Thread Manipulation and Synchronization

Threads used 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 with a data structure for the TCB (thread control block).
- "`Fork`" gives a new thread a function to run.
  It allocates a stack for the thread, sets up the TCB, then puts the thread on a thread ready queue.
- **Setup TCB:** Fill the stack pointer. Set the PC to be the first instruction in the function, set a register to the first parameter.

- The system maintains a thread ready queue. Whenever a processor becomes idle, the scheduler grabs a thread to run. Then system restores the state from its TCB to run the selected function.

Problems with Concurrent Threads

Two threads increment the same variable "a".

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

- The desired result: `a` is 2 after both threads finish.
- Possible results when execute concurrently:

```
T0 : a=1                  T0 : a=2
T1 : a=2                  T1 : a=1

T0 : 1
T1 : 1
```


Atomic Operations & Mutual Exclusion

- An atomic operation is one that executes without any interference from other operations. i.e. it executes as one unit.
- If "a=a+1" is an atomic operation, the final result is guaranteed to be the same as if the operations executed in some serial order.
- Use *mutual exclusion* to make operations atomic:
  Only one thread is allowed to update "a" at a time (called *a mutual exclusion*).
- The code that performs the atomic operation is called *a critical section.*
- Use synchronization operations to implement mutual exclusions.

Semaphore for synchronization

A semaphore is a counter that support two atomic operations, P and V.

The Semaphore interface from Nachos:

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

- 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.

The Sum example

```
int a = 0;
Semaphore *s;
void sum(int p) {
  int t;
  s->P();
  a=a+1;
  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);
}
```

A semaphore is used to do mutual exclusion.

Semaphores for other problems

Semaphores can do more than mutual exclusion: e.g. synchronize a producer/consumer or a pipe problem.

Producer is generating data and the consumer is consuming data. e.g. One person types a keyboard as a producer and the Unix shell reads characters as a consumer.

```
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);
}
```

Bounded Buffers

Producer/consumer with a limited buffer.

- Consumer cannot run forever without data provided by producer.
- Producer cannot run forever.

```
Semaphore *full;
Semaphore *empty;
void consumer(int dummy) {
  while (1) {
    data->P(); // wait for data
    consume the next unit of data
    space->V(); //release space
  }
}
void producer(int dummy) {
  while (1) {
    space->P(); //wait for space
    produce the next unit of data
    data->V(); //more data
  }
```



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


Other synchronization abstraction

Locks and **Condition variables** (e.g. provided in POSIX Pthreads).

- Locks are an abstraction specifically for mutual exclusion.
- The Nachos lock interface:

```
class Lock {
  public:
    Lock(char* debugName);
    ~Lock();
    void Acquire();
    void Release();
}
```

- A lock can be in one of two states: locked and unlocked.
- Semantics of **atomic** lock operations:
  - Lock(name) : creates a free lock with 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.

How to 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.
- Lock allows threads to perform complicated atomic operations on each piece of data.

Requirements for lock implementation

- **Safty, or called mutual exclusion**. Only one thread can acquire lock at a time.
- **Progress**. If multiple threads try to acquire an unlocked lock, one of the threads will get it.
- **Bouned waiting.** Lock acquiring completes in finite time.

The need for condition varilables

If we use locks for unbounded buffer:

- if a consumer wants to consume before the producer produces data, it must wait
- Thus consumder needs to acquire a lock, and then check if something can be consumed.
- Consumer does not know when data is available. Thus it must loop, checking again and again until the data is ready.

This is bad because it wastes CPU resources.

Condition variables

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 is reacquired.
- Signal(Lock *l) : Enables one waiting thread to run. When Signal returns, lock is still acquired.
- Broadcast(Lock *l) : Enables all waiting threads to run. When Broadcast returns, the lock is still acquired.

How to use condition variables

- 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 (condition is not satisfied), it uses the condition variable to wait until it can perform the operation.
- In some cases you may need more than one condition variable.
- A programming abstraction, which automatically associates locks and condition variables with data, is called a **monitor.**

  A monitor is a data structure plus a set of operations. The monitor also has a lock and, optionally, one or more condition variables. See OSC Section 6.7.

Condition variables for unbounded buffer

```
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);
}
```


Two variants of condition variables

- **Hoare condition variables.** When One thread performs a `Signal`, the very next thread to run is the waiting thread.
- **Mesa condition variables.** 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.

Put `while`'s around condition variables:

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


Laundromat Example

- **N laundry machines,** numbered 1 to N.
- **P allocation stations.**
  When you want to wash, go to an allocation station and put in your coins. The allocation station gives you a machine number that you use.
- **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.

Laundromat code: alpha release

```
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(n);
  }
}
main() {
  for (i = 0; i < P; i++) {
    t = new Thread("allocate");
    t->Fork(allocate, 0);
    t = new Thread("deallocate");
    t->Fork(deallocate, 0);
  }}
```


Code for get() and put()

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;
}
```


Lock for concurrent access

Two people may be assigned to the same machine.

```
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();
}
```


Condition variables for waiting

If someone waits when all machines are taken:

```
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();
}
```

When to use broadcast()

Whenever want to wake up all waiting threads.

Example: a broadcast for allocation/deallocation of variable sized units. e.g. 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.

m() → malloc()   f → free()

Process 1	Process 2	Process 3
m(10) - succ	m(5) - suspend	m(5)-suspend
	gets lock - wait	
		gets lock - wait
f(10) - broadcast		
	resume m(5)-succ	
		resume m(5)-succ
m(7) - wait		
		m(3) - wait
	f(5) - broadcast	
resume m(7)-wait		
		resume m(3)-succ

Deadlock: Example

```
Lock *l1, *l2;
void p() {
  l1->Acquire();
  l2->Acquire();
  Manipulate data that l1/l2 protect;
  l2->Release();
  l1->Release();
}
void q() {
  l2->Acquire();
  l1->Acquire();
  Manipulate data that l1/l2 protect;
  l1->Release();
  l2->Release();}
```

If `p` and `q` execute concurrently, they may wait forever (called deadlock).

- First, `p` acquires `l1` and `q` acquires `l2`.
- Then, `p` waits to acquire `l2` and `q` waits to acquire `l1`.

Conditions for deadlock

Deadlock if the following conditions are true:

- **Mutual Exclusion:** Only one thread can hold lock at a time.
- **Hold and Wait:** At least one thread holds a lock and is waiting for another process to release a lock.
- **No preemption:** Only the process holding the lock can release it.
- **Circular Wait:** There is a set $t_1, \ldots, t_n$ such that t_1 is waiting for a lock held by t_2, ..., t_n is waiting for a lock held by t_1.

How to avoid such a deadlock?

- Order the locks, and always acquire the locks in that order.
- Eliminates the circular wait condition.