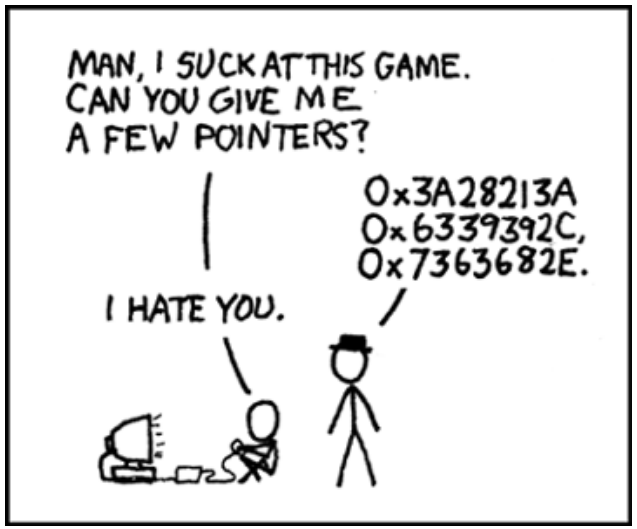
Week 11: Memory Management

CS211: Programming and Operating Systems

Niall Madden (Niall.Madden@NUIGalway.ie)

Wednesday and Thursday, 28+29 April 2021



https://xkcd.com/138

CS211 Assessment

Grades for CS211 will be based on

1. Four programming assignments: 40%. The final component, Lab 6, is due **5pm. Tuesday 4 May**.
2. Homework assignment 20%. Details are on Blackboard. Deadline is 5pm, **Friday 30 April**.
3. Online exam: 40%.

Would you like to hear all about...?

1. Part 1: Memory Management
   - Terminology
   - Memory and Storage
   - Main topics
   - Logical and Physical Addresses
2. Part 2: Contiguous Memory Allocation
   - Algorithms
   - Fragmentation
3. Part 3: Paging
   - `01PageSize.c`
   - Pages and Frames
   - Demand Paging
   - Page Fault
4. Part 4: Page Replacement
   - Page Replacement
   - First-In-First-Out (FIFO) Algorithm
   - Other Algorithms
5. Part 5: Memory management in C
   - Arrays V pointers
   - The `sizeof()` things
6. Part 6: Dynamic memory allocation

CS211
Week 11: Memory Management

Start of ...

PART 1: Memory Management

Arguably, the most precious resource an OS can allocated to a process is memory.

Why even more precious than, say, CPU time? Enter your answer in the Chat.

Summary of main ideas

- "***Storage***" is how/where long-term data is maintained, particularlly while a program is not running.
- During execution, a program (now, a process) has data stored in ***main memory***, also known as "primary storage".
- Data belonging to a process is stored in "actual" memory: integrated circuits physically located on the device.
- The OS presents this location to the device as a logical address. (A little like we referred to our physical class-room as "AC202", rather than $53 \circ 16'49.1''N\ 9°03'38.4''W$)
- The OS can use (backing) storage as temporary main memory, using "swapping".

Memory and Storage Management

So far, in the "OS" component of ***CS211*** we have focused on

- OS Stucture
- Processes – what are they?
- Process scheduling (w.r.t. CPU time)
- Process resource allocation.

We didn't really discuss what these "resources" are. In this final section of CS211 we will look at the management of the most important resources (other than CPU time): **memory** and **storage**.

In the textbook (OSTEP),

- ***Memory*** is covered Chapters 13–23 (which shows that we don't do it in full detail here), as part of ***Virtualisation***.
- ***Storage*** is covered in Chapters 36–50, under the heading of ***Persistence***. Unfortunately, along with **security**, it is largely omitted from CS211.

Roughly, "**storage**" refers to how long-term data is maintained, particularlly while a program is not running. Traditionally, this was on disks (drives), more recently it is on networked (cloud) devices.

But during execution, a program (now, a process) has data stored in main memory.

Management of both main memory and long-term storage require a high degree of sophistication on the part of the operating system.

This section consists of

- Memory Management
- Virtual Memory
- Programming with memory addresses (pointers).

A program must be brought into (main) memory and placed within a process for it to be executed. An ***Input queue*** is maintained, i.e., a collection of jobs on the disk that are waiting to be brought into memory for execution. User programs go through several steps before being executed, in particular they must be allocated space in memory.

Furthermore, recalling ***CPU Scheduling***: the OS attempts to maximise CPU usage and responsiveness by switching between processes. This means that memory must be allocated to a (possibly large) number of processes at any given time.

This leads to several complications.

(a) Since there are many processes running, the OS must have a fair way of allocating memory to all running processes.
(b) As we've seen with `fork()`'ed processes, two processes can think they have access to the same memory address. So some ***virtualization*** is needed.
(c) Most computers have different types of "memory", such as `cache`, `RAM`, and `backing storage`. The OS presents a single coherent view of all this to processes.
(d) Physical memory is not necessarily large enough to accommodate all the procsess, and so secondary storage must be used as "*over-flow*" memory.

A logical address is generated by the CPU; as far as the process is concerned, it is the address that it uses. It is also referred to as a ***virtual address***.
A ***physical address*** is that what is used by the memory unit, i.e., the one loaded into the memory address register.

The ***virtual address*** is unique within a process. That is, no two variables belonging to a process can be stored at the same virtual address.

You can check the virual address used by a C program as follows:

```
1   int x;
    fork();
3   x = getpid();
    printf("x is stores %d at %p\n", x, &x);
```

The output I get is:

```
x is stores 21106 at 0x7fff1650fdc4
2 x is stores 21107 at 0x7fff1650fdc4
```

There must be a mechanism for associating (or ***binding***) abstract memory, e.g., program variables, to actual memory address. This can happen at

1. Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
2. Load time: relocatable code is generated if memory location is not known at compile time.
3. **Execution time**: Binding delayed until run time, so a process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers). Most modern operating systems use execution-time binding.

CS211
Week 11: Memory Management

END OF PART 1

CS211
Week 11: Memory Management

Start of ...

PART 2: Contiguous Memory Allocation

Part 2: Contiguous Memory Allocation

In a **multiprogramming** environment, memory space is occupied by the operating system and a collection of user processes.

In order to maximise the degree of multiprogramming on the system, the OS will load as many procs into memory as there is space for.

When new processes arrive in the ***input queue***, it picks the first one and loads it into memory unless there is not enough room to accommodate it ("Banker's Algorithm").

In this case, it may search through to procs in the Input queue and load the first one for which there is a large enough space to accommodate.

When user processes terminate, memory “holes” are created. The operating system must allocated one of these hole to a new proc that is starting.

If the hole it too large, only part of it is allocated and a new smaller hole is created. Also, when a process terminates, if its space is adjacent to a free hole, they are amalgamated to create a larger one.

The scheduler then checks if they new hole is large enough to accommodate the next job in the Input queue.

Analogy: books in a book-case (where we can’t change the position of any book on the case)

If this procedure of splitting and amalgamating of holes continues for a while, we end up with blocks of available memory of various size are scattered throughout memory. This is called **Fragmentation** and is to be avoided.

Strategies/Algorithms for allocating memory to procs is a crucial part of memory management.

Three different strategies (see Section 17.3) are:

1. **First Fit (FF):** allocate the first hole that is large enough to accommodate the new proc. This is the fastest method, but may cause the most fragmentation–i.e., the most small “pockets” of unused non-contiguous memory.
2. **Best Fit (BF):** search all available holes and allocate the smallest hole that is big enough to accommodate the new proc. This is slower than best-fit, but leads to the smallest “pocket” sizes.
3. **Worst Fit (WF):** search all available holes and assign part of the the largest available hole. This is the slowest method but may leave a smaller number of larger holes than either first or best fit.

Example

Suppose that a system has four free memory holes orders as follows:

$$H_1 = 100\text{k}, H_2 = 500\text{k}, H_3 = 200\text{k}, H_4 = 300\text{k}.$$

Four jobs (i.e., processes) requiring (contiguous) memory space of various sizes are submitted at the same in the order given below.

Process	P_1	P_2	P_3	P_4
Size	140k	450k	200k	300k

Show how these would be allocated by the FF and BF strategies.

Process	P_1	P_2	P_3	P_4
Size	140k	450k	200k	300k

First Fit

H_1 (100)	H_2 (500)	H_3 (200)	H_4 (300)

Best Fit

H_1 (100)	H_2 (500)	H_3 (200)	H_4 (300)

Process	P_1	P_2	P_3	P_4
Size	140k	450k	200k	300k

Worst Fit

H_1 (100)	H_2 (500)	H_3 (200)	H_4 (300)

Memory is ***partitioned*** into contiguous segments and allocated to different processes. The methods for contiguous memory allocation described above can lead to

- ***External fragmentation:*** total memory space exists to satisfy a request, but it is not contiguous. That is, memory is available but is broken up into "pockets" between partitions that are too small to be used.
- ***Internal fragmentation:*** Suppose there is a free hole of size 12,100 bytes and we require a partition of size 12,080 bytes. The OS may require more that 20 bytes to keep track of the unused portion, so it can be more economical to allocate all 12,100 bytes even though this is slightly larger than requested memory; this is called **"internal fragmentation**

CS211
Week 11: Memory Management

END OF PART 2

CS211
Week 11: Memory Management

Start of ...

PART 3: Paging

Part 3: Paging

(See Chapter 18 of the text-book:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf)

Basic idea:

- Logical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available ("chop up the processes").
- Divide physical memory into fixed-sized blocks called **frames**. All frames on the system are of the same size, usually some power of 2, determined by the hardware ("chop up the space").

To find out the page size on a Linux system, run this program:

From en.wikipedia.org/wiki/Page_(computer_memory)

```c
// Try this for Linux
#include <stdio.h>
#include <unistd.h> /* sysconf(3) */
int main(void) {
        printf("The page size for this system is %ld bytes.\n",
               sysconf(_SC_PAGESIZE)); /* _SC_PAGE_SIZE is OK too. */
        return 0;
}
```

The output I get on my laptop (and https://www.onlinegdb.com) is

```
The page size for this system is 4096 bytes.
```

Try this on a Windows system:

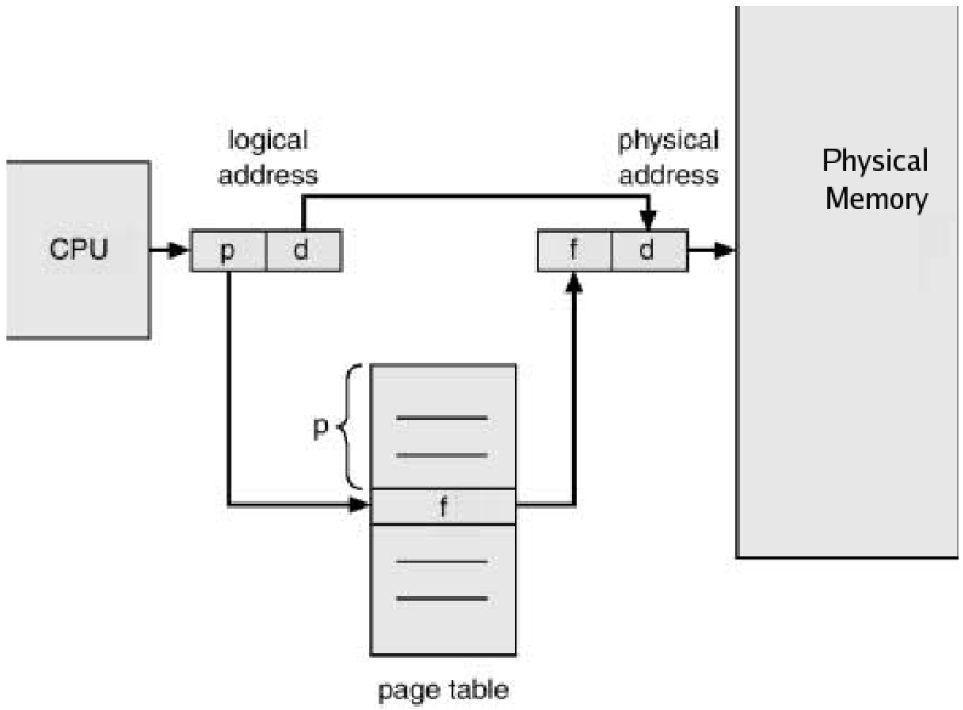
From en.wikipedia.org/wiki/Page_(computer_memory)

```
 // Try this for Windows
#include <stdio.h>
#include <windows.h>
int main(void) {
  SYSTEM_INFO si;
  GetSystemInfo(&si);
  printf("The page size for this system is %u bytes.\n",
     si.dwPageSize);
  return 0;
}
```

What output do you get?

- Logical memory into blocks called **pages**.
- **Frames** and **Pages** are of the same size.
- the OS keeps track of all free frames in memory.
- To run a program of size *n* pages, we need to find *n* free frames and load program.
- ***Page Table*** maintained to translate logical to physical addresses.

Address Translation Scheme – An address generated by CPU is divided into:
Page number (*p*), used as an index into a page table which contains base address of each page in physical memory;
Page offset (*d*), combined with base address to define the physical memory address that is sent to the memory unit.



Basic ideas:

- Bring a page into memory only when it is needed (***lazy paging***).
- Pages belonging to a process may be ***memory resident*** or not.
- The system needs some mechanism for distinguishing between resident and non-resident pages. Therefore the ***page table*** associates a valid/invalid bit with each page.
- 1 $\Rightarrow$ **valid/resident** 0 $\Rightarrow$ **invalid/not in physical memory**.
- If a process wishes to access a valid page (a **"HIT"**) it can do so.
- If a process wishes to access an invalid page (a **"MISS"**) then the paging hardware generates a **Page Fault**.

If there is a reference to an invalid page, reference will trap to OS.

- reference to nonexisting page $\rightarrow$ abort
- Just not in memory $\rightarrow$ must bring page into memory.

1. Reference is made to invalid page
2. Page fault generated – call to Operating system.
3. Locate empty frame in physical memory.
4. Swap page into frame.
5. Reset tables, flip validation bit (change to 1)
6. Restart user process.

Of course, this assumes that there is an empty frame in memory...

CS211
Week 11: Memory Management

END OF PART 3

CS211
Week 11: Memory Management

Start of ...

PART 4: Page Replacement

What to do if there is no empty frame

What happens if there is no free frame?

Page replacement

Find some page in memory that is not in use, and swap it out. We need an algorithm for selecting such a page. The algorithm will be evaluated on the basis of the generation of a minimum number of page faults.

Page fault handling is slow, so the performance of the whole system depends heavily on having an efficient ***Page Replacement*** algorithm – one with the lowest possible ***page fault rate***.

To evaluate a page replacement algorithm, we will consider

- An example where a program has the following stream of virtual pages (also called the “page reference string”)

$$\{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\}.$$

  This means it first references page #1, then page #2, then page #3,..., finally page #5. ...
- F, the number of frames on the system
- the algorithm.

For these we shall compute the ***Frame Hit Rate:***

$$\frac{number\ of\ Hits}{Length\ of\ page\ reference\ string}$$

The greater the hit rate, the better.

When a free frame is needed, the victim is the page that has been in memory the longest.
Suppose that $F = 3$, and that at the start no pages are in memory. Then we get

1	2	3	4	1	2	5	1	2	3	4	5
MISS	MISS	MISS	MISS	MISS							
1	2 1	3 2 1									

If we wanted to improve the hit rate, we could try increasing the number of frames. So, let's redo the previous example, but with $F = 4$.

1	2	3	4	1	2	5	1	2	3	4	5
MISS	MISS	MISS	MISS								
1	2 1	3 2 1	4 3 2 1								

This is called **Bélády's Anomaly:** sometimes ***increasing*** the number of frames can ***increase*** the number of page faults!

Optimal algorithm requires the least number of page faults and does not suffer for Belady's Anomaly: *Replace page that will not be used for longest period of time.* However, one needs to know what pages will be referenced in the future. So this is difficult to implement.

LRU: The ***Least Recently Used (LRU)*** algorithm is similar to FIFO but, rather than removing the frame that has been in memory the longest, we remove the one that has not been referenced for the longest time.

Exercise

Re-do the examples from Slides 34 and 35. How many Page faults are generated?

CS211
Week 11: Memory Management

END OF PART 4

CS211
Week 11: Memory Management

Start of ...

PART 5: Memory management in C

Part 5: Memory management in C

We'll finish with a short note about how a process can request memory from the OS. We'll need to recall, from Week 4 (Part 2):

- **A Pointer is special variable that has as its value a memory location**.
- Declaring a pointer to an integer is done by:

```
int *p;
```

- The variable `p` can contains an address.
- Access the contents of that address with `*p`.
  In this context, the `*` symbol is called a ***dereferencer***.

It is reasonable to think of `&` as the inverse of the operator `*`. This is because `&i` means "the address of the variable stored in `i`", while `*p` means "the value stored at the address `p`.

If `i` is an integer with value `23` then

`*(&i)` will evaluate as `23`. However,

`&(*i)` is illegal. ***Why?***

If we declare `p` as a pointer to an integer and set

`p=&i;` then

`&(*p)` will evaluate as the address of `i`.

Question: Is `*(&p)` legal? If so, what will it evaluate as?

When an array of integers is declared, e.g.,

```
int a[10];
```

what actually happens is:

- the system declares a pointer to an integer called `a`.
- the pointer is to `a[0]`, the "***base address***" of the array.
  (Note: this means that `a` is the same as `&a[0]`.)
- space is allocated to the addresses pointed to by `a`, `a+1`, ..., `a+9`.

It is **not** true that arrays and pointers are exactly the same. If we declare:

```
int *p, a[10];
```

the value of the pointer `p` can be reassigned anytime we like, but the value of `a` is fixed.

However, because of this, we use pointers when we wish to pass arrays as functions.

Here is an example of passing an array as an argument to a function.

Given an integer array a, we'll calculate $\sum_{i=0}^{n} a_i$.

02Sum.c

```
4  int sum_pointer(int *p, int n);
   int sum_array(int a[], int n);
6  int main()
   {
8    int a[4] = {1, 99, 40, 60};
     printf("a[0]+a[1]+a[2]=%d\n", sum_array(a,3));
10   printf("a[1]+a[2]+a[3]=%d\n", sum_pointer(a+1,3));
     return(0);
12 }
```

02Sum.c

```
14 int sum_pointer(int *p, int n)
   {
16   int i, sum=0;
     for (i=0; i<n; i++)
18     sum+=*(p+i);
     return(sum);
20 }

22 int sum_array(int a[], int n)
   {
24   int i, sum=0;
     for (i=0; i<n; i++)
26     sum+=a[i];
     return(sum);
28 }
```

Before studying dynamic memory allocation, we will work out how much storage is required by `int`egers, `float`s, `char`acters, and even pointers.

This is because we need to tell the system how many ***bytes*** are required to store an array. Although we can hard-code this into our program, we prefer not to since

- we might make a mistake;
- some systems store data types differently from others (standards for the C language specify the minimum number of bytes used, but this can be exceeded);
- There are more complex `struct`ures which have variable memory requirements.

So we use the `sizeof()` operator. (Note: you've already seen this when using the `read` and `write` functions for `pipe`s).

The `sizeof()` operator returns the number of bytes for a particular data type.

It can take data types (e.g, `float`, `char`) or variable names as its argument.

It returns an unsigned integer.

03SizeOf.c

```
   int x=-123,  *p;   char name[6]="CS211";
16 printf("A char takes      %3lu bytes;\n",
     sizeof(char));
18 printf("A float uses      %3lu bytes;\n", sizeof(float));
   printf("but a double uses %3lu bytes;\n",  sizeof(double));
20 printf("x is requires     %3lu bytes;\n", sizeof(x));
   printf("A pointer needs   %3lu bytes;\n", sizeof(p));
22 printf("Array %s is stored in %3lu bytes;\n",
          name, sizeof(name));
24 printf("enum MONTH takes  %3lu bytes;\n", sizeof(MONTH));
   printf("struct Date takes %3lu bytes.\n", sizeof(Date));
```

The output I get is

```
A char takes          1 bytes;
A float uses          4 bytes;
but a double uses     8 bytes;
x is requires         4 bytes;
A pointer needs       8 bytes;
Array CS211 is stored in   6 bytes;
enum MONTH takes      4 bytes;
struct Date takes    12 bytes.
```

CS211
Week 11: Memory Management

END OF PART 5

CS211
Week 11: Memory Management

Start of ...

PART 6: Dynamic memory allocation

Part 6: Dynamic memory allocation

Having to declare the size of an array in a function header is a huge restriction. Either we have to modify the program every time we change the size of the array, or we have to define the array to be as big as possible.

This is wasteful of time and resources.

To minimise the amount of coding we have to do, and to use resources well, we simply declare an appropriate variable with type

- ***pointer to int*** for an `int`eger vector: `int *v`

Part 6: Dynamic memory allocation

Next we must ask the system to reserve some memory. There are **four** important commands:

- `sizeof()` (see above)
- `calloc(n, sizeof(x))`: `C`ontinuous Memory `Alloc`ation. It will reserve enough space to *n* variables each with same size as *x*. It sets than all to zero.
- `malloc(n*sizeof(int))`: `M`emory `Alloc`ation. As above, but it doesn't do any initialization.
- `free(ptr)`: deallocate the space the begins at the address stored in `ptr`.

The headers for these functions is in `stdlib.h`.

Part 6: Dynamic memory allocation

The important thing is that we can call `calloc()`, `malloc()` and `free()` any time we like. This is what is ***dynamic*** about it.

Both `malloc()` and `calloc()` return pointers to the base of the memory they allocated. However, because they are not for specific pointer types (e.g., pointer to `int` or pointer to `char`) they return ***"void pointers"***.

These ***"void pointers"*** are then re-cast. E.g.,

```
c = (char *) malloc(7*sizeof(char));
d = (float *) calloc(10, sizeof(float));
```

In the example below, the size of the vector `v` is not fixed until the program is run. Note that the `sum` function will work for any sized array.

04Dynamic.c

```
 8  int main(void )
    {
10    int *v, n,  i, ans;
      printf("How many elements are there in v? :");
12    scanf("%d", &n);
      v=(int *)calloc(n, sizeof(int));
14    for (i=0; i < n; i++)
      {
16      printf("Enter v[%d]: ", i);
        scanf("%d", &v[i]);
18    }
      ans= sum(v, n);
20    printf("The sum of the entries of v is %d \n", ans);
      free(v);
22    return(0);
    }
```

CS211
Week 11: Memory Management

END OF PART 6 (and of the Week 11)