Overview: Shared Memory Programming

- process-level fork-join
- process vs threads
- Posix threads: creation, example and its performance
- locks for safe sharing of data
  - support by test-and-set and other atomic operations
  - phases and properties
  - implementation; Posix thread API
- semaphores and condition variables
- barriers
- O/S support for shared memory systems
  - user-space vs kernel-space threads
  - threads overheads
  - interrupt handling

Refs: Grama et al Ch 7; Lin & Snyder Ch 6, Wilkinson & Allen Ch 8; Posix Threads Tutorial from LLNL
Unix supports concurrent processing via `fork()`, `{shmget(), semget() etc` these utilities can be (are) used for parallel processing.
(Heavyweight) Unix Processes

- an O/S like Unix is based on the notion of a process
  - the CPU is shared between different processes
- Unix processes are created via `fork()` call
  - the child is an (almost) exact copy of the parent, and has a new, unique process ID
- processes are joined using the system call `wait()`
  - this introduces a synchronization
Unix Fork Example

```
pid = fork();
if (pid == 0) {
    // code to be executed by the child
} else {
    // code to be executed by the parent
}
if (pid == 0)
    exit(0);
else
    wait(0);
```

- the new process is an exact copy of the parent except for the value of `pid`
- often use `exec()` to overwrite the child process with a new executable
- what is a `zombie` process?
- why might fork be viewed as a heavyweight operation?
  - how could this cost be reduced?
Processes and Threads

A process

Two threads
Process/Thread Comparison

- process creation is relatively expensive
  - O/S must set up process control blocks, page tables
  - replicates memory image (although reduced with copy-on-write semantics)
  Communication and synchronization also require O/S calls

- thread creation is relatively cheap
  - the O/S need only set up lightweight control blocks
  Synchronization and variable sharing is (mostly) in user space.

- in processes, all data is private (the MPI model)
  - communication is via shared memory segments, sockets or files
  This may minimize cache coherency traffic!

- in threads, by convention
  - data in the heap/static/global memory segments is shared, stack is private
  communication occurs via shared data structures
Why Threads

- software portability
  - applications can be developed on a serial machine and run on parallel machines without changes (is this really true?)

- latency hiding
  - ability to mask access to memory, I/O or communication by having another thread execute in the meantime (but how quickly can execution switch between threads?)

- scheduling and load balancing
  - for unstructured and dynamic applications (e.g. game playing) loading balancing can be very hard. One option is to create more threads than CPU resources and let the O/S sort out the scheduling

- ease of programming, widespread use
  - due to above, threaded programs are deemed easier to write (or develop incrementally), so there has been widespread acceptance of the POSIX thread API (generally referred to as Pthreads)
main program

pthread create(
    pthread_t *thread_handle,
    const pthread_attr_t *attribute,
    void * (*)(thread_function)(void *),
    void *arg);

pthread join(
    pthread_t thread_handle,
    void **status);

thread_function(arg)
{
    return status;
}

Threads and Threaded Code

- **pthread_self()** provides the (thread) ID of the calling thread
  - what is the analogous MPI call?
  - **pthread_equal(thread1, thread2)** tests IDs

- detached threads
  - threads that will never synchronize via a join operation
  - specified via an attribute
  - why is this useful?

- re-entrant or thread-safe functions are those than can be safely called when another instance has been suspended in the middle of its invocation
  - can you give an example of when a function might not be re-entrant?
Example: Computing Pi

- ratio of area of circle to the square is $\frac{\pi}{4}$
- guess points with domain of square at random
- identify those that are distance less than 1 from origin
- ratio of points in circle to total points is $\approx \frac{\pi}{4}$
Example: Computing Pi

```c
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512

void *compute_pi (void *); 

main() {
  ... 
  pthread_t p_threads[MAX_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  for (i=0; i < num_threads; i++) {
    hits[i] = i;
    pthread_create(&p_threads[i], &attr, compute_pi,
                   (void *) &hits[i]);
  }
  for (i=0; i < num_threads; i++) {
    pthread_join(p_threads[i], NULL);
    total_hits += hits[i];
  }
  ... 
}
```
Example: Computing Pi (cont)

```c
void *compute_pi (void *s) {
    int *hit_pointer = (int *) s, seed = *hit_pointer;
    int i, local_hits = 0;
    double rx, ry;
    for (i = 0; i < sample_points_per_thread; i++) {
        rx = ((double) rand_r(&seed)) / RAND_MAX - 0.5;
        ry = ((double) rand_r(&seed)) / RAND_MAX - 0.5;
        if (rx*rx + ry*ry < 0.25)
            local_hits++;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Notes:

- note the use of the function `rand_r()` (instead of superior random number generators such as `drand48()`)

- what would happen if we added into `*hit_pointer` each time?
Performance of the Pi Program

- 4 processor SGI Origin system using up to 32 threads
- original code: 3.91 fold speedup! (|| efficiency of 0.98!)
  This was on 32 threads (why 32?).
- instead of incrementing local hits, we add to a shared array using strides of 1, 16 and 32
- what performance problem will this introduce, and what does this say about the line size of the secondary cache (which implements cache coherency)?

(Fig 7.2 Grama et al, Intro to Parallel Computing)
Sharing Data

- shared memory provides the ability to share data directly without having to pass messages
- for Unix heavyweight processes, additional shared memory system calls are necessary
  - each process has its own virtual address space
  - shared memory system calls allow the processes to attach to a segment of shared memory
  - use `shmget()` to allocate a shared memory segment, `shmat()` to attach to it (either at a specific virtual address or allow the O/S to return a suitable address)

- for threads, variables on the heap are shared
  - what variables reside on the heap?
  - what variables reside on the stack?
Accessing Shared Data

- concurrent read is no problem
- but what about concurrent write?

<table>
<thead>
<tr>
<th>Code</th>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=x+1</td>
<td>ld r0, addr(x)</td>
<td>ld s0, addr(x)</td>
</tr>
<tr>
<td></td>
<td>add, r1,r0,#1</td>
<td>add, s1,s0,#1</td>
</tr>
<tr>
<td></td>
<td>st addr(x), r1</td>
<td>st addr(x), s1</td>
</tr>
</tbody>
</table>

- result could be \( x+1 \) or \( x+2 \)
- similar problems with access to shared resources like I/O
- critical sections: a mechanism to ensure controlled access to shared resources
  - processes must enter the critical section under mutual exclusion
Locks

- simplest mechanism for providing mutual exclusion
- a lock is a 1-bit variable, a value of
  - 1 indicates a process is in the critical section
  - 0 indicates no process is in the critical section
- at its lowest level a lock is a protocol for coordinating processes/threads,
  - the CPU is not physically prevented from executing those instructions

```c
while (lock == 1) do nothing; /*busy−wait loop*/
lock = 1;                      /*enter critical section*/
// critical section
lock = 0;                      /*exit critical section*/
```
- the above is an incorrect example of a spin-lock, that uses a mechanism called busy waiting
  - what is wrong with the above?
  - what low-level hardware support needs to be provided to address this?
The Assembly

lock:  ld  RO, mem[addr]               ; load word into RO
       cmp  RO, #0                     ; if 0, store 1
       bnz  lock                      ; else, try again
       st  mem[addr], #1

unlock: st  mem[addr], #0             ; store 0 to address

Problem: data race because load-test-store is not atomic!

- processor 0 loads address X, observes 0
- processor 1 loads address X, observes 0
- processor 0 writes 1 to address X
- processor 1 writes 1 to address X

(see http://15418.courses.cs.cmu.edu/spring2015/lecture/synchronization)
Test and Set Lock

- uses the test-and-set instruction:

```plaintext
ts    RO, mem[addr]    ; atomically load mem[addr] into RO
    ; and, if mem[addr] is 0,
    ; set mem[addr] to 1

lock:  ts    RO, mem[addr]    ; load word into RO
    cmp    RO, #0    ; if 0, lock obtained
    bnz    lock

unlock: st  mem[addr], #0    ; store 0 to address
```

What does ‘atomic’ mean in this context?

- \( p \) threads contending for the lock will require \( O(p) \) attempts, thus \( O(p^2) \) time (and energy!)

- we can reduce the number of attempts (but still \( O(p) \)) by only trying the `ts` when we see `mem[addr]` is 0
Common Atomic Operations

- **Test-and-Set:**
  ```
  bool testAndSet(bool *lock) {
    bool lockInit = lock;
    lock = true;
    return lockInit;
  }
  ```

- **Fetch-and-Add (operate):**
  ```
  int fetchAndAdd(int *location, int inc) {
    int value = *location
    *location = value + inc;
    return value;
  }
  ```

- **Compare-and-Swap (on Intel x86, cmpxchg):**
  ```
  bool compareAndSwap(int *p, int old, int new) {
    if (*p != old)
      return false;
    *p = new;
    return true;
  }
  ```
Locks: Phases and Properties

- **phases:**
  - waiting to acquire lock (busy wait or de-schedule and woken later)
  - lock acquisition
  - lock release

- **desirable properties:**
  - fast to acquire in absence of contention
  - generate low interconnect traffic
  - good scalability
  - low storage cost
  - fairness
  - do not degrade overall system (e.g. when more threads than CPUs contend for a lock)
Lock Implementation

- each atomic operation synchronizes the memory system (down to the LLC), and also causes a cache line transfer
- can we do better than the test-and-set lock?
- ticket lock: analogous to protocol of buying seafood at Woolworths
  - initialize: `ticketNumber = screenNumber = 0;`
  - acquire:
    ```
    myTicket = fetchAndAdd(&ticketNumber, 1);
    wait until (myTicket == screenNumber)
    ```
  - release: `screenNumber++;`
- this has a reduced number of atomics, but has a large amount of cache line invalidations due to all threads accessing the ‘scoreboard’ variable `screenNumber`
  It is however fair! Atomic-less version is known as Lamport’s bakery algorithm
- MCS lock: Woolworths analogy: instead of watching a screen, the person just ahead tells you when they are served.
  This restricts the threads accessing their ‘scoreboard’ to just 2
Pthread Lock Routines

- Locks are implemented as mutually exclusive lock variables (mutex variables)
- In Pthreads, the type is `pthread_mutex_t`

```c
#include <pthread.h>

pthread_mutex_t mutex1;
pthread_mutex_init(&mutex1, NULL /*default mutex attributes*/);

pthread_mutex_lock(&mutex1);
/* CRITICAL SECTION */
pthread_mutex_unlock(&mutex1);
```
Deadlock

- Can occur with multiple threads seeking to acquire resources that cannot be shared (circular wait condition)

Pthreads provides `pthread_mutex_trylock()` to help address these circumstances.
Semaphores: Concepts

- devised by Dijkstra in 1968
- a positive integer (including zero) operated on by two indivisible (atomic) operations named \( P \) (passeren, down) and \( V \) (vrijgeven, up)
  - \( P(s) \) waits until \( s \) is greater than 0 and decrements \( s \) by 1 and allows the process to continue
  - \( V(s) \) increments \( s \) by 1 to release any one of the waiting processes (if any)

- in the following, \( s \) is initialized to 1

<table>
<thead>
<tr>
<th>Process 1:</th>
<th>Process 2:</th>
<th>Process 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(s)</td>
<td>P(s)</td>
<td>P(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V(s)</td>
<td>V(s)</td>
<td>V(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
</tbody>
</table>
Semaphores: Implementation

- **Binary semaphores** have only a value of 0 or 1
  - hence can act as a lock
- **Waking up of threads via a V(s)** occurs via an OS signal
- **Posix semaphores API**
  ```c
  int sem_init (sem_t *sem, int pshared, unsigned int value);
  int sem_wait (sem_t *sem); // P(), down
  int sem_trywait (sem_t *sem);
  int sem_post (sem_t *sem); // V(), up
  int sem_getvalue (sem_t *sem, int *value);
  ```
- **pshared**: if non-0, generate a semaphore for usage between processes
- **value** delivers the number of waiting processes/threads as a negative integer, if there are any waiting on this semaphore
- **Posix also provides a semaphores IPC (inter-process communication) via**
  ```c
  semget(), semctl(), semop()
  ```
- **What's bad about semaphores?**
Condition Variables

- Condition variables allow threads to synchronize based on the value of some data, e.g. when a certain value of a (global) variable is reached.
  - Their implementation avoids the polling of the variable.
- The condition check must occur in a critical region (why?), so a mutex $m$ is associated with a condition variable $c$.

- `pthread_cond_signal(c)`: signals that the condition has occurred. This wakes up one of the threads (if any) waiting on, via an OS signal.
- `pthread_cond_broadcast(c)`: wakes up all threads waiting on $c$.
- `pthread_cond_wait(c,m)`: wait for the condition to occur. This blocks the calling thread, releasing $m$ and re-acquiring it on wakeup.

- E.g. thread 0 waits for the condition $x$ becoming 0:

```
// Thread 0
pthread_mutex_lock(&m);
while (x != 0)
    pthread_cond_wait(c, m);
//critical_section
pthread_mutex_unlock(&m);
```

```
// Thread 1
pthread_mutex_lock(&m);
x--;  //waiting
if (x == 0)
    pthread_cond_signal(c);
pthread_mutex_unlock(&m);
```
Monitors in Posix via Mutexes and Condition Variables

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t bufferNotFull;
    pthread_cond_t bufferNotEmpty;
    int count;
    int buf[BUFF_SIZE];
} bbuf_t;

int take(int *item, bbuf_t *B) {
    pthread_mutex_lock(&B->mutex);
    while (B->count == 0) {
        pthread_cond_wait(
            &B->buffer_not_empty,
            &B->mutex);
    }
    B->count--;
    // remove item from buffer
    pthread_mutex_unlock(&B->mutex);
    pthread_cond_signal(
        &B->buffer_not_full);
    return 0;
}

int append(int item, bbuf_t *B) {
    pthread_mutex_lock(&B->mutex);
    while (B->count == BUFF_SIZE) {
        pthread_cond_wait(
            &B->bufferNotFull, &B->mutex);
    }
    B->count++;
    // add item to buffer
    pthread_mutex_unlock(&B->mutex);
    pthread_cond_signal(
        &B->bufferNotEmpty);
    return 0;
}
Barriers

- as with message passing, process/thread synchronization is often required
- Pthreads did not originally provide a native barrier so it had to be coded (e.g. using an atomically-decremented counter)
- now supported with the functions

  ```c
  pthread_barrier_init(pthread_barrier_t *b, ..., unsigned count);
  count is the number of threads that are required to be synchronized
  
  pthread_barrier_destroy(pthread_barrier_t *b);
  
  pthread_barrier_wait(pthread_barrier_t *b);
  
  we can apply message passing barrier algorithms to the shared memory case
Barrier Implementation

- central barrier with \( p \) threads (initialize \( \text{ctr} \) to \( p \), \( \text{sense} \) to 0)
  
  ```
  localSense = sense;
  if (fetchAndAdd(&ctr, -1) == 1) // last to reach
      ctr = p, sense = !sense;
  while (localSense != sense) //spin*
  
  *sense is required for repeated barriers; gets toggled between
  *not scalable: \( p \) atomics per barrier, contention on \( \text{ctr}, \text{sense} \)

- tree-based implementation (\( \text{id} \) is thread id, arrays are initialized to 0):
  
  ```
  arrived[id] = 1; localSense = sense[id];
  for (d = 1; d < p && id % (2*d) == 0; d *= 2)
      while (!arrived[id+d]) /*wait for right node*/;
  if (id == 0) arrived[id] = 0, sense[id] = !localSense;
  for (d = p/2; d >= 1; d /= 2)
      if (id % (2*d) == 0) //release right node
          arrived[id+d] = 0, sense[id+d] = !localSense;
      else if (id % (2*d) == d)
          while (sense[id] == localSense) /*wait for left node*/;
  ```

  No atomics, \( \lg p \) complexity.
  Can pad arrays so that only 2 threads contend on any array element.
Concurrent vs Parallel Processing

- **concurrent processing:** multiple tasks are defined and (sections of each are) allowed to execute in any order
  - does not imply a multiple processor environment
  - e.g. spawn a separate thread to do I/O while the CPU-intensive thread continues to do another operation

- **parallel processing:** the simultaneous execution of concurrent tasks on different CPUs

All parallel processing is concurrent, but NOT all concurrent processing is parallel
O/S Thread Support

- support thread creation and termination (exit, cancel by peer thread), as well as scheduling-related system calls (e.g. yield, park)
- schedule threads fairly, managing transitions from runnable, running, sleeping and stopped states
- deliver a signal to the correct thread
- ensure all OS mechanisms (e.g. access to kernel data structures) are highly scalable (e.g. use scalable locks or if possible lockless methods)
- balance resource (CPU) usage across cores, core groups, and sockets for aggregate system performance
- provide support for binding threads to particular cores (e.g. Linux `get/set_cpu_affinity()`)  
- provide appropriate defaults and system calls for NUMA
- on large systems: avoid ‘jitter’ (e.g. IBM Blue Gene devotes 1 core to OS to avoid this)
- user-level libraries need to be thread-safe and support efficient synchronization (e.g. adaptive spinlocks: yield the thread after the spin-loop times out)
Thread Implementations

three categories of threads
  - pure user-space
  - pure kernel-space
  - mixed

user mode: when a process executes instructions within its program or linked libraries

kernel mode: when the operating system executes instructions on behalf of a user program
  - often as a result of a system call, or an interrupt directed at the program
  - in kernel mode, the program’s thread of execution can access kernel data structures
User-Space Threads

- no kernel involvement in providing a thread
- the kernel has no knowledge of threads and continues to schedule processes only
- thread library selects which thread to run

OK for concurrency, no good for parallel programming
Kernel-Space Threads

- A kernel-level thread is created for each user thread.
- The O/S must manage on a per-thread basis information typically managed on a process basis.
  - In Solaris, threads are called lightweight processes (LWP).
- Potentially poor scaling when too many threads as O/S gets overloaded.
User-space vs Kernel-space

- **user-space advantages**
  - no changes to core O/S
  - no kernel involvement means they may be faster for some applications
  - can create many threads without overloading the system

- **user-space disadvantages**
  - potentially long latency during system service blocking (e.g. 1 thread stalled on I/O request)
  - all threads compete for the same CPU cycles
  - no advantage gained from multiple CPUs

- **mixed scheme**
  - a few user threads mapped to one kernel thread
Thread Overheads

- thread creation may require the library or OS to:
  - allocate new data structures for the thread state:
    - id, status, attributes (e.g. scheduling policy & priority), register contents, stack
  - place newly the created thread into the system’s scheduling queues

Better to create a pool of threads once rather than continuously changing the number of threads

- context switching:
  - swapping threads requires its private context and registers to be stored to kernel memory
  - may occur when waiting for I/O or a lock to be freed as well as when natural time quanta exhausted
  - best if the OS maintains thread affinity: restart thread on the last core it ran on:
    - otherwise may get L1 cache and TLB pollution
  - undesirable to swap out a thread holding any resource (e.g. lock) that other threads may be waiting on

Too many context switches may imply too many threads are being used.
Interrupt Handling

- by default, all cores handle incoming interrupts equally (SMP)
- potentially, interrupts cause high (L1) cache and TLB pollution, as well as delay (switch to kernel context, time to service) threads running on the servicing core

solutions:

- OS can consider handling all on one core (which has no compute-bound threads allocated to it)
- two-level interrupt handling:
  - top-half interrupt handler simply saves any associated data and initiates the bottom-half handler
    e.g. (for a network device) handler simply deposits incoming packets into an appropriate queue
  - ideally, the core with the interrupt’s destination thread services the bottom-half interrupt