CIS 415
Operating Systems
Threads

Prof. Allen D. Malony
Department of Computer and Information Science
Fall 2014

UNIVERSITY OF OREGON
Outline

- Processes revisited
- Threads, threads, threads, ....

Reminders

- Assignment 1 due Friday, October 17
**Process Control Block**

- Information associated with each process
  - Also called task control block
- Process state
  - Running, waiting, …
- Program counter
  - Location of instruction to next execute
- CPU registers
  - Contents of all process-centric registers
- CPU scheduling information
  - Priorities, scheduling queue pointers, …
- Memory-management information
  - Memory allocated to the process
- Accounting information
  - CPU used, elapsed time, time limits, …
- I/O status information
  - I/O devices allocated to process, list of open files

**PCB**

<table>
<thead>
<tr>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process number</td>
</tr>
<tr>
<td>program counter</td>
</tr>
<tr>
<td>registers</td>
</tr>
<tr>
<td>memory limits</td>
</tr>
<tr>
<td>list of open files</td>
</tr>
</tbody>
</table>
Process Context Switch

How long does this take?
Both are determined by the PCB size

execute $P_0$

save state into PCB$_0$

interrupt or system call

reload state from PCB$_1$

execute $P_1$

idle
What must the OS do to run these processes?

The child processes do not exec() another program, but continue to use the parent’s program code. They do, however, get a copy of all the parent’s process state.
Why Threads?

- A process is “a program in execution”
  - Memory address space containing code and data
  - Other resources (e.g., open file descriptors)
  - State information (PC, register, SP) => PCB details

- Consider a process in 2 respects (categories)
  - Collection of resources required for execution
    - code, addr space, open files, …
  - “Thread” of execution
    - current state of execution (CPU state)

- Can think about these separately
**Process Model**

- Much of the OS’s job is keeping processes from interfering with each other … Why?
- Each process has its own resources to use
  - Program code to execute, address space, files, …
- Good for isolation, bad because of context switching between processes
  - Full *process swap* required
  - OS must intervene to save/restore all process state
  - Some apps could contain multiple threads of execution but only need one grouping of resources
Terminology

- Multiprogramming
  - Run multiple processes concurrently on a single processor
  - OS choose which process to run out of multiple

- Multiprocessing
  - Run multiple processes on multiple processors
  - OS manages mapping of processes to processors

- Multithreading
  - Define multiple execution contexts (threads) in a single address space (of a process)
  - OS manages mapping of threads to an address space
  - OS manages mapping of threads to processor(s)
What’s a Thread?

- Thread of execution through a program on CPU
  - Program counter
  - Registers

- Memory
  - Address space (process)
    - address space is shared!
  - Stack (per thread)
    - each thread has its own stack pointer
  - Heap (process) + private dynamic memory (per thread)

- I/O
  - Share files, sockets, … (process)
Multithreaded Applications

- Multiple threads sharing a common address space
- Why would you want to do that?
- Some applications could be written to support concurrency
  - How do you get concurrency?
  - One way is to create processes

- Some applications want to share data structures among concurrently executing parts of the computation
  - Is this possible with processes?
  - Is it difficult with processes?

- Threads could be used if there is no need for the OS to enforce resource separation (trust amongst the threads)
Advantages of Threads

- Improve responsiveness
  - Ideally, a thread is always ready

- Resource sharing
  - All the stuff is easily accessible

- Economy of resources
  - Thread resources are cheaper than process resources

- Utilization of multiprocessors
  - Get all of them running
Single-Threaded vs. Multi-Threaded

- Regular UNIX process can be thought of as a special case of a multithreaded process
  - A process that contains just one thread
Working with Threads

- In a C program
  - `main()` procedure defines the first thread
  - C programs always start at `main`

- Create a second thread
  - Allocate resources to maintain a second execution context in same address space
    - think about what process fields will be necessary for a thread
  - Supply a procedure name to start the new thread’s execution
Threads vs. Processes

- Easier to create than a new process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Less communication overheads
  - Communicating between the threads of one process is simple because the threads share everything
  - Address space is shared
Which is Cheaper?

- Create new process or create new thread (in existing process)
- Context switch between processes or threads
- Interprocess or inter-thread communication
- Sharing memory between processes or threads
- Terminate a process or terminate a thread (not last one)

<table>
<thead>
<tr>
<th>Process creation method</th>
<th>Time (sec), elapsed (real)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>22.27 (7.99)</td>
</tr>
<tr>
<td>vfork()</td>
<td>3.52 (2.49)</td>
</tr>
<tr>
<td>clone()</td>
<td>2.97 (2.14)</td>
</tr>
</tbody>
</table>

Time to create 100,000 processes (Linux 2.6 kernel, x86-32 system)
Implications?

- 0.22 ms per fork
  - Maximum of \((1000 / 0.22) = 4545.5\) connections per second
  - 0.45 billion connections per day per machine
    - fine for most servers
    - too slow for a few super-high-traffic front-line web services

- Facebook serves \(\mathcal{O}(750\ \text{billion})\) page views per day
  - Guess ~1-20 HTTP connections per page
  - Would need 3,000 -- 60,000 machines just to handle `fork()`, without doing any work for each connection!
Thread Attributes

• Global to process:
  – memory
  – PID, PPID, GID, SID
  – controlling term
  – process credentials
  – record locks
  – FS information
  – timers
  – resource limits
  – and more...

• Local to specific thread:
  – thread ID
  – stack
  – signal mask
  – thread-specific data
  – alternate signal stack
  – error return value
  – scheduling policy/priority
  – Linux-specific (e.g., CPU affinity)
Threading Models

- Programming: Library or system call interface
  - Kernel Threading
    - thread management support in the kernel
    - invoked via system call
  - User-space threading
    - thread management support in user-space library
    - linked into your program

- Scheduling: Application or kernel scheduling
  - May create user-level or kernel-level threads
    - NOTE: CPU only runs kernel threads!!!
User-Space Threads

- Thread management support in user-space library
  - Sets of functions for creating, invoking, and switching among threads
  - Need to switch threads stacks in user space …

- Linked into your program
  - Thread libraries

- Examples
  - POSIX Threads (PThreads)
  - Win32 Threads
  - Java Threads
Implementing Threading

- Threads can perform operations in user mode that are usually handled by the OS
  - Assumes cooperating threads so hardware enforcement of separation not required

- Idea: “dispatcher” subroutine in the process is called when a thread is ready to relinquish control to another thread
  - Manages stack pointer, program counter
  - Switches process’s internal state among threads
Kernel Threads

- Thread management support in kernel
  - Sets of system calls for creating, invoking, and switching among threads
- Supported and managed directly by the OS
  - Thread objects in the kernel
- Nearly all OSes support a notion of threads
  - Linux -- thread and process abstractions are mixed
  - Solaris
  - Mac OS X
  - Windows XP
  - …
Many-to-one Thread Model

- Many user-level threads correspond to a single kernel thread
  - Kernel is not aware of the mapping
  - Handled by a thread library

- How does it work?
  - Create and execute a new thread
  - Upon yield, switch to another user thread in the same process
    - Kernel is unaware
  - Upon wait, all threads are blocked
    - Kernel is unaware there are other options
    - Can not wait and run at the same time
One-to-one Thread Model

- One user-level thread per kernel thread
  - A kernel thread is allocated for every user-level thread
  - Must get the kernel to allocate resources for each new user-level thread

- How does it work?
  - Create new thread
    - system call to kernel
  - Upon yield, switch to another kernel thread in system
    - kernel is aware
  - Upon wait, another thread in the process may run
    - only the single kernel thread is blocked
    - kernel is aware there are other options in this process
Many-to-many Thread Model

- A pool of user-level threads maps to a pool of kernel threads
  - Pool sizes can be different (kernel pool is no larger)
  - A kernel thread is pool is allocated for every user-level thread
  - No need for the kernel to allocate resources for each new user-level thread

- How does it work?
  - Create new thread
    - may map to kernel thread dynamically
  - Upon yield, switch to another thread
    - kernel is aware
  - Upon wait, another thread in the process may run
    - if a kernel thread is available to be scheduled to that process
    - kernel is aware of the mapping between process threads and kernel threads
Problems solved with threads

- Imagine you are building a web server
  - You could allocate a pool of threads, one for each client
    - thread would wait for a request, get content file, return it
  - How would the different thread models impact this?

- Imagine you are building a web browser
  - You could allocate a pool of threads
    - some for user interface
    - some for retrieving content
    - some for rendering content
  - What happens if the user decided to stop the request?
    - mouse click on the stop button
Multithreaded Server Architecture

(1) request

(2) create new thread to service the request

(3) resume listening for additional client requests
Linux Threads

- Linux uses a one-to-one thread model
  - Threads are called tasks

- Linux views threads as “contexts of execution”
  - Threads are defined separately from processes
  - That is, a thread is assigned an address space
Linux Threads

- Linux system call
  - `clone(int (*fn)(), void **stack, int flags, int argc, ...)`
  - Create a new thread (Linux task)

- May be created in the same address space or not
  - Flags (on means “share”):
    - Clone VM
    - Clone Filesystem
    - Clone Files
    - Clone Signal Handlers
  - If `clone` with all these flags off, what system call is `clone` equal to?
POSIX Threads

- POSIX Threads or Pthreads is a thread API specification
  - Not directly an implementation
  - Could be mapped to libraries or system calls
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - Specification, not implementation
  - Provided either as user-level or kernel-level
  - API specifies behavior of the thread library
  - Implementation is up to development of the library
- Common in UNIX operating systems
  - Solaris, Linux, Mac OS X
POSIX Threads

- Interface

- `pthread_create()`
  - Thread ID {of the new thread}
    - set on return
  - Attributes {of the new thread}
    - stack size, scheduling information, etc.
  - Function {one arg only}
    - start function
  - Arguments {for function}
  - Return value, status {of the system call}
POSIX Threads

• `pthread_create()`
  – start the thread

• `pthread_self()`
  – return thread ID

• `pthread_equal()`
  – for comparisons of thread ID's

• `pthread_exit()`
  – or just return from the start function

• `thread_join()`
  – wait for another thread to terminate & retrieve value from `pthread_exit()`

• `pthread_cancel()`
  – terminate a thread, by TID

• `pthread_detach()`
  – thread is immune to join or cancel & runs independently until it terminates

• `pthread_attr_init()`
  – thread attribute modifiers
POSIX Threads FAQ

- How do pass multiple arguments to start a thread?
  - Build a struct and pass a pointer to it
- Is the pthreads id unique to the system?
  - No, just process -- Linux task ids are system-wide
- After `pthread_create`, which thread is running?
  - Like fork
- How many threads do exit terminate?
  - `pthread_exit`?
  - All in process; only caller
- How are variables shared by threads?
  - Globals, local static, dynamic data (heap)
Pthreads Example

#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>\n");
        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);
}
```
Pthreads Code for Joining 10 Threads

#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);
Concurrency with threads

- A single process handles all of the connections
  - But, a parent thread forks (or dispatches) a new thread to handle each connection
  - The child thread:
    - handles the new connection
    - exits when the connection terminates
Graphically

```
s server
accept()
```
Graphically
Graphically

client

pthread_create()

server
Graphically
Graphically

client

pthread_create()

server

client
Graphically
Implications?

- 0.0297 ms per thread create; 10x faster than process forking
  - Maximum of \( \frac{1000}{0.0297} \approx 33,670 \) connections per second
  - 3 billion connections per day per machine
    - much, much better

- But, writing safe multithreaded code can be complicated
Concurrent threads

- **Benefits**
  - Straight-line code, line processes or sequential
    - still the case that much of the code is identical!
  - Parallel execution; good CPU, network utilization
    - lower overhead than processes
  - Shared-memory communication is possible

- **Disadvantages**
  - Synchronization is complicated
  - Shared fate within a process; one rogue thread can hurt you badly
Inter-Thread Communication

- Can you use shared memory?
  - Already have it
  - Just need to allocate memory in the address space
    - No need for shm
    - Programming to pipes provides abstraction

- Can you use message passing?
  - Sure
  - Would have to build infrastructure
Fork/Exec Issues

- Semantics are ambiguous for multi-threaded processes

- `fork()`
  - How does it interact with threads?

- `exec()`
  - What happens to the other threads?

- `fork`, then `exec`
  - Should all threads be copied?
Thread Cancellation

- So, you want to stop a thread from executing
  - Do not need it anymore
- Two choices
  - Synchronous cancellation
    - wait for the thread to reach a point where cancellation is permitted
    - no such operation in Pthreads, but can create your own
  - Asynchronous cancellation
    - terminate it now
    - `pthread_cancel(thread_id)`
Signal Handling

- What’s a signal?
  - A form of IPC
  - Send a particular signal to another process

- Receiver’s signal handler processes signal on receipt

- Example
  - Tell the Internet daemon (inetd) to reread its config file
  - Send signal to inetd: kill -SIGHUP <pid>
  - inetd’s signal handler for the SIGHUP signal re-reads the config file

- Note: some signals cannot be handled by the receiving process, so they cause default action (kill the process)
Signal Handling

- Synchronous Signals
  - Generated by the kernel for the process
  - Due to an exception -- divide by 0
    - events caused by the thread receiving the signal

- Asynchronous Signals
  - Generated by another process

- Asynchronous signals are more difficult for multithreading
So, you send a signal to a process

- Which thread should it be delivered to?

**Choices**

- Thread to which the signal applies
- Every thread in the process
- Certain threads in the process
- A specific signal receiving thread

It depends…
Signal Handling and Threads

- UNIX signal model created decades before Pthreads: conflicts arise
- Synchronous vs. asynchronous cases
  - Synchronous
    - Signal is delivered to the same process that caused the signal
    - Which thread(s) would you deliver the signal to?
  - Asynchronous
    - Signal generated by another process
    - Which thread(s) in this case?
**Thread Pools**

- Pool of threads
  - Create (all) at initialization time
  - Assign task to a waiting thread
    - it is already made
  - Use all available threads

- What about when that task is done?
  - Suppose another request is in the queue…
  - Should we use running thread or another thread?

- Problem: setup time

- Faster than setting up a process, but what is necessary?
  - How do we improve performance?
So how many kernel threads should be available for a process?
- In M:N model

Suppose the last kernel thread for an application is to be blocked
- Recall the relationship between kernel and user threads
- What happens?
Scheduling

- Wouldn’t it be nice if the kernel told the application and the application had a way to get more kernel threads?
  - Scheduler activation
    - at thread block, the kernel tells the application via an upcall
    - aside: An upcall is a general term for an invocation of application function from the kernel
  - Application can then get a new thread created
    - See lightweight threads
Reentrance and Thread-Safety

- Terms that you might hear

- Reentrant code
  - Code that can be run by multiple threads concurrently

- Thread-safe libraries
  - Library code that permits multiple threads to invoke the safe function

- Requirements
  - Rely only on input data
    - or some thread-specific data
  - Must be careful about locking (later)
Why not threads?

- Threads can interfere with one another
  - Impact of more threads on caches
  - Impact of more threads on TLB
  - Bug in one thread...

- Executing multiple threads may slow them down
  - Impact of single thread vs. switching among threads

- Harder to program a multithreaded program
  - Multitasking hides context switching
  - Multithreading introduces concurrency issues
Summary of Threads

- Threads
  - A mechanism to improve performance and CPU utilization

- Kernel-space and user-space threads
  - Kernel threads are real, schedulable threads
  - User-space may define its own threads (but not real)

- Threading models and implications
  - Programming systems
  - Multi-threaded design issues

- Useful, but not a panacea
  - Slow down system in some cases
  - Can be difficult to program

- Multiprogramming and multithreading are vital concepts