1.

**Classical flow model**

The process model is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them from each other, and here the streams come to the fore. The process is a way of grouping interdependent resources into a single whole. The process has an address space that contains program text and data, as well as other resources. These resources can include open files, raw alarms, signal handlers, accounting information, etc. Managing these resources can be greatly facilitated by combining them as a process.

Threads add to the process model the ability to implement several, largely independent of each other, tasks in a single process environment. Threads use a single address space, but different resources.

Like a traditional process (ie a process with only one thread), the thread must be in one of the following states: executable, blocked, ready, or completed.

Note that each thread has its own stack, as shown in Fig. The stack of each thread contains one frame for each procedure already called but not yet completed. This frame contains local procedure variables and the control return address when it is called. Each thread will usually call different procedures so each thread needs its own stack.

Some difficulties that arise when working with threads:

a. If a parent process has multiple threads, should the child process have them?

b. What happens if any of the parent process threads have been blocked by a

read system call used, for example, to read from the keyboard? Will there now be two threads, in the parent and child processes, blocked on keyboard input? If a string is entered, will both threads receive a copy?

3. Another class of problems is that threads share many data structures. What happens if one thread closes a file at a time when another thread has not yet read data from it?

4. Suppose that one thread noticed a lack of free memory and began to allocate additional volume. Halfway there is a switching of threads, and the new thread also notices a lack of free memory and begins to allocate additional volume. It is possible that additional memory will be allocated twice.

2.

**POSIX**

POSIX Threads or Pthread is a POSIX thread implementation standard that defines APIs for creating and managing threads.

Pthreads defines a set of data types, functions, and constants in the C programming language format. They are described in the pthread.h header file and implemented as a library.

All Pthreads procedures have names with the prefix "pthread\_" and can be divided into 4 categories by purpose:

• Flow management - creating, merging flows, etc .;

• Mutexes;

• Conditional variables;

• Synchronize streams using locks and barriers to reading / writing data.

----------------------------------------------------------------------------------------------------

The main functions of the standard

• Data types:

-pthread\_t: flow descriptor

-pthread\_attr\_t: a set of flow attributes

• Flow management functions:

-pthread\_create (): create a stream

-pthread\_exit (): end the thread

-pthread\_cancel (): cancel flow

-pthread\_join (): block a thread until the end of another thread specified in the function call

-pthread\_detach (): release resources occupied by the flow (if the flow is running, the release of resources will occur after its completion)

-pthread\_attr\_init (): initialize the structure of the flow attributes

-pthread\_attr\_setdetachstate (): an indication to the system that it can free up resources occupied by the thread after the thread completes

-pthread\_attr\_destroy (): freeing memory from the structure of stream attributes (destroy handle)

• Stream synchronization functions:

-pthread\_mutex\_init (), pthread\_mutex\_destroy (), pthread\_mutex\_lock (), pthread\_mutex\_trylock (), pthread\_mutex\_unlock (): using mutexes

-pthread\_cond\_init (), pthread\_cond\_signal, pthread\_cond\_wait (): using conditional variables

3.

**Implementation of flows in user space**

If you place threads in user space, the kernel will know nothing about the existence of threads, it will control normal, single-threaded processes.

The first and most obvious advantage is that a set of user-level threads can be implemented in an operating system that does not support threads.

Threads are run on top of a run-time system, which is a set of procedures that manage flows. We have already considered four of them, these are pthread\_create, pthread\_exit, pthread\_join and pthread\_yield, but usually there are other procedures in the set.

When flow control is performed in user space, each process must have its own flow table to keep track of the flows available in that process. This table is analogous to the process table available in the kernel, except that it contains only the properties that belong to each thread.

The flow table is managed by the program execution support system. When a thread is put on standby or locked, the information needed to resume execution is stored in the thread table, just as the kernel stores process information in the process table. Switching threads done in this way is at least an order of magnitude, and perhaps more, faster than intercepting kernel control, which is a strong argument in favor of a set of threads implemented at the user level.

The procedure that preserves the state of the thread and the scheduler are just local procedures, so calling them is much more efficient than calling the kernel. Among other things, you do not need to intercept kernel control by the trap instruction, you do not need to switch the context, the memory cache does not need to be reset to disk, etc. Due to all this, the thread scheduler works very quickly.

Another advantage of implementing threads at the user level is the ability to give each process its own scheduling algorithm settings.

Problems with the flow of threads in user space:

The complexity of implementing blocking system calls

4.

**Implementing Threads in the Kernel**

Now let us consider having the kernel know about and manage the threads. No run-time system is needed in each, as shown in [FIG. 2-8(b)](about:blank). Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread management overhead is much smaller, there is less incentive to do this.

5.

**Hybrid implementation of threads**

A short-term research study will address the challenge of achieving near PetaOps scale performance within a decade through innovative systems that exhibit wide applicability and ease of use. The objective of the research is to determine the feasibility and detailed structure of a parallel architecture integrating the combined capabilities of semiconductor, superconductor, and optical technologies. This hybrid architecture approach is motivated by the realization that a mix of technologies may yield superior operational attributes to those possible based solely on semiconductor technology in the same time frame. An interdisciplinary team of collaborators has been assembled to conduct the required point-design study in order to devise a complete system structure and evaluate its performance characteristics using a small set of scientific application kernels.

The hybrid technology approach exploits critical opportunities enabled by key emerging devices. Specifically, computational performance can be dramatically improved through recent advances in Superconducting Rapid Single Flux Quantum logic which will make 100 GHz clock rates feasible in the next two years. Memory capacity may be drastically increased through a new memory hierarchy merging advanced semiconductor high density memory and future (admittedly more speculative) optical 3-D holographic storage capable of storing 0.1 Petabits or more. Interconnection bandwidth will be greatly enhanced by means of optical networks with 100 Gbps channels.