A process is an instance of a running program

- A thread is an execution context
- Process can have one or more threads
- Threads share address space (code, data, heap), open files
- Threads have their own stack and register state

POSIX Thread APIs:

- `pthread_create()` - Creates a new thread
- `pthread_exit()` - Destroys current thread
- `pthread_join()` - Waits for thread to exit
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
            mutex_unlock (&mutex); /* <--- Why? */
            thread_yield ();
            mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
void consumer (void *ignored) {
    for (; ;) {
        mutex_lock (&mutex);
        while (count == 0) {
            mutex_unlock (&mutex);
            thread_yield ();
            mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
Condition variables

- Busy-waiting in application is a bad idea
  - Consumes CPU even when a thread can’t make progress
  - Unnecessarily slows other threads/processes or wastes power

- Better to inform scheduler of which threads can run

- struct cond_t;
  - pthread_cond_t or cv in OS
- void cond_init (cond_t *, ...);
- void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock m and sleep until c signaled
  - Then re-acquire m and resume executing
- void cond_signal (cond_t *c);
- void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on c
Condition variables

- Busy-waiting in application is a bad idea
  - Consumes CPU even when a thread can’t make progress
  - Unnecessarily slows other threads/processes or wastes power
- Better to inform scheduler of which threads can run
- Typically done with condition variables

```c
struct cond_t;  \(\text{pthread_cond_t or cv in OS/161}\)
void cond_init (cond_t *, ...);
void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock \(m\) and sleep until \(c\) signaled
  - Then re-acquire \(m\) and resume executing
void cond_signal (cond_t *c);
void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on \(c\)
mutex_t mutex = MUXTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    item *nextProduced = produce_item ();

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE)
      cond_wait (&nonfull, &mutex);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal (&nonempty);
    mutex_unlock (&mutex);
  }
}
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
            cond_wait (&nonempty, &mutex);

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
Re-check conditions

- **Always re-check condition on wake-up**
  
  ```c
  while (count == 0) /* not if */
  cond_wait (&nonempty, &mutex);
  ```

- **Otherwise, breaks with spurious wakeup or two consumers**
  
  - Start where Consumer 1 has mutex but buffer empty, then:

  ```c
  Consumer 1
  cond_wait (...);

  Consumer 2
  mutex_lock (...);
  if (count == 0)
  
  use buffer[out]...
  count--;
  mutex_unlock (...);

  Producer
  mutex_lock (...);
  count++;
  cond_signal (...);
  mutex_unlock (...);
  ```

  `use buffer[out]... ← No items in buffer`
- Why must `cond_wait` both release mutex & sleep?
- Why not separate mutexes and condition variables?

```c
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```
Condition variables (continued)

- **Why must** `cond_wait` **both** release mutex & sleep?
- **Why not** separate mutexes and condition variables?

```c
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```

- Can end up stuck waiting when bad interleaving

**Producer**
```
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
}
```

**Consumer**
```
mutex_lock (&mutex);
...
count--;  
cond_signal (&nonfull);
```

- **Problem:** `cond_wait` & `cond_signal` do not commute
Condition variables (continued 2)

- Should you hold the mutex when calling signal/broadcast?

  - Case one.pnum: Holding the mutex
    - Waiter is woken up by signal
    - Waiter immediately sleeps waiting for mutex
    - This causes two context switches
    - Pthread implementations solve this through wait morphing
      - Thread is automatically moved from the cv to mutex wait queue

  - Case two.pnum: Not holding the mutex
    - Signal occurs just before call to cond_wait
    - Stuck in infinite wait
• Should you hold the mutex when calling signal/broadcast?

• Case 1: Holding the mutex
  - Waiter is woken up by signal
  - Waiter immediately sleeps waiting for mutex
  - This causes two context switches
  - Pthread implementations solve this through wait morphing
  - Thread is automatically moved from the cv to mutex wait queue
• Should you hold the mutex when calling signal/broadcast?

• Case 1: Holding the mutex
  - Waiter is woken up by signal
  - Waiter immediately sleeps waiting for mutex
  - This causes two context switches
  - Pthread implementations solve this through wait morphing
  - Thread is automatically moved from the cv to mutex wait queue

• Case 2: Not holding the mutex
  - Signal occurs just before call to cond_wait
  - Stuck in infinite wait
A **Semaphore** is initialized with an integer $N$

- `sem_create(N)`

**Provides two functions:**

- `sem_wait(S)` (originally called $P$)
- `sem_signal(S)` (originally called $V$)

**Guarantees** `sem_wait` will return only $N$ more times than `sem_signal` called

- Example: If $N == 1$, then semaphore acts as a mutex with `sem_wait` as lock and `sem_signal` as unlock

**Semaphores give elegant solutions to some problems**

**Linux primarily uses semaphores for sleeping locks**

- `sema_init`, `down_interruptible`, `up`, ...
- Also weird reader-writer semaphores, `rw_semaphore` [Love]
• We can use a semaphore as a mutex

```c
semaphore *s = sem_create(1);

/* Acquire the lock */
sem_wait(s); /* Semaphore count is now 0 */
/* critical section */
/* Release the lock */
sem_signal(s); /* Semaphore count is now 1 */
```
Using a Semaphore as a Mutex

- We can use a semaphore as a mutex

```c
semaphore *s = sem_create(1);

/* Acquire the lock */
sem_wait(s); /* Semaphore count is now 0 */
/* critical section */
/* Release the lock */
sem_signal(s); /* Semaphore count is now 1 */
```

- Couple important differences:
  - Mutex requires the same thread to acquire/relase the lock
  - Allows mutexes to implement priority inversion
Semaphore producer/consumer

- **Initialize** `full` to 0 (block consumer when buffer empty)
- **Initialize** `empty` to `N` (block producer when queue full)

```c
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait (&empty);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&full);
    }
}

void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&empty);
        consume_item (nextConsumed);
    }
}
```
Various synchronization mechanisms

- Other more esoteric primitives you might encounter
  - Plan 9 used a **rendezvous** mechanism
  - Haskell uses MVars (like channels of depth 1)
- Many synchronization mechanisms equally expressive
  - Pintos implements locks, condition vars using semaphores
  - Could have been vice versa
  - Can even implement condition variables in terms of mutexes
- Why base everything around semaphore implementation?
  - High-level answer: no particularly good reason
  - If you want only one mechanism, can’t be condition variables (interface fundamentally requires mutexes)
  - Unlike condition variables, `sem_wait` and `sem_signal` commute, eliminating **problem of condition variables w/o mutexes**
struct semaphore *sem_create(const char *name, int count);
void sem_destroy(struct semaphore *sem);
void P(struct semaphore *sem);
void V(struct semaphore *sem);

struct cv *cv_create(const char *name);
void cv_destroy(struct cv *cv);
void cv_wait(struct cv *cv, struct lock *lock);
/* Ignore the lock parameter on signal and broadcast */
/* We will discuss this next class */
void cv_signal(struct cv *cv, struct lock *lock);
void cv_broadcast(struct cv *cv, struct lock *lock);
Implementation of P and V

- See os161/kern/thread/synch.c

```c
void P(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        wchan_lock(sem->sem_wchan);
        spinlock_release(&sem->sem_lock);
        wchan_sleep(sem->sem_wchan);
        spinlock_acquire(&sem->sem_lock);
    }
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}

void V(struct semaphore *sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```