Announcement

• Due date of HW#2 is extended to next Monday

• Office hours for HW#2 on M W F (check Piazza)

• Please check chapters on processes and interprocess communication (IPC) in Linux on Piazza + Makefile for HW # 2

• Exercise # 3 has been posted
Learning Objectives

• What are threads? Explain the purpose of threads

• Why threads? Identify difference between threads and processes

• Identify advantages and disadvantages of threads

• Identify differences between kernel level threads and user level threads

• Programming threads in Linux- API
Outline of Chapter 4

• **Process characteristics**
  – Need for threads
  – Similarity and difference between threads and processes

• Advantages and complications with threads

• User level threads

• Kernel level threads
Process and Threads

Processes have two characteristics that can be treated independently by the OS:

i) Resource Ownership

Process includes:
- a virtual address space to hold the process image (data + PCB)
- control of some resources (files, I/O devices…)

ii) Scheduling/Execution

• Unit of dispatching - follows an execution path through one or more programs
- execution may be interleaved with other process
- the process has an execution state and a dispatching priority
Processes vs Threads

i) Resource Ownership

- The unit of **dispatching** (scheduling/execution) is referred to as a **thread** or **lightweight process**

ii) Scheduling/Execution

- The unit of **resource ownership** is referred to as a **process** or **task**

- **Multithreading** - The ability of an OS to support multiple, **concurrent** paths of execution within a single process
Processes and Threads: Resource Ownership vs Execution

- **Single-threaded process**
  - Code
  - Data
  - Files
  - Registers
  - Stack

- **Multithreaded process**
  - Code
  - Data
  - Files
  - Registers
  - Stack

**Thread**
- Threads are within the same process address space, thus, **much of the information** present in the memory description of the process can be shared across threads.

- **Some information cannot be replicated**, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information suffices to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.
An execution state (running, ready, etc.): one thread might be running and other one is blocked.

Saved context of thread when not running

Has an execution stack: to keep the information of functions calls of thread

Access to the memory and resources of its process
  - all threads of a process share this
  - when one thread alters a (non-private) memory item, all other threads (of the process) see it.
  - a file opened by one thread is available to others

[One way to view a thread is as an independent program counter (trace) operating within a process.]
Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - all threads of a process share the same address space and other resources
  - any alteration of a resource by one thread affects the other threads in the same process
Threads: Management Point of View

Figure 4.2 Single Threaded and Multithreaded Process Models
Multithreading vs Single Threading

- **Multithreading** - The ability of an OS to support multiple, *concurrent* paths of execution within a single process

- **Single threading**: when the OS does not recognize the concept of thread
Classification of OS

MS-DOS supports a single user process and a single thread.

Some UNIX variants support multiple user processes but only support one thread per process.

Java run-time environment (JRE) has a single process with multiple threads.

Multiple processes and threads are found in Windows, Solaris, and many modern versions of UNIX.

Figure 4.1  Threads and Processes [ANDE97]
Threads and Thread Usage

Multiple tasks of an application can be implemented by separate threads.

Word Processing application with multiple tasks:

- Update display
- Spell checking
- sentence alignment program
- Answer a network request

independent process for each program or a single process with multiple threads??
Thread Use in a Single-User System

It has many applications:

- **Foreground and background work**

  one thread could display menus and read user input, while another thread executes user commands

- **Asynchronous processing**

- **Speed of execution**

  even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing.

- **Modular program structure**

  Programs that involve a variety of activities
Threads and Thread Usage

- **Responsiveness**
  - One thread blocks, another one runs.
  - One thread may always wait for the user

- **Resource Sharing**
  - Threads can easily share resources

- **Economy**
  - Creating a thread is fast
  - Context switching among threads may be faster!
how a process can be run faster by multi-threading?
Multithreading Execution

CPU

single-threaded process

multi-threaded process
Multithreading Concept

Schedulable Entities
We can select one of them and run
Example

• Consider an application that consists of several independent parts that do not need to run in sequence

• Each part can be implemented as a thread

• Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of switching to another process)

• Word Processor Editor + Spell Checker + etc
function1(…)
{
    ....
}

function2(…)
{
    ....
}

main()
{
    ....
    thread_create (function1 ,...);
    ....
    thread_create (function2, ...);
    ....
    thread_create (function1, ...);
    ....
}
Parallel Computation: Merge Sort

- Sorting a huge list, e.g., MergeSort or QuickSort
- Can be parallelized by multi-thread programming?
- Compare it to multi-processes?
• Process characteristics
  – Need for threads
  – Similarity and difference between threads and processes

• Advantages and complications with threads

• User level threads

• Kernel level threads
Benefits of Threads

• Managing threads (such as creation and termination) is cheaper: Compared to processes, threads take less time to create or terminate.

• Switching between two threads within the same process takes less time than a process switch: thread switching happens inside a process and is much faster.

• Threads can communicate with each other without invoking the kernel (since they share resources such as main memory).

[Shared memory versus Inter-Process Communication (IPC)]

we need to be careful!!!!!!
Multithreaded Process

concurrent execution without the overhead of switching multiple processes

Achieves concurrency without the overhead of using multiple processes

Threads within the same process can exchange information through their common address space and have access to the shared resources of the process

Threads in different processes can exchange information through shared memory that has been set up between the two processes

Fast data exchange by shared memory
Caution: An Example of Inconsistency

- Three variables: A, B, C which are shared by thread T1 and thread T2
- T1 computes C = A+B
- T2 transfers amount X from A to B
  - T2 must do: A = A -X and B = B+X (so that A+B is unchanged)

- But if T1 computes A+B after T2 has done A = A-X but before B = B+X
- then T1 will not obtain the correct result for C = A + B

Protecting Shared Resources: Mutual Exclusion
Threads

- Process-level actions may affect all of the threads in a process

- Suspending a process involves suspending all threads of the process since all threads share the same address space

- Termination of a process, terminates all threads within the process

What operations would change the thread execution state?
The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn: Spawn another thread (into ready state)
- Block (per process or per thread): Implementation issue: block entire process, or just that thread?
- Unblock
- Finish (thread): Deallocate register context and stacks

Suspend is a process-level concepts: if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process
Consider a program that makes two remote procedure calls requests of two different hosts to obtain a combined result.

the results are obtained in sequence so the program has to wait for a response from each server in turn.
Rewriting the program to use a separate thread for each RPC results in a substantial speedup. The program waits concurrently for the two replies.
Multithreading on a Uniprocessor

Figure 4.4  Multithreading Example on a Uniprocessor
Concurrent Execution on a Single-core System
Parallel Execution on a Multicore System
Outline of Chapter 4

• Process characteristics
  – Need for threads
  – Similarity and difference between threads and processes

• Advantages and complications with threads

• **User level threads**

• Kernel level threads
Categories of Thread Implementation
[Types of Threads]

• User Level Threads (ULT)
• Kernel Level Threads (KLT) also called:
  • kernel-supported threads
  • lightweight processes.
Kernel Space

Process A

Process B

PCB A

PCB B

Thread table

Kernel process table

User Space

Process A

Process B

PCB A

PCB B

Thread

Run-time System (library)

Thread table

Kernel process table
User-Level Threads (ULTs)

- All thread management is done by the application typically by using a thread library
- The kernel is not aware of the existence of threads
- All thread management is done by the application
- Thread states and scheduling are managed within a process, by that process
Threads library

• Contains code for:
  – creating and destroying threads
  – passing messages and data between threads
  – scheduling thread execution
  – saving and restoring thread contexts
Kernel activity for ULT

- The kernel is NOT aware of thread activity but it is still manages process activity
- When a thread makes a system call, the whole process will be blocked

- Jacket (up-call) is a call from the kernel space to user space
- Observe that all other calls are from user space to kernel space
Advantages and inconveniences of ULT

• Advantages
  – Thread switching does not require kernel privileges (i.e., no mode switch involved)
  – Scheduling can be application specific: choose the best algorithm.
  – ULTs can run on any OS. Need only a thread library

• Inconveniences
  – Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked (Blocking system calls block whole process)
  – The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors (Concurrent execution of multiple threads within a process is not possible)
ULTs are OS Independent

Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS
Disadvantages of ULTs

- In a typical OS many system calls are **blocking**
- as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application **cannot** take advantage of multiprocessing (multi-processor or multi-core)
Overcoming ULT Disadvantages

(1) Jacketing -- converts a blocking system call into a non-blocking system call

(2) Writing an application as multiple processes rather than multiple threads
• Process characteristics
  – Need for threads
  – Similarity and difference between threads and processes

• Advantages and complications with threads

• User level threads

• **Kernel level threads**
Kernel-Level Threads (KLTs)

- All management is done by the kernel
- Kernel maintains context information for the process and the threads
- **no thread management is done by the application** (No thread library)
- API to kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling is done on a thread basis
- MS Windows is an example of this approach
Advantages of KLTs

- The kernel can simultaneously schedule many threads from the same process on many processors.
- Blocking is done on a thread level: If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.
Disadvantage of KLTs

• The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

• Thread switching within the same process involves the kernel and results in a significant overhead.
Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- Multiple ULTs are mapped onto smaller or equal number of KLTs
- The programmer may adjust the number of KLTs
- May combine the best of both approaches
- Example: Solaris
Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example
Pthreads

- POSIX standard (IEEE 1003.1c)
- Defining an API for thread creation and synchronization
- A specification for thread behavior, not an implementation
- OS designers may implement the specification in any way they wish
- UNIX-based systems, such as Solaris 2
Summary

- **User-level threads**
  - created and managed by a threads library that runs in the user space of a process
  - a mode switch is not required to switch from one thread to another
  - only a single user-level thread within a process can execute at a time
  - if one thread blocks, the entire process is blocked

- **Kernel-level threads**
  - threads within a process that are maintained by the kernel
  - a mode switch is required to switch from one thread to another
  - multiple threads within the same process can execute in parallel on a multiprocessor
  - blocking of a thread does not block the entire process

- **Process/related to resource ownership**

- **Thread/related to program execution**