Operating Systems

Memory

Carlos Alberto Llano R.

15 de octubre de 2015

Contents

Basic concepts

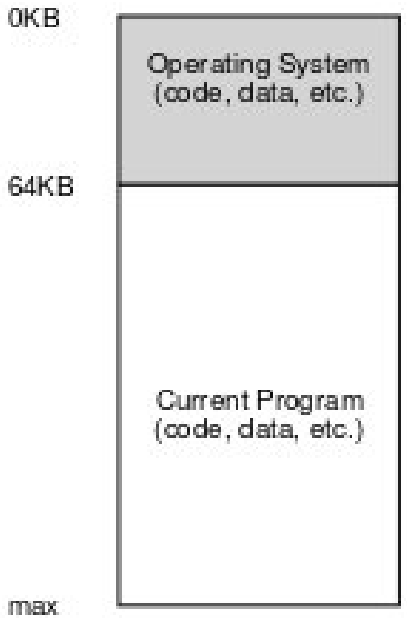
Memory is central to the operation of a modern computer system.

Memory consists of a large array of bytes, each with its own address.

The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

Basic concepts - Multiprogramming and Time Sharing

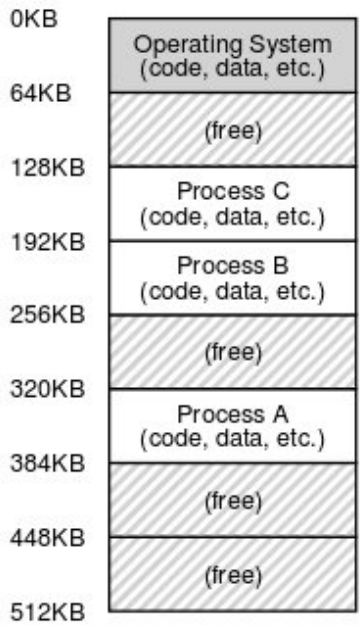
One way to implement time sharing would be to run one process for a short while, giving it full access to all memory, then stop it, save all of its state to some kind of disk (including all of physical memory).



Basic concepts - Multiprogramming and Time Sharing

Saving and restoring register-level state (the PC, general-purpose registers, etc.) is relatively fast, saving the entire contents of memory to disk is brutally non-performant.

Thus, what we'd rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently.

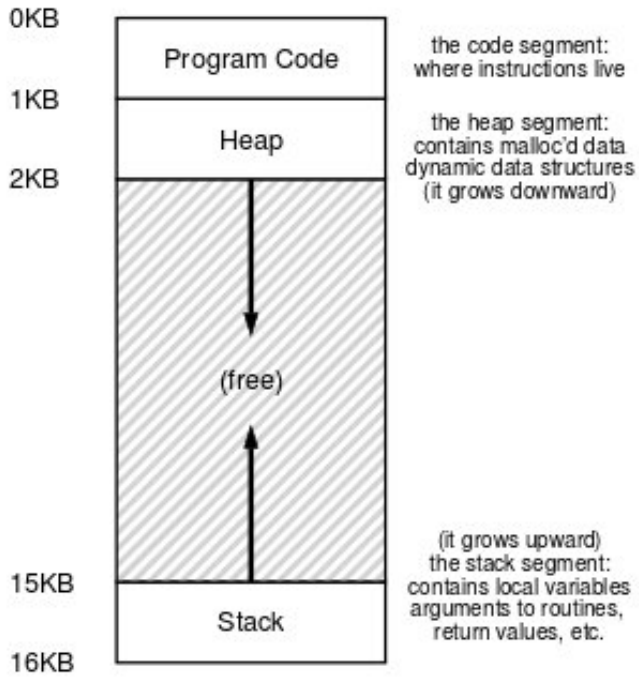


Basic Hardware

- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- The main memory is accessed via a transaction on the memory bus.
- Cache: fast memory between the CPU and main memory.

Basic Hardware - Address Space

- Each process has a separate memory space (or address space).
  - An address space is the set of addresses that a process can use to address memory.



Basic Hardware - Address Space

The address space of a process contains:

The code of the program (the instructions) have to live in memory somewhere, and thus they are in the a

A stack to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and fro routines.

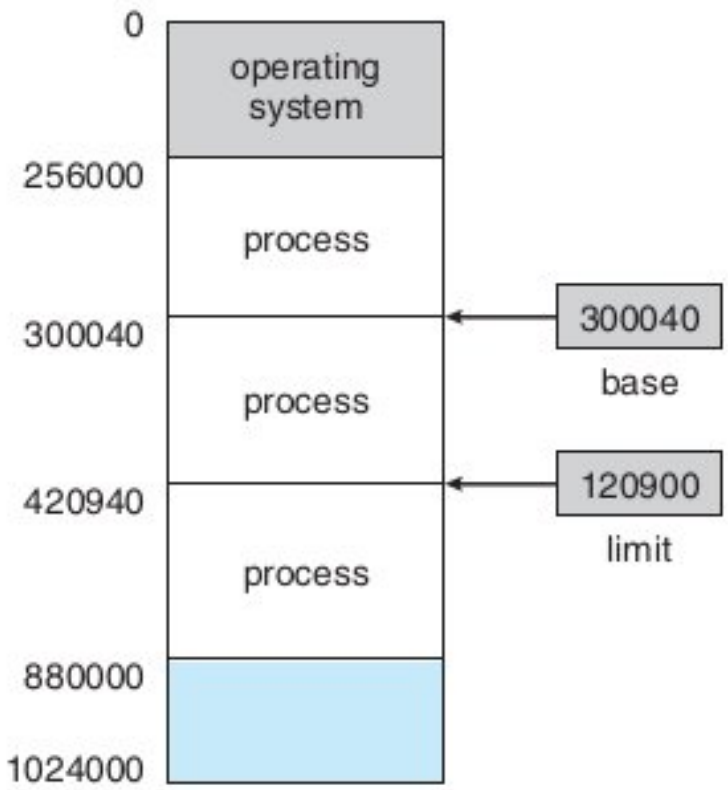
The heap is used for dynamically-allocated, user-managed memory.

Basic Hardware

- Is fundamental to having multiple processes loaded in memory for concurrent execution.
- Ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

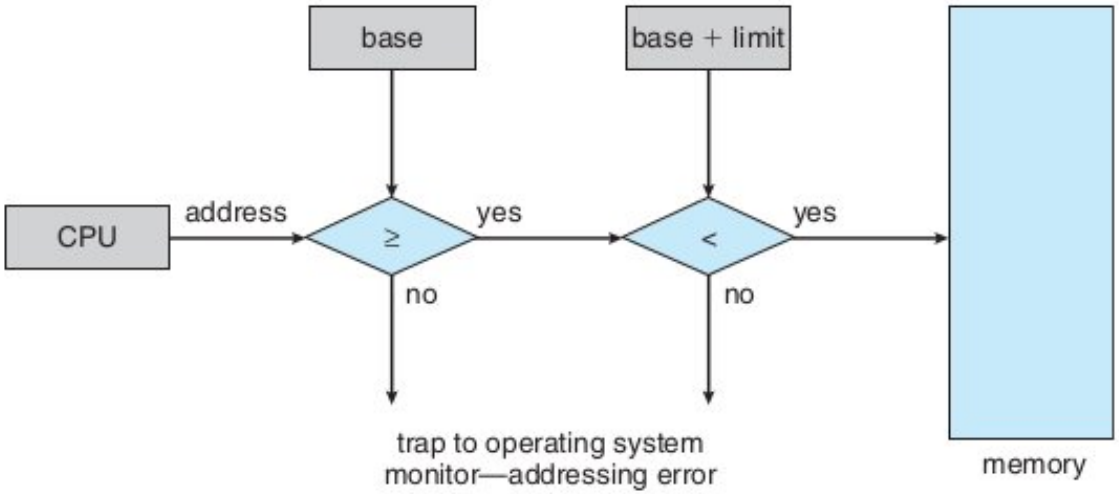
  base register holds the smallest legal physical memory address
  limit register specifies the size of the range

Basic Hardware - Registers



Basic Hardware - Protection with base and limit registers

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



Address Binding

Addresses in the source program are generally symbolic (e.g. the variable count).

A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").

The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014).

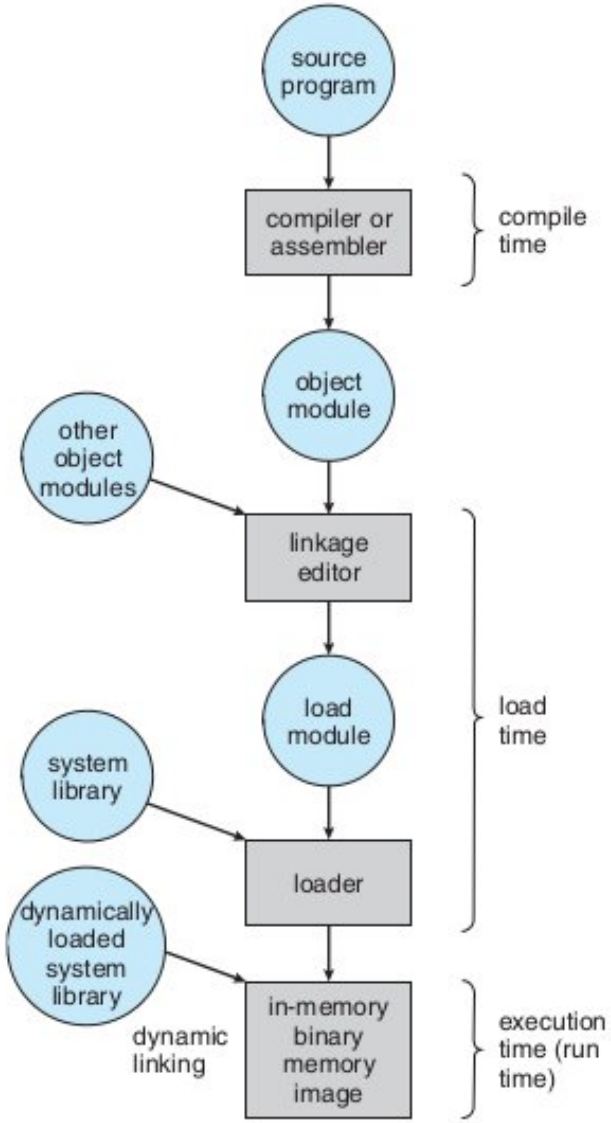
Address Binding

Compile time If you know at compile time where the process will reside in memory, then absolute code can be generated.

Load time If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time.

Execution time If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Address Binding

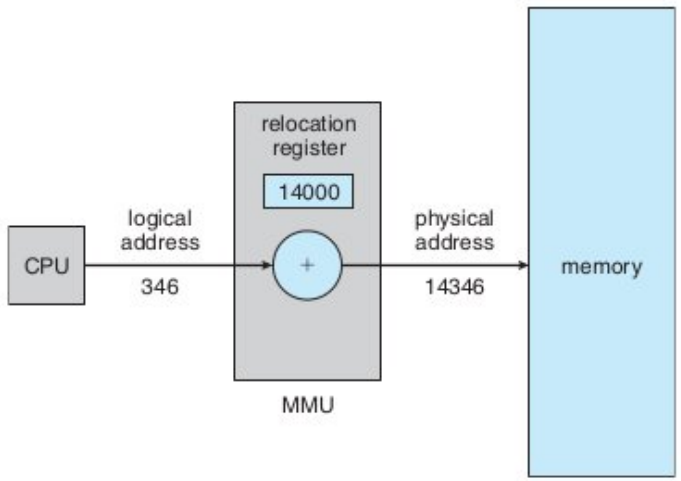


Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a *logical address*, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a *physical address*.

The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

Logical Versus Physical Address Space



The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.

The memory-mapping hardware converts logical addresses into physical addresses.

Logical Versus Physical Address Spaceg

Ever write a C program that prints out a pointer? The value you see (some large number, often printed in hexadecimal), is a virtual address.

This little program that prints out the locations of the main() routine (where code lives), the value of a heap-allocated value ret urned from malloc(), and the location of an integer on the stack

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        printf("location of code  : %p\n", (void*) main);
        printf("location of heap  : %p\n", (void*) malloc(1));
        int x = 3;
        printf("location of stack : %p\n", (void*) &x);
        return x;
}
```

Dynamic Loading

With dynamic loading, a routine is not loaded until it is called.

1. The main program is loaded into memory and is executed.
2. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
3. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
4. Then control is passed to the newly loaded routine.

Dynamic Linking and Shared Libraries

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run.

Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.

A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library.

Dynamic Linking and Shared Libraries

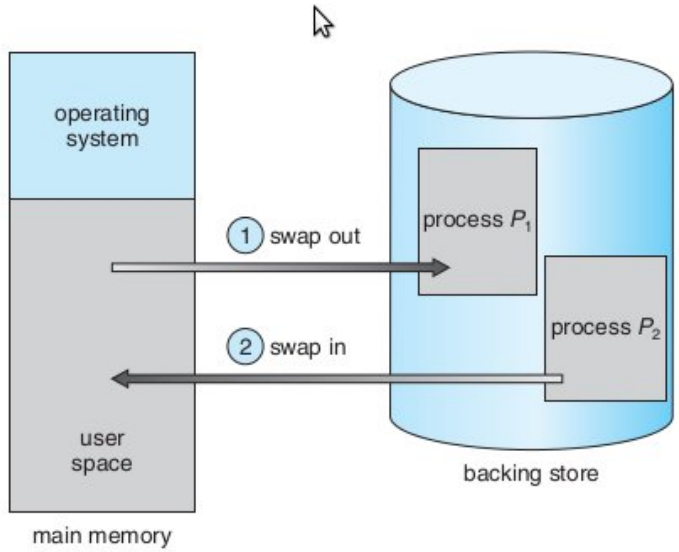
More than one version of a library may be loaded into memory.

Each program uses its version information to decide which copy of the library to use.

Only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as shared libraries.

Swapping

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.



Swapping

The context-switch time in such a swapping system is fairly high. Let's assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100-MB process to or from main memory takes 100 MB/50 MB per second = 2 seconds.

Swapping on Mobile Systems

Mobile systems typically do not support swapping in any form.

It may terminate a process if insufficient free memory is available. However, before terminating a process, Android and iOS writes its application state to flash memory so that it can be quickly restarted.

Types of memory - stack

```
void func() {
        int x; // declares an integer on the stack
        ...
}
```

The compiler does the rest, making sure to make space on the stack when you call into func(). When your return from the function, the compiler deallocates the memory for you; thus, if you want some information to live beyond the call invocation, you had better not leave that information on the stack.

Types of memory - heap

Where all allocations and deallocations are explicitly handled by you, the programmer.

```
void func() {
        int *x = (int*) malloc(sizeof(int));
        ...
}
```

malloc

The malloc() call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL.

```
void func() {
        int *x = (int*) malloc(sizeof(int));
        ...
}
```

malloc

```
void func() {
        int *x = malloc(10*sizeof(int));
        printf("%d\n", sizeof(x));
        ...
}
```

When we use sizeof() in the second line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines) why?

malloc

The reason is that in this case, sizeof() thinks we are simply asking how big a pointer to an integer is, not how much memory we have dynamically allocated.

However, sometimes sizeof() does work as you might expect:

```
void func() {
        int x[10];
        printf("%d\n", sizeof(x));
        ...
}
```

In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.

free

To free heap memory that is no longer in use, programmers simply call free():

```
void func() {
        int *x = malloc(10*sizeof(int));
        printf("%d\n", sizeof(x));
        ...
        free(x);
}
```

calloc

The calloc() allocates space for an array elements, initializes to zero and then returns a pointer to memory.

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

```
void func() {
        int *x = (int *)calloc(100, sizeof(int));
        printf("%d\n", sizeof(x));
        ...
        free(x);
}
```

realloc

The realloc() change the size of previously allocated space.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    printf("Enter size of array: ");
    scanf(" %d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf(" %u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf(" %d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0;i<n2;++i)
        printf(" %u\t",ptr+i);
    return 0;
}
```

Common Errors - Forgetting To Allocate Memory

```
void func() {
        char *src = "hello";
        char *dst;          // oops! unallocated
        strcpy(dst, src); // segfault and die
        ...
}
```

Common Errors - Forgetting To Allocate Memory

```
void func() {
        char *src = "hello";
        char *dst;          // oops! unallocated
        strcpy(dst, src); // segfault and die
        ...
}

void func() {
        char *src = "hello";
        char *dst = (char*) malloc(strlen(src) + 1);
        strcpy(dst, src); // segfault and die
        ...
}
```

Common Errors - Not Allocating Enough Memory

```
void func() {
        char *src = "hello";
        char *dst = (char*) malloc(strlen(src)); //
    too small!
        strcpy(dst, src);
        ...
}
```

Common Errors - Forgetting to Initialize Allocated Memory

With this error, you call malloc() properly, but forget to fill in some values into your newly-allocated data type.

Common Errors - Forgetting To Free Memory

Another common error is known as a memory leak, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem.

Common Errors - Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it.

Common Errors - Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the double free. The result of doing so is undefined.

Common Errors - Calling free() incorrectly

When you pass in some other value, bad things can (and do) happen. Thus, such invalid frees are dangerous and of course should also be avoided.

valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for Linux when on the x86 and, as of version 3, AMD64, architectures. It allows you to run your program in Valgrind's own environment that monitors memory usage such as calls to malloc and free (or new and delete in C++).

valgrind - Install

sudo apt-get install valgrind

or

http://www.valgrind.org/downloads/current.html

bzip2 -d valgrind-XYZ.tar.bz2
tar -xf valgrind-XYZ.tar
./configure
make
make install

valgrind - Examples

view:

Examples/memory/c/val0X.c

and

Examples/memory/c++/fibonacci.cc

gdb

Gdb is a debugger for C (and C++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface.

gdb - commands

- gdb executable: To start gdb
- run: will start the program running under gdb.
- break: A 'breakpoint' is a spot in your program where you would like to temporarily stop execution in order to check the values of variables.
- delete: will delete all breakpoints that you have set.
- clear function: will delete the breakpoint set at that function.
- continue: will set the program running again, after you have stopped it at a breakpoint.
- step: will go ahead and execute the current source line, and then stop execution again before the next source line.

gdb - commands

- next: will continue until the next source line in the current function.
- print: expression will print out the value of the expression, which could be just a variable name.
- backtrace: If you want to know where you are in the program's execution.

gdb - Examples

view Examples/memory/gdb/c++/series.cc