Thread-level Parallelism

Mikhail Chalabine
Linköping University

TDDC 78 Labs: Memory-based Taxonomy

Memory	Lab(s)	Use
Distributed	1	MPI
Shared	2 3	Posix threads OpenMP
Distributed	4	MPI

LAB 5 (tools) at every stage; save your time

Lab 3 – Stationary Heat Equation

- **Problem**
  - Find stationary temperature distribution in a square given some boundary temperature distribution
  - SHMEM, OpenMP
  - Serial code in Fortran
- **Solution**
  - Requires solving differential equation
  - Iterative Jacobi method
  - Algorithm detailed in Compendium
- **Primary concern**
  - Synchronize access
  - *O(N)* extra memory

Why multithreading?

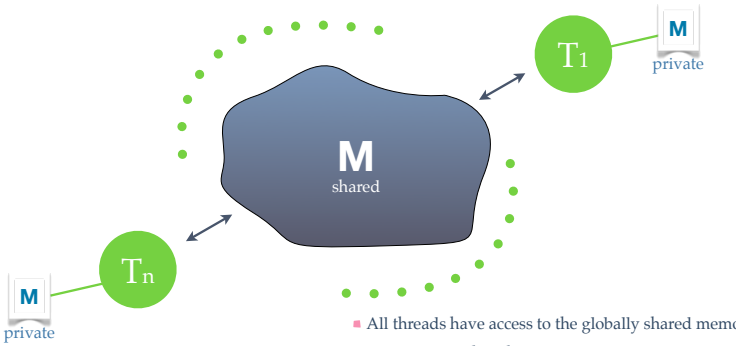
- **Improve application responsiveness**
  - I/O in parallel with other tasks
- **Use multiprocessors more efficiently**
  - Parallel threads in parallel processes (unbound!)
- **Use fewer system resources than processes**
  - A thread is faster to create/terminate than a process
  - Less time to context switch
  - Efficient memory utilization: local and global SHMEM
- **Easier to program and debug**
  - Shared-memory *vs.* Distributed-memory model

Process vs. Thread

- **Process**
  - Running program, UNIX environment created with fork
  - Unit of resource ownership
    - Virtual address space
    - Control of resources (I/O, file system, etc.)
  - Unit of dispatching / scheduling (system)
    - A process execution can be interleaved with execution of other processes
    - The process has an execution state and a dispatching priority
- **Thread**
  - Sequence of instructions executed within the context of a process

Simultaneous multithreading = N threads + N hardware threads + superscalar architecture

Threads and Shared Memory

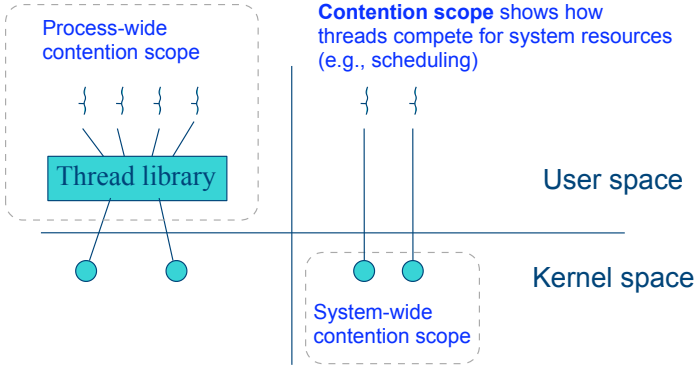


- All threads have access to the globally shared memory
- Data can be either shared or private:
  - Shared data is visible to all threads
  - Private data is visible to one thread only
- Data transfer is transparent to the programmer

Thread Taxonomy (1)

- **Thread libraries**
  - **Solaris** threads
  - **Native implementations**, e.g., Native POSIX Thread Library (NPTL) on Linux
    - **POSIX** is a standard (free to implement)
- **Thread levels**
  - **Application-level** threads (user-level)
    - Visible to the application programmer
  - **Lightweight processes** (LWP) (kernel threads)
    - Visible to the operating system kernel
    - Virtual CPUs (> physical CPUs)
    - Thread libraries schedule threads on LWPs

Thread Taxonomy (2)



Unbound
TL schedules threads
to available LWPs

Bound
Threads contend with other
processes in the system.
All are scheduled by the kernel.

Thread Taxonomy (3)

- **Solaris threads**
  - **Many-to-many** mapping between CPUs, LWPs and Threads (*N*:*M*)
  - Fast context switch between threads
    - No kernel system calls required
  - Complex implementation to schedule ***N*** user threads onto ***M*** kernel threads while preserving efficiency
  - Priority inversion can be a serious issue

Thread Taxonomy (4)

- **Linux threads**
  - **One-to-one mapping** (bound threads only)
    - One thread per LWP, *i.e.*, 1:1 Thread-to-LWP mapping
    - Threads are schedulable entities (systemwide contention scope)
    - `PTHREAD_SCOPE_PROCESS` not supported:
    - **NGPT** IBM (Next Generation POSIX Threading) development stopped in 2003
    - **NPTL** (Native POSIX Thread Library) Red Hat, > Kernel 2.6
      - Threads look as processes to the kernel, *i.e.*,
        - Bound threads only
        - Need for *clone()* system call to create a thread
      - Better kernel support
      - Simpler implementation than for N:M
      - Simpler scheduling algorithms (efficiency, speed)

NPTL

- POSIX compliance
- Effective Use Of SMP
- Low Startup Cost
- Low Link-In Cost
- Hardware Scalability
- Software Scalability
- NUMA Support
- Integration With C++
- Higher cost for context switch than for unbound
- 1:1 mapping implies simpler handling of
  - Priority inversion; Race conditions; etc.

NPTL on Windows

- No native support
- CYGWIN implementation
- pthreads-win32

- Windows threads are claimed to be
  - More efficient (Windows)
  - Simpler to program

Concurrency vs. Parallelism

- **Parallel**
  - Two or more threads are executing simultaneously
  - Not possible on uniprocessor machines
- **Concurrent**
  - Two or more threads are making progress
  - More generic situation, can include time-slicing
  - Tasks run in any order and possibly in parallel

Main Concept: Synchronization

- Different from MPI's Send-Receive
- Thread safety = protect shared data
- Deterministic behavior
- Synchronization objects:
  - **Mutex Locks** (Mutual Exclusion)
    - Serialize access to shared resources
  - **Condition Variables**
    - Block a thread until a (global) condition is true
  - **Semaphores**
    - Block a thread until count is positive

Hello world

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   4

void *PrintHello(void *threadId) {
   long tId;
   tId = (long)threadId;
   printf("Hello World! It's thread #%ld!\n", tId);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
pthread_t threads[NUM_THREADS];
int ret;
long t;
for(t=0;t<NUM_THREADS;t++){
  printf("In main: creating thread %ld\n", t);
  ret = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  if (ret){
    printf("ERROR! Return code: %d\n", ret);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```


Passing a single parameter

```
long *param[NUM_THREADS];
...
for(t=0; t<NUM_THREADS; t++)
{
   param[t] = (long *) malloc(sizeof(long));
   *param[t] = t;
   printf("Creating thread %ld\n", t);
   ret = pthread_create(&threads[t], NULL, PrintHello, (void *) param[t]);
   ...
}
```

```
long param = ...;
...
for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   ret = pthread_create(&threads[t], NULL, PrintHello, (void *) &param);
   ...
}
```

Some thread can modify content at the &param address before all threads are reated!

Passing multiple parameters

```
struct thread_data{
       int  threadId;
       char *msg;
    };

    struct thread_data thread_data_array[NUM_THREADS];

    void *PrintHello(void *tParam)
    {
       struct thread_data *myData;
       ...
       myData = (struct thread_data *) tParam;
       taskId = myData->threadId;
       helloMsg = myData->msg;
       ...
    }

    int main (int argc, char *argv[])
    {
       ...
       thread_data_array[t].threadId = t;
       thread_data_array[t].Msg = msgPool[t];
       rc = pthread_create(&threads[t], NULL, PrintHello,
            (void *) &thread_data_array[t]);
       ...
    }
```


Mutex lock example

```
#include<pthread.h>

pthread_mutex_t count_mutex;
long count;

void increment_count(){
  pthread_mutex_lock(&count_mutex);
  count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long get_count(){
    long  c;
     pthread_mutex_lock(&count_mutex);
     c = count;
    pthread_mutex_unlock(&count_mutex);
     return (c);
}
```

Attach locks to resources

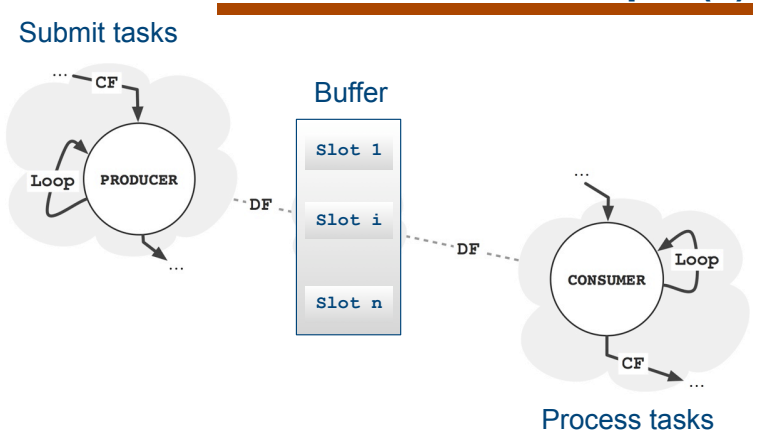
Conditional variables example

```
pthread_mutex_t count_lock;
pthread_cond_t count_positive;
long count;

decrement_count() {
   pthread_mutex_lock(&count_lock);
   while (count <= 0)
      pthread_cond_wait(&count_positive, &count_lock);
   count = count - 1;
   pthread_mutex_unlock(&count_lock);
}

increment_count() {
   pthread_mutex_lock(&count_lock);
   count = count + 1;
   if (count > 0)
      pthread_cond_signal(&count_positive);
   pthread_mutex_unlock(&count_lock);
}
```


Semaphores

- Coordinate access to resources
  - **Initialize** to the number of free resources
  - Atomically **increment** the count when resources are added
  - Atomically **decrement** the count when resources are removed.
  - Threads **block wait** on decrement until the count becomes greater than zero.

Producer-consumer Example (1)



Producer-consumer Example (2)

- **Synchronization requirements**
  - Mutual exclusion
    - When working with the buffer
  - Among **producers**
    - Waiting for an available slot
  - Among **consumers**
    - Waiting for an available task

Producer-consumer Example (3)

Buffer data structure

```
typedef struct {
 char buf[BSIZE];          /* Slots */
 int nextin;               /* Next slot to be used by producer */
 int nextout;              /* Next slot to be used by consumer */
 pthread_mutex_t mutex;    /* Buffer access mutex  */
 sem_t occupied;           /* Number of occupied slots */
 sem_t empty;              /* Number of empty slots */
} buffer_t;
```


Producer-consumer Example (4)

Initialization

```
sem_init( &buffer.occupied, 0, 0);           // Set to no tasks
sem_init( &buffer.empty,    0, BSIZE);       // Set to all slots free

buffer.nextin = buffer.nextout = 0;          // Set start to buffer top

pthread_mutex_init( &buffer.mutex, NULL);    // Init mutex

pthread_attr_t  tattr;                       // Set thread attributes
pthread_attr_init( &tattr);
pthread_attr_setscope( &tattr, PTHREAD_SCOPE_SYSTEM);

pthread_t       tid;
void*           start_routine(void*);
void*           arg;
pthread_create (&tid, &tattr, start_routine, arg); /* Create */
```

Producer-consumer Example (5)

Producer

```
void producer( buffer_t *b, char item) {
  sem_wait( &b->empty);              // Wait for empty slot
  pthread_mutex_lock( &b->mutex);    // Lock the buffer

  b->buf[b->nextin] = item;          // Put an item into buffer
  b->nextin++;                       // Set new buffer top
  b->nextin %= BSIZE;

  pthread_mutex_unlock( &b->mutex);  // Release the lock
  sem_post(&b->occupied);            // Wake up consumer
}
```


Producer-consumer Example (5)

Consumer

```
char consumer(buffer_t *b) {
  char item;                         // Item to process

  sem_wait(&b->occupied);            // Wait for task
  pthread_mutex_lock(&b->mutex);     // Lock the buffer

  item = b->buf[b->nextout];         // Get item
  b->nextout++;                      // Set new buffer top (out)
  b->nextout %= BSIZE;
  pthread_mutex_unlock(&b->mutex);   // Release the lock
  sem_post(&b->empty);               // Signal empty slot
  return(item);
}
```


Compiling and linking

- Don't forget to include
  - `pthread.h`, `semaphore.h`
- Link with
  - `-lpthread`, `-lposix4`

Typical problems

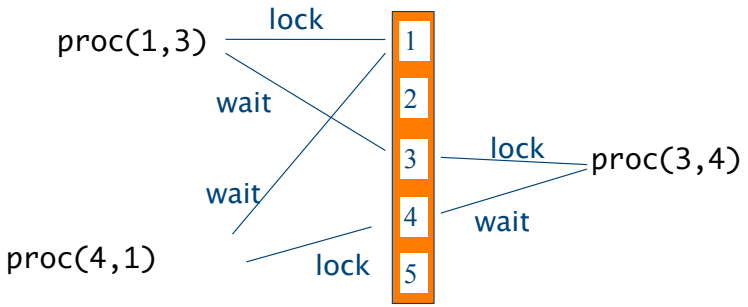
- **Uninitialized variables**
  - Uninitialized synchronization objects lead to strange behavior
  - Tip: check the return codes!
- **Deadlocks (>= 2 waiting for each other)**
- **Race conditions (one misbehaves)**
- **Poor performance**
  - To many synchronizations
  - Cache effects kill gains of using multiprocessors

Deadlock example (1)

- Suppose we have a shared array of items and we need to process two items at a time

```
typedef struct {
     .....
     pthread_mutex_t mutex;
    } item_t;
    void proc(unsigned i, unsigned j) {
        item_t tmp;
        pthread_mutex_lock(&(array[i].mutex));
        pthread_mutex_lock(&(array[j].mutex));
        ...
        pthread_mutex_unlock(&(array[i].mutex));
        pthread_mutex_unlock(&(array[j].mutex));
    }
    item_t array[ASIZE];
```


Deadlock example (2)



Order resources to be locked!

Deadlock example (3)

```
    void proc(unsigned i, unsigned j) {
    item_t tmp;
    int itmp;

    /* keep indices in order ! */
    if( i > j) {
     itmp = i;
     i = j;
     j = itmp;
    }
    pthread_mutex_lock(&(array[i].mutex));
    pthread_mutex_lock(&(array[j].mutex));
    ...
    pthread_mutex_unlock(&(array[i].mutex));
    pthread_mutex_unlock(&(array[j].mutex));
}
```

Order resources to be locked!

Summary and goals for your lab

- Understand
  - Threads and their use
  - Synchronization vs. Send / Receive
  - Resource ordering
  - Low level parallelism vs. Higher-level specification in OpenMP
- Implement
  - Filters as in Lab 1