ECE 669

Parallel Computer Architecture

Lecture 19

Processor Design

Overview

- Special features in microprocessors provide support for parallel processing
  - Already discussed bus snooping
- Memory latency becoming worse so multi-process support important
- Provide for rapid context switches inside the processor
- Support for prefetching
  - Directly affects processor utilization

Why are traditional RISCs ill-suited for multiprocessing?

- Cannot handle asynchrony well
  - Traps
  - Context switches
- Cannot deal with pipelined memories - (multiple outstanding requests)
- Inadequate support for synchronization

  (Eg. R2000 ——— No synchro instruction)

  (SGI ——— Had to memory map synchronization)

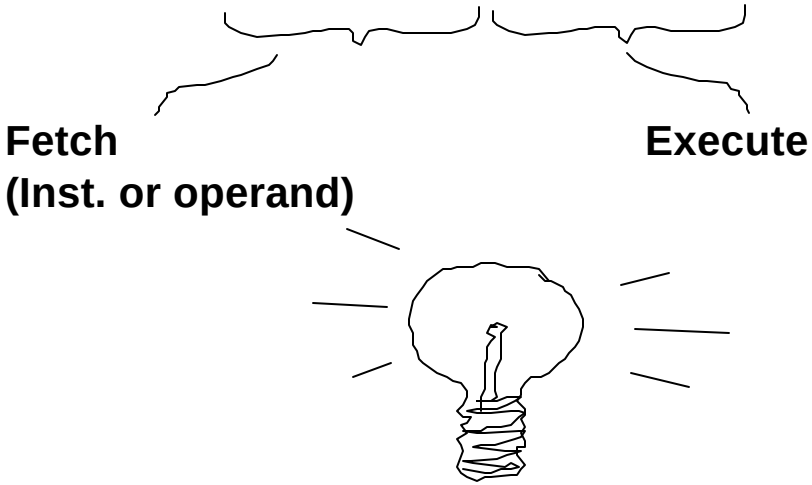
Three major topics

- Pipeline processor-memory-network
  - Fast context switching
  - Prefetching

  (Pipelining: Multithreading)
- Synchronization
- Messages

Pipelining – Multithreading Resource Usage

- Mem. Bus
- ALU

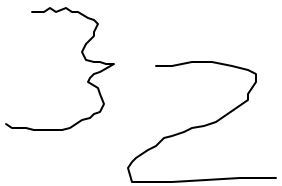


- Overlap memory/ALU usage
  - More effective use of resources
  - Prefetch
  - Cache
  - Pipeline (general)

RISC Issues

- 1 Inst/cycle
  - Huge memory
  - bandwidth requirements
    - Caches: 1 Data Cache

      or
    - Separate I&D caches
  - Lots of registers, state
- Pipeline Hazards
  - Compiler
  - Reservation bits
  - Bypass Paths



Interlocks

  - More state!
- Other “stuff” - register windows
  - Even more state!

Fundamental conflict

- **Better single-thread performance (sequential)**
  - **More on-chip state**
- **More on-chip state**
  - **Harder to handle asynchronous events**
    - **Traps**
    - **Context switches**
    - **Synchronization faults**
    - **Message arrivals**

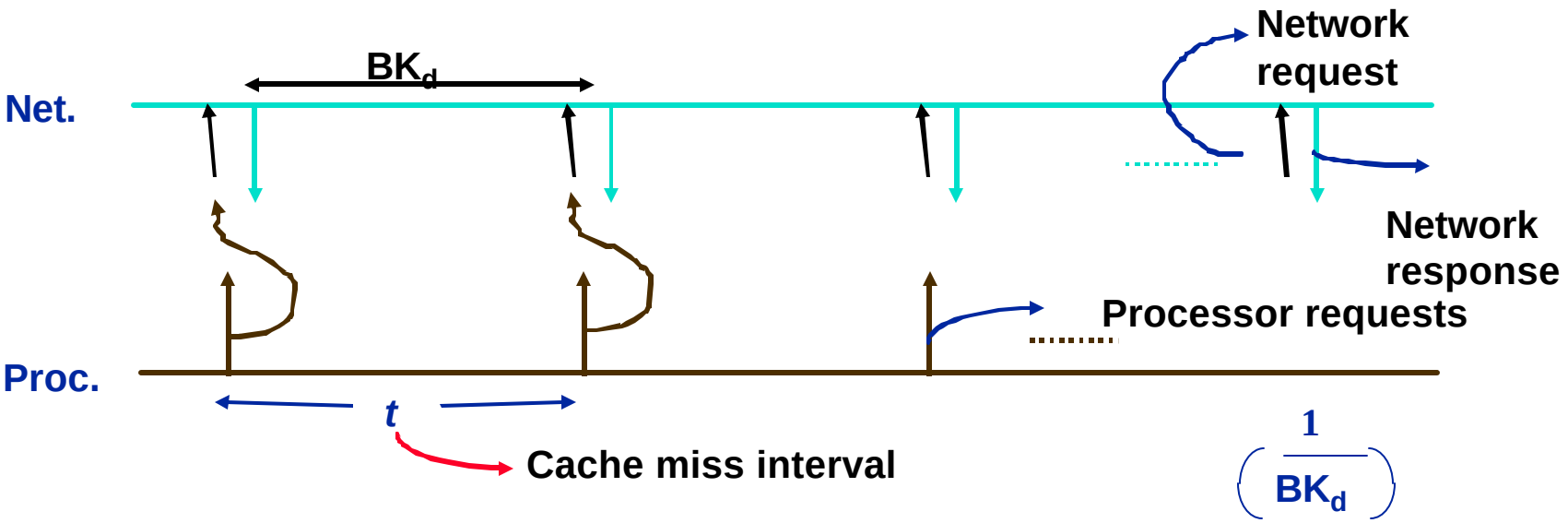
But, why is this a problem in MPs?

Makes pipelining proc-mem-net harder.

Consider...

Ignore communication system latency (T=0)

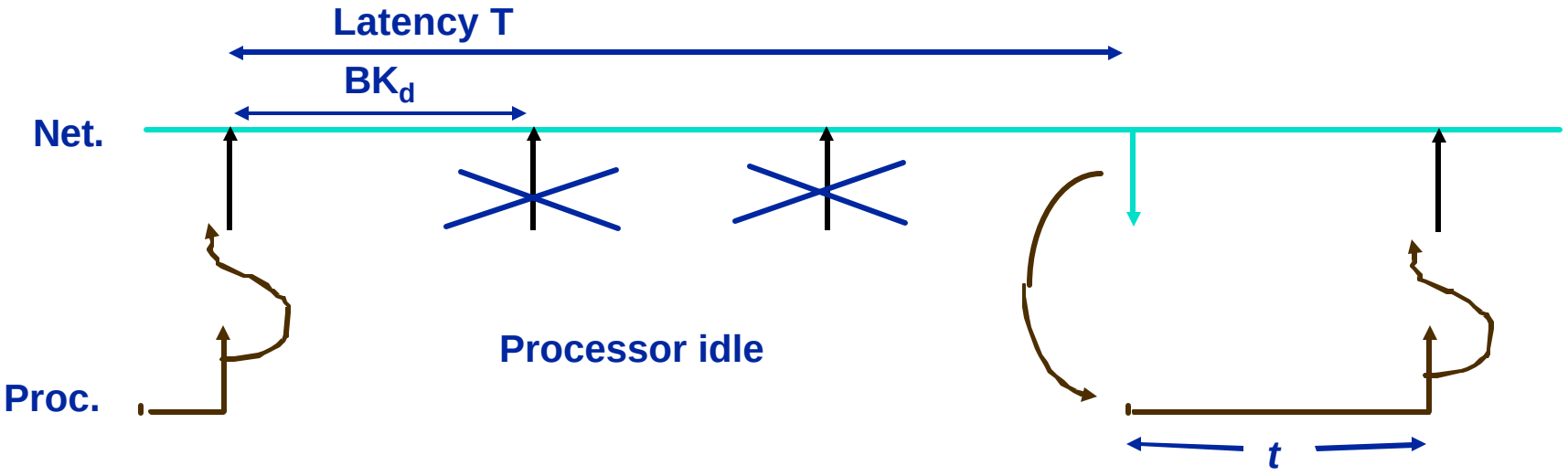
- Then, max bandwidth per node limits max processor speed



- Above
  - Processor-network matched rate=net bandwidth i.e. proc request
  - If processor has higher request rate, it will suffer idle time

Now, include network latency

- Each request suffers T cycles of latency



$$\frac{t}{t + T} = \frac{1}{1 + mT}$$

- Processor utilization = $\left(t = \frac{1}{m} \right)$

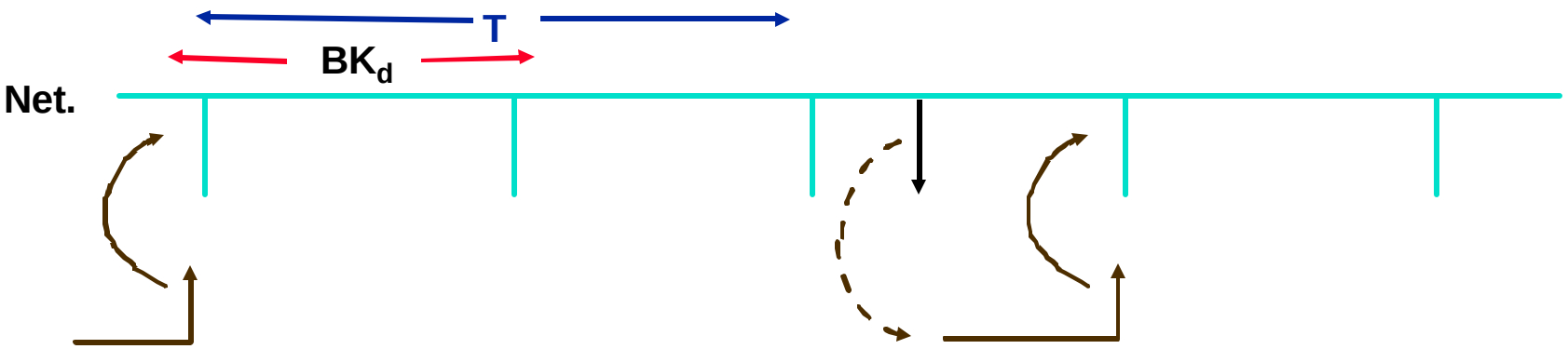
  Processor utilization

  Network bandwidth also wasted because of lost issue opportunities!

  FIX?

One solution

- **Overlap communication with computation.**



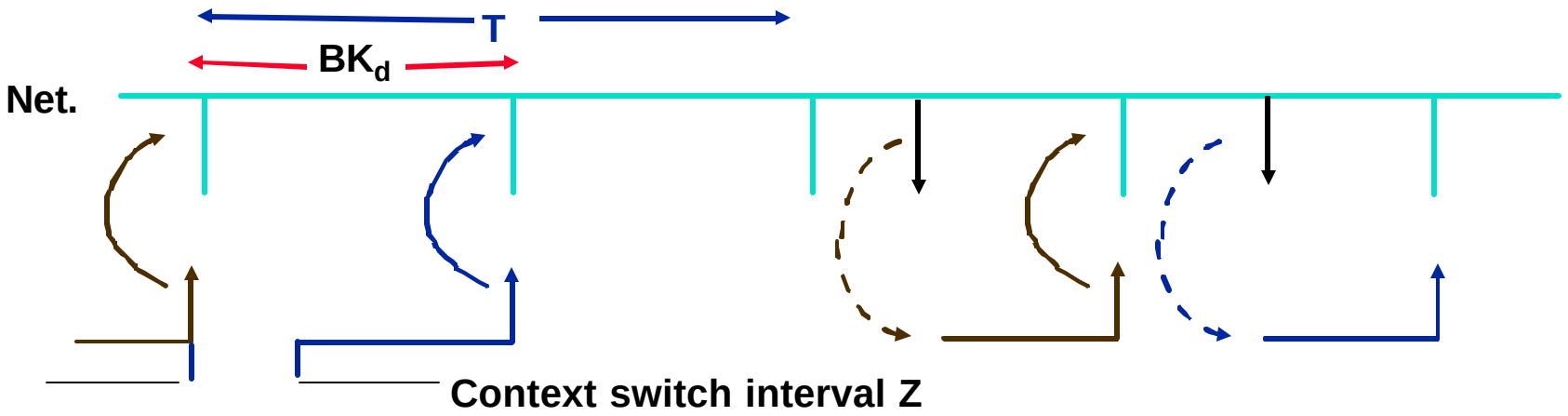
- **"Multithread" the processor**
  - **Need rapid context switch. See HEP, Sparcle.**

Processor utilization $= \dfrac{pt}{t + T}$ if $pt < (t + T)$

- **And/or allow multiple outstanding requests -- non-blocking memory**

One solution

Overlap communication with computation.



- "Multithread" the processor
  - Need rapid context switch. See HEP, Sparcle.

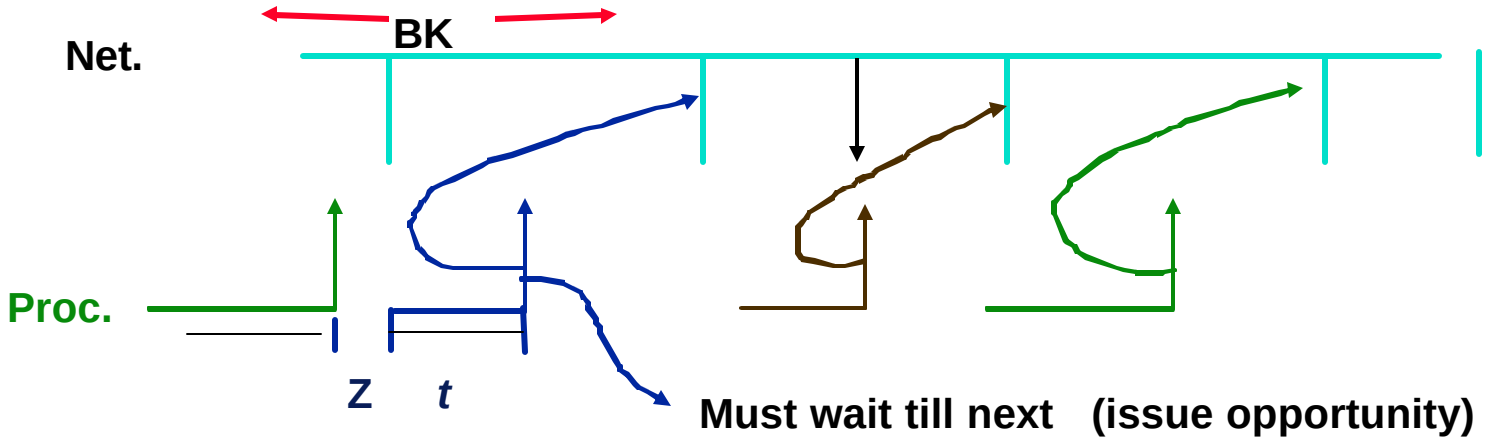
$$\textbf{Processor utilization} = \frac{pt}{t + T} \quad \text{if } pt < (t + T)$$

$$\textbf{or} = \frac{t}{t + Z} \quad \text{otherwise}$$

- And/or allow multiple outstanding requests -- non-blocking memory

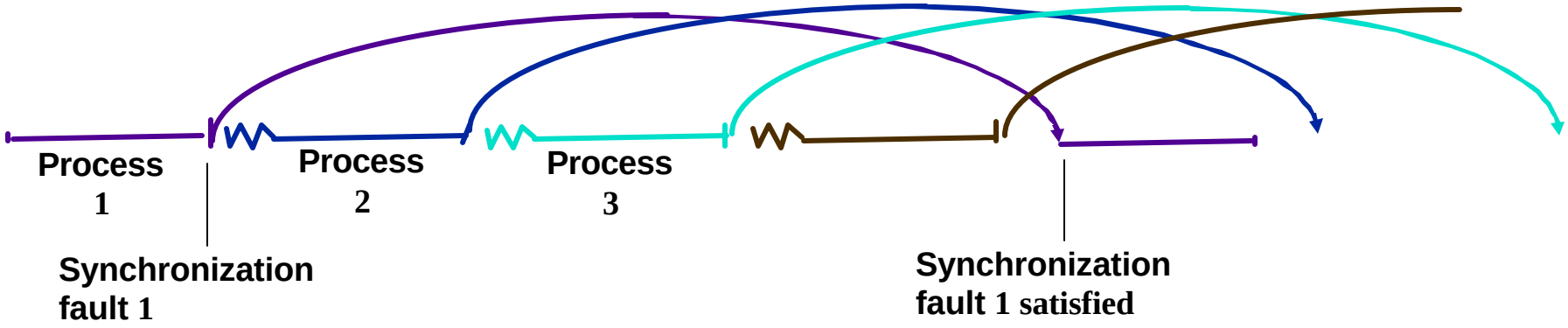
Caveat!

- Of course, previous analysis assumed network bandwidth was not a limitation.
- Consider:

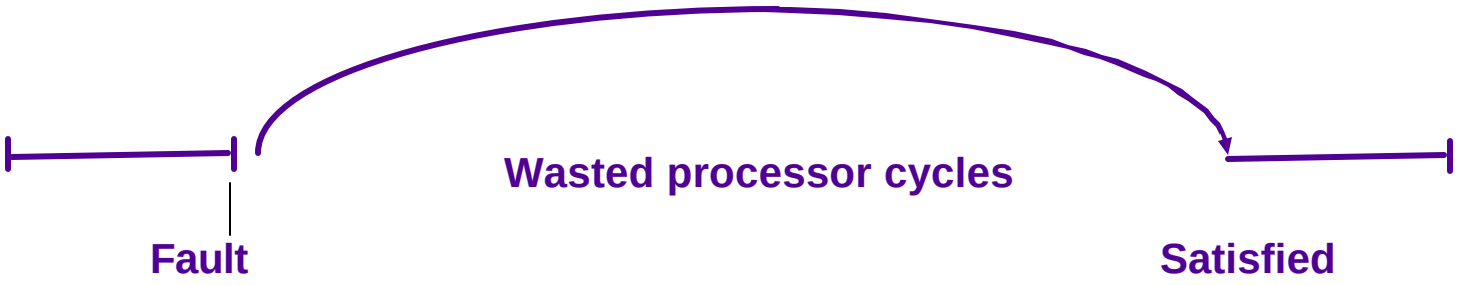


- Computation speed (proc. util.) limited by network bandwidth.
- Lessons: Multithreading allows full utilization of network bandwidth. Processor util. will reach 1 only if net BW is not a limitation.

Same applies to synchronization delays as well



- If no multithreading



Requirements for latency tolerance (comm or synch)

- Processors must switch contexts fast
- Memory system must allow multiple outstanding requests
- Processors must handle traps fast (esp synchronization)
- Can also allow multiple memory requests

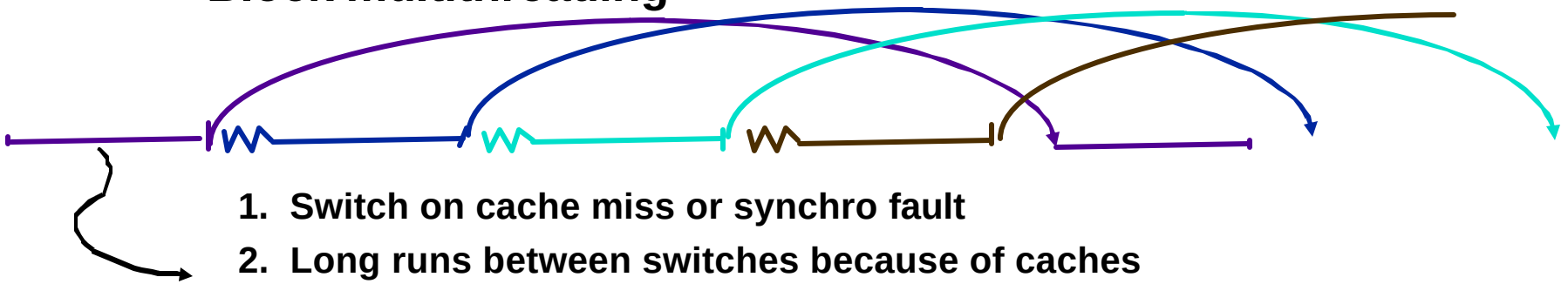
° **But, caution:**

Latency tolerant processors are no excuse for not exploiting locality and trying to minimize latency

° **Consider...**

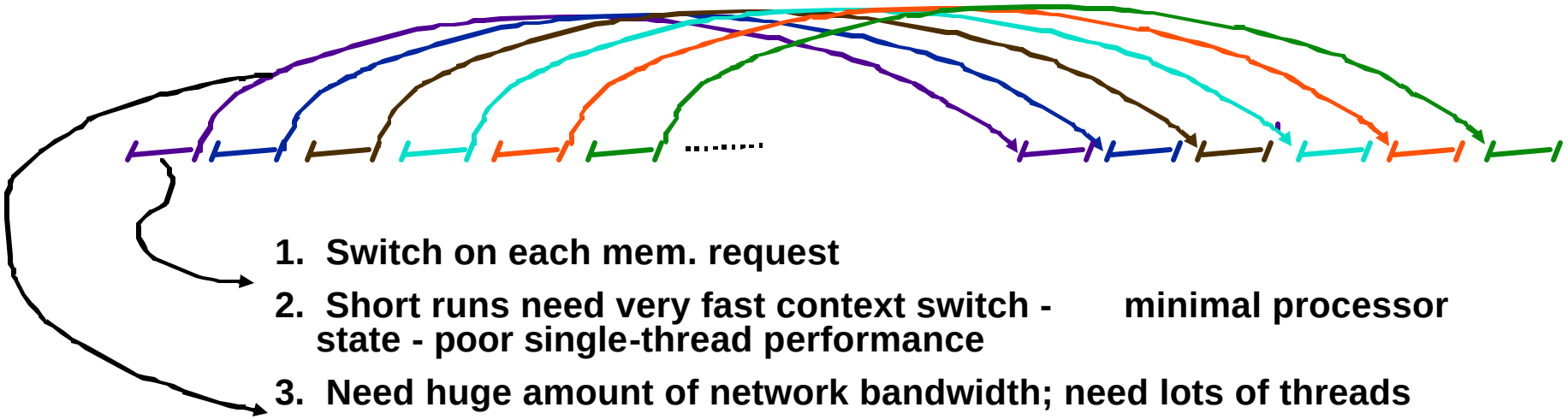
Fine multithreading versus block multithreading

- **Block multithreading**



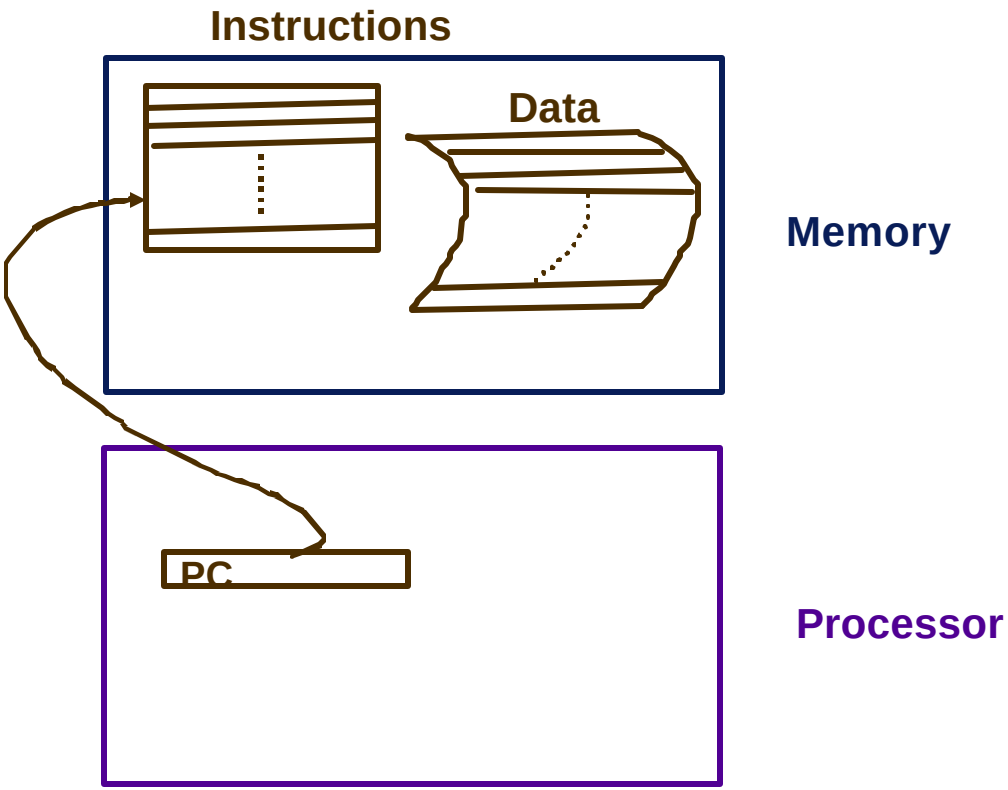
1. Switch on cache miss or synchro fault
2. Long runs between switches because of caches
3. Fewer request in network

- **Fine multithreading**



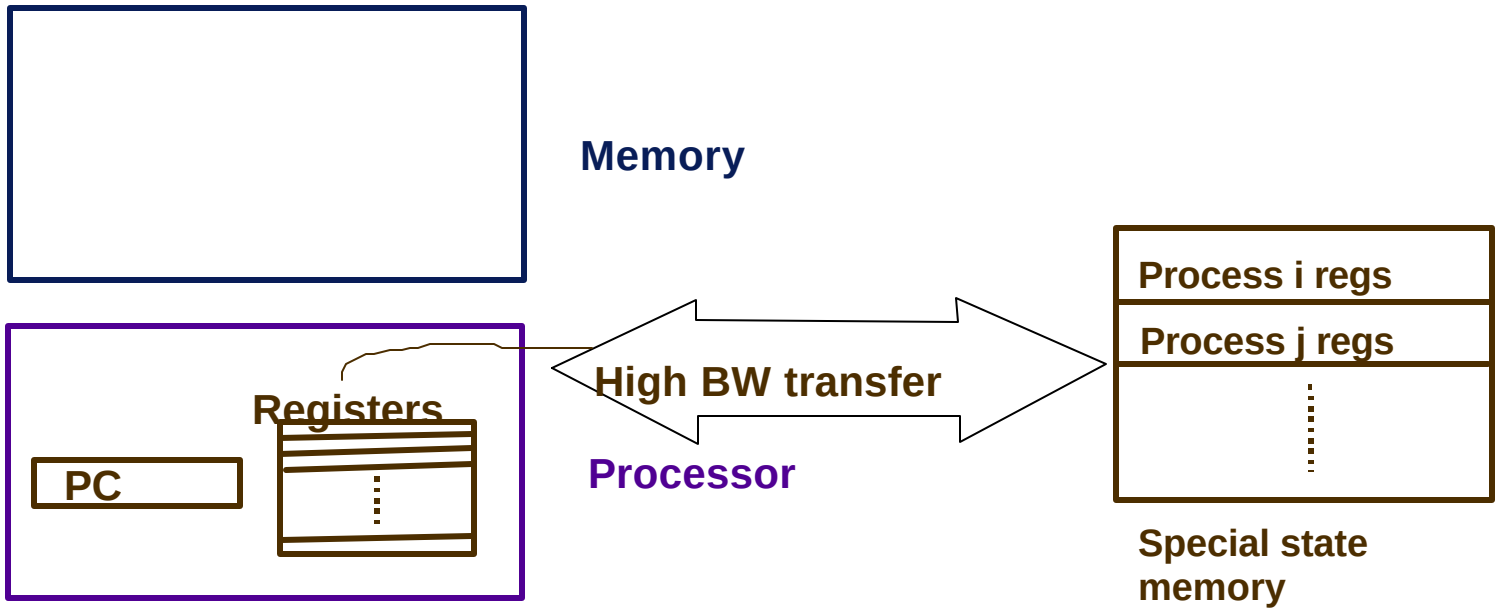
1. Switch on each mem. request
2. Short runs need very fast context switch - minimal processor state - poor single-thread performance
3. Need huge amount of network bandwidth; need lots of threads

How to implement fast context switches?



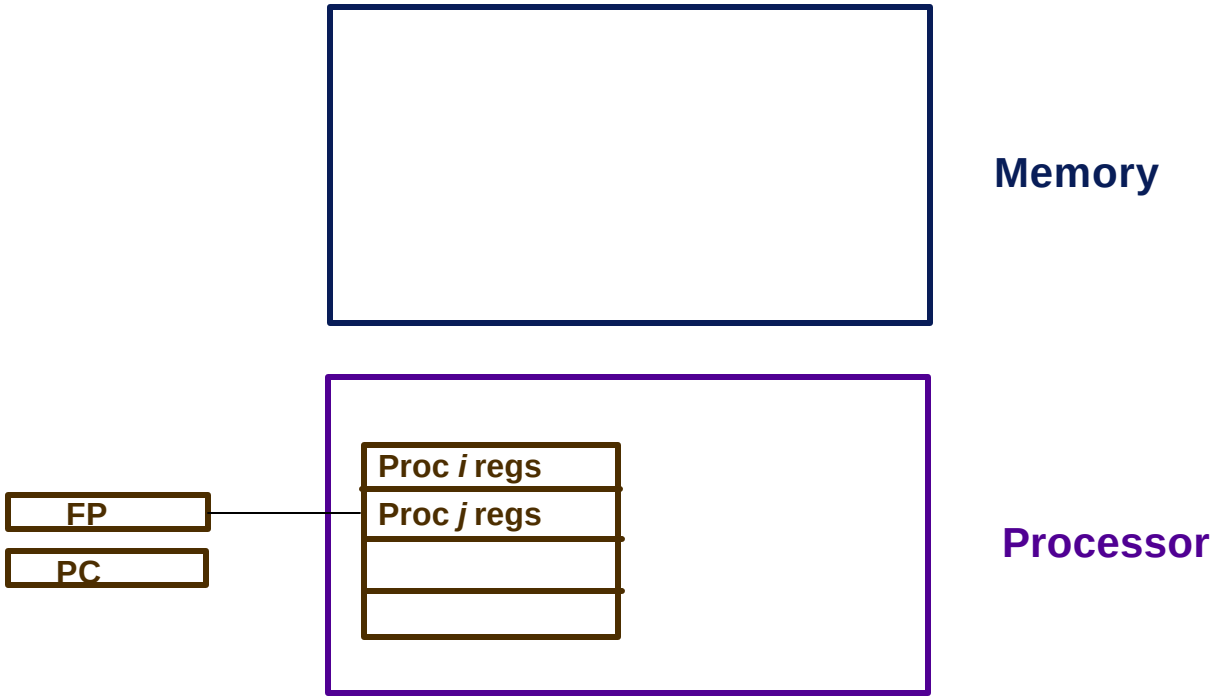
- Switch by putting new value into PC
- Minimize processor state
- Very poor single-thread performance

How to implement fast context switches?



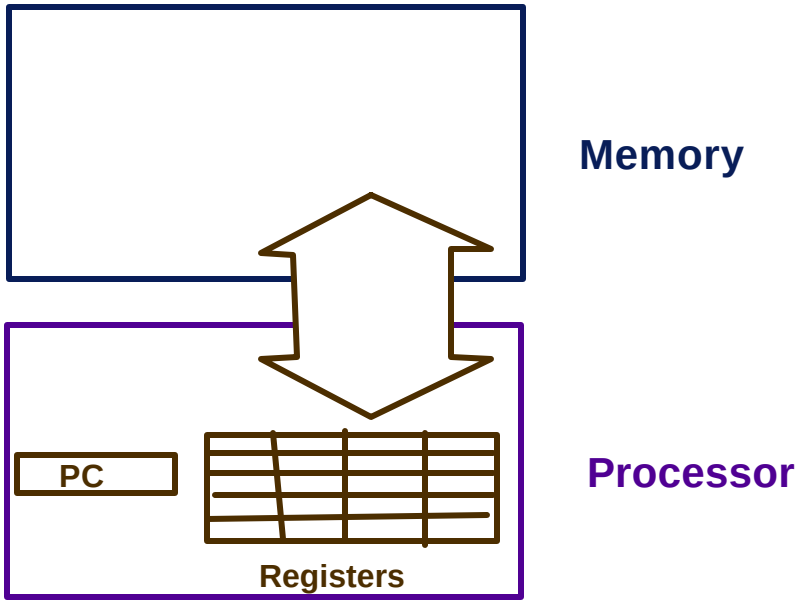
- Dedicate memory to hold state & high bandwidth path to state memory
- Is this best use of expensive off-chip bandwidth?

How to implement fast context switches?



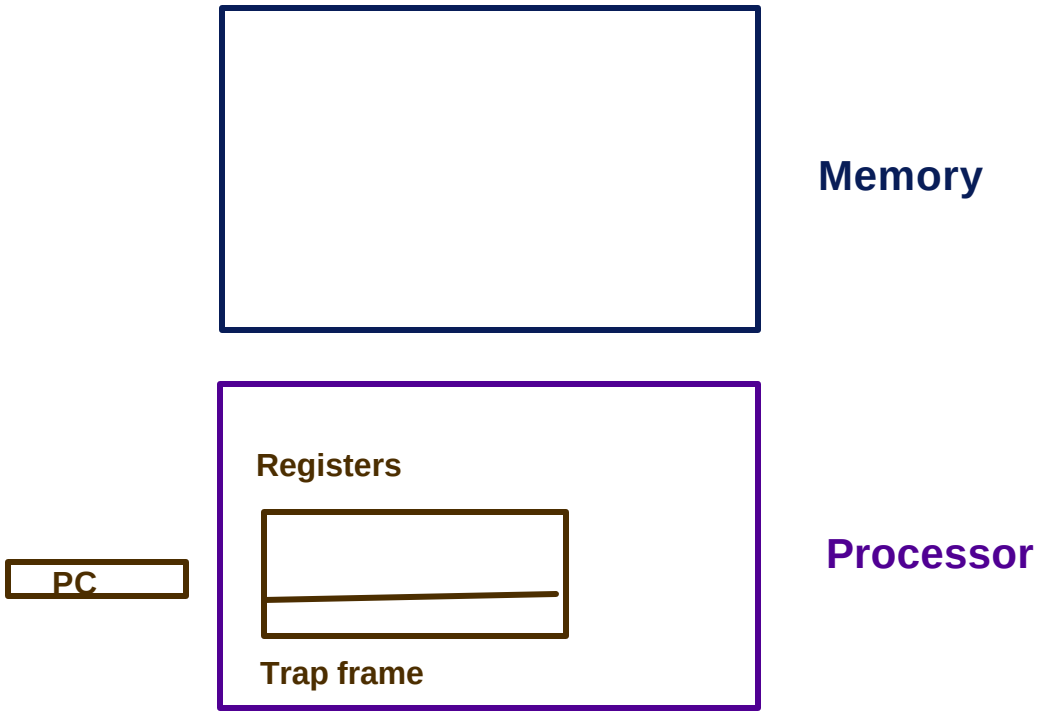
- Include few (say 4) register frames for each process context.
- Switch by bumping FP (frame pointer)
- Switches between 4 processes fast, otherwise invoke software loader/unloader - Sparcle uses SPARC windows

How to implement fast context switches?



- **Block register files**
- **Fast transfer of registers to on-chip data cache via wide path**

How to implement fast context switches?

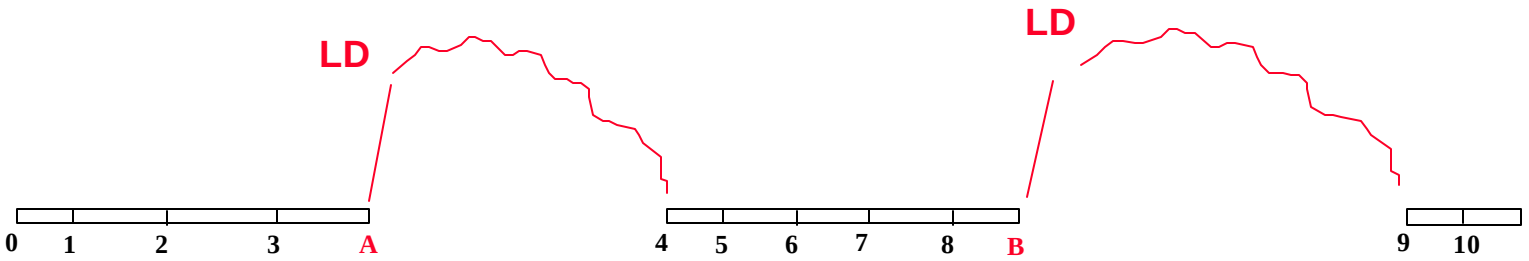


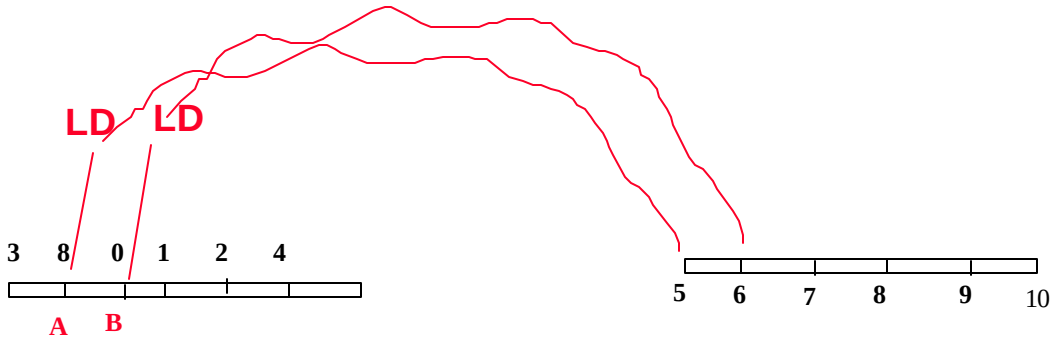
Fast traps also needed.

- Also need dedicated synchronous trap lines --- synchronization, cache miss...
- Need trap vector spreading to inline common trap code

Pipelining processor - memory - network

- **Prefetching**

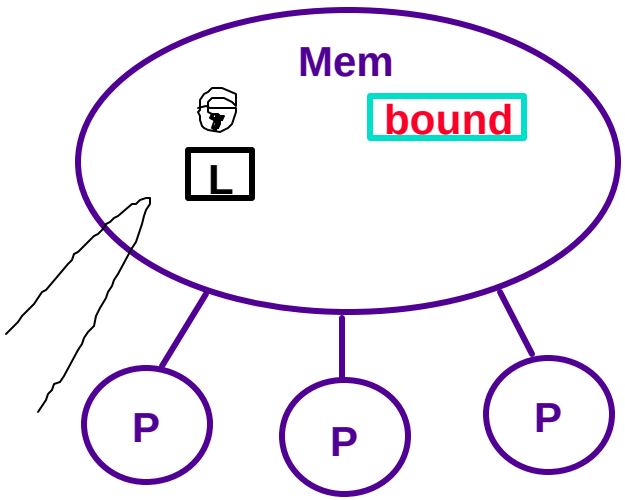




Synchronization

- **Key issue**
  - **What hardware support**
  - **What to do in software**
- **Consider atomic update of the “bound” variable in traveling salesman**

Synchronization




```
While (LOCK(L)==1);          Loop
          read bound
          incr bound
          store bound
          unlock(L)

Lock(L)
   read L                          } test
   if (L==1) return 1;
   else L=1                        } set
   store L
   return o;
```

Atomic

- Need mechanism to lock out other request to L

In uniprocessors

- Raise interrupt level to max, to gain uninterrupted access

° In multiprocessors
  - Need instruction to prevent access to L.

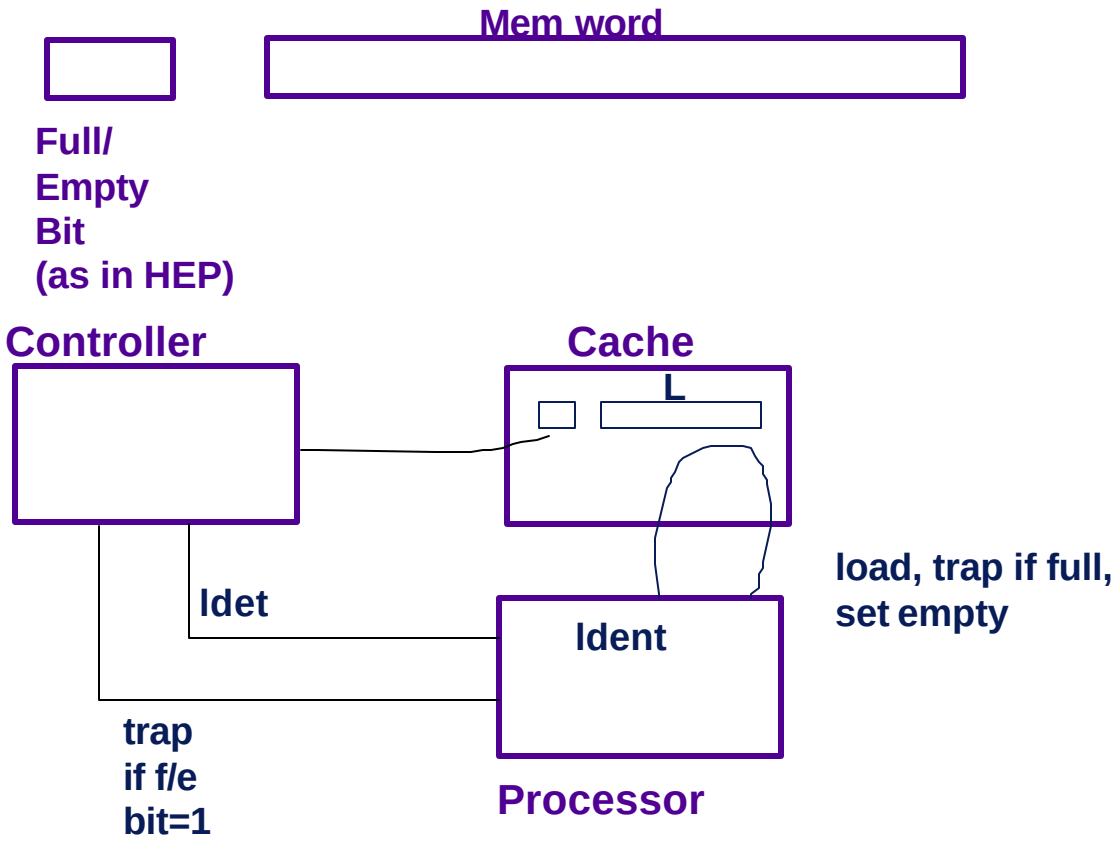
° Methods
  - Keep synchro vars in memory, do not release bus
  - Keep synchro vars in cache, prevent outside invalidations

° Usually, can memory map some data fetches such that cache controller locks out other requests

Data-parallel synchronization

Can also allow controller to do update of L.

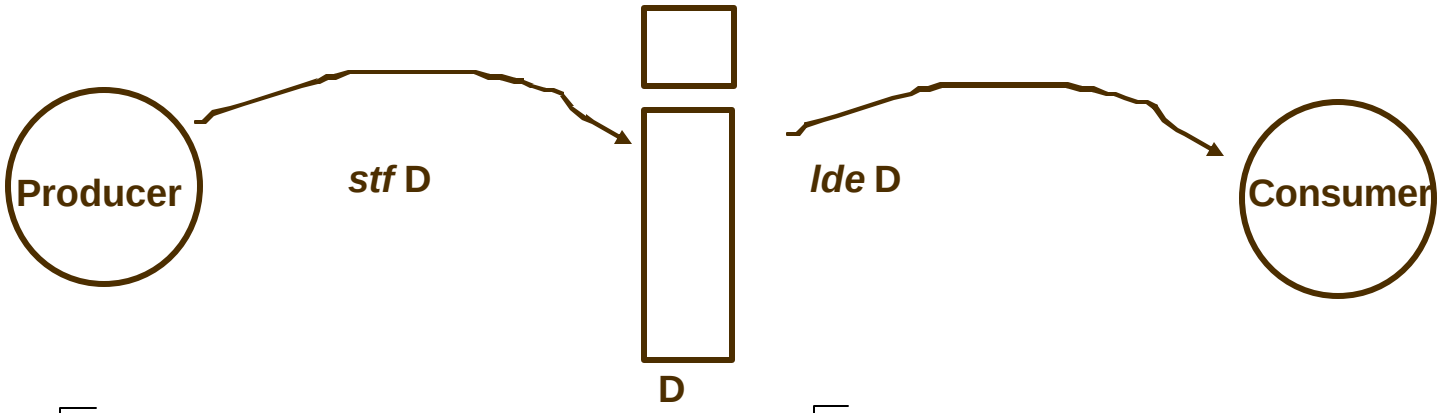
Eg. Sparcle (in Alewife machine)



Given primitive atomic operation can synthesize in software higher forms

Eg.

1. Producer-consumer

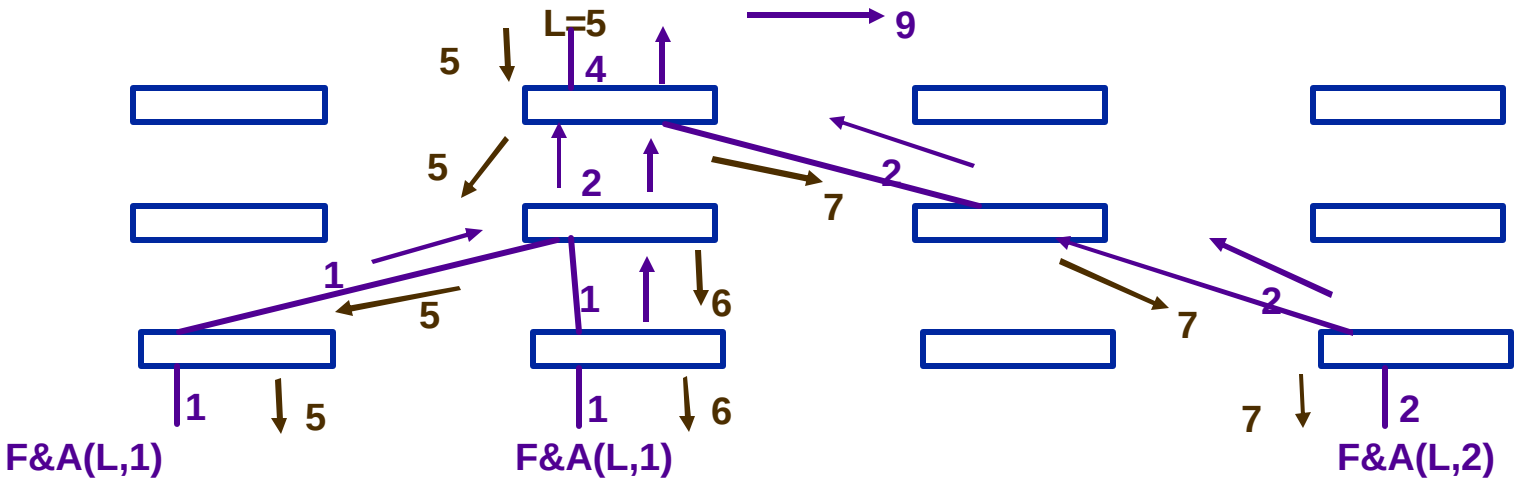



```
Store if
f/e=0
set f/e=1

trap
otherwise...retry
```

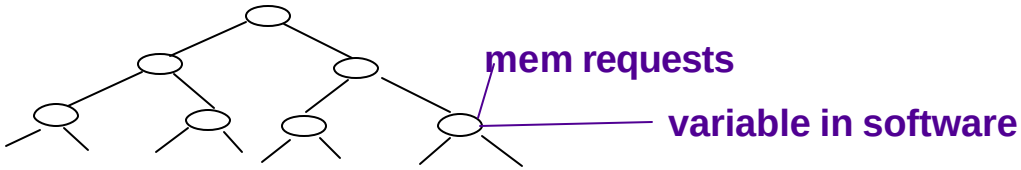
```
Load if
f/e=1
set f/e=0

trap
otherwise...retry
```

Some provide massive HW support for synchronization -- eg. Ultracomputer, RP3

- Combining networks.
- Say, each processor wants a unique *i*



- Switches become processors -- slow, expensive
- Software combining -- implement combining tree in software using a tree data structure



Summary

- Processor support for parallel processing growing
- Latency tolerance supports by fast context switching
  - Also more advanced software systems
- Maintaining processor utilization is a key
  - Ties to network performance
- Important to maintain RISC performance
- Even uniprocessors can benefit from context switches
  - Register windows