

Administrivia

- lab2 revised
  - minor changes
  - -Wall → -O0
- detached vs joinable threads (pthread)
- shorter recap (by popular demand)

mmap

## Our Story So Far (abbreviated)

- interprocess communication
  - pipes (regular <sup>or</sup> named)
  - Posix vs SysV IPC
  - SysV IPC
    - message queue
    - semaphores
    - shared memory - special handling of memory maps in kernel process control blocks
- 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 machine
  - unless you need faster performance

## Our Story So Far (abridged) (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 course hall: more loosely 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 cheat 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  
sizeof (char int char) != char char int  
(C++ class is struct with hidden pointer field)
- array:  $a[i] = *(address\ of\ a[0] + i \times size\ of\ a)$ 
  - $p+i$  vs  $(char *) p + i$ 
    - ↳ add  $i \times size\ of\ (*p)$      $C$  add  $C \times 1$
  - stepping through memory by increments of size of object  
 $a[i] = *(a + i) = *(i + a) = i[a]$
- casting: lets programmer assert that this block of memory corresponds to type  
↳ counter-intuitive
- void \*: untyped address
  - can't do arithmetic because we don't know the size
  - traditionally: used  $char*$  for this purpose  
 $char * p \rightarrow p+1$  because as you would expect

## More About Pointers (cont.)

- Points 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:  $\left\{ \begin{array}{l} \text{parameterizes behavior} \\ \text{abstracts out decision} \end{array} \right.$

e.g.: Sorting:  $\left\{ \begin{array}{l} \text{ptr, size of element,} \\ \text{number of elements,} \\ \text{how to compare element} \end{array} \right.$

most general interface about higher-level computer constructs (e.g. type parameters)

e.g. dynamic dispatch

- vptr

```
struct Animal {  
    void * speak (Animal *)  
};  
    dog  
    wolf  
    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
    - when the critical section is open, the system must decide among waiting processes
    - i.e. something must be chosen
  - 2) progress
    - a process cannot wait indefinitely
      - starvation problem
  - 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 + purchase  
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

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

interesting  
for historical  
context

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:  $\begin{cases} \text{int turn} = \text{self/other} \\ \text{bool flags}[2] = \{ \text{false}, \text{false} \} \end{cases}$

P0:

flags[self] = true // I want

while (flag[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

if flag[other] is false at this point, other cannot become true without going into their critical section

if other wanted too, one of the two processes will proceed at their turn

assert:

flag[self] == true  
flag[other] == false

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

## Hardware Support

- test-and-set instructions
  - swap
- } "atomic" operations

eg: test & set

- 1 = has lock

0 = lock free

test & set to 1 to acquiring 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 0, wakes up waiting process

- mutex: more-or-less binary-value semaphore
  - but purists will cringe when you say that
  - google it
    - understand it, impress your interviewers
  - 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()

- Threading hard problem

## Pthread Condition Variables

```
int pthread_cond_init(  
    pthread_cond_t * cond  
    const pthread_condattr_t * attr);  
    → null for default attrs
```

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

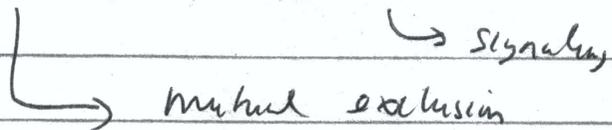
```
int pthread_cond_wait(  
    pthread_cond_t * cond  
    pthread_mutex_t * mutex)
```

- pthread\_cond\_wait ~~wait~~ timed wait
- wait with timeout

- \* all waiters must use same mutex
- x possible to have spurious returns from pthread\_cond\_wait(), so code must check

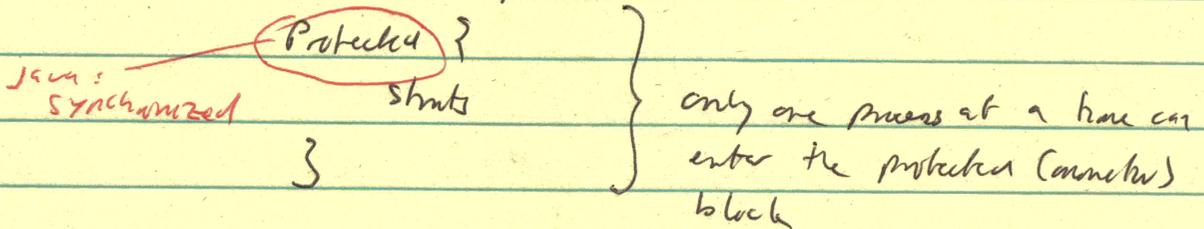
## Semaphores (again)

- semaphores may be implemented using  
mutex + counter + condition variable



# Monitors

- language-level implementation



- 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 automatically, releases lock
```