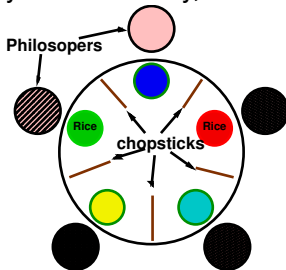


Week 10: Deadlock Handling; User-defined types

CS211: Programming and Operating Systems

Niall Madden (Niall.Madden@NUIGalway.ie)

Wednesday and Thursday, 21+22 April 2021



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%.

31st May

Final Reminder :

VOTE

This is the right place if you'd like to learn about ...

1 Part 1: Those Philosophers again

- Deadlock handling
- Semaphores
- Solutions

2 Part 2: Deadlock Detection

- Recall: Resource Allocation Graph

3 Part 3: Deadlock Avoidance

- Safe states and sequences
- Exercise

4 Part 4: User-defined types

- typedef
- enum
- enum+typedef
- How it really works

5 Part 5: struct

- struct+typedef
- typedef+enum+struct
- "Textbook" examples

(Today)

Wednesday.

finish Thursday.

Thursday

CS211

Week 10: Deadlock Handling; User-defined types

Start of ...

PART 1: The Dining Philosophers again

Part 1: Those Philosophers again

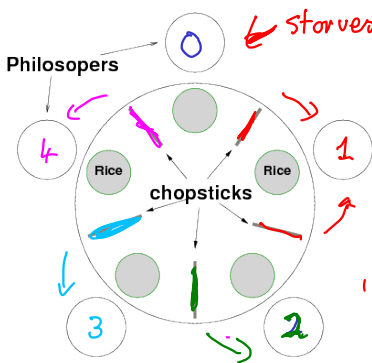
(See also Section 31.6 of the textbook)

Starvation occurs when a particular process is always waiting for a particular semaphore to become available.

The ideas of **Deadlock** and **Starvation** are exhibited in the classic synchronization problem: ***The Dining Philosophers Problem.***

- There are five philosophers seated at a round table.
- Each has a bowl of **rice** in front of them.
- There are **five chopsticks** on the table, each between a pair of philosophers.
- They spend their day alternating between eating and thinking.
- But to eat, they need both the chopsticks to their left and right...

Part 1: Those Philosophers again



If a philosopher is hungry, she will try to pick up the chopstick to the left and then the chopstick to the right. If she manages to do this they will eat for a while before putting down both and thinking for a while. However, if she picks up one, she will not let go of it until she can pick up the second and eat.

Suppose each of the picks up the chopstick to their left. No chopsticks remain on the table so we reach a state of deadlock.

The challenge is to find a solution so that

- **Deadlock** does not occur.
- neither does **starvation** where one philosopher never gets to eat.

In general operating systems take one of three approaches to deal with deadlock:

- 1 Ensure that the system will never enter a deadlock state.
- 2 Allow the system to enter a deadlock state and then recover.

In Case 1, there are two possibilities:

- **Prevention:** we ensure at least one of the four necessary conditions never hold.
- **Deadlock avoidance:** where the OS uses *a priori* information about the procs the devise an algorithm to circumvent deadlock.

In Case 2, the OS must have mechanisms for first detecting deadlock and then dealing with it. (More about this in a minute).

4 conditions: *mutual Exclusion*
No preemption
Circular Wait
?

Recall that a **Semaphore**, S , is an integer variable that can only be accessed via one of two operations:

- **Test/sem_wait**, $P(S)$, and
- **Increment/sem_post**, $V(S)$.

It is “special” in the sense that the two operations on it can’t be interrupted.

In Lab 6, we implement a solution to a **Race Condition** when we have multiple fork’ed subprocesses, problem by implementing a semaphore via a pipe.

However, POSIX systems already come with a **semaphore** solution when working with pthreads. To use it

- sem_t S; declares S to be a semaphore. *(recall from Week 8)*
- sem_init(&S, 0, 1); initialises it as a binary semaphore. *ignore (means “shored”). semaphore count: 1 for a binary semaph*
- sem_wait(&S) checks if it is available and, if it is, grabs it.
- sem_post(&S) releases it, causing awaiting thread to wake up and try taking it. *ie increments S.*

If interested, see 01_pthread_semaphore.c for an example.

The Dining Philosophers Problem can be represented as follows.

"forever"

From section 31.6 of the textbook

```

1 while (1) {
2   think();
3   get_chopsticks();
4   eat();
5   put_chopsticks();
6 }

```

Handwritten notes:
 - An arrow points from "forever" to the while loop.
 - A red line under "think();" is labeled "ie 'sleep'".
 - A bracket on the right side of the loop body is labeled "need to get left & get right chopstick."
 - An arrow points from "get_chopsticks();" to the bracket.

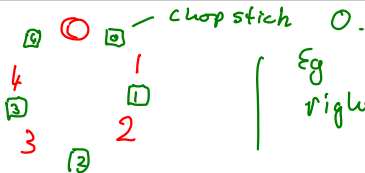
Here the key is to write versions of `get_chopsticks()` and `put_chopsticks()` which result in no deadlock, none starves (also, ideally, concurrency is optimised).

Within these functions we will have others:

```

1 int left(int p) { return p; }
2 int right(int p) { return (p + 1) % 5; }

```



Next we will use semaphores to control access to the chopsticks. Let us suppose there are five:

```
sem_t chopsticks[5].
```

When we can write the get/put functions as

```
1 void get_chopsticks() {  
3     sem_wait(chopsticks[left(p)]); ← pick up left  
    sem_wait(chopsticks[right(p)]); ← pick up right .  
}  
  
7 void putchopsticks() {  
    sem_post(chopsticks[left(p)]);  
9     sem_post(chopsticks[right(p)]);  
}
```

Implemented like this, we get the expected deadlock. We'll now try to get a solution.

The most famous solution, which was proposed by the problem's inventor, **Edsger Dijkstra**, is simply to have one philosopher attempt to pick up the chopstick to their right first, while everyone else tries to get the one on their left first.

```
1  void getchopsticks() {  
2      if (p == 4) {  
3          sem_wait(chopsticks[right(p)]);  
4          sem_wait(chopsticks[left(p)]);  
5      } else {  
6          sem_wait(chopsticks[left(p)]);  
7          sem_wait(chopsticks[right(p)]);  
8      }  
9  }
```

Can you convince yourself this works?

Exercise.

CS211

Week 10: Deadlock Handling; User-defined types

END OF PART 1

CS211
Week 10: Deadlock Handling; User-defined types

Start of ...

PART 2: Deadlock Detection

Competition for resources can be described using a directed graph called a **resource allocation graph**.

This graph has two sets of **vertices**:

■ **Processes:** P_0, P_1, \dots, P_n and

/ ■ **Resources:** R_0, R_1, \dots, R_m

and **Edges**

■ from P_j to R_k is process j as requested resource ~~resource~~ k but not yet been allocated it,

■ from R_k to P_k is process j as been allocated resource ~~resource~~ k and not yet released it.



R_k

R_k



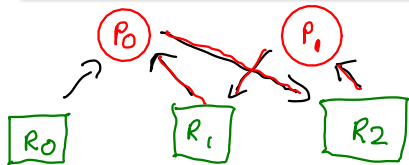
- If there are no cycles, there is no deadlock,
- If there is deadlock, there must be a cycle
- If there is a cycle, there **may** be deadlock
- If each resource has only one instance, and there is a cycle, then there is deadlock

Example

There are two processes, P_0 and P_1 , and three resources, R_0 , R_1 and R_2 .

- P_0 requests all three resources. It is allocated R_0 and R_1 .
- P_1 requests R_1 and R_2 . It is allocated R_2 .

Is the system in deadlock?



This is deadlocked,
since

$P_0 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1 \rightarrow P_0$
is a cycle.

Detection of deadlock is an important, but difficult problem.

Using ideas like the Resource Allocation Graph, the system's needs can be represented mathematically, and then a deadlock state checked for.

identical.

Idea

Suppose that a system has n processes, and a total of m resources that it can allocate. Resources can only be requested or released one at a time. Then the system is *Deadlock free* if the following conditions hold:

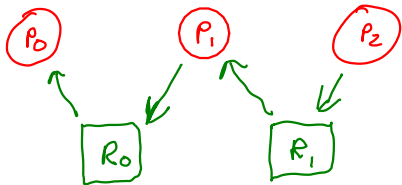
- (i) each process requires at least 1 resource and at most m .
- (ii) the sum of all their requirements is less than $m + n$.

Example

Use a resource allocation graph to show that *there is no deadlock*

- if there are $n = 3$ processes, and
- $m = 2$ resources
- Resources can only be requested or released one at a time.
- (i) each process requires at least 1 resource and at most 2.
- (ii) the sum of all their requirements is at most 4 (i.e., less than $m + n = 5$).

P_0 has R_0
 P_2 requests R_1
 P_1 requests R_0
 P_1 has R_1

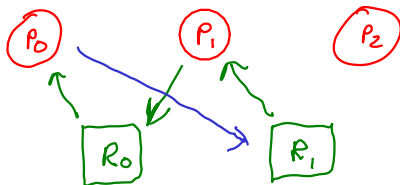


Try examples yourself.

Example

In previous example, suppose we remove the requirement that all processes request at least one resource. Might there be deadlock?

P_0 has R_0
 P_0 requests R_1
 P_1 requests R_0
 P_1 has R_1



DEADLOCK.

CS211

Week 10: Deadlock Handling; User-defined types

END OF PART 2

CS211

Week 10: Deadlock Handling; User-defined types

Start of ...

PART 3: Deadlock Avoidance

Esp: Banker's Alg.

(Again!!)

Part 3: Deadlock Avoidance

One possible solution to the deadlock problem, but which requires extra information is represented as the **Banker's Algorithm**. Simply put:

Banker's Algorithm

Only allocate resources to a process if you can allocate **all** the resources it requests.

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. *Eventually.*
- A state is **safe** if the system can allocate 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.

It is, perhaps, easiest explained through an example.

Usually, not every sequence is a safe sequence.

Example (From old exam paper)

Suppose we have a system with

- three resource types, A , B and C . There are in total 10 instances of type A , 5 instances of type B and 7 instances of type C .
- 5 processes, P_0, P_1, \dots, P_4 .

At a given point in time the allocations, maximal demands and availability of each of the resources is given below. Determine if it is a safe state, and give a safe sequence.

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

25

Note: can't make all allocations at once.

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

	Total Reqs			Current Alloc			Needs			
	A	B	C	A	B	C	A	B	C	
P_0	7	5	3	0	1	0	7	4	3	
P_1	3	2	2	2	0	0	1	2	2	
P_2	9	0	2	3	0	2	6	0	0	
P_3	2	2	2	2	1	1	0	1	1	
P_4	4	3	3	0	0	2	4	3	1	
				7	2	5				

Safe Seq:
 $P_1, P_1, P_4, P_2,$
 then P_0 .

The current allocation is $(7, 2, 5)$ i.e. $7 \times A$ and 2 B's and 5 of C. So we have
 $(10, 5, 7) - (7, 2, 5) = (3, 3, 2)$ available.
 So we can allocate $(1, 2, 2)$ to P_1 , leaving $(2, 1, 0)$ available. When P_1 finishes, it releases $(3, 2, 2)$ giving $(5, 3, 2)$ available. Can now do P_3 , then P_4 , then P_0

**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

Handwritten annotations below the table: $\overline{6}$ under A, $\overline{4}$ under B, $\overline{11}$ under C.

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

Part 3: Deadlock Avoidance

Safe states and sequences

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

	Total Reqs			Current Alloc			Needs		
	A	B	C	A	B	C	A	B	C
P ₁	3	2	2	1	0	0	2	2	2
P ₂	6	1	3	6	1	2	0	0	1
P ₃	3	1	4	2	1	1	1	0	3
P ₄	4	2	2	0	0	2	4	2	0
	9 2 5								

Notation: (x, y, z)

Mean x of A
 y of B
 z of C

Available: $(9, 3, 6) - (9, 2, 5) = (0, 1, 1)$

Step 1: $(0, 1, 1) \xrightarrow[\text{allocate to } P_2]{(0, 0, 1)} (0, 1, 0) \xrightarrow[\text{P}_2 \text{ finish}]{(6, 1, 3)} (6, 2, 3) \text{ available}$

Step 2: $(6, 2, 3) \xrightarrow[\text{P}_1]{(2, 2, 2)} (4, 0, 1) \xrightarrow[\text{P}_1 \text{ finish}]{(3, 2, 2)} (7, 2, 3)$

Step 3: $(7, 2, 3) \xrightarrow[\text{P}_3]{(1, 0, 3)} (6, 2, 0) \xrightarrow[\text{P}_3 \text{ finish}]{(9, 3, 4)}$

Step 4: Allocate to P₄.

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

1. This is a safe state.
2. $\{P_2, P_1, P_3, P_4\}$ is an example of a safe sequence.

Exercise (10.1)

A system has $m = 4$ identical resources, and $n = 3$ processes, P_1 , P_2 and P_3 , which make a request for 1, 2, and 3 resources, respectively. Draw the resource allocation graph for the scenario. Can the system reach deadlock?

Exercise (10.2)

A system has $m = 4$ identical resources, and $n = 3$ processes. The processes make a total request for 6 resources. If one process makes no request, can the system reach deadlock?

Exercise (10.3)

Recall an example where we had 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 (take notes). Determine if it is a safe state, and give a safe

		Total Reqs			Current Alloc		
		A	B	C	A	B	C
sequence.	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

Could we make an additional initial allocation of $(0, 2, 0)$ to P_0 ? That is, would that lead to a safe state?

CS211

Week 10: Deadlock Handling; User-defined types

END OF PART 3

CS211

Week 10: Deadlock Handling; User-defined types

Start of ...

PART 4: User-defined types

Recorded

Thursday, 22 Apr
at 13:20

Part 4: 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.

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

Consider the following piece of code:

```
1 float compute(float Angle)
2 {
3     ...
4     if (Angle == 90)
5         ...
6 }
```

Expected angle to
be in degrees

And the function is called as

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

angle is in radians.

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**.

~

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 *original-name* as *MY-ALIAS*.

If we follow the example given above, then we can use the following in our code:

```
2 ...  
   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 monetary values.]
- ✓ → self documenting code.

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)  
4 {  
5     .  
7     if (Angle == 90)  
8         ...  
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).

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 **enumerative** data type.

enum syntax (V1)

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

Example: `enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1;`

variable .

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)

So can set `s1 = CLUBS;` or
check if `(s1 == HEARTS)`

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      ...  
    }
```

This version is the most commonly used!

Under the hood, `enum` is really just creating constants whose values are `ints`.

For example, if we use

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

Then we are just defining the constants

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

And the data type `SUIT` is really just an `int`.

For example, if we call

```
SUIT s1=DIAMONDS; → sets s1 = 1  
s1 ++; → sets s1 = 2 (ie s1 == HEARTS).
```

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

Also: we can change the default values.
`typedef enum {CLUBS=1, DIAMOND=2, etc};`

CS211

Week 10: Deadlock Handling; User-defined types

END OF PART 4

CS211

Week 10: Deadlock Handling; User-defined types

Start of ...

PART 5: struct

Part 5: struct

Finally we get to our first *real* derived data type: structures.

Structures 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.

Part 5: struct

struct syntax (V1)

Syntax:

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

Example:

```
struct card {  
    int pips; -ie number 1,...,13.  
    char suit;  
};
```

We declare variables of this type as follows:

```
2 struct card ace_of_hearts;  
   ace_of_hearts.suit = 'h'; ← h for heart.  
   ace_of_hearts.pips = 1;
```

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

```
1 struct card c2;  
   c2 = ace_of_hearts; // legal  
3 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;  
    ...  
    typek member-namek;  
} struct-name;
```

Example:

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

From
Port 4

Must be

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

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

defined
earlier in
the code.

Example: to use `enum`, `struct` and `typedef` to implement an ADT called `Date` 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;
5 typedef struct
6 {
7     int DayofMonth;
8     MONTH Month;
9     int Year;
10 } Date;
```

ADT = Abstract
Data Type

The xv6 Proc Structure, taken from Figure 4.5 of the text-book

<http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-intro.pdf>

```

1 // the different states a process can be in
2 enum proc_state { UNUSED, EMBRYO, SLEEPING,
3   RUNNABLE, RUNNING, ZOMBIE };
4
5 // the information xv6 tracks about each process
6 // including its register context and state
7 struct proc {
8   char *mem; // Start of process memory
9   uint sz; // Size of process memory
10  char *kstack; // Bottom of kernel stack for this process
11  enum proc_state state; // Process state
12  int pid; // Process ID
13  struct proc *parent; // Parent process
14  void *chan; // If non-zero, sleeping on chan
15  int killed; // If non-zero, have been killed
16  struct file *ofile[NOFILE]; // Open files
17  struct inode *cwd; // Current directory
18  struct context context; // Switch here to run process
19  struct trapframe *tf; // Trap frame for the current interrupt
20 };

```


Handwritten annotations:

- waiting* (above EMBRYO)
- ready* (above RUNNABLE)
- terminated.* (above ZOMBIE)
- returned by getpid().* (with an arrow pointing to the `pid` field)


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

4 struct process
5 {
6     int burst; // remaining CPU time required.
7     int finish_time; // value of t at which process finished
8     int response_time; // time process first gets to RUNNING queue
9     State my_state;
10 };

```



CS211

Week 10: Deadlock Handling; User-defined types

END OF PART 5

and
of
Week 10.