CS 33 Week 8

Section 1G, Spring 2015
Prof. Eggert (TA: Eric Kim)
v1.0

Announcements

- Midterm 2 scores out now
  - Check grades on my.ucla.edu
  - Mean: 60.7, Std: 15 Median: 61
- HW 5 due date updated
  - Due: May 26th (Tuesday)
- Lab 4 released soon
  - OpenMP Lab
  - Tutorial on OpenMP
    - http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Overview

- More concurrency
- File I/O
- MT 2

Example: sum

```
int result = 0;
void sum_n(int n) {
  if (n == 0) {
    result = n;
  } else {
    sum_n(n-1);
    result = result + n;
  }
}
```

Suppose Louis Reasoner tries to make this code thread safe...

Example: fib

```
int result = 0;
sem_t s; // sem_init(&s,1);
void sum_n(int n) {
  if (n == 0) {
    P(&s); result = n; V(&s);
  } else {
    P(&s);
    sum_n(n-1);
    result = result + n;
    V(&s);
  }
}
```

Question: Is there anything wrong with this code?

Answer: Yes, deadlock! sum_n(5) calls sum_n(4), but sum_n(4) can't acquire mutex. sum_n(5) can't make progress without sum_n(4) - thread is stuck.

Thread-safe functions

Definition: A function is **thread-safe** if functions correctly during simultaneous execution by multiple threads. [http://stackoverflow.com/questions/261683/what-is-meant-by-thread-safe-code]

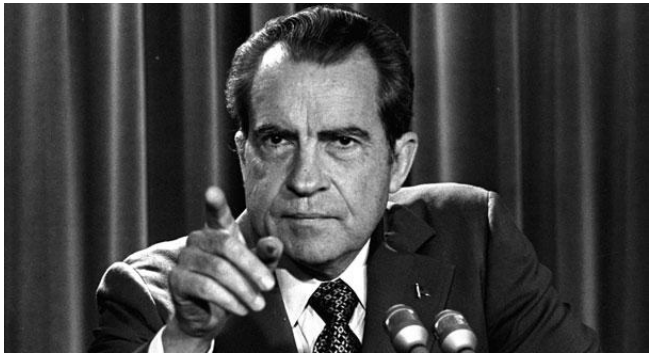
Alt: In computer programming, **thread-safe** describes a program portion or routine that can be called from multiple programming threads without *unwanted interaction* between the threads. [http://whatis.techtarget.com/definition/thread-safe]

Thread-safe functions

Typically achieve thread-safety by mechanisms:

- Synchronization (ie semaphores/mutexes)
- Careful handling of shared data

"To write code that will run stably for weeks takes extreme paranoia."



(Not actually said by Nixon)

Reentrant Functions

A "stronger" version of thread-safe (in a sense).

Some conditions/characteristics of a reentrant function:

- Does not use any shared data
- Any "state" is passed in as parameters

Reentrant Functions

Following example is from:

https://www-01.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.genprogc/writing_reentrant_thread_safe_code.htm?cp=ssw_aix_61%2F13-3-12-18

Helpful reading on thread safety and reentrancy.

Ex: strtoupper

```
/* non-reentrant function */
char *strtoupper(char *string) {
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Question: Is this threadsafe?

Answer: Nope! Two threads running strtoupper() will write to shared buffer.

Ex: strtoupper

```
/* reentrant function (a poor solution) */
char *strtoupper(char *string) {
  char *buffer;
  int index;
  /* error-checking should be performed! */
  buffer = malloc(MAX_STRING_SIZE);
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Ex: strtoupper

```
/* reentrant function (a better solution) */
char *strtoupper_r(char *in_str, char *out_str) {
  int index;
  for (index = 0; in_str[index]; index++)
    out_str[index] = toupper(in_str[index]);
  out_str[index] = 0
  return out_str;
}
```

Ex: numseq()

```
/**
 * Returns the sequence of numbers: 2*(n-1), ie:
 *   2, 4, 6, 8, ...
 **/
int numseq() {
  static unsigned int n = 0;
  n = n + 1;
  return 2*n;
}
```

Question: Is this reentrant? Threadsafe?

Answer: Not reentrant.
Not threadsafe.

Ex: numseq()

```
/**
 * Returns the sequence of numbers: 2*(n-1), ie:
 *   2, 4, 6, 8, ...
 **/
int numseq(int* n) {
  *n = *n + 1;
  return 2*(*n);
}
```

Question: Is this reentrant? Threadsafe?

Answer: This is reentrant ***if*** "int* n" does not point to a shared variable.
Book calls this "implicitly" reentrant.
Similarly, this is threadsafe as long as "int* n" is not a variable shared among threads.

Reentrancy vs Thread Safety

Question: Are threadsafe functions always reentrant?

```
void f() {
  mutex_acquire();
  // suppose signal handler gets invoked here!
  do_important_stuff();
  mutex_release();
}
```

Answer: Nope! Suppose function f() is used as a signal handler. Suppose we are executing f(), and acquire the mutex. Then, suppose signal handler gets invoked again, and we invoke f() again. The signal handler will get stuck trying to acquire the mutex!

Reentrancy vs Thread Safety

Question: Are reentrant functions always threadsafe?

Answer: According to your textbook, yes. This is using the definition that reentrant functions never access shared data.

For fun, let's consider a slightly more nuanced definition of reentrant functions.

An Alternate Definition of Reentrant

History: Reentrancy is an idea that originated in single-threaded environments.

People wanted functions that worked correctly even if a hardware interrupt happened during a function execution.

Thus, reentrancy and thread-safety are *actually* two separate ideas.

Reentrancy vs Thread Safety

```
int t;
void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    *y = t;
}
```

This is not reentrant, not threadsafe.

Exercise: Show why this is not reentrant from a hardware-interrupt perspective.

Reentrancy vs Thread Safety

A reentrant version of swap:

```
int t;
void swap(int *x, int *y) {
    int s;
    s = t; // save global variable
    t = *x;

    *x = *y;
    *y = t;
    t = s; // restore global variable
}
```

Suppose we get interrupted here, and another swap() call is made to completion. Since the second swap() call is careful to restore the value of t, the first swap() call will still produce the correct output.

Reentrancy vs Thread Safety

A reentrant version of swap:

```
int t;
void swap(int *x, int *y) {
    int s;
    s = t; // save global variable
    t = *x;

    *x = *y;
    *y = t;
    t = s; // restore global variable
}
```

Question: Is this threadsafe?

Answer: Nope! Say two threads T1, T2 call swap() concurrently. Say T1 finished "t = *x", then we switch to T2.
Say T2 does "t = *x", then we switch back to T1.
T1 will be using the wrong value of t!

Thoughts from the peanut gallery

The previous reentrancy example uses a definition of reentrancy that the textbook doesn't seem to use.

Perhaps the textbook is offering a simplified view of reentrancy?

File I/O

Do read the textbook (Chapter 10).

Tip: Actually write some C programs that read/write to files, using stdio.h function: fopen, fclose, fputc/fgetc, fread/fwrite.

Exercise: read()

```
char buf[10];
size_t n;
int i;
while (1) {
  n = read(f, buf, 10);
  printf("  Read in %d bytes!\n", n);
  for (i=0; i<n; ++i) {
    printf("  [byte %d/%d]: %d\n",i,n,buf[i]);
  }
}
```

Answer: read() returns a ssize_t, ie a *signed* int, since it may return -1. Thus, if read() returned -1, then the size_t would interpret -1 as a very big number - oops!

Question: read() returns 0 if EOF, -1 if error, or # bytes read. What is the bug in this code?

POSIX vs C library

POSIX library (ie <unistd.h>) provides low-level functions that allow user programs to talk to the operating system. Unix, MacOSX use this interface ("POSIX compliant").

Includes I/O functions like open(),close(),read(),write().

Also: fork().

POSIX vs C library

In theory, can write C programs to do all file I/O using only POSIX functions.

But, would be kind of annoying.

Instead, C library <stdio.h> includes a bunch of convenience functions, such as: fprintf()/fscanf(), fgets()/fputs(), fopen()/fclose(), etc:

http://www.cplusplus.com/reference/cstdio/

In practice, typically use <stdio.h> for your programs.

POSIX vs C library

<stdio.h> is built **on top of** <unistd.h> functions.

Quick Demo: C File I/O Examples

(1) Write C program to display text file contents

    v1: Use only POSIX functions

    v2: Use C library

(2) Write a C chatter bot.

Midterm 2

Let's go over each question.

(Can't release this part online, sorry!)

Remind class about OpenMP tutorial