Lecture 21: Threads Introduction

- Multiple strands of execution within a single process are called __________.
- Threads share the same global memory (data and heap), but each thread has its own stack (automatic variables).
- Threads also share the same process ID, controlling terminal, user and group IDs, open file descriptors, signal handlers, and more (see the pthreads man page).

Lecture 21: Threads

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
- Reading and writing to the same memory locations is possible, and therefore requires ___________ by the programmer.

Lecture 21: Threads Disadvantages of Threads

- Writing multithreaded programs requires very careful design. Errors in synchronization or shared memory access are easy to make (and hard to __________).
- Splitting a large calculation into two parts as different threads does not necessarily result in better performance.
- If you use a function from more than one thread it must be re-entrant.

Lecture 21: Threads Linux Threads

- The POSIX ___________ API is currently the most popular API for multi-threaded programming and will be the only threading API discussed.
- The latest ISO C standard (C11) includes threads. The ISO C threading API differs from the pthreads API. ISO C threads are not currently widely available.
Lecture 21: Threads
Linux Threads

- Programs using the POSIX thread routines should include the `pthread.h` header file.
- Compile programs using the "-pthread" option. This option sets ______ for both the compiler and linker.

Lecture 21: Threads
Thread Management - Creating

- All processes contain a main thread. Additional threads are created using the `pthread_create()` routine:
  ```c
  int pthread_create(
      pthread_t *thread,
      pthread_attr_t *attr,
      void *(*startRoutine)(void *),
      void *arg);
  ```
- On success 0 is returned and the thread ________ is returned in thread.

Lecture 21: Threads
Thread Management - Creating

- The `attr` argument specifies thread attributes. See the `pthread_attr_init` man page for a list of attributes. It can be NULL if the default attributes are to be applied.
- The new thread executes the `startRoutine` function. `arg` is passed as the first argument to the function.
- All threads are ________. New threads can create other threads.

Lecture 21: Threads
Thread Management - Exiting

- You can use `pthread_exit()` to explicitly exit a thread. (A simple return from the thread routine will also terminate the thread.)
- If `main()` exits with `pthread_exit()` other threads will continue to run. Otherwise they will be automatically ____________.
- `pthread_cancel()` can be used by one thread to cancel another one.

Lecture 21: Threads
Thread Management - Joining

- Use `pthread_join()` to ______ for a specific thread to terminate:
  ```c
  int pthread_join(
      pthread_t thread,
      void **thread_return);
  ```
- `pthread_join()` waits on the thread with ID `thread` to complete. If `thread_return` is not NULL, the thread return value is stored at the location pointed to by `thread_return`.

Lecture 21: Threads
Thread Management - Joining

- A thread may have either the `joinable` or `detached` attribute. Only threads that are created as `joinable` (the ________) can be joined. If a thread is created as `detached`, it can never be joined.
- `pthread_detach()` can be used to detach a thread created as `joinable`. This can be done to conserve resources.
- See `pthread_intro.cpp` for an example.
Lecture 21: Threads
Thread Management - Misc.

- `pthread_self()` returns the unique, system assigned thread ID of the calling thread. `pthread_equal()` compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

```c
pthread_t pthread_self();
int pthread_equal(pthread_t thr_id2, pthread_t thr_id2);
```

- The `pthread_once()` routine can be used to ensure that _______ code is run exactly one time.

Lecture 21: Threads
Thread Synchronization

- A `join` is one mechanism for obtaining synchronization between threads.

- The threads library also provides _______ and condition variables for synchronization. These will be discussed in the next lecture.

- SYS V or POSIX semaphores can be used for thread synchronization. POSIX unnamed semaphores are especially attractive.

- See the `pthread_sem.cpp` example.

Lecture 21: Threads
Reentrancy

- The threaded routines must call functions which are re-entrant or thread safe. A re-entrant routine is one that can be safely executed concurrently; that is, the routine can be re-entered while it is already running.

- To be re-entrant a routine: (1) must not use _______ variables, (2) must not return the address to static data, (3) must work only on the data provided to it by the caller, (4) must not call non-reentrant functions.

Lecture 21: Threads
Reentrancy

- If static variables are used then mutexes (semaphores) must be applied or the functions must be re-written to avoid the use of these variables.

- In C, local variables are dynamically allocated on the _______. Therefore, any function that does not use static data or other shared resources is thread-safe.

Lecture 21: Threads
Reentrancy

- Thread-unsafe functions may be used by only one thread (mutexes may be used to ensure that this is so).

- Many non-reentrant functions return a _______ to static data. This can be avoided by returning dynamically allocated data or using caller-provided storage. An example of a non-thread safe function is `strtok` which is also not re-entrant. The "thread safe" version is the re-entrant version `strtok_r`.

Lecture 21: Threads
Reentrancy

- An example of a non-reentrant function:

```c
int g_var = 1; // global counter
void inc_g() { g_var++; }
```

- A slight alteration for reentrancy:

```c
void inc_g()
{
    sem_wait(&g_var_sem);
    g_var++;
    sem_post(&g_var_sem);
}
```