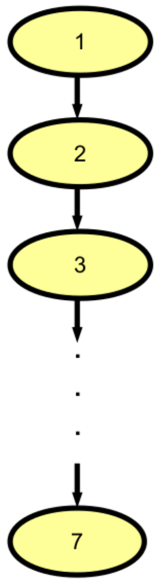
Linux Threads

Sequential Programming



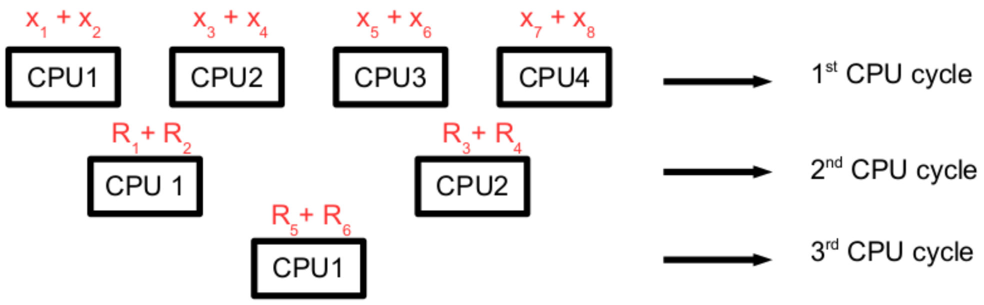
Suppose we want to add eight numbers $x_1, x_2, x_3, \ldots\ x_8$

There are seven addition operations and if each operation take 1 CPU cycle, the entire operation will take seven cycles

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$

Concurrent/Parallel Programming

Suppose we have 4xCPUs or a 4xCore CPU, the seven addition operations can now be completed in just three CPU cycles, by dividing the task among different CPUs



Ways to Achieve Concurrency

Multiple single threaded processes

- Use `fork()` to create a new process for handling every new task, the child process serves the client process, while the parent listens to the new request
- Possible only if each slave can operate in isolation
- Need IPC between processes
- Lot of memory and time required for process creation

Multiple threads within a single process

- Create multiple threads within a single process
- Good if each slave need to share data
- Cost of creating threads is low, and no IPC required

Single process multiple events

- Use non-blocking or asynchronous I/O, using `select()` and `poll()` system calls

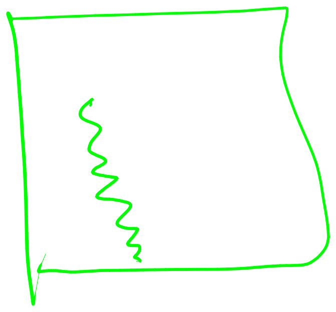
Processes and Threads

Every process has two characteristics:

- **Resource ownership**- process includes a virtual address space to hold the process image
- **Scheduling**- follows an execution path that may be interleaved with other processes

These two characteristics are treated independently by the operating system. The unit of resource ownership is referred to as a process, while the unit of dispatching is referred to as a thread

A thread is an execution context that is independently scheduled, but shares a single addresses space with other threads of the same process



Processes and Threads (cont...)

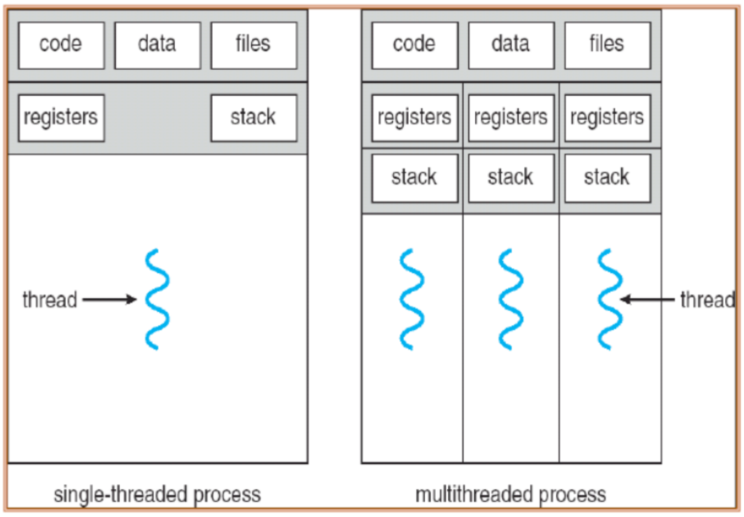
Similarities between Processes & Threads:

- Like a process, a thread can also be in one of many states (new, ready, running, block, terminated)
- Only one thread can be in running state (single CPU)
- Like a process a thread can create a child thread

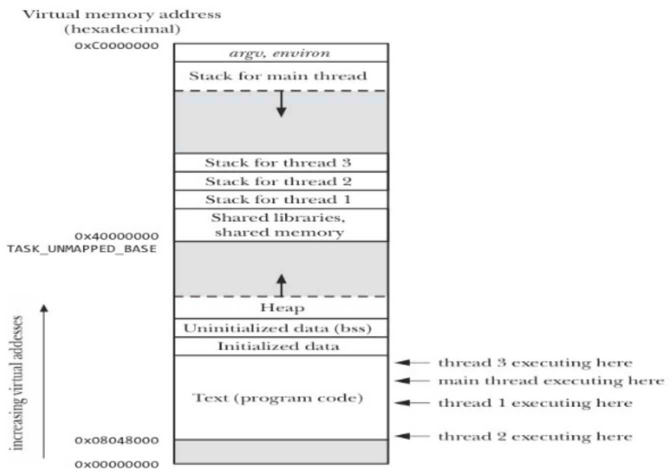
Differences between Processes & Threads:

- No automatic protection in threads
- Every process has its own address space, while all other threads within a process executes within the same address space

Single vs Multi-threaded Process



Pictorial View of a Multi-threaded Process



Temporal Multi-threading: Only one thread of instruction can execute in any given pipeline stage at a time
Simultaneous Multi-threading (SMT/HT): More than one thread of instruction can execute in any giver pipeline stage at a time. (SMT/HT is a multi-threading on a super scalar architecture)

Multi-Threaded Process

Threads within a process share :

- PID, PPID, PGID, SID, UID, GID
- Code and Data Section
- Global Variables
- Open files via PPFDT
- Signal Handlers
- Interval Timers
- CPU time consumed
- Resources Consumed
- Nice value
- Record locks (created using fcntl())

Threads have their own:

- Thread ID
- CPU Context (PC, and other registers)
- Stack
- State
- The `errno` variable
- Priority
- CPU affinity
- Signal mask

Thread Implementation Models (M:1)

In Many-to-one (M:1) threading implementation, all of the details of thread creation, termination, scheduling, synchronization, and so on are handled entirely within the user-space. Kernel knows nothing about the existence of multiple threads within the process

Advantages:

- Thread operations are fast as no mode switch is required
- User level threads can be used even if the underlying platform does not support multithreading

Disadvantages:

- When a user-level thread makes a blocking system call, e.g., `read()`, the entire process is blocked
- Since the kernel is unaware of the existence of multiple threads within the process, it CANNOT schedule separate threads to different CPUs on multiprocessor hardware

Thread Implementation Models (1:1)

In one-to-one (1:1) threading implementation, each thread maps onto a separate kernel scheduling entity (KSE). All of the details of thread creation, termination, scheduling, synchronization and so on are handled by system calls inside the kernel

Advantages:

- When a kernel-level thread makes a blocking system call, e.g., `read()`, only that thread is blocked
- Since the kernel is aware of the existence of multiple threads within the process, it can schedule separate threads to different CPUs on multiprocessor hardware

Disadvantages:

- Thread operations are slow as a switch into kernel mode is required
- Overhead of maintaining a separate KSE for each of the threads in an application place a significant load on the kernel scheduler, degrading overall system performance

Thread Implementation Models (M:N)

The many-to-many (M:N) threading implementation, aim to combine the advantages of the 1:1 and M:1 models, while eliminating their disadvantages. Each process can have multiple associated KSEs, and several threads may map to each KSE

Disadvantages:

- The major disadvantage of M:N model is its complexity. The task of thread scheduling is shared between the kernel and the user-space threading library, which must cooperate and communicate information with one another

The M:N model was initially considered for the NPTL threading implementation, but rejected as it required much changes to the Kernel. The Linux threading implementations **LinuxThreads** and **NPTL** employ the 1:1 model

LinuxThreads

LinuxThreads is the original Linux threading implementation, developed by Xavier Leroy. In addition to the threads created by the application, LinuxThreads creates an additional "manager" thread that handles thread creation and termination. Threads are created using a **`clone()`**, with the flags mentioned below: (threads share virtual memory, file descriptors, file system-related information (umask, root directory, pwd,...) and signal disposition)

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

Deviations from specified behavior

- **`getpid()`** returns a different value in each of the threads of a process
- **`getppid()`** returns the PID of the manager thread
- If one thread creates a child using **`fork()`**, then only the thread that created the child process can **`wait()`** for it
- If a thread calls **`exec()`**, then SUSv3 requires that all other threads are terminated. While this is not so in LinuxThreads
- Threads don't share PGIDs, and SIDs
- Threads don't share resource limits
- Some versions of `ps`(1) show all of the threads in a process (including the manager thread) as separate items with distinct PIDs
- CPU time returned by **`times()`** and resource usage information returned by `getrusage()` are per thread
- Threads don't share nice value set by **`setpriority()`**

NPTL Threads

The Native POSIX Threads Library (NPTL) is is the modern Linux Threading implementation, developed by Drepper and Ingo Molnar, designed to address most of the shortcomings of LinuxThreads. It adheres more closely to SUSv3 specification. Applications that employ large number of threads scale much better under NPTL than under LinuxThreads. NPTL threads does not require an additional manager thread. Supported by Linux 2.6 onwards. Threads are created using **`clone()`**, that specifies all the flags of LinuxThreads and more:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND |
CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID |
        CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
```

To discover thread implementation on your system give following command:

```
$ getconf GNU_LIBPTHREAD_VERSION
 $ getconf GNU_LIBC_VERSION
```

On systems that provides both NPTL and LinuxThreads, if you want to run a multithreaded program with LinuxThreads, you set the following environment variable to a kernel version that doesn't provided support for NPTL (e.g., 2.2.5)

```
$ export LD_ASSUME_KERNEL=2.25
```

Pthreads API

The pthread API defines a number of data types and should be used to ensure the portability of programs and mostly defined in `/usr/include/x86_64-linux-gnu/bits/pthreadtypes.h`. Remember you should not use the C == operator to compare variables of these types

Data Type	Description
`pthread_t`	Used to identify a thread
`pthread_attr_t`	Used to identify a thread attributes object
`pthread_mutex_t`	Used for mutex
`pthread_mutexattr_t`	Used to identify mutex attributes object
`pthread_cond_t`	Used for condition variable
`pthread_cond_attr_t`	Used to identify condition variable attributes object
`pthread_key_t`	Key for thread specific data
`pthread_once_t`	One-time initialization control context
`pthread_spinlock_t`	Used to identify spinlock
`pthread_rwlock_t`	Used for read-write lock
`pthread_rwlockattr_t`	Used for read-write lock attributes
`pthread_barrier_t`	Used to identify a barrier
`pthread_barrierattr_t`	Used to identify a barrier attributes object

Pthreads API (cont...)

```
int pthread_create(pthread_t *tid, const pthread_attr_t
              *attr, void *(*start)(void *), void *arg) ;
```

- This function starts a new thread in the calling process. The new thread starts its execution by invoking the start function which is the 3rd argument to above function
- On success, the TID of the new thread is returned through 1st argument to above function
- The 2nd argument specifies the attributes of the newly created thread. Normally we pass `NULL` pointer for default attributes.
- The 4th argument is a pointer of type void which points to the value to be passed to thread start function. It can be `NULL` if you do not want to pass any thing to the thread function. It can also be address of a structure if you want to pass multiple arguments

Pthreads API (cont...)

```
void pthread_exit(void *status);
```

- This function terminate the calling thread
- The `status` value is returned to some other thread in the calling process, which is blocked on the `pthread_join()` call
- The pointer `status` must not point to an object that is local to the calling thread, since that object disappears when the thread terminates

Ways for a thread to terminate:

- The thread function calls the `return` statement
- The thread function calls `pthread_exit()`
- The main thread returns or call `exit()`
- Any sibling thread calls `exit()`

Pthreads API (cont...)

```
int pthread_join(pthread_t tid, void **retval);
```

- Any peer thread can wait for another thread to terminate by calling `pthread_join()` function, similar to `waitpid()`. Failing to do so will produce the thread equivalent of a zombie process
- The 1st argument is the ID of thread for which the calling thread wish to wait. Unfortunately, we have no way to wait for any of our threads like `wait()`
- The 2nd argument can be `NULL`, if some peer thread is not interested in the return value of the new thread. Otherwise, it can be a double pointer which will point to the status argument of the `pthread_exit()`

Returning value from a Thread Function

- A thread function can return a pointer to its parent/calling thread, and that can be received in the 2nd argument of the `pthread_join()` function
- The pointer returned by the `pthread_exit()` must not point to an object that is local to the thread, since that variable is created in the local stack of the terminating thread function
- Making the local variable `static` will also fail. Suppose two threads run the same `thread_function()`, the second thread may over write the static variable with its own return value and return value written by the first thread will be over written
- So the best solution is to create the variable to be returned in the heap instead of stack

Creating Arrays of Threads

- You may need to create large number of threads for dividing the computational tasks as per your program logic
- At compile time, if you know the number of threads you need, you can simply create an array of type `pthread_t` to store the thread IDs
- If you do not know at compile time, the number of threads you need, you may have to to allocate memory on heap for storing the thread IDs
- The maximum number of threads that a system allow can be seen in `/proc/sys/kernel/threads-max` file. There are however, other parameters that limit this count like the size of stack the system needs to give to every new thread

Thread Attributes

Every thread has a set of attributes which can be set before creating it. If we pass a `NULL` as second argument to `pthread_create()` function, the default thread attributes are used. The default value of thread attributes are shown in table below:

Attribute	Default Value	Description
`detachstate`	PTHREAD_CREATE_JOINABLE	Joinable by other threads
`stackaddr`	NULL	Stack allocated by system
`stacksize`	NULL	2 MB
`priority`	---	Priority of calling thread is used
`policy`	SCHED_OTHER	Determined by system
`inheritsched`	PTHREAD_INHERIT_SCHED	Inherit scheduling attributes from creating thread

Detach State (Avoiding Zombie Threads)

Joinable Thread:

A joinable thread (like a process) is not automatically cleaned up by GNU/LINUX when it terminates. The thread's exit status hangs around in system until another thread calls `pthread_join()` to obtain its return value. Only then its resources are released. For example whenever we want to return data from child thread to parent thread the child thread must be a joinable thread

Detached Thread:

A detachable thread is cleaned up automatically when it terminates. Since a detached thread is immediately cleaned up, another thread may not wait for its completion by using `pthread_join()` to obtain its return value. For example suppose the main thread crates a child thread to do back up of a file and the main thread continue its execution. When the backup is finished , the second thread can just terminate. There is no need for it to rejoin the main thread. A thread can detach itself using `pthread_detach(pthread_self())` call

Steps to Specify Customized Thread Attributes

- Create a `pthread_attr_t` object
- Call `pthread_attr_init()`, passing it a pointer of above object
- Modify the attribute object to contain the desired attribute value using the appropriate setters
- Pass a pointer to the attribute object when calling `pthread_create()`
- Destroy pthread attribute object by calling `pthread_attr_destroy()`

Pthreads API (cont...)

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- The `pthread_attr_init()` function initializes the thread attributes object pointed to by `attr` with default attribute values. After this call, individual attributes of the object can be set using various related functions (next slide), and then the object can be used in one or more `pthread_create()` calls
- When a thread attributes object is no longer required, it should be destroyed using the `pthread_attr_destroy()` function. Destroying a thread attributes object has no effect on threads that were created using that object