UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Threads and Cooperation

Questions answered in this lecture:

- Why are threads useful?
- How does one use POSIX pthreads?
- What are user-level versus kernel-level threads?
- How do processes (or threads) communicate (IPC)?

Why support Threads?

Divide large task across several cooperative threads
Multi-threaded task has many performance benefits

- Adapt to slow devices
  One thread waits for device while other threads computes
- Defer work
  One thread performs non-critical work in the background, when idle
- Parallelism
  Each thread runs simultaneously on a multiprocessor

Common Programming Models

Multi-threaded programs tend to be structured in one of three common models:

- Manager/worker
  Single manager handles input and assigns work to the worker threads
- Producer/consumer
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- Pipeline
  Task is divided into series of subtasks, each of which is handled in series by a different thread

What do threads share?

Multiple threads within a single process share:

- Process ID (PID)
- Address space
  - Code (instructions)
  - Most data (heap)
- Open file descriptors
- Signals and signal handlers
- Current working directory
- User and group id

Each thread has its own

- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses
- Signal mask

Thread Operations

Variety of thread systems exist

- Win32 threads
- C-Threads
- POSIX Pthreads

Common thread operations

- Create
- Exit
- Suspend
- Resume
- Sleep
- Wake
- Join (instead of wait())

Operations for threads usually faster than for processes

PThread Example

```
Main()
{
   pthread_t t1, t2;
   char *msg1 = "Thread 1"; char *msg2 = "Thread 2";
   int ret1, ret2;
   ret1 = pthread_create(&t1, NULL, print_fn, (void *)msg1);
   ret2 = pthread_create(&t2, NULL, print_fn, (void *)msg2);
   if (ret1 || ret2) {
        fprintf(stderr, "ERROR: pthread_created failed.\n");
        exit(1);
   }
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   printf("Thread 1 and thread 2 complete.\n");
}
Void print_fn(void *ptr)
{
   printf("%s\n", (char *)ptr);
}
```

Output???

OS Support for Threads

Three approaches for thread support

- User-level threads
- Kernel-level threads
- Hybrid of User-level and Kernel-level threads

Thread Model #1

User-level threads: Many-to-one thread mapping

- Implemented by user-level runtime libraries
  - Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads
  - OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
- Can tune scheduling policy to meet application demands
- Lower overhead thread operations since no system calls

Disadvantages

- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks

Thread Model #2

Kernel-level threads: One-to-one thread mapping

- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages

- Each kernel-level thread can in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages

- Higher overhead for thread operations
- OS must scale well with increasing number of threads

Thread Model #3

Hybrid of Kernel and user-level threads: m-to-n thread mapping

- Application creates m threads
- OS provides **pool** of n kernel threads
- Few user-level threads mapped to each kernel-level thread

Advantages

- Can get best of user-level and kernel-level implementations
- Works well given many short-lived user threads mapped to constant-size pool

Disadvantages

- Complicated…
- How to select mappings?
- How to determine the best number of kernel threads?
  - User specified
  - OS dynamically adjusts number depending on system load

Interprocess Communication (IPC)

To cooperate usefully, threads must communicate with each other

How do processes and threads communicate?

- Shared Memory
- Message Passing
- Signals

IPC: Shared Memory

Processes

- Each process has private address space
- Explicitly set up shared memory segment within each address space

Threads

- Always share address space (use heap for shared data)

Advantages

- Fast and easy to share data

Disadvantages

- Must **synchronize** data accesses; error prone

Synchronization: Topic for next few lectures

IPC: Message Passing

Message passing most commonly used between processes

- Explicitly pass data btwn **sender** (src) + **receiver** (destination)
- Example: Unix pipes

Advantages:

- Makes sharing explicit
- Improves modularity (narrow interface)
- Does not require trust between sender and receiver

Disadvantages:

- Performance overhead to copy messages

Issues:

- How to name source and destination?
  - One process, set of processes, or mailbox (port)
- Does sending process wait (I.e., block) for receiver?
  - Blocking: Slows down sender
  - Non-blocking: Requires buffering between sender and receiver

IPC: Signals

Signal

- Software interrupt that notifies a process of an event
- Examples: SIGFPE, SIGKILL, SIGUSR1, SIGSTOP, SIGCONT

What happens when a signal is received?

- Catch: Specify signal handler to be called
- Ignore: Rely on OS default action
  - Example: Abort, memory dump, suspend or resume process
- Mask: Block signal so it is not delivered
  - May be temporary (while handling signal of same type)

Disadvantage

- Does not specify any data to be exchanged
- Complex semantics with threads

Threads and Signals

Problem: To which thread should OS deliver signal?

Option 1: Require sender to specify thread id (instead of process id)

- Sender may not know about individual threads

Option 2: OS picks destination thread

- POSIX: Each thread has signal mask (disable specified signals)
- OS delivers signal to all threads without signal masked
- Application determines which thread is most appropriate for handing signal