Programming with Shared Memory

Nguyễn Quang Hùng

Outline

- Introduction
- Shared memory multiprocessors
- Constructs for specifying parallelism
  - Creating concurrent processes
  - Threads
- Sharing data
  - Creating shared data
  - Accessing shared data
  - Language constructs for parallelism
  - Dependency analysis
  - Shared data in systems with caches
- Examples
  - Pthreads example
- Exercises

Introduction

- This section focuses on programming on shared memory system (e.g SMP architecture).
- Programming mainly discusses on:
  - Multi-processes: Unix/Linux fork(), wait()…
  - Multithreads: IEEE Pthreads, Java Thread…

Multiprocessor system

- Multiprocessor systems: two types
  - Shared memory multiprocessor.
  - Message-passing multicomputer.
  - *In "Parallel programming:Techniques & applications using networked workstations & parallel computing" book.*
- Shared memory multiprocessor:
  - SMP-based architecture: IBM RS/6000, Big BLUE/Gene supercomputer, etc.

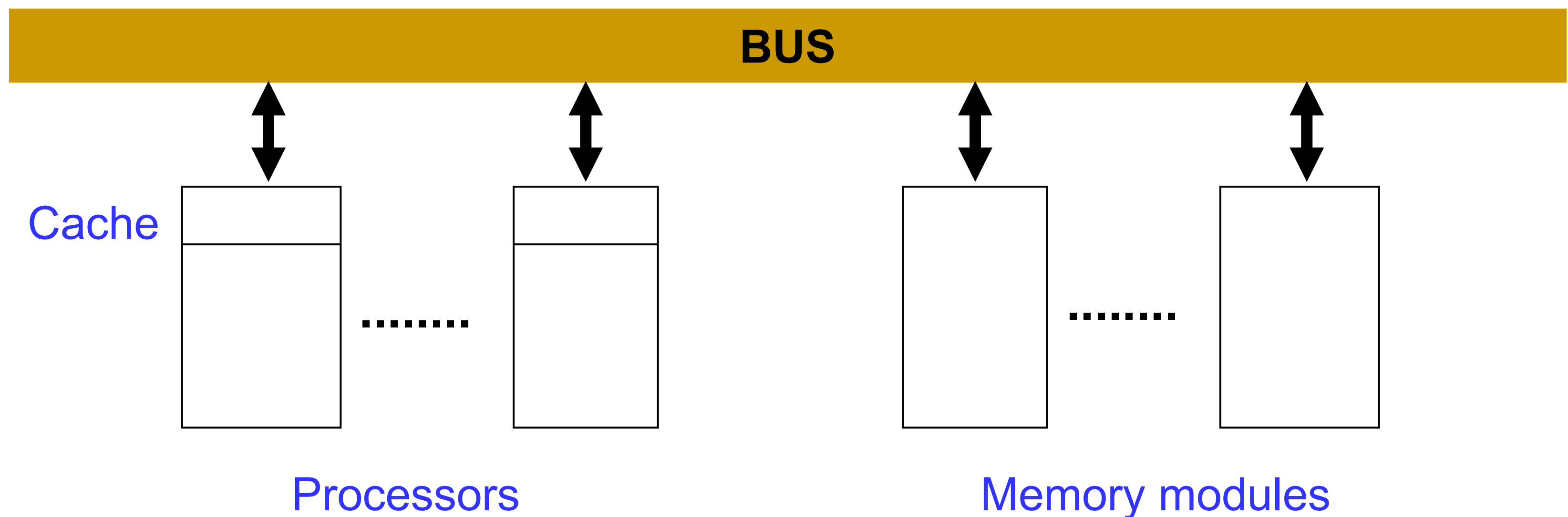
Read more & report:

IBM RS/6000 machine.
http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html
http://docs.hp.com/en/B6056-96002/ch01s01.html

Shared memory multiprocessor system

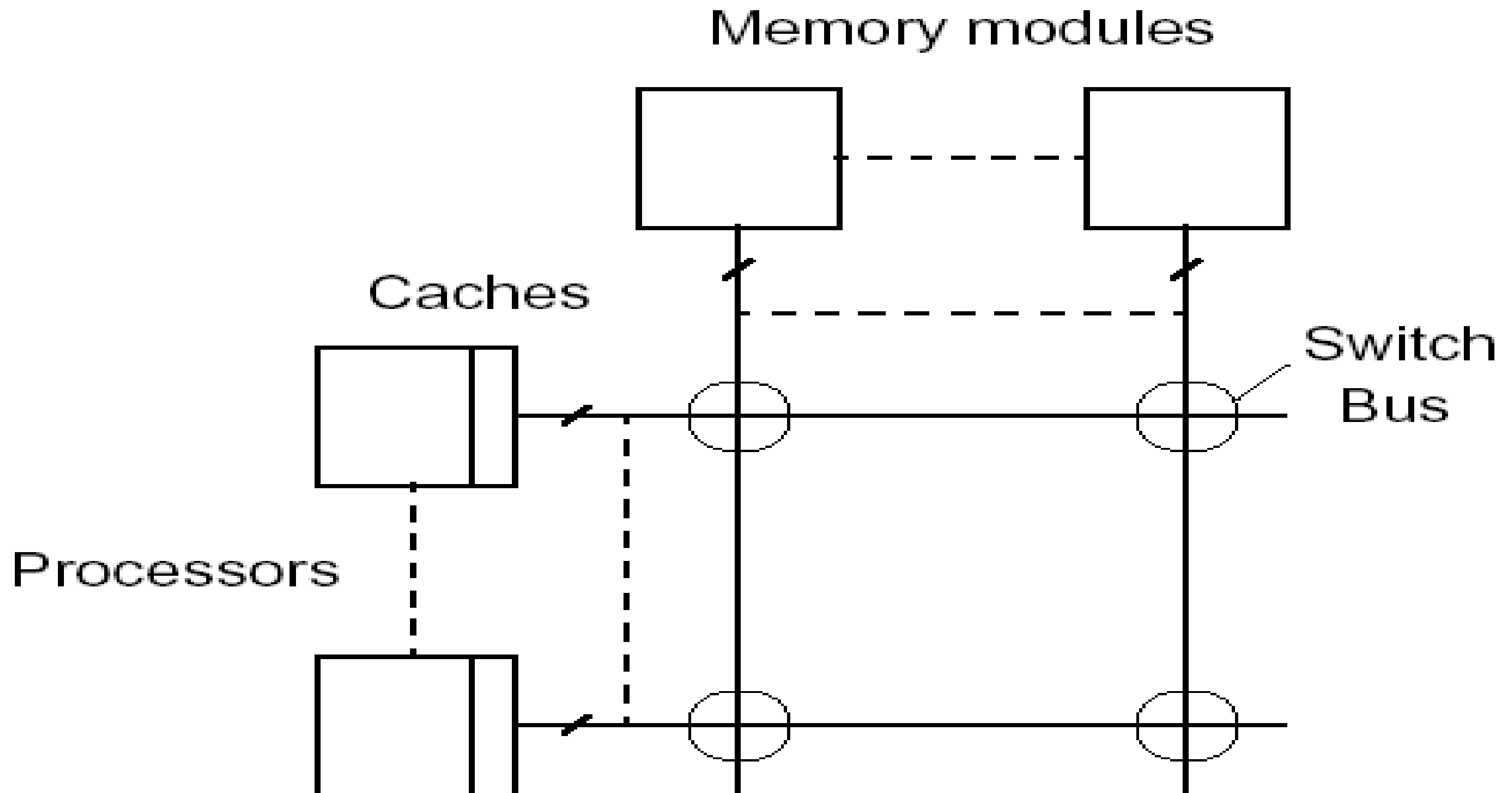
- Based on SMP architecture.
- Any memory location can be accessible by any of the processors.
- A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.
- Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections: semaphore, lock, monitor…).

Shared memory multiprocessor using a single bus

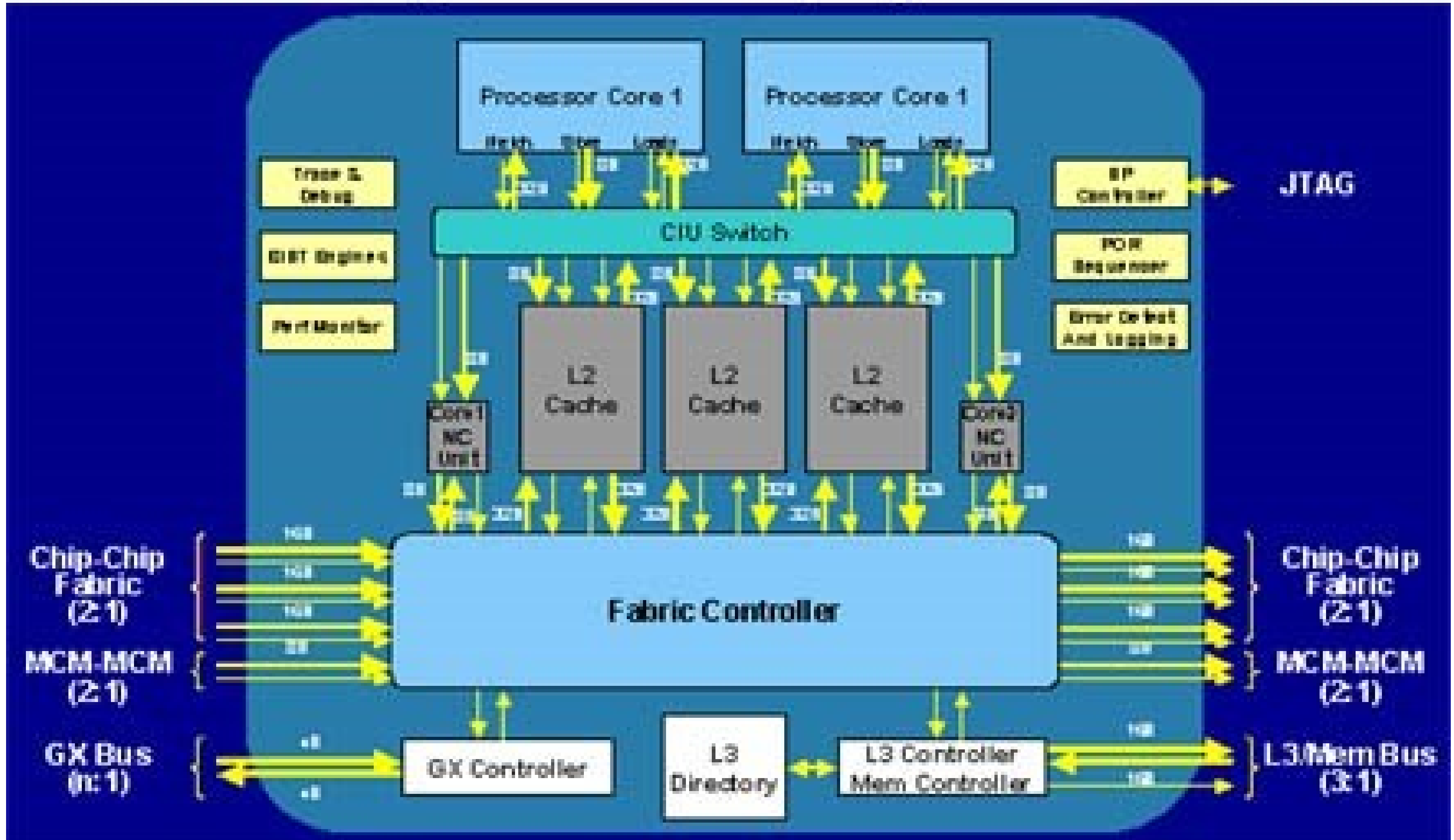


- A small number of processors. Perhaps, Up to 8 processors.
- Bus is used by one processor at a time. Bus contention increases by #processors.

Shared memory multiprocessor using a crossbar switch



IBM POWER4 Chip logical view



Source: www.ibm.com

Several alternatives for programming shared memory multiprocessors

- Using library routines with an existing sequential programming language.
  - Multiprocesses programming:
    - **fork(), execv()…**
  - Multithread programming:
    - **IEEE Pthreads library**
    - **Java Thread. http://java.sun.com**

- Using a completely new programming language for parallel programming - not popular.
  - High Performance Fortran, Fortran M, Compositional C++….

- Modifying the syntax of an existing sequential programming language to create a parallel programming language. Using an existing sequential programming language supplemented with compiler directives for specifying parallelism.
  - OpenMP. **http://www.openmp.org**

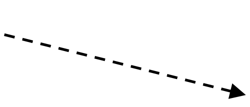
Multi-processes programming

- Operating systems often based upon notion of a process.
- Processor time shares between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.
- Offers opportunity to de-schedule processes blocked from proceeding for some reasons, e.g. waiting for an I/O operation to complete.
- Concept could be used for parallel programming. Not much used because of overhead but fork/join concepts used elsewhere.

FORK-JOIN construct

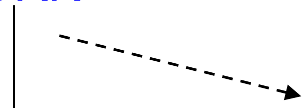
Main program

FORK

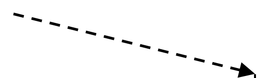


Spawned processes

FORK



FORK



JOIN

JOIN

JOIN

JOIN

UNIX System Calls

- No join routine - use exit() and wait()

- SPMD model

```
..
pid = fork(); /* fork */
Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0); /* join */
...
```

UNIX System Calls (2)

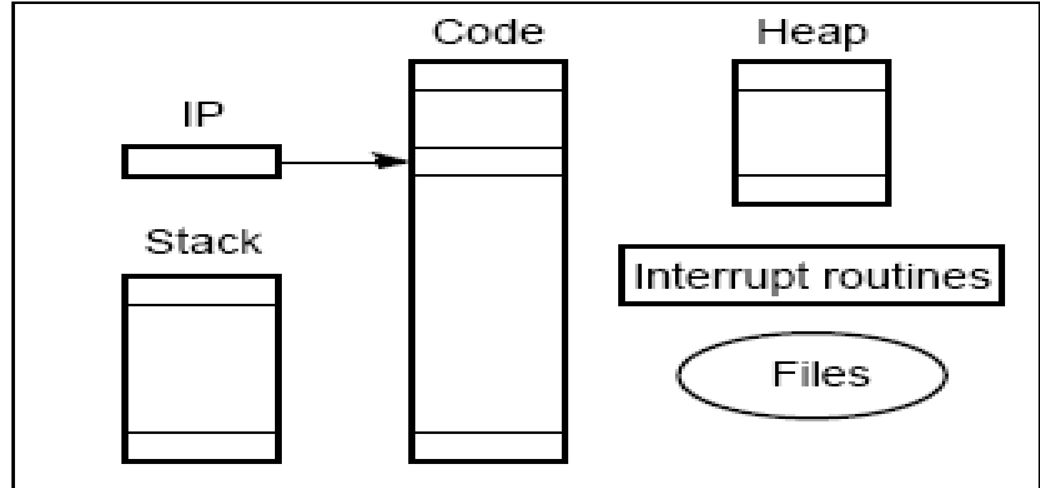
- SPMD model: master-workers model.

```
1. …
2. pid = fork();
3. if (pid == 0) {
4.       Code to be executed by slave process
5. } else {
6.       Code to be executed by master process
7. }
8. if (pid == 0) exit(0); else wait(0);
9. ...
```

Process vs thread

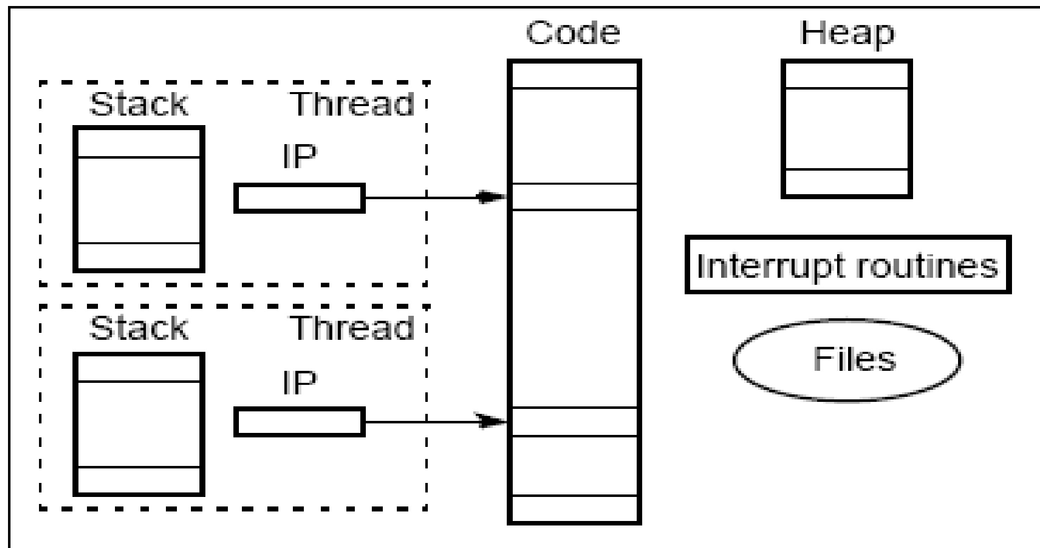
Process

- Completely separate program with its own variables, stack, and memory allocation.



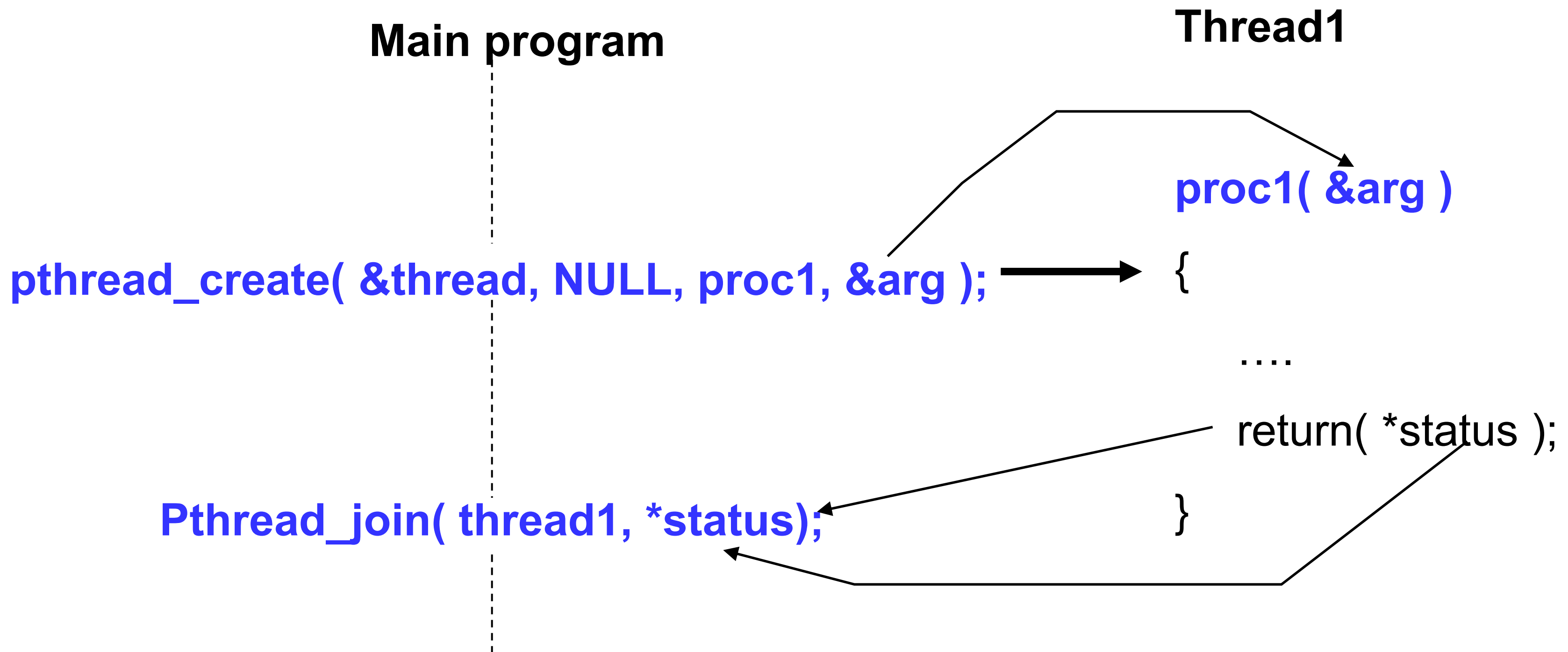
Threads

– Share the same memory space and global variables between routines



IEEE Pthreads (1)

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



The pthread_create() function

- #include <pthread.h>
- int pthread_create(

```
                pthread_t *threadid,
                pthread_attr_t * attr,
                void * (*start_routine)(void *),
                void * arg);
```

- The *pthread_create()* function creates a new thread storing an identifier to the new thread in the argument pointed to by *threadid*.

The pthread_join() function

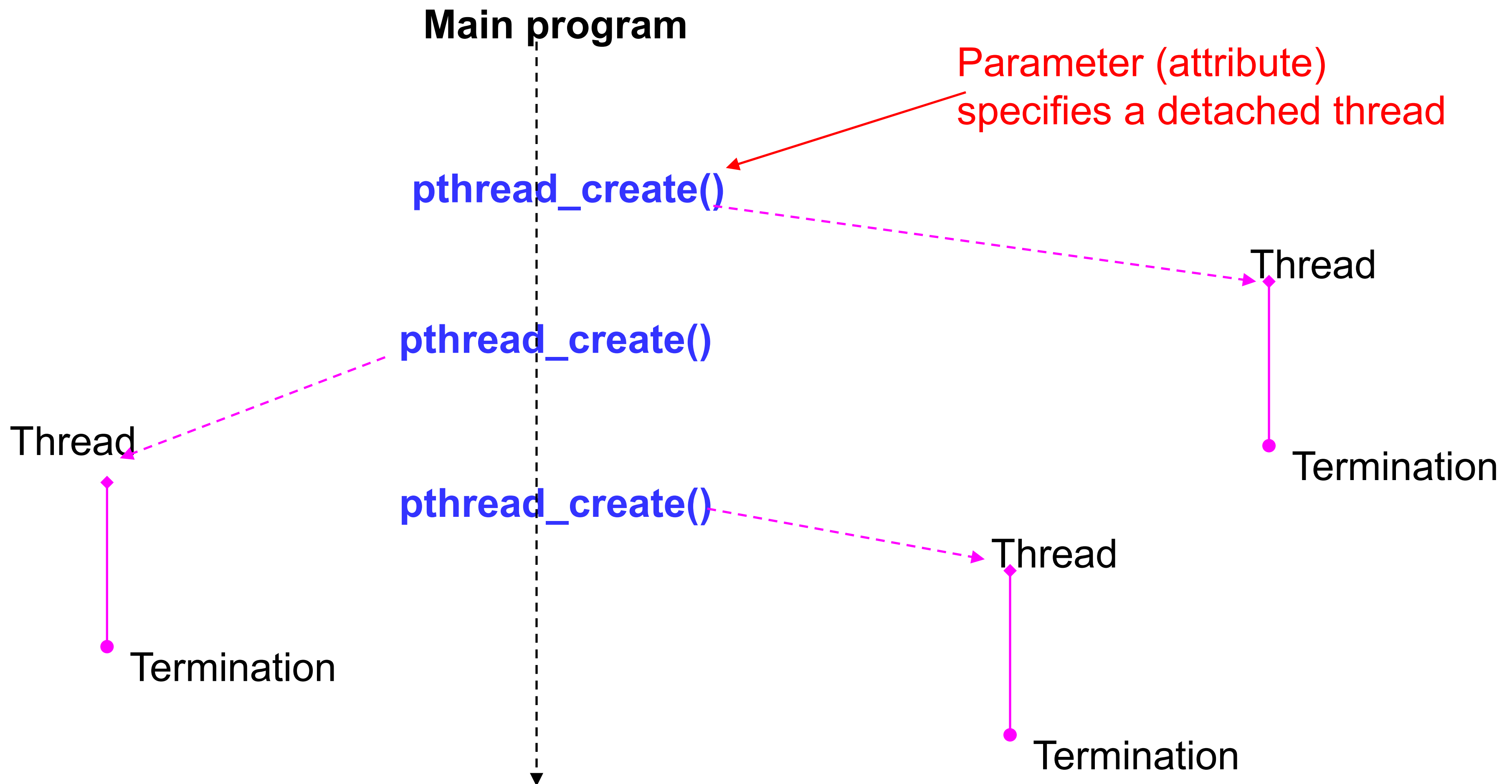
- **#include <pthread.h>**
- **void pthread_exit(void *retval);**
- **int pthread_join(pthread_t threadid,**
  void **retval);

- The function *pthread_join()* is used to suspend the current thread until the thread specified by *threadid* terminates. The other thread's return value will be stored into the address pointed to by ***retval*** if this value is not NULL.

Detached threads

- It may be that threads 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.

Pthread detached threads



The pthread_detach() function

- **#include <pthread.h>**
- **int pthread_detach(pthread_t threadid);**

• Put a running thread into detached state.

• Can't synchronize on termination of thread *threadid* using *pthread_join()*.

TABLE 11.2 THREAD ATTRIBUTES

Attribute	*Value*	*Meaning*
detachstate	PTHREAD_CREATE_JOINABLE*	Joinable state
	PTHREAD_CREATE_DETACHED	Detached state

Thread cancellation

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

- The *pthread_cancel* function allows the current thread to cancel another thread, identified by *thread*.
- Cancellation is the mechanism by which a thread can terminate the execution of another thread. More precisely, a thread can send a cancellation request to another thread. Depending on its settings, the target thread can then either ignore the request, honor it immediately, or defer it till it reaches a cancellation point.

Other Pthreads functions

- #include <pthread.h>
- int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void));

Thread pools

- Master-Workers Model:
  - A master thread controls a collection of worker thread.
  - Dynamic thread pools.
  - Static thread pools.
  - Threads can communicate through shared locations or signals.

Statement execution order

- Single processor: Processes/threads typically executed until blocked.
- Multiprocessor: Instructions of processes/threads interleaved in time.

Example

Process 1	**Process 2**
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

- Several possible orderings, including

  Instruction 1.1
  Instruction 1.2
  Instruction 2.1
  Instruction 1.3
  Instruction 2.2
  Instruction 2.3

assuming instructions cannot be divided into smaller interruptible steps.

Statement execution order (2)

- If two processes were to print messages, for example, the messages could appear in different orders depending upon the scheduling of processes calling the print routine.
- Worse, the individual characters of each message could be interleaved if the machine instructions of instances of the print routine could be interleaved.

Compiler/Processor optimization

- Compiler and processor reorder instructions for optimization.
- **Example:** the statements

  ```
  a = b + 5;
  x = y + 4;
  ```

  could be compiled to execute in reverse order:

  ```
  x = y + 4;
  a = b + 5;
  ```

  and still be logically correct.

  May be advantageous to delay statement a = b + 5 because a previous instruction currently being executed in processor needs more time to produce the value for b. Very common for processors to execute machines instructions out of order for increased speed .

Thread-safe routines

- *Thread safe* if they can be called from multiple threads simultaneously and always produce correct results.
- Standard I/O thread safe:
  - printf(): prints messages without interleaving the characters.
- NOT thread-safe functions:
  - System routines that return time may not be thread safe.
- Routines that access shared data may require special care to be made thread safe.

SHARING DATA

SHARING DATA

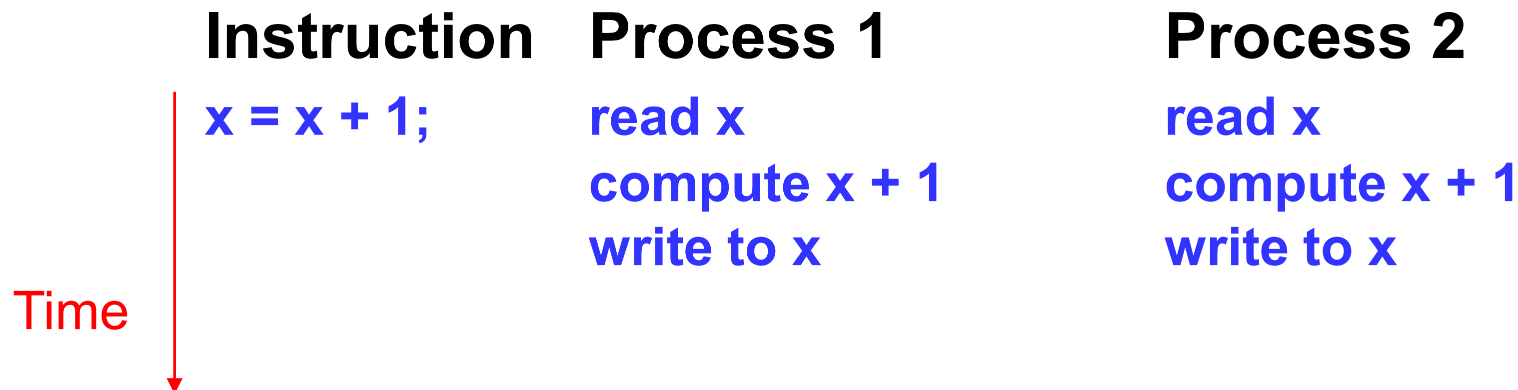
- Every processor/thread can directly access shared variables, data structures rather than having to the pass data in messages.
- Solution for critical sections:
  - Lock
  - Mutex
  - Semaphore
  - Conditional variables
  - Monitor

Creating shared data

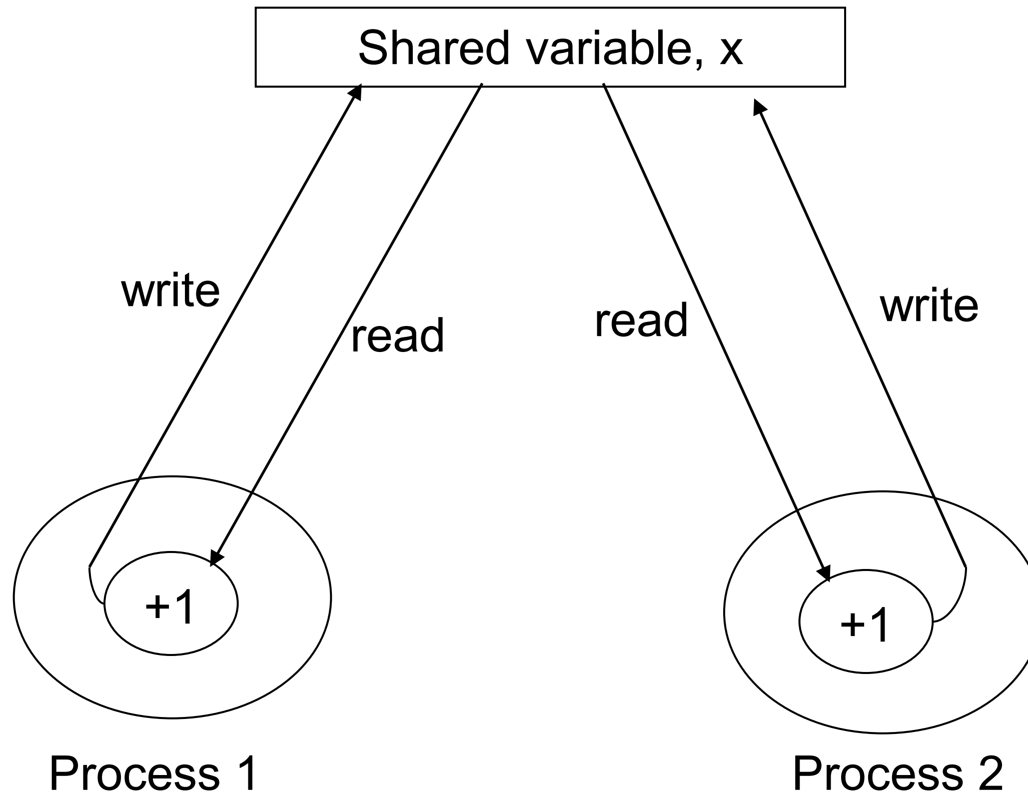
- UNIX processes: each process has its own virtual address space within the virtual memory management system.
  - Shared memory system calls allow processes to attach a segment of physical memory to their virtual memory space.
    - shmget() – creates, returns shared memory segment identifier.
    - shmat() – returns the starting address of data segment.
- It's NOT necessary to create shared data items explicity when using threads.
  - Global variables: available to all threads.

Acsessing shared data

- Accessing shared data needs careful control.
- Consider two processes each of which is to add one to a shared data item, x. Necessary for the contents of the location x to be read, x + 1 computed, and the result written back to the location:



Conflict in accessing shared variable



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*.

- This concept also appears in an operating systems.

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.

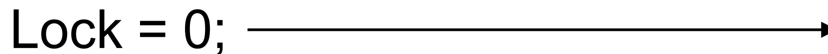
Control of critical sections through busy waiting

Processs 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;

Pthreads lock functions

- Pthreads implements lock by *mutally exclusive lock variables (mutex* variables*)*.

```
pthread_mutex_t mutex1;

pthread_mutex_init( &mutex1, NULL );

……..

pthread_mutex_lock ( &mutex1 );

        /// Critical section code here

pthread_mutex_unlock( &mutex1 );
```

Only 1 thread can enter the critical section code or wait

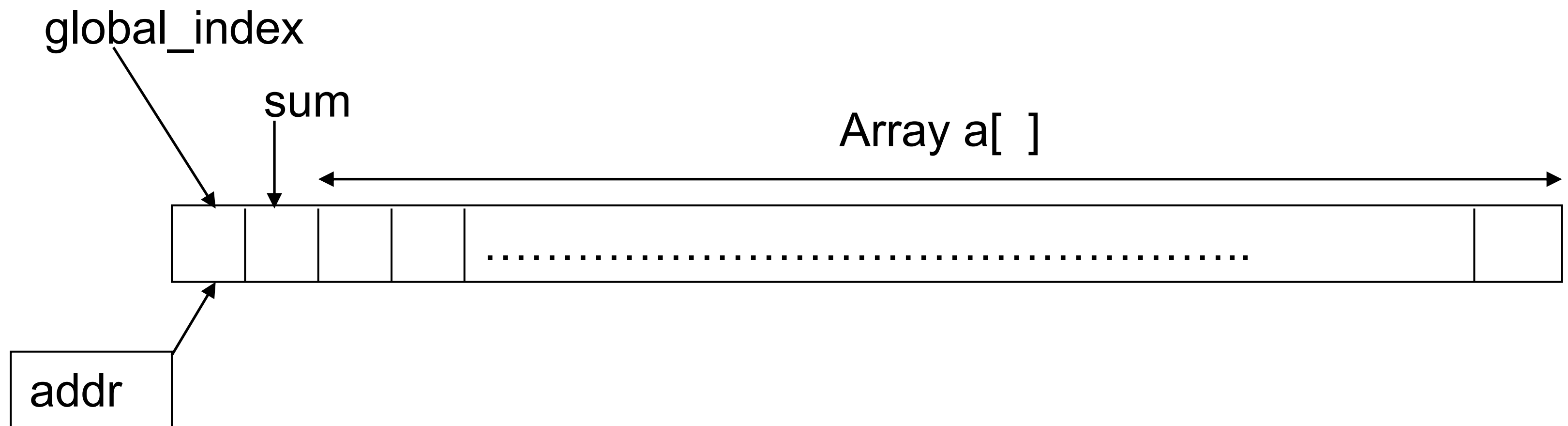
Only the thread that locks a mutex can unlock it. Otherwise, throws an error.

IEEE Pthreads example

- Calculating sum of an array a[ ].
- N threads created, each taking numbers from list to add to their sums. When all numbers taken, threads can add their partial results to a shared location sum.
- The shared location global_index is used by each thread to select the next element of a[].
- After index is read, it is incremented in preparation for the next element to be read. The result location is sum, as before, and will also need to be shared and access protected by a lock.

IEEE Pthreads example (2)

- Calculating sum of an array a[ ].



Code at page 254

IEEE Pthreads example (3)

```
1.  #include <stdio.h>
2.  #include <pthread.h>
3.  #define ARRAY_SIZE 1000
4.  #define NUM_THREADS 10

5.  // Global Variables, Shared data
6.  int a[ ARRAY_SIZE ];
7.  int global_index = 0;
8.  int sum = 0;

9.  pthread_mutex_t mutex1; // mutually exclusive lock variable
10. pthread_t worker_threads[ NUM_THREADS ];
```

IEEE Pthreads example (4)

```
1.  // Worker thread
2.  void *worker(void *ignored ) {
3.          int local_index, partial_sum = 0;
4.          do      {
5.                  pthread_mutex_lock ( &mutex1 );
6.                          local_index = global_index;         global_index++;
7.                  pthread_mutex_unlock( &mutex1 );
8.                  if (local_index < ARRAY_SIZE)   {
9.                          partial_sum += a [ local_index ];
10.                 }
11.         }
12.         while ( local_index < ARRAY_SIZE );
13.         pthread_mutex_lock( &mutex1 );
14.                 sum += partial_sum;
15.         pthread_mutex_unlock( &mutex1 );
16. }
```

IEEE Pthreads example (5)

```
1.  void master() {
2.          int i;
3.          // Initialize mutex
4.          pthread_mutex_init( &mutex1, NULL );
5.          init_data();
6.          create_workers( NUM_THREADS );
7.          // Join threads
8.          for (i = 0; i < NUM_THREADS ; i++ )     {
9.                  if ( pthread_join( worker_threads[i], NULL ) != 0 )      {
10.                         perror( "PThread join fails" );
11.                 }
12.         }
13.         printf("The sum of 1 to %i is %d \n" , ARRAY_SIZE, sum );
14. }
```

IEEE Pthreads example (6)

```
1.  void init_data() {
2.          int i;
3.          for (i = 0;  i < ARRAY_SIZE ;  i++ )    {   a[i] = i + 1;      }
4.  }

5.  // Create some worker threads
6.  void create_workers(int n){
7.          int i;
8.          for (i = 0; i < n ; i++ )    {
9.                  if (pthread_create(&worker_threads[i], NULL,
                                worker, NULL ) != 0 )       {
10.                         perror( "Pthreads create fails" );    }
11.         }
12. }
```

Java multithread programming

- A class extends from java.lang.Thread class.
- A class implements java.lang.Runnable interface.

```
// A sample Runner class
public class Runner extends Thread
{
String name;
public Runner(String name) {
    this.name = name;
}
public void run() {
    int N = 10;
    for (int i = 0; i < N ; i++)      {
    System.out.println("I am "+
    this.name + "runner at " + I + " km.");
    thread.delay(100);
    }
}
```

```
public static void main(String[] args)
{
    Runner hung = new Runner("Hung");
    Runner minh = new Runner("Minh");
    Runner ken = new Runner("Ken");
    hung.start();
    minh.start();
    ken.start();
    System.out.println("Hello World!");
}
} // End main
```

Language Constructs for Parallelism

Language Constructs for Parallelism - Shared Data

- Shared Data:
  - shared memory variables might be declared as shared with, say,

```
shared int x;
```

- Par Construct

```
par {
        S1;
        S2;
        .
        .
        Sn;
}

par {
         proc1();
         proc2();
         ...
}
```

Forall Construct

- Keywords: forall or parfor
- To start multiple similar processes together: which generates *n* processes each consisting of the statements forming the body of the for loop, S1, S2, …, Sm. Each process uses a different value of *i*.

```
forall (i = 0 ; i < N; i++ ) {
    S1;
    S2;
    …..
    Sm;
}
```

- Example:

```
forall (i = 0; i < 5; i++)
        a[i] = 0;
```

clears **a[0]**, **a[1]**, **a[2]**, **a[3]**, and **a[4]** to zero concurrently.

Dependency analysis

- To identify which processes could be executed together.
- **Example: c**an see immediately in the code

```
forall (i = 0; i < 5; i++)
a[i] = 0;
```

- that every instance of the body is independent of other instances and all instances can be executed simultaneously.
- However, it may not be that obvious. Need algorithmic way of recognizing dependencies, for a *parallelizing compiler*.

Bernstein's Conditions

- Set of conditions sufficient to determine whether two processes can be executed simultaneously. Given:
  - I_i is the set of memory locations read (input) by process P_i.
  - O_j is the set of memory locations written (output) by process P_j.
- For two processes P1 and P2 to be executed simultaneously, inputs to process P1 must not be part of outputs of P2, and inputs of P2 must not be part of outputs of P1; i.e.,
  - $I_1 \cap O_2 = \phi$
  - $I_2 \cap O_1 = \phi$
- where f is an empty set. Set of outputs of each process must also be different; i.e.,
  - $O_1 \cap O_2 = \phi$
- If the three conditions are all satisfied, the two processes can be executed concurrently.

Example

- **Example:** suppose the two statements are (in C)
  - **a = x + y;**
  - **b = x + z;**
- We have
  - $\boldsymbol{I1 = (x, y)}$ $\boldsymbol{O1 = (a)}$
  - $\boldsymbol{I2 = (x, z)}$ $\boldsymbol{O2 = (b)}$
- and the conditions
  - $I1 \cap O2 = \phi$
  - $I2 \cap O1 = \phi$
  - $O1 \cap O2 = \phi$
- are satisfied. Hence, the statements **a = x + y** and **b = x + z** can be executed simultaneously.

OpenMP

- An accepted standard developed in the late 1990s by a group of industry specialists.
- Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base language Fortran and C/C++.
- The compiler directives can specify such things as the par and forall operations described previously.
- Several OpenMP compilers available.
- Exercise: read more & report:
  - http://www.openmp.org

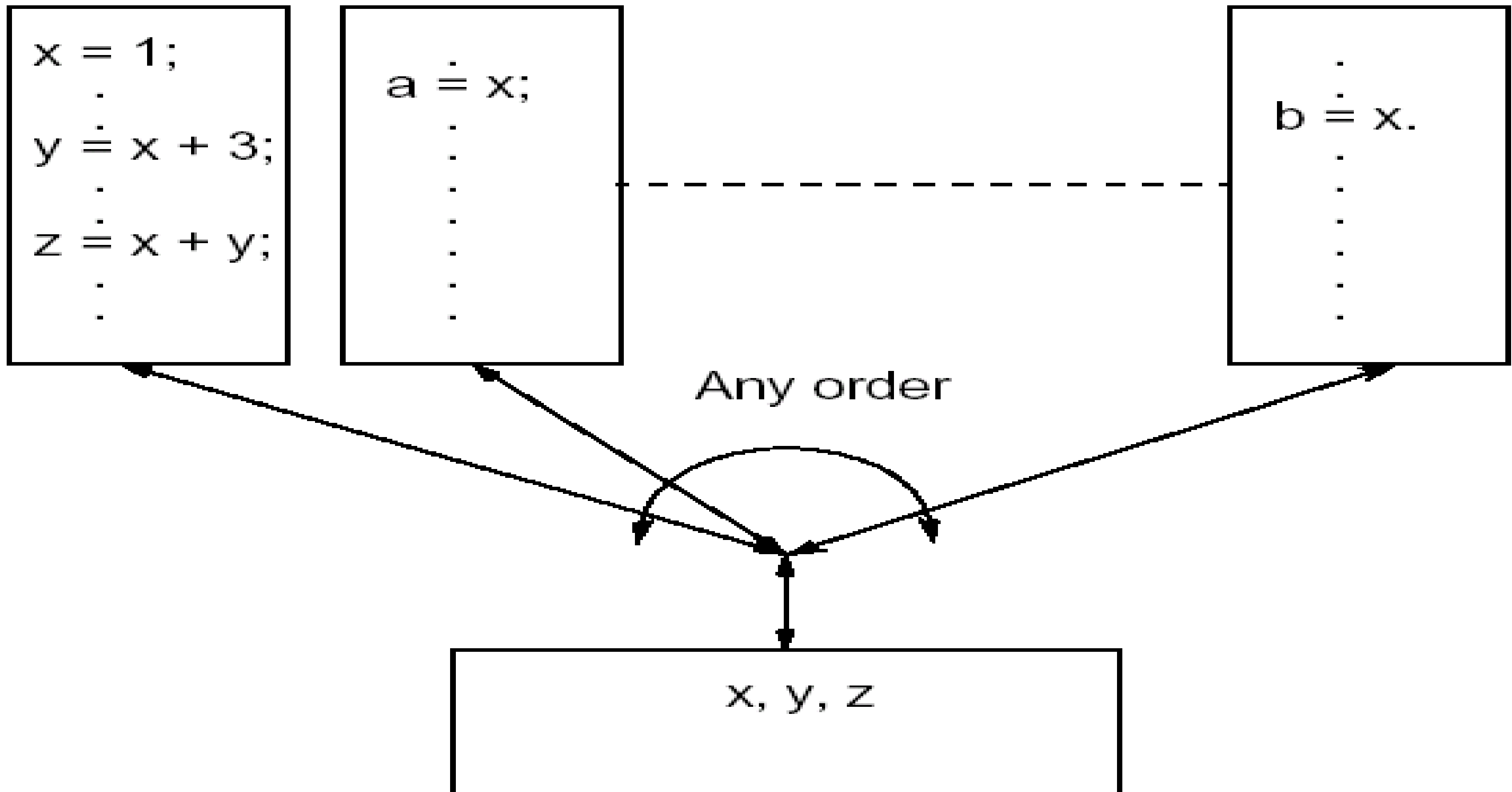
Shared Memory Programming Performance Issues

- Shared data in systems with caches
  - Cache coherence protocols
  - False Sharing:
    - Solution: compiler to alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.
- High performance programs should have as few as possible critical sections as their use can serialize the code.

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 (2)



Sequential consistency (2)

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