CSE 374: Lecture 26

Concurrency

Function Pointers

Can point to code the way we point to data.

In C, the syntax is:

<return_type> (*<pointer_name>) (function_arguments);

Set equal to 'address of function' (&f)

```
double two(double x) {
  return 2.0;
}

printf("int two(x) = %e\n",
    integrate(&two, 0.0,
              2.0, 1.0));
```

```
double integrate(
    double (*f)(double),
    double lo, double hi,
    double delta) {
  ...
   ans += (*f)(x) *
          ((hi-lo) / ((n+1));
...
```

Function Pointers: nicer syntax

Typedef can be used to shorten datatype:

```
typedef double (*fdd)(double);
```

The C compiler is smart enough to know what is a function and what is a variable:

```
ans += (*f)(x) * ((hi-lo) / (n+1));
ans += f(x) * ((hi-lo) / (n+1));
```

Also interprets function name as a pointer to the code:

```
integrate(&sin, 0.0, PI/2.0, 0.01));
integrate(sin, 0.0, PI/2.0, 0.000001));
```

What is Concurrency?

- Running multiple processes simultaneously
  - Running separate programs simultaneously
  - Running two different ‘threads’ in one program
- Each ‘process’ is one ‘thread’
- Parallelism refers to running things simultaneously on separate resources (ex. Separate CPUS)
- Concurrency refers to running multiple threads on SHARED resources

Sequential programming demands finishing sequence before starting the next one

Previously, performance improvements could be made by improving hardware
- no longer (Goodbye Moore’s Law)

Allows processes to run ‘in the background’

★ Responsiveness - allow GUI to respond while computation happens
★ CPU utilization - allow CPU to compute while waiting (for data, input, etc)
★ Isolation - keep threads separate so errors in one don’t affect the others

‘Nice’ linux parallel processes

NAME

nice - run a program with modified scheduling priority

SYNOPSIS

nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION

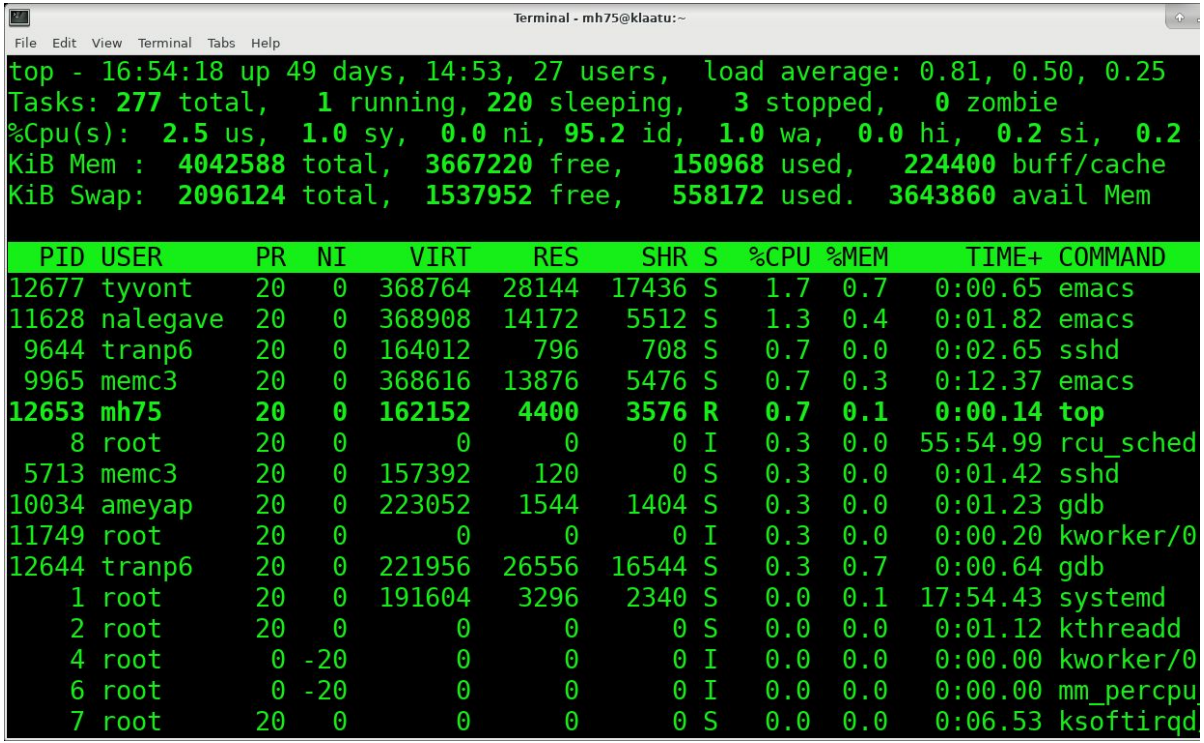
Run COMMAND with an adjusted niceness, which affects process scheduling. With no COMMAND, print the current niceness. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process).

Other Linux tools

Top - shows all processes with ‘niceness’ (NI)

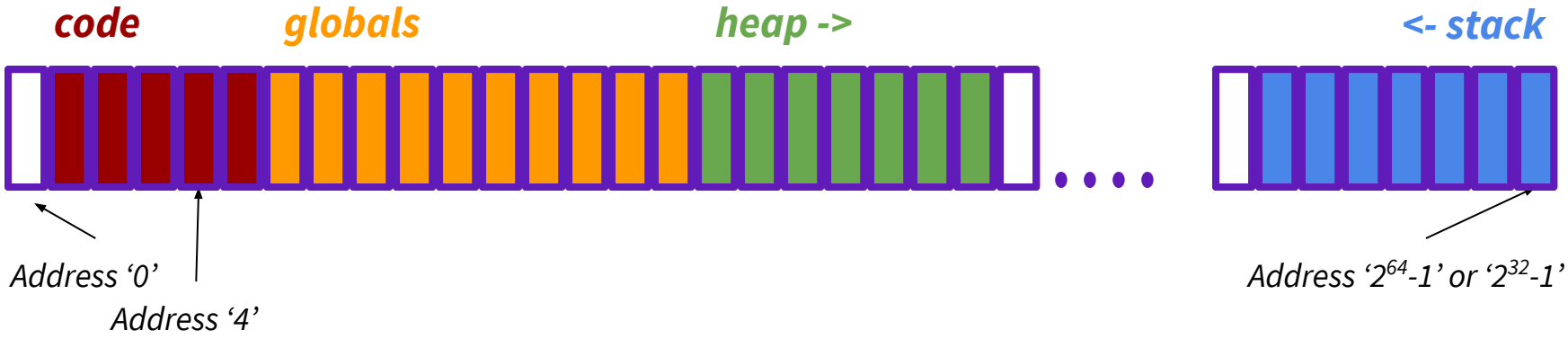
[mh75@klaatu ~]$ ps -o pid,comm,nice

PID COMMAND NI
11483 bash 0
13034 ps 0



Concurrency in Detail

- C, Java support parallelism similarly (other languages can be different)
  - one pile of code, globals, heap
  - multiple “stack + program counter”s — called threads
  - threads are run or pre-empted by a scheduler
  - threads all share the same memory
- Various synchronization mechanisms control when threads run
  - “don’t run until I’m done with this”



Concurrency in C & Java

C: the POSIX Threads (pthreads) library

- #include <pthread.h>
- pass -lpthread to gcc (when linking)
- pthread_create takes a function pointer and arguments, runs as a separate thread

Java: built into the language

- Subclass java.lang.Thread, and override the run method
- Create a Thread object and call its start method
- Any object can “be synchronized on” (later today)

(Aside: POSIX)

“ The Portable Operating System Interface (POSIX)[1] is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.[2][3]” - Wikipedia

The C ‘pthread’ conforms to the POSIX standard for threading.

Pthread functions

```
Pthread_t threadID;
```

The threadID keeps track of to which thread we are referring.

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

Note - pthread_create takes two generic (untyped) pointers
interprets the first as a function pointer and the second as an argument pointer.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Puts calling thread ‘on hold’ until *‘thread’* completes - useful for waiting to thread to exit

https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

Memory Consideration

(ex. pthreadex.c)

- If one thread did nothing of interest to any other thread, why bother running?
- Threads must communicate and coordinate
  - Use results from other threads, and coordinate access to shared resources
- Simplest ways to not mess each other up:
  - Don’t access same memory (complete isolation)
  - Don’t write to shared memory (write isolation)
- Next simplest:
  - One thread doesn’t run until/unless another is done

Parallel Processing

Common pattern for expensive computations (such as data processing)

1. split the work up, give each piece to a thread (fork)
2. wait until all are done, then combine answers (join)

To avoid bottlenecks, each thread should have about the same amount of work

Performance will always be less than perfect speedup

What about when all threads need access to the same mutable memory?

Multiple threads with one memory

Often you have a bunch of threads running at once and they might need the same mutable (writable) memory at the same time but probably not

- Want to be correct, but not sacrifice parallelism

Example: bunch of threads processing bank transactions

- withdraw, deposit, transfer, currentBalance, etc…

unlikely two will overlap, but there's a chance

very important that answer is correct when they overlap

Data races

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
    if (a->balance < amt) return FAIL;
    a->balance -= amt; return SUCCESS;
}
```

This code is correct in a sequential program

It may have a race condition in a concurrent program, allowing for a negative balance

Discovering this bug with testing is very hard

A Data Race - *two threads withdraw $100 simultaneously*

Thread 1

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }




     a->balance -= amt; return SUCCESS;
}
```

Thread 2

```
struct Acct {int balance; /*etc…*/ };
int withdraw(struct Acct* a, int amt) {
     if (a->balance < amt) {
          return FAIL; }
     a->balance -= amt; return SUCCESS;
}
```

Atomic Operations

- An operation we want to be done all at once
  - No interruptions
- Note: Must be the right size
  - Too big - program runs sequentially
  - Too small - program has potential races
- ‘Atomic’ requires a hardware primitive

We can wrap the hardware primitive with a lock

In C : ‘mutex’

```
std::mutex BankAccount::m_;
void BankAccount::withdraw(double amount) {
    m_.lock();
    if (getBalance() > b) {
        throw std::invalid_argument();
    }
    setBalance(getBalance() - amount);
    m_.unlock();
}
```

C mutex lock

1. Create a lock for specific data
2. Lock before atomic part of code
3. Unlock after atomic operation

What happens if more than one piece of code affects the data?

Idea: Use same mutex (‘m’) for each piece of code that modifies ‘balance_’

```
std::mutex BankAccount::m_;
void BankAccount::withdraw(double amount)
{
    m_.lock();
    if (getBalance() > b) {
        throw std::invalid_argument();
    }
    setBalance(getBalance() - amount);
    m_.unlock();
}
```

Deadlock

Problem:

If every method that modifies balance_ is locked with mutex m, that balance can not be updated.

Solution:

Must create helper function that allows for modifying balance_ under the lock.

```
void BankAccount::withdraw(double amount) {
  m_.lock();
  if (getBalance() < amount) {
    throw std::invalid_argument();
  }
  setBalanceUnderLock(getBalance() - amount);
  m_.unlock();
}

void setBalance(double amount) {
  m_.lock();
  setBalanceUnderLock(amount);
  m_.unlock();
}

void setBalanceUnderLock(double amount) {
  balance_ = amount;
}
```