

Continuing w/ the "using threads" paper, a few observations on priorities

Many thread systems provide what is called a priority scheduler (or strict priority scheduler).

Principle if T_1 is running then \nexists ready but not running thread T_2 with $\text{Priority}(T_2) > \text{Priority}(T_1)$.

(note this is phrased for multiprocessors). On a uniprocessor it reduces to "a highest priority ready thread is running".

In thread systems it is common for threads to have static priorities — the priority is assigned when the thread is created and only changed (seldom) under explicit program control (The CML scheduler does not work this way). Contrast OS process priorities which vary in response to the way a program uses system resources.

Thread

Priorities should never be used to solve synchronization issues (Process priority is sometimes used this way but even that is fairly evil in my opinion.)

Similarly timeouts should not be used for synchronization.

There are really two scheduling issues that priorities attempt to address.

At a low level they help the Thread dispatcher choose which one thread should be run on a processor at any given time. For that you really want something easy to work with because dispatching happens frequently.

More globally priority is an expression of relative importance of different tasks that a processor could be working on. My experience is that this understanding of importance does not translate well to the exclusivity of SP scheduling. Just because activity A is more important than activity B in the long term does not mean it is ^{always} more important in the short term.

Strict priority also leads to the problem of stable priority inversion that we've mentioned before:

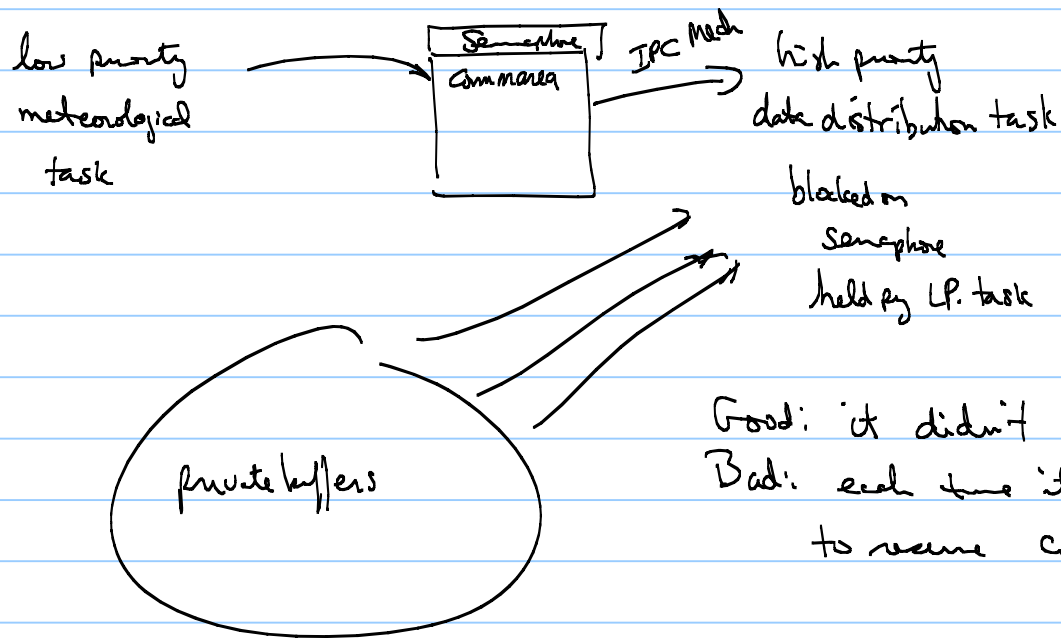
H wants a lock

M is CPU bound

L holds the lock.

Solvable for locks w/ priority inheritance.

Mars Pathfinder priority in version July 1997



unrelated
med^l priority comm task ran much more than expected.
watch dog saw that data dist didn't finish in allowed time and reset the system.

Good: it didn't permanently wedge.
Bad: each time it reset had to wait a day to resume communication

Solution: enable prio inheritance on the Semaphore.

Attended view of priorities: express the long-term share of CPU that a task should receive or even the share needed over intervals of specified length.

All threads make progress but w/ different resource consumption limits. Downside - more costly dispatching decisions because the dispatching priority is constantly changing.

N.B. This would probably not have solved the pathfinder problem - it was related to holding a specific lock for too long (Note These ideas also show up in network packet scheduling literature.)

This approach partitions a resource, isolating threads from bad independent behavior of other threads. In the "many threads" paper we observed enough instances of priority inversion that we had to ensure progress by all threads using a "donate some cycles" hack.

Finally, want to talk about one more concurrency pattern. This from

Pattern Oriented Software Architecture Volume 2: Patterns for Concurrent & Networked Objects,
by Schmidt, Stael, Robert & Buschman

First, consider an alternative to threads called the event driven model or reactor

```
while (1) {
```

```
    e ← request from any of multiple sources
```

```
    process e appropriately
```

```
}
```

// Key Press
// mouse click
// network packet
// ...

Good - all single threaded; no worries about synchronization

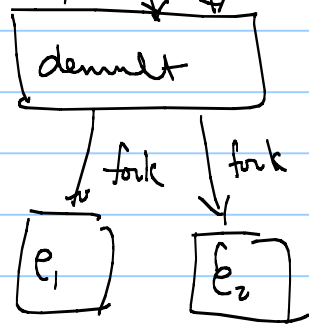
Bad - different event streams might be quite unrelated, have state that needs to be remembered from one event to the next. state Mgmt has to be explicitly programmed using custom data structures instead of residing in the stack of a thread. Doesn't help w/ mult-processors

Light weight, fast, when retained context is not needed.

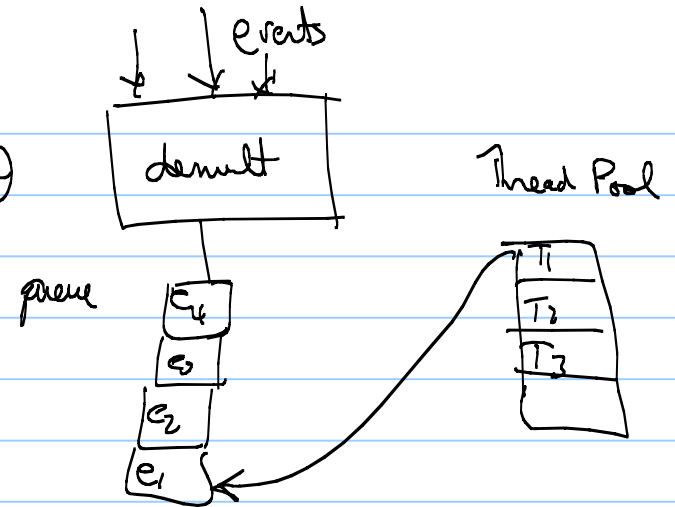
Basic Unix server of
events processes
w/ mult. threads

Alternatives

1)



2)

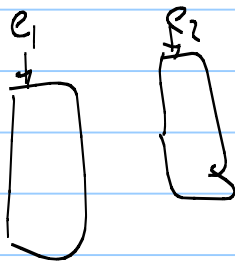


0) High overhead assoc. } not practical for low # of sources
memory overhead

1) Has very high overhead associated w/ forking

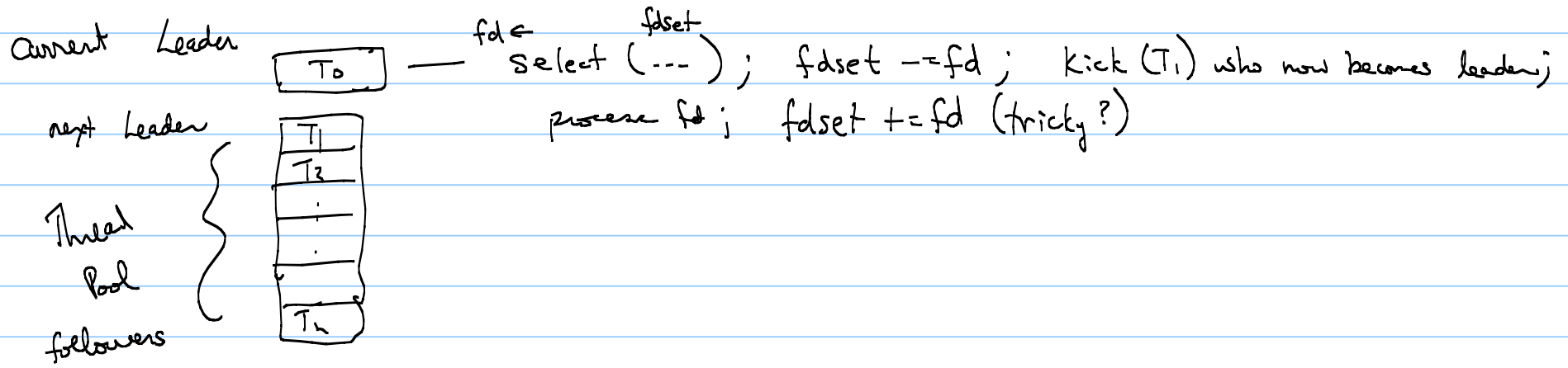
2) Has high overhead for allocation of queue elements
and synchronization on the queue.

3)



Thread per source.

Leaders and followers: Threads take turns doing the demultiplexing step.



Variants win32 Wait for Multiple Events doesn't need the manipulation of the $fdset$. (maybe)

Advantage: The leader does not have to allocate or thread switch to begin processing an fd event; large amounts of data — e.g. — result of $\text{read}()$ do not have to pass between threads. (cache coherency cost is reduced).

Disadvantages: complexity of implementation
No explicit queue for priority event processing.