CS410: Parallel Computing

Spring 2024

Nikhil Hegde
Milind Chabbi

Shared-Memory Programming (using POSIX Threads)

Recall: Why Threads?

- Portable, widely-available programming model
  —use on both serial and parallel systems
- Useful for hiding latency
  —e.g. latency due to I/O, communication
- Useful for scheduling and load balancing
  —especially for dynamic concurrency
- Relatively easy to program
  —significantly easier than message-passing!

Recall:

Shared Address Space Programming Models

A brief taxonomy

- Lightweight processes and threads
  - —all memory is global
  - —examples: Pthreads, Cilk (lazy, lightweight threads)
- Process-based models
  - —each process's data is private, unless otherwise specified
  - —example: Linux shget/shmat/shmdt
- Directive-based models, e.g., OpenMP
  - —shared and private data
  - —facilitate thread creation and synchronization
- Global address space programming languages
  - —shared and private data
  - —hardware typically distributed memory, perhaps not shared
  - —examples: Unified Parallel C, Co-array Fortran

Programming Parallel Computers - Approaches

1. Extend existing languages:
   - Add parallel constructs (cilk_spawn, omp_for etc.)
     (requires development of compiler system support)
   - Add parallel operations (e.g. `fork, pthread_create,` etc.)
2. Define totally new parallel language and compiler system
3. Extend existing compilers so that they translate sequential programs into parallel programs
4. Add a parallel language layer on top of the sequential language

Extend Language

- Add operations to a sequential language
  - Create and terminate processes/threads
  - Synchronize processes/threads
  - Allow processes/threads to communicate

  Advantages:
  - Easiest, quickest, and least expensive
  - Allows existing compiler technology to be leveraged
  - New libraries can be ready soon after new parallel computers are available

  Disadvantages
  - Lack of compiler support to catch errors
  - Easy to write programs that are difficult to debug

Why Pthreads?

- Lightweight

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

[Source: Why Pthreads? | LLNL HPC Tutorials]

Numbers are in seconds and are obtained with 50,000 process/thread creation

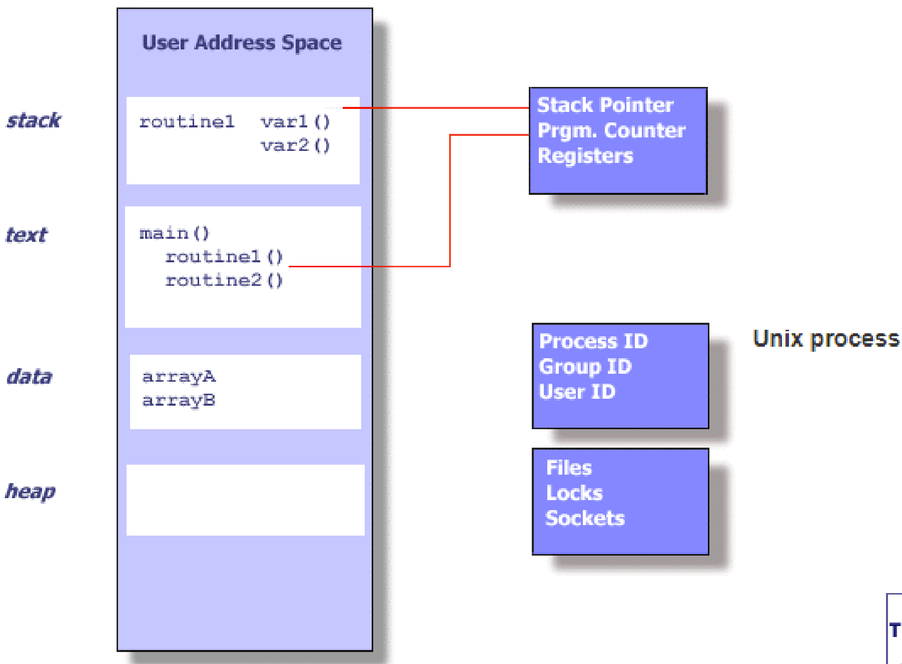
Recap: Threads and Processes

- Abstraction provided by the OS

Process	Thread
Self-contained i.e. has its own private resources to execute/run programs. E.g. of a resource: memory.	Belongs to a process. Share memory and other resources among threads of the same process.
Is an instance of a running program.	Can be considered as a subroutine in the 'main' program
Have an illusion that *entire computer* is for itself.	Have an illusion that *entire processor* is for itself.

Recap: Threads and Processes

[Source: what is a Thread? | LLNL HPC Tutorials]

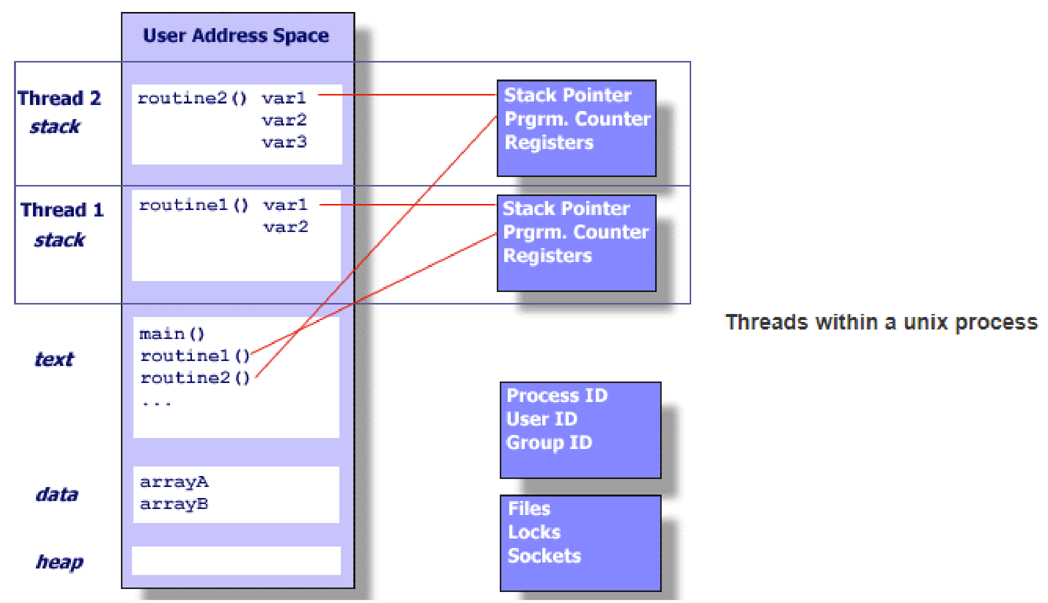


Each process maintains its own:

- Process ID, Group ID, User ID,
- Environment,
- Working directory,
- Registers, stack, heap, program instructions
- File handles, Shared libraries, ..

Each thread in a process maintains its own copy of:

- Stack pointer,
- Program counter,
- Registers,
- Thread-specific data,
- Scheduling policy



POSIX Threads

- Hardware vendors implemented proprietary versions of threads earlier
  - Software portability was a concern

- IEEE POSIX 1003.1c standard
  - Standardized C language threads programming interface for UNIX like systems - **Pthreads API**

  - Implemented as a set of C language programming types and procedure calls
    - Include a header file `pthread.h` and use a library (linking may be implicit in some cases)
    - Fortran Programmers can use wrappers around C procedure calls. Some Fortran compilers provide Fortran Pthreads API.

POSIX Threads

- Pthreads implementation now supported by most vendors
  - In addition to proprietary implementation
- Concepts are broadly applicable to other implementations (independent of Pthreads API)
  - NT Threads
  - Solaris Threads
  - Java Threads
  - C++ Threads (C++11 onwards)

Pthreads API

- Subroutines grouped into four major categories:
  1. **Thread Management –**
     Routines that deal with thread creation, join, detach, etc. Also, query thread attributes
  2. **Mutexes –**
     Routines that deal with synchronization. Creating, destroying, locking, and unlocking a mutex (abbr. for “mutual exclusion”). Also, set mutex attributes
  3. **Condition Variables** –
     Routines addressing communication between threads that share a mutex. Based upon programmer specified conditions. Includes functions to create, destroy, wait and signal based upon specified variable values. Also, set/query condition variable attributes.
  4. **Synchronization**
     Routines that manage read-write locks and barriers

Contains around 100 routines

Pthreads API

- Subroutines grouped into four major categories:
  1. **Thread Management –**

     Routines that deal with thread creation, join, detach, etc. Also, query thread attributes
  2. **Mutexes –**

     Routines that deal with synchronization. Creating, destroying, locking, and unlocking a mutex (abbr. for “mutual exclusion”). Also, set mutex attributes
  3. **Condition Variables** –

     Routines addressing communication between threads that share a mutex. Based upon programmer specified conditions. Includes functions to create, destroy, wait and signal based upon specified variable values. Also, set/query condition variable attributes.
  4. **Synchronization**

     Routines that manage read-write locks and barriers

Creating Pthreads

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

unique thread ID Returned upon successful creation.

Thread attributes specified with object of type `pthread_attr_t`. Can leave it as NULL to set default attribute values.

The routine that the thread executes once created.

Single argument passed to `start_routine` (type-casted to void *). Can leave it as NULL to omit arguments.

Creates a new thread and makes it ready-to-execute

- Usage: Typically, `main` function first creates threads, which in turn may create other threads. No hierarchy exists among threads; All threads created are peers.
- Return value: an integer indicating status of creation

Pthread ID `pthread_t`

```
int pthread_create(pthread_t* threadID,..
```

- `pthread_t` object is initialized by the `pthread_create` API.

Pthreads Attributes `pthread_attr_t`

```
int pthread_create(pthread_t* threadID,
const pthread_attr_t* attribute,..
```

- `pthread_attr_init` and `pthread_attr_destroy` `APIs` create and destroy thread attribute object.

  The attribute object contains options to specify:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling priority
  - Scheduling contention scope
  - Stack size
  - Stack address

Start Routine

```
int pthread_create(pthread_t* threadID,
const pthread_attr_t* attribute,
void * (*start_routine)(void*),..
```

- Called upon thread execution

Arguments to Start Routine

```
int pthread_create(pthread_t* threadID,
const pthread_attr_t* attribute,
void * (*start_routine)(void*),
void * arg)
```

- Single argument typecasted to void *
- How do you pass multiple arguments?
  - Pack the arguments in a structure object and pass the pointer to the structure object (cast as void *)
- Since thread start time is non-deterministic, do not pass as arguments data that is modified by other threads

Terminating Pthreads

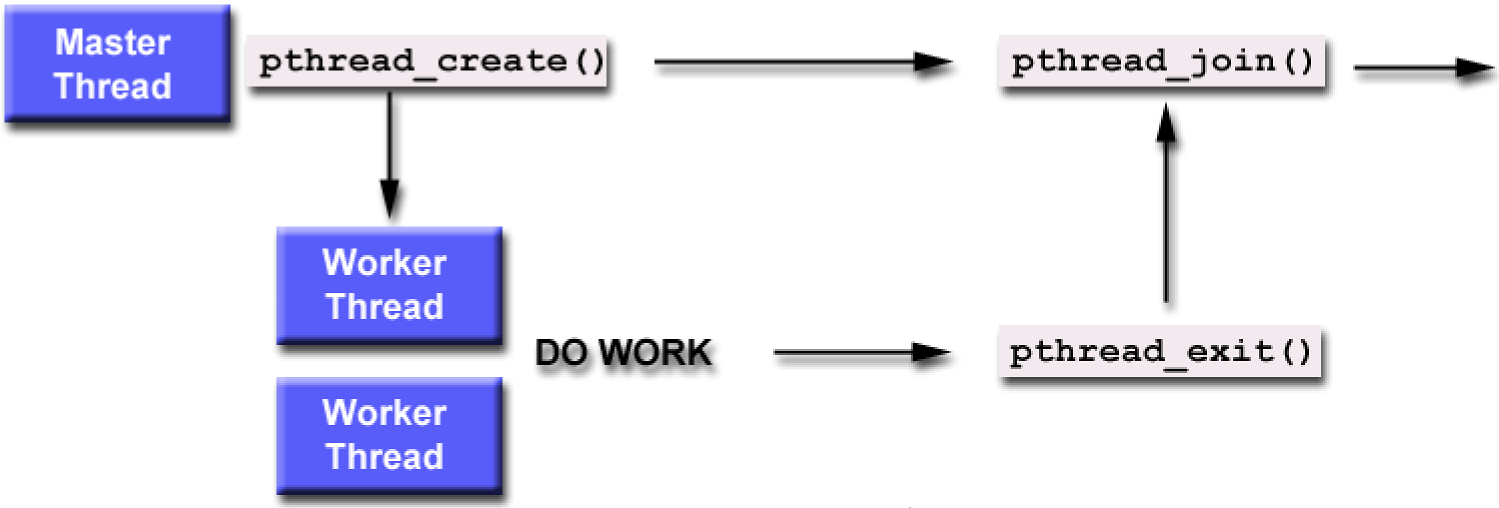
- `pthread_exit(status)`
- `pthread_cancel(thread)` routine can be used by a thread to cancel another thread
- `exit(int)` terminates the entire process (including all threads)

pthread_exit(status)

- `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist
- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute.
  - Otherwise, they will be automatically terminated when `main()` finishes
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread
- Cleanup
  - `pthread_exit()` routine does not close files
  - Recommended to use `pthread_exit()` to exit from all threads...especially `main()`.

Joining and Detaching from Pthreads

- `pthread_join(threadid, status)` is one way to accomplish synchronization between threads
  - Blocks the calling thread until the specified thread terminates
  - Only threads created as joinable can be joined (other option is to create a thread as *detached*, where the thread can never be joined.)
    - Can Use `pthread_detach()` to detach a thread after it was created as joinable



[Source: Joining and Detaching Threads | LLNL HPC Tutorials]