Paper 2 Report

z5089812, Xuan Tong

1. Summary

This paper presents the idea which make use of off-shelve CODOM architecture to accelerate inter-process communication, named as Direct Inter-Process Communication (dIPC). It addresses the slow IPC problem comes with components isolation. dIPC as an OS extension that repurposes and extends the CODOMs architecture to allow threads to cross process boundaries. Process isolation is the segregation of different software processes to prevent them from accessing memory space they do not own, component isolation is critical for contemporary software system from security and fault-tolerance perspective, so we definitely need isolation, and dIPC can help to improve its performance to make it usable.

Source of overheads for normal IPC includes resource isolation, state isolation, its instrinsic generality and semantic complexity etc. dIPC is accelerated mainly by fulfilling the following 3 requirements: no Operating System intervention for IPC, processes and domains in dIPC can be created and destroyed dynamically, untrusted application programmers can define their own isolation policies.

CODOM provides two important concepts worth knowing for dIPC, 1. CODOMs verifies whether the current instruction is allowed to access a specific memory address, by using the instruction pointer as the subject of access control checks instead of using the process as subject. 2. Domains can also share arbitrary data buffers through capabilities, like seL4, it can create capability by delegating, and delete by revoking.

dIPC-enabled processes are loaded into a global virtual address space, which means in dIPC, multiple processes share one page table, each page table entry is tagged with corresponding domain, and each tag associate with an Access Protection List, which indicates the permission grant to that domain, there are 3 permissions: Call, Read and Write.

There are 3 types of stack for primary thread in dIPC, Data stack, Capability Stack and Kernel Control Stack. dIPC enforces the confidentiality and integrity in these stacks and registers by autogenerated proxies, which in turn help to implement crossdomain exception recovery, maintain the stability of OS.

Lastly, this paper presents the performance comparison after running microbenchmark and macrobenchmark, we can tell that this solution brings component/process isolation with surprisingly low overhead, its speed is close to syscall in traditional monolithic kernel.

2. Pros

The authors have successfully built compiler and corresponding system that can be used to perform IPC calls in CODOM, demonstrate to successfully manage the threads and memory and gives out a relatively good performance. Compiler is critical for this project, it generate code stubs and additional binary information, we can tell that the authors spent great effort in fine-tuning the compiler and dIPC implementation.

Successfully extend CODOM architecture, make it capable to do process isolation, as well as add dIPC to interact with thread management, memory management.

3. Cons

Does not scale well with single address space. Resource contention is a potential problem, how it going to do swapping, worth some consideration. Also the efficiency of tagging and permission(r,w,c) may deteriorate when the page table grows larger and larger.

Argument immutability is implemented in non-privileged user code by simply copying argument content, can be improved by using shared memory or passing pointers

4. Criticisms

Didn't benchmark on large software system, all the processes within dIPC share the same address space, this may definitely cause resource contention when it comes large software system.

In the beginning, it mentions that it is 8.87× faster than IPC in the L4 microkernel, however in chapter 7, it didn't actually give out any detail on how it is measured.

Large Trust Code Base, over 9 K lines of new code for Linux (version 3.9.10) and 2 K for the runtime, which is hard to verify its correctness.

In 7.2 case study, it mentions that different asymmetric policies in dIPC can have up to a 8.47× performance difference, however it didn't mention what asymmetric policy it uses.

`dIPC-enabled processes are loaded into a global virtual address space (which in turn allows using a shared page table), while other processes are loaded normally` I think it worth articulating how to deal with concurrency between dIPC-enabled processes and normal processes.