Part of Chapter 4 and Chapter 5 from the book.

Thread Libraries: PThreads, Windows, Java

Pthreads (POSIX -- UNIX)
- POSIX Pthreads -- as specification
- Each OS provides their own implementation
  - Most UNIX systems provide Pthreads
  - Windows does not, 3rd party implementations
- Large specification ...
- Core routines:
  - `pthread_attr_init(&attr)` - initialize an attribute variable
  - `pthread_create(&tid, &attr, funcname, voidptr)` -- ID is stored in tid.
    - void * funcname (void *argfromvoidptr) -- arg from pthread_create
    - `pthread_exit(void *)`
  - `pthread_join(tid, void **ptr)` -- ptr to place to store void * to exit
- Book has simple example of starting a pthread and then joining it.
- Synchronization in chapter 5.
Windows Threads - Similar concept, different API

Core routines:

- `ThreadHandle = CreateThread (security, stacksize, funcname, &parameter, flags, &id)`
- `Function: DWORD WINAPI funcname (LPVOID parameter)`
  - function just returns, no "exit" call like pthreads
- "parent"
  - `WaitForSingleObject(ThreadHandle, INFINITE)`
  - `CloseHandle(ThreadHandle)`
  - or: `WaitForMultipleObjects(....)`

Java Threads -- actually tends to use (at the lower levels) one of the above

- `public interface Runnable { public abstract void run (); }`
- The `run()` method is run by the thread
- Runnable does not have the thread ... there is another Thread class
  - `class usesthreads implements Runnable { ... public usesthreads(args...) {constructor} public void run() { code for thread } }`
- `Thread thrd = new Thread (new usesthreads(...))`
- `thrd.join()`
Other methods ...

Chapel -- parallel features that generate threads, some short running, some long running
- foreach i in 1 .. 1000 do A[i] = i;
- coforall l in Locales { on L { callfunc(); } }
- coforall i in 1 .. 100 { callfunc(); }

OpenMpc -- Another library used by compiler directives

Apple -- Grand Central Dispatch
- ^{ printf ("I am a block") } 
- compiler schedules blocks specified this way
Race Condition ... bounded buffer / consumer producer
Producer: // another version
while (true) {
    item = produce_item()
    while (counter == BUFFER_SIZE) /* do nothing */;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
Consumer:
while (true) {
    while (counter == 0) /* do nothing */;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    consume_item(item)
}
□ problem with concurrent execution, possible incorrect value of counter
□ Race condition: results depend on order of execution of the "processes"
Critical-Section problem

- Want concurrent code to execute correctly all the time, e.g. no race conditions
- Critical Section: a section of code that may be changing shared variables
- Protocol so that only one process may be in their critical section at the same time
- A solution must satisfy four requirements
  - Mutual exclusion -- only one process is in its critical section at any time
  - Progress -- only processes wanting in their critical sections participate
  - Bounded waiting -- bounded time a process must wait for entry into its critical region
- Switching between processes to provide concurrency
  - Preemptive kernels -- CPU is taken away from a thread
  - Non-preemptive kernels -- thread may have CPU as long as needed, yield() gives up CPU
- Non-preemptive are free from race conditions in a single CPU system
- Multi-CPU and preemptive kernels may have race conditions
- Most kernels are preemptive or have multi-CPU (more recently)
Peterson’s Solution

- Pure software solution -- Taking turns

- `int turn;`
- `boolean flag[2];`

```java
    do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j) /*spin*/
            -- critical section
        flag[i] = false;
            -- remaining section
    }
```

- This code is for P_i (P sub i) -- saying I want in to the CR
- Does this meet the three requirements?
Hardware synchronization

- Locks -- the basis for synchronization
- Solutions -- (will be used by blitz!)
  - Turn interrupts off -> no preemption
    - Doesn’t work well for multi-CPU systems
  - Test-and-set instruction (atomically done)
    - \( \text{rv} = \text{variable}; \text{variable} = \text{some\_value}; \text{return} \ \text{rv} \)
    - Often values are 0 and 1 (or true and false)
  - Use:
    ```
    do {
      while (test_and_set(&lock)) /* do nothing */;
      -- critical section
      lock = false;
      -- remainder section
    } while (true);
    ```
Compare-and-swap

```c
int compare_and_swap (int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Use:

```c
do {
    while (compare_and_swap(&lock, 0, 1) != 0) /* do nothing */
    -- critical section
    lock = 0;
    -- remaining section
}
```
Mutex - (short for mutual exclusion)

- based on a lock, acquiring the lock, releasing the lock
- simple version: 
  ```
  acquire() {
      while (!available);
      available = false;
  }
  
  release() {
      available = true;
  }
  ```
- Both done atomically (acquire tests and sets atomically)
- Issue: busy waiting (e.g. spin lock)
  - can be useful in situations where spin is very short
- Issue 2: This solves the mutual exclusion problem, not the other synchronization issues
Semaphores

- A more robust tool
- Core implementation: A semaphore is an integer variable S with atomic operations
  ```c
  wait(S) {
    while (S <= 0) /* wait */;
    S--;
  }
  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-n)
- Issue: busy waiting
- Can provide Mutex
- Can provide other synchronization solutions:
  ```c
  p1: S1; p2: wait(S);
  signal(S); 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);
    }
}
```
Deadlock and starvation

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
□ More on deadlock later

□ Another Issue?

□ starvation ... queue management
□ which can lead to priority inversion
□ a lower priority process holding a mutex stopping a higher priority process
□ read book about the mars pathfinder
□ Solution: priority inheritance
□ lower priority processes run at highest priority of any process on the queue
Bounded-buffer problem

shared variables:

- int n;
- item buffer[n];
- semaphore mutex=1;
- semaphore empty=n;
- semaphore full=0;

Producer

do {
  item = produce_item();
  wait(empty);
  wait(mutex);
  wait(mutex);
  -- add item to buffer
  signal(mutex);
  signal(mutex);
  signal(full);
} while (true);
Consumer
  do {
    wait(full);
    wait(mutex);
    item = get item from buffer
    signal(mutex);
    signal(empty);
  } while (true);

Readers/Writers problem
  □ DB readers and writers
  □ Any number of readers, one writer
  □ Readers should not block writers indefinitely
  □ Solution in book blocks writers indefinitely

Dining Philosophers Problem
  □ Five philosophers -- eat and think
  □ Eating requires two utensils (fork, chopstick)
  □ Picks up only one at a time, shares utensils with neighbors (ignore sanitary concerns)
  □ Read book and implement in assignment 2
Monitors

Semaphores -- with OS support to block processes, a solution for synchronization

- Can have problems -- not so easy to detect
- Programs have to be written correctly
- Programmers have to write correct synchronization code

Consider:

```c
wait(mutex);
...
critical section ...
wait(mutex);
```

To help "fix" these issues, language designers have added constructs to languages

- Monitors -- a object based synchronization construct

  ```c
  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

condition variables -- "condition x;"

Operations:

x.wait() -- blocks the process in the monitor

x.signal() -- restarts one process blocked

no blocked processes? no-op

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
Monitors (page 3)

- Read 5.8.2 -- dining-philosophers solution using monitors
- One more possible function in a monitor
  - \texttt{x.broadcast} -- add all blocked processes to a queue to run next
- Most monitors are actually implemented by semaphores
  - Semaphores for Mutex is "easy"
  - Semaphores for condition variables?
    - Not a direct replacement
    - No signal call if no process is blocked
      - requires extra shared variables

PThread synchronization
- \texttt{pthread_mutex_t}, \texttt{pthread_mutex_init()}, \texttt{pthread_mutex_lock()}, \texttt{pthread_mutex_unlock()}
- condition variables are also implemented
- POSIX defines a set of semaphore ... not part of Pthreads
  - \texttt{man sem}

Read all sections in Chapter 5