CIS520 – Operating Systems
Handout 3
Thread Creation, Manipulation and Synchronization

- We first must postulate a thread creation and manipulation interface. Will use the one in Nachos:

```
class Thread {
  public:
    Thread(char* debugName);
    ~Thread();
    void Fork(void (*func)(int), int arg);
    void Yield();
    void Finish();
}
```

- The `Thread` constructor creates a new thread. It allocates a data structure with space for the TCB.
- To actually start the thread running, must tell it what function to start running when it runs. The `Fork` method gives it the function and a parameter to the function.
- What does `Fork` do? It first allocates a stack for the thread. It then sets up the TCB so that when the thread starts running, it will invoke the function and pass it the correct parameter. It then puts the thread on a run queue someplace. `Fork` then returns, and the thread that called `Fork` continues.
- How does OS set up TCB so that the thread starts running at the function? First, it sets the stack pointer in the TCB to the stack. Then, it sets the PC in the TCB to be the first instruction in the function. Then, it sets the register in the TCB holding the first parameter to the parameter. When the thread system restores the state from the TCB, the function will magically start to run.
- The system maintains a queue of runnable threads. Whenever a processor becomes idle, the thread scheduler grabs a thread off of the run queue and runs the thread.
- Conceptually, threads execute concurrently. This is the best way to reason about the behavior of threads. But in practice, the OS only has a finite number of processors, and it can't run all of the runnable threads at once. So, must multiplex the runnable threads on the finite number of processors.
- Let's do a few thread examples. First example: two threads that increment a variable.

```
int a = 0;
void sum(int p) {
  a++;
  printf("%d : a = %d\n", p, a);
}
void main() {
  Thread *t = new Thread("child");
  t->Fork(sum, 1);
  sum(0);
}
```

- The two calls to `sum` run concurrently. What are the possible results of the program? To understand this fully, we must break the `sum` subroutine up into its primitive components.
- `sum` first reads the value of `a` into a register. It then increments the register, then stores the contents of the register back into `a`. It then reads the values of of the control string, `p` and `a` into the registers that it uses to pass arguments to the `printf` routine. It then calls `printf`, which prints out the data.
- The best way to understand the instruction sequence is to look at the generated assembly language (cleaned up just a bit). You can have the compiler generate assembly code instead of object code by giving it the -S flag. It will put the generated assembly in the same file name as the .c or .cc file, but with a .s suffix.

```
la      a, %r0
ld      [%r0],%r1
add     %r1,1,%r1
st      %r1,[%r0]

ld      [%r0], %o3 ! parameters are passed starting with %o0
mov     %o0, %o1
la      .L17, %o0
call    printf
```

- So when execute concurrently, the result depends on how the instructions interleave. What are possible results?

```
0 : 1                                   0 : 1
1 : 2                                   1 : 1

1 : 2                                   1 : 1
0 : 1                                   0 : 1

1 : 1                                   0 : 2
0 : 2                                   1 : 2

0 : 2                                   1 : 2
1 : 1                                   0 : 2
```

  So the results are nondeterministic - you may get different results when you run the program more than once. So, it can be very difficult to reproduce bugs. Nondeterministic execution is one of the things that makes writing parallel programs much more difficult than writing serial programs.
- Chances are, the programmer is not happy with all of the possible results listed above. Probably wanted the value of `a` to be 2 after both threads finish. To achieve this, must make the increment operation atomic. That is, must prevent the interleaving of the instructions in a way that would interfere with the additions.
- Concept of atomic operation. An atomic operation is one that executes without any interference from other operations - in other words, it executes as one unit. Typically build complex atomic operations up out of sequences of primitive operations. In our case the primitive operations are the individual machine instructions.
- More formally, if several atomic operations execute, the final result is guaranteed to be the same as if the operations executed in some serial order.
- In our case above, build an increment operation up out of loads, stores and add machine instructions. Want the increment operation to be atomic.
- Use synchronization operations to make code sequences atomic. First synchronization abstraction: semaphores. A semaphore is, conceptually, a counter that support two atomic operations, P and V. Here is the Semaphore interface from Nachos:

```
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    void P();
    void V();
}
```

- Here is what the operations do:
  - Semphore(name, count) : creates a semaphore and initializes the counter to count.
  - P() : Atomically waits until the counter is greater than 0, then decrements the counter and returns.
  - V() : Atomically increments the counter.
- Here is how we can use the semaphore to make the `sum` example work:

```
int a = 0;
Semaphore *s;
void sum(int p) {
  int t;
  s->P();
  a++;
  t = a;
  s->V();
  printf("%d : a = %d\n", p, t);
}
void main() {
  Thread *t = new Thread("child");
  s = new Semaphore("s", 1);
  t->Fork(sum, 1);
  sum(0);
}
```

- Are using semaphores here to implement a mutual exclusion mechanism. The idea behind mutual exclusion is that only one thread at a time should be allowed to do something. In this case, only one thread should access `a`. Use mutual exclusion to make operations atomic. The code that performs the atomic operation is called a critical section.
- Semaphores do much more than mutual exclusion. Can also be used to synchronize producer/consumer programs. The idea is that the producer is generating data and the consumer is consuming data. So a Unix pipe has a producer and a consumer. You can also think of a person typing at a keyboard as a producer and the shell program reading the characters as a consumer.
- Here is the synchronization problem: make sure that the consumer does not get ahead of the producer. But, would like the producer to be able to produce without waiting for the consumer to consume. Can use semaphores to do this. Here is how it works:

```
Semaphore *s;
void consumer(int dummy) {
  while (1) {
    s->P();
    consume the next unit of data
  }
}
void producer(int dummy) {
  while (1) {
    produce the next unit of data
```

```
    s->V();
  }
}
void main() {
  s = new Semaphore("s", 0);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

In some sense the semaphore is an abstraction of the collection of data.

- In the real world, pragmatics intrude. If we let the producer run forever and never run the consumer, we have to store all of the produced data somewhere. But no machine has an infinite amount of storage. So, we want to let the producer to get ahead of the consumer if it can, but only a given amount ahead. We need to implement a bounded buffer which can hold only N items. If the bounded buffer is full, the producer must wait before it can put any more data in.

```
Semaphore *full;
Semaphore *empty;
void consumer(int dummy) {
  while (1) {
    full->P();
    consume the next unit of data
    empty->V();
  }
}
void producer(int dummy) {
  while (1) {
    empty->P();
    produce the next unit of data
    full->V();
  }
}
void main() {
  empty = new Semaphore("empty", N);
  full = new Semaphore("full", 0);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

In assignment 1 you will implement a producer and consumer for the console (a device that reads and writes characters from and to the system console). You will probably use semaphores to make sure you don't try to read a character before it is typed.

- Semaphores are one synchronization abstraction. There is another called locks and condition variables.
- Locks are an abstraction specifically for mutual exclusion only. Here is the Nachos lock interface:

```
class Lock {
  public:
    Lock(char* debugName);              // initialize lock to be FREE
    ~Lock();                            // deallocate lock
    void Acquire(); // these are the only operations on a lock
```

```
    void Release(); // they are both *atomic*
}
```

- A lock can be in one of two states: locked and unlocked. Semantics of lock operations:
  - Lock(name) : creates a lock that starts out in the unlocked state.
  - Acquire() : Atomically waits until the lock state is unlocked, then sets the lock state to locked.
  - Release() : Atomically changes the lock state to unlocked from locked.

  In assignment 1 you will implement locks in Nachos on top of semaphores.
- What are requirements for a locking implementation?
  - Only one thread can acquire lock at a time. (safety)
  - If multiple threads try to acquire an unlocked lock, one of the threads will get it. (liveness)
  - All unlocks complete in finite time. (liveness)
- What are desirable properties for a locking implementation?
  - Efficiency: take up as little resources as possible.
  - Fairness: threads acquire lock in the order they ask for it. Are also weaker forms of fairness.
  - Simple to use.
- When use locks, typically associate a lock with pieces of data that multiple threads access. When one thread wants to access a piece of data, it first acquires the lock. It then performs the access, then unlocks the lock. So, the lock allows threads to perform complicated atomic operations on each piece of data.
- Can you implement unbounded buffer only using locks? There is a problem - if the consumer wants to consume a piece of data before the producer produces the data, it must wait. But locks do not allow the consumer to wait until the producer produces the data. So, consumer must loop until the data is ready. This is bad because it wastes CPU resources.
- There is another synchronization abstraction called condition variables just for this kind of situation. Here is the Nachos interface:

```
class Condition {
  public:
    Condition(char* debugName);
    ~Condition();
    void Wait(Lock *conditionLock);
    void Signal(Lock *conditionLock);
    void Broadcast(Lock *conditionLock);
}
```

- Semantics of condition variable operations:
  - Condition(name) : creates a condition variable.
  - Wait(Lock *l) : Atomically releases the lock and waits. When Wait returns the lock will have been reacquired.
  - Signal(Lock *l) : Enables one of the waiting threads to run. When Signal returns the lock is still acquired.
  - Broadcast(Lock *l) : Enables all of the waiting threads to run. When Broadcast returns the lock is still acquired.

  All locks must be the same. In assignment 1 you will implement condition variables in Nachos on top of semaphores.

- Typically, you associate a lock and a condition variable with a data structure. Before the program performs an operation on the data structure, it acquires the lock. If it has to wait before it can perform the operation, it uses the condition variable to wait for another operation to bring the data structure into a state where it can perform the operation. In some cases you need more than one condition variable.
- Let's say that we want to implement an unbounded buffer using locks and condition variables. In this case we have 2 consumers.

```
Lock *l;
Condition *c;
int avail = 0;
void consumer(int dummy) {
  while (1) {
    l->Acquire();
    if (avail == 0) {
      c->Wait(l);
    }
    consume the next unit of data
    avail--;
    l->Release();
  }
}
void producer(int dummy) {
  while (1) {
    l->Acquire();
    produce the next unit of data
    avail++;
    c->Signal(l);
    l->Release();
  }
}
void main() {
  l = new Lock("l");
  c = new Condition("c");
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 1);
  Thread *t = new Thread("consumer");
  t->Fork(consumer, 2);
  t = new Thread("producer");
  t->Fork(producer, 1);
}
```

- There are two variants of condition variables: Hoare condition variables and Mesa condition variables. For Hoare condition variables, when one thread performs a `Signal`, the very next thread to run is the waiting thread. For Mesa condition variables, there are no guarantees when the signalled thread will run. Other threads that acquire the lock can execute between the signaller and the waiter. The example above will work with Hoare condition variables but not with Mesa condition variables. Why, and how can we fix it?
- Replace the `if` with a `while`.

```
void consumer(int dummy) {
  while (1) {
    l->Acquire();
    while (avail == 0) {
      c->Wait(l);
    }
    consume the next unit of data
```

```
    avail--;
    l->Release();
  }
}
```

In general, this is a crucial point. Always put `while`'s around your condition variable code. If you don't, you can get really obscure bugs that show up very infrequently.

- In this example, what is the data that the lock and condition variable are associated with? The `avail` variable.
- People have developed a programming abstraction that automatically associates locks and condition variables with data. This abstraction is called a monitor. A monitor is a data structure plus a set of operations (sort of like an abstract data type). The monitor also has a lock and, optionally, one or more condition variables. See OSC Section 6.7.
- The compiler for the monitor language automatically inserts a lock operation at the beginning of each routine and an unlock operation at the end of the routine. So, programmer does not have to put in the lock operations.
- Monitor languages were popular in the 80's - they are in some sense safer because they eliminate one possible programming error. But more recent languages have tended not to support monitors explicitly, and expose the locking operations to the programmer. So the programmer has to insert the lock and unlock operations by hand.
- Laundromat Example: A local laudromat has switched to a computerized machine allocation scheme. There are N machines, numbered 1 to N. By the front door there are P allocation stations. When you want to wash your clothes, you go to an allocation station and put in your coins. The allocation station gives you a number, and you use that machine. There are also P deallocation stations. When your clothes finish, you give the number back to one of the deallocation stations, and someone else can use the machine. Here is the alpha release of the machine allocation software:

```
allocate(int dummy) {
  while (1) {
    wait for coins from user
    n = get();
    give number n to user
  }
}
deallocate(int dummy) {
  while (1) {
    wait for number n from user
    put(i);
  }
}
main() {
  for (i = 0; i < P; i++) {
    t = new Thread("allocate");
    t->Fork(allocate, 0);
    t = new Thread("deallocate");
    t->Fork(deallocate, 0);
  }
}
```

- The key parts of the scheduling are done in the two routines `get` and `put`, which use an array data structure `a` to keep track of which machines are in use and which are free.

```
int a[N];
int get() {
  for (i = 0; i < N; i++) {
    if (a[i] == 0) {
      a[i] = 1;
      return(i+1);
    }
  }
}
void put(int i) {
  a[i-1] = 0;
}
```

- It seems that the alpha software isn't doing all that well. Just looking at the software, you can see that there are several synchronization problems.
- The first problem is that sometimes two people are assigned to the same machine. We can fix this with a lock:

```
int a[N];
Lock *l;
int get() {
  l->Acquire();
  for (i = 0; i < N; i++) {
    if (a[i] == 0) {
      a[i] = 1;
      l->Release();
      return(i+1);
    }
  }
  l->Release();
}
void put(int i) {
  l->Acquire();
  a[i-1] = 0;
  l->Release();
}
```

So now, have fixed the multiple assignment problem. But what happens if someone comes in to the laundry when all of the machines are already taken? What does the machine return? Must fix it so that the system waits until there is a machine free before it returns a number. The situation calls for condition variables.

```
int a[N];
Lock *l;
Condition *c;
int get() {
  l->Acquire();
  while (1) {
    for (i = 0; i < N; i++) {
      if (a[i] == 0) {
        a[i] = 1;
        l->Release();
        return(i+1);
      }
    }
    c->Wait(l);
```

```
  }
}
void put(int i) {
  l->Acquire();
  a[i-1] = 0;
  c->Signal();
  l->Release();
}
```

- What data is the lock protecting? The `a` array.
- When would you use a broadcast operation? Whenever want to wake up all waiting threads, not just one. For an event that happens only once. For example, a bunch of threads may wait until a file is deleted. The thread that actually deleted the file could use a broadcast to wake up all of the threads.
- Also use a broadcast for allocation/deallocation of variable sized units. Example: concurrent malloc/free.

```
Lock *l;
Condition *c;
char *malloc(int s) {
  l->Acquire();
  while (cannot allocate a chunk of size s) {
    c->Wait(l);
  }
  allocate chunk of size s;
  l->Release();
  return pointer to allocated chunk
}
void free(char *m) {
  l->Acquire();
  deallocate m.
  c->Broadcast(l);
  l->Release();
}
```

- Example with malloc/free. Initially start out with 10 bytes free.

Time	Process 1	Process 2	Process 3
0	malloc(10) - succeeds	malloc(5) - suspends lock	malloc(5) suspends lock
1		gets lock - waits	
2			gets lock - waits
3	free(10) - broadcast		
4		resume malloc(5) - succeeds	
5			resume malloc(5) - succeeds
6	malloc(7) - waits		
7			malloc(3) - waits
8		free(5) - broadcast	
9	resume malloc(7) - waits		
10			resume malloc(3) - succeeds

  What would happen if changed `c->Broadcast(l)` to `c->Signal(l)`? What would happen if changed `while` loop to an `if`?

- You will be asked to implement condition variables as part of assignment 1. The following implementation is INCORRECT. Please do not turn this implementation in.

```
class Condition {
  private:
    int waiting;
    Semaphore *sema;
}
void Condition::Wait(Lock* l)
{
  waiting++;
  l->Release();
  sema->P();
  l->Acquire();
}
void Condition::Signal(Lock* l)
{
  if (waiting > 0) {
    sema->V();
    waiting--;
  }
}
```

Why is this solution incorrect? Because in some cases the signalling thread may wake up a waiting thread that called Wait after the signalling thread called Signal.