Software Distributed Shared Memory

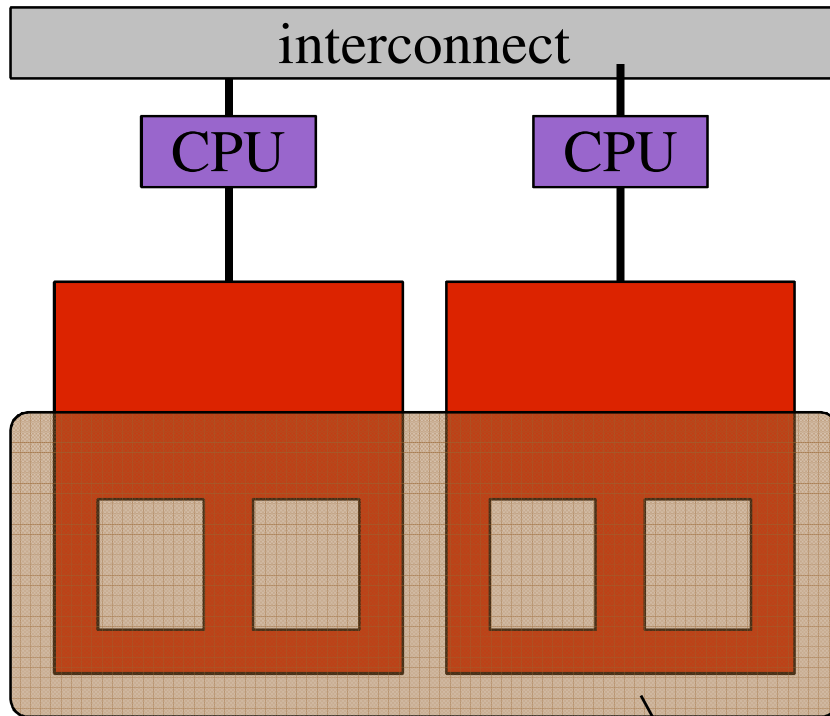
Alistair Rendell

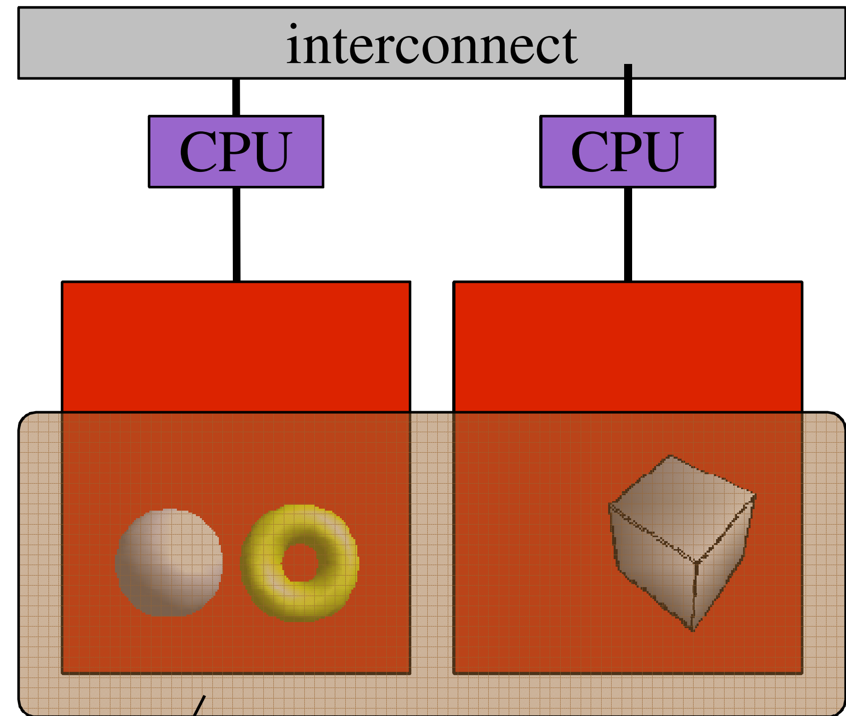
Taken from work with PhD Student H'sien Wong and slides from Wilkinson and Allen

Why Shared Memory on Clusters?

- Provide a more flexible environment
  - programs not so limited by memory per node
  - ease the transition from sequential to (MPI) parallel
- Some applications are too complicated for message passing
  - knowing when to send/receive data
  - even with MPI-2
- MPI is “the assembly language of parallel programming”

Distributed Shared Memory (DSM)





DSM administered memory

Controlling DSM Access

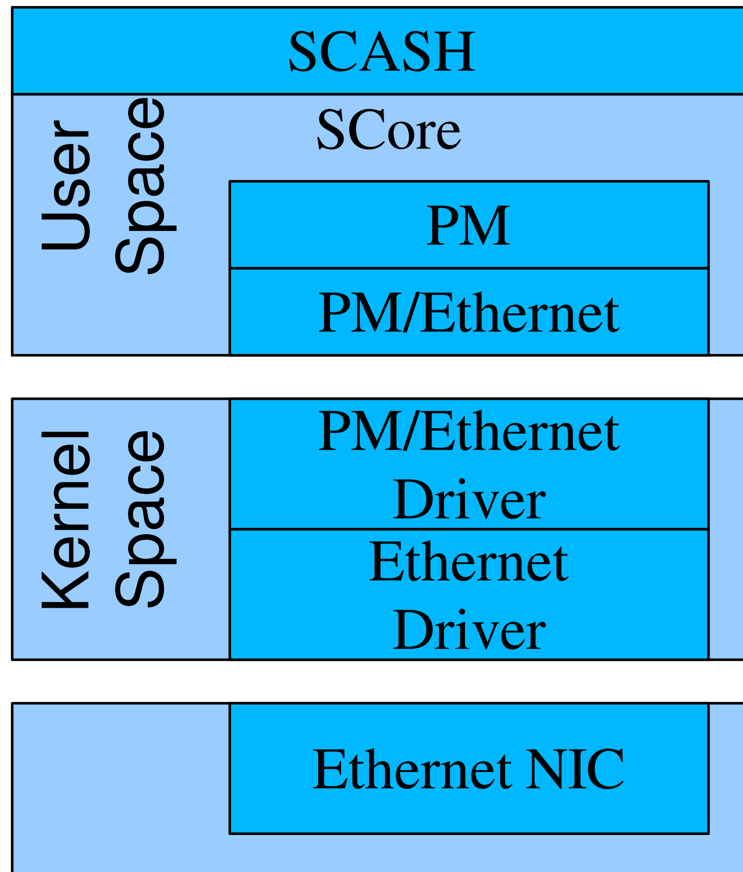
- Detect memory access
  - programmer explicitly requests for access (more object based), or
  - mprotect is used together with a handler for SIGSEGV signals (natural for page based systems)
- Ensure necessary conditions for memory access are in place
  - if the data is already available, maintain page/object table attributes
  - else also fetch:
    - the page/object (migration)
    - a copy of the page/object – DSM has to deal with coherency issues.

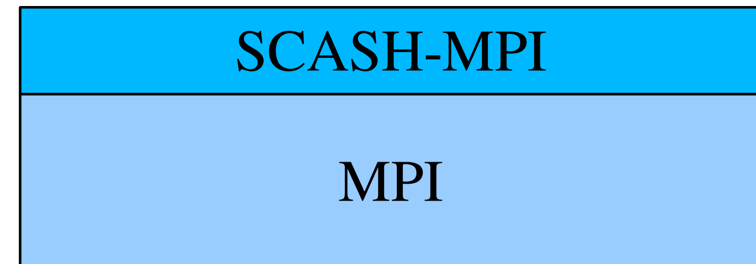
Some Software DSM Systems

- **SCASH**
  - Page based system as part of SCORE
- Treadmarks
  - Page based system now available from Intel as their Cluster OpenMP environment (ClOMP)
- **Adsmith**
  - C++ object based DSM system (studied by Wilga Hawkins as part of her honours work)
  - Library based
- Linda
  - Object based from Yale, now available from commercial company
  - Requires compiler support

Paged Based DSM
SCASH

The SCASH DSM





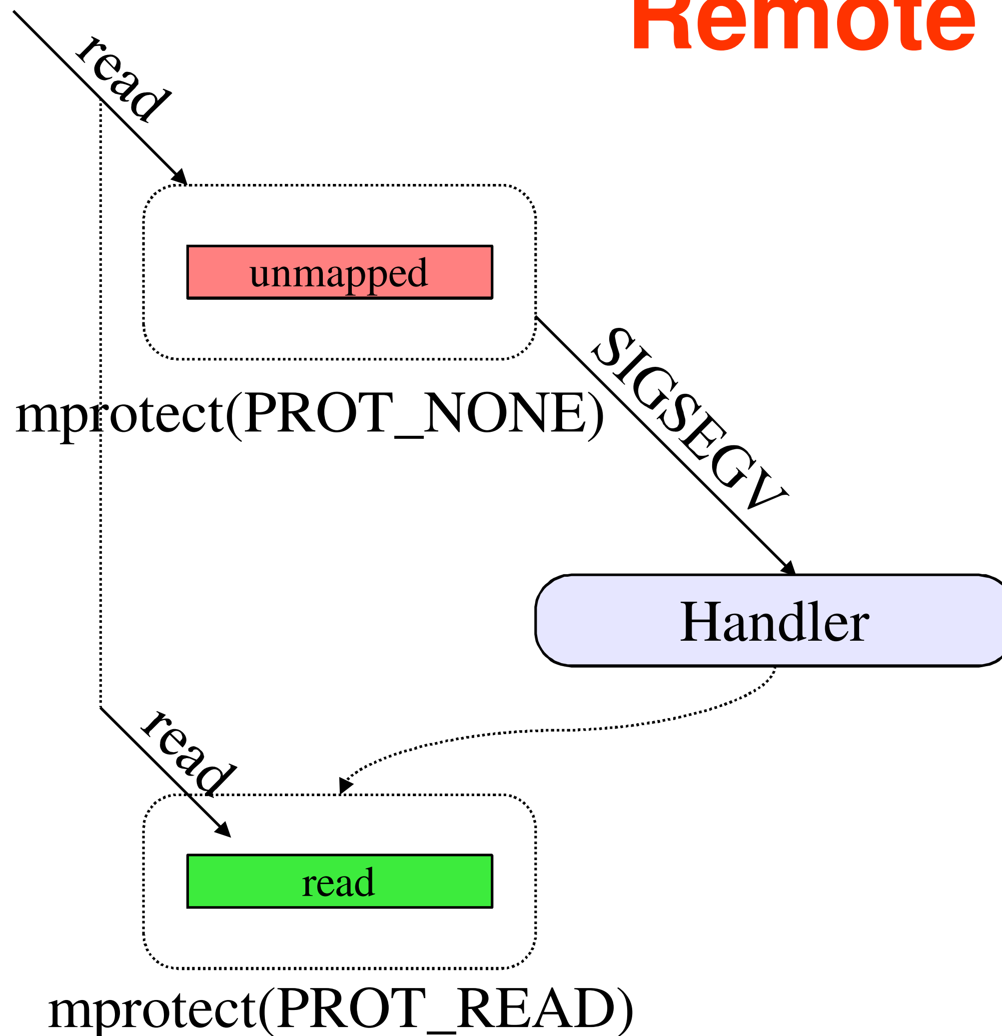
(experimental)

- Part of SCore
  - www.pccluster.org
- PM Layer provides:
  - Message Passing
  - Remote Memory Access (RMA)

The SCASH DSM

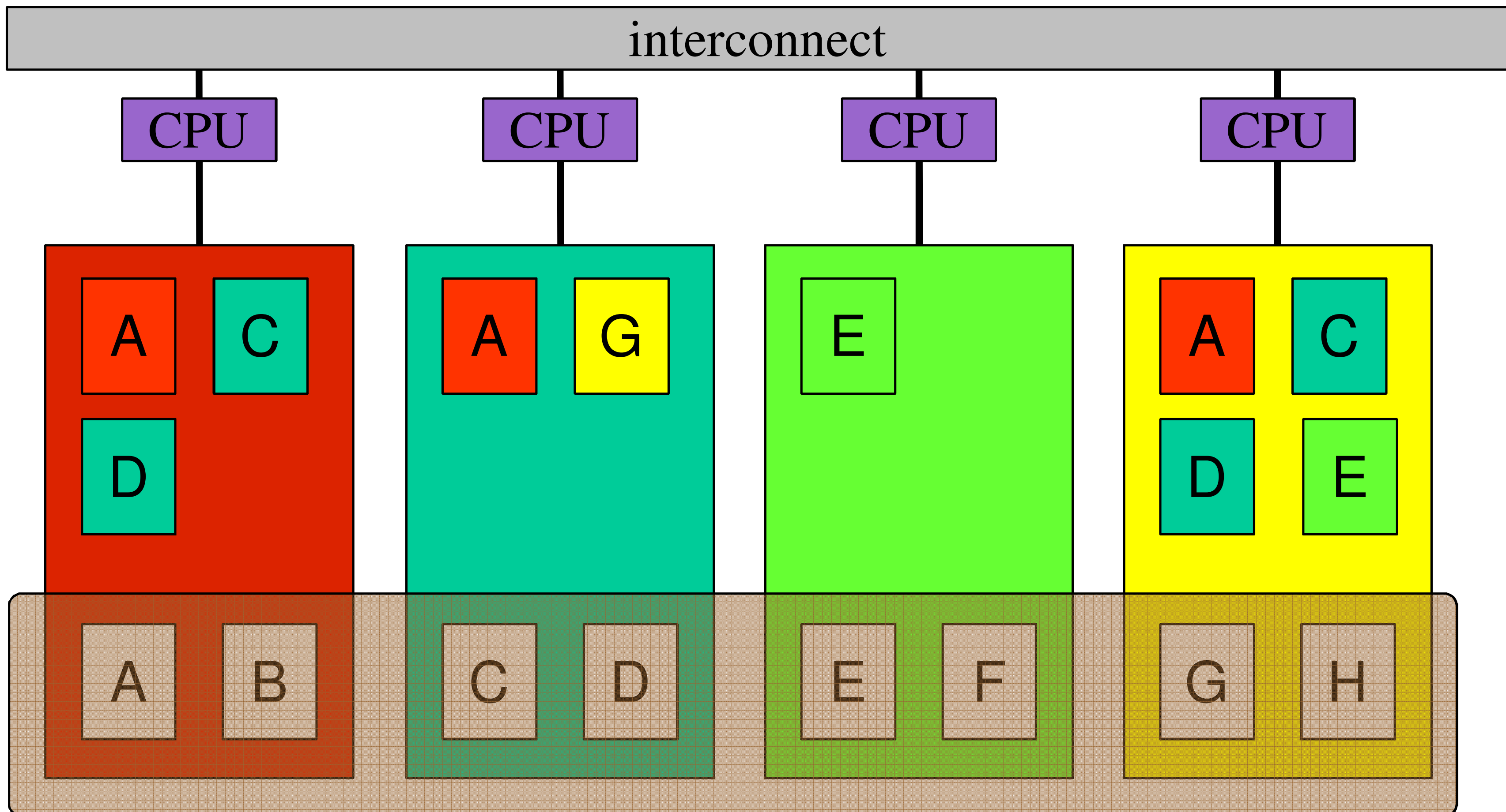
- Each page of DSM memory has a home process.
- Page permissions:
  - “unmapped”, “read-only”, or “read-write”
- Page fetches are copies.
  - supports multiple writers through twinning-and-diffing.
- Uses PM for remote memory access.
- Consistency maintained at *barrier* points.

Reading from an Unmapped Remote Page



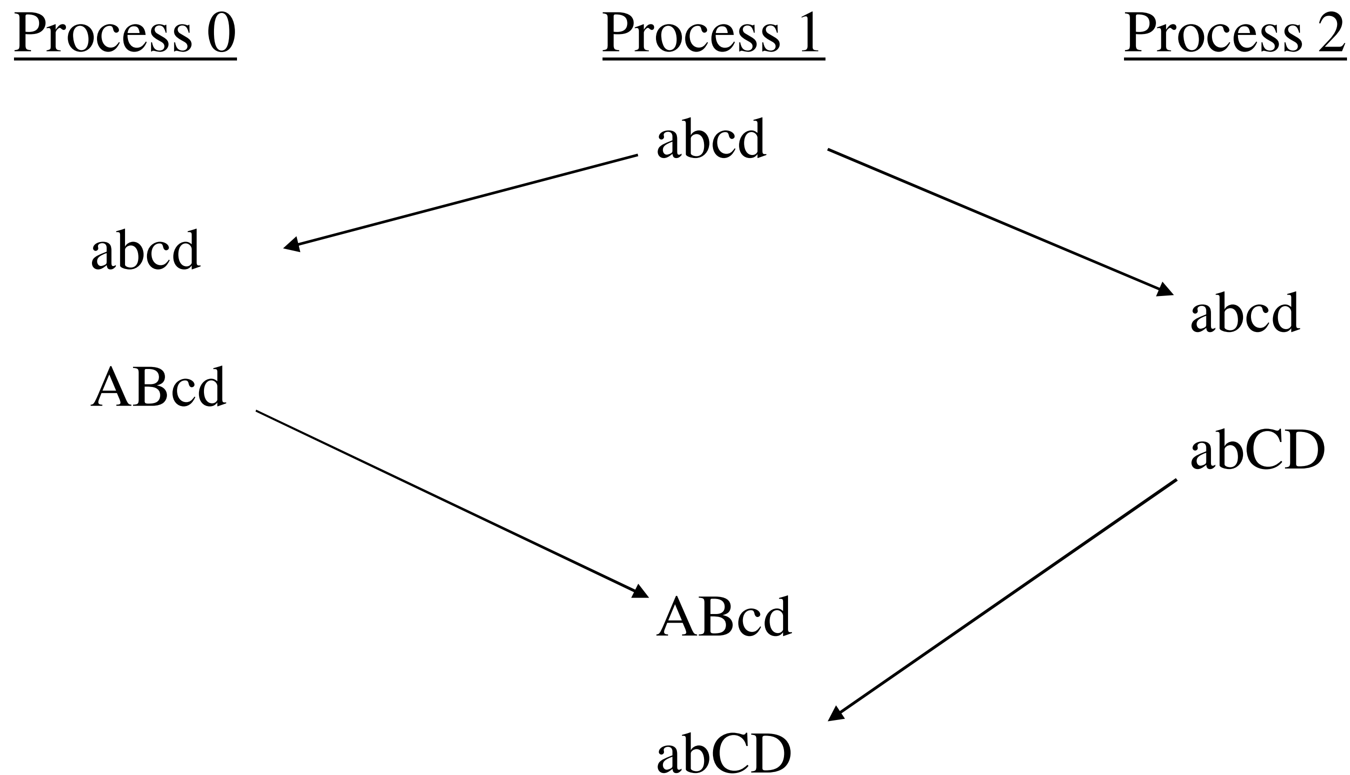
- Use PM RMA to fetch page data
- Notify page owner to update the copyset of the page
  - leave message in page owner's mail box
- Change attributes and mprotect

View of Memory Pages After Running for Some Time and Before Barrier



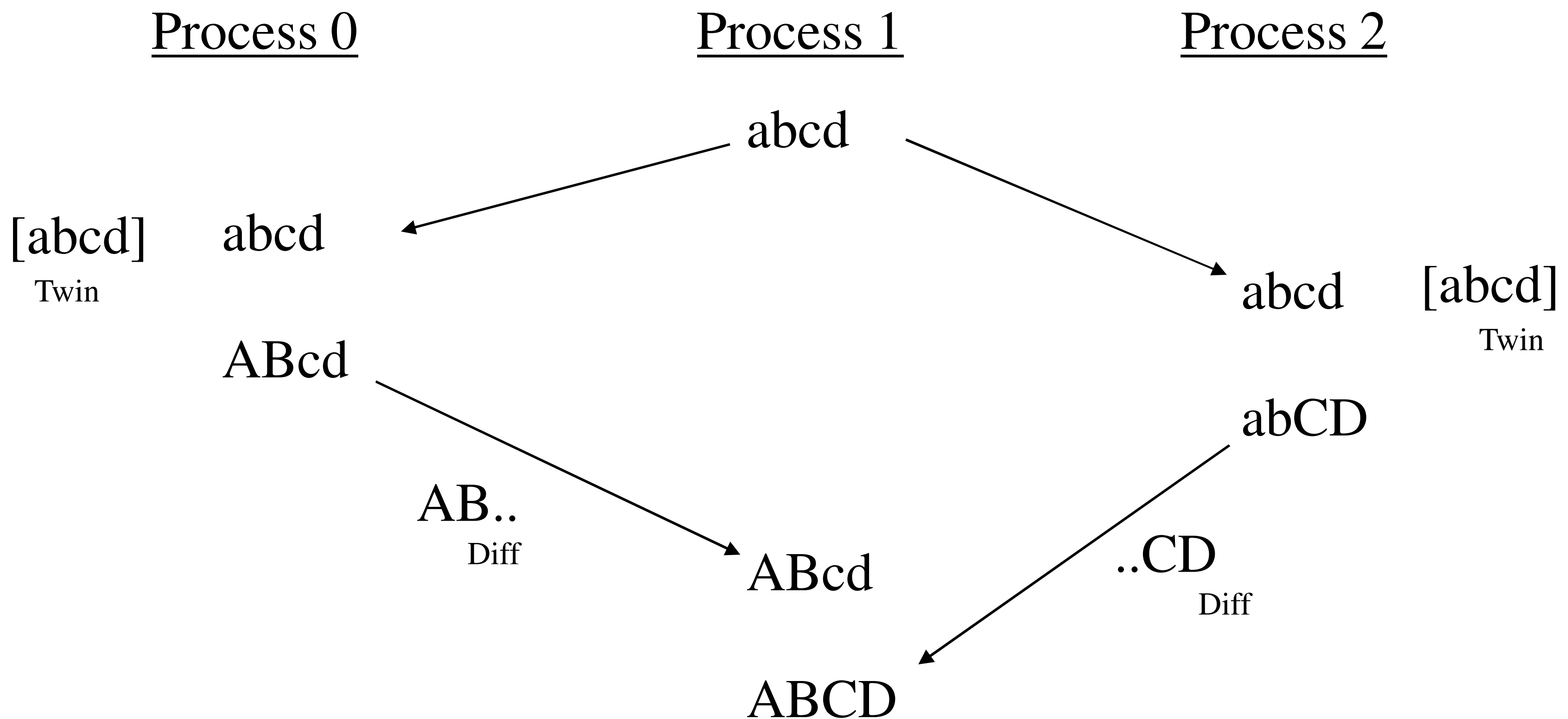
What About Writing!

False Sharing or Multiple Writers Problem



Like false cache line sharing – but much worse as each cache line is now the size of a memory page!

Twinning-and-Diffing



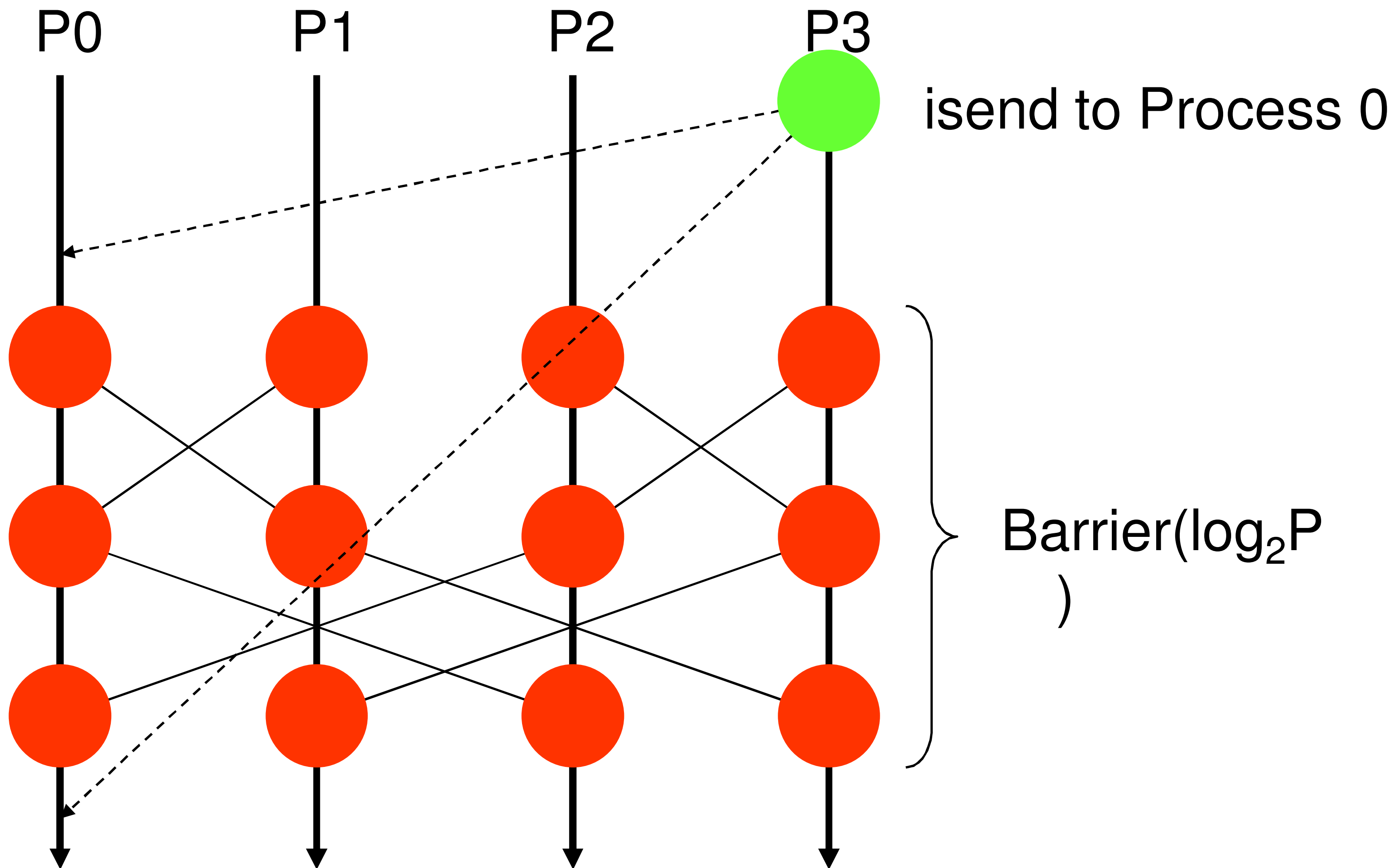
- On a write fault we make a copy and, when required, send changes relative to the copy

Updating Global Memory/ Consistency Points

SCASH Barrier (Consistency Point)

- Synchronise
- Flush
  - for each page written to, create a diff and send the diff to the page owner
  - for each diff received, apply the diff and add the page to a list of flushed pages (F)
- Synchronise
- Invalidate
  - For each page f in F, for each process p in f's copyset, tell p to invalidate page f
- Synchronise

Ordering Operations Before a Barrier
(A Software Fence Operation)



How do we fix this?

Cost of SCASH Page Faults

Pentium III Cluster with 100Mbit Ethernet Connection

Access	Permission Prior	Where	usec	Notify	Fetch	Twin
Write	Read	Home	8	-	-	-
Write	Read	Remote	51	Yes	-	Yes
Write	Unmapped	Remote	599	Yes	Yes	Yes
Read	Unmapped	Remote	587	Yes	Yes	-

- Compare to
  - memory latency 132nsec (LMBench)
  - MPI latency of ≈ 30usec

Comparative Cost of OpenMP Constructs

Elapsed time (usec)

	PIII (nodes x threads/node)			Sun V1280 (Threads)		
Directive	**1x1**	**2x1**	**4x1**	**1**	**2**	**4**
parallel	0.8	1762	13556	0.3	5.3	6.8
for	0.5	662	7731	0.5	2.1	2.9
parallel for	1.2	1797	13498	0.7	6.0	7.3
barrier	0.2	661	7305	0.1	1.1	1.8
single	0.9	4371	15330	0.1	0.9	1.3
critical	1.3	179	260	0.2	0.3	0.5
lock	0.6	52	74	0.2	0.4	0.5
ordered	1.7	1154	3712	0.2	0.6	0.6
atomic	1.3	3446	4776	0.1	0.5	0.8
reduction	1.1	29994	47836	0.5	5.5	7.8

Solution of 2D Laplace Equation

```
for (i=0; i<no_of_iterations; i++){
  for (y=0; y<no_of_rows; x++)
    for (x=0; x< no_of_columns; x++)
       new[x,y] = (old[x+1,y]+old[x-1,y]+
                   old[x,y+1]+old[x,y-1])/4.0
   tmp=new; new=old; old=tmp;
} /* next iteration */
```

	Implementation	Time (sec)			Speedup	
		1x1	2x1	4x1	2x1	4x1
0	Sequential	8.2	-	-		
1	Naïve	8.2	75.7	86.9	0.11	0.09
2	Barrier Opt.	8.2	63.7	101.1	0.13	0.08
3	Page Fault Opt.	8.2	6.9	5.3	1.19	1.55
4	Barrier&Page Opt.	8.2	6.7	4.3	1.22	1.91

Object Based DSM
Adsmith
(Slides from Wilkinson and Allen)

Adsmith

- User-level libraries that create distributed shared memory system on a cluster.
- Object based DSM - memory seen as a collection of objects that can be shared among processes on different processors.
  - much less likely to have false sharing
- Written in C++
- Built on top of PVM
- Freely available
  - I don't believe that it is being actively developed
- User writes application programs in C or C++ and calls Adsmith routines for creation of shared data and control of its access

Adsmith Uses Release Consistency

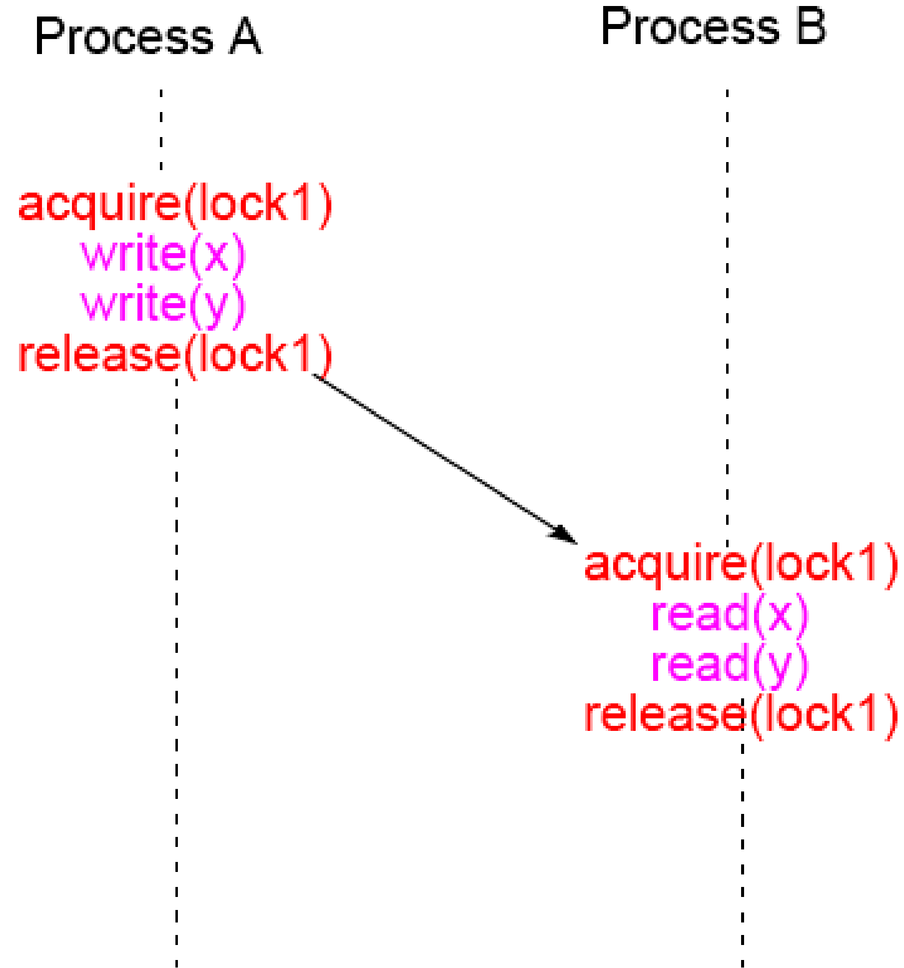
An extension of weak consistency in which the synchronization operations have been specified:

- acquire operation - used before a shared variable or variables are to be read.

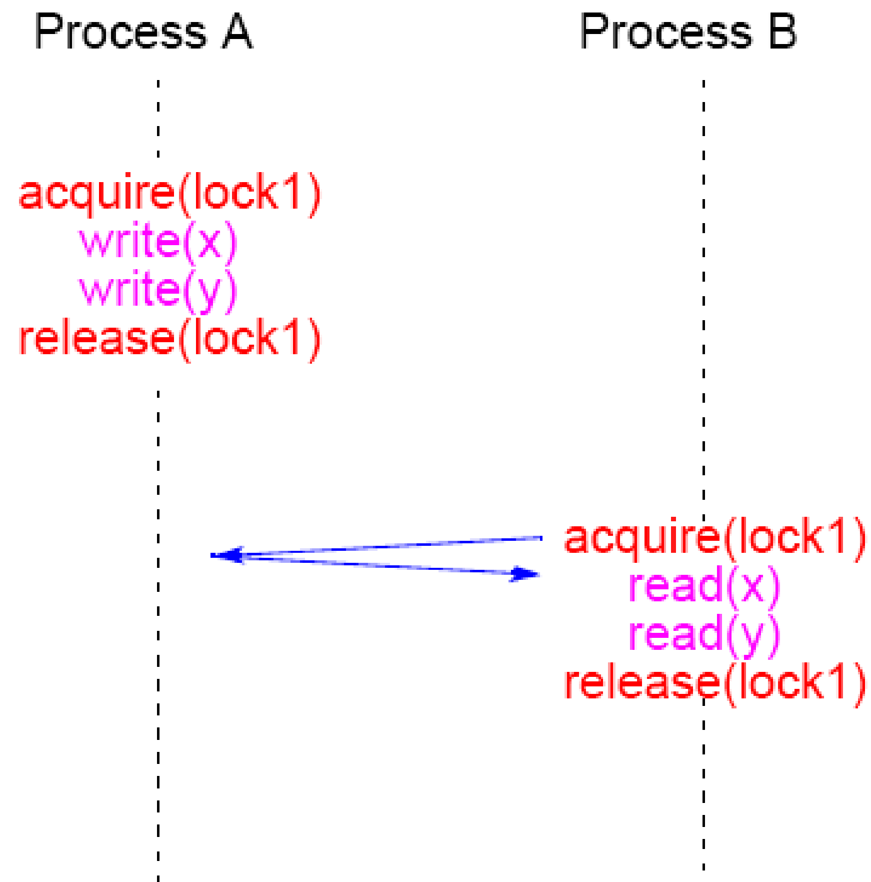
- release operation - used after the shared variable or variables have been altered (written) and allows another process to access to the variable(s)

Typically acquire is done with a lock operation and release by an unlock operation (although not necessarily).

Release Consistency



Lazy Release Consistency



- Delay consistency until an acquire posted
  - Gives fewer messages

Adsmith Routines

These notes are based upon material in Adsmith User Interface document.

Process

To start a new process or processes:

adsm_spawn(filename, count)

Example

adsm_spawn(“prog1”,10);

starts 10 copies of prog1 (10 processes). Must use Adsmith routine to start a new process. Also version of adsm_spawn() with similar parameters to pvm_spawn().

Process “join”

adsmith_wait();

will cause the process to wait for all its child processes (processes it created) to terminate.

Versions available to wait for specific processes to terminate, using pvm tid to identify processes. Then would need to use the pvm form of adsmith() that returns the tids of child processes.

Access to shared data (objects)

Adsmith uses "release consistency." Programmer explicitly needs to control competing read/write access from different processes.

Three types of access in Adsmith, differentiated by the use of the shared data:

- Ordinary Accesses - For regular assignment statements accessing shared variables.
- Synchronization Accesses - Competing accesses used for synchronization purposes.
- Non-Synchronization Accesses - Competing accesses, not used for synchronization.

Ordinary Accesses - Basic read/write actions

Before read, do:

adsm_refresh()

to get most recent value - an "acquire/load." After write, do:

adsm_flush()

to store result - "store"

Example

```
int *x;                    //shared variable
.
.
adsm_refresh(x);
a = *x + b;
```

Synchronization accesses

To control competing accesses:

- Semaphores
- Mutex’s (Mutual exclusion variables)
- Barriers.

available. All require an identifier to be specified as all three class instances are shared between processes.

Semaphore routines

Four routines:

wait()
signal()
set()
get().

```
class AdsmSemaphore {
	public:
		AdsmSemaphore( char *identifier, int init = 1 );
		void wait();
		void signal();
		void set( int value);
		void get();
};
```

Mutual exclusion variables – Mutex

Two routines

lock

unlock()

```
class AdsmMutex {
        public:
                AdsmMutex( char *identifier );
                void lock();
                void unlock();
};
```

Example

```
int *sum;
AdsmMutex x(“mutex”);
x.lock();
        adsm_refresh(sum);
        *sum += partial_sum;
        adsm_flush(sum);
x.unlock();
```

Barrier Routines

One barrier routine

barrier()

```
class AdsmBarrier {
        public:
                AdsmBarrier( char *identifier );
                void barrier( int count);
};
```

Example

```
AdsmBarrier barrier1("sample");
.
.
barrier1.barrier(procno);
```

Non-synchronization Accesses

For competing accesses that are not for synchronization:

adsm_refresh_now(void *ptr);

And

adsm_flush_now(void *ptr);

refresh and flush take place on home location (rather than locally) and immediately.

Linda

Linda

- Not a language, a small set of parallel extensions to an existing language
  - A method to create new threads running asynchronously to creating process
  - A shared ``tuple space'' that multiple threads can access
- Tuples space is a shared data space that can be accessed by any thread
  - Communication involves putting and retrieving data from tuple space
- Four basic operations supported (in addition to standard language
  - **eval:** spawns an additional asynchronous thread of execution that evaluates a tuple and places the answer in tuple space
  - **out:** places a tuple into tuple space
  - **in:** reads a tuple from tuple space but leaves the data there
  - **rd:** reads a tuple from tuple space but removes the tuple
- Linda programs are compiled with a Linda compiler
- Compiler and/or runtime environment are supposed(!) to optimise the actual physical location of data on distributed memory machines

```
real_main()
{  read_mats(rows,columns,a_rows,dim,b_cols)
   for (i=0, i<nworkers; i++) eval('worker',worker(a_rows,dim,b_cols));
   for (i=0, i<a_rows; i++)   out('row',i,rows[i]:dim)
   out ('columns',columns:dim*b_cols);
   out('task',0)
   for (i=0; i<l, i++)        in('result',i,?result[i]:len);
}
worker(a_row,dim,b_cols)
{  rd('columns',?columns:lens)
   while(1){
      in('task',?i);
      if (i > a_rows){
         out('task',i);
         break;
      }else{
         out('task',i+1);
      }
      rd('row',i,?row:len);
      for (j=0; j<b_cols; j++)result[j]=dot(row,columns[j*dim],dim);
      out('result',i,result:b_cols);
   }
return;}
```