(292)

Why to debug.

(i) **Misbehavior:**

20 packet, start showing 10 packets linkedlist.

20 packet comes start showing 10 packets.

(ii) **Memory leak:**

(iii) **Invalid / Illegal memory access.**

segmentation fault, it kernel the crash.

(a) ~~we~~ 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 it's read only But we are trying to write it

(c) allocated memory and free it and again trying to access it then it will give us segmentation fault.

(d) As an appln we should not touch the kernel mem., But some time we pass some api and ~~try~~ directly touch the kernel memory.

(iv) **Invalid instruction**

compile for processor X and then run on another processor Y

(v) **Synchronization issue among the threads or many cores**

global data between threads

Context

↓

w.r.t to the kernel

there may be diff context like process context, Interrupt context, Softirq context.

* we have stored 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 so that I can find out where the problem is

(I) I want to know the exact line of the code where this problem is happening

(II) I want to have the local variables data at that particular time when that segmentation fault has occurred.

(III) want to have global data when this segmentation fault has occurred

(IV) All other Dr I want to have. current

(V) I want to have the backtrace from where this particular fn has called.

(194) example

```
char *ptr = NULL
char *fn = "google.com"
```

we need to find the following

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

→ to find out this we need Instruction pointer (this contain 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 contain at time of

Heap segment

→ this register in CPU

can have info we need stack

* customer page.

* This all info will be present in core file

Coredile

(1)

footprint of process i.e. whole address of process

* which signal has generated in coredile, i.e. what information a coredump file contains

(1) process state:

→ signal info like signal no, error no

which signal has generated this core file.

→ set of pending signals

→ set of held signals (blocked signals)

→ proc id, proc parent id etc..

→ set of registers (sp or esp)

(2) process info:

→ state of the current process

→ state in string like R, S, Z...

→ whether the process is in Zombie state

→ Nice value of the process

→ process flags

→ pid, ppid

→ process name

→ command line argument of the prgm.

(3) Thread specific info:

→ proc status for the thread

thread

↓

current thread + other thread info.

(4) Info of memory mapped file

→ No of files mapped

→ Name of files mapped

→ start add

→ end add

(5) proc add space:

→ like data segment, stack segment, code segment and heap.

Who generate the core file and when will this be generated.

NULL pointer

CPU particular add (virtual add)

↓

map to phy add

↓

Not mapped

↓

page fault

↓

address valid or not

↓

Bad add (Now kernel will generate signal to proc (segmentation fault signal in our case)

297 Once it generates the segmentation fault signal to the process, it will try to create a file core file and embedd all 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 us to generate a core dump whenever coredump occurs. By default it is enabled

* each process has some resource limit

* kernel maintains list of resource for each process

* we have to tell the kernel to generate a core file size.

* For some of 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: relocatable obj contains section and executable object contains segments

(298) Debugging without symbol and with symbol

* When we dont have any di debugging symbol then we disassemble the code.

* For each frame stack frame will be allocated

* ABL: App Binary interface that explains how the parameter is passed to the fn.

(299) Lambda expression C++11/C++14

Lambda expn is a shorthand syntax for creating a function object.

auto is odd = [](int n){ return n%2; }

auto a = isOdd(3) // return bool.

declare, define and instantiate one of these function object in line in the code.

It actually does like

class isOdd

{ public:

bool operator()(int n)

{ return n%2;

}

};

or

count_if(begin(v), end(v), isOdd())

predicate

This creating a unnamed temp. of type isOdd. It can be replaced with lambda exp like

count_if(begin(v), end(v), [](int n){ return n%2; });

* Before calling () for any func we need to create an object of that class

Closure type: This is 200 generated type compiler generate for that lambda exp.

This compile time construct. The syntax like shell.

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

then compiler generate

class isOdd

{ bool operator(int n)

{ return n%2;

}

};

Closure: Actually the instance of closure type.

This object generated for a class.

This runtime construct. i.e instance shell.

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

auto fn = isOdd

Declaration, definition

tion:

* For calling lambda we need an object

* ClassName() or ClassName; will create temp

Objects

Lambda capture

↓

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

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

Ans:

ex:

```
auto mult=3
auto isMultipleOf = [] (int x){
                    return x%mult;};
```

How the lambda know about mult

So capture block is used

So why since lambda and multi is in diff scope. It will treat mult as its member variable in class generated by compiler like

```
class isMultOf
{   int m_mult;
    public:
        isMultOf(int mult): m_mult(mult)
        {}
        bool operator(int x)
        { return x%m_mult;}
};
```

how to assign value to local variable using lambda ?.

Explicit capture:

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

// explicit capture by reference.

```
vector<int>v.col{....};
int sum = 0;
for_each(begin(v), end(v)
        [&sum](int x) {sum = sum+x;});
```

ex-2

```
auto mult = 3
for_each(begin(v) end(v), [mult](int x)
    { return x * mult;});
```

This capture is by value so we have copy of mult in lambda exp.

If we want to change the captured value then we have to ~~use~~ by capture it was so by reference.

like:

```
auto sum = 0
auto ismultiple = [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
{ sum(int sum): m_sum(sum){}
  void operator()(int n) const
  { m_sum = m_sum + n; }
  int m_sum
};
```

this is generated

lambda

So we are not able to change m_sum

To resolve this we will capture by reference.

like

```
int sum=0
for_each( begin(v), end(v), [&sum](int n
    { 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{...};
int sum=0;
for_each( begin(v), end(v), [&]
    (int n){ sum = sum + n; });
```

default capture by value.

```
[=](int n){ return n%2==0; };
```

Init capture:

C++14

It allows naming and initializing of capture.

ex: explicit capture by reference.

```
int sum=0
for_each( begin(v), end(v), [&my_sum=
    sum](int x){ my_sum+=x;});
```

problem with c++11 capture:

One of the problem was that we could not actually capture a variable into a lambda.

lambda does not add anymore expressive power to the language.

allows us to name the internal data members in lambda and then initialize it in the same stmt.

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

```
struct one
{   void fun( vector<int> const& v)
    {   count_if( begin(v), end(v)
        [=](int x){ return ! no_multiple;}
        );
    }
    int mult;
};
```

(306) If we are using above lambda outside of Object One. result will be valid bcz we are not actually capturing result pointer but we are capturing this→result and this will be a dangling ref. why if we return any fn from fun and after one (Object) gets disappear then this pointer will be dangling.

How to resolve it

Create a local and capture result number through it or go with init capture

Solution ① void func()

{ int local = result;

[local] () { return local; }; or

}

or

count_if(begin(v), end(v), [localm=mul]
(int x){ return !(x%localm); });

or

[mul=mul] (int x){ return !(
x%mul); };
since mul and mul are in two
diff scope

Note: Generic lambda:
~~Integeneric~~ lambda is a lambda which
uses auto in its parameter list

for_each (begin(v), end(v), [&sum=sum]
(auto x){ sum=sum+x; });

How to pass ~~lambda~~ lambda to fn

~~By~~ by using template fn.

Smart pointer

Why

* Memory management. (specially Heap)
* Need to have the pointer with new and delete
* When we have pointer we don't know what kind of pointer it is.
  Eg: Is it a dangling or referring to something

Ex.

```
MyClass *ptr;
ptr = new MyClass;
ptr->doSomeWork();
delete p;
```

← what happens when exception occurs Here. This will lead to memory leak. may be we have taken a lock on that memory.

Simple soln

```
MyClass *ptr;
try {
    p = new MyClass;
    p->doSomeWork();
    delete p;
}
catch(...)
{
    delete p;
    throw;
}
```

// This can be done

~~Disadvantage~~ Soln we need a mechanism where when (ptr) goes out of scope its pointed memory will be deleted automatically.

Soln create a template class wrapper over original ptr. like.

smart pointer

```
template <class T>
class auto_ptr
{
    T *ptr;
public:
    explicit 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->doSomething();

No if exception occurs p will go out of scope and its destructor will be called. So it will release the pointed mem.

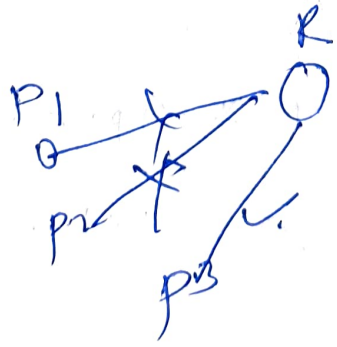
adv. (i) no need to delete or free these smart pointers

(ii) No memory leak.

Note: auto_ptr: based on exclusive model.

i.e when we assign it to other ptr then the source will give the ownership

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



* copying or assigning by either copy constructor or copy assigning changes ownership. one has to give ownership

ex: class test

{ public:

void print()

{ cout << "test" << endl;

}

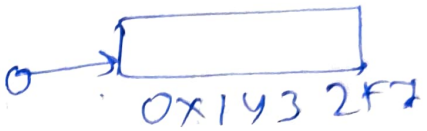
};

int main()

{ auto_ptr <test> aptr1 (new test);

aptr1 → print()

cout << aptr1 → get() —



returns address of containing ptr.

Now auto_ptr <test> aptr2 (aptr1)

aptr2 → print()

copy constructor.

Now

aptr1 will no longer be pointing to dictated memory

NULL → aptr1

0X1432F7

Disadv copying and assigning modifies ownership. src → leaves ownership

dst → get "

So basically

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 can lead to mem leak.

* class wrapper over primitive, custom Data and function

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

(313) Unique_ptr also container over raw pointer

It prevents copying of its contained pointer.

like

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

constructor (new)

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

cannot copy unique ptr

ptr1 is holding a pointer which is pointing to address of type A.

(II) unique_ptr can be moved using the move semantics

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

Now ptr1 will release ownership and ptr1 will be null

helps get pointed address

exclusive ownership

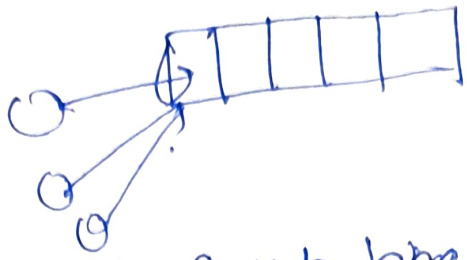
When should we use unique_ptr?

When single 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

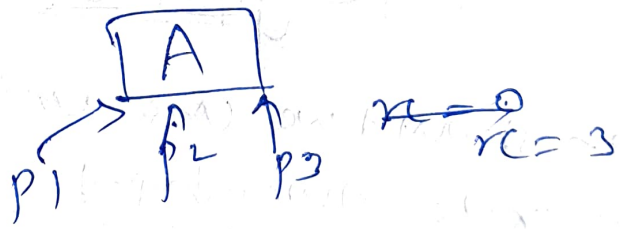


(counter) incremented each time a new pointer points to the resource. decremented when destructor of object called

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 <A> p1 (new A());

// p2 = p1; p3(p1)

When to use shared_ptr?

When sharing of resource is required.

multiple shared_ptr can point to a single resource.

(315) ex: stored_ptr<A> p1(new A());

stored_ptr<A> p2(p1);

stored_ptr<A> p3 = p2

p1.get()

p1.reset() will reset p1 ptr w.r.t inside p1 Object to null.

p2.reset()

p3.reset() // Now Object will be deleted.

problem with stored pointer

* cyclic reference problem.

class A

{ stored_ptr<A> adjacent

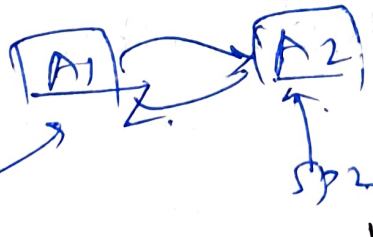
};

sp1(new A)

sp2(new A)

sp1->setAdj(sp2)

sp2->setAdj(sp1);

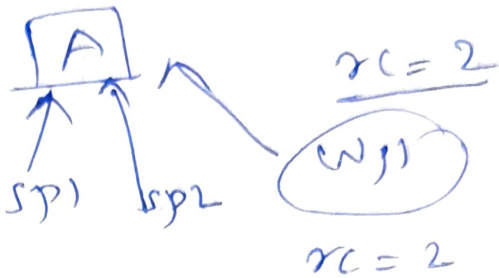


because cyclic reference Objects referred by stored_ptr will never be deleted which leads to memory leak

How to resolve this

Using weak_ptr.

weak_ptr: holds a non-owning ("weak") reference to an object that is managed by shared_ptr



* weak_ptr is created as a copy of shared_ptr

* 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

* It used to check the validity of underlying resource. using expired() fn

expired()

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

wp1.expired() → return true if ref count = 0 or else false.