Threads (part 1)

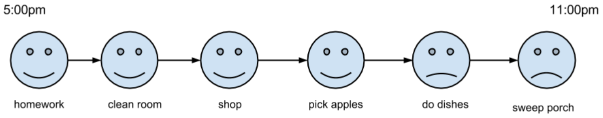
August 25, 2017

Threads

- Threads (an abbreviation of *threads of control*) are how we can get more than one thing to happen simultaneously in a program.
- Threads can be used for several reasons:
  - To improve responsiveness by handling asynchronous events simultaneously.
  - To improve run-time by parallelizing a computation (breaking it into smaller parts, doing each part in parallel and then combining results if needed).
  - To handle background operations.
  - Other reasons: sharing resources, modeling natural processes etc.

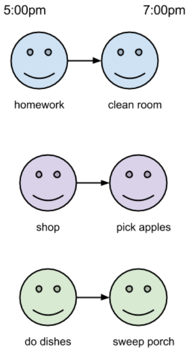
Thread Example: Parallelization (1)

- You have a set of 6 tasks you have to do:
  - do your homework
  - clean up your room
  - do the shopping
  - pick the apples in the garden
  - wash the dishes
  - sweep the porch
- Each task takes 1 hour to do.
- If you have to do all the work by yourself, it will take you 6 hours (If you start at 5:00, all tasks will be completed at 11:00).



Thread Example: Parallelization (2)

- If you find two friends and divide the work and you all start working at the same time, you will finish all the tasks sooner.
  - All 3 of you will do 2 tasks each, which will take 2 hours (If you all start at 5:00, all tasks will be completed at 7:00! Plenty of time to do even more homework!).



Thread Examples: Handling Asynchronous Events

Life is asynchronous!

Asynchronous means things can happen independently unless there's some enforced dependency.

- *Handling events in GUIs*. A thread can watch for user clicking on a button (or some part of the GUI) and activate a listener to handle the event. Another thread can do the graphics while the first thread keeps the GUI responsive.
- *Overlapping computation with I/O*. For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.

Thread Examples: Handling Asynchronous Events

- *Asynchronous event handling*. Tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests by using two separate threads.
- *Prioritizing tasks*. Tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.

Thread Examples

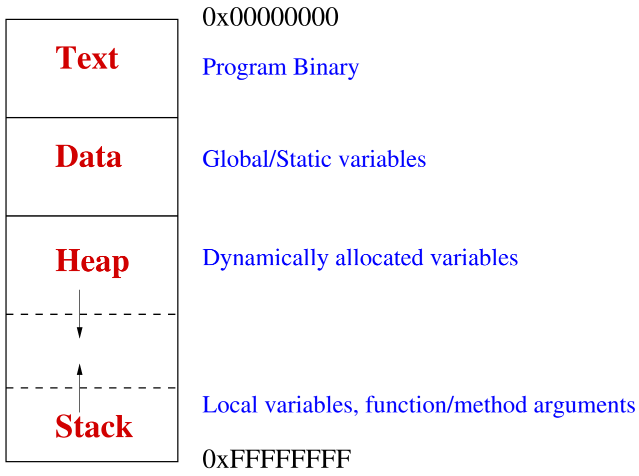
Some examples of where threads are useful:

- Windowing systems
- GUI applications
- Web browsers
- Database servers
- Web servers

Threads

- A thread (a.k.a. lightweight process) is associated with a particular process (a.k.a. heavyweight process, a retronym).
- A heavyweight process may have several threads and a thread scheduler.
- Review the Linux/UNIX process model.

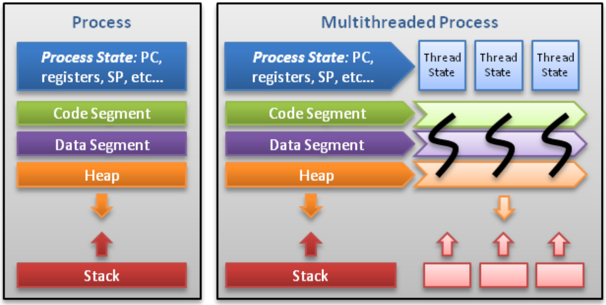
The Linux/UNIX Process model



Thread model

- Threads exist within a process and use the processes resources.
- A thread is a sequence of instructions within a program that can be executed indepenently of other code.
- Threads share the text, data and heap segments. Each thread has its own stack and status.
- Allows threads to be scheduled by the OS and run as independent entities.

Thread and Process models

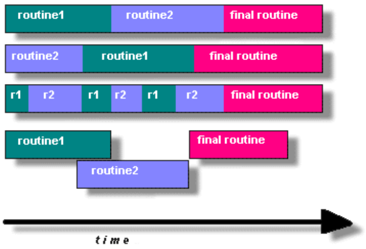


Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

© Alfred Park, http://randu.org/tutorials/threads

Threaded Programs

- In general, in order for a program to take advantage of threads, it must be able to 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.



Pthreads: POSIX Threads Library

A standardized threads library that is the native threads library under Linux. Contains around a hundred functions. We will examine a few core functions.

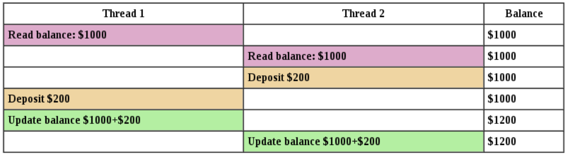
- `pthread_create()` Creates and starts running a new thread from the specified start function.
- `pthread_exit()`: Terminates the calling thread.
- `pthread_join()`: Wait for the specified thread to finish.
- `pthread_self()`: Prints the thread id of the calling thread.

Threading Examples (1)

- threads/thread-hello-world.c
- threads/thread-better-hello-world.c
- threads/thread-test.c
  - Use the command `ps xm` to see all threads and processes. Or use the KDE system guard `ksysguard` to monitor processes and number of threads. Also see threads in Task Manager for Microsoft Windows.
- threads/thread-ids.c

Threading Examples (2)

- threads/thread-sum.c A multithreaded parallel sum program.
- threads/bad-bank-balance.c Illustrates race conditions when multiple threads access the same global variable and the result depends on the interleaving of threads.



Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Thread Synchronization

- Mutex - A construct used to protect access to a shared bit of memory.
- Think of a lock that only has one key. If you want to open the lock you must get the key. If you don't have the key you must wait until it becomes available.
- A Mutex, short for *Mutual exclusion object*, is an object that allows multiple program threads to share the same resource, such as a data structure or file access, but not simultaneously. Each thread locks the mutex to gain access to the shared resource and then unlocks when it is done. We can use mutexes to prevent race conditions.

Threading Examples (3)

- See the example lab/threads/safe-bank-balance.c for a solution to the race condition using a Mutex lock. Mutexes will be studied in depth in the Operating Systems class.
- See the folder threads/account for a better structured solution to the same problem using Mutex objects.

In-class Exercise (1)

Evil in the Garden of Threads? The following code shows the usage of two threads to sum up a large array of integers in parallel.

```c
/* appropriate header files */
static void *partial_sum(void *ptr);
static int *values, n;
static int result[2]; /* partial sums arrays */

int  main( int argc, char **argv) {
    int i;
    pthread_t thread1, thread2;
    if (argc != 2)
      {fprintf(stderr, "Usage: %s <n> \n", argv[0]); exit(1);}
    n = atoi(argv[1]);
    values = (int *) malloc(sizeof(int)*n);
    for (i=0; i<n; i++)
        values[i] = 1;
    pthread_create(&thread1, NULL, partial_sum, (void *) "1");
    pthread_create(&thread2, NULL, partial_sum, (void *) "2");
    do_some_other_computing_for_a_while();
    printf("Total sum = %d \n", result[0] + result[1]);
    exit(0);
}
void *partial_sum(void *ptr) {
        /* same as the example earlier */
}
```

In-class Exercise (1)

Choose the statements that best explains how the code runs.

1. The code always adds the `values` array correctly.
2. The code never adds the `values` array correctly.
3. The code sometimes adds the `values` array correctly.
4. The code will corrupt the `values` array because of a race condition.

In-class Exercise (2)

Did you forget to Lock?

```
/* appropriate header files */
void *run(void *ptr);
pthread_mutex_t mutex;
int count = 1000;

int  main( int argc, char **argv) {
    pthread_t thread1, thread2;
    struct list *sharedList = create_and_populate_list();
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&thread1, NULL, run, (void *) sharedList);
    pthread_create(&thread2, NULL, run, (void *) NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_mutex_destroy(&mutex);
    exit(0);
}
void *run(void *ptr) {
    struct list *sharedList = (struct list *) ptr;
    for (i = 0; i < count; i++)
        update_data(sharedList);
}
void update_data(struct list *sharedList) {
    pthread_mutex_lock(&mutex);
    if (sharedList == NULL) return;
    // update the list as needed */
    pthread_mutex_unlock(&mutex);
}
```

In-class Exercise (2)

Choose the statements that best explains how the code runs.

1. Both threads update the list in a thread-safe manner.
2. The threads sometimes terminate and sometimes deadlock.
3. The threads deadlock every time and never stop.
4. The threads will corrupt the list because of a race condition.

Multithreading support in GDB

The gdb debugger provides these facilities for debugging multi-thread programs:

- Automatic notification of new threads.
- thread threadno, a command to switch among threads
- info threads, a command to inquire about existing threads
- thread apply [threadno] [all] args, a command to apply a command to a list of threads
- Thread-specific breakpoints

See GDB manual for more details.

Exercises

1. **Thread Dance**. Convert the program in `lab/threads/random` to use multiple threads and measure how much speedup you get relative to the number of CPUs on the system for generating the given number of random values. Also experiment with increasing the number of threads to be higher than the number of CPUs. Allow the user to specify the number of threads via a command line argument as follows:
   `random <numberOfRandoms> <numThreads>`
2. **The fault in our threads**. Modify the `TestList.c` driver program from your linked list project so that it creates multiple threads that run random tests on the same linked list. Also modify the driver program to accept an additional command line argument to specify the number of threads. Since all threads are sharing the same list without any protection, expect many segmentation faults due to race conditions. The purpose of this assignment is to explore race conditions in multi-threaded code.
3. **Safety in the mosh pit!** Study the `lab/threads/safe-bank-balance.c` example to see the use of Pthread mutexes. Try to implement the same idea to protect your linked list from the previous assignment. Can you get it to work safely? Can you prove that it is safe? These topics will be covered in much more depth in the Operating Systems class.
4. **Multithreaded Chat Server**. Write a multithreaded server that can chat with multiple clients simultaneously. Also write a simple client program to test the server. Use named pipes for the communication between clients and servers.