Thread

Sunate Hwang
Kookmin University

References - Threads Primer, Bil Lewis & Daniel J. Berg, SunSoft Press

THREAD PROGRAMMING “FOUNDATIONS”

Thread Libraries

- There are two fundamentally different ways of implem enting threads.
  - The first- write a user-level library that is substantially self-con tained.
    - library를 위한 모든 code나 structure는 user space에 있다.
  - The second way - write a library that is inherently a kernel-lev el implementation.
    - Kernel routine에 의존하고있다.
- 양쪽 경우 다 프로그래머는 thread library에의해 구현된 API(Application Programmer's Interface)를 이용한다.
- 이런 library는 MT program을 작성하기 위한 기초적인 것만을 제공한다. 보다 복잡한 것은 프로그래머가 librar y를 이용하여 구축한다

Process Structure

- The only the kernel knows about is the process structure.
  - Solaris 2에서는 CPU state가 분리되어 새로운 structure를 형성하는데 이를 LWP(lightweight process)라 부른다.
  - 이 LWP는 이전의 SunOS 4.x의 LWP library와는 완전히 다른 것이다.

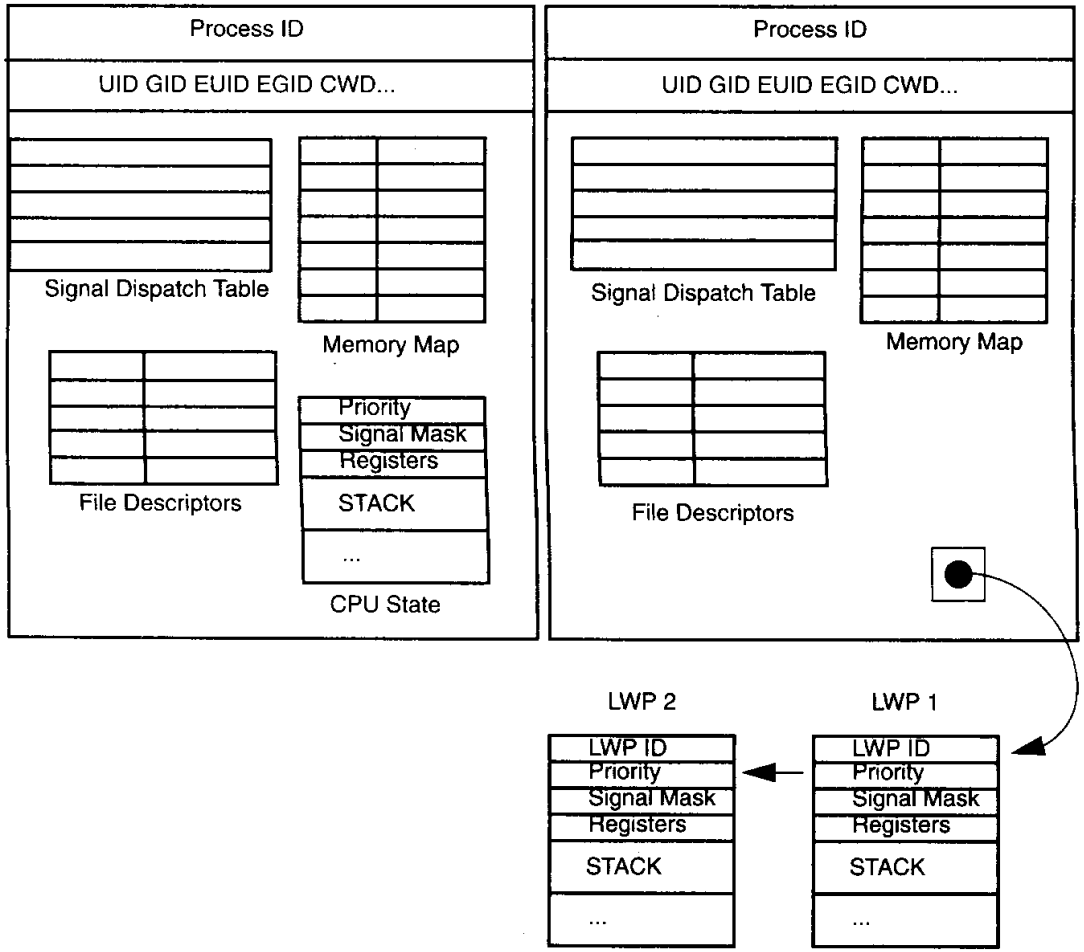


Figure 3-1 The Process Structure in Traditional UNIX and in Solaris 2

Lightweight Processes

- A lightweight process can be thought of as a virtual CPU that is a vailable for executing code.
  - 각 LWP는 kernel에 의해 독립적으로 처리
  - 독립적으로 system call을 수행하고 독립적인 page fault를 발생할 수 있다.
- Multiple LWPs in the same process can run in parallel on multiple processors.
  - LWP는 scheduling class와 priority에 따라서 가용한 CPU에 scheduled된다.
  - LWP 단위로 schedule되므로 각 LWP는 자신의 kernel statistic을 수집해야한다. - user time, system time, page faults, etc
  - LWP는 thread interface를 지원하기 위해 kernel-level의 concurrency와 parallelism을 제공하기 위한 한 구현 기술이다. LWP interface를 직접 사용해도 되지만 호환성만 잃게 되고 특별히 더 좋은 것은 없다. 따라서 thread interface를 이용할 것.

Threads and LWPs

- Process in a traditional multitasking OS
  - memory, the CPU register state, some system state(file descriptors, user ID, working directory, etc., all stored in the *process structure*) 로 구성
  - context switch
    - save registers in the process structure, change some virtual memory pointers, loads the CPU registers with data from the other process structure, and continue.

- Unbound thread
  - context switch
    - process와 비슷, 그러나 모든 것이 user space에서 thread library에의해서 이루어짐
    - saving the registers into one thread structure and replacing them with the saved state from another thread structure. The virtual memory page table and all the system state **remain unchanged**.
    - The idea is that you have a single program, in one memory space, with many virtual CPUs running different parts of the program concurrently. It's a user-level version of multitasking.

What actually makes up a threads are

- its own stack and stack pointer; a program counter; some thread information, such as scheduling priority and signal mask, stored in the thread structure; and the CPU registers (the stack pointer and program counter are actually just registers).

  - Global data와 마찬가지로 code는 thread의 일부분이 아니다.
  - thread는 그 code를 실행하고 그 global data를 사용할 것이다.

  - Stack과 같은 thread structure는 user space에 있으므로 다른 thread에서 변경할 수 도 있다. 그러나 결코 좋은 방법이 아니다.

Solaris Multithreaded Model

- Threads are the portable application-level interface.
- Programmers write applications using the threads library.
- The library schedules the threads onto LWPs.
- The LWPs in turn are implemented by kernel threads in the Solaris kernel.
- These kernel threads are then scheduled onto the available CPUs by the standard kernel scheduling routine, completely invisibley to the user.

- The importance of two-level model
  - ability to meet the demands of different programming requirements.
  - Large amount of concurrency - such as a window system that provides each widget with one input handler and one output handler.
  - parallel computation onto the actual processors available
  - currency provided by being able to have numerous threads involved with blocking system calls simultaneously.

System calls

- A system call is the way multitasking operating systems allow user processes to get information or request services from the kernel.
  - Blocking system call
  - Nonblocking system call

1. **The process traps to the kern el.**
2. **The trap handler runs in kern el mode, and saves all of the registers.**
3. **It sets the stack pointer to th e process structure's kernel stack.**
4. **The kernel runs the system c all.**
5. **The kernel places any reques ted data into the user-space structure that the programm er provided.**
6. **The kernel changes any proc ess structure values affected .**
7. **The process returns to user mode, replacing the registers . And returns the appropriate value from the system call.**

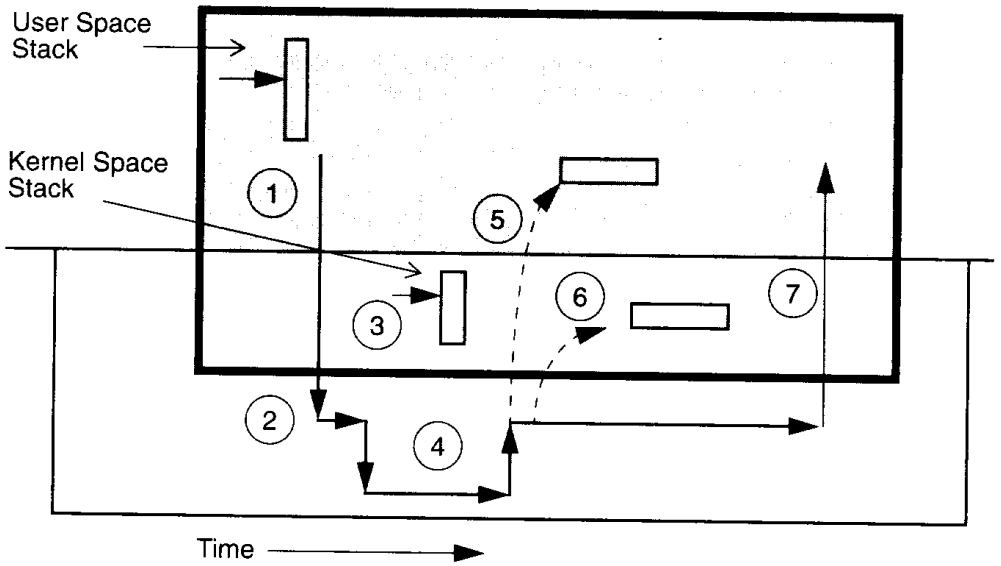


Figure 3-5 The Operation of a System Call

- LWP에서도 거의 같은 과정으로 system call 이 이루어진다. 각 LWP는 각각 stack을 가지고 있으므로 독립적으로 system call을 수행한다.
- 여러개의 thread가 독립적으로 system call을 수행하기 위해서는 그 system call이 blocking인 것이 바람직하다.

Signals

- Signals are the mechanism that UNIX uses in order to get asynchronous behavior in a program.

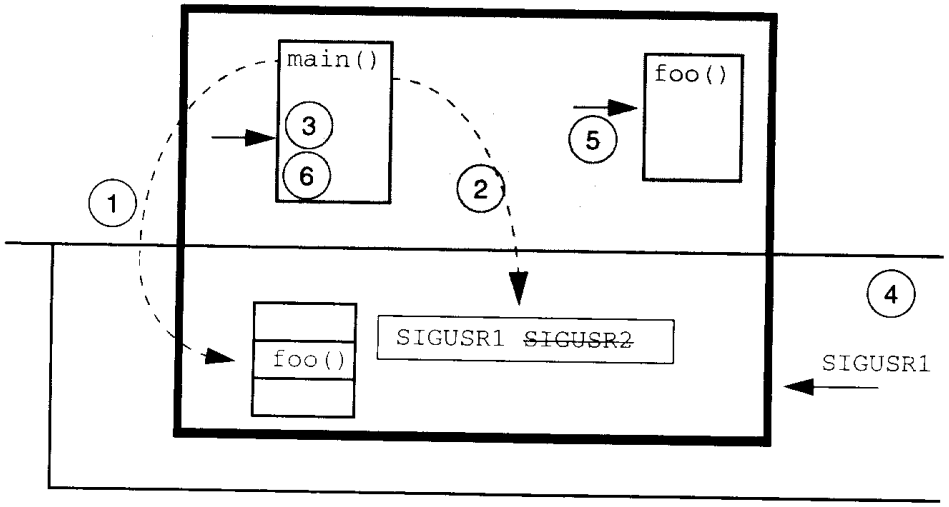


Figure 3-6 The Operation of a Signal

- A typical program will start up in main(), as in the above figure. The program will then call sigaction() (1) to declare some function of your choosing to be the handler for a signal(say, function foo() will handle SIGUSR1.) The kernel will put a pointer to that handler into the process structure's signal dispatch table(see Figure 3-1). Next, your program will call sigpromask() (2) to tell the kernel which signals it is willing to accept. Finally your program takes off and starts doing what you wrote it to do (3).
- Now, when some other process sends your process SIGUSR1 (4), your program will stop what it's doing and run the handler code you wrote (5). You have no idea what your program might be doing when the signal arrives. That's the idea with signals, they can be completely asynchronous. When the signal handler is done, it typically just does a return, and your program continues where it left off, as if nothing had happened.

- POSIX의 정의에 의하면 외부에서 발생한 signal은 process를 향하도록 되어 있다.(LWP나 thread가 아님).

- 보통 library가 어떤 thread가 signal을 처리할 지를 정한다.

- Synchronous signal(traps - divide by zero, illegal memory reference)은 반드시 해당 thread로 전달 된다.

- Programming을 간단히 하려면 모든 asynchronous signal을 한 thread에서 처리하도록 하는 것이 좋다.
  1. signal mask를 이용하여 한 thread를 제외하고는 모든 thread에서 signal을 금지한다.
  2. 더 좋은 방법은 sigwait()를 이용하는 것이다.

Scheduling

□Different Models of Kernel Scheduling

1) Many Threads on One LWP (M:1)

- The thread creation, scheduling and synchronization is all done 100% in user space, so it's fast and cheap.
- 그러나 blocking system call에서 한 thread가 block 되면 process 전체가 blcok된다. → a clever hack : "jacket"이라는 routine을 blocking system call 주변에 삽입하여 non-blocking으로 만든다
- 예: DCE threads library on HP-UX

2) One Thread per LWP (1:1)

- 한 thread가 non-blocking system call 때문에 block 되어도 다른 thread는 계속 실행할 수 있다. 그러나 tread creation은 LWP creation을 의미하므로 cost가 발생한다.
- 예: Windows NT, OS/2

3) Many Threads on Many LWPs (M:M)

- user space에서의 creation, scheduling, synchronization
- blocking system call이 process 전체를 block할 필요가 없다.

4) The Solaris Two-level Model

Thread Scheduling

- Local scheduling
  - thread를 위한 scheduling mechanism이 process에 local이다.
  - No system call

- Global scheduling
  - by kernel.
  - Time slicing

- Solaris implements both
- POSIX allows both
- NT and OS/2 - only global scheduling

- In Solaris
  - Global scheduling -- thread를 LWP에 binding하여 성취
  - Local scheduling – thread library에 의해서

- 실행 중인 thread, T1이 context switching되는 4가지 경우
  1. synchronization
     - 가장 흔한 경우는 mutex lock을 request했으나 얻지 못한 경우이다.
  2. suspension
     - thr_suspend()에 의해서
     - thr_continue()가 호출될 때 까지 stopped queue에서 대기
     - solaris, NT, OS/2는 suspension을 정의 한다. 그러나 POSIX에는 없다.
  3. preemption
     - T1이 실행 중일 때 더 높은 priority의 thread가 runnable하게 되면
  4. yielding
     - thr_yield()에 의해서
     - solaris와 POSIX에서만 정의함

- 1,2,4번의 경우 완전히 user space에서 실행될 수 있다.
  그러나 preemption은 system call이 개입 된다.

- Global scheduling에서 preemption은 kernel에 의해서 handling된다.
  Local scheduling에서는 특별히 만들어진 signal을 이용할 수 있다. 즉 thread library는 이 특별한 signal을 running중인 thread에게 보내고 이를 위한 signal handler는 context switching을 수행한다.

Thread state

1. **Active : LWP**에 있음
2. **Runnable :** 실행 준비가 되었으나 충분한 **LWP**가 없음
3. **Sleeping : synchronization variable**을 기다림
4. **Stopped : thr_suspend()**가 호출되었음을 의미, **thr_continue()**를 기다림
5. **Zombie :** 죽은 **thread**이나 **resource**를 가져가기를 기다리고 있음

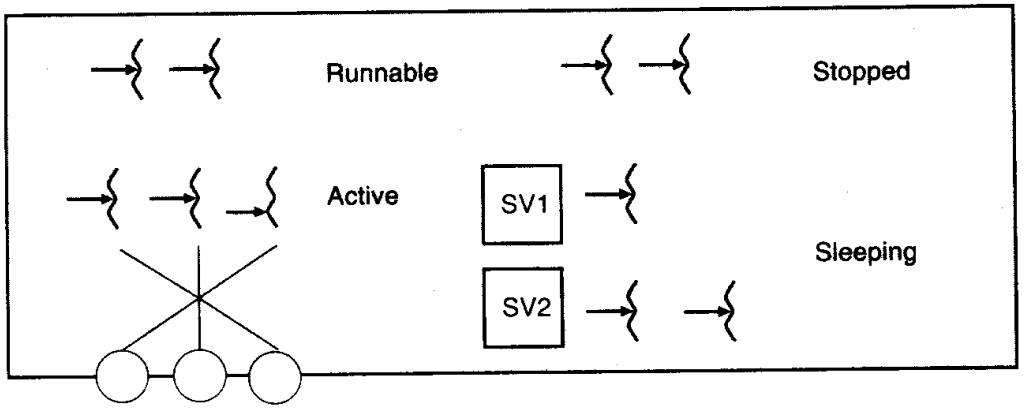


Figure 4-1 Some Threads in Various States

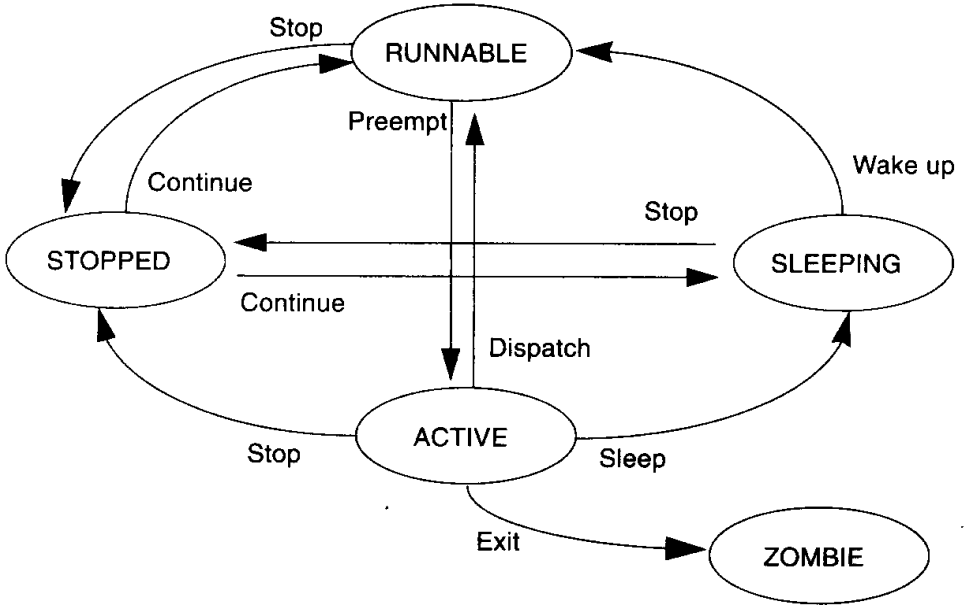


Figure 4-2 Simplified View of Thread State Transitions

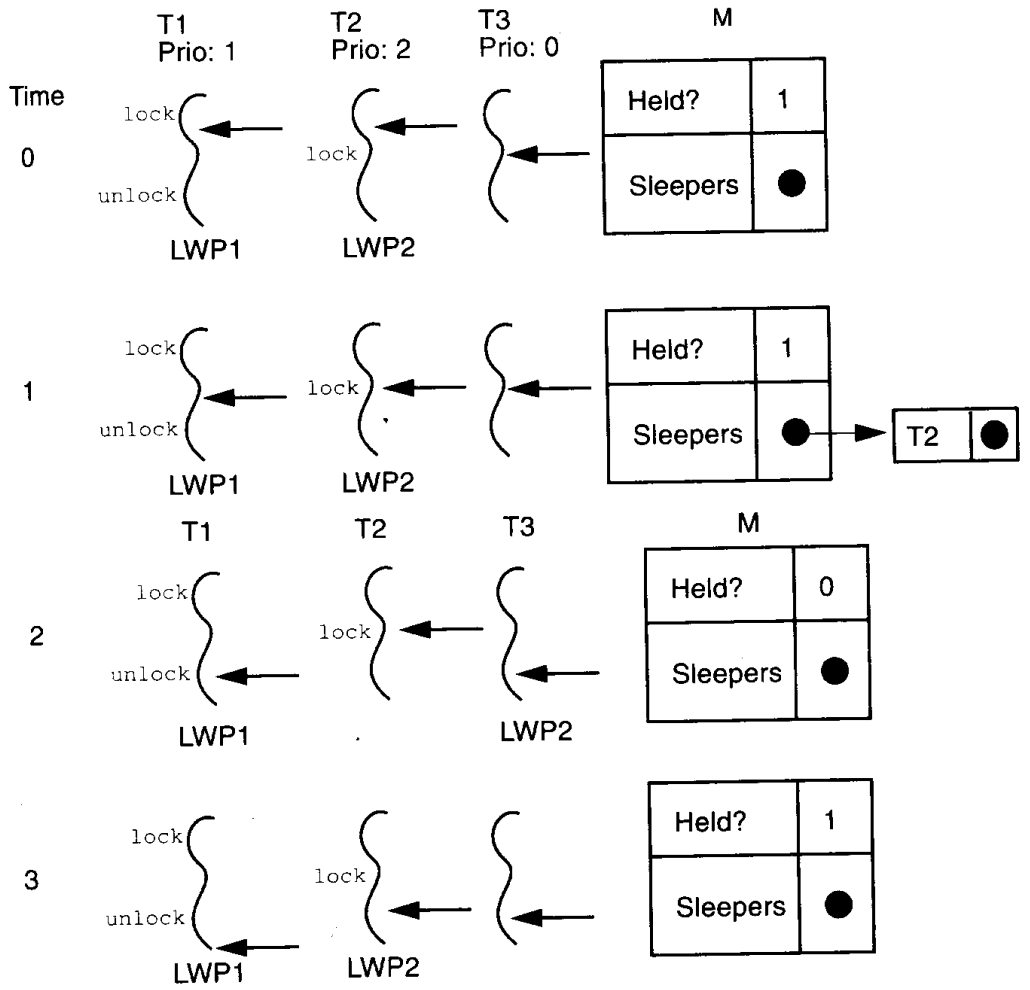
- POSIX는 3가지 scheduling algorithm을 정의 한다.
  - strict priority-based scheme : Solaris local model과 같다.
  - true time-sliced model - LWP schedule in time sharing class
  - priority-based FIFO scheme

- POSIX는 scheduling scope를 정의한다.
  - PTHREAD_SCOPE_SYSTEM - the equivalent of bound threads
  - PTHREAD_SCOPE_PROCESS - the equivalent of unbound threads

- Suspension은 Solaris에서 2경우에 유용하다. (POSIX에서는 정의하지 않음)
  - creation time에
  - Garbage Collection과 같이 모든 thread를 halt해야 할 경우

Context Switching

1. **LWP/CPU term**
2. **thread/LWP term**



Bound Threads and Real-time LWPs

- A bound thread is nothing more than a typical thread t hat is permanently tied to a specific LWP. (Globally sc heduled thread in POSIX)

1. need to ensure that it gets a CPU in second
   rely upon the normal time-slicing scheduling
2. the order of 100ms

raise the timesharing class priority of the LWP

3. 2-100ms

put the LWP into the real-time scheduling class.

Thread Creation and Destruction

thr_creation

thr_exit()

thr_join()

detached thread: THR_ DETACHED flag

□모든 **thread**는 어떤 **non-detached thread**라도 **thr_join()**을 호출하여 **join**할 수 있다. 하지만 **exiting thread**는 단 한번만 **joined**된다.

□**Thread ID** 대신에 **NULL**을 **passing**하여 바로 다음에 **exit**하는 어떤 **nondetached thread**와 **join**할 수 있다.

Cancellation

- Sometime you have reason to get rid of a thread before it has completed its work.

- Under Solaris:
  1. 주기적으로 global variable을 check - 주기가 규칙적이지 않아 매우 오랫동안 check gkwl ahtgkf tn dlTek.
  2. signal을 보낸다 - lock이 걸린 상태에서 cancel될 수 있다.

- POSIX threads has a formal solution to this problem.
  1. bracket sections of your code with calls that enable and disable cancellation.
  2. establishes cancellation points

POSIX THREADS

1) Introduction

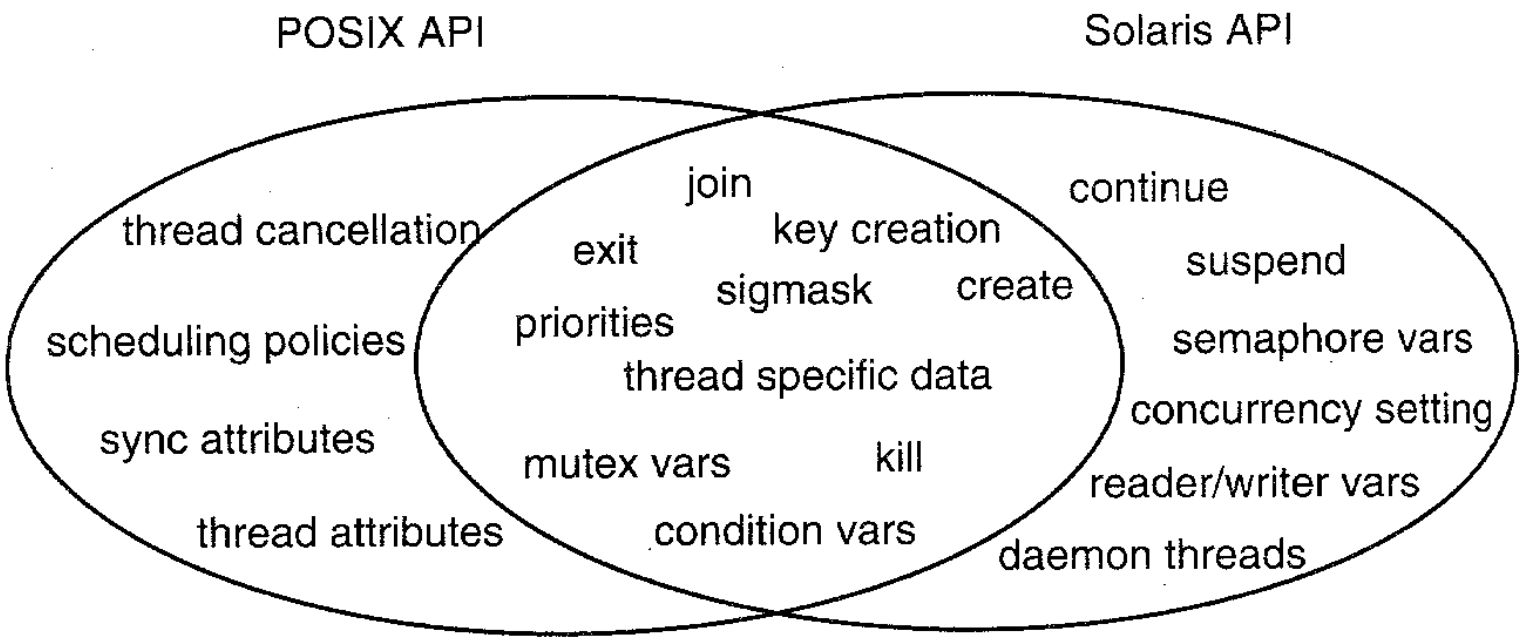


Figure 7-1 POSIX and Solaris API Differences

2) Attribute Objects

- The pthread API permits the programmer to create threads and synchronization variables in many different sate.

- Solaris - using flags
- POSIX - using attribute obkects

- The state information defined in the attribute object is used only on the creation or initialization of the thread or synchro nization variable.

- Two major advantages to using attribute objects:
  - the capability for attribute objects to be implemented on many diff erent systems. - It defines only state not how to implement it.
  - The redability of application code

- But, they require memory

a) Thread and attribute Objects

- The advantage of the attribute mechanism is its simplicity.
- The thread attribute objects can be created and redefined in one section of code and then used throughout the application.

- Scope
  - process-wide: PTHREAD_SCOPE_PROCESS, unbound
  - system-wide: PTHREAD_SCOPE_SYSTEM, bound, kernel entity

- Detach State
  - PTHREAD_CREATE_DETACHED
    - when the thread terminates, all of its resources and exit status will be discarded immediately.
  - PTHREAD_CREATE_JOINABLE
    - thread exit status and resources will be retained until the thread is joined by another thread.

- Sack Address
  - the base address (starting address) of the stack for the thread
    - NULL / non-NULL

- Stack Size
  - if given, it must be at least PTHREAD_STACK_MIN bytes in size.

- Scheduling Policy
  - define how the thread is scheduled and set the priority for the thread

b) Synchronization Variable and Attribute Objects

- scope - private to the process / shared among other processes.
- Once the attribute has been created, it can be used in the initialization of one or more synchronization variable of the same type.
- Mutex Variables
- Condition Variables
- Semaphore Variables
  - Semaphore variables are not officially part of the pthreads specification; they are actually part of the POSIX real-time specification.

3) Cancellation

- Thread cancellation should be used when the further execution of a thread is no longer required.

- Cancellation provides a way of controlling where and when a thread can be canceled, allowing the thread to clean up any resources used and return to a known state.

- Without cancellation:
  - deadlock is possible - terminated while holding a lock
  - memory leak exists - terminated without freeing the memory

- permit / disallow cancellation
- cancellation handler

Cancellation State and Type

- PTHREAD_CANCEL_DISABLE - all cancellation requests are held in a pending state.
- PTHREAD_CANCEL_ENABLE - any pending cancels will be acted upon as defined

- Cancellation type
  - Asynchronous - at any point, dangerous
  - deferred
    - Standard execution points - standard predefined locations
      - A thread blocked in a pthread_cond_wait() call
      - A thread blocked in a pthread_cond_timewait() call
      - A thread waiting for the termination of another thread in pthread_join() call
      - A thread blocked in a sigwait() call
      - In general, any pthread call that may cause a thread to block is a cancellation point. - exception : pthread_mutex_lock() too costly
    - Discrete points - application programmer can set up specific locations
      - pthread_testcancel(): 만약에 pendng하고있는 cancel request가 있으면 자신을 kill 하고 아니면 continue 한다.

Cancellation Cleanup Handles

- When a thread is canceled, pthreads provides a way to clean up the thread's state.
- This is done through a set of cleanup handlers that are called upon cancellation of a thread.

- pthread_cleanup_pop()
- pthread_cleanup_push()

- manual page를 참조할 것: $man pthread_celanup_push

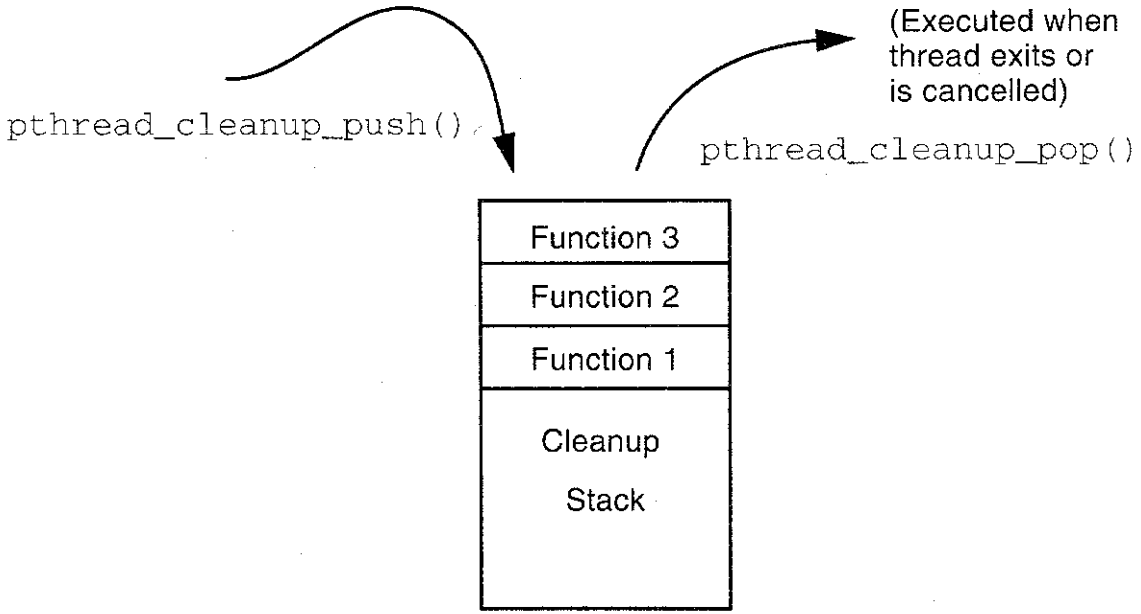


Figure 7-2 Cancellation Cleanup Handler Functions.

- When a thread is cancelled or exits, a call to pthread_cleanup_pop() is effectively made with a non-zero argument.

4) Thread Scheduling

- Pthreads offer a flexibility in scheduling threads onto processors, giving the programmer more options on how a thread will be hand led by the operating system.

- Two major controls: scheduling scope and scheduling policy

- Three scheduling policies defined in POSIX threads:
  - SCHED_FIFO
  - SCHED_RR
  - SCHED_OTHER - like Solaris scheduling

- Three different scheduling classes for mutex locks:
  - PTHREAD_PRIO_NONE - ignore any thread priority
  - PTHREAD_PRIO_INHERIT - upgrade priority of the mutex variable
  - PTHREAD_PRIO_PROTECT - sets a fixed mutex priority. Any thread tha t locks the mutex will inherit the priority of the mutex, if ...