Linux | DB | Open Source | Web

≡ Menu

- Home
- Free eBook
- Start Here
- Contact
- About

Introduction to Linux Threads – Part I

by Himanshu Arora on March 30, 2012

Like 5 Tweet

A thread of execution is often regarded as the smallest unit of processing that a scheduler works on.

A process can have multiple threads of execution which are executed asynchronously.

This asynchronous execution brings in the capability of each thread handling a particular work or service independently. Hence multiple threads running in a process handle their services which overall constitutes the complete capability of the process.

In this article we will touch base on the fundamentals of threads and build the basic understanding required to learn the practical aspects of Linux threads.

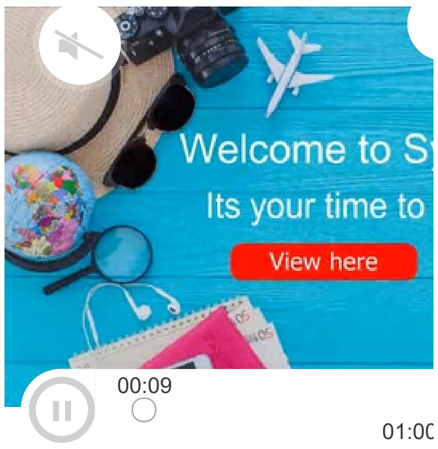
Linux Threads Series: part 1 (this article), part 2, part 3.

Why Threads are Required?

Now, one would ask why do we need multiple threads in a process?? Why can't a process with only one (default) main thread be used in every situation.

Well, to answer this lets consider an example :

Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output. Now, if the process is not multi-threaded ie if the process does not involve multiple threads, then the whole processing in the process becomes synchronous. This means that the process takes an input processes it and produces an output.



The limitation in the above design is that the process cannot accept an input until its done processing the earlier one and in case processing an input takes longer than expected then accepting further inputs goes on hold.

To consider the impact of the above limitation, if we map the generic example above with a socket server process that can accept input connection, process them and provide the socket client with output. Now, if in processing any input if the server process takes more than expected time and in the meantime another input (connection request) comes to the socket server then the server process would not be able to accept the new input connection as its already stuck in processing the old input connection. This may lead to a connection time out at the socket client which is not at all desired.

This shows that synchronous model of execution cannot be applied everywhere and hence was the requirement of asynchronous model of execution felt which is implemented by using threads.

Difference Between threads and processes

Following are some of the major differences between the thread and the processes :

- Processes do not share their address space while threads executing under same process share the address space.
- From the above point its clear that processes execute independent of each other and the synchronization between processes is taken care by kernel only while on the other hand the thread synchronization has to be taken care by the process under which the threads are executing
- Context switching between threads is fast as compared to context switching between processes
- The interaction between two processes is achieved only through the standard inter process communication while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

User threads Vs Kernel Threads

Threads can exist in user space as well as in kernel space.

A **user space** threads are created, controlled and destroyed using user space thread libraries. These threads are not known to kernel and hence kernel is nowhere involved in their processing. These threads follow co-operative multitasking where-in a thread releases CPU on its own wish ie the scheduler cannot preempt the thread. Th advantages of user space threads is that the switching between two threads does not involve much overhead and is generally very fast while on the negative side since these threads follow co-operative multitasking so if one thread gets block the whole process gets blocked.

A **kernel space** thread is created, controlled and destroyed by the kernel. For every thread that exists in user space there is a corresponding kernel thread. Since these threads are managed by kernel so they follow preemptive multitasking where-in the scheduler can preempt a thread in execution with a higher priority thread which is ready for execution. The major advantage of kernel threads is that even if one of the thread gets blocked the whole process is not blocked as kernel threads follow preemptive scheduling while on the negative side the context switch is not very fast as compared to user space threads.

If we talk of Linux then kernel threads are optimized to such an extent that they are considered better than user space threads and mostly used in all scenarios except where prime requirement is that of cooperative multitasking.

Problem with Threads

There are some major problems that arise while using threads :

- Many operating system does not implement threads as processes rather they see threads as part of parent process. In this case, what would happen if a thread calls fork() or even worse what if a thread execs a new binary?? These scenarios may have dangerous consequences for example in the later problem the whole parent process could get replaced with the address space of the newly exec'd binary. This is not at all desired. Linux which is POSIX complaint makes sure that calling a fork() duplicates only the thread that has called the fork() function while an exec from any of the thread would stop all the threads in the parent process.
- Another problem that may arise is the concurrency problems. Since threads share all the segments (except the stack segment) and can be preempted at any stage by the scheduler than any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function and uses the same variables or data structures.

For the problem 1 mentioned above, all we can say is that its a design issue and design for applications should be done in a way that least problems of this kind arise.

For the problem 2 mentioned above, using locking mechanisms programmer can lock a chunk of code inside a function so that even if a context switch happens (when the function global variable and data structures were in inconsistent state) then also next thread is not able to execute the same code until the locked code block inside the function is unlocked by the previous thread (or the thread that acquired it).

Tweet Like 5 > Add your comment

If you enjoyed this article, you might also like..

1. 50 Linux Sysadmin Tutorials
2. 50 Most Frequently Used Linux Commands (With Examples)
3. Top 25 Best Linux Performance Monitoring and Debugging Tools
4. Mommy, I found it! – 15 Practical Linux Find Command Examples
5. Linux 101 Hacks 2nd Edition eBook Free

- Awk Introduction – 7 Awk Print Examples
- Advanced Sed Substitution Examples
- 8 Essential Vim Editor Navigation Fundamentals
- 25 Most Frequently Used Linux IPTables Rules Examples
- Turbocharge PuTTY with 12 Powerful Add-Ons

{ 18 comments… add one }

- Jalal Hajigholamali March 30, 2012, 2:41 am

  Hi,

  Thanks a lot, i am waiting for part-II

  Link
- Marco March 31, 2012, 5:16 am

  You can't even imagine how much I appreciate this website! It's been a hell of a discovery! I'm a CS student attending the course of Operating Systems and your articles are almost going synchronously with my professor's lectures. 😛 Thank you very much!

  Link
- Hasitah Jayasooriya April 2, 2012, 4:01 am

  Really good post at all. thanks, also waiting for part II

  Link
- bob April 2, 2012, 7:14 am

  good article that gets to the kernel of the issue.

  Link
- Himanshu Arora April 3, 2012, 5:33 am

  @All
  Thanks for the appreciation

  Link
- Narendra Yadav April 20, 2012, 3:32 am

  Thanks for this valuable knowledge.
  But I have a doubt, may be I am wrong..
  In the above Kernel space paragraph you are saying that "For every thread that exists in user space there is a corresponding kernel thread." But I think kernel can only assign processes to processors — Two threads within the same process cannot run simultaneously on two processors, so it is the user process that is seen by the kernel not the user thread.
  Plesae help me to understand this.
  Thanks

Link

- Ron Marques July 18, 2012, 5:25 am

  Hopefully you can elaborate more on the topic. Waiting for the next part

  Link
- Mohanasundaram October 8, 2012, 4:44 am

  Excellent article with some great examples. Thanks a lot.

  Link
- gururaj March 13, 2013, 8:23 am

  I have a doubt here.
  These two statements of the article contradicts with each other.

  “These(user space) threads follow co-operative multitasking where-in a thread releases CPU on its own wish ie the scheduler cannot preempt the thread.”

  “Another problem that may arise is the concurrency problems. Since threads share all the segments (except the stack segment) and can be preempted at any stage by the scheduler than any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function and uses the same variables or data structures. ”

  Please correct me if my understanding is wrong.

  Thank you

  Link
- Ammar Khan June 3, 2013, 11:53 am

  Really well written article, explains concepts very succinctly. Keep it up. Thank you.

  Link
- Sandeep August 11, 2013, 11:17 pm

  Is it possible to monitor a thread real time status like we do for process using top or htop command. Question is if i have a process with 500 thread running (cat /proc/PID/status). this process is consuming high resources. Now how can i monitor which thread is consuming most of resources, i tried to use htop but the problem is if i have 500 threads running under a process its difficult to find which thread is consuming high bcz it does not shows high utilizing process first as top command does.

  May be my question or approach is worng. Please let me know.

  Thanks
  Sandeep

  Link
- Shitian Long August 28, 2013, 3:08 am

  I have hard time to understand this paragraph.
  Many operating system does not implement threads as processes rather they see threads as part of parent process. In this case, what would happen if a thread calls fork() or even worse what if a thread execs a new binary?? These scenarios may have dangerous consequences for example in the later problem the whole parent process could get replaced with the address space of the newly exec’d binary. This is not at all

desired. Linux which is POSIX complaint makes sure that calling a fork() duplicates only the thread that has called the fork() function while an exec from any of the thread would stop all the threads in the parent process.

I am wondering if you could make an example of this problem. Thanks

Link

- Prasad September 24, 2013, 12:31 am

  if a thread calls fork() system call which one gets duplicated thread or master thread contents can u please explain once and share where exactly this scenario

  Link
- Gopi July 14, 2014, 7:39 pm

  I have a great day. Even I know user level threads but some confusion. From today I can forget my confusion on threads. Thanks for clear explanation.

  Link
- selva December 30, 2014, 8:46 am

  hi,
  really nice article to understand the concept of thread.
  Thank you.

  Link
- Rezwanul islam September 6, 2015, 8:27 pm

  Really written of thread of LINUX

  Link
- Manjunath January 8, 2016, 12:18 pm

  Very Nice explanation

  Link
- Neelkanth February 17, 2017, 5:05 am

  Awesome Explanation. Thanks for sharing the knowledge.

  Link

Leave a Comment

Name

Email

Website

Comment

Submit

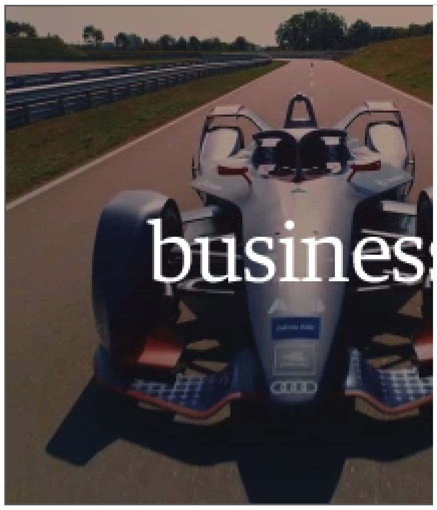
Notify me of followup comments via e-mail

Next post: How to Create, Compile, Load Linux LKM Loadable Kernel Modules

Previous post: How to Setup VirtualBox Guest Additions and Network

RSS | Email | Twitter | Facebook | Google+

Custom Search Search



EBOOKS

- Free Linux 101 Hacks 2nd Edition eBook - Practical Examples to Build a Strong Foundation in Linux
- Bash 101 Hacks eBook - Take Control of Your Bash Command Line and Shell Scripting
- Sed and Awk 101 Hacks eBook - Enhance Your UNIX / Linux Life with Sed and Awk
- Vim 101 Hacks eBook - Practical Examples for Becoming Fast and Productive in Vim Editor
- Nagios Core 3 eBook - Monitor Everything, Be Proactive, and Sleep Well

The Geek Stuff
17,381 likes

Like Page Share

Be the first of your friends to like this

POPULAR POSTS

- 15 Essential Accessories for Your Nikon or Canon DSLR Camera
- 12 Amazing and Essential Linux Books To Enrich Your Brain and Library

- 50 UNIX / Linux Sysadmin Tutorials
- 50 Most Frequently Used UNIX / Linux Commands (With Examples)
- How To Be Productive and Get Things Done Using GTD
- 30 Things To Do When you are Bored and have a Computer
- Linux Directory Structure (File System Structure) Explained with Examples
- Linux Crontab: 15 Awesome Cron Job Examples
- Get a Grip on the Grep! – 15 Practical Grep Command Examples
- Unix LS Command: 15 Practical Examples
- 15 Examples To Master Linux Command Line History
- Top 10 Open Source Bug Tracking System
- Vi and Vim Macro Tutorial: How To Record and Play
- Mommy, I found it! -- 15 Practical Linux Find Command Examples
- 15 Awesome Gmail Tips and Tricks
- 15 Awesome Google Search Tips and Tricks
- RAID 0, RAID 1, RAID 5, RAID 10 Explained with Diagrams
- Can You Top This? 15 Practical Linux Top Command Examples
- Top 5 Best System Monitoring Tools
- Top 5 Best Linux OS Distributions
- How To Monitor Remote Linux Host using Nagios 3.0
- Awk Introduction Tutorial – 7 Awk Print Examples
- How to Backup Linux? 15 rsync Command Examples
- The Ultimate Wget Download Guide With 15 Awesome Examples
- Top 5 Best Linux Text Editors
- Packet Analyzer: 15 TCPDUMP Command Examples
- The Ultimate Bash Array Tutorial with 15 Examples
- 3 Steps to Perform SSH Login Without Password Using ssh-keygen & ssh-copy-id
- Unix Sed Tutorial: Advanced Sed Substitution Examples
- UNIX / Linux: 10 Netstat Command Examples
- The Ultimate Guide for Creating Strong Passwords
- 6 Steps to Secure Your Home Wireless Network
- Turbocharge PuTTY with 12 Powerful Add-Ons

CATEGORIES

- Linux Tutorials
- Vim Editor
- Sed Scripting
- Awk Scripting
- Bash Shell Scripting
- Nagios Monitoring
- OpenSSH
- IPTables Firewall
- Apache Web Server
- MySQL Database
- Perl Programming
- Google Tutorials
- Ubuntu Tutorials
- PostgreSQL DB
- Hello World Examples
- C Programming
- C++ Programming
- DELL Server Tutorials
- Oracle Database
- VMware Tutorials

About The Geek Stuff

My name is **Ramesh Natarajan**. I will be posting instruction guides, how-to, troubleshooting tips and tricks on Linux, database, hardware, security and web. My focus is to write articles that will either teach you or help you resolve a problem. Read more about Ramesh Natarajan and the blog.

Contact Us

Email Me : Use this Contact Form to get in touch me with your comments, questions or suggestions about this site. You can also simply drop me a line to say hello!.

Follow us on Google+

Follow us on Twitter

Become a fan on Facebook

Support Us

Support this blog by purchasing one of my ebooks.

Bash 101 Hacks eBook

Sed and Awk 101 Hacks eBook

Vim 101 Hacks eBook

Nagios Core 3 eBook