ADVANCED OPERATING SYSTEMS

Threads, Duality, SEDA

An introduction to Programming with Threads (1989)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

An introduction to Programming with Threads

- What kind of paper is this?
  - A tutorial

Two key OS functions

- Multiplex resources
  - Allow multiple users or programs to use the same set of hardware resources (processors, memory, disks, network connection) **safely and efficiently**
- Provide abstractions

Operating System Abstractions

- Abstractions simplify use of the hardware by applications
- Abstractions *simplify* application design by
  - Hiding undesirable properties
  - Adding new capabilities
  - Organizing information

Example Abstraction: file

- What undesirable properties do files hide?
  - Disks are slow!
  - Chunks of storage are actually distributed all over the disk
  - Disk storage may fail!
- What new capabilities do files add?
  - Growth and shrinking
  - Organization into directories
- What information do files help organize?
  - Ownership and permissions
  - Access time, modification time, type, etc.

Abstractions

- Files abstract the disk
- Address spaces abstract what?

Abstractions

- Files abstract the disk
- Address spaces abstract memory
- Threads abstract what ?

Abstractions

- Files abstract the disk
- Address spaces abstract memory
- Threads abstract the CPU
- Sockets abstract what?

Abstractions

- Files abstract the disk
- Address spaces abstract memory
- Threads abstract the CPU
- Sockets abstract the network
- What about processes?

The Process

- Processes are the most fundamental operating system abstraction
  - processes organize information about other abstractions and represent a single thing that the computer is "doing"
  - You know processes as applications

Organizing information

- Unlike threads, address spaces, and files, processes are **not tied to a hardware component.** Instead, they contain other abstractions
- Processes contain
  - One or more threads
  - An address space, and
  - Zero or more open file handles representing files

Back to Birrell's threads tutorial

- What is a multi-threaded program?

Back to Birrell's threads tutorial

- A multi-threaded program has multiple points of execution, one in each of its threads
- Threads execute within a single address space
- How are the following shared between threads?
  - Registers?
  - Stack?
  - Memory?
  - File descriptor table?

Back to Birrell's threads tutorial

- A multi-threaded program has multiple points of execution, one in each of its threads
- Threads execute within a single address space
- How are the following shared between threads?
  - Register values not shared
  - Stack not shared
  - Memory – global variables, heap, code are shared
  - File descriptor table shared

Why use threads?

- Take advantage of multiprocessors
- Do something useful while waiting for a slow device (disks, network, etc)
- To interact with humans in a non-sluggish manner
- Defer work to improve responsiveness of a program
  - Push non-urgent work (e.g., rebalancing a red-black tree post update) to another thread
- Build a networked server that serves incoming clients in parallel

Threads share global memory

- Shared memory must be protected via mutual exclusion mechanism
  - Mutex: a resource scheduling mechanism
    - Resource being scheduled is shared memory
    - Scheduling policy is: one thread at a time
  - Difference between mutexes, locks, and semaphores?

Condition variables

- Mutex: a resource scheduling mechanism
  - Resource being scheduled is shared memory
  - Scheduling policy is: one thread at a time
  - What happens if programmer needs a more complicated scheduling policy?

Condition variables

- Example: thread A reads data off network, places in shared buffer; thread B reads data from buffer and processes it
- Does the example below work correctly?

Thread A loop
1) Read data
2) Lock mutex
3) Add new data to buffer
4) Unlock mutex
5) Return to step 1

Thread B loop
1) Lock mutex
2) If buffer not empty
    1) Read data from buffer
3) Unlock mutex
4) Return to step1

Condition variables

- Example: thread A reads data off network, places in shared buffer; thread B reads data from buffer and processes it
- Does the example below work correctly?
  - Works fine but depletes resources
  - Potential for busy-waiting, why?
- Condition variables provide mechanisms for programmers to express that a thread must block until particular event occurs

Thread A loop
1) Read data
2) Lock mutex
3) Add new data to buffer
4) Unlock mutex
5) Return to step 1

Thread B loop
1) Lock mutex
2) If buffer not empty
    1) Read data from buffer
3) Unlock mutex
4) Return to step1

Condition variables

- Condition variables in combination with mutexes can implement any thread scheduling policy desired
- Condition variable always associated with a particular mutex and with the data protected by that mutex
- Monitor consists of
  - Data
  - A mutex
  - 0 or more condition variables
  - Will come back to monitors in the future…

Example: shared buffer

- Example: thread A reads data off network, places in shared buffer; thread B reads data from buffer and processes it
- How to avoid busy-waiting?
  - Introduce condition variables
  - What are the semantics of wait() call?
  - Why *while* instead of *if* in condition check?

```
Thread A loop
Lock M  {
   while (buffer is full) wait(M, non-full);
   read data
   signal non-empty
Unlock M
```

```
Thread B loop
Lock M {
    while (buffer is empty)
        wait(M, non-empty);
     read data
     signal non-full
Unlock M
```

Paper provides guidelines for multi-threaded programming

- Think of mutex as protecting the *invariant* of the associated data
- Beware of unsynchronized access to shared data
  - Can result in bizarre results depending on real-time scheduling of the threads
  - What kinds of simple things could a compiler do to support multi-threaded programming?

Paper guidelines (cont'd)

- Fine-grained vs course-grained locking
  - If you have one big lock for all global data,
    - Simpler, but
    - What problem does it cause?
  - If you have lots of fine-grained locking
    - More concurrency, but
    - What problems does it cause?
- Avoid deadlocks
  - How?
- What is priority inversion?

Paper guidelines (cont'd)

- Signal versus broadcast
  - When to use each?

Paper guidelines (cont'd)

- Signal versus broadcast
  - Signal useful if you know that at most one thread (and any of the threads) waiting on the condition variable can make progress
  - Broadcast useful when you don't know
    - Can I use signal and broadcast interchangeably?
    - Can I replace signal() with broadcast() call or vice versa? What is the effect?
- Spurious lock conflicts
  - What is effect of moving signal or broadcast outside the lock clause?

Paper guidelines (cont'd)

- Separate blocked threads onto different condition variables if their functions are different (e.g., multiple readers waiting to read/single writer waiting to write)
- What is starvation?

Paper guidelines (cont'd)

- Separate blocked threads onto different condition variables if their functions are different (e.g., multiple readers waiting to read/single writer waiting to write)
- What is starvation?
  - Some thread never makes progress
- Nested monitor problem
  - What is this?

Paper guidelines (cont'd)

- Nested monitor problem
  - "Risky to call into a lower level abstraction while holding a mutex without knowing exactly what the lower level abstraction does"
    - Could lead to deadlock
    - Modularity and APIs versus deadlock avoidance
- Add threads to defer work until later
  - Adv: E.g., Procedure returns more quickly as soon as result is ready
  - Disadv: creating a thread each time you defer work may be costly on uniprocessor, but encouraged on multiprocessor…. up to a point

Paper guidelines (cont'd)

- Consider pipelining to speed up your application
  - Almost linear speed-up possible
  - Is pipelining useful on a uniprocessor?
    - Yes, if each thread will "encounter some real-time delays (such as page faults, device handling or network communication"
    - Pipelining similar to SEDA concept
- Beware of creating too many threads!
- To really make your program efficient, need performance profiling
  - Statistics on lock conflicts: how often threads block on a mutex, how long they wait, etc.

On the Duality of Operating System Structures (1979)

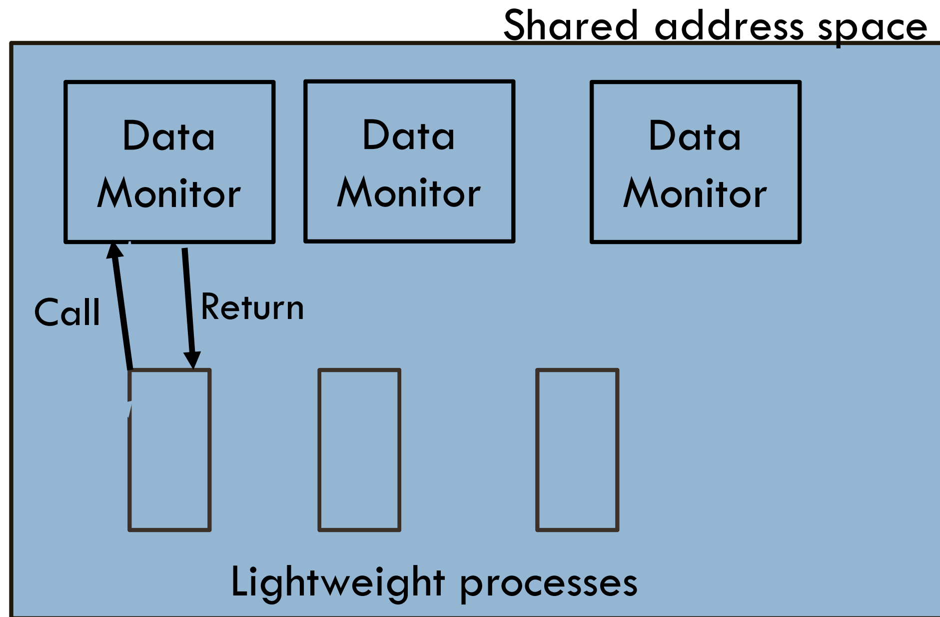
- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

On the Duality of Operating System Structures (1979)

- What kind of paper is this?
  - A position paper

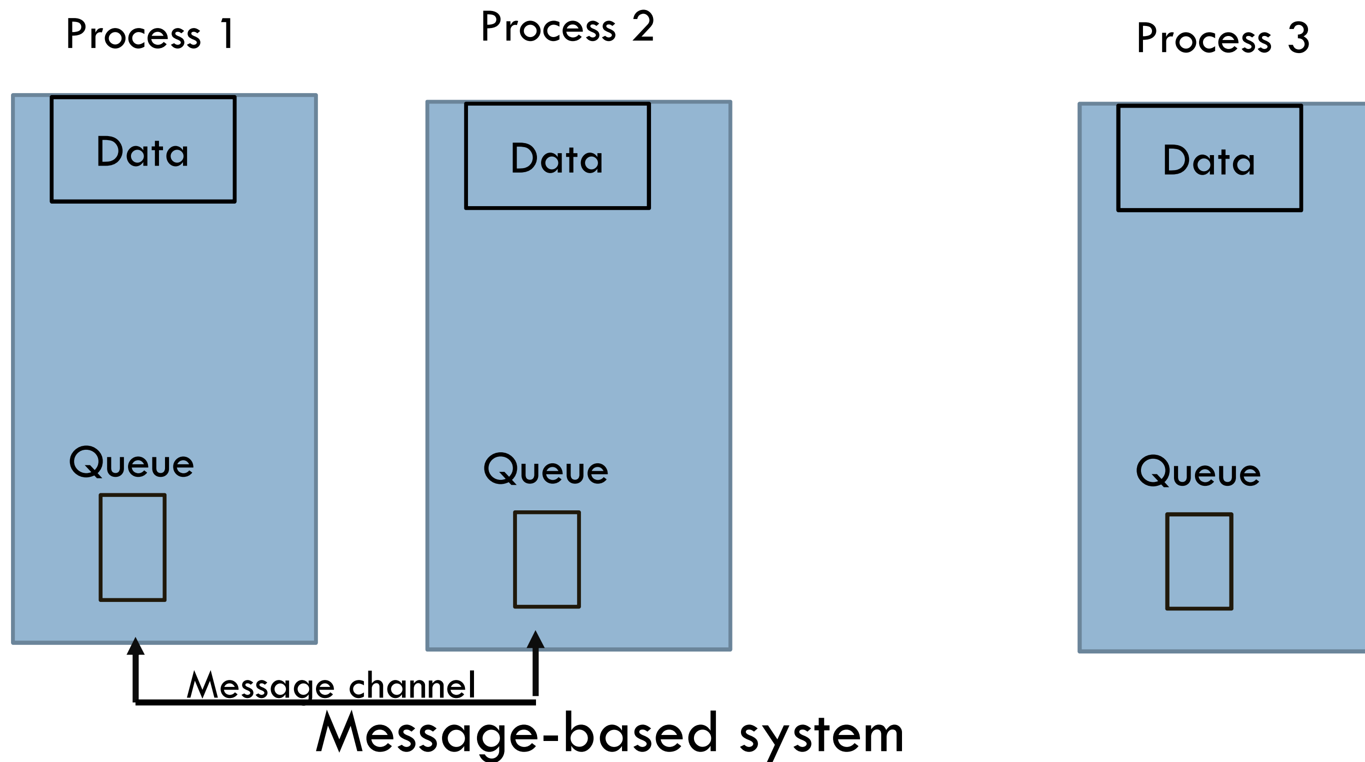
Controversy

- Message-based versus monitor-based systems
- Paper's claim:
  - In power
  - In performance
- Controversy still exists



Procedure-based system

- Process corresponds to task rather than resource
- Resource manager is a shared data structure along with synchronized procedures to manage it
- Locks are associated with data structures, data is shared between processes
- Module structure is relatively static



Message-based system

- Resource manager is a process that receives requests over message channels
- Inner loop = get message, parse, reply, get another message
- Synchronization handled with message channels
- Process structure is relatively static
- "Task" passes from process to process through message channels

Duality mapping: structures are semantically equivalent

- Process
- Message channel
- Send msg + wait reply
- Send msg…wait reply
- Send reply
- Selective message wait

- Monitor
- Entry procedure name
- Procedure call
- Async procedure call
- Return from procedure
- Condition variables

Claim: Neither is more powerful

- Most PLs accommodate procedures more easily than messages
  - Type-checking
  - Synchronization mechanism/support

Claim: Performance is also equivalent

- Debatable
- Procedure call is faster than message exchange
  - Call time << context switch time
  - Factor 100-1000 times faster
- Asynchronous message is easier than async call
- What happens under contention?
  - Monitors – Queue up on locks
  - Messages – Queue up in message channels

SEDA: An Architecture for Well-Conditioned Scalable Internet Services

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?
  - Something else?

Examples of event-driven-like programming

- Unix signal handler
- Select/poll/epoll call
- GUIs -- Javascript, node.js, etc
- Asynchronous I/O?

Events versus Threads (over) simplified

- Threads can block, so we make use of the CPU by switching between threads
- Event handlers cannot block, so we can make use of the CPU by simply running events to completion
  - We must write handlers so they are so small they don't ever block, and if they need to block, they create an event that someone else will have to handle

Events versus Threads (over) simplified

- With threads, getting good concurrency relies on switching between them
- With event-based programming, getting good concurrency relies on writing events in a way that I can process a lot of them in a row really quickly and keep the system busy