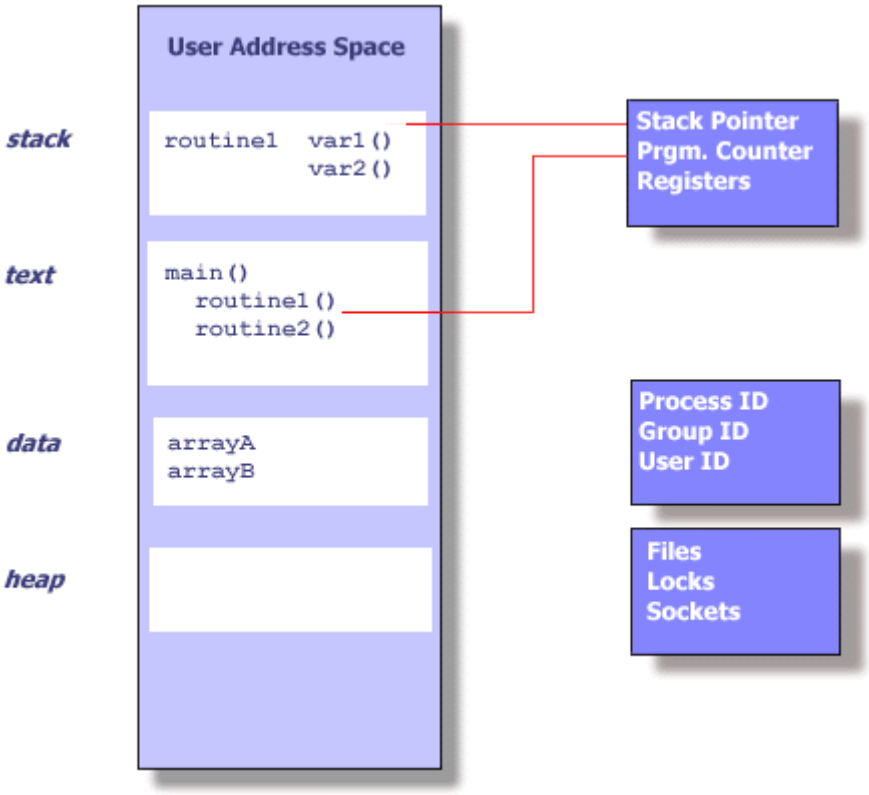
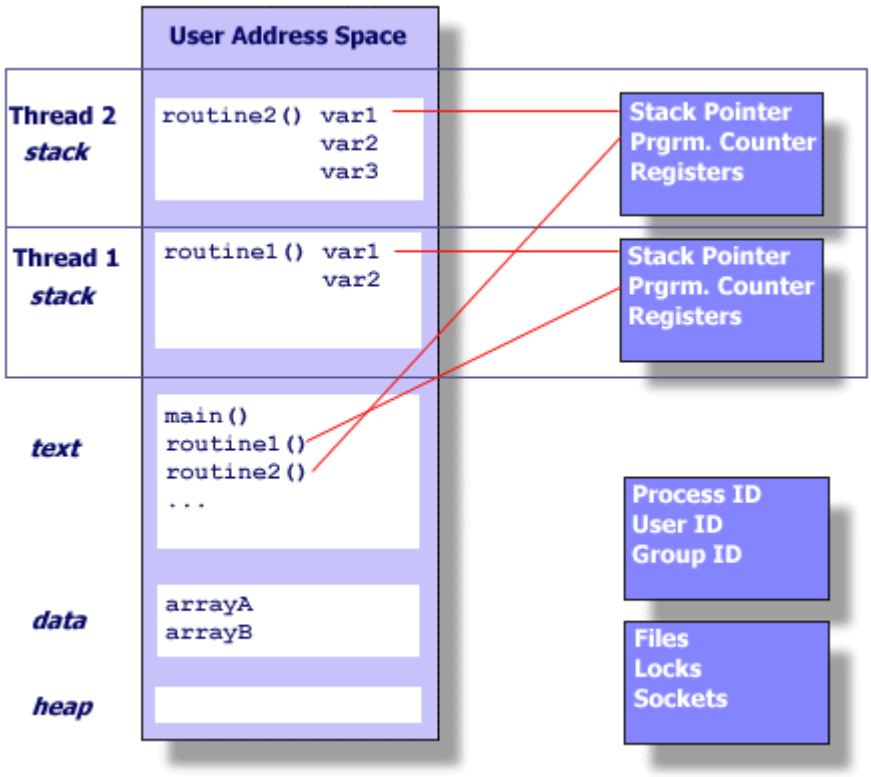
What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.
- How is this accomplished?
- Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:
    - Process ID, process group ID, user ID, and group ID
    - Environment
    - Working directory.
    - Program instructions
    - Registers
    - Stack
    - Heap
    - File descriptors
    - Signal actions
    - Shared libraries
    - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.
- This independent flow of control is accomplished because a thread maintains its own:
- Stack pointer
- Registers
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Thread specific data.



UNIX PROCESS



THREADS WITHIN A UNIX PROCESS

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.

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
    - For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
    - Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
    - Most hardware vendors now offer Pthreads in addition to their proprietary API's.

- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.

- Some useful links:
    - standards.ieee.org/findstds/standard/1003.1-2008.html
    - www.opengroup.org/austin/papers/posix_faq.html
    - www.unix.org/version3/ieee_std.html

- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library - though this

library may be part of another library, such as `libc`, in some implementations.

Why Pthreads?

Light Weight:

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

Efficient Communications/Data Exchange:

- The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.
- MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).
- For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
- In the worst case scenario, Pthread communications become more of a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.

Other Common Reasons:

- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
    - Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
    - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
    - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Designing Threaded Programs

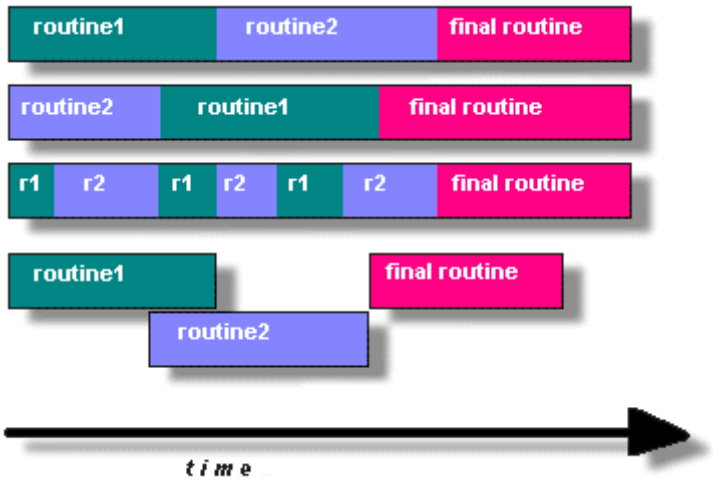
Parallel Programming:

- On modern, multi-core machines, pthreads are ideally suited for parallel programming, and

whatever applies to parallel programming in general, applies to parallel pthreads programs.

- There are many considerations for designing parallel programs, such as:
  - What type of parallel programming model to use?
  - Problem partitioning
  - Load balancing
  - Communications
  - Data dependencies
  - Synchronization and race conditions
  - Memory issues
  - I/O issues
  - Program complexity
  - Programmer effort/costs/time

In general though, in order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



Programs having the following characteristics may be well suited for pthreads:

- Work that can be executed, or data that can be operated on, by multiple tasks simultaneously:
- Block for potentially long I/O waits
- Use many CPU cycles in some places but not others
- Must respond to asynchronous events
- Some work is more important than other work (priority interrupts)

- Several common models for threaded programs exist:

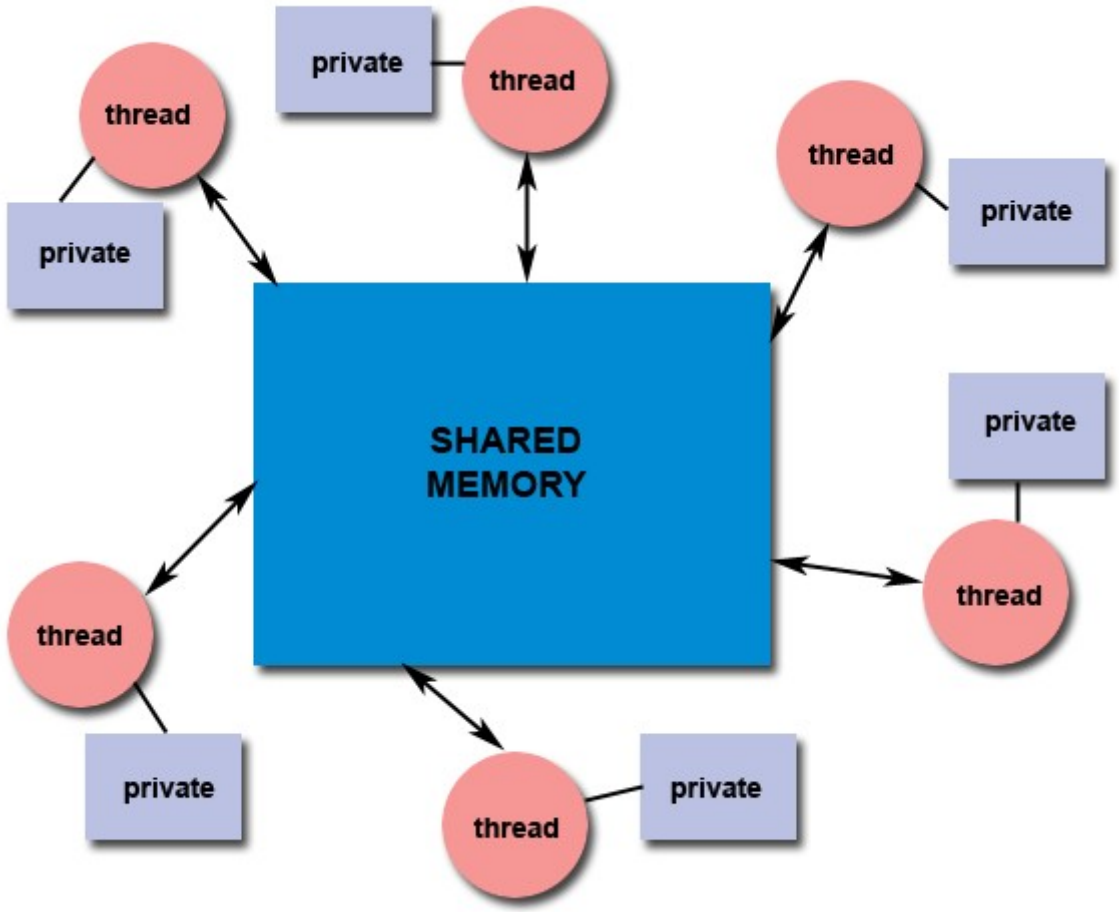
- ***Manager/worker:*** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks. At least two

forms of the manager/worker model are common: static worker pool and dynamic worker pool.

- ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
- ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Shared Memory Model:

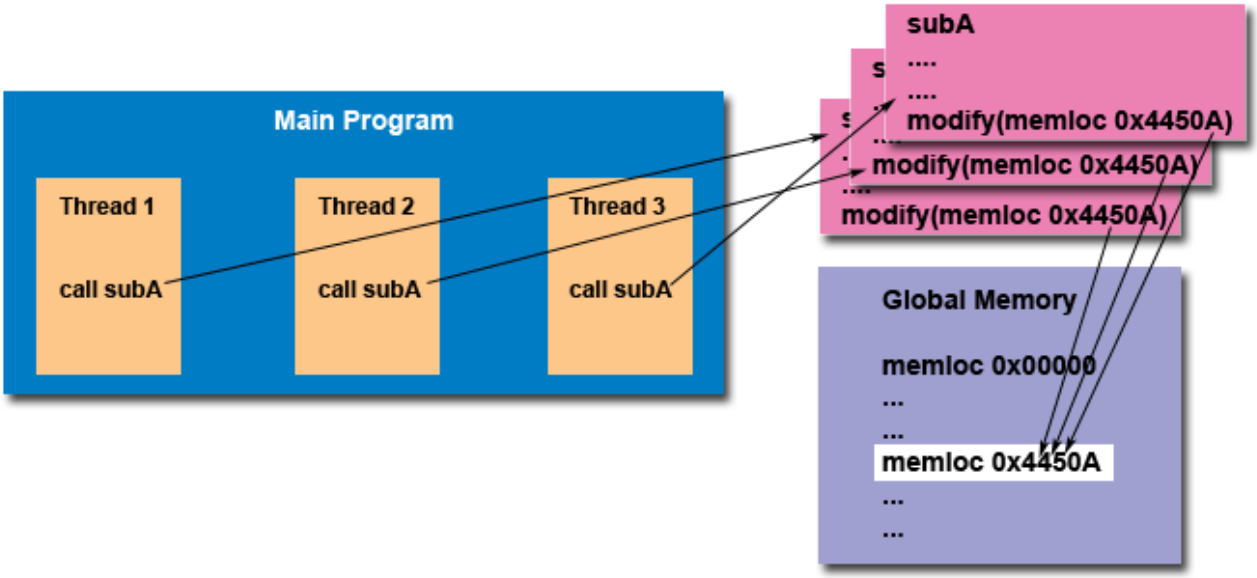
- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Thread-safeness:

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose that your application creates several threads, each of which makes a call to the same library routine:
    - This library routine accesses/modifies a global structure or location in memory.

- As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.



- The implication to users of external library routines is that if you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.
- Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc.

Thread Limits:

- Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.
- Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform.
- For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program.
- Several thread limits are discussed in more detail later in this tutorial.

The Pthreads API

The original Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. The POSIX standard has continued to evolve and undergo revisions, including the

Pthreads specification.

- Copies of the standard can be purchased from IEEE or downloaded for free from other sites online.
- The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
4. **Synchronization:** Routines that manage read/write locks and barriers.

- Naming conventions: All identifiers in the threads library begin with **pthread_**. Some examples are shown below.

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
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.

- The Pthreads API contains around 100 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning

Pthreads programmer.

- For portability, the `pthread.h` header file should be included in each source file using the Pthreads library.
- The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Some Fortran compilers may provide a Fortran pthreads API.

Thread Management

Creating and Terminating Threads

Routines:

```
pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)

pthread_cancel (thread)

pthread_attr_init (attr)

pthread_attr_destroy (attr)
```

Creating Threads:

- Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- `pthread_create` arguments:
  - `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
  - `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or 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.
- The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.
- Querying and setting your implementation's thread limit - Linux example shown. Demonstrates querying the default (soft) limits and then setting the maximum number of processes (including threads) to the hard limit. Then verifying that the limit has been overridden.

bash / ksh / sh	tcsh / csh

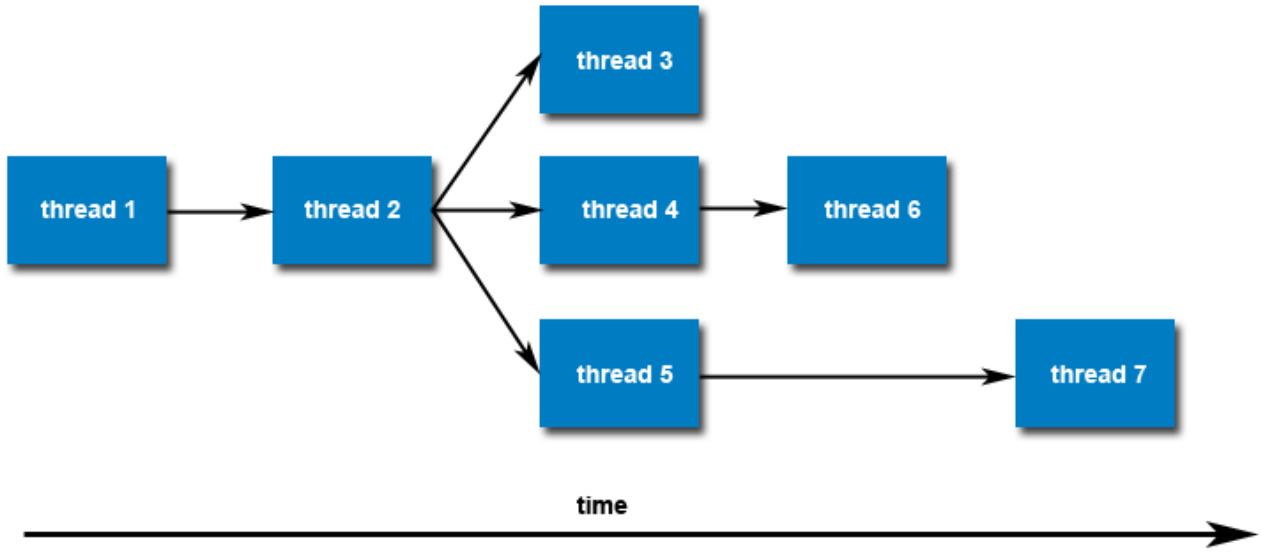
`$ ulimit -a`
`core file size (blocks, -c) 16`
`data seg size (kbytes, -d) unlimited`
`scheduling priority (-e) 0`
`file size (blocks, -f) unlimited`
`pending signals (-i) 255956`
`max locked memory (kbytes, -l) 64`
`max memory size (kbytes, -m) unlimited`
`open files (-n) 1024`
`pipe size (512 bytes, -p) 8`
`POSIX message queues (bytes, -q) 819200`
`real-time priority (-r) 0`
`stack size (kbytes, -s) unlimited`
`cpu time (seconds, -t) unlimited`
`max user processes (-u) 1024`
`virtual memory (kbytes, -v) unlimited`
`file locks (-x) unlimited`
 	
`$ ulimit -Hu`
`7168`
 	
`$ ulimit -u 7168`
 	
`$ ulimit -a`
`core file size (blocks, -c) 16`
`data seg size (kbytes, -d) unlimited`
`scheduling priority (-e) 0`
`file size (blocks, -f) unlimited`
`pending signals (-i) 255956`
`max locked memory (kbytes, -l) 64`
`max memory size (kbytes, -m) unlimited`
`open files (-n) 1024`
`pipe size (512 bytes, -p) 8`
`POSIX message queues (bytes, -q) 819200`
`real-time priority (-r) 0`
`stack size (kbytes, -s) unlimited`
`cpu time (seconds, -t) unlimited`
`max user processes (-u) 7168`
`virtual memory (kbytes, -v) unlimited`
`file locks (-x) unlimited`
	`% limit`
	
	`cputime unlimited`
	`filesize unlimited`
	`datasize unlimited`
	`stacksize unlimited`
	`coredumpsize 16 kbytes`
	`memoryuse unlimited`
	`vmemoryuse unlimited`
	`descriptors 1024`
	`memorylocked 64 kbytes`
	`maxproc 1024`
	
	`% limit maxproc`
	`unlimited`
	
	`% limit`
	`cputime unlimited`
	`filesize unlimited`
	`datasize unlimited`
	`stacksize unlimited`
	`coredumpsize 16 kbytes`
	`memoryuse unlimited`
	`vmemoryuse unlimited`
	`descriptors 1024`
	`memorylocked 64 kbytes`
	`maxproc 7168`

Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

Thread Attributes:

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.

- Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling parameters
  - Scheduling contention scope
  - Stack size
  - Stack address
  - Stack guard (overflow) size



Thread Binding and Scheduling:

Question: After a thread has been created, how do you know a)when it will be scheduled to run by the operating system, and b)which processor/core it will run on?

ANSWER Unless you are using the Pthreads scheduling mechanism, it is up to the implementation and/or operating system to decide where and when threads will execute. Robust programs should not depend upon threads executing in a specific order or on a specific processor/core.

- The Pthreads API provides several routines that may be used to specify how threads are scheduled for execution. For example, threads can be scheduled to run FIFO (first-in first-out), RR (round-robin) or OTHER (operating system determines). It also provides the ability to set a thread's scheduling priority value.
- These topics are not covered here, however a good overview of "how things work" under Linux can be found in the `sched_setscheduler` man page.

- The Pthreads API does not provide routines for binding threads to specific cpus/cores. However, local implementations may include this functionality - such as providing the non-standard `pthread_setaffinity_np` routine. Note that "_np" in the name stands for "non-portable".
- Also, the local operating system may provide a way to do this. For example, Linux provides the `sched_setaffinity` routine.

Terminating Threads & pthread_exit():

- There are several ways in which a thread may be terminated:
    - The thread returns normally from its starting routine. Its work is done.
    - The thread makes a call to the `pthread_exit` subroutine - whether its work is done or not.
    - The thread is canceled by another thread via the `pthread_cancel` routine.
    - The entire process is terminated due to making a call to either the `exec()` or `exit()`
    - If main() finishes first, without calling `pthread_exit` explicitly itself
- The `pthread_exit()` routine allows the programmer to specify an optional termination *status* parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).
- In subroutines that execute to completion normally, you can often dispense with calling `pthread_exit()` - unless, of course, you want to pass the optional status code back.
- Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- **Discussion on calling `pthread_exit()` from main():**
    - There is a definite problem if main() finishes before the threads it spawned if you don't call `pthread_exit()` explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.
    - By having main() explicitly call `pthread_exit()` as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

Thread Management

Passing Arguments to Threads

- The `pthread_create()` routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread_create()` routine.

- All arguments must be passed by reference and cast to (void *).

Question: How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?

ANSWER: Make sure that all passed data is thread safe - that it can not be changed by other threads.

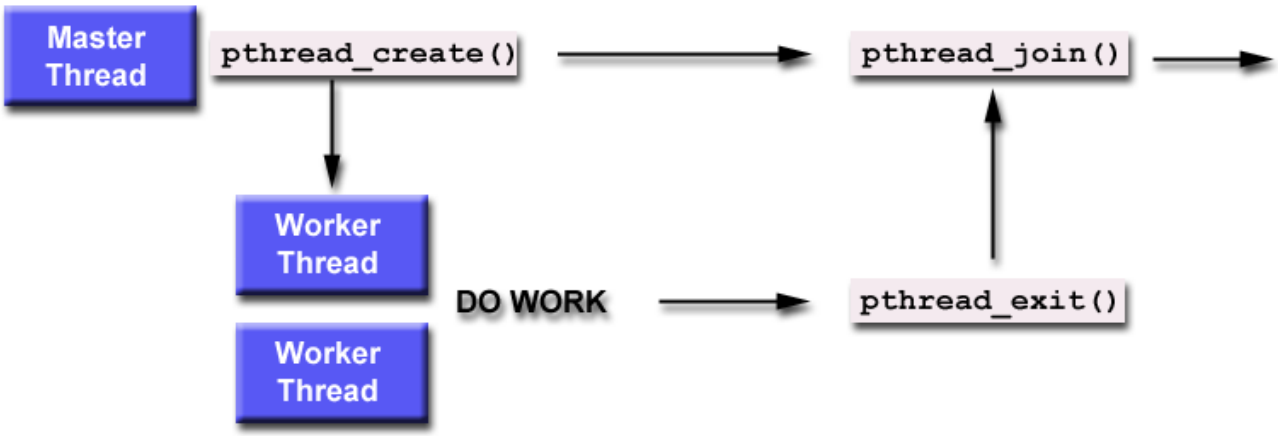
Joining and Detaching Threads

Routines:

```
pthread_join (threadid,status)

pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)
```

Joining:

- "Joining" is one way to accomplish synchronization between threads. For example:



The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.

- The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`.
- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable.
- To explicitly create a thread as joinable or detached, the `attr` argument in the `pthread_create()` routine is used. The typical 4 step process is:
    1. Declare a pthread attribute variable of the `pthread_attr_t` data type
    2. Initialize the attribute variable with `pthread_attr_init()`
    3. Set the attribute detached status with `pthread_attr_setdetachstate()`
    4. When done, free library resources used by the attribute with `pthread_attr_destroy()`

Detaching:

- The `pthread_detach()` routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.

Recommendations:

- If a thread requires joining, consider explicitly creating it as joinable. This provides portability as not all implementations may create threads as joinable by default.
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state. Some system resources may be able to be freed.