Threads

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from
Modern Operating Systems, 3rd ed., by Andrew Tanenbaum
and from Operating System Concepts, 7th ed., by Silbershatz, Galvin, & Gagne)
Review — Processes in Unix, Linux, and Windows

- an address space – usually protected and virtual – mapped into memory
- the code for the running program
- the data for the running program
- an execution stack and stack pointer (SP); also heap
- the program counter (PC)
- a set of processor registers – general purpose and status
- a set of system resources
  - files, network connections, pipes, …
  - privileges, (human) user association, …
- …
Processes in Unix, Linux, and Windows (continued)

• Evolutionary direction: – Isolation

• I.e., main goal was to keep processes separate from each other
  • Multi-user computer systems (time-sharing)
  • Independent applications

• Communication and cooperation among processes is limited
Processes in Unix, Linux, and Windows (continued)

- Processes in Unix, Linux, and Windows are “heavyweight”
  - Lots of resources
  - Expensive context switches
  - Inflexible
  - ...

CS-3013, A-Term 2011
OS-Centric View of Problem

- Lots of data in PCB & other data structures
  - Even more when we study memory management
  - More than that when we study file systems, etc.
- Processor caches a lot of information
  - Memory Management information
  - Caches of active pages
- Costly context switches and traps
  - Many 10s of microseconds (even 100s of microseconds)
Application-Centric View of Problem

- Separate processes have separate address spaces
  - Shared memory is limited or non-existent
  - Applications with internal concurrency are difficult

- Isolation between independent processes vs. cooperating activities
  - Fundamentally different goals
Example

- **Web Server**
  - Need to support multiple concurrent requests pertaining to common data

- **One solution:**
  - create several processes that execute in parallel
  - Use shared memory — `shmget()` — to map to the same address space into multiple processes
  - have the OS schedule them in parallel

- **Clumsy and inefficient**
  - *Space and time:* PCB, page tables, cloning entire process, etc.
  - *Programming:* `shmget()` is really hard to program!
Example 2

- Transaction processing systems
  - E.g., airline reservations or bank ATM transactions
- 1000’s of transactions *per second*
  - Very small computation per transaction
  - Long wait times for data base access
- Separate processes per transaction are too costly
- Other techniques (e.g., message passing) are much more complex
Example 3

- Games have multiple active characters
  - Independent behaviors
  - Common context or environment
- Need “real-time” response to user
  - For interactive gaming experience
- Programming all characters in separate processes is really, really hard!
- Programming them in a single process is much harder without concurrency support.
This problem ...

- ... is partly an artifact of
  - Unix, Linux, and Windows
  - Big, powerful processors (e.g., Pentium, Athlon)

- ... partly a consequence of history
  - Shared computers rather than personal computers

- ... tends to occur in most large systems

- ... is infrequent in small-scale systems
  - PDAs, cell phones
  - Closed systems (i.e., controlled applications)
**Solution:– Threads**

- A *thread* is a particular execution of a program, function, or procedure *within the context* of a Linux or Windows process
  - i.e., a specialization of the concept of *process*
- A thread has its own
  - Program counter, registers, PSW
  - Stack
- A thread shares
  - Address space, heap, static data, program code
  - Files, privileges, all other resources
  - with all other threads of the same process
Reading Assignment

- Robert Love, *Linux Kernel Development*
  - Chapter 3 – “Process Management”
- Tanenbaum
  - §2.2 – “Threads”

*Chapters 1 & 2 are useful, too!*
Definition:– Thread

- A thread is a particular execution of a program or procedure within the context of a Unix, Linux, or Windows process
  - i.e., a specialization of the concept of process
- A thread has its own
  - Program counter, registers, PSW
  - Stack
- A thread shares
  - Address space, heap, static data, program code
  - Files, privileges, all other resources
  with all other threads of the same process
void main(int argc, char** argv);
char *f(char *c, int i);

main(...)
f(...)

int j = ...;
char *msg = malloc(...);
thread_t T;
T = new_thread f(msg,j);
...
while (...) {
    /* loop */
}
...

char *d = malloc(...);
for (i=0;i<j;i++){
    ...
    d[i] = c[i];
}
return d;

char *result = join T;

Same heap
Address Space
Linux-Windows process

From previous lesson

Virtual address space

0xFFFFFFFFF

0x00000000

stack
(dynamically allocated)

heap
(dynamically allocated)

static data

program code
(text)

PC

SP
Address Space for Multiple Threads

Virtual address space

0xFFFFFFFF

0x00000000

Thread 1 stack

Thread 2 stack

Thread 3 stack

Heap

Static data

Code (text)

PC (T1)

SP (T1)

PC (T2)

SP (T2)

PC (T3)

SP (T3)
Basic Thread Functions

- **Create a thread**
  - Make a function call that executes *concurrently* with caller

- **Exit a thread**
  - *i.e.*, thread terminates or returns from its function

- **Join a thread**
  - Wait until the designated thread terminates, capture its return value, delete its stack

- …
Basic Thread Functions (continued)

- ... 
- **Detach a thread**
  - Separate it from its creators
  - Becomes a standalone, independent execution
  - Not “joinable”; no value returned
  - Stack goes away upon termination
- **Get own thread ID**
  - *i.e.*, get the number of my own thread
- ...
Another Example

// web server
while (true) {
    listen for web req.
    allocate priv. socket
    create & detach new
    thread to serve req.
    ...
}

// individual request
//   handler
interpret request
while (!done) {
    interact with client
    over private socket
    ...
}
exit thread
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Better utilization of multi-processor architectures than achievable with just processes
Using Threads

• Everyone uses threads nowadays
  • Many purposes
  • Many languages

• Programming applications with concurrent activity within them has become an essential skill
  • I.e., some form of threads
Questions?
Tools for using Threads

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
Thread Interface

- E.g., POSIX pthreads API:
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(start_routine) (void), void *arg)`
    - creates a new thread of control
    - new thread begins executing at start_routine
  - `pthread_exit(void *value_ptr)`
    - terminates the calling thread
  - `pthread_join(pthread_t thread, void **value_ptr)`
    - blocks the calling thread until the specified thread terminates
  - `pthread_t pthread_self()`
    - Returns the calling thread's identifier

See the `man` pages for these functions
Project 3 will use pthreads
Implementing Threads

- **In User space**
  - User-space function library
  - Runtime system – similar to process management except in user space
  - Windows NT – fibers: a user-level thread mechanism

- **In Kernel**
  - Primitive objects known to and scheduled by kernel
  - Linux: lightweight process (LWP)
  - Windows NT, XP, Vista, 7:– threads
Implementing Threads

• In **User space**
  – User-space function library
  – Runtime system – similar to process management except in user space
  – Windows NT – *fibers*: a user-level thread mechanism

• In **Kernel**
  – Primitive objects known to and scheduled by kernel
  – Linux: *lightweight process* (LWP)
  – Windows NT, XP, Vista, 7:– *threads*
Threads Implemented in User Space

• Thread management done by non-privileged threads library
• Runs in process address space
User-Space Threads (continued)

- Can be implemented without kernel support
  - ... or knowledge!
- Program links with a runtime system that does thread management
  - Operations are very efficient (function calls)
  - Space efficient and all in user space (TCB)
  - Task switching is very fast
- Since kernel not aware of threads, there can be scheduling inefficiencies or issues
  - E.g., blocking I/O calls
  - Non-concurrency of threads on multiple processors
User-Space Threads

- Obsolete because all modern kernels support threads at kernel level
- Still (somewhat of) a performance issue
- Research focused on how to avoid system calls for thread synchronization, creation, deletion, etc.
Questions?
Threads Implemented in Kernel

• Supported by the Kernel
  • OS maintains data structures for thread state and does all of the work of thread implementation.

• Examples
  • Windows XP/2000/Vista/7
  • Solaris
  • Linux version 2.6
  • Tru64 UNIX
  • Mac OS X
Kernel-level Threads (continued)

- OS schedules threads instead of processes

Benefits
- Overlap I/O and computing in a process
- Creation is cheaper than processes
- Context switch can be faster than processes

Negatives
- System calls needed for thread operations
  - High overhead
- Additional OS data space for each thread
Threads Supported by Processor

- E.g., Pentium 4 with Hyperthreading™
  - [www.intel.com/products/ht/hyperthreading_more.htm](http://www.intel.com/products/ht/hyperthreading_more.htm)
- Multiple processor cores on a single chip
  - True concurrent execution within a single process
- Requires kernel support
- Exposes many issues
  - Critical section management of synchronization primitives at kernel level
  - Multiple threads scheduling from same ready queue
  - Multiple interrupts in progress at one time
Unix Processes vs. Threads

- On a 700 Mhz Pentium running Linux
  - Processes:
    - `fork()`/`exit()`: 250 microsec
  - Kernel threads:
    - `pthread_create()`/`pthread_join()`: 90 microsec
  - User-level threads:
    - `pthread_create()`/`pthread_join()`: 5 microsec
Some Issues Pertaining to Threads

- Process global variables
  - E.g., **ERRNO** in Unix — a static variable set by system calls
- Semantics of *fork()* and *exec()* system calls for processes
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations
Semantics of `fork()` and `exec()`

- Does `fork()` duplicate only the calling thread or all threads?
  - Easy if user-level threads
    - All threads are duplicated, unbeknownst to kernel
  - Not so easy with kernel-level threads
    - Linux has special `clone()` operation
    - Windows XP/Vista has something similar
Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - *Asynchronous cancellation* terminates the target thread immediately
  - *Deferred cancellation* allows the target thread to periodically check if it should be cancelled
Signal Handling

• **Signals** are used in UNIX/Linux systems to notify a process that a particular event has occurred
  - AKA software interrupts

• A **signal handler** is used to process a class of signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal handling implemented as forced function call by process

• **Options for multi-threaded process:**
  - Deliver the signal to the thread to which the signal applies
    - How does system know which one?
  - Deliver the signal to every thread in the process
  - Assign specific threads to receive specific signals for the process
Models for Kernel Implementation

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

This was a big issue a decade ago
- Still discussed in textbooks

Hardly relevant today
- Windows, Linux, and most Unix systems provide kernel threads

See extra slides on course web site at end of this topic
Modern Linux Threads

• Implemented in kernel
• “A thread is just a special kind of process.”
  • Robert Love, *Linux Kernel Development*, p.23
• The primary unit of scheduling and computation implemented by Linux 2.6 kernel
• Every thread has its own task_struct in kernel
• …
Modern Linux Threads (continued)

- *Process* `task_struct` has pointers to own memory & resources
- *Thread* `task_struct` has pointer to process’s memory & resources

- `fork()` and `thread_create()` are library functions implemented by `clone()` kernel call
- ...

CS-3013, A-Term 2011
Modern Linux Threads (continued)

- Threads are scheduled independently of each other
- Threads can block independently of each other
  - Even threads of same process
- Threads can make their own kernel calls
  - Kernel maintains a small *kernel stack* per thread
  - During kernel call, kernel is in *process context*
Digression – Process Address Space

- Linux includes (parts of) *kernel* in every address space
  - Protected
  - Easy to access
  - Allows *kernel* to see into client processes
    - Transferring data
    - Examining state
    - ...

- Also many other operating systems
Processes – Address Space

Virtual address space

Kernel Space

User Space

Kernel Code and Data

- stack (dynamically allocated)
- heap (dynamically allocated)
- static data
- code (text)

32-bit Linux & Win XP – 3G/1G user space/kernel space
Linux Kernel Implementation

- Kernel may execute in either *Process context* vs. *Interrupt context*
- In *Process context*, kernel has access to
  - Virtual memory, files, other process resources
  - May sleep, take page faults, etc., on behalf of process
- In *Interrupt context*, no assumption about what process was executing (if any)
  - No access to virtual memory, files, resources
  - May not sleep, take page faults, etc.
Modern Linux Threads (continued)

• Multiple threads can be executing in kernel at same time
• When in process context, kernel can
  • sleep on behalf of its thread
  • take pages faults on behalf of its thread
  • move data between kernel and process or thread
  • ...

CS-3013, A-Term 2011
Threads in Linux Kernel

- Kernel has its own threads
  - No associated process context
- Supports concurrent activity within kernel
  - Multiple devices operating at one time
  - Multiple application activities at one time
  - Multiple processors in kernel at one time
- A useful tool
  - Special kernel thread packages, synchronization primitives, etc.
  - Useful for complex OS environments
Windows Vista Threads

• Much like to Linux 2.6 threads
  • Primitive unit of scheduling defined by kernel
  • Threads can block independently of each other
  • Threads can make kernel calls
  • …

• Process
  • A higher level (non-kernel) abstraction
  • A container

• See Tanenbaum, §11.4
Threads – Summary

• Threads were invented to counteract the heavyweight nature of *Processes* in Unix, Windows, etc.

• Provide lightweight concurrency *within* a single address space

• Have evolved to become *the* primitive abstraction defined by kernel
  • Fundamental unit of scheduling in Linux, Windows, etc
Reading Assignment

- Robert Love, *Linux Kernel Development*
  - Chapter 3 – “Process Management”
- Tanenbaum
  - §2.2 – “Threads”
Questions?
Single and Multithreaded Processes

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

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