Frequently asked questions from the previous class survey

- Why use the Berkeley algorithm?
- What are the times that are exchanged in Berkeley and Cristian’s algorithm?
- Why should N > 3f where f is number of faulty clocks?
- Logical clocks and the banking transaction example
  - Why should all processes acknowledge?
  - Are logical clocks more accurate?
  - Are there skews and drifts for logical clocks?
- Offset calculation in wireless settings?
  - A and B know their offsets from K; so know their offsets from each other. How many K’s are needed? What does this achieve?
- Size disadvantages for Vector and Matrix clocks?
  - Is this why folks still use Lamport’s clocks?

Topics covered in this lecture

- Vector clocks
- Causally ordered multicasting using vector clocks
- Threads
  - Rationale
  - Implications for throughput at a server

Vector clocks

- Developed by Mattern [1989] and Fidge [1991] to overcome shortcomings of Lamport’s clocks
  - i.e. if C(a) < C(b) then we cannot conclude a \( \rightarrow \) b
  - A vector clock for a system of N processes is an array of N integers
  - Each process keeps its own vector clock VC_i
  - Process uses it vector clock to timestamp messages

Causality can be captured by Vector clocks

- Event a is known to causally precede event b iff VC(a) < VC(b)
  - VC(a) < VC(b) iff VC(a)[k] \leq VC(b)[k] for all k and at least one of those relationships is strictly smaller
- Each process P_i maintains a vector VC_i
  - VC_i[k] is number of events so far at P_i
  - If VC_i[k] = k
    - P_i knows k events occurred at P_i
    - P_i’s knowledge of local time at P_i

Vectors are piggybacked along with any messages that are sent

1. Before executing an event (sending, delivering, or internal) P_i executes
   - VC_i[j] = VC_i[j] + 1
2. When P_i sends a message m to P_j
   - Set m’s timestamp ts(m) to VC_i after doing (1)
3. After receiving m, process P_j adjusts its vector
   - VC_j[k] = max{VC_j[k], ts(m)[k]} for each k
   - Execute step (1) and deliver
Vector clocks example 1

A
[1,0,0] [2,0,0]
B
[2,1,0] [2,2,0]
C
[0,0,1] [2,2,2]

Vector clocks example 2

A
[1,0,0] [5,4,0] [7,4,4]
B
[1,3,0] [1,4,0] [5,4,0] [7,4,4]
C
[1,3,3] [1,3,4] [5,4,0] [7,4,4]

Vector timestamps allow us to determine causality and concurrency

- Event \( a \) happened before event \( b \) iff
  - \( ts(a) \leq ts(b) \) for each process \( i \)
  - And one of those relationships is strictly smaller
- If this is not true
  - Events \( a \) and \( b \) are concurrent

Vector Clocks: Other aspects

- If event \( a \) has timestamp, \( ts(a) \):
  - \( ts(a)[i] - 1 \)
    - Denotes number of events at \( P_i \) that precede \( a \)
- When \( P_j \) receives message \( m \) from \( P_i \) with timestamp \( ts(m) = VC_i \):
  - \( P_j \) knows about number of events at \( P_i \) that causally preceded \( m \)
  - Also, \( P_j \) knows about how many events at other processes have preceded the sending of \( m \), and on which \( m \) may causally depend

Vector clocks: Disadvantages

- Storage and message payload is proportional to \( N \), the number of processes
- It’s been shown ([Charron-Bost 1991]) that if we are to tell if two events are concurrent by inspecting timestamps?
  - The dimension of \( N \) is unavoidable

Using Vector Clocks for Causally Ordered Multicasting
Contrasting totally-ordered and causally-ordered multicasting

- Causally-ordered multicasting is weaker than totally-ordered multicasting
- If two messages are not in any way related to each other?
  - We do not care about the order in which they are delivered to applications
  - Could be delivered in different order at different applications

Using Vector Clocks for causally-ordered multicasting

- Clocks are only adjusted when sending and receiving messages
- Upon sending a message, process $P_i$ will only increment $VC_i[i]$ by 1
- When $P_i$ delivers a message $m$ with timestamp $ts(m)$ it only adjusts $VC[k]$ to $\max(VC[k], ts(m)[k])$ for each $k$

When process $P_j$ receives a message $m$ from $P_i$

- Delivery of the message $m$ to the application layer is delayed until 2 conditions are met:
  1. $ts(m)[i] = VC_j[i] + 1$
     - This means $m$ is the next message that $P_i$ was expecting from $P_j$
  2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$
     - This means that $P_j$ has seen all messages that have been seen by $P_i$ when it receives $m$

An example showing enforcement of causal communications

Matrix clocks

- Generalizes the notion of vector clocks
- Processes keep estimates of other processes’ vector time [Raynal & Singhal, 1996]
- Essentially, a vector of vector clocks for each of the communicating processes
A quick look at threads and processes

- From an Operating Systems (OS) perspective
  - Management and scheduling of processes are the most important issues to deal with
- When it comes to distributed systems
  - Threads are highly important
    - Overlap processing and I/O
    - Asynchronous communications

The evolution of Operating Systems

- Computers didn’t have OSes
- Single program executed from start to finish
  - On bare metal
- Running only a single program was inefficient
  - Resources were scarce and expensive
- OS evolved to allow more than one program to execute in processes

Processes are isolated, independently executing programs

- Often defined as a program in execution
- The OS ensures that processes, inadvertently or maliciously:
  - Cannot affect the correctness of each other
- Processes may share the same CPU, memory, and other resources
  - But this concurrent sharing is transparent

Concurrency transparency comes at a high price

- Each time a process is created the OS creates an independent address space
  - Allocation involves initializing memory segments
    - Zeroing the data segment
    - Copying program into text segment
    - Setting up a stack (temporary data)
  - Switching CPU between processes is expensive

A process in memory

Function parameters, return addresses, and local variables

Memory allocated dynamically during runtime

Global variables

Program code

A note on the program in memory

Program image appears to occupy contiguous blocks of memory
- OS maps programs into non-contiguous blocks
- Mapping divides program into equal-sized pieces: pages
- OS loads pages into memory
- When processor references memory on page
  - OS looks up page in table, and loads into memory
Advantages of the mapping process

- Allows large logical address space for stack and heap
- No physical memory used unless actually needed
- OS hides the mapping process
  - Programmer views program image as logically contiguous
  - Some pages may not reside in memory

Process state transition diagram: When a process executes it changes state

- New
- Ready
- Running
- Waiting
- I/O or event wait
- I/O or event completion
- Exit
- Interrupt
- Terminated

Each process is represented by a process control block (PCB)

- Process state
- Process number
- Program counter
- Registers
- Memory limits
- List of open files

PCB is a repository for any information that varies from process to process.

An example of CPU switching between processes

- Process A
- Operating System
- Process B

Speed of the context switch depends on

- Memory speed
- Number of registers to copy
- Special instructions for loading/storing registers
- Memory management: Preservation of address spaces

Broad breakdown of switching costs

- CPU context
  - Register values, program counter, stack pointer, etc
- Memory management
  - Modify memory management registers
  - Invalidate address translation caches
    - Translation lookaside buffer (TLB)
  - Swapping related to paging
    - Move pages between memory and disk: expensive!
Processes form a building block of distributed systems, but …

- Granularity provided by processes is insufficient
- Multiple threads of control per process
  - Easier to build distributed systems
  - Better performance

Threads execute their own piece of code independently of other threads, but …

- No attempt is made to achieve high-degree of concurrency transparency
  - Especially, not at the cost of performance
- Only maintains information to allow a CPU to be shared among several threads
- Thread context
  - CPU Context + Thread Management info
  - List of blocked threads

Information not strictly necessary to manage multiple threads is ignored

- Protecting data against inappropriate accesses by multiple threads in a process?
  - Developers must deal with this

A process with multiple threads of control can perform more than 1 task at a time

- Interactive multithreaded application
  - Parts of program may be blocked or slow
  - Remainder of program may still chug along
- A single threaded process can ONLY run on 1 processor
  - Regardless of how many are available
  - Underutilization of compute resources

Implications?

- Performance of a multithreaded application is seldom worse than a single threaded one
  - Actually leads to performance gains
- Development requires additional effort
  - No automatic protection against each other

Thread use in non-distributed settings

- Interactive multithreaded application
  - Parts of program may be blocked or slow
  - Remainder of program may still chug along
- A single threaded process can ONLY run on 1 processor
  - Regardless of how many are available
  - Underutilization of compute resources
Another drawback of processes is the overheads for IPC.

Switch from user space to kernel space

Switch from kernel space to user space

Applications can be constructed using separate threads

- Communications dealt entirely using shared data
- Performance is much better
- Software engineering
  - Collection of several (generally independent) tasks
  - Word Processor
    - Input handling, spell check, layout, index generation...

Summarizing threading models

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

Summarizing threading models

- Many-to-Many
- Two-level

An example of performance improvements with threads

Client and Server with Threads

- Client
- Request Queue
- Server
- Server may have up to N threads
- Requests

Instructor: SHRIDEEP PALLICKARA

September 3, 2015

CS555: Distributed Systems [Fall 2015]
Dept. Of Computer Science, Colorado State University
**Server side processing**

- Server has **queue** of requests received from clients
- Server also has a **pool** of one or more threads
  - Each thread repeatedly removes requests & processes it
- Each thread applies the same methods to process the requests
  - Each request takes 2 ms of processing PLUS 8 ms of I/O (when server reads from disk i.e. no caching)

**Maximum server throughput with 1 thread**

- The turnaround time for handling any request is $2+8 = 10$ ms
- The server can handle 100 requests per second
- Any new requests that arrive while the thread is handling a request?
  - These will be queued

**Server throughput with 2 threads**

- We assume that the threads are independently schedulable
  - One thread can be scheduled while the other is blocked for I/O
- Thread T2 can process a second request when thread T1 is blocked, and vice versa
- This increases throughput ... but both threads may be blocked for I/O on the single disk drive
- If all I/O requests are serialized and take 8 ms each?
  - Maximum throughput is $1000/8 = 125$ requests/second

**Server throughput with disk block caching**

- Server keeps data that it reads in buffers
- When a server thread tries to retrieve data
  - It first examines the cache and avoids disk accesses if it finds data element there
- If the hit rate is 75%?
  - The mean I/O time per-request reduces to $(0.75 \times 0 + 0.25 \times 8) = 2$ milliseconds
- Maximum theoretical throughput?
  - Becomes 500 requests per second

**But there are costs associated with caching**

- Average processor time for a request increases
  - This is because it takes time to search for cached data for every operation
  - Let us assume that this is now 2.5 milliseconds
  - The server can now handle $1000/2.5 = 400$ requests per second i.e. 400

**Let’s look at caching plus multiple threads**

- Each request takes about $2.5$ (processing) + $2$ (I/O)
  - Total time per request is now $4.5$ mSecs when disk accesses are serialized
  - Each thread can do $1000/4.5$ requests per second i.e. 222 requests/second
- With two threads?
  - 444 requests/second
- With three threads?
  - 500 requests (bound by the I/O time)
The contents of this slide-set are based on the following references

- http://en.wikipedia.org/wiki/Matrix_clocks