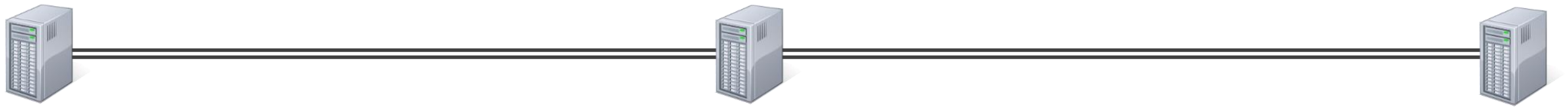
Threads

(Operating Systems)



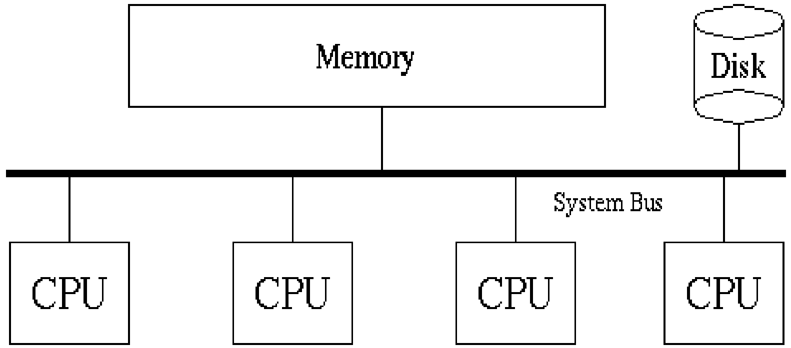
Deepika H V

C-DAC Bengaluru

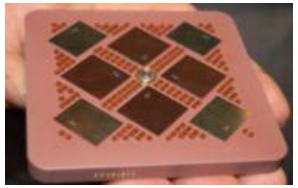
(deepikahv@cdac.in)

Symmetric MultiProcessor

- **SMP – computer architecture where two or more identical processors can connect to a single shared memory.**



Shared Memory Machine



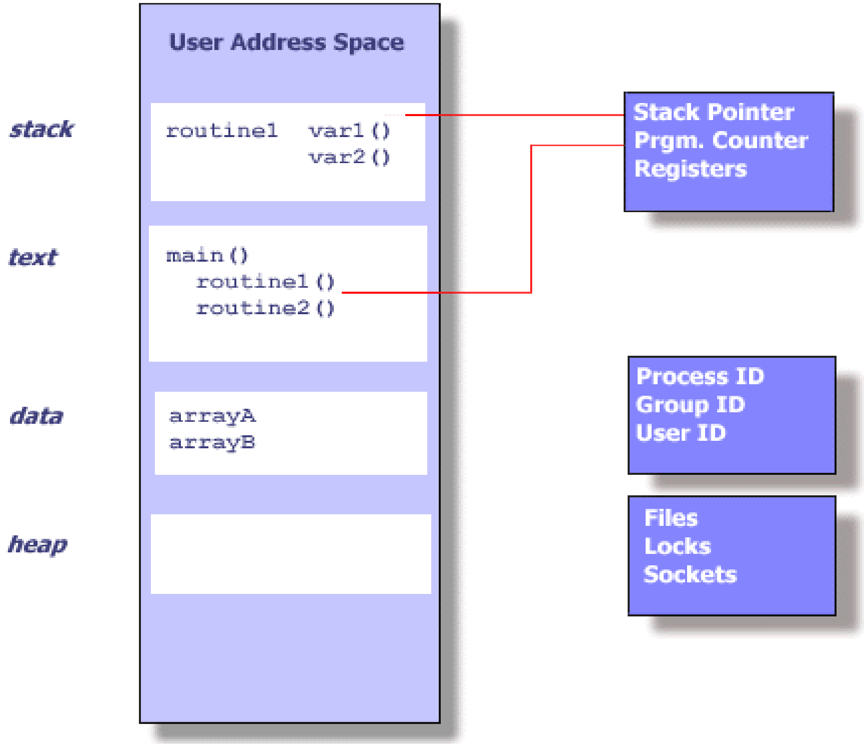
- **What is process?**
  - Program in execution

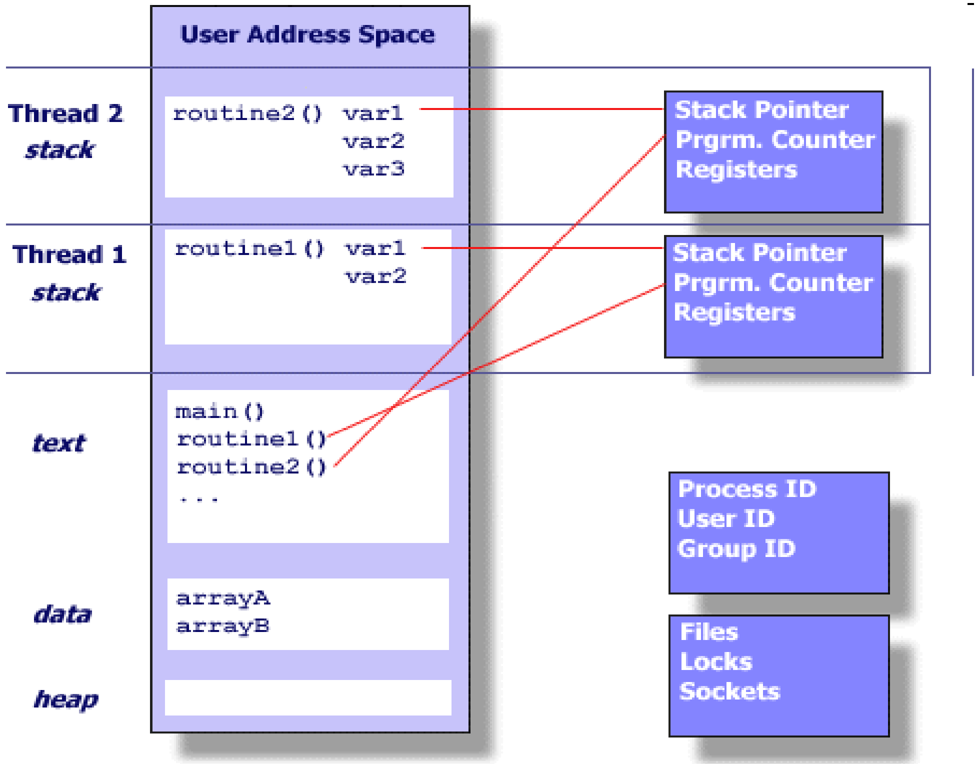
- **What is thread?**
  - Is an independent /different stream of control that can execute its instructions independently and can use the process resources

- **What is the connection b/w process and thread ?**

Thread

- Imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to run simultaneously and/or independently. That would describe a "multi-threaded" program





Thread Features

- **So, in summary, in the UNIX environment a thread:**
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently (and dependently)
  - Dies if the parent process dies
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

- **As threads within the same process share resources:**
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

	Process	Threads
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
4	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
5	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Value of Using Threads

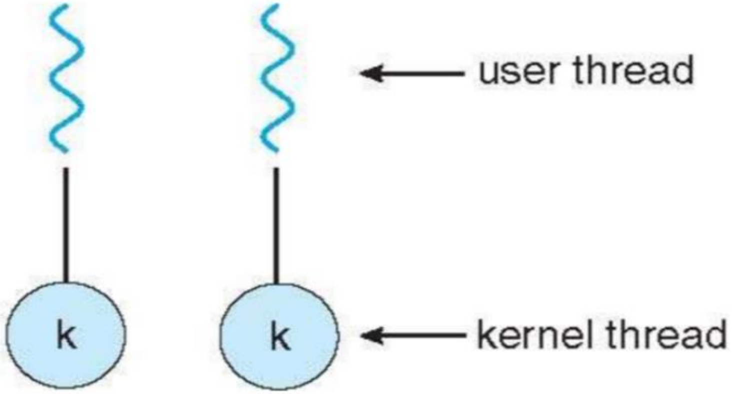
- Performance gains from multiprocessing hardware (parallelism)
- Increased application throughput
- Increased application responsiveness
- Replacing process-to-process communications
- Efficient use of system resources
- Simplified signal handling
- The ability to make use of the inherent concurrency of distributed objects

Types of Threads

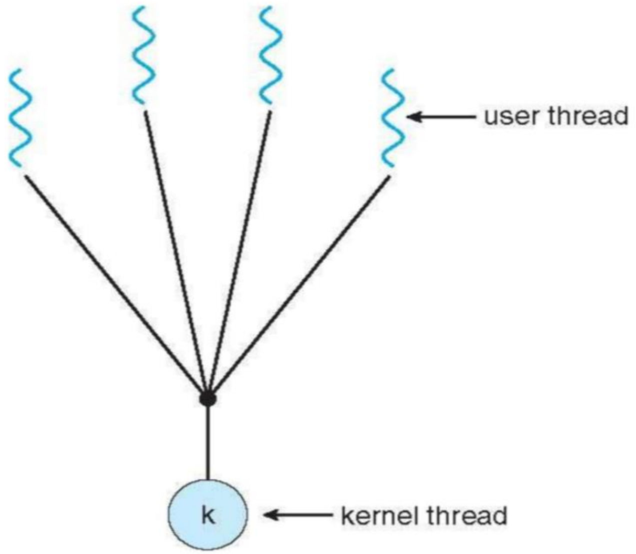
	User Level Thread	Kernel Level Threads
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.

Multithreading Models

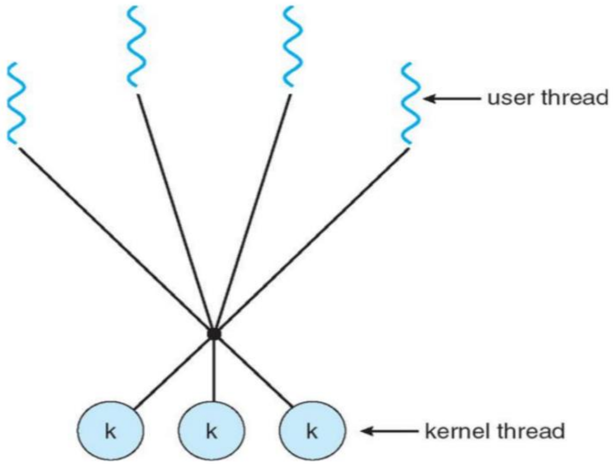
- **One to One**
  - Windows NT, Linux



- **Many to one**
  - Solaris Green Threads and GNU portable threads



- **Many to Many**
  - Solaris prior to ver 9
  - Windows 2000



Posix Threads (pthreads)

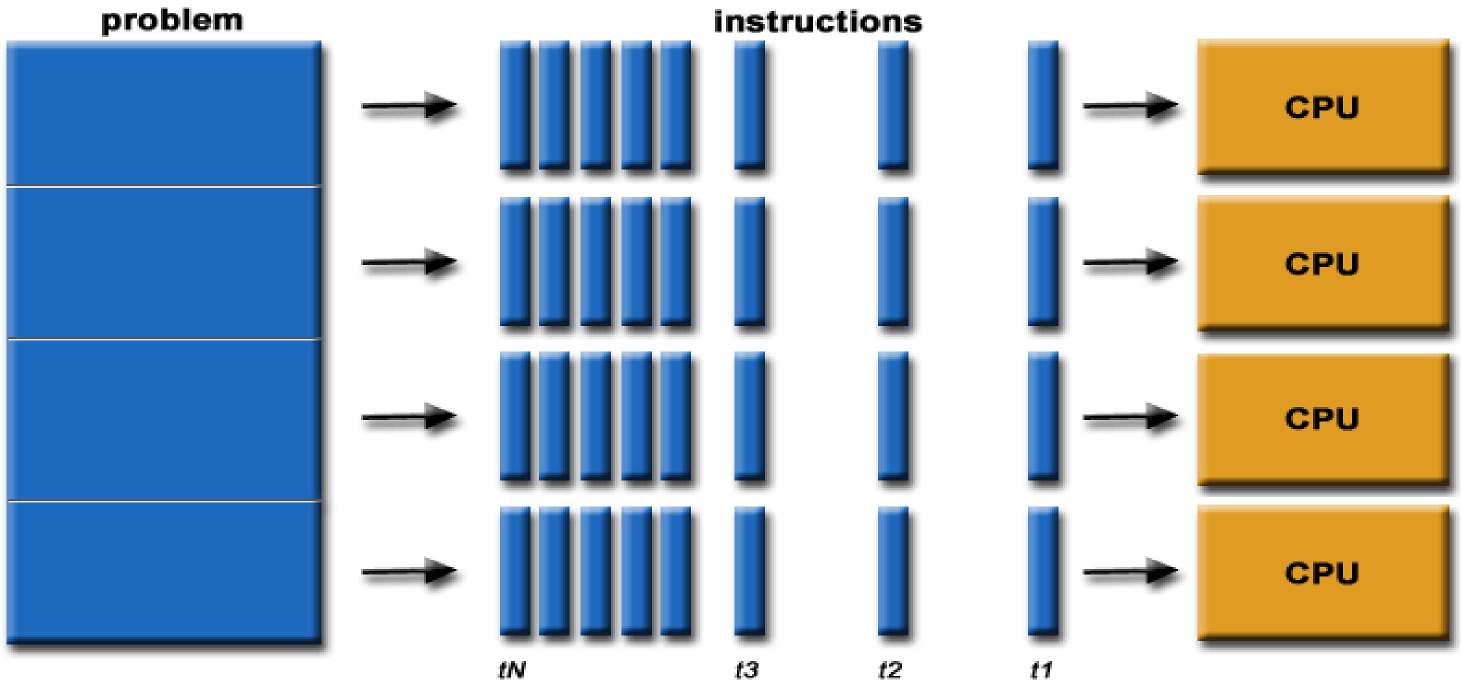
- – specified by the IEEE POSIX 1003.1c standard (1995).
- – set of C programming types & procedure calls, implemented with a pthread.h header file and a thread library.

- **Why Pthreads.**
  - – 5000 threads/process.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

When is threading Useful

- Independent Tasks
- Servers
- Repetitive tasks
- Asynchronous events



- **Considerations For Thread Programming**
  - Problem partitioning and complexity
  - Load balancing
  - Data dependencies
  - Synchronization and race conditions
  - Data communications
  - Memory, I/O issues

Naming convention

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

Compilation

Compiler / Platform	Compiler Command	Description
IBM AIX	`xlc_r / cc_r`	C (ANSI / non-ANSI)
	`xlC_r`	C++
	`xlf_r -qnosave` `xlf90_r -qnosave`	Fortran - using IBM's Pthreads API (non-portable)
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PathScale Linux	`pathcc -pthread`	C
	`pathCC -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, AIX	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++

Concept

- Concept of opaque objects pervades the design of API.
- Pthreads has over 100 subroutines
- For portability, pthread.h header file should be used for accessing pthread library.
- POSIX standard defined only for C language
- Once threads are created they are peers and may create other threads.
- Maximum number of threads created is implementation dependent.

1. Thread Management

- pthread_create (thread,attr,start_routine,arg)
- pthread_exit (status)
- pthread_attr_init (attr)
- pthread_attr_destroy (attr)
- pthread_join (threadid,status)
- pthread_detach (threadid,status)

Pthread_create

- **pthread_create (thread, attr, start_routine, arg)**
  - creates a new thread and makes it executable.

- **thread: An unique identifier for the new thread returned by the subroutine.**
- **attr: An attribute object that may be used to set thread attributes. NULL for the default values.**
- **start_routine: the C routine that the thread will execute once it is created.**
- **arg: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.**

Termination

- Thread returns from main routine.
- Thread calls pthread_exit (status) . This is used to explicitly exit a thread
- the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- Thread is cancelled by other thread – pthread_cancel()
- Entire process is terminated.

Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Example 2 - Thread Argument Passing

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

- **pthread_attr_getstacksize (attr, stacksize)**
- **pthread_attr_setstacksize (attr, stacksize)**
- **pthread_attr_getstackaddr (attr, stackaddr)**
- **pthread_attr_setstackaddr (attr, stackaddr)**

Mutex

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- Mutexes can be used to prevent "race" conditions.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Sequence

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Mutex Routines

- pthread_mutex_init (mutex,attr)
- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutexattr_destroy (attr)
- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Condition Variable

- Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread

Join / Continue

- pthread_cond_init (condition,attr)
- pthread_cond_destroy (condition)
- pthread_condattr_init (attr)
- pthread_condattr_destroy (attr)
- pthread_cond_wait (condition,mutex)
- pthread_cond_signal (condition)
- pthread_cond_broadcast (condition)

Thank You