Thread

What is Thread?

A thread is a flow of execution through the process code, with its own program counter, system registers and stack.

A thread is also called a lightweight process. Threads provide a way to improve application performance through parallelism.

Thread

Motivation for Threads

1) The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

2) Since they are lighter weight than processes, they are easier and faster to create and destroy than processes. Creating a thread goes 10–100 times faster than creating a process.

3) Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

4) Threads are useful on systems with multiple CPUs, where real parallelism is possible.

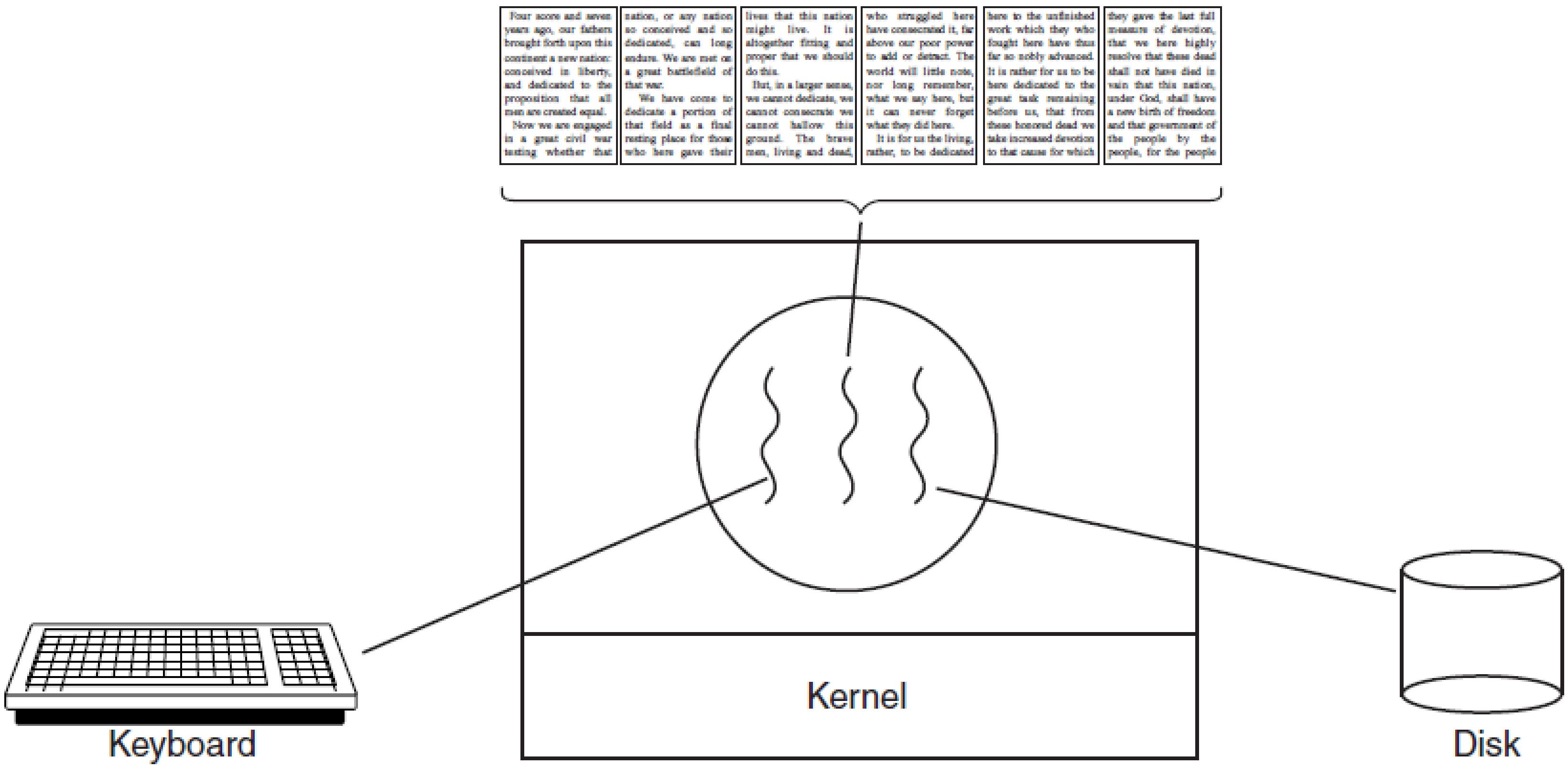
Thread Usage

Example 1:

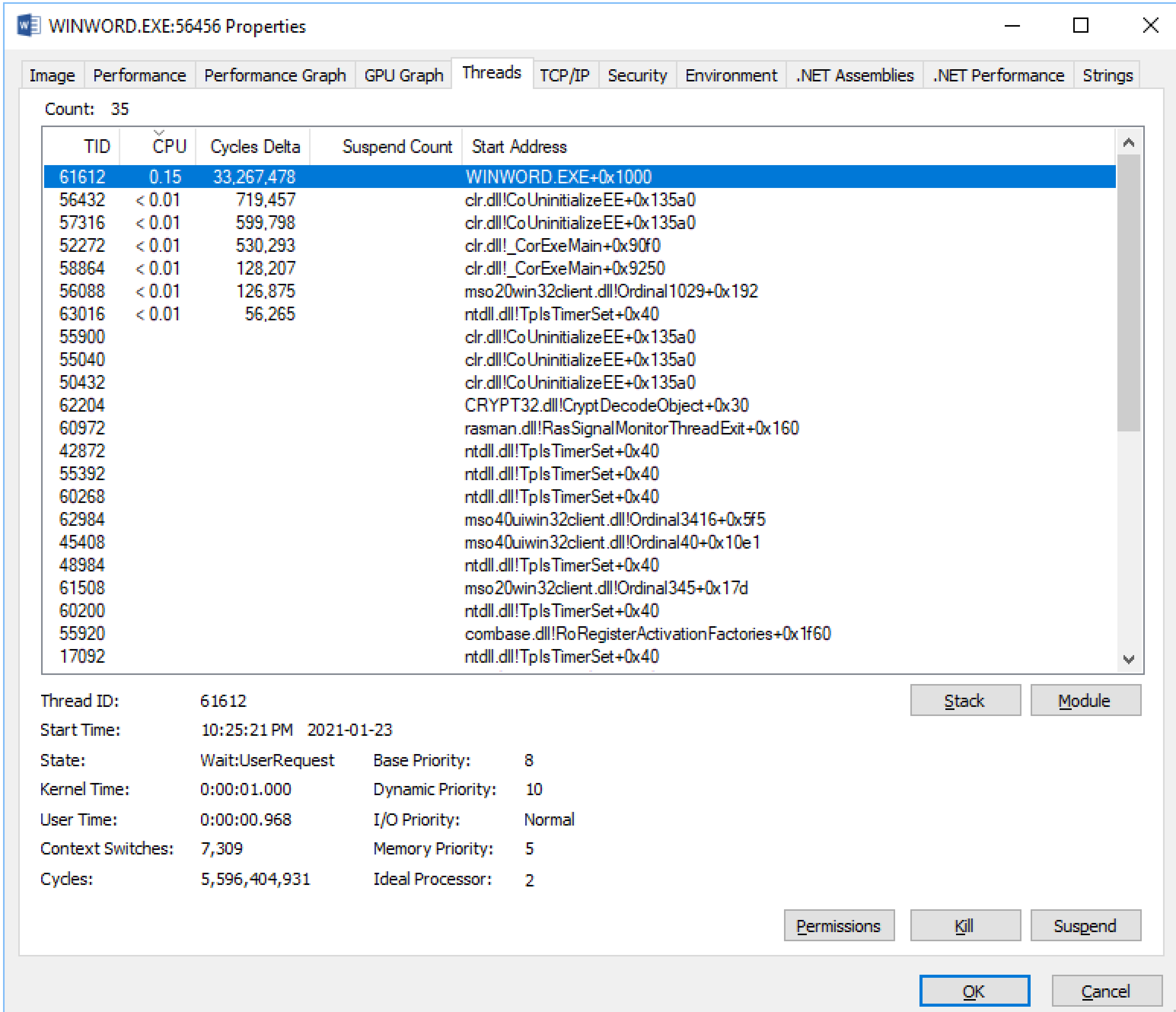
One thread interacts with the user and the other handles reformatting in the background.

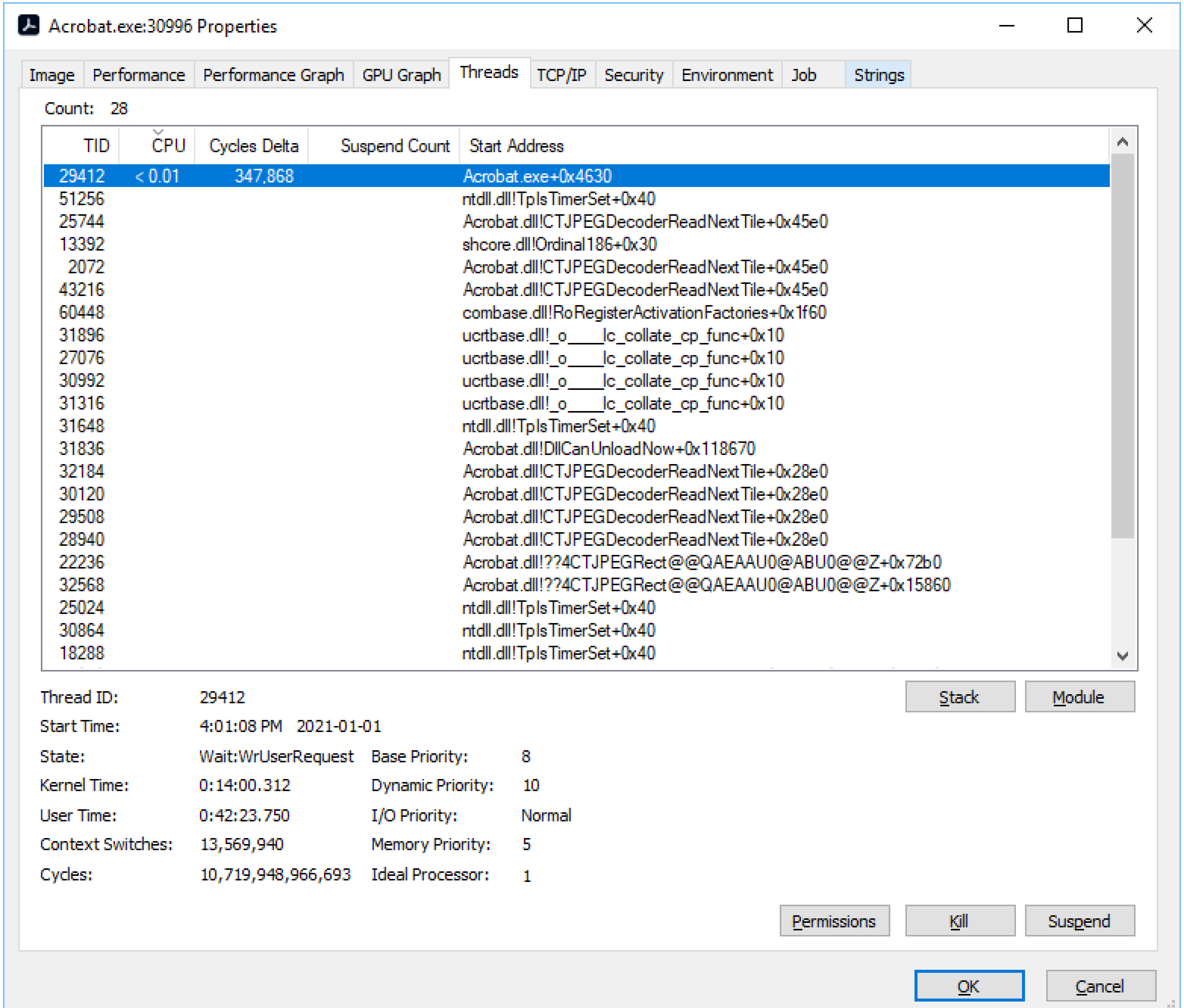
The third thread can handle the disk backups without interfering with the other two e.g. automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure.

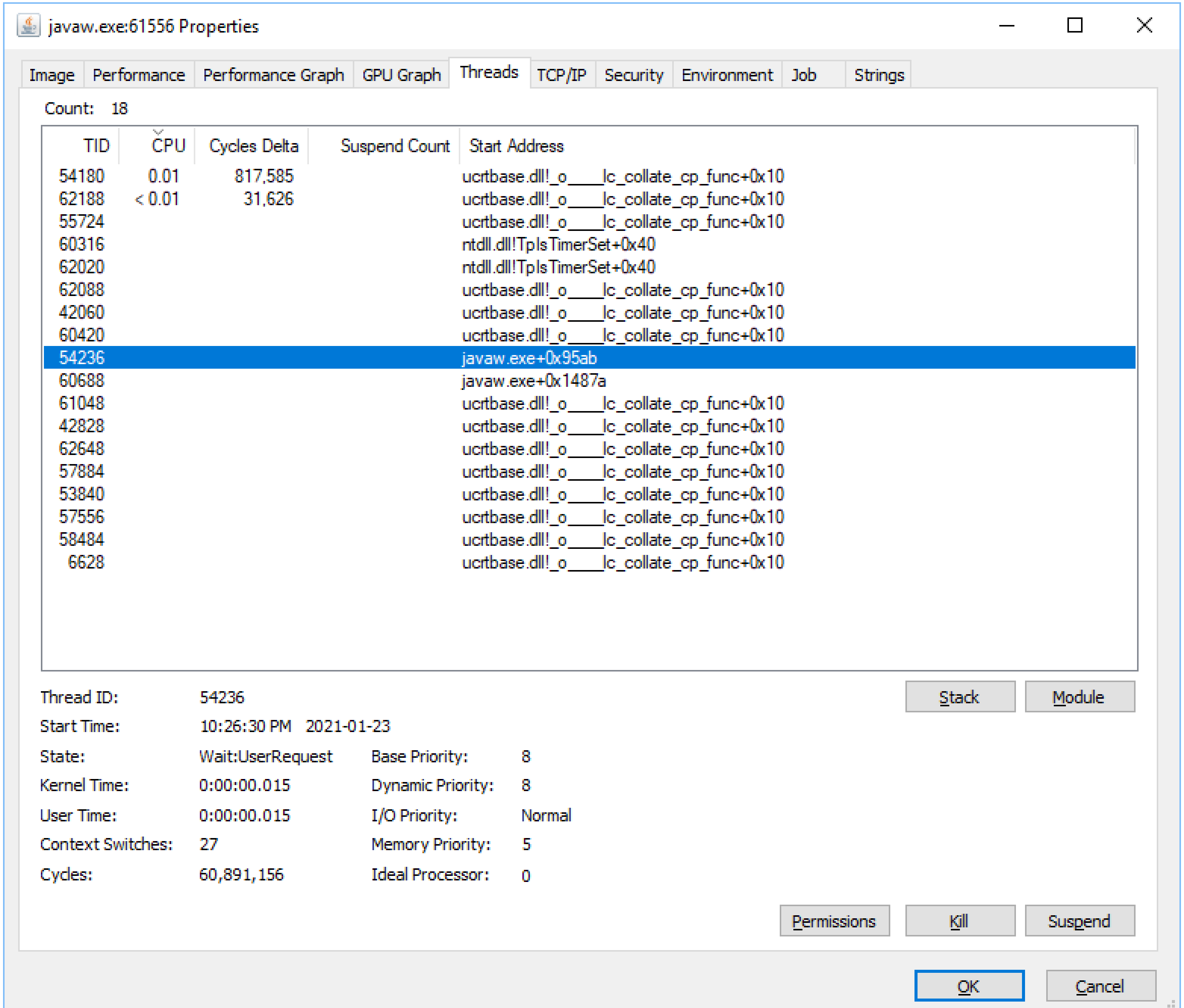
Thread Usage



A word processor with three threads.


WINWORD.EXE:56456 Properties
Image
Performance
Performance Graph
GPU Graph
Threads
TCP/IP
Security
Environment
.NET Assemblies
.NET Performance
Strings
Count: 35
TID CPU Cycles Delta Suspend Count Start Address
61612 0.15 33,267,478 WINWORD.EXE+0x1000
56432 < 0.01 719,457 clr.dll!CoUninitializeEE+0x135a0
57316 < 0.01 599,798 clr.dll!CoUninitializeEE+0x135a0
52272 < 0.01 530,293 clr.dll!_CorExeMain+0x90f0
58864 < 0.01 128,207 clr.dll!_CorExeMain+0x9250
56088 < 0.01 126,875 mso20win32client.dll!Ordinal1029+0x192
63016 < 0.01 56,265 ntdll.dll!TpIsTimerSet+0x40
55900 clr.dll!CoUninitializeEE+0x135a0
55040 clr.dll!CoUninitializeEE+0x135a0
50432 clr.dll!CoUninitializeEE+0x135a0
62204 CRYPT32.dll!CryptDecodeObject+0x30
60972 rasman.dll!RasSignalMonitorThreadExit+0x160
42872 ntdll.dll!TpIsTimerSet+0x40
55392 ntdll.dll!TpIsTimerSet+0x40
60268 ntdll.dll!TpIsTimerSet+0x40
62984 mso40uiwin32client.dll!Ordinal3416+0x5f5
45408 mso40uiwin32client.dll!Ordinal40+0x10e1
48984 ntdll.dll!TpIsTimerSet+0x40
61508 mso20win32client.dll!Ordinal345+0x17d
60200 ntdll.dll!TpIsTimerSet+0x40
55920 combase.dll!RoRegisterActivationFactories+0x1f60
17092 ntdll.dll!TpIsTimerSet+0x40
Thread ID: 61612
Start Time: 10:25:21 PM 2021-01-23
State: Wait:UserRequest
Base Priority: 8
Kernel Time: 0:00:01.000
Dynamic Priority: 10
User Time: 0:00:00.968
I/O Priority: Normal
Context Switches: 7,309
Memory Priority: 5
Cycles: 5,596,404,931
Ideal Processor: 2
Stack
Module
Permissions
Kill
Suspend
OK
Cancel


Acrobat.exe:30996 Properties
Image
Performance
Performance Graph
GPU Graph
Threads
TCP/IP
Security
Environment
Job
Strings
Count: 28
TID
CPU
Cycles Delta
Suspend Count
Start Address
29412 < 0.01 347,868 Acrobat.exe+0x4630
51256 ntdll.dll!TpIsTimerSet+0x40
25744 Acrobat.dll!CTJPEGDecoderReadNextTile+0x45e0
13392 shcore.dll!Ordinal186+0x30
2072 Acrobat.dll!CTJPEGDecoderReadNextTile+0x45e0
43216 Acrobat.dll!CTJPEGDecoderReadNextTile+0x45e0
60448 combase.dll!RoRegisterActivationFactories+0x1f60
31896 ucrtbase.dll!_o____lc_collate_cp_func+0x10
27076 ucrtbase.dll!_o____lc_collate_cp_func+0x10
30992 ucrtbase.dll!_o____lc_collate_cp_func+0x10
31316 ucrtbase.dll!_o____lc_collate_cp_func+0x10
31648 ntdll.dll!TpIsTimerSet+0x40
31836 Acrobat.dll!DllCanUnloadNow+0x118670
32184 Acrobat.dll!CTJPEGDecoderReadNextTile+0x28e0
30120 Acrobat.dll!CTJPEGDecoderReadNextTile+0x28e0
29508 Acrobat.dll!CTJPEGDecoderReadNextTile+0x28e0
28940 Acrobat.dll!CTJPEGDecoderReadNextTile+0x28e0
22236 Acrobat.dll!??4CTJPEGRect@@QAEAAU0@ABU0@@Z+0x72b0
32568 Acrobat.dll!??4CTJPEGRect@@QAEAAU0@ABU0@@Z+0x15860
25024 ntdll.dll!TpIsTimerSet+0x40
30864 ntdll.dll!TpIsTimerSet+0x40
18288 ntdll.dll!TpIsTimerSet+0x40
Thread ID: 29412
Stack
Module
Start Time: 4:01:08 PM 2021-01-01
State: Wait:WrUserRequest
Base Priority: 8
Kernel Time: 0:14:00.312
Dynamic Priority: 10
User Time: 0:42:23.750
I/O Priority: Normal
Context Switches: 13,569,940
Memory Priority: 5
Cycles: 10,719,948,966,693
Ideal Processor: 1
Permissions
Kill
Suspend
OK
Cancel



javaw.exe:61556 Properties
Image
Performance
Performance Graph
GPU Graph
Threads
TCP/IP
Security
Environment
Job
Strings
Count: 18
TID
CPU
Cycles Delta
Suspend Count
Start Address
54180 0.01 817,585 ucrtbase.dll!_o____lc_collate_cp_func+0x10
62188 < 0.01 31,626 ucrtbase.dll!_o____lc_collate_cp_func+0x10
55724 ucrtbase.dll!_o____lc_collate_cp_func+0x10
60316 ntdll.dll!TpIsTimerSet+0x40
62020 ntdll.dll!TpIsTimerSet+0x40
62088 ucrtbase.dll!_o____lc_collate_cp_func+0x10
42060 ucrtbase.dll!_o____lc_collate_cp_func+0x10
60420 ucrtbase.dll!_o____lc_collate_cp_func+0x10
54236 javaw.exe+0x95ab
60688 javaw.exe+0x1487a
61048 ucrtbase.dll!_o____lc_collate_cp_func+0x10
42828 ucrtbase.dll!_o____lc_collate_cp_func+0x10
62648 ucrtbase.dll!_o____lc_collate_cp_func+0x10
57884 ucrtbase.dll!_o____lc_collate_cp_func+0x10
53840 ucrtbase.dll!_o____lc_collate_cp_func+0x10
57556 ucrtbase.dll!_o____lc_collate_cp_func+0x10
58484 ucrtbase.dll!_o____lc_collate_cp_func+0x10
6628 ucrtbase.dll!_o____lc_collate_cp_func+0x10
Thread ID: 54236
Start Time: 10:26:30 PM 2021-01-23
State: Wait:UserRequest
Base Priority: 8
Kernel Time: 0:00:00.015
Dynamic Priority: 8
User Time: 0:00:00.015
I/O Priority: Normal
Context Switches: 27
Memory Priority: 5
Cycles: 60,891,156
Ideal Processor: 0
Stack
Module
Permissions
Kill
Suspend
OK
Cancel

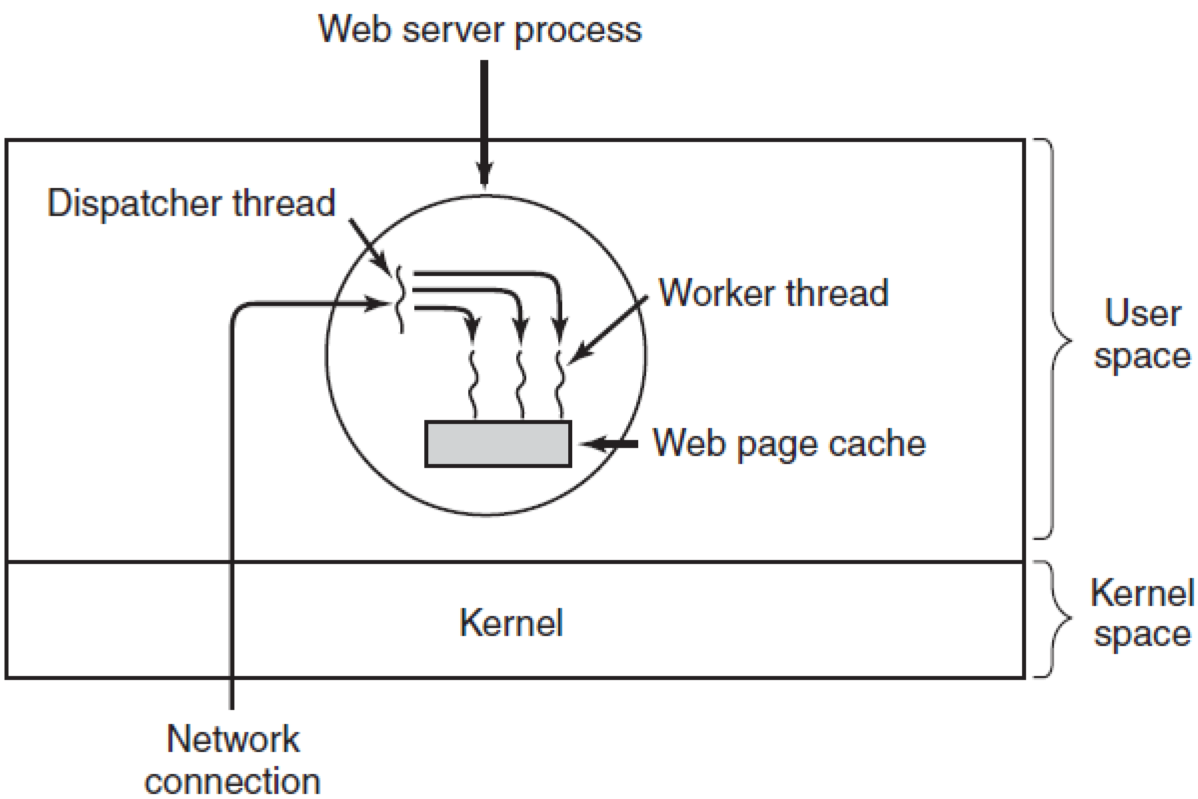
Thread Usage

Example 2:

The **dispatcher thread**, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher.

Thread Usage



A multithreaded Web server.

Thread Usage

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

A rough outline of the code for the previous figure . (a) Dispatcher thread. (b) Worker thread.

Thread Usage

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server.

The Classical Thread Model

The process model is based on two independent concepts: resource grouping and execution(thread)

- A process has an address space containing program text and data, as well as other resources. These resources 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 Classical Thread Model

Different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables.

Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack.

There is no protection between threads because (1) it is impossible, and (2) it should not be necessary

The Classical Thread Model

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	

The first column lists some items shared by all threads in a process.

The second one lists some items private to each thread.

The Classical Thread Model

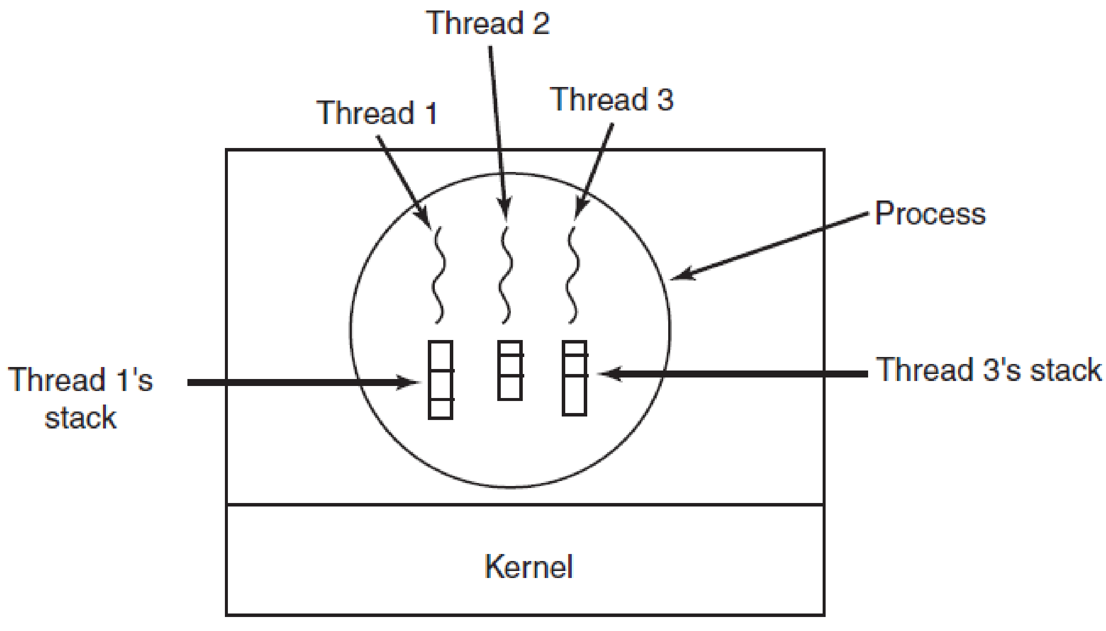
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

The first column lists some items shared by all threads in a process.

The second one lists some items private to each thread.

The Classical Thread Model

Each thread has its own stack



The Classical Thread Model

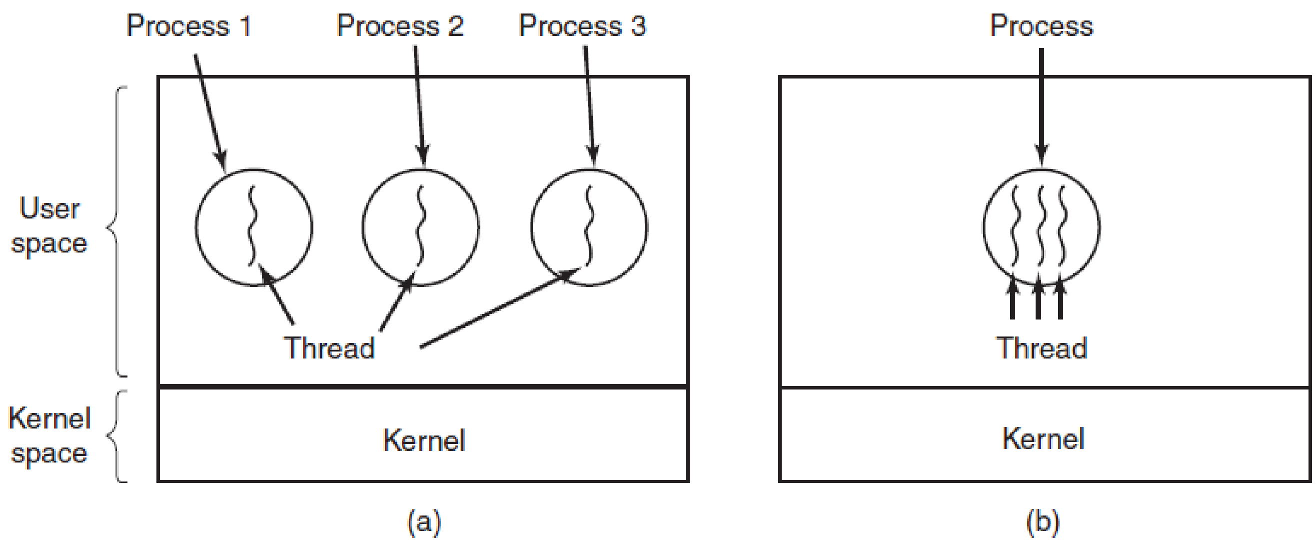
A thread can be in any one of several states: **running**, **blocked**, **ready**, or **terminated**.

- A running thread currently has the CPU and is active.
- A blocked thread is waiting for some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it.
- A ready thread is scheduled to run and will as soon as its turn comes up.

The transitions between thread states are the same as those between process states.

The Classical Thread Model

The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process. In this case the CPU switches rapidly back and forth among the threads, providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one.



(a) Three processes each with one thread.
(b) One process with three threads.

POSIX Threads

The threads package : Pthreads

- Supported by most UNIX systems
- Defines over 60 function calls

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Some of the Pthreads function calls.

POSIX Threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
```

An example program using threads.

POSIX Threads

```
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

An example program using threads.

POSIX Threads

```c
int main()
{
    int thread_number_1 = 1;
    int thread_number_2 = 2;
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, func, &thread_number_1);
    pthread_create(&tid2, NULL, func, &thread_number_2);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

```
~$ gcc main.c -lpthread
~$ ./a.out
Thread #2
Thread #1
Thread #2
Thread #1
Thread #2
```

```
~$
~$ pstree -p | grep a.out
tini(1)---sh(6)---node(7)-+-bash(439)---a.out(1182)-+-{a.out}(1183)
                          |                          `-{a.out}(1184)

~$ ls /proc/1182/task/
1182  1183  1184
~$
```

POSIX Threads

```
GETTID(2)                  Linux Programmer's Manual                 GETTID(2)

NAME
       gettid - get thread identification

SYNOPSIS
       #include <sys/types.h>

       pid_t gettid(void);

DESCRIPTION
       gettid()  returns  the caller's thread ID (TID).  In a single-threaded process,
       the thread ID is equal to the process ID (PID, as returned by getpid(2)).  In a
       multithreaded process, all threads have the same PID, but each one has a unique
       TID.  For further details, see the discussion of CLONE_THREAD in clone(2).

RETURN VALUE
       On success, returns the thread ID of the calling thread.
```

```
~$ ./a.out
Thread #1 ID :  2272
Thread #2 ID :  2273
Thread #1 ID :  2272
Thread #2 ID :  2273
```

```
~$ pstree -p | grep a.out
tini(1)---sh(6)---node(7)-+-bash(439)---a.out(2271)-+-{a.out}(2272)
                          |                         `-{a.out}(2273)
~$
~$
```

Implementing Threads

- Implementing Threads in User Space
- Implementing Threads in the Kernel
- Hybrid Implementations

Implementing Threads

Implementing Threads in User Space

The threads package is put entirely in user space.
The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes.

A user-level threads package can be implemented on an operating system that does not support threads. With this approach, threads are implemented by a library.

The threads run on top of a run-time system, which is a collection of procedures that manage threads e.g. *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_yield*.

Implementing Threads

Implementing Threads in User Space

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system.

- ❖ No trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. <u>This makes thread scheduling very fast</u>.
- ❖ User-level threads also have other advantages. They allow each process to have its own <u>customized scheduling algorithm</u>.
- ❖ <u>They also scale better</u>, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Implementing Threads

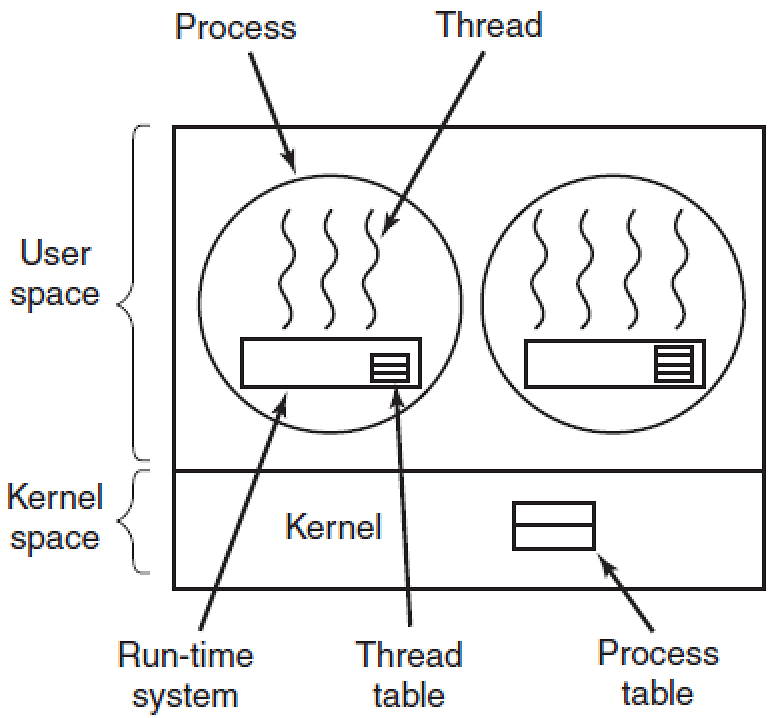
Implementing Threads in User Space

Despite their better performance, user-level threads packages have some major problems.

- The problem of how blocking system calls are implemented. Suppose that a thread reads from the keyboard before any keys have been hit. Letting the thread actually make the system call is unacceptable, since this will stop all the threads.

- Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule processes round-robin fashion (taking turns). Unless a thread enters the run-time system of its own free will, the scheduler will never get a chance.

Implementing Threads in User Space



A user-level threads package.

Implementing Threads

Implementing Threads in the Kernel

- No run-time system is needed in each.
- No thread table in each process.
- The kernel has a thread table that keeps track of all the threads in the system.
- The information is the same as with user-level threads, but now kept in the kernel instead of in user space
- The kernel also maintains the traditional process table to keep track of processes.

Implementing Threads

Implementing Threads in the Kernel

- When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready) or a thread from a different process.

- With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Implementing Threads

Implementing Threads in the Kernel

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk.

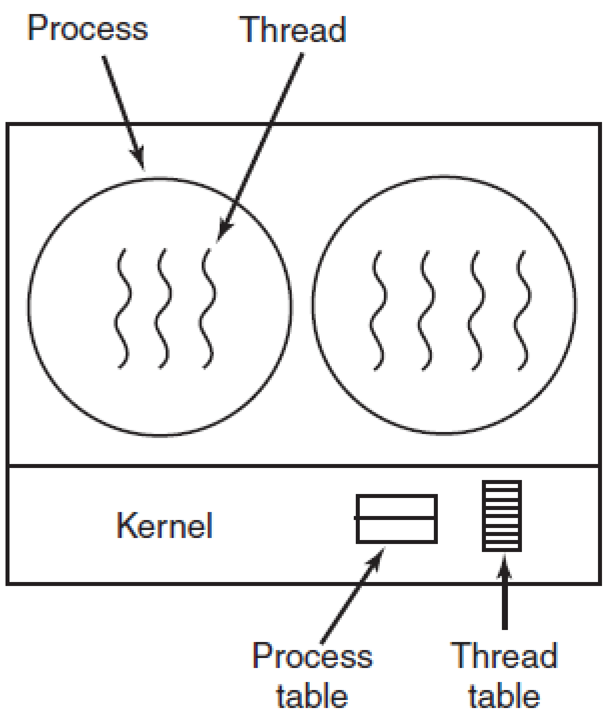
<u>Their main disadvantage is that the cost of a system call is substantial</u>, so if thread operations (creation, termination, etc.) a common, much more overhead will be incurred.

Implementing Threads

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Implementing Threads in the Kernel



A threads package managed by the kernel.

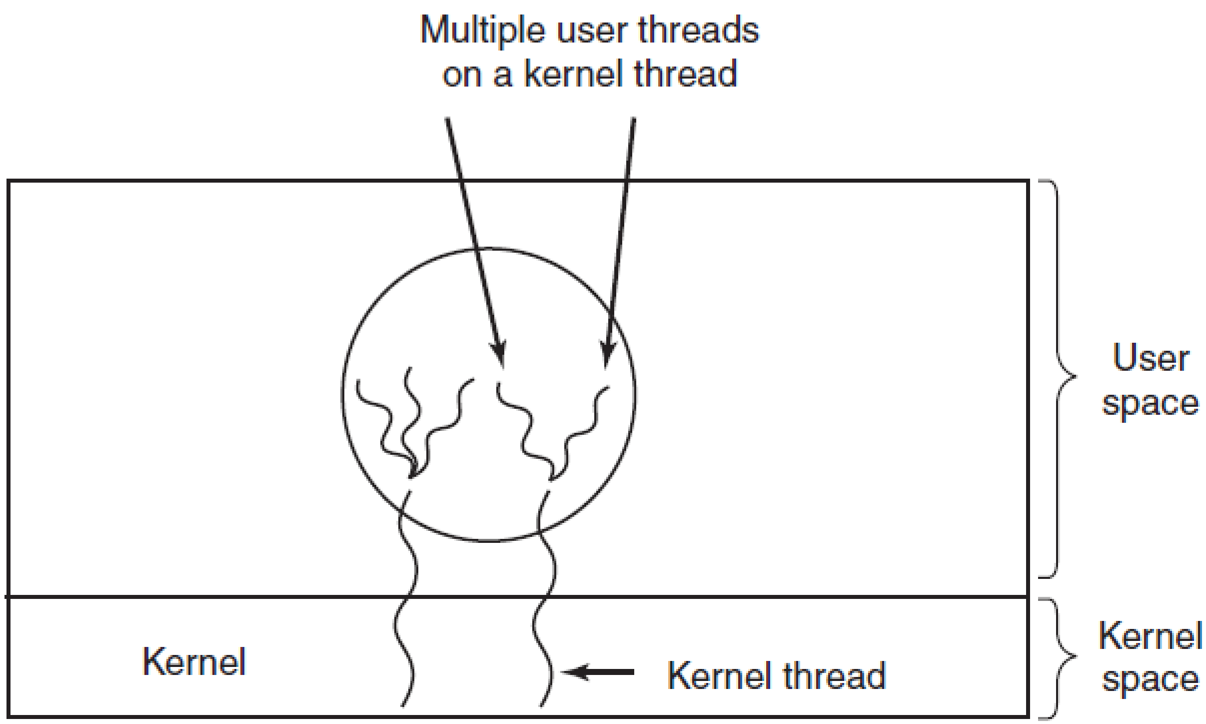
Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user level threads with kernel-level threads.

One way is use kernel-level threads and then multiplex user-level threads onto some or all of them. the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility.

With this approach, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

Hybrid Implementations



Multiplexing user-level threads
onto kernel-level threads.

Hybrid Implementations

- Many to one relationship

  many user threads to one kernel thread

- Many to many relationship

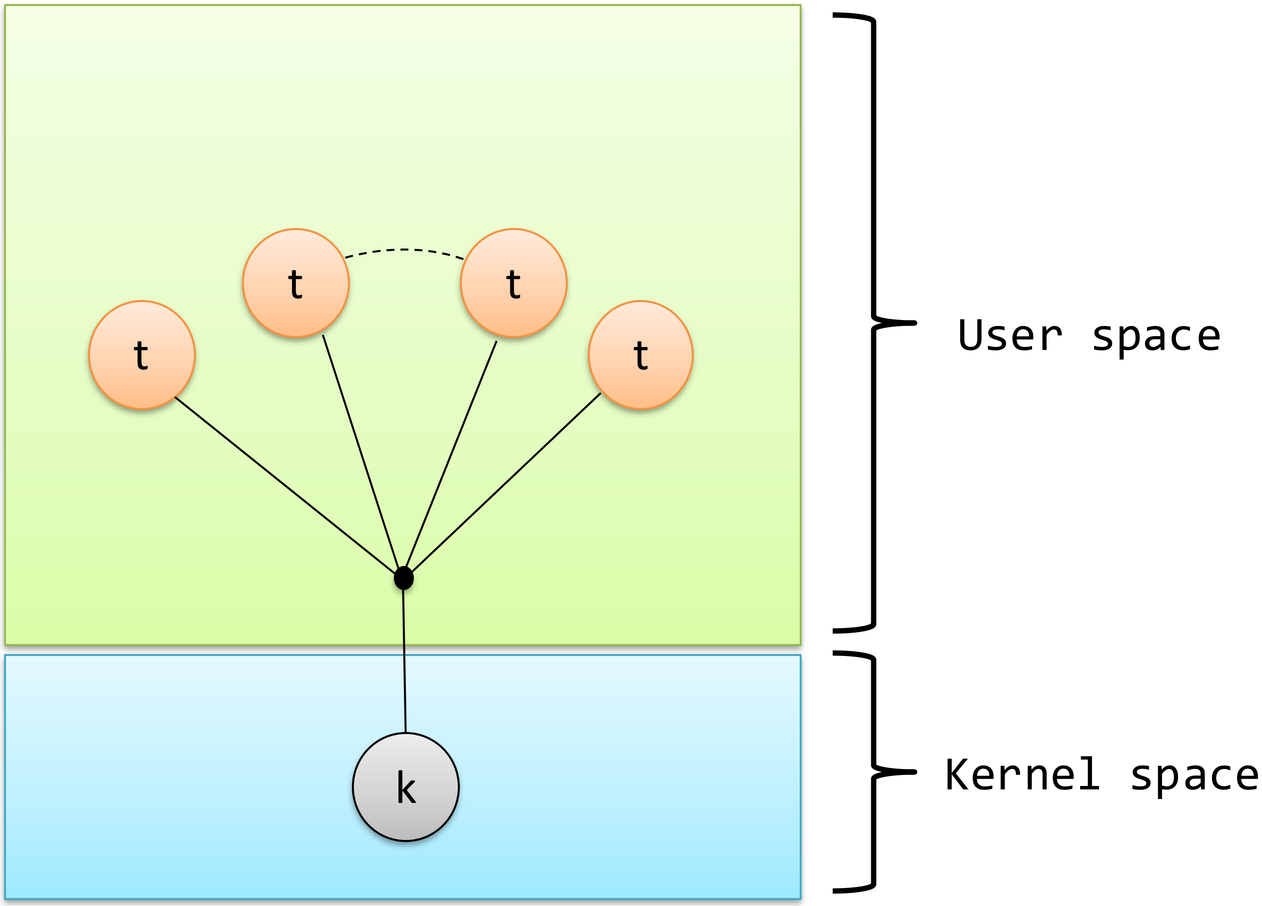
  many user-level threads to many kernel-level threads

- One to one relationship

  one user thread to one kernel thread

Hybrid Implementations

Many to one relationship



Hybrid Implementations

Many to one relationship

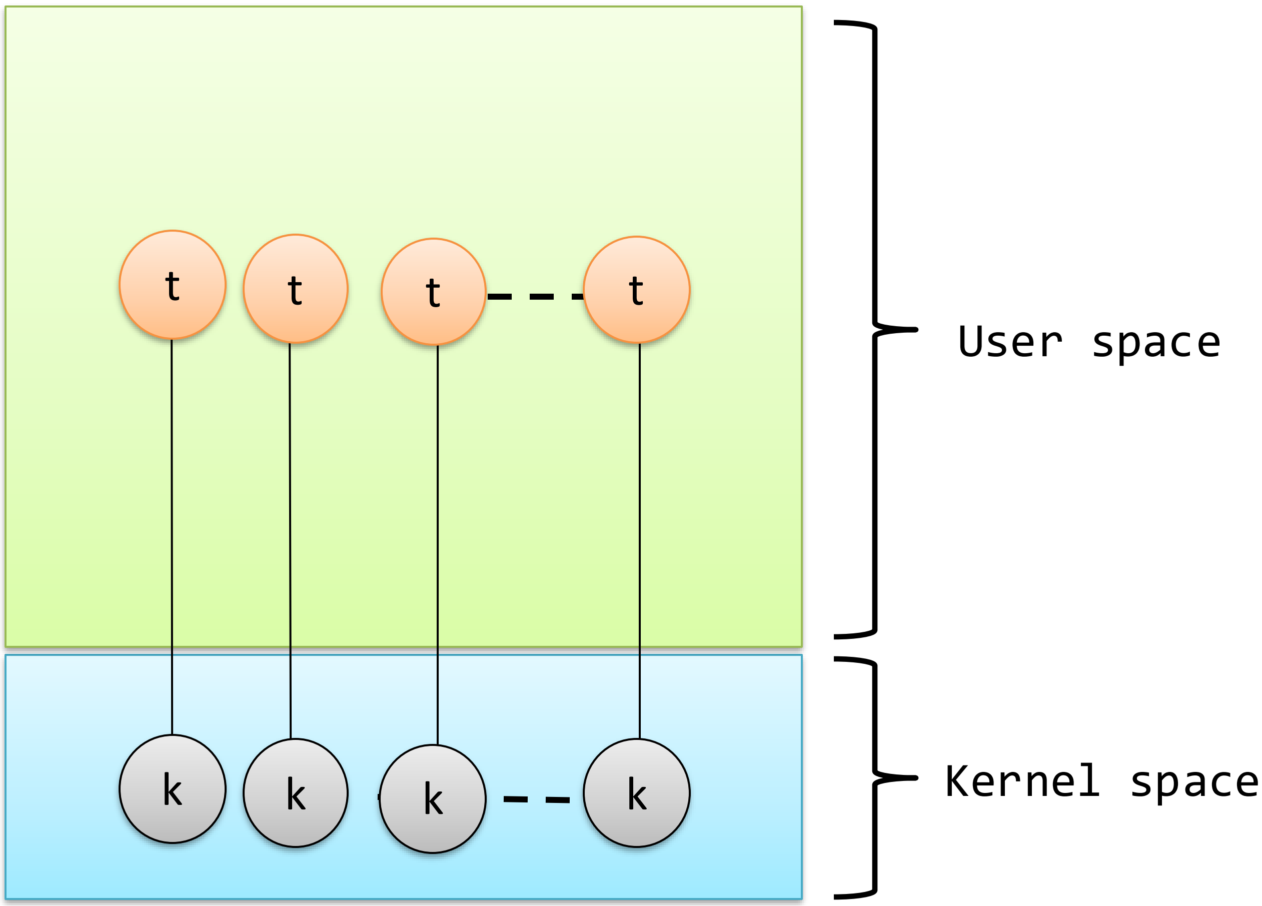
Multiple user threads are mapped onto a single kernel thread.

All user threads within a process share the same kernel thread.
Managed by the user-level thread library, and the kernel remains unaware of individual user threads.

Easy thread creation and management, it suffers from a lack of true concurrency since all threads are bound to a single kernel thread.

Hybrid Implementations

One to one relationship



Hybrid Implementations

One to one relationship

Each user thread is mapped onto a separate kernel thread.

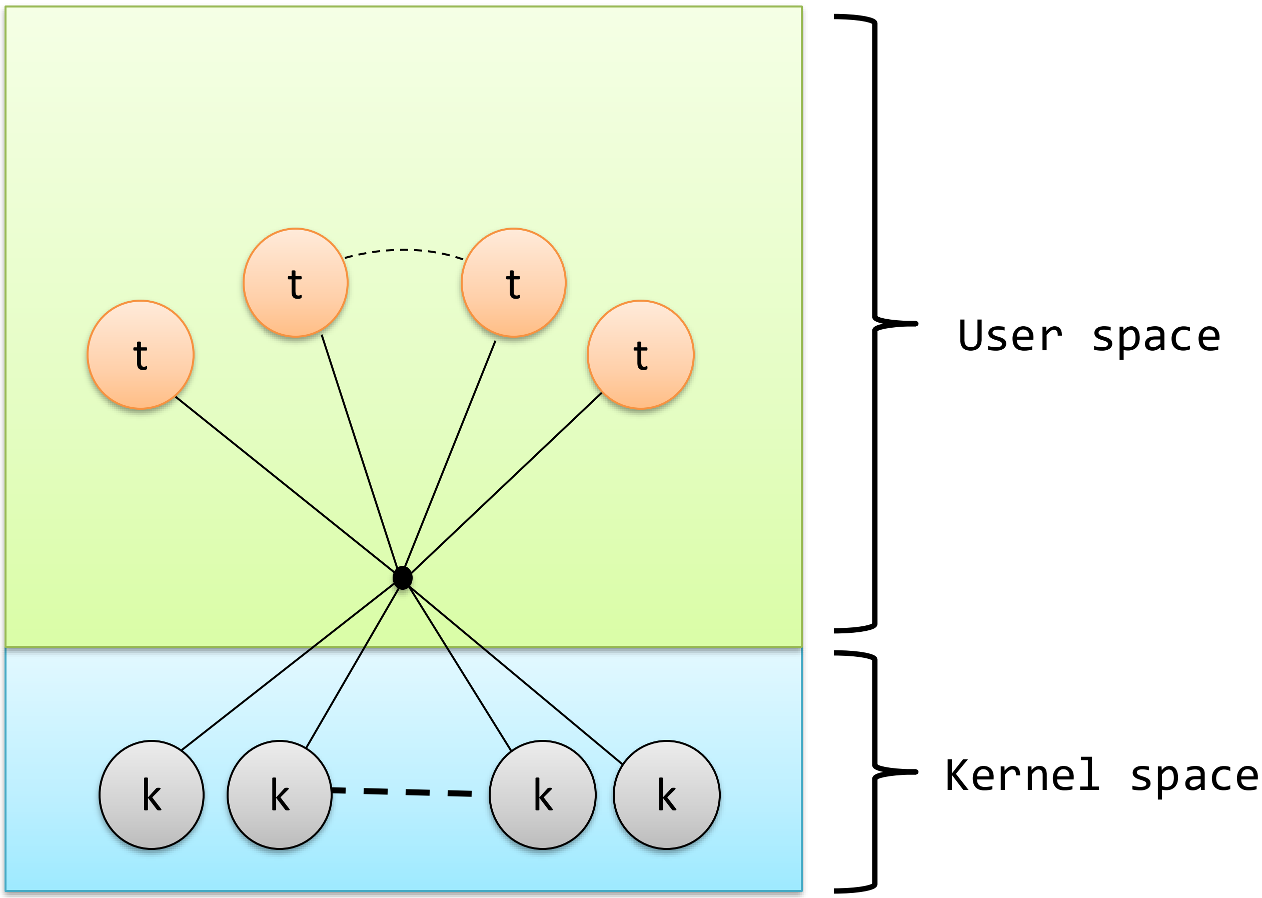
Each user thread has a dedicated kernel thread for execution.

Provides true concurrency as each user thread can be scheduled independently by the kernel.

However, creating and managing many kernel threads can introduce overhead.

Hybrid Implementations

Many to many relationship



Hybrid Implementations

Many to many relationship

Combines aspects of the previous two models.

Multiple user threads are mapped onto an equal or smaller number of kernel threads.

It provides a balance between concurrency and thread creation overhead.

The user-level thread library and the kernel work together to schedule and manage the execution of threads.

Java Threads

Platform Threads

Method 1 (Extending the Thread Class):

```java
class MyThread extends Thread{
        public void run(){
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName);
        }
    }

public class Main {

    public static void main(String[] args) {

        MyThread t1 = new MyThread();
        t1.setName("T1");
        t1.start();
        System.out.println("Hello World");
        MyThread t2 = new MyThread();
        t2.setName("T2");
        t2.start();

    }
}
```

```
Hello World
T1
T2
```

Java Threads

Platform Threads

Method 2 (Implementing the Runnable Interface):

```java
public class Main {
    static class MyRunnable implements Runnable {
        @Override
        public void run(){
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName);
        }
    }
    public static void main(String[] args) {

        Thread t1 = new Thread( new MyRunnable());
        t1.start();
        System.out.println("Hello World");
        Thread t2 = new Thread( new MyRunnable());
        t2.start();
    }
}
```

```
Hello World
Thread-0
Thread-1
```

Java Threads

Platform Threads

Method 3 (using Lambda Expressions):

```
public class Main {

    public static void main(String[] args) {
    Runnable r = () -> {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName);
    };
        Thread t1 = new Thread(r, "T1");
        t1.start();
        System.out.println("Hello World");
        Thread t2 = new Thread(r, "T2");
        t2.start();
    }
}
```

```
Hello World
T1
T2
```

Java Threads

Virtual Threads

```java
public class VThread {
    public static void main(String args[]){
        String name = Thread.currentThread().getName();
        System.out.println(name + ", is virtual ?  " + Thread.currentThread().isVirtual());
        System.out.println(name + ", state ?  " + Thread.currentThread().getState());
        System.out.println(name + ", is alive ?  " + Thread.currentThread().isAlive());
    }
}
```

```
asp@asp-VB:~/Desktop/javaExam$ javac --release 19 --enable-preview  VThread.java
Note: VThread.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
asp@asp-VB:~/Desktop/javaExam$ java  --enable-preview  VThread
main, is virtual ?  false
main, state ?  RUNNABLE
main, is alive ?  true
asp@asp-VB:~/Desktop/javaExam$
```

Java Threads

Virtual Threads

VThread.java

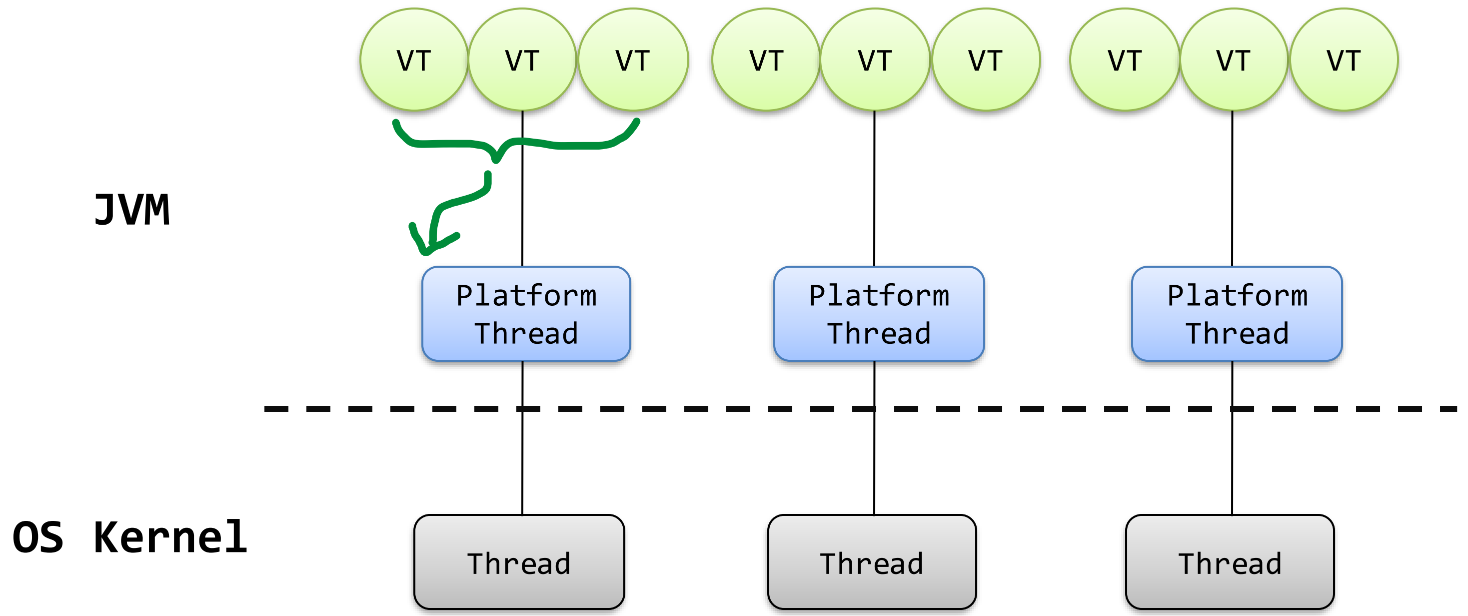
```java
public class VThread {
    public static void main(String args[]){
        Runnable r1 = () ->{
        String name = Thread.currentThread().getName();
        System.out.println(name + ", is virtual ?  " + Thread.currentThread().isVirtual());
        };

        Thread vT1 = Thread.ofVirtual().unstarted(r1);
        vT1.setName("VT");
        vT1.start();
        Thread T2 = new Thread(r1);
        T2.setName("T");
        T2.start();

        try{
            T2.join();
            vT1.join();
        }catch (InterruptedException e){
            e.printStackTrace();
         }
    }
}
```

```
asp@asp-VB:~/Desktop/javaExam$ javac --release 19 --enable-preview  VThread.java
Note: VThread.java uses preview features of Java SE 19.
Note: Recompile with -Xlint:preview for details.
asp@asp-VB:~/Desktop/javaExam$ java  --enable-preview  VThread
VT, is virtual ?  true
T, is virtual ?  false
asp@asp-VB:~/Desktop/javaExam$
```

Java Threads

Platform Threads

1. They are managed by the operating system.
2. Each thread has its own execution context, including its own stack, program counter, and registers.
3. Creating and managing threads can be resource-intensive, especially when a large number of threads are involved.
4. Threads are heavyweight in terms of resource usage, which can limit the scalability of applications that require a high degree of concurrency.



Java Threads

Virtual Threads

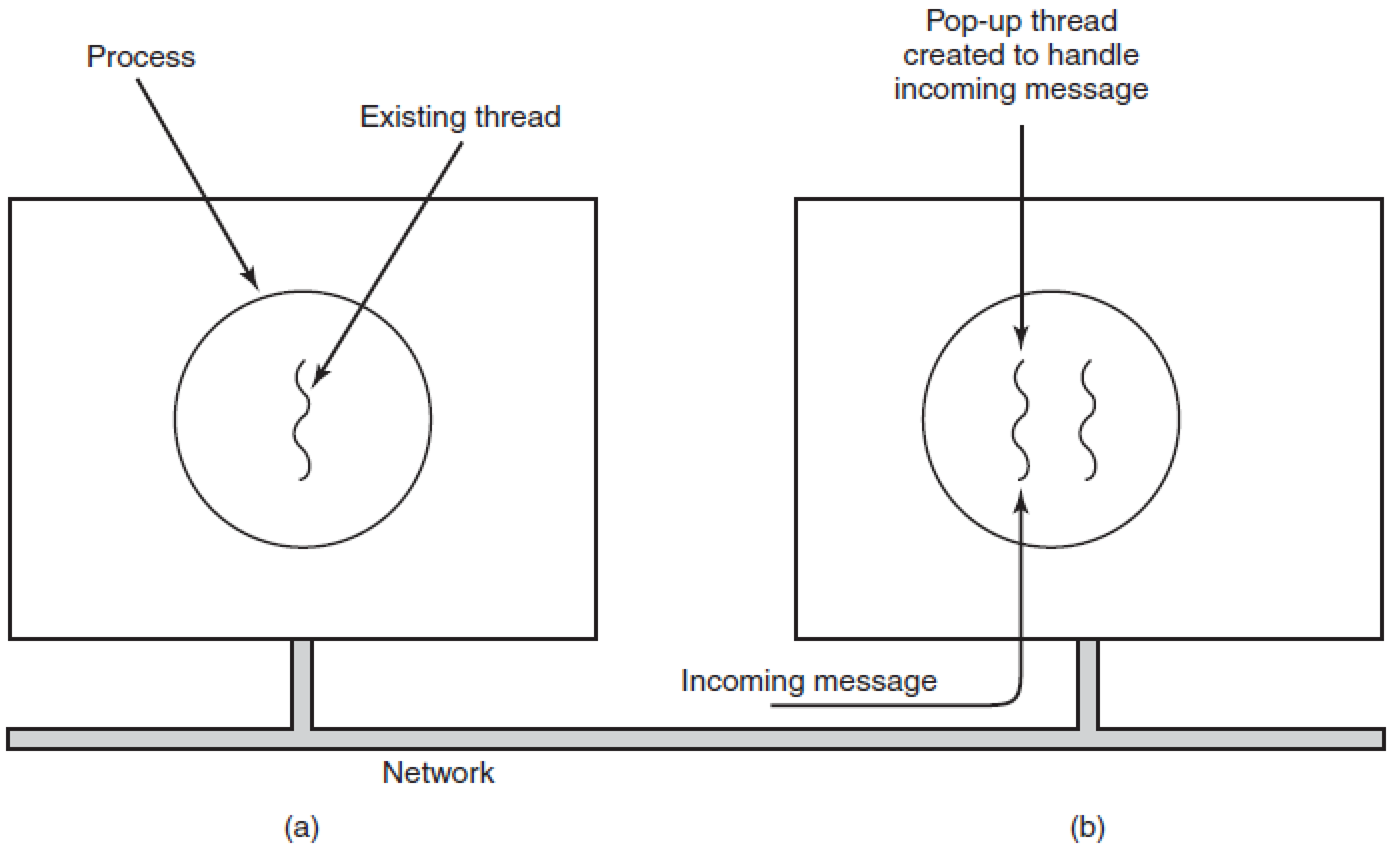
1. Virtual threads (also known as "Project Loom" virtual threads) are a new concurrency mechanism introduced as an experimental feature in more recent versions of Java.
2. Virtual threads are lightweight, user-space threads that are managed by the Java Virtual Machine (JVM) itself, rather than relying on native operating system threads.
3. Virtual threads are much more memory-efficient compared to traditional threads, as they share resources such as stacks and registers.
4. They are designed to be easy to create and manage, allowing developers to create large numbers of concurrent tasks without incurring the overhead associated with creating many native threads.
5. Virtual threads are particularly suited for tasks with high concurrency requirements, such as handling network connections or processing asynchronous tasks.
6. While virtual threads provide a more lightweight concurrency model, they are still subject to the same programming models as traditional threads and require proper synchronization to ensure thread safety.

Pop-Up Threads

The arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a <u>pop-up thread</u>. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.

A key advantage of pop-up threads is that since <u>they are brand new, they do not have any history—registers, stack, whatever—that must be restored</u>. Each one starts out fresh and each one is identical to all the others. This makes it possible to create such a thread quickly.

Pop-Up Threads



Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.