## Lecture Topics

- a few useful ideas (wrap-up)
- shared memory models
- POSIX Threads (+ semaphores)
  - what and why
  - basics
  - semaphores and their uses
  - dependence and condition variables
- performance reasoning for shared memory threads (a future lab)

## Administrivia

- …

- many pushes for programmers to produce smaller grain sizes
  - pros **[review]**
    - more parallelism (scalable)
    - lots of tasks per processor simplifies load balancing (efficient)

[this part was left as a thought problem]

  - cons (challenges for parallel languages/models)
    - data locality may be hard to express/exploit
    - more variance, so you need dynamic load balancing
      - splitting natural grain size increases variance
      - microarchitectural affects induce higher variance (one cache miss means more in a short task)
    - sometimes harder to exploit (explicit approaches)
      - more packaging for inter-task communication
      - more dependences
      - more scheduling overhead
      - more communication overhead
      - more complex work distribution (need >1 queue)
        - » contention means more overhead
        - » group tasks dynamically to amortize

- scheduling parallel jobs
  - time-sharing or space-sharing?
    - sequential OS scheduler is time-sharing
    - if you have several processors, why pay the overhead?
      - allocate blocks of processors to parallel applications
      - leave a couple free for interactive jobs
  - example: Livermore gang scheduling (space-sharing)
    - three classes of jobs
      - batch: get it done before morning
      - compute: I'll be back after coffee
      - interactive: I'm trying to debug!
    - allocate "gangs" of processors to queued apps
    - try to prioritize queue by class
  - example: co-scheduling
    - Ousterhout's Sprite OS at Berkeley, late 1980s
    - coordinate schedulers across distributed workstations
    - start/stop parallel job threads simultaneously
    - overhead high, and outlook bleak for interactive jobs
- fitting into general systems
  - a Glunix result ca. 1995 (global OS for NOW project at Berkeley)
    - for some apps, a few % skew led to 100× slowdown
    - idle waiting for communicated data
    - slow to barrier, but barrier is more messages…
    - a source of skew? Glunix daemons…
  - implicit scheduling (Andrea Arpaci-Dusseeau's Ph.D.)
    - use message semantics to adapt local schedule
    - dynamically find co-scheduled solution
- scheduling hierarchical parallelism well?
  open problem, I think; see Soumen Chakrabarti's thesis for starter

# Shared Memory Models

- shared memory threads (e.g., Posix)
  - dominate the server market
  - sometimes encapsulated in language (e.g., Java)
  - dynamic number of threads
    - sharing an address space
    - thread to processor mapping varies over time
  - explicit parallelism: each thread has a main function
  - asynchronous sharing: shared memory machines only

- openMP (MP = multiprocess…)
  - combination of data parallelism and shared memory threads
  - viewed for some time as competitor to MPI
    - particularly for large shared memory systems
    - which have essentially vanished
    - OpenMP requires shared memory
  - hybrids with MPI still sometimes used (clusters of SMP/multi-core)
  - dynamic number of threads
    - parallel regions
    - new thread creation
    - thread joining (or waiting)
  - explicit parallelism (several levels possible, chosen by programmer)
  - asynchronous sharing

# POSIX Threads: What and Why

- POSIX = Portable Operating System Interface for uniX
  - threads becoming common in mid-1990s
  - commercial implementations available (e.g., Sun Solaris)
  - no agreement on how to provide
  - first POSIX thread extension standard in 1995
- some problems to solve…
  - non re-entrant routines
  - error model
  - common abstractions
- non-reentrant routines
  - one thread calls routine
  - before first thread uses answer, second thread calls same routine
  - answer changes!
  - common problem in many UNIX interfaces
    - not designed for parallelism
    - used static storage to hold answers
    - rather than paying cost of dynamic allocation
    - or requiring caller to allocate storage
  - mostly fixed with new names ("_r") that add a storage pointer
- UNIX error model (errno) is not re-entrant!
  - errno is static storage!
  - fixed using macro to use thread-specific storage for existing library
  - all new routines use POSIX error model
    - return 0 on success
    - return positive error code on failure

- common abstractions (what to standardize)
  - threads (creation, destruction, cancellation, etc.)
  - mutexes (mix of Linux kernel's spin locks and semaphores)
  - semaphores
  - reader-writer locks
  - condition variables (similar to wait queues, which students will see later)
  - barriers (not defined in Linux kernel)
  - plus how they all interact with
    - scheduling
    - existing abstractions (e.g., signals);
      this part took a while to get right!

- a generic mechanism: attributes
  - certain aspects of a given abstraction must be set at creation time
  - encapsulated in an attributes structure
    - initialize and modify parameters
    - often keep one per "type" of the object in question
    - use to create new instances
  - typically can use NULL attribute for default behavior
  - POSIX has attributes for [all but semaphores…]
    - threads
    - mutexes
    - condition variables
    - reader-writer locks
    - barriers

# Basics of POSIX Threads

- generally, you have to…
  - `#include <pthread.h>`
  - link with `-lpthread`
  - on some systems, you have to do more (not in Linux)
- threads are represented by an opaque type: `pthread_t`
- creation
  - spawn new thread with

```
int pthread_create (pthread_t* new_thread,
                    const pthread_attr_t* attr,
                    void* (*thread_main)(void*), void* arg);
```

  - if successful, new (opaque) thread identifier returned in *thread
  - thread main function takes and returns a pointer
  - `arg` is passed as argument to new thread's main function
- what about the return value?
  - returned to any thread that asks (once the thread has finished)
  - called "joining" with the thread that has ended

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

  - caller blocks until thread terminates
  - returned value is then delivered to join call
- ending the program?
  - termination of main thread immediately ends all others
  - calling exit still ends the program
  - call `pthread_exit` to terminate thread immediately
- cleanup routines called on thread exit (API discussed previously)

```
void pthread_cleanup_push (void (*routine) (void* arg),
                           void* arg);
void pthread_cleanup_pop (int execute);
```

- thread attributes
  - certain aspects of threads must be set at creation time
  - encapsulated in attributes structure: **`pthread_attr_t`**
  - initialize with **`pthread_attr_init`**
  - scheduling (**`pthread_attr_setscope`**)
    - Posix supports threads that run until they yield (**`sched_yield`** or possibly on other library calls)
    - Linux does not (except for real-time threads)
  - detach/join
    - default behavior is joinable thread
      - thread function returns void* value
      - some other thread receives at join point
      - **`pthread_join`** is blocking
    - avoid need to join with **`pthread_attr_setdetachstate`**
    - warning: pthreads supports thread cancellation, but only for joinable threads

- attribute properties for synchronization abstractions
  - pshared (process shared)
    - PTHREAD_PROCESS_PRIVATE:
      only used by threads in one process
    - PTHREAD_PROCESS_SHARED:
      can be used by threads in more than one process
  - How could more than one process see a mutex?
    Shared memory segments allowed between processes (e.g., SysV IPC).
  - Why does it matter?
    - some OS's implement private mutexes purely in library (without entering OS); performance boost
    - shared mutexes must enter OS

# Posix Mutexes [NOTES ONLY]

- pthread_mutex_t
  - static init: = PTHREAD_MUTEX_INITIALIZER;
  - dynamic init

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

- typical use
  - **`pthread_mutex_lock`**
  - **`pthread_mutex_unlock`**

- semantics
  - mutual exclusion
    - non-recursive by default (lock on self-held lock deadlocks)
    - debugging option available (**`set_mutexattr_type`**)
    - recursive option also available (same call)
  - normally puts process to sleep if mutex held (try lock calls do exist)
  - processes are queued in priority order
  - attribute settings specify priority inheritance rules
    - can inherit highest priority of any process waiting on a lock held by current process (**`set_mutexattr_protocol`**)
    - can set minimum priority when holding the lock (protocol **and `set_mutexattr_prioceiling`**)