Thread Programming (Pthread)

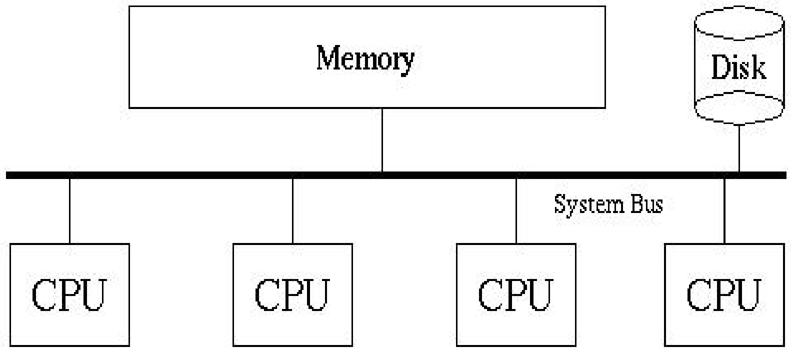
Deepika H.V

System Software Development Group,

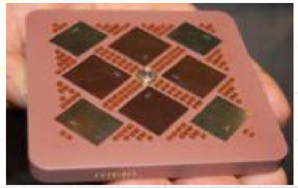
C-DAC,Bangalore

Symmetric MultiProcessor

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

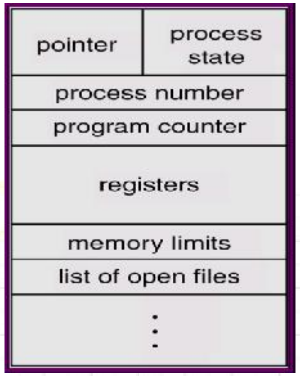


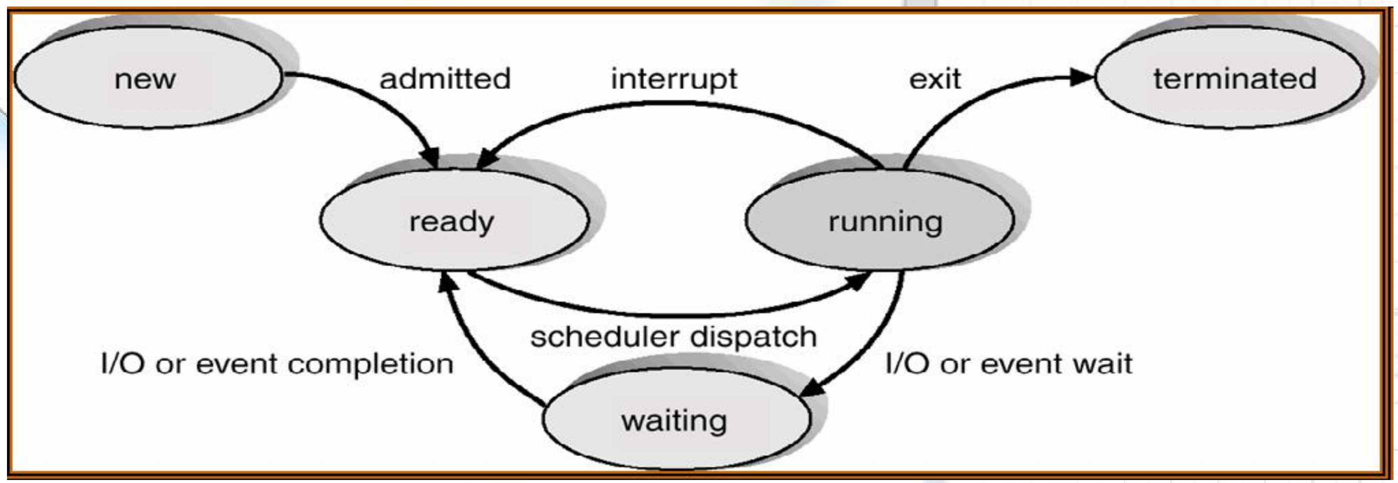
Shared Memory Machine



Process

- **Program in execution is called a process the address space map,**
  - the current status of the process,
  - the execution priority of the process,
  - the resource usage of the process,
  - the current signal mask,
  - the owner of the process.





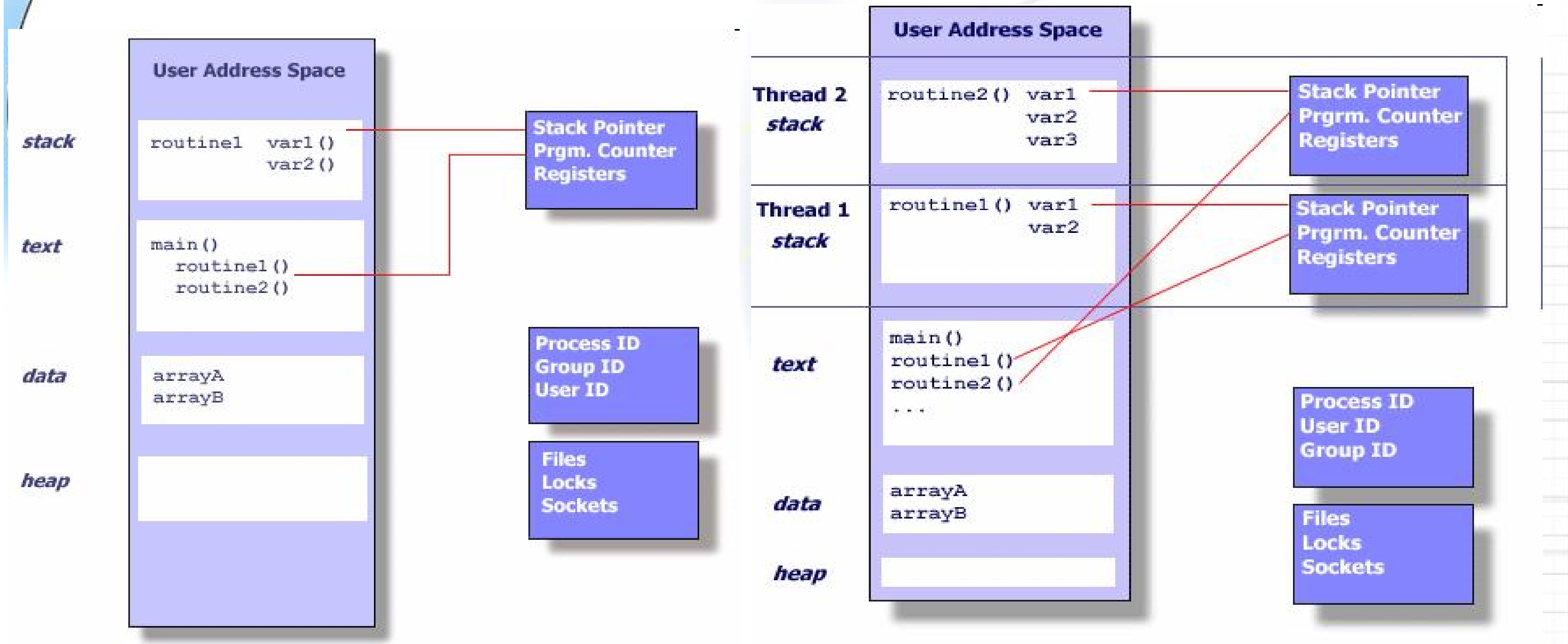
What is Thread?

- Is an independent /different stream of control that can execute its instructions independently and can use the process resources.

- What is 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 - or something similar
- -- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

- **Because 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.

Kernel Interactions of Thread

- **Kernel Interaction/Entities?**
  - a. concurrency v/s parallelism
  - b. system calls
  - c. signals
  - d. synchronization
  - e. scheduling -- One to One
    - Many to One
    - Many to Many

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

Opearation of System Call

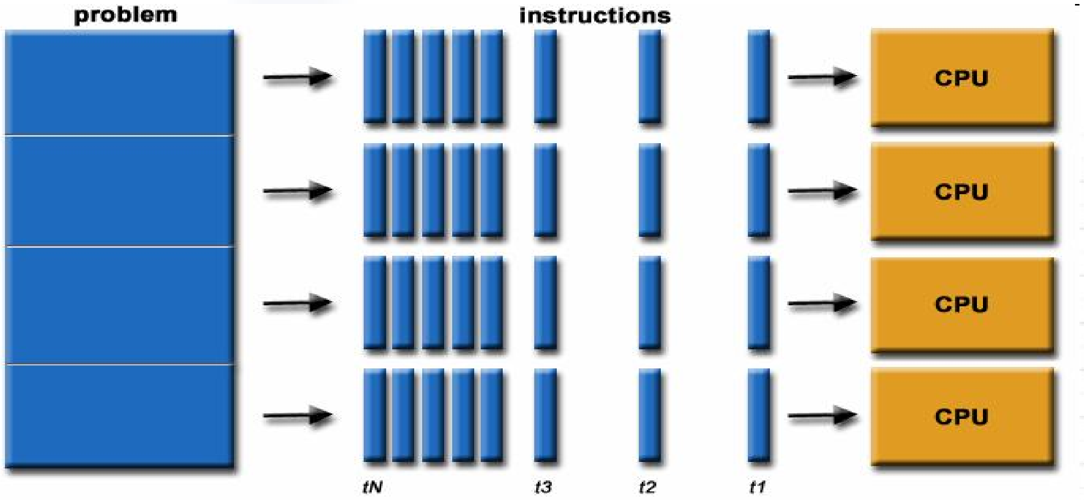
- When a process makes a system call, the following events occur:
- **1.** The process traps to the kernel.
- **2.** The trap handler runs in kernel mode, and saves all of the registers.
- **3.** It sets the stack pointer to the process structure's kernel stack.
- **4.** The kernel runs the system call.
- **5.** The kernel places any requested data into the user-space structure that the programmer provided.
- **6.** The kernel changes any process structure values affected.
- **7.** The process returns to user mode, replacing the registers and stack pointer,and returns the appropriate value from the system call.

Operation of Signal

- 1. The program will call sigaction() to declare some function to be the *handler* for a given signal (say, function foo() will handle SIGUSR1). The kernel will put a pointer to that handler into the process structure's signal dispatch table.
- **2.** Next, your program will call sigprocmask() to tell the kernel which signals it is willing to accept (here, SIGUSR1, yes; SIGUSR2, no).
- **3.** Finally your program takes off and starts doing what you wrote it to do.
- **4.** Now, when some other process sends your process SIGUSR1, your program will stop what it's doing...
- **5.** and run the handler code you wrote. 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.
- **6.** 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.

When threading is useful

- Independent tasks
- Servers
- Repetitive tasks



Thread Programming Models

- *Manager/worker*
- Pipeline
- Peer

Considerations For Thread Programming

- Problem partitioning and complexity
- Load balancing
- Data dependencies
- Synchronization and race conditions
- Data communications
- Memory, I/O issues

POSIX threading APIs

The Supercomputing People

Overview

- **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.

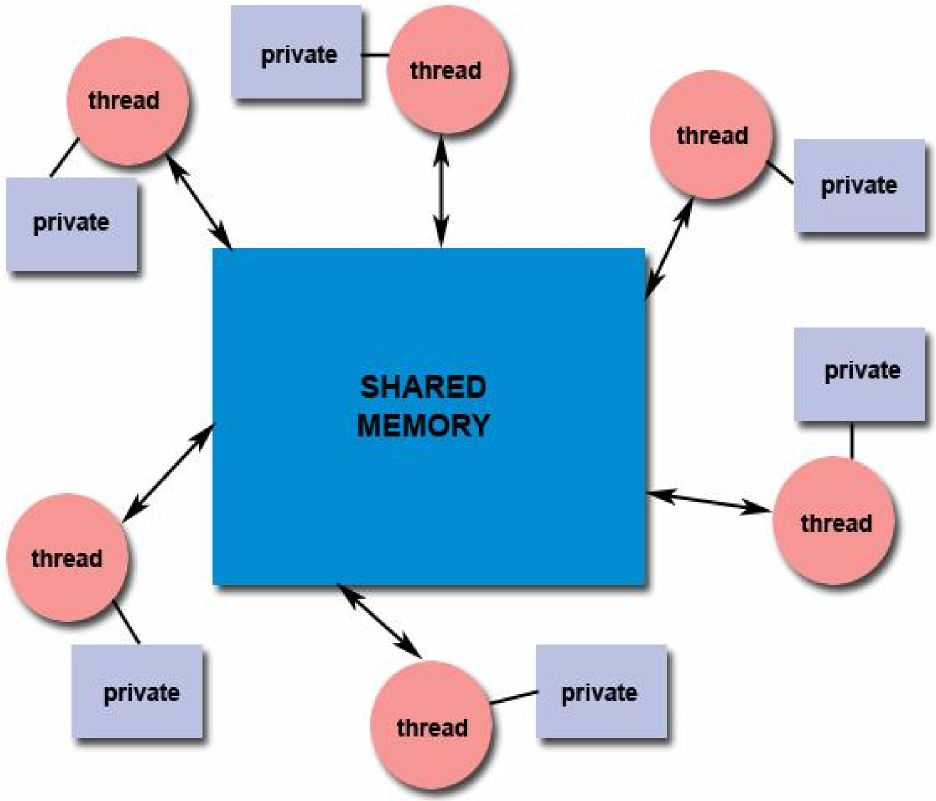
- **Why Pthreads.**

  Eg: 5000 threads/process creation.

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

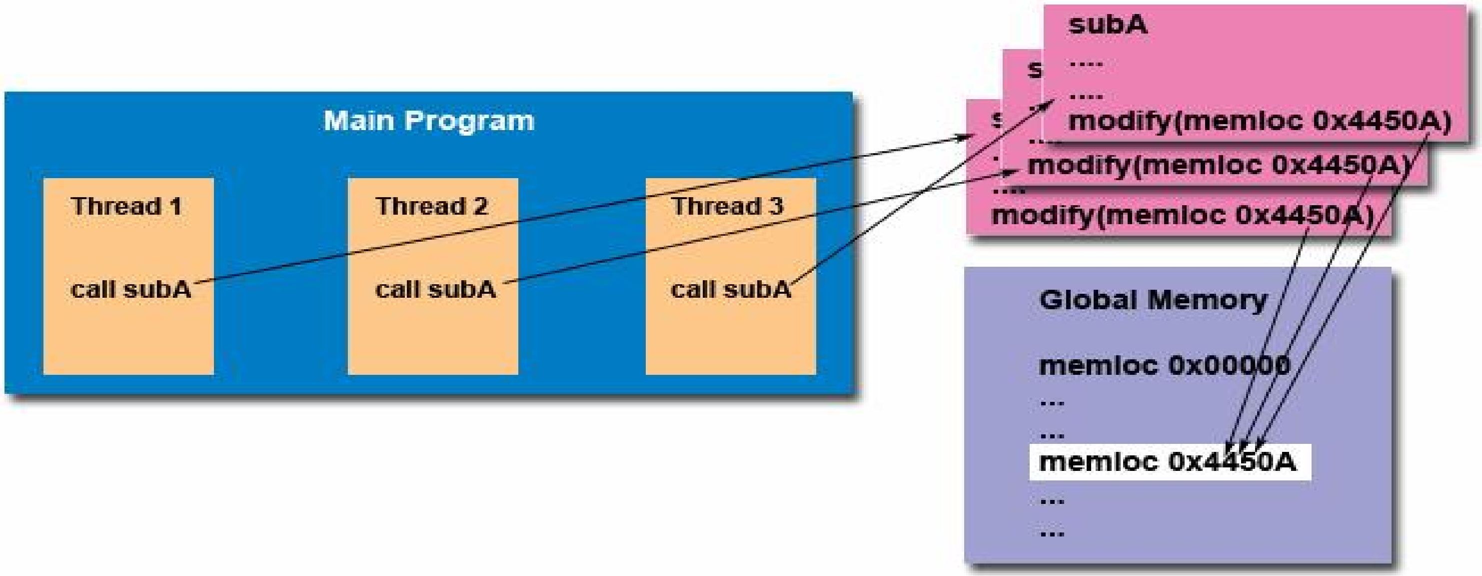
Thread - Shared Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Thread Safeness

The ability to execute multiple threads simultaneously without clobberring shared data or creating race conditions



Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness

Classification of Pthread APIs

- Pthreads API can be informally grouped into three major classes:
- **Thread management**: work directly on threads - creating, detaching, joining, etc.
- **Mutexes:** deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- **Condition variables**: address communications between threads that share a mutex. This class includes functions to create, destroy, wait and signal based upon specified variable values.

Naming Convention in Pthread Prog.

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

Compiling Threaded Programs

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

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 passed.**

Terminating threads

- 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.

The Supercomputing People

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);
}
```

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!

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]);
   ...
}
```

The Supercomputing People

```
#define NUM_THREADS     4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join()
```

```
                is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Contd…

- pthread_attr_getstacksize (attr, stacksize)
- pthread_attr_setstacksize (attr, stacksize)
- pthread_attr_getstackaddr (attr, stackaddr)
- pthread_attr_setstackaddr (attr, stackaddr)
- pthread_self ()
- pthread_equal (thread1,thread2)
- pthread_once (once_control, init_routine)

Mutex Variables

- 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

A typical sequence in the use of a mutex

- 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

- Statically, when it is declared. For example:
  pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
- Dynamically :
  pthread_mutex_init (mutex,attr)

- pthread_mutex_destroy (mutex)
- pthread_mutexattr_init (attr)
- pthread_mutex_destroy (attr)
- pthread_mutex_lock (mutex)
- pthread_mutex_trylock (mutex)
- pthread_mutex_unlock (mutex)

Condition Variables

- 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.

Condition Variables implementation

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

Condition Variables Routines

- Dynamically, when declared
  pthread_cond_init (condition,attr) or
  Statically, when it is declared.
  pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;

- 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