P56

Third Lecture (Threads Locks Logs)

Friday 15:00-17:00 SE02

Andrew.W.Moore@cl.cam.ac.uk

Disk Reminder

- **Home directory** (‘root’ aka /) and **/home/l51** is a micro SD
  rated at 170MB/s Read rate

- **/flash** is a Samsung FIT Plus 256 GB Type-A 300 MB/s USB 3.1 Flash

- On one in four pi has a /**nvme**
  a SanDisk 1TB Extreme PRO Portable SSD, USB 3.2 Gen 2x2
  External NVMe Solid State Drive up to 2000 MB/s

- l51-pi046:**/hdd5400** 5400RPM 320GB SATA / USB

- l51-pi047:**/hdd7200** 7200RPM 500GB SATA / USB

The Four Fundamental Hardware Resources

CPU

Memory

Disk/SSD

Network

Five

The ~~Four~~ Fundamental ~~Hardware~~ Resources

CPU

Memory

Disk/SSD

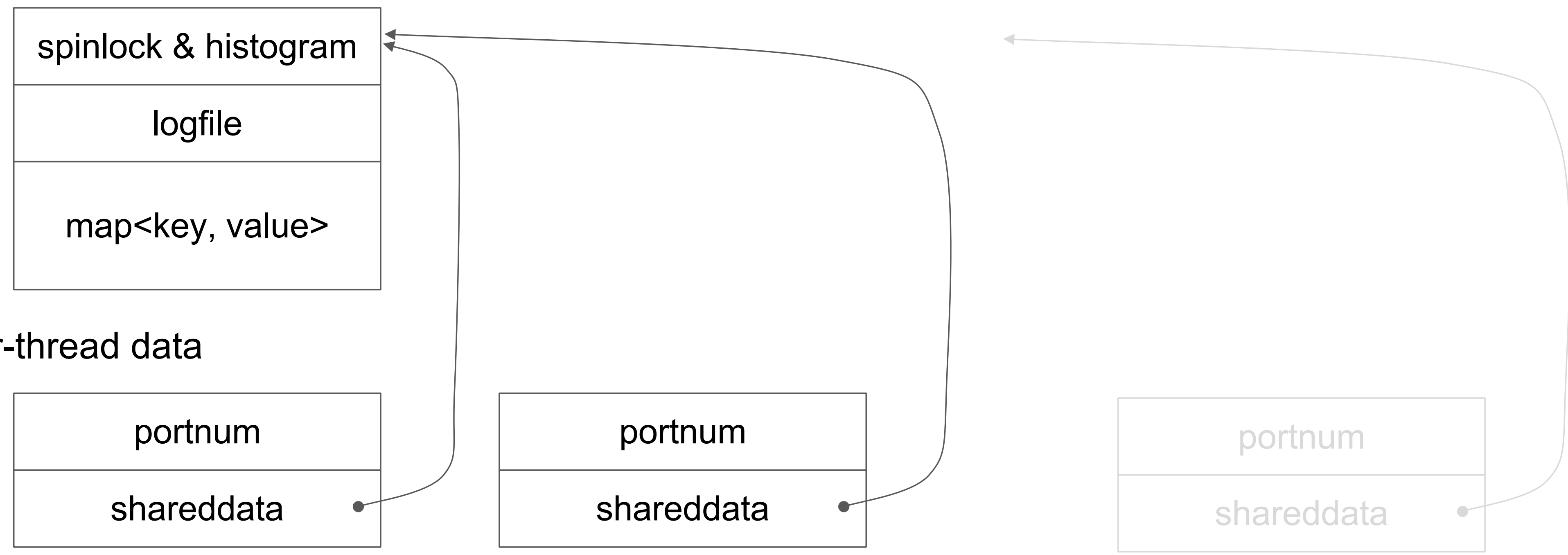
Network

Critical software sections (locks)

Threads, locks

Multiple server threads

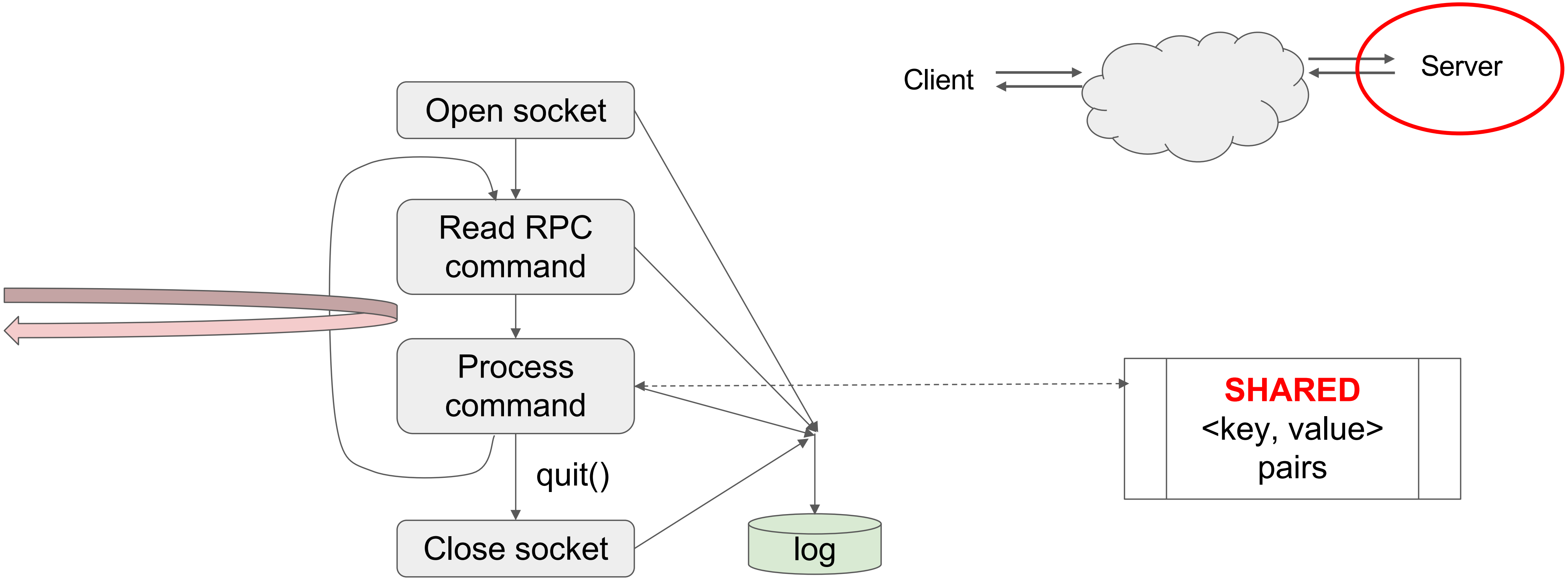
Shared data



Multiple server threads

```
void* SocketLoop(void* arg) {
  PerThreadData* ptd = reinterpret_cast<PerThreadData*>(arg);
  open socket
    …
  close socket
}
for (int i = 0; i < n; ++i) {
        Build per-thread data, perthreaddata
        pthread_create(&thread, &attr, SocketLoop(), (void*) &perthreaddata);
}
```

RPC simple server4



Software locks

```
void* SocketLoop(void* arg) {
  PerThreadData* ptd = reinterpret_cast<PerThreadData*>(arg);
  open socket
  ...
            for(;;) {
                        Receive request; Log request [2]
                        Process request uses shared data
                        Send response; Log response [3]
            }
    …
  close socket
}
```

Software locks

```
do {
  old_value = __atomic_test_and_set(&lock)
} while (old_value != 0)
```

Thread A	0=>1	...have lock...			1=>0				
Thread B		1=>1	1=>1	1=>1		0=>1	...have lock…		
Thread C			1=>1	1=>1			1=>1	1=>1	etc.

Software locks, cache thrashing

```
do {
  old_value = __atomic_test_and_set(&lock)
} while (old_value != 0)
```

Thread A   0=>1   ...have lock...   **1=>0**
Thread B   **1=>1**   **1=>1**   **1=>1**   **0=>1**   ...have lock…
Thread C   **1=>1**   **1=>1**   **1=>1**   1=>1   etc.

Writing inside the loop causes **cache thrashing**

Software locks

```
do {
  while (lock != 0) {}    // Spin here mostly
  old_value = __atomic_test_and_set(&lock)
} while (old_value != 0)
```

Thread A   0=>1   ...have lock...   **1=>0**
Thread B   **1** 1 1 1 1 1 1 1 1   **0** **0=>1** ...have lock…
Thread C   **1** 1 1 1 1 1 1 1 1   **0**   **1=>1** 1 1 1 etc. (‘missed’)

Software locks, yielding

```
do {
  while (lock != 0) {🔴}    // Spin here mostly
  old_value = __atomic_test_and_set(&lock)
while (old_value != 0)
```

Thread A   0=>1   ...have lock...   **1=>0**

Thread B   1 1 1 1 1 1 1 1 1 ...

Thread C   1 1 1 1 1 1 1 1 1 ...

What happens if there are only two cores, and Thread A is not running?

Enter the secret world of lock and priority inversion…..

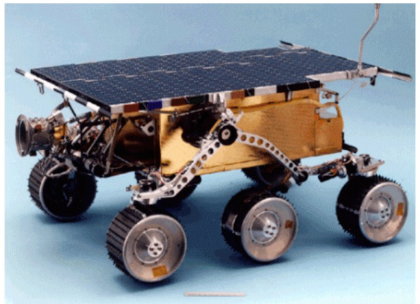
Mars Pathfinder Incident

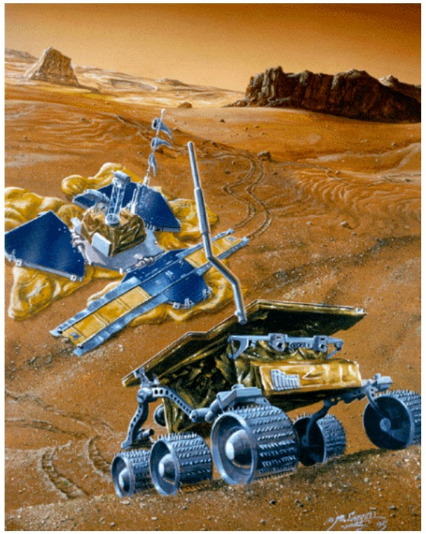
- Landing on July 4, 1997
- "…experiences software glitches…"
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.

- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.

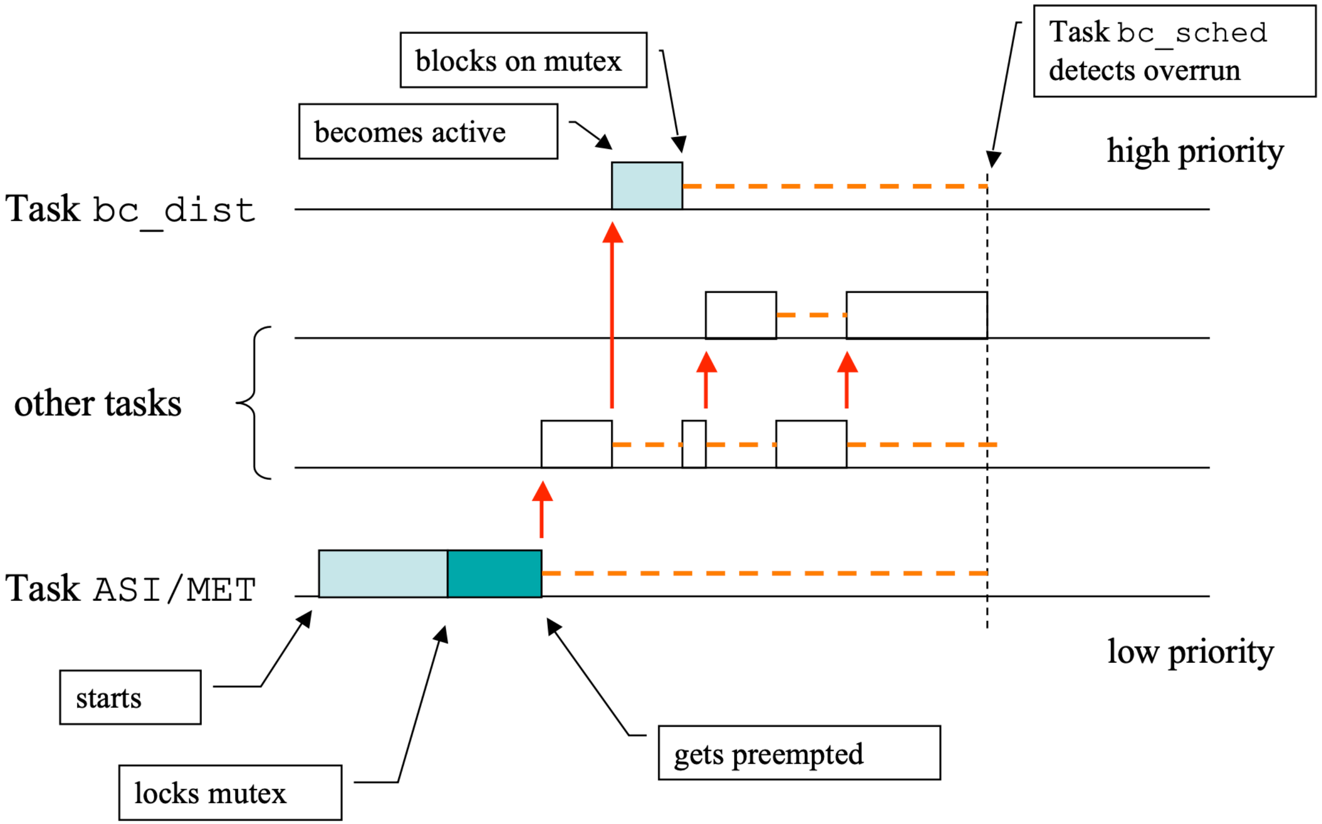
- Resources:

  `research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html`





Priority Inversion on Mars Pathfinder



Software locks, memory barrier

Instructions:

load x

load y

x happens to be in main memory, while y is in the L1 cache. What happens?

1) load y waits for load x to finish 200 cycles later, OR
2) load y is executed without waiting, finishing long before load x
3) if x is the lock acquire, a memory barrier instruction is needed to *force* that y follows x. Same game with two stores at lock release

Software locks, robust critical sections

```
void* SocketLoop(void* arg) {  ...
        Receive request; Log request [2]
Process read(shareddata, request, response) {
        key = f(request data)
        {
                SpinLock sp(shareddata->lockandhist)    // What's This Function?
                it = shareddata->find(key)
                response data = it->value
        }
}
        Send response; Log response [3] ...
  close socket
}
```

Software locks

```
class SpinLock {   // The compiler does the work for us making sure we release
  …
}
SpinLock::SpinLock(&lock) {            // Constructor
          …  __atomic_test_and_set(&lock, _ATOMIC_ACQUIRE)  ...
}
SpinLock::~SpinLock() {                // Destructor
          __atomic_clear(&lock, _ATOMIC_RELEASE)
}
```

Logging

Logging

Both client and server log every transaction: each send and each receive

Log records are binary, fixed-size (96 bytes), written to disk

Designed to be relatively low bandwidth and low overhead

O(10000) RPCs/sec * **O(100) bytes per RPC** = O(1MB/sec)

Capture all the timestamps and the RPCIDs so full dynamics can be reconstructed

Logging

Client:
[1] 20171012_134525.753148 00.000000 00.000000 00.000000 unk:unk 128.178.052.211:12345 0.0 0.0
b6b2abaa 00000000 **ReqSend** ping Success 0

Server:
[2] 20171012_134525.753148 25.753225 00.000000 00.000000 128.178.052.211:41104 128.178.052.211:12345 0.0 0.0
b6b2abaa 00000000 **ReqRcv** ping Success 0
[3] 20171012_134525.753148 25.753225 25.753248 00.000000 128.178.052.211:41104 128.178.052.211:12345 0.0 0.0
b6b2abaa 00000000 **RespSend** ping Success 0

Client:
[4] 20171012_134525.753148 25.753225 25.753248 25.753303 128.178.052.211:41104 128.178.052.211:12345 0.0 0.0
b6b2abaa 00000000 **RespRcv** ping Success 0

Brown: client clock. Blue: server clock. **b6b2abaa** rpcid 00000000 parent id

Histograms

Simple histogram of latencies is **much** more useful than just an average.
Buckets that cover increasing powers of two ranges give tight bound on number of buckets. Thirty-two buckets of, for example, microseconds of latency:

[0..1]
[2..4)
[4..8)
[8..16)
etc.
[2*31 .. 2**32)

Histograms

Simple histogram of latencies is **much** more useful than just an average.

we have a connection from 0100007f:32c5
53.385ms 9.892ms 6.110ms 5.618ms 5.394ms … 95.516ms 6.337ms 5.278ms 3.511ms 3.268ms
...

Histogram of floor log 2 buckets of usec response times

```
Fits in 1 bit          Fits in 11 bits          Fits in 21 bits       Fits in 31 bits
1 2+ 4+ us             1+ 2+ 4+ msec            1+ 2+ 4+ sec          1K+ 2k+ secs
|                      |                        |                     |
0 0 0 0 0 0 0 0 0 0    0 213 284 2 0 1 0 0 0 0  0 0 0 0 0 0 0 0 0 0   0 0
```

500 RPCs, 2228.8 msec, 524.332 TxMB, 524.332 RxMB
224.3 RPC/s (4.458 msec/RPC), 235.3 TxMB/s, 235.3 RxMB/s

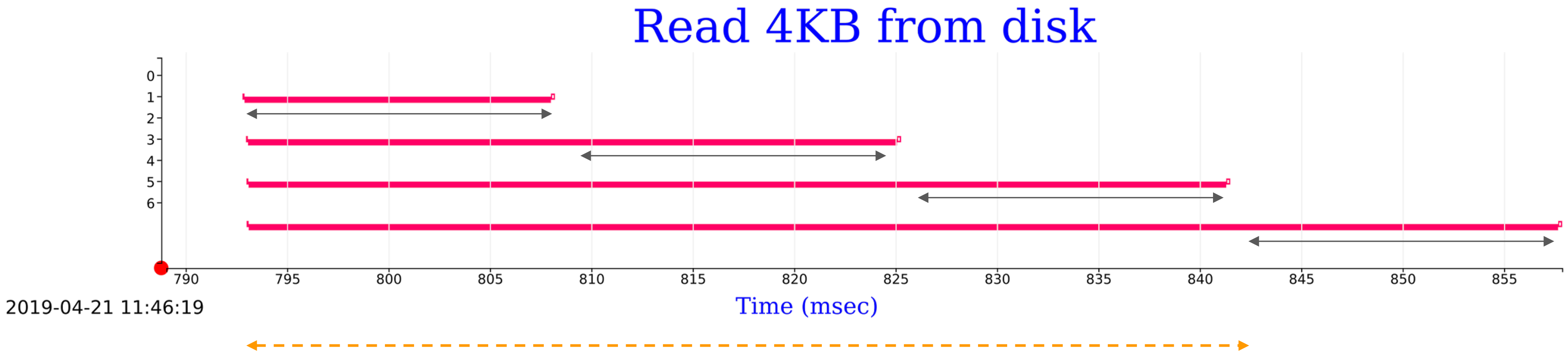
Locks

Locks are the most common source of RPC delays

Holding time is evil
Easy bugs to create: do unneeded work while holding lock
Oversight: no systematic way to detect too-long locks

Solutions:
Build results outside of lock, just swap pointers under lock
Read shared data, leaving note to fail if another updates
Code that uses atomic operations but no locks
Lock library that tracks 90th %ile acquire time
Programmer declares machine-readable expected 90th %ile acquire time;
    library checks

Locks: holding time vs. acquire time



Holding time: black arrows

Acquire time: orange dashed arrow

= holding time * number ahead in queue

Long **acquire** times produce 99th-percentile slow transactions

Software locks: acquire

Spinlock:

```
do {
    while (lock != 0) {}    // Spin here mostly
    old_value = __atomic_test_and_set(&lock)
} while  (old_value != 0)
```

1 nsec to spin again

Blocking lock:

```
do {
    futex(&lock, FUTEX_WAIT, 1);    // Blocks until lock = 0
    old_value = __atomic_test_and_set(&lock)
} while  (old_value != 0)
```

2x 1000 nsec to context switch

Software locks: acquire

Combined lock:

```
        do {
            for() {if (lock==0) break; pause;} // Spin but not too long

            futex(&lock, FUTEX_WAIT, 1);    // Blocks until lock = 0

            old_value = __atomic_test_and_set(&lock)
        } while  (old_value != 0)
  critical_section:
```

~50ns to spin again

2x 1000ns to context switch

Software locks: acquire

Combined lock:

```
do {
    for() {if (lock==0) break; pause;} // Spin but not too long

    futex(&lock, FUTEX_WAIT, 1);    // Blocks until lock = 0

    old_value = __atomic_test_and_set(&lock)
} while  (old_value != 0)
critical_section:
```

~50ns to spin again

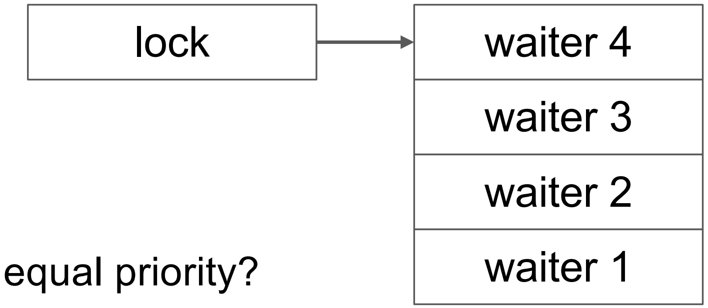
2x 1000ns to context switch

Optimal design: spin for about two context switch times (2-10 usec) then block

Software locks: acquire

Optimal design: spin for about two context switch times (2-10 usec) then block

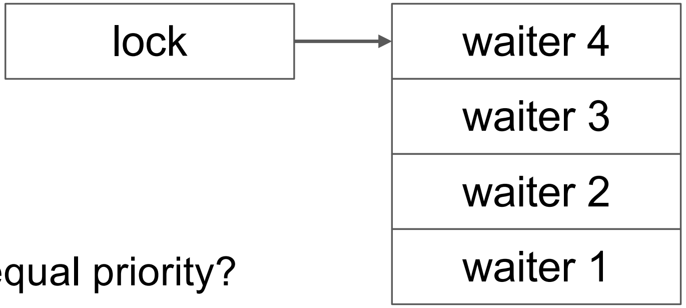
Software locks: release



Which to run next, if all equal priority?

- Strategy 1: make them all runnable
- Strategy 2: make most recent runnable
- Strategy 3: make oldest runnable

Software locks: release



Which to run next, if all equal priority?

- Strategy 1: make them all runnable **cache thrashing, 3 wasted tries**
- Strategy 2: make most recent runnable **starvation of oldest**
- Strategy 3: make oldest runnable better, but want lockless queueing

Locks: track acquire time

Combined lock:

```
        do {
            for() {if (lock==0) break; pause;} // Spin but not too long
            futex(&lock, FUTEX_WAIT, 1);    // Blocks until lock = 0
            old_value = __atomic_test_and_set(&lock)
        } while  (old_value != 0)
   critical_section:
```

Locks: track acquire time

Combined lock:

```
        old_value = __atomic_test_and_set(&lock)
        if (old_value == 0) {goto critical_section}
        // Contended lock
        start = gettimeofday()
        do {
           for() {if (lock==0) break; pause;} // Spin but not too long
           futex(&lock, FUTEX_WAIT, 1);    // Blocks until lock = 0
           old_value = __atomic_test_and_set(&lock)
        } while  (old_value != 0)
        elapsed = gettimeofday() - start   // Use to update histo after freeing lock
  critical_section:
```

Time alignment

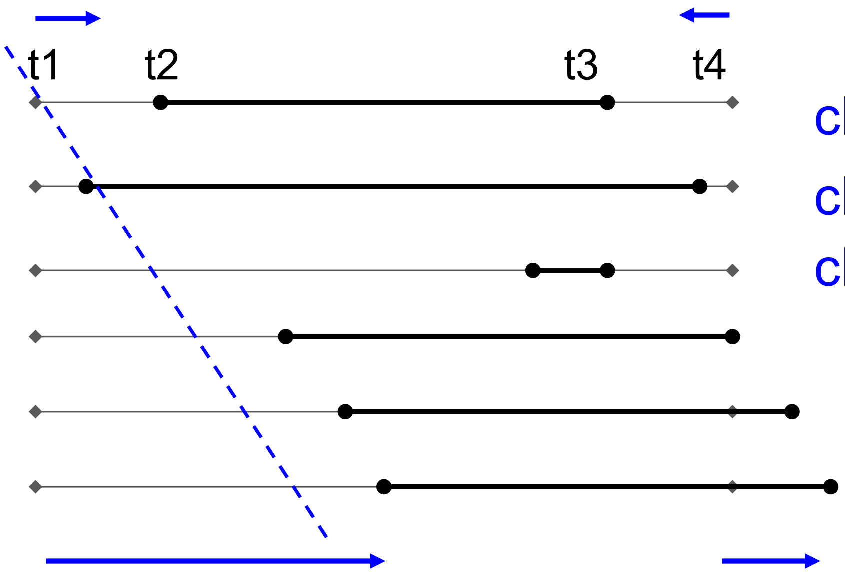
A flawed alignment algorithm

Compute the offset as follows, using only complete entries in the client:

biggest_left_shift = Min(req_rcv_i - req_send_i) over all i
biggest_right_shift = Min(resp_rcv_i - resp_send_i) over all i

offset = (biggest_left_shift + biggest_right_shift) / 2

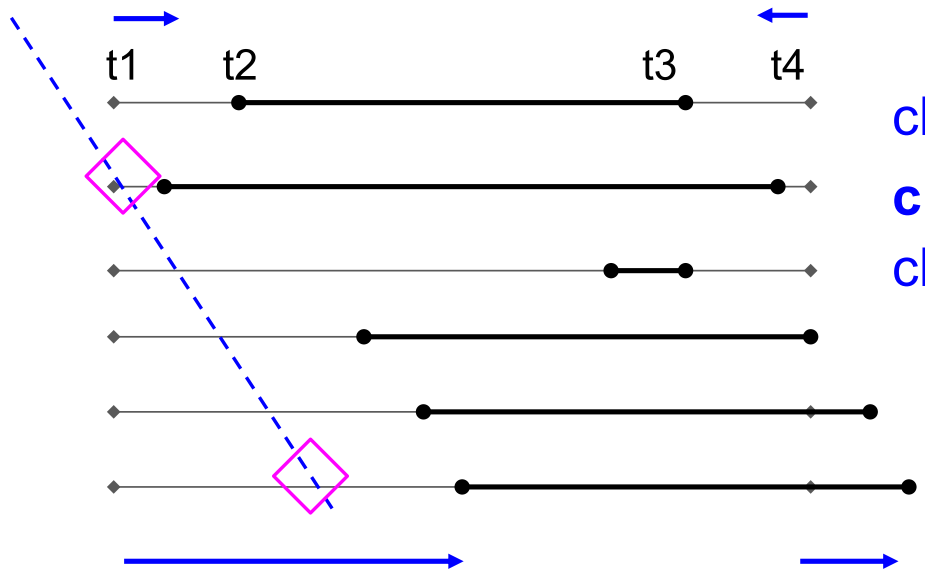
Consider these client/server times



client offset = [-3 .. 3]
client offset = [-1 .. 1]
client offset = [-3 .. 10]
client offset = [0 .. 5]
client offset = [1 .. 6]
client offset = [2 .. 7]

NO single shift fits all the ranges well. Client-server offset **changes** over time, due to clock drift between the two machines -- need a slope term

Consider these client/server times



client offset = [-3 .. 3]

client offset = [-1 .. 1] slop = 2

client offset = [-3 .. 10]

client offset = [0 .. 5]

client offset = [1 .. 6]

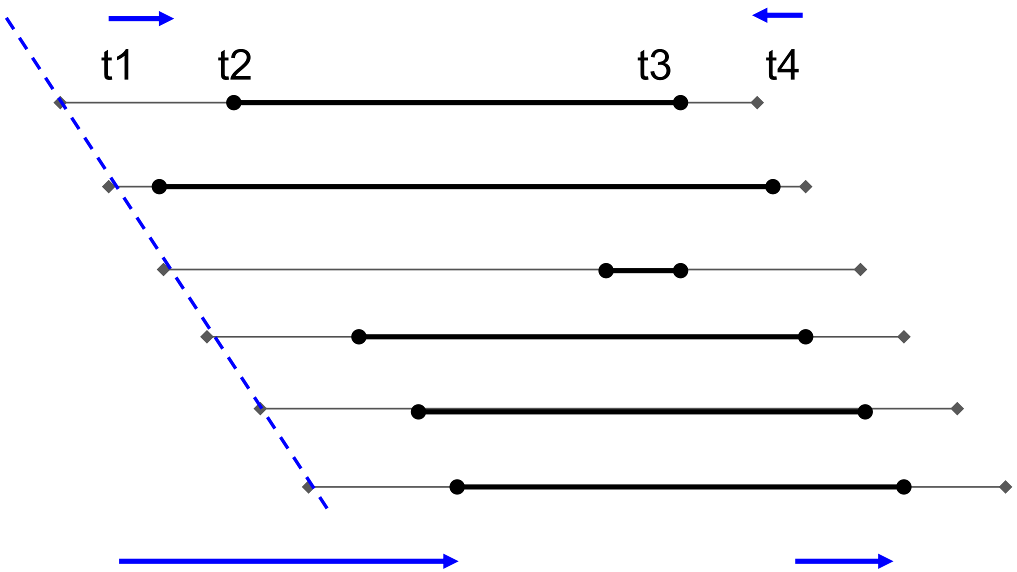
client offset = [2 .. 7] slop = 5

Mid-shift of 0 fits lowest-slop in first three lines.

Mid-shift of 4.5 fits lowest-slop in last three lines

Slope is 4.5 over four lines

Consider these client/server times



Client times adjusted via
newT = (oldT - basetime) * M + B
for slope M and offset B