OS: C Threading API (pthreads)

CIT 595
Spring 2010

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- API specifies behavior of the thread library, implementation is up to development of the library
- Note: To compile program
  - gcc –lpthread filename.c
  - Must provide –lpthread flag as this library is dynamically linked (i.e. linked at runtime)

Thread Creation

```c
int pthread_create (pthread_t * thread, pthread_attr_t * attr, void (* start_routine) (void *), void * arg)
```

- `thread`: unique identifier for the new thread returned by the subroutine
- `attr`: attribute object that may be used to set thread attributes
  - You can specify a thread attributes object, or NULL for the default values
  - Attributes such as stack size, priority, joinable or detachable
- `start_routine`: the C routine that the thread will execute once it is created
  - function that thread perform must be void * funcname (void *)
- `arg`: A single argument that may be passed to start_routine.
  - It must be passed by reference as a pointer cast of type void
  - NULL may be used if no argument is to be passed
  - It should be cast to its correct type in the function
- returns 0 when successful

Designing Concurrent Programs with Pthreads

- Take advantage of Pthreads, the work should be organized into discrete, independent tasks which can execute concurrently
- For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading

Source: https://computing.llnl.gov/tutorials/pthreads/
Example thread1.c

```c
struct threadData{
    int id;
};
typedef struct threadData thData;

int main()
{
    thData thDataArray[NUM_THREADS];
    int rc, t;
    for(t=0; t < NUM_THREADS; t++)
    {
        thDataArray[t].id = t;
        printf("In main: creating thread %d
", t);
        rc = pthread_create(&threads[t], NULL, PrintHello,
            (void *)&thDataArray[t]);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d
", rc);
            exit(-1);
        }
    }
    return 0;
}

void *PrintHello(void *threadid)
{
    thData * temp = (thData *)threadid;
    printf("Hello World! It's me, thread #%d!
", temp->id);
    pthread_exit(NULL);
}
```

Example thread1.c contd..

```c
//Main contd
for (t = 0 ; t < NUM_THREADS ; t++) {
    rc = pthread_join( threads[t], NULL );
    if (rc != 0) {
        printf("Error joining thread %d
", thDataArray[t].id);
        exit(-1);
    }
}

return 0;
```

Incorrect argument passing

- The code below passes the address of variable `t`, which
  is shared memory space and visible to all threads.
- As the loop iterates, the value of this memory location changes,
  possibly before the created threads can access it.

```c
int rc; int t;
for(t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
    ...}
```

Thread Termination

- void pthread_exit(void *value_ptr);
- `value_ptr` is a pointer to the object (variable, array, structure) returned by the thread.
- The value must not be of local scope otherwise it won’t exist after the thread is destroyed.
- This value can available to another thread in the same process.
- Can be NULL if not returning anything.
Thread join

- One way to accomplish synchronization between threads
- Causes the calling thread to wait for another thread to terminate
- For threads, it is important as we run the risk of executing an exit (reach the end of main) which will terminate the process and all threads before the threads have completed.

```c
int pthread_join(pthread_t thread, void ** value_ptr)
```
- `thread` is the thread to wait on
- `value_ptr` is the value given to `pthread_exit()` by the terminating thread
- returns 0 to indicate success

Usage:
```c
void * return_val;
...
//after code creating threads
if(pthread_join(worker_thread, &return_val)){
    printf("Error while waiting on thread
");
    exit(1);
}
```

Mutex

- Mutex is short for mutual exclusion
- Primary means of implementing thread synchronization and for protecting shared data when multiple writes occur
- Provides lock mechanism for shared data

To create a mutex:
```c
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t * attr)
```
- `mutex` is the lock (of type pthread_mutex_t)
- `attr` is the lock attributes
  - NULL by default

To lock:
```c
int pthread_mutex_lock(pthread_mutex_t *mutex)
```
- If lock is already locked, the calling thread is blocked
- If lock is not locked, the calling thread acquires it
- returns 0 on success

To unlock:
```c
void pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Typical sequence when using a mutex

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one succeeds and that thread owns the mutex
- The owner thread performs some set of actions
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- Finally the mutex is destroyed

Mutex (contd..)

- Declare variables as globals i.e. outside all methods
  - E.g. `pthread_mutex_t myMutex;`

- For synchronization:
  ```c
  pthread_mutex_lock(&myMutex)
  //critical section code
  pthread_mutex_unlock(&myMutex)
  ```

- After all work is done, need to destroy them
  ```c
  pthread_mutex_destroy(&myMutex);
  ```

- See example thread2.c
Deadlock in Resource Sharing Environment

- A **deadlock** occurs when 2 or more tasks (processes/threads) permanently block each other by each having a lock on a resource which the other tasks are trying to lock.

- Deadlock can occur due to:
  - Locks: Waiting to acquire locks on resources, such as objects, pages etc.
  - Sharing resources such as I/O devices printer, disks etc.

Example

- 2 Threads access 2 shared variables A and B
- Variable A is protected by lock x and variable B by lock y
- Here’s what Thread 1 and Thread 2 need to do:

  **Thread 1**
  - A = A + 10
  - B = B + 20
  - A = A + B
  - A = A + 30

  **Thread 2**
  - B = B + 10
  - A = A + 20
  - A = A + B
  - B = B + 30

- Each must acquire locks for A and B
- Lets look at one way to do this

Approach

**Thread 1**
- Lock(x)
- A = A + 10
- Lock(y)
- B = B + 20
- A = A + B
- Unlock(y)
- A = A + 30
- Unlock(x)

**Thread 2**
- Lock(y)
- B = B + 10
- Lock(x)
- A = A + 20
- A = A + B
- Unlock(x)
- B = B + 30
- Unlock(y)

- Can we see a problem with this approach?
- How can we avoid the problem?

Cond and Wait

- Another means of synchronization
- Condition variables allow threads to synchronize based upon the actual value of data
  - If we want one thread to signal an event to another we need to use Conditional variables
    - `pthread_cond_t condVariable`
  - Idea is one thread wait until a certain condition is true
    - Test condition
      - If not true, calls `pthread_cond_wait(…)` to block until it is
  - Another thread makes the condition true and call `pthread_cond_signal(…)` to unblock the thread waiting
  - To avoid race conditions, the conditional variable must use a mutex
Example

- Signalling thread
  - `pthread_mutex_lock(&mutex);`
  - `flag = 1; pthread_cond_signal(&condition);`
  - `pthread_mutex_unlock(&mutex);`
  - **condition** is conditional variable
    - type `pthread_cond_t`
  - **mutex** is mutex variable
    - type `pthread_mutex_t`

- Waiting thread
  - `pthread_mutex_lock(&mutex);`
  - `if(flag == 0)`
    - `pthread_cond_wait(&condition, &mutex);`
  - `pthread_mutex_unlock(&mutex);`

- Wait will automatically release the mutex while it waits
- After signal is received and thread is awakened, mutex will be automatically locked for use by the thread.
- Programmer is then responsible for unlocking mutex when the thread is finished with it

Producer Consumer Example

- Single producer thread, single consumer thread
  - Single shared buffer between producer and consumer
  - E.g. A fixed-size queue of print requests
    - One thread produces information – adds a print request to the queue
    - Other thread consumes information – takes a print request and prints it

- Condition and Wait
  - Involves mutual exclusion between producer and consumer
    - Due to use of same buffer
    - After producing an item, a producer should signal the consumer
    - After consuming, consumer should signal the producer

- See example thread3.c for illustration

Misc.

- **pthread_self()** returns the unique, system assigned thread ID of the calling thread
- **Thread cancelation**
  - `int pthread_cancel(pthread_t thread)`
    - Causes the thread to be canceled (or terminated)
- **Thread Attributes**
  - By default, a thread is created with certain attributes
  - `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object
  - `pthread_attr_getXXX (...)` gets the attribute and `pthread_attr_setXXX (...)` sets the attribute
- **Mutex Attributes**
  - Like threads, mutexes also have attributes
  - Related to thread scheduling (more details in scheduling topics)