Threads (Chapter 11)

- Process -- Program, Memory (text, data, bss, heap, stack), execution
  - stack - directly linked to execution
  - function call frame, on the stack
- CPU -- execution "engine"
- Early computers: one CPU, memory ...
  - Shared computing, multiple processes on one CPU
  - Typical use: round robin, quantum
- Mid 1970s -- Big Iron (cray 1) down to microprocessors (intel 8080)
- Idea at that point: Elephant (one big CPU) vs army of ants (microprocessors)
- Parallel Computing -- army of ants, each ant ran "sequential"
- Now: One "box", multiple CPUs (10s/box), one big memory
  - Possibility of having multiple execution points inside one process
- Concurrency -- multiple execution points inside one process
  - But, not new, used in 1980 (and possibly earlier)
  - At UW: simulate 4096 CPUs on a single CPU?
    - Multiple threads of control in one process
    - Multiple stacks (one for each thread of control)
    - Round-robin the threads
Concurrent, Parallel and Distributed

What is the difference between these ideas?
- Nelson’s definitions -- may not be accepted industry wide, but ...
- Concurrency -- Multiple threads of control in a single process
- Parallel -- A collection of processes, running on a collection of CPUs, cooperating to solve a single problem. CPUs geographically close (e.g. same room or building.)
- Distributed -- A collection of CPUs, most likely geographically distributed, providing services for a variety of uses. e.g. google

Note: Concurrency can be used without multiple CPUs. Parallel and Distributed can’t work on a single CPU.
- Concurrency can be very useful with a GUI, one thread per visual element.

Threads -- concurrency mechanism
- Note: threads can be useful in the parallel and distributed processes.
- Early Threads:
  - User space (OS didn’t know anything about them)
- Now:
  - Thread packages, language features, OS support
Basic Thread Ideas

- Single process, multiple threads (stack and execution)
- Can simplify code for asynchronous code (e.g. GUI)
- Threads can share global or heap memory. (Typically not stack)
  - Process can share memory, but it is more difficult. (ch 15 & 17)
- With multiple CPUs available, "clock" time can be reduced.
- Interactive programs can "spawn CPU intensive tasks on a thread" and come back to user quickly.
- Threads are useful even on a uniprocessor.
  - Simulation example
  - GUI example
  - Issue of blocked vs running threads
    - With OS support, blocked threads don’t block other threads

Thread consists of:
- stack (local variables in functions, call sequence)
- CPU registers (PC, status, ....)
- "Thread Local Storage" -- Global to thread, Local in process
  - errno -- two threads calling a system call at the same time
- Shares the rest of the process ... pid, CWD, files, heap, ...
PThreads -- POSIX threads

A thread library defined by the POSIX group (POSIX.1-2001)
- Various implementations have been done.
- Need a thread ID ... but may be a struct ... so
  - int pthread_equal (pthread_t tid1, pthread_t tid2);
    - compare twothreads, return non-zero => equal, 0 is not equal
  - pthread_t pthread_self(void);
    - Gets the current thread id.
- Thread Creation
  - Program after execxx() starts as a single thread program.
  - Threaded program then starts threads
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine) (void *), void *arg);
    - thread -- pointer to a pthread_t variable
    - attr -- May be NULL (more later)
    - start_routine -- pointer to thread's "main" function
    - arg -- pointer passed to the start routine
  - See pth-id.c: a program to print the "thread id"
Thread Termination & Joining

- Any thread calling exit(3) or _exit(2) exits the process and kills all threads
- A default action signal that terminates will terminate the entire process
- Single thread "exit" terminates only calling thread
  - void pthread_exit (void *rval_ptr)
    - may return a pointer
  - "start routine" can just return and return value is pthread_exit parameter
- "Joining a thread"
  - Similar to a wait(), but for a thread
  - int pthread_join(pthread_t thread, void **retval);
    - Calling thread blocks if "thread" is still running
    - After "thread" returns, calls pthread_exit() or is canceled, returns from join
      - PTHREAD_CANCELED is a possible return value
  - return value is 0 for success, non-zero is an error number
    - EDEADLK - mutual joins or join calling thread
    - EINVAL - thread already joined or process waiting to join
    - ESRCH - no thread with that ID
  - Do not return pointers to local variables ... "automatic variable mis-use"
  - Joined thread reclaims the resources of the thread
Other Calls

- `int pthread_cancel(pthread_t thread);`
  - Causes thread to terminate as if it did `pthread_exit(PTHREAD_CANCELED)`

- `void pthread_cleanup_push(void (*routine)(void *), void *arg);`
  - Schedules a thread to be run at `pthread_exit` or `pthread_cancel` time
  - Adds "routine" to a stack of routines for the calling thread

- `void pthread_cleanup_pop(int execute);`
  - Pops the top routine off the thread’s cleanup stack, not run if `execute` is 0

- `int pthread_detach(pthread_t thread);`
  - Sets the thread to be "un-joinable" and automatically reclaims resources at thread exit
  - `pthread_join()` on this thread will return an error
Thread Synchronization

Race Conditions
- Multiple processes working together
- Results depend on the order of the processes running. (Example race.c)
- Example -- character at a time output
- Accessing a file, "lock file"

Race conditions happen even easier in threads
- Consider pth-race.c
- What happens?
- Why does that happen?
  - What about "read only" variables?
- How can you fix it?
  - Critical section -- section of code that must happen "atomically" -- no interruption of the process
  - Software -- Peterson’s solution
    - Turn based approach -- but works only for two threads
  - Hardware assist approaches: Mutex -- Mutual Exclusion
    ```c
    int mutex = 0;
    while (test_and_set(&mutex) == 1) /* spin */;
    critical-section;
    mutex = 0;
    ```
- Problem -- busy wait.
- Solution -- have the OS block the thread until it can enter the critical section.
Mutex -- solution for mutual access to a shared variable

Mutex -- a lock to block access to a critical section
- one thread in the critical section at a time
- all access to shared variable covered by a mutex
- Pthreads -- need to initialize it:
  ```c
  int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);
  ```
  - `pthread_mutexattr_t *` may be NULL for standard attributes
- When done, the mutex may be destroyed
  ```c
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```
  - no need to destroy if calling exit
- Entry to critical section -- lock, block if held
  ```c
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  ```
  - return value 0 if successful, should check
- Non-blocking try to lock
  ```c
  int pthread_mutex_trylock(pthread_mutex_t *mutex);
  ```
  - Error if locked, `errno` is EBUSY
- Exit to critical section -- unlock and let others in
  ```c
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```
Static initialization of a mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

Typical code outline:
if (pthread_mutex_lock(&mutexvar)) {
    /* Error condition */
} else {
    /* critical section */
    if (pthread_mutex_unlock(&mutexvar)) {
        /* error condition */
    }
}

Critical section should be
- as short as possible
- have no loops unless absolutely necessary
- never block in the critical section

See pth-mutex.c
Mutex issues

- Deadlock -- circular waiting
  - ush: child blocked writing to pipe, parent waiting on child
  - mutex: tries to lock a mutex already held
- multiple mutexes:
  - A holds M1, tries for M2 (blocked)
  - B holds M2, tries for M1 (blocked)
- longer chains of holding and waiting with circular wait are possible

Avoid deadlocks with a mutex lock ordering
- can’t order? Use pthread_mutex_trylock and don’t block
Problem: Readers and Writers

- Shared Data Structure ... e.g. Balanced Binary Tree
- Reader threads -- doing lookups in the tree
- Writer threads -- doing inserts into the tree

Problems?
- race without mutex protection
- problem here?
  - two readers can share at the same time with no race
  - writers need exclusive access, can’t share with readers

Solution?
- A new kind of a lock: A read-write lock
- Two kinds of locking:
  - read_lock -- I promise to only read
  - write_lock -- I will modify data, need exclusive access
- read_lock: allows locking if lock is read locked
  - blocks if write lock held
  - typically blocks if thread waiting to write lock
- write_lock: blocks until all locks (both read and write) are unlocked
Pthread reader-writer locks

- reader/writer lock initialization
  ```c
  int pthread_rwlock_init(pthread_rwlock_t * lock, const pthread_rwlockattr_t * attr);
  pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
  ```

- reader/writer lock destruction
  ```c
  int pthread_rwlock_destroy(pthread_rwlock_t *lock);
  ```

- reader/writer locks/unlock routines
  ```c
  int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
  int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
  int pthread_rwlock_unlock(pthread_rwlock_t *lock);
  int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
  int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
  ```

- 0 return if OK, error number on failure
- Read man pages for full information
Bounded Buffer Problem

- multiple "producers", produce item, add it to shared queue
- multiple "consumers", grab an item from the shared queue and "consume it"
- shared queue has a shared size N
- How does this get synchronized?

Semaphores

- A more robust tool (Not part of P-Threads)
- Core implementation: A semaphore is an integer variable $S$ with atomic operations
  - $\text{wait}(S) \{ \text{while} (S \leq 0) /* wait */; S--; \}$
  - $\text{signal}(S) \{ S++; \}$
- Original names by E.W. Dijkstra were $P()$ and $V()$ ... proberen and verhogen
- Kinds of semaphores: Binary (0,1) and Counting (0,1,2,3,...,n)
- Issue: busy waiting
- Can provide Mutex (e.g. binary semaphore is essentially a mutex)
- Can provide other synchronization solutions, wait for another process
  - $t1: S1; \quad t2: \text{wait}(S);$  
    - $\text{signal}(S); \quad S2;$
Semaphore implementations

- Issue of busy waiting ...
- Instead of busy waiting, a process could block (give up the CPU)
- Consider the following implementation, each function "atomic"

```c
typedef struct { int value; struct process *list; } semaphore;
void wait (semaphore *S)
{  S->value--;
   if (S->value < 0) {
      add process to S->list;
      block();
   }
}
void signal (semaphore *S)
{  S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```
Solution of bounded buffer problem:

Queue of size N:
Semaphore empty = N, Semaphore full = 0, Semaphore mutex = 1

producer:
  while true
    produce item
    wait(&empty)
    wait(&mutex)
    Add item to Queue
    signal(&mutex)
    signal(&empty)
    signal(&full)

Consumer:
  while true
    wait(&full)
    wait(&mutex)
    delete item from Queue
    signal(&mutex)
    signal(&empty)
    consume item
Issues with semaphores

P0         P1
wait(S)    wait(Q)
wait(Q)    wait(S)
...
signal(S)  signal(Q)
signal(Q)  signal(S)

□ Issue?
□ Deadlock ... P0 gets S and waits on Q, P1 gets Q and waits on S
□ Easy to get with semaphores if not careful
□ Programs have to be written correctly
□ Programmers have to write correct synchronization code
□ Consider:
  wait(S);
  ... critical section ...
  wait(S);
□ To help "fix" these issues, language designers have added constructs to languages
Monitors

- Monitors -- a object based synchronization construct
  - monitor name {
    // shared variable definitions
    function f1(args) { .... with access to shared vars and arguments only.... }  
    function f2(args) { .... with access to shared vars and arguments only.... }  
  
  
  initialization (...) {....}
  }

- functions in the monitor all run with mutual exclusion
- shared vars may be accessed only by functions in the monitor
- programmer does not need to code mutual exclusion
- needs something more for full synchronization, e.g. bounded buffer
  - need to wait in a method for some condition to be true
  - don’t want to block other threads from entering
condition variables -- "condition x;"

- Operations:
  - `x.wait()` -- blocks the process in the monitor
  - `x.signal()` -- restarts one process blocked
    - no blocked processes? no-op
  - `x.broadcast()` -- restarts all processes blocked

- Issues:
  - call to `x.signal()` -- who runs?
    - caller waits
    - signaled waits
  - compromise for Concurrent Pascal: signaler must exit

- A number of languages have implemented monitors

- Path Pascal -- a slightly different approach
  - Object, functions, specification of order/number of operations
  - `path 1:(a,b) , n:(a;b) end`
    - `1:(a,b)` -- a and b need mutual exclusion
    - `n:(a;b)` -- an a must run before b, at most n more as than bs
Condition variables and Pthreads

Pthreads have condition variables ... but monitors!

Functions for condition variables in Pthreads

int pthread_cond_init(pthread_cond_t * restrict cond, const pthread_condattr_t * restrict attr);

or declaration init: pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond); // May be implemented as broadcast!
int pthread_cond_wait(pthread_cond_t * restrict cond, pthread_mutex_t * restrict mutex);
int pthread_cond_timedwait(pthread_cond_t * restrict cond, pthread_mutex_t * restrict mutex,
const struct timespec * restrict abstime);

Use?

Must be used in conjunction with mutexes

pthread_mutex_lock(&mutex)
while (something requires us to wait) {
    pthread_cond_wait (&condvar,&mutex);
}
critical section
pthread_mutex_unlock(&mutex)
Condition variables and Pthreads (page 2)

- Wait code again
  ```c
  pthread_mutex_lock(&mutex)
  while (something requires us to wait) {
    pthread_cond_wait (&condvar,&mutex);
  }
  critical section
  pthread_mutex_unlock(&mutex)
  ```
- While needed due to possibility signal is implemented as broadcast
- Signal/Broadcast code
  ```c
  pthread_mutex_lock(&mutex)
  critical section
  pthread_cond_signal(&condvar)
  pthread_mutex_unlock(&mutex)
  ```
- Want to signal and exit (like required for in some languages)
- Best implementations
  - "implement a monitor in C, mymonitor.h/.c" (glo.c,glo.h,pth-mon.c)
  - Other parts of program just call the functions
Dining philosophers -- another classic synchronization problem

- 5 philosophers
- jobs: eat & think (over and over, ....)
- seated at a round table
- one plate in front of each philosopher
- one fork in between plates
- philosopher needs two forks to eat

How do you synchronize access to the forks?
- status [5] -- THINKING, HUNGRY, EATING
- condition [f] -- to wait for all forks
- check_forks(p) -- if hungry(p) and +/- not eating then set status to EATING, signal(p)
- pickupForks(p) -- monitor routine
  - status[p] = HUNGRY, check_forks(p), if status[p] != EATING, wait(p)
- putdownForks(p) -- another monitor routine
  - status[p] = THINKING, check_forks(-), check_forks(+)
Banking Example

- Customer Accounts and transactions (customer/account the thread)
  - Deposits -- simple
  - Check/Withdrawls -- simple
  - Transfers --- ?
    - Synchronize from and to accounts
    - Rendezvous
      - Implemented as part of several languages, Ada is one of them
      - How to implement in Pthreads?
Barriers -- multi-thread synchronization

All thread need to wait for slowest thread?
- Use a "barrier" -- all threads have to stop at same time until all are stopped
- Implementation with a condition variable (and mutex)?
- Pthread version:
  ```c
  int pthread_barrier_init(pthread_barrier_t * restrict barrier, const
  pthread_barrierattr_t * restrict attr,
  unsigned int count);
  ```
  - 0 return -> successful
  ```c
  int pthread_barrier_destroy(pthread_barrier_t *barrier);
  ```
  - 0 return -> successful
  ```c
  int pthread_barrier_wait(pthread_barrier_t *barrier);
  ```
  - 0 => successful for all but one, one gets PTHREAD_BARRIER_SERIAL_THREAD
  "thread may be used to update shared data."

Parallel version across multiple machines
- Issues of speed across a barrier -- as few barriers as possible!
- Tree -- across multiple machines / or even on a GPU ...
- Sequential is BAD
Amdahl’s Law

- Sequential part of a problem dictates limit on speed up
- \( p \) -- fraction of work that can be parallelized
- \( T = (1-p)T + pT \) -- total time
- Now add parallelization ...
  - \( s \) is a speed up factor on the parallel code

- \( T' = (1-p)T + pT/s \)
Thread Attributes -- for use at thread creation time

```c
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Attributes you can get and set

- `pthread_attr_getdetachstate(3)` thread detach state
- `pthread_attr_getguardsize(3)` thread guard size
- `pthread_attr_getinheritsched(3)` inherit scheduler attribute
- `pthread_attr_getschedparam(3)` thread scheduling parameter
- `pthread_attr_getschedpolicy(3)` thread scheduling policy
- `pthread_attr_getscope(3)` thread contention scope
- `pthread_attr_getstack(3)` thread stack
- `pthread_attr_getstacksize(3)` thread stack size
- `pthread_attr_getstackaddr(3)` thread stack address

Mutex & read/write lock attributes

- control how locks work -- not covered here
Reentrancy (aka thread safe)
- multiple threads can call same function at the same time
- is it safe to do it?
  - Yes -> thread safe!
- What would make it un-safe?
  - return a pointer to a single static struct
  - second call changes static struct
- System functions? Are they thread safe?
  - Not all -- see Figure 12.9 -- not guaranteed to be thread safe
  - Things like getpwent(), getgrgid() ....
- How do you use them?
  - critical section with mutexes()
- How about your code? Can it be thread safe?
- How about access to errno?
Thread Control (page 3)

Thread specific data
- can’t use thread ID and an array ...
- local variables in thread_main() are "thread specific" .... but
  - errno needs to have one variable per thread
  - (GCC did some compiler tricks ... but not portable)
- Idea of a "key" -- for accessing the data
  - int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
  - int pthread_key_delete(pthread_key_t key);
    - Creates and destroys a key
    - Only want to do this once, not for every thread
    - destructor called at thread termination time if not NULL
  - int pthread_once(pthread_once_t *once_control, void (*init_routine)(void *));
    - allows first thread to call init_routine and not others
- Access to thread specific data:
  - void *pthread_getspecific(pthread_key_t key);
  - int pthread_setspecific(pthread_key_t key, const void *value);
  - set first, then get. get before set gets NULL
Errno? no longer a variable but a define to call a function

```
#define errno (*__errno())
```

Similar to windows GetLastErrorCode!

Cancel Options

- Can control how a thread can be canceled -- read section 12.7

Final two issues: signals and forks

- Each thread has its own signal mask (pthread_sigmask())
  - sigaction is still for the entire process
- Signals are delivered to a single thread
  - hardware issue -> delivered to thread that caused it
- No thread caused the signal -> delivered to an arbitrary thread!
- Control can be had with the per thread signal mask and sigwait(2)
  - e.g. one thread can catch all the generic signals
- Note: read about Linux, signals and threads at the end of 12.8
Forking with threads:

- fork(2) - creates a new process with ONE thread running
  - What about all the mutexes, r/w locks and cond variables?
- fork()/exec() -> no problems ... memory image destroyed
- fork() and continue execution
  - Don’t use locks/threads ... no problem
  - Start threads, using locks ... big problem!
    - If you do this ... read how to do it
    - can use pthread_atfork() to help clean up locks.

I/O in threads

- reads/writes -- use "file pointer"
- can interfere with each other
- Solution?
  - ssize_t pread(int d, void *buf, size_t nbytes, off_t offset);
  - ssize_t pwrite(int d, const void *buf, size_t nbytes, off_t offset);