Dynamic Software Updating:
Introduction and Foundation

Presented by

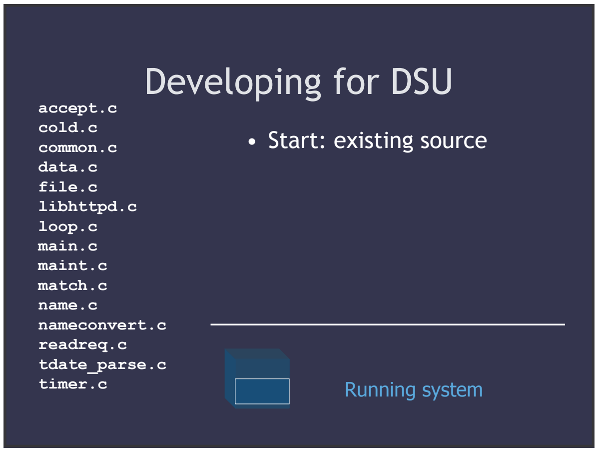
Michael Hicks

Oregon Summer School 2006

Dynamic Software Updating
(DSU)

- Update a running program with new code and data
  - Preserves state and processing
- Critical for non-stop systems
  - Air-traffic control, financial transaction processing, network components, …
- Convenient for other systems
  - No need to reboot your OS after a patch!

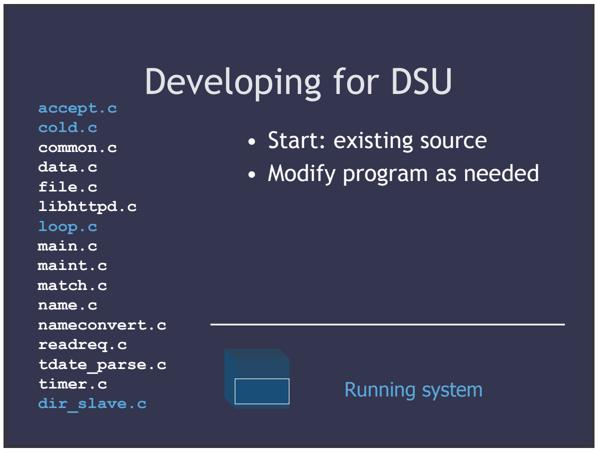
Developing for DSU



```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
```

- Start: existing source

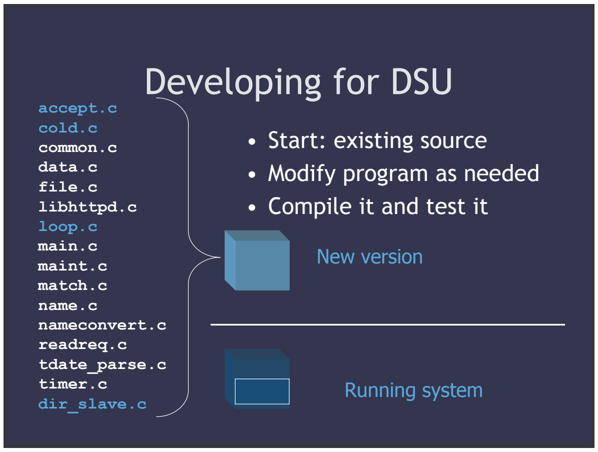
Developing for DSU



```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

- Start: existing source
- Modify program as needed

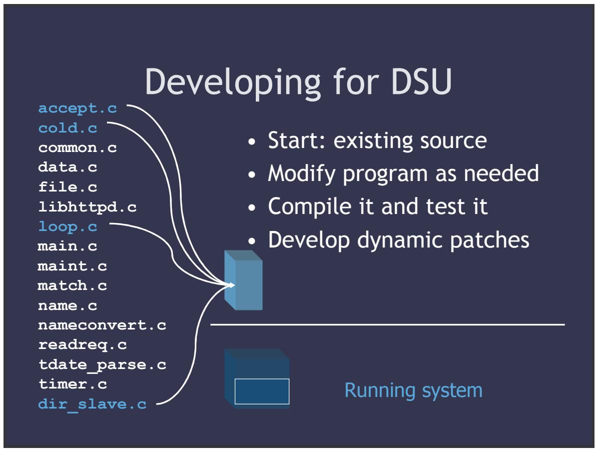
Developing for DSU



```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

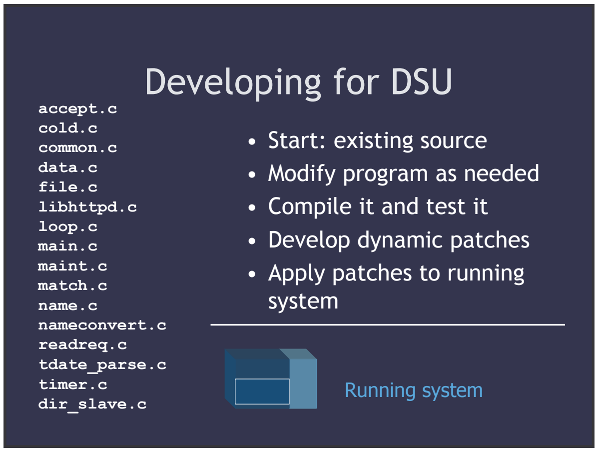
- Start: existing source
- Modify program as needed
- Compile it and test it

Developing for DSU



```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

- Start: existing source
- Modify program as needed
- Compile it and test it
- Develop dynamic patches

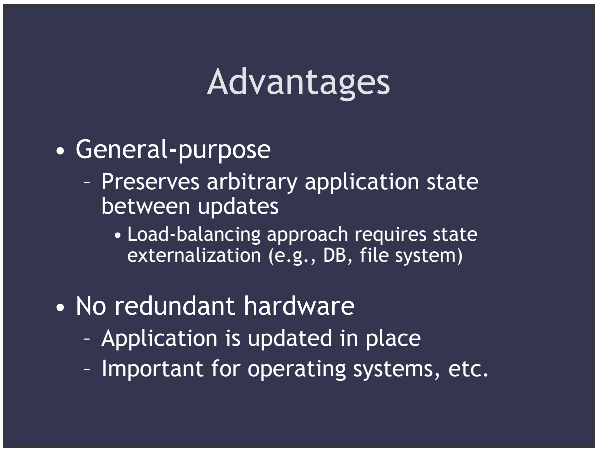


Developing for DSU

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

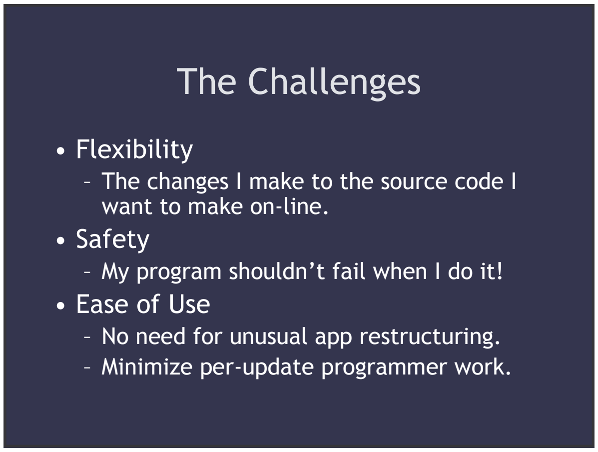
- Start: existing source
- Modify program as needed
- Compile it and test it
- Develop dynamic patches
- Apply patches to running system

Running system



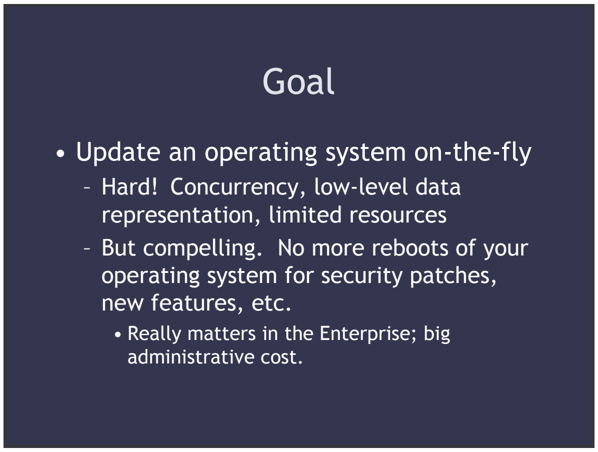
Advantages

- General-purpose
  - Preserves arbitrary application state between updates
    - Load-balancing approach requires state externalization (e.g., DB, file system)
- No redundant hardware
  - Application is updated in place
  - Important for operating systems, etc.



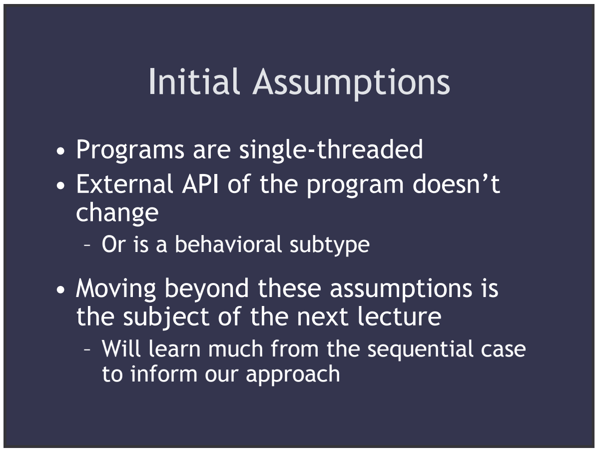
The Challenges

- Flexibility
  - The changes I make to the source code I want to make on-line.
- Safety
  - My program shouldn’t fail when I do it!
- Ease of Use
  - No need for unusual app restructuring.
  - Minimize per-update programmer work.



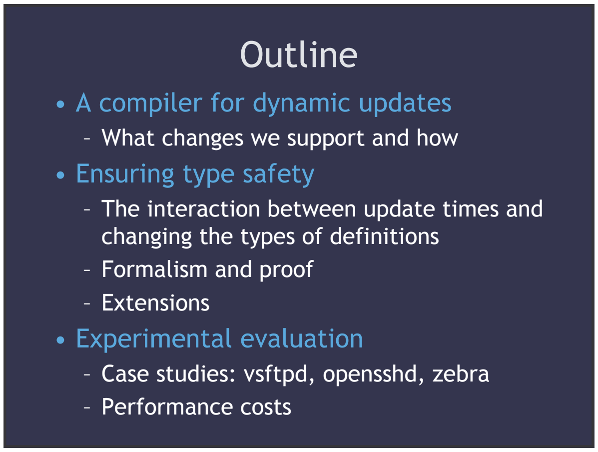
Goal

- Update an operating system on-the-fly
  - Hard! Concurrency, low-level data representation, limited resources
  - But compelling. No more reboots of your operating system for security patches, new features, etc.
    - Really matters in the Enterprise; big administrative cost.



Initial Assumptions

- Programs are single-threaded
- External API of the program doesn’t change
  - Or is a behavioral subtype
- Moving beyond these assumptions is the subject of the next lecture
  - Will learn much from the sequential case to inform our approach



Outline

- A compiler for dynamic updates
  - What changes we support and how
- Ensuring type safety
  - The interaction between update times and changing the types of definitions
  - Formalism and proof
  - Extensions
- Experimental evaluation
  - Case studies: vsftpd, opensshd, zebra
  - Performance costs

Software Evolution Trends

- Observed changes in popular apps
  - OpenSSH, vsftpd, Linux, Bind, Apache
- Results
  - Many functions added, existing functions change frequently, few functions deleted
  - Type signatures change, generally simply
    - Less often: typedefs, structs
    - More often: function prototypes
    - Almost never: global variables

Dynamic Updates: Form

- Replace, add, or delete definitions
  - Functions, globals, and type definitions
  - Updated functions may have different types
- To update a type definition **t**, user provides a *type transformer* function **c**
  - Used by the runtime to convert values of type **t** to the new representation

Compilation Techniques

- Function indirection: compiler adds an indirection between each caller and called function
  - Each function call will always be to the most recent version
- Type wrapping: compiler makes accesses to values of named type to be through special functions
  - May run type transformers on the accessed value if its type has been updated

Example

```
struct T { int x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

Example: Type wrapping

```
struct __T0 { int x; int y; };

struct T {
  unsigned int version;
  union { struct __T0 data;
          char slop[…]; } u;
};

struct __T0* __con_T(struct T* abs){
  __DSU_transform(abs);
  return &abs->u.data;
}
```

Alternative: Add Indirection

```
struct __T0 { int x; int y; };

struct T {
  unsigned int version;
  struct __T0 *data;
};

struct __T0* __con_T(struct T* abs){
  __DSU_transform(abs);
  return abs->data;
}
```

Example: Accessing Types

```
void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

Example: Accessing Types

```
void call() {
  struct T t =
    { 0, {.data={1,2}} };
  foo(&t.x);
}
```

Example: Accessing Types

```
void call() {
  struct T t =
    { 0, {.data={1,2}} };
  foo(&(__con_T(&t))->x);
}
```

Example: Function Indirection

```
void foo(int* x) { *x = 1; }

void call() {
  struct T t = …;
  foo(&(__con_T(&t))->x);
}
```

Example: Function Indirection

```
void __foo_v0(int* x) { *x = 1; }

void __call_v0() {
  struct T t = …;
  foo(&(__con_T(&t))->x);
}
```

Example: Function Indirection

```
struct __fun { void* fptr; …};
struct __fun foo = { __foo_v0,…};

void __foo_v0(int* x) { *x = 1; }

void __call_v0() {
  struct T t = …;
  foo(&(__con_T(&t))->x);
}
```

Example: Function Indirection

```
struct __fun { void* fptr; …};
struct __fun foo = { __foo_v0,…};

void __foo_v0(int* x) { *x = 1; }

void __call_v0() {
  struct T t = …;
  (foo.fptr)(&(__con_T(&t))->x);
}
```

Updating code on the stack

- Dynamic updates take effect at function calls
  - A function call is always to the most recent version
- What about code that is on the stack?
  - Long running loops
  - Code that is returned to

Loop extraction

- Extract out loop body into function
  - Argument is *loop state*: consists of all locals and parameters in the host function
- Loop actions (break, continue, etc.) become return codes handled in host
- Reuses existing updateability mechs.
- Can be used for arbitrary code S by changing that code to be
  - while (1) { S; break; }

Example: vsftpd

```
main() {
  … init …
  if (tunable_listen)
    standalone_main();
  … handle conn …
}
```

```
standalone_main() {
  … init listen sock l …
  while (1) {
    if (x = acceptconn(l))
      fork and return
      in child
  }
}
```

Example: vsftpd

```
main() {
  … init …
  if (tunable_listen)
    standalone_main();
  while (1) {
    … handle conn …
    break;
  }
}
```

```
standalone_main() {
  … init listen sock l …
  while (1) {
    if (x = acceptconn(l))
      fork and return
      in child
  }
}
```

Notes on Mechanisms

- Compilation is not the only way to effect changes
  - Could rewrite program text to redirect function calls
  - Could overwrite data in-place, at update-time
- But it's simple and flexible, so we use it for now

Problem: Bad Timing

- Updating **t** when some existing code still expects the old representation could lead to a type error.
  - This situation is timing dependent.

***Question**: when during a program's execution is it safe to update the representation of a type **t**?*

Example

```
struct T { int x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

Example: version 2

```
struct T { int *x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  int z = 1;
  struct T t = {&z,2};
  foo(&t.x);
}
```

Starting execution

```
struct T { int x; int y; };
  struct T { int *x; int y; };
void foo(int* x) { *x = 1; }

void call() {
> struct T t = {1,2};
  foo(&t.x);
}
```

Attempting update now

```
struct T { int x; int y; };
  struct T { int *x; int y; };
void foo(int* x) { *x = 1; }

void call() {
  struct T t = {1,2};
> foo(&t.x);
}
```

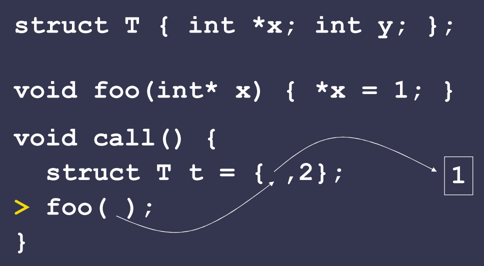
Run type transformer

```
struct T { int *x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  struct T t = {  ,2};
> foo(&t.x);
}
```

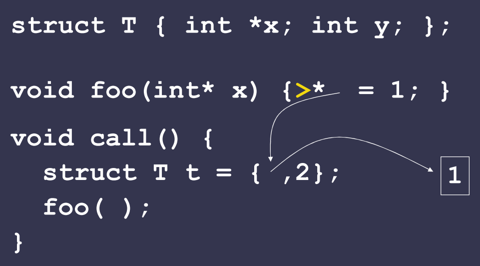
Taking the address



```
struct T { int *x; int y; };

void foo(int* x) { *x = 1; }
void call() {
  struct T t = { ,2};
> foo( );
}
```

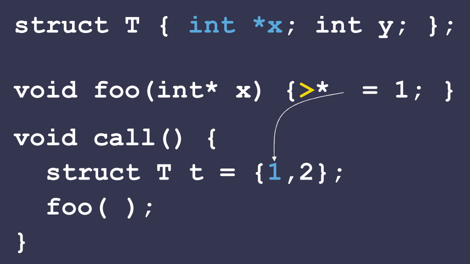
Call foo()



```
struct T { int *x; int y; };

void foo(int* x) {>*  = 1; }
void call() {
  struct T t = { ,2};
  foo( );
}
```

Doing the assignment: error!



```
struct T { int *x; int y; };

void foo(int* x) {>*  = 1; }
void call() {
  struct T t = {1,2};
  foo( );
}
```

The problem

- The new program was type correct
- But the old version of `call` was active at the time of the update, and expected the *old* `struct T` rep
  - It uses it *concretely*
- A similar situation occurs when changing the types of functions or global variables

Possible Solution #1

- Copy and transform values whose types have changed to the new code, leaving the existing ones as is (Hicks 2001).
- Problem
  - Old code could operate on stale data, or call old versions of functions
  - Update point must be chosen carefully

Possible Solution #2

- Allow it, but require *backward* type transformers for each updated type **T** (Duggan 2002) and *stubs* for functions that changed type (Segal 1990)
- Problems
  - May not be possible to convert a type backwards, particularly since type changes often add information
  - Hard to reason about program behavior
    - Convert forward, back, forward = ?

Possible Solution #3

- Disallow updates to active code (Gilmore 1997, Malabarba 2000, …)
- Problems:
  - Updates less available (loops)

Our Approach: Safety Analysis

- *__con_T* functions identify when a type is used concretely
- Dynamically prevent updates that could lead to old code concretely using a transformed value
  - Calculate dependencies at compile-time
  - Apply same idea to function calls, global variable references

Example revisited

```
void foo(int* x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3
  foo(&t.x);
4
}
```

Example revisited

```
void foo(int* x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3 {T,foo}
  foo(&t.x);
4
}
```

Dependence on type of T and foo

Example revisited

```
void foo(int* x) {
1 {}
  *x = 1;
2 {}
}

void call() {
  struct T t = {1,2};
3 {T,foo}
  foo(&t.x);
4 {}
}
```

No type dependencies

Dependence on type of T and foo

Formalism: Proteus

- Soundness (POPL 2005)
  - Type system of a simple imperative language called Proteus
    - Update points made explicit in program text
    - Efficient constraint-based inference
  - Well-formed and well-timed updates will not cause the program to go wrong
- Adapted approach to updating security policies (FCS 2005, CSFW 2006)

Proteus Typing Judgments

$$\Delta; \Gamma \mid\!\text{-}\ e : t; \Delta'$$

- Δ is a *capability*
  - set of type names that can be accessed concretely
- Read judgment as:
  - e can be typed with capability Δ, and the evaluation of e results in capability Δ'

Typing: **Con** and **Update**

$$\frac{\Delta; \Gamma \mid\!\text{-}\ e : \mathbf{t}; \Delta' \quad \Gamma|_{\Delta'}(t) = \tau}{\Delta; \Gamma \mid\!\text{-}\ \mathrm{con}_t\ e : \tau; \Delta'}$$

$$\frac{\Delta \subseteq \Delta'}{\Delta; \Gamma \mid\!\text{-}\ \mathrm{update}^{\Delta'} : \mathrm{int}; \Delta'}$$

Typing: **App** (Intuition)

- We would expect that to call a function **f**, it must have an input capability the caller must satisfy
  - This is unnecessary: at update-time we ensure that all functions are consistent with the current type definitions (condition shown later).
  - However, Function **f**'s output capability will impact the capability of the caller, since **f** could perform an update.

Typing: **App**

$$\frac{\Delta; \Gamma \mid\!\text{-}\ e_1 : \tau_1 \rightarrow^{\Delta e} \tau_2; \Delta' \qquad \Delta; \Gamma \mid\!\text{-}\ e_2 : \tau_1; \Delta''}{\Delta; \Gamma \mid\!\text{-}\ e_1\ e_2 : \tau_2; \Delta'' \cap \Delta_e}$$

Abstraction-violating Aliases

- Cannot transform a value when there exists an alias into it
  - Reveals its representation indirectly
- An alias into a value of type T should prevent T's update

Example revisited

```
void foo(int*    x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3
  foo(&t.x);
4
}
```

Example revisited

```
void foo(int*{T} x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3
  foo(&t.x);
4
}
```

Parameter is suspect

Creates suspect alias

Example revisited

```
void foo(int*{T} x) {
1 {T}
  *x = 1;
2 {T}
}

void call() {
  struct T t = {1,2};
3
  foo(&t.x);
4
}
```

Suspect alias active

Example revisited

```
void foo(int*{T} x) {
1 {T}
  *x = 1;
2 {T}
}

void call() {
  struct T t = {1,2};
3 {}
  foo(&t.x);
4 {}
}
```

Suspect alias active

No active suspect aliases

Combining the Analyses

```
void foo(int*{T} x) {
1 {T}
  *x = 1;
2 {T}
}

void call() {
  struct T t = {1,2};
3 {T,foo}
  foo(&t.x);
4 {}
}
```

Implementation

- Compiler
  - implemented using CIL framework
- Safety analysis
  - Extended to support changes to function types and the & operator
- Patch generation tool
  - Constructs default type transformers
- DLOPEN library for loading patches

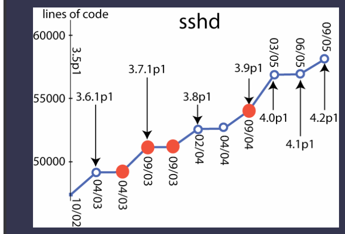
Three Years of Changes

3 years
13 releases

3 years
11 releases

4 years
6 releases

Sshd Evolution History



- Functions
  - 131 added, 19 deleted
  - 85 proto changed
  - 752 body changed
- Types
  - 27 added, 2 deleted
  - 19 changed
- Global variables
  - 70 added, 19 deleted
  - 29 changed

3 years
11 releases

Dynamic Update Catalysts

Or: why does DSU work??

1. Quiescence
2. Functional state transformation
3. Type-safe programs
4. Robust design

Quiescence

- No in-flight transactions
- Consistent global state
- Shallow stack
- Quiescent point → update point

Quiescent Points Are Easy to Find

```
while(1) {
   …
   newsock = accept();
   fork_child(newsock);

   …

   update(); ←———        quiescent point
}
```

Functional State Transformation

- Assumption: can convert global state
  - New_state = f(Old_state)
- No guarantees
  - Assumption might not hold (2 out of 27 updates)
  - Can recover/compensate

Type-safe Programs

- Good news: C programmers generally adhere to type safe programming style
- Low-level idioms hamper updateability
  - Illegal casts, inline assembly
  - Non-updateable types
  - Restrict range of updates
- **`void *`**
  - C lacks polymorphism
  - Usually benign

Robust Design

- Global invariants
  - Updates must preserve invariants
  - Usually implicit
  - Explicit invariants - `assert`
- Test suites

Experiments

- Throughput
  - Transfer rate in `vsftpd`, `sshd`: **unaffected**
- Overhead
  - Connection setup+tear in `vsftpd`, `sshd`: 0..32%
  - Route setup/route redist in `zebra`: 4..12%
- Memory footprint
  - 0..40% (no old code/data unloading)
- Update application time
  - Less than 5 ms

Programming Effort

	Source code (LOC)		
App	Application changes	Type+state transformers	Patch generator (automatic)
vsftpd	< 50	162	83965
sshd	< 50	125	248587
zebra	< 50	49	43173

Challenging Assumptions

- So far, we have assumed that dynamically updateable programs
  - Are sequential, not multi-threaded
  - Do not change their external (communication) interfaces
- But many long-running programs are multi-threaded, and upgrade their communication protocols
  - Medium-term goal: robust upgrades of OSs

Multi-threaded Problems

- Cannot apply an update at the first-reached update point in some thread
  - Other threads could be at arbitrary points
  - How should safety analysis treat thread-spawn?
- Lazy transformation of named-type values may introduce data races not in the original program
  - Atomic operations compiled to non-atomic ones

Basic Approach

- Require *all* threads to reach safe update points (or terminate) before applying the dynamic patch
  - Updates will occur at well-defined points
- Eagerly transform named-type data while program is paused
  - No change to data representation

Review: the (App) rule

$$\frac{\Delta;\ \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Delta e} \tau_2;\ \Delta' \qquad \Delta;\ \Gamma \vdash e_2 : \tau_1;\ \Delta''}{\Delta;\ \Gamma \vdash e_1\ e_2 : \tau_2;\ \Delta'' \cap \Delta_e}$$

Thread-spawn rule

$$\frac{\Delta;\ \Gamma \vdash e_1 : \tau_1 \rightarrow^{\Delta e} \tau_2;\ \Delta' \qquad \Delta;\ \Gamma \vdash e_2 : \tau_1;\ \Delta''}{\Delta;\ \Gamma \vdash \text{spawn}\ e_1\ e_2 : \tau_2;\ \Delta''}$$

- The output capability of e_1 does not affect the caller's output capability

Eager Transformation

- Need a way to "find" the data in the program so that it can be changed
  - Use the *factory* pattern to keep track of typed data when it is created
  - At update-time, iterate over all of the data and transform it

Tradeoffs

- Fairly simple departure from sequential approach, but
- Forces program to wait while
  - All threads barrier synchronize
  - All data is transformed
- Could create an unacceptable pause
  - Or deadlock

Observation

- We can improve availability by only pausing threads whose actions might conflict with a dynamic patch
- This is a separation property
  - a la *separation logic*
  - But rather than reasoning about heap locations, we reason about concrete uses of named-type data or definitions

Thread separation

- No need to pause any thread whose definitions/types are disjoint with a patch's definitions/types

type **t** = τ

fun **f**$^{\{t\};\{\}}$ (**x**:int) : int = … (con$_t$ e) … in

fun main() = spawn **f** **x**; update$^{\Delta}$; …

No need to wait for child **f** to terminate
if dynamic patch does not mention **t**

ADT Separation

- ADTs' maintain internal invariants distinct from the rest of the program
  - Abstract type & attendant functions
  - Object, as in Java or C++
- Idea: permit updating an ADT while the ADT code is inactive
  - Ensures invariants are preserved

K42 Operating System

- OS components written as individual objects in C++
  - File cache manager
  - Scheduler
- Permits hot-swapping individual objects at run time
  - To fix bugs
  - To improve performance

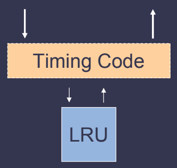
K42 Implementation

- Designed to scale to large SMP machines
  - Preemptive kernel
  - Actions performed by lightweight, short-lived threads
  - Uses an *object translation table* to insert a level of indirection between callers of object methods and the objects

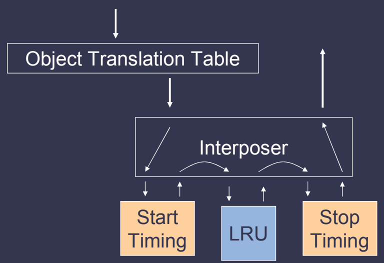
Enforcing ADT separation

- Hot-swapping in K42 only occurs when the object is inactive
  - Enforced by a dynamic quiescence protocol
- Two mechanisms [Soules et al 2003]
  - *Interposition* of a mediator object, to shepherd the update
  - Means to track when threads are accessing a given object using *thread epochs*

Interposition



Interposition



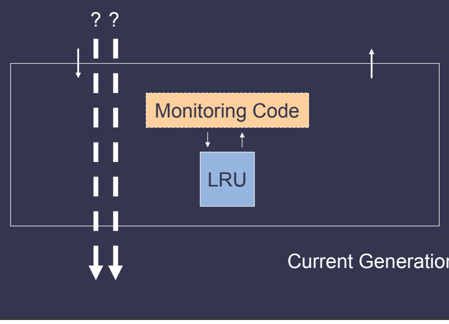
Applications of Interposition

- Counters
- Timers
- Logging
- Debugging
  - Check arguments coming in
  - Modify arguments coming in
- Replication
- …

Quiescence in K42

- Use a thread generation count
  - Maintain a global generation marker
  - Mark each new thread with a generation
  - Keep a counter of live threads for each active generation
- Implements a form of *Read-Copy-Update* (RCU) synchronization

Quiescence in K42

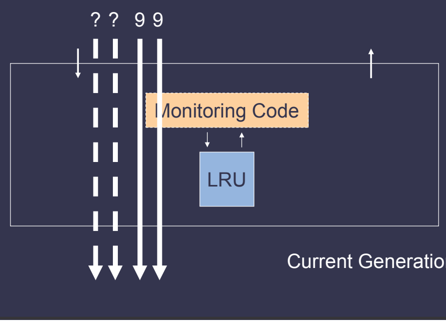

? ?

Monitoring Code

LRU

Current Generation: 8

Quiescence in K42

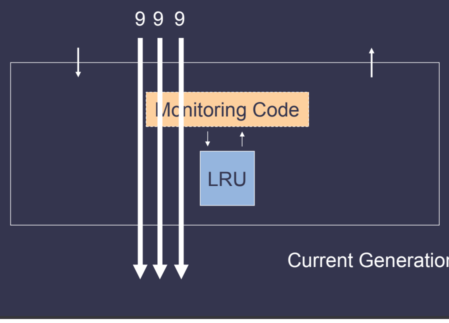

? ? 9 9

Monitoring Code

LRU

Current Generation: 9

Quiescence in K42

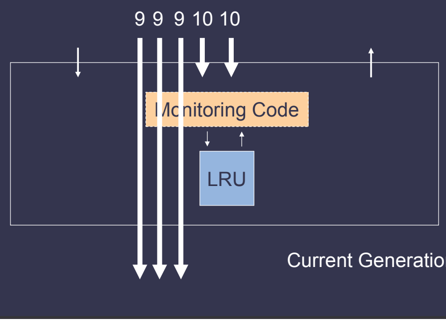

9 9 9

Monitoring Code

LRU

Current Generation: 9

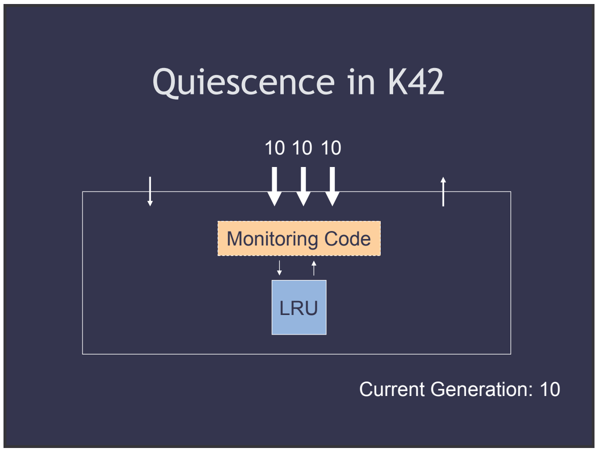
Quiescence in K42


9 9 9 10 10

Monitoring Code

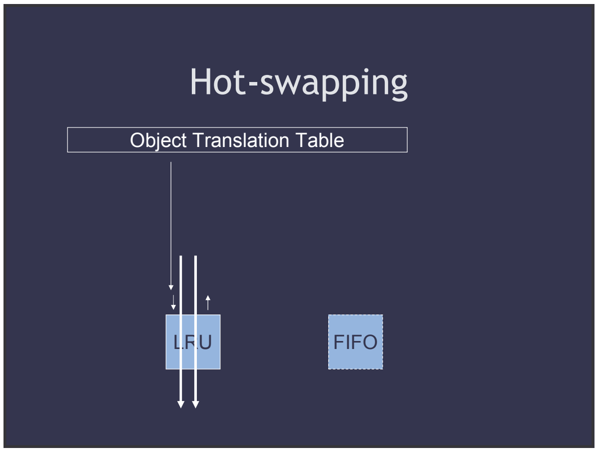
LRU

Current Generation: 10

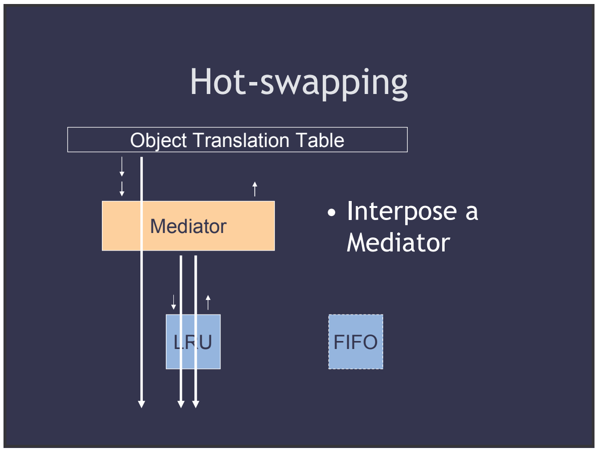


Quiescence in K42

Current Generation: 10

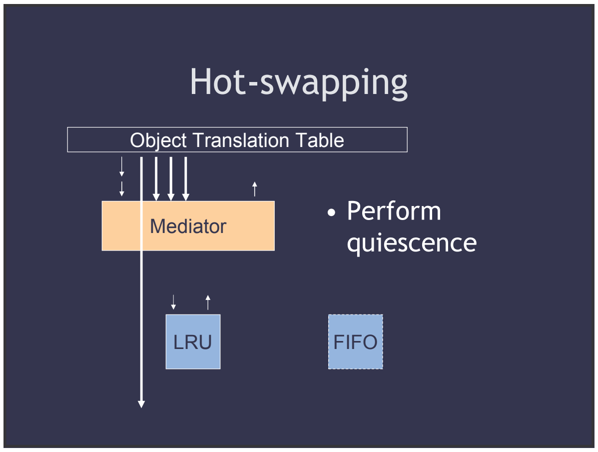


Hot-swapping



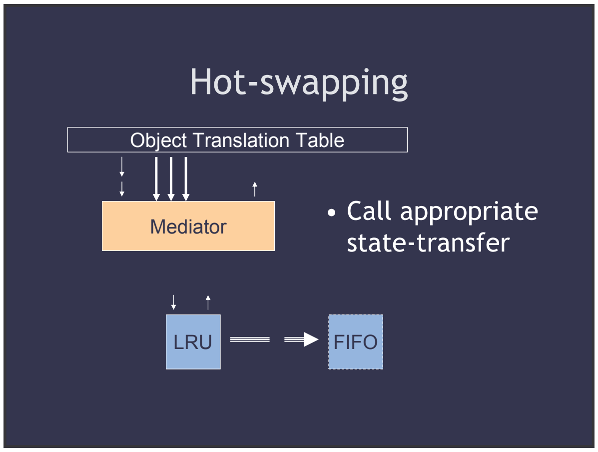
Hot-swapping

- Interpose a Mediator



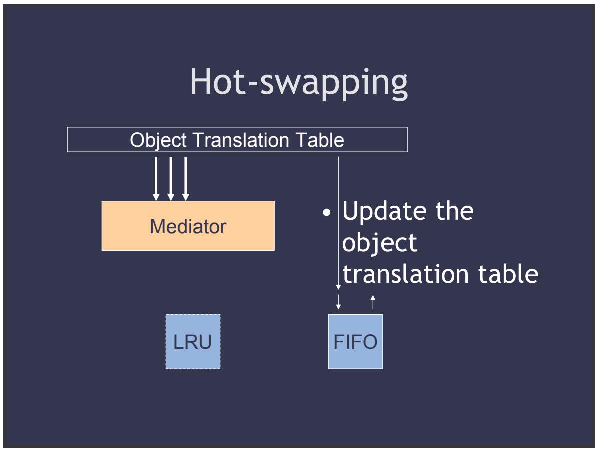
Hot-swapping

- Perform quiescence



Hot-swapping

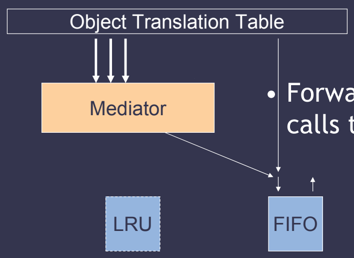
- Call appropriate state-transfer



Hot-swapping

- Update the object translation table

Hot-swapping



- Forward blocked calls to FIFO

Caveats

- RCU/thread generation reduces overhead
- Problems updating multiple objects simultaneously
  - Could lead to deadlock
  - Possible I/O invariants violations
- Not straightforward to change method types
  - Requires a “stub” to mediate old caller to new method

How to adapt to our DSU framework?

Warning! What follows is half-baked …

Adapting K42 approach

- Define an ADT as a type **t** and the set of functions **f_1**, **f_2**, …, **f_n** that use **t** concretely
  - they contain an operation (con_t e)
- A call to an ADT function logically represents a *transaction*
  - Object invariant satisfied on entry and exit

Updates & Transactions

- Earlier, we said that dynamic updates must occur when the program is quiescent
  - K42 allows updating object **o** when it is quiescent (inactive)
- In our DSU system, we can think of an update occurring at a transaction boundary
  - Enforces atomicity of *program versions* (vs. atomicity of heap effects)

Common DSU structure …

```
while (1) {
  update;
  // perform processing
}
```

… viewed as a transaction

```
while (1) {
  update;
  begin transaction
  // perform processing
  end transaction
}
```

Processing is atomic with respect to updates

Updating Rule

- An update *within* a transaction must not change any code or data within that transaction
  - In our example, the update point was defined *outside* the transaction boundary, respecting this rule vacuously
  - When might updates inside be sensible?

Nested Transactions

- To support the finer-granularity transactions of ADTs, we are likely to have nesting
  - But the prior rule would have outer transactions subsume inner ones
- Rule amendment: outer transactions do not restrict updates to code within nested transactions
  - Modulo restrictions to ensure type safety

Synchronizing Updates

- Strategy 1: optimism and rollback
  - When an update is available, abort the the transaction(s) in each thread until the update rule is satisfied
- Benefit: updates take place very quickly
- Drawbacks: overhead to support undo; may not be able to undo side-effects (I/O)

Synchronizing Updates

- Strategy 2: roll-forward and block
  - Conflicting threads proceed until ok
  - Nonconflicting threads proceed until they are about to conflict, and then block
  - Update when all threads non-conflicting
- Benefit: no need to support rollback, no worry about undoing effects
- Drawback: longer to converge

Detecting conflicts

- Cannot wait until a transaction completes to know whether it might conflict
  - Otherwise would have to roll back the update itself
- Instead: use static analysis
  - Soundly approximate all of those functions, types, etc. that could be accessed during the transaction

Adding Flow Sensitivity

- While the whole of a transaction may conflict with an update, it may be
  - The part of the transaction that conflicts has *already* completed
  - The part of the transaction that will conflict has *not yet* taken place
- In both cases, we can perform the update safely right away
  - The former simulates no-op roll-forward
  - The latter simulates no-op rollback

Updating Model

- The prior discussion has assumed that updates always “march forward”
  - The old program transitions to the new program (almost) immediately
  - Challenge is to reduce pauses by being fine-grained about where/when updates can take place
- What if we need pieces of the program to have different versions?
  - E.g., in a distributed system, different nodes under different administrative control

Updating Distributed Systems

[Ajmani et al 2006]

- Upgrade the entire system in a decentralized way
  - No synchronization required
- Implication: different nodes might be running different versions of the software
- Question: how do we reason about this situation to ensure it’s OK?

Modeling Distributed Updates

- Each node has a single object
  - Simple, but good for abstract thinking
- Each message sent to a node is an RPC
- Objects have versions
- Messages to nodes include the sender’s expected version

Simulation

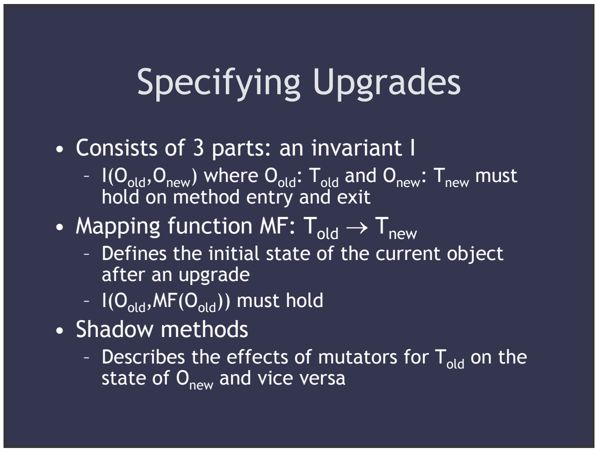
- Each node/object has a “current version” but may *simulate* the other versions
- An upgrade from T_{old} to T_{new} yields an object with a compound type $T_{old\&new}$
  - contains the state of both objects
  - has the methods of both types

Implementing Simulation

- Messages whose version is not the current version N handled by *simulation objects*
  - Past SO: one for each version L < N
  - Future SO: one for each version F > N
- Typically implemented by *delegation* to the current object

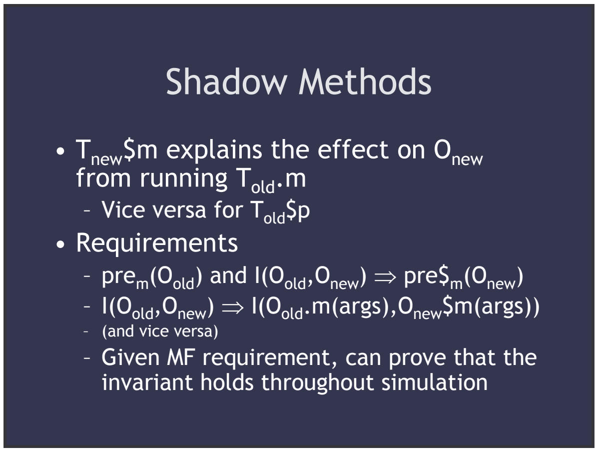
Multi-version Nodes

QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.



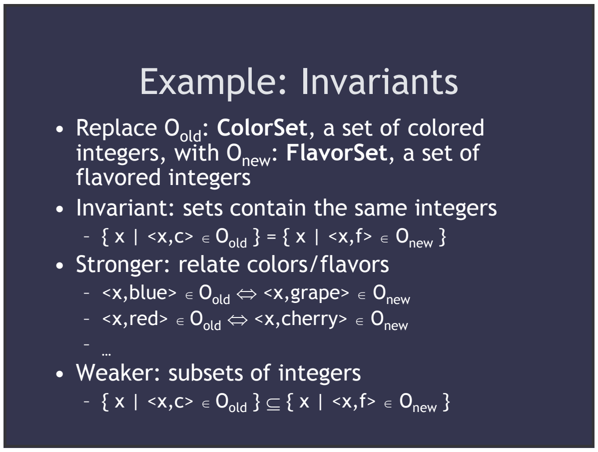
Specifying Upgrades

- Consists of 3 parts: an invariant I
  - $I(O_{old},O_{new})$ where O_{old}: T_{old} and O_{new}: T_{new} must hold on method entry and exit
- Mapping function MF: $T_{old} \to T_{new}$
  - Defines the initial state of the current object after an upgrade
  - $I(O_{old},MF(O_{old}))$ must hold
- Shadow methods
  - Describes the effects of mutators for T_{old} on the state of O_{new} and vice versa



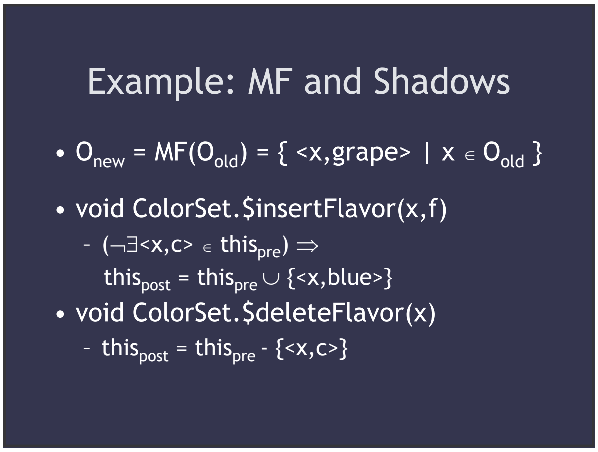
Shadow Methods

- $T_{new}\$m$ explains the effect on O_{new} from running $T_{old}.m$
  - Vice versa for $T_{old}\$p$
- Requirements
  - $pre_m(O_{old})$ and $I(O_{old},O_{new}) \Rightarrow pre\$_m(O_{new})$
  - $I(O_{old},O_{new}) \Rightarrow I(O_{old}.m(args),O_{new}\$m(args))$
  - (and vice versa)
  - Given MF requirement, can prove that the invariant holds throughout simulation



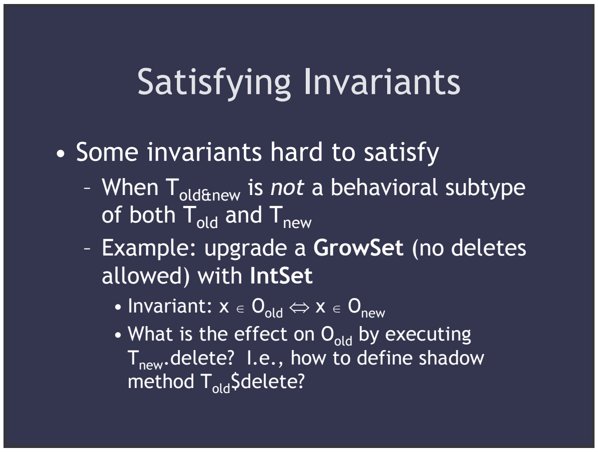
Example: Invariants

- Replace O_{old}: **ColorSet**, a set of colored integers, with O_{new}: **FlavorSet**, a set of flavored integers
- Invariant: sets contain the same integers
  - $\{ x \mid \langle x,c\rangle \in O_{old} \} = \{ x \mid \langle x,f\rangle \in O_{new} \}$
- Stronger: relate colors/flavors
  - $\langle x,blue\rangle \in O_{old} \Leftrightarrow \langle x,grape\rangle \in O_{new}$
  - $\langle x,red\rangle \in O_{old} \Leftrightarrow \langle x,cherry\rangle \in O_{new}$
  - …
- Weaker: subsets of integers
  - $\{ x \mid \langle x,c\rangle \in O_{old} \} \subseteq \{ x \mid \langle x,f\rangle \in O_{new} \}$



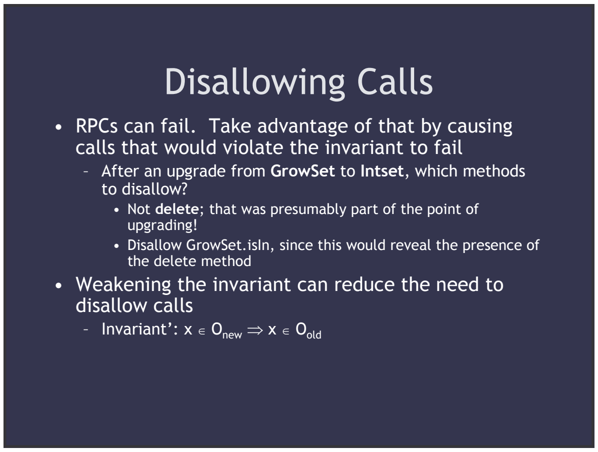
Example: MF and Shadows

- $O_{new} = MF(O_{old}) = \{ \langle x,grape\rangle \mid x \in O_{old} \}$
- void ColorSet.$insertFlavor(x,f)
  - $(\neg\exists\langle x,c\rangle \in this_{pre}) \Rightarrow$ $this_{post} = this_{pre} \cup \{\langle x,blue\rangle\}$
- void ColorSet.$deleteFlavor(x)
  - $this_{post} = this_{pre} - \{\langle x,c\rangle\}$



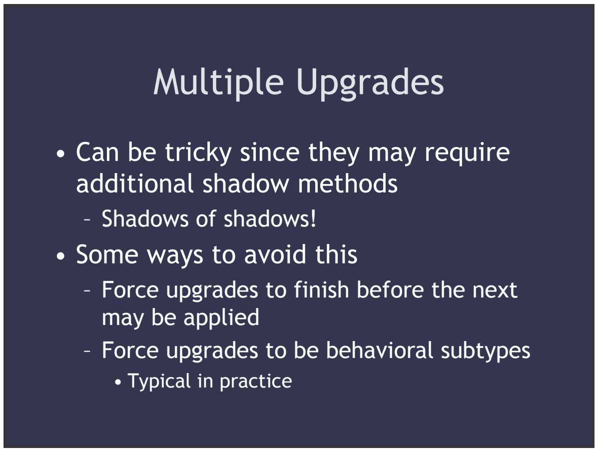
Satisfying Invariants

- Some invariants hard to satisfy
  - When $T_{old\&new}$ is *not* a behavioral subtype of both T_{old} and T_{new}
  - Example: upgrade a **GrowSet** (no deletes allowed) with **IntSet**
    - Invariant: $x \in O_{old} \Leftrightarrow x \in O_{new}$
    - What is the effect on O_{old} by executing T_{new}.delete? I.e., how to define shadow method $T_{old}\$delete$?



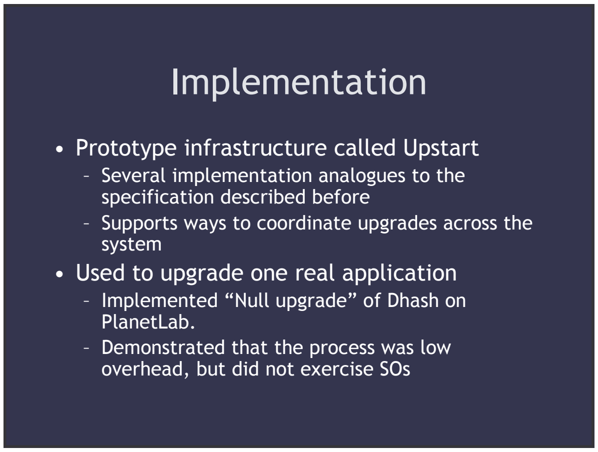
Disallowing Calls

- RPCs can fail. Take advantage of that by causing calls that would violate the invariant to fail
  - After an upgrade from **GrowSet** to **Intset**, which methods to disallow?
    - Not **delete**; that was presumably part of the point of upgrading!
    - Disallow GrowSet.isIn, since this would reveal the presence of the delete method
- Weakening the invariant can reduce the need to disallow calls
  - Invariant': $x \in O_{new} \Rightarrow x \in O_{old}$



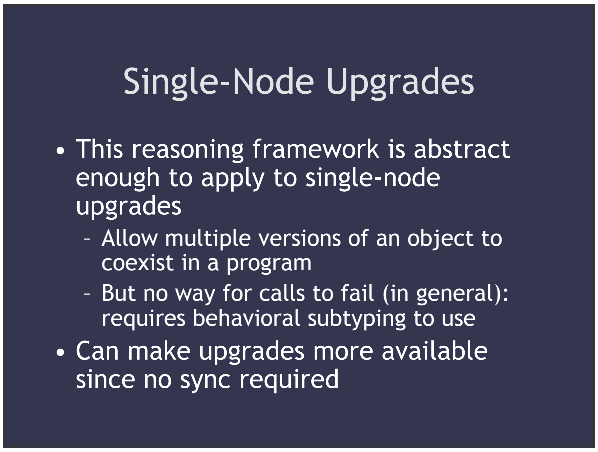
Multiple Upgrades

- Can be tricky since they may require additional shadow methods
  - Shadows of shadows!
- Some ways to avoid this
  - Force upgrades to finish before the next may be applied
  - Force upgrades to be behavioral subtypes
    - Typical in practice



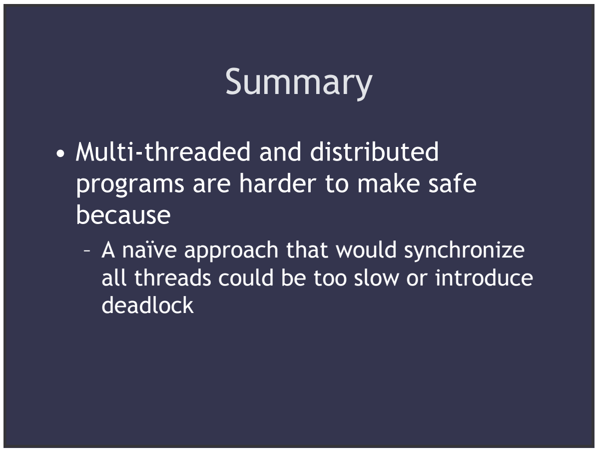
Implementation

- Prototype infrastructure called Upstart
  - Several implementation analogues to the specification described before
  - Supports ways to coordinate upgrades across the system
- Used to upgrade one real application
  - Implemented “Null upgrade” of Dhash on PlanetLab.
  - Demonstrated that the process was low overhead, but did not exercise SOs



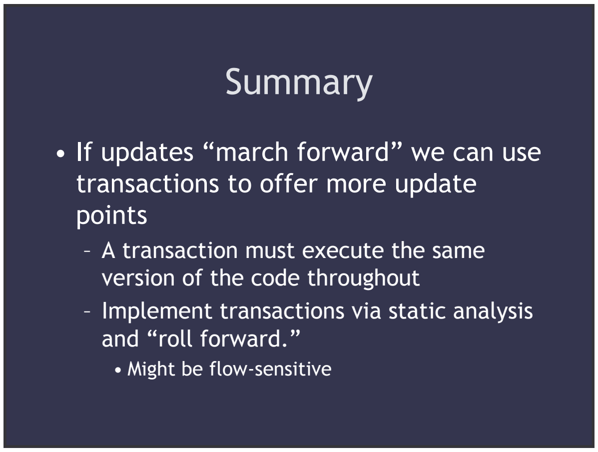
Single-Node Upgrades

- This reasoning framework is abstract enough to apply to single-node upgrades
  - Allow multiple versions of an object to coexist in a program
  - But no way for calls to fail (in general): requires behavioral subtyping to use
- Can make upgrades more available since no sync required



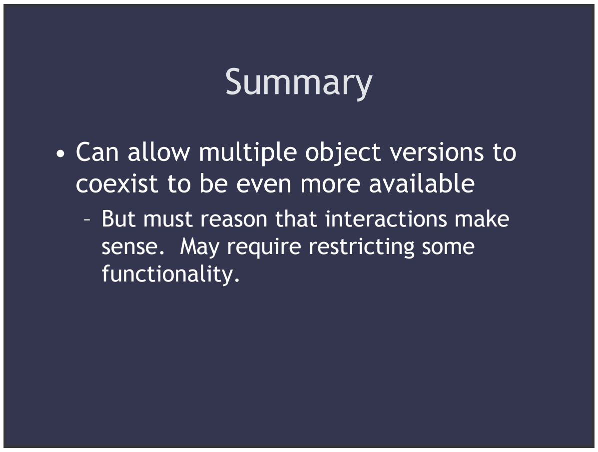
Summary

- Multi-threaded and distributed programs are harder to make safe because
  - A naïve approach that would synchronize all threads could be too slow or introduce deadlock



Summary

- If updates “march forward” we can use transactions to offer more update points
  - A transaction must execute the same version of the code throughout
  - Implement transactions via static analysis and “roll forward.”
    - Might be flow-sensitive



Summary

- Can allow multiple object versions to coexist to be even more available
  - But must reason that interactions make sense. May require restricting some functionality.

Related Work

- Dynamic Software Updating
  - K42 @ IBM
  - Erlang @ Ericsson
  - Various others
- Safety analysis
  - Gupta (TSE `96)
  - Duggan (Acta Inf. `02)
  - Boyapati et al. (OOPSLA `03)
  - CL (POPL `99)

Other Work

- Live Updating of Operating Systems using Virtual Machines (VEE 2006)
  - Uses VM to sync whole system
  - Almost no notion of safety
- OPUS: updating multi-threaded programs (simply) to fix security bugs
  - Only applies to code

For More Information

- Papers
  - POPL 2005 paper on analysis
    - FCS 2005 paper on application to security
  - PLDI 2006 paper for implementation and experience with C
- Compiler and tools available

http://www.cs.umd.edu/projects/dsu/