

(1) misbehavior:

20 panes, start showing 10 panes  
Uncheck list.

20 panes, start showing 10 panes

(2) memory leak:(3) Invalid / illegal memory access

segmentation fault, it kernel. fine work.

(a) ~~use~~ our application has address space  
But in our code we are trying to access  
memory that is Not ours

(b) read only memory, page given to us  
is read only but we are trying to write to it

(c) allocated memory and free it and  
again try to access it. Then it will  
give segmentation fault.

(d) As an appn we should not

touch the kernel mem, But some time  
we pass some api and says directly  
touch the kernel memory.

(4) invalid instructions

compile for processor X and run  
on another processor

(5) Synchronization issue among the  
threads or many workers

global data has been modified (293)

### Context

w.r.t to the kernel

there may be diff context like process context, interrupt context, switching context.

\* we have shared data and these data may not be accessed by all context simultaneously, we should allow only one context.

\* Debugging is a technique to find where a particular problem is in our program. what we want to debug, or what we want to find out in our program.

\* What kind of info I need when I can find out where the problem is

① I want to know the exact line of the code where this problem is happening.

② I want to have the local variables data at the particular time when the segmentation fault has occurred.

③ I want to have global data when the segmentation fault has occurred.

④ All other pr I want to have.

⑤ I want to have the backtrace from where this particular function has called.

(294) Example

char \*ptr = NULL

char \*A = "google.com"

We need to find the following

- (I) Instruction address where problem is  
(II) what local variable contains  
(III) what were the parameters to the fn.  
who has called this fn.  
(IV) what were the data of global variables  
(V) information of registers i.e. contents of register when problem has occurred

→ to find out this we need  
Instruction pointed (this contains address of fault instruction)

→ to get local variable we need stack (we need the whole stack data)

\* one of ways to pass parameter to fn is stack or register

Data segment (what global variable contains at least)  
Heap segment

[this register in CPU]

can have info we need state (299)

\* customer place.

\* this all info will be present in corefile

Corefile

↓

footprint of process i.e. core dump address of process.

\* which signal has generated in corefile?

what information a coredump file contains

① process state:

→ signal info like signal no, error

which signal has <sup>no</sup> generated → set of pending signals  
process → set of held signals (blocked  
file → signal  
→ process id, process parent id and ...  
→ set of registers (sp or esp)

② process info:

→ state of the current process

→ state in string like R, S, Z ...

→ where the process is in Zombie state

→ nice value of the process

→ process flags

→ pid, ppid

→ process name

→ command line arguments of the program

### (3) Thread specific info:

(296)

→ process status for the thread  
from a thread



current thread + other threads  
info

### (4) memory info of memory mapped file

→ No of files mapped

→ No of files mapped

→ start add

→ end add

### (5) process add space:

→ like data segment, stack segment,  
code segment or heap

who generate the code file and when  
will this be generated

NULL pointer

CPU particular add & virtual address



map to phy ADD



Not mapped



page fault



address valid or not

Bad addr Now kernel will

generate signal to prevent segmentation fault signal in our case

- (297) Once it generates the segmentation fault signal to be process, it will try to create a core file and embed all the above info.
- \* It will create the core file in current working directory.
  - \* most of the time core file is platform dependent.
  - \* core object file.
- Elf format.
- \* we have to tell kernel to generate a core dump whenever core dump occurs. By default is enabled.
  - \* each process has some resource limit.
  - \* kernel maintains what resource for each process and tells kernel to generate a core file size.
  - \* for some or any signal, core file will be generated.
  - \* By default limit of core dump file is 0.
- Note: ELF file is collection of sections or collection of segments

Basically: relocation obj contains section and executable obj contains segments

## 298 Debugging without symbol and with symbol

- When we don't have any debugging symbols we disassemble the code.
- \* For each frame stack frame will be allocated.
- \* ABI: Appn Binary interface that explains how the parameter is passed to the fn.

(299)

## Lambda expression C++11/C++14

Lambda expression a shorthand syntax for creating a function object.

auto isOdd =  $\lambda$  (int n) { return n % 2; }

Auto a = isOdd(3) // return true.

~~def~~ declare, define and instantiate one of these function objects inline in the code.

If many. do like

class isOdd

\* Before calling  
() for using from  
where scope  
or object or self  
close

L

public:

int bool operator () (int n)

{  
    return n % 2;  
}

}

or

predicate

count\_if(begin(v), end(v), isOdd v)  
     $\lambda$  (int n) { return n % 2; }

✓ This creating a unnamed

temp. dt type is Odd. It can be

replaced with Lambda exp like

count\_if(begin(v), end(v),  $\lambda$  (int n) { return  
    n % 2; })

ClosureType: This hybrid generated type  
compiler generate for most lambda exp:  
like lbes.

auto isOdd = [ ](int) & return n%2;

then compiler generate

class isOdd

1. operator(int)  
& return n%2;  
}

Closure: Actually the instance of closure.

This object generated for most class.

This runtime construct, i.e. instance itself.

auto isOdd = [ ](int) & return n%2;

auto fn = isOdd

Decorators, defn and inheritance

fn

\* For calling lambda we need objects

\* class\_name() or class\_name() will create temp  
Objects

## Lambda Capture

(30)

↓  
It specifies what and how local variable and parameters will be available to the lambda.

auto fn = [...] (int n) { ... };

Ans:

Ex:

auto mult = 3

auto ismult() = [ ](int n){

return n \* mult; }

Name lambda know about mult

so capture block is used

So why some lambda and multi in edit scope. It will treat multi as member variable in class generated by compiler like

class ismult{  
int m\_mult;

h. public:

ismult(int mult): m\_mult(mult)

{ }

    bco1 operator(int a)

    { return a \* m\_mult; }

how to assign value. value to local variable using lambda v. (302)

### Explicit capture:

Specifies how and what to capture, by reference or value.

A explicit capture by reference.

```
#include<iostream>  
int main()
```

```
{ int sum = 0;
```

```
for_each(begin(v), end(v),
```

```
[&sum](int n) {sum = sum + n;});
```

### Ex-2

```
auto mult = 3;
```

```
for_each(begin(v), end(v), [mult](int n))
```

```
{ return n * mult; }
```

This captures by value \$0

We have copy of mult in lambda exp.

~~If we want to change the captured  
values then we have to use by capture  
it using by reference.~~

like:

auto sum = 0

auto isMultiple = ~~function~~  
sum (int n) {

~~return~~ sum = sum + n; }

It will not allow to change sum.  
because # actually the compiler will  
generate Overload of operator () (parenthesis)  
as constant. i.e why we are not able to  
change sum by using reference.

like

struct sum

L sum (int sum): m\_sum (sum) h. }

void operator () (int n) const

L' m\_sum = m\_sum + n; }

}

int m\_sum

this is generated

?

lambda

change m\_nm

so we are not able to

To resolve this we will capture by  
reference.

like

(304)

int sum = 0

for each ( begin(v), end(v), [8] ) ( int  
{ sum = sum + n; } );

Default capture:

It specifies how to capture by reference or value, what is captured is determined by lambda exp.

Ex: default capture by reference

vector<int> v; d ... { }

int sum = 0;

for each ( begin(v), end(v) [8]  
( int n ) { sum = sum + n; } )

default capture by value

$\lambda = \lambda (int n) \{ sum = sum + n; \}$

Unit capture:

It avoids naming and initializing of capture.

Ex: explicit capture by reference.

(305)

int sum = 0

for\_each(begin(v), end(v), [&mySum =  
sum](int n) & mySum += n);

problem with C++11 capture:

one of the problem was that we could not  
actually copy more c. variable into a  
lambda.

~~Lambda does not add anymore expressive~~  
~~ness to the language.~~

~~it allows to name the internal param.~~  
~~members in lambda and then initialize it in~~  
~~free some space.~~

prob. when we are using lambda inside  
a class member fn. like

just one

↳ void fun (vector<int> v){}

↳ count\_if(begin(v), end(v),

[=](int n) & return !no manipulate;

)

int mult;

};

(306) If we are using above lamb def. outside  
of Object One, mult will forget be  
valid becoz we are not returning capturing  
mult base numbe but we are capturing  
capturing this->mult and this will be  
a dangerous ret. why if we return obj  
fn from fn and after one (Object) you  
disappear then this pointer will be  
dangerous.

How to resolve it  
create a local func and capture  
base number through it or go with  
initial capture.

so ① void func)

↳ int locnum = mult;

[locnum] (int) ↳ return n°  
locnum}); or

↳ or

(30)

count\_if(begin(v), end(v), [locam=mult])  
(int n) & return !(\*n, locam); };

or

[mult=mult] (int n) & return !(\*

n \* mult); };

since mult and mult are in two

diff scope

Note: Generic lambda:

~~Generic lambda~~ is a lambda which uses auto in its parameters list

for\_each (begin(v), end(v), [&sum=sum]  
(auto n) & sum = sum + n); };

How to pass ~~lambda~~ lambda to fn

~~by using template fn.~~

## Smart pointer

(308)

Why

- \* ~~Memory management.~~ (specifically Heap)
- \* Need to have me pointers with new and delete
- \* when we have pointer we don't know what kind of pointer is it.
  - (i) Is it a ~~danger~~ or referring to something

Ex.

myclass \*ptr;

ptr = new myclass();

ptr → do some work();

← what happens after  
delete p; exception occurs here. this

will lead to memory leak,  
may be more taken

or back on free memory.

Simple soln

myclass \*ptr;

try {

ptr = new myclass();

ptr → do some work();

delete ptr;

}

catch (...) {

} } delete ptr;

If circumstances

~~Simplest way~~ we need a mechanism where when ptr goes out of scope its pointed memory will be deleted automatically.

So create a templated wrapper over original pr. like.

template <class>

Smart pointer

class auto\_ptr

{

    T \*ptr;

public:

    implicit auto\_ptr(T \*p=0):

        ptr(p) {}

    ~auto\_ptr() { delete ptr; }

T & operator \*()

{ return \*ptr; }

}

T \* operator ->()

{ return ptr; }

}

## Use of above code

(310)

auto\_ptr<myclass> p1(new myclass())

p → destructor →

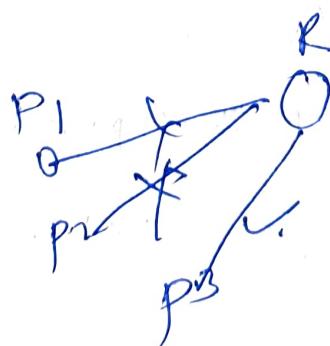
✓ If no exception occur p will go  
out of scope and its destructor will be  
called. So it will release the memory  
mem.

adv. ~~i) no need to delete or free these  
smart pointers~~

~~ii) no memory leak~~

Note: auto\_ptr based on exclusive model.  
i.e. we can assign it to other ptr.  
then the source itself give the ownership

→ two pointers of same type cannot point  
to the same resource.



→ copying or assigning by either way  
constructor or copy assignment changes  
ownership. one has to solve ownership

ex: class test

(31)

↳ public:

void print()

↳ const test \*K const;

};

int main()

↳ auto\_ptr<test> opt1(new test);

opt1 → print()

const opt1 → get() —  
return the address of owning pn.

Now auto\_ptr<test> opt2(opt1)

opt2 → print()

copy constr.  
move

opt2 will no longer be  
pointing to old return

NULL → opt1

→ 0x1432f7

Diadv copying and assigning makes  
no ownership. src → leaves ownership

ptr → get "

## Smart Pointers (312)

Defn : smart pointers are template class that uses operator overloading to provide the functionality of a pointer, while providing additional features to support improved memory mgmt. and safety.

\* wrapper around standard c pointer.

like T\* foo() returns a pointer.

what we are supposed to do with it  
once we are done with it?

i) should we free the mem.

ii) should we expect that some other fn  
will free the mem.

iii) Is it statically or dynamically  
allocated

iv) should we use delete or free?  
there is no way of knowing. and control  
to mem leak.

\* class wrapper over primitive, custom Data and framework

If we use smart pointers we don't have to worry  
about mis.

(313)

### Unique-Ptr

also contains raw pointer

✓ It prevents copying of its contained pointer

like

→ smart ptr  
(C++11)

`unique_ptr<A> ptr1(new A);`

`unique_ptr<A> ptr2 = ptr1; // error`

cannot copy unique

~~ptr1 is holding pointer~~

which is pointing to address of type A.

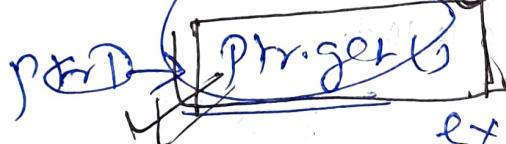
ptr

② unique\_ptr can be moved using the move scenario

`unique_ptr<A> ptr2 = move(ptr1);`

Now ptr1 will release ownership and

ptr1 will be null



objects get pointed

exclusive ownership

When should we use unique\_ptr?

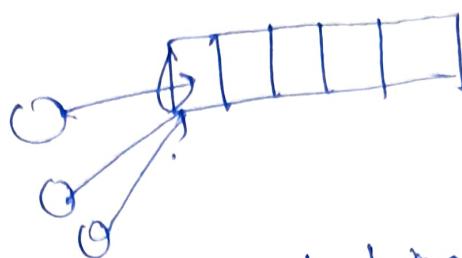
Open style ownership of resource is required.

only one unique\_ptr can point to one resource so one unique\_ptr cannot be copied to another

## Shared pointer

(314)

Based on reference count model

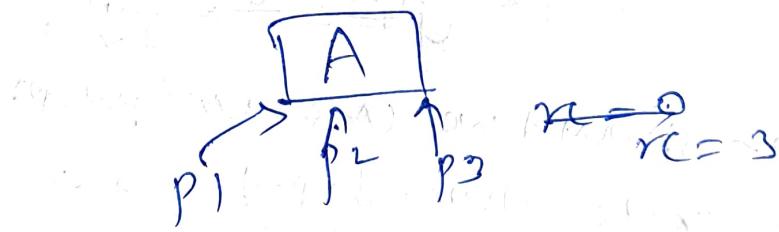


(Counter) incremented each time a new pointer points to the resource. decremented when destructor of object called

also contains of raw pointer.

Copy is allowed here.

~~Note: object referenced by the contained raw pointer will not be destroyed until ref-count > 0 i.e. until all copies of shared\_ptr have been deleted.~~



shared\_ptr<AD> p1(new A());

11      p2 = p1; p3(p1)

When to use shared\_ptr?

When sharing of resource is required.

multiple shared\_ptr can point to a single resource.

(3) ex. stored- $\text{ptr} \leftarrow A \rightarrow p_1(\text{new } A)$ ,

stored- $\text{ptr} \leftarrow A \rightarrow p_2(p_1)$ ;

stored- $\text{ptr} \leftarrow A \rightarrow p_3 = P^2$

~~p1.res()~~

$\text{p1.res()}$  will release ~~p1~~

$\text{ptr$  is inside  $p1$

~~p2.res()~~

Object to now.

$p_3$  ~~res()~~ // Now Object

will be deleted.

### problem with stored pointer

A cyclic reference problem.

Class A

A. stored- $\text{ptr} \leftarrow A \rightarrow$  adjacent

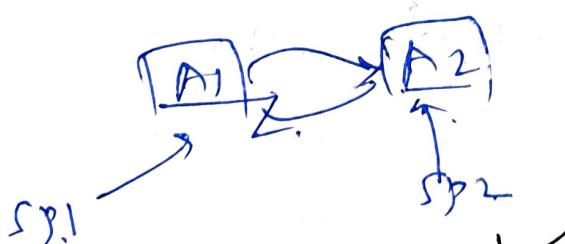
?'

$s_{p1}(\text{new } A)$

$s_{p2}(\text{new } A)$

$s_{p1} \rightarrow s_{p2} \text{Adj}(s_{p2})$

$s_{p2} \rightarrow s_{p1} \text{Adj}(s_{p1})$



because cyclic reference Object referred by stored- $\text{ptr}$  will never be deleted which leads to memory leak

## How to resolve tr1

(316)

Using weak\_ptr.

~~weak\_ptr~~: holds a non-owning ("weak") reference to an object that's managed by shared\_ptr



~~x~~ weak\_ptr is created as a copy of shared\_ptr.

~~x~~ provides access to an object that is owned by one or more shared\_ptr instances but does not participate in reference counting

How to use weak\_ptr

~~x~~ It is used to check the validity of underlying resource using expired()

~~weak\_ptr~~.expired()

It is used to check if the underlying resource is valid or not by checking reference count

~~weak\_ptr~~.expired() → return true if

refcount = 0 or else false.