Specifying and Checking Stateful Software Interfaces (Lecture 2)

Manuel Fähndrich maf@microsoft.com
Microsoft Research

2005 Summer School on Reliable Computing
Eugene, Oregon

Lecture 1 recap

- Goal: Specify and check stateful interfaces
- Techniques
  - Linear type systems
  - Type system based on capabilities (permissions)
- Modeling
  - allocation/deallocation
  - type state protocols
  - locking

Lecture 2

- Frame axiom
- Type-states using capabilities
- Vault: W2K driver case study
- Recursive data structures
- Unifying non-linear data structures and linear data

Lambda abstraction

$$\tau ::= \ldots \mid \forall[\Delta].(C;\sigma) \to (\sigma;C)$$

- We can abstract allocation sequence

$$\mathsf{init} : \forall[\rho].(\underbrace{\{\rho \mapsto \langle s(0), s(0)\rangle\}}_{\text{pre-heap}}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \underbrace{\{\rho \mapsto \mathsf{t}\}}_{\text{post-heap}})$$

Recall examples

- Function taking a list argument (but not consuming it!)

$\mathsf{length} : \forall[\rho].(C_{\mathsf{List}}\langle\rho\rangle, \mathsf{pt}(\rho)) \rightarrow (\mathsf{int}, C_{\mathsf{List}}\langle\rho\rangle)$

- Function freeing entire list

$\mathsf{freeAll} : \forall[\rho].(C_{\mathsf{List}}\langle\rho\rangle, \mathsf{pt}(\rho)) \rightarrow (\mathsf{unit}, \cdot)$

- Application rule

$$\frac{\begin{array}{l} A; C_1 \vdash e_1 : (C_3, \sigma_1) \rightarrow (\sigma_2, C_4); C_2 \\ A; C_2 \vdash e_2 : \sigma_1; C_3 \end{array}}{A, C_1 \vdash e_1\ e_2 : \sigma_2;\ C_4}\,[\mathsf{app}]$$

The frame rule

$$\frac{\begin{array}{c} C_1 = C_r \otimes C_2 \\ A; C_2 \vdash e : \sigma; C_3 \end{array}}{A, C_1 \vdash e : \sigma;\ C_r \otimes C_3}[\text{frame}]$$

- Example

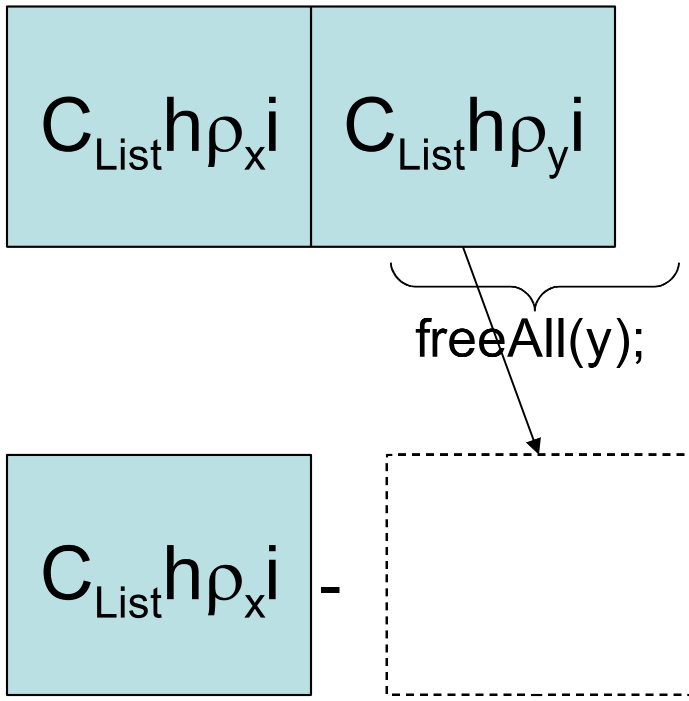
  x : pt(ρ_x), y : pt(ρ_y)

  freeAll(y);

  int z = length(x);

  freeAll(x);

- Modifications?



Specification tasks

- Allocation/Deallocation ⋗
- Memory initialization ⋗
- Locks ⋗
- Events
- Type states ⋗
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom

Let’s look again at type-states.

Type-states with capabilities

- Still one type per type-state

$$\begin{tikzcd}
 & & R \arrow[loop above] \arrow[dd, "promote"] \arrow[dr, "close"] & & \\
{} \arrow[r, "alloc"] & A \arrow[ur, "openR"] \arrow[dr, "openW"'] & & C \arrow[r, "free"] & {} \\
 & & W \arrow[loop below] \arrow[ur, "close"'] & &
\end{tikzcd}$$

$$\mathsf{openR} : \forall[\rho].(\{\rho \mapsto \mathsf{AFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{RFile}\})$$

$$\mathsf{openW} : \forall[\rho].(\{\rho \mapsto \mathsf{AFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{WFile}\})$$

$$\mathsf{promote} : \forall[\rho].(\{\rho \mapsto \mathsf{RFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{WFile}\})$$

$$\mathsf{close} : \forall[\rho].(\{\rho \mapsto \mathsf{RFile} \cup \mathsf{WFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{CFile}\})$$

$$\mathsf{free} : \forall[\rho].(\{\rho \mapsto \mathsf{CFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \cdot)$$

Observation about type states

- A type state is just a type!
- Type = Predicate over values and heap fragments
- A physical block of memory can have different types, thus different states/properties at different times.

Heavy notation?

- Vault programming language
  - Try to make capabilities available to programmers
  - Type-states as family of some base type
    File@A, File@R, File@W, File@C

```
void openR( tracked(ρ) File file ) [ ρ@A ! R ];
```

$$\mathsf{openR} : \forall[\rho].(\{\rho \mapsto \mathsf{AFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{RFile}\})$$

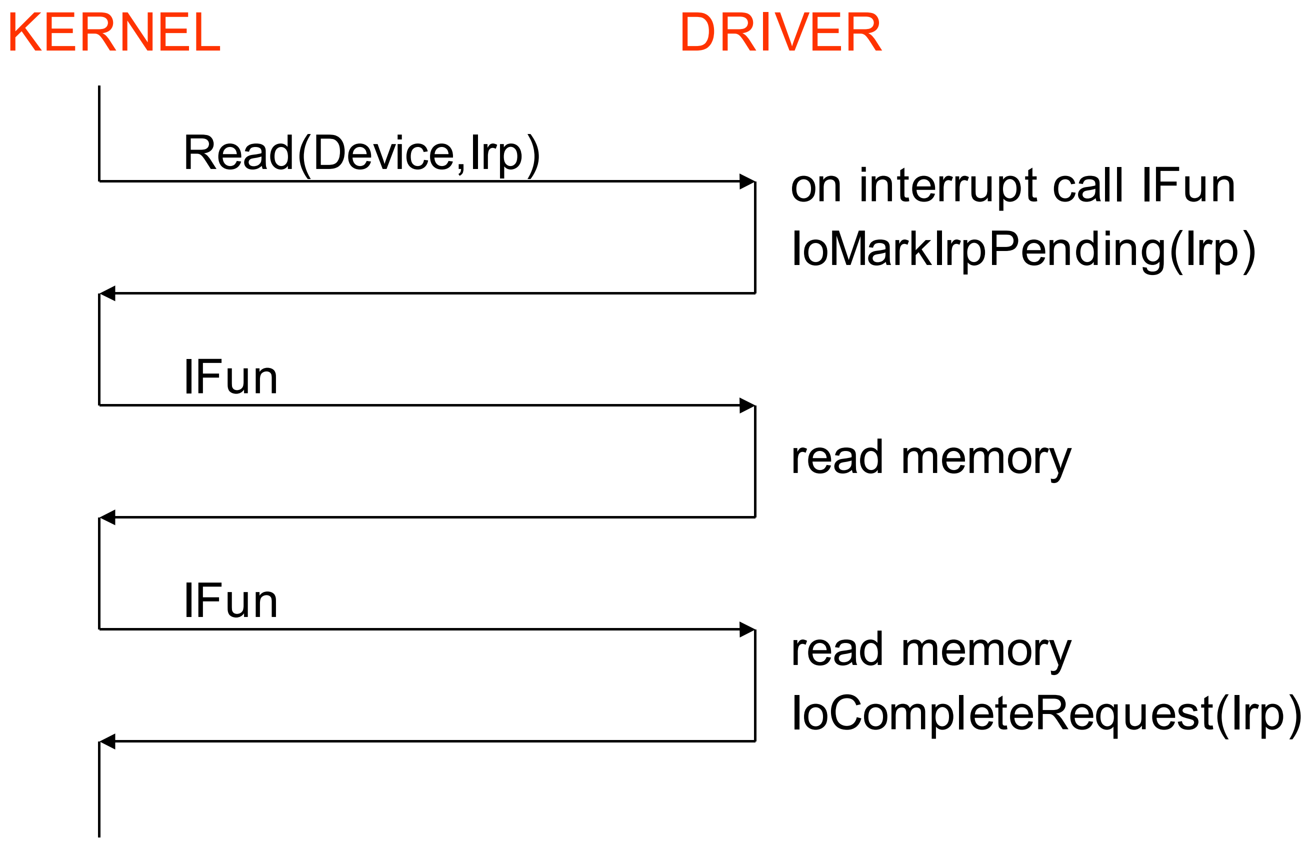
```
void closeR( tracked(ρ) File file ) [ --ρ@A ];
```

$$\mathsf{closeR} : \forall[\rho].(\{\rho \mapsto \mathsf{RFile}\}, \mathsf{pt}(\rho)) \to (\mathsf{unit}, \{\rho \mapsto \mathsf{CFile}\})$$

Case Study: Windows Drivers

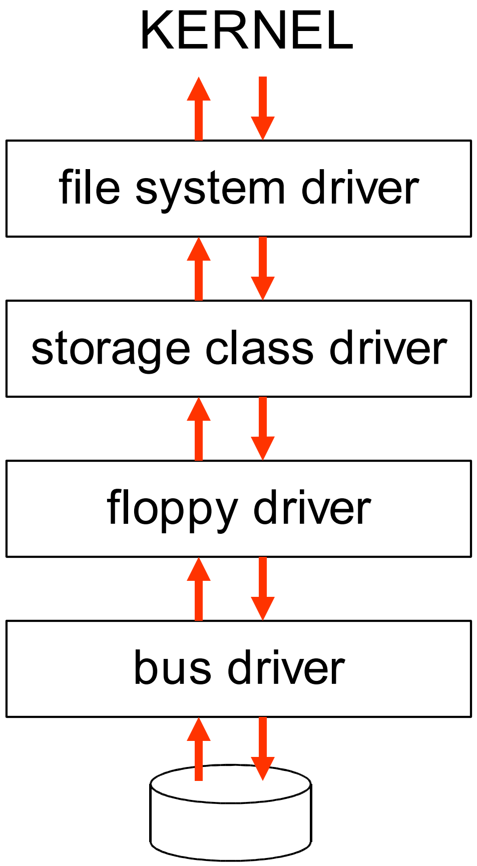
- Driver handles requests from the kernel
  - e.g. start, read, write, shutdown, ...
  - driver exports a function for each request type
  - lifetime of request ≠ lifetime of function call
- Request is encapsulated in a data structure
  - I/O Request Packet (IRP)
  - Driver handles request by side-effecting IRP
  - IRP ownership and lifetime are important

Request often lives across calls

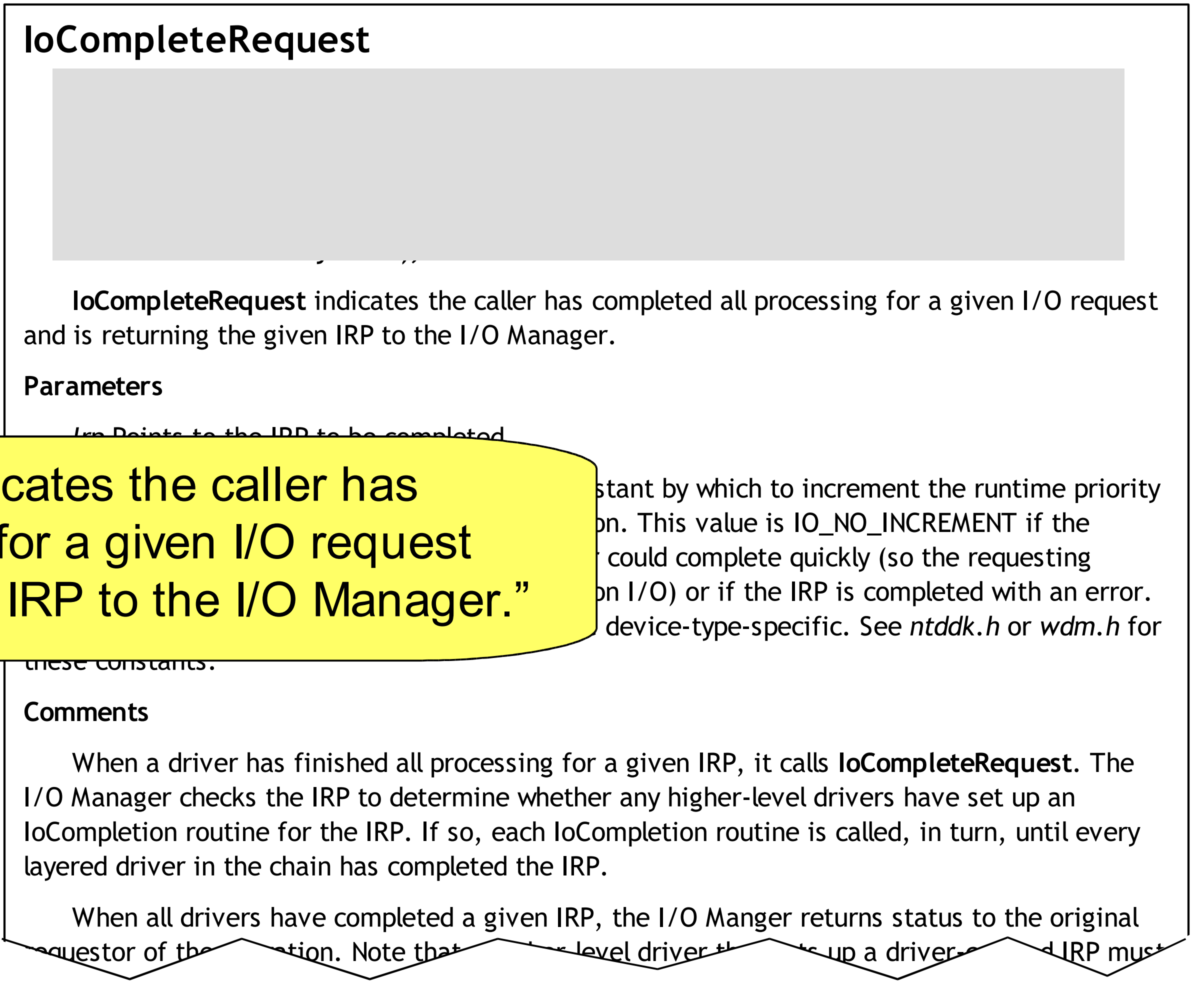


Drivers form a stack

- Kernel sends IRP to top driver in stack
- Driver may...
  - handle IRP itself
  - pass IRP down
  - pass new IRP(s) down



IRP Ownership



IoCompleteRequest

IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager.

Parameters

Irp Points to the IRP to be completed.

stant by which to increment the runtime priority
on. This value is IO_NO_INCREMENT if the
could complete quickly (so the requesting
on I/O) or if the IRP is completed with an error.
device-type-specific. See *ntddk.h* or *wdm.h* for
these constants.

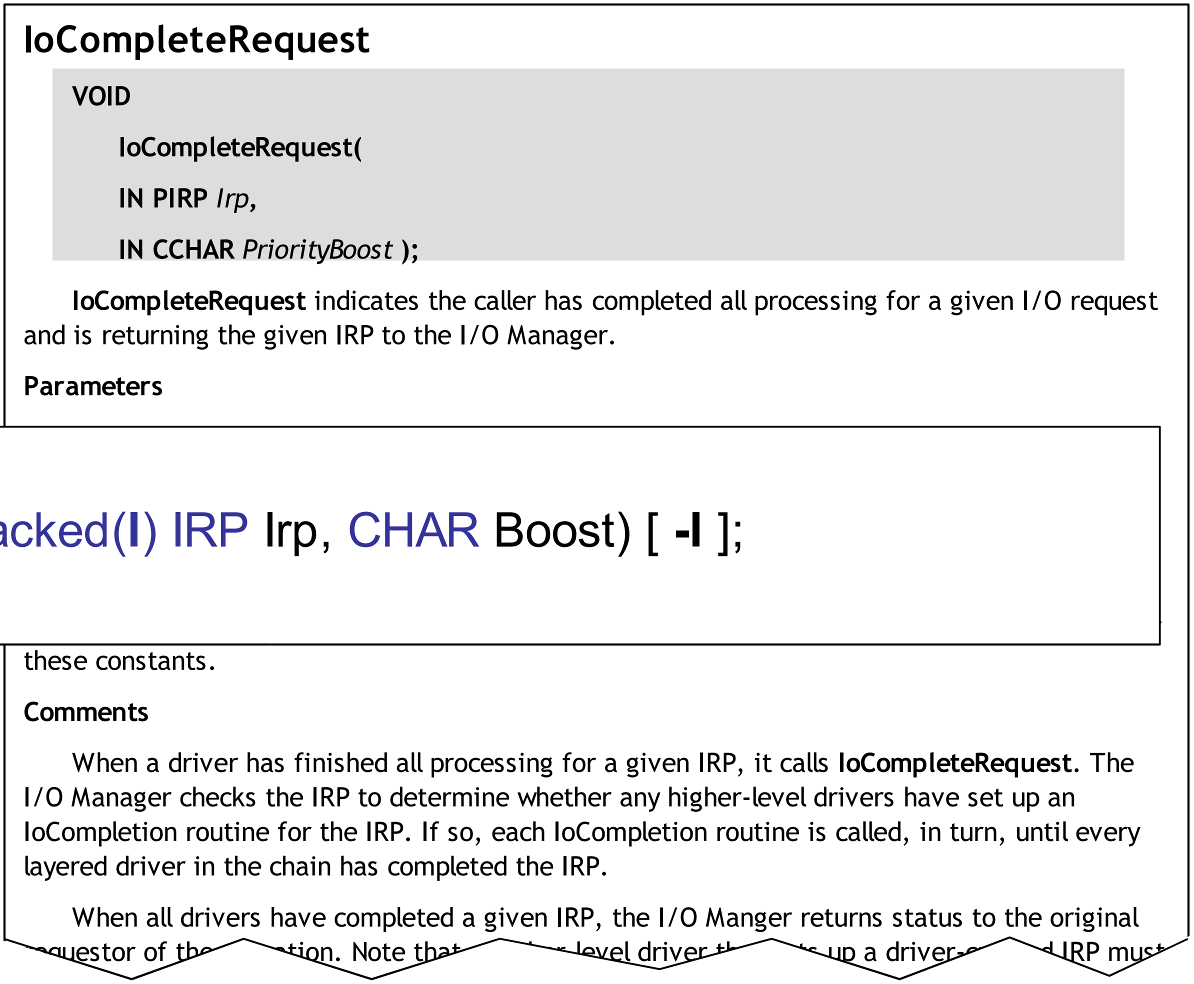
Comments

When a driver has finished all processing for a given IRP, it calls **IoCompleteRequest**. The I/O Manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine for the IRP. If so, each IoCompletion routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed a given IRP, the I/O Manger returns status to the original requestor of the operation. Note that

"**IoCompleteRequest** indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager."

IRP Ownership



IoCompleteRequest

```
VOID
    IoCompleteRequest(
    IN PIRP Irp,
    IN CCHAR PriorityBoost );
```

IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager.

Parameters

these constants.

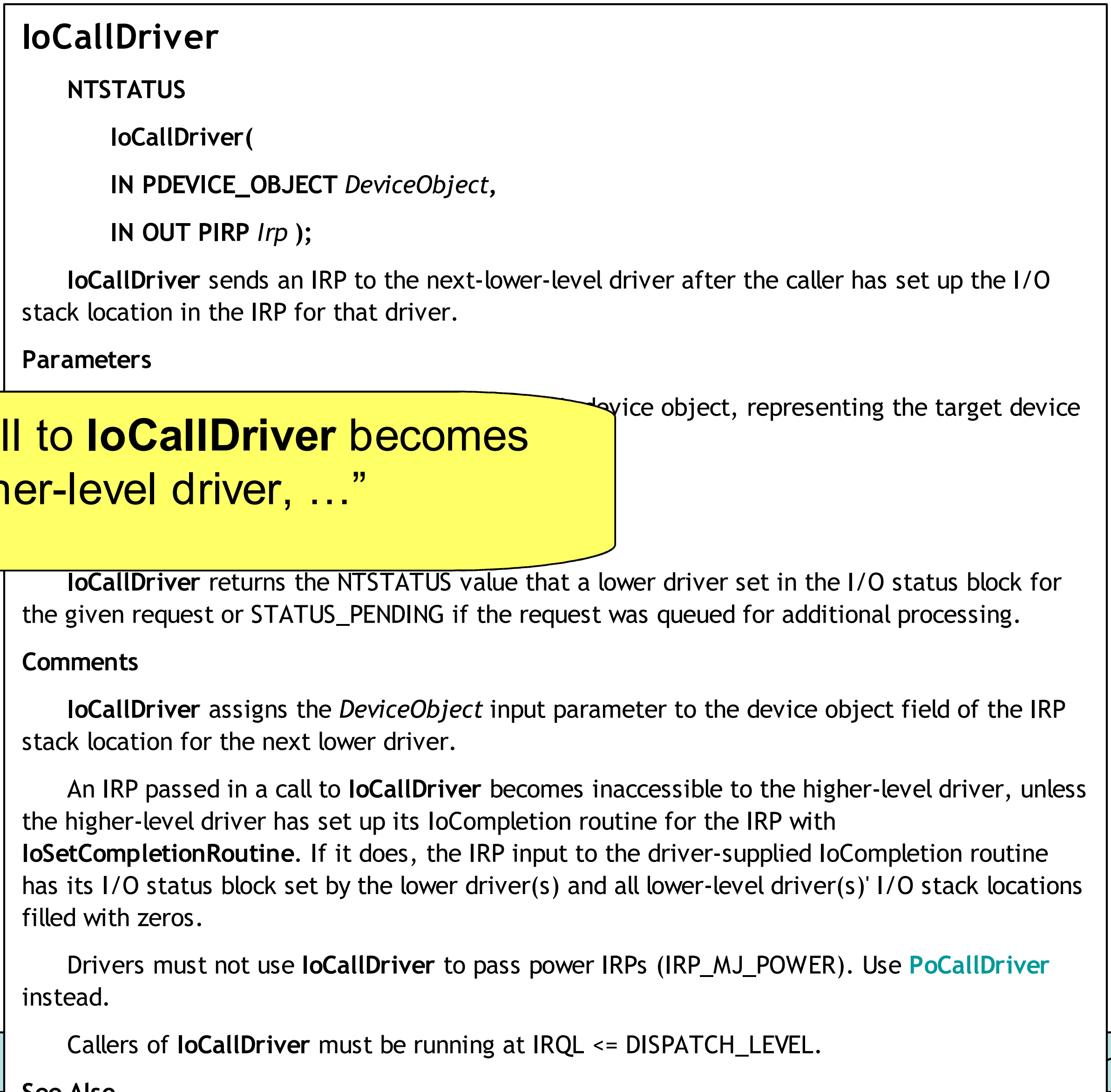
Comments

When a driver has finished all processing for a given IRP, it calls **IoCompleteRequest**. The I/O Manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine for the IRP. If so, each IoCompletion routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed a given IRP, the I/O Manger returns status to the original requestor of th ation. Note that -level driver t up a driver- IRP must

```
void IoCompleteRequest( tracked(I) IRP Irp, CHAR Boost) [ -I ];
```

IRP Ownership



IoCallDriver

NTSTATUS

IoCallDriver(

IN PDEVICE_OBJECT *DeviceObject*,

IN OUT PIRP *Irp* **);**

IoCallDriver sends an IRP to the next-lower-level driver after the caller has set up the I/O stack location in the IRP for that driver.

Parameters

evice object, representing the target device

IoCallDriver returns the NTSTATUS value that a lower driver set in the I/O status block for the given request or STATUS_PENDING if the request was queued for additional processing.

Comments

IoCallDriver assigns the *DeviceObject* input parameter to the device object field of the IRP stack location for the next lower driver.

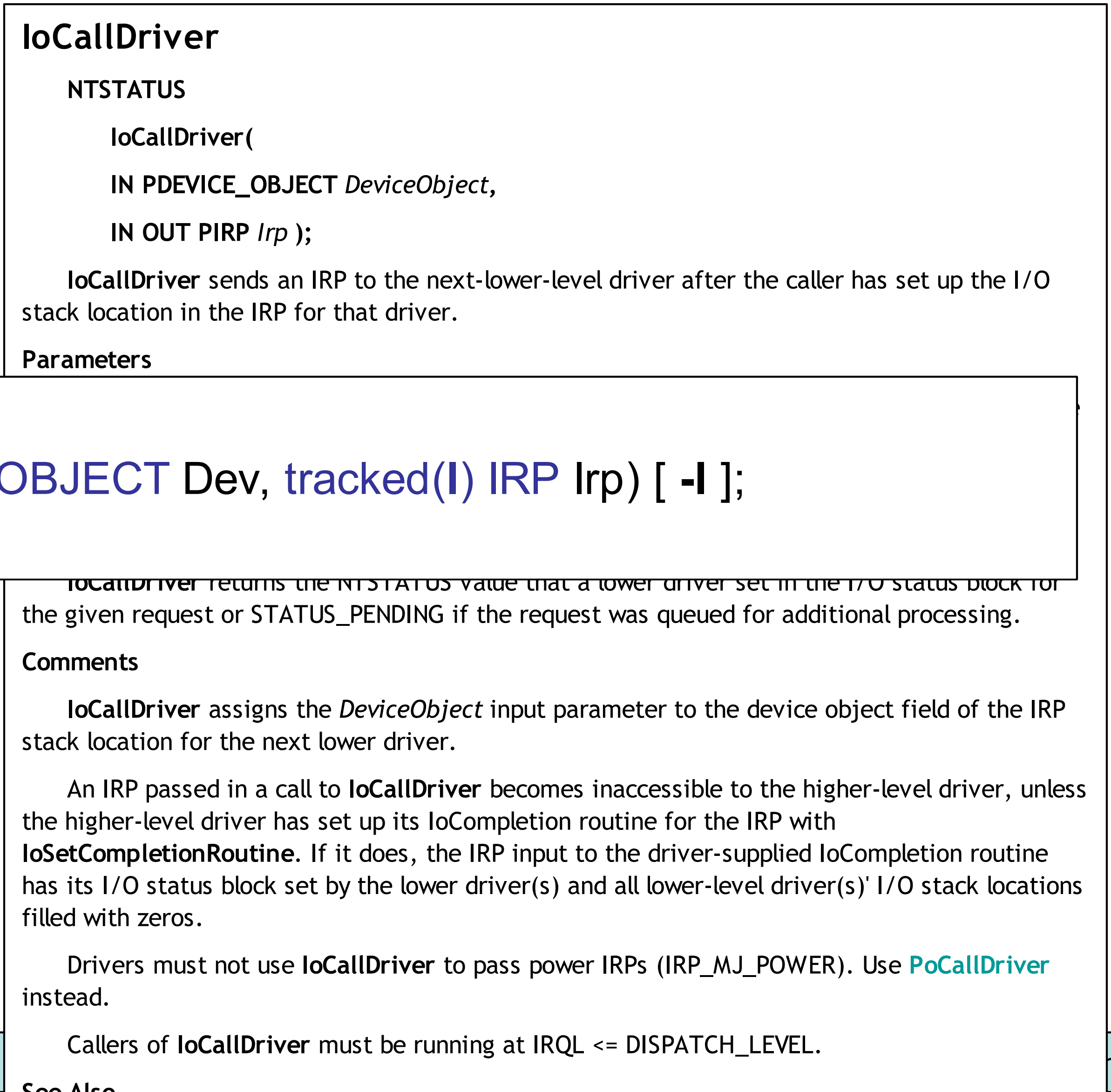
An IRP passed in a call to **IoCallDriver** becomes inaccessible to the higher-level driver, unless the higher-level driver has set up its IoCompletion routine for the IRP with **IoSetCompletionRoutine**. If it does, the IRP input to the driver-supplied IoCompletion routine has its I/O status block set by the lower driver(s) and all lower-level driver(s)' I/O stack locations filled with zeros.

Drivers must not use **IoCallDriver** to pass power IRPs (IRP_MJ_POWER). Use **PoCallDriver** instead.

Callers of **IoCallDriver** must be running at IRQL <= DISPATCH_LEVEL.

“An IRP passed in a call to **IoCallDriver** becomes inaccessible to the higher-level driver, …”

IRP Ownership



IoCallDriver

NTSTATUS

IoCallDriver(

IN PDEVICE_OBJECT *DeviceObject*,

IN OUT PIRP *Irp* **);**

IoCallDriver sends an IRP to the next-lower-level driver after the caller has set up the I/O stack location in the IRP for that driver.

Parameters

IoCallDriver returns the NTSTATUS value that a lower driver set in the I/O status block for the given request or STATUS_PENDING if the request was queued for additional processing.

Comments

IoCallDriver assigns the *DeviceObject* input parameter to the device object field of the IRP stack location for the next lower driver.

An IRP passed in a call to **IoCallDriver** becomes inaccessible to the higher-level driver, unless the higher-level driver has set up its IoCompletion routine for the IRP with **IoSetCompletionRoutine**. If it does, the IRP input to the driver-supplied IoCompletion routine has its I/O status block set by the lower driver(s) and all lower-level driver(s)' I/O stack locations filled with zeros.

Drivers must not use **IoCallDriver** to pass power IRPs (IRP_MJ_POWER). Use PoCallDriver instead.

Callers of **IoCallDriver** must be running at IRQL <= DISPATCH_LEVEL.

```
void IoCallDriver(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [ -I ];
```

Example: Driver request

```
NTSTATUS Read(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [ -I ] {
    if (GetRequestLength(Irp) == 0) {
        NTSTATUS status = `STATUS_SUCCESS(`TransferBytes(0));
        IoCompleteRequest(Irp, status);
        return status;
    } else
        return IoCallDriver(NextDriver,Irp);
}
```

Example: Driver request

```
NTSTATUS Read(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [ -I ] {      { I }
    if (GetRequestLength(Irp) == 0) {                              { I }
        NTSTATUS status = `STATUS_SUCCESS(`TransferBytes(0));      { I }
        IoCompleteRequest(Irp, status);                            {}
        return status;                                             {}
    } else                                                         { I }
        return IoCallDriver(NextDriver,Irp);                       {}
}
```

IRP completion routines

- Getting IRP ownership back
  - driver A hands IRP to B and wants it back after B is done
  - driver A sets “completion routine” on IRP

```
void IoSetCompletionRoutine(tracked(K) IRP Irp,
                            COMPLETION_ROUTINE<K> Fun) [K];

type COMPLETION_ROUTINE<key K> =
    tracked COMPLETION_RESULT<K>(DEVICE_OBJECT Dev,
                                 tracked(K) IRP Irp) [-K];

tracked variant COMPLETION_RESULT<key K> [
        | `MoreProcessingRequired
        | `Finished(NTSTATUS) {K} ];
```

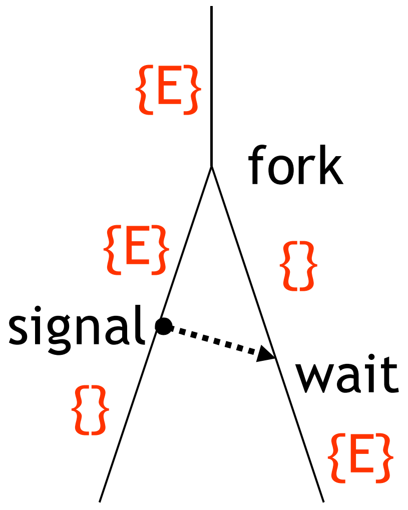
Events

```
type KEVENT<key R>;

KEVENT<E> KeInitializeEvent<type T> (tracked(E) T Obj) [ E ];

NTSTATUS KeSignalEvent(KEVENT<E> Event) [ -E ];

NTSTATUS KeWaitForEvent(KEVENT<E> Event) [ +E ];
```



Completion routine example

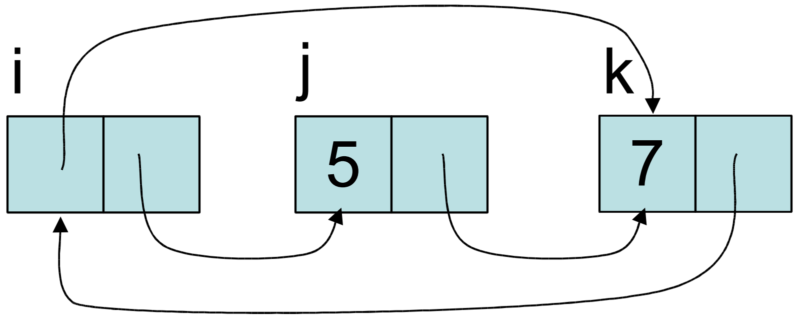
```
NTSTATUS PlugPlay(DEVICE_OBJECT Dev, tracked(R) IRP Irp) [-R] {          {R}
    KEVENT<R> DoneEvent = KeInitializeEvent(Irp);                        {R}

    tracked COMPLETION_RESULT<I>
     CompletePnP(DEVICE_OBJECT Dev, tracked(I) IRP Irp) [-I] {           {I=R}
       KeSignalEvent(DoneEvent);                                         {}
       return `MoreProcessingRequired;                                   {}
    }
                                                                         {R}
    IoSetCompletionRoutine(Irp, CompletePnP<R>);                         {R}
    CALL_RESULT<R> result = IoCallDriver(lowerDriver, Irp);              {}
    KeWaitForEvent(DoneEvent);                                           {R}
    ...
}
```

Specification tasks

- Allocation/Deallocation ⋗
- Memory initialization ⋗
- Locks ⋗
- Events ⋗
- Type states ⋗
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom
- Non-tree data structures?

Non-tree data structures?



$$\{i \mapsto \langle \mathsf{pt}(k), \mathsf{pt}(j)\rangle\}\otimes$$
$$\{j \mapsto \langle s(5), \mathsf{pt}(k)\rangle\}\otimes$$
$$\{k \mapsto \langle s(7), \mathsf{pt}(i)\rangle\}\otimes$$

- arbitrary finite graphs and
- a form of regular recursive graphs via existential abstraction over pointer names
  and heap fragments

Recursive data structures

- Consider a linear list

  List^{2} = Nil j Cons of int * List^{2}

  “Each Cons cell *owns* the rest of the list”

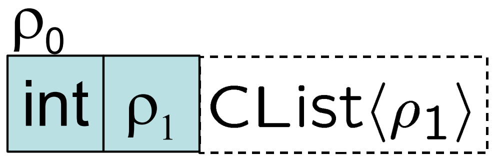
- Using capabilities:
  - Use pt(0) for Nil
  - Package a heap fragment with non-zero pointer
  - Abstract over the pointer value

  List^{2} , 9[ρ j C_{List}h ρi].pt(ρ)

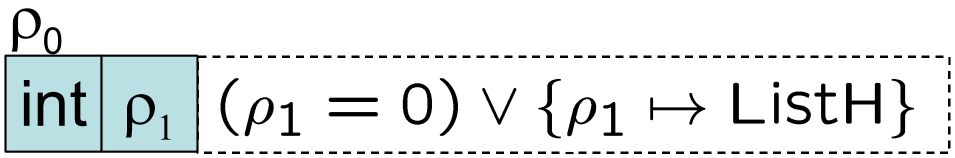
  C_{List}hρi = (ρ=0) Ç {ρ $\mapsto$ ListH}

  ListH = 9[ρ j C_{List}hρi].h int, pt(ρ) i

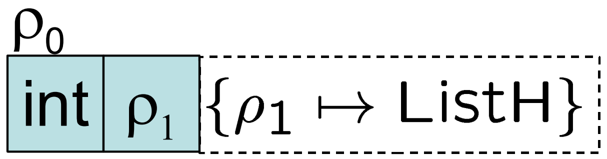
Linear list unpacking and packing



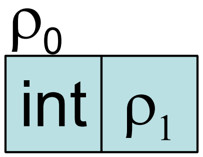
Linear list unpacking and packing



Linear list unpacking and packing

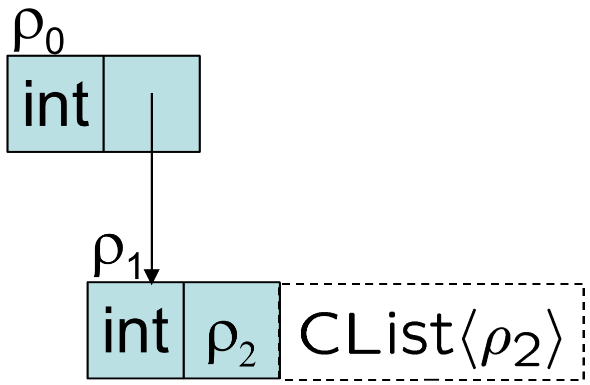


Linear list unpacking and packing

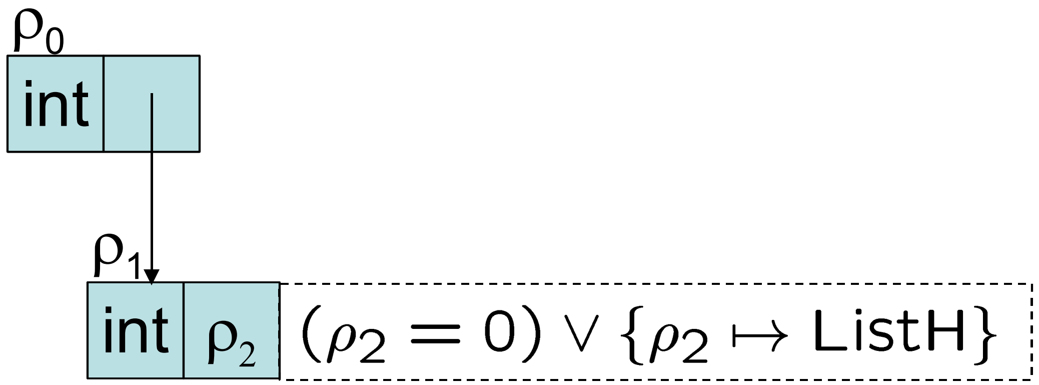


$$\{\rho_1 \mapsto \exists[\rho_2 \mid \mathsf{CList}\langle\rho_2\rangle].\langle \mathsf{int}, \mathsf{pt}(\rho_2)\rangle\}$$

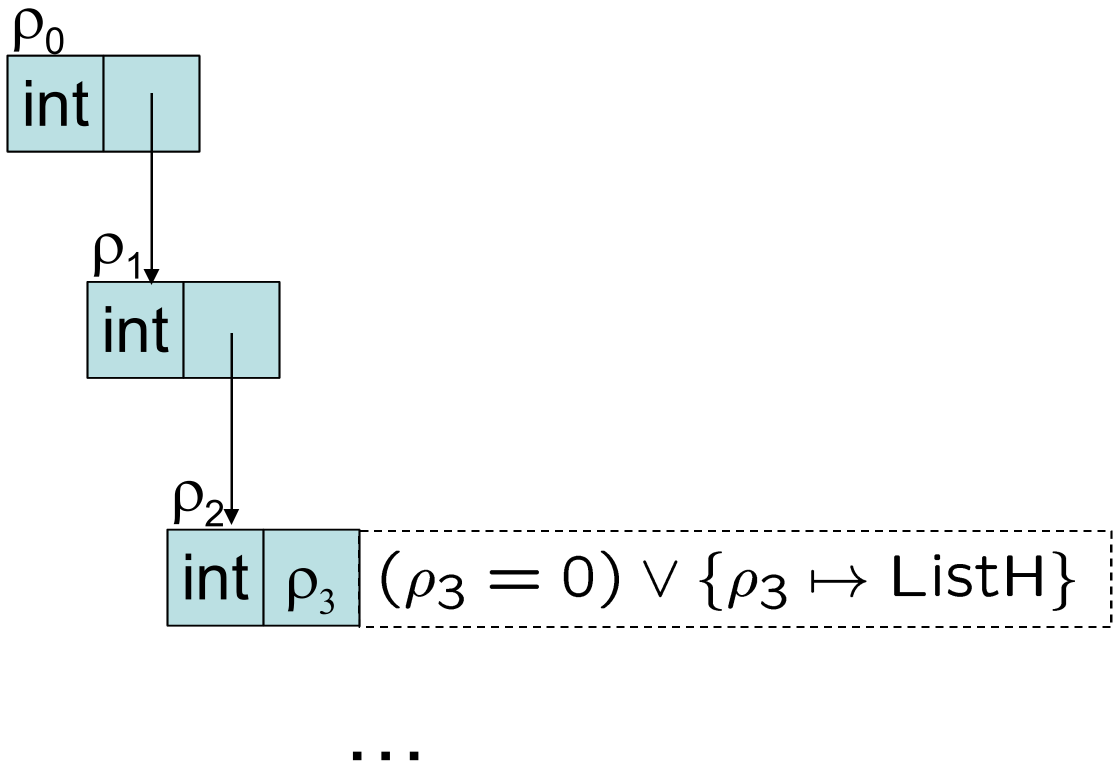
Linear list unpacking and packing



Linear list unpacking and packing



Linear list unpacking and packing



Packing and Unpacking

$$\frac{A; C_1 \vdash e : \exists[\Delta \mid C].\sigma; C_2}{A; C_1 \vdash \mathsf{unpack}\, e : \sigma; C \otimes C_2}[\mathsf{unpack}]$$

$$\frac{\begin{array}{l} A; C_1 \vdash e : S(\sigma);\ S(C) \otimes C_2 \\ S = [\Delta'/\Delta] \end{array}}{A; C_1 \vdash \mathsf{pack}[\Delta \mid C].e : \exists[\Delta \mid C].\sigma; C_2}[\mathsf{pack}]$$

Summary of Capability Type Systems

- Capabilities are single-threaded in type system (heap is single-threaded in dynamic semantics)
  - Linear treatment of capabilities
- Splitting and joining of heap fragments
- Relaxed single pointer requirement
  - Single heap fragment invariant
- Natural imperative programming style
  - Can use pointers as often as we like
    - as long as we can prove suitable capability is present
  - Explicit treatment of dangling pointers

Programming languages

- Based on capabilities or similar concepts

Vault	*resource management and type states*
Fugue	*object type states*
Sing#	*resource management and channels*
Cyclone	*safe C replacement with regions*
Clay	*low-level memory management (GC)*
ATS	*low-level memory management*
…	

PL Characteristics

- Dichotomy between precisely tracked data and non-linear data   (exception Clay)
- Surface specification language vs. internal specification language
  - Has to be concise, otherwise it's a calculus
  - Difficult to find good trade-off between expressiveness and conciseness
- How much is inferred, how much is explicit?
  - Coercions, Instantiations, Proof terms

Arbitrary data structures?

- Arbitrary graphs are difficult to express, but not impossible
  - O'Hearn et.al. have done specifications and hand proofs of complicated graph algorithms
    - graph copying and freeing
  - But automated systems with such expressive power are still under development
    - Clay (Hawblitzel et. al) and ATS (Xi et.al.) come close.
  - Different domains require different expressiveness
    - Specifying and checking copying GC
    - Application program dealing with sockets and files

Non-linear data structures

- Mere mortals need way to express data structures with less detailed capability specifications
  - Who owns the observer in the view-observer pattern?
  - Who owns call-back closures on GUI elements?
    - Where is the permission?
    - How is it threaded to place of use?
- Require some way to abstract over individual permissions
- Necessary evil

Specifications

- Allocation/Deallocation ⋗
- Memory initialization ⋗
- Locks ⋗
- Events ⋗
- Type states ⋗
- Object states
- Regions
- Reference counting
- Sharing
- Channels
- Deadlock freedom
- Use $\neq$ Consume ⋗
- Non-tree data structures ⋗

Regions

- Rather than handling individual capabilities for individual objects, need a mechanism to abstract over the capabilities for a set of objects.
- Well-known abstraction: Regions
  - A region is a named subset of the heap
  - Objects are individually allocated from a region
  - A region is deallocated as a whole
    - Common lifetime for all objects within region
  - ρBT denotes an object of type T in region ρ

Regions

- A region has type pt(ρ), where {ρ↦ Region}
- An object in a region has type ρBT
- Can define specialized type rules

$$\frac{\begin{array}{c} A; C_0 \vdash e_0 : \mathsf{pt}(\rho); C_1 \\ A; C_i \vdash e_i : \tau_i;\ C_{i+1} \\ C_{n+1} = \{\rho \mapsto \mathsf{Region}\} \otimes \_ \end{array}}{A; C_0 \vdash alloc(e_0)\langle e_1..e_n\rangle : \rho \triangleright \langle \tau_1..\tau_k\rangle; C_{n+1}}\,[\mathsf{alloc\text{-}r}]$$

$$\frac{\begin{array}{c} A; C_1 \vdash e : \rho \triangleright \langle \tau_1..\tau_n\rangle; C_2 \\ C_2 = \{\rho \mapsto \mathsf{Region}\} \otimes C_3 \end{array}}{A; C_1 \vdash e.k : \tau_k; C_2}\,[\mathsf{read\text{-}r}]$$

Region example in Vault

```
void main() {                                          {}
    tracked(R) region reg = Region.create();           {R}
    R:point pt = new(reg) point {x=4, y=2};            {R}
    int y;                                             {R}
    if (pt.x > 0) {                                    {R}
        Region.delete(reg);                            {}
        y = 0;                                         {}
    } else {                                           { R }
        y = pt.x;                                      { R }
        Region.delete(reg);                            {}
    }                                                  {}   post condition!
}
```

Bug 1: Dangling reference

```
void main() {                                         {}
    tracked(R) region reg = Region.create();          {R}
    R:point pt = new(reg) point {x=4, y=2};           {R}
    int y;                                            {R}
    if (pt.x > 0) {                                   {R}
        Region.delete(reg);                           {}
        y = 0;                                        {}
    } else {                                          {R}
        Region.delete(reg);                           {}
        y = pt.x;                                     bug! R ∉ {}
    }                                                 {}
}
```

Bug 2: Memory leak

```
void main() {                                              {}
    tracked(R) region reg = Region.create();               {R}
    R:point pt = new(reg) point {x=4, y=2};                {R}
    int y;                                                 {R}
    if (pt.x > 0) {                                        {R}
        y = 0;                                             {R}
    } else {                                               {R}
        y = pt.x;                                          {R}
    }                                                      {R}
}                                                          {R} bug! leaking key R
```

Discussion of Regions

- Different objects in region can have the same type

  R:point x; R:point y; …

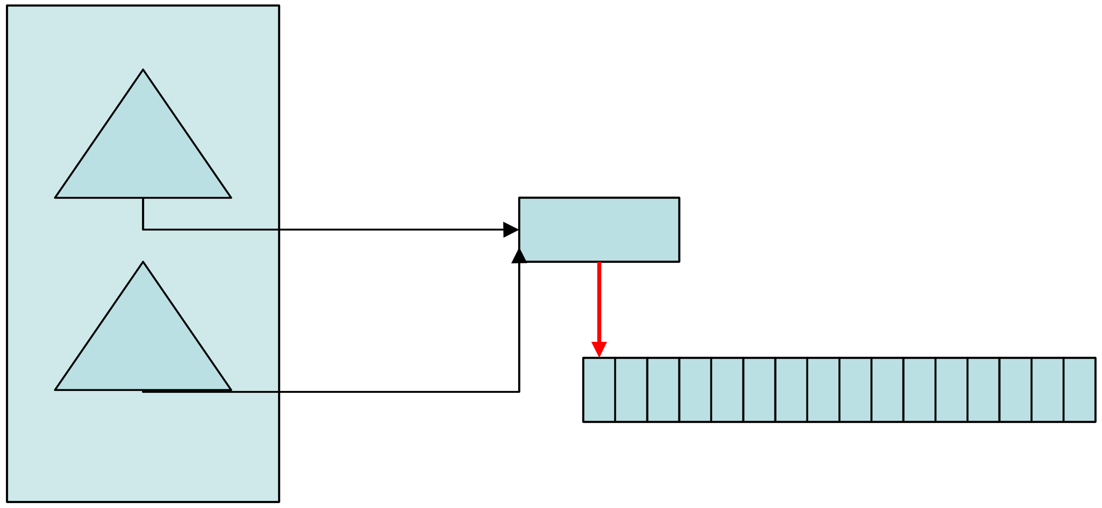
- Non-region pointers and pointers into regions have distinct types
  - pt(ρ) with $\{\rho \mapsto T\}$ vs. ρBT
- Decision for what kind of object is used is done at allocation, and fixed throughout
  - Can't do incremental initialization e.g.
- Component restriction of linear types:
  - can't have linear components in region types

Motivating example

Dictionary example:

- map keys to resizable arrays
- sharing of cells suggests cells and contents in a region

  ρBrefh ρBint[] i
- But, resize can't free old array
- ref<int[]> ?



Generalizing the region idea

- Goal: Uniform object model
  - Birth and death as linear objects
  - Switch from linear to non-linear and back
  - Switch from non-linear to linear and back
- Any resource can serve as a region (a lifetime delimiter)
- Call such a resource an adopter ρ_a
- For adoptee:
  - use type pt(ρ_1)
  - non-linear predicate (*adoption fact*): $\{\rho_a \mapsto T_a\}$ B $\rho_1{:}T_1$
    - “given cap $\{\rho_a \mapsto T_a\}$, can deduce ρ_1 is a pointer to T_1”
    - delegates permissions

[illegible]

Adoption (Freezing)

- Explicit act to introduce adoption fact

$\{\rho_0: \tau_0\}$ B $\rho_1:\tau_1$ from $\{\rho_1 \mapsto \tau_1\}$

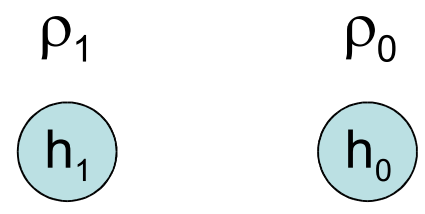
$$\frac{\begin{array}{l} A; C_i \vdash e_i : \mathsf{pt}(\rho_i);\ C_{i+1} \qquad\qquad i = 0, 1 \\ C_2 = \{\rho_0 \mapsto h_0\} \otimes \{\rho_1 \mapsto h_1\} \otimes C_3 \\ C_4 = \{\rho_0 \mapsto h_0\} \otimes C_3 \otimes \{\rho_0 \mapsto h_0\} \triangleright \rho_1 : h_1 \end{array}}{A; C_0 \vdash \mathsf{adopt}\, e_1\, \mathsf{by}\, e_0 : \mathsf{pt}(\rho_1);\ C_4}\,[\mathsf{adopt}]$$

- Abbreviation
  - CBT , 9[ρ j C B ρ : T].pt(ρ)
- Linear components in non-linear objects
  - T abitrary
  - But, cannot access linear T's via non-linear permission

Adoption graphically

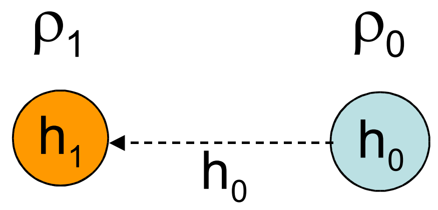
adopt e_1 by e_0

Capability



Before:

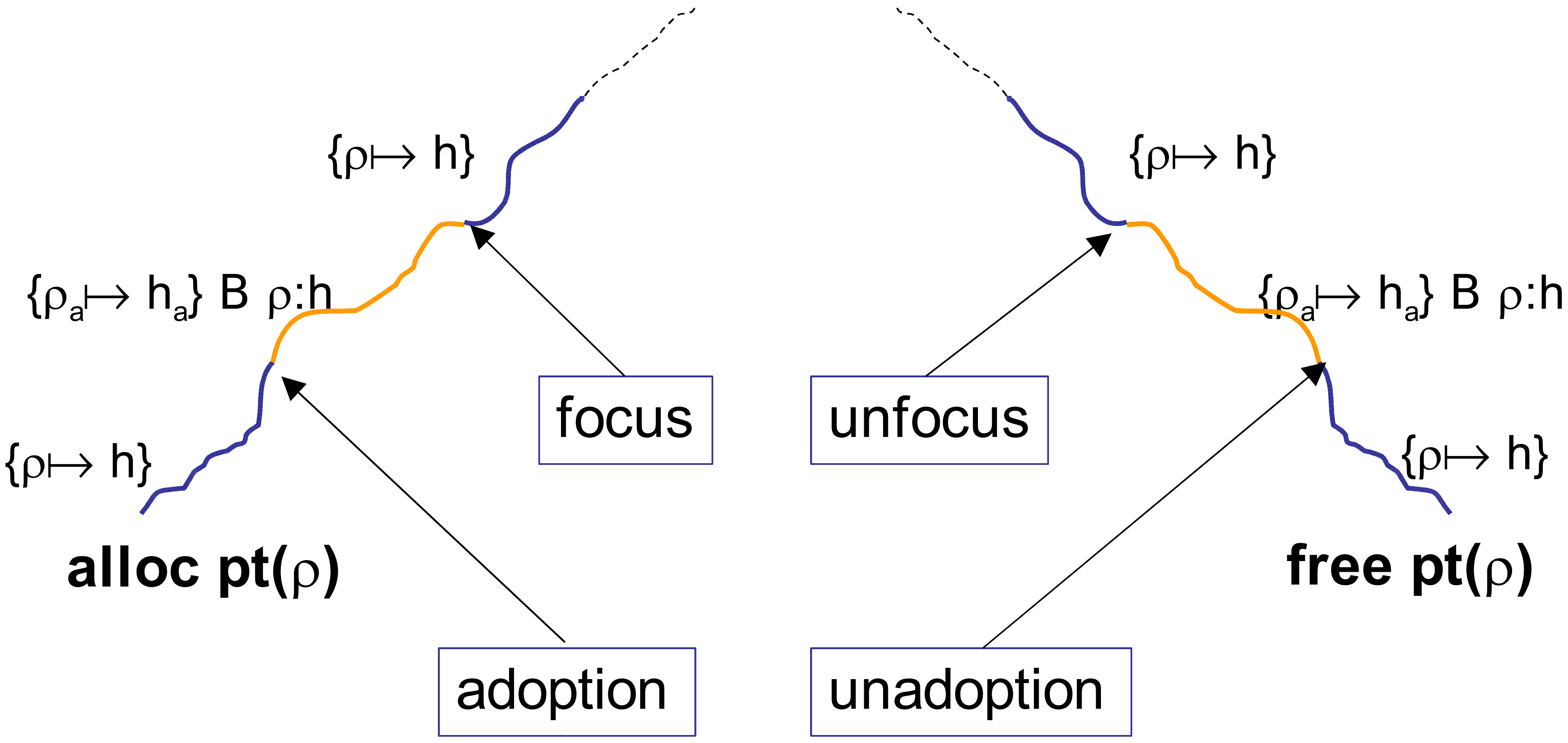
$\{\rho_0 \mapsto h_0\} - \{\rho_1 \mapsto h_1\}$



After:

$\{\rho_0 \mapsto h_0\} - \{\rho_0 \mapsto h_0\} B \rho_1 : h_1$

Data lifetime model (types)



Example Adoption

ACellPhρ_Di newCell(pt(ρ_D) Dict d) {

  pt(ρ_c) Cell c = new Cell;

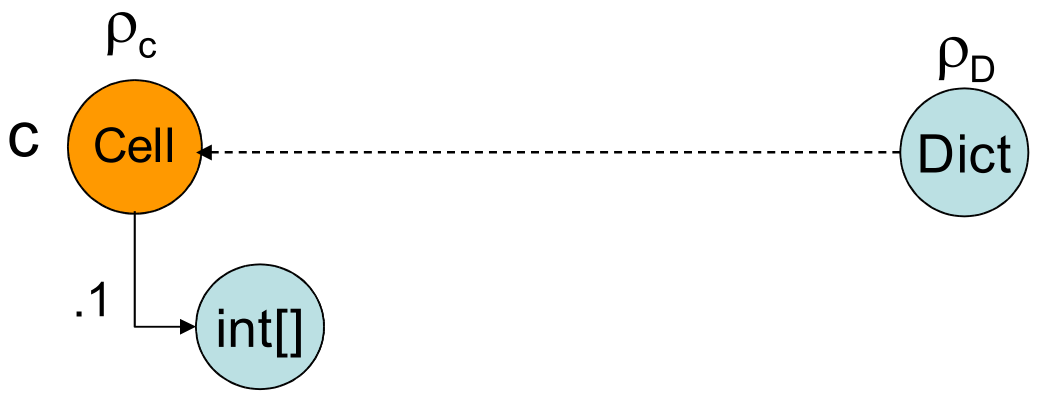
  c.data = new int[];

  return(adopt c by d);

}

$$\mathsf{Cell} = \langle \exists[\rho' \mid \{\rho' \mapsto \mathit{int}[]\}].\mathsf{pt}(\rho')\rangle$$

$$\mathsf{ACellP}\langle \rho_D \rangle = \exists[\rho' \mid \{\rho_D \mapsto \mathsf{Dict}\} \triangleright \rho' : \mathsf{Cell}].\mathsf{pt}(\rho')$$

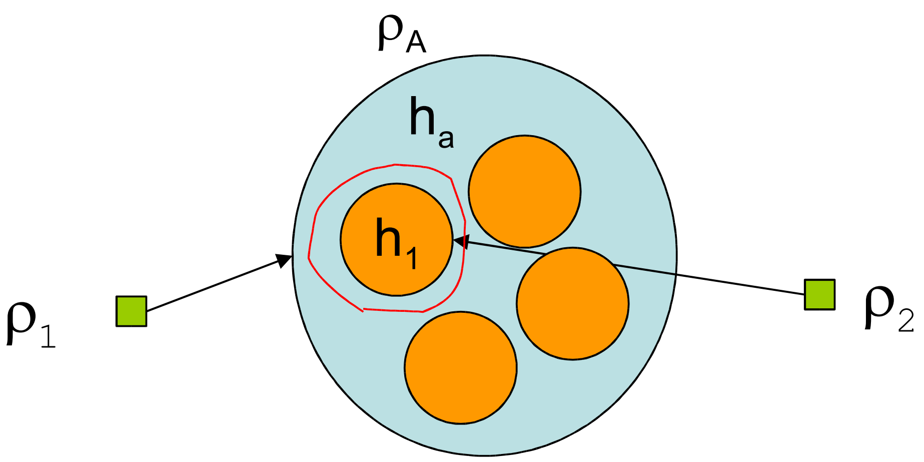


Adoption is related to **let!**

- Wadler 90:
  let! (x) y = e_1 in e_2
  linear type of x is non-linear during e_1.
- Problems:
  - Scoped
  - How to enforce escaping of components of x
  - Unsound with mutability:
    Consider ref<int[]> → ref<int[]>

Focus

focus e_1 in e_2

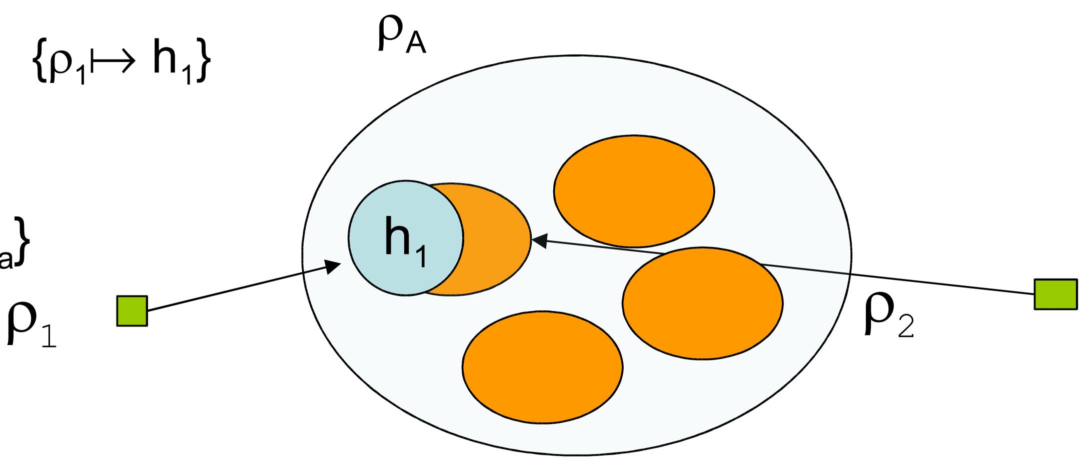


Revoke ρ_A

Restore ρ_A capability for ρ_1 $\{\rho_1 \mapsto h_1\}$

Fact to restore ρ_A

$\{\rho_1 \mapsto h_1\}$ ($\{\rho_a \mapsto h_a\}$



Example focus

void resize(ACellPhρ_Di c) {

  focus c {

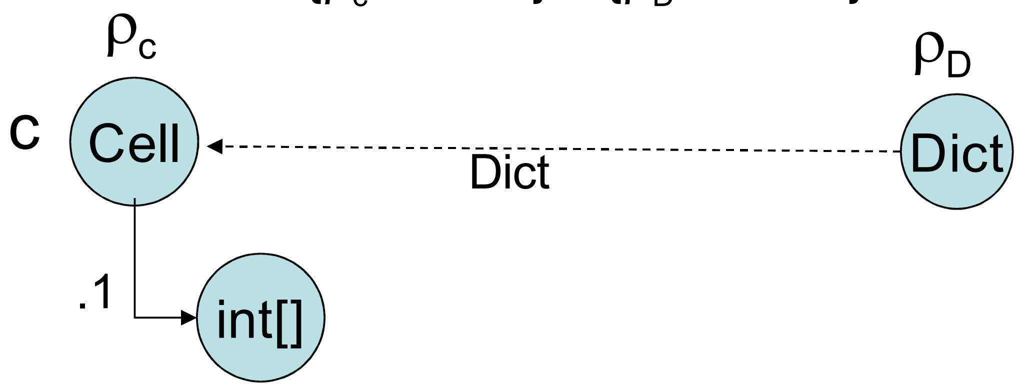
    free c.data;

    c.data = new int[];

  }

}

$\{\rho_c \mapsto \text{Cell}\} (\{\rho_D \mapsto \text{Dict}\}$



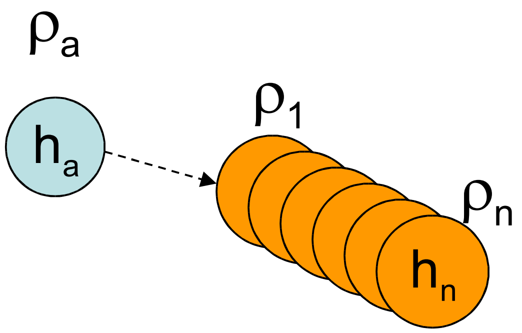
Unfocus

- $\{\rho_1 \mapsto h_1\}$ ($\{\rho_A \mapsto h_A\}$
- Can be seen as an implication or coercion function
- Explicit implication allows for non-lexical scopes
  - Right to unfocus can be passed up/down to other functions
  - Useful for inferring scopes locally

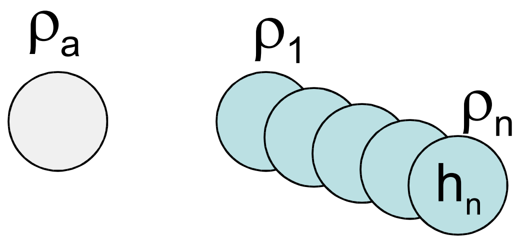
Unadoption

free ρ_a

Before:



After:



Generalizations

- Adoption facts: C B ρ:τ
- Abstract over capabilities (symbolic cap G)
- Resize function does not need to know details of adopter
- 8[ρ,G]. (G - (G B ρ:Cell), pt(ρ)) ! (void, G)
- Temporary view of non-adopted pointer as adopted
  - {ρ↦ h} ! {ρ↦ h} - {ρ↦ h} Bρ:h
  - can write functions that work over adopted and non-adopted data!

Generalizations (cont)

- Can handle interior pointers

  $\{\rho \mapsto h\}$

  h =h T_1, T_2 i

  Want pt(ρ_1) to 1st field of type T_1

    $\{\rho \mapsto h\}$ Bρ_1:T_1

- Pointers to the stack

Lecture 3

- Permission sharing
- Type states for objects
- Techniques for message based systems

Backups

Locking (3)

- Lingering problems
  - Release wrong lock

τ ::= … j Lockhρ, σi j RTokenhρi

$$\mathsf{acquire} : \forall[\rho].(\cdot, \mathsf{Lock}\langle\rho,\sigma\rangle) \to \exists[\rho'](\sigma, \{\rho' \mapsto \mathsf{RToken}\langle\rho\rangle\})$$

$$\mathsf{release} : \forall[\rho,\rho'].(\{\rho' \mapsto \mathsf{RToken}\langle\rho\rangle\}, \mathsf{Lock}\langle\rho,\sigma\rangle, \sigma) \to (\mathsf{unit}, \cdot)$$

- Code looks as expected:
  - token not passed explicitly

```
T x = acquire(lock);
…
release(lock, x);
```

Packing and unpacking of h

$$\frac{\begin{array}{l} A; C_1 \vdash e : \mathsf{pt}(i);\ C_2 \\ C_2 = \{i \mapsto S(h)\} \otimes S(C) \otimes C_3 \\ S = [\Delta'/\Delta] \end{array}}{A; C_1 \vdash \mathsf{pack}[\Delta \mid C].e : \mathsf{pt}(i);\ \{i \mapsto \exists[\Delta \mid C].h\} \otimes C_3}\,[\text{pack-h}]$$

$$\frac{\begin{array}{l} A; C_1 \vdash e : \mathsf{pt}(i);\ C_2 \\ C_2 = \{i \mapsto \exists[\Delta \mid C].h\} \otimes C_3 \end{array}}{A; C_1 \vdash \mathsf{unpack}\, e : \mathsf{pt}(i);\ \{i \mapsto h\} \otimes C \otimes C_3}\,[\text{unpack-h}]$$