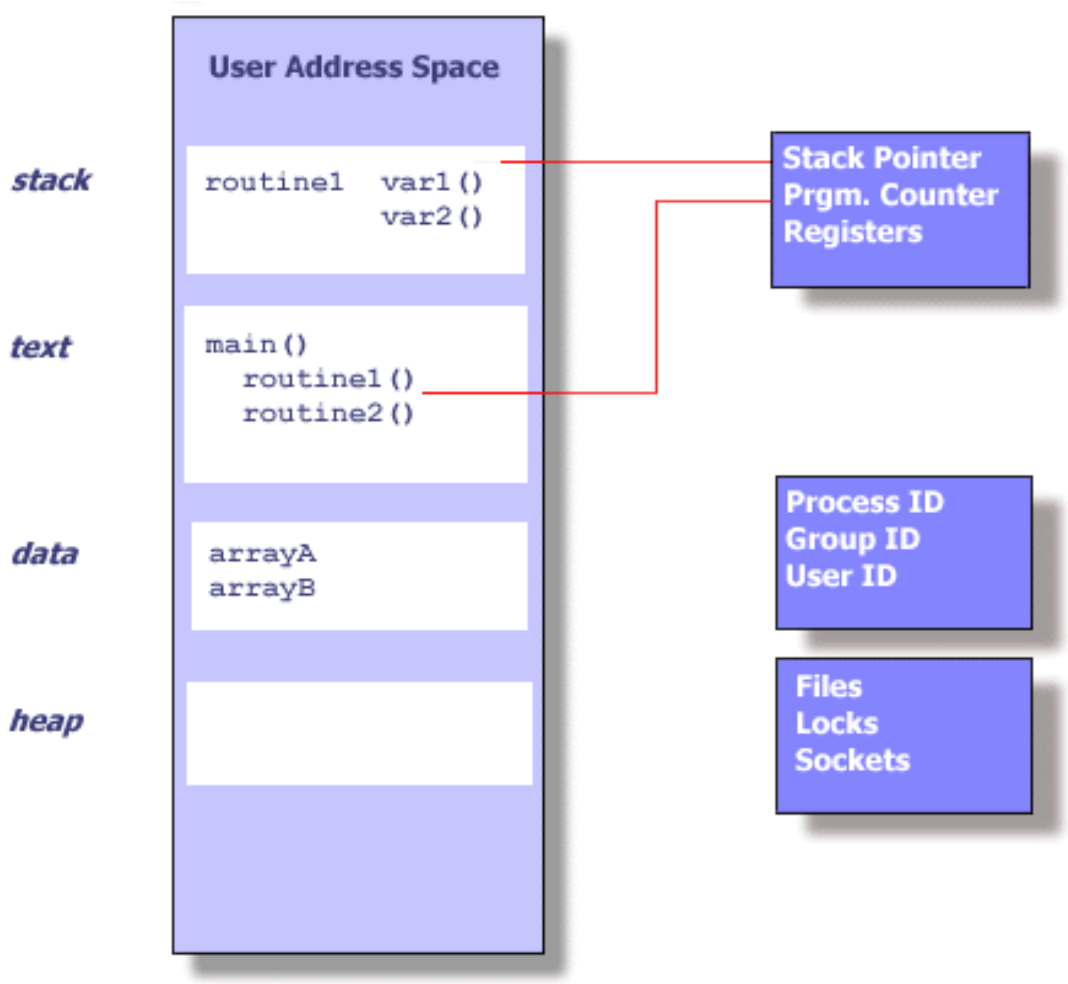
cse5441 - parallel computing

threads

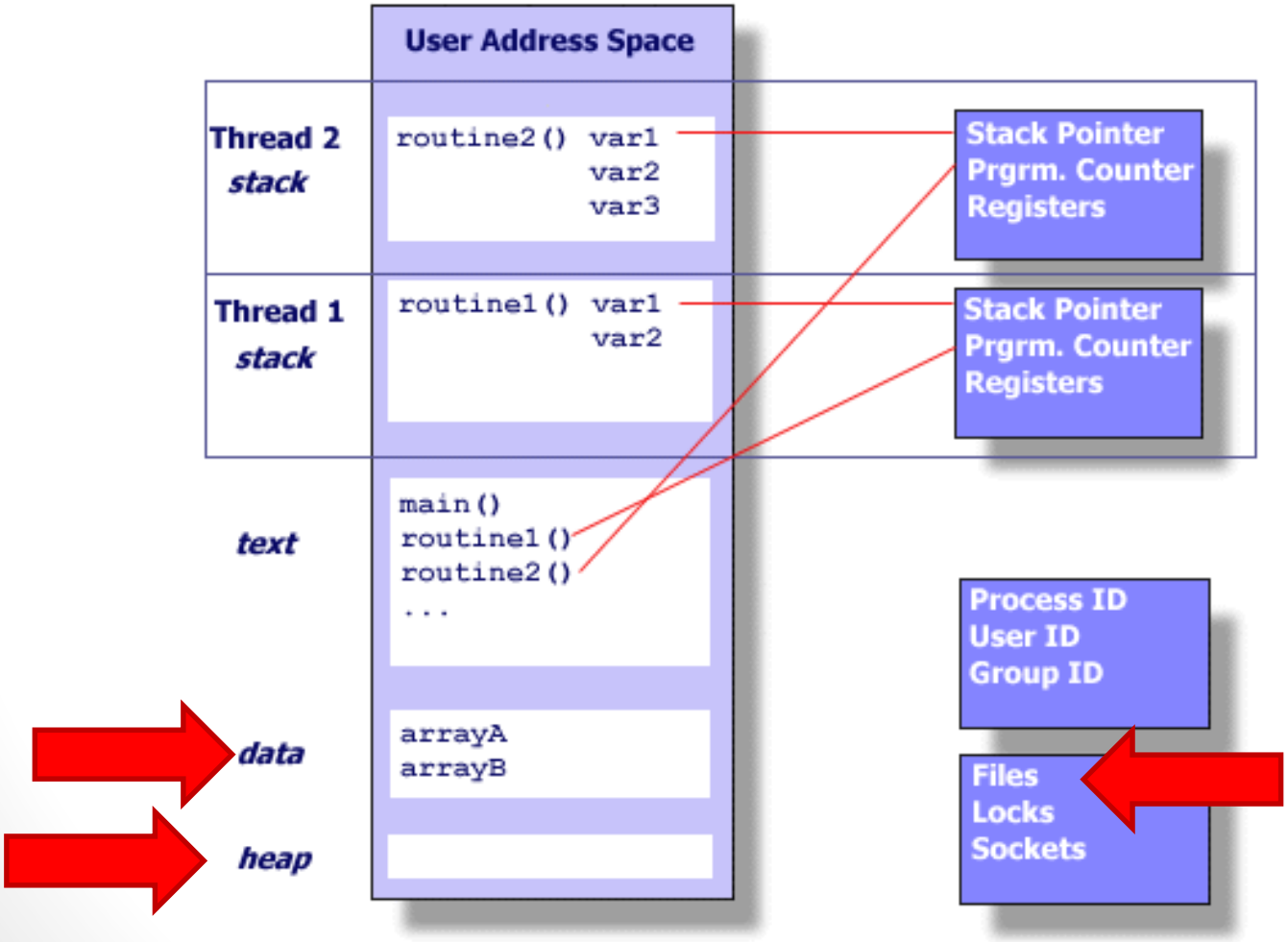
for a comprehensive on-line reference, see:
https://computing.llnl.gov/tutorials/pthreads/

UNIX processes



https://computing.llnl.gov/tutorials/pthreads/images/process.gif

POSIX threads



https://computing.llnl.gov/tutorials/pthreads/images/thread.gif

execution events

Like a process, a thread may be:

scheduled:	pid/tid placed in run queue for next available processor
interrupted:	stopped on the current cpu, may then • await a signal • return to run queue
swapped:	stopped on the current cpu • removed from run queue • memory pages backed to disk

process creation -- fork and exec

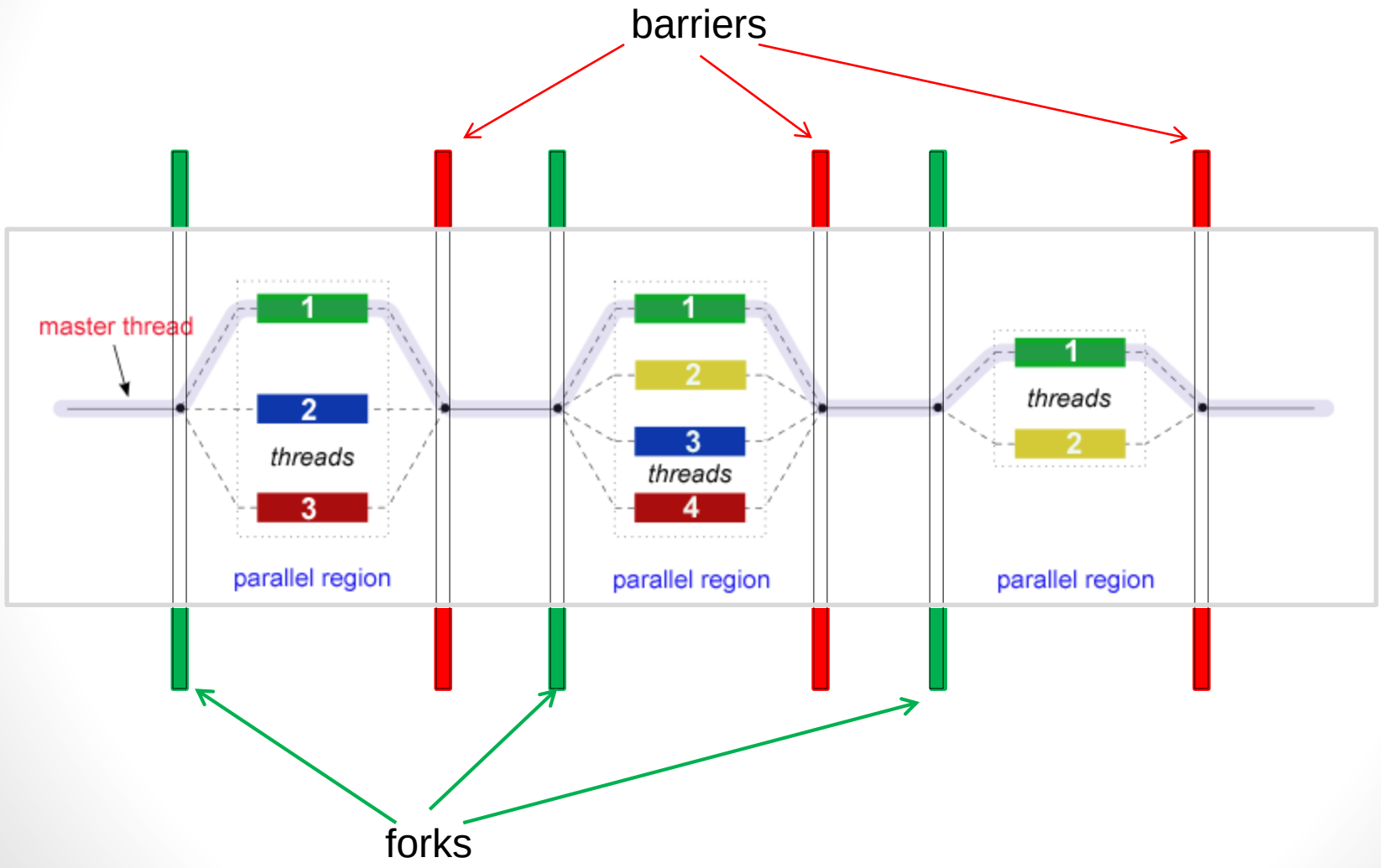
```
printf ("Parent: Hello, World!\n");

f_id = fork ();
if (f_id == 0)
{
        // I am the child
        execvp ("./child", NULL);
}

// I am the parent
```

for threads, the analog is a call to the threads library

threads

	pthreads	OpenMP	MPI
MP model	thread parallel routine	thread parallel region	message passing
memory architecture	local shared	local shared	distributed and shared
communication architecture	shared address	shared address	message passing
MP granularity	coarse or fine	fine	coarse
synchronization	explicit	implicit or explicit	implicit or explicit
API implementation	library	compiler directives	library

fork – join parallelism



https://computing.llnl.gov/tutorials/openMP/images/fork_join2.gif

thread concurrency

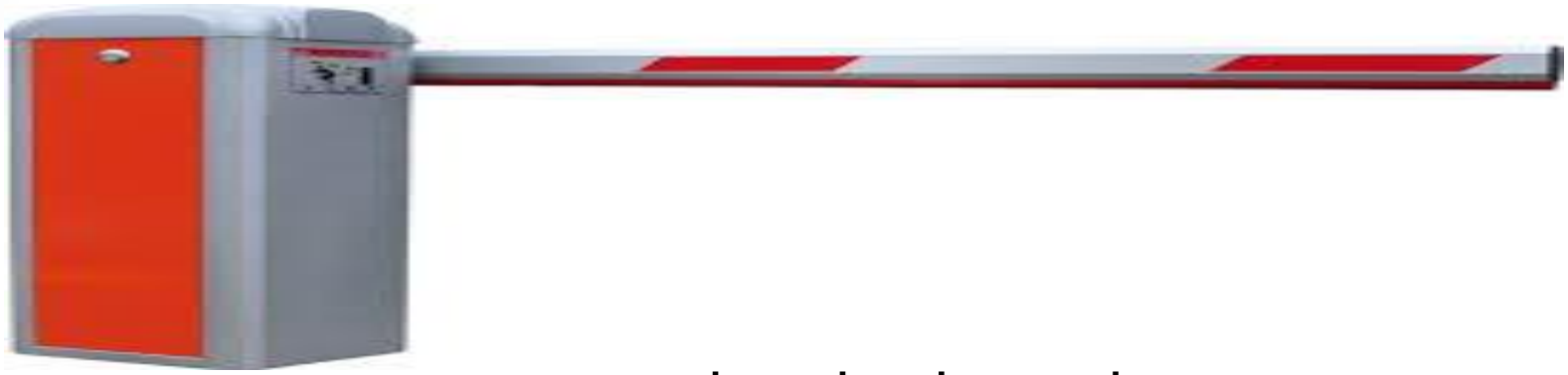
create 10 threads

foreach thread

    print “hello my thread number is” threadnum

```
hello my thread number is 0
hello my thread number is 1
hello my thread number is 4
hello my thread number is 3
hello my thread number is 2
hello my thread number is 7
hello my thread number is 5
hello my thread number is 6
hello my thread number is 9
hello my thread number is 8
```

barriers



- a synchronization point
- arriving threads wait on other threads
- threads released once all have arrived

https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcRL6lfRkguZzUId6rBy4AZxmgNhc9MXS4IhMV92Yp_eOcW1tqsyWg

typical serial scientific app

```
repeat until converged
    for each container
        update domain specific values (DSV)
    communicate updated DSVs
```

typical serial scientific app example

what happens when we want threads?

```
while (( cur_max_dsv – cur_min_dsv) / cur_max_dsv) > epsilon)
{
   for ( cur = 0; cur < num_boxes; cur++ )
   {
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
         cur_n     = grid_boxes[cur].n_top[tn];
         ov_start = max(cur.upper_left_x, cur_n.upper_left_x);
         ov_end   = min(cur.upper_left_x+w, cur_n.upper_left_x+w);
         overlap   = ov_end - ov_start;
         dsv_s     += overlap * cur.dsv_temperature;
      }
      for ( bn = 0; bn < grid_boxes[cur].n_bottom.size(); bn++ )
      {
         cur_n     = grid_boxes[cur].n_bottom[bn];
         ov_start = max(cur.lower_left_x, cur_n.lower_left_x);
         ov_end   = min(cur.lower_left_x+w, cur_n.lower_left_x+w);
         overlap   = ov_end - ov_start;
         dsv_s     += overlap + cur.dsv_temperature;
      . . .
   }
   commit updated DSVs
   update convergence condition
}
```

thread 0

thread 2

thread 1

dsv_s contains newly-computed values which have not been “committed”

typical serial scientific app example

what happens when we want threads?

```
while (( cur_max_dsv – cur_min_dsv) / cur_max_dsv) > epsilon)
{
   for ( cur = 0; cur < num_boxes; cur++ )
   {
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
         cur_n     = grid_boxes[cur].n_top[tn];
         ov_start = max(cur.upper_left_x, cur_n.upper_left_x);
         ov_end  = min(cur.upper_left_x+w, cur_n.upper_left_x+w);
         overlap  = ov_end - ov_start;
         dsv_s[tn]  += overlap * cur.dsv_temperature;
      }
      dvs_s = sum(dsv_s[]);

      for ( bn = 0; bn < grid_boxes[cur].n_bottom.size(); bn++ )
      {
         . . .
         dsv_s[bn]  += overlap + cur.dsv_temperature;
       }
       dvs_s = sum(dsv_s[]);
}
   commit updated DSVs
   update convergence condition
}
```

thread 0

thread 2

thread 1

now each loop is independent

critical “segment” (note this is really a loop ...)

threaded scientific app

disposable threads

```
while (( cur_max_dsv – cur_min_dsv) / cur_max_dsv) > epsilon)
{
   initialize thread data structures;
   create desired number of threads
   for ( cur = 0; cur < num_boxes; cur++ )
   {
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
         ...
         complex computations
         compute DSV updates into working (“temporary”) variables
         ...
      }
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
          ...
         complex computations
         compute DSV updates into working (“temporary”) variables

         ...
      }
   }
   await completion of thread group  (threads exit)
   commit updated DSVs
   update convergence condition
}
```

partial pthreads API

pthread_t ***my_threads**;

pthread_create(*&my_threads[tnum]*, NULL, *threadsafe_function*, (void *)*param*);

pthread_exit((void *)*return_code*);

void **thread_status*;
pthread_join(*my_threads[tnum]*, *&thread_status*);

threaded scientific app

example

```
void  dissapate(float epsilon)
{
 . . .
pthread_t      *threads;
void           *th_status;

   updated_dsv_temperature = new float[num_boxes];
   threads = new pthread_t[num_threads];

   while ( ((cur_max_dsv - cur_min_dsv) / cur_max_dsv) > epsilon )
   {
                //fire up threads to process boxes
                for (long tn = 0; tn < num_threads; tn++)
                {
                   pthread_create(&threads[tn], NULL, dissapate_box, (void *) &tn);
                }

                //join threads prior to updating dsvs
                for (long tn = 0; tn < num_threads; tn++)
                {
                   pthread_join(threads[tn], &th_status);
                }

                //update new dsv values
                for (int i = 0; i < num_boxes; i++)
                {
                   grid_boxes[i].dsv_temperature = updated_dsv_temperature[i];
                }
   }
```

hold on, there ...

all child treads exit

single threaded

parameters -- the gnarly truth

(the dark side) a quick and easy (but incorrect) path ...

passing an integer parameter to thread-safe function:

```
int i;
for (i = 0; i < 10; i++)
{
      pthread_create(&my_threads[i],  NULL, thread_safe_func,  (void *) &i);
}
```

receiving an integer parameter in thread-safe function:

```
void*  thread_safe_func( void*  i )
{
        printf("%d", *((int *) i));
}
```

parameters -- the gnarly truth

(the jedi way) a better way ...

passing an integer parameter to thread-safe function:

```
int    i;
int*  param;
for (i = 0; i < 10; i++)
{
        param  = malloc(sizeof(int));
       *param  = i;
        pthread_create(&my_threads[i],  NULL, thread_safe_func,  (void *) param);
 }
```

receiving an integer parameter in thread-safe function:

```
    void*  thread_safe_func( void*  i )
    {
          printf("%d", *((int *) i));
          free(i);
          pthread_exit((void *) NULL);
    }
```

it's your turn . . .

putting it all together:

```
pthread_t    *my_threads;
pthread_create(&my_threads[tnum],  NULL,  threadsafe_function,  (void *)tnum);

pthread_exit((void *)return_code);

void    *thread_status;
pthread_join(my_threads[tnum],  &thread_status);


void*  thread_safe_func( void*  my_parameter )
```

create program "say_hello," which creates 10 pthreads and prints the message:
"hello world from thread <tid>"

for each thread (where tid is a unique, sequential thread identifier)

threaded scientific app

persistent threads

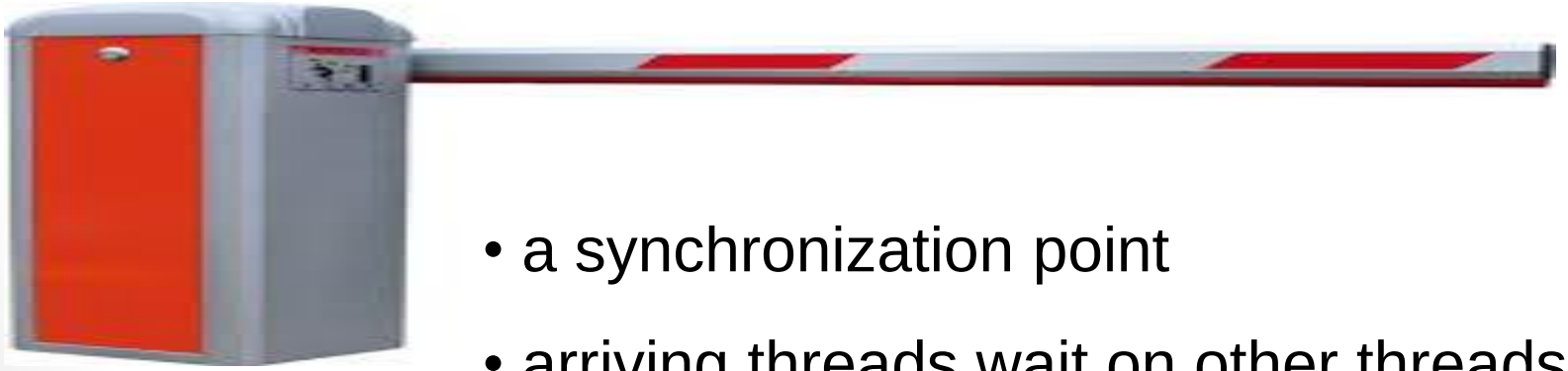
```
initialize thread data structures;
create desired number of threads
while (( cur_max_dsv – cur_min_dsv) / cur_max_dsv) > epsilon)
{
 for ( cur = 0; cur < num_boxes; cur++ )
   {
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
         ...
         complex computations
         compute DSV updates into working (“temporary”) variables
         ...
      }
      for ( tn = 0; tn < grid_boxes[cur].n_top.size(); tn++ )
      {
          ...
         complex computations
         compute DSV updates into working (“temporary”) variables
         ...
      }
   }
   synchronize thread group
   commit updated DSVs
   synchronize thread group
   single thread - update convergence condition
   synchronize thread group
}
await completion of thread group  (threads exit)
```

pthread barrier API

int **pthread_barrier_wait**(pthread_barrier_t **barrier*);

int **pthread_barrier_init**(pthread_barrier_t **barrier*,
pthread_barrierattr_t **attr*, unsigned *count*);

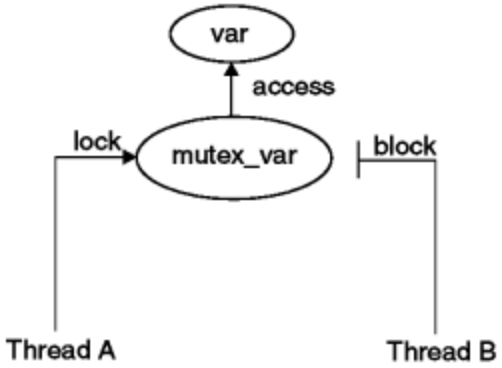
int **pthread_barrier_destroy**(pthread_barrier_t **barrier*);



- a synchronization point
- arriving threads wait on other threads
- threads released once all have arrived

https://encrypted-tbn3.gstatic.com/images?q=tbn:ANd9GcRL6lfRkguZzUId6rBy4AZxmgNhc9MXS4IhMV92Yp_eOcW1tqsyWg

mutual exclusion



int **pthread_mutex_init** (pthread_mutex_t **mutex_lock*,
const pthread_mutexattr_t **lock_attr*);

int **pthread_mutex_lock** (pthread_mutex_t **mutex_lock*);

int **pthread_mutex_unlock** (pthread_mutex_t **mutex_lock*);

https://i.stack.imgur.com/CKmOx.gif

from serial to pthread app

- move partitionable loop into separate function
- consider each data structure used in thread function
  - should it be private to a thread, or master scope?
  - consolidate all shared data structures to single struct
  - instantiate private data structures onto the thread stack
  - for performance, de-couple work buffers
  - object-oriented variable use parameters promote to global
- C++: beware member functions (may prefer to disband class)
- instantiate thread data structures
- insert thread create/exit calls
- insert barriers (join, wait)
- suggested: make num_threads a program parameter

when to use pthreads

- multiple cores available
- large data structures with independent objects
- multi-user time-shared environment
- need for shared memory architecture
- want highest level of thread control

some useful pthread applications

- producer / consumer
- visualization
- sorting
- scientific applications with heavy FP DSVs

when not to use pthreads

- on a single-core single-user system with a single disk
- when application data cannot be segregated into independent pieces
- when functional modules are small, and computations are short (no FP)
- remember Amdahl's law

for a comprehensive on-line reference, see:

https://computing.llnl.gov/tutorials/pthreads/

cse5441 - parallel computing

threads

threads - answers

slide 18:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* say_hello(void* i)
{
    printf("hello world from thread %d\n", *((int *) i));
    free(i);
    pthread_exit((void *) NULL);
}

int main()
{
pthread_t   my_threads[10];
void*               thread_status;
int                 i;
int*                param;

    for (i = 0; i < 10; i++)
    {
                param  = malloc(sizeof(int));
                *param = i;
                pthread_create(&my_threads[i],  NULL,  say_hello,  (void *) param);
    }
    for (i = 0; i < 10; i++)
    {
                pthread_join(my_threads[i],  &thread_status);
    }
}
```