

Ans: Pthreads API can be informally grouped into three major classes:

1. Thread management
2. Mutexes
3. Condition variables: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

In a critical section (i.e. where a mutex has been used), a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.

– It will suspend by waiting on a condition variable.

– It will, however, release the critical section lock (mutex).

– When that condition variable is signaled, it will become ready again; it will attempt to re-acquire

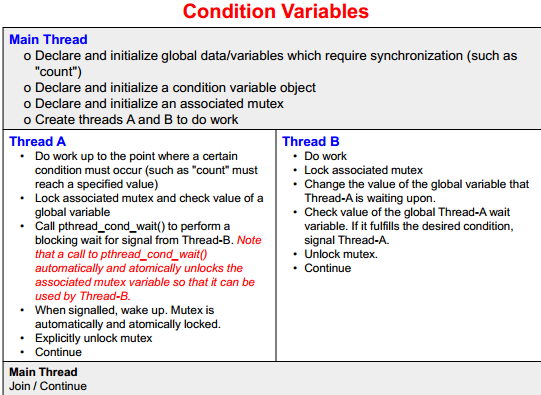
that critical section lock and only then will be able proceed.

With Posix threads, a condition variable can be associated with only one mutex variable!

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 variable!



Associated with a specific mutex. Given declarations:

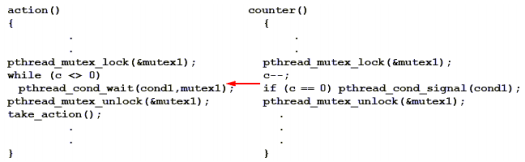
pthread\_cond\_t cond1;

pthread\_mutex\_t mutex1;

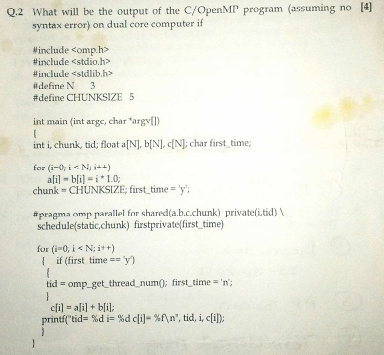
pthread\_cond\_init(&cond1, NULL);

pthread\_mutex\_init(&mutex1, NULL);

the Pthreads arrangement for signal and wait is as follows:



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



#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 50

#define CHUNKSIZE 5

int main (int argc, char \*argv[])

{

int i, chunk, tid;

float a[N], b[N], c[N];

char first\_time;

/\* Some initializations \*/

for (i=0; i < N; i++)

a[i] = b[i] = i \* 1.0;

chunk = CHUNKSIZE;

first\_time = 'y';

#pragma omp parallel for \

shared(a,b,c,chunk) \

private(i,tid) \

schedule(static,chunk) \

firstprivate(first\_time)

for (i=0; i < N; i++)

{

if (first\_time == 'y')

{

tid = omp\_get\_thread\_num();

first\_time = 'n';

}

c[i] = a[i] + b[i];

printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);

}

}

The output is as expected:

Hide   Copy Code

tid= 0 i= 0 c[i]= 0.000000

tid= 1 i= 5 c[i]= 10.000000

tid= 0 i= 1 c[i]= 2.000000

tid= 1 i= 6 c[i]= 12.000000

tid= 0 i= 2 c[i]= 4.000000

tid= 1 i= 7 c[i]= 14.000000

tid= 0 i= 3 c[i]= 6.000000

tid= 1 i= 8 c[i]= 16.000000

tid= 0 i= 4 c[i]= 8.000000

tid= 1 i= 9 c[i]= 18.000000

. . . . . and so on

tid= 1 i= 47 c[i]= 94.000000

tid= 0 i= 43 c[i]= 86.000000

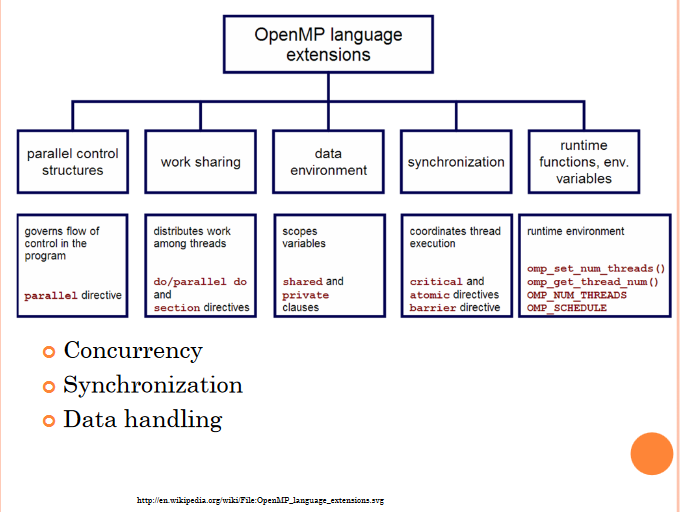
tid= 1 i= 48 c[i]= 96.000000

tid= 0 i= 44 c[i]= 88.000000

tid= 1 i= 49 c[i]= 98.000000

Source: https://www.codeproject.com/Articles/60176/WebControls/



Ans 3) 

Main categories of OpenMP’s constructs:

* Directives
* Parallel Regions
* Work-sharing
* Data Environment
* Synchronization

OpenMP: Work-sharing Construct

It distributes the execution of the associated statement among the members of the team that encounter it

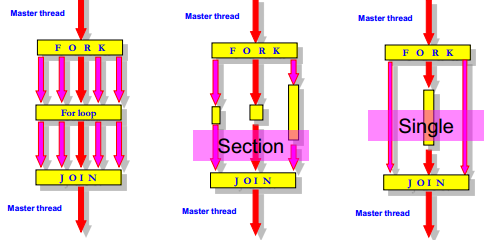
* Work sharing construct do not launch new threads
* There is no barrier upon entry to work-sharing construct.
* There is an implied barrier at the end of a work sharing construct

Restrictions

* Must be enclosed in the parallel region for parallel execution.
* Must be encountered by all the members of the team or none of them.

OpenMP defines the following work-sharing constructs.

* for directive
* sections directive
* single directive



**For directive (Represents a type of “data parallelism”)**

For directive identifies the iterative work-sharing construct.

#pragma omp for [clause[[,]clause]…] new-line

for-loop Clause is one of the following:

scheduled (type [,chunk])

private(variable list)

firstprivate (variable list)

lastprivate (variable list)

reduction (variable list)

ordered, nowait

**For directive**

The “for” Work-Sharing construct splits up loop iterations among the threads in a team

#pragma omp parallel

#pragma omp for

for (I=0; I<N; I++) {

NEAT\_STUFF(I);

}

Note: By default, there is a barrier at the end of the “omp for”.

***Important:*** The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

#include <omp.h>

#define CHUNKSIZE 100

#define N 1000

main()

{

int i, chunk; float a[N], b[N], c[N];

/\* Some initializations \*/

for (i=0; i < N; i++)

a[i] = b[i] = i \* 1.0;

chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i) {

#pragma omp for schedule(dynamic,chunk) nowait

for (i=0; i < N; i++)

c[i] = a[i] + b[i]; }/\* end of || section \*/

}

sections directive

Can be used to implement a type of "functional parallelism".

sections directive gives different structured blocks to each

thread.

#pragma omp parallel

#pragma omp sections {

#pragma omp section

x\_calculation();

#pragma omp section

y\_calculation();

……

}

* Independent SECTION directives are nested within a SECTIONS

directive.

* Each SECTION is executed once by a thread in the team. Different

sections will be executed by different threads.

Simple vector-add program

* The first n/2 iterations of the for loop will be distributed to the first thread, and the rest will be distributed to the second thread.
* When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)

#include <omp.h>

#define N 1000

main()

{

int i;

float a[N], b[N], c[N];

for (i=0; i < N; i++)

a[i] = b[i] = i \* 1.0;

#pragma omp parallel shared(a,b,c) private(i)

{

#pragma omp sections nowait

{

#pragma omp section

for (i=0; i < N/2; i++)

c[i] = a[i] + b[i];

#pragma omp section

for (i=N/2; i < N; i++)

c[i] = a[i] + b[i];

}

}

}

single directive

This identifies that the associated structured block is to be executed by only one thread in the team (It can be any thread including

master thread).

#pragma omp single [clause[[,] clause] ...]

new-line

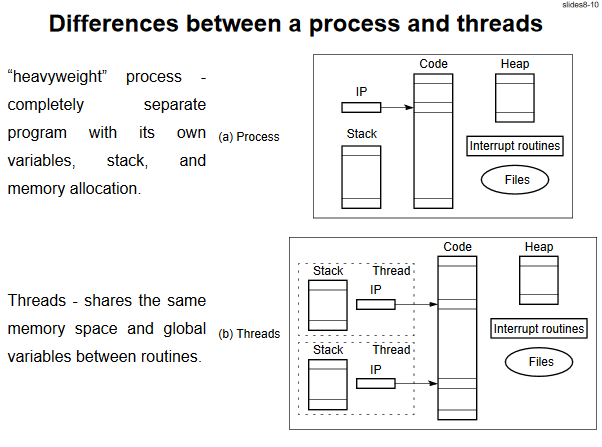
structured-block

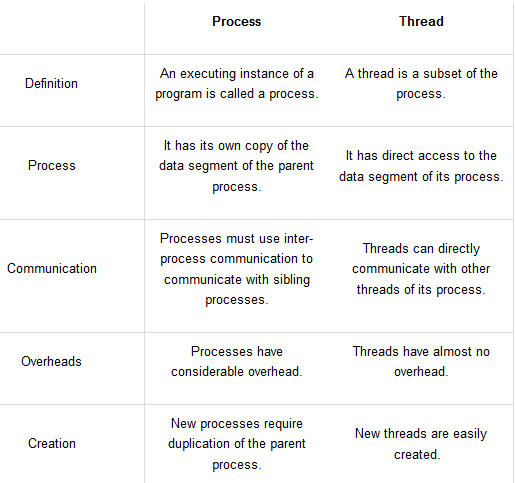
Restrictions

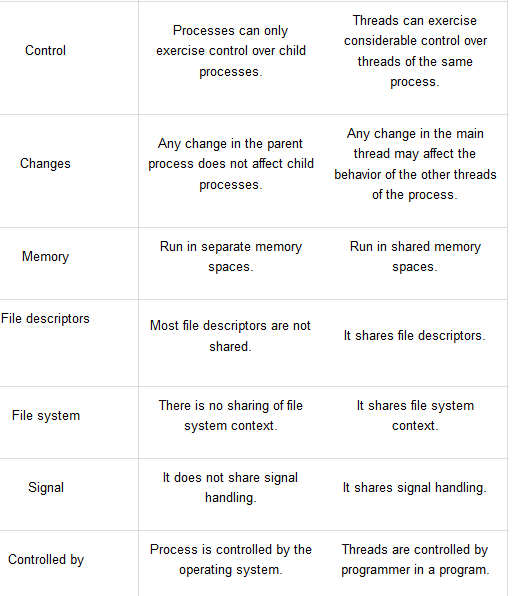
* A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
* Work-sharing constructs must be encountered by all members of a team or none at all.
* Successive work-sharing constructs must be encountered in the same order by all members of a team.



Ans:







Fork – Join Model

* OpenMP uses fork and join model for parallel execution
* OpenMP programs begin with single process: master thread.
* FORK : Master thread creates a team of parallel threads
* JOIN: When the team threads complete the statements in parallel region, they synchronize and terminate leaving master thread.
* Parallelism is added incrementally conversion from a sequential to parallel is a little bit at a time
* Threads based parallelization

Open MP is based on the existence of multiple threads in the shared memory programming paradigm

* Explicit parallelization

It is an explicit (not automatic) programming model, and offers full control over parallelization to the programmer

* Compiler directive based

All of OpenMP parallelization is supported through the use of compiler directives

* Nested parallelism support

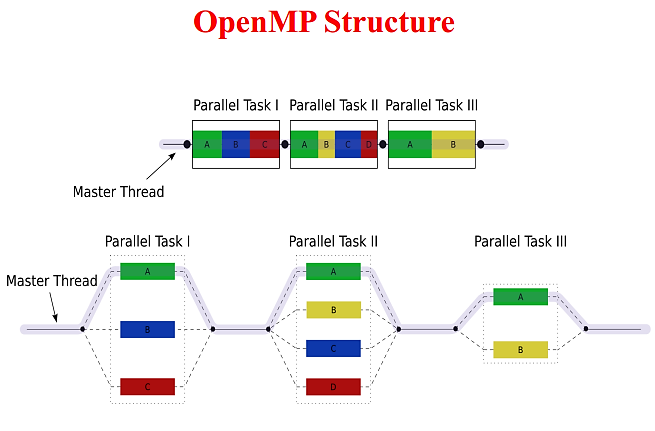
The API support placement of parallel construct inside other parallel construct

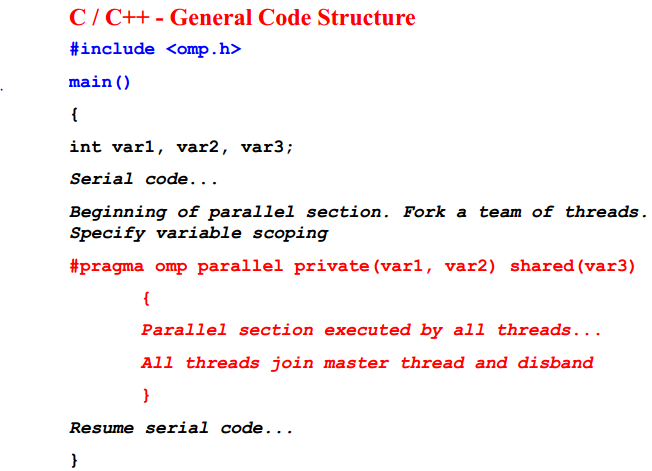
* Dynamic threads

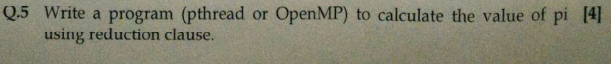
The API provides dynamic altering of number of threads (Depends on the implementation)

How do threads interact?

OpenMP is shared memory model. Threads communicate by sharing variables







Reductions

• Reductions are so common that OpenMP provides support for them

• May add reduction clause to parallel for pragma

• Specify reduction operation and reduction variable

• OpenMP takes care of storing partial results in private variables and combining partial results after the loop

• Local copies are reduced into a single global copy at the end of the construct.

