Light-Weight Contexts:

An OS Abstraction for Safety and Performance

James Litton

Anjo Vahldiek-Oberwagner
Deepak Garg
Eslam Elnikety
Bobby Bhattacharjee
Peter Druschel

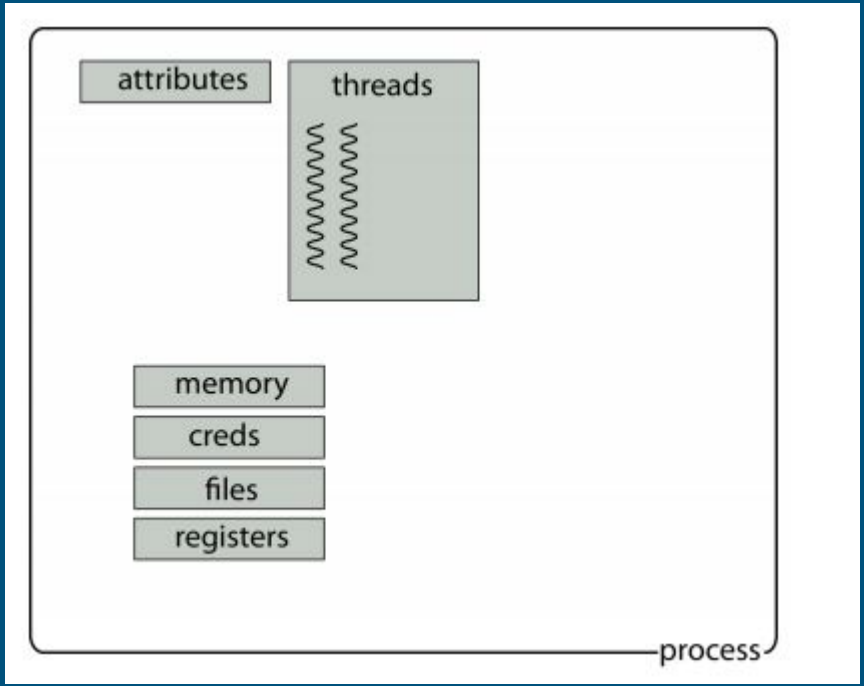
Aseem Saxena
Shubhani

Introduction

Process --

Unit of

- Isolation
- privilege separation
- program state

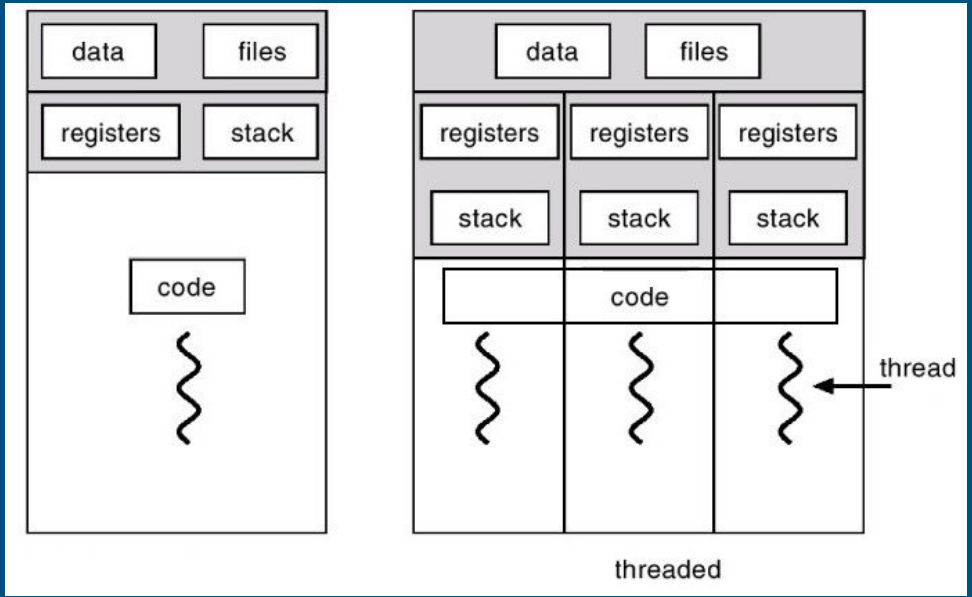


Introduction

Threads separate the unit of execution from a process

But do not provide isolation and privilege separation

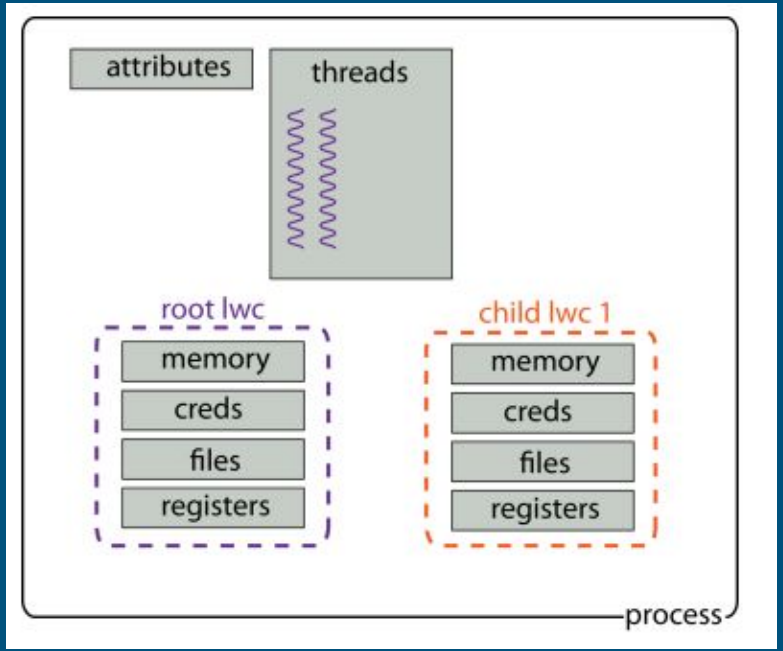
Process vs Thread



Introduction

Light Weight Context (lwC)

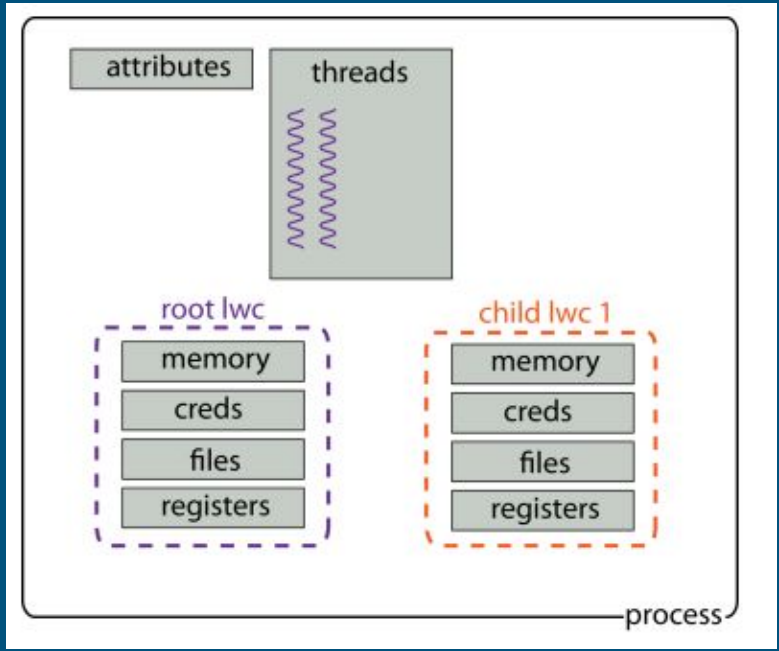
- OS abstraction - that provides independent units of protection, privilege, and execution state within a process.
- A process may contain multiple lwCs, each with their own virtual memory mappings, file descriptor bindings, and credentials.



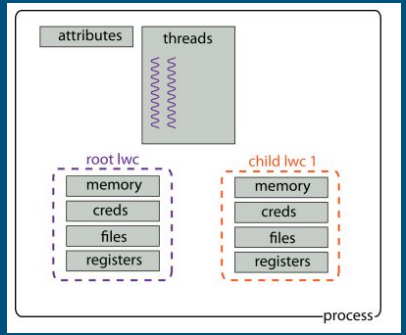
Introduction

lwCs enable a range of new in-process capabilities

- fast roll-back
- protection rings (by credential restriction)
- session isolation
- protected compartments
- Reference Monitors



Design -- Creating lwCs

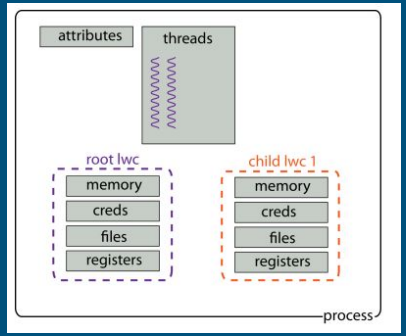


- lwCs are named using file descriptors.
- Each process starts with one root lwC, which has a well-known file descriptor number.
- lwCreate semantics similar to fork

lwCreate

```
new,caller,args ← lwCreate(spec, options)
```

Design -- Creating lwCs



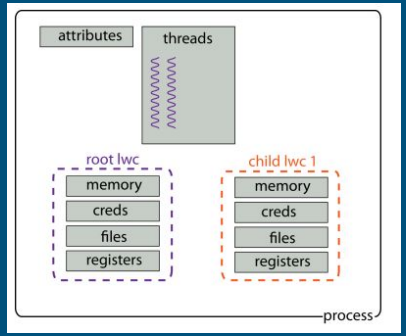
Arguments

- child lwC receives a copy-on-write snapshot of all its parent's resources by default
- resource-spec : array of C unions
- each array element specifies either a range of FDs, VM addresses, or credentials.
- For each range, one of the following sharing options can be specified:
  - LWC_COW
  - LWC_SHARED
  - LWC_UNMAP
- Options : systrap etc.

lwCreate

```
new,caller,args ← lwCreate(spec, options)
```

Design -- Creating lwCs



Return Values

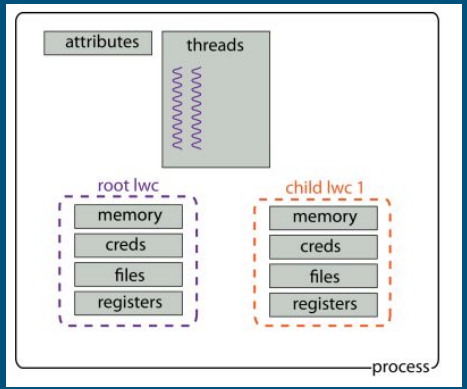
- New
  - Parent lwC's FD in child lwC
  - Child lwC's FD in parent lwC
- Caller
  - -1 in parent
  - Caller lwC's FD in child lwC
- args
  - Arguments passed while switching to child lwC

lwCreate

```
new,caller,args ← lwCreate(spec, options)
```

Design -- lwC vs Thread

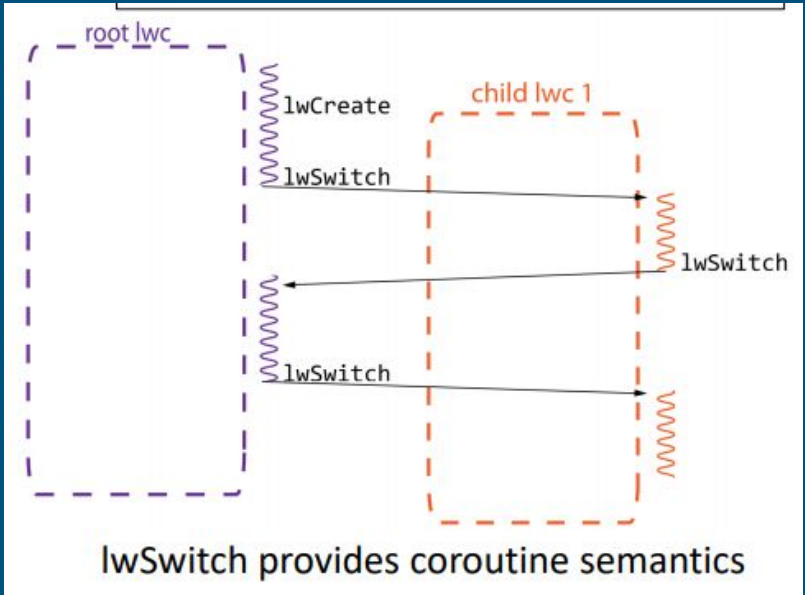
- Threads and lwCs are independent.
- Within a process, a thread executes within one lwC at a time and can switch between lwCs.
- lwSwitch retains the state of the calling thread in the present lwC.



Design -- Switching between lwCs

lwSwitch

```
caller,args ← lwSwitch(target, args)
```



switches the calling thread to the lwC with descriptor target, passing args as parameters.

Returns from a lwSwitch and lwCreate, any signal handlers that were installed are the only possible entry points into a lwC.

Usecase 1 -- Snapshot and rollback

Algorithm 1 Snapshot and rollback

```
1:  function SNAPSHOT()
2:      new,caller,arg = lwCreate(default_spec, ...)
3:      if caller = -1 then                          ▷ parent
4:          return new
5:      else
6:          close(caller)
7:          return snapshot()
8:  function ROLLBACK(snap)                          ▷ never returns
9:      lwSwitch(snap, 0)
10: function MAIN()
11:     ...                                          ▷ initialize state
12:     snap = snapshot()
13:     ...                                          ▷ serve request
14:     rollback(snap)
            ▷ kills current lwC, continues at line 12 in snap
```

As compared to a setup where the process manually cleans up request-specific state after each request, the snapshot and rollback can improve performance by efficiently discarding the request-specific state with a single call, and also improves security by isolating sequential requests served by the same task from each other.

Usecase 2 -- Isolating sessions in an event-driven server

Algorithm 2 Event-driven server with session isolation

```
1:  function SERVE_REQUEST(retlwc, client)
2:    loop
3:      if would_block(client) then
4:        lwSwitch(retlwc, 0);
5:      else if finished(client) then
6:        lwSwitch(retlwc, 1);
7:      else
8:        serve(client)
9:  function MAIN
10:   descriptors = { accept_ descriptor }
11:   file2lwc_map = { accept_descriptor => root }
12:   loop
13:     next = descriptors.ready()
14:     if next = accept_descriptor then
15:       fd = accept(next)
16:       descriptors.insert(fd)
17:       specs = { ... }        ▷ Share fd descriptor only
18:       new,caller,arg = lwCreate(specs, ...)
19:       if caller = -1 then              ▷ context created
20:         file2lwc_map[fd] = new
21:       else
22:         serve_request(root, fd)
23:     else
24:       lwc = file2lwc_map[next]
25:       from, done = lwSwitch(lwc, ...)
26:       if done = 1 then
27:         close(next);close(from)
28:         descriptors.remove(next)
29:         file2lwc_map.unset(next)
```

Since all worker lwCs obtain a private copy of the root's state, no worker sees session-specific state of other workers. This isolates the sessions from each other.

Design -- Resource access APIs

Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

Design -- Dynamic Resources

Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

A lwC may dynamically map/overlay resources from another lwC into its address space.

Arguments

- lwc - FD of Target lwc for resource overlaying
- Spec - Regions of a given resource type (FD or VM) that are to be overlayed, and whether the specified region should be copied or shared.

Caller should hold access capabilities for the requested resources.

Design -- Access Capabilities/Restrictions

Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

- Each lwC holds a descriptor with a universal access capability for itself.
- Parent receives a descriptor with a universal access capability for the child.
- Parent lwC may grant a child lwC access capabilities for the parent lwC selectively by LWC_MAY_ACCESS in the resource-spec passed to the lwCreate call.
- lwRestrict, restricts the set of resources that may be overlayed or accessed by any context that holds the lwC descriptor l

Usecase 3 -- Sensitive data isolation

Isolation of a private signature key that is available to a signing function, but kept hidden from the rest of the network-facing program.

The child lwC is granted the privilege to overlay any part of the parent's virtual memory by using MAY_OVERLAY flag in lwCreate

Parent lwC revokes its privilege to overlay any part of the child lwC's memory using lwRestrict

Algorithm 3 Sensitive Data Isolation

```
1:  function SIGN(key, data, out_buffer)
2:  function SIGN_SSTUB(caller,arg)
3:    loop
4:      lwOverlay(caller,{VM,arg,sizeof(arg),SHARE})
5:      sign(privkey, arg.in, arg.out)
6:      lwOverlay(caller,{VM,arg,sizeof(arg),UNMAP})
7:      caller,arg = lwSwitch(caller, 0)
8:  function SIGN_CSTUB(buf)
9:    caller,res = lwSwitch(child, buf)
10: function MAIN
11:   ...                          ▷ initialization, load privkey
12:   child,caller,arg =
13:   lwCreate({VM,0,MAX,MAY_OVERLAY}, 0)
14:   if caller != -1 then
15:     sign_sstub(caller,arg)
16:   privkey = 0                                  ▷ erase key
17:   lwRestrict(child, {VM,0,MAX,NO_ACCESS})
18:   loop
19:     ...
20:     sign_cstub(buf)
21:     ...
```

Design -- System call emulation

Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

- Child lwC created with LWC_SYSTRAP in options flag
- If a thread in Child lwC invokes a system call for which it
- does not hold a capability, the thread is switched to its parent lwC by sandbox.
- The parent can choose to :
  - decline the syscall and return an error to the child,
  - perform a syscall on behalf of the child(with different argument also) using the lwSyscall.

Design -- System call emulation

Resource Access		
status	←	lwRestrict(lwc, spec)
status	←	lwOverlay(lwc, spec)
status	←	lwSyscall(target, mask, syscall, args)

Arguments:

- Target : Child lwC for which syscall is being made
- Mask parameter allows the caller to modify the behavior by specifying aspects of its own context that are to be put in place for the duration of the system call
- Syscall args can be modified by parent from as provided by child lwC

Usecase 4 -- Protected reference monitor

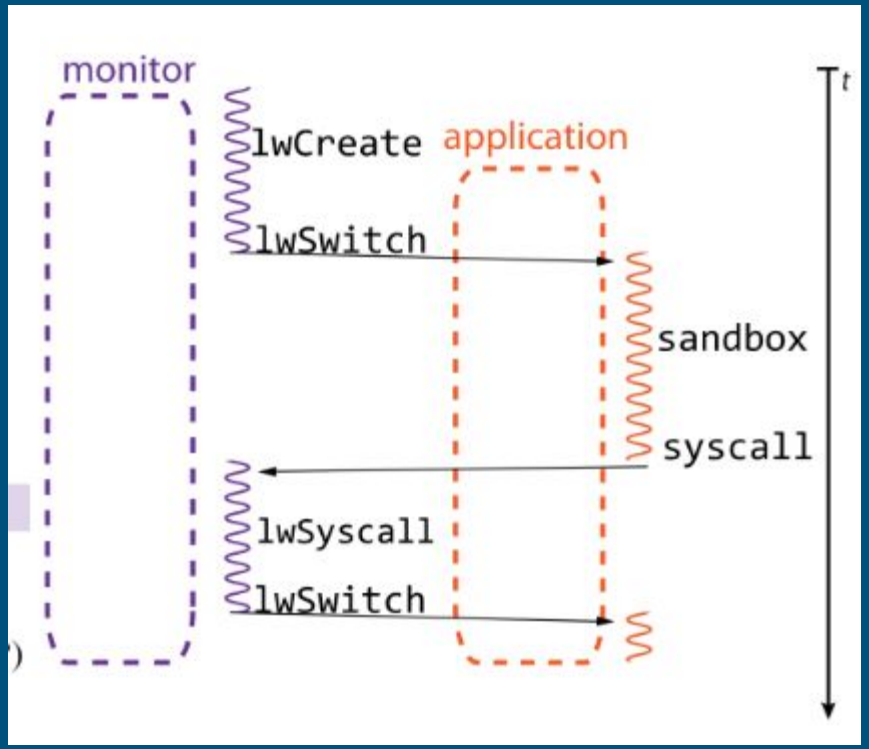
Allows a parent lwC to intercept any subset of system calls made by its child and monitor those calls.

A more interesting monitor could inspect the system call arguments or other parts of the child's state by overlaying in the appropriate regions.

Algorithm 4 Reference Monitor

```
1:  function MONITOR(child)
2:      _,call = lwSwitch(child, NULL)
3:      loop
4:          if is_allowed(call) then
5:              spec = { type = CRED, SANDBOX }
6:              rv = lwSyscall(child, spec,
                                call.num, call.params)
7:              out.err,out.rv = errno, rv;
8:          else
9:              out.err,out.rv = EPERM, -1;
10:         _,call = lwSwitch(child, out)
11: function MAIN
12:     specs = { ... } ▷ Share (COW) all but private data
13:     child,c,_ = lwCreate(specs, LWC_SYSTRAP)
14:     if c = -1 then              ▷ parent becomes refmon
15:         monitor(child)                  ▷ Never returns
16:     privdrop() && run()             ▷ Child starts here
```

Usecase 4 -- Protected reference monitor



Algorithm 4 Reference Monitor

```
1:  function MONITOR(child)
2:      _,call = lwSwitch(child, NULL)
3:      loop
4:          if is_allowed(call) then
5:              spec = { type = CRED, SANDBOX }
6:              rv = lwSyscall(child, spec,
                               call.num, call.params)
7:              out.err,out.rv = errno, rv;
8:          else
9:              out.err,out.rv = EPERM, -1;
10:         _,call = lwSwitch(child, out)
11: function MAIN
12:     specs = { ... } ▷ Share (COW) all but private data
13:     child,c,_ = lwCreate(specs, LWC_SYSTRAP)
14:     if c = -1 then              ▷ parent becomes refmon
15:         monitor(child)                  ▷ Never returns
16:     privdrop() && run()             ▷ Child starts here
```

lwC Signal handling Semantics

- Attributable signals(which can be attributed to the execution of a particular instruction such as SIGSEGV or SIGFPE), are delivered to the lwC that caused the signal immediately.
- Non-attributable signals(SIGKILL or SIGUSR1), are delivered to the root lwC and any lwCs in the process that were created with the LWC_SHARESIGNALS option.
- A non-attributable signal is delivered to a lwC upon the next switch to the lwC.

lwC System Call Semantics

- During fork system call, all lwC s in the calling process are duplicated in the child process.
- Any memory region with MAP_SHARED flag in some lwCs of the calling process are shared with the corresponding lwCs in the new child process, within and across the two processes.
- Any memory regions that are shared among lwCs in the parent process using the LWC_SHARED option in lwCreate are shared among the corresponding lwCs within the child process only.
- Exit system call in any lwC of a process terminates the entire process

lwC Abstraction : Threat model & Security Properties

- When a lwC is created, its parent has universal privileges on the lwC. Consequently, the security of a lwC assumes that its parent (and, by transitivity, all its ancestors) cannot be hijacked to abuse these privileges.

- In practice, the parent should drop all unnecessary privileges on the child immediately after the child is created, so this assumption is needed only with respect to the remaining privileges.

lwC Abstraction : Threat model & Security Properties

- A lwC cannot attain privileges that exceed those of its process, and the confidentiality and integrity properties of any lwC cannot be weaker than those of its process.
- The properties of the root lwC are those of the process.
- The root lwC should run a high-assurance component, near to the root, to minimize its dependencies.
- More complex, less stable, network or user-facing components should be encapsulated in de-privileged lwCs at the leaves of a process's lwC tree and should execute with the least privileges required.

lwC Abstraction

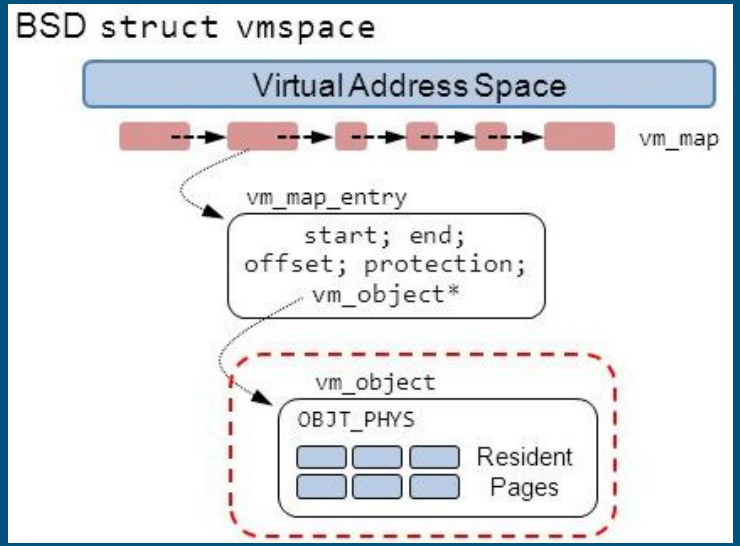
- Why lighter than process Abstraction?
  - Space overhead
  - Time overhead

lwC Implementation

- Implemented lwCs in the **FreeBSD 11.0**
- FreeBSD is an OS for a variety of platforms which focuses on features, speed, and stability.
- FreeBSD is derived from BSD
- lwC implementation required modifications in FreeBSD kernel data structures corresponding to process memory, file tables and credentials.

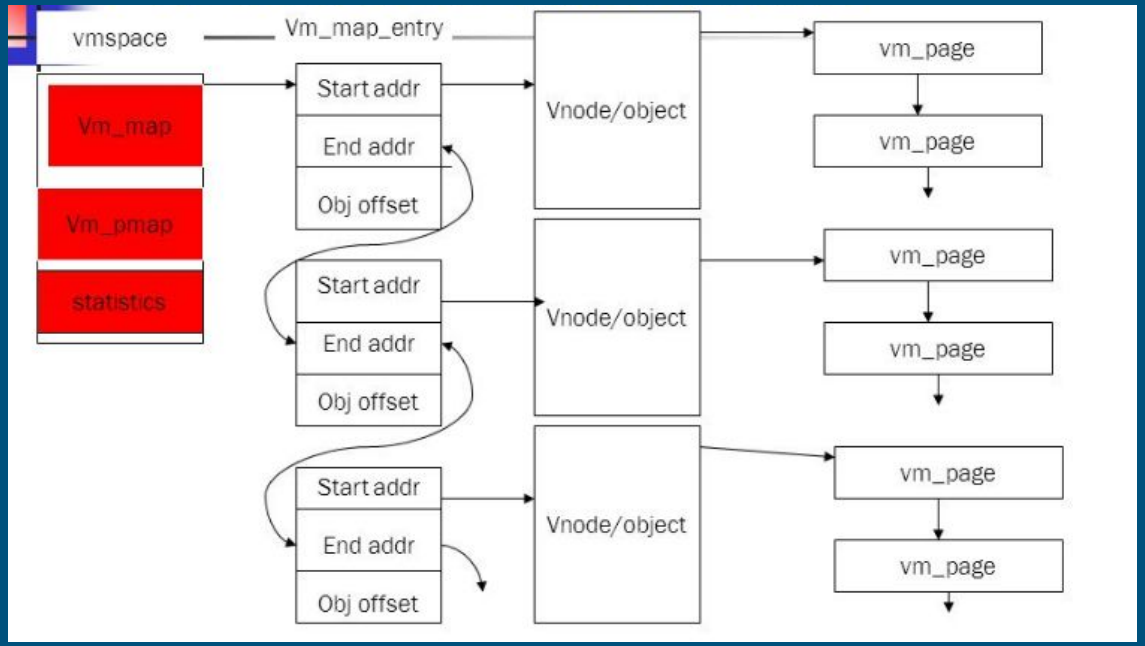
lwC Implementation -- Memory

- The address space of a process is organized under a vmspace structure



lwC Implementation -- Memory

- Per Process Resources



lwC Implementation -- Memory

- The number of memory regions within a process is typically small (few tens).
- lwCreate replicates the vmspace associated with the parent lwC in exactly the same manner as fork.
- LWC_UNMAP during the lwCreate call are not mapped into the new lwC's
- LWC_SHARE are mapped into the lwC as memory that differs from shared memory in only one respect: a subsequent fork will not share this region with its parent.
- Work performed during fork and lwCreate is proportional to the number of vm_map_entry structures.

lwC Implementation -- Memory

- FreeBSD 11.0 supports PCIDs
- Earlier context switch (lwC or process) required a TLB flush.
- Modern processors include a “process context identifier” (PCID)that can be used to distinguish pages that belong to different page tables.
- TLB entries are tagged with the PCID that was active when they were resolved.
- At the time of switch, the kernel sets the CR3 register to a value containing the PCID and the address of the first page directory entry.
- each lwC is assigned a unique PCID.

lwC Implementation -- File Table

- all files, sockets, devices, etc. open in a process are accessible via the process's file table.
- lwCs are also accessed via file-table entries.
- Upon fork, the file table is copied from the parent to the child process.
- In lwCreate all FD are copied into the child lwC file table in the same manner as fork except that any associated FD overlay rights are copied as well.
- For LWC_UNMAP, the descriptors are not copied into the file table.
- Upon lwCreate, if the file table or a lwC descriptor's is not shared by parent, then the child lwC is not able to access the parent's lwCs

lwC Implementation -- Permissions and Overlays

- Lwc interaction using lwSwitch or lwOverlay
- lwC ‘a’ may switch to lwC ‘b’ only if b’s descriptor is present in a’s file table.
- Overlay permissions are more fine-grained(LWC_MAY_OVERLAY flag)
- lwCreate call (p creating c) results in two file descriptors.
  - c and has full overlay rights, and is inserted into p’s file table.
  - p is given to c and only allows overlays on the descriptor as specified by p in the lwCreate call.

Experimental Results

Experiments were performed on

- Dell R410 servers, each with 2x Intel Xeon X5650 2.66GHz 6 core CPUs
- hyper-threading and Speed-Step disabled,
- 48GB main memory,
- OS FreeBSD 11.0
- OpenSSL 1.0.2 for TLS.
- Servers connected via Cisco switches with 1Gbit Ethernet links.
- Each server has a 1TB Seagate ST31000424SS disk formatted under UFS.

Experimental Results

- Microbenchmarks
  - lwCreate
  - lwSwitch
  - Reference monitoring (SYSTRAP)

- Apache (v. 2.4.18)
  - Session isolation
  - Reference monitoring

- Nginx (v. 1.9.15)
  - Session isolation
  - Reference monitoring
- SSL private key isolation

- PHP (v. 7.0.11)
  - Fast startup via snapshots

Experimental Results -- lwC Creation

- Total cost of creating, switching to, and destroying lwCs with resources as COW, within a single process
  - No dirty pages - 87.7 µs
  - COW cost per dirty page - 3.4 µs(approx)

- Cost of maintaining a separate lwC is linearly dependent on the number of unique pages it creates.

- These results can be used to estimate the cost of using lwCs in an application, given an estimate of the rate of lwC creations and switches, and the number of unique pages in each lwC.

Experimental Results -- lwC Switch

lwC	process	k-thread	u-thread
2.01 (0.03)	4.25 (0.86)	4.12 (0.98)	1.71 (0.06)

- Above table compares the time to execute a lwSwitch call compared to context switching between processes (using semaphore), between kernel threads (using semaphore) and user threads.
- lwC switch takes less than half the time of a process or kernel thread switch.
- Kernel thread switch is on par with a process context switch when both use the same form of synchronization.
- The user threads use the getcontext and setcontext calls in FreeBSD 11. In Linux glibc, it is implemented in userspace library(run in 6% of the time required by semaphore-based kernel thread switches)

Experimental Results -- Reference Monitor

- **lwc-mon**
  - When a process starts, the reference monitor gains control first and creates a child lwC, which executes the server application.
  - Child lwC is sandboxed using FreeBSD Capsicum .
  - Child lwC disallowed from using certain system calls, which are instead redirected to the parent lwC using the LWC_SYSTRAP.

Experimental Results -- Reference Monitor

- Evaluated against:
  - **Inline Monitoring** using LD_PRELOAD, provides a lower bound on overhead, but does not provide security.
  - **Process Separation** provides a secure reference monitor in a separate process. But lwC outperform it by a factor of two.

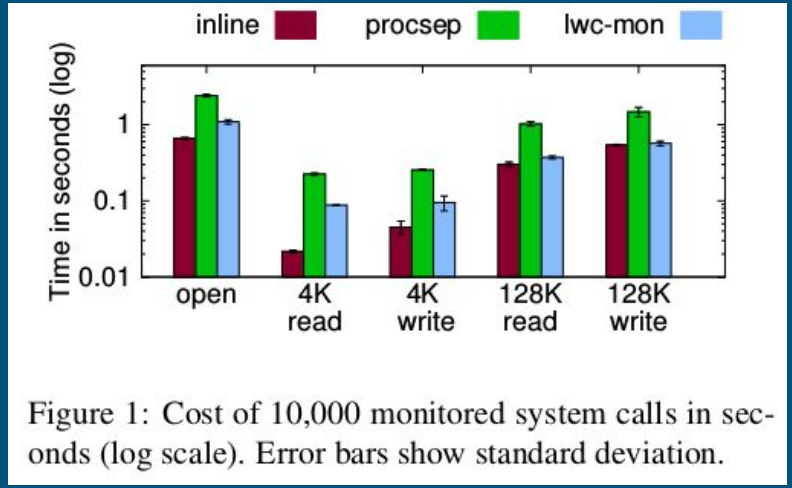


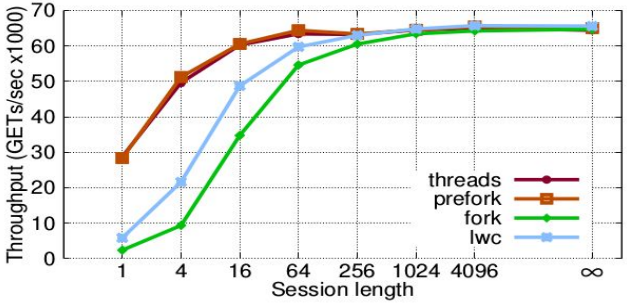
Figure 1: Cost of 10,000 monitored system calls in seconds (log scale). Error bars show standard deviation.

Experimental Results -- Apache

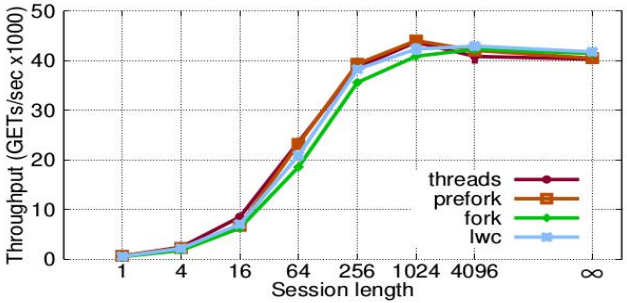
- Apache provides services in two configurations:
  - Kernel **threads**
  - **Pre-forked processes** that map to different cores.
- Another server Nginx use event loop within a process and have option of swapping multiple process that map to cores.
- These configurations do not provide privilege separation in different user sessions and per user information flow control.
  - Multi threaded and event driven configurations serve different sessions concurrently in the same process.
  - Pre-forked processes sequentially share among different sessions.
- Apache can be configured to fork new process for each user session to provide memory isolation and privilege separation.

Experimental Results -- Apache

- Results alongside indicate that apache in configuration to provide security services have reduced performance.
- Apache is augmented with lwC to provide security services using snapshot and rollback.
- This apache with lwC has better throughput in all cases and has significant advantage for short sessions.
- Moreover, lwC achieves performance comparable to the best configuration without isolation for sessions lengths of 256 and larger.



(a) HTTP

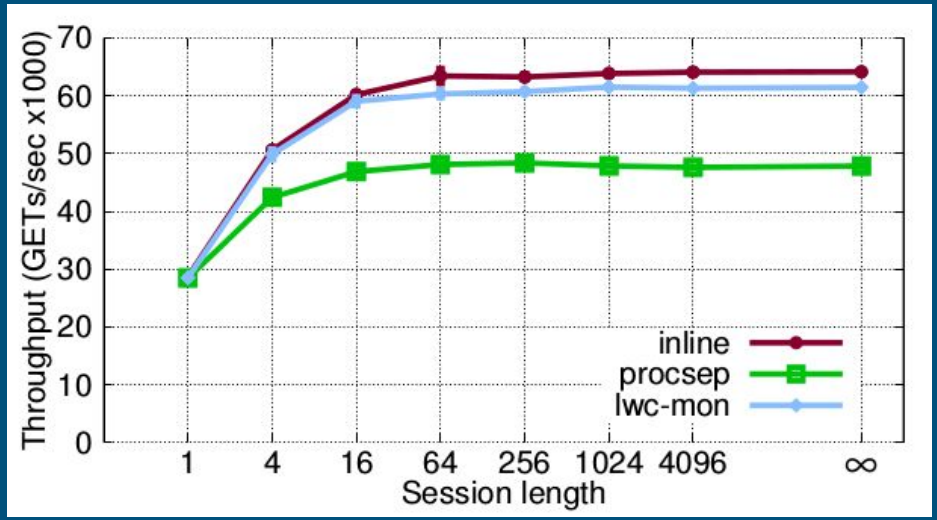


(b) HTTPS

Figure 2: Apache throughput in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 3.7%.

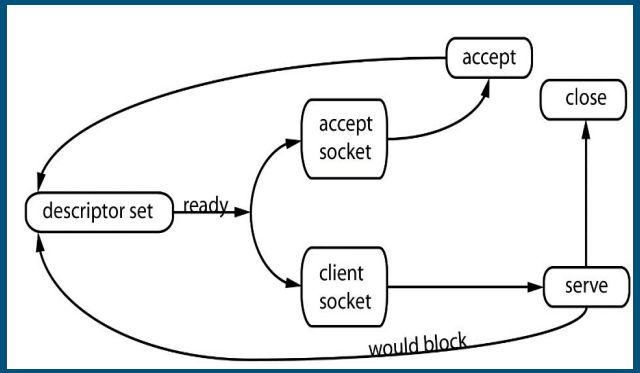
Experimental Results -- Apache

- Reference monitoring was integrated within Apache.
- Figure on the right shows the throughput of Apache prefork in different reference monitor configurations when used to serve short (45 byte) documents.
- The overhead of reference monitoring increases with session length due to the increase in relative number of reference monitored system calls compared to other system calls.

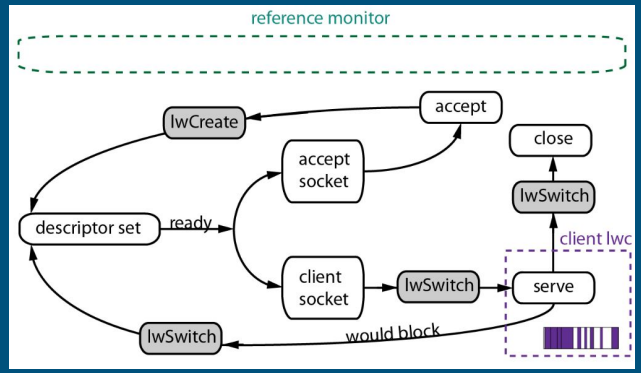


Experimental Results -- Nginx

- It is a high performance web server and it also do not provide session isolation.
- It is augmented to provide reference monitoring and session isolation as shown in figure.



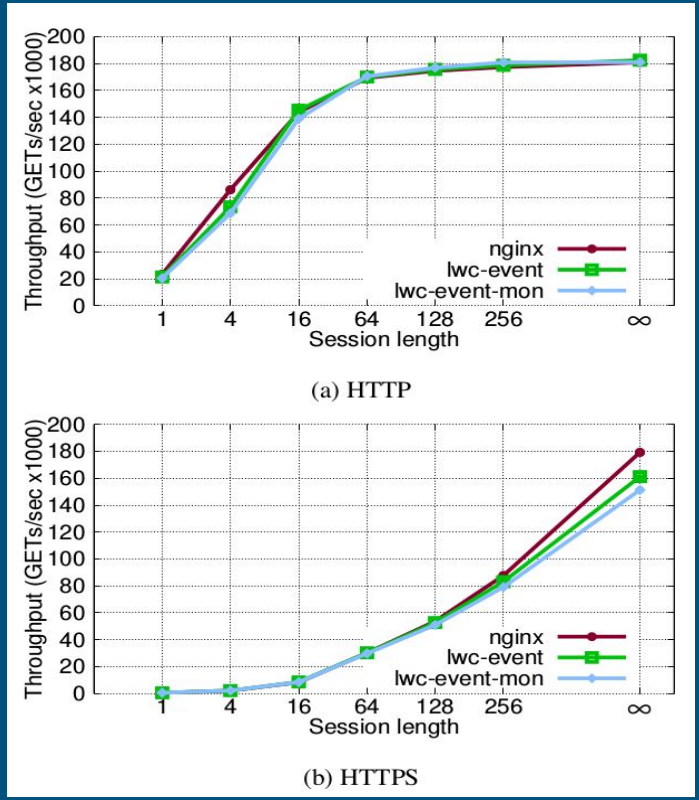
Original Nginx working



Augmented Nginx working

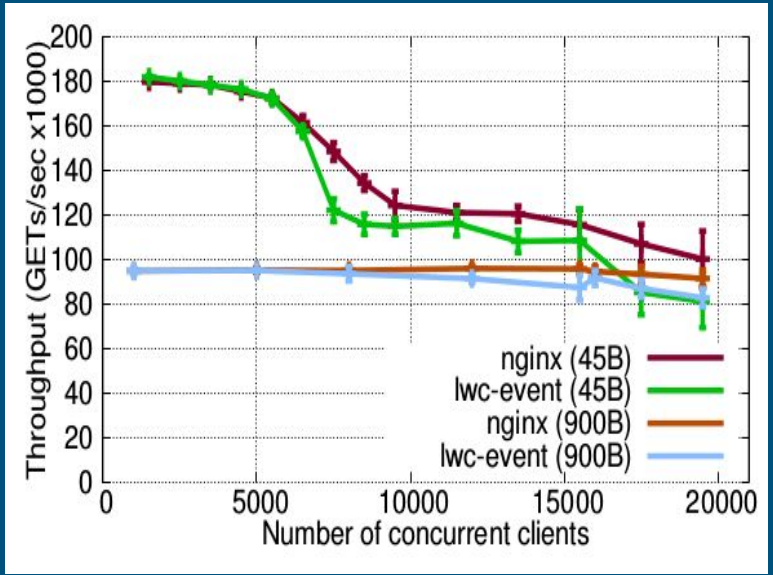
Experimental Results -- Nginx

- nginx is considered the state of the art high-performance server. It is about 2.88x quicker than apache.
- Introducing lwCs in base configuration has no significant impact on its throughput.
- Reference monitoring adds only minimal overhead.
- For both HTTP and HTTPS, with isolation and reference monitoring, lwC-augmented nginx performs comparably to native nginx.



Experimental Results -- Nginx

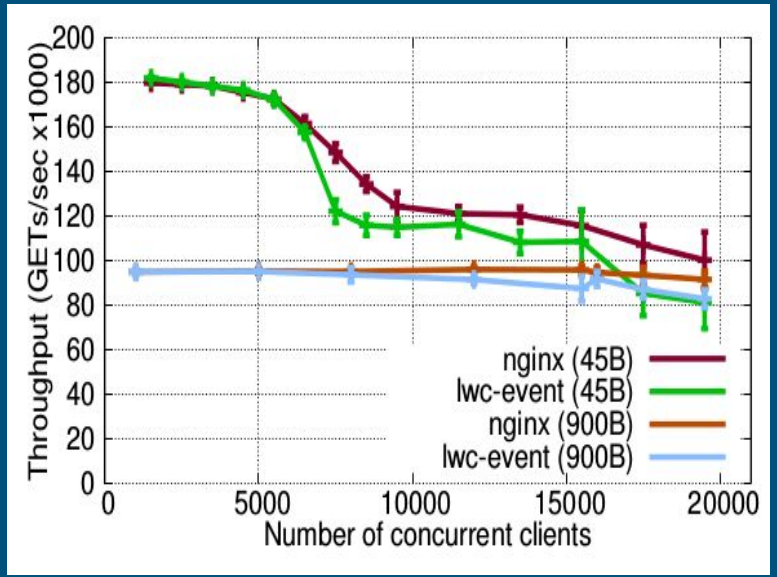
- Effect of using lwCs under a large number of concurrent client connections. Two configurations:
  - Between 6 and 76 ApacheBench instances, and each instance issues 250 concurrent requests for a 45 byte document.
  - Identical except the ApacheBench instances request 900 byte documents.



Experimental Results -- Nginx

- For small documents, lwc-event matches the performance of native nginx up to 6500 clients. Beyond, the performance of both configurations declines.

- In FreeBSD kernel threads, the interrupt handler thread, gets CPU bound after 6500 clients, and the CPU consumption of the nginx worker threads reduces with higher numbers of clients as the nginx worker threads block. The lwc-event configuration further pays an extra cost of lwC switches, which reduces performance compared to stock nginx. However, given that lwc-event provides session isolation, this is a still a strong result.



Experimental Results -- PHP FCGI

- Use lwC snapshots to speed up PHP.
- Before a PHP script performs any computation that depends on request-specific parameters, the script may invoke the pagecache call, which implements the snapshot pattern, and then revert to it on subsequent requests to the same URL.
- With or without the opcode cache, the lwC snapshot is able to provide significant performance benefit to highly optimized end-to-end applications such as web frameworks, while adding isolation between user requests.

stock php no cache	*lwC* php no cache	stock php cache	*lwC* php cache
226.1	615.8	1287.5	1701.4

Experimental Results -- Isolating OpenSSL Keys

- lwCs provide an effective way to isolate sensitive data from network-based attacks like buffer overflows or overreads. The sensitive data is stored in a lwC, within the process, such that the network-facing code has no visibility to it.
- As an example, parts of the OpenSSL library that manipulate web server certificate private keys were isolated.
- In experiments, native nginx required **99.7 seconds** to complete ten thousand SSL handshakes, whereas the configuration with a lwC isolated SSL library required **100.4 seconds**.
- Hence, with lwCs, isolating SSL private keys is essentially free.

Thank You