What is the difference between a process and a thread?

Asked 11 years, 2 months ago Active 1 month ago Viewed 1.1m times

1532

828

What is the technical difference between a process and a thread?

I get the feeling a word like 'process' is overused and there are also hardware and software threads. How about light-weight processes in languages like Erlang? Is there a definitive reason to use one term over the other?

multithreading process

edited Feb 11 '18 at 15:25
too honest for this site
10.9k 3 25 48

asked Oct 14 '08 at 9:13
James Fassett
33k 11 32 42

2 Related: stackoverflow.com/questions/32294367/… – zxq9 Aug 16 '17 at 14:24

3 It probably warrants saying that each OS has a different idea of what is a 'thread' or 'process'. Some mainstream OS' don't have a concept of 'thread', there are also some embedded OS' that only have 'threads'. – Neil Aug 22 '18 at 16:39

34 Answers

1 2 next

1380

✓

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

I'm not sure what "hardware" vs "software" threads you might be referring to. Threads are an operating environment feature, rather than a CPU feature (though the CPU typically has operations that make threads efficient).

Erlang uses the term "process" because it does not expose a shared-memory multiprogramming model. Calling them "threads" would imply that they have shared memory.

edited Aug 15 '17 at 17:28

answered Oct 14 '08 at 9:15

p1100i
3,278 2 23 42

Greg Hewgill
730k 160 1047 1192

52 Hardware threads are probably referring to multiple thread contexts within a core (e.g. HyperThreading, SMT, Sun's Niagara/Rock). This means duplicated register files,extra bits carried around with the instruction through the pipelines,and more complex bypassing/forwarding logic,among other things. – Matt J Mar 6 '09 at 6:10

4 @greg, one doubt I have in threads. let me consider I have a process A, which got some space in RAM. If the process A creates a thread, the thread also need some space to execute. So will it increase size of the space which is created for process A, or space for thread created somewhere else ? so what is that virtual space process creates ? Please correct me if my question is wrong. Thanks – duslabo Sep 20 '12 at 15:45

9 @JeshwanthKumarNK: Creating a new thread allocates at least enough memory for a new stack. This memory is allocated by the OS in process A. – Greg Hewgill Sep 20 '12 at 19:20

19 This answer seems wrong. If both processes and threads were independent sequences of execution, then a process that contained two threads would have to have three sequences of execution, and that can't be right. Only a thread is a sequence of execution -- a process is a container that can hold one or more sequences of execution. – David Schwartz May 28 '16 at 0:37

8 "Hardware threads" are threads that are given individual hardware resources (a separate core, processor, or hyperthread). "Software threads" are threads that have to compete for the same processing power. – jpmc26 Jan 19 '17 at 19:02

767

Process
Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

Thread
A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Found this on MSDN here:
About Processes and Threads

> Microsoft Windows supports preemptive multitasking, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

edited Jan 10 '17 at 18:30

answered Oct 14 '08 at 9:43
Scott Langham
48.6k 31 115 181

17 For people who want to know why cant you format a floppy at the same time : stackoverflow.com/questions/20708707/… – Computernerd Dec 20 '13 at 17:34

7 @LuisVasconcellos - If there were no threads, then the process wouldn't do anything. The process would only be some code and program state loaded into memory. It's not much use. It'd be like having a road with no vehicles travelling along it. – Scott Langham Mar 31 '15 at 12:59

3 @LuisVasconcellos - Good. Yes, you can think of a thread as something that moves through the process's code and that carries out the instructions in that code. – Scott Langham Mar 31 '15 at 16:12

7 This answer is way better than the accepted answer because it talks about the *ideal* of processes and threads: They should be separate things with separate concerns. The fact is, most operating systems have history that goes back farther than the invention of threads, and consequently, in most operating systems, those concerns are still somewhat entangled, even if they are slowly improving over time. – Solomon Slow Mar 23 '16 at 13:50

4 @BKSpurgeon With every explanation one gives, you have to take your reader from one level of understanding to the next level. Unfortunately, I can't tailor the answer to every reader and so have to assume a level of knowledge. For those who don't know, they can make further searches of terms I use they don't understand, can't they, until they reach a base point they do understand. I was going to suggest you offer your own answer, but am happy to see you already have. – Scott Langham Jul 6 '16 at 20:48

295

Process:

- An executing instance of a program is called a process.
- Some operating systems use the term 'task' to refer to a program that is being executed.
- A process is always stored in the main memory also termed as the primary memory or random access memory.
- Therefore, a process is termed as an active entity. It disappears if the machine is rebooted.
- Several process may be associated with a same program.
- On a multiprocessor system, multiple processes can be executed in parallel.
- On a uni-processor system, though true parallelism is not achieved, a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency.

- **Example:** Executing multiple instances of the 'Calculator' program. Each of the instances are termed as a process.

Thread:

- A thread is a subset of the process.
- It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel.
- Usually, a process has only one thread of control – one set of machine instructions executing at a time.
- A process may also be made up of multiple threads of execution that execute instructions concurrently.
- Multiple threads of control can exploit the true parallelism possible on multiprocessor systems.
- On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time.
- All the threads running within a process share the same address space, file descriptors, stack and other process related attributes.
- Since the threads of a process share the same memory, synchronizing the access to the shared data within the process gains unprecedented importance.

I borrowed the above info from the **Knowledge Quest! blog**.

edited Jul 9 '18 at 8:46
Ganesh Kamath - 'Code Frenzy'
4,061 31 44

answered Mar 19 '10 at 14:17
Kumar
3,015 1 11 2

86 Kumar: From my knowledge, threads do not share the same stack. Otherwise it wouldn't be possible to run different code on each of them. – Mihai Neacsu Apr 23 '13 at 23:51

26 Yup I think @MihaiNeacsu is right. Threads share "code, data and files" and have their own "registers and stack". Slide from my OS course: i.imgur.com/Iq1Qprv.png – Shehaaz Oct 24 '13 at 17:22

1 Kquest.co.cc links are dead. – Elijah Lynn Apr 27 '15 at 12:50

119

First, let's look at the theoretical aspect. You need to understand what a process is conceptually to understand the difference between a process and a thread and what's shared between them.

We have the following from section *2.2.2 The Classical Thread Model* in Modern Operating Systems 3e by Tanenbaum:

> The process model is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in....

He continues:

> One way of looking at a process is that it is a way to group related resources together. A process has an address space containing program text and data, as well as other resources. These resource may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily. The other concept a process has is a thread of execution, usually shortened to just thread. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

Further down he provides the following table:

```
Per process items              | Per thread items
------------------------------|-----------------
Address space                  | Program counter
Global variables               | Registers
Open files                     | Stack
Child processes                | State
Pending alarms                 |
Signals and signal handlers    |
Accounting information         |
```

Let's deal with the hardware multithreading issue. Classically, a CPU would support a single thread of execution, maintaining the thread's state via a single program counter, and set of registers. But what happens if there's a cache miss? It takes a long time to fetch data from main memory, and while that's happening the CPU is just sitting there idle. So someone had the idea to basically have two sets of thread state (PC + registers) so that another thread (maybe in the same process, maybe in a different process) can get work done while the other thread is waiting on main memory. There are multiple names and implementations of this concept, such as HyperThreading and Simultaneous Multithreading (SMT for short).

Now let's look at the software side. There are basically three ways that threads can be implemented on the software side.

1. Userspace Threads
2. Kernel Threads
3. A combination of the two

All you need to implement threads is the ability to save the CPU state and maintain multiple stacks, which can in many cases be done in user space. The advantage of user space threads is super fast thread switching since you don't have to trap into the kernel and the ability to schedule your threads the way you like. The biggest drawback is the inability to do blocking I/O (which would block the entire process and all it's user threads), which is one of the big reasons we use threads in the first place. Blocking I/O using threads greatly simplifies program design in many cases.

Kernel threads have the advantage of being able to use blocking I/O, in addition to leaving all the scheduling issues to the OS. But each thread switch requires trapping into the kernel which is potentially relatively slow. However, if you're switching threads because of blocked I/O this isn't really an issue since the I/O operation probably trapped you into the kernel already anyway.

Another approach is to combine the two, with multiple kernel threads each having multiple user threads.

So getting back to your question of terminology, you can see that a process and a thread of execution are two different concepts and your choice of which term to use depends on what you're talking about. Regarding the term "light weight process", I don't personally see the point in it since it doesn't really convey what's going on as well as the term "thread of execution".

answered Oct 22 '13 at 12:42
Robert S. Barnes
34.7k 26 119 171

4 Outstanding answer! It breaks down a lot of the jargon and assumptions. That does make this line stand out as awkward, though: "So someone had the idea to basically have two sets of thread state (PC + registers)" -- what is the "PC" referred to here? – Smithers Jan 20 '15 at 17:50

2 @Smithers The PC is the program counter, or instruction pointer, which gives the address of the next instruction to be executed: en.wikipedia.org/wiki/Program_counter – Robert S. Barnes Jan 21 '15 at 10:47

1 @stackoverflowuser2010 no only threads have stacks. What you call a process is a process with a single thread of execution and it is the thread that has the stack not the process. – Robert S. Barnes May 16 '16 at 4:59

92

To explain more with respect to concurrent programming

1. A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
2. Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Keeping average person in mind,

On your computer, open Microsoft Word and web browser. We call these two *processes*.

In Microsoft word, you type some thing and it gets automatically saved. Now, you would have observed editing and saving happens in parallel - editing on one thread and saving on the other thread.

edited Feb 14 '18 at 11:49
Palak Jain
473 4 11

answered Dec 24 '12 at 7:04
Reachgoals
1,383 1 11 9

12 Outstanding answer, it keeps things simple and provides an example every user even viewing the question can relate to. – Smithers Jan 20 '15 at 18:00

6 editing/saving was a nice example for multiple threads inside a process! – user645579 Mar 30 '15 at 15:47

51

> An application consists of one or more processes. A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

Stolen from here.

answered Oct 14 '08 at 9:16
Node
18.9k 2 28 35

1 Good answer (especially with crediting), as it shows the relation between the two and segues into an easily expected "next question" (about fibers). – Smithers Jan 20 '15 at 18:04

27

A process is a collection of code, memory, data and other resources. A thread is a sequence of code that is executed within the scope of the process. You can (usually) have multiple threads executing concurrently within the same process.

answered Oct 14 '08 at 9:30
Gerald
21.1k 9 64 93

24

Real world example for Process and Thread *This will give you the basic idea about thread and process*

Process and Thread real world example

Here, The Process (or Task) name is known as "**Construct road**".

Ok, how can I construct a road, so I need some resource for construction, right?

Resources:

1) Boundary of road(Area)
    a. In technically we can call it as virtual address space, it has unique process identifier to identfy the work
    b. Access to limited boundary of area | security context
    c. Also, you can match the other properties of **process** with the above example (environment, priority, etc.)
2) Hardware or labor
    a. Number of "hardware or labor" count is based on road contractor mind set; suppose he/she want to finish the work quickly then he/she has to assign more people on this work
    b. I.e. each worker can access this limited area (Shared boundary space)
    c. At least one person needed to start work on this
    d. Each person has their own identifier
    e. So, can we call it as **Thread**? if yes match the other properties of thread with this example (exception handling, thread context etc.)

I borrowed the above info from Scott Langham's Answer - thanks

answered Dec 6 '16 at 23:24
Ratheesh
481 4 7

23

- Every process is a thread (primary thread).
- But every thread is not a process. It is a part(entity) of a process.

edited May 4 '16 at 19:42
ROMANIA_engineer
40k 22 171 156

answered Aug 9 '13 at 20:28
karthikeyan_somu
265 2 4

3 Can you explain that a bit further and/or include some evidence? – Zim84 Aug 9 '13 at 20:47

23

Process:

1. Process is a heavy weight process.
2. Process is a separate program that has separate memory,data,resources ect.
3. Process are created using fork() method.
4. Context switch between the process is time consuming.

Example:
Say, opening any browser (mozilla, Chrome, IE). At this point new process will start to execute.

Threads:

1. Threads are light weight processes.Threads are bundled inside the process.
2. Threads have a shared memory,data,resources,files etc.
3. Threads are created using clone() method.
4. Context switch between the threads are not much time consuming as Process.

Example:
Opening multiple tabs in the browser.

answered Sep 1 '17 at 4:49

community wiki
ANK

14

Both threads and processes are atomic units of OS resource allocation (i.e. there is a concurrency model describing how CPU time is divided between them, and the model of owning other OS resources). There is a difference in:

- Shared resources (threads are sharing memory by definition, they do not own anything except stack and local variables; processes could also share memory, but there is a separate mechanism for that, maintained by OS)
- Allocation space (kernel space for processes vs. user space for threads)

Greg Hewgill above was correct about the Erlang meaning of the word "process", and here there's a discussion of why Erlang could do processes lightweight.

edited Oct 14 '08 at 9:35

answered Oct 14 '08 at 9:29
Sergey Mikhanov
7,760 9 38 52

13

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

Process

Is a program in execution. it has text section i.e the program code, current activity as represented by the value of program counter & content of processors register. It also includes the process stack that contains temporary data(such as function parameters, return addressed and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

Thread

A thread is a basic unit of CPU utilisation; it comprises a thread ID, a program counter, register set, and a stack. it shared with other threads belonging to the same process its code section, data section and other operating system resources such as open files and signals.

-- Taken from Operating System by Galvin

edited Jul 12 '16 at 8:49
Henry Pham
1,779 3 14 19

answered May 10 '15 at 8:43
Saurabh Juneja
789 1 6 12

11

Trying to answer this question relating to Java world.

A process is an execution of a program but a thread is a single execution sequence within the process. A process can contain multiple threads. A thread is sometimes called a **lightweight process**.

For example:

Example 1: A JVM runs in a single process and threads in a JVM share the heap belonging to that process. That is why several threads may access the same object. Threads share the heap and have their own stack space. This is how one thread's invocation of a method and its local variables are kept thread safe from other threads. But the heap is not thread-safe and must be synchronized for thread safety.

Example 2: A program might not be able to draw pictures by reading keystrokes. The program must give its full attention to the keyboard input and lacking the ability to handle more than one event at a time will lead to trouble. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this. Here Drawing picture is a process and reading keystroke is sub process (thread).

edited Apr 3 '13 at 18:57

answered Apr 3 '13 at 18:29
Rupesh
2,081 1 20 36

1 Good answer, I like that it defines its scope (Java world) and provides some applicable examples--including one (#2) that anyone who has to ask the original question can immediately relate to. – Smithers Jan 20 '15 at 17:55

10

http://lkml.iu.edu/hypermail/linux/kernel/9608/0191.html

> Linus Torvalds (torvalds@cs.helsinki.fi)
>
> Tue, 6 Aug 1996 12:47:31 +0300 (EET DST)
>
> Messages sorted by: [date][thread][subject][author]

> Next message: Bernd P. Ziller: "Re: Oops in get_hash_table"
>
> Previous message: Linus Torvalds: "Re: I/O request ordering"
>
> On Mon, 5 Aug 1996, Peter P. Eiserloh wrote:
>
> > We need to keep a clear the concept of threads. Too many people seem to confuse a thread with a process. The following discussion does not reflect the current state of linux, but rather is an attempt to stay at a high level discussion.
>
> NO!
>
> There is NO reason to think that "threads" and "processes" are separate entities. That's how it's traditionally done, but I personally think it's a major mistake to think that way. The only reason to think that way is historical baggage.
>
> Both threads and processes are really just one thing: a "context of execution". Trying to artificially distinguish different cases is just self-limiting.
>
> A "context of execution", hereby called COE, is just the conglomerate of all the state of that COE. That state includes things like CPU state (registers etc), MMU state (page mappings), permission state (uid, gid) and various "communication states" (open files, signal handlers etc). Traditionally, the difference between a "thread" and a "process" has been mainly that a threads has CPU state (+ possibly some other minimal state), while all the other context comes from the process. However, that's just *one* way of dividing up the total state of the COE, and there is nothing that says that it's the right way to do it. Limiting yourself to that kind of image is just plain stupid.
>
> The way Linux thinks about this (and the way I want things to work) is that there *is* no such thing as a "process" or a "thread". There is only the totality of the COE (called "task" by Linux). Different COE's can share parts of their context with each other, and one *subset* of that sharing is the traditional "thread"/"process" setup, but that should really be seen as ONLY a subset (it's an important subset, but that importance comes not from design, but from standards: we obviusly want to run standards-conforming threads programs on top of Linux too).
>
> In short: do NOT design around the thread/process way of thinking. The kernel should be designed around the COE way of thinking, and then the pthreads *library* can export the limited pthreads interface to users who want to use that way of looking at COE's.
>
> Just as an example of what becomes possible when you think COE as opposed to thread/process:
>
> - You can do a external "cd" program, something that is traditionally impossible in UNIX and/or process/thread (silly example, but the idea is that you can have these kinds of "modules" that aren't limited to the traditional UNIX/threads setup). Do a:
>
> clone(CLONE_VM|CLONE_FS);

> child: execve("external-cd");
>
> /* the "execve()" will disassociate the VM, so the only reason we used CLONE_VM was to make the act of cloning faster */
>
> - You can do "vfork()" naturally (it meeds minimal kernel support, but that support fits the CUA way of thinking perfectly):
>
> clone(CLONE_VM);
>
> child: continue to run, eventually execve()
>
> mother: wait for execve
>
> - you can do external "IO deamons":
>
> clone(CLONE_FILES);
>
> child: open file descriptors etc
>
> mother: use the fd's the child opened and vv.
>
> All of the above work because you aren't tied to the thread/process way of thinking. Think of a web server for example, where the CGI scripts are done as "threads of execution". You can't do that with traditional threads, because traditional threads always have to share the whole address space, so you'd have to link in everything you ever wanted to do in the web server itself (a "thread" can't run another executable).
>
> Thinking of this as a "context of execution" problem instead, your tasks can now chose to execute external programs (= separate the address space from the parent) etc if they want to, or they can for example share everything with the parent *except* for the file descriptors (so that the sub-"threads" can open lots of files without the parent needing to worry about them: they close automatically when the sub-"thread" exits, and it doesn't use up fd's in the parent).
>
> Think of a threaded "inetd", for example. You want low overhead fork+exec, so with the Linux way you can instead of using a "fork()" you write a multi-threaded inetd where each thread is created with just CLONE_VM (share address space, but don't share file descriptors etc). Then the child can execve if it was a external service (rlogind, for example), or maybe it was one of the internal inetd services (echo, timeofday) in which case it just does it's thing and exits.
>
> You can't do that with "thread"/"process".
>
> Linus

answered Apr 17 '18 at 13:29
陳 力

3,100 1 10 31

9

Difference between Thread and Process?

A process is an executing instance of an application and A thread is a path of execution within a process. Also, a process can contain multiple threads.It's important to note that a thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a 'lightweight' process. Thus, the essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications.

Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication – is quite difficult and resource-intensive.

> Here's a summary of the differences between threads and processes:

1. Threads are easier to create than processes since they don't require a separate address space.
2. Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.
3. Threads are considered lightweight because they use far less resources than processes.
4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other.
   This is really another way of stating #2 above.
5. A process can consist of multiple threads.

edited Nov 25 '13 at 10:14

answered Nov 21 '13 at 9:45

Carlos
101 1 4

9

The following is what I got from one of the articles on The Code Project. I guess it explains everything needed clearly.

> A thread is another mechanism for splitting the workload into separate execution streams. A thread is lighter weight than a process. This means, it offers less flexibility than a full blown process, but can be initiated faster because there is less for the Operating System to set up. When a program consists of two or more threads, all the threads share a single memory space. Processes are given separate address spaces. all the threads share a single heap. But each thread is given its own stack.

edited Dec 17 '16 at 13:38
Peter Mortensen
24.4k 19 89 118

answered Feb 21 '13 at 17:03
Carthi
247 3 6

1 Not sure if this is clear, unless coming from a perspective that already understands threads vs processes. Adding in how they relate to each other might be useful. – Smithers Jan 20 '15 at 18:01

8

From the point of view of an interviewer, there are basically just 3 main things that I want to hear, besides obvious things like a process can have multiple threads:

1. Threads share same memory space, which means a thread can access memory from other's thread memory. Processes normally can not.
2. Resources. Resources (memory, handles, sockets, etc) are release at process termination, not thread termination.
3. Security. A process has a fixed security token. A thread, on the other hand, can impersonate different users/tokens.

If you want more, Scott Langham's response pretty much covers everything. All these are from the perspective of an operating system. Different languages can implement different concepts, like tasks, light-wigh threads and so on, but they are just ways of using threads (of fibers on Windows). There are no hardware and software threads. There are hardware and software **exceptions** and **interrupts**, or user-mode and kernel **threads**.

edited Apr 1 '15 at 8:23

answered Feb 13 '15 at 9:24
AndreiM
627 6 14

8

1. A thread runs in a shared memory space, but a process runs in a separate memory space
2. A thread is a light-weight process, but a process is a heavy-weight process.
3. A thread is a subtype of process.

edited Dec 17 '16 at 13:33
Peter Mortensen
24.4k 19 89 118

answered Dec 21 '12 at 4:22
Sushil kumar
97 1 1

7

Process:

Process is basically a program in execution. It is an active entity. Some operating systems use the term 'task' to refer to a program that is being executed. A process is always stored in the main memory also termed as the primary memory or random access memory. Therefore, a process is termed as an active entity. It disappears if the machine is rebooted. Several process may be associated with a same program. On a multiprocessor system, multiple processes can be executed in parallel. On a uni-processor system, though true parallelism is not achieved, a process scheduling algorithm is applied and the processor is scheduled to execute each process one at a time yielding an illusion of concurrency. Example: Executing multiple instances of the 'Calculator' program. Each of the instances are termed as a process.

Thread:

A thread is a subset of the process. It is termed as a 'lightweight process', since it is similar to a real process but executes within the context of a process and shares the same resources allotted to the process by the kernel. Usually, a process has only one thread of control – one set of machine instructions executing at a time. A process may also be made up of multiple threads of execution that execute instructions concurrently. Multiple threads of control can exploit the true parallelism possible on multiprocessor systems. On a uni-processor system, a thread scheduling algorithm is applied and the processor is scheduled to run each thread one at a time. All the threads running within a process share the same address space, file descriptors, stack and other process related attributes. Since the threads of a process share the same memory, synchronizing the access to the shared data withing the process gains unprecedented importance.

ref-https://practice.geeksforgeeks.org/problems/difference-between-process-and-thread

edited Aug 19 at 8:40
marsze
6,509 3 22 44

answered Nov 27 '18 at 10:03
Lovekush Vishwakarma
1,615 15 18

6

Coming from the embedded world, I would like to add that the concept of processes only exists in "big" processors (*desktop CPUs, ARM Cortex A-9*) that have MMU (memory management unit) , and operating systems that support using MMUs (such as *Linux*). With small/old processors and microcontrollers and small RTOS operating system (*real time operating system*), such as freeRTOS, there is no MMU support and thus no processes but only threads.

Threads can access each others memory, and they are scheduled by OS in an interleaved manner so they appear to run in parallel (or with multi-core they really run in parallel).

Processes, on the other hand, live in their private sandbox of virtual memory, provided and guarded by MMU. This is handy because it enables:

1. keeping buggy process from crashing the entire system.
2. Maintaining security by making other processes data invisible and unreachable. The actual work inside the process is taken care by one or more threads.

edited Jan 13 '16 at 7:10
Anand M Arora
342 3 8

answered Oct 10 '15 at 20:23
Divergence
61 1 1

6

1. Basically, a thread is a part of a process without process thread wouldn't able to work.
2. A thread is lightweight whereas the process is heavyweight.
3. communication between process requires some Time whereas thread requires less time.
4. Threads can share the same memory area whereas process lives in separate.

answered Jul 20 '17 at 12:59
dinesh sharma
182 1 13

6

Process: program under execution is known as process

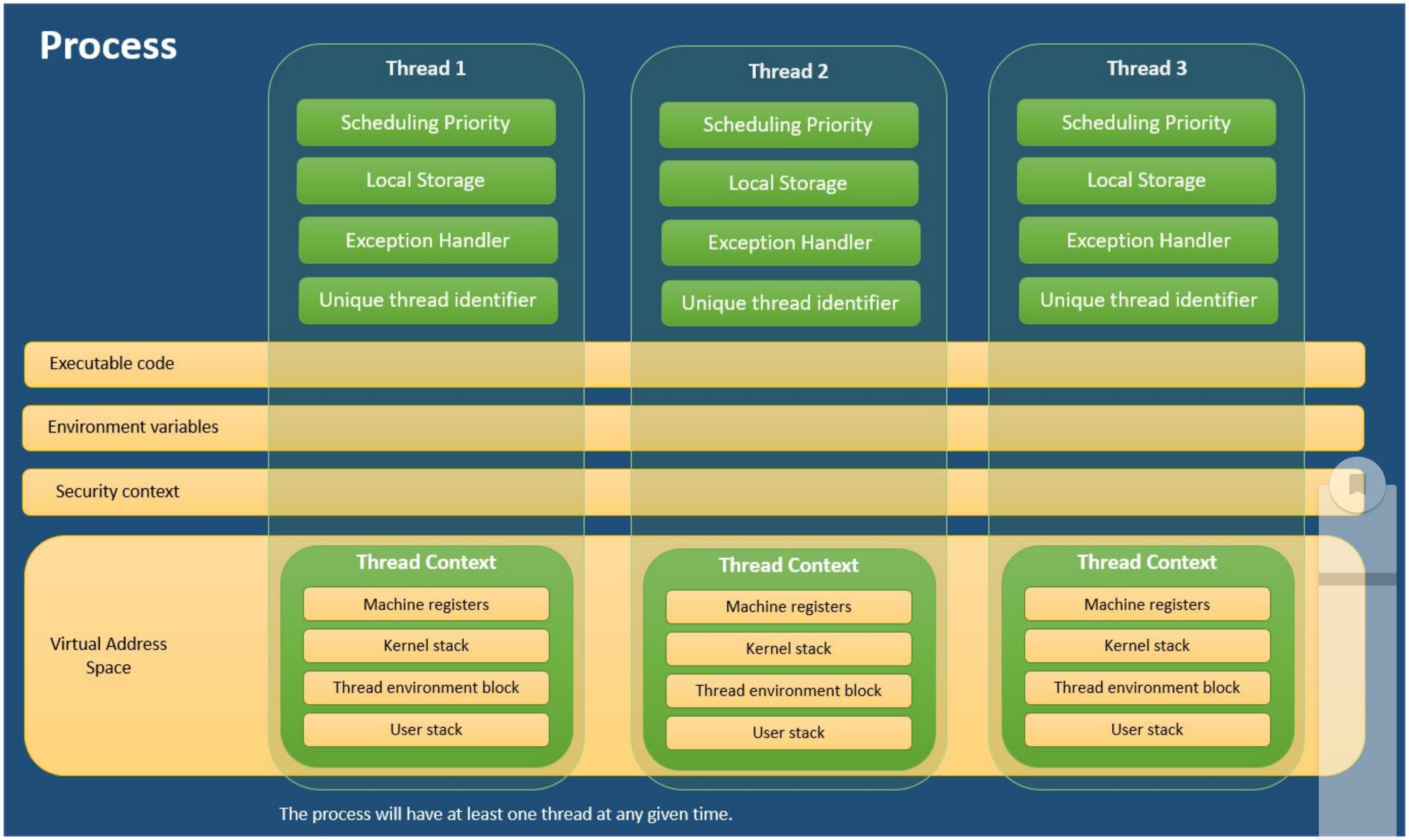
Thread: Thread is a functionality which is executed with the other part of the program based on the concept of "one with other"so thread is a part of process..

edited Apr 11 '18 at 18:22
Alf Moh
4,862 3 26 38

answered Dec 24 '12 at 7:19
saidesh kilaru
548 2 9 17

6

For those who are more comfortable with learning by visualizing, here is a handy diagram I created to explain Process and Threads. I used the information from MSDN - About Processes and Threads



answered Apr 15 '18 at 12:12

Saurabh R S

2,494 1 23 35

6

Trying to answer it from Linux Kernel's OS View

A program becomes a process when launched into memory. A process has its own address space meaning having various segments in memory such as `.text` segement for storing compiled code, `.bss` for storing uninitialized static or global variables, etc.
Each process would have its own program counter and user-space *stack*.

Inside kernel, each process would have its own kernel stack (which is separated from user space stack for security issues) and a structure named `task_struct` which is generally abstracted as the process control block, storing all the information regarding the process such as its priority, state,(and a whole lot of other chunk).
A process can have multiple threads of execution.

Coming to threads, they reside inside a process and share the address space of the parent process along with other resources which can be passed during thread creation such as filesystem resources, sharing pending signals, sharing data(variables and instructions) therefore making threads lightweight and hence allowing faster context switching.

Inside kernel, each thread has its own kernel stack along with the `task_struct` structure which defines the thread. Therefore kernel views threads of same process as different entities and are schedulable in themselves. Threads in same process share a common id called as thread group id(`tgid`), also they have a unique id called as the process id (`pid`).

edited Nov 15 at 7:35

answered Aug 14 '16 at 17:47
Rachit Tayal
433 4 12

5

While building an algorithm in Python (interpreted language) that incorporated multi-threading I was surprised to see that execution time was not any better when compared to the sequential algorithm I had previously built. In an effort to understand the reason for this result I did some reading, and believe what I learned offers an interesting context from which to better understand the differences between multi-threading and multi-processes.

Multi-core systems may exercise multiple threads of execution, and so Python should support multi-threading. But Python is not a compiled language and instead is an interpreted language[1]. This means that the program must be interpreted in order to run, and the interpreter is not aware of the program before it begins execution. What it does know, however, are the rules of Python and it then dynamically applies those rules. Optimizations in Python must then be principally optimizations of the interpreter itself, and not the code that is to be run. This is in contrast to compiled languages such as C++, and has consequences for multi-threading in Python. Specifically, Python uses the Global Interpreter Lock to manage multi-threading.

On the other hand a compiled language is, well, compiled. The program is processed "entirely", where first it is interpreted according to its syntactical definitions, then mapped to a language agnostic intermediate representation, and finally linked into an executable code.

This process allows the code to be highly optimized because it is all available at the time of compilation. The various program interactions and relationships are defined at the time the executable is created and robust decisions about optimization can be made.

In modern environments Python's interpreter must permit multi-threading, and this must both be safe and efficient. This is where the difference between being an interpreted language versus a compiled language enters the picture. The interpreter must not to disturb internally shared data from different threads, while at the same time optimizing the use of processors for computations.

As has been noted in the previous posts both a process and a thread are independent sequential executions with the primary difference being that memory is shared across multiple threads of a process, while processes isolate their memory spaces.

In Python data is protected from simultaneous access by different threads by the Global Interpreter Lock. It requires that in any Python program only one thread can be executed at any time. On the other hand it is possible to run multiple processes since the memory for each process is isolated from any other process, and processes can run on multiple cores.

[1] Donald Knuth has a good explanation of interpretive routines in The Art of Computer Programming: Fundamental Algorithms.

answered Jul 17 '15 at 18:55

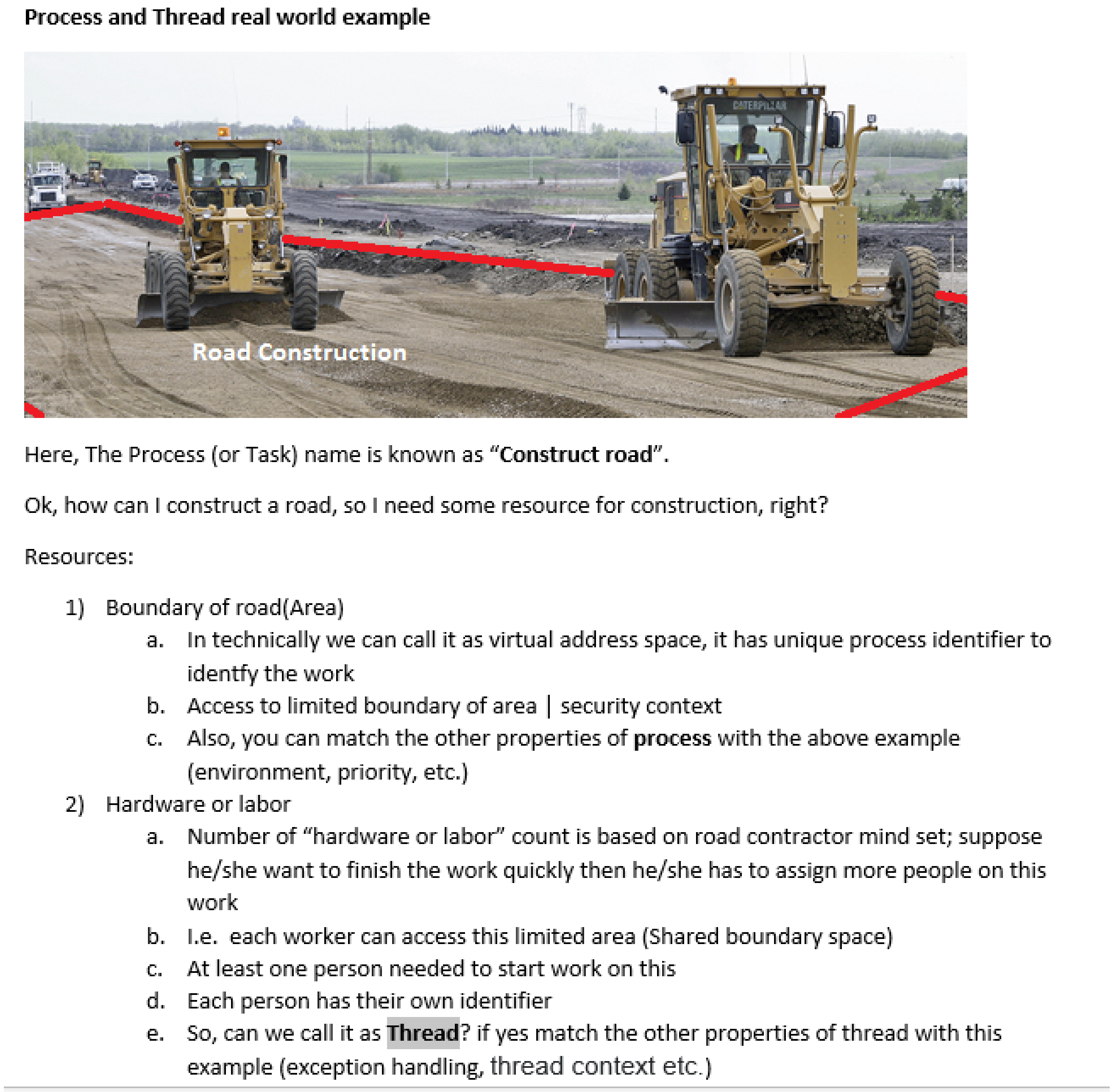
Aaron

609 7 14

5

I've perused almost all answers there, alas, as an undergraduate student taking OS course currently I can't comprehend thoroughly the two concepts. I mean most of guys read from some OS books the differences i.e. threads are able to access to global variables in the transaction unit since they make use of their process' address space. Yet, the newly question arises why there are processes, cognizantly we know already threads are more lightweight vis-à-vis processes. Let's glance at the following example by making use of the image excerpted from [one of the prior answers](),

We have 3 threads working at once on a word document e.g. [Libre Office](). The first does spellchecking by underlining if the word is misspelt. The second takes and prints letters from keyboard. And the last does save document in every short times not to lose the document worked at if something goes wrong. *In this case, the 3 threads cannot be 3 processes since they share a common memory which is the address space of their process and thus all have access to the document being edited.* So, the road is the word document along with two bulldozers which are the threads though one of them is lack in the image.



Process and Thread real world example

Here, The Process (or Task) name is known as "**Construct road**".

Ok, how can I construct a road, so I need some resource for construction, right?

Resources:

1) Boundary of road(Area)
    a. In technically we can call it as virtual address space, it has unique process identifier to identfy the work
    b. Access to limited boundary of area | security context
    c. Also, you can match the other properties of **process** with the above example (environment, priority, etc.)
2) Hardware or labor
    a. Number of "hardware or labor" count is based on road contractor mind set; suppose he/she want to finish the work quickly then he/she has to assign more people on this work
    b. I.e. each worker can access this limited area (Shared boundary space)
    c. At least one person needed to start work on this
    d. Each person has their own identifier
    e. So, can we call it as **Thread**? if yes match the other properties of thread with this example (exception handling, thread context etc.)

answered Mar 12 at 15:41
snr
8,239 2 27 51

4

Threads within the same process share the Memory, but each thread has its own stack and registers, and threads store thread-specific data in the heap. Threads never execute independently, so the inter-thread communication is much faster when compared to inter-process communication.

Processes never share the same memory. When a child process creates it duplicates the memory location of the parent process. Process communication is done by using pipe, shared memory, and message parsing. Context switching between threads is very slow.

edited Apr 19 '16 at 2:24
Mogsdad
36k 12 112 223

answered Apr 19 '16 at 2:01
S.Adikaram
298 2 12

3

They are almost as same... But the key difference is a thread is lightweight and a process is heavy-weight in terms of context switching, work load and so on.

edited Dec 17 '16 at 13:39
Peter Mortensen
24.4k 19 89 118

answered Apr 27 '14 at 21:27
Nasir Ul Islam Butt
109 1 7

3 Thread is a sub-process,they share common resources like code,data ,files within a process.Whereas the two processes cant share resources (Exceptions are if a process(parent) fork to make another process(child) then by default they can share resources.),demands high payload to resources to CPU whereas threads are much lighter in this Context .Although both posses same things.Scenario,consider a single threaded process is blocked due to an I/0,then the whole 1 will go to the waiting state but when multithreaded process is blocked by i/o,then its only 1 i/o concerned thread will be blocked . – Nasir Ul Islam Butt May 10 '14 at 18:40

3

The best answer I've found so far is Michael Kerrisk's 'The Linux Programming Interface':

> In modern UNIX implementations, each process can have multiple threads of execution. One way of envisaging threads is as a set of processes that share the same virtual memory, as well as a range of other attributes. Each thread is executing the same program code and shares the same data area and heap. However, each thread has it own stack containing local variables and function call linkage information. [LPI 2.12]

This book is a source of great clarity; Julia Evans mentioned its help in clearing up how Linux groups really work in this article.

edited Feb 12 at 5:29

answered Dec 15 '17 at 0:44
Zach Valenta
997 1 9 25

2

Example 1: A JVM runs in a single process and threads in a JVM share the heap belonging to that process. That is why several threads may access the same object. Threads share the heap and have their own stack space. This is how one thread's invocation of a method and its local variables are kept thread safe from other threads. But the heap is not thread-safe and must be synchronized for thread safety.

answered May 13 '14 at 16:34
user3633458
21 1

1 2 next

Highly active question. Earn 10 reputation in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.