Memory Management

Dr. Dipali Meher

Source: The Design of Univ Operating System- Maurice Bach

Introduction

Memory is a basic essential resource, available to a process. The memory management process involves:

- Allocation
- Manipulation
- Release

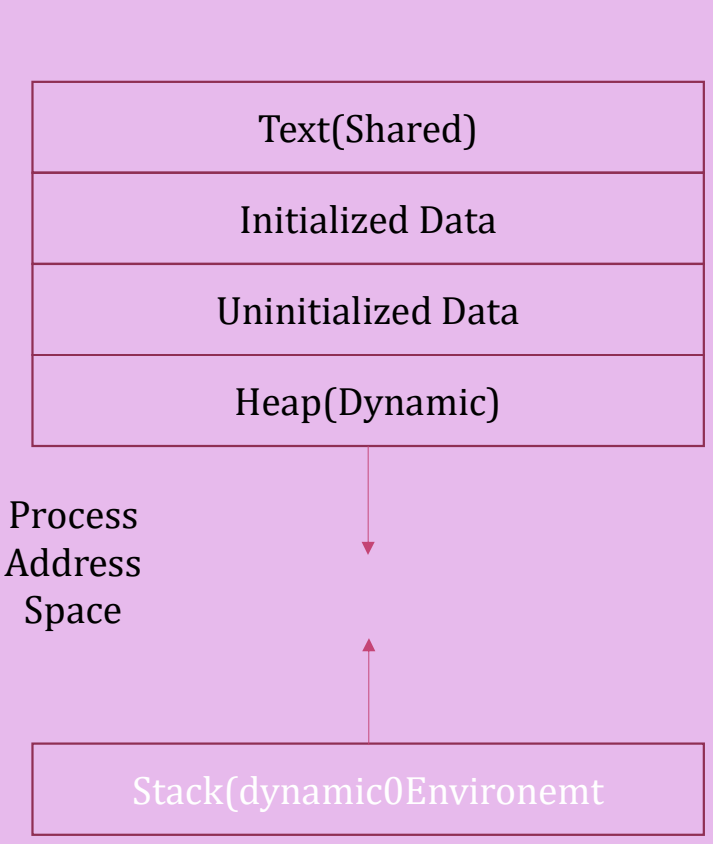
Process Address Space

Linux uses virtual address space instead of direct memory address space. This task is done by kernel.

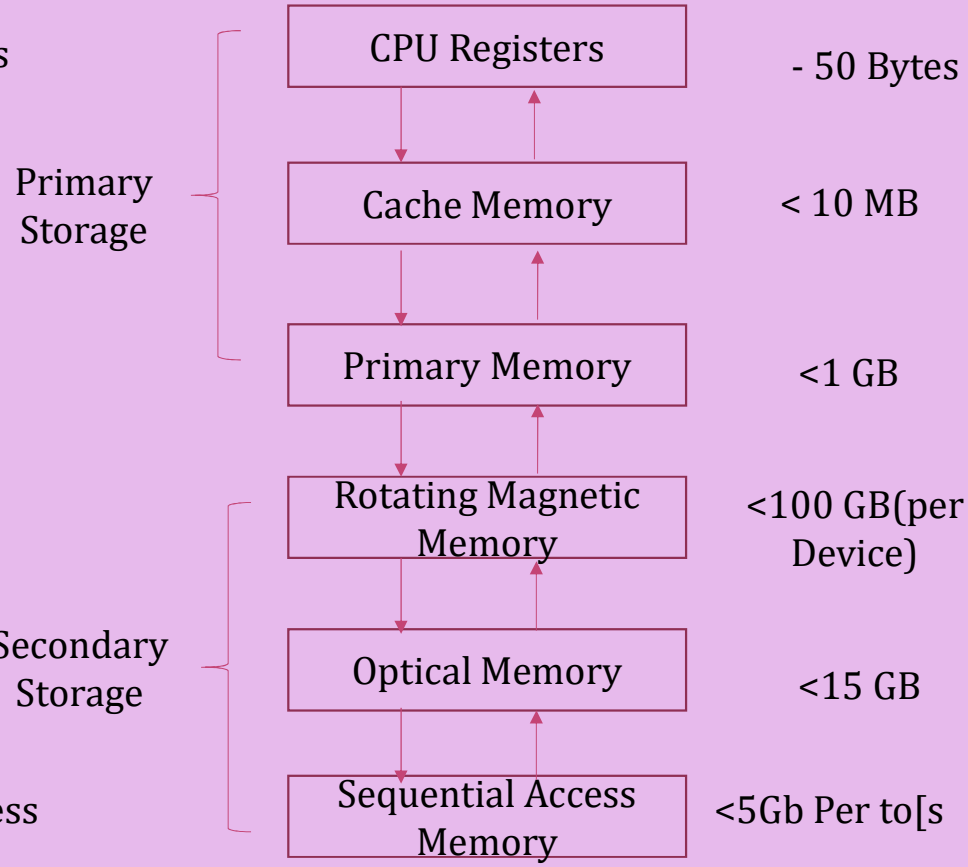
Kernel associate each process to its virtual address space.

This address space is linear starting from 0 upto max no

UNIX Process Address Space



Memory Hierarchy



Pages(Valid and Invalid Pages)

The virtual address space consists of pages of fixed sizes typically 4KN (for 32 bit sys) and 8KN for(64 bit sys).

Pages can be labelled as valid or invalid.

Valid Page: It is an actual page in physical memory or some secondary storage such as swap partition or a file on the disk.

Invalid Page: It indicates unused/unallocated address space.

Memory Regions

The kernel arranges page into blocks that share certain properties such as access permission. These block are called memory regions segments or mappings.
Different Memory Regions are:

1) Text segment(program code, const, var)
2) Stack(Process execution stack contain var and fun)
3) Data segment(process dynamic memory)
4) BSS segment(block started by symbol)
    1) Text segment: Machine instructions that CPU executes, text segment is sharable so that only a single copy needs to be in memory for frequently executed programs such as text editor, c compiler.
    2) Stack: Stack is place where automatic variables are stored along with the information that is saved each time a function is called. Each time a function is called the address of where to return to and certain information about the callers environment such as some of the machine registers are saved in the stack.
    3) Data Segment:
        1) Initialized data segment: It is usually called simply the data segment containing variables that are specifically initialized in the program.
        2) Uninitialized data segment: known as BSS.(Block Started for Symbol) Data in this segment is initialized by the kernel to arithmetic 0 or null pointer before the program starts executing

Dynamic Memory Management

Malloc : allocate memory dynamically

#include<stdlib.h>

Void * malloc(size_t size)

Malloc allocates size bytes of memory and returns pointer to start of newly allocated region.

Calloc: Dynamic allocation of array

Void * calloc(size_t n, size_t size)

The C library function void *calloc(size_t nitems, size_t size) allocates the requested memory and returns a pointer to it. The difference in malloc and calloc is that malloc does not set the memory to zero where as calloc sets allocated memory to zero.

Realloc:is used to resize the memory block pointed to a pointer that was previously allocated to the variable by the malloc() or calloc() function. Realloc stands for reallocation

void * realloc(void *ptr,size_t size)

Free :**free**() **function** only frees the memory from the heap and it does not call the destructor

Void free(void *ptr)

Alignment

Th alignment of data refers to the relation between its address and memory chunks. A variable located at a memory address that is a multiple of its size is said to be naturally aligned.

e.g 32 bytes memory allocated is multiple of 4

Progammers require dynamic memory aligned along a larger boundary such as page

The valloc() function has the same effect as malloc() , except that the allocated memory will be aligned to a multiple of the value returned by sysconf(_SC_PAGESIZE) .

Note: When free() is used to release storage obtained by valloc() , the storage is not made available for reuse.

The memalign function allocates a block of size bytes whose address is a multiple of boundary . The boundary must be a power of two! The function memalign works by allocating a somewhat larger block, and then returning an address within the block that is on the specified boundary.

Data Segment

A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Before loading program data is set here.

Data segment is not read only as values of variables can be changed at runtime.

The computer program memory is organized into the following:

1) Data Segment(Data+BSS+Heap)
2) Stack
3) Code Segment

Data

The data area contains global and static variables used by the program that are initialized.

This segment can be further classified into initialized read only area and initialized read write area.

BSS: This segment is known as uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

e.g. Static int i; would be contained in the BSS segment.

Heap: This area begins at the end of BSS segment and grows to larger address from here. It is managed by malloc, realloc and free. (which uses brk or sbrk system calls. When brk or sbrk system calls doe not complete the malloc or ralloc function mmap function is used.

The heap are is shared by all shared libraries and dynamically loaded modules in a process.

Data

brk, sbrk - change the amount of space allocated for the calling process's data segment

```
#include <unistd.h>

int brk(void *endds);

void *sbrk(intptr_t incr);
```

The brk() function is used to change the space allocated for the calling process. The change is made by setting the process's break value to addr and allocating the appropriate amount of space. The amount of allocated space increases as the break value increases. The newly-allocated space is set to 0.
In general brk() asks the kernel to let you you read and write to a contiguous chunk of memory called the heap.

Stack

Stack are is joined to heap area and grew the opposite direction. When the stack pointer met the heap pointer free memory was exhausted.

(heap and stack grow in opposite directions)

The stack are contains the program stack, a LIFO structure. A stack pointer register tracks the top of the stack; it is adjusted each time value is pushed onto the stack. The set of values pushed for one function call is termed a 'stack frame'. A stack frame consists of minimum of return address.

Default value for stack is 64MB

For 32 bit UNIX OS stack can be increased upto 600MB

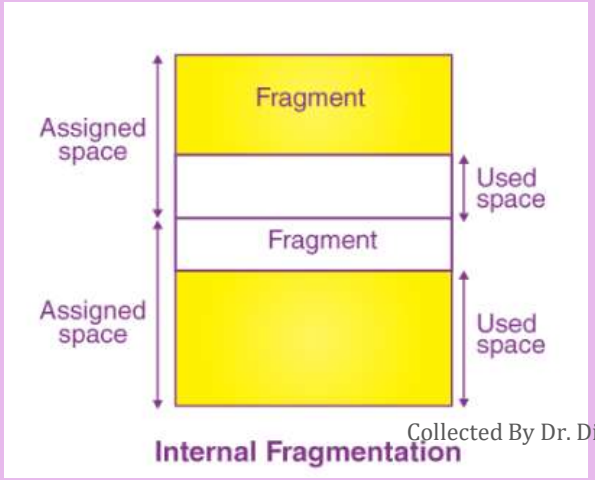
For 64 bit UNIX OS stack can be increased upto 2048MB

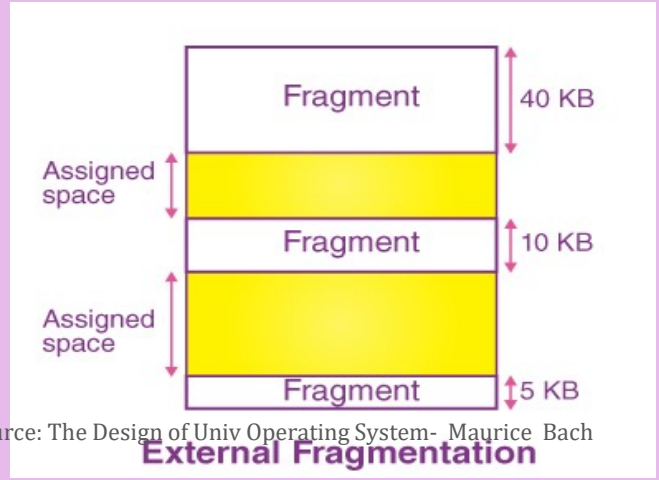
Anonymous memory mappings

The algorithm is known as 'buddy memory allocation scheme' which works using two types of fragmentation.

i) Internal fragmentation: It occurs when more memory that requested is used to satisfy an allocation. This result an efficient use of the available memory

ii) External fragmentation: It occurs when sufficient memory is free to satisfy a request but it is split into two or more non adjacent chunks. This can result in inefficient use of memory or failed memory allocation. Allocating memory via anonymous mappings has several benefits





Source: The Design of Univ Operating System- Maurice Bach

Anonymous memory mappings: Advantages

Allocating memory via anonymous mappings has several benefits:

1) No fragmentation concern: When the program no longer nees an anonymous memory mapping the mapping is unmapped and the memory is immediately returned to the system.
2) Anonymous memory mappings are resizable and have adjustable permissions can receive advice just line normal mappings
3) Each allocation exists in a separate memory mappings. There is no need to manage the global heap.

Anonymous memory mappings: Disadvantages

1) Memory mappings are always an integer number of pages in size. If file size is small than page size the memory is wasted e.g. for file size 4KB 7 byte mapping(7 byte page size waste is 3 bytes)
2) The memory mapping must be fit into process address space. For 32 bit address space various size mappings create fragmentation. In case of 64 bit address space this problem does not occur.
3) There is overhead of creating and maintain the memory mappings and associated data structures inside the kernel.

Creating anonymous memory mappings

This can be done using mmap function and memory will be destroyed using munmap function.

#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: set to NULL, signifies that the memory mapping may begin any where in memory that the kernel specifies

prot: It sets PROT_READ,PROT_WRITE bits making the mapping readable and writable

flags: It sets the MAP_ANONYMOUS bit making this mapping anonymous of MAP_PRIVATE bit to make it private

fd: file descriptor and offset are ignored if MAP_ANONUMOUS is set.

Advanced Memory Allocation

The advanced memory allocation can be done by mallopt() system call.

#include<malloc.h>

int mallopt(int param int value)

Linux support the following values for param defined in<malloc.h>

i) M_CHECK_ACTION

ii) M_MAP_MAX

iii) M_MAP_THRESHOLD

iv) M_MXFAST

v) M_TOP_PAD

vi) M_TRIM_THRESHOLD

M_CHECK_ACTION:

Values are

0:ignore error condition ; continue execution with undefined results

1: Print a detailed error message and continue execution

2: abort the program

3:Print detailed error message; stack trace and memory mappings and abort the program

5: Print a simple error message and continue the execution

7: Print simple error message; stack trace and memory mappings and abort the progra,

M_MMAP_MAX

This is maximum number of mappings that the system will make to satisfy dynamic memory requests. When this limit is reached the data segment will be used for all allocations until one of these mappings is freed

Value 0 will disable all use of anonymous mappings as a basis for dynamic memory allocation.

M_MMAP_THRESHOLD

The size threshold over which an allocation request will be satisfied via anonymous mapping instead of the data segment.

Value 0: enables the use of anonymous mappings for all allocations effectively disabling use of data segment for dynamic memory allocation

M_MXFAST

The maximum size of a fast bin. Fast bins are special chunks of memory in the heap that never merged with adjacent chunks and never returned to the system. Which allows fast allocations

Value 0:disables all use of fast bins

M_TOP_PAD

The amount of padding(in bytes) used wjen adjusting the size of the data segment.

Debugging memory allocations

Memory allocations can be debugged with the help of MALLOC_CHECK as an environment variable in a program. Linux also provides mallinfo() function for obtaining statistics related to the memory allocation system.

#include<malloc.h>

struct mallinfo(vod)

Another function which returns the memory related statistics is malloc_stats()

#include<malloc.h>

Void malloc_stats(void)

This provides information such as system bytes, used bytes, max mmap regions as well as mmap bytes.

Stack based allocations

Dynamic memory can be obtained by using heap or memory mappings to stack. A stack contains programs automatic variables. To make a dynamic allocation from the stack use of alloc() system call is made.

#include<alloca.h>

void * alloca(size_t size);

The memory allocated with alloca() is automatically freed as the stack unwinds back to the invoking function upon return.

Choosing a memory allocation mechanism

Various memory mechanisms

i)malloc

ii)calloc

iii)realloc

iv)brk() and sbrk()

v) Anonymous memory mappings

vi)posix_memaligh()

vii)valloc()

viii)alloc()

ix) Variable length arrays

Manipulating memory

Raw bytes of memory can be manipulated using 'C' functions like strcpy() and strcmp() provided with a user defined buffer size.

i)memset(): is used to set the specified bytes. A call to memset() sets n bytes starting at S to the byte C and return s

void * memset(void *s,int c,size_t n);

ii)memchr()is used tosearch a given byte in a block of memory

void * memchr(const void *s, int cm size_t n);

iii)memcmp(): is used to compare two chunks of memory for equivalence

Int memcmp(const void *s1,const void *s2,size_t n);

iv)memmove(): is used to copy the first n bytes of src to dst

Void * memmove(void *dst, const void *src, size_t n);

Locking Memory

The concept of demand paging is well supported by linux. Demand Paging swaps the required pages in the memory with the pages no longer required

Locking means locking some part of address space with the help of mlock() function where as locking of all the address space can be done with mlockall() function.

Linux implements demand paging which allows to swap in the pages required from the disk as well as swapping out of pages if the are no longer needed. This allows the virtual address spaces of process on the system to have no direct relationship to the total amount of physical memory.

Two situations of systems paging behavior

i) Determinism: Applications with timing constraints require deterministic behavior. If some memory accesses result in page faults which incur costly disk I/O operations-applications can over run their timing needs.

By ensuring that the pages it needs are always in the physical memory and never pages to disk assures the memory accessed will not result in page faults, providing consistency , determination and improved performance.

ii) Security: certain file are required to be stored in the physical memory/.

Unlocking memory

Unlocking the memory requires the kernel to swap the pages out to disk as needed this can be done using tow system calls munlock() and munlockall()

#include<sys/nman.h>

int munlock(const void * addr, int size_t len)

int munlockall(void)

The system call munlock() unlocks the pages starting at addr and extending for len bytes where as the call to munlockall() undoes the effect of mlockall()

Locking Limits

Linux decides the max number of pages to be locked to maintain the overall performance of the system. This can be done using RLIMIT_MEMLOCK bytes. The default is 32 KB.

For debugging and diagnostic purpose the mincore() function is used to determine whether the range of given memory is in physical memory or swapped out to disk.

mincore() returns a vector that indicates whether pages of the calling process's virtual memory are resident in core (RAM), and so will not cause a disk access (page fault) if referenced.#include<unistd.h>

#include<sys/„an.h>

int mincore(void *start, size_t length, unsigned char(uec)

A call to mincore function gives information about which pages of mapping are in physical memory time of the system call.

The mincore function requests a vector describing which pages of a file are in core and can be read without disk access. The kernel will supply data for length byes following the start address. On return the kernel will have filled vec with bytes of which the lease significant bit indicates of a page is core resident.

Opportunistic Allocation

Linux implements the allocation strategy by creating new memory mapping for a processes request for a an additional memory.

It actually allocates the memory only when the process writes to a newly allocated memory. This is done by the kernel on page by page basis, paging and copy-on-writes needed.

Advantages:

1) No need to allocate memory in advance so the wastage is avoided.
2) The requests are satisfied in page by page manner

SWAPPING

Maurice-Bach-Notes/9-Memory-Management-Policies.md at master · suvratapte/Maurice-Bach-Notes · GitHub

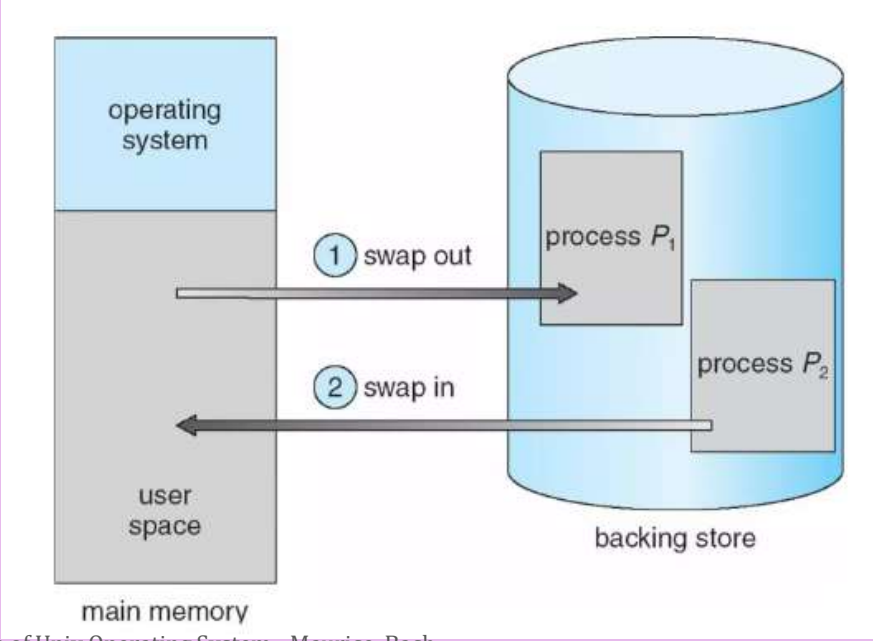
Swapping is process of exchanging pages, segment of memory and values to another location.

UNIX systems transferred entire processes between primary memory and the swap device. Such a memory management policy is called swapping.

The entire process does not have to reside in main memory to execute, and the kernel loads pages for a process on demand when the process references the pages. This permits greater flexibility in mapping the virtual address space of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of available physical memory and allowing more processes to fit simultaneously in main memory. Swapping policy is easier to implement and has less overhead.

Swapping

- Swapping has 3 parts to it:
- managing space on the swap device
- swapping processes out of main memory
- swapping processes into main memory



To increase CPU utilization in multiprogramming, a memory management scheme known as swapping can be used.

Swapping is the process of bringing a process into memory and then temporarily copying it to the disc after it has run for a while.

The purpose of swapping in an operating system is to access data on a hard disc and move it to RAM so that application programs can use it.

It's important to remember that swapping is only used when data isn't available in RAM.

Although the swapping process degrades system performance, it allows larger and multiple processes to run concurrently.

Because of this, swapping is also known as memory compaction.

The CPU scheduler determines which processes are swapped in and which are swapped out.

Consider a multiprogramming environment that employs a priority-based scheduling algorithm.

When a high-priority process enters the input queue, a low-priority process is swapped out so the high-priority process can be loaded and executed.

When this process terminates, the low priority process is swapped back into memory to continue its execution.

- Swapping has been subdivided into two concepts: swap-in and swap-out.
- Swap-out is a technique for moving a process from RAM to the hard disc.
- Swap-in is a method of transferring a program from a hard disc to main memory, or RAM.

Advantages of Swapping

- If there is low main memory so some processes may has to wait for much long but by using swapping process do not have to wait long for execution on CPU.
- It utilize the main memory.
- Using only single main memory, multiple process can be run by CPU using swap partition.
- The concept of virtual memory start from here and it utilize it in better way.
- This concept can be useful in priority based scheduling to optimize the swapping process.

Disadvantages of swapping

- If there is low main memory resource and user is executing too many processes and suddenly the power of system goes off there might be a scenario where data get erase of the processes which are took parts in swapping.
- Chances of number of page faults occur
- Low processing performance

- The procedure by which any process gets removed from the **hard disk** and placed in the **main memory or RAM** commonly known as **Swap In.**
- On the other hand, **Swap Out** is the method of removing a process from the **main memory or RAM** and then adding it to the **Hard Disk.**

Swapping process out

A process is swapped out if the kernel needs space in memory. It needs space in memory under following situations:

- The *fork* system call must allocate space for a child process.
- The *brk* system call increases the size of a process.
- A process becomes larger by the natural growth of its stack.
- The kernel wants to free space in memory for processes it had previously swapped out and should now swap in.

Demand Paging

Machines whose memory architectures is based on pages and whose CPU has restartable instructions can support a kernel that implements a demand paging algorithm, swapping pages of memory between main memory and a swap device.

Demand paging systems free processes from size limitations otherwise imposed by the amount of physical memory available on a machine.

The kernel still has a limit on the virtual size of a process, dependent on the amount of virtual memory the machine can address.

Demand paging is transparent to the user except for the virtual size permissible to a process.

Data Structures for Demand Paging

The kernel contains 4 major data structures to support low-level memory management functions and demand paging:

- page table entries
- *disk block descriptors*
- *page frame data table* (called *pfdata* for short)
- swap-use table

pfdata table is allocated once for the lifetime of the system, but allocations for other structure are dynamic.