CSC2/452 Computer Organization
Threading and Parallelism

Sreepathi Pai

URCS

November 18, 2019

Outline

Outline

Administrivia

- Assignment #4 is out
  - Due date: Tuesday, Nov 26, 7PM
- Assignment #5 (last!) will be out Dec 2
  - Due on Dec 11

Outline

Process-based Concurrency

- Create multiple processes to handle multiple "work items" (or requests)
  - `fork` is cheap
  - Resources (CPU, Memory) managed by OS
- Use OS-based interprocess communication (IPC) mechanisms to communicate
  - Shared memory
  - Semaphores (Named and unnamed)
  - Files/Pipes, etc.

Some properties of process-based concurrency

- Each process is isolated from the other
  - By design
  - "Shared nothing" concurrency
- Processes have to opt-in to share data
  - Must use OS services (i.e. system calls) to do so
- Cost of sharing mechanisms can vary
  - Some are cheap like `mmap`
  - Others are less cheap, like semaphores

Where is process-based concurrency used?

- Web Browsers
  - Each tab of your web browser is a separate process
  - Security? Reliability?
- Distributed Computing
  - The processes are usually not on the same machine
  - Remote procedure calls (RPC)
  - e.g., SETI@HOME or Folding@HOME
  - e.g., nearly every website
- High-performance Cluster Computing
  - A cluster is a connected network of computers
  - One of the many possible designs for supercomputers (but most popular right now)
  - "Message-passing"

An alternative: Thread-based parallelism

- Thread-based parallelism
  - Notion of a "thread of execution"
  - A thread is nearly always just a program counter + stack
  - Compare to process which has its own address space, etc.
- Also called "shared memory parallelism"
  - All threads share the same address space
- Each thread can read/write other threads data directly
  - Without going through the OS

Two prominent implementations of threading

- User-mode threading
  - Threads are invisible to OS
  - OS only sees a process
  - Process manages creation, termination and *scheduling* of threads
- OS-level threading
  - Threads are visible to OS
  - OS sees both processes and threads
  - Process uses OS facilities to create and terminate threads
  - OS schedules threads

Our focus today: OS-level threading

- OS-level threading is supported in Unix-like systems through POSIX Threads
  - Usually referred to as `pthreads`
- You need to link your program with `libpthread`
  - `gcc -pthread yourfile.c`

Outline

Threads Concepts

- Threads are created using `pthread_create`
  - Processes created using `fork`/`execve`
  - Each thread has a unique thread ID (within the process)
- A new thread starts at a specified function
  - A `fork` starts at instruction after call to `fork`
  - `execve` starts in `main`
- A thread exits using `pthread_exit`
  - This terminates the process *only* if this was the last thread
- Threads can wait for each other using `pthread_join`
  - Just like `waitpid`

Advantages of threads

- Very lightweight (compared to processes)
- Share the same address space as other threads
  - Same (global) data and heap
  - So all global variables and `malloc`'ed data is shared by default
  - But different stacks, so all function-local variables are still private
- Threads can read/write each others data directly
  - Using load/store instructions

Disadvantages of threads

- Share the same address space as other threads
  - Same (global) data and heap
  - So all global variables and `malloc`'ed data is shared by default
  - But different stacks, so all function-local variables are still private
- Threads can read/write each others data directly
  - "Shared everything"
  - No protection
  - Programmers must carefully control access to all shared data

Issues with Shared Everything: Thread safety

- If you're writing a program using pthreads, you're aware of what data is shared and not in your code
  - And you will use mutual exclusion mechanisms to correctly order accesses to shared data
- But what about all the code written by others that you're using?
  - e.g. `printf`, `fopen`, etc.
  - Some of these functions were designed in a pre-threads world
  - Do they store internal data that might inadvertently be shared by multiple threads?

The rand Function

```
#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);
```

- The rand function returns a (pseudo-)random value
- The srand function sets the seed for the next invocation of rand
  - The same seed produces the same random number sequence
- How does srand communicate the seed value to rand?

One possible implementation

```
unsigned int glseed;

void srand(unsigned int seed) {
   glseed = seed;
}

int rand() {
   ... read glseed to produce next random number ...
   ... store next random number in glseed ...
}
```

- This implementation uses global variables (i.e. `glseed`) to communicate the seed from `srand` to `rand`
- What will happen when `rand` is called by different threads?

Thread Unsafe Functions

- A thread unsafe function is a function that is not designed to be called (at the same time) by multiple threads
- Some functions in C and POSIX cannot be used in a thread-safe manner
  - A list of such functions is available in the `pthreads` manual page
  - They usually have thread-safe replacements, e.g. `rand_r` for `rand`

Thread-Safe Functions

```
int rand_r(unsigned int *seedp);
```

- A thread-safe function is a function that can be safely called (at the same time) by multiple threads
- Unless explicitly noted, functions in C and POSIX are required to thread-safe
  - The manual page for each function contains a note on this (e.g. "MT-Safe")
- Common strategy is to expose any hidden state to the user
  - e.g., `rand_r` takes the seed as input

Thread Unsafe Data

```
#include <errno.h>

int errno;
```

- The `errno` global variable contains the error code of the last system or library call
- If two threads both encounter an error, what should the value of `errno` contain?

Thread-safe data

- Newer versions of POSIX redefine `errno` to be thread-specific global data
  - instead of process-specific global data
- Each thread gets its own copy of `errno`

A related concept: Re-entrant Functions

- A re-entrant function is a function that can be "re-entered" even when another call to it is in progress
  - POSIX calls these "async-signal-safe"
- A program installs a signal handler for `SIGCHLD`
- It then calls `printf` in `main`
  - But while `printf` is executing, you receive the signal
  - And in the signal handler you call `printf` to print a debug message
  - What happens to the first `printf` call still in progress?
- Note: no threads are used in this example

Undefined behaviour

- None of the functions in `stdio.h` are re-entrant
- Invoking any of these functions when a call is in progress (in the same thread) results in undefined behaviour
- However, `printf` is thread safe (required since C11)
  - It can be called even when a call is in progress in a different thread
- All re-entrant functions are thread safe IF called on thread-private data
  - Not all thread-safe functions are re-entrant
  - Usually applies when writing signal handlers

Writing Thread-Safe Functions

- Thread-unsafe functions access shared data without synchronization
- Therefore, to make a function thread safe, add synchronization
  - e.g. use semaphores for mutual exclusion

Writing Re-entrant Functions

- Re-entrant functions should not use synchronization
- Re-entrant functions must only access data that is:
  - a function-local variable, or
  - an argument passed to the function
- Re-entrant functions are also thread safe if their arguments are thread-private
  - Shared data as arguments would violate thread-safety because re-entrant functions do not use synchronization

Pitfalls of Threads

- Synchronization and ordering must be used for correctness
  - Similar to processes
- Shared by default data
  - All global variables and `malloc`
  - All pointers are in the same address space
- Must only use thread-safe functions
  - Or re-entrant functions with thread-private data
  - A consequence of shared everything...

Outline

Sum of n numbers, using threads

```
/* holds arguments to thread function */
struct thread_arg {
  int i;
  int *a;
  int N;
  int NPERTHREAD;
  atomic_uint *sum;
  pthread_t tid;
};

void *tsum(void *arg) {
  struct thread_arg *ta = (struct thread_arg *) arg;

  printf("In thread %d, adding array elements from %d\n", ta->i,
         ta->i * ta->NPERTHREAD);

  for(int j = ta->i * ta->NPERTHREAD;
      j < (ta->i * ta->NPERTHREAD + ta->NPERTHREAD) && j < ta->N; j++) {
    *ta->sum += ta->a[j];
  }

  printf("In thread %d, sum is %d\n", ta->i, *ta->sum);
  pthread_exit(NULL);
}
```

Code explanation

- Thread entry functions (here `tsum`) can only accept a single `void *` argument
- We use that to send a structure containing all the arguments
  - We could avoid this in `fork` because child processes would start after fork in `main`
  - But now, the thread will start in `tsum`, which has no access to variables in `main`
- The code within `tsum` is not much altered from the code in the `fork()` variant
  - Except all arguments are read from the `ta` structure
- Note the `void *` return type of thread function
  - This function calls `pthread_exit` explicitly
  - If you used `return` instead, `pthread_exit` would be called implicitly with the return value: `return NULL` is equivalent to `pthread_exit(NULL)`

Threads: Initializing arguments

```
atomic_uint sum = 0;
NPERTHREAD = (N+nthread-1)/nthread;

struct thread_arg *ta = calloc(nthread, sizeof(struct thread_arg)

for(int i = 0; i < nthread; i++) {
  ta[i].a = a;
  ta[i].NPERTHREAD = NPERTHREAD;
  ta[i].N = N;
  ta[i].sum = &sum;
}
```

Threads: Creating threads

```
/* loop that creates threads */
for(int i = 0; i < nthread; i++) {
  ta[i].i=i;
  if(pthread_create(&ta[i].tid, NULL, tsum, &ta[i]) != 0) {
    fprintf(stderr, "ERROR: Failed to create thread\n");
    exit(1);
  }
}
```

Threads: Waiting for threads

```
int s = 0;
void *res;
for(int i = 0; i < nthread; i++) {
  s = pthread_join(ta[i].tid, &res);
  if(s != 0) {
    fprintf(stderr, "ERROR: Could not join\n");
    exit(2);
  }
}

printf("In main, sum is %d\n", sum);
```

Synchronization

- Semaphores can still be used with threads
  - `sem_init`, `sem_wait`, and `sem_post`
- But PThreads also offers other synchronization mechanisms
  - Mutexes: `pthread_mutex_init`, ...
  - Barriers: `pthread_barrier_init`, ...
  - Condition variables: `pthread_cond_signal`, ...

Mutexes

- Like binary semaphores
- Call `pthread_mutex_init` to initialize a mutex variable
- All threads wishing to enter a critical section call `pthread_mutex_lock` on a shared mutex variable
  - This attempts to "obtain a lock"
  - It will wait if the lock is already taken by another thread
- A thread that has the lock will call `pthread_mutex_unlock` to exit the critical section
  - One of the waiting threads will then be allowed in

Barriers

- Barriers cause threads to wait until a pre-determined number of threads arrive at the barrier
  - Usually, the number of threads is all the threads
- Barriers are commonly used to order phases of program execution
  - Each thread executes a phase independently, and then waits for all other threads to complete the phase before moving to the next
- Call `pthread_barrier_init` to initialize a barrier variable
  - You need to specify the number of threads
- Each thread calls `pthread_barrier_wait` on the barrier variable
  - This will force thread to wait until all other threads reach the barrier
- When the last thread arrives at the barrier, all threads proceed

Condition Variables

- Condition variables allow threads to wait for condition to be true
  - Efficient alternative to "spinning" (i.e. a loop that constantly checks a variable)
- A threads locks a mutex, and waits on a condition variable that becomes associated with that mutex
  - The mutex is unlocked and the thread put to sleep in one atomic action
  - When the condition becomes true, the thread wakes after the lock is re-acquired
  - Example: Producer puts item in queue, consumer thread wakes up and can immediately dequeue it
- A thread can "wake up" the waiting threads by "signalling" the condition

Condition Variables: API

- Call `pthread_cond_init` to initialize a condition variable
- A thread calls `pthread_cond_wait` on the condition variable and a mutex
  - This causes it to wait until condition variable is "signalled"
- A thread calls `pthread_cond_signal` to wake up waiting threads
  - Wakes up one thread
  - Can also call `pthread_cond_broadcast` to wake up all waiting threads

Outline

Multiple Processors and Cores

- You can run concurrent code on a system with 1 processor
  - Thanks to time sharing
- But most computers have multiple cores today
  - Each core is an independent computational unit
- Systems can also have multiple processors
  - Each processor contains multiple cores
  - Rare in consumer-grade systems

Mapping Processes and Threads to Cores

- The OS scheduler maps processes and threads to cores
- It is possible to "pin" threads/processes to certain cores
  - Avoids scheduling overhead
  - Can improve performance in some situations
- On Linux, the `sched_setaffinity` function allows you to set thread affinities
  - Can also use the `pthread_setaffinity_np` function

A Sneak Peek at Cache Coherence

- Recall that caches contain copies of data variables
  - This is fine when only one process/thread is accessing the data
- What happens when different threads access shared data?
  - Core 1 has shared variable `sum` in its cache
  - Will Core 2 try to get `sum` from memory?

Cache Coherence

- Cache coherence is a hardware mechanism to locate copies of a piece of data and use the "latest" version
  - Usually, the last written version
- Core 2 will send a request for `sum`
  - Core 1 will reply to that request
  - RAM may also reply, but Core 1 has more recent version and will be used by Core 2
- Coherence protocols also prevent multiple cores from writing to the same piece of data
- Cache coherence is covered in CS2/458, and also in CS2/451 and (possibly) ECE404

References

- Chapter 12
  - Except 12.2 (I/O Multiplexing)