Today’s Topics

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples
Processes

- Heavy-weight
  - A process includes many things:
    - An address space (all the code and data pages)
    - OS resources (e.g., open files) and accounting info.
    - Hardware execution state (PC, SP, registers, etc.)
  - Creating a new process is costly because all of the data structures must be allocated and initialized
    - Linux: over 100 fields in task_struct
      (excluding page tables, etc.)
  - Inter-process communication is costly, since it must usually go through the OS
    - Overhead of system calls and copying data
Concurrent Servers: Processes

- Web server example
  - Using fork() to create new processes to handle requests in parallel is overkill for such a simple task

```c
While (1) {
    int sock = accept();
    if ((pid = fork()) == 0) {
        /* Handle client request */
    } else {
        /* Close socket */
    }
}
```
Cooperating Processes

- **Example**
  - A web server, which forks off copies of itself to handle multiple simultaneous tasks
  - Any parallel program on a multiprocessor

- **We need to**
  - Create several processes that execute in parallel
  - Cause each to map the same address space to share data (e.g., shared memory)
  - Have the OS schedule these processes in parallel

- **This is very inefficient!**
  - Space: PCB, page tables, etc.
  - Time: creating OS structures, fork and copy address space, etc.
Rethinking Processes

- What’s similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privilege
  - They all share the same resources (files, sockets, etc.)

- What’s different?
  - Each has its own hardware execution state: PC, registers, SP, and stack.
Key Idea (1)

- Separate the concept of a process from its execution state
  - Process: address space, resources, other general process attributes (e.g., privileges)
  - Execution state: PC, SP, registers, etc.

- This execution state is usually called
  - a thread of control
  - a thread or
  - a lightweight process (LWP)
Key Idea (2)
Key Idea (3)
What is a Thread?

- A thread of control (or a thread)
  - A sequence of instructions being executed in a program
  - Usually consists of
    - a program counter (PC)
    - a stack to keep track of local variables and return addresses
    - registers
  - Threads share the process instructions and most of its data
    - A change in shared data by one thread can be seen by the other threads in the process
- Threads also share most of the OS state of a process
Concurrent Servers: Threads

- Using threads
  - We can create a new thread for each request

```c
webserver ()
{
    While (1) {
        int sock = accept();
        thread_fork (handle_request, sock);
    }
}
handle_request (int sock)
{
    /* Process request */
    close (sock);
}
```
Concurrent Servers: Threads

(1) request

(2) create new thread to service the request

(3) resume listening for additional client requests
Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures
- **Programming Easiness**
Process Address Space

- **0xFFFFFFFF**
- **address space**
- **0x00000000**

**Stack**
- Dynamically allocated memory

**Heap**
- Dynamically allocated memory

**Static Data**
- Data segment

**Code**
- Text segment

**PC**
- Program Counter

**SP**
- Stack Pointer
Address Space with Threads

Address space

0xFFFFF00000000

0x00000000

PC

PC

PC

SP

SP

SP

THREE THREADS

code (text segment)

static data (data segment)

heap (dynamically allocated mem)

thread 3 stack

thread 2 stack

thread 1 stack

x00000000 x00000000
User Threads and Kernel Threads

- **User threads**
  - Management done by user-level threads library
  - Three primary thread libraries
    - POSIX Pthreads
    - Windows threads
    - Java threads

- **Kernel threads**
  - Supported by the Kernel
  - Examples – virtually all general purpose operating systems, including
    - Windows
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X
Kernel-level Threads

- The OS manages threads and processes
- All thread operations are implemented in the kernel
- The OS schedules all of the threads in a system.
  - If one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
  - Possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - Less state to allocate and initialize
Kernel-level Threads

- They can still be too expensive.
  - For fine-grained concurrency, we need even cheaper threads.
  - Ideally, we want thread operations as fast as a procedure call.
- Thread operations are all system calls.
  - The program must cross an extra protection boundary on every thread operation, even when the processor is being switched between threads in the same address space.
  - The OS must perform all of the usual argument checks.
- Must maintain kernel state for each thread
  - Can place limit on the number of simultaneous threads.
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.
Implementing Kernel-level Threads

- Kernel-level threads
  - Kernel-level threads are similar to original process management and implementation
User-level Threads

- **Motivation**
  - To make threads cheap and fast, they need to be implemented at the user level
  - Portable: User-level threads are managed entirely by the runtime system (user-level library)

- **User-level threads are small and fast**
  - Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - Creating a thread, switching between threads, and synchronizing threads are done via procedure calls (No kernel involvement)
  - User-level thread operations can be 10-100x faster than kernel-level threads
User-level Threads

- Limitations
  - User-level threads are invisible to the OS
    - They are not well integrated with the OS
  - As a result, the OS can make poor decisions
    - Scheduling a process with only idle threads
    - Blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run
    - Unscheduling a process with a thread holding a lock
  - Solving this requires coordination between the kernel and the user-level thread manager
    - e.g., all blocking system calls should be emulated in the library via non-blocking calls to the kernel
Implementing User-level Threads

- User-level threads

Diagram showing the relationship between processes and threads within the user and kernel spaces.
Implementing User-level Threads

- **Thread context switch**
  - Very simple for user-level threads
    - Save context of currently running thread
      : push all machine state onto its stack
    - Restore context of the next thread
      : pop machine state from next thread’s stack
    - The next thread becomes the current thread
    - Return to caller as the new thread
      : execution resumes at PC of next thread
  - All done by assembly languages
    - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls
Implementing User-level Threads

- Thread scheduling
  - A thread scheduler determines when a thread runs
    - Just like the OS and processes
    - But implemented at user-level as a library
  - It uses queues to keep track of what threads are doing
    - Run queue: threads currently running
    - Ready queue: threads ready to run
    - Wait queue: threads blocked for some reason (maybe blocked on I/O or a lock)
  - How can we prevent a thread from hogging the CPU?
Implementing User-level Threads

- Non-preemptive scheduling
  - Force everybody to cooperate
    - Threads willingly give up the CPU by calling yield()
  - yield() calls into the scheduler, which context switches to another ready thread

```c
Thread ping ()
{
    while (1) {
        printf ("ping\n");
        yield();
    }
}

Thread pong ()
{
    while (1) {
        printf ("pong\n");
        yield();
    }
}
```

- What happens if a thread never calls yield()?
Implementing User-level Threads

- Preemptive scheduling
  - Need to regain control of processor asynchronously
  - Scheduler requests that a timer interrupt be delivered by the OS periodically
    - Usually delivered as a UNIX signal
    - Signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - At each timer interrupt, scheduler gains control and context switches as appropriate
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

Examples
- Windows
- Linux
- Solaris 9 and later
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the ThreadFiber package
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
Threads Interface

- Pthreads
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
  - API specifies behavior of the thread library.
  - Implementation is up to development of the library.
  - Common in UNIX operating systems.
Threads Interface

- POSIX-style threads
  - Pthreads
  - DCE threads (early version of Pthreads)
  - Unix International (UI) threads (Solaris threads)
    - Sun Solaris 2, SCO Unixware 2

- Microsoft-style threads
  - Win32 threads
    - Microsoft Windows 98/NT/2000/XP
  - OS/2 threads
    - IBM OS/2
Pthreads Example

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
```
Pthreads Example (Cont.)

```c
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
Threading Issues

- **fork() and exec()**
  - When a thread calls fork(),
    - Does the new process duplicate all the threads?
    - Is the new process single-threaded?
  - Some UNIX systems support two versions of fork().
    - In Pthreads,
      - fork() duplicates only a calling thread.
    - In the Unix International standard,
      - fork() duplicates all parent threads in the child.
      - fork1() duplicates only a calling thread.
  - Normally, exec() replaces the entire process.
Threading Issues

- Thread cancellation
  - The task of terminating a thread before it has completed
  - Asynchronous cancellation
    - Terminates the target thread immediately
    - What happens if the target thread is holding a resource, or it is in the middle of updating shared resources?
  - Deferred cancellation
    - The target thread is terminated at the cancellation points
    - The target thread periodically check if it should be cancelled
- Pthreads API supports both asynchronous and deferred cancellation
Threading Issues

- **Signal handling**
  - Where should a signal be delivered?
  - To the thread to which the signal applies
    - for synchronous signals
  - To every thread in the process
  - To certain threads in the process
    - Typically only to a single thread found in a process that is not blocking the signal
    - Pthreads: per-process pending signals, per-thread blocked signal mask
  - Assign a specific thread to receive all signals for the process
    - Solaris 2
Threading Issues

- **Using libraries**
  - **errno**
    - Each thread should have its own independent version of the errno variable.

- **Multithread-safe (MT-safe)**
  - A set of functions is said to be multithread-safe or reentrant, when the functions may be called by more than one thread at a time without requiring any other action on the caller’s part.
  - Pure functions that access no global data or access only read-only global data are trivially MT-safe.
  - Functions that modify global state must be made MT-safe by synchronizing access to the shared data.