Lecture 10

Shared Memory Multiprocessors

Institute of Computer Science & Information Technology,
Faculty of Management & Computer Sciences,
The University of Agriculture, Peshawar, Pakistan.

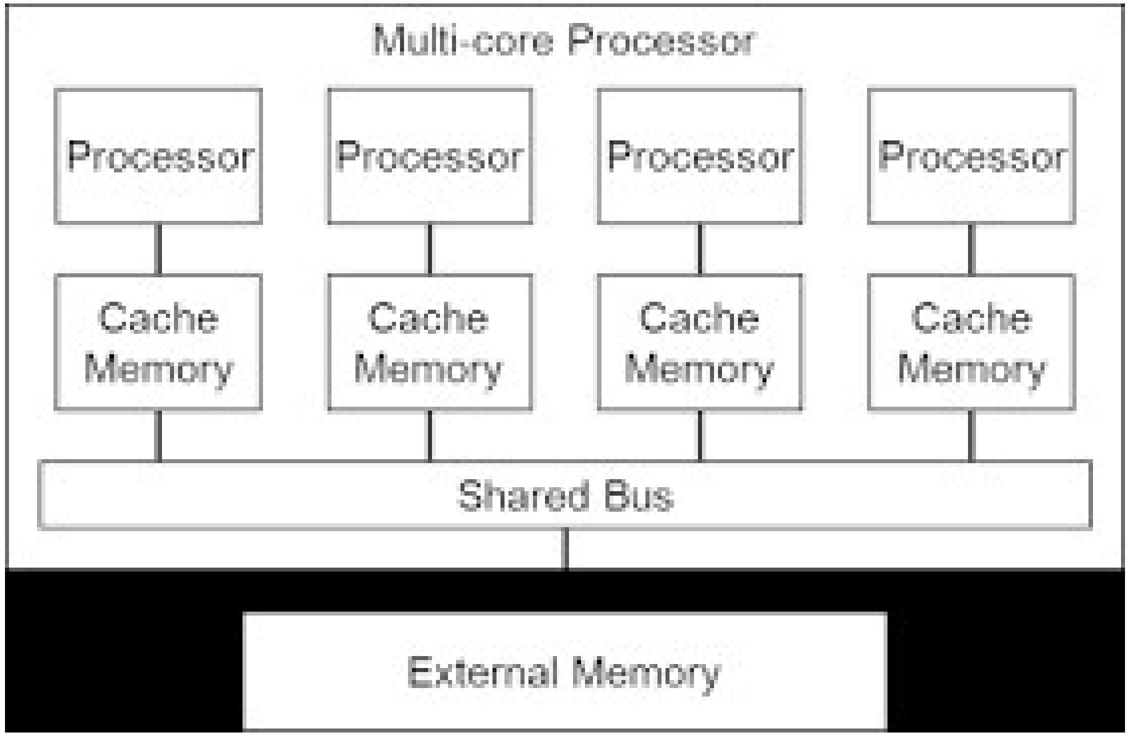
Shared Memory Multiprocessors

- Any memory location can be accessible by any of the processors.

- A single address space exists, meaning that each memory location is given a unique address within a single range of address.

- For small number of processors, common architecture is the single bus architecture:

Shared Memory Multiprocessors (1)



- This architecture is only suitable for, perhaps, up to eight processors, because the bus can only be used by one processor at a time.

Programming Alternatives

1. Using a supportive programming language.
2. Using library routines with an existing sequential language.
3. Using a sequential programming language and ask a parallelizing compiler to convert it into parallel executable code.
4. UNIX processes.
5. P-Threads (POSIX thread)
6. Using an existing sequential programming language supplemented with compiler directives for specifying parallelism., e.g., OpenMP.

Constructs for Parallelism

- Creating Concurrent Processes
  - FORK-JOIN was described by Conway in 1963, and was known before 1960.
  - In the original construct a FORK statement generates one new path for a concurrent process and the concurrent processes use the JOIN statement at their ends.

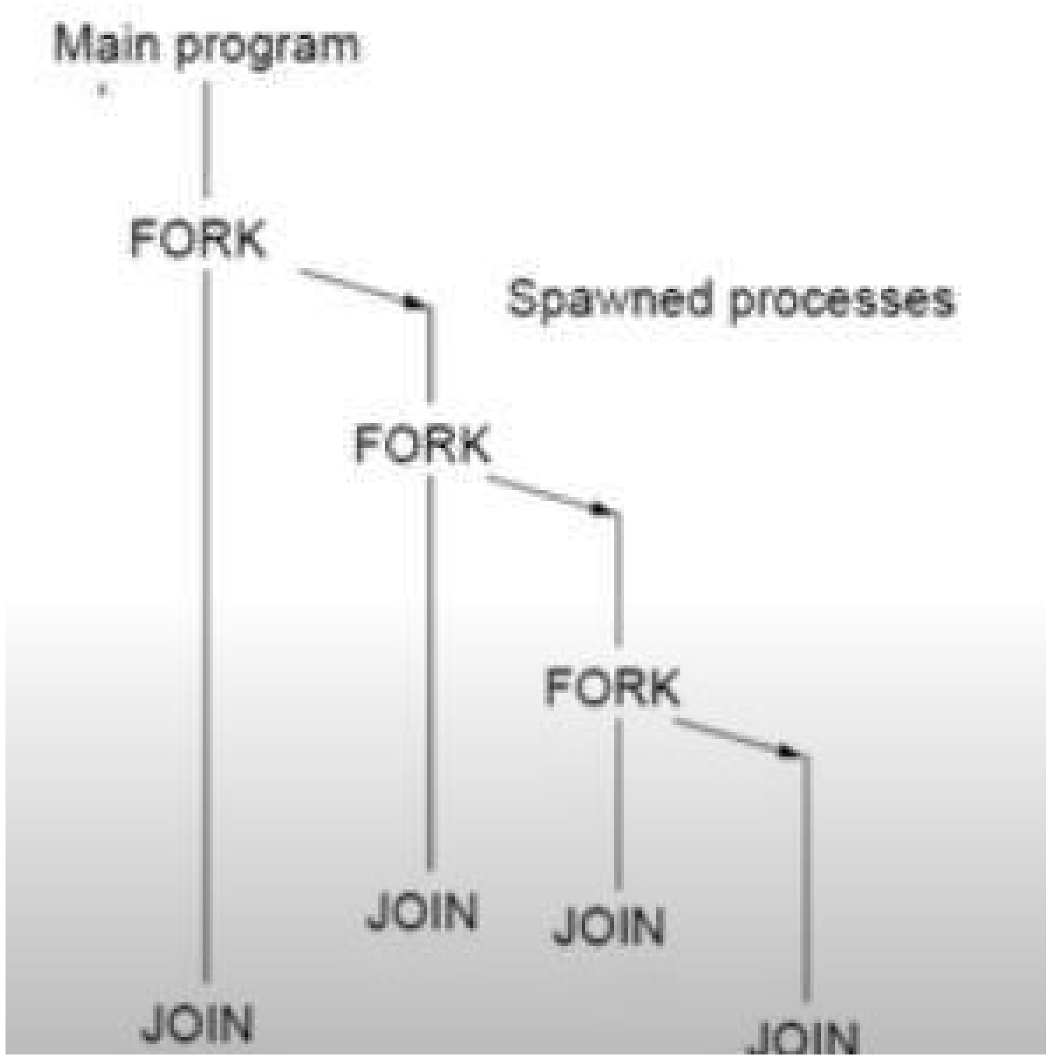
UNIX Heavyweight Processes

- The UNIX system call fork() creates a new process.
- The new process(child process) is an exact copy of the calling process except that it has a unique process ID.
- It has its own copy of the parent's variables.
- They are assigned the same values as the original variables initially.

- The forked process starts execution at the point of the fork.

- On success, fort() returns 0 to the child process and returns the process ID of the child process to the parent process.

- Processes are ‘joined’ with the system calls wait() and exit():

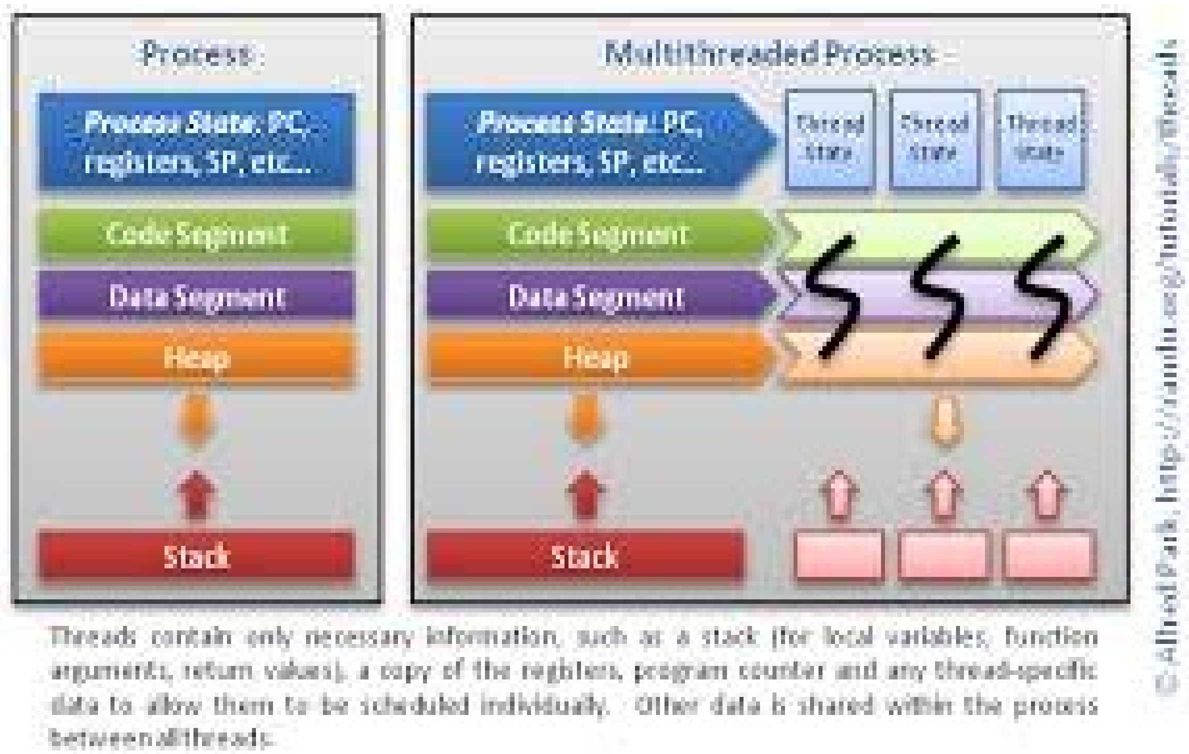

Main program
FORK
Spawned processes
FORK
FORK
JOIN
JOIN
JOIN
JOIN

OS Review: Processes

- 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).
- When we run a program, a process is created
  - E.g. ./a.out, ./axpy, etc
  - fork () system call

Threads

- Threads use, and exist within, the process resources.
- Scheduled and run as independent entities.
- Duplicate only the bare essential resources that enable them to exist as executable code.



What is a Thread in Real

- OS view
  - – An independent stream of instructions that can be scheduled to run by the OS.
- Software developer view
  - – A "procedure" that runs independently from the main program
    - Imagine multiple such procedures of main run simultaneously and/or independently
  - – Sequential program: a single stream of instructions in a program.
  - – Multi-threaded program: a program with multiple streams
    - Multiple threads are needed to use multiple cores/CPUs

POSIX threads (PThreads)

- Threads used to implement parallelism in shared memory multiprocessor systems, such as SMPs

- Historically, hardware vendors have implemented their own proprietary versions of threads
  – Portability a concern for software developers.

- For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.
  – Implementations that adhere to this standard are referred to as POSIX threads

The POSIX Thread API

- Commonly referred to as PThreads, POSIX has emerged as the standard threads API, supported by most vendors.
  - Implemented with a pthread.h header/include file and a thread library

- Functionalities
  - Thread management, e.g. creation and joining
  - Thread synchronization primitives
    - Mutex
    - Condition variables
    - •Reader/writer locks
  - Thread-specific data

PThread API

- #include <pthread.h>

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

- gcc -lpthread

Thread Creation

- Initially, main() program comprises a single, default thread
  – All other threads must be explicitly created

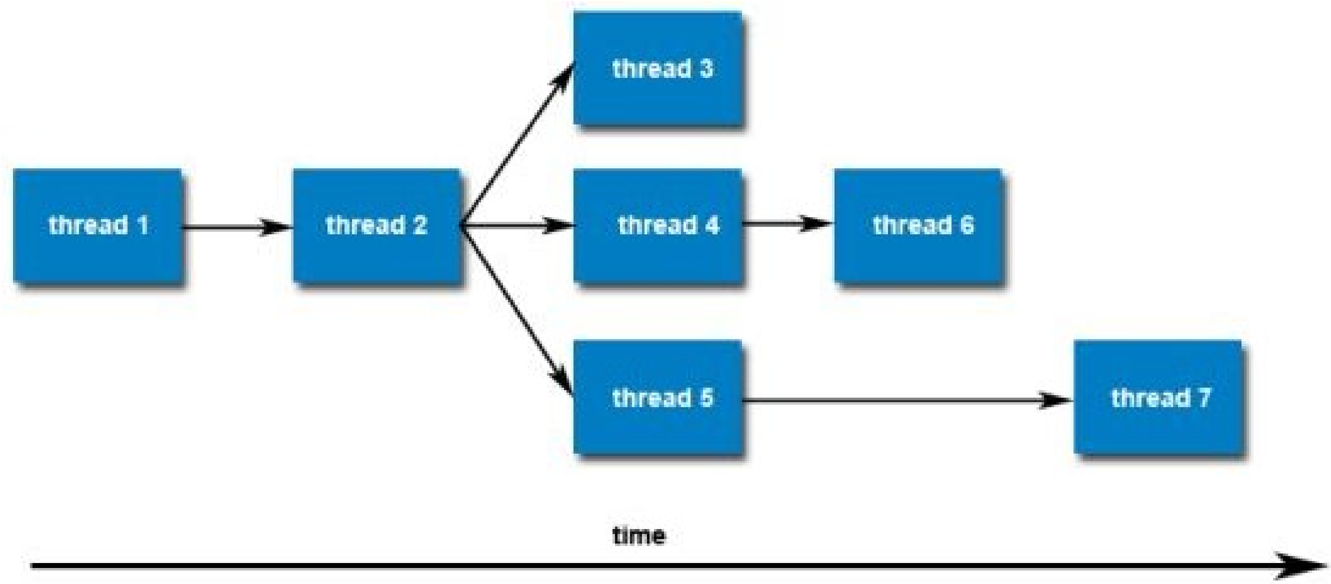
```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void * arg);
```

- 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.

Opaque object: A letter is an opaque object to the mailman, and sender and receiver know the information.

Thread Creation

- • pthread_create creates a new thread and makes it executable, i.e. run immediately in theory
  - – can be called any number of times from anywhere within your code
- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency between threads



Terminating Threads

- pthread_exit is used to explicitly exit a thread
  - Called after a thread has completed its work and is no longer required to exist
- If main()finishes before the threads it has created
  - If 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: the pthread_exit()routine does not close files
  - Any files opened inside the thread will remain open after the thread is terminated

Thread Attribute

```
int pthread_create(
    pthread_t *thread,
     const pthread_attr_t *attr,
    void *(*start_routine)(void *),
     void * arg);
```

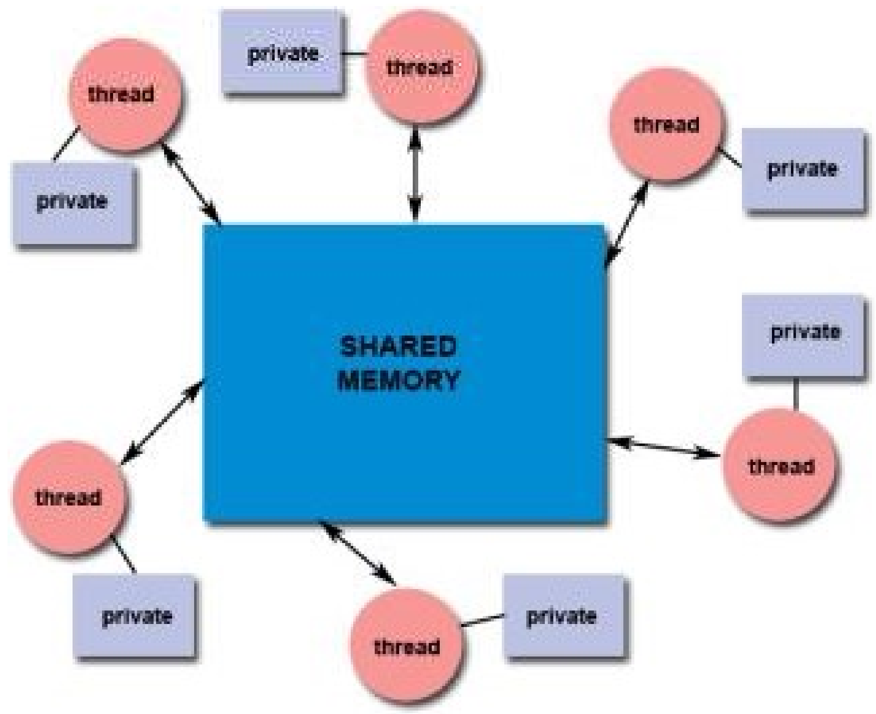
- Attribute contains details about
  - whether scheduling policy is inherited or explicit
  - scheduling policy, scheduling priority
  - stack size, stack guard region size
- pthread_attr_init and pthread_attr_destroy are used to initialize/destroy the thread attribute object
- Other routines are then used to query/set specific attributes in the thread attribute object

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:
  - Create a structure which contains all of the arguments
  - Then pass a pointer to the object of that structure in the pthread_create()routine.
  - All arguments must be passed by reference and cast to (void *)
- Make sure that all passed data is thread safe: data racing
  - it can not be changed by other threads
  - It can be changed in a determinant way
    - Thread coordination

Shared Memory and Threads

- 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 Consequences

- • Shared State!
  - Accidental changes to global variables can be fatal.
  - 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
  - Therefore requires explicit synchronization by the programmer
- Many library functions are not thread-safe
  - Library Functions that return pointers to static internal memory. E.g. gethostbyname()
- Lack of robustness
  - Crash in one thread will crash the entire process

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
- Example: an 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 threadsafe.

Why PThreads (not processes)?

- The primary motivation
  - To realize potential program performance gains
- Compared to the cost of creating and managing a process
  - A thread can be created with much less OS overhead
- Managing threads requires fewer system resources than managing processes
- All threads within a process share the same address space
- Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication