Preview

- Thread Implementation
  - User level thread
  - Kernel level thread
- Multithreading Model
  - One-to-Many
  - One-to-One
  - Many-to-Many
- The Thread Library
- The Threading Issues
  - Issues with fork(), exec()
  - Signal handling
  - Thread termination

Thread Implementation

- Threads may be provided either at user level or kernel level.
  - User level thread – thread are managed by runtime system
  - Kernel level thread– threads are manage by operating system

Thread Implementation
(User Level Thread)

- User level thread
  - kernel is not aware of the existence of threads.
  - Runtime system (thread library) control thread with
    - Creation
    - Destroying
    - Scheduling
  - The application starts with a single thread

Thread Implementation
(User Level Thread)

- User level thread Cont.
  - Advantage
    - Thread switching does not require Kernel mode privileges.
    - User level thread can run on any operating system.
    - Scheduling can be application specific in the user level thread.
    - User level threads are fast to create and manage.
  - Disadvantage
    - In a typical operating system, most system calls are blocking.
    - Multithreaded application cannot take advantage of multiprocessing.

Thread Implementation
(Kernel Level Thread)

- Kernel level thread
  - The thread management is done by the operating system.
  - The Kernel maintains context information for the process as a whole and for individuals threads within the process.
  - Scheduling by the Kernel is done on a thread basis.
  - The Kernel performs thread creation, scheduling and management in Kernel space.

Thread Implementation
(Kernel Level Thread)

- Kernel level thread cont.
  - Advantages
    - Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
    - If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
    - Kernel routines themselves can be multithreaded.
  - Disadvantages
    - Kernel threads are generally slower to create and manage than the user threads.
    - Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

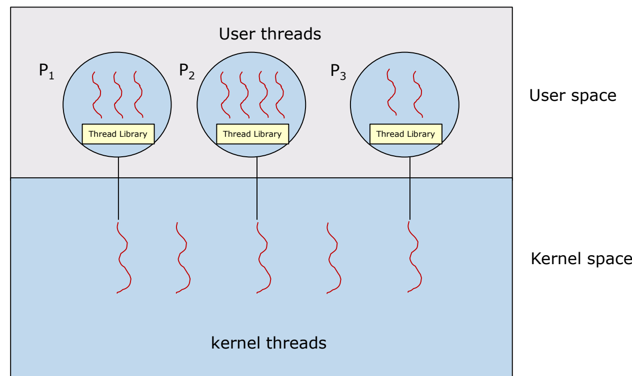
Multithreading Models

- Some operating system provide a combined user level and Kernel level thread facility.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types:
  - Many to many relationship.
  - Many to one relationship.
  - One to one relationship.

Multithreading Models
(Many-to-One Model)

- The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.
- When thread makes a blocking system call, the entire process will be blocked.
- Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

Multithreading Models
(Many-to-One Model)

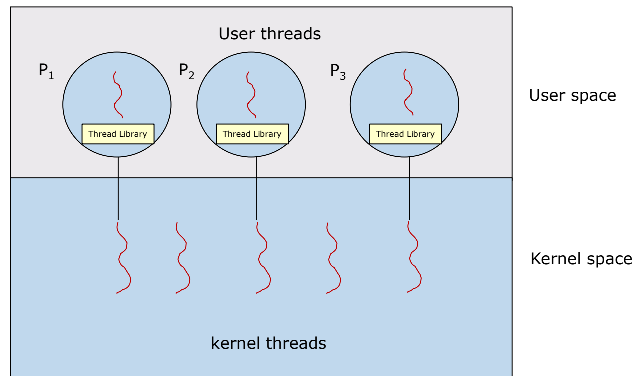


Many-to-One Model

Multithreading Models
(One-to-One Model)

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- Drawback
  - Since creating a user thread requires creating the corresponding kernel thread, a large number of kernel threads may burden the performance of a system.

Multithreading Models
(One-to-One Model)

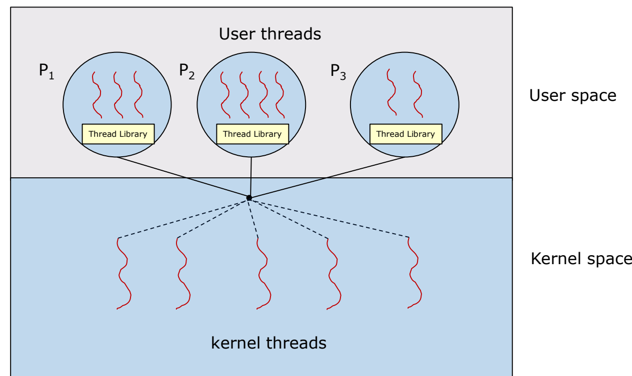


One-to-One Model

Multithreading Models
(Many-to-Many Model)

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- Developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.
- This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Multithreading Models
(Many-to-Many Model)



Many-to-Many Model

Multithreading Models
(Summary)

- The many-to-one model allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time.
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
- The many-to-many model suffers from neither of these shortcomings:
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
  - When a thread performs a blocking system call, the kernel can schedule another thread for execution.

Thread Libraries

- There are two primary ways of implementing a thread library.
  - User-level Library entirely in user space with no kernel support - All code and data structures for the library exist in user space.
  - Kernel-level library supported directly by the operating system - code and data structures for the library exist in kernel space. Invoking a function in the API for the library results in a system call to the kernel.
- Three Main thread libraries in use today
  - Pthread – user level
  - Windows – kernel-level
  - Java - generally implemented using a thread library available on the host system.–Windows API, Linux, typically pthread.

Implicit Threading (optional)

- Implicit Threading?
  - Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads. (programmer need take care of race condition between threads)
  - Creation and management of threading is done by compilers and run-time libraries
  - Application developers need identify tasks—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model
  - Developers only need to identify parallel tasks.
  - Libraries determine the specific details of thread creation and management.

Implicit Threading

- Five methods in text book
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

Threading Issues
(The fork() and exec() system calls)

- When a thread calls fork() system call, there is two possible options
  - Duplicates all threads inside the process
  - Duplicate only the thread that invoked the fork()
- When a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process.
- If exec() is called immediately after forking, the program specified in the parameters to exec() will replace the process.
- If the separate process does not call exec() after forking, the separate process should duplicate all threads.

Threading Issues
(Signal Handling)

- A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled.
- Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-define signal handler that is called to handle the signal.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.

Threading Issues
(Signal Handling)

- Where should a signal be delivered? In general, the following options exist:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.
- What if a thread call `kill(pid t pid, int signal)` to a multithreaded process?
  - Because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.

Threading Issues
(Signal Handling)

- Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid)

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);

                        returns 0 on Ok, returns an error number on error
```

- `pthread_kill` sends the signal sig to thread, a thread in the same process as the caller. The signal is asynchronously directed to thread.

Threading Issues
(Thread Cancellation)

- Thread cancellation involves terminating a thread before it has completed.
  - Ex) if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Ex) A web browser load a web page by several threads. When a user presses a top button on the browser, all threads must be cancelled.
- Cancellation of a target thread may occur in two different scenarios.
  - Asynchronous cancellation – One thread immediately terminates the target thread.
  - Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Threading Issues
(Thread Cancellation)

- The **pthread_cancel**() function requests cancellation of the target thread.
- The target thread is cancelled, based on it's ability to be cancelled.
- When cancel ability is deferred, all cancels are held pending in the target thread until the thread changes the cancel ability, calls a function that is a cancellation point.

Threading Issues
(Thread Cancellation)

- Cancellation points:
  - pthread_cond_timedwait()
  - pthread_cond_wait()
  - pthread_delay_np()
  - pthread_join()
  - pthread_join_np()
  - pthread_extendedjoin_np()
  - pthread_testcancel()

```c
/* thcancel.c demonstrate pthread_cancel() function */
#include <pthread.h>
#include <stdio.h>

void *threadfunc(void *);
void err_sys(char *, int );

int main(int argc, char **argv)
{
   pthread_t tid;
   int rc=0;

   printf("Entering testcase\n");
   /* create a thread */
   if ((rc = pthread_create(&tid, NULL, threadfunc, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   sleep(2);
   printf("Now Canceling the thread\n");
   /* try to calcel the thread created */
   if ((rc = pthread_cancel(tid)) != 0)
      err_sys("ERROR; return code from pthread_cancel() is", rc);
   if ((rc = pthread_join( tid, NULL)) != 0)
         err_sys("ERROR; return code from pthread_join() is", rc);
   sleep(3);
   printf("Main completed\n");
   return 0;
}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
void *threadfunc(void *parm)
{
   printf("Entered secondary thread\n");
   while (1)
  {
     printf("Secondary thread is looping\n");
     pthread_testcancel(); /* cancel point */
     sleep(1);
  }
}
```


Threading Issues
(Thread Cancellation)

```c
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
```

- The **pthread_cleanup_push**() function pushes a clean up function routine, to be called with the single argument, arg, when the thread performs one of the following actions.
  - pthread_exit()
  - pthread_cancel()

Threading Issues
(Thread Cancellation)

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

- The **pthread_cleanup_pop**() function pops the last cleanup handler from the cancellation cleanup stack.
- If the *execute* parameter is nonzero, the handler is called with the argument specified by the **pthread_cleanup_push**() call with which the handler was registered.


```
/*thpushpop.c: demonstrate
  pthread_cleanup_push() and pthread_cleanup_pop() */

#include <pthread.h>
#include <stdio.h>
void err_sys(char *, int );
void cleanupHandler(void *);
void *threadfunc(void *);

int main()
{
   pthread_t tid;
   int rc=0;

   printf("Entering testcase\n");
   /* now creating a thread */
   if ((rc = pthread_create(&tid, NULL, threadfunc, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   sleep(2);

   printf("Now Canceling the thread\n");
   /* now cancelling the created thread */
   if ((rc = pthread_cancel(tid)) != 0)
      err_sys("ERROR; return code from pthread_cancel() is", rc);
   sleep(3);
   if ((rc = pthread_join( tid, NULL)) != 0)
         err_sys("ERROR; return code from pthread_join() is", rc);
   printf("Main completed\n");
   return 0;
}
```

Threading Issues
(Thread Cancellation)

```
void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
void cleanupHandler(void *arg)
{
  printf("Master ask me terminate myself\n");
  sleep(2);
  printf("I will be back!\n");
}
void *threadfunc(void *parm)
{
  printf("Entered secondary thread\n");
  /* push clearnup function after cancell */
  pthread_cleanup_push(cleanupHandler, NULL);
  while (1) {
    printf("Master! Don't terminate me! I want live forever!\n");
    pthread_testcancel(); /* cancel point */
    sleep(1);
  }
  pthread_cleanup_pop(0);
  return NULL;
}
```


Threading Issues
(Thread-Local Storage (TSL))

- Since threads belonging to a process , threads could share the data of the process which is one of the benefits of multithreaded programming.
- In some cases, each thread might need its own copy of certain data (Thread-local storage).
  - For example, in a transaction process system, separate thread are assigned on different transaction service. Each transaction might be assigned a unique ID.
  - To associate each thread with its unique transaction ID, we could use TLS.
  - Local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations.
  - Most thread libraries and compilers provide support for TLS.
    - In pthread : pthread_key_t can be used to declare TLS
    - In Java : ThreadLocal<T> objects with get() and set() method

Threading Issues
(Scheduler Activations)

- Threads Implementation in a process
  - Option A: user-level library (runtime system), within a single-threaded process
    - Runtime system manage whole threads.
    - Library (runtime system) does thread switch
  - Option B: use kernel threads
    - Kernel manage processes and threads
    - Kernel does thread context switching.
    - Simple, but huge transitions between user and kernel mode
  - Option C: Scheduler activations (Hybrid)
    - Kernel allocates lightweight process to user-level runtime system for thread
    - The thread library (runtime system) implements context switch
    - System call I/O that blocks thread and triggers upcall.

Threading Issues
(Scheduler Activations)

- One scheme for communication between the user-thread library(runtime system) and the kernel is known as scheduler activation which is required in many-to-many and two-level models.
- Many system place an intermediate data structure called lightweight process(LWP) between the user and kernel threads.
- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run
- Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also block

Threading Issues
(Scheduler Activations)

- An application may require any number of LWPs to run efficiently.
- Consider a CPU-bound application running on a single processor.
- In this scenario, only one thread can run at a time, so one LWP is sufficient.
- However, an application that is I/O intensive may require multiple LWPs to execute. Typically, an LWP is required for each concurrent blocking system call.
- Suppose that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

Threading Issues
(Scheduler Activations)

- It works following way:
  - The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
  - The kernel must inform an application about certain events. This procedure is known as an upcall. Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.
  - One event that triggers an upcall occurs when an application thread is about to block.
  - The kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
  - The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.