Chapter 4

Threads

Chapter Objectives

To understand concepts of Threads Programs used over the duration of the course.

Objectives

For this chapter, the following are the objectives:
- Introduction to Threads.
- Understanding Architecture and Design of Threads in Linux.

Slide #4-1

Notes

In this chapter, we examine the terms programs, process and threads.
The objective of this chapter is to provide an understanding of the concepts on processes and threads that would be used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - Threads
2. **Description**: A thread is very similar to a process. Threads are abstractions based on the process model, and have current relevance. This chapter provides an introduction to the concepts used in course.
3. **Concepts Covered in Chapter**:
   - Introduction to Threads.
   - Design of Threads.
   - LINUX Notes

4. **Prior Knowledge**:
   Discussion questions are,
   - What are Programs? .. Processes? .. and, Threads?

5. **Teaching & Learning Strategy**:
   - UNIX Processes and Threads: Structure and Layout?,
   - Threads created using `p_thread_create()`
   - Thread Synchronization
   - Thread Cancellation
   - clone()

6. **Teaching Format**:
   Theory + Homework Assignments
7. **Study Time**: 150 Minutes (Lecture & Theory) +
   ~45 minutes (Homework Assignments)
8. **Chapter References**:

Overview of Threads

Why Threads?
- Process Overview
- Disadvantages of Threads

Threads
- Standard and Lightweight.
- Similar to Process
- Kernel View of Threads

Notes:

1. Process Overview

 1.1. One process creates another process using fork() and exec(). It Implemented as a simple, hierarchical parent-child model. The parent process would typically either exit immediately or wait until the child returns.

 fork() returns two completely independent copies of the original process. Each copy of the process has its own address space with its own copy of variables, totally distinct from the same variables in the other process.

 Each Process has Code and Data. Each Process has its own address space, even though they may be running the same "physical program file".

 1.2. However, there are problems.

 The process model has significant system overheads. Each child process inherits a copy of the complete execution context and state of its parent including global variables, open fd's etc. Inter communication between processes is relatively complex, and is extremely resource intensive.

 - high cost of switching between processes Each Process has significant overheads when switching. Process "Context Switch" is expensive.
 - scheduler limits .. ability to efficiently handle number of processes
 - IPC mechanisms are typically slow.

 In retrospect, processes are referred to as "Heavyweight Processes".

2. Threads Overview

   2.1. Threads are similar to proceses. Threads can be very useful.

   Threads **share** a common address space, and, avoid a lot of the inefficiencies of the multi-process IPC model.

   Threads share majority of resources as the parent – address space, global variables

   2.2. Each thread has its own state – program counter, registers, stack etc.

   2.3. There are different thread implementations.
   The major differences are in kernel threads, user-space threads and processes.

3. Historically, there have been two types of threads:

   3.1. User-Level

   Switching is handled in the user space component of the process. The determination of the "next" runnable task is handled by the "coordinator thread" within a process.

   - It can be implemented as a "co-operative multitasking" algorithm. Each task occupies the CPU for some duration, and, voluntarily relinquishes the CPU to the coordinator thread upon expiration of the duration.

   - It can be implemented as a "pre-emptive multitasking" algorithm. The coordinator thread forces the switch by sending a signal or by using timer signals.

   The coordinator thread is performing functions already available as part of the operating system.

   The disadvantages with user-level threads are that a single thread can consume the time slice, and starve other "runnable" threads in the process. Depending on the implementation, user-level threads may not take advantage of multiple CPU's in SMP.

   3.2. Kernel-Level

   Switching is handled in the kernel space component of the process. The determination of the "next" runnable task is handled by the scheduler.

   pthreads, also known as POSIX threads, P1003.1c, or ISO/IEC 9945-1:1990c, provides an API implementation for enabling a multi-threading in an application.

4. Before Linux 2.4, a thread was just a special case of a process.

   4.1. Therefore, one thread could not wait on the children of another thread, even when the latter belongs to the same thread group.

   4.2. However, as POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.

```
int pid;
if (pid=fork()) {
  /* parent code */
  exit(1);
} else {
  /* child code */
  execl( "command", "arg1", "arg2", ...);
  printf("Should never get here...\n");
  exit (-1);
}
```

Overview of Threads .. 2

- Multi-threaded programming vs multi process programming
- Threads in an SMP system.

Slide# 4-3

Notes

1. Overview

   1.1. A thread is a sequential flow of control through a program.

   1.2. Multi-threaded programming is a form of parallel programming, where several threads of executing concurrently in the context of a process.

It differs from multi-process programming in that, all threads share the same resources (memory space, file descriptors), instead of running in their own memory space as is the case with Unix processes.

i.e. all threads are in the same memory space, and can therefore work concurrently on shared data.

2. Threads are useful for several reasons.

   2.1. If there are several threads of a process that can divide the task or communicate together, rather than as one big monolithic sequential program.

   2.2. A program can take advantage of SMP capability on multi-processor machines

   Threads can run in parallel on several processors concurrently. It allows a single process to divide its work between several processors, and be faster than a single-threaded program, which runs on only one processor at a time.

   2.3. Even on uniprocessor machines, threads allow overlapping I/O and computations in a simple way.

Overview of Threads .. 2

- Threads Implementation

Slide# 4-4

Notes

1. Thread implementations adhere to different models:

   1.1. "one-to-one" model,

   Each thread is a separate process in the kernel.

These threads are created with the Linux clone() system call, which is a generalization of fork() allowing the new process to share the memory space, file descriptors, and signal handlers of the parent.

The kernel scheduler schedules threads, similar to scheduling for any regular process.

Advantages of the "one-to-one" model include:

- minimal overhead on CPU-intensive multiprocessing (with about one thread per processor);

- minimal overhead on I/O operations;

- a simple and robust implementation (the kernel scheduler does most of the hard work of scheduling).

The main disadvantage is that context switches are more expensive on mutex and condition operations, which must go through the kernel. This is mitigated to some extent, by the fact that context switches in the Linux kernel are pretty efficient.

1.2. "many-to-one" model uses a user-level scheduler that context-switches between the threads, entirely in user code.

There is only one process running, when viewed from the kernel. This model does not take advantage of multiple processors in a system. The handling of blocking I/O operations may have issues. There are several user-level thread libraries available for Linux.

1.3. The "many-to-many" model combines both kernel-level and user-level scheduling:

several kernel-level threads run concurrently, each executing a user-level scheduler that selects between user threads.

Most commercial Unix systems (Solaris, Digital Unix, IRIX) implement POSIX threads this way.

This model combines the advantages of both the "many-to-one" and the "one-to-one" model, and is attractive because it avoids the worst-case behaviors of both models -- especially on kernels where context switches are expensive, such as Digital Unix.

2. The Linux kernel has an interesting and unique view of threads.

2.1. Essentially, the kernel has no such concept.

2.2. To the Linux Kernel all threads are unique processes.

2.3. At a high-level there is no difference between two unrelated processes and two threads in a single process.

2.4. The kernel views threads as processes that share resources: a process consisting of two threads is considered as two distinct processes that share the kernel resources ... address space, open files etc.

The pThreads Model

Overview of pThreads

- Standard and Lightweight.
- Similar to Process
- API Returns ‘0’ upon success, and error code upon failure. (MT safe)

Slide# 4-5

Notes:

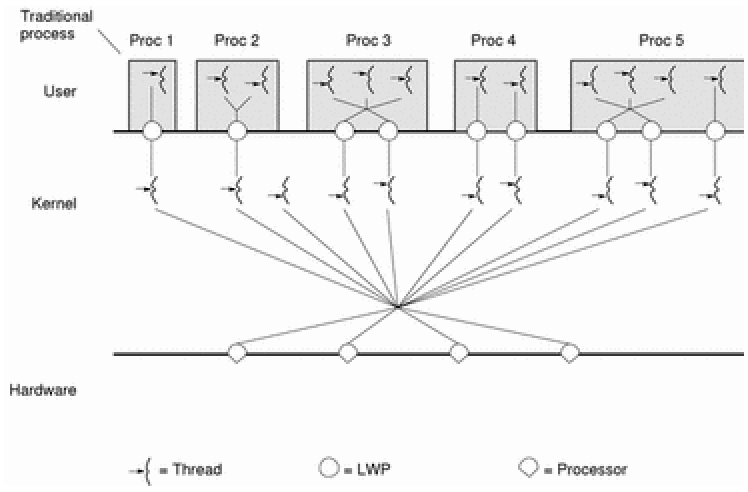
1. pthreads
   1.1. pthreads, also known as POSIX threads, P1003.1c, or ISO/IEC 9945-1:1990c, provides an API implementation for enabling a multi-threading in an application.
2. Posix Threads (or “pThreads”) provides a rich API for developing threaded applications.
   2.1. provides an easy way to create, maintain and destroy threads.
   2.2. provides an easy way to share data between threads.
   2.3. also, provides an easy way to synchronize threads.
3. In general pThread APIs return 0 upon success, and error code upon failure.
   3.1. This is because, the global variable errno is shared between all the threads for a process.
   3.2. Each thread can potentially overwrite the value of errno.
   3.3. Global variables are shared by all the threads including the parent. Variables local to the thread including parameters are private to the thread.

4. All threads within a process share the same address space.

   4.1. A list of created threads, is not maintained.
   A thread does not know the thread that created it.

   4.2. Threads in the same process share:
   - instructions .. code/text
   - data .. almost all
   - fd's
   - signals and handlers
   - cwd
   - uid and gid

   4.3. Each thread has a unique:
   - TID .. thread ID
   - set of registers .. stack pointer
   - stack for local variables, and function return addresses
   - signal mask
   - priority
   - errno



Implementation of Threads for Linux

- Standard and Lightweight.
- Similar to Process
- API Returns ‘0’ upon success, and error code upon failure. (MT safe)

Slide# 4-6

Notes

1. Implementation of Threads for Linux. POSIX threads is a popular API, and there are a variety of implementations available for Linux.

   1.1. GNU Standard C Library

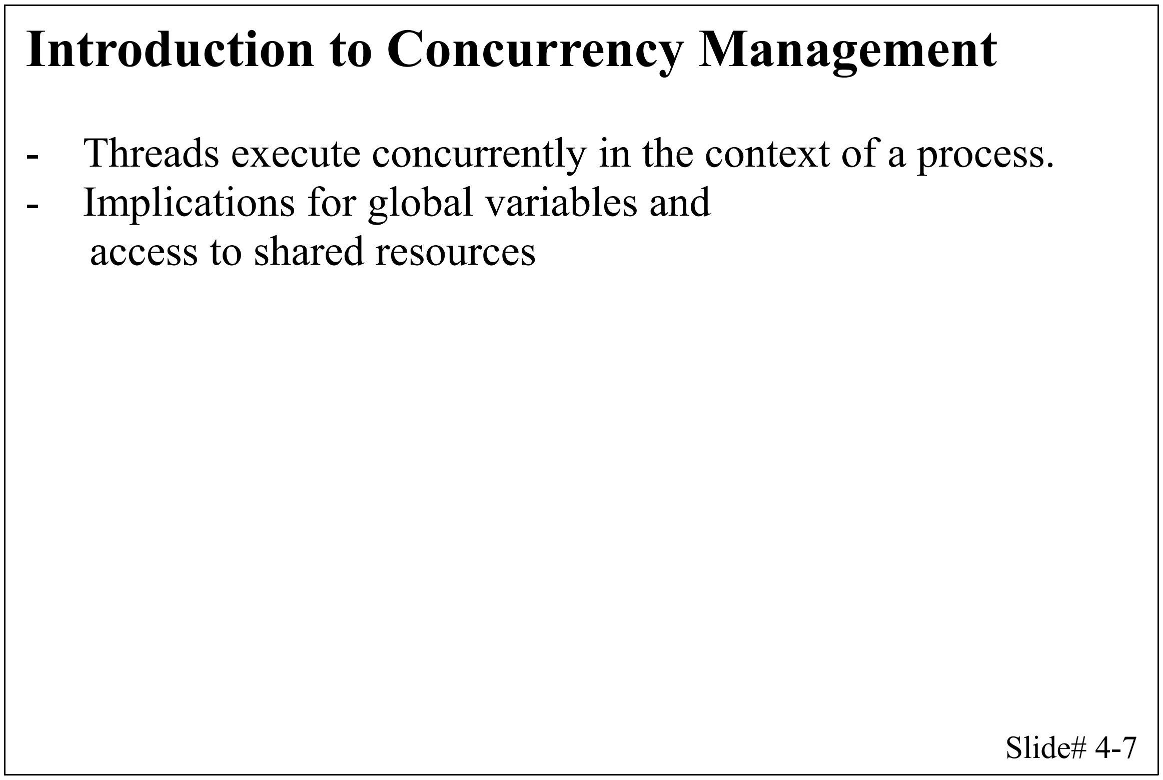
   1.2. Florida State University

1.3. IBM Next Generation Threads

1.4. RTLinux

1.5. RTAI

1.6. Native Posix Threads Library



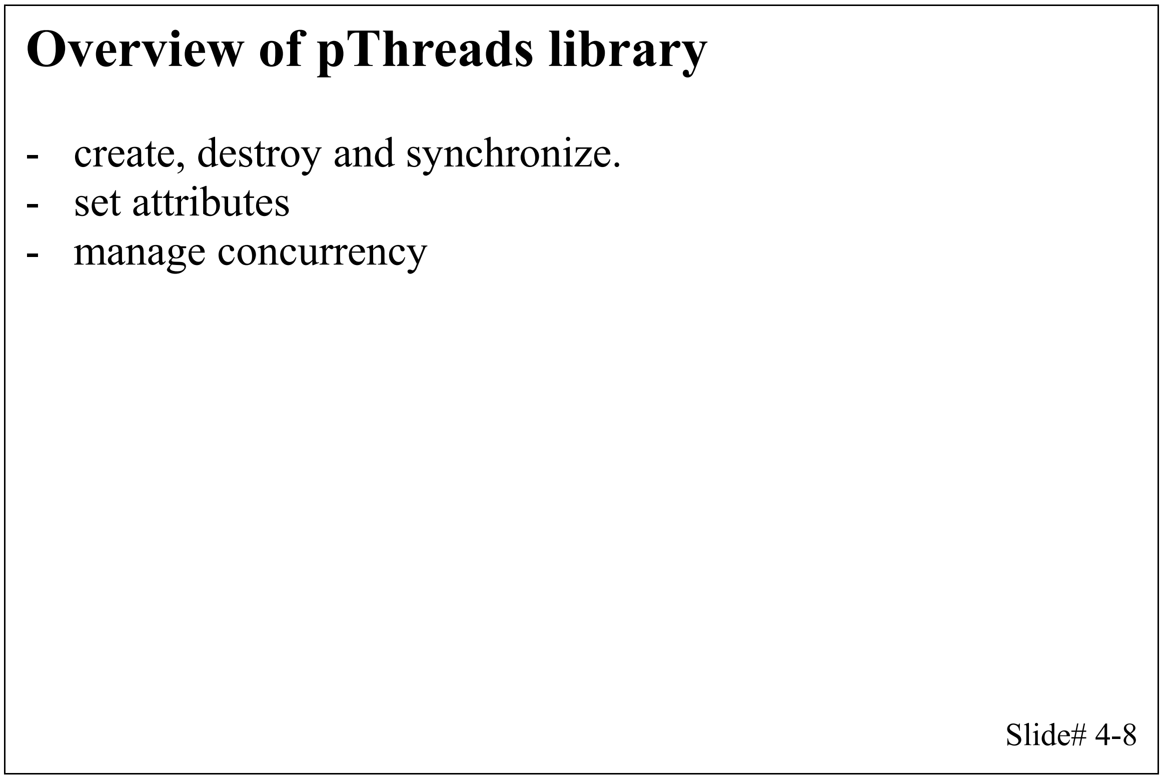
Introduction to Concurrency Management

- Threads execute concurrently in the context of a process.
- Implications for global variables and
  access to shared resources

Slide# 4-7

1. Th
e operating environment decides which LWP should run on which processor and when. It has no knowledge about what user threads are or how many are active in each process.
2. The kernel schedules LWPs onto CPU resources according to their scheduling classes and priorities. The threads library schedules threads on the process pool of LWPs in much the same way.
3. Each LWP is independently dispatched by the kernel, performs independent system calls, incurs independent page faults, and runs in parallel on a multiprocessor system.
4. An LWP has some capabilities that are not exported directly to threads, such as a special scheduling class.

The pThreads Library



Overview of pThreads library

- create, destroy and synchronize.
- set attributes
- manage concurrency

Slide# 4-8

1. Thread operations include creation, termination, synchronization, joins, blocking, scheduling.

create, destroy and synchronize.

- pthread_create()
- pthread_join()
- pthread_exit()

pthread creation

pthread_create()

pthread join

pthread_join()

Slide# 4-9

Thread Creation Guidelines
Here are some simple guidelines for using threads.

- Use threads for independent activities that must do a meaningful amount of work.
- Use threads to take advantage of CPU concurrency.

1. pthread_create()
```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

2. returns ‘0’ upon success and error code upon failure.
3. Args

   3.1. First arg .. pthread_t contains Thread ID or “TID”.
TID is unlike PID .. very little can be done with a TID.

   3.2. Second arg .. prhread_attr_init points to a pthread attribute structure. Attributes include scheduling policy, joining back and scope.

   3.3. Third arg .. start_routine is address of a function executed by the newly created thread.
if and when the function returns, the thread is destroyed.

   3.4. fourth arg .. is the argument to start_routine.
this is the one and onl y argument to start_routine

Notes:

1. pthread_join()
2. Main Ideas
    2.1. cleans up resources .. and, is called once for each created thread.
    2.2. suspends the calling thread until the referenced thread terminates.
    2.3. returns immediately if the referenced thread is already terminated.
    2.4. allows any sibling thread to join another thread.
3. The first argument is the thread id.
4. The second argument is used to obtain the result from the terminating thread.
5. Code Example -- pthread_create() and pthread_join()

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *myfn( void *fnptr );

main()
{
     pthread_t t1, t2;
     char *s1 = "Thread 1";
     char *s2 = "Thread 2";
     int  r1, r2;

     r1 = pthread_create( &t1, NULL, fn, (void*)s1);
     r2 = pthread_create( &t2, NULL, fn, (void*)s2);

     pthread_join( t1, NULL);
     pthread_join( t2, NULL);

     printf("Thread 1 returns: %d\n",r1);
     printf("Thread 2 returns: %d\n",r2);
     exit(0);
}
void *myfn( void *fnptr )
{
     char *msg = (char *) ptr;
     printf("%s \n", msg);
}
```

pthread exit

pthread_exit()

Slide# 4-6

Notes

1. pthread_exit()
   ```
   void pthread_exit(void *retval);
   ```
2. Threads terminate by

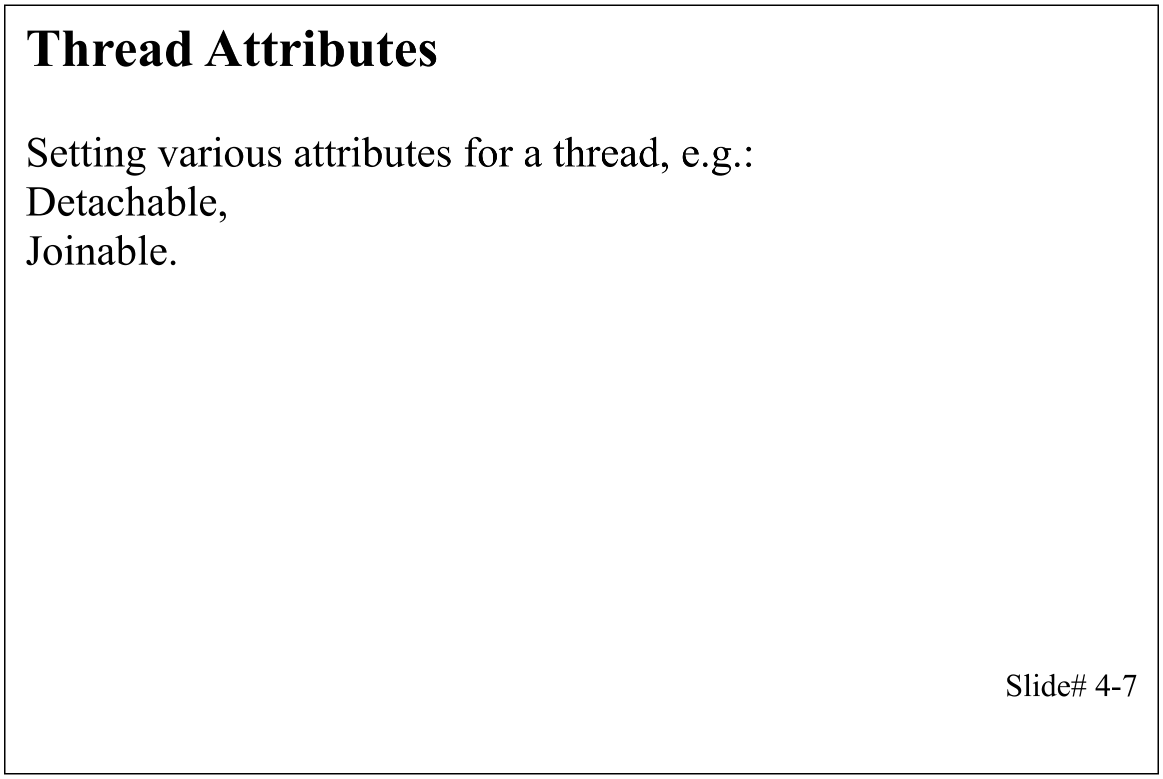
   2.1. explicitly calling pthread_exit, pthread_exit() kills the thread.
pthread_exit never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join. the return pointer *retval, must not be of local scope otherwise it would cease to exist once the thread terminates.
pthread_exit does NOT close files or free memory. This things has to be done by user at appropriate places

   2.2. letting the function return, or

   2.3. a call to the function exit which will terminate the process including any threads.

3. Args
retval - Return value of thread.

Thread Attributes



Thread Attributes

Setting various attributes for a thread, e.g.:
Detachable,
Joinable.

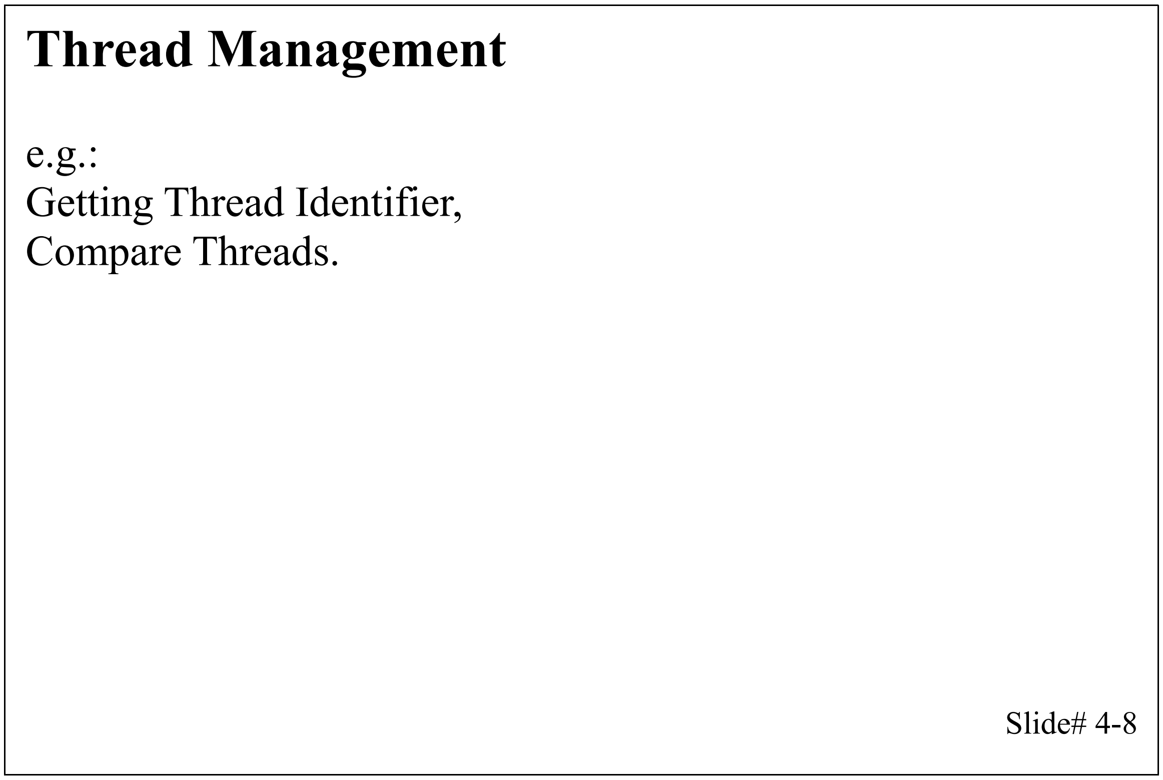
Slide# 4-7

Notes

1. Type: pthread_attr_t
   1.1. Before using, initialize the thread attribute by using pthread_attr_init(pthread_attr_t *attr)
   1.2. Set various attributes by using pthread_attr_set…() function
   1.3. Destroy the attribute by using pthread_attr_destroy(pthread_attr_t *attr)
2. Consider two alternatives during designing the system.
   2.1. Explicitly make the thread “Joinable.
   2.2. Free resources when no one is going to wait for the thread.
3. Thread joining
   3.1. Wait for another thread to complete before continuing
   3.2. Get the return status of the completed thread – Remember pthread_exit call?
   3.3. Signature: pthread_join(pthread_t thread, void **status)
4. Thread Detaching
   4.1. Disallow other threads using “join” on a given thread.
   4.2. Tells the thread library to reclaim storage for the thread after it is done. No one will be waiting to join the thread

5. Join and Detach are set by using thread attributes in pthread_attr_t
   5.1. Creating a joinable or detached thread during pthread_create using pthread_attr – Function pthread_attr_setdetachstate(). Default is 'joinable'.
   5.2. Detaching a thread explicitly using pthread_detach(pthread_t thread)
6. POSIX standard specifies that threads should be created as joinable. However, many implementations allow detaching threads.
7. Code Example -- pthread_create(), pthread_exit() and pthread_join()

```
#include<stdio.h>
#include<pthread.h>

void* fn(void* argval){
    sleep(5);
    fprintf(stderr,"Inside fn with message \"%s\".\n",argval);
    pthread_exit(NULL);
}

int main(){
    pthread_t tid;
    int st, retval;
    pthread_attr_t tattr;
    char* targ=(char*)malloc(sizeof(char)*10);

    pthread_attr_init(&tattr);
    pthread_attr_setdetachstate(&tattr,
PTHREAD_CREATE_JOINABLE);

    strcpy(targ,"abcd");

    pthread_create(&tid, NULL, fn, (void*)targ);

    retval = pthread_join(tid, (void **)&st);
    printf("Exiting ... \n");
    return 0;
}
```



Thread Management

e.g.:
Getting Thread Identifier,
Compare Threads.

Slide# 4-8

Notes

1. Getting the thread identifier
   `pthread_t pthread_self()`
2. Comparing the threads
   `int pthread_equal(pthread_t t1, pthread_t t2)`

   2.1. Do not use == operator on pthread_t
3. A thread can yield the processor, go back and wait in its run queue
   `void sched_yield()`

   3.1. #include sched.h and compile with -lrt if using sched_yield()

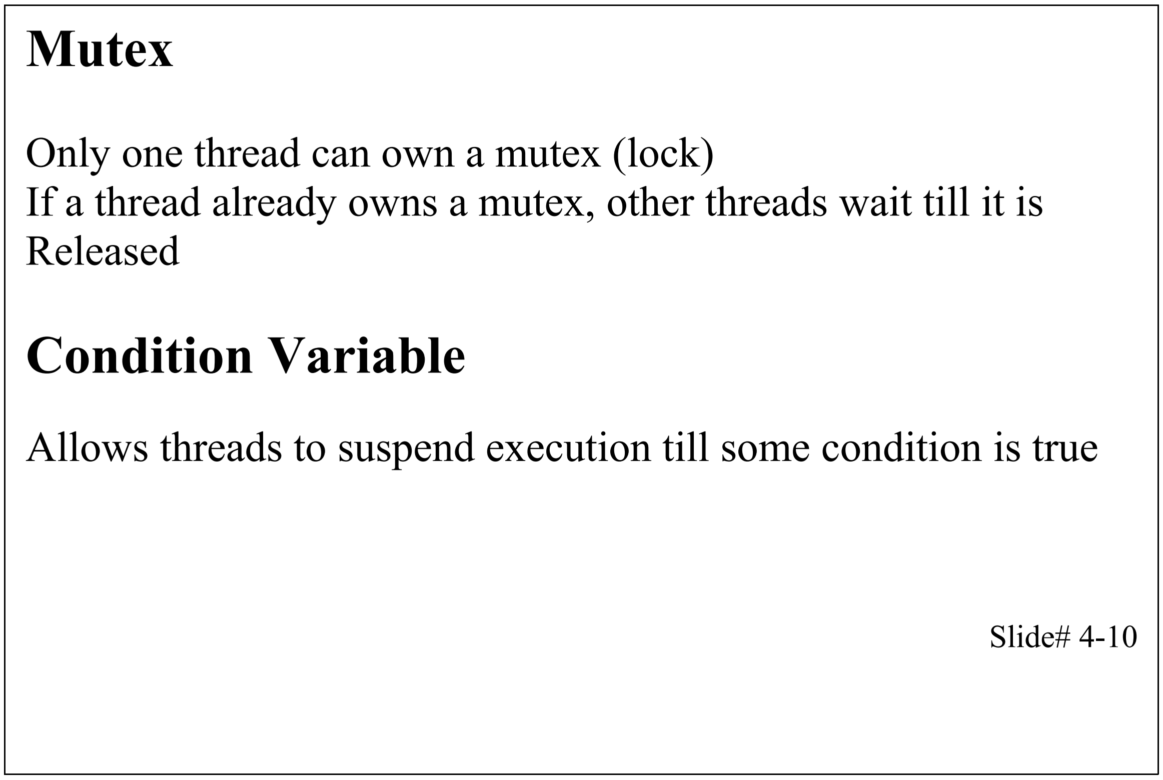
Concurrency Control in pThreads

Mutex
Condition Variables
Joins

Slide# 4-9

Notes

1. Concurrency
2. Mutex.: Use a mutex variable; Acquire the mutex before entering critical section
3. Threads can greatly simplify writing efficient programs.
   These include: Blocking IO: where IO can be done:
   - serially, waiting for each to complete before commencing the next.
   - Using asynchronous IO with signals, polls / selects.
   - Using synchronous IO, with a separate thread for each IO.
4. Multiple Processors.: if the threads library supports SMP, threads can be run on each processor. This is efficient if the process is cpu-intensive.
5. There are problems when multiple threads share a common address space.
   - 5.1. The biggest problem is when two threads race with each other to access the same resource. This is termed as race condition. The outcome is non-deterministic, non-repeatable.
   - 5.2. When accessing global variables, data access is typically non-atomic, the results can be completely non-deterministic.
6. The solution is to use concurrency control mechanism functions that block, if another thread is using the resource.
7. Pthreads use mutex to implement concurrency control.



8.

Mutex

Only one thread can own a mutex (lock)
If a thread already owns a mutex, other threads wait till it is
Released

Condition Variable

Allows threads to suspend execution till some condition is true

Slide# 4-10

Notes:

1. Many threads may read/write the same shared data and can be scheduled independently:
2. Keep the data consistent. Protect critical sections by using locks – Mutex (Mutual exclusion).
3. Declare
   `pthread_mutex_t`
4. Initializing:
   4.1. Static initialization :
   `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER`
   4.2. Dynamic initialization :
   ```
   pthread_mutex_t *mutex;
   pthread_mutex_init(pthread_mutex_t *, pthread_mutexattr_t*)
   ```
5. Set Mutex attributes type
   `pthread_mutexattr_t`
6. Destroying mutex attributes:
   `pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`
7. Destroying the mutex. Used when the mutex is no longer needed.
   `pthread_mutex_destroy()`
8. Locking the mutex. If mutex is free then it locks the mutex. If mutex is owned by another thread then it blocks till the mutex is released
   `pthread_mutex_lock()`

9. Lock a mutex only if it is free. Otherwise return.
   `pthread_mutex_trylock()`
   9.1. If the mutex was acquired, then it returns a value 0.
   9.2. Else will return appropriate busy or error code.
   9.3. Useful for Deadlock prevention.
10. Unlocking a mutex
   `pthread_mutex_unlock()`
11. Mutex does not inherently provide concurrency control or deadlock prevention
   11.1. explicitly use it to avoid deadlocks. check before locking two or more mutexes
   11.2. Acquire the mutex only when needed and release it as soon as it done

12. Code Example

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
int counter=0;
/* Function C */
void functionC()
{
   pthread_mutex_lock( &mymutex );
   counter++;
   pthread_mutex_unlock( &mymutex );
}
```

13. Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *fn();
int  counter = 0;

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;

main()
{
   int retval1, retval2;
   pthread_t t1, t2;

   if( (retval1=pthread_create( &t1, NULL, &fn, NULL)) ) {
      printf("Thread creation failed: %d\n", retval1);
   }
```

```
    if( (retval2=pthread_create( &t2, NULL, &fn, NULL)) ) {
       printf("Thread creation failed: %d\n", retval2);
    }

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    exit(0);
 }

 void *fn()
 {
    pthread_mutex_lock(&m1);
    counter++;
    printf("Inside fn(): Counter value: %d.\n",counter);
    pthread_mutex_unlock(&m1);
 }
```

Notes

1. Creating, Destroying:
   1.1. pthread_cond_init
   1.2. pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
   1.3. pthread_cond_destroy
2. Waiting on condition:
   2.1. pthread_cond_wait
   2.2. pthread_cond_timedwait - place limit on how long it will block.
3. Waking thread based on condition:
   3.1. pthread_cond_signal
   3.2. pthread_cond_broadcast - wake up all threads blocked by the specified condition variable
4. Condition variables help to access data at a given data state efficiently.

4.1. A condition variable must always be associated with a mutex to avoid a race condition. created by one thread preparing to wait, and
another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock.
The thread will be perpetually waiting for a signal that is never sent.

5. Two generic operations:
   5.1. Wait: Wait till the data is in a given state

5.2. Signal: Wake up the waiting threads

6. Code Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

pthread_mutex_t counter_mtx     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condvar_mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condvar_cv  = PTHREAD_COND_INITIALIZER;

void *fn1();
void *fn2();
int  count = 0;
#define COUNT_END  10
#define COUNT_LIMIT1  3
#define COUNT_LIMIT2  6

main()
{
   pthread_t t1, t2;

   pthread_create( &t1, NULL, &fn1, NULL);
   pthread_create( &t2, NULL, &fn2, NULL);
   pthread_join( t1, NULL);
   pthread_join( t2, NULL);

   exit(0);
}

void *fn1() {
   while(1) {
      pthread_mutex_lock( &condvar_mtx );
      while( count >= COUNT_LIMIT1 &&
             count <= COUNT_LIMIT2 ) {
         pthread_cond_wait( &condvar_cv, &condvar_mtx );
      }
      pthread_mutex_unlock( &condvar_mtx );

      pthread_mutex_lock( &counter_mtx );
      count++;
      printf("Inside fn1: Counter value: %d\n",count);
```

```
        pthread_mutex_unlock( &counter_mtx );

        if(count >= COUNT_END) return(NULL);
      }
  }

  void *fn2() {
      while(1) {
         pthread_mutex_lock( &condvar_mtx );
         if( count < COUNT_LIMIT1 || count > COUNT_LIMIT2 )
         {
            pthread_cond_signal( &condvar_cv );
         }
         pthread_mutex_unlock( &condvar_mtx );

         pthread_mutex_lock( &counter_mtx );

         count++;
         printf("Inside fn2: Counter value: %d\n",count);
         pthread_mutex_unlock( &counter_mtx );

         if(count >= COUNT_END) return(NULL);
      }

  }
```

Thread Scheduling

Each thread can have its own scheduling properties

Slide#4-12

Notes

1. Each thread can have its own scheduling properties, specified by
   1.1. thread creation
   1.2. changing attributes of a thread already created
   1.3. defining the effect of mutex on thread scheduling when creating a mutex
   1.4. changing the scheduling of a thread during synchronization operations.
2. default values that are sufficient for most cases.

.

Thread Guidelines

```
- Avoiding Deadlocks and Race Conditions
- Thread Safety
```

Slide# 4-13

Notes

1. Race conditions: Issues: Threads are scheduled by the operating system. Threads are not executed in the order they are created. Threads may execute at different speeds.
   1.1. Mutexes and joins must be utilized to achieve a predictable execution order and outcome.
2. Thread Safety:
   2.1. functions should be "thread safe".

   no static or global variables which other threads may clobber or access assuming single threaded operation.

   For static or global variables, use mutexes. Any function that does not use static data or other shared resources is thread-safe.

   Thread-unsafe functions may be used by only one thread at a time in a program.

   2.2. Many non-reentrant functions return a pointer to static data. It can be avoided by returning dynamically allocated or using caller provided storage. strtok() is non-thread safe function and is also not re-entrant, while the "thread safe" and reentrant version is strtok_r.
3. Mutex Deadlock:
   3.1. occurs when a mutex is never "unlocked".

   3.2. can also be caused by poor application of mutexes or joins especially when applying two or more mutexes to a section of code. threads may wait indefinitely for the resource to become free causing a deadlock.

Chapter 4

Threads

Terms & Concepts Worksheet

Table #1 ("Basic")

1. process.	a. one process creates another process using fork() and exec() b. Implemented as a simple, hierarchical parent-child model. The parent process would typically either exit immediately or wait until the child returns. c. This model has significant system overheads. Each child process inherits the a copy of the complete execution context and state of its parent including global variables, open fd's etc. Inter communication between processes is relatively complex, and is extremely resource intensive.
2. threads.	a. similar to proceses. b. share majority of resources as the parent – address space, global variables c. each thread has its own state – program counter, registers, stack etc. d. There are different thread implementations. The major differences are in kernel threads, user-space threads and processes. e. Before Linux 2.4, a thread was just a special case of a process. Therefore, one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, as POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the same thread group.
3. threads.	a. kernel and system changes that affect how applications spawn and manage other processes and threads in 2.6 .. new threading model implemented through NPTL (Native POSIX Threading Library). mainly, in reference to system and application run-time libraries (.so). b. LinuxThreads is the standard threads library prior to 2.6 .. largely POSIX compliant. c. Both LinuxThreads and NPTL use the same names for libraries libpthread.a and, libpthread.so – in order to minimize changes to existing Makefiles and build environments. - LinuxThreads uses a compile-time setting for the number of threads that a single process can create. It uses a per process manager thread to create and coordinate threads in a process. - There is a higher overhead of creating and destroying threads. Due to the lack of per-thread synchronization primitives, there is a limitation on the number of threads that can be running in a LinuxThreads implementation.

	- Under LinuxThreads, each thread had a unique PID. - Certain LinuxThreads functions are not available in NPTL. d. IBM introduced Next Generation POSIX Threads (NGPT) as an external threading library in 2.4. It worked complementary to LinuxThreads and provided more POSIX compliance and better performance than LinuxThreads. e. NPTL replaces LinuxThreads and NGPT. It provides high-performance threading support needed for high capacity or high traffic requirements. f. NPTL supports mutexes for concurrency management and higher parallelism. NPTL is a POSIX compliant threading implementation. It handles signals between processes and between all threads within a process. g. In NPTL, each thread shares the same PID, and the Thread ID is used to uniquely identify individual threads. h. fork() and exec()are thread aware. The thread inherits the PID of its caller. The parent of a multi-threaded application is notified of a child's termination only when the entire process terminates. Functions registered with the pthread_at_fork() are no longer run when a vfork() occurs. i. NPTL does not have a manager thread – therefore, applications that kept track of threads and identified manager threads for a specific purpose.

Chapter 4

Threads

Assignment Questions

1.1 Terms Review: (One–Liners)
a) pthreads b) mutex c) condition variables d) race condition

Chapter 4

Threads

Assignment Questions

1. Modify the simple program from Chapter 4 that prints “Hello World” in the background .. to use 2 threads in a loop 5 times with a sleep 10 secs.
2. Modify the word count assignment from Chapter 4 to count characters, words and lines individually as a thread, and print the output from main().

Extra Credit

Chapter 4

Threads

Useful Links

http://linas.org/linux/threads-faq.html#ThreadsDefinition

http://www.linuxdevices.com/articles/AT6753699732.html

pthreads:

https://computing.llnl.gov/tutorials/pthreads/

https://computing.llnl.gov/LCdocs/pthreads/index.jsp

NPTL in particular:
https://computing.llnl.gov/LCdocs/pthreads/index.jsp?show=s9.3

Also, as a general interest

https://computing.llnl.gov/?set=training&page=index