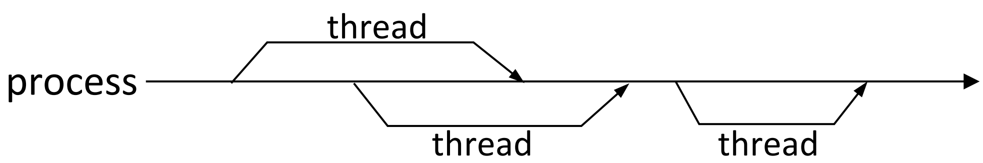
Shared-memory Programming with PThreads (MK book, Ch.4)

PThreads (POSIX Threads) – UNIX based multithreaded programming API

Single process with multiple threads of control



fork/join

<u>Issues:</u>

Critical section – code that updates shared location

Thread synchronization, etc.

ex)
```
#include <pthread.h>
    . . . .
    int thread_count;  //global to all threads (shared)
    . . . .
    int main(..)
    { thread_handles;
      pthread_create( );
      . . . .
      pthread_join( );
    }
    func A (..) { . . . . }
    func Hello (..) { . . . . }
```

```
//more specifically,
   pthread_t* thread_handles;
   thread_handles = malloc( thread_count *
                                      sizeof(pthread_t));
   pthread_create(&thread_handles[index],
                          NULL, Hello, (void*) index);
```

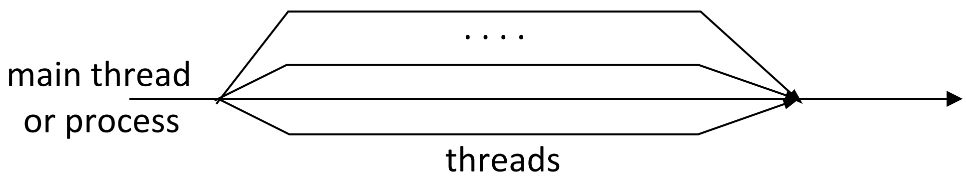
slave func (Hello)   *para to func* ((void*) index)

```
$>gcc –g –wall –o xx  xx.c –lpthread
   //-g for debug,  -wall for warning msgs
```

Each thread has its own stack and heap, i.e., if multiple threads call the same func, each thread has its own copy of func's local vars and parameters.

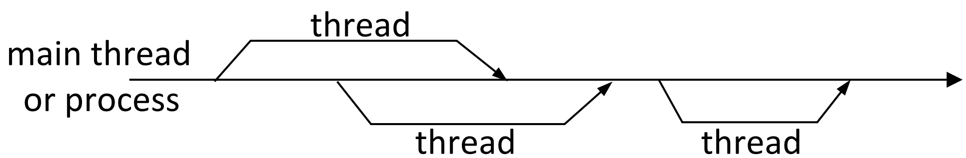
Static vs. dynamic threads:

static: create all threads at the beginning



If no task for a thread, the thread sits idle

dynamic: create thread(s) when needed



more overheads of thread creation

Matrix-vector multiplication with Pthread

$$\underset{A(m*n)}{\begin{bmatrix} a_{0,0} \;.\;.\;.\;.\; & a_{0,n-1} \\ a_{1,0} \;.\;.\;.\;.\; & a_{1,n-1} \\ .\;.\;.\;. & \\ .\;.\;.\;. & \\ a_{m-1,0} \;.\;.\;.\;.\; & a_{m-,\,n-1} \end{bmatrix}} * \underset{X(n*1)}{\begin{bmatrix} x_0 \\ x_1 \\ . \\ . \\ x_{n-1} \end{bmatrix}} = \underset{Y(m*1)}{\begin{bmatrix} y_0 \\ y_1 \\ . \\ . \\ y_{m-1} \end{bmatrix}}$$

$y_0 = a_{0,0}*x_0 + a_{0,1}*x_1 + a_{0,2}*x_2 + \ldots + a_{0,n-1}*x_{n-1}$

$\rightarrow y_i = \sum_{j=0}^{n-1} a_{ij} * x_j$

serial code: vs.

```
for (i=0; i < m; i++)
{ yi = 0.0 ;
  for (j=0; j < n; j++)
      yi += ai,j * xj ;
}
```

parallel scheme:

idea – use 1 thread for multiple i's,
i.e., 1 thread for multiple y_i's.
Threads share accessing X vector, i.e., each thread accesses a part of A and all X.

$$
Y \begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{m-1} \end{bmatrix}
$$

Thread_0 --- $\frac{m}{t}$ entries, where t is the total # of threads

Thread_1 --- $\frac{m}{t}$ entries, where t is the total # of threads

. . .

Thread_q is assigned y_start ~ y_end,

where, start = q * $\frac{m}{t}$; end = (q+1) * $\frac{m}{t}$ - 1

Thread function: – each thread calls
//assume that A, X, Y, m, n are global

```
void* matrix_vector_mult (void* rank)
{ int my_rank = (long) rank;  //type casting from void* to long
  int i, j;
  int local_m = m/thread_count;  //assume m is evenly divisible by t
  int my_first_row = my_rank * local_m;          //q * m/t
  int my_last_row = (my_rank+1)*local_m – 1;  //(q+1) * m/t - 1
  for (i=my_first_row; i<=my_last_row; i++)
  { y[i] = 0.0;  //flush
    for (j=0; j<n; j++)
       y[i] += A[i][j] * X[j];
   }
   return NULL;
 }  //thread_q computes  y (q * m/t) ~ y ((q+1) * m/t – 1)
```

Topics to study:

critical section, mutex
producer/consumer synchronization, semaphores
barrier synchronization, condition variables
read/write locks
cache coherence, false sharing

Critical section

Multiple threads try to access (update) shared memory (globals)

→race condition

ex) compute π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + \boxed{(-1)^n \frac{1}{2n+1}} + \ldots\right); \quad //n=0,1,2, \ldots$$

serial algo:

```
factor = 1.0;
sum = 0.0;
for (i=0; i < n; i++, factor = –factor)
   sum += factor / (2*i + 1);    //sum = sum + (+/-) factor / (2*i+1)
pi = 4.0 * sum;
```

PThread version – <u>first trial</u>: divide n iterations by P

```
//assume n is evenly divisible by P
Slave funcrion(..)
{ . . . .
  my_n = n / num_threads;
  my_first_i = my_rank * my_n;
  my_last_i = my_first_i + my_n;
  if (my_first_i % 2 == 0)   //if my_frist_i is even, +factor; else –factor
     factor = 1.0;
  else
     factor = -1.0;
  for (i=my_first_i; i < my_last_i; i++, factor = –factor)
     sum += factor / (2*i+1);    //sum is global → race condition (c.s.)
 }                               //Sol: use mutex
```

→ Problem: updating global sum is C.S. – not protected

2nd trial: busy-waiting (fine-grained)

```
Slave func(..)
{  . . . .
   for (i=my_first_i; i < my_last_i; i++, factor = –factor)
   {  while (flag != my_rank);   //busy-waiting
      sum += factor / (2*i+1);
      flag = (flag+1) % num_threads;
    }
 }
```

→ Problem: threads are switched for every term → too much overhead

3rd trial: compute local sum and then, busy-waiting (coarse grained)

```
Slave func(..)
{  . . . .
   for (i=my_first_i; i < my_last_i; i++, factor = –factor)
       local_sum += factor / (2*i+1);
   while (flag != my_rank);   //busy-waiting
   sum += local_sum;
   flag = (flag+1) % num_threads;
 }
```

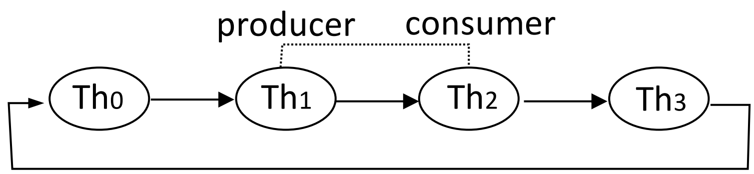
→ still problem of busy-waiting – consumes CPU time uselessly

Note: one advantage of using busy-waiting is that we can order threads.

➔ better solution is using mutex/semaphore – shorter exec time

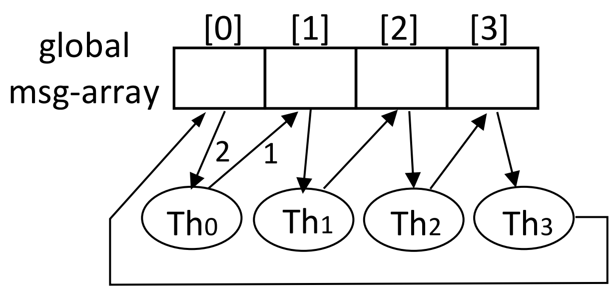
```
pthread_mutex_lock(&mutex1);
sum += local_sum;
pthread_mutex_unlock(&mutex1);
```

Producer/consumer synchronization – with semaphores



In shared-memory Pthread model,

ex)



Task:

1. each thread sends msg to the next thread;
2. then, reads msg sent from the previous thread, and terminates.

Issue: How to synchronize?

e.g., Th_0 should read (receive) msg after Th_3 writes (sends).

Sol: mutex? – No!

Busy-waiting? – OK, but costly

<u>Semaphore</u> – best solution

ref) global sum computation – any order of combine is OK (commutative);

vs. matrix computation, e.g., A * B * C * D = E *//A,B,C,D,E are n*n matrices*

order of computation is important – mutex cannot provide solution.

Thread slave function: a primitive approach with <u>synchronization problem</u>

```
. . . .
my_mag ← “xxx . . . “;
msg_array[dest] ← my_msg;
if (msg_array[my_rank] != NULL)
   display received msg;
else  display “no msg received”;
```

Sol1?: define 1 mutex for each thread?

Since mutex is initialized with 1, it may cause problem.

Sol2: busy-waiting – OK, but costly

```
while (msg_array[my_rank] == NULL);
display received msg;
```

Sol3: <u>semaphores</u> (not a part of PThread; #include<semaphore.h>)

Concept:

```
    msg_array[dest] ← my_msg;
    notify dest;
    wait signal from source (sender);
    display received msg;
```

```
//assume that semaphores are initialized 0 (locked)
msg_array[dest] ← my_msg;
sem_post (&semaphores[dest]);  //notify

sem_wait(&semaphores[my_rank]);
display received msg;
```

```
//program 4.7. A first attempt at sending msgs using Pthreads (not correct)
void *Send_msg(void* rank)
{
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}
-------------------------

//Program 4.8. Using semaphores so that threads can send msgs (correct)
void *Send_msg(void* rank)
{
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank - 1 + thread_count) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   /* Notify destination thread that it can proceed */
   sem_post(sems[dest]);

   /* Wait for source thread to say OK */
   sem_wait(sems[my_rank]);
   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}
```

```
//////////////////////////////////////
//// Park -- semaphore solution to producer-consumer synchronization -- C++ version.
////
//// Each thread sends a message to the next thread, and displays msg received from
//// one previous thread.
//// This version uses named semaphores, since unnamed semaphores aren't available
//// in some systems (e.g., MacOS X, as of 10.6).
////
//// $> g++ -lpthread Prog-4-8-Park.cpp
//// $>./a.out 4  //or any
//////////////////////////////////////

#include <iostream>
#include <pthread.h>   //for pthread
#include <semaphore.h> //for semaphore
#include <cstdlib>     //for atoi()
#include <string>
#include <sstream>     //for int to string conversion
//#include <time.h>    //for checking exec time
using namespace std;

//globals --accessible to all threads
int num_threads;
string* msg_array;     //dynamic array of messages
string* semName_array; //dynamic array of semaphore names (needed for MacOSX)
sem_t** sem_array;     //dynamic array of semaphore ptr's
pthread_mutex_t mutex1;//for atomic cout statement

void *Send_msg(void* rank); //prototype - thread slave function

///////////////////////////////////
int main(int argc, char* argv[])
{
   long thread_id; //long for type casting with void*
   num_threads = atoi(argv[1]); //command line arg - tot num of threads

   pthread_t* myThreads; myThreads=new pthread_t[num_threads];
   //pthread_t myThreads[num_threads]; //simple way --this also works
   pthread_mutex_init(&mutex1, NULL);

   msg_array = new string[num_threads];
   for(thread_id = 0; thread_id < num_threads; thread_id++)
     msg_array[thread_id] = ""; //initialize with empty strings

   sem_array = new sem_t*[num_threads];
   semName_array = new string[num_threads]; //MacOSX needs semaphore name

   for (thread_id = 0; thread_id < num_threads; thread_id++)
   { semName_array[thread_id] = "sem_"+thread_id;
     sem_array[thread_id] = sem_open(semName_array[thread_id].c_str(), O_CREAT, 0777, 0);
     //initialize sem to 0 (locked); sem_open needs c_str type;
   }
   for (thread_id = 0; thread_id < num_threads; thread_id++)
     pthread_create(&myThreads[thread_id], NULL, Send_msg, (void*) thread_id);

   for (thread_id = 0; thread_id < num_threads; thread_id++)
     pthread_join(myThreads[thread_id], NULL);

   for (thread_id = 0; thread_id < num_threads; thread_id++)
   { sem_unlink(semName_array[thread_id].c_str());
     sem_close(sem_array[thread_id]);
   }
   //delete[] sem_array; delete[] semName_array; delete[] msg_array; delete[] myThreads;
   return 0;
}//main
```

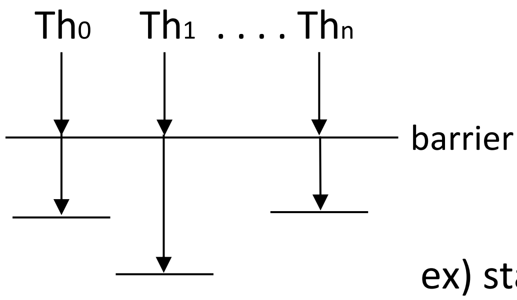
```
////////////////////////////
void *Send_msg(void* rank)
{
   int my_rank = (long) rank;
   int dest = (my_rank + 1) % num_threads; //dest is one next thread (rotational)
   int source = (my_rank - 1 + num_threads) % num_threads; //source is one prev thread

   //type casting from int to string --needs #include <sstream>
   stringstream out1, out2;
   out1<<dest;
   out2<<my_rank;
   string dest_str = out1.str();
   string my_rank_str = out2.str();

   string my_msg = "Hello to Thread_"+dest_str+" from Thread_"+my_rank_str;
   msg_array[dest] = my_msg;  //send msg to dest
   sem_post(sem_array[dest]); //notify to dest

   sem_wait(sem_array[my_rank]);  //wait until source notifies me
   if (msg_array[my_rank] != "")
   { pthread_mutex_lock(&mutex1); //make cout atomic
     cout<<"Thread_"<<my_rank<<" > "<<msg_array[my_rank]<<endl;
     pthread_mutex_unlock(&mutex1);
   }
   else
   { pthread_mutex_lock(&mutex1); //make cout atomic
     cout<<"Thread_"<<my_rank<<" > No message from Thread_"<<source<<endl;
     pthread_mutex_unlock(&mutex1);
   }
   return NULL;
}
```

Barriers and Condition variables



In the case, all threads should be at the same point in a program.

ex) start from the same point in a program,

check for the slowest thread's finish time.

Implementation of barrier

Most Pthread implementations do not support barrier construct (Open Group provides).

busy-waiting/mutex?

semaphore?

condition variables

1. Busy-waiting + mutex?

```
global counter = 0;
. . . .
in slave function,
    . . . .
    mutex_lock(&barrier_mutex);
    counter++;
    mutex_unlock(&barrier_mutex);
    while (counter < num_threads);  //barrier
    . . . .
```

→ Problem when we need to use 2nd barrier, etc.

Sol: using 1 counter (global) per barrier event.

2. Semaphore?

globals: counter = 0; count_sem = 1; barrier_sem = 0;

. . . .

In slave function,

```
sem_wait (&count_sem);
If (counter == num_threads -1)  //if last thread
{  counter = 0;  //for next barrier event
   sem_post (&count_sem);
   for (i=0; i < num_threads-1; i++)  //serial, awakens all other threads
      sem_post (&barrier_sem);
}
else
{  counter++;
   sem_post (&count_sem);
   sem_wait (&barrier_sem);
}
```

ref:
mutex – always starts from 1 and value (1/0);
semaphore – can be initialized with any value;

→ not a complete solution: may cause <u>race condition</u> – when reusing barrier semaphore for further barrier events.

3. <u>Condition variables</u> – Pthread supports

suspends execution until a condition (event) occurs;

In fact, condition var. and mutex together work.

Concept:

```
lock (mutex);
if condition
   signal thread(s);
else
   unlock mutex;
   block;
unlock (mutex);
```

C.S.

wait

```
pthread_cond_signal (&cond_var);       //unblocks 1 blocked thread

pthread_cond_broadcast (&cond_var);  //unblocks all blocked threads
```

unblock

```
pthread_cond_wait (&cond_var, &mutex);  //unblocks mutex and
                                        //blocks threads on cond_var
```

<u>Barrier with cond_var</u>

```
globals:   counter = 0;
           mutex;
           cond_var;
 . . . .
 in slave func,
    pthread_mutex_lock (&mutex);
    counter++;
    if (counter == num_threads)  //if condition
        counter = 0;
        pthread_cond_broadcast (&cond_var)
    else
        while (pthread_cond_wait (&cond_var, &mutex) != 0) ;
    pthread_mutex_unlock (&mutex);
```

C.S.

lock on this   *unlock this*

If some other signal causes unblock, return value ≠ 0, so, wait again (using while loop).

Read/write locks -- Pthread supports

Problem: multiple threads read/write on a shared big data structure;

Sol: 1. one mutex for the entire data structure;

2. one mutex for each element in the data structure; --worst timing

3. using pthread read/write locks (CREW); -- best timing

concept (CREW):

if one is reading, any writer should wait;

if one is writing, any reader/writer should wait;

→ implemented with 3 operations on rwlock:

rdlock, wrlock, unlock

syntax:

```
pthread_rwlock_t  rwlock;
pthread_rwlock_init (&rwlock);
. . . .
pthread_rwlock_destroy (&rwlock);
```

ex)

```
pthread_rwlock_rdlock (&rwlock);
    reading operation here;
pthread_rwlock_unlock (&rwlock);

pthread_rwlock_wrlock (&rwlock);
    writing (updating) operation here;
pthread_rwlock_unlock (&rwlock);
```

ref: in the reality, using serial code (1 thread) takes shorter than using multiple threads with heavy lock/unlock operations.

Caches, cache coherence and false sharing

Cache access pattern and program execution performance:

ex) C/C++ use row-major layout (e.g., 2D array → row-major 1D layout)

4x4 array A stored in memory (block size = 4)

mem_block_0:	A_{00}	A_{01}	A_{02}	A_{03}
mem_bleck_1:	A_{10}	A_{11}	A_{12}	A_{13}
mem_block_2:	A_{20}	A_{21}	A_{22}	A_{23}
mem_block_3:	A_{30}	A_{31}	A_{32}	A_{33}

Assume: direct-mapped cache;
cache total size = 2 blocks;

cache_block_0:	x	x	x	x
cache_bleck_1:	x	x	x	x

1. row-major access → yields 4 cache misses

```
for (i=0; i<4; i++)
    for (j=0; j<4; j++)
        y[i] += A[i][j]*s;
```

vs. 2. column-major access → yields 16 cache misses

```
for (j=0; j<4; j++)
    for (i=0; i<4; i++)
        y[i] += A[i][j]*s;
```

Cache coherence:

3 sources of incoherency:

- sharing of writable data
- process migration
- I/O bypasses cache (DMA – direct memory access between mem and I/O)

Solutions:

- common cache
- snoopy bus: write update vs. write invalidate (e.g., MESI protocol)
- directory based protocol (DSM): local/remote/home node

False sharing:

Multiple threads access different variables, which belong to the same cache line (block).

Updating thread makes the line invalid and other threads miss the cache line and access lower level memory.