PThreads for Shared Memory Systems

Chris Kauffman

Last Updated:
Tue Mar 22 09:37:27 AM CDT 2022

Logistics

This Week

- POSIX Threads Briefly
- Java Threads (maybe)
- OpenMP - automated threads

Reading

- Grama 7.1-9 (PThreads)
- POSIX Threads Programming Tutorial

Mini-Exam 1 Grades Up

- Regrade requests until Tue 3/29 11:59pm
- Mini-Exam 2 Grades later this week

A2 Coming

- Look for post later this week
- K-means implementation
- Serial, MPI (distributed), and OpenMP (Shared Mem)

Preamble

Assumptions

- You've taken an Intro OS Course (like CSCI 4061)
- You're familiar with Unix Processes
- You've probably seen threaded programming before

PThreads Learning Approach

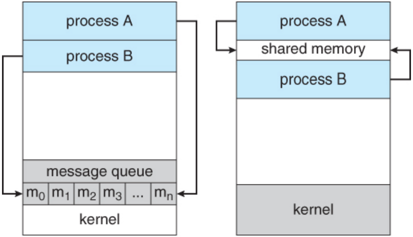
- Review functions/data to run threads
- Introductory example to demonstrate threads doing cooperative computation
- Surmount difficulties associated with coordinating threads AND maintain speed
- Later will look at OpenMP: an easier approach to shared memory programming

Processes vs Threads

Process in IPC	Threads in pthreads
(Marginally) Longer startup	(Marginally) Faster startup
Must share memory explicitly	Memory shared by default
Good protection between processes	Little protection between threads
`fork() / waitpid()`	`pthread_create() / _join()`

Modern systems (Linux) can use semaphores / mutexes / shared memory / message queues / condition variables to coordinate Processes or Threads

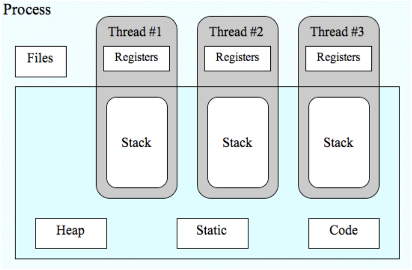
IPC Memory Model



(a) (b)

Source

Thread Memory Model



Source

PThreads Library and Shared Memory Parallelism

- **POSIX** Threading Library - POSIX is a UNIX standard adhered to by many OS's (Linux, BSD, MacOSX, even Windows [sort of])
- PThreads are reasonably portable (run the same between different architectures / OS's)
- PThreads allow use of **shared memory parallelism** on single machines with multiple processors / cores as the OS can execute each thread on a different core

Process and Thread Functions

- Threads and process both represent "flows of control"
- Most ideas have analogs for both

Processes	Threads	Description
`fork()`	`pthread_create()`	create a new flow of control
`waitpid()`	`pthread_join()`	get exit status from flow of control
`getpid()`	`pthread_self()`	get "ID" for flow of control
`exit()`	`pthread_exit()`	exit (normally) from an existing flow of control
`abort()`	`pthread_cancel()`	request abnormal termination of flow of control
`atexit()`	`pthread_cleanup_push()`	register function to be called at exit from flow of control

Stevens/Rago Figure 11.6: Comparison of process and thread primitives

Thread Creation

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);

int pthread_join(pthread_t thread, void **retval);
```

- Start a thread running function `start_routine`
- `attr` may be `NULL` for default attributes
- Pass arguments `arg` to the function
- Wait for thread to finish, put return in `retval`

Minimal Example of PThreads

```c
// pthreads_minimal.c: Minimal example of starting a
// pthread, passing a parameter to the thread function, then
// waiting for it to finish. Two threads are launched.
#include <pthread.h>
#include <stdio.h>

void *fx(void *param){
  int p=(int) param;
  p = p*2;
  return (void *) p;
}

int main(){
  pthread_t thread_1, thread_2;
  pthread_create(&thread_1, NULL, fx, (void *) 42);
  pthread_create(&thread_2, NULL, fx, (void *) 65);
  int res1, res2;
  pthread_join(thread_1, (void **) &res1);
  pthread_join(thread_2, (void **) &res2);
  printf("results are: %d %d\n",res1,res2);
  return 0;
}
```

Compilation

```
>> gcc pthreads_minimal.c -lpthread
pthreads_minimal.c: In function ‘fx’:
pthreads_minimal.c:8:9:
   warning: cast from pointer to integer
   of different size [-Wpointer-to-int-cast]
    8 |   int p=(int) param;
      |         ^
pthreads_minimal.c:10:10:
   warning: cast to pointer from integer
   of different size [-Wint-to-pointer-cast]
   10 |   return (void *) p;
      |          ^

>> ./a.out
results are: 84 130
```

- Note compiler complaints about casting
- In recent `gcc` + `glibc`, may no longer need `-lpthread`

Exercise: Observe this about pthreads

1. Where does a thread start execution?
2. What does the parent thread do on creating a child thread?
3. How much compiler support do you get with pthreads?
4. How does one pass multiple arguments to a thread function?
5. If multiple children are spawned, which execute?
6. What is the arrangement of the function call stack for threads?

Answers: Observe this about pthreads

1. Where does a thread start execution?
   - Child thread starts running code in the function passed to `pthread_create()`, function `doit()` in example
2. What does the parent thread do on creating a child thread?
   - Continues immediately, much like `fork()` but child runs the given function while parent continues as is
3. How much compiler support do you get with pthreads?
   - Little: must do a lot of casting of arguments/returns
4. How does one pass multiple arguments to a thread function?
   - Create a struct or array and pass in a pointer
5. If multiple children are spawned, which execute?
   - Can't say which order they will execute in, similar to `fork()` and children
6. What is the arrangement of the function call stack for threads?
   - Each thread has its own function call stack within the same memory image of the managing process

Motivation for Threads

- Like use of `fork()`, threads increase program complexity
- **Improving execution efficiency** is a primary motivator
- Assign independent tasks in program to different threads
- 2 common ways this can speed up program runs

1 Parallel Execution with Threads

- Each thread/task computes part of an answer and then results are combined to form the total solution
- Discuss in Lecture (Pi Calculation)
- REQUIRES multiple CPUs to improve on Single thread

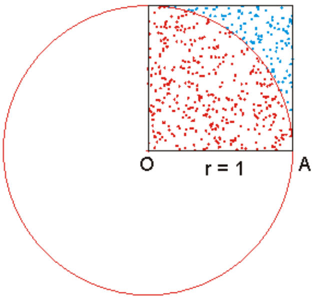
2 Hide Latency of Slow Tasks in a Program

- Slow tasks block a thread but Fast tasks can proceed independently allowing program to stay busy while running
- Does NOT require multiple CPUs to get benefit Why?

Model Problem: A Slice of Pi

- Calculate the value of $\pi \approx 3.14159$
- Simple *Monte Carlo* algorithm to do this
- Randomly generate positive `(x,y)` coords
- Compute distance between `(x,y)` and `(0,0)`
- If distance ≤ 1 increment "hits"
- Counting number of points in the positive quarter circle
- After large number of hits, have approximation

$$\pi \approx 4 \times \frac{\text{total hits}}{\text{total tries}}$$



Algorithm generates dots, computes fraction of red which indicates area of quarter circle compared to square

picalc_serial.c and picalc_pthreads_broken.c

- Examine source code for picalc_serial.c
- Uses rand_r() function to generate random numbers rather than more typical rand() function
- Will become apparent why in a moment
- Note basic algorithm is simple and easily parallelizable
- Done in obvious way in picalc_pthreads_broken.c
- Observe incorrect results and attempt to explain why

Why is `pthreads_picalc_broken.c` so wrong?

- The instructions `total_hits++;` is **not atomic**
- Translates to assembly

```
// total_hits stored at address #1024
30: load  REG1 from #1024
31: increment REG1
32: store REG1 into #1024
```

- Interleaving of these instructions by several threads leads to undercounting `total_hits`

Mem #1024 `total_hits`	Thread 1 Instruction	REG1 Value	Thread 2 Instruction	REG1 Value
100				
	30: load REG1	100		
	31: incr REG1	101		
101	32: store REG1			
			30: load REG1	101
			31: incr REG1	102
102			32: store REG1	
	30: load REG1	102		
	31: incr REG1	103		
			30: load REG1	**102**
			31: incr REG1	103
103			32: store REG1	
103	32: store REG1			

Critical Regions and Mutex Locks

- Access to shared variables must be coordinated among threads
- A **mutex** allows *mutual exclusion*
- Locking a mutex is an atomic operation like incrementing/decrementing a semaphore

```
pthread_mutex_t lock;

int main(){
  // initialize a lock
  pthread_mutex_init(&lock, NULL);
  ...;
  // release lock resources
  pthread_mutex_destroy(&lock);
}

void *thread_work(void *arg){
  ...
  // block until lock acquired
  pthread_mutex_lock(&lock);

  do critical;
  stuff in here;

  // unlock for others
  pthread_mutex_unlock(&lock);
  ...
}
```

Protecting Critical Region in picalc

```
int total_hits=0;
int points_per_thread = ...;
pthread_mutex_t lock;                  // initialized in main()

void *compute_pi(void *arg){
  long thread_id = (long) arg;
  unsigned int rstate = 123456789 * thread_id;
  for (int i = 0; i < points_per_thread; i++) {
    double x = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
    double y = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
    if (x*x + y*y <= 1.0){
      pthread_mutex_lock(&lock);    // lock global variable
      total_hits++;                 // update
      pthread_mutex_unlock(&lock);  // unlock global variable
    }
  }
  return NULL;
}
```

time Utility Reports 3 Times

```
# 'time prog args' reports 3 times for program runs
# - real: amount of "wall" clock time, how long you have to wait
# - user: CPU time used by program, sum of ALL threads in use
# - sys : amount of CPU time OS spends in system calls for program

> time seq 10000000 > /dev/null          # print numbers in sequence
real    0m0.081s                         # real == user time
user    0m0.081s                         # 100% cpu utilization
sys     0m0.000s                         # 1 thread, few syscalls

> time du ~ > /dev/null                  # check disk usage of home dir
real    0m2.012s                         # real >= user + sys
user    0m0.292s                         # 50% CPU utilization, lots of syscalls for I/O
sys     0m0.691s                         # I/O bound: blocking on hardware stalls

> time ping -c 3 google.com > /dev/null  # contact google.com 3 times
real    0m2.063s                         # real >>= user+sys time
user    0m0.003s                         # low cpu utilization
sys     0m0.007s                         # lots of blocking on network

> time make > /dev/null                  # make with 1 thread
real    0m0.453s                         # real == user+sys time
user    0m0.364s                         # ~100% cpu utilization
sys     0m0.089s                         # syscalls for I/O but not I/O bound

> time make -j 4 > /dev/null             # make with 4 "jobs" (threads/processes)
real    0m0.176s                         # real <= user+sys
user    0m0.499s                         # syscalls for I/O and coordination
sys     0m0.111s                         # parallel execution gives SPEEDUP!
```

Exercise: Speedup on Picalc via Mutex

Using a mutex fixes the approximation but breaks speedup

```
> gcc -Wall picalc_serial.c
> time a.out 100000000 > /dev/null         # SERIAL version
real    0m1.553s                           # 1.55 s wall time
user    0m1.550s
sys     0m0.000s
> gcc -Wall picalc_pthreads_mutex_contention.c -lpthread
> time a.out 100000000 1 > /dev/null       # PARALLEL 1 thread
real    0m2.442s                           # 2.44s wall time ?
user    0m2.439s
sys     0m0.000s
> time a.out 100000000 2 > /dev/null       # PARALLEL 2 threads
real    0m7.948s                           # 7.95s wall time??
user    0m12.640s
sys     0m3.184s
> time a.out 100000000 4 > /dev/null      # PARALLEL 4 threads
real    0m9.780s                          # 9.78s wall time???
user    0m18.593s                         # wait, something is
sys     0m18.357s                         # terribly wrong...
```

How do we get both accuracy AND speedup?

Answers: Local count then merge

- Contention for locks creates tremendous overhead
- Classic divide/conquer or map/reduce or split/join paradigm works here
- Each thread counts its own local hits, combine **only** at the end with single lock/unlock

```
void *compute_pi(void *arg){
  long thread_id = (long) arg;
  int my_hits = 0;                             // private count for this thread
  unsigned int rstate = 123456789 * thread_id;
  for (int i = 0; i < points_per_thread; i++) {
    double x = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
    double y = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
    if (x*x + y*y <= 1.0){
      my_hits++;                              // update local
    }
  }
  pthread_mutex_lock(&lock);                 // lock global variable
  total_hits += my_hits;                     // update global hits
  pthread_mutex_unlock(&lock);               // unlock global variable
  return NULL;
}
```

Speedup!

- This problem is almost **embarassingly parallel**: very little communication/coordination required
- Solid speedup gained but note that the user time increases as # threads increases due to overhead

```
# 8-processor desktop
> gcc -Wall pthreads_picalc_mutex_nocontention.c -lpthread
> time a.out 100000000 1 > /dev/null   # 1 thread
real    0m1.523s                       # 1.52s, similar to serial
user    0m1.520s
sys     0m0.000s
> time a.out 100000000 2 > /dev/null   # 2 threads
real    0m0.797s                       # 0.80s, about 50% time
user    0m1.584s
sys     0m0.000s
> time a.out 100000000 4 > /dev/null   # 4 threads
real    0m0.412s                       # 0.41s, about 25% time
user    0m1.628s
sys     0m0.003s
> time a.out 100000000 8 > /dev/null   # 8 threads
real    0m0.238s                       # 0.24, about 12.5% time
user    0m1.823s
sys     0m0.003s
```

Exercise: A Viable Alternative?

Discuss correctness and likely performance of this version

```
// picalc_pthreads_falseshare.c
#define MAX_THREADS 32
int thread_hits[MAX_THREADS];   // counts of hits for each thread
...
void *compute_pi(void *arg){
  long thread_id = (long) arg;
  ...
    if (x*x + y*y <= 1.0){
      thread_hits[thread_id]++; // update this thread's hit count
    }
...
}

int main(int argc, char **argv) {
  ...
  for(int p=0; p<num_threads; p++){
    pthread_join(threads[p], (void **) NULL);
  }
  int total_hits=0;                 // sum up hits over all
  for(int i=0; i<num_threads; i++){
    total_hits += thread_hits[i];
  }
```

Answers: A Viable Alternative?

- Correctness is fine for `picalc_pthreads_falseshare.c`
- Lacking speedup due to **false sharing**
- Array `thread_hits[]` is all on the same cache line
- Causes each thread to invalidate the cache on other processors slowing things down

```
>> gcc picalc_pthreads_falseshare.c -lpthread
>> time a.out 100000000 4
npoints: 100000000
hits:    78541395
pi_est:  3.141656

real    0m0.925s
user    0m3.292s
sys     0m0.001s
```

```
|------------------+-------+-------+-------|
| samples          |   75M |   75M |   75M |
| threads          |     1 |     2 |     4 |
|------------------+-------+-------+-------|
| serial           | 1.023 |     - |     - |
| mutex_fast       | 1.032 | 0.521 | 0.268 |
| mutex_contention | 1.614 | 3.790 | 3.920 |
| falseshare       | 1.044 | 0.764 | 0.723 |
|------------------+-------+-------+-------|
```

Atomic Types

- Lock / Update / Unlock pattern observed for a long time
- Works great but somewhat tedious, requires OS calls
- The C11 (2011) standard introduced **atomic** types into C at the language level so OS calls can be avoided
- Supported by many compilers including GCC now

```
// picaclc_pthreads_atomic_contention.c
#include <stdatomic.h>     // provides some atomic types
atomic_int total_hits=0;   // synced across procs / threads

void *compute_pi(void *arg){
  ...
    if (x*x + y*y <= 1.0){
      total_hits++;        // update okay but creates contention
    }
}
```

- Aside from `atomic_int`, various other pre-defined types like `atomic_char` and `atomic_size_t`
- Also `_Atomic` qualifier for user-defined types

Implementation of Atomics in GCC

Assembly code from `picalc_pthreads_atomic_contention.c`

```
compute_pi:
        ...
        lock addl $1, total_hits(%rip)
        ...
```

- `addl` adds source to destination
- `total_hits(%rip)` is RIP-relative location of global
- `lock` is an instruction prefix which locks the memory bus
  - Ensures proc has exclusive access to cache location of var
  - Invalidates other proc caches with the var

New Syntax, Same Tactics

samples	75M	75M	75M	
threads	1	2	4	
serial	1.023	-	-	
falseshare	1.044	0.764	0.723	
mutex_contention	1.614	3.790	3.920	every time
mutex_fast	1.032	0.521	0.268	end only
atomic_contention	1.102	2.212	2.290	every time
atomic_fast	1.025	0.519	0.267	end only

- Atomic updates cause Bus contention, degrade performance
- Doing them less frequently leads to better performance
- Follow the same pattern as for mutexes:
  - Update locals as much as possible
  - Update global at the end of local computations

Exercise: Array Sum via PThreads

```
// Sums the given array of integers 'array' with length
// 'len'. Launches specified number of threads to parallelize the
// process. Returns the array sum as its return value.
long arraysum_pthreads(int *array, int len, int nthreads);
```

Questions

1. Discuss overall strategy to get parallelism using threads
2. Note difficulties balancing work or ensuring correctness (ensure all array elements counted)
3. Give specific tactics about how threads will know what portion of the work to do.
4. Discuss C programming language constructs required to make the whole thing work. Avoid global variables.

Answers: Array Sum via PThreads

See `arraysum_pthread.c`

1. Discuss overall strategy to get parallelism using threads
   Have each thread sum a portion of the array. Store thread sums someplace, have master thread sum these.

2. Note difficulties balancing work or ensuring correctness (ensure all array elements counted)
   Balance work by splitting array evenly: 4 threads, each gets ~25% block of array, have last thread deal with ending elements.

3. Give specific tactics about how threads will know what portion of the work to do.
   Will need to communicate array location (not a global), length, total threads, logical thread ID to each thread. Need a place for each thread to communicate back its results.

4. Discuss C programming language constructs required to make the whole thing work.
   Define a struct with fields for arguments and local sum for thread. `arraysum_pthreads()` allocates an array of such structs, launches threads with appropriate struct data. Threads run a "worker" function which sums data and stores in its struct data.

Lessons from `arraysum_pthreads()`

- Significant tedium / boilerplate code involved
  - Requires a struct for thread arguments
  - Requires an additional “worker” function
  - Master thread launches workers in a loop, waits for completion, accumulates results
- Same basic pattern would be present for several variants
  - Other reductions like `min / max / product`
  - `arrayadd(a[], b[])` or `dotproduct(a[], b[])`
- Same ideas would be at play but magnified in more complex settings like matrix-vector multiply, matrix-matrix multiply

OpenMP provides a higher-level, more ergonomic means of executing this pattern through parallel **directives** - next topic of study.

Exercise: Heat Problem in PThreads

```
// Simulate the temperature changes for internal cells
for(t=0; t<max_time-1; t++){
  for(p=1; p<width-1; p++){
    double left_diff  = H[t][p] - H[t][p-1];
    double right_diff = H[t][p] - H[t][p+1];
    double delta = -k*( left_diff + right_diff );
    H[t+1][p] = H[t][p] + delta;
  }
}
```

Questions

1. Discuss parallelization with PThreads, high-level strategy
2. Is the strategy very different from the `array_sum()` setting?
3. What sources of parallel overhead do you see here?

Answers: Heat Problem in PThreads

1. Discuss parallelization with PThreads, high-level strategy

   Due to data dependence, parallelize the inner loop with each processor/thread handling a portion of a row at iteration t.

2. Is the strategy very different from the `array_sum()` setting?

   No: one would start P threads at each outer loop iteration to split up the inner loop iterations. This will require passing worker threads similar parameters likely via a struct and construction of a "worker" function for those threads to use.

3. What sources of parallel overhead do you see here?

   *Each iteration threads must be created and destroyed which will induce overhead. With more work, one could implement a version which starts P threads once. This requires synchronizing them across outer loop iterations likely via a **barrier** call of some type.*

PThread Barriers

```
pthread_barrier_t barrier;
// data type used to manage barriers

int pthread_barrier_wait(pthread_barrier_t *barrier);
// Blocks calling thread until a specified number of other threads
// wait on barrier. All threads proceed once count is reached.

int pthread_barrier_init(pthread_barrier_t *barrier,
                         pthread_barrierattr_t *attr,
                         unsigned count);
// Initialize data associated with barrier. Parameter `count` is the
// number of threads which must wait before all proceed.

int pthread_barrier_destroy(pthread_barrier_t *barrier);
// De-allocate barrier data
```

- Construct that allows bulk synchronization between threads
- Can ensure all threads reach a certain point before proceeding
- In Heat calculation, can be used to ensure that threads are in sync across outer loop iterations

Barrier use for in PThreads Heat

```
void *heat_worker(void *arg){
  workdata_t *wd = (workdata_t *) arg;
  ...;
  for(t=0; t<max_time-1; t++){
    for(p=mystart; p<mystop; p++){
      double left_diff  = H[t][p] - H[t][p-1];
      double right_diff = H[t][p] - H[t][p+1];
      double delta = -k*( left_diff + right_diff );
      H[t+1][p] = H[t][p] + delta;
    }
    pthread_barrier_wait(wd->barrier); // ensure all threads complete
  }                                    // row before proceeding
}

void heat_pthreads(...){
  pthread_barrier_t barrier;         // initialize barrier
  pthread_barrier_init (&barrier, NULL, nthreads);
  ...;
  for(int i=0; i<nthreads; i++){
    ...;
    workdata[i].barrier = &barrier;  // threads get reference to barrier
    pthread_create(&threads[i],NULL, heat_worker, &workdata[i]);
  }
  ...;                               // join all threads, perform reduction
  pthread_barrier_destroy(&barrier); // destroy barrier
  ...;
}
```

Additional Synchronization in PThreads Library

Condition Variables

- Wait/notification queue capable of blocking and waking up threads
- Can be used to implement Barriers but allow finer-grained control
- Always used with a Mutex and some state variables which give "conditions" of interest

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Read/Write Locks

- Distinguishes between readers and writers of data
- Allows multiple readers to lock but writer blocks until readers release
- When #readers > #writers, allows greater concurrency

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

(Optional) Thread Pools

- For some parallel applications, tasks arise over time rather than all at once
- A worker **thread pool** is often a mechanism to allow parallel execution for this
  1. At startup, some number of worker threads are started (the pool)
  2. Often #threads == #cores
  3. When work is identified, it is placed in a queue
  4. Threads pick up tasks from the queue, execute them, then look for more work
  5. When no tasks are available, threads idle
- **Advantage**: Avoid thread startup/shutdown overhead
- **Consideration**: Building a thread pool can be tricky, look for an existing library
- **Disadvantage**: Must learn the API of the pool or build your own (tricky and involves use of condition variables for efficient idling)