the address of the newly allocated block together with the current program break address:

```
$ ./malloc
Allocation size: 65536
Address : 0x561a835a8670, Break : 0x561a835c9000
Address : 0x561a835b8680, Break : 0x561a835c9000
Address : 0x561a835c8690, Break : 0x561a835f9000
Address : 0x561a835d86a0, Break : 0x561a835f9000
Allocation size: 67108864
Address : 0x7f882981f010, Break : 0x561a835f9000
Address : 0x7f882581e010, Break : 0x561a835f9000
Address : 0x7f882181d010, Break : 0x561a835f9000
Address : 0x7f881d81c010, Break : 0x561a835c9000
$
```

The larger 64 MB allocations all hold addresses larger than the current program break

- allocated as anonymous memory mappings outside the global heap

Example

Running the program through `strace` provides us with more insight:

```
$ strace ./malloc
...
write(1, "Allocation size: 65536\n", 23Allocation size: 65536) = 23
write(1, "Address : 0x5566bad8c680, Break "..., 49Address : 0x5566bad8c680, Break : 0x5566bad9d000) = 49
brk(0x5566badcd000)
...
write(1, "Allocation size: 67108864\n", 26Allocation size: 67108864) = 26
mmap(NULL, 67112960, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3d43c80000
write(1, "Address : 0x7f3d43c80010, Break "..., 49Address : 0x7f3d43c80010, Break : 0x5566badcd000) = 49
...
munmap(0x7f3d43c80000, 67112960)        = 0
...
exit_group(0)                           = ?
+++ exited with 0 +++
```

Example

Running the program through `strace` provides us with more insight:

```
$ strace ./malloc
...
write(1, "Allocation size: 65536\n", 23Allocation size: 65536) = 23
write(1, "Address : 0x5566bad8c680, Break "..., 49Address : 0x5566bad8c680, Break : 0x5566bad9d000) = 49
brk(0x5566badcd000)
...
write(1, "Allocation size: 67108864\n", 26Allocation size: 67108864) = 26
mmap(NULL, 67112960, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3d43c80000
write(1, "Address : 0x7f3d43c80010, Break "..., 49Address : 0x7f3d43c80010, Break : 0x5566badcd000) = 49
...
munmap(0x7f3d43c80000, 67112960)        = 0
...
exit_group(0)                           = ?
+++ exited with 0 +++
```

While the smaller allocations use calls to `brk`, the larger ones use the `mmap` and `munmap` system calls.

The mmap function creates a new mapping in the virtual address space of the calling process, while munmap undoes this mapping:

```c
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

The **mmap** function creates a new mapping in the virtual address space of the calling process, while **munmap** undoes this mapping:

```c
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

A call to **mmap** to establish an anonymous memory mapping requires `fd` be set to -1 and `offset` to 0.

- These arguments are used for memory-mapped files (more on this later)

The **mmap** function creates a new mapping in the virtual address space of the calling process, while **munmap** undoes this mapping:

```c
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

`start` determines the start address of the mapping

- setting this value to `NULL` lets the kernel choose the address at which to create the mapping

The **mmap** function creates a new mapping in the virtual address space of the calling process, while **munmap** undoes this mapping:

```c
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

`length` determines the size of the mapping

- value must be greater than 0

The **mmap** function creates a new mapping in the virtual address space of the calling process, while **munmap** undoes this mapping:

```
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

`prot` describes the desired memory protection of the mapping:

- PROT_NONE stipulates that pages may not accessed
- PROT_READ, PROT_WRITE, and PROT_EXEC, bitwise OR'd depending on requirements

Anonymous memory mapping

The mmap function creates a new mapping in the virtual address space of the calling process, while munmap undoes this mapping:

```c
#include <sys/mman.h>

void * mmap (void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t offset);

int munmap (void *start, size_t length);
```

flags describes the behaviour of the map

- for use as an anonymous map (not backed by a file), flags is set to MAP_ANONYMOUS | MAP_PRIVATE

Shared memory

Shared memory

Systems with virtual memory allow physical memory to be mapped into multiple process address spaces

- e.g. one copy of a program or library binary in memory mapped into multiple virtual address spaces

This ability can be exploited for *inter-process communication* (IPC), by establishing a region of memory that can be accessed by two or more processes

- processes read from and write to shared memory as any other allocated memory
- writes are reflected in the address space of each process attaching to the shared memory
- access by several processes may give rise to contention and concurrency problems such as race hazards

Creating shared memory segments

A shared memory segment may be allocated through the System V interface:

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

The function `shmget` returns the identifier of the shared memory segment associated with the value of the argument `key`.

- if a new segment is being created, `size` is the minimum length of the segment rounded up to a multiple of page size; segment is initialised with zeroes
- if a client is attaching to an existing segment, `size` can be 0

Creating shared memory segments

A shared memory segment may be allocated through the System V interface:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

A new segment is always created if the key has value `IPC_PRIVATE`, or no shared memory segment corresponding to `key` exists and `IPC_CREAT` is specified in `shmflg`.

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `cmd` entry may take one of these values (among others):

`IPC_STAT` Copy information from kernel datastructre related to segment `shmid` into `buf`

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `cmd` entry may take one of these values (among others):

`IPC_STAT` Copy information from kernel datastructre related to segment `shmid` into `buf`

`IPC_SET` Write the values of some of the `shmid_ds` data structure to the kernel copy associated with segment `shmid`

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `cmd` entry may take one of these values (among others):

- `IPC_STAT` Copy information from kernel datastructre related to segment `shmid` into `buf`
- `IPC_SET` Write the values of some of the `shmid_ds` data structure to the kernel copy associated with segment `shmid`
- `IPC_RMID` Remove shared memory segment; the segment is deleted only if its reference count is zero

Controlling shared memory segments

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `cmd` entry may take one of these values (among others):

- `SHM_LOCK` Lock pages of shared memory segment in memory, preventing them from being paged out
- `SHM_UNLOCK` Unlock pages of shared memory segment, allowing them to be paged out when needed

Controlling shared memory segments

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The `cmd` entry may take one of these values (among others):

- `SHM_LOCK` Lock pages of shared memory segment in memory, preventing them from being paged out
- `SHM_UNLOCK` Unlock pages of shared memory segment, allowing them to be paged out when needed

Note

These commands may only be executed by superuser.

Controlling shared memory segments

The `shmctl` method provides an interface for controlling aspects of the shared memory segment, including its deletion:

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

A successful call to `shmctl` with `SHM_STAT` returns the identifier of the shared memory segment whose index was given in `shmid`. Other operations return 0 on success.

On error, -1 is returned, and `errno` is set appropriately.

Attaching shared memory segments

Once a shared memory segment has been created, a process may attach it to its address space via the `shmat` interface:

```c
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

The attaching address is specified by `shmaddr`:

Attaching shared memory segments

Once a shared memory segment has been created, a process may attach it to its address space via the `shmat` interface:

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

The attaching address is specified by `shmaddr`:

- if `shmaddr` is NULL, the system chooses a suitable (unused) page-aligned address to attach the segment (recommended)

Attaching shared memory segments

Once a shared memory segment has been created, a process may attach it to its address space via the `shmat` interface:

```c
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

The attaching address is specified by `shmaddr`:

- if `shmaddr` is NULL, the system chooses a suitable (unused) page-aligned address to attach the segment (recommended)
- if `SHM_RND` is specified in `shmflg`, the address is rounded down to the nearest multiple of `SHMLBA`, the low boundary address multiple (always a power of 2, currently equal to page-size)

Attaching shared memory segments

Once a shared memory segment has been created, a process may attach it to its address space via the `shmat` interface:

```c
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

The attaching address is specified by `shmaddr`:

- if `shmaddr` is NULL, the system chooses a suitable (unused) page-aligned address to attach the segment (recommended)
- if `SHM_RND` is specified in `shmflg`, the address is rounded down to the nearest multiple of `SHMLBA`, the low boundary address multiple (always a power of 2, currently equal to page-size)
- otherwise `shmaddr` must be a page-aligned address at which the attach occurs

Once a shared memory segment has been created, a process may attach it to its address space via the `shmat` interface:

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

On success, `shmat` returns the address of the attached shared memory segment

- on error, `(void*) -1` is returned, and `errno` is set to indicate the cause of the error

When done with a shared memory segment, it can be detached from the address space using the `shmdt` interface:

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

Detaching shared memory segments

When done with a shared memory segment, it can be detached from the address space using the `shmdt` interface:

```
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

The `shmaddr` argument is the value that was previously returned by a call to `shmat`

- a successful call to `shmdt` updates members of the associated `shmid_ds` structure
- decrements `shm_nattach` by 1; if the segment is marked for deletion and this value is 0, then the segment is deleted

Detaching shared memory segments

When done with a shared memory segment, it can be detached from the address space using the `shmdt` interface:

```c
#include <sys/types.h>
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

A successful call to `shmdt` returns 0

- on error -1 is returned, and `errno` is set to indicate the cause of the error

Shared memory up-time server

To exemplify the use of shared memory segments, we're going to write a small up-time server:

- creates a shared memory segment with id `0x666`, 256 bytes in size
- every second increments a timer monotonically by 1
- stores a textual representation of the timer in the shared memory buffer

Code listing for the shared memory up-time server:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char *argv[]) {
  int timer = 0; // monotonically increasing timer
  int shmid; // shared memory segment id
  char *shm; // shared memory attached address

  if ((shmid = shmget(0x666, 256, IPC_CREAT | 0666)) == -1) { // create shared memory segment
    perror("shmget() failed:");
    exit(EXIT_FAILURE);
  }

  if ((shm = shmat(shmid, NULL, 0)) == (char *)-1) { // attach to segment
    perror("shmat() failed:");
    exit(EXIT_FAILURE);
  }

  for(;; ++timer) { // loop forever, increment timer by 1 every iteration
    // convert timer to string; buffer is shared memory
    sprintf(shm, "%02d:%02d:%02d", (timer / 3600) % 24, (timer / 60) % 60, timer % 60);
    sleep(1); // sleep for 1 second
  }
}
```

We also provide an up-time client that connects with the shared memory segment to read the current up-time:

- attaches to shared memory segment with id `0x666`
- prints the up-time value set by the up-time server

Shared memory up-time client

Code listing for client:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char *argv[]) {
  int shmid; // shared memory segment id
  char *shm; // shared memory attached address

  if ((shmid = shmget(0x666, 0, 0666)) == -1) { // acquires id of segment with key 0x666
    perror("shmget() failed:");
    exit(EXIT_FAILURE);
  }

  if ((shm = shmat(shmid, NULL, 0)) == (char *)-1) { // attaches to segment
    perror("shmat() failed:");
    exit(EXIT_FAILURE);
  }

  for(;;) { // loop forever
    printf("Timer is : [%s]\n", shm); // print value in shared memory buffer
    sleep(1); // sleep for 1 second
  }
}
```

Run the up-time server in the background `./shm_srv &` and start the client `shm_cli` immediately after:

```
$ ./shm_srv &
$ ./shm_cli
Timer is : [00:00:03]
Timer is : [00:00:04]
Timer is : [00:00:05]
Timer is : [00:00:06]
^C
$ ./shm_cli
Timer is : [00:00:15]
Timer is : [00:00:16]
Timer is : [00:00:17]
^C
$
```

Example

Run the up-time server in the background `./shm_srv &` and start the client `shm_cli` immediately after:

```
$ ./shm_srv &
$ ./shm_cli
Timer is : [00:00:03]
Timer is : [00:00:04]
Timer is : [00:00:05]
Timer is : [00:00:06]
^C
$ ./shm_cli
Timer is : [00:00:15]
Timer is : [00:00:16]
Timer is : [00:00:17]
^C
$
```

Stopping and restarting the client shows that the displayed time is independent of it.

Questions?