Logistics

Reading

▸ Stevens/Rago
  Ch 11-12

▸ Robbins and Robbins
  Ch 12-13

Lab10: Proc to pthread port

▸ Due next Monday
▸ How did it go?

Goals

▸ Condition Variables
▸ Thread wrap-up

Date | Event
--- | ---
Tue 11/14 | Threads/Condvars
Thu 11/16 | Review (how?)
Mon 11/20 | Lab 11 (10 due)
Tue 11/21 | Exam 2
Thu 11/23 | ↓

Project 2

▸ Post prior to break
▸ Due last week of semester
Threads of Control within the Same Process

- Parallel execution path within the same process
- Multiple threads execute different parts of the same code for the program concurrently
  - Concurrent: simultaneous or in an unspecified order
- 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
### Threads vs IPC

<table>
<thead>
<tr>
<th>Process in IPC</th>
<th>Threads in pthreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Marginally) Longer startup</td>
<td>(Marginally) Faster startup</td>
</tr>
<tr>
<td>Must share memory explicitly</td>
<td>Memory shared by default</td>
</tr>
<tr>
<td>Good protection between processes</td>
<td>Little protection between threads</td>
</tr>
<tr>
<td><code>fork()</code> / <code>waitpid()</code></td>
<td><code>pthread_create()</code> / <code>_join()</code></td>
</tr>
<tr>
<td>Queues, Semaphores, Shared Mem</td>
<td>Queues, Semaphores, Mutexes, CondVars</td>
</tr>
</tbody>
</table>

### IPC Memory Model

- Process A
  - shared memory
  - process B
  - message queue
    - $m_0, m_1, m_2, m_3, \ldots, m_n$
  - kernel

- Process B
  - shared memory
  - process A

### Thread Memory Model

- Process
  - Files
  - Registers
  - Stack
  - Heap
  - Static
  - Code

- Thread #1
  - Registers
  - Stack

- Thread #2
  - Registers
  - Stack

- Thread #3
  - Registers
  - Stack
### Process and Thread Functions

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

<table>
<thead>
<tr>
<th>Processes</th>
<th>Threads</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>pthread_create()</td>
<td>create a new flow of control</td>
</tr>
<tr>
<td>waitpid()</td>
<td>pthread_join()</td>
<td>get exit status from flow of control</td>
</tr>
<tr>
<td>getpid()</td>
<td>pthread_self()</td>
<td>get &quot;ID&quot; for flow of control</td>
</tr>
<tr>
<td>exit()</td>
<td>pthread_exit()</td>
<td>exit (normally) from an existing flow of control</td>
</tr>
<tr>
<td>abort()</td>
<td>pthread_cancel()</td>
<td>request abnormal termination of flow of control</td>
</tr>
<tr>
<td>atexit()</td>
<td>pthread_cleanup_push()</td>
<td>register function to be called at exit from flow of control</td>
</tr>
</tbody>
</table>

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

// 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
   -lpthreads
▶ 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
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?
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
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 $\leq 1$ 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.
Serial Code picalc.c and picalc_rand.c

- Examine source code for picalc_rand.c
- Note basic algorithm is simple and easily parallelizable
- Discuss trouble with the rand() function: non-reentrant
- Examine source code for picalc.c
- Contrast the rand_r() function: reentrant version
Exercise: pthreads_picalc.c

http://cs.umn.edu/~kauffman/4061/pthreads_picalc.c

- Examine source code for pthreads_picalc.c
- How many threads are created? Fixed or variable?
- How do the threads cooperate? Is there shared information?
- Do the threads use the same or different random number sequences?
- Will this code actually produce good estimates of $\pi$?
Answers: pthreads_picalc.c

http://cs.umn.edu/~kauffman/4061/pthreads_picalc.c

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

```bash
> 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

```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`

<table>
<thead>
<tr>
<th>Mem #1024</th>
<th>Thread 1 Instruction</th>
<th>REG1 Value</th>
<th>Thread 2 Instruction</th>
<th>REG1 Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>total_hits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>30: load REG1</td>
<td>100</td>
<td>30: load REG1</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>31: incr REG1</td>
<td>101</td>
<td>31: incr REG1</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>32: store REG1</td>
<td></td>
<td>32: store REG1</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30: load REG1</td>
<td>102</td>
<td>30: load REG1</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>31: incr REG1</td>
<td>103</td>
<td>31: incr REG1</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>32: store REG1</td>
<td></td>
<td>32: store REG1</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30: load REG1</td>
<td>103</td>
<td>30: load REG1</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>31: incr REG1</td>
<td></td>
<td>31: incr REG1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32: store REG1</td>
<td></td>
<td>32: store REG1</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```c
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: 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
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;
}
```
Answer: 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
- If it were busy should be roughly
  - real: 2.000s
  - user: 1.001s
- `pthread_spinlock_*` like mutex locks but more likely to busily wait

```c
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;
}
```
Exercise: Protect critical region of picalc

▶ Insert calls to pthread_mutex_lock() and pthread_mutex_unlock()

▶ Protect the critical region

▶ Predict effects on code

```c
1 int total_hits = 0;
2 int points_per_thread = ...;
3 pthread_mutex_t lock; // initialized in main()
4
5 void *compute_pi(void *arg){
6    long thread_id = (long) arg;
7    unsigned int rstate = 123456789 * thread_id;
8    for (int i = 0; i < points_per_thread; i++) {
9       double x = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
10      double y = ((double) rand_r(&rstate)) / ((double) RAND_MAX);
11      if (x*x + y*y <= 1.0){
12         total_hits++; // update
13      }
14   }
15   return NULL;
16 }
```
Answer: Protect critical region of picalc

- Naive approach
  
  ```c
  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
Speedup?

- Dividing work among workers should decrease wall (real) time
- Shooting for **linear speedup**

\[
\text{Parallel Time} = \frac{\text{Serial Time}}{\text{Number of Workers}}
\]

```
> gcc -Wall picalc.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
sys   0m18.357s  # wait, something is
```
Alternative Approach: 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

```c
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

```bash
# 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
```
Mutex Gotchas

- Managing multiple mutex locks is fraught with danger
- Must choose protocol carefully: similar to discussion of Dining Philosophers with semaphores
- 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
- Robbins/Robbins 13.8 discusses some problems with the Mars Pathfinder probe resulting from threads/mutex locks
  - 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
get_thread_id()???

As noted in other answers, pthreads does not define a platform-independent way to retrieve an integral thread ID. This answer http://stackoverflow.com/a/21206357/316487 gives a non-portable way which works on many BSD-based platforms.
– Bleater on Stack Overflow

// Standard opaque object, non-printable??
pthread_t opaque = pthread_self();

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

// Linux only
pid_t tid = syscall( __NR_gettid );
printf("Thread %d reporting for duty\n",tid);
Thread ID work-arounds

- In many cases pid_t is just a unsigned long
  
  ```c
  // /usr/include/bits/pthreadtypes.h
  typedef unsigned long int pthread_t;
  ```

- Allows simple printf printing as in
  ```c
  void *doit(void *param){
      pthread_t tid = pthread_self();
      printf("doit: I am thread %ld\n",tid);
      ...;
  }
  ```

- Thread ids are often LARGE numbers

- See pthread_ids.c for full example

- Use this technique for debugging, remove for production and NOT for algorithms

- Establish own logical thread IDs if required by passing parameters to thread worker function
Exercise: Odd-Even workers

```c
int count = 0; // global variable all threads are modifying
pthread_mutex_t count_mutex; // mutex to check/alter count

// Run by TWO even child threads, increment count when it is even 5 times
void *even_work(void *t) {
    ...
}
// Run by TWO odd child threads, increment count when it is odd 5 times
void *odd_work(void *t) {
    ...
}
int main(){
    int tids[] = {0, 1, 2, 3}; pthread_t threads[4];
    pthread_create(&threads[0], NULL, even_work, &(tids[0]));
    pthread_create(&threads[1], NULL, odd_work, &(tids[1]));
    pthread_create(&threads[2], NULL, even_work, &(tids[2]));
    pthread_create(&threads[3], NULL, odd_work, &(tids[3]));
    // join threads, WANT: count = 20
}
```

- Propose code which uses a mutex to lock `count`
- Even/Odd threads update only if it is appropriate
- What kind of control structure must be used?
- What consequences does this have for performance?
Need a loop that

- Acquires a lock
- Checks count, proceeds if odd/even
- Otherwise release and try again

Results in busy waiting: can repeatedly get lock despite condition of odd/even not changing

```c
int count = 0;
pthread_mutex_t count_mutex;

void *even_work(void *t) {
    int tid = *( (int *) t);
    for(int i=0; i<THREAD_ITERS; i++){
        while(1){
            pthread_mutex_lock(&count_mutex);
            if(count % 2 != 0){ // check if even
                break; // yup: move on
            } // nope: try again
            pthread_mutex_unlock(&count_mutex);
        }
        count++; // locked and even
        pthread_mutex_unlock(&count_mutex);
    }
    return NULL;
}
```
Condition Variables

- Major limitation for locks: can only lock/unlock (duh?)
- Frequently want to check shared resource, take action only under specific conditions associated with resource
  - Queue of work is non-empty
  - Two utensils are immediately available
  - It is this thread's turn to go
- Mutex on its own is ill-suited for this problem:
  In a loop
  - Lock variables indicating condition
  - Check condition
  - Break from loop if condition is true
  - Unlock and try again if not true
- Semaphores may be of some help but they have to do with counts only
- For this, condition variables or monitors are often used
Condition Variable Operations

- Condition variables would be more appropriately named notification queue
- Always operate in conjunction with a mutex
- Threads acquire mutex, check condition, block if condition is unfavorable, get notified of changes, automatically relock mutex on wakeup

```c
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t attr);
int pthread_cond_destroy(pthread_cond_t *cond);
// Initialize and destroy
```

```c
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
// atomically release mutex and block/sleep until notified that given condition has changed
```

```c
int pthread_cond_signal(pthread_cond_t *cond);
// wake up a single thread waiting on the given condition
// woken up thread automatically locks the mutex specified
// in pthread_cond_wait()
```

```c
int pthread_cond_broadcast(pthread_cond_t *cond);
// wake up all threads waiting on the given condition
// woken up threads automatically lock the mutex specified
// when it is their "turn"
```
Odds/Evens with Condition Variables

- odds_evens_condvar.c
- Worker loop now uses `pthread_cond_wait()`
- Blocks and gets notification of changes to `count`
- Threads call `pthread_cond_broadcast()` to wake up other threads when `count` changes: no busy lock/unlock while waiting
- **Question:** Would `pthread_cond_signal()` which wakes up a single other thread work here?

```c
int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_condv;

void *even_work(void *t) {
    int tid = *( (int *) t);
    for(int i=0; i<THREAD_ITERS; i++) {
        pthread_mutex_lock(&count_mutex);
        while(count % 2 != 0) {
            pthread_cond_wait(&count_condv, &count_mutex);
        }
        count++;
        pthread_mutex_unlock(&count_mutex);
        pthread_cond_broadcast(&count_condv);
    }
    return NULL;
}
```
Bounded Buffer: Classic Model Problem

- **Producers** add items to buffer *if space available*
- **Consumers** remove from buffer *if items present*
- Lock buffer to check/alter it
- Lock-only solution involves repeated lock/discard
  
  Producer A locks, no space, unlocks
  Producer B locks, no space, unlocks
  Producer A locks, no space, unlocks
  Producer B locks, no space, unlocks
  ...

- **CondVars/Semaphore** add efficiency through sleep/signal
  
  Producer A locks, no space, sleeps
  Producer B locks, no space, sleeps
  ...
  ...
  Consumer C locks, removes, signals
  Producer A locks, adds, unlocks

- Shared, fixed sized buffer of items
- Multiple threads/processes acting on buffer
Exercise: Reentrant?

- Recall the meaning of reentrant
- Describe dangerous place to call non-reentrant functions
- What are some notable non-reentrant functions?
- Does this have play in our current discussion of threads?
Reentrant and Thread-Safe

- A variety of VERY useful functions are non-reentrant, notably `malloc()` / `free()`
- Use some global state manipulate the heap
- Dangerous to call these during a signal handler as they are not async-signal-safe
- However, many of these are thread-safe: can be called from multiple threads safely (MT-Safe for Muti-Thread Safe)
- This is good as it means multiple threads can allocate/free memory safely which would be close to crippling if not allowed
- Check manual pages for library/system calls you plan to use
- Q: Prof Kauffman: how can something be thread-safe but not re-entrant?
- A: I’ll give 5 cards to someone who can put up a good Piazza post explaining this by next class. There’s a lot of StackOverflow to read and I’ve got a project to get ready for you.
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

- Strongly suggest you examine Stevens and Rago 12.8-12.10 to find out the following pitfalls:

  - Threads have individual signal masks but share signal disposition (!?)
  - 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
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… fork() calls clone(least sharing) and pthread_create() calls 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
End Message: Threads are not a first choice

- 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...