CS3503A

Lecture #6

2018-04-17

Administrivia

- lab2 revised
    - minor changes
    - -Wall → oops
- detatched vs joinable threads (pthreads)
- shorter recap (by popular demand)

mmap

Our Story So Far (abbreviated)

- interprocess communication
  - pipes (regular ~~vs~~ or named)
  - Posix vs SysV IPC
  - SysV IPC
    - message queue
    - semaphores
    - shared memory — special handling of memory maps in kernel process control block
- low-level interfaces: { void pointer (typeless) & size }
  - cast pointers to type in client code
  - use sizeof operator to get size (# bytes) of object

  Most general possible interface, given that library has no type info

- message passing is safer (easier to reason about, hence less error-prone) than shared memory
- networking: preferred IPC mechanism because you can put processes on different machines
  - unless you need faster performance

Our Story So Far (abbreviated) (cont)

- threads vs. processes
  - threads: single process running single program
    - multiple concurrent paths of execution
    - shared resources (memory, etc.)
    - separate call stacks
    - wild pointers can clobber other threads
      - difficult to debug
    - advantages: lighter weight, faster context switches

- kernel, library, or hybrid implementation

  1:1   many:one   { many: many

  green threads   { multiplex

- distributed convex hull: more logically implemented using threads
  - except for pedagogical goals

- Posix threads:
  - pass in pointer to function that takes void pointer & returns void pointer
  - pass in void pointer argument
  - use casting in client code
  - join terminating threads to get return value

More About Pointers

- types: compile-time concept
  - machine code deals with addresses, bytes, words
  - higher-level languages include type info in runtime
    - e.g. Java reflection

- C - designed as a high-level assembler
  - considered low-level by modern standards
  - allows precise control of memory

- struct: layout in memory may include padding
  proof sizeof (char int char) != char char int
  (12) (8)
  (C++ class is struct with hidden pointer field)

- array: a[i] = *(address of a[0] + i × sizeof a)
  - (p + i) vs (char *) p + i
    add i × sizeof(*p)   add i × 1
  stepping through memory by increments of size of object
  a[i] = *(a + i) = *(i + a) = i[a]
  counter-intuitive
- casting: lets programmer assert that this block of memory corresponds to type

- void * : untyped address
  - can't do arithmetic because we don't know the size
  - traditionally: used char* for this purpose
    char * p → p+1 behaves as you would expect

More About Pointers (cont.)

- Pointers to functions: address of function entry point

- Can't do pointer arithmetic on function pointer
  - not meaningful to deref fn ptr
  - but we can call a function through a function pointer

- uses: { parameterizing behavior
  abstracting out decision

- e.g.: sorting: { ptr, size of element,
  number of elements,
  how to compare element

most general interface absent higher-level compiler constructs (e.g. type parameters)

- e.g. dynamic dispatch
  - vptr

```
struct Animal {
    void* speak(Animal*)
};
```

{ dog.speak() → woof (dog)
  cat.speak() → meow (cat)

→ C++ compiler does this for you

Critical Section

- region of code which only one process at a time may execute
  - i.e. concurrent execution would be problematic

- may be a performance bottleneck

- requirements:
  1) mutual exclusion
  2) progress
     - when the critical section is open, the system must decide among waiting processes
     - i.e. something must be chosen
  3) bounded waiting
     - a process cannot wait indefinitely
       → starvation problem

Synchronization

- race condition
  - two threads need to update same data structure
  - update is not atomic
  - bad luck (worst-case) timing:
    updates overlap

- canonical example:
  - read current_balance
    new_balance = current_balance + paycheque
    save new_balance
  - read current_balance
    new_balance = current_balance - withdrawal
    save new_balance

- sequential execution ok: either may come first
- concurrent execution: problematic
  - one transaction disappears

Synchronization Techniques

- mutex (mutual exclusion)
- semaphore
- condition variable
- monitor

} interview question: what's the difference

interesting for historical value

- Software solution:
  - solution for 2 threads
  - won't work on modern hardware (caching)

Naive approaches (fail!)

1) declare I'm using → race condition
2) yielding by turns ("your turn") → deadlock

intrinsically interesting - beautiful

Dekker's algorithm: combine both

reasoning about concurrent processes is hard

Decker's Algorithm

globals: { int turn = self/other
bool flags [2] = { false, false }

P∅:

```
flags [self] = true          // I want
while ( flags[other] ) {     // other wants too
    flag [self] = false      // allow other to proceed
    while ( turn == other ){ wait until my turn
    }
    flag [ self ] = true
}
{
    ⋮   critical section
}
turn = other
flag [ self ] = false
```

assert:
{ flag [self] == true
flag [other] == false

if other tries to get into critical section, they will go into while loop above

if flag [other] is false at this point, other cannot become true without going into their version of the loop

if other wanted too one of the two more will proceed because it is their turn

Hardware Support

- test-and-set instruction
- swap

} "atomic" operations

e.g: test & set
- 1 = has lock
  0 = lock free

test & set to 1 ~~to~~ acquire lock
- if was already 1, someone else has lock and you left lock value unchanged

Semaphore vs Mutex

- For most practical purposes, a mutex is a binary semaphore
- subtle semantic differences
  - mutex is tied to process/thread (ownership)
    - may allow recursive locks
    - may protect against priority inversion
    - may be automatically released if process dies
  - binary semaphore is tied to resource (signaling)
    - process requesting resource waits until the resource is available

- example: protecting a dictionary
  vs process that requires exclusive access to the dictionary

  ⇒ difference is subtle
  (& important only to purists)

Semaphores

~1962/63 Dijkstra
- Dutch Computer Scientist (you may have heard of him)

- OS-level service: "convenient" interface
  - avoid busy waiting (spin locks)

Semaphore S
- initial value
- 2 operations: P(), V()

P() "Proeben"
aka wait() or down()
- decrement (waits if counter = 0)

V() "Verhogen"
aka signal(), up()
- increments
  - if counter was Ø, wakes up waiting process

- mutex: more-or-less binary-value semaphore
  - but purists will cringe when you say this
  - google it
    - understand it, impress your interviewer
  - semantic difference: ownership
    - mutex protects block
    - semaphore is for signalling

Pthread Mutex

```
#include <pthread.h>

int pthread_mutex_init(
        pthread_mutex_t *mutex,
        const pthread_mutexattr_t *attr)
```

- NULL attr : use default values

```
pthread_mutex_lock(
        pthread_mutex_t* mutex)

pthread_mutex_unlock(
        pthread_mutex_t* mutex)
```

- polling : pthread_mutex_trylock

Condition Variables

- waiting for some condition: polling is expensive

```
while !done
    get lock
    if condition
        process
        done = true
    release lock
```

Operations

wait()

signal() Coops.

- Semaphore uses signal
- System: signal(2))

broadcast()

- thundering herd problem

Pthread Condition Variables

```
int pthread_cond_init(
        pthread_cond_t* cond
        const pthread_condattr *attr);
```

(const pthread_condattr *attr → null for default settings)

```
int pthread_cond_signal(
        pthread_cond_t * cond)
```

```
int pthread_cond_wait(
        pthread_cond_t * cond
        pthread_mutex_t * mutex)
```

- pthread_cond_~~wait~~ timedwait
  - wait with timeout

* all waiters must use same mutex
* possible to have spurious returns from pthread_cond_wait(), so code must check

Semaphores (again)

- semaphores may be implemented using
  mutex + counter + condition variable
  - mutex → mutual exclusion
  - condition variable → signaling

Monitors

- language-level implementation

```
Protected {
    stmts
}
```

Java: synchronized

only one process at a time can enter the protected (monitor) block

- process may wait on condition (variable) inside monitor – then other process may enter
  - when condition is true and process regains exclusive access, original process may continue
  - → requires another process to signal

- Can implement monitor in C++ using RAII

```
{
    Monitor m()        ← constructor acquires lock
       ⋮
    m.wait_for(signal_var)
       ⋮
}                      ← destructor called automagically, releases lock
```