Abstract

Mutually trusting processes can be safely run in the same address-space (effectively running *processes-as-threads*) without having source-level changes. This has compelling performance improvements for programs that can be realized with very little work. Some programs such as Python do not support parallel threads, and this may be the only way to unlock high performance on multicore architecture.

I can implement this abstraction of processes-as-threads without modifying the underlying process's code or the kernel. In just one semester I have implemented a prototype of the process-as-threads abstraction for CPython, which allows it to run parallel threads safely. I demonstrate massive performance benefits from this approach with comparatively little work.

1 Motivation

Modern systems need distinct virtual-to-physical mappings (*address space*) for several reasons. While extremely important in the general case, I argue that there is a specific case where we can relax them for the sake of performance. Multithreading and virtual memory-aliasing (`mmap`) already relax this constraint in *non*-memory-safe languages, which is more dangerous than my proposal.

Stability One malfunctioning process should not be able to cause other processes to malfunction . When a programmer wants to parallelize code, they implicitly trust themselves, so the security of processes is not necessary here .

Security Processes do not necessarily trust each other with their volatile data . In a memory-safe programming languages, direct memory access is not possible, eliminating this class of bugs .

On the other hand, running in the same address space has important performance benefits:

Reduced context-switching time When the OS switches from one process to another it has to flush the TLB, file-descriptor table, and other per-process resources; When switching between threads, this state is shared .

Reduced communication/synchronization time Inter-process communication (such as Unix pipe) has to copy data into and out-from the kernel; Inter-thread communication can leveraged the shared memory-space copying once or zero times (sending a reference) .

Reduced memory fragmentation Huge pages are intended to extend the reach of the TLB, however this faces implementation problems. If each process gets a huge page, physical memory runs out, even though each process might only be using a fraction of its page. Multiple threads however could share a huge page between themselves .

This workload is becoming increasingly important due to the limitations of single-core performance and the move to multi-core . Python is often used for big-data applications, with frameworks such as Spark and Dask .

Python cannot support parallel multithreading . It has a `threading` module, but this can only one one thread of Python bytecode at a time . If the Python bytecode calls into an external extension or waits on an OS event, then other Python bytecode can execute, giving us *concurrency* but not *parallelism* . Concurrency accelerates I/O-bound tasks, but does not accelerate CPU-bound tasks. The state-of-the-art for Python is to use multiple processes (`multiprocessing` module) .

Many interpreted-languages, such as Python, have that constraint. The interpreter is not thread-safe, so it has to be protected behind a Global Interpreter Lock (GIL). Neither normal processes nor normal threads can provide the benefits of running in the same address-space. However the process-as-threads abstraction can provide this benefit without modifying CPython nor the OS kernel at all. Compare this to other proposals which involve a massive overhaul of the CPython interpreter–a so-called 'GIL-ectomy'.

In just one semester I have implemented a prototype of the process-as-threads abstraction. I have applied it parelellizing Python with extremely positive performance results. The degree of performance improvement I could get on a project as mature as Python in one semseter by myself is extremely promising. My vision is that with one more year I could develop a polished process-as-threads which could parallelize any programming language interpreter without any change to process-as-threads or the language. Rewriting an interpreter using multithreading natively is going to be the best long-run move, but this provides close to the same benefit with much less work. It could even apply to run two different lanuage interpreters in the same addres-space, accelerating polyglot applications (i.e. a system utilizing code in multiple languages simultaneously).

2 Footnotes, Verbatim, and Citations

Footnotes should be places after punctuation characters, without any spaces between said characters and footnotes, like so.[1]

Now we're going to cite somebody. Watch for the cite tag. Here it comes. Arpachi-Dusseau and Arpachi-Dusseau co-authored an excellent OS book, which is also really funny [1], and Waldspurger got into the SIGOPS hall-of-fame due to

[1]Remember that USENIX format stopped using endnotes and is now using regular footnotes.

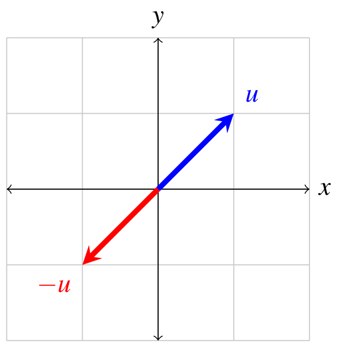

Figure 1: Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as p caption's text. Text size inside figure should be as big as caption's text.

his seminal paper about resource management in the ESX hypervisor [2].

And the 'cite' package sorts your citations by their numerical order of the corresponding references at the end of the paper, ridding you from the need to notice that, e.g, "Waldspurger" appears after "Arpachi-Dusseau" when sorting references alphabetically [1, 2].

It'd be nice and thoughtful of you to include a suitable link in each and every bibtex entry that you use in your submission, to allow reviewers (and other readers) to easily get to the cited work, as is done in all entries found in the References section of this document.

Now we're going take a look at Section 3, but not before observing that refs to sections and citations and such are colored and clickable in the PDF because of the packages we've included.

3 Floating Figures and Lists

Here's a typical reference to a floating figure: Figure 1. Floats should usually be placed where latex wants then. Figure1 is centered, and has a caption that instructs you to make sure that the size of the text within the figures that you use is as big as (or bigger than) the size of the text in the caption of the figures. Please do. Really.

In our case, we've explicitly drawn the figure inlined in latex, to allow this tex file to cleanly compile. But usually, your figures will reside in some file.pdf, and you'd include them in your document with, say, \includegraphics.

Lists are sometimes quite handy. If you want to itemize things, feel free:

fread a function that reads from a `stream` into the array `ptr` at most `nobj` objects of size `size`, returning returns the number of objects read.

Fred a person's name, e.g., there once was a dude named Fred who separated usenix.sty from this file to allow for easy inclusion.

The noindent at the start of this paragraph in its tex version makes it clear that it's a continuation of the preceding paragraph, as opposed to a new paragraph in its own right.

3.1 LaTeX-ing Your TeX File

People often use `pdflatex` these days for creating pdf-s from tex files via the shell. And `bibtex`, of course. Works for us.

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.

References

[1] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. `http://pages.cs.wisc.edu/~remzi/OSTEP/`.

[2] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002. `https://www.usenix.org/legacy/event/osdi02/tech/waldspurger/waldspurger.pdf`.