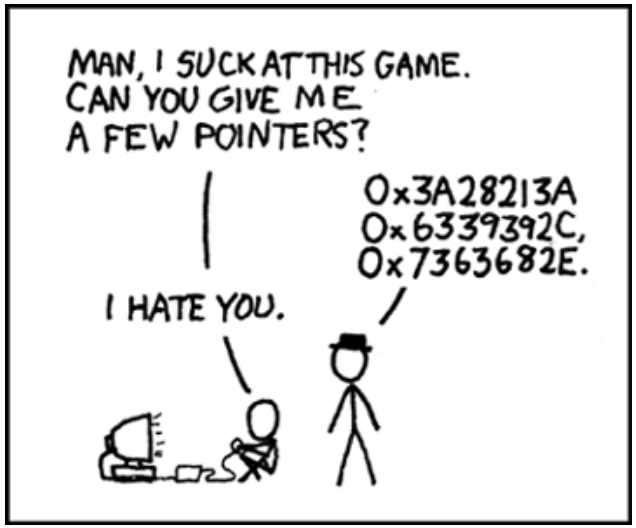
Week 11: (Safe states and) User-defined types in C, and Memory Management

CS211: Programming and Operating Systems

25 + 26 March 2020



https://xkcd.com/138

CS211 – Week 11

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: `https://bit.ly/2wEsbyK`
   Thu 1pm: `https://bit.ly/3bqi27C`
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 the rest of the semester

Week 11 (this week)

- Lecture Wednesday at 3pm;
- Lecture Thursday at 1pm;
- Lab Thursday at 3pm;
- Lab Friday at 10am.

Week 12 (next week)

- Lecture Wednesday at 3pm;
- Lecture Thursday at 1pm;
- Lab or tutorial Thursday at 3pm;
- Lab or tutorial Friday at 10am.

There will be no scheduled labs or tutorials after next week. However, I will host office hours at times to be arranged.

Assessment for the rest of the semester

1. Your work for Lab 6 is due 5pm, Friday 3 April.
2. An written assignment will be given by Friday of this week, with a deadline of Thursday, 9 April.
3. An on-line exam, based on a multiple choice questions (or similar) will be held a time to be determined by the central university administration (later this week, I hope!)
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, and solutions, will be provided at least 3 weeks in advance of the exam.

Assessment for the rest of the semester

Suggestion for calculation of your final grade:

- 10% for each lab assignment
- 30% for the take-home assignment.
- 40% for the on-line exam.

Discuss:

Today, in CS211, . . .

1. Schedule for the rest of the semester
2. Assessment for the rest of the semester
3. Recall... Deadlock
   - Safe sequences
4. User-defined types
5. `typedef`
6. `enum`
   - `enum+typedef`
   - How it really works
7. `struct`
   - `struct+typedef`
   - `typedef+enum+struct`
8. “Textbook” examples
9. Memory and Storage Management
10. Memory management
11. Logical and Physical Addresses
12. Swapping
13. Exercise

Recall... Deadlock

Deadlock is when two or more procs are waiting indefinitely for an event that can only be caused by one of the waiting processes. It can arise if four conditions hold simultaneously

1. ***Mutual exclusion***: only one process can have access to a particular resource at any given time.
2. ***Hold and wait***: a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. ***No preemption***: a resource can only be released voluntarily by the process holding it, after that process has completed its task.
4. ***Circular wait***.

One possible solution to the deadlock problem, but which requires extra information is represented as the **Banker's Algorithm**. In particular we need to know:

- The number of resources the system has;
- The number currently allocated to each process;
- **The maximum that any process might request**.

With this information, it should be possible to ensure that a circular wait condition does not hold. To understand this, we need to concepts of **safe states** and **safe sequences**.

- A **resources-allocation state** is the number of available and allocated resources, and the maximum demands of processes.
- A state is **safe** if the system can allocated resources to each process, and still avoid deadlock.
- A **safe sequence** is a sequence of processes such that their resource requests can be granted, in order, with no process having to wait indefinitely.

We did one example at the end of the last lecture. Now we'll do another one.

Example (From 2017/2018 CS211 exam)

Suppose we have a system with three resource types:

9 instances of A, 3 instances of B and 6 instances of C.

Consider four processes P_1, P_2 , P_3 , P_4 , with, at a given point in time, current allocations and total requirements given by

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_1	3	2	2	1	0	0
P_2	6	1	3	6	1	2
P_3	3	1	4	2	1	1
P_4	4	2	2	0	0	2

Is this a safe state? If it is, give a safe sequence?

9 instances of A, 3 instances of B and 6 instances of C.

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_1	3	2	2	1	0	0
P_2	6	1	3	6	1	2
P_3	3	1	4	2	1	1
P_4	4	2	2	0	0	2

User-defined types

In C (and other languages) we often define data types of their own. This can be done for two primary reasons:

(a) It makes the code more readable and, thus, more likely to be correct;

(b) It allows us to build more complicated data-types.

We will look at doing this with

1. `typedef`: Give an intuitive alias to an existing type.
2. `enum`: create an enumerated listed type. Underlying data type is usually integer, but the programmer doesn't need to know which.
3. `struct`: build heterogeneous data types where different elements can be different types and of different dimensions.

User-defined types

Motivation

We have two reasons for studying the use of `typedef`, `enum`, and `struct`.

1. They are key concepts in the C programming language;
2. They are widely used in Operating System code

typedef

Consider the following piece of code:

```
1 float compute(float Angle)
  {
3     ...
    if (Angle == 90)
5     ...
  }
```

And the function is called as

```
  float theta, pi=3.14159;
2 theta = compute(pi/2);
```

Obviously this will compile OK, but could give very surprising results.
The problem here is that, although the variable *Angle* is a double, we don't know if it is measured in e.g., Radians or Degrees.
One way to resolve this is to make a date-type for, say, degrees, using a ***type definition***

typedef

In C, we can define our own types using the keyword `typedef`.

`typedef` syntax

Syntax: `typedef` *original-name MY-ALIAS*;
Example: `typedef` *float EURO*;

This allows us to refer to the data type *origin-name* as *MY-ALIAS*.
If we follow the example given above, then we can use the following in our code:

```
  typedef float EURO;
...
  EURO gross_cost, VAT;
```

Advantages of using `typedef`

- Makes the code more readable;
- Makes it clearer to someone using our code what was intended;
- Makes it easier to make change later, e.g., to use `double` to store monitary values.

typedef

Returning to our example from earlier, we could create a data-type for angles measured in degrees (as opposed to radians):

```
1  typedef float Degree;

3  Degree compute(Degree Angle)
   {
5    .
     .
7    if (Angle == 90)
       ...
9  }
```

If the programmer checks the function header before writing the calling function, at least she will know how to call it.

(This idea is sometimes referred to as ***information hiding*** in software engineering).

enum

Sometimes we have a variable of a particular value that can only take on one of a certain set of values.
For example, consider a program that simulates a card game. Each card has a **suit** that is one of **clubs** ♣, **diamonds** ♢, **hearts** ♡, or **spades** ♠.
This is an example of where we might like to use an `enum`erative data type.

`enum` syntax (V1)

Syntax: `enum {list, of, values} var-name`
Example: `enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1;`

Now *s1* is a variable that can take on any of the values *CLUBS*, *DIAMONDS*, *HEARTS*, *SPADES*.
However, what we probably *really* want is to create a new type called suit. (PTO)

enum syntax (V2)

Syntax: `enum type-name {list, of, values}`

Example: `enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};`

Now we have a data type called *enum suit*, and we can declare variables of that type, e.g.,

```
1  enum suit {CLUBS, DIAMONDS, HEARTS,  SPADES};
...
3  enum suit s1=CLUBS;
```

But, even better would be to combine `enum` and `typedef`. (PTO)

enum+typedef syntax (V3)

Syntax: `typedef enum {list, of, values} type-name`

Example: `typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} SUIT ;`

Now we have a data type called *SUIT*, and we can declare variables of that type, e.g.,

```
1  typedef enum {CLUBS, DIAMONDS, HEARTS,  SPADES} SUIT;
...
3  SUIT s1=CLUBS;
...
5  if (s1 == HEARTS)
   {
7     ...
   }
```

Under the hood, `enum` is really just creating constants whose values are `int`s.
For example, if we use

```
typedef enum {CLUBS, DIAMONDS, HEARTS,  SPADES} SUIT;
```

Then we are just defining the constants

```
1 int CLUBS=0, DIAMONDS=1, HEARTS=2,  SPADES=3.
```

And the data type `SUIT` is really just an `int`.
For example, if we call

```
1 SUIT s1=DIAMONDS;
  s1 ++;
```

Then `s1==HEARTS` will evaluate as true.

`struct`

Finally we get to our first *real* derived data type: `struct`ures.

`Struct`ures provide a way to *aggregate* data types. It is *heterogeneous*, meaning that a `struct` can be made up of different data types (unlike, say, and array, where all entries are of the same type).

The components of a `struct` are called ***members***, and are accessed using the "." (dot) operator.

struct syntax (V1)

Syntax:

```
struct struct-name {
    type1 member-name1;
    type2 member-name2;
        ...
    type2 member-name2;
};
```

Example:

```
struct card {
    int pips;
    char suit;
};
```

We declare variables of this type as follows:

```
struct card ace_of_hearts;
ace_of_hearts.suit = 'h';
ace_of_hearts.pips = 1;
```

We can copy structs, but not check for equality. E.g.,

```
struct card c2;
c1 = ace_of_hearts; // legal
if (c1 == ace_of_hearts) // not legal
```

It is very common to use `typedef` when defining a `struct`.

`struct+typedef` syntax (V2)

Syntax:

```
typedef {
    type1 member-name1;
    type2 member-name2;
        ...
    type2 member-name2;
} struct-name;
```

Example:

```
typedef {
    int pips;
    SUIT suit;
} CARD;
```

One can also create arrays of structures, e.g.,

```
CARD deck[52];
```

Example: to use `enum`, `struct` and `typedef` to implement an ADT called `Date` that stores that stores the day, month, and year as elements:

```
1  typedef enum {Jan=1, Feb, Mar, Apr, May, Jun,
       Jul, Aug, Sep, Oct, Nov, Dec} MONTH;

   typedef struct
5  {
     int DayofMonth;
7    MONTH Month;
     int Year;
9  } Date;
```

“Textbook” examples

The xv6 Proc Structure, taken from Figure 4.5 of the text-book
http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf

```
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
    RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
   char *mem;                  // Start of process memory
   uint sz;                    // Size of process memory
   char *kstack;               // Bottom of kernel stack for this process
   enum proc_state state;      // Process state
   int pid;                    // Process ID
   struct proc *parent;        // Parent process
   void *chan;                 // If non-zero, sleeping on chan
   int killed;                 // If non-zero, have been killed
   struct file *ofile[NOFILE]; // Open files
   struct inode *cwd;          // Current directory
   struct context context;     // Switch here to run process
   struct trapframe *tf;       // Trap frame for the current interrupt
};
```

“Textbook” examples

```
typedef enum _state {RUNNING, FINISHED, READY} State;

struct process
{
  int burst; // remaining CPU time required.
  int finish_time;   // value of t at which process finished
  int response_time; // time process first gets to RUNNING queue
  State my_state;
};
```

Memory and Storage Management

So far, in the "OS" component of ***Programming and Operating Systems*** 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.

Memory and Storage Management

Roughly, “storage” refers to how long-term data is maintained, particularlly while a program is not running. Traditionally, this was on disks (drives), more typically it is in the cloud.
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
- File Systems

Memory management

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 scheduling algorithm 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.

We shall see that physical memory is not large enough to accommodate all the procs, and so secondary storage must be used as "*over-flow*" memory.

Logical and Physical Addresses

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
```

Logical and Physical Addresses

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.

Swapping

In a multiprogramming environment, **swapping** occurs when memory space belonging to a particular process is moved temporarily from main memory to a ***backing store***.

Backing store is usually a specially allocated section of a fast magnetic disk. Often (e.g., on Linux) it is referred to as the **swap space** (some times “page space”, e.g., on Windows).

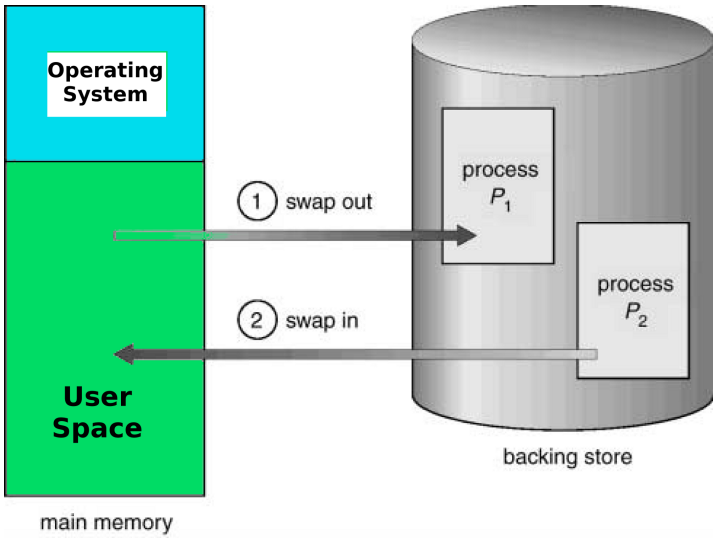
Swapping is used because the space allocated to one process in main memory is required by another. When execution of the original process is to be continued, it is swapped back into main memory, usually to the location is previously occupied.

Swapping

When the scheduler decides to run the next process:

- The **dispatcher** checks if it is already in (main) memory
- If it is not, and if there is no free space, it chooses a proc to swap out and moves its allocated memory to the swap space
- The new process is ***swapped*** into its space. The CPU registers are loaded and execution of the new proc begins.

For CPU usage to be efficient, the time quantum (or time slice) of the scheduler must be longer than the time required for the swapping operation.



Exercise

Exercise (11.1)

Suppose we have three resource types, A, B and C, and 5 processes, P_0, ..., P_4. We have

10 instances of type A, 5 instances of type B, and
7 instances of type C.

At a given point in time the allocations, maximal demands and availability of each of the resources is given as follows.

	Total Reqs			Current Alloc		
	A	B	C	A	B	C
P_0	7	5	3	0	1	0
P_1	4	2	2	**3**	0	**2**
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

We know (from class work) that this is a safe state. However, could we make an additional initial allocation of $(0, 2, 0)$ to P_0? That is, would that lead to a safe state?