Revitalizing ns-3's Direct Code Execution

Parth Pratim Chatterjee
Kalinga Institute of Industrial Technology
parth27official@gmail.com

Thomas R. Henderson
University of Washington
tomhend@u.washington.edu

ABSTRACT

This paper describes the design, implementation and validation of the ns-3 model of the Licklider Transmission Protocol, the standard transport protocol used to provide transmission reliability in Delay Tolerant Networks (DTNs). DTNs are an emerging field whose principles are used to provide communications in extreme and performance-challenged environments, such as spacrecraft,underwater, or disaster response scenarios. Evaluation of such environments requires the use of simulation tools. As of now, there is a lack of precise simulation models of these protocols, and concretely within the ns-3 simulator. The ns-3 model presented in this paper accurately models the LTP protocol and offers ...

Keywords

ns-3

1. INTRODUCTION

- Introduce ns-3
- Motivation of DCE: make complex network implementations available to ns-3
- History of DCE, and challenge of maintaining it

The rest of this paper is organized as follows: Section 2 provides an overview on the technical challenges due to Linux evolution. Section 3 describes solutions. Section 4 presents the results. Section 5 related work. Section 6 conclusions and future work.

2. CHALLENGES

2.1 libio vtable mangling

The vtable is a table maintaining references to functions called for virtual functions defined for a class or an entity. These functions can be overriden dynamically by user defined functions and the respective call for the corresponding virtual function in a derived class object can be bound to that function at runtime, unlike pre-defined functions which are static, fixed, and can be determined during compile time. The libc on Linux provides this highly flexible feature for most user defined classes, but the case with the FILE structure is not the same.

The FILE structure is a library defined structure which defines the overall organization, orientation and properties of any file I/O stream opened by the host application. It maintains different parameters for storing useful operational fields like the UNIX based file descriptor number of the opened stream, the read/write offsets and buffer addresses of the stream. The pseudoname for the FILE structure as seen inside libc is *_IO_FILE*. Since, FILE is a library defined entity, the library provides its own set of implementation for all possible operations on an open FILE stream. Whenever an *_IO_FILE* stream is allocated by the kernel, a contiguous memory location is allocated as a block called *_IO_FILE_plus*. The *_IO_FILE_plus* structure looks like this.

```
1 struct _IO_FILE_plus
2 {
3   FILE file;
4   const struct _IO_jump_t *vtable;
5 };
```

By nature of the implementation of the kernel's memory allocation processes, the contents of a struct are allocated in contiguous memoory locations. This can be verified by the *sizeof* operation of C showing that the sum of sizes of the individual members of a struct is equal to the size of the struct object. Similarly, the FILE and the *_IO_jump_t* objects are allocated in contiguous memory locations. Specifically, the *_IO_jump_t* areas is interesting to DCE, as it defines the callbacks or reference pointers to the functions handlers for each supported file operation. Some of the callbacks which are interesting to DCE and its use cases are highlighed below.

```
1 struct _IO_jump_t
2 {
3 ...
4 ssize_t(*) __read (FILE *, void *, ssize_t)
      ;
5 ssize_t(*) __write (FILE *, const void *,
      ssize_t);
6 off64_t (*) __seek (FILE *, off64_t, int);
7 int (*) __close (FILE *);
8 int (*) __stat (FILE *, void *);
```

```
 9 ...
10 };
```

This structure acts like the vtable for the FILE structure, but it does not behave like the oridinary vtable seen when working with virtual functions and derived classes, which are dynamic and supports run time bindings. This vtable is instead expected to behave as a statically bound vtable. There exist other libc functions, like *fopencookie*, to override some of the FILE operation implementations, but not all operations are supported, and *fopencookie* also does not attach itself to a standard file I/O stream, instead working with user defined buffers a.k.a cookies).

A key aspect of DCE is that it needs to *hijack* application operations like system calls, file I/O operations, and networking system calls, and re-route them through corresponding handlers based on the application logic and simulation script implementation. Considering file I/O operations, DCE needs to have control over read/write/close/seek/stat operations of each open file, which requires overwriting the vtable handlers with the corresponding handlers defined in DCE's stdio definition source files.

Taking advantage of the contiguous memory allocation of the FILE and *_IO_jump_t*, one can overwrite the vtable with a custom vtable definition for all the operations needed for DCE. One can make a dummy *_IO_FILE_plus* pointer point to the explicitly casted FILE object. *memcpy* the existing vtable to a local copy, modify and overwrite the stream operations with a custom written implementation, and then re-point the vtable field of the dummy *_IO_FILE_plus* to the local modified vtable. This allows control over stream operations, which can now be routed through and and to behave as ns-3 streams, Unix FD streams, etc. based on the type of file descriptor that is defined. Although this control over FILE streams to regulate stream buffer flushing and data redirection is productive for DCE, these techniques can also be used maliciously for *buffer overflow attacks*, as it lets penetration testers to make use of tools like pwntools etc. to gain control over application execution and important run time CPU register values such as the *rip* which is used for the relative addressing of application components(which is also how position-independent-executables work), which is more secure as compared to static addressing, where fixed address values of symbols and pointers could be gained by static analysis tools for run time application exploitations.

Post libc-2.25, security features have been implemented to glibc to identify and block buffer overflow attacks on the FILE object. Whenever any FILE operation is executed, glibc verifies if the FILE object's vtable can be trusted and is not corruputed or manipulated. To verify this, it makes a call to *_IO_validate_vtable*. Every libio vtable is defined in a unique section called *libio_IO_vtables*. By definition, libc trusts the vtable if the vtable of the current FILE object lies within this section. It checks if the offsets of this vtable lies between *__stop__libc_IO_vtables* and *__start__libc_IO_vtables*, if it does, we can continue with the operation, if not, libc conducts a final check by calling *_IO_vtable_check* which makes final checks on the FILE vtable pointer location, namespace and edge cases where FILE * objects are passed to threads which are not in the currently linked executable. When DCE overflows the *_IO_FILE_plus* and overwrites the *_IO_jump_t*, it does not lie in *libio_IO_vtables* section and it also does not pass the pointer mangling sanity checks, leading to a *__libc_fatal (T̈fatal error: glibc detected an invalid stdio handle)̈;*

2.2 PIE loading and usage

Position-independent-executables (PIE) are applications compiled with special compiler flags that allow the application to be loaded into an arbitrary memory address, not depending on absolute symbol addresses. PIE executables avoid exploits that hijack the call stack by referencing fixed function/symbol addresses that can be found in executables not compiled in this manner. Every memory address is accessed with reference to what is called the *%rip* (instruction pointer). The *%rip* is computed at the time of execution when the application is loaded into virtual memory. This approach makes it difficult for attackers to determine symbol locations in memory.

DCE supports the execution of real host applications in simulation, bridging the networking layer between the host and the specified networking stack, and also supporting other system calls made by the host applications. Since several applications may need to be loaded into memory, and also have control over the position of the main symbol of the loaded application, DCE needs to have position independent executables, so that when they are loaded into memory, the symbol positions in memory are dynamic, giving control over when an application is launched in a simulation (which can be configured in the script using available ns-3 programming constructs for the *DceApplicationManager* class). To implement this, DCE uses the *CoojaLoader* which uses *dlmopen* under the hood to load the executable into memory. In glibc library versions newer than 2.25, security checks have been introduced to identify such PIE objects being loaded through *dlmopen*, and in case the **DF_1_PIE** flag is found in the object's ELF Dynamic headers, it will abort with an error.

2.3 Linux Networking Stack for DCE

DCE also provides implementation options for ns-3 simulations for the networking stack at the TCP/IP layers. Linux and FreeBSD stacks are available as alternatives to the native ns-3 TCP/IP implementation. Script writers can configure the *DceManager* class to use the chosen stack. Currenly, the Linux network stack is based on a project named *net-next-nuse-4.4.0*, which is built on top of base Linux kernel version 4.4.0.

The process for supporting portions of the Linux or FreeBSD kernel in DCE is much more extensive than that for a typical user-space application, because of the size of the kernel and because the kernel provides its own scheduler and has internal implementations of functions such as memory management. It is not simply a matter of compiling kernel code as PIE library code; more intrusive changes are necessary. *net-next-nuse* is a library port of the Linux kernel that performs selective linking of required kernel modules and components such as the networking, virtual file system (VFS), memory management unit (MMU), and other layers. The library port also performs additional functions such as exporting DCE useful callback structures, abstracting the internal network data flow, coordinating Linux kernel synchronization, process creating, and DCE-kernel task, IRQ and tasklet scheduling and synchronization, to expose a Linux-like execution environment for host applications.

Since Linux kernel 4.4.0, Linux kernel releases have had major developements in almost all kernel components, but

for our specific use case, the networking stack has seen major changes in several components, including the TCP timer, jiffies and clock HZ usage, packet processing (napi) framework, internal enums and state definitions and checks, packet flows, checksum and offload hardenings, newer congestion control algorithms, ucounts API, etc. The Linux kernel version 4.4 networking stack is useful for some ns-3 simulation work but lacks implementations found in more recent kernels that are of interest to current researchers. Therefore, to upgrade the supported kernel version to something more current, we considered both an evolution of the existing approach, as well as an evaluation of two similar projects, LKL and LibOS, described next.

2.3.1 LKL

The Linux Kernel Library (LKL) is a library port of the Linux kernel that, through some pre-shipped helper shell scripts, can be used to hijack all system calls made by a host application and map them through the ported Linux kernel rather than through system defined implementations. It also allows one to setup network interfaces such as TAP, RAW, VDE , etc. with custom gateways, netmasks, IP addresses, etc. with the help of JSON configuration files. These helper shell scripts make use of `LD_PRELOAD` to reorder library loading to LKL written system calls to take control in place of the libc defined routines.

2.3.2 LibOS

LibOS uses the same internal architecture as does *net-next-nuse*. It is also a Linux kernel port that works on the principle of selective kernel module linking and patching. It defines special link time constructs to include only specific kernel files and symbols that are needed for executing on top of the Linux kernel with *nuse*, which works similar to LKL by hijacking system calls and rerouting them through nuse and kernel defined routines. Only kernel components that let LibOS start the kernel and run all *__initcall*(s), which are defined with a *__init* and registered as as a initcall using special macros, are linked. These routines are linked into special *.INIT* setions of the final linked library. These components include critical parts such *kernel*, *net* and other selective parts of *proc*, *mm*, *fs* , *drivers*, etc.

2.3.3 LKL vs. LibOS

We conducted a comparison of LKL and LibOS approaches according to different design parameters and considerations for a complex application framework such as DCE with very specific demands from it's underying network stack.

2.3.3.1 Linux Kernel Support.

LKL, which was primarily designed to work as a Linux-as-userspace-library interface for applications to be dynamically linked to at runtime, is built on top of Linux 5.2. The kernel port design of LKL facilitates kernel version upgrades with little to no effort. Abstractly, the kernel upgrade process would permit a git rebase on the kernel version the user would want to use, and the project should compile with no major issues to deal with (some minor compiler, and Linux kernel header definition changes might come up, which should be resolvable with a bit of effort).

LibOS, which makes use of dynamic, selective Linux kernel linking, bridges the gap between application workspace and Linux kernel networking stack with the help of glue code, kernel component connector code, and user application provided exported functions and callbacks for proper execution. This architecture required LibOS to modify some of the internal Linux kernel files for additional components such as the slab memory allocator, which requires LibOS to setup preprocessor directives to select a slab allocator for specific Kconfig-defined compiler directives, to pass on control to LibOS routines whenever required. Currently, LibOS supports Linux kernel version 4.4.0. Upgrading to a newer Linux kernel version might be an intense process requiring one to deal with issues from header file changes, to complete changes in kernel components like the networking layer, memory maqnagement, namespace manager and kernel boot process.

2.3.3.2 In-Library Kernel Boot Order.

Apart from certain functions that are initialized only in a user OS, such as hardware drivers and devices, network buses, NICs, etc., LKL spins up a high level CPU lock controlled thread, which makes calls to `start_kernel`. Since LKL is a uniprocessor system, it initializes the kernel on a singular LKL thread, locks of which are synchronized with Linux scheduler calls, which are called when the scheduler decides to switch execution control to kernel level tasks for preemption, and other tasks, which require certain memory level moderation to achieve atomic operation. Also, since the LKL CPU thread needs to be initialized before any Linux functions can be used by the application that is using the LKL library, it eventually disrupts the flow of DCE scripts, which work on a scheduling algorithm and specific ns-3 task context switch paradigms that are different from the Linux kernel, making it wait for LKL's internal Linux kernel opertions to finish, before it can schedule other ns-3 lightweight threads. The net result of this tension is to cause simulation result to differ from real world observations.

LibOS does not depend on the actual `start_kernel`, particularly because, as opposed to LKL which sets up an environment for the Linux kernel, all of the basic requirements for the kernel to assume an actual Linux workspace making it easy for LKL to access most of Linux functionalities without any change to kernel internals, LibOS on the other hand makes use of its own `lib_init` fuction which calls specific setups calls required by the network stack of the kernel to work, such as proc, VFS, ramfs, scheduler, etc. It also overrides scheduler member functions, syncing it with ns-3's scheduler. LibOS creates an ns-3 task copy for each kernel task which is created by the `copy_process`, `create_process`, etc. kernel functions. Each such task maps with itself a callback function which should be called once the wait time for a task is over, or has been invoked as a part of a regular scheduling process. Once tasks are processed, they are also popped off the ns-3 task queue.

2.3.3.3 Maintenance.

LKL was designed to be a low-maintenance, Linux-as-a-userspace-library interface, which exposes Linux subsystems through overidden system calls. Since LKL is an architecture-level port of the Linux kernel, isolating LKL specific code to the arch and tools Linux folders, moving on to newer or custom releases induces less cost as compared to LibOS, which works on dynamic component compiling and linking the build system through a custom Makefile, specifying Linux files and modules to be compiled and linked to the final nuse shared object file, and the selection of files to

compile must be complete, assuring that no function calls are being made by any critical component to another component that does not exist, and as necessary, writing glue code to fulfill the depedency graph for all such modules and files. This makes migrating to newer versions of Linux kernel more time consuming, and even simply debugging kernel internals requires more effort since one is not sure if it is the glue code, the allocator, or the ns-3 synced scheduler that is malfunctioning.

3. SOLUTIONS

3.1 Custom glibc-2.31-based build

Given that the standard glibc library, since version 2.25, contains security features that block how DCE makes use of it, the alternatives are to either use a different standard library implementation without such features, or to use a modified version of glibc. Two options are *libc* and *musl*. The post-linking structure of the generated binary and the libary vary, in terms of the number of static linkages, symbol tables, etc. For instance, musl works on the idea of single static linkage, in which, the library or executable compiled with musl-gcc, is statically linked to only one musl linked library called `ld-musl.so.1`, which defines all the symbols required by the application. This reduces the size of the executable by a huge extent, but also does more harm than good to DCE. The initial build step of DCE includes calling a script named `dcemakeversion.c`. This script is responsible for extracting the symbol table of the libc currently being linked to (all symbols for the libraries libc, libpthread, librt, lib and libdl). These libraries are the various modular extensions of the glibc providing features such as pthreads, math, dynamic library loading and the base libc library as well. The symbols are read from the Elf headers of the respective shared library .so file, and stored in a local .version file.

The symbol table for all of these libraries are important, as DCE generates a shared library called `libc-ns3.so` which is a for the local libc and DCE implementations, on top of which host applications are executed. This shared library defines all symbols which are defined in the local libc, as `NATIVE` and all the features implemented inside DCE as `DCE`. All other symbols that are new to DCE but not implemented by DCE, but that are a part of the local system libc, are then referenced in the .version files. These symbols are then defined as well, to avoid any runtime symbol lookup errors. DCE then generates preprocessor mappings for all of the symbols. All DCE defined symbols will natively be mapped to DCE implemented versions of them, rather than to the system libc implementation, and all NATIVE defined symbols will be mapped to global namespace implementations, which are the ones already implemented in the system libc. Users would compile their applications on top of the system libc itself, but with an extra -fPIC and -pie flag, which allows us to load the applications dynamically into the DCE process address space. The next step is to load out the libc-ns3.so shared object file, and to call the `libc_setup` function, which initializes the system call mappings. DCE also subsequently loads other libraries that have been generated in a similar way, such as `libpthread-ns3.so`, `librt-ns3.so`, `libm-ns3.so`, and `libdl-ns3.so`. As a final step, the host application is loaded and the main function of the application is called, which starts to now work on top of the custom libraries. `musl-gcc`, on the other hand, does not support modular libraries for these features, and for all cases would link the single `ld-musl.so.1`, making it impossible to segregate the symbols for the different libraries. glibc is a better match for what DCE requires, and links all the standard libraries needed.

To override the vtable mangling security checks of glibc, it is necessary to use a modified libc version that matches the system's default libc version, so that symbol lookup errors are avoided at runtime by applications built on a different library and being loaded into an available namespace using `dlmopen`. It is also necessary to override the security checks on vtable pointer mangling. The gcc compiler and linker options could be used to reconfigure the default build environment to build DCE on top of a customized glibcc. The default root directory where gcc starts to look for libraries, header files, etc. is `'/'` on Linux. It is necessary to repoint this directory to the custom glibc root. This is where the `-sysroot` option is used to set it to correct bake build directory. Following this, one can then add the custom glibc prioritized directory for library and header file lookup using the -L and -I options, respectively. Next, the rpath and rpath-link paths for ELF executables that could be linked to the shared objects at run time or link time, respectively, are set. Finally,the dynamic linker is set to the newly built library, using the -Wl–dynamic-linker flag. All these changes are placed under an unclosed -Wl –start-group, as DCE requires other linker flags, which can be added before inserting the ending -Wl –end-group.

3.2 Bake Build Automation

Bake is a Python-based build system invented for DCE. It orchestrates the build and local installation of several DCE dependencies before managing the DCE build itself, by calling other build tools native to the respective projects, such as make, Waf, or CMake. Bake works on a build configuration script called `bakeconf.xml`. The bakeconf.xml file is converted to `bakefile.xml` after the configuration file is parsed by the Bake module. This creates a build dependency graph, maintaining build steps, parameters and post build commmands for each dependency node. Bake dynamimcallly binds dependency modules based on configuration options specified for each build module defined in the bakeconf.xml. The three major steps of a Bake build are: 1) configure, 2) download and 3) build. Bake supports almost all major types of source code fetching methods, including git, mercurial and the fetching of compressed archive files. In the build step, dependencies are built in the required order, starting with modules having no dependencies on other modules, and finishing with the module having all dependencies already satisfied. To support the custom glibc build, the dependency graph looked something like this:

ToDo : Attach Bake Dependency graph ?

Bake has a source configuration option named *patch*, which can safely apply a patch, without re-applying if it has already been applied before. This allows, for example, a large codebase to be fetched in its unaltered form, while Bake must maintain and apply only a small patch file to it. Using this option, Bake applies the DCE-specific glibc patch, which disables the security checks on vtable mangling, and also disables the PIE object checks for dlmopen position independent executable loading. The glibc is then build using its standard build steps. The linux kernel headers files are

then installed into the `/usr` directory of a custom glibc's sysroot. This finishes with a standard Linux-like system root to use for building DCE without any issues.

3.3 Docker environnment for DCE

3.4 net-next-nuse-5.10

Net-next-nuse, introduced above, is an architecture port of the Linux kernel, which resorts to selective kernel module linking and which takes control over critical kernel components such as the slab allocator, task scheduler, workqueue-waitqueue handling, timer based function invocation, as well as some kernel utilities such as jiffies and random number generators. It also sets up emulations of the network system calls for both general socket networking operations as well netdevice based operations. All of this is exposed through an API initialised by a simulator initialisation call, `sim_init`, which does a bidirectional mapping of the imported (DCE to Linux) and exported (Linux to DCE) function utilities.

3.4.1 Background

The implementation of net-next-nuse considers only the use case of DCE and thus might not find application outside of DCE, unlike LKL which is designed for running a plethora of networking applications, but it fits in really well when it comes to DCE.

When Linux builds, apart from all other files, it generates two important files: `vmlinuz` and `initramfs`. The file vmlinuz is where all the kernel code is compiled and linked. LKL performs a raw objcopy on the generated object files vmlinux and copies all symbols from vmlinux along with the global functions exported from the LKL arch port, and combines them into a final library called `lkl.o`, which is then passed on to the tools/lkl, to link the host, netdevice, utility, filesystem, and networking handler codes to make the final `liblkl.so` shared library. In the initial stages of the DCE modernization effort, this approach sounded promising, but as explained below, it can become a bottleneck.

In contrast, net-next-nuse understands that the way Linux Kernel behaves on the host, and the way the kernel interacts with the task scheduler and how the scheduler operates on the tasks, is completely different from how DCE manages fibers (called tasks in DCE terminology). DCE is based on context-based, lightweight threads called fibers, which require a custom scheduler, called the TaskManager in DCE. On an ordinary host machine, when a user runs any application that spins up normal threads, the user does not need to bother about scheduling the threads on the different cores of the host CPU and maintaining mutex locks of the CPU, making the kernel responsible for jumping from one thread to another. But when using fibers, one can jump from one thread to another only when one of the fibers yields to another fiber, and this context switch is done by a custom scheduler, which is written in DCE's TaskManager::Schedule(). This is beneficial because when using ordinary threads, the number of context switches the CPU has to do (push your thread data stack on to you address space, in and out), is a lot more expensive than making switches in fibers.

Another challenge is as follows. Separate schedulers exist inside DCE and within the Linux kernel library. As a result, every new operation that you make inside the kernel creates something called a (virtual) kernel thread. However, managing those threads is challenging because the DCE TaskManager does not have visibility to the (virtual) threads. The net-next-nuse approach to solving this issue is to perform a selective linking of kernel modules and utilities and to avoid linking the scheduler, workqueue, waitqueue and other such linked components, opting to instead implement them using DCE imported functions mapped to TaskManager utilities. As a result, every time that some process wants to preempt and block, the kernel calls `schedule()` or `schedule_timeout()`, and control is passed to DCE to take care of it.

3.4.2 Slab Allocator

When the kernel is booted up by net-next-nuse in `lib_init(...)`, it calls some of the required initialization functions, and all included, initcalls. Most of these initcalls require the creation of a memory cache object called `kmem_cache`. These memory slabs are used to allocate memory to child objects, using functions like `kmem_cache_alloc(...)` and functions as simple as `kmalloc(...)` to allocate space for a pointer object. The kernel already has slab allocators (two such allocators are SLUB and SLOB), but all of them allocate space internally, giving no control to DCE. We thus use a custom SLAB allocator called SLIB, which calls internal `DCE's malloc(...)` functions and sets up `kmem_cache` data structures and constructors etc. It also implements compound and single page operations like`_put_page(...)`, `kfree(...)` and array space allocation.

3.4.3 Virtual Kernel Task and its Real Fiber Equivalent

Thread management in the kernel is another important consideration. Every new task inside the kernel– for example, a syscall– creates an internal kernel thread. When a system boots up, there has to be some initial task that invokes the `start_kernel(...)` function. This is called the `init_task`. Whenever a new kernel thread is to be created, a call to `kernel_thread(...)` is made. Along with the function that is to be invoked in the thread, and the arguments to be passed, one must also pass along one more argument, known as clone flags. Internally, the kernel will try to clone a previous task as much as possible. These flags basically determine how far the kernel should go, as in cloning the structures. Some of the flags are `CLONE_FS`, `CLONE_VM`, `CLONE_FILES`, etc., and `kernel_clone(...)` passes a special clone flags data structure. This function further calls `copy_process(...)`, which is responsible for checking which flags are enabled using a bitwise `&` operator and calling the corresponding copy utility; for example, `copy_fs(...)`, `copy_files(...)`, and `copy_cred(...)`. It also makes a call to `dup_task_struct(...)`, which will bind changes of the current `task_struct` to a new one and return it back. After receiving back a proper `task_struct`, the `kernel_clone(...)` schedules a fork for the newly created task.

However, DCE is not a complete architecture-level port and thus lacks a clear definition of `init_task`. So, for the very first `kernel_thread`, the current task would be `init_task`, so not having a proper definition of it creates a domino effect making all subsequent tasks have an incomplete structure, leading to possible segmentation faults. Furthermore, DCE does not use the kernel scheduler, but instead forks a kernel thread in such a way that the DCE scheduler can see it, and every time TaskManager::TaskWait(...)

is called on a particular task, DCE can put the current job to sleep, and give the other tasks/fibers a chance to execute their set of functions.

Therefore, to take control over the kernel thread creation, we rewrite the `kernel_thread(...)` function to call `lib_task_start(...)`, which calls the internal TaskManager::Start(...) to create a DCE task, fills up the task_struct, sets the SimTask context with the task_struct and returns back the process ID (pid) of the task. Also, `current`, as previously referred to above, is not a variable, but a macro, which calls `get_current(...)`, which DCE hijacks and leads to TaskManager::TaskCurrent(...), which checks if the current task has a SimTask context. If it does, it returns it back. If it does not, it calls TaskManager::Start(...) to set it up.

3.4.4 Scheduler Workqueue/Waitqueue Implementation

Most of the scheduling code can be found in the `/kernel` directory of the Linux kernel. DCe does not compile the original kernel implementation for the scheduler, but rather implements all of the key functions used by the networking stack. One may ask, if no applications are running inside the kernel, when would anything ever need to be scheduled? The answer is that blocking network calls require such support. Imagine creating a socket, binding and listening to it and then making an `accept(...)` call, and the client application doesn't seem to ever come up. The system would go to a standstill if it did not schedule the client application for it to issue a corresponding `connect(...)` call. This is one of the issues with LKL [4.2], and is why we avoid linking those files, and rather pass the majority of the control to DCE, but without making a single change in code, because DCE also has to work when simulation scripts use the default ns-3 network stack. DCE therefore rewrites certain functions. Below, a few function and concepts have been discussed :

- schedule() :
- schedule_timwout() :
- lib_task_wait() :
- lib_task_yield() :
- add_timer() :
- mod_timer() :
- init_timer_key() :
- do_softirq() :
- open_softirq() :
- queue_work_on() :

3.4.5 Jiffies timer

The Linux kernel management of time is based on the use of the global jiffies variable, which contains a 32-bit integer that reports the number of ticks elapsed since the boot of the kernel. The duration of each tick depends on the way the kernel was configured, but in modern Linux systems, it is usually configured to be one millisecond. The jiffies variable is normally increased whenever the kernel timer interrupt is triggered. This timer interrupt is also responsible for executing any expired kernel timers every ten milliseconds, etc. [22, 19] both deal with this by executing periodically a per-node event that increases the jiffies variable and relies on the Linux kernel code to deal with its internal timers. This approach suffers from one major drawback: even if there are no timers scheduled to expire for the next 10 or 80 milliseconds, the simulation will keep running and executing events, just for the sake of incrementally increasing the value of this jiffies variable. Rather than waste time to do this, DCE instead configures the Linux kernel to not use periodic ticks with `CONFIG NOHZ` and then replaces the kernel timer facility entirely to schedule simulation events for each kernel timer instead of keeping track of the kernel events in a data structure separate from the main simulation event list. The resulting kernel network stack thus runs in tickless mode and does not waste time scheduling unnecessary events.

3.4.6 Kernel initialization

The kernel initializers are divided into base init functions and roughly eight levels of initcalls, and export a start and end pointer of the list. Init functions are the ones directly called in the `start_kernel(...)` or by certain device drivers, whereas initcalls are the ones which are stored in a special .initcall.init section of the final linked library. These are then copied to their respective positions through the linker.lds script. In the `lib_init` function, I call specific init functions, namely :

- vfs_caches_init_early
- rcu_init
- devices_init
- buses_init
- radix_tree_init
- timekeeping_init
- cred_init
- uts_ns_init
- vfs_caches_init
- seq_file_init
- proc_root_init

I also changed the initcall invocation loop of lib_init to this :

```
1    initcall_entry_t *call;
2    extern initcall_entry_t __initcall_start
     [], __initcall_end[];
3
4    call = __initcall_start;
5    do {
6        initcall_from_entry(call)();
7        call++;
8    } while (call < __initcall_end);
```

Before we call all the init functions, I also added these two lines of code :

```
1    files_cachep = kmem_cache_create("
     files_cache",
2            sizeof(struct files_struct), 0,
3            SLAB_HWCACHE_ALIGN|SLAB_PANIC|
     SLAB_ACCOUNT,
```

```
4               NULL);
5
6       fs_cachep = kmem_cache_create("fs_cache"
        ,
7               sizeof(struct fs_struct), 0,
8               SLAB_HWCACHE_ALIGN|SLAB_PANIC|
        SLAB_ACCOUNT,
9               NULL);
```

These caches are required while setting up the `fs_struct` and `files_struct` for each new task in its corresponding `task_struct`. Please see [5.9].

3.4.7 Security LSM Module

The security module of the Linux kernel depends on a special configuration called the `CONFIG_LSM`. The LSM module requires a few architecture defined variables. Surprisingly, one cannot find them in the source code (.c or .h files) anywhere; rather, they are placed in a specific section of the library/binary called .init.data section, through assembly code. net-next-nuse makes use of a linker script which can help us position variables inside the library under specific sections. I had to put this code in the linker.lds to initialise the start and end of the LSM table.

```
1    . = ALIGN(CONSTANT (MAXPAGESIZE));
2    .init.data : AT(ADDR(.init.data)) {
3      __start_lsm_info = .;
4      KEEP(*(.lsm_info.init))
5      __end_lsm_info = .;
6      __start_early_lsm_info = .;
7      KEEP(*(.early_lsm_info.init))
8      __end_early_lsm_info = .;
9    }
```

`vfs_kern_mount(...)` validates and parses params using `security_fs_context_parse_param(...)`, which checks if the requested operation was blocked in the LSM tables. If this module is not enabled, it would return back ENOPARAM by default and fail.

3.4.8 Kernel code and net-next-nuse alignment

In updating net-next-nuse, the `panic_on_taint` symbol had to be removed. This symbol is exported while linking kernel/panic.c (needed for acknowledging kernel runtime errors) and accessing it in kernel/sysctl.c caused errors because it could not find it back. Also, including this symbol was unnecessary because its value defaults to zero and never changes during program execution.

When the kernel is initialized, it calls `mnt_init`, which is responsible for setting up the kernel file system (kernfs), the sysfs, ramfs, etc. and then making a call to `init_mount_tree`, which sets up the required namespace for all mounts. This function will then make a corresponding call to `vfs_kern_mount`, which will use the vfs setup to initialize the needed file systems. `vfs_kern_mount` will set up the file context and mount it. In this process it checks for the security module and the LSM entries, to check if the kernel was not previously configured to avoid declaration of the specified namespace. If all checks pass, it returns back a vfsmount pointer. This pointer is used to set up a mount namespace which initializes the data structure `mnt_namespace`, using `alloc_mnt_ns`. To initialize the `mnt_namespace` object, the current tasks nsproxy is required, which is the user namespace proxy member. Every task is required to define some initial maximum value for each namespace it might require during the execution of the task. Similarly, for allocating a mount namespace, it increases the `UCOUNT_MNT_NAMESPACES` of the current task's nsproxy, which has an upper bound of `init_task`'s value. Since a complete architecture port is not needed, this value is not defined, and would throw a segmentation fault, when `READ_ONCE(...)` tries to make an atomic operation on it to access it and increase the value by 1 if doing so doesn't exceed the max limit, using the `atomic_inc_below(..)` macro. We thus manually set the value for it, during task initialisation in `lib_task_start(...)`.

```
1 ucounts->ns->ucount_max[
      UCOUNT_MNT_NAMESPACES] = MNS_MAX;
```

Although the mount namespace is set up for the init task, it is needed for other tasks as well. We therefore created a global variable `def_mnt_ns` and stored a backup of the first mount namespace using an extern variable and then later on set it up in the `lib_task_start(...)`

```
1  extern struct path def_root;
2  extern struct mnt_namespace *def_mnt_ns;
3
4  static void __init init_mount_tree(void){
5    ...
6  struct nsproxy *ns;
7  ns->uts_ns = 0;
8  ns->ipc_ns = 0;
9  ns->mnt_ns = def_mnt_ns;
10 ns->pid_ns_for_children = 0;
11 ns->net_ns = &init_net; // global struct *
       net
12 task->kernel_task.nsproxy = ns;
13   ...
14
15 def_root = root;
16 def_mnt_ns = ns;
17
18   ...
19 }
```

This seems good right ? But what is def_root ? Ok, I had to hack this again XD. Remember init_mount_tree, we discussed it in the pointer above. What does it do ? It sets up a vfs mount. Okay, does that mean a path somewhere in the kernel ? Yes it should have. I also found another line of code there : init_task.nsproxy->mnt_ns = ns;

Okay, does this mean that the first task has the mount namespace generated while we mount and allocate the mount tree and acquire an mnt_namespace. What else can we observe here ? How does the Linux Kernel create tasks then ? Do you remember doing these when you create processes on your OS ? No, right ? Which means, do we reuse some of the previous data from the init_task and replicate them in new tasks. Yes, we do. Whenever we create a kernel_thread(..) it calls copy_process(...) which copies some of the required parameters from the host task, depending upon the flags you pass it. For example, If you had set the CLONE_FS flag, it would copy the fs_struct from the init_task to the newly made task, by calling copy_fs(..) in kernel/fork.c. Which means, if we never have diverse task member requirements we can just reuse the previous task, implying the first task ? Right ! So, we can then just declare a global def_root and then by declaring an extern variable for it, and keep reusing !

The netif_napi_add tests for the NAPI_STATE_LISTED bit to be enabled in the state of the napi_struct passed on to it from cgroup init calls. We never set this flag directly,

so this check was removed from the kernel code. sock_init is one of the many initcalls which are called when sim_init is invoked by DCE. First of all before we proceed with anything, it is necessary that we have initialised the proc file system completely, so that the sysctl interface could be initialised by the net_sysctl_init(...). We then register the sockfs filesystem using register_filesystem(...) and go for a kern_mount(...) which invokes the VFS kern mount function and as discussed above, it requires the mnt_init(...) to correctly setup the mount environment. Initially, in net-next-nuse-4.4.0, mnt_init(...) was made a blank function, and we have reasons to support it. In older kernel releases, sockfs file system init did not depend on a file system context, and required a mount function (sockfs_mount) which would directly mount the file system onto the kernel, but in commit [6.1], the socket file system mounting process was handed over to the internal VFS mounting mechanism, thus breaking our setup. So, now we need to have the mnt_init(...) and put in patches and glue codes wherever needed.

TODO: include/arm changes to files, for example, atomic, barrier, ptrace, user, user32/64. . . .

TODO: Rump kernel header file double include, adding flag to disable linux include

3.4.9 Native Kernel NetDevices

Certain components of ns-3 require the setup of custom NetDevices which require a custom MTU and other flags, such as whether the device should enable multicast, or is a point-to-point device, etc. These flags can be put together, and one can allocate a Linux netdevice struct for each such requirement using the `alloc_netdev(...)`, passing all of the configuration needed for details such as MTU, address length, and destructors, along with a `register_netdev(...)` call. This returns back a forward declared struct SimDevice which is used as a NetDevice object inside ns-3.

- Requests are provided in the form of API functions ...
- Notifications are provided as callback functions ...

4. RESULTS

4.1 Docker vs Native DCE Simulation Tests

4.2 Performance : DCE vs. ns-3

4.3 Google BBR v1 Validation Results

5. RELATED WORK

6. CONCLUSIONS

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] Space networks user's guide (snug). Technical Report Revision 9, Goddard Space Flight Center, Greenbelt, Maryland, August 2007.

[2] O. R. Helgason and K. V. Jónsson. Opportunistic networking in omnet++. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 82:1–82:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] A. Keränen, J. Ott, and T. Kärkkäinen. The one simulator for dtn protocol evaluation. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 55:1–55:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[4] Ltplib documentation and test. http://dtn.dsg.cs.tcd.ie/sft/ltplib/.

[5] Ltp-ri specification and implementation. http://irg.cs.ohiou.edu/ocp/ltp.html.

[6] The interplanetary overlay network (ion). http://ion-dtn.sourceforge.net/.

[7] Common open research emulator (core). http://www.nrl.navy.mil/itd/ncs/products/core.

[8] R. Wang, S. C. Burleigh, P. Parikh, C.-J. Lin, and B. Sun. Licklider transmission protocol (ltp)-based dtn for cislunar communications. *IEEE/ACM Trans. Netw.*, 19(2):359–368, Apr. 2011.