

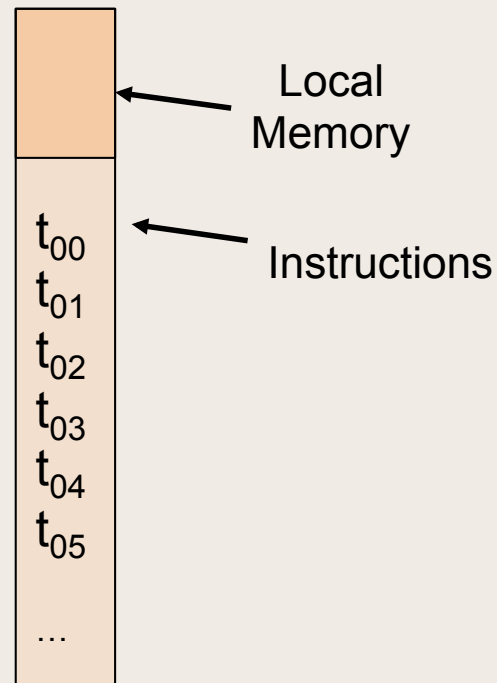
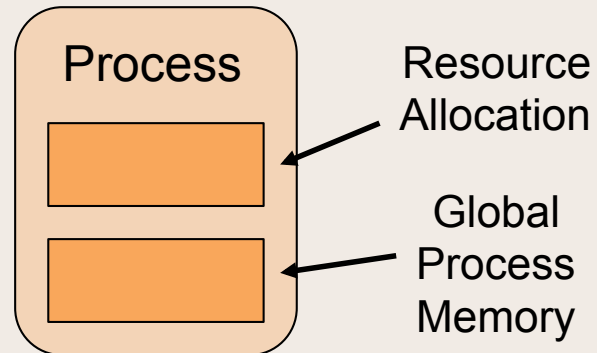
CS 6230: High-Performance Computing and Parallelization – Introduction to OpenMP

Dr. Mike Kirby

School of Computing and
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT, USA

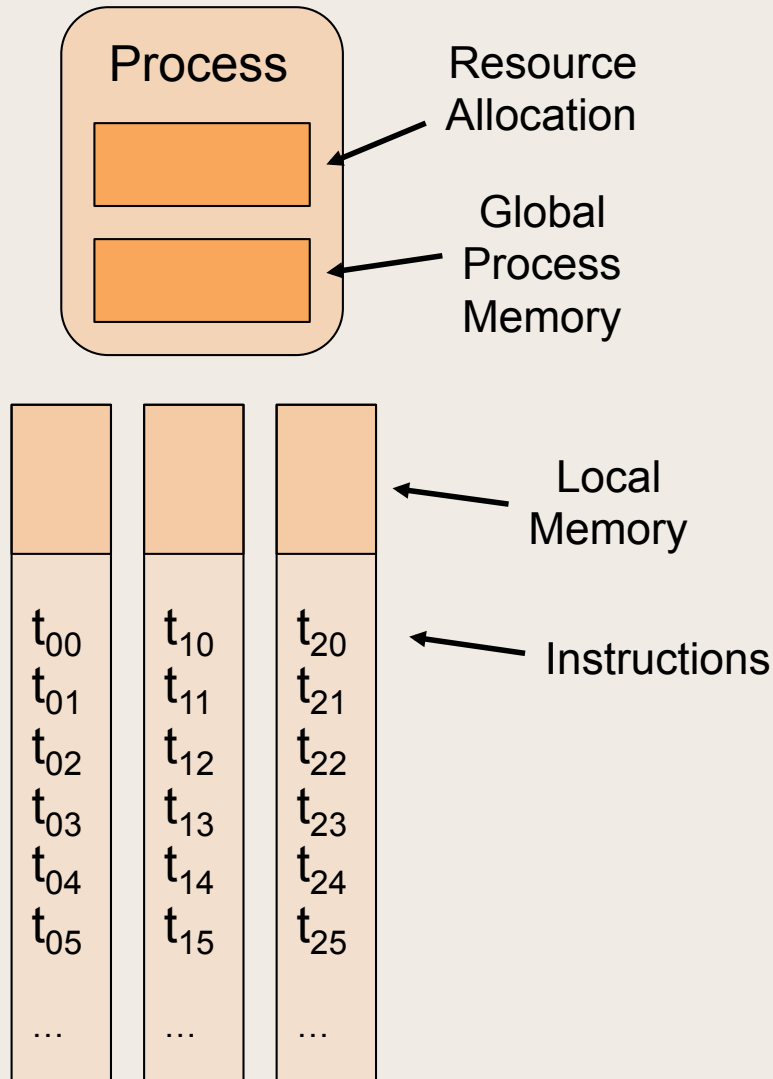


What is a Process?



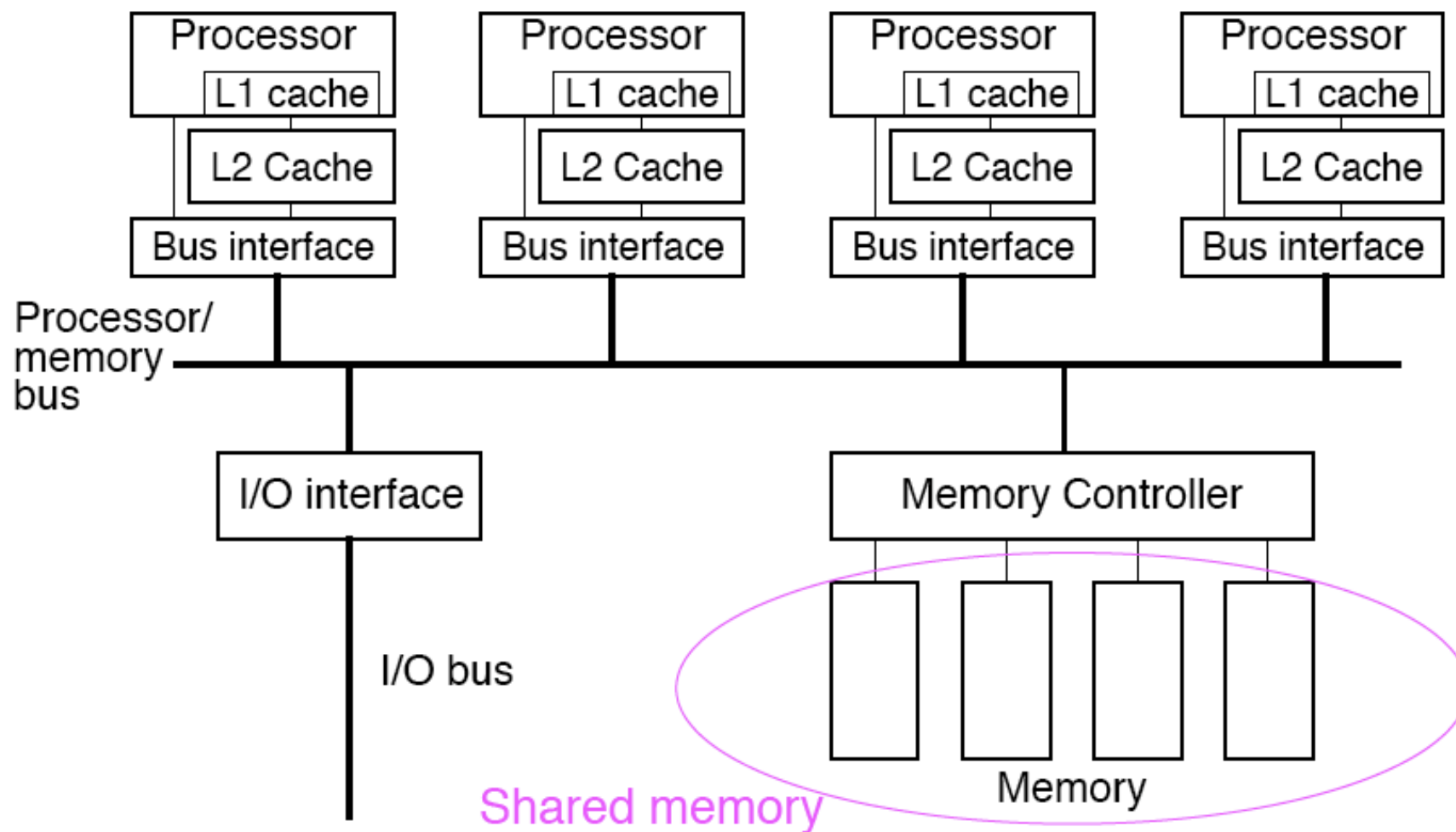
- A process is usually defined as an instance of a program that is being executed, including all variables and other information describing the program's state.
- Each process is an independent entity to which system resources (CPU time, memory, etc.) are allocated.
- Each processes is executed in a separate address space. Thus, one process cannot access variables or other data structures that are defined in another process.
- If two processes want to communicate, they have to use inter-process communication mechanisms like files, pipes, or sockets.

What then is a “Thread”?



- A “Thread” is a “Thread of Execution”. Each process upon its creation has at least one thread of execution.
- A multi-threaded program is one in which a single process has multiple threads of execution. All threads have access to the global process memory and process resources. Each thread maintains its own local memory and list of instructions.
- Threads can communicate via the “shared” global memory.

Shared-Memory Computing



OpenMP

- OpenMP is based on the fork/join programming model.
- OpenMP was designed around two key concepts:
 - sequential equivalence: a program yields the same results whether it executes using one thread or many threads.
 - incremental parallelism: A style of programming in which a program evolves from a sequential program into a parallel program.

Example 1

Fortran Version

```
program simple  
print *, "E pur si muove"  
stop end
```

C Version

```
#include <stdio.h>  
  
int main()  
{  
    printf("E pur si muove \n");  
}
```

OpenMP Pragmas

```
C$OMP PARALLEL  
C$OMP END PARALLEL
```

```
#pragma omp parallel
```

Example 1 (Continued)

Fortran Version

```
program simple
C$OMP PARALLEL
print *, "E pur si muove"
C$OMP END PARALLEL
stop end
```

Output (if OMP_NUM_THREADS=3):

```
E pur si muove
E pur si muove
E pur si muove
```

C Version

```
#include <stdio.h>
#include "omp.h"

int main()
{
    #pragma omp parallel
    {
        printf("E pur si muove \n");
    }
}
```

Example 2

Fortran Version

```
program simple
integer i
i = 5
C$OMP PARALLEL
print *, "E pur si muove", i
C$OMP END PARALLEL
stop end
```

Output (if OMP_NUM_THREADS=3):

```
E pur si muove 5
E pur si muove 5
E pur si muove 5
```

C Version

```
#include <stdio.h>
#include "omp.h"

int main()
{
    int i=5;
    #pragma omp parallel
    {
        printf("E pur si muove %d \n", i);
    }
}
```

Considered shared (visible by all threads)

Example 3

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int i=5; // a shared variable

    #pragma omp parallel
    {
        int c; // a variable local or private to each thread
        c = omp_get_thread_num();
        printf("c = %d, i = %d\n",c,i);
    }
}
```

Considered shared (visible by all threads)

Considered local or private to thread

Output (if OMP_NUM_THREADS=3):

c = 0, i = 5

c = 2, i = 5

c = 1, i = 5

Setting the number of Threads

Environment Variable:

```
setenv OMP_NUM_THREADS 3
```

At Runtime:

```
#pragma omp parallel num_threads(3)
```

OpenMP: Structured Blocks and Directive Formats

An *OpenMP construct* is defined to be a directive (or pragma) plus a block of code. Not just any block of code will do. It must be a structured block:

- Block with one point of entry at the top
- Block with one point of exit at the bottom

OpenMP – Pragma-based Library

In general, the form is:

```
#pragma omp directive-name [clause[ clause] ...]
```

Specific examples:

```
#pragma omp parallel private(ii, jj, kk)  
#pragma omp barrier  
#pragma omp for reduction(+:result)
```

Worksharing

- When using a parallel construct alone, every thread executes the same block of statements.
- Times exist in which we want different code to map onto different threads. The most common example *is loop splitting*.

Worksharing

The OpenMP parallel construct is used to create a team of threads. This can then be followed by a worksharing construct. Often the pragma is shortened.

```
#pragma omp parallel  
{  
  #pragma omp for  
    ...  
}
```

```
#pragma omp parallel for  
{  
    ...  
}
```

Typical Loop-Oriented Program

```
#include <stdio.h>
#define N 1000
extern void combine(double, double);
extern double big_comp(int);

int main()
{
    int i;
    double answer, res;
    answer = 0.0;
    for (i=0;i<N;i++)
    {
        res = big_comp(i);
        combine(answer,res);
    }
    printf("%f\n", answer);
}
```

Typical Loop-Oriented Program (Continued)

```
#include <stdio.h>
#define N 1000
extern void combine(double, double);
extern double big_comp(int);
```

```
int main()
{
```

```
    int i;
    double answer, res[N];
    answer = 0.0;
```

Extra Memory




```
    for (i=0; i<N; i++)
        res[i] = big_comp(i);
```

Decoupled Computation



```
    for(i=0; i<N; i++)
        combine(answer, res[i]);
```

Serial Computation



```
    printf("%f\n", answer);
}
```


Typical Loop-Oriented Program (Continued)

```
#include <stdio.h>
#include <omp.h>
#define N 1000
extern void combine(double, double);
extern double big_comp(int);
```

```
int main()
{
    int i;
    double answer, res[N];
    answer = 0.0;
```

```
#pragma omp parallel
{
    #pragma omp for
        for (i=0;i<N;i++)
            res[i] = big_comp(i);
}
```

← Decoupled Computation

```
    for(i=0;i<N;i++)
        combine(answer,res[i]);

    printf("%f\n", answer);
}
```

Wait/Nowait

By default there is an implicit barrier at the end of any OpenMP workshare construct; that is, all the threads wait at the end of the construct and only proceed after all the threads have arrived.

This barrier can be removed by adding a nowait clause to the worksharing construct:

```
#pragma omp for nowait
```

Commonly-Used Types of Worksharing Constructs

The **single** construct defines a block of code that will be executed by the first thread that encounters the construct. Other threads arriving later skip the construct and wait (implicitly) at the barrier at the end of the construct.

```
#pragma omp single  
    num_threads = omp_get_num_threads();
```

The **sections** construct is used to set up a region of the program where distinct blocks of code are to be assigned to different. Each block is defined with a section construct.

Data Environment Clauses: Example 1

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
```

```
int main()
{
```

```
    int i;
    int num_steps = 1000000;
    double x, pi, step, sum = 0.0;
```

```
    step = 1.0/(double) num_steps;
```

```
#pragma omp parallel for private(x) reduction(+:sum)
```

```
    for (i=0;i<num_steps;i++)
```

```
    {
```

```
        x = (i+0.5)*step;
```

```
        sum += 4.0/(1.0+x*x);
```

```
    }
```

```
    pi = step * sum;
```

```
    printf("pi %lf\n",pi)
```

```
    return 0;
```

```
}
```

Target:
$$\int_0^1 \frac{4}{1+x^2} dx$$

Private variable with for

Reduction on sum variable

Data Environment Clauses

- In OpenMP, the variable that is bound to a given name depends on whether the name appears prior to a parallel region, inside a parallel region, or following a parallel region.
- When the variable is declared prior to a parallel region, it is by default shared and the name is always bound to the same variable.
- A `private(list)` clause directs the compiler to create, for each thread, a private (or local) variable for each name included in the list.

Data Environment Clauses

- **firstprivate(list)**. Just as with private, for each name appearing in the list, a new private variable for that name is created for each thread. Unlike private, however, the newly created variables are initialized with the value of the variable that was bound to the name in the region of the code preceding the construct containing the firstprivate clause. This clause can be used on both parallel and the worksharing constructs.
- **lastprivate(list)**. Once again, private variables for each thread are created for each name in the list. In this case, however, the value of the private variable from the sequentially last loop iteration is copied out into the variable bound to the name in the region following the OpenMP construct containing the lastprivate clause. This clause can only be used with the loop-oriented workshare constructs.

Data Environment Clauses: Example 2

```
#include <stdio.h>
#include <omp.h>
```

```
int main()
{
    int h, i, j, k;
    h = 1; j = 2; k = 0;
```

Another variable called h is allocated to each thread. It is uninitialized from the shared j and k.

Private variables called j and k are allocated to each thread. They are uninitialized from the shared j and k.

```
#pragma omp parallel for private(h) firstprivate(j,k) \
                           lastprivate(j,k)

    for (i=0;i<N;i++)
    {
        k++;
        j = h + i; // ERROR: h, and therefore j, is undefined
    }

    printf("h = %d, j = %d, k = %d\n",h, j, k); // ERROR j and h are undefined

    return 0;
}
```

Data Environment Clauses

- A variable can be defined in both the firstprivate and lastprivate clause. It is entirely possible that a private variable will need both a well-defined initial value and a value exported to the region following the OpenMP construct in question.
- In lastprivate used with the for construct, the value of the variables is the value of the last iteration of the loop (from whichever thread was responsible for that iteration).
- OpenMP stipulates that after a name appears in any of the private clauses, the variable associated with that name in the region of code following the OpenMP construct is undefined.

OpenMP Runtime Library

- `void omp_set_num_threads(int)` – takes an integer argument and requests that the operating system provide that number of threads in subsequent parallel regions.
- `int omp_get_num_threads(void)` – return the actual number of threads in the current team of threads.
- `int omp_get_thread_num(void)` – return the ID of a thread where the ids range from 0 to N-1. Thread 0 is the master thread.

OpenMP Runtime Library

```
#include <stdio.h>
#include <omp.h>
```

```
int main()
{
```

```
    int id, numb;
```

```
    omp_set_num_threads(3);
```

```
#pragma omp parallel private (id, numb)
{
```

```
    id = omp_get_thread_num();
```

```
    numb = omp_get_num_threads();
```

```
    printf("I am thread %d out of %d \n",id,numb);
```

```
}
```

```
    return 0;
```

```
}
```

I am thread 2 out of 3

I am thread 0 out of 3

I am thread 1 out of 3

Synchronization

- **flush** – defines a synchronization point at which memory consistency is enforced.

```
#pragma omp flush [(list)]
```

- **critical** – implements a critical section for mutual exclusion.

```
#pragma omp critical [(name)]  
{ a structured block }
```

- **barrier** – provides a synchronization point at which the threads wait until every member of the team has arrived before any threads continue.

```
#pragma omp barrier
```

Synchronization

```
#include <stdio.h>
#include <omp.h>
#define N 1000
extern void combine(double,double);
extern double big_comp(int);

int main()
{
    int i;
    double answer, rea;
    answer = 0.0;
    #pragma omp parallel for private (res)
    for (i=0;i<N;i++)
    {
        res = big_comp(i);
        #pragma omp critical
        combine(answer,res);
    }
    printf("%f\n", answer);

    return 0;
}
```

Each thread serializes on this statement

Synchronization

- `void omp_init_lock(omp_lock_t *lock)` – initialize lock.
- `void omp_destroy_lock(omp_lock_t *lock)` – destroy the lock, thereby freeing any memory associated with the lock.
- `void omp_set_lock(omp_lock_t *lock)` – set or acquire the lock. If the lock is free, then the thread calling `omp_set_lock()` will acquire the lock and continue. If the lock is held by another thread, the thread is calling `omp_set_lock()` will wait on the call until the lock is available.

Synchronization

- `void omp_unset_lock(omp_lock_t *lock)` – unset or release the lock so some other thread can acquire it.
- `int omp_test_lock(omp_lock_t *lock)` – test or inquire if the lock is unavailable. If it is, then the thread calling this function will acquire the lock and continue. The test and acquisition of the lock is done atomically. If it is not, the function returns false (nonzero) and the calling thread continue. This function is used so a thread can do useful work while waiting for a lock.

Synchronization

```
#include <stdio.h>
#include <omp.h>
```

```
extern void do_something_else(int); extern void go_for_it(int);
```

```
int main(){
```

```
    omp_lock_t lck1, lck2;
    int id;
```

Declare locks

```
    omp_init_lock(&lck1);
    omp_init_lock(&lck2);
```

Initialize locks

```
#pragma omp parallel shared(lck1, lck2) private(id)
{
```

```
    id = omp_get_thread_num();
```

```
    omp_set_lock(&lck1);
```

Set lock

```
    printf("thread %d has the lock \n", id);
```

```
    printf("thread %d ready to release the lock \n", id);
```

```
    omp_unset_lock(&lck1);
```

Release lock

```
    while(!omp_test_lock(&lck2)){
```

```
        do_something_else(id); // do something useful while waiting
                                // for the lock
```

```
    }
```

```
    go_for_it(id); //Thread has the lock
```

```
    omp_unset_lock(&lck2);
```

```
}
```

```
    omp_destroy_lock(&lck1);
```

```
    omp_destroy_lock(&lck2);
```

```
}
```

Scheduling Clause

```
#include<stdio.h>
#include<stdlib.h>
#define NX 100
#define LEFTVAL 1.0
#define RIGHTVAL 10.0
#define NSTEPS 10000

void initialize(double uk[], double ukp1[]) { /* NOT SHOWN */}
void printValues(double uk[], int step) { /* NOT SHOWN */}

int main(void)
{
    /*pointers to arrays for two iterations of the algorithm */
    double * uk = new double[NX];
    double * ukp1 = new double[NX];
    double *temp;
    int i,k;

    double dx = 1.0/NX;
    double dt = 0.5*dx*dx;

#pragma omp parallel private (k, i)
    {
        initialize(uk, ukp1);

        for (int k=0; k<NSTEPS; ++k)
        {
            for (int i=1; i<NX-1; ++i)
            {
                ukp1[i] = uk[i] + (dt/(dx*dx)) * (uk[i+1] - 2.0*uk[i] + uk[i-1]);
            }
        }
        #pragma omp single
        {
            temp = ukp1; ukp1 = uk; uk = temp;
        }
        delete[] uk;
        delete[] ukp1;
    }
    return 0;
}
```

`schedule(static [,chunk])` – the iteration space is divided into blocks of size chunk. If chunk is omitted, then the block size is selected to provide one approximately equal size block per thread.

`schedule(dynamic, [,chunk])` – The iteration space is divided into blocks of size chunk. If chunk is omitted, the block size is set to 1. The remaining blocks are placed in a queue. When a thread finishes with its current block, it pulls the next block of iterations that need to be computed off the queue.

Glossary

Atomic: An atomic operation at the hardware level is uninterruptible, for example load and store, or atomic test-and-set introduction.

ccNUMA: Cache-coherent NUMA. A NUMA model where data is coherent at the level of the cache.

Condition variable: Condition variables are part of the monitor synchronization mechanism. A condition variable is used by a process or thread to delay until the monitor's state satisfies some condition; it is also used to awaken a delayed process when the condition becomes true.

Glossary

Fork/join: A programming model used in multithreaded APIs Such as OpenMP. A thread executes a fork and creates additional threads. The threads (called a team in OpenMP) execute concurrently. When the members of the team complete their concurrent tasks, they execute joins and suspend until every member of the team has arrived at the join. At that point, the members of the team are destroyed and the original thread continues.

Mutex: A mutual exclusion lock. A mutex serializes the execution of multiple threads.

Glossary

NUMA: This term is used to describe a shared-memory computer system where not all memory is equidistant from all processors. Thus, the time required to access memory locations is not uniform, and for good performance the programmer usually needs to be concerned with the placement of data in memory.

POSIX: The Portable Operating System Interface as defined by the Portable Applications Standards Committee (PASC) of the IEEE Computer Society. Whereas other operating systems follow some of the POSIX standards, the primary use of this term refers to the family of standards that define the interfaces in UNIX and UNIX-like (Linux) operating systems.

Glossary

Pthreads: Another name for POSIX threads, that is, the definition of threads in the various POSIX standards.

Semaphore: An ADT used to implement certain kinds of synchronization. A semaphore has a value that is constrained to be a nonnegative integer and two atomic operations. The allowable operations are V (sometimes called up) and P (sometimes called down). A V operation increases the value of the semaphore by one. A P operation decreases the value of the semaphore by one, provided that can be done without violating the constraint that the value be nonnegative. A P operation that is initiated when the value of the semaphore is 0 suspends.