CSCI216: Threads in a Nutshell

Chris Kauffman

Last Updated:
Wed Dec 4 04:28:34 PM EST 2024

Logistics

Reading Bryant/O'Hallaron

Ch	Read?	Topic
Ch 12		**Concurrent Programming**
12.1	opt	Conc Progr. w/ Processes
12.2	opt	Conc Progr. w/ I/O Multiplexing
12.3	READ	Conc Progr. w/ Threads
12.4	READ	Shared Vars in Threaded Programs
12.5	READ	Synchronizing Threads w/ Semaphores
12.6	READ	Using Threads for Parallelism
12.7	opt	Other Concurrency Issues

- B&H use **Semaphores** in text to coordinate threads in Ch 12.5
- We will use **Mutexes** instead
- Will explain the minor difference

Assignments

Last Lab / HW for semester

- Lab12: Threads/Matrix Opt
- HW12: mmap() / pmap
- P5 Up, Due Mon 09-Dec

Questions on anything?

Date	Event
Mon 02-Dec	Dis: Lab12 Threads
	Lab11/HW11 Due
Tue 03-Dec	**Threads Wrap**
Wed 04-Dec	Dis: Lab12 Threads
Thu 05-Dec	Lec: Practice Exam
Mon 09-Dec	Dis: Review
	Lab12 / HW12 Due
	P5 Due
	Course Evals Due
Thu 12-Dec	Final Exam
6:30-8:30pm	Lec 1xx: IRB 0324
	Lec 2xx: ESJ 0202

Announcements: Student Feedback Opportunities

Course Experiences Now Open

e.g. Rate your Professor

- https://www.courseexp.umd.edu/
- **If response rate reaches 80% for all sections...**
- **by Mon 09-Dec 11:59pm...**
- **I will reveal a Final Exam Question**
- No answers but public discussion welcome
- Feedback open through Tue 10-Dec

Canvas Exit Survey

- Now open on ELMS/Canvas
- https://umd.instructure.com/courses/1368155/quizzes/1722546
- Worth 1 Full Engagement Point for completion
- Due prior to Final Exam (Wed 11-Dec 11:59pm)

Threads of Control within the Same Process

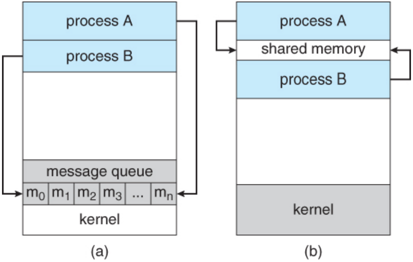
- Multiple threads execute different parts of the same code for the program concurrently
  - Concurrent: simultaneous or in an unspecified order
  - Parallel: simultaneous
- Threads each have their own "private" function call stack
- CAN share stack values by passing pointers to them around
- Share the heap and global area of memory
- In Unix, **Posix Threads (pthreads)** is the most widely available thread library

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()`

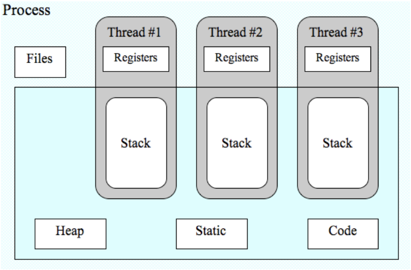
Modern systems (Linux) can use semaphores / mutexes / shared memory / message queues / condition variables to coordinate Processes or Threads

IPC Memory Model



Source

Thread Memory Model



Source

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

Code

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

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

int main(){
  pthread_t thread_1;
  pthread_create(&thread_1, NULL,
                 doit, (void *) 42);
  int xres;
  pthread_join(thread_1, (void **) &xres);
  printf("result is: %d\n",xres);
  return 0;
}
```

Compilation

- Link thread library
  `-lpthread`
- Lots of warnings

```
> gcc pthreads_minimal_example.c -lpthread
pthreads_minimal_example.c: In function 'doit':
pthreads_minimal_example.c:7:9: warning:
 cast from pointer to integer of different
 size [-Wpointer-to-int-cast]
   int p=(int) param;
         ^
pthreads_minimal_example.c:9:10: warning:
 cast to pointer from integer of different
 size [-Wint-to-pointer-cast]
   return (void *) p;
          ^
> a.out
result is: 84
```

Observations About Pthreads

1. Child thread starts running code in the function passed to `pthread_create()`, function `doit()` in example
2. Main Thread continues immediately, much like `fork()` but child runs the given function while parent continues as is
3. Compilers provide Little syntax support for threads: must do a lot of casting of arguments/returns
4. Thread Entry Functions can take a single pointer argument; passing multiple arguments is usually done via a `struct`
5. Can't say in which order Main/Children threads will execute; identical to `fork()`'d processes

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; Why?

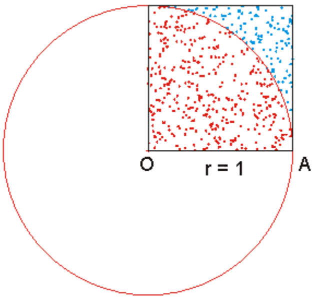
(2) Hide Latency of Slow Tasks via Threads

- Slow tasks block a thread but Fast tasks can proceed independently allowing program to stay busy while running
- Textbook coverage (I/O latency reduction)
- 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

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



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

Exercise: picalc_pthreads_broken.c

Serial Version (Single Thread)

- picalc_serial.c codes Monte Carlo approximation for Pi
- Uses rand_r() to generate pseudo-random numbers
- picalc_rand.c uses traditional rand(), discuss more later

Parallel Version (Multiple Threads)

Examine source code for pthreads_picalc_broken.c
Discuss following questions with a neighbor

1. How many threads are created? Fixed or variable?
2. How do the threads cooperate? Is there shared information?
3. Do the threads use the same or different random number sequences?
4. Will this code actually produce good estimates of π?

Exercise: pthreads_picalc_broken.c

```c
long total_hits = 0;  long points_per_thread = -1;

void *compute_pi(void *arg){
  long thread_id = (long) arg;
  unsigned int rstate = 123456789 * thread_id;   // unique seed per thread
  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){
      total_hits++;
    }
  }
  return NULL;
}
int main(int argc, char **argv) {
  long npoints = atol(argv[1]);                      // number of samples
  int num_threads = argc>2 ? atoi(argv[2]) : 4;  // number of threads
  points_per_thread = npoints / num_threads;      // init global variables
  pthread_t threads[num_threads];                    // track each thread
  for(long p=0; p<num_threads; p++){                // launch each thread
    pthread_create(&threads[p],NULL,compute_pi, (void *) (p+1));
  }
  for(int p=0; p<num_threads; p++){                 // wait for each thread to finish
    pthread_join(threads[p], (void **) NULL);
  }
  double pi_est = ((double)total_hits) / npoints * 4.0;
  printf("npoints: %8ld\n",npoints);
  printf("hits:    %8ld\n",total_hits);
  printf("pi_est:  %f\n",pi_est);
  return 0;
}
```

Answers: `pthreads_picalc_broken.c`

1. How many threads are created? Fixed or variable?
   - Threads specified on command line
2. How do the threads cooperate? Is there shared information?
   - Shared global variable `total_hits`
3. Do the threads use the same or different random number sequences?
   - Different, seed is based on thread number
4. Will this code actually produce good estimates of π?
   - Nope: not coordinating updates to `total_hits` so will likely be wrong

```
> gcc -Wall pthreads_picalc_broken.c -lpthread
> a.out 10000000 4
npoints: 10000000
hits:     3134064
pi_est:  1.253626   # not a good estimate for 3.14159
```

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`[1]

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			

[1]CSAPP Ch 12.5 discusses similar code for another example

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);
  ...
}
```

Exercise: Protect critical region of `picalc`

- Insert calls to `pthread_mutex_lock() / _unlock()`
- Protect the critical region and Predict effects on execution

```
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){
      total_hits++;                                  // update
    }
  }
  return NULL;
}
```

Answers: Protect critical region of `picalc`

- Naive approach

```
if (x*x + y*y <= 1.0){
  pthread_mutex_lock(&lock);    // lock global variable
  total_hits++;                 // update
  pthread_mutex_unlock(&lock);  // unlock global variable
}
```

- Ensures correct answers but...
- Severe effects on performance (next slide)

Speedup?

- Multiple threads should decrease wall (real) time and give **Speedup**:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

- Ideally want **linear speedup**: 2X speedup for 2 Threads, etc.

```
> gcc -Wall picalc_serial.c -lpthread
> time a.out 100000000 > /dev/null          # SERIAL version
real    0m1.553s                             # 1.55 s wall time
user    0m1.550s
sys     0m0.000s
> gcc -Wall pthreads_picalc_mutex.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...
```

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!
```

Avoiding Mutex Contention for Efficiency

- Locking/Unlocking Mutexes is a **system call**, takes time for the OS to coordinate threads
- Avoiding repeated lock/unlock cycles saves time
- Often necessitates **private data per thread** to contention
- In this case, private data is just a single integer but it may be more complex in other settings (e.g. whole vector, matrix, data structure, etc.)

```
// picalc_pthreads_mutex.c
// LOTS of lock contention: slow down
  for (int i=0; i<points_per_thread; i++) {
    double x = ...;
    double y = ...;
    if (x*x + y*y <= 1.0){
      pthread_mutex_lock(&lock);
      total_hits++;
      pthread_mutex_unlock(&lock);
    }
  }
```

```
// picalc_pthreads_mutex_nocontention.c
// LITTLE lock contention: speedup
  int my_hits = 0;  // private per thread
  for (int i=0; i<points_per_thread; i++) {
    double x = ...;
    if (x*x + y*y <= 1.0){
      my_hits++;
    }
  }
  pthread_mutex_lock(&lock);
  total_hits += my_hits;
  pthread_mutex_unlock(&lock);
```

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 picalc_pthreads_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
```

Alternative Approach: Lock Free

As an alternative, can completely avoid the global variable / lock by having working threads return private sums which are received by `main()` and totaled in it, a more *functional* approach

```
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
    }
  }
  return (void *) my_hits;
}
int main(){
  ...
  int total_hits = 0;
  for(int p=0; p<nthreads; p++){
    int hits;
    pthread_join(threads[p], (void **) &hits);
    total_hits += hits;
  }
}
```

rand() vs rand_r() Function Usage

Consider left/right examples below

- Very similar except use of rand_r() vs rand() functions
- Note the usage differences, rand_r() has state in its parameter, rand() uses hidden global variable for its state

```c
// picalc_pthreads_mutex_nocontention.c:
int main(){
  ...;
  pthread_create(...,compute_pi,i+1);
  ...;
}

void *compute_pi(void *arg){
  long thread_id = (long) arg;
  unsigned int rstate =
    123456789 * thread_id;
  int my_hits = 0;
  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++;
    }
  }
  ...
```

```c
// picalc_pthreads_rand.c:
int main(){
  ...;
  srand(123456789);  // seed generator
  ...;
}

void *compute_pi(void *arg){
  // rand() uses a hidden global variable
  // for the state of the random number
  // generator
  int my_hits = 0;
  for (int i = 0; i < points_per_thread; i++){
    double x = ((double) rand())
               / ((double) RAND_MAX);
    double y = ((double) rand())
               / ((double) RAND_MAX);
    if (x*x + y*y <= 1.0){
      my_hits++;
    }
  }
```

Exercise: rand() vs rand_r() Function Performance

Which of these to seems to scale better with the number of threads? Why do you think the slower suffers?

```
val>> gcc -o p_rand_r picalc_pthreads_rand_r.c

val>> time ./p_rand_r 1000000 1
npoints:  1000000
hits:      785235
pi_est:  3.140940
real    0m0.060s
user    0m0.054s
sys     0m0.004s

val>> time ./p_rand_r 1000000 2
npoints:  1000000
hits:      784938
pi_est:  3.139752
real    0m0.038s
user    0m0.061s
sys     0m0.004s

val>> time ./p_rand_r 1000000 4
npoints:  1000000
hits:      785398
pi_est:  3.141592
real    0m0.023s
user    0m0.061s
sys     0m0.004s
```

```
val>> gcc -o p_rand picalc_pthreads_rand.c

val>> time ./p_rand 1000000 1
npoints:  1000000
hits:      785229
pi_est:  3.140916
real    0m0.136s
user    0m0.133s
sys     0m0.001s

val>> time ./p_rand 1000000 2
npoints:  1000000
hits:      784982
pi_est:  3.139928
real    0m1.018s
user    0m1.166s
sys     0m0.855s

val>> time ./p_rand 1000000 4
npoints:  1000000
hits:      785589
pi_est:  3.142356
real    0m0.522s
user    0m0.970s
sys     0m0.954s
```

Answers: `rand()` vs `rand_r()` Function Performance

- `rand_r()` is faster out of the gate and runs faster with more threads
- `rand()` runs slower for 1 thread, slows down significantly at 2 threads, still slow at 4 threads
- `rand()` must protect the global variable representing the random number state with **mutual exclusion**: each call to `rand()` likely involves some sort lock/compute/unlock
- This slows things down for the `rand()` version
- `rand_r()` puts the random number generation state in each thread so no coordination is needed: **unshared data leads to speed**

```c
// GLIBC rand.c
int rand (void) {
  return (int) __random ();
}
```

```c
// GLIBC random.c
static struct random_data unsafe_state = {...}

long int __random (void) {
  int32_t retval;
  __libc_lock_lock (lock);
  (void) __random_r (&unsafe_state, &retval);
  __libc_lock_unlock (lock);
  return retval;
}
```

Meaning of Thread Safety

Thread safety is achieved in one of two ways

1. Use local data only: no shared data
2. Protect shared data with mutex locking/unlocking around critical regions

Historically many Unix library functions were not thread-safe

- `malloc() / free()` operated on the heap, a shared data structure; not initially thread-safe but modern incarnations are using combinations of (hidden) local data and mutexs
- `rand()` function was historically NOT thread-safe
  - used a global variable as the state of the random number generator
  - multiple threads calling it would corrupt the state leading too... random numbers (unpredictable random numbers)
  - `rand_r()` was introduced to fix this, use local state
  - Most `rand()` implementations are now thread-safe and `rand_r()` has been deprecated: will be eventually removed
  - Switch to the `jrand48()` function for similar functionality to `rand_r()`

Thread-Safe Functions Documentation

Manual pages for library functions often describe whether they are safe for multiple threads to use or not

```
MALLOC(3)                  Library Functions Manual                  MALLOC(3)

NAME
       malloc, free, calloc, realloc, reallocarray - allocate and free dynamic
       memory
...
ATTRIBUTES
       |---------------------------------------+--------------+---------|
       | Interface                             | Attribute    | Value   |
       |---------------------------------------+--------------+---------|
       | malloc(), free(), calloc(), realloc() | Thread safety | MT-Safe |
       |---------------------------------------+--------------+---------|

==============================================================================
CRYPT(3)                   Library Functions Manual                   CRYPT(3)

NAME
       crypt, crypt_r, crypt_rn, crypt_ra - passphrase hashing
...
       char * crypt(  const char *phrase, const char *setting);
       char * crypt_r(const char *phrase, const char *setting,
                      struct crypt_data *data);

ATTRIBUTES
       |-----------------------------+---------------+----------------|
       | Interface                   | Attribute     | Value          |
       |-----------------------------+---------------+----------------|
       | crypt                       | Thread safety | MT-UnSafe race |
       |-----------------------------+---------------+----------------|
       | crypt_r, crypt_rn, crypt_ra | Thread safety | MT-Safe        |
       |-----------------------------+---------------+----------------|
```

Reentrant Functions

A related concept to Thread Safe functions are **Reentrant Functions**

> *. . . reentrant if it can be interrupted in the middle of its execution, and then be safely called again ("re-entered") before its previous invocations complete execution.*
> – *Wikipedia: Reentrancy*

General hierearchy is:

Quality	Probable Causes
Thread Unsafe	Uses shared data without coordination
Thread Safe	Uses shared data (e.g. mutex locking), not necessarily reentrant
Reentrant	Uses local data, Thread-safe by default

Reentrant functions are important as one would write **signal handlers** as handlers can be interrupted and lead to re-entering a function

Thread IDs: OS-Level vs Logical

OS Thread ID Functions

Thread ID functions exist on most UNIX platforms but. . .

```
// treat thread as a big integer
unsigned long = pthread_self();

// Linux only
pid_t tid = gettid(); // system call
printf("Thread %d reporting for duty\n",
       tid);

// Non-portable, non-linux
pthread_id_np_t tid =
   pthread_getthreadid_np();
```

NONE of the above are likely give thread ids numbered 0,1,2,3. . . on all systems and should not be used when such logic is desired

Logical Thread IDs

When logical IDs (0,1,2,..) are required, can be created simply and passed via “context” data

```
// pthread_sum_array.c:
typedef struct {
  int threadid;
  ...
} work_context_t;

void *worker_func(void *arg){
  work_context_t *ctx =
     (work_context *) arg ;
  int my_id = ctx->threadid;
  ...;
}

int main(){
  ...;
  work_context_t ctxs[4]={};
  for(int i=0; i<4; i++){
    ctxs[i].thread_id = i;
    pthread_create(&threads[i],NULL
                   worker_func, &ctxs[i]);
  }
  ...;
}
```

Examine: `pthread_sum_array.c`

- Common thread code patterns demonstrated there
- To make threaded functions more general **avoid use of global variables**
- Commonly requires passing pointers to a `struct` as the argument to worker threads; Kauffman uses the term "context" for this struct but that is not in wide use
- The `struct` usually carries essential information into a worker thread function:
  - Thread's ID and total # threads
  - Pointers to data on which to operate
  - Pointers to any data needed to coordinate (e.g. Mutexes)
- Context struct provides all that's needed for threads to do their share of work
- Avoids the need to use a global variable: code is more self-contained
- **Use this idea in Project 5 to set up coordination**

Mutex vs Semaphore

Similarities

- Both used to protect critical regions of code from other processes/threads
- Both use non-busy waiting
  - process/thread blocks if locked by another
  - unlocking wakes up a blocked process/thread
- Both can be process private or shared between processes
  - Shared mutex requires shared memory
  - Private semaphore with option `pshared==0`

Differences

- Semaphores loosely associated to Process coordination
- Mutexes loosely associated to to Thread coordination
- Both can be used for either with correct setup
- Semaphores posses an arbitrary **natural number**, usually `0` for locked, `1,2,3`,.. for available
- Mutexes are either locked/unlocked
- Mutexes have a busy locking variant: `pthread_spinlock_t`

Semaphore Terminology and History

- "Semaphore" generally some sort of signaling mechanism to control a shared resource, usage in computing originated from Railway Semaphores used to control Single Train Tracks to avoid collisions
- Use in computing attributed to Edsger Dijkstra, slightly more general than typical Mutex lock, slightly different terminology

	Acquire	Release
Mutex	`lock()`	`unlock()`
Semaphore	`wait()`	`post() / signal()`

- Technically `post()` will increment the semaphore value but often they are used just as 0 "locked" and 1 "unlocked"
- There are two major UNIX versions of Semaphores
  - POSIX Semaphores which are newer, widely available, have a relatively clean design, should be used in new code
  - System V IPC Semaphores which are old, a bit nutty, and should be avoided in new code if at all possible

Mutex Gotchas

- Managing multiple mutex locks is tricky: wrong protocol may result in **deadlock**, threads waiting for each other to release locks
- Same thread locking same mutex twice can cause deadlock depending on options associated with mutex
- Interactions between threads with different scheduling priority are also tough to understand and the source of trouble
- Notable Mutex problem in the Mars Pathfinder Onboard Computer
  - Used multiple threads with differing priorities to manage limited hardware
  - Shortly after landing, started rebooting like crazy due to odd thread interactions
  - Short-lived, low-priority thread got a mutex, pre-empted by long-running medium priority thread, system freaked out because others could not use resource associated with mutex
  - Search for articles on "Thread Priority Inversion" problems which is the class of problems that nearly derailed the mission

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
- `pthread_barrier_demo.c`: shows basic purpose of barriers

Exercise: Scaling an Array

- Adapt the approach of the earlier sum example to **scale** elements of an array by dividing each element by the sum
- Use a `pthread_barrier_t` with `pthread_barrier_wait()` to coordinate parts of the computation

```c
void *workfunc(void *arg){
  ...;
  double my_sum = 0.0;
  for(long i=start; i<stop; i++){
    my_sum += ctx.array[i];
  }

  pthread_mutex_lock(ctx.lock);
  *ctx.total_sum += my_sum;
  pthread_mutex_unlock(ctx.lock);

  // ADD COORDINATION / SCALING HERE

  return NULL;
}
```

```c
// MODIFY TO INCLUDE BARRIER DATA
int main() {
  ...;
  pthread_mutex_t lock;
  pthread_mutex_init(&lock,NULL);

  pthread_t threads[num_threads];
  work_context_t context[num_threads];

  for(int i=0; i<num_threads; i++){
    ...;
    context[i].lock = &lock;

    pthread_create(&threads[i],NULL,
                   workfunc, &context[i]);
  }
  ...;
}
```

Answers: Scaling an Array

See `pthread_scale_array.c` for full solution

```c
void *workfunc(void *arg){
  ...;
  double my_sum = 0.0;
  for(long i=start; i<stop; i++){
    my_sum += ctx.array[i];
  }

  pthread_mutex_lock(ctx.lock);
  *ctx.total_sum += my_sum;
  pthread_mutex_unlock(ctx.lock);

  // ADD COORDINATION / SCALING HERE
  pthread_barrier_wait(ctx.barrier);
  my_sum = *ctx.total_sum;
  for(long i=start; i<stop; i++){
    ctx.array[i] /= my_sum;
  }

  return NULL;
}
```

```c
// MODIFY TO INCLUDE BARRIER DATA
int main() {
  ...;
  pthread_mutex_t lock;
  pthread_mutex_init(&lock,NULL);
  pthread_barrier_t barrier;
  pthread_barrier_init(&barrier,NULL,
                       num_threads);

  pthread_t threads[num_threads];
  work_context_t context[num_threads];

  for(int i=0; i<num_threads; i++){
    ...;
    context[i].lock = &lock;
    context[i].barrier = &barrier;

    pthread_create(&threads[i],NULL,
                   workfunc, &context[i]);
  }
  ...;
}
```

==== END FALL 2024 CONTENT =====

Remaining content is optional but informative

(Optional) Exercise: Mutex Busy wait or not?

- Consider given program
- Threads acquire a mutex, sleep 1s, release
- **Predict** user and real/wall times if
  1. Mutex uses busy waiting (polling)
  2. Mutex uses interrupt driven waiting (sleep/wakup when ready)
- Can verify by compiling and running

  `time a.out`

```c
// Busy?
int glob = 1;
pthread_mutex_t glob_lock;

void *doit(void *param){
  pthread_mutex_lock(&glob_lock);
  glob = glob*2;
  sleep(1);
  pthread_mutex_unlock(&glob_lock);
  return NULL;
}

int main(){
  printf("BEFORE glob: %d\n",glob);

  pthread_mutex_init(&glob_lock, NULL);
  pthread_t thread_1;
  pthread_create(&thread_1, NULL, doit, NULL);
  pthread_t thread_2;
  pthread_create(&thread_2, NULL, doit, NULL);

  pthread_join(thread_1, (void **) NULL);
  pthread_join(thread_2, (void **) NULL);

  printf("AFTER glob: %d\n",glob);
  pthread_mutex_destroy(&glob_lock);

  return 0;
}
```

Answers: Mutex Busy wait or not? NOT

- Locking is **Not** a busy wait
- Either get the lock and proceed OR
- Block and get woken up when the lock is available
- Timing is
  - real: 2.000s
  - user: 0.001s
- Contrast with `time_spinlock.c`:
  - real: 2.000s
  - user: 1.001s
- `pthread_spinlock_*` like mutex but wait "busily": faster access for more CPU

```c
// time_mutex_.c: Not busy, blocked!
int glob = 1;
pthread_mutex_t glob_lock;

void *doit(void *param){
  pthread_mutex_lock(&glob_lock);
  glob = glob*2;
  sleep(1);
  pthread_mutex_unlock(&glob_lock);
  return NULL;
}

int main(){
  printf("BEFORE glob: %d\n",glob);

  pthread_mutex_init(&glob_lock, NULL);
  pthread_t thread_1;
  pthread_create(&thread_1, NULL, doit, NULL);
  pthread_t thread_2;
  pthread_create(&thread_2, NULL, doit, NULL);

  pthread_join(thread_1, (void **) NULL);
  pthread_join(thread_2, (void **) NULL);

  printf("AFTER glob: %d\n",glob);
  pthread_mutex_destroy(&glob_lock);

  return 0;
}
```

Mixing Processes and Threads

- You can mix IPC and Threads if you hate yourself enough.

  Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated.
  – Stevens/Rago Ch 12.8[2]

- Strongly suggest you examine Stevens and Rago 12.8-12.10 to find out the following **pitfalls**:
- Threads have individual Signal Masks (for blocking) but share Signal Disposition (for handling funcs/termination)
- Calling `fork()` from a thread creates a new process with all the locks/mutexes of the parent but only one thread (!?)
  - Usually implement a `pthread_atfork()` handler for this
- Multiple threads should use `pread() / pwrite()` to read/write from specific offsets; ensure that they do not step on each other's I/O calls

[2] Advanced Programming in the Unix Environment, 3rd Ed by Richard Stevens and Stephen A. Rago

Are they really so different?

- Unix standards strongly distinguish between threads and processes: different system calls, sharing, etc.
- Due to their similarities, you should be skeptical of this distinction as smart+lazy OS implementers can exploit it:
  Linux uses a 1-1 threading model, with (to the kernel) no distinction between processes and threads – everything is simply a runnable task.
  On Linux, the system call `clone()` clones a task, with a configurable level of sharing...

Unix Syscall	*Linux implementation*
`fork()`	`clone(LEAST sharing)`
`pthread_create()`	`clone(MOST sharing)`

– *Ryan Emerle, SO:"Threads vs Processes in Linux"*

The "1-1" model is widely used (Linux, BSD, Windows(?)) but conventions vary between OSs: check your implementation for details

Lightweight Threads of Various Colors

- Pthreads are (almost) guaranteed to interact with the OS
- On Linux, a Pthread is a "schedulable" entity which is automatically given time on the CPU by the scheduler
- Other kinds of threads exist with different properties with various names, notably **lightweight / green threads**

  > ***Green threads*** *are threads that are scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system (OS).*
  > – *Wikip: Green Threads*

- Lightweight/Green thread library usually means OS only sees a single process
- Process itself must manage its internal threads with its own scheduler / yield semantics
  - **Advantage**: Fast startup `:-D`
  - **Drawback**: No parallelism `:-(`

(Optional) Exercise: Processes vs Threads

Processes when. . .

Identify some obvious signs your application should you use processes vs. . .

Threads when. . .

Identify some obvious signs your application should you use threads instead

Answers: Processes vs Threads

Processes when...

- Limited amount of sharing needed, file or single block of memory
- Want ability to monitor/manage/kill distinct tasks with standard OS tools
- Plan to make use of signals in any appreciable way

Threads when...

- Tasks must share a lot of data
- Likely that won't need to individually monitor tasks
- Absolutely need fastest possible startup of subtasks

Threads Should be Chosen Cautiously

- Managing concurrency is hard
- Separate processes provide one means to do so, often a good start as defaults to nothing shared
- Performance benefits of threads come with MANY disadvantages and pitfalls
- If forced to use threads, consider design carefully
- If possible, use a higher-level thread manager like OpenMP, well-suited for parallelizing loops for worker threads
- Avoid mixing threads/IPC if possible
- Prepare for a tough slog...