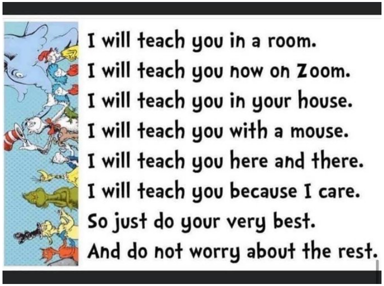
Week 12: Memory Management and Allocation; Review

CS211: Programming and Operating Systems

1 + 2 April, 2020



I will teach you in a room.
I will teach you now on Zoom.
I will teach you in your house.
I will teach you with a mouse.
I will teach you here and there.
I will teach you because I care.
So just do your very best.
And do not worry about the rest.

Source: `sbagley` on Twitter

In our last CS211 class, . . .

In our last CS211 class, . . .

CS211 – Week 12

Tips and Protocols on online Lectures

1. Access through Blackboard... CS211... Virtual Classroom.
2. It can help to join through two devices: laptop for looking at slides, and phone for audio/video. If doing that, use these lines for the second device:

   Wed 3pm: `http://tiny.cc/1920-CS211-W12-L1`
   Thu 1pm: `http://tiny.cc/1920-CS211-W12-L2`
3. Turn video **off** at all times; turn on your mic only when asked.
4. When you enter add a "Chat" message to say "hello" (accessed through the "Collaborate panel" on the bottom left).
5. If you have a question, raise your hand (icon bottom centre).
6. Or ask the question in the Chat section. That's very helpful, since it doesn't cause any interruption, the whole class can see the question, and I can pace my answer.

Schedule for this week

Week 12 (next week)

- Lecture Wed at 3pm; `http://tiny.cc/1920-CS211-W12-L1`
- Lecture Thu at 1pm; `http://tiny.cc/1920-CS211-W12-L2`
- Lab/tutorial Thursday at 3pm;
- Lab/tutorial Friday at 10am.

There will be office hours next and in Study Week; times to be arranged.

Assessment for the rest of the semester

1. Your work for Lab 6 is due 5pm, Friday 3 April.
2. Written assignment, with a deadline of Thursday, 9 April. See `http://www.maths.nuigalway.ie/~niall/CS211/CS211-Assignment.pdf`
3. An on-line exam, based on a multiple choice questions (or similar) scheduled for **12 May 2020**.
4. The on-line exam will follow, roughly, the format of Q1 from last year's exam paper. See `https://www.mis.nuigalway.ie/papers_public/2018_2019/MA/2018_2019_CS211_1_1_2.PDF` A sample version of the paper (using the same software), and solutions, will be provided at least 3 weeks in advance of the exam.

Proposal for calculation of your final grade:

- Labs: 30% (10% for each lab assignment)
- Take-home assignment: 30%
- On-line exam: 40%

Memory management

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

Discussion: why even more precious than, say, CPU time?

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 $53 \circ 16'49.1''N\ 9^\circ 03'38.4''W$)
- The OS can use (backing) storage as temporary main memory, using "swapping".

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.

Contiguous Memory Allocation

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)

Contiguous Memory Allocation

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.

Contiguous Memory Allocation

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.

Contiguous Memory Allocation

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.

Contiguous Memory Allocation

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)

Fragmentation

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**

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").

Paging

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

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

```
// 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.
```

Paging

Try this on a Windows system:

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

```c
// 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;
}
```

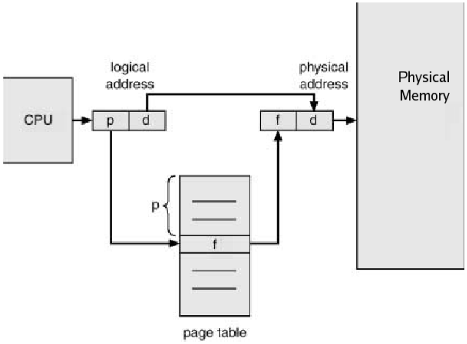
What output do you get?

Paging

- 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.

Paging

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.



Demand Paging

Basic ideas:

- Bring a page into memory only when it is needed (***lazy paging***).
- Pages belonging to a proc 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 $\Rightarrow$ resident   0 $\Rightarrow$ not in physical memory.
- If a proc wishes to access a valid page (a **"HIT"**) it can do so.
- If a proc wishes to access an invalid page (a **"MISS"**) then the paging hardware generates a **Page Fault**.

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.

Page Fault

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***.

Page Replacement

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.

First-In-First-Out (FIFO) Algorithm

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	3									
	1	2									
		1									

First-In-First-Out (FIFO) Algorithm

Suppose instead that $F = 4$:

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

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

Other Algorithms

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 24 and 25. How many Page faults are generated?

Memory management - by a process

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:

- **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***.

Memory management - by a process

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?

Arrays V pointers

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.

Arrays V pointers

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

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

Arrays V pointers

02Sum.c

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

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

The `sizeof()` things

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. Two problems with this might be:

- we have enough things to remember without knowing how many bytes in a float.
- some systems store data types differently from others.
- There are more complex `struct`ures which have variable memory requirements.

The `sizeof()` things

The good news is that 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 (Why?).

03SizeOf.c

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

The `sizeof()` things

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.
```

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`

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`.

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));
```

Dynamic memory allocation — An example

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

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

Module review

The topics we have covered (not necessarily in order) are:

(a) What is an OS?

(b) Computer History: from batch systems to distributed systems.

(c) Programming with processes: `fork`, `getpid`

(d) Interprocess communication with `pipe`

(e) Threads

(f) Scheduling Algorithms.

(g) Concurrency; race conditions; Critical Sections; locks; Semaphores;

(h) The dining Philosophers problem

(i) Memory management.

Module review

In C Programming, we had

(i) Basic strcture

(ii) `if else`, etc

(iii) Loops: `for`, `while`, `do... while`;

(iv) Input (`printf`) and output `scanf`

(v) Functions, including argument lists, return values, void, call-by-value; call-by-reference.

(vi) Pointers

(vii) Strings

(viii) Files: `fopen`, `fclose`, reading and writing

(ix) User-defined types `enum`, `struct`, `typedef`

(x) Dynamic memory allocation: `calloc`, `malloc`, `free`

THE END!!

I hope you have enjoyed CS211, and found it interesting and/or useful.
Thank you for your commitment, collaboration, interest and insights.