**COMSATS UNIVERSITY ISLAMABAD**

**ATTOCK CAMPUS**

**ASSIGNMENT#02**

**SUBMITTED TO:MR IRSHADULLAH**

**SUBMITTED BY:M MOIZ**

**REGISTERATION NO:FA21-BSE-060**

**SUBJECT:OPERATING SYSTEMS**

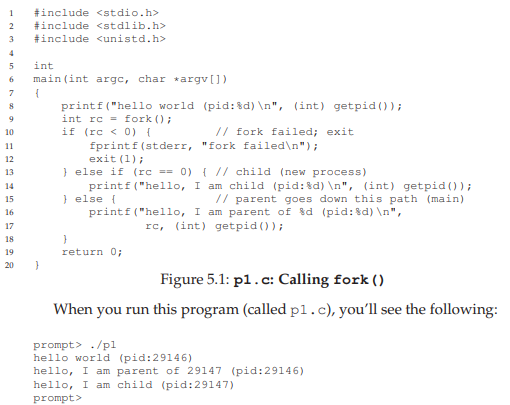
**DATE:28th MARCH 2024**

**THREADING ISSUES**

**SEMANTICS OF FORK() AND EXEC() SYSTEM CALLS**

**FORK()SYSTEM CALL:**

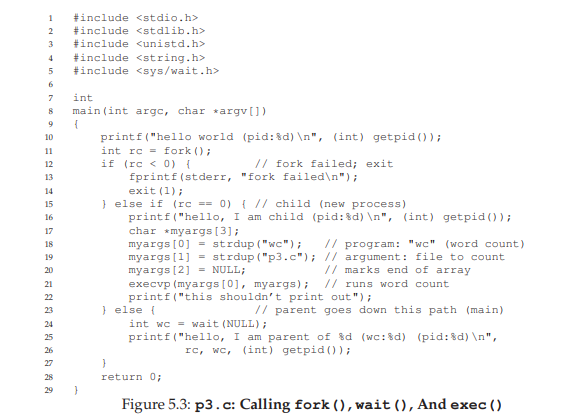
The fork() system call is used to create a new process .



When it first started running, the process prints out a hello world message; included in that message is its process identifier, also known as a PID. The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for example) stop it from running. So far, so good. Now the interesting part begins. The process calls the fork() system call, which the OS provides as a way to create a new process. The odd part: the process that is created is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program p1 running, and both are about to return from the fork() system call. The newly-created process (called the child, in contrast to the creating parent) doesn’t start running at main(), like you might expect (note, the “hello, world” message only got printed out once); rather, it just comes into life as if it had called fork() itself. You might have noticed: the child isn’t an exact copy. Specifically, although it now has its own copy of the address space (i.e., its own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of fork() is different. Specifically, while the parent receives the PID of the newly-created child, the child is simply returned a 0. This differentiation is useful, because it is simple then to write the code that handles the two different cases (as above).[1]

**EXEC()SYSTEM CALL:**

A final and important piece of the process creation API is the exec() system call . This system call is useful when you want to run a program that is different from the calling program. For example, calling fork() in is only useful if you want to keep running copies of the same program. However, often you want to run a different program; exec() does just that. In this example, the child process calls execvp() in order to run the program wc, which is the word counting program. In fact, it runs wc on the source file p3.c, thus telling us how many lines, words, and bytes are found in the file:[1]



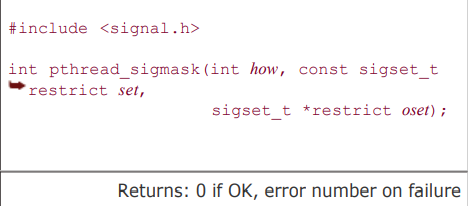
If fork() was strange, exec() is not so normal either. What it does: given the name of an executable (e.g., wc), and some arguments (e.g., p3.c), it loads code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are reinitialized. Then the OS simply runs that program, passing in any arguments as the argv of that process. Thus, it does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.[1]

**SIGNAL HANDLING:**

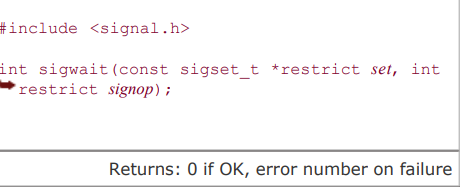
**SYNCHRONIZED AND ASYNCHRONIZED:**

Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated. Each thread has its own signal mask, but the signal disposition is shared by all threads in the process. This means that individual threads can block signals, but when a thread modifies the action associated with a given signal, all threads share the action. Thus, if one thread chooses to ignore a given signal, another thread can undo that choice by restoring the default disposition or installing a signal handler for the signal. Signals are delivered to a single thread in the process. If the signal is related to a hardware fault or expiring timer, the signal is sent to the thread whose action caused the event. Other signals, on the other hand, are delivered to an arbitrary thread.

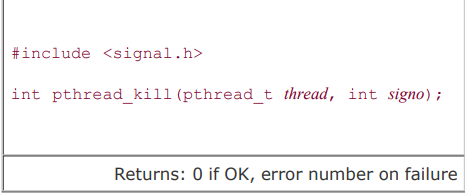
Processes can use sigprocmask to block signals from delivery. The behavior of sigprocmask is undefined in a multithreaded process. Threads have to use pthread\_sigmask instead.[2]



The pthread\_sigmask function is identical to sigprocmask, except that pthread\_sigmask works with threads and returns an error code on failure instead of setting errno and returning -1. A thread can wait for one or more signals to occur by calling sigwait.

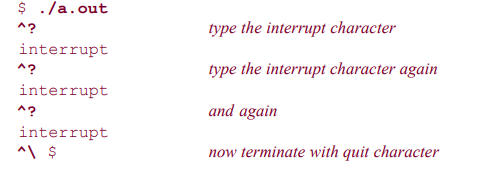


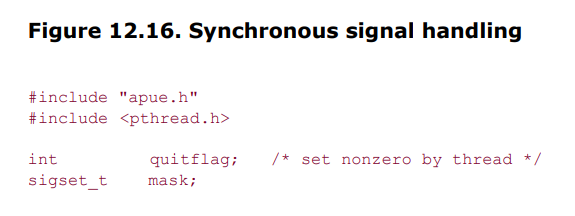
The set argument specifies the set of signals for which the thread is waiting. On return, the integer to which signop points will contain the number of the signal that was delivered. If one of the signals specified in the set is pending at the time sigwait is called, then sigwait will return without blocking. Before returning, sigwait removes the signal from the set of signals pending for the process. To avoid erroneous behavior, a thread must block the signals it is waiting for before calling sigwait. The sigwait function will atomically unblock the signals and wait until one is delivered. Before returning, sigwait will restore the thread's signal mask. If the signals are not blocked at the time that sigwait is called, then a timing window is opened up where one of the signals can be delivered to the thread before it completes its call to sigwait. The advantage to using sigwait is that it can simplify signal handling by allowing us to treat asynchronously-generated signals in a synchronous manner. We can prevent the signals from interrupting the threads by adding them to each thread's signal mask. Then we can dedicate specific threads to handling the signals. These dedicated threads can make function calls without having to worry about which functions are safe to call from a signal handler, because they are being called from normal thread context, not from a traditional signal handler interrupting a normal thread's execution. If multiple threads are blocked in calls to sigwait for the same signal, only one of the threads will return from sigwait when the signal is delivered. If a signal is being caught (the process has established a signal handler by using sigaction, for example) and a thread is waiting for the same signal in a call to sigwait, it is left up to the implementation to decide which way to deliver the signal. In this case, the implementation could either allow sigwait to return or invoke the signal handler, but not both. To send a signal to a process, we call kill. To send a signal to a thread, we call pthread\_kill.[2]

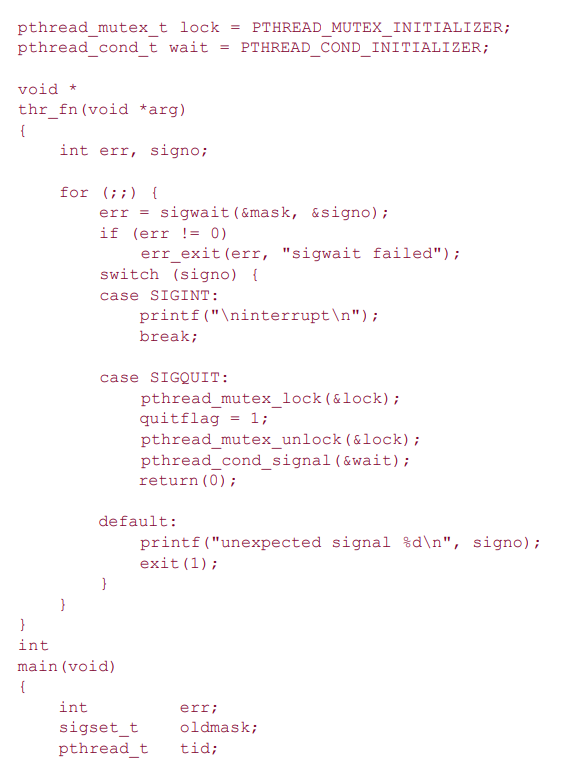


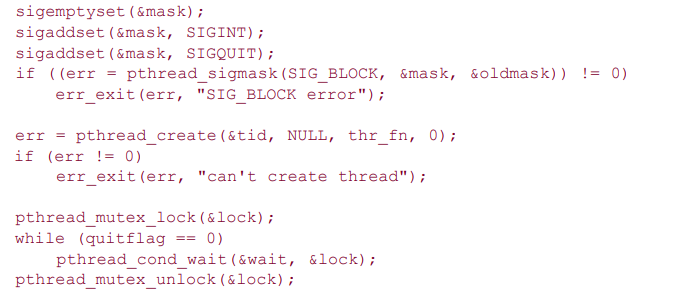
We can pass a signo value of 0 to check for existence of the thread. If the default action for a signal is to terminate the process, then sending the signal to a thread will still kill the entire process. Note that alarm timers are a process resource, and all threads share the same set of alarms. Thus, it is not possible for multiple threads in a process to use alarm timers without interfering (or cooperating) with one another

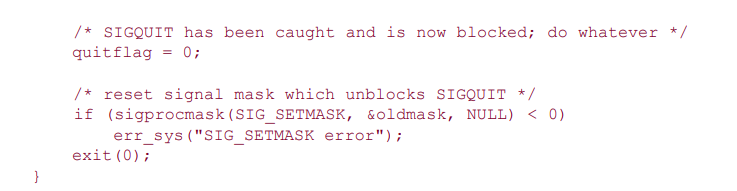
Recall that in Figure 10.23, we waited for the signal handler to set a flag indicating that the main program should exit. The only threads of control that could run were the main thread and the signal handler, so blocking the signals was sufficient to avoid missing a change to the flag. With threads, we need to use a mutex to protect the flag, as we show in the program in Figure 12.16. Instead of relying on a signal handler that interrupts the main thread of control, we dedicate a separate thread of control to handle the signals. We change the value of quitflag under the protection of a mutex so that the main thread of control can't miss the wake-up call made when we call pthread\_cond\_signal. We use the same mutex in the main thread of control to check the value of the flag, and atomically release the mutex and wait for the condition. Note that we block SIGINT and SIGQUIT in the beginning of the main thread. When we create the thread to handle signals, the thread inherits the current signal mask. Since sigwait will unblock the signals, only one thread is available to receive signals. This enables us to code the main thread without having to worry about interrupts from these signals. If we run this program, we get output similar to that from Figure 10.23:[2]





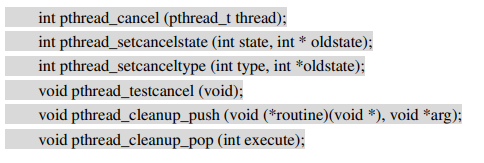






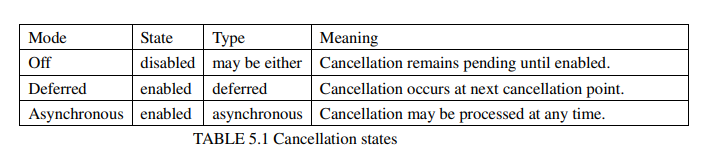
Linux implements threads as separate processes, sharing resources using clone(2). Because of this, the behavior of threads on Linux differs from that on other implementations when it comes to signals. In the POSIX.1 thread model, asynchronous signals are sent to a process, and then an individual thread within the process is selected to receive the signal, based on which threads are not currently blocking the signal. On Linux, an asynchronous signal is sent to a particular thread, and since each thread executes as a separate process, the system is unable to select a thread that isn't currently blocking the signal. The result is that the thread may not notice the signal. Thus, programs like the one in Figure 12.16 work when the signal is generated from the terminal driver, which signals the process group, but when you try to send a signal to the process using kill, it doesn't work as expected on Linux.p[2]

**THREAD CANCELLATION:**



Most of the time each thread runs independently, finishes a specific job, and exits on its own. But sometimes a thread is created to do something that doesn't necessarily need to be finished. The user might press a CANCEL button to stop a long search operation. Or the thread might be part of a redundant algorithm and is no longer useful because some other thread succeeded. What do you do when you just want a thread to go away? That's what the Pthreads cancellation interfaces are for. Cancelling a thread is a lot like telling a human to stop something they're doing. Say that one of the bailing programmers has become maniacally obsessed with reaching land, and refuses to stop rowing until reaching safety (Figure 5.1). When the boat finally runs up onto the beach, he's become so fixated that he fails to realize he's done. The other programmers must roughly shake him, and forcibly remove the oars from his blistered hands to stop him--but clearly he must be stopped. That's cancellation. Sort of. I can think of other analogies for cancellation within the bailing programmer story, but I choose to ignore them. Perhaps you can, too. Cancellation allows you to tell a thread to shut itself down. You don't need it often, but it can sometimes be extremely useful. Cancellation isn't an arbitrary external termination. It is more like a polite (though not necessarily "friendly") request. You're most likely to want to cancel a thread when you've found that something you set it off to accomplish is no longer necessary. You should never use cancellation unless you really want the target thread to go away. It is a termination mechanism, not a communication channel. So, why would you want to do that to a thread that you presumably created for some reason? An application might use threads to perform long-running operations, perhaps in the background, while the user continues working. Such operations might include saving a large document, preparing to print a document, or sorting a large list. Most such interfaces probably will need to have some way for the user to cancel an operation, whether it is pressing the ESC key or Ctrl-C, or clicking a stop sign icon on the screen. The thread receiving the user interface cancel request would then determine that one or more background operations were in progress, and use pthread\_cancel to cancel the appropriate threads.[3]

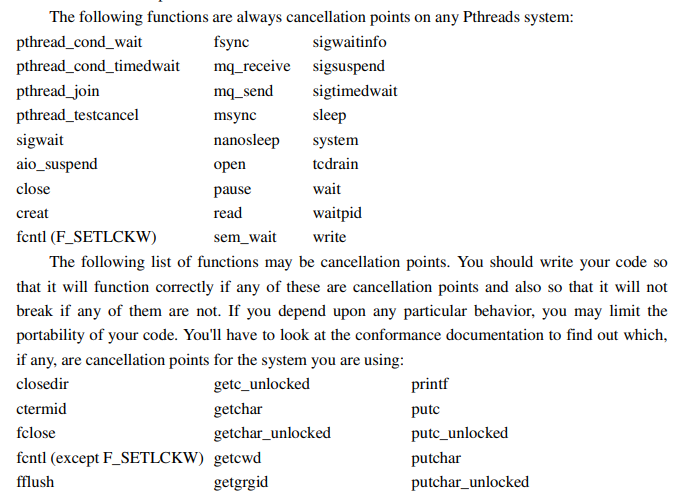
Often, threads are deployed to "explore" a data set in parallel for some heuristic solution. For example, solving an equation for a local minimum or maximum. Once you've gotten one answer that's good enough, the remaining threads may no longer be needed. If so, you can cancel them to avoid wasting processor time and get on to other work. Pthreads allows each thread to control its own termination. It can restore program invariants and unlock mutexes. It can even defer cancellation while it completes some important operation. For example, when two write operations must both complete if either completes, a cancellation between the two is not acceptable. Pthreads supports three cancellation modes, described in Table 5.1, which are encoded as two binary values called "cancellation state" and "cancellation type." Each essentially can be on or off. (While that technically gives four modes, one of them is redundant.) As shown in the table, cancellation state is said to be enabled or disabled, and cancellation type is said to be deferred or asynchronous.

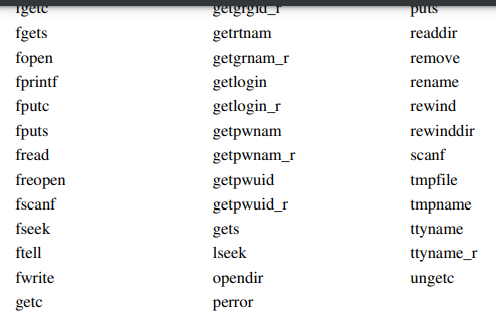


By default, cancellation is deferred, and can occur only at specific points in the program that check whether the thread has been requested to terminate, called cancellation points. Most functions that can wait for an unbounded time should be deferred cancellation points. Deferred cancellation points include waiting on a condition variable, reading or writing a file, and other functions where the thread may be blocked for a substantial period of time. There is also a special function called pthread\_testcancel that is nothing but a deferred cancellation point. It will return immediately if the thread hasn't been asked to terminate, which allows you to turn any of your functions into cancellation points. Some systems provide a function to terminate a thread immediately. Although that sounds useful, it is difficult to use such a function safely. In fact, it is nearly impossible in a normal modular programming environment. If a thread is terminated with a mutex locked, for example, the next thread trying to lock that mutex will be stuck waiting forever. It might seem that the thread system could automatically release the mutex; but most of the time that's no help. Threads lock mutexes because they're modifying shared data. No other thread can know what data has been modified or what the thread was trying to change, which makes it difficult to fix the data. Now the program is broken. When the mutex is left locked, you can usually tell that something's broken because one or more threads will hang waiting for the mutex. The only way to recover from terminating a thread with a locked mutex is for the application to be able to analyze all shared data and repair it to achieve a consistent and correct state. That is not impossible, and it is worth substantial effort when an application must be fail-safe. However, it is generally not practical for anything but an embedded system where the application designers control every bit of shared state in the process. You would have to rebuild not only your own program or library state, but also the state affected by any library functions that might be called by the thread (for example, the ANSI C library). To cancel a thread, you need the thread's identifier, the pthread\_t value returned to the creator by pthread\_create or returned to the thread itself by pthread\_self. Cancelling a thread is asynchronous--that is, when the call to pthread\_cancel returns, the thread has not necessarily been canceled, it may have only been notified that a cancel request is pending against it. If you need to know when the thread has actually terminated, you must join with it by calling pthread\_join after cancelling it. If the thread had asynchronous cancelability type set, or when the thread next reaches a deferred cancellation point, the cancel request will be delivered by the system. When that happens, the system will set the thread's cancelability type to PTHREAD\_CANCEL\_DEFERRED and the cancelability state to PTHREAD\_CANCEL\_DISABLE. That is, the thread can clean up and terminate without having to worry about being canceled again. When a function that is a cancellation point detects a pending cancel request, the function does not return to the caller. The active cleanup handlers will be called, if there are any, and the thread will terminate. There is no way to "handle" cancellation and continue execution--the thread must either defer cancellation entirely or terminate. This is analogous to C++ object destructors, rather than C++ exceptions--the object is allowed to clean up after itself, but it is not allowed to avoid destruction. The following program, called cancel.c, shows how to write a thread that responds "reasonably quickly" to deferred cancellation, by calling pthread\_testcancel within a loop. 11-19 The thread function thread\_routine loops indefinitely, until canceled, testing periodically for a pending cancellation request. It minimizes the overhead of calling pthread\_testcancel by doing so only every 1000 iterations (line 17). 27-35 On a Solaris system, set the thread concurrency level to 2, by calling thr\_setconcurrency. Without the call to thr\_setconcurrency, this program will hang on Solaris because thread\_routine is "compute bound" and will not block. The main program would never have another chance to run once thread routine started, and could not call pthread\_cancel. 36-54 The main program creates a thread running thread\_routine, sleeps for two seconds, and then cancels the thread. It joins with the thread, and checks the return value, which should be PTHREAD\_CANCELED to indicate that it was canceled, rather than terminated normally. cancel.c A thread can disable cancellation around sections of code that need to complete without interruption, by calling pthread\_setcancelstate. For example, if a database update operation takes two separate write calls, you wouldn't want to complete the first and have the second canceled. If you request that a thread be canceled while cancellation is disabled, the thread remembers that it was canceled but won't do anything about it until after cancellation is enabled again. Because enabling cancellation isn't a cancellation point, you also need to test for a pending cancel request if you want a cancel processed immediately. When a thread may be canceled while it holds private resources, such as a locked mutex or heap storage that won't ever be freed by any other thread, those resources need to be released when the thread is canceled. If the thread has a mutex locked, it may also need to "repair" shared data to restore program invariants. Cleanup handlers provide the mechanism to accomplish the cleanup, somewhat like process atexit handlers. After acquiring a resource, and before any cancellation points, declare a cleanup handler by calling pthread\_cleanup\_push. Before releasing the resource, but after any cancellation points, remove the cleanup handler by calling pthread\_cleanup\_pop. If you don't have a thread's identifier, you can't cancel the thread. That means that, at least using portable POSIX functions, you can't write an "idle thread killer" that will arbitrarily terminate threads in the process. You can only cancel threads that you created, or threads for which the creator (or the thread itself) gave you an identifier. That generally means that cancellation is restricted to operating within a subsystem.[3]

**DEFERRED CANCELLABILITY:**

"Deferred cancelability" means that the thread's cancelability type has been set to PTHREAD \_CANCEL\_DEFERRED and the thread's cancelability enable has been set to PTHREAD\_ CANCEL\_ENABLE. The thread will only respond to cancellation requests when it reaches one of a set of "cancellation points."





Pthreads specifies that any ANSI C or POSIX function not specified in one of the two lists cannot be a cancellation point. However, your system probably has many additional cancellation points. That's because few UNIX systems are "POSIX." That is, they support other programming interfaces as well--such as BSD 4.3, System V Release 4, UNIX95, and so forth. POSIX doesn't recognize the existence of functions such as select or poll, and therefore it can't say whether or not they are cancellation points. Yet clearly both are functions that may block for an arbitrary period of time, and programmers using them with cancellation would reasonably expect them to behave as cancellation points. X/Open is currently addressing this problem for UNIX98 (X/Open System Interfaces, Issue 5), by extending the Pthreads list of cancellation points. Most cancellation points involve I/O operations that may block the thread for an "unbounded" time. They're cancelable so that the waits can be interrupted. When a thread reaches a cancellation point the system determines whether a cancel is pending for the current ("target") thread. A cancel will be pending if another thread has called pthread\_cancel for the target thread since the last time the target thread returned from a cancellation point. If a cancel is pending, the system will immediately begin calling cleanup functions, and then the thread will terminate. If no cancel is currently pending, the function will proceed. If another thread requests that the thread be canceled while the thread is waiting for something (such as I/O) then the wait will be interrupted and the thread will begin its cancellation cleanup. If you need to ensure that cancellation can't occur at a particular cancellation point, or during some sequence of cancellation points, you can temporarily disable cancellation in that region of code. The following program, called cancel\_disable.c, is a variant of cancel.c. The "target" thread periodically calls sleep, and does not want the call to be cancelable. 23-32 After each cycle of 755 iterations, thread\_routine will call sleep to wait a second. (The value 755 is just an arbitrary number that popped into my head. Do arbitrary numbers ever pop into your head?) Prior to sleeping, thread\_routine disables cancellation by setting the cancelability state to PTHREAD\_CANCEL\_DISABLE. After sleep returns, it restores the saved cancelability state by calling pthread\_setcancelstate again. 33-35 Just as in cancel.c, test for a pending cancel every 1000 iterations. cancel\_disable.c[3]

**ASYNCHRONOUS CANCELLABILITY:**

Asynchronous cancellation is useful because the "target thread" doesn't need to poll for cancellation requests by using cancellation points. That can be valuable for a thread that runs a tight compute-bound loop (for example, searching for a prime number factor) where the overhead of calling pthread\_testcancel might be severe. | Avoid asynchronous cancellation! | It is difficult to use correctly and is rarely useful. The problem is that you're limited in what you can do with asynchronous cancellation enabled. You can't acquire any resources, for example, including locking a mutex. That's because the cleanup code would have no way to determine whether the mutex had been locked. Asynchronous cancellation can occur at any hardware instruction. On some computers it may even be possible to interrupt some instructions in the middle. That makes it really difficult to determine what the canceled thread was doing. For example, when you call malloc the system allocates some heap memory for you, stores a pointer to that memory somewhere (possibly in a hardware register), and then returns to your code, which probably moves the return value into some local storage for later use. There are lots of places that malloc might be interrupted by an asynchronous cancel, with varying effects. It might be interrupted before the memory was allocated. Or it might be interrupted after allocating storage but before it stored the address for return. Or it might even return to your code, but get interrupted before the return value could be copied to a local variable. In any of those cases the variable where your code expects to find a pointer to the allocated memory will be uninitialized. You can't tell whether the memory really was allocated yet. You can't free the memory, so that memory (if it was allocated to you) will remain allocated for the life of the program. That's a memory leak, which is not a desirable feature. Or when you call pthread\_mutex\_lock, the system might be interrupted within a function call either before or after locking the mutex. Again, there's no way for your program to find out, because the interrupt may have occurred between any two instructions, even within the pthread\_mutex\_lock function, which might leave the mutex unusable. If the mutex is locked, the application will likely end up hanging because it will never be unlocked. | Call no code with asynchronous cancellation enabled unless you | wrote it to be async-cancel safe--and even then, think twice! You are not allowed to call any function that acquires resources while asynchronous cancellation is enabled. In fact, you should never call any function while asynchronous cancellation is enabled unless the function is documented as "async-cancel safe." The only functions required to be async safe by Pthreads are pthread\_cancel, pthread\_setcancelstate, and pthread\_setcanceltype. (And there is no reason to call pthread\_cancel with asynchronous cancelability enabled.) No other POSIX or ANSI C functions need be async-cancel safe, and you should never call them with asynchronous cancelability enabled. Pthreads suggests that all library functions should document whether or not they are async-cancel safe. However if the description of a function does not specifically say it is async-cancel safe you should always assume that it is not. The consequences of asynchronous cancellation in a function that is not async-cancel safe can be severe. And worse, the effects are sensitive to timing--so a function that appears to be async-cancel safe during experimentation may in fact cause all sorts of problems later when it ends up being canceled in a slightly different place. The following program, cancel\_async.c, shows the use of asynchronous cancellation in a compute-bound loop. Use of asynchronous cancellation makes this loop "more responsive" than the deferred cancellation loop in cancel.c. However, the program would become unreliable if any function calls were made within the loop, whereas the deferred cancellation version would continue to function correctly. In most cases, synchronous cancellation is preferable. 24-28 To keep the thread running awhile with something more interesting than an empty loop, cancel\_async.c uses a simple matrix multiply nested loop. The matrixa and matrixb arrays are initialized with, respectively, their major or minor array index. 34-36 The cancellation type is changed to PTHREAD\_CANCEL\_ASYNCHRONOUS, allowing asynchronous cancellation within the matrix multiply loops. 39-44 The thread repeats the matrix multiply until canceled, on each iteration replacing the first source array (matrixa) with the result of the previous multiplication (matrixc). 66-74 Once again, on a Solaris system, set the thread concurrency level to 2, allowing the main thread and thread\_routine to run concurrently on a uniprocessor. The program will hang without this step, since user mode threads are not timesliced on Solaris. cancel\_async.c.[3]

**THREAD LOCAL-STORAGE:**

* Thread-local storage enables each thread to have its own private instance of a variable.
* This is useful when you need data specific to a thread and avoid sharing issues between threads.
* Declared with the thread\_local keyword before the variable declaration.
* Accessible only by the thread that created it.
* Different threads can have the same variable name with different instances.
* Thread-local variables are typically destroyed when the thread exits.

thread\_local int myThreadData;

void someFunction() {

myThreadData = 0; // Initialize to 0 for this thread

// Use myThreadData specific to this thread

}

**Benefits:**

* Improves performance by avoiding synchronization overhead for thread-specific data.
* Simplifies code by eliminating the need for complex data structures to manage thread-specific information.

**Potential Drawbacks:**

* Increased memory usage if many threads create large thread-local variables.
* Debugging challenges as thread-local variables are not visible from other threads.

**Applications:**

* Thread-specific counters or state variables.
* Caching frequently accessed data for a thread.
* Implementing thread-safe random number generators.[4]

**SCHEDULER ACTIVATIONS:**

While kernel threads are better than user-level threads in some key ways, they are also indisputably slower. As a consequence, researchers have looked for ways to improve the situation without giving up their good properties. Below we will describe an approach devised by Anderson et al. (1992), called scheduler activations. Related work is discussed by Edler et al. (1988) and Scott et al. (1990). The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready. Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks waiting for another thread to do something, for example, there is no reason to involve the kernel, thus saving the overhead of the kernel-user transition. The user-space run-time system can block the synchronizing thread and schedule a new one by itself. When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to processors. This mechanism can also be used on a multiprocessor where the virtual processors may be real CPUs. The number of virtual processors allocated to a process is initially one, but the process can ask for more and can also return processors it no longer needs. The kernel can also take back virtual processors already allocated in order to assign them to more needy processes. The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked (e.g., by its having executed a blocking system call or caused a page fault), the kernel notifies the process’ run-time system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the run-time system at a known starting address, roughly analogous to a signal in UNIX. This mechanism is called an upcall. Once activated, the run-time system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the run-time system to inform it. The run-time system can either restart the blocked thread immediately or put it on the ready list to be run later. When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process’ I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process’ threads, the interrupted thread is not restarted. Instead, it is suspended, and the run-time system is started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the run-time system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice. An objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer n offers certain services that layer n + 1 can call on, but layer n may not call procedures in layer n + 1. Upcalls do not follow this fundamental principle[5].

**REFERENCES:**

[1]"Operating Systems: Three Easy Pieces" by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau(Book)

[2**]** "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago(Book)

[3] "Programming with POSIX Threads" by David R. Butenhof(Book)

[4] "C++ Concurrency in Action" by Anthony Williams(Book)

[5] "Modern Operating Systems" by Andrew S. Tanenbaum and Herbert Bos(Book)

**THE END**