Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity OS Kernels

Kristis Makris <kristis.makris@asu.edu>
Arizona State University

Kyung Dong Ryu <kryu@us.ibm.com>
IBM T.J. Watson Research Center

Overview

- Motivation
- Dynamic Kernel Updates Categorization
- System Architecture
- Adaptive Function Cloning
- Synchronized Updates
- Applications
- Conclusion

Motivation

- **Dynamic kernel updates are essential**
- **Existing updating methods are inadequate**
- **Two approaches**
  - **Build adaptable OS**
    - Specially crafted (K42, VINO, Synthetix)
    - Require OS and application restructuring
  - **Dynamic code instrumentation**
    - No kernel source modification (KernInst, GILK)
    - Basic block code interposition
    - Currently limited
      - No procedure replacement
      - No autonomous kernel adaptability
      - No safe, complete subsystem update guarantees

Dynamic Updates Categorization (1)

- Updating variable values
  - Update an entry in system call table
  - Update owner (uid) of an inode
    - Needs synchronized update
  - Count number of system calls of a process
    - Needs state tracking

- Updating datatypes
  - Add new fields in Linux PCB for process checkpointing
    - Update all functions that use the old datatype, or
    - Maintain new fields in separate data structure
      - Does not need state transfer

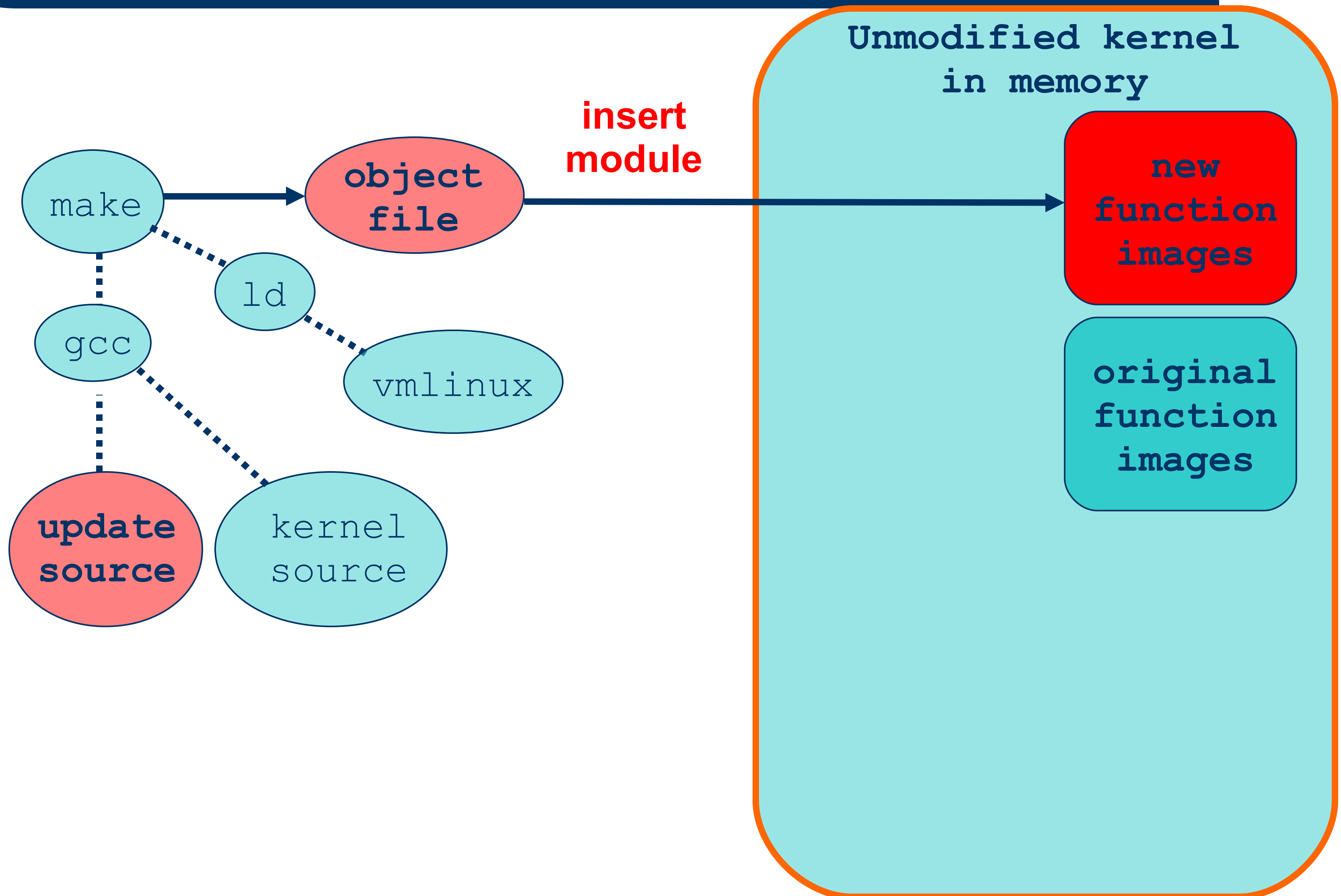
Dynamic Updates Categorization (2)

- Updating single function
  - Correct a defect
- Updating kernel threads
  - Update memory paging subsystem
    - Needs update during infinite loop
- Updating function groups
  - Update `pipefs` subsystem
    - Needs synchronized update

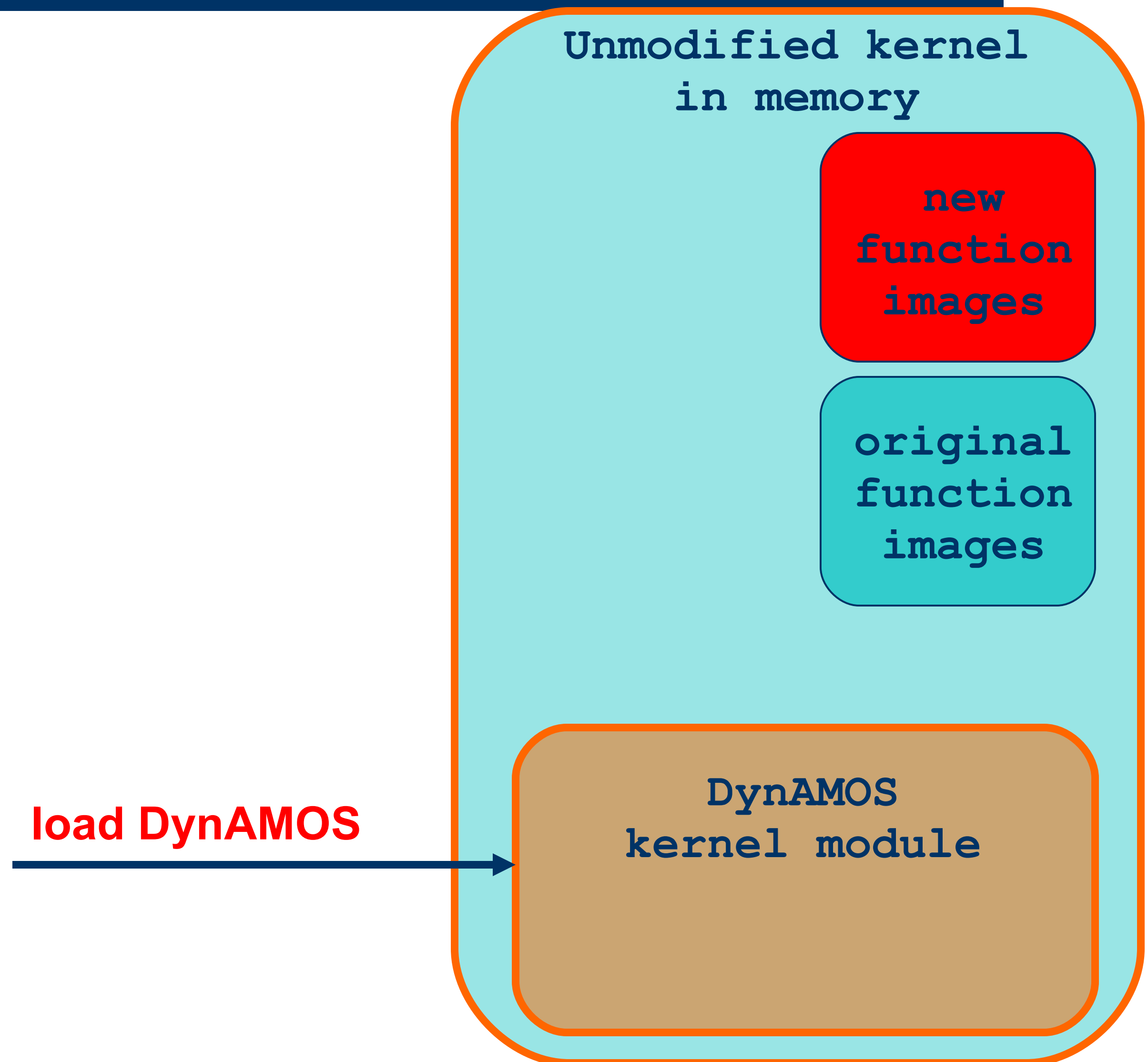
Our Approach

- **DynAMOS**
  - **Prototype for i386 Linux 2.2-2.6**
- **Dynamic code instrumentation**
  - No kernel source modification or reboot
  - Procedure replacement
- **Adaptive updates**
  - Concurrent execution of multiple versions
  - State tracking
  - Autonomous kernel adaptability
- **Safe updates of complete subsystems**
  - Quiescence detection
  - Update synchronization (non-quiescent subsystems)
  - Datatype updates
  - State transfer

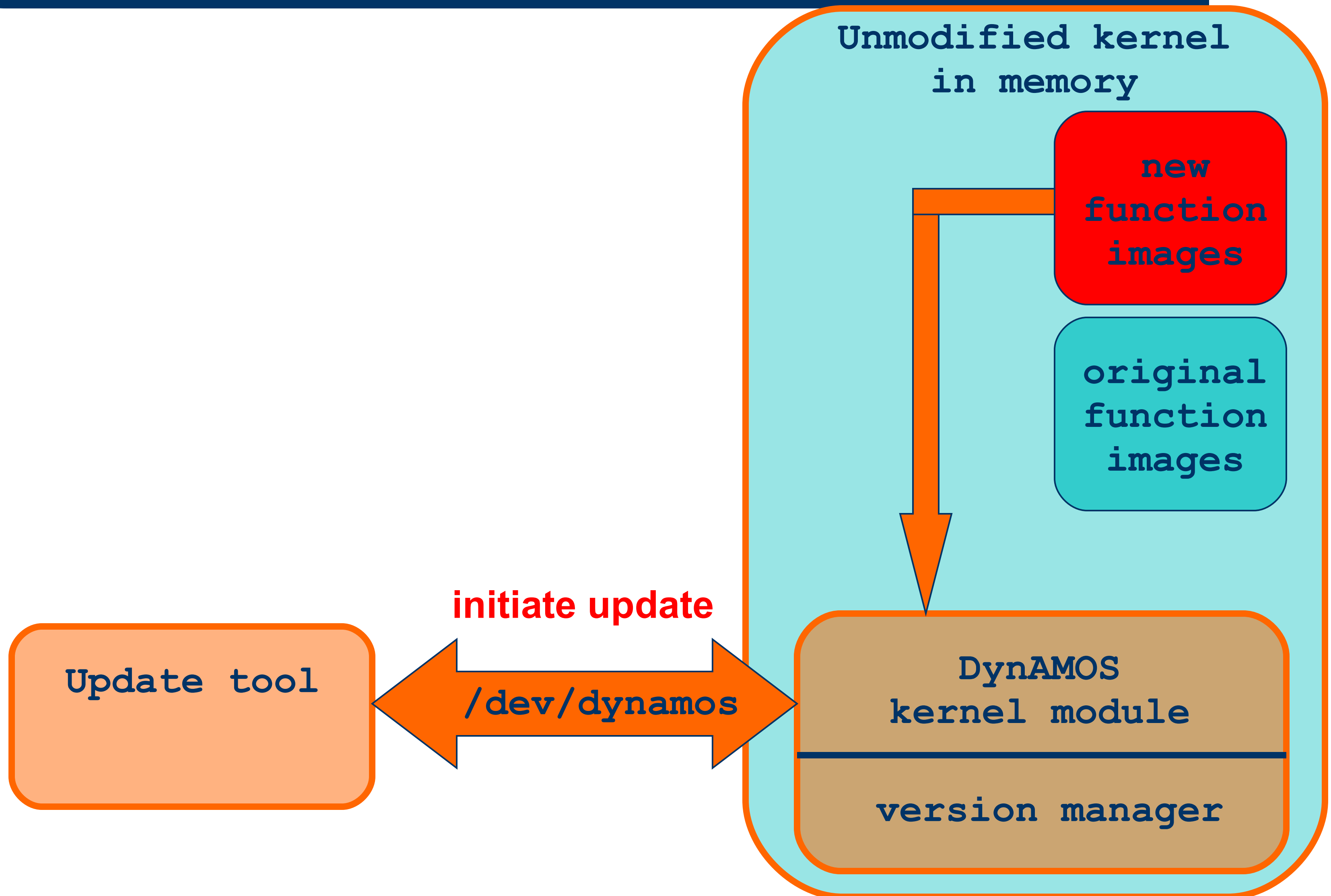
DynAMOS System Architecture



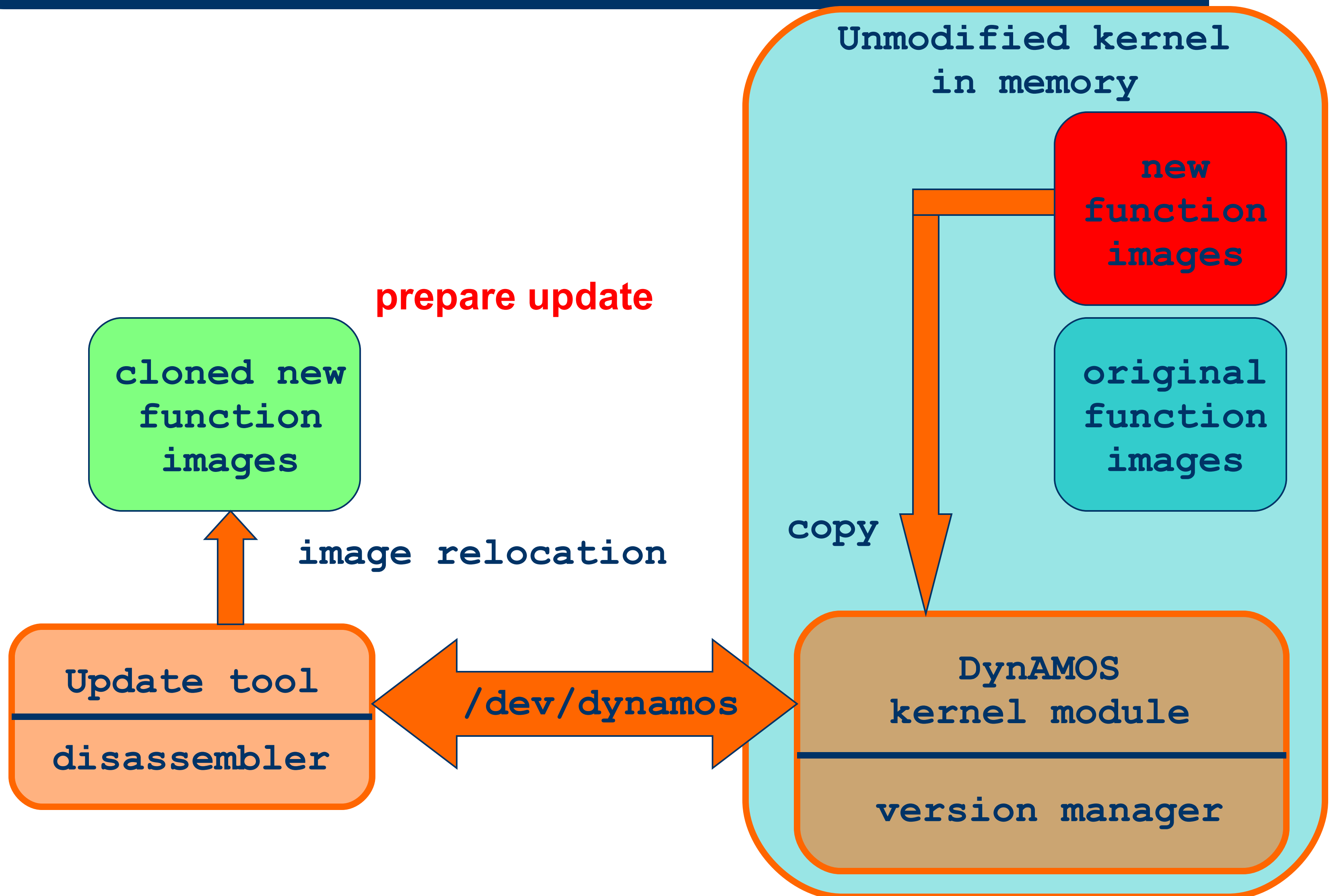
DynAMOS System Architecture



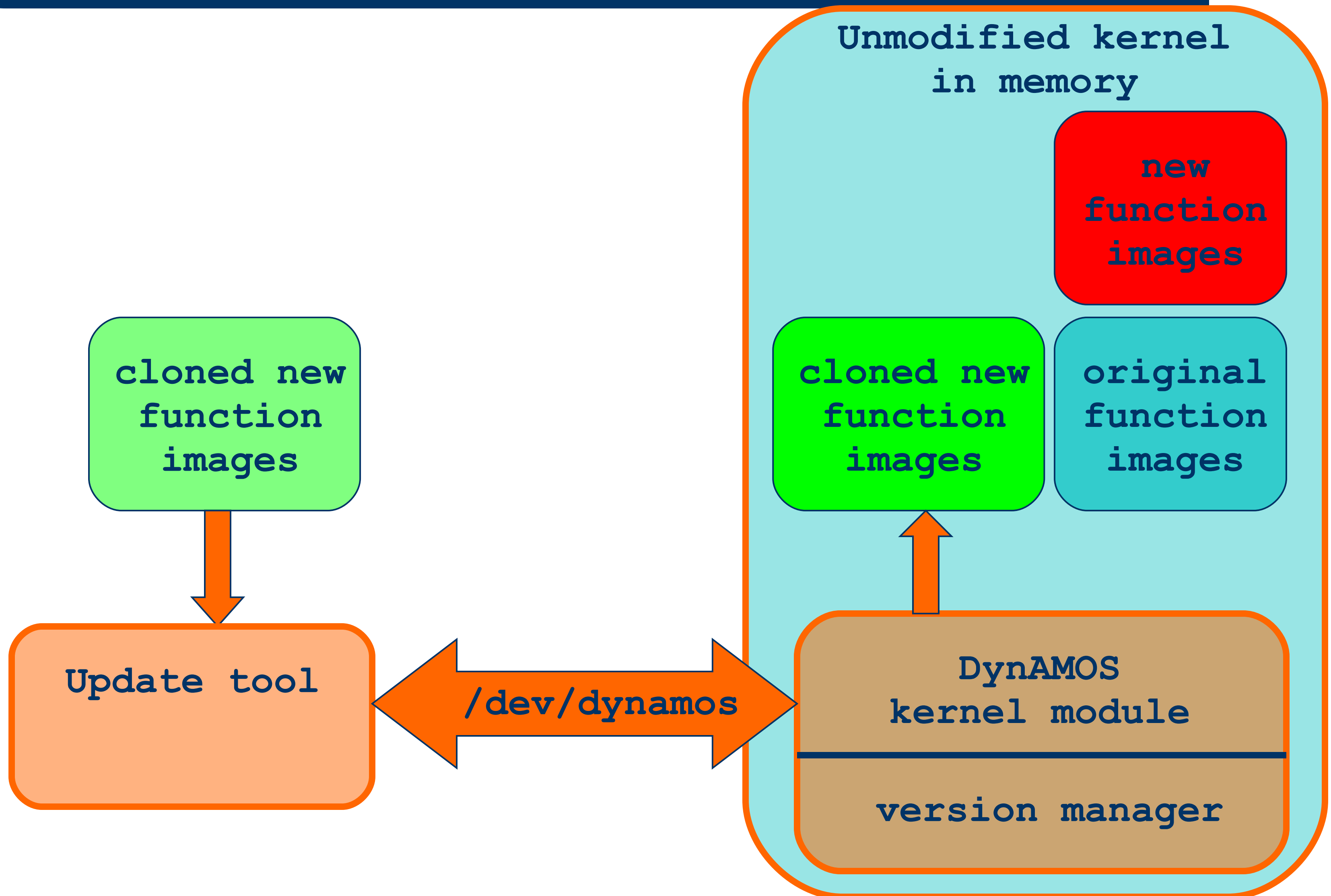
DynAMOS System Architecture



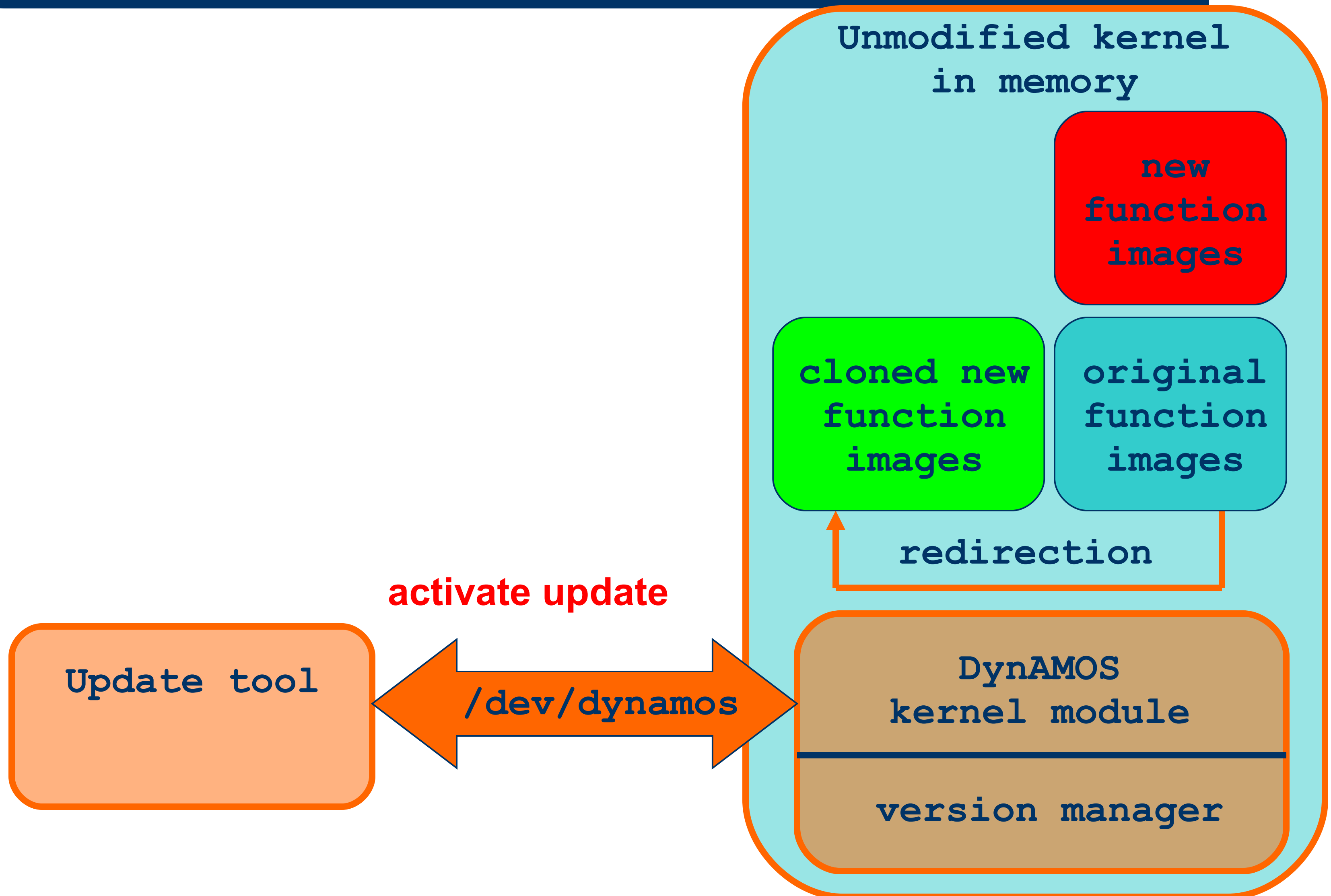
DynAMOS System Architecture



DynAMOS System Architecture



DynAMOS System Architecture



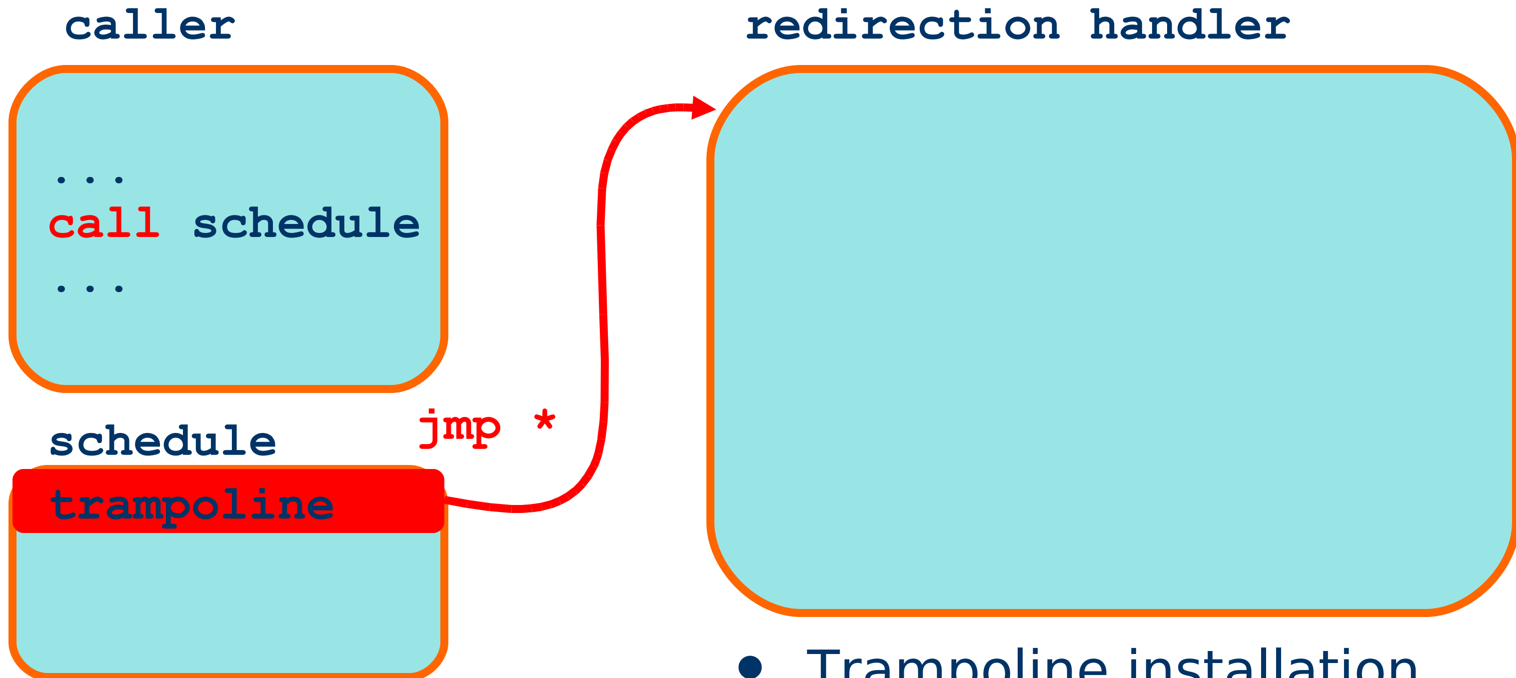
Execution Flow Redirection

```
caller

...
call schedule
...

schedule
```

- Apply Linger-Longer scheduler
  - Unobtrusive fine-grain cycle stealing
  - Implemented in `schedule_LL` as a scheduling policy

step 1

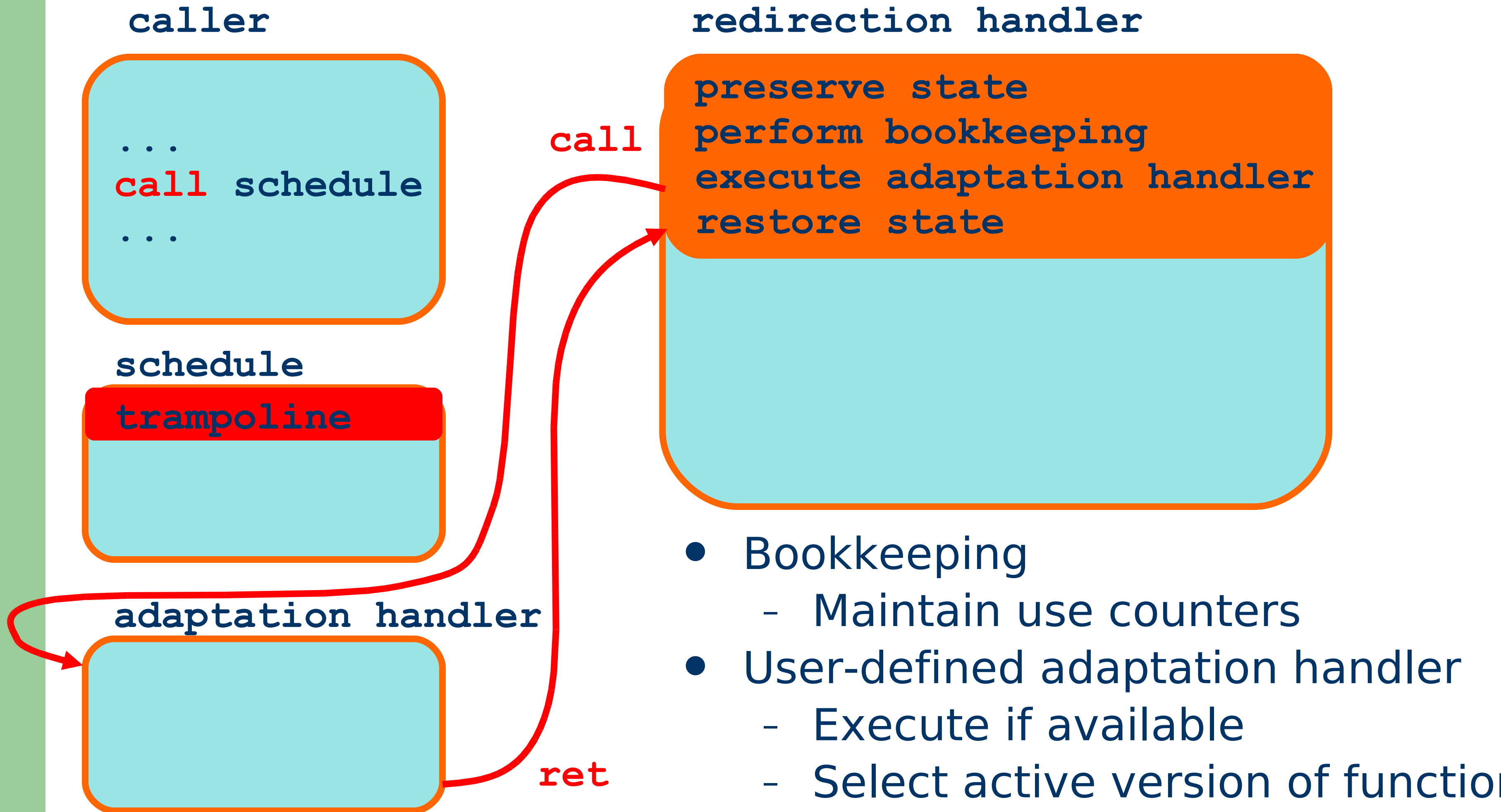
Execution Flow Redirection



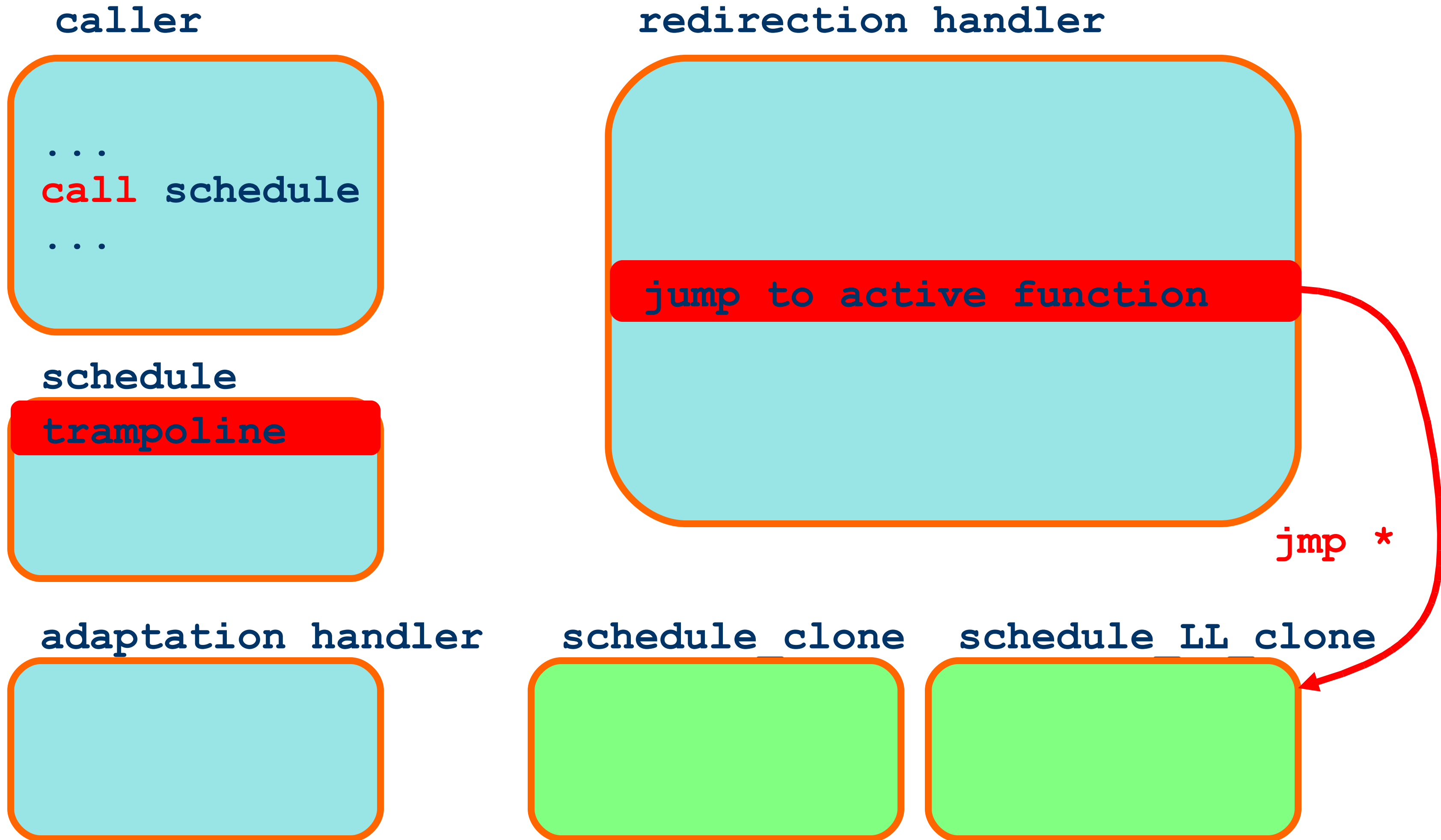
- Trampoline installation
  - Disable processor interrupts
  - Flush I-cache
- Indirect jump
  - Don’t modify page permissions

step 2

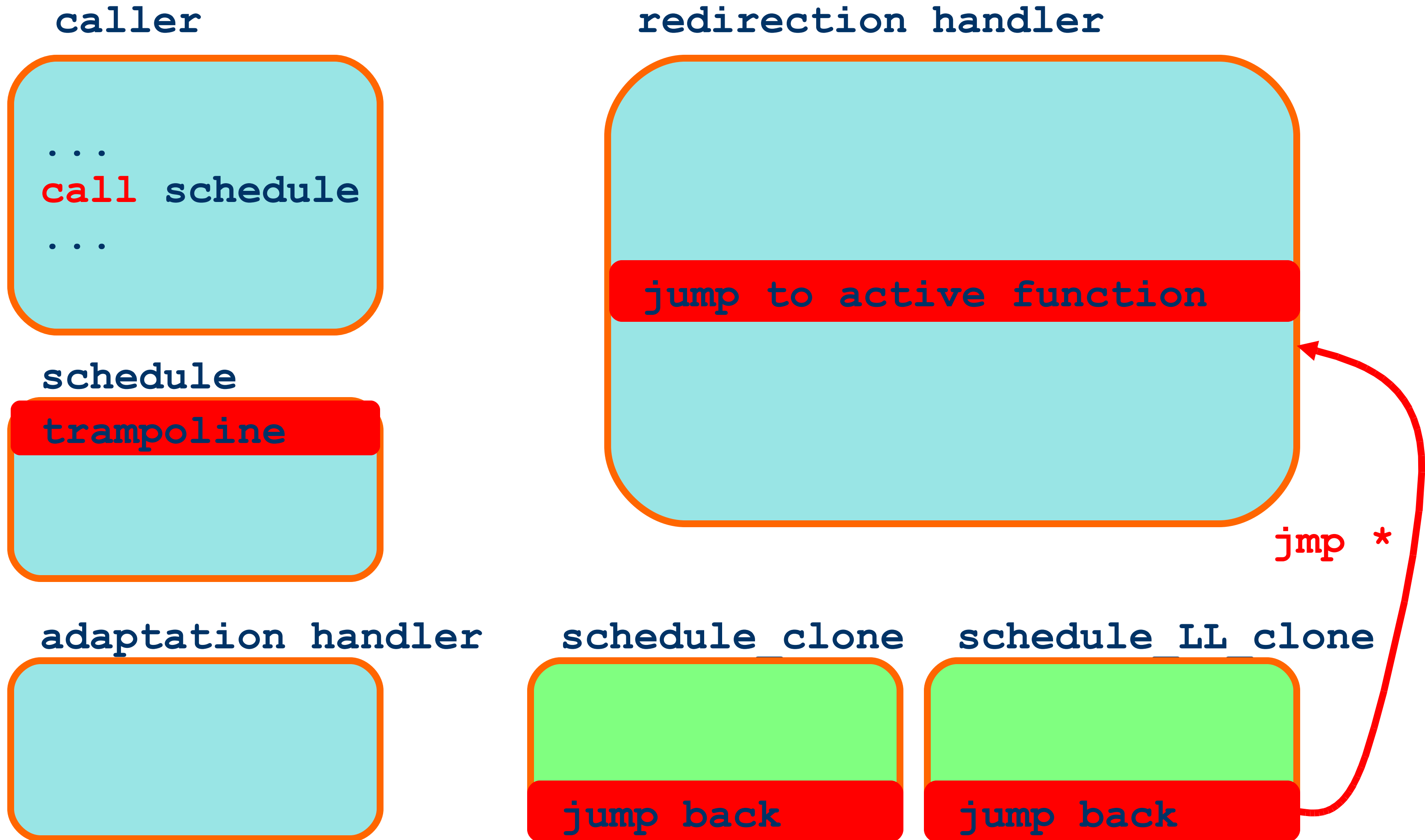
Execution Flow Redirection



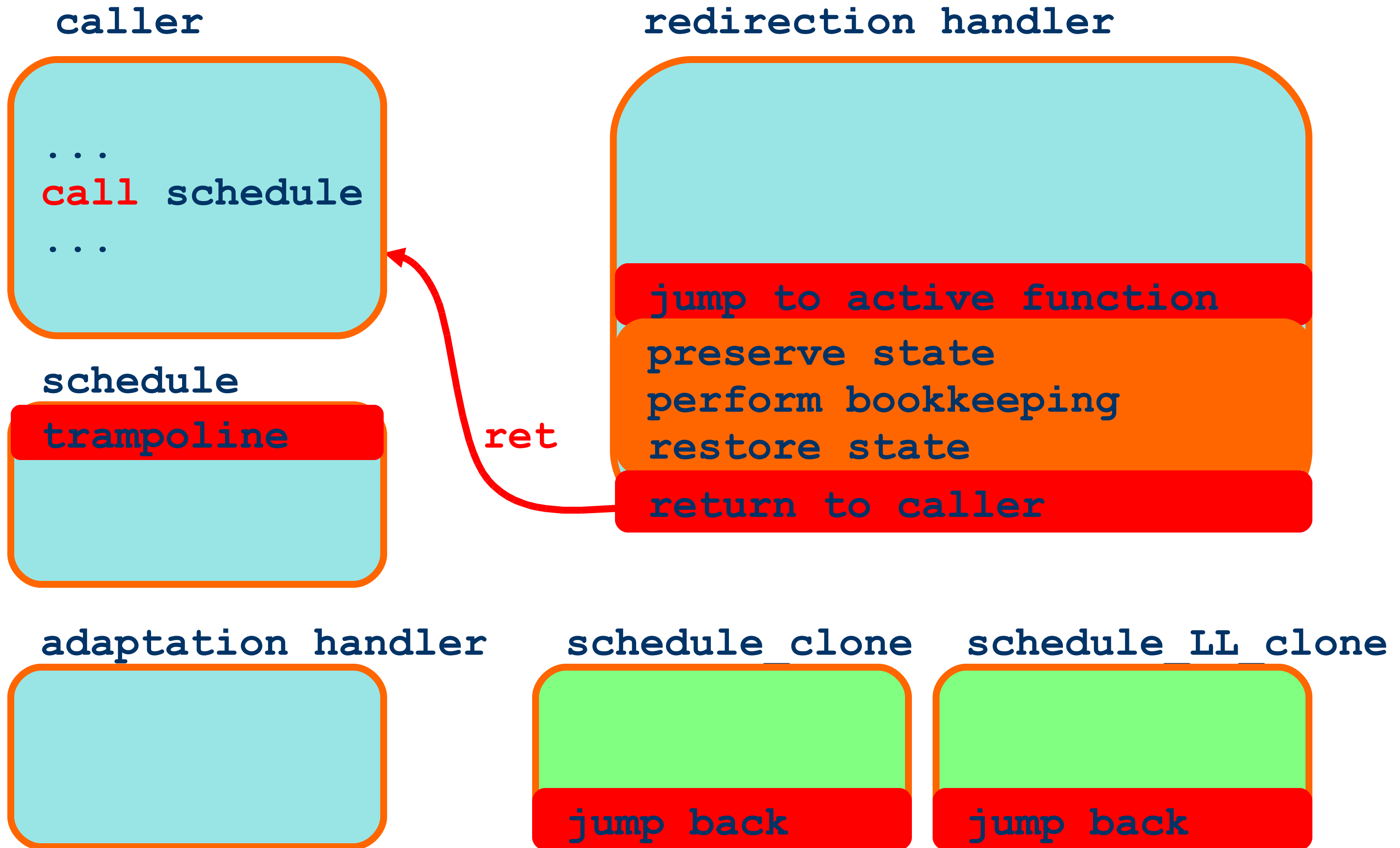
Execution Flow Redirection



Execution Flow Redirection



Execution Flow Redirection



Adaptive Function Cloning Benefits

- No processor state saved on stack
  - Function arguments accessed directly
- Autonomous kernel determination of update timeliness
  - Using adaptation handler
- Function-level updates
  - Basic blocks can be bypassed (no control-flow graph needed)
  - Function modifications developed in original source language

Function Relocation Issues

- **Replace `ret` (1-byte) with `jmp *` (6-byte) back to handler**
  - Adjust inbound (`jmp`) and outbound (`call`) relative offsets

- **Safely detect**
  - Backward branches: `jmp` to code overwritten by trampoline
  - Outbound branches: `jmp` to code outside function image
  - Indirect outbound branches: `jmp *` from indirection table
  - Data-in-code
    - Need user verification
  - Multiple entry-points: e.g. produced by Intel C Compiler

Performance

- **Small memory footprint (42k)**
- **Indirect addressing (`jmp *`) hurts branch prediction**
  - Can use direct addressing (`jmp`)
  - Overhead not correlated to path length
  - Mostly 1-8%

Function	Size (bytes)	Average execution time (μs)	Overhead (%)
do_fork	1811	26.622	1.71
sys_brk	247	0.295	43.48
do_execve	487	79.631	0.83
sys_open	127	5.759	8.04
sys_read	235	3.537	1.67
sys_write	235	9.407	2.00
do_page_fault	1127	2.092	5.82
sys_kill	79	0.865	43.92

Quiescence Detection

- Needed to
  - Atomically update function groups
    - e.g. Count number of processes using a filesystem
  - Safely reverse updates

- Implemented by
  - Usage counters
    - On entry and exit
  - Stack walk-through
    - For non-returning calls (`do_exit` in Linux; no `ret` instruction)
    - Examine stack and program counter of all processes
    - Default kernel compilation (works without frame pointers)

Non-quiescent Subsystems

reader and writer are synchronized with each other

```
pipe_read()
{
    ...
    acquire Sem
    while (buffer_empty) {
        ...
        release Sem
L1:     sleep
        acquire Sem
    }
    read from data buffer
    release Sem
    return
}
```

wait for new data in buffer

```
pipe_write()
{
    ...
    acquire Sem
    while (buffer_full) {
        ...
        release Sem
L2:     sleep
        acquire Sem
    }
    write in data buffer
    release Sem
    return
}
```

wait for more room in buffer

Adaptively enlarge `pipefs` 4k copy buffer during large data transfers

Non-quiescent Subsystems

non-quiescent; sleeping

```
pipe_read()
{
   ...
   acquire Sem
   while (buffer_empty) {
       ...
       release Sem
L1:    sleep
       acquire Sem
   }
   read from data buffer
   release Sem
   return
}
```

quiescent

```
pipe_write()
{
   ...
   acquire Sem
   while (buffer_full) {
       ...
       release Sem
L2:    sleep
       acquire Sem
   }
   write in data buffer
   release Sem
   return
}
```

subsystem may never quiesce
cannot update atomically

Synchronized update of `pipefs`

```
pipe_read() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 4k_buffer
   release Sem
   return
}
```

```
pipe_read_v3() {
   acquire Sem
   while (1mb_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

Phase 1

Synchronized update of `pipefs`

```
pipe_read() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 4k_buffer
   release Sem
   return
}
```

```
pipe_read_v2() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
       if (must_update) {
           phase = 3
           STATE TRANSFER
           goto new
       }
   }
   read data from 4k_buffer
   release Sem
   return

   new:

}
```

```
pipe_read_v3() {
   acquire Sem
   while (1mb_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

Semantically equivalent version at sou

Wait for pipe_read to become inactive

Phase 2

Synchronized update of `pipefs`

```
pipe_read() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 4k_buffer
   release Sem
   return
}
```

```
pipe_read_v2() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
       if (must_update) {
           phase = 3
           STATE TRANSFER
           goto new
       }
   }
   read data from 4k_buffer
   release Sem
   return
```

```
while (1mb_buffer_empty) {
    release Sem
    sleep
    acquire Sem
new:
}
read data from 1mb_buffer
release Sem
return
}
```

```
pipe_read_v3() {
   acquire Sem
   while (1mb_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

Inline updated version

Phase 2

Synchronized update of `pipefs`

```
pipe_read() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 4k_buffer
   release Sem
   return
}
```

```
pipe_read_v2() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
       if (must_update) {
           phase = 3
           STATE TRANSFER
           goto new
       }
   }
   read data from 4k_buffer
   release Sem
   return

   while (1mb_buffer_empty) {
       release Sem
       sleep
       acquire Sem
   new:
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

```
pipe_read_v3() {
   acquire Sem
   while (1mb_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

Phase 3

Synchronized update of `pipefs`

Adaptive update

30-90% improvement in Linux 2.6

3.2% overhead when not adapting

```
pipe_read_adaptation_handler() {
   if (phase == 3)
      activate pipe_read_v3
   else
      activate pipe_read_v2

   if (this process read
       more than 64k)
      must_update = 1
}
```

Phase 3

```
pipe_read_v2() {
   acquire Sem
   while (4k_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
       if (must_update) {
           phase = 3
           STATE TRANSFER
           goto new
       }
   }
   read data from 4k_buffer
   release Sem
   return

   while (1mb_buffer_empty) {
       release Sem
       sleep
       acquire Sem
   new:
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

```
pipe_read_v3() {
   acquire Sem
   while (1mb_buffer_empty) {
       release Sem
L1:    sleep
       acquire Sem
   }
   read data from 1mb_buffer
   release Sem
   return
}
```

Sleep in original version
Awake in new version

Multi-phase approach

Adaptive Memory Paging For Efficient Gang Scheduling

- Kernel thread update (`kswapd`), Linux 2.2
  - Infinite loop
  - Awaken by other subsystems
  - Goes back to sleep
    - e.g. calls `interruptible_sleep_on` in Linux
- To update
  - Activate `interruptible_sleep_on_v2`
    - Save state, exit
    - Start new version of kernel thread, restore state

Kernel-Assisted Process Checkpointing

- **Datatype update for EPCKPT in Linux 2.4**
  - Compact datatypes in commodity kernel. No extra room
    - `struct task_struct`: semaphores, pipes, memory mapped files
    - `struct file`: checkpoint filename
- **Shadow data structures**
  - Instantiation (`do_fork, sys_open`): map memory address of original variable to shadow using hash table
  - Removal (`do_exit, fput`): free shadow too
  - Already instantiated variables
    - Shadow missing: idempotent use of new fields
  - Update only functions that use new fields
    - No state transfer needed

Related Work

- K42
  - Specially designed with hot-swappable capabilities
  - Guarantees quiescence
- Ginseng
  - User-level software updates; requires recompilation
- KernInst, GILK, Detours, ATOM, EEL
  - Do not facilitate adaptive execution
  - Do not safely replace complete subsystems

On-going and Future Work

- Automatically produce updates given a patch
  - Apply MOSIX, Superpages: parallel applications
  - Apply Nooks: OS reliability
  - Upgrade Linux kernel
- Multiprocessor support
  - Safely install trampoline: freeze other processors using single-byte trap instruction (`ud2`)
- Kernel module port
  - FreeBSD, OpenSolaris

Conclusion

- **Dynamic Kernel Updates**
  - Dynamic code instrumentation
  - Commodity operating system (prototype for i386 Linux 2.2-2.6)
- **Adaptive function cloning**
  - Concurrent execution of multiple function versions
- **Safe updates of non-quiescent subsystems**
  - Scheduler, kernel threads, synchronized updates
- **Datatype updates**
- **Demonstrated updates**
  - Synchronized pipefs adaptation, process checkpointing, adaptive memory paging for efficient gang-scheduling, unobtrusive fine-grain cycle stealing, public security fixes
- **Small memory footprint (42k), 1-8% overhead**

Björn's questions

- How to handle false positives produced by “stack walk-through” approach?
- Datatype updates: is it possible to add new fields in the middle of a struct or only at the end?
- I didn't understand why they need indirect addressing in the trampoline.