Overview of last lecture

- Concurrency vs Parallelism, how threads enable both.
- What are threads, how do they change the address space.
- How access to shared data leads to **race condition**.
- Identifying **critical section**.
- **Atomicity**.
- **Mutual exclusion**.
- **Locks**, **spin-locks**, how to implement locks with hardware support.
- Evaluating lock implementations.
- Last question from last lecture: how would spin-locks perform on a machine with multiple CPUs?

Condition Variables

- A thread often wishes to check for a condition before continuing its execution.

```
void *child(void *arg) {
  printf("child\n");
  // XXX how to indicate that we are done?
  return NULL;
}

int main(void) {
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL, child, NULL);
  // XXX how to wait for child?
  printf("parent: end\n");
  return 0;
}
```

parent: begin
child
parent: end

Condition Variables

```
volatile int done = 0;

void *child(void *arg) {
  printf("child\n");
  done = 1;
  return NULL;
}

int main(void) {
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(&c, NULL,
                 child, NULL);
  while (done == 0)
    ;
  printf("parent: end\n");
  return 0;
}
```

- Problem: wastes CPU cycles by spinning

Condition Variables

- Condition variable is an explicit queue.
- Threads whose execution is blocked by certain condition being true, but is false at the moment, can put themselves to the queue by **waiting** on the condition.
- Threads whose execution lead to the condition becoming true, can let the waiters know by **signalling** on the condition

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thread_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void *child(void *arg) {
  printf("child\n");
  thread_exit();
  return NULL;
}

void thread_join() {
  pthread_mutex_lock(&m);
  while (done == 0)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}

int main(void) {
  printf("parent: begin\n");
  pthread_t p;
  pthread_create(&p, NULL, child, NULL);
  thread_join();
  printf("parent: end\n");
  return 0;
}
```

- What happens when parent creates the child thread and continues to run?
- What happens when parent creates the child thread and child thread runs immediately?

Condition Variables

- Do we need the state variable?
- What happens below if child runs immediately?
- State variable is what both threads are interested in. Without state, threads cannot communicate.

- Do we need holding the lock before waiting/signaling?
- What happens below if parent runs first, but is interrupted right before wait?
- “Check for predicate and block if true” should be atomic to avoid race condition.

```
void thread_exit() {
  pthread_mutex_lock(&m);
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}

void thread_join() {
  pthread_mutex_lock(&m);
  pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

```
void thread_exit() {
  done = 1;
  pthread_cond_signal(&c);
}

void thread_join() {
  if (done == 0)
    pthread_cond_wait(&c);
}
```

Producer/Consumer Problem

```
int buffer;
int count = 0;


void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}


int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```

```
void *producer(void *arg) {
  int loops = (int) arg;
  for (int i = 0; i < loops; ++i)
    put(i);
}



void *consumer(void *arg) {
  int i;
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer Problem – Broken Solution

- Works only for one producer and one consumer.
- Two problems with it, what are those?

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

```
void *producer(void *arg) {
  int loops = (int) arg;
  for (int i = 0; i < loops; ++i) {
    pthread_mutex_lock(&m);
    if (count == 1)
      pthread_cond_wait(&c, &m);
    put(i);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
  }
}
```

```
void *consumer(void *arg) {
  int loops = (int) arg;
  for (int i = 0; i < loops; ++i) {
    pthread_mutex_lock(&m);
    if (count == 0)
      pthread_cond_wait(&c, &m);
    int tmp = get();
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer Problem – General Solution

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % MAX;
  ++count;
}

int get() {
  int tmp = buffer[use];
  use = (use + 1) % MAX;
  --count;
  return tmp;
}
```

```
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *arg) {
  int loops = (int) arg;
  for (int i = 0; i < loops; ++i) {
    mutex_lock(&mutex);
    while (count == MAX)
      pthread_cond_wait(&empty, &m);
    put(i);
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *arg) {
  int loops = (int) arg;
  for (int i = 0; i < loops; ++i) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
      pthread_cond_wait(&full, &m);
    int temp = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&m);
    printf("%d\n", tmp);
  }
}
```

Semaphores

- A synchronization primitive introduced by Edsger Dijkstra.
- It is complete – can be used to implement locks and condition variables.
- It is an object with integer value that can be manipulated with two routines. POSIX calls them **sem_wait()** and **sem_post()**.
- Initial value defines its behavior.

Semaphores

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);

int sem_wait(sem_t *s) {
  wait until value of semaphore s is greater than 0
  decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
  increment the value of semaphore s by 1
  if there are 1 or more threads waiting, wake 1
}
```

- Bodies are executed atomically.

Semaphors as Locks (Binary Semaphores)

```
sem_t m;
sem_init(&m, 0, 1);
```

```
sem_wait(&m);
// critical section here
sem_post(&m);
```

Semaphores as Condition Variables

```
sem_t s;

void *child(void *arg) {
  printf("child\n");
  sem_post(&s);
  return NULL;
}

int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0);
  printf("parent: begin\n");
  pthread_t c;
  pthread_create(c, NULL, child, NULL);
  sem_wait(&s);
  printf("parent: end\n");
  return 0;
}
```

Producer/Consumer Problem with Semaphores

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % MAX;
}

int get() {
  int tmp = buffer[use];
  use = (use + 1) % MAX;
  return tmp;
}
```

```
sem_t empty;
sem_t full;
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);
    tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
int main(int argc, char *argv[]) {
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);
}
```

Producer/Consumer Problem with Semaphores

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value;
  fill = (fill + 1) % MAX;
}

int get() {
  int tmp = buffer[use];
  use = (use + 1) % MAX;
  return tmp;
}
```

- There is a race condition for the case of multiple producers/consumers

```
sem_t empty;
sem_t full;
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    put(i);
    sem_post(&full);
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);
    tmp = get();
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
int main(int argc, char *argv[]) {
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);
}
```

Producer/Consumer Problem with Semaphores

```
sem_t empty;
sem_t full;
sem_t mutex;

int main(int argc, char *argv[])
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);

  // We introduce lock for
  // mutual exclusion to buffer.
  sem_init(&mutex, 0, 1);
  ...
}
```

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(i);
    sem_post(&full);
    sem_post(&mutex);
  }
}
```

```
void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer Problem with Semaphores

```
sem_t empty;
sem_t full;
sem_t mutex;

int main(int argc, char *argv[])
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);

  // We introduce lock for
  // mutual exclusion to buffer.
  sem_init(&mutex, 0, 1);
  ...
}
```

- Mutex scope is is too wide, leads to a deadlock.

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&empty);
    put(i);
    sem_post(&full);
    sem_post(&mutex);
  }
}

void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&mutex);
    sem_wait(&full);
    int tmp = get();
    sem_post(&empty);
    sem_post(&mutex);
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer Problem with Semaphores – Working Solution

```
sem_t empty;
sem_t full;
sem_t mutex;

int main(int argc, char *argv[])
  sem_init(&empty, 0, MAX);
  sem_init(&full, 0, 0);

  // We introduce lock for
  // mutual exclusion to buffer.
  sem_init(&mutex, 0, 1);
  ...
}
```

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
  for (int i = 0; i < loops; i++) {
    sem_wait(&full);
    sem_wait(&mutex);
    int tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

Reader-Writer Lock

```
typedef struct _rwlock_t {
  sem_t lock;
  sem_t writelock;
  int readers;
} rwlock_t;


void rwlock_init(rwlock_t *rw) {
  readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

- As implemented here, not fair, readers can starve writers.

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Deadlocks

- Example

```
Thread 1:
lock(L1);
lock(L2);
```

```
Thread 2:
lock(L2);
lock(L1);
```

- Why deadlocks occur:
  - Circular dependencies, encapsulation
    - Vector v1, v2; v1.AddAll(v2);
- Four conditions required for a deadlock to occur
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait

Deadlocks – Prevention

- Circular wait
  - Provide a total ordering on lock acquisition.
- Hold-and-wait
  - Acquire “prevention” lock before acquiring others, and release it at the end.
- No preemption
  - Can lead to **livelock.**
- Mutual exclusion
  - **Wait-free** data structures, data structures that do not require explicit locking.

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

Monitors

- An object or a module to be used safely by more than one thread. (Wikipeda)
- Invented by Per Brinch Hansen and Tony Hoare.
- A monitor guarantees that **only one thread can be active within the monitor at a time.**

Monitors

```
class account {
 private:
  int balance = 0;

 public:
  void deposit(int amount) {
    balance = balance + amount;
  }

  void withdraw(int amount) {
    balance = balance - amount;
  }
};
```

```
class account {
 private:
  int balance = 0;
  pthread_mutex_t monitor;

 public:
  void deposit(int amount) {
    pthread_mutex_lock(&monitor);
    balance = balance + amount;
    pthread_mutex_unlock(&monitor);
  }

  void withdraw(int amount) {
    pthread_mutex_lock(&monitor);
    balance = balance - amount;
    pthread_mutex_unlock(&monitor);
  }
};
```

Monitors

```
monitor class BoundedBuffer {
 private:
  int buffer[MAX];
  int fill, use;
  int fullEntries = 0;
  cond_t empty, full;

 public:
  void produce(int element) {
    if (fullEntries == MAX)
      wait(&empty);
    buffer[fill] = element;
    fill = (fill + 1) % MAX;
    fullEntries++;
    signal(&full);
  }
  int consume() {
    if (fullEntries == 0)
      wait(&full);
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    fullEntries--;
    signal(&empty);
    return tmp;
  }
}
```

- The pseudo-code on the left is written using **Hoare semantics**.
- It will lead to a race if **Mesa semantics** is assumed.
- Always recheck the condition after being woken up.