Chapter 8

Shared Memory Systems and Programming

Topics:

- Regular shared memory systems and programming
- Distributed shared memory on a cluster
  (not done in 1st edition of textbook)

Programming with Shared Memory

Shared memory multiprocessor system

Any memory location can be accessible by any of the processors.

A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

Several Alternatives for Programming Shared Memory Multiprocessors:

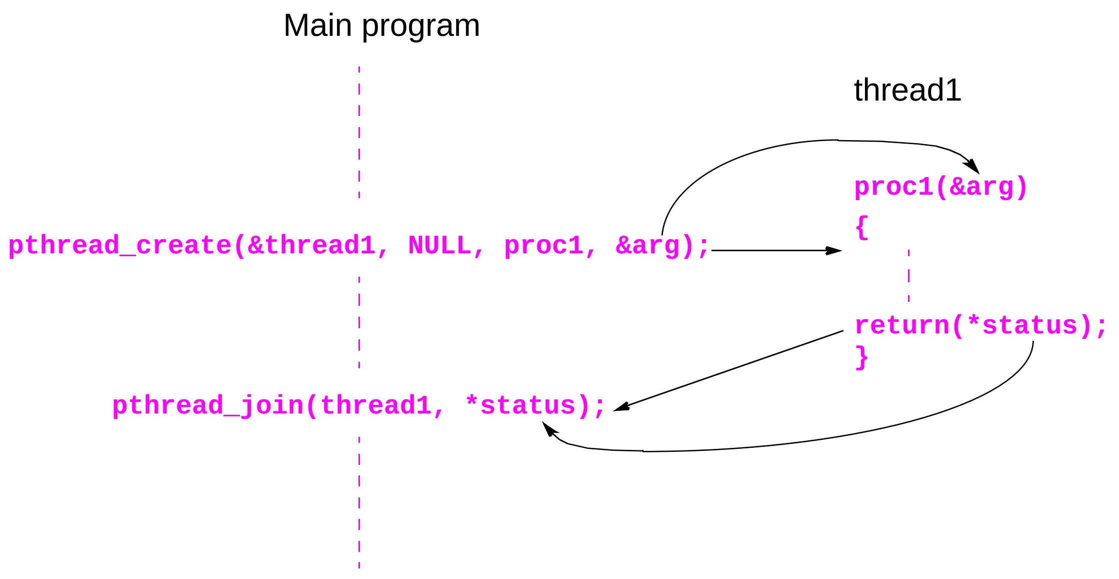
Using:

- Threads (Pthreads, Java, ..) in which the programmer decomposes the program into individual parallel sequences, each being thread, and each being able to access variables declared outside the threads.
- A sequential programming language with preprocessor compiler directives to declare shared variables and specify parallelism. Example OpenMP - industry standard
- A sequential programming language with user-level libraries to declare and access shared variables.
- A parallel programming language with syntax for parallelism, in which the compiler creates the appropriate executable code for each processor (not now common)
- A sequential programming language and ask a parallelizing compiler to convert it into parallel executable code. - also not now common

Pthreads

IEEE Portable Operating System Interface, POSIX, sec. 1003.1 standard

Executing a Pthread Thread


Main program
thread1
proc1(&arg)
{
pthread_create(&thread1, NULL, proc1, &arg);
return(*status);
}
pthread_join(thread1, *status);

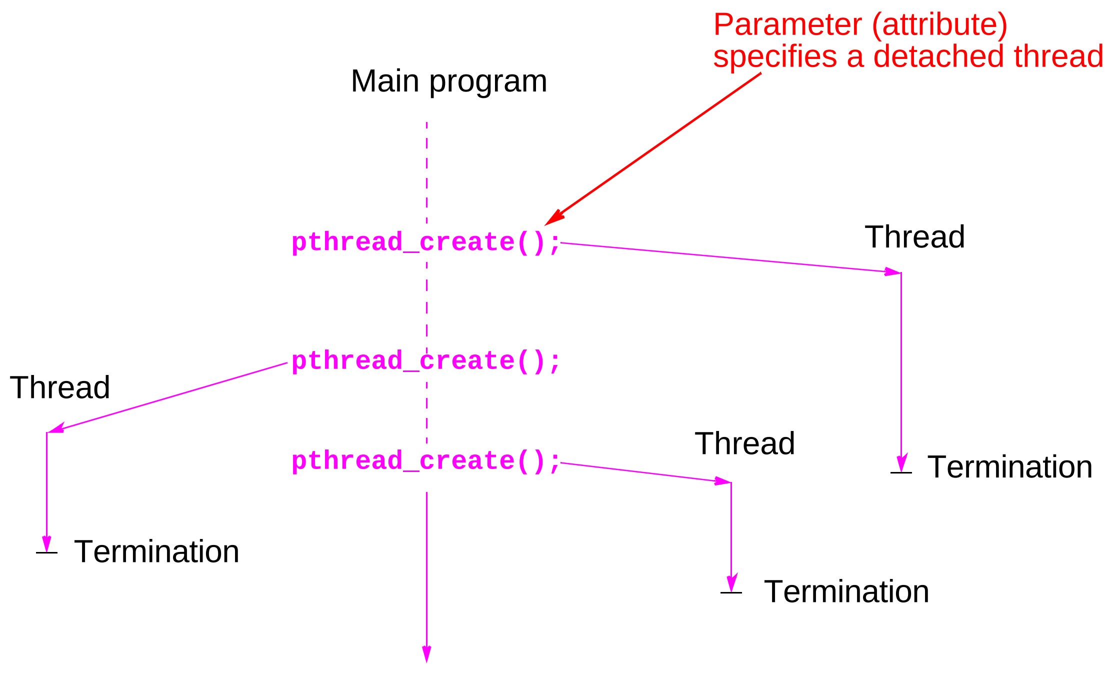
Detached Threads

It may be that thread are not bothered when a thread it creates terminates and then a join not needed.

Threads not joined are called *detached threads*.

When detached threads terminate, they are destroyed and their resource released.

Pthreads Detached Threads



Thread-Safe Routines

Thread safe if they can be called from multiple threads simultaneously and always produce correct results.

Standard I/O thread safe (prints messages without interleaving the characters).

System routines that return time may not be thread safe.

Routines that access shared data may require special care to be made thread safe.

Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called *critical sections* and arrange that only one such critical section is executed at a time

This mechanism is known as *mutual exclusion*.

Students usually have come across this concept in an operating system course. In that case, simply review in context of shared memory programming.

Locks

Simplest mechanism for ensuring mutual exclusion of critical sections.

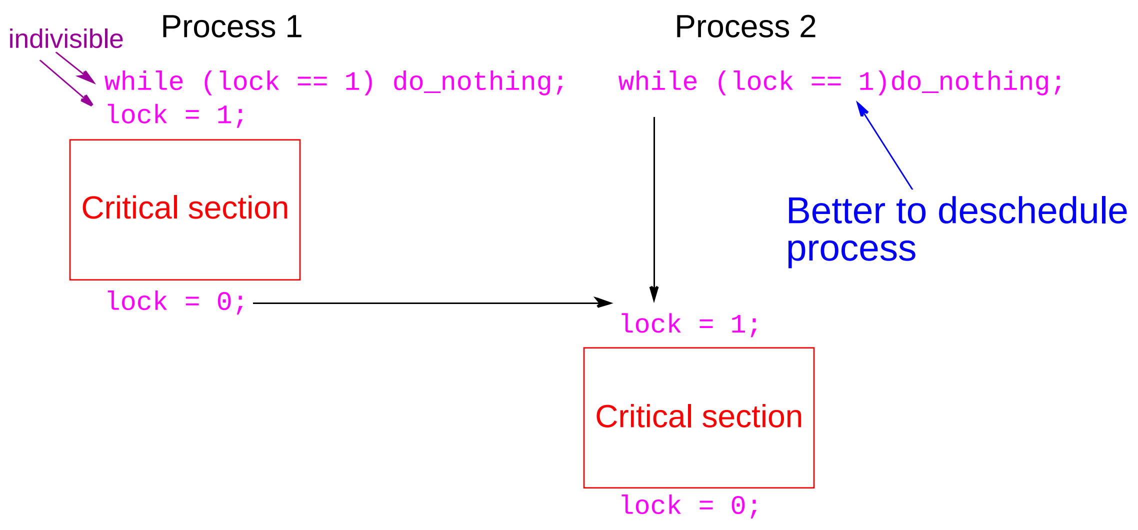
A lock is a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

Operates much like that of a door lock:

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Binary semaphores can achieve similar results, but usually locks are sufficient.

Control of critical sections through busy waiting



Process 1:

```
while (lock == 1) do_nothing;
lock = 1;
    Critical section
lock = 0;
```

Process 2:

```
while (lock == 1)do_nothing;
lock = 1;
    Critical section
lock = 0;
```

Pthread Lock Routines

Locks are implemented in Pthreads with *mutually exclusive lock variables*, or "mutex" variables:

```
                .
    pthread_mutex_lock(&mutex1);
        critical section
    pthread_mutex_unlock(&mutex1);
                .
```

If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open. If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed. Only the thread that locks a mutex can unlock it.

Critical Sections Serializing Code

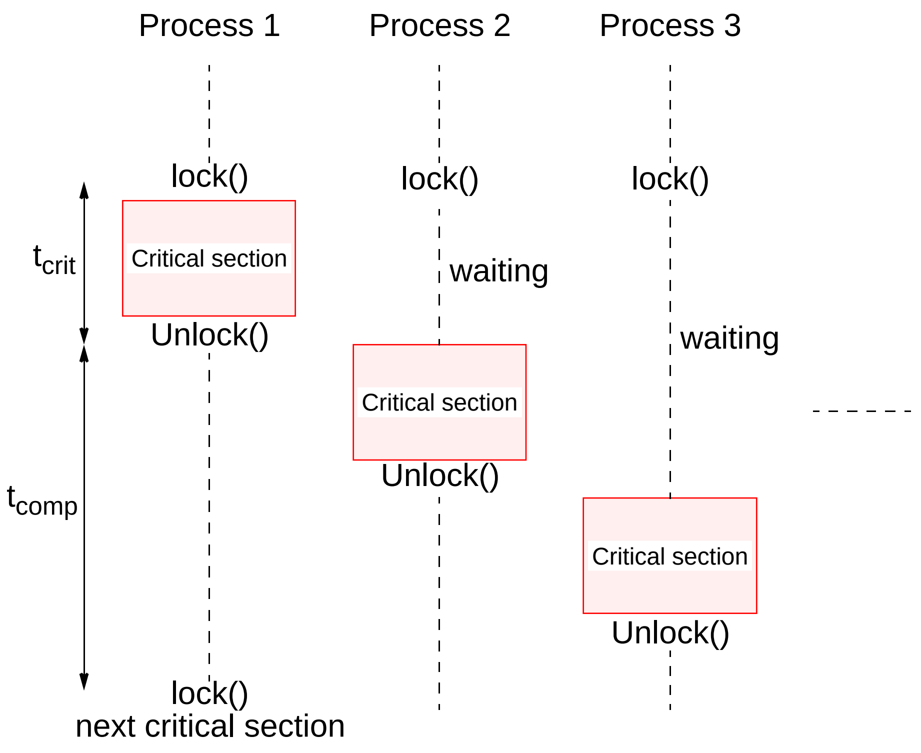
High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.

Illustration

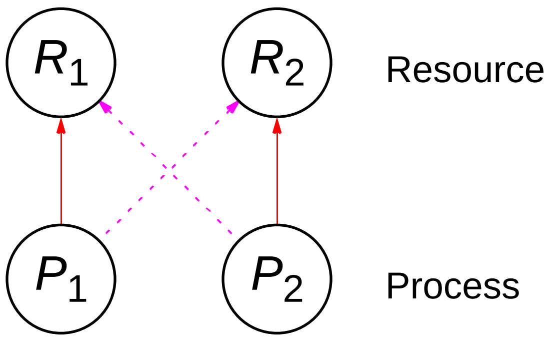


When $t_{comp} < pt_{crit}$, less than p processor will be active

Deadlock

Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Two-process deadlock



Can also occur in a circular fashion with several processes having a resource wanted by another.

Pthreads

Offers one routine that can test whether a lock is actually closed without blocking the thread:

```
pthread_mutex_trylock()
```

This routine will lock an unlocked mutex and return 0 or will return with **`EBUSY`** if the mutex is already locked – might find a use in overcoming deadlock.

Monitor

Suite of procedures that provides only way to access shared resource. Only one process can use a monitor procedure at any instant.

Could be implemented using a semaphore or lock to protect its entry; i.e.,

Java, of course, has this built-in.

```
monitor_proc1()
{
  lock(x);
      .
    monitor body
      .
  unlock(x);
  return;
}
```

Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

This is a very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variables*.

Pthread Condition Variables

Associated with a specific mutex. Given declarations:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:

```
action()                              counter()
{                                       {
         .                                  .
         .                                  .
pthread_mutex_lock(&mutex1);            pthread_mutex_lock(&mutex1);
while (c <> 0)                          c--;
  pthread_cond_wait(cond1,mutex1);      if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);          pthread_mutex_unlock(&mutex1);
take_action();                             .
         .                                 .
         .                                 .
}                                       }
```

Signals are *not* remembered - threads must already be waiting for a signal to receive it.

Basic Tenets of Shared Memory Programming

- Protect access to shared data through use of critical sections, locks, semaphores, etc.
- Synchronize processes using barriers

Shared Memory Program Examples

In textbook, various ways to to sum the elements of an array, using processes and threads.

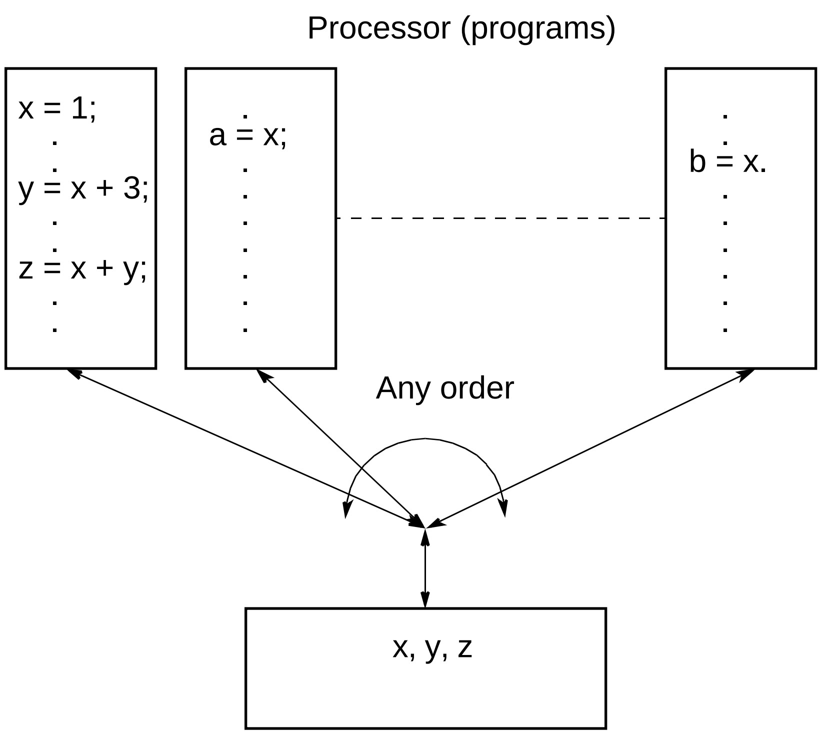
Sequential Consistency

Formally defined by Lamport (1979):

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processors occur in this sequence in the order specified by its program.

i.e. the overall effect of a parallel program is not changed by any arbitrary interleaving of instruction execution in time.

Sequential Consistency



Writing a parallel program for a system which is known to be sequentially consistent enables us to reason about the result of the program.

Example

```
Process P1                Process 2
    .                         .
data = new;                   .
flag = TRUE;                  .
    .                         .
    .                     while (flag != TRUE) { };
    .                     data_copy = data;
    .                         .
```

Expect data_copy to be set to new because we expect the statement data = new to be executed before flag = TRUE and the statement while (flag != TRUE) { } to be executed before data_copy = data. Ensures that process 2 reads new data from another process 1. Process 2 will simple wait for the new data to be produced.

Program Order

Sequential consistency refers to “operations of each individual processor .. occur in the order specified in its program” or program order.

In previous figure, this order is that of the stored machine instructions to be executed.

Compiler Optimizations

The order is not necessarily the same as the order of the corresponding high level statements in the source program as a compiler may reorder statements for improved performance. In this case, the term program order will depend upon context, either the order in the souce program or the order in the compiled machine instructions.

High Performance Processors

Modern processors usually reorder machine instructions internally during execution for increased performance.

This does not alter a multiprocessor being sequential consistency, if the processor only produces the final results in program order (that is, retires values to registers in program order which most processors do).

All multiprocessors will have the option of operating under the sequential consistency model. However, it can severely limit compiler optimizations and processor performance.

Example of Processor Re-ordering

```
Process P1                Process 2
    .                         .
new = a * b;                  .
data = new;                   .
flag = TRUE;                  .
    .                         .
    .                     while (flag != TRUE) { };
    .                     data_copy = data;
    .                         .
```

Multiply machine instruction corresponding to new = a * b is issued for execution. The next instruction corresponding to data = new cannot be issued until the multiply has produced its result. However the next statement, flag = TRUE, is completely independent and a clever processor could start this operation before the multiply has completed leading to the sequence:

```
Process P1                Process 2
    .                         .
new = a * b;                  .
flag = TRUE;                  .
data = new;                   .
    .                         .
    .                     while (flag != TRUE) { };
    .                     data_copy = data;
    .                         .
```

Now the while statement might occur before new is assigned to data, and the code would fail.

All multiprocessors will have the option of operating under the sequential consistency model, i.e. not reorder the instructions and forcing the multiply instruction above to complete before starting the subsequent instruction which depend upon its result.

Relaxing Read/Write Orders

Processors may be able to relax the consistency in terms of the order of reads and writes of one processor with respect to those of another processor to obtain higher performance, and instructions to enforce consistency when needed.

Examples

Alpha processors

Memory barrier (MB) instruction - waits for all previously issued memory accesses instructions to complete before issuing any new memory operations.

Write memory barrier (WMB) instruction - as MB but only on memory write operations, i.e. waits for all previously issued memory write accesses instructions to complete before issuing any new memory write operations - which means memory reads could be issued after a memory write operation but overtake it and complete before the write operation. (check)

SUN Sparc V9 processors

memory barrier (MEMBAR) instruction with four bits for variations

Write-to-read bit prevent any reads that follow it being issued before all writes that precede it have completed. Other: Write-to-write, read-to-read, read-to-write.

IBM PowerPC processor

SYNC instruction - similar to Alpha MB instruction (check differences)

Weak Consistency

Synchronization operations are used by the programmer whenever it is necessary to enforce sequency consistency. The compiler/ processor is allowed in other places to reorder instructions without regard to sequential consistency.

This is quite reasonable model since any accesses to shared data should be provided with synchronization operations (locks etc.).

We will see more about this in the next section when considering shared memory programming on a cluster(distributed shared memory systems).

Distributed Shared Memory

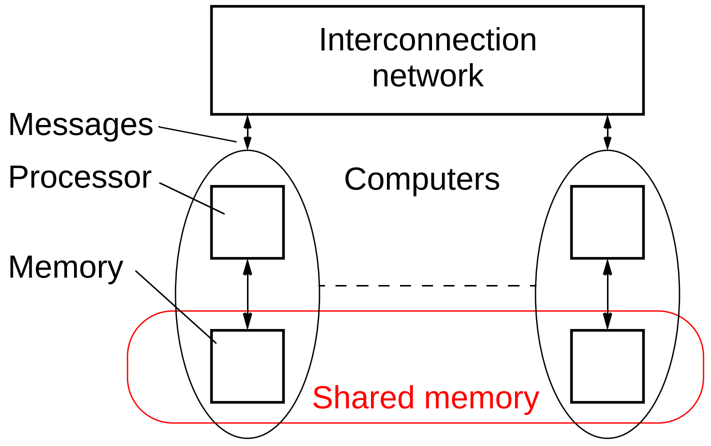
Distributed Shared Memory

Making the main memory of a cluster of computers look as though it is a single memory with a single address space.

Then can use shared memory programming techniques.

DSM System

Still need messages or mechanisms to get data to processor, but these are hidden from the programmer:



Advantages of DSM

- System scalable
- Hides the message passing - do not explicitly specific sending messages between processes
- Can us simple extensions to sequential programming
- Can handle complex and large data bases without replication or sending the data to processes

Disadvantages of DSM

- May incur a performance penalty
- Must provide for protection against simultaneous access to shared data (locks, etc.)
- Little programmer control over actual messages being generated
- Performance of irregular problems in particular may be difficult

Methods of Achieving DSM

- Hardware

  Special network interfaces and cache coherence circuits

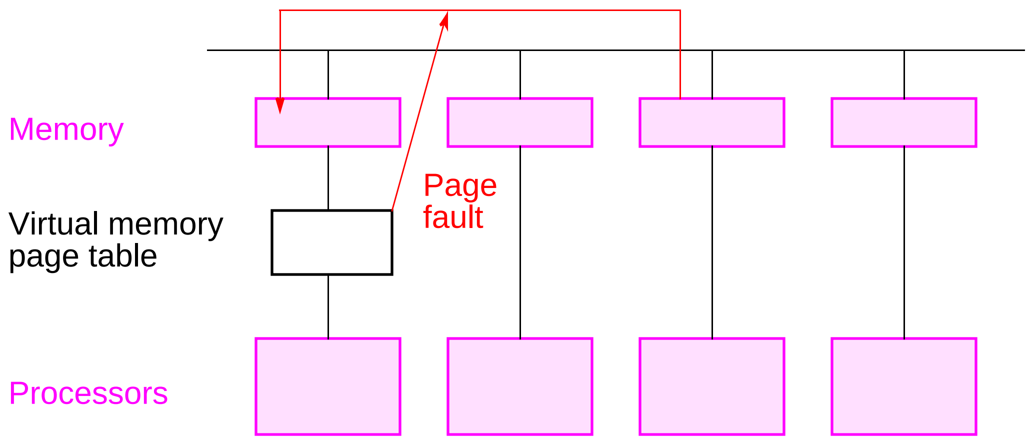
- Software

  Modifying the OS kernel
  Adding a software layer between the operating system and the application - most convenient way for teaching purposes

Software DSM Implementation

- Page based - Using the system’s virtual memory
- Shared variable approach- Using routines to access shared variables
- Object based- Shared data within collection of objects. Access to shared data through object oriented discipline (ideally)

Sofware Page Based DSM Implementation



Some Software DSM Systems

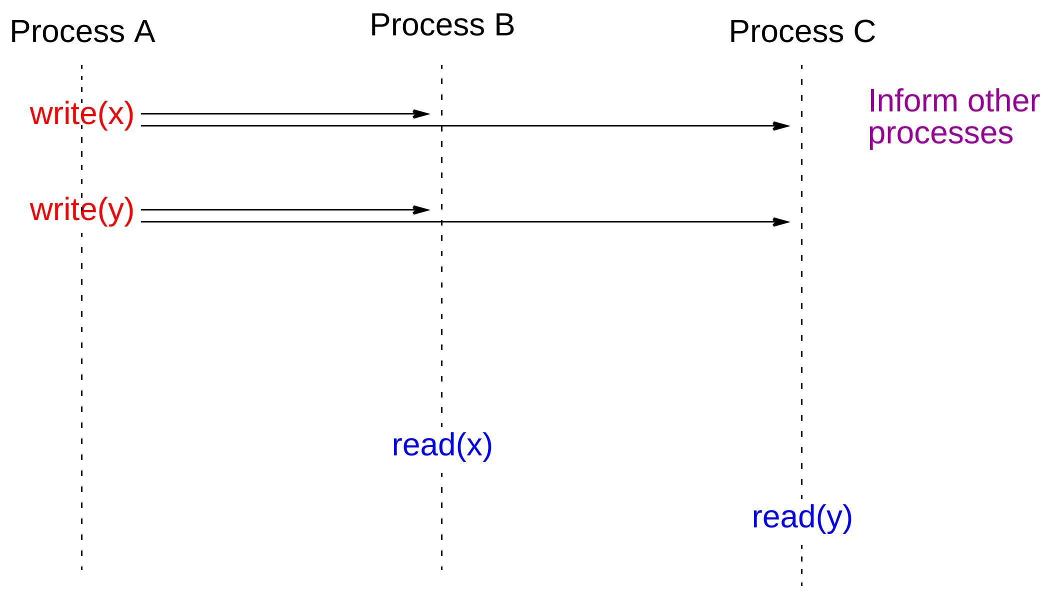
- Treadmarks
  - Page based DSM system
  - Apparently not now available
- JIAJIA
  C based
  - Obtained at UNCC but required significant modifications for our system (in message-passing calls)
- Adsmith object based
  - C++ library routines
  - We have this installed on our cluster - chosen for teaching

Consistency Models

- Strict Consistency - Processors sees most recent update, i.e. read returns the most recent wrote to location.
- Sequential Consistency - Result of any execution same as an interleaving of individual programs.
- Relaxed Consistency- Delay making write visible to reduce messages.
- Weak consistency - programmer must use synchronization operations to enforce sequential consistency when necessary.
- Release Consistency - programmer must use specific synchronization operators, acquire and release.
- Lazy Release Consistency - update only done at time of acquire.

Strict Consistency

Every write immediately visible



Disadvantages: number of messages, latency, maybe unnecessary.

Consistency Models used on DSM Systems

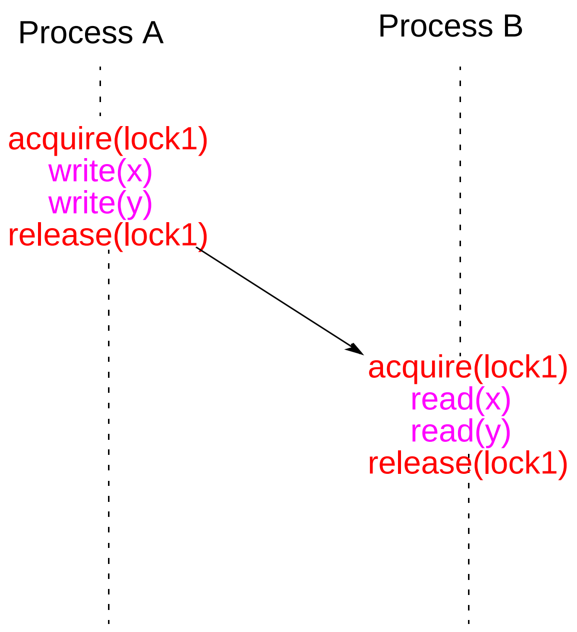
Release Consistency

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)

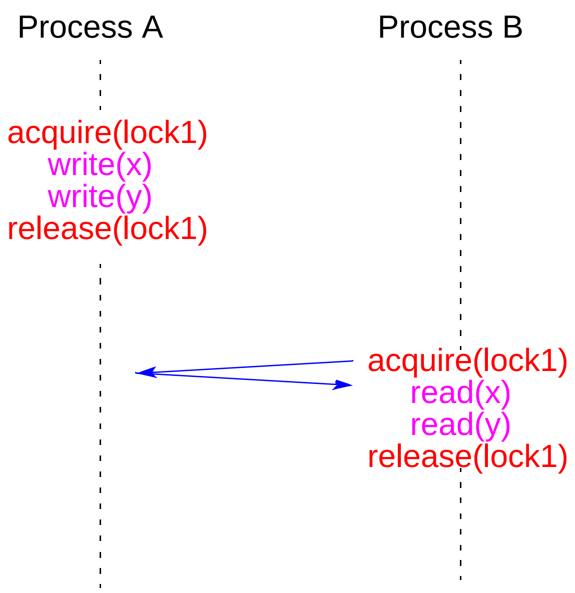
Typically acquire is done with a lock operation and release by an unlock operation (although not necessarily).

Release Consistency



Lazy Release Consistency

Get modification at acquire time rather than sent at release time.



Advantages: Fewer messages

Adsmith

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.
- Written in C++
- Built on top of pvm
- Freely available - installed on UNCC cluster

User writes application programs in C or C++ and calls Adsmith routines for creation of shared data and control of its access.
If required, can also call pvm routines in same program.

Adsmith Routines

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

Initialization/Termination

Explicit initialization/termination of Adsmith not necessary.

Process Creation

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.

Features to Improve Performance

Routines that can be used to overlap messages or reduce number of messages:

- Prefetch
- Bulk Transfer
- Combined routines for critical sections

Prefetch

```
adsm_prefetch( void *ptr )
```

used before adsm_refresh() to get data as early as possible.

Non-blocking so that can continue with other work prior to issuing refresh.

Bulk Transfer

Combines consecutive messages to reduce number. Can apply only to “aggregating”:

```
adsm_malloc( AdsmBulkType *type );
adsm_prefetch( AdsmBulkType *type )
adsm_refresh( AdsmBulkType *type )
adsm_flush( AdsmBulkType *type )
```

where AdsmBulkType is defined as:

```
enum AdsmBulkType {
     adsmBulkBegin,
     AdsmBulkEnd
}
```

Use parameters **`AdsmBulkBegin`** and **`AdsmBulkEnd`** in pairs to “aggregate” actions.
Easy to add afterwards to improve performance.

Example

```
adsm_refresh(AdsmBulkBegin);
     adsm_refresh(x);
     adsm_refresh(y);
     adsm_refresh(z);
adsm_refresh(AdsmBulkEnd);
```

Routines to improve performance of critical sections

Called “Atomic Accesses” in Adsmith.

```
adsm_atomic_begin()
adsm_atomic_end()
```

Replaces two routines and reduces number of messages.

```
Acquire    ↕    adsm_atomic_begin()
Refresh
local code
Flush
Release    ↕    adsm_atomic_end()
```

Sending an expression to be executed on home process

Can reduce number of messages. Called “Active Access” in Adsmith. Achieved with:

```
adsm_atomic(void *ptr, char *expression);
```

where the expression is written as [type] expression.

Object pointed by ptr is the only variable allowed in the expression (and indicated in this expression with the symbol @).

Example

```
int *x = (int*)adsm_malloc)”x”,sizeofint(int));
adsm_atomic(x,”[int] @=@+10”);
```

Collect Access

Efficient routines for shared objects used as an accumulator:

```
void adsm_collect_begin(void *ptr, int num);
void adsm_collect_end(void *ptr);
```

where num is the number of processes involved in the access, and *ptr points to the shared accumulator

Example

(from page 10 of Adsmith User Interface document):

```
int partial_sum = ... ;  // calculate the partial sum
adsm_collect_begin(sum,nproc);
sum+=partial_sum;             //add partial sum
adsm_collect_end(sum);   //total; sum is returned
```

Other Features

Pointers

Can be shared but need to use adsmith address translation routines to convert local address to a globally recognizable address and back to an local address:

To translates local address to global address (an int)

```
int adsm_gid(void *ptr);
```

To translates global address back to local address for use by requesting process

```
void *adsm_attach(int gid);
```

Message passing

Can use PVM routines in same program but must use adsm_spawn() to create processes (not pvm_spawn(). Message tags MAXINT-6 to MAXINT used by Adsmith.

Information Retrieval Routines

For getting host ids (zero to number of hosts -1) or process id (zero to number of processes -1):

```
int adsm_hostno(int procno = -1);
```

- Returns host id where process specified by process number procno resides. (If procno not specified, returns host id of calling process).

```
int adsm_procno();
```

- Returns process id of calling process.

```
int adsm_procno2tid(int procno);
```

- Translates process id to corresponding PVM task id.

```
int adsm_tid2procno(int tid)
```

translates PVM task id to corresponding process id.

UNCC DSM Home page

Link from the main parallel programming home page:

http://www.cs.uncc.edu/par_prog

Gives instructions on setting directories etc.

Adsmith User Instructions

Adsmith only supports PVMs of a single architecture, so it will work with only Linux-only or Solaris-only virtual machines.

1) Make sure that you're set up to use PVM first off.

Adsmith User Instructions continued

Compilation

2) Compile adsmith runtime support + examples into one's home directory:

```
% cd /afs/uncc/cs/linux/apps20/parallel/adsmith
% make
```

Compiles adsmd, adsmgate, plus demo programs into $HOME/pvm3/bin/$PVM_ARCH/. adsmd and adsmgate are required to be in $HOME/pvm3/bin/$PVM_ARCH.

3) To build your own adsmith + pvm programs, use one of the example's aimk makefiles.

Examples installed in: /afs/uncc/cs/linux/apps20/parallel/adsmith/

Adsmith User Instructions continued

Examples

Makefile: use ...examples/hello/Makefile.aimk, modified for your program name. The GNU C++ compiler (g++) must be used to compile the code, as that it was the compiler that was used to compile the adsmith libraries. All of the example aimkfiles are set up to explicitly use g++ -- do not reset this value.

Adsmith User Instructions continued

Runtime

4) To run an example (matmul):

a) Invoke pvm console, and add as many machines as you wish. Machines must be the same architecture as the machine that you first invoke "pvm" on. If you're sitting at a Linux terminal and wish to make use of the PVM cluster machines, be sure to invoke the pvm command from a "Solaris Terminal" window. After adding the machines, quit (not halt!) out of the pvm console.

b) Invoke the example program "matmul", which should have been compiled in step 2.

Adsmith User Instructions continued

Sample output

```
% cd ~/pvm3/bin/LINUX
/afs/uncc.edu/usr6/jlrobins/pvm3/bin/LINUX
% ./matmul
2 host detected
1: my range is 0 to 10
2: my range is 10 to 20
Time Computing and Refresh = 0.085689 sec
Time Including Prefetch = 0.086026 sec
```

DSM Implementation Projects

Using underlying message-passing software

- Easy to do
- Can sit on top of message-passing software such as MPI or PVM
- Several UNCC student projects currently doing this in different ways

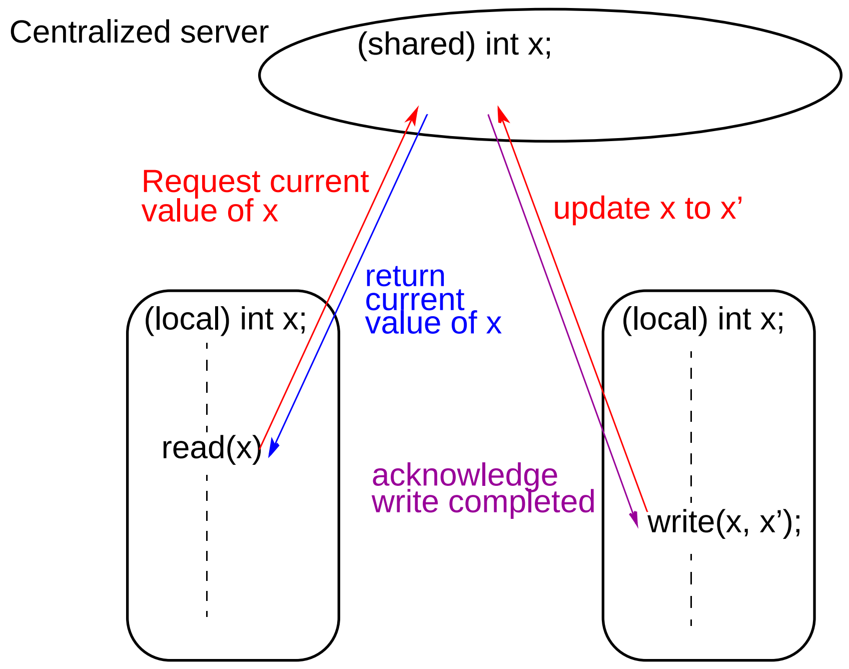
Issues in Implementing a DSM System

- Managing shared data - reader/writer policies
- Timing issues - relaxing read/write orders

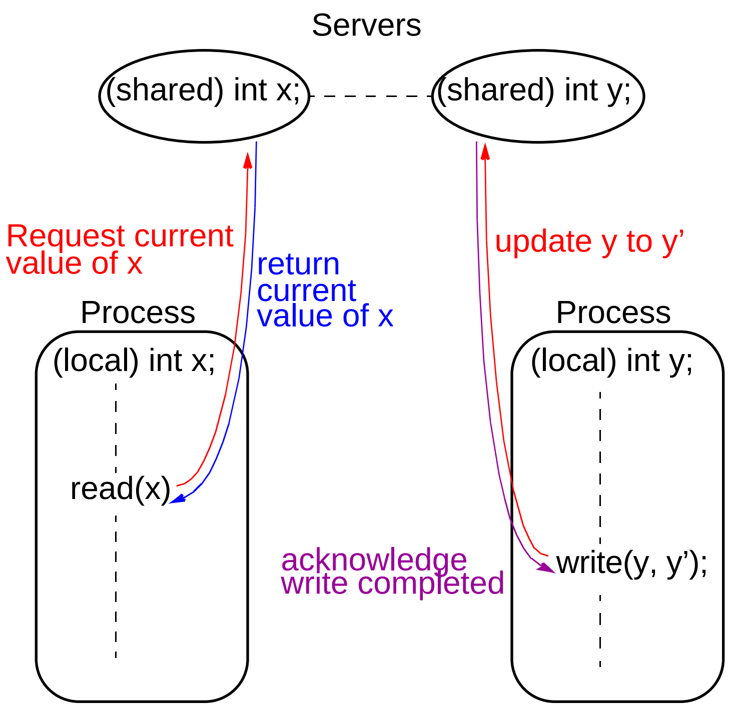
Reader/Writer Policies

- Single reader/single writer policy - simple to do with centralized servers
- Multiple reader/single writer policy - again quite simple to do
- Multiple reader/multiple writer policy - tricky

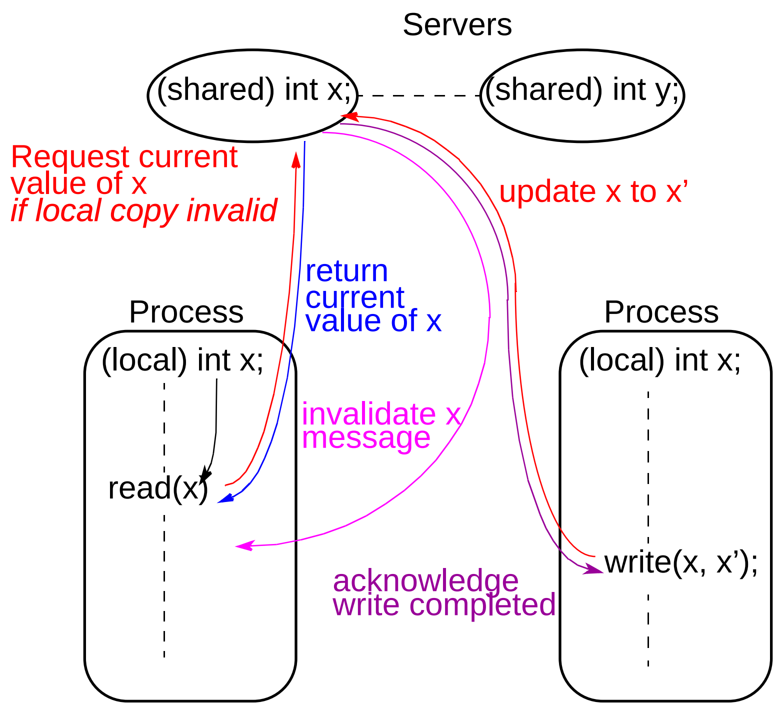
Simple DSM system using a centralized server



Simple DSM system using multiple servers



Simple DSM system using multiple servers and multiple reader policy



Shared Data with Overlapping Regions A New Concept Developed at UNCC

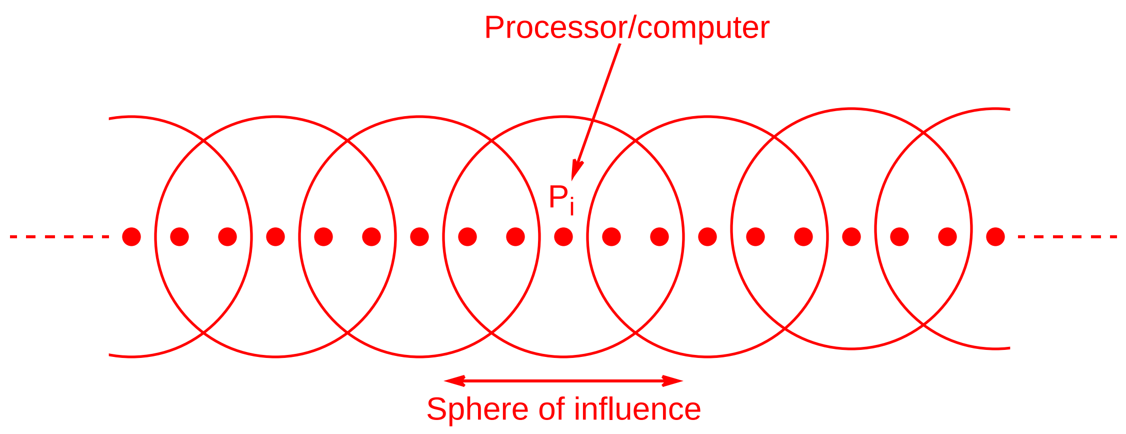
Based upon earlier work on so-called over-lapping connectivity interconnection networks

A large family of scalable interconnection networks devised - all have characteristic of overlapping domains that nodes can interconnect

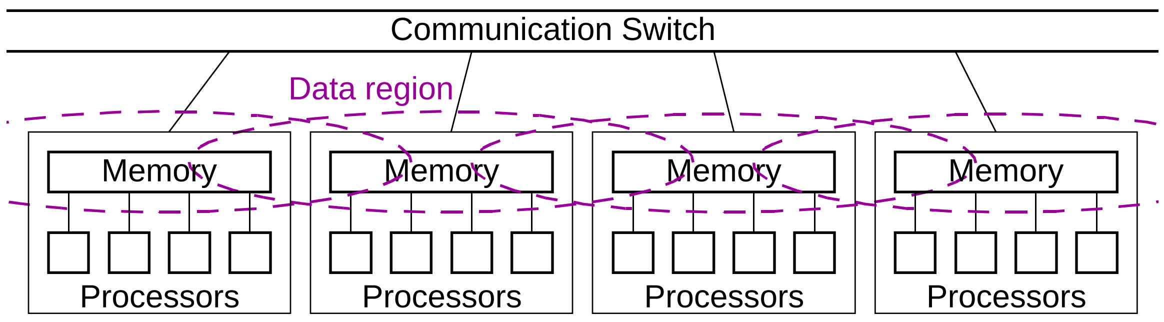
Many applications require communication to logically nearby processors

Overlapping Regions

Example



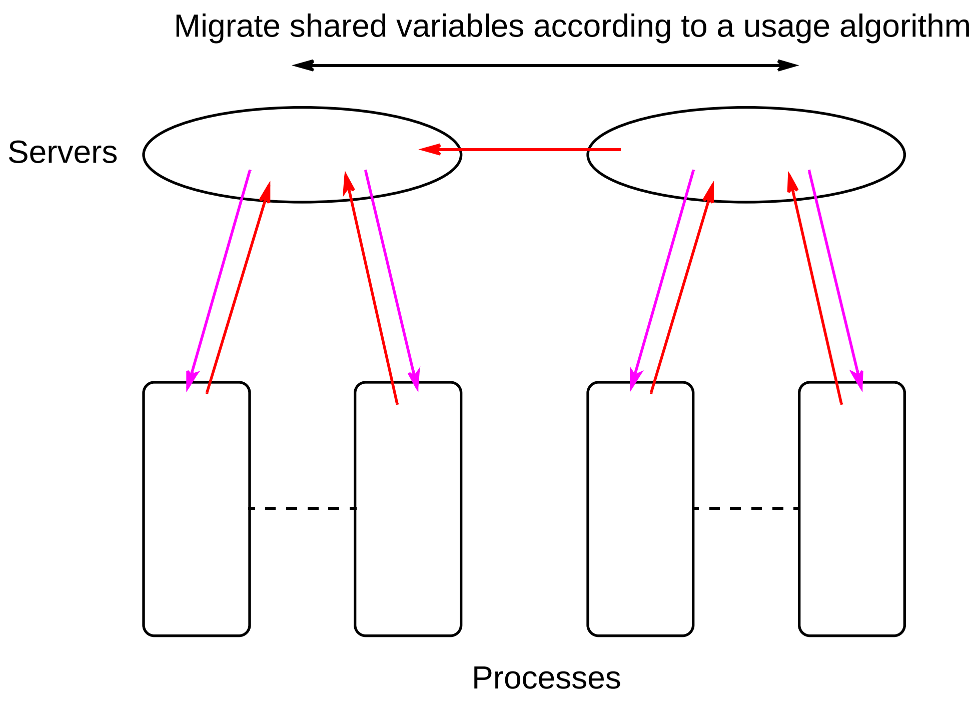
Symmetrical Multiprocessor System with Overlapping Data Regions



Static and Dynamic Overlapping Groups

- Static - defined prior to program execution - add routines for declaring and specifying these groups
- Dynamic - shared variable migration during program execution

Shared Variable Migration between Data Regions



UNCC DSM Systems

Several competing ones being developed:

- Using PVM as underlying message-passing software with C routines (Done by previous senior project students.)
- Using MPI as underlying message-passing software with C++ routines
- 3 versions, one is embodying static and dynamic overlapping groups
- Using MPI as underlying message-passing software with Java routines

DSM Projects

- Write a DSM system in C using PVM for the underlying message-passing and process communication.
- Write a DSM system in C++ using MPI for the underlying message-passing and process communication.
- Write a DSM system in Java using MPI for the underlying message-passing and process communication.
- (More advanced) One of the fundamental disadvantages of software DSM system is the lack of control over the underlying message passing. Provide parameters in a DSM routine to be able to control the message-passing. Write routines that allow communication and computation to be overlapped.