**POSIX THREADS**

* Introduction to Threads
* POSIX Threads
* The Pthread API
* Threads vs Processes

Threads

Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently. A single process can contain multiple threads

All threads are independently executing the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments.

The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute parallel. If one thread is blocked on I/O, other threads are still eligible to execute.

THREADS ENABLES USER TO DO CONCURRENT PROGRAMMING WITH MULTIPLE TASKS.

WITH PROCESSES WE USE SO MANY SYSTEM RESOURCES. FOR SOME OF THE APPLICATIONS, WE DON’T HAVE TO USE THAT MUCH OF RESOURCES. ALSO IPC CAUSES A LOT OF WORK.

THREADS CAN EXECUTE CONCURRENTLY ON A MULTIPROCESSOR SYSTEM.

IF ONE OF THE THREADS BLOCKED, OTHER THREADS CAN CONTINUE EXECUTION.

Threads in a Process (USER-LEVEL)

Diagram, table

Description automatically generatedAs can be seen from the figure all of the per-thread stacks reside within the same virtual address space. This means that, given a suitable pointer, it is possible for threads to share data on each others stacks.

This is occasionally useful, but it requires careful programming to handle the dependency that results from the fact that a local variable remains valid only for the lifetime of the stack frame in which it resides.

Failing to correctly handle this dependency can create bugs that are hard to track down.



THREADS CAN USE RED PORTION OF THE PROCESS’S MEMORY

VARIABLES INSIDE STACK FOR THREAD 1 ARE ONLY AVAILABLE FOR THAT THREAD.

KIRMIZI KISMA ERİŞİRKEN CRITICAL SECTION YARATMALISIN. STACKLERİ THREAD LIBRARY HALLEDER.

Threads: Introduction

Threads offer advantages over processes in certain applications.

To give an example consider a network server design in which a parent process accepts incoming connections from clients, and then uses fork() to create a separate child process to handle communication with each client.

FOR EACH CLIENT, WE HAVE OWN MEMORY. EXTRA USE OF SYSTEM RESOURCES. KERNEL DOESN’T KNOW HOW MANY SMALL EXECUTIONS (THREADS) WE HAVE INSIDE THE PROCESS IF WE DIDN’T LET IT KNOW.

While this approach works well for many scenarios, it does have the following limitations in some applications:

* It is difficult to share information between processes. Since the parent and child don’t share memory, we must use some form of interprocess communication in order to exchange information between processes.
* Process creation with fork() is relatively expensive. The need to duplicate various process attributes such as page tables and file descriptor tables means that a fork() call is still time-consuming.

WITH THREADS FOR EACH CLIENT, WE REMOVE THE REQUIREMENT FOR IPC.

Threads: Motivation

Threads address both of these problems:

* Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables. However, in order to avoid the problems that can occur when multiple threads try to update the same information, we must employ some synchronization techniques.
  + IF MULTIPLE THREADS ARE WORKING INSIDE PROCESS, IF ONE OF THEM CHANGES STH INSIDE WHILE OTHER IS TRYING TO READ IT, DATA CORRUPTION MIGHT OCCUR. YOU HAVE TO USE SYNCHRONIZATION.
* Thread creation is faster because many of the attributes that must be duplicated in a child created by fork() are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables
  + WITH THREADS, YOU DON’T HAVE TO GO OUT AND TELL EACH OTHER CHILD PROCESS THAT THIS VALUE HAS BEEN CHANGED WHEN YOU CHANGE A VARIABLE.

Threads : function call vs a Thread

Diagram

Description automatically generated

-When a new thread is created it runs concurrently with the creating process.

-When creating a thread you indicate which function the thread should execute.

EXECUTION OF THREAD IS DONE JUST LIKE THE FUNCTION CALL.

THE WAY RETURN VALUE OF THREAD FUNCTION AND THE ARGUMENTS PASSED INTO THREAD FUNCTION MUST BE FLEXIBLE SO WE USE VOID POINTERS FOR BOTH OF THEM. YOU ARE ASSIGNING YOUR RESOURCE FOR THREAD EXECUTION SO RETURN VALUE SHOULD BE FLEXIBLE.

A function that is used as a thread must have a special format.

It takes a single parameter of type pointer to void and returns a pointer to void.

The parameter type allows any pointer to be passed. This can point to a structure, so in effect, the function can use any number of parameters.

The processfd function used above might have prototype:

void \*processfd(void \*arg);

Instead of passing the file descriptor to be monitored directly, we pass a pointer to it.

A picture containing text

Description automatically generated

ID OF THREAD

ARGUMENTS

IF YOU USE NULL, SYSTEM WILL USE DEFAULT ATTRIBUTES.

IN REGULAR FUNCTION CALL, ALL VALUES CREATED IN FUNCTION WILL BE DISCARDED.

WHEN YOU USE PTHREAD\_CREATE, FROM THAT POINT ON, UP TILL THE TIME YOU REMOVE THAT THREAD FROM THE MEMORY BY EITHER JOINING OR DETACHING IT, WE HAVE 2 EXECUTIONS AND OS, WHEN CONTEXT SWITCH APPEARS, WILL GIVE ONE OF THEM THE RESOURCES. BOTH OF THEM CANNOT USE THE RESOURCES AT THE SAME TIME.

Thread Management

A thread package usually includes functions for thread creation and thread destruction, scheduling, enforcement of mutual exclusion and conditional waiting

A typical thread package also contains a runtime system to manage threads transparently (i.e., the user is not aware of the runtime system).

When a thread is created, the runtime system allocates data structures to hold the thread's ID, stack and program counter value.

The thread's internal data structure might also contain scheduling and usage information. The threads for a process share the entire address space of that process.

* THERE IS A SOME KIND OF SMALL SCHEDULER INSIDE THE PROCESS’ MAIN EXECUTION THAT CONTROLS THE ALLOCATION OF THE STACK MEMORY INSIDE THAT PROCESS
* THREAD LIBRARY SHOULD BE ABLE TO MANAGE THIS SO THAT USER THAT USES LIBRARY AND KERNEL IS UNAWARE OF THIS EXECUTION.
* THIS MANAGER EXECUTION IS ALSO RESPONSIBLE FOR WHICH EXECUTION WILL BE USE CPU RESOURCES WHEN CONTEXT SWITCH HAPPENS. THREAD LIBRARY HANDLES FAIRNESS.

When a thread allocates space for a return value, some other thread IN BACKGROUND is responsible for freeing that space IF THE THREAD DOESN’T FREE SPACE BY ITSELF. Whenever possible, a thread should clean up its own mess (CLOSE FILES, FREE RESOURCES, ETC.) rather than requiring another thread to do it TO REDUCE WORKLOAD ON THIS OTHER THREAD.

When creating multiple threads, do not reuse the variable holding a thread's parameter until you are sure that the thread has finished accessing the parameter. As the variable is passed by reference, it is a good practice to use a separate variable for each thread.

WE DIDN’T SAY THESE VARIABLES ARE STORED IN SPECIFIC WAY INSIDE THE STACK OF PROCESS WITH THREAD. THREAD LIBRARY HAS TO MAKE SURE THAT PORTION OF THE STACK SHOULD BE AVAILABLE TO THAT EXECUTION FOR ITS OWN VARIABLES WHEN THAT THREAD IS CREATED. THAT PORTION SHOULDN’T BE ACCESSIBLE BY THE OTHER EXECUTIONS INSIDE THAT PROCESS. THREAD LIBRARY SHOULD CREATE STACK SPECIFIC FOR THE THREAD WHILE CREATING THE THREAD. ALSO IT SHOULD ENABLE GLOBAL VARIABLES OR DATA AVAILABLE TO EXECUTION WHEN IT IS CREATED. USER AND KERNEL MUST BE UNAWARE OF WHATS GOING ON.

SO THREAD ID IS IMPORTANT. CORRESPONDING TO THIS ID STACK PORTION IS ALLOCATED SPECIFICALLY FOR THE THREAD WITH THAT SPECIFIC ID.

1 CPU VARSA MULTIPROCESS OLDUĞU ZAMAN NASILSA, MULTITHREADED OLDUĞU ZAMAN DA BİRDEN ÇOK İŞ AYNI ANDA YAPILIYORMUŞ GİBİ GÖZÜKÜR HALBUKİ SEQUENTIAL.

BİRDEN FAZLA PROCESSOR VARSA HEPSİ ÜSTÜNDE AYRI AYRI EXECUTION ÇALIŞTIRABİLİRSİN. KERNEL THREADLER ASLINDA ONUN İÇİN VAR.

Thread Safety

A hidden problem with threads is that they may call library functions that are not thread-safe, possibly producing spurious results

A function is thread-safe if multiple threads can execute simultaneous active invocations of the function without interference.

POSIX specifies that all the required functions, including the functions from the standard C library, be implemented in a thread-safe manner except for the specific functions

Those functions whose traditional interfaces preclude making them thread-safe must have an alternative threadsafe version.

Another interaction problem occurs when threads access the same data. YOU HAVE TO USE SYNCHRONIZATION.

In more complicated applications, a thread may not exit after completing its assigned task. Instead, a worker thread may request additional tasks or share information.

MOST OF THE INITIAL FUNCTIONS THAT ARE USED IN STANDARD C LIBRARY ARE NOT THREAD SAFE. WHEN A CONTEXT SWITCHING OCCURS YOU DON’T KNOW WHICH EXECUTION WILL HAVE THE RESOURCES ON THE NEXT TIME WHEN THE PROCESS HAS THE RESOURCES. FOR MOST FUNCTIONS, WHEN CONTEXT SWITCH HAPPENS DURING EXECUTION OF FUNCTION, NEXT TIME FUNCTION IS CALLED, IT HAS TO RESTART TO REMOVE POSSIBILITY OF DATA CORRUPTION.

MOST OF THE STANDARD C LIBRARY FUNCTIONS DOESN’T DO THAT, DOESN’T RESTART. SO AS IN THE CASE OF MULTIPLE PROCESS EXECUTIONS, THEY ARE NOT THREAD SAFE. THEY DON’T RESTART BY THEMSELVES.

USE ONLY THREAD-SAFE STANDARD C LIBRARY FUNCTIONS INSIDE YOUR MULTI-THREADED PROGRAMS.

User Threads versus Kernel Threads

The two traditional models of thread control are user-level threads and kernel-level threads.

User-level threads usually run on top of an existing operating system. These threads are invisible to the kernel and compete among themselves for the resources allocated to their encapsulating process

The threads are scheduled by a thread runtime system that is part of the process code

Programs with user-level threads usually link to a special library in which each library function is enclosed by a jacket

The jacket function calls the thread runtime system to do thread management before and possibly after calling the jacketed library function.

SOMETIMES YOU WANT YOUR KERNEL TO BE AWARE OF THREADS. WHEN YOU HAVE MULTIPLE CPUs SPECIALLY. IF KERNEL KNOWS YOU ARE CREATING AN EXECUTION INSIDE YOUR PROGRAM, THAT CAN BE SCHEDULED TO DIFFERENT CPUs. KERNEL IS AWARE OF EXECUTIONS YOU ARE CREATING 🡪 KERNEL LEVEL THREADS. IF YOU LET KERNEL SCHEDULER KNOWS THAT YOU ARE CREATING AN EXECUTION, THEN YOU CANNOT ALLOCATE SOME MEMORY IN PROGRAM STACK FOR YOURSELF. YOU HAVE TO HAVE ANOTHER PLACE, SPECIAL ID, THAT KERNEL IS AWARE OF BUT STILL YOU SHOULD BE ABLE TO ACCESS RED PART STUFF IN THE FIRST FIGURE. SO CREATION OF KERNEL THREAD REQUIRES MORE RESOURCES. KERNEL LEVEL THREAD CAN NEARLY BE AS BIG AS YOUR PROCESS. STILL OVERHEAD OF CREATING A KERNEL LEVEL THREAD IS LESS THAN CREATE A WHOLE PROCESS BY ITSELF.

SOME COMPANIES DIDN’T HAVE LUXURY OF HAVING MANY CPUs AND THEY WERE GOOD AT SOFTWARE SO THEY DEVELOPED SYNCHRONIZATION METHODOLOGIES. SO USER THREADS WHICH ARE MORE LIGHTWEIGHTED AND 1 CPU BOUNDED OCCURRED.

BOTH USER AND KERNEL LEVEL THREADS HAVE TO ACCESS RED PART IN FIRST FIGURE. YOU HAVE TO DO SYNCHRONIZATION ON THIS PART.

User Level Threads

Functions such as read or sleep can present a problem for user-level threads because they may cause the process to block.

To avoid blocking the entire process on a blocking call, the user-level thread library replaces each potentially blocking call in the jacket by a nonblocking version.

The thread runtime system tests to see if the call would cause the thread to block. If the call would not block, the runtime system does the call right away. If the call would block, however, the runtime system places the thread on a list of waiting threads, adds the call to a list of actions to try later, and picks another thread to run.

All this control is invisible to the user and to the operating system.

User-level threads have low overhead, but they also have some disadvantages.

The user thread model, which assumes that the thread runtime system will eventually regain control, can be thwarted by CPU-bound threads (YOU BOUND TO 1 CPU – YOUR EXECUTION WILL BE DONE IN 1 CPU – KERNEL WILL NOT BE ABLE TO SCHEDULE OTHER EXECUTIONS INSIDE YOUR PROCESS TO ANOTHER CPU).

A CPU-bound thread rarely performs library calls and may prevent the thread runtime system from regaining control to schedule other threads.

The programmer has to avoid the lockout situation by explicitly forcing CPU-bound threads to yield control at appropriate points.

Kernel-level Threads

With kernel-level threads, the kernel is aware of each thread as a schedulable entity and threads compete systemwide for processor resources

The scheduling of kernel-level threads can be almost as expensive as the scheduling of processes themselves, but kernel-level threads can take advantage of multiple processors.

The synchronization and sharing of data for kernel-level threads is less expensive than for full processes, but kernel-level threads are considerably more expensive to manage than user-level threads.

CAN EXECUTE MULTIPLE EXECUTIONS IF MULTIPLE CPUs ON THAT SYSTEM ARE AVAILABLE.

Hybrid Threads – PTHREAD/POSIX LIBRARY HAS IMPLEMENTED IN THIS WAY

Hybrid thread models have advantages of both user-level and kernel-level models by providing two levels of control

The user writes the program in terms of user-level threads and then specifies how many kernelschedulable entities are associated with the process

The user-level threads are mapped into the kernelschedulable entities at runtime to achieve parallelism. The level of control that a user has over the mapping depends on the implementation

IF YOU CHANGE ATTRIBUTE IN PTHREAD\_CREATE FROM NULL, YOU CAN CREATE CPU-BOUND (PROCESS LEVEL / USER LEVEL) OR SYSTEM-LEVEL (KERNEL LEVEL) THREAD.

PTHREAD İLE USER-LEVEL THREAD CREATE ETTİĞİNDE AYNI PID’YE SAHİP 3 TANE PROCESS GÖRÜRSÜN (ps -a 🡪 KOMUTU İLE DENEYEBİLİRSİN). 1 TANESİ THREADLERİN USER’A BELLİ ETMEDEN HER THREAD İÇİN STACKTE YER AYIRAN VE BİRBİRLERİNİ ETKİLEMEMESİNİ SAĞLAYAN, SANKİ O PROGRAMIN INSIDE BİR SCHEDULERI GİBİ DAVRANAN BİR EXECUTION. İKİNCİSİ KERNEL’A THREAD OLUŞTURDUĞUMUZU SÖYLER – FOR EXTERNAL SCHEDULER TO KEEP TRACK OF EXECUTIONS. SONRA 100 TANE DAHA GENERATE ETSEN 3 TANE GÖRÜRSÜN. ÜÇÜNCÜSÜ ZATEN MAIN PROCESS.

BU DİĞER PROCESSLER SAYESİNDE HANGİ THREAD DURDU VS. ÖĞRENEBİLİRİZ.

User Threads versus Kernel Threads

The user-level threads are called threads and the kernel-schedulable entities are called lightweight processes

The POSIX thread scheduling model is a hybrid model that is flexible enough to support both userlevel and kernel-level threads in particular implementations of the standard.

The model consists of two levels of scheduling— threads and kernel entities. The threads are analogous to user-level threads. The kernel entities are scheduled by the kernel. The thread library decides how many kernel entities it needs and how they will be mapped.

User-level threads run on top of an operating system

* Threads are invisible to the kernel.
* Link to a special library of system calls that prevent blocking
* Have low overhead
* CPU-bound threads can block other threads
  + IF ONE OF THE THREADS IS BLOCKED, REST OF THE EXECUTIONS MAY BE BLOCKED.
* Can only use one processor at a time.

Kernel-level threads are part of the OS.

* Kernel can schedule threads like it does processes.
* Multiple threads of a process can run simultaneously on multiple CPUs.
* Synchronization more efficient than for processes but less than for user-level threads.

Pthreads API – POSIX THREAD API

Pthread data types:

* The Pthreads API defines a number of data types, some of which are listed

Table

Description automatically generated



ID OF THREAD

IF YOU WANT TO USE THREAD JUST ONCE.

Threads and errno:

* In the traditional UNIX API, errno is a global integer variable. However, this doesn’t suffice for threaded programs.
  + WHEN YOU RETURN 0, THERE IS AN ERROR IN UNIX API AND CORRESPONDING ERRNO IS CHANGED ACCORDINGLY.
  + IT IS NOT SIMPLE TO WHICH ERROR OCCURRED ON WHICH EXECUTION. THEREFORE WE USE RETURN VALUE.
* If a thread made a function call that returned an error in a global errno variable, then this would confuse other threads that might also be making function calls and checking errno (race condition)
* Therefore, in threaded programs, each thread has its own errno value. The errno mechanism has been adapted for threads in a manner that leaves error reporting unchanged from the traditional UNIX API.

Return value from Pthreads functions:

* The traditional method of returning status from system calls and some library functions is to return 0 on success and –1 on error, with errno being set to indicate the error. SO YOU CAN KNOW THIS ERROR HAPPENED WHILE EXECUTING THIS SPECIFIC THREAD.
* The functions in the Pthreads API do things differently. All Pthreads functions return 0 on success or a positive value on failure. The failure value is one of the same values that can be placed in errno by traditional UNIX system calls

POSIX Thread management functions

Text

Description automatically generated with low confidence

Most POSIX thread functions return 0 if successful and a nonzero error code if unsuccessful. They do not set errno, so the caller cannot use perror to report errors

FROM A THREAD, YOU CAN CANCEL THE EXECUTION OF ANOTHER THREAD WITH PTHREAD\_CANCEL.

IF YOU DON’T CARE ABOUT THE RETURN VALUE OF THE THREAD, MAKE IT DETACHABLE. WHEN THREAD İS DETACHED, YOU DON’T HAVE TO WAIT FOR ITS RETURN VALUE. THEN THREAD HAS TO DO RESOURCE CLEANING ITSELF.

WITH ONLY EXIT, YOU STOP EXECUTION OF ALL PROCESS SO WE NEED PTHREAD\_EXIT.

JOIN IS DONE BY MAIN PROCESS WHO CREATED THE THREAD. YOU SHOULD USE IF YOU WANT TO LEARN RETURN VALUE.

Pthreads API: Thread Creation

When a program is started, the resulting process consists of a single thread, called the initial or main thread.

The pthread\_create() function creates an additional (new) thread.

Text, letter

Description automatically generated

The new thread commences execution by calling the function identified by start with the argument arg. The thread that calls pthread\_create() continues execution with the next statement that follows the call.

Pthreads API: Thread Termination

The execution of a thread terminates in one of the following ways:

* The thread’s start function performs a return specifying a return value for the thread.
* The thread calls pthread\_exit() .
* The thread is canceled using pthread\_cancel()
* Any of the threads calls exit(), or the main thread performs a return (in the main() function), which causes all threads in the process to terminate immediately.

DON’T CREATE PROCESS INSIDE A THREAD.

The pthread\_exit() function terminates the calling thread, and specifies a return value (RETVAL) that can be obtained in another thread by calling pthread\_join().

Shape, rectangle

Description automatically generated

FINISHES EXECUTION OF THREAD FUNCTION. IF THE THREAD IS JOINABLE, ALL THE RESOURCES (STACK FOR THREAD) WILL STILL BE THERE, THE RETURN VALUE (RETVAL) WILL STILL BE AVAILABLE UNTIL JOIN FUNCTION CALL IS MADE. EXECUTION IS NOT DONE ANYMORE WITH PTHREAD\_EXIT BUT RESOURCES ALLOCATED ARE STILL AVAILABLE.

Calling pthread\_exit() is equivalent to performing a return in the thread’s start function, with the difference that pthread\_exit() can be called from any function that has been called by the thread’s start function.

If the main thread calls pthread\_exit() instead of calling exit() or performing a return , then the other threads continue to execute.

Pthreads API: Thread IDs

POSIX threads are referenced by an ID of type pthread\_t. A thread can find out its ID by calling pthread\_self.

A picture containing text

Description automatically generated

Since pthread\_t may be a structure, use pthread\_equal to compare thread IDs for equality. The parameters of pthread\_equal are the thread IDs to be compared.

Shape

Description automatically generated with medium confidence

If t1 equals t2, pthread\_equal returns a nonzero value. If the thread IDs are not equal, pthread\_equal returns 0

Pthreads API : Detaching and Joining

When a thread exits, it does not release its resources unless it is a detached thread.

The pthread\_detach function sets a thread's internal options to specify that storage for the thread can be reclaimed when the thread exits.

Detached threads do not report their status when they exit.

Threads that are not detached are joinable and do not release all their resources until another thread calls pthread\_join for them or the entire process exits. The pthread\_join function causes the caller to wait for the specified thread to exit (similar to waitpid at the process level)

To prevent memory leaks, long-running programs should eventually call either pthread\_detach or pthread\_join for every thread.

DETACHING A THREAD 🡪 THREAD WILL DO WHATEVER IT SUPPOSED TO DO BUT RETURN VALUE IS NOT IMPORTANT FOR THE EXECUTION THAT HAS CREATED THAT THREAD. YOU HAVE TO DO CLEAN UP BY YOURSELF.

JOINING A THREAD 🡪 MAIN EXECUTION WAITS FOR THREAD FUNCTION TO COMPLETE WHATEVER IT IS DOING AND LEARN RETURN VALUE FOR IT. IF YOU CREATE A JOINABLE THREAD, MAIN EXECUTION HAS TO WAIT FOR THAT THREAD TO FINISH ITS EXECUTION. JOINING IS DONE WITH ID OF THE THREAD. YOU HAVE TO WAIT FOR EACH JOINABLE THREAD WITH CORRESPONDING ID. CLEANING IS DONE BY INTERNAL THREAD LIBRARY SCHEDULER.

Pthreads API: Joining

A nondetached thread's resources are not released until another thread calls pthread\_join with the ID of the terminating thread as the first parameter.

The pthread\_join function suspends the calling thread until the target thread, specified by the first parameter, terminates.

The value\_ptr parameter provides a location for a pointer to the return status that the target thread passes to pthread\_exit or return. If value\_ptr is NULL, the caller does not retrieve the target thread return status

A picture containing shape

Description automatically generated

VALUE\_PTR IS RETURN VALUE. YOU CAN USE IT IF YOU WISH.

Text, application

Description automatically generated

Pthreads API: Detaching – CALL IT FROM MAIN THREAD

The pthread\_detach function has a single parameter, thread, the thread ID of the thread to be detached.

A picture containing text

Description automatically generated

If successful, pthread\_detach returns 0. If unsuccessful, pthread\_detach returns a nonzero error code

Once a thread has been detached, it is no longer possible to use pthread\_join() to obtain its return status, and the thread can’t be made joinable again. Detaching a thread doesn’t make it immune to a call to exit() in another thread or a return in the main thread. In such an event, all threads in the process are immediately terminated, regardless of whether they are joinable or detached. To put things another way, pthread\_detach() simply controls what happens after a thread terminates, not how or when it terminates.

Pthreads API: Thread Attributes

As mentioned earlier that the pthread\_create() attr argument, whose type is pthread\_attr\_t, can be used to specify the attributes used in the creation of a new thread. We won’t go into the details of these attributes (see the CSE 244 slides for details) or show the prototypes of the various Pthreads functions that can be used to manipulate a pthread\_attr\_t object.

We’ll just mention that these attributes include information such as the location and size of the thread’s stack, the thread’s scheduling policy and priority and whether the thread is joinable or detached.

ALL THE ATTRIBUTES EITHER DETACHABLE, JOINABLE, SYSTEM-BOUNDED, CPU-BOUNDED… CAN BE ADJUSTED INTO ATTRIBUTES.

YOU CAN EVEN ADJUST THE AMOUNT OF MEMORY ALLOCATED INSIDE THE PROCESS SPECIFIC FOR THAT THREAD USING ATTRIBUTE VARIABLES.

EVEN SCHEDULER (THREAD LIBRARY SCHEDULER OR KERNEL SCHEDULER) PRIORITY FOR THREADS CAN BE ADJUSTED USING ATTRIBUTES. YOU CAN GIVE PRIORITY BUT THERE IS NO DETERMINISTIC EXECUTION.

Thread Attributes Example code

Text

Description automatically generated

IF YOU WANT TO CREATE ANOTHER THREAD WITH SAME ATTRIBUTE THEN USE BEFORE DESTROYED.

NOW MAIN PROCESS DOESN’T HAVE TO WAIT FOR THREAD TO REMOVE ITS RESOURCES.

The code shown creates a new thread that is made detached at the time of thread creation.

This code first initializes a thread attributes structure with default values, sets the attribute required to create a detached thread, and then creates a new thread using the thread attributes structure.

Once the thread has been created, the attributes object is no longer needed, and so is destroyed.

Threads Versus Processes

This lecture we briefly considered some of the factors that might influence our choice of whether to implement an application as a group of threads or as a group of processes.

We begin by considering the advantages of a multithreaded approach:

* Sharing data between threads is easy. By contrast, sharing data between processes requires more work. YOU DON’T HAVE TO USE IPC WITH THREADS.
* Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.

USAGE OF SYSTEM RESOURCES IS MUCH MORE LESS COMPARED TO PROCESSES.

Using threads can have some disadvantages compared to using processes

* When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner. Multiprocess applications don’t need to be concerned with this.
  + YOU HAVE TO CALL THREAD-SAFE FUNCTIONS IN MULTITHREADED PROGRAMMING AS OPPOSED TO YOU CAN USE STANDARD C LIBRARY, WHICHEVER IT IS, ON MULTIPROCESSED PROGRAMMING.
* A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.
  + YOU CAN EASILY TRACK THE ERROR WITH PROCESSES.
  + TRACKING ONE THREAD IS EASIER. IF YOU HAVE ONLY ONE THREAD, BLOCKING THAT THREAD WILL BLOCK THE WHOLE PROCESS. BUT IF ONE THREAD IS BLOCKED IN MULTITHREADED PROCESS, REST OF THE THREADS CAN STILL DO EXECUTIONS.
* Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread’s stack and thread-specific data (or thread local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large, this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory.

The following are some other points that may influence our choice of threads versus processes:

* Dealing with signals in a multithreaded application requires careful design. (As a general principle, it is usually desirable to avoid the use of signals in multithreaded programs.)
  + YOU CAN ASSIGN SPECIFIC THREAD TO SPECIFIC SIGNAL. INSTEAD OF WHOLE PROCESS WAITING FOR A SIGNAL, YOU CAN HAVE SMALL RESOURCE WAIT FOR THE SIGNAL AND DO WHATEVER IT HAS TO DO WHEN THAT SIGNAL OCCURS.
* In a multithreaded application, all threads must be running the same program. In a multiprocess application, different processes can run different programs.
* Aside from data, threads also share certain other information (e.g., file descriptors, signal dispositions, current working directory, and user and group IDs). This may be an advantage or a disadvantage, depending on the application.

Summary

In a multithreaded process, multiple threads are concurrently executing the same program. All of the threads share the same global and heap variables, but each thread has a private stack for local variables. The threads in a process also share a number of other attributes, including process ID, open file descriptors, signal dispositions, current working directory, and resource limits.

The key difference between threads and processes is the easier sharing of information that threads provide, and this is the main reason that some application designs map better onto a multithread design than onto a multiprocess design. Threads can also provide fast performance for some operations, but this factor is usually secondary in influencing the choice of threads versus processes.

Threads are created using pthread\_create(). Each thread can then independently terminate using pthread\_exit() (If any thread calls exit(), then all threads immediately terminate).

Unless a thread has been marked as detached, it must be joined by another thread using pthread\_join(), which returns the termination status of the joined thread.

THREAD SHOULDN’T CREATE A PROCESS. YOU HAVE JUST STACK AND SOME GLOBAL PART. YOU ARE TRYING TO CREATE A PROCESS WHICH DUPLICATES STACK AND GLOBAL PART. THEN YOU GIVE ID TO IT. MAIN OS SCHEDULER DOESN’T KNOW THREAD BUT IT GOTTA KNOW THIS PROCESS WITH THAT ID. SO OUTCOME IS UNPREDICTABLE.

CLEVER THREAD LIBRARIES WILL GIVE YOU ERROR WHEN YOU TRY TO CREATE A PROCESS FROM A THREAD.

POSIX THREAD İÇİNDEN PROCESS CREATE ETMEYİN YASAK DİYE BİR STANDART KOYMAMIŞ.