Today

- Threads
  - Multithreaded programming
  - Thread Pools
- Concurrency Problems

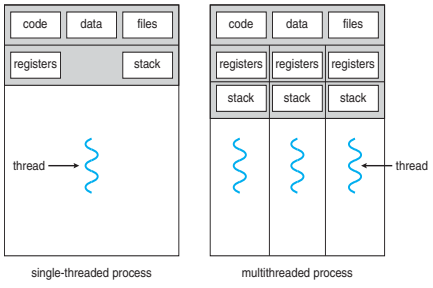
Review

- How are threads similar to yet different from processes?

Review: Threads vs Processes

- Threads executing within the same process share *most* of their address space.
- All threads in a process share the same:
  - Code segment
  - Data segment
  - Heap
- Each thread must have its own:
  - Program counter
  - Register values
  - Stack segment (i.e., local variables and parameters)

Single and Multithreaded Processes



Shared Address Space Code Example

Consider: how are threads and processes with shared memory different?

Multithreading vs Alternatives

- Anything that can be done with a multithreaded program can also be done:
  - With a single-threaded program
  - With cooperating processes and IPC

How will the multithreaded version compare to these alternatives?

Multithreading Efficiency

- Compared to a single-threaded version of the same program, a multithreaded version may exhibit
  - better responsiveness
  - improved performance
- Compared to an implementation using cooperating processes, a multithreaded implementation will be
  - *more economical* in terms of system resource usage
  - *more efficient* in terms of execution speed
    - Creation
    - Context Switching
    - Communication

Multicore Programming

- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation
  - Usually implement hybrid of these
- As # of threads grows, so does architectural support for threading

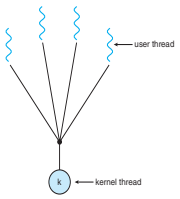
MULTITHREADING MODELS

User Threads and Kernel Threads

- User threads - management done by user-level threads library, without kernel support
  - Three primary thread libraries:
    - POSIX Pthreads
    - Java threads
    - Windows threads
- Kernel threads - Supported by kernel
  - Virtually all general-purpose operating systems, including Windows, Solaris, Linux, Mac OS X, …
- Need a relationship between user and kernel threads

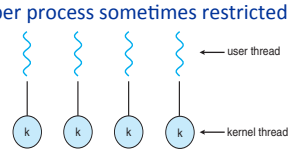
Many-to-One

- 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
- Can't take advantage of multiple cores → few systems currently use this model



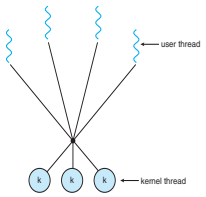
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Kernel thread creation is expensive
  - Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows, Linux
  - Solaris 9 and later



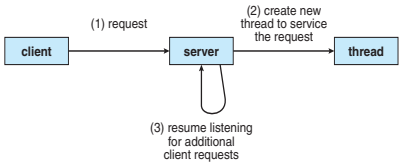
Many-to-Many Model

- Allows many user-level threads to be mapped to many kernel threads
  - # user >= # kernel
- Allows the operating system to create a sufficient number of kernel threads
- Variation: two-level model
  - Some user threads are matched to kernel thread (one-to-one)



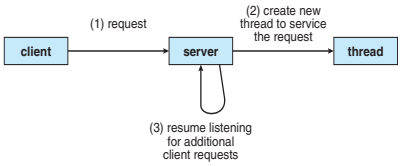
IMPROVING EFFICIENCY

Multithreaded Server Architecture



What happens under a large load?

Multithreaded Server Architecture

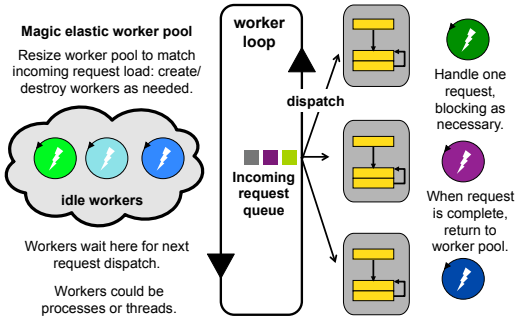


- Under large load, resources used to spawn threads (requires memory and CPU resources)
- Could have _lots_ of threads under huge load
  - Thrashing/competition for resources

Thread Pools

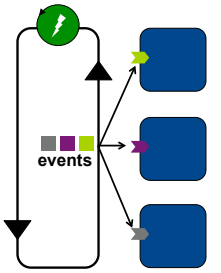
- Create a number of threads in a pool where the threads await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - Tasks could be scheduled to run periodically

Thread pool: idealized



Event-driven programming

- Event-driven programming is a design pattern for a thread's program.
- The thread receives and handles a sequence of typed *messages* or *events*.
  - Handle one event at a time, in order.
- In its pure form, the thread never blocks, except to wait for the next event.
  - Blocks only if no events to handle (idle).
- Program is like a set of handler routines for the event types.
  - The thread upcalls the handler to dispatch or "handle" each event.
- A handler should not block: if it does, the thread becomes unresponsive to events.



Dispatch events by invoking handlers (upcalls).

But what's an "event"?

- A system can use an event-driven design pattern to handle any kind of asynchronous event.
  - Arriving input (e.g., GUI clicks/swipes, requests to a server)
  - Notify that an operation started earlier is complete
    - E.g., I/O completion
  - Subscribe to events published/posted by other threads
  - Including status of children: stop/exit/wait, signals, etc.
- You can use an "event" to represent any kind of message that drives any kind of action in the receiving thread.
- But the system must be designed for it, so that operations the thread requests do not block
  - The request returns immediately ("asynchronous") and delivers a completion event later.

Events vs. Threading

- Classic Unix system call APIs are blocking
  - Requires multiple processes/threads to build responsive/efficient systems.
- Kernel networking and I/O stacks are mostly event-driven
  - interrupts, callbacks, event queues, …
- Some system call APIs may be non-blocking
  - Ex: asynchronous I/O
  - notify thread by an event when operation completes
- Modern systems combine events and threading
  - Event-driven model is natural for GUIs, servers.
  - But to use multiple cores (every modern system) effectively, we need multiple threads.
  - Multi-threading also enables use of blocking APIs without compromising responsiveness of other threads in the program.

SEDA paper
Swing concurrency

THREAD CHALLENGES

Signal Handling

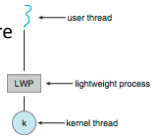
- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers:
    - default
    - user-defined
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

Signal Handling

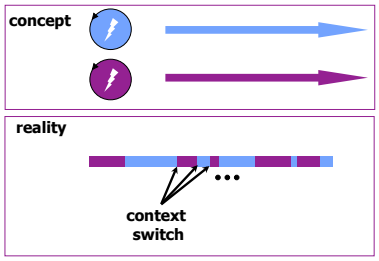
- Where should a signal be delivered for multi-threaded?
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- Option chosen depends on the signal

Scheduler Activations

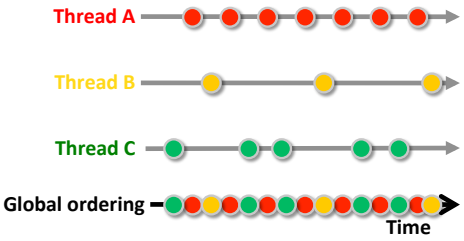
- Some multithreading models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – ***lightweight process (LWP)***
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide ***upcalls***
  - a communication mechanism from the kernel to the upcall handler in the thread library



Two threads sharing a CPU

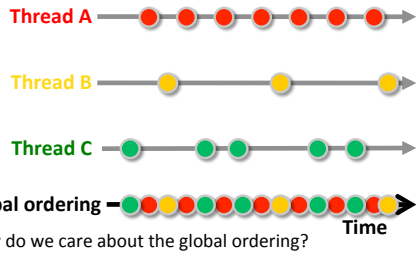


Non-determinism and ordering



- Why do we care about the global ordering?
- Why is this ordering unpredictable?

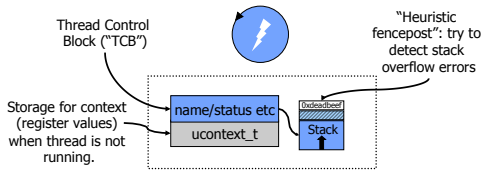
Non-determinism and ordering



- Why do we care about the global ordering?
  - Might have **dependencies** between events
  - Different orderings can produce different results
- Why is this ordering unpredictable?
  - Can't predict how fast processors will run

Portrait of a thread

Each thread is represented by a data struct.
We call it a "**thread object**" or "**Thread Control Block**".
It stores information about the thread, and
may be linked into other system data structures.



Each thread also has a runtime stack for its own use. As a running thread calls procedures in the code, frames are pushed on its stack.

Non-determinism example

- `y=10;`
- Thread A: `x = y+1;`
- Thread B: `y = y*2;`
- Possible results?
  - A goes first: x = 11 and y = 20
  - B goes first: y = 20 and x = 21
- Variable y is shared between threads.

Another example

- Two threads (A and B)
  - A tries to increment i
  - B tries to decrement i

```
i = 0;

Thread A:
while (i < 10){
    i++;
}
print "A done."

Thread B:
while (i > -10){
    i--;
}
print "B done."
```


Looking Ahead

- Project 2 due Wednesday
- Reading Chapter 4