A good review video

- https://www.youtube.com/watch?v=1BvYJMcI AeU

- Correction: CPU doesn’t do scheduling. It is the OS who schedules which thread/process to run
6. void producer(void)  
7. { int item;  
8.  
9.   while (TRUE) {  
10.      item = produce_item ();  
11.      P(&empty);  
12.      P(&mutex);  
13.      insert_item(item);  
14.      V(&mutex);  
15.   }  
16. }  

17. void consumer(void)  
18. { int item;  
19.  
20.   while (TRUE) {  
21.      P(&mutex);  
22.      P(&full);  
23.      item= remove_item();  
24.      V(&mutex);  
25.      consume_item(item);  
26.   }  
27. }
Implementing lock Using Test-And-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```
Next: Use Swap to Implement Lock

- Swap = TSL

```c
void Swap (char* x,* y);
\ All done atomically
{
    char temp = *x;
    *x = *y;
    *y = temp
}
```
Use Swap to implement lock

- Is the following code correct?

```c
typedef struct{
    int held=0;
}Lock;

void Acquire(Lock l)
{
    int tmp = 1;
    do{
        Swap(&tmp, &(l->held));
    } while(l->held != 0);
}
```

This is an access to a shared variable
Use Swap to implement lock-
Continue

- Correct implementation

```c
typedef struct {
    int held=0;
} Lock;

void Acquire(Lock l) {
    int tmp = 1;
    while(tmp)
        Swap(&tmp, &(l->held));
}
```
Barriers

(a) Process

(b) Barrier

(c) Barriers
Implement Barrier

- Use the following to implement a barrier for 5 threads

// types that you can use
- Mutex // binary semaphore
- Condition // condition variable
- int
- Boolean
Functions you can use

Functions that you can use

- **acquire(Lock m)**
  - Semantics: acquires the lock on m; other threads invoking lock on m before it is unlocked are appended to the waiting queue on m and put to sleep

- **release(Lock m)**
  - Semantics: releases the lock on m; removes one thread(if any) from the waiting queue on m and awakens it

- **wait(Condition cond, Lock m)**
  - Semantics: appends the thread to the waiting queue on cond; automatically unlocks m; the thread is then put to sleep

- **signal(Condition cond)**
  - Semantics: removes one of the threads(if any) from the waiting queue on cond and awakens it

- **broadcast(Condition cond)**
  - Semantics: removes all threads(if any) from the waiting queue on cond and awakens them
Implement Barrier

// thread code
// declarations

// WRITE SOMETHING HERE

// code doing some useful work here, blah blah blah YOU DO NOT NEED to WRITE HERE....

// now reach barrier
// WRITE SOMETHING HERE TOO

// now all threads continue running

Solutions are in https://cseweb.ucsd.edu/classes/fa15/cse120-a/hw/hw2-sol.html
Step 1: Implement only functionality

```c
int numReadyThreads = 0;

void Barrier(){
    numReadyThreads ++;
    while(numReadyThreads < 5);
}
```
int numReadyThreads = 0;
Lock lck;

void Barrier()
{
    acquire(lck);
    numReadyThreads ++;
    while(numReadyThreads < 5);
    release(lck);
}

Is the release in the right place?
Step 2: Add Mutual Exclusion (2nd attempt)

```c
int numReadyThreads = 0;
Lock lck;

void Barrier(){
    acquire(lck);
    numReadyThreads ++;
    while(numReadyThreads < 5){
        release(lck);
    }
}
```
Step 2: Add Mutual Exclusion

int numReadyThreads = 0;
Lock lck;

void Barrier(){
    acquire(lck);
    numReadyThreads ++;
    while(numReadyThreads < 5){
        release(lck);
        acquire(lck);
    }
}

int numReadyThreads = 0;
Lock lck;

void Barrier(){
    acquire(lck);
    numReadyThreads ++;
    while(numReadyThreads < 5){
        release(lck);
        threadYield();
        acquire(lck);
    }
    Release(lck)
}
Step 3: Avoid Busy Waiting with a condition variable

```c
int numReadyThreads = 0;
Lock lck;
Condition waitQ;
void Barrier(){
    acquire(lck);
    numReadyThreads ++;
    if(numReadyThreads < 5){
        wait(waitQ, lck)
    } else{
        broadcast(waitQ);
    }
    release(lck);
}
```
Old Bridge

- An old bridge has only one lane and can only hold at most 3 cars at a time without risking collapse.
- Implement
  - `ArriveBridge(int direction)` and `ExitBridge()`
  - To control traffic so that at any given time, there are at most 3 cars on the bridge, and all of them are going the same direction.
  - A car calls `ArriveBridge` when it arrives at the bridge and wants to go in the specified direction (0 or 1);
  - `ArriveBridge` should not return until the car is allowed to get on the bridge.
  - A car calls `ExitBridge` when it gets off the bridge, potentially allowing other cars to get on.
  - Don't worry about starving cars trying to go in one direction; just make sure cars are always on the bridge when they can be.
int currentDirection; // current direction 0 or 1

int cars; // number of cars on the bridge

int waiters[2]; // number of waiters in each direction

Condition waitingToGo[2]; // waiting queues for each direction

Lock lock; // a lock for mutual exclusion to access above shared variables
ArriveBridge(int direction) {  //direction is either 0 or 1
    acquire(lock);

    // while can't get on the bridge, wait
    while ((cars == 3) ||
           (cars > 0 && currentdirection != direction)) {
        waiters[direction]++;
        wait(waitingToGo[direction], lock);
        waiters[direction]--;
    }

    // get on the bridge
    cars++;
    currentdirection = direction;

    release(lock);
}
ExitBridge() {
    acquire(lock);

    // get off the bridge
    cars--;

    // if anybody wants to go the same direction, wake them
    if (waiters[currentdirection] > 0)
        signal(waitingToGo[currentdirection]);

    // else if empty, try to wake somebody going the other way
    else if (cars == 0)
        broadcast(waitingToGo[1-currentdirection]);

    release(lock);
}
Counting Semaphore

- Given an implementation of binary semaphores (i.e., a V operation on a binary semaphore with a value of 1 does not change the value of the semaphore), implement counting semaphores.

- Show any variables or semaphores you use and their initial values.
CONTROL VARIABLES:

    mutex: semaphore, initial value 1 (FREE)
    delay: semaphore, initial value 0
    csem: integer, initialized to the number of units of the resource

P(CSEM):

    /* claim exclusive access to csem */
    P(mutex);

    /* decrement csem, either to claim a resource unit or to record a new waiter */
    csem = csem - 1;

    /* if there were no units available, release csem and join the queue */
    if (csem <= -1) {
        V(mutex);
        P(delay);
    }

    /* otherwise, release csem and proceed to use (one unit of) the resource */
    else V(mutex);

V(CSEM):

    /* claim exclusive access to csem */
    P(mutex);

    /* increment csem, to release a waiter (if any) or announce a free unit */
    csem = csem + 1;

    /* if there were waiters in the queue, release one */
    if (csem < 0) V(delay);

    /* release csem */
    V(mutex);
Another Exercise

● Spoiled parents----my dream 😊
  - A Kid thread executes function \textit{Kid()}:
    ● If two parents have not come back home from shopping, read books
    ● Once both parents have come back, start cooking
    ● After food is ready, each with parents
  - Two spoiled parents, each execute \textit{Parent()}:
    ● Go back home
    ● If food is not ready, take a nap
    ● If food is ready, eat dinner
A thread-safe Queue

- Queue Data Structure
  - Queue is a data structure that maintain "First In First Out" (FIFO) order

- Queue operations
  - 1. enqueue - insert item at the back of queue Q
  - 2. dequeue - return (and virtually remove) the front item from queue Q
  - 3. init - initialize queue Q, reset all variables.
```c
#include <stdio.h>
define QUEUE_SIZE 100
typedef struct {
  int q[QUEUE_SIZE];
  int first, last;
  int count;
} queue;
void init_queue(queue *q)
{
  q->first = 0;
  q->last = QUEUE_SIZE - 1;
  q->count = 0;
}
void enqueue(queue *q, int x)
{
  q->last = (q->last + 1) % QUEUE_SIZE;
  q->q[q->last] = x;
  q->count = q->count + 1;
}
int dequeue(queue *q)
{
  int x = q->q[q->first];
  q->first = (q->first + 1) % QUEUE_SIZE;
  q->count = q->count - 1;
  return x;
}
```
How to make it synchronization-safe using locks?

```c
#include <stdio.h>
#define QUEUE_SIZE 100
typedef struct {
    int q[QUEUE_SIZE];
    int first, last;
    int count;
} queue;

void init_queue(queue *q) {
    q->first = 0;
    q->last = QUEUE_SIZE - 1;
    q->count = 0;
}

void enqueue(queue *q, int x) {
    q->last = (q->last + 1) % QUEUE_SIZE;
    q->q[ q->last ] = x;
    q->count = q->count + 1;
}

int dequeue(queue *q) {
    int x = q->q[ q->first ];
    q->first = (q->first + 1) % QUEUE_SIZE;
    q->count = q->count - 1;
    return x;
}
```
#include <stdio.h>
#define QUEUE_SIZE 100
typedef struct {
    int q[QUEUE_SIZE];
    int first, last;
    int count;
} queue;
void init_queue(queue *q) {
    q->first = 0;
    q->last = QUEUE_SIZE - 1;
    q->count = 0;
}
void enqueue(queue *q, int x) {
    q->last = (q->last + 1) % QUEUE_SIZE;
    q->q[q->last] = x;
    q->count = q->count + 1;
}
int dequeue(queue *q) {
    int x = q->q[q->first];
    q->first = (q->first + 1) % QUEUE_SIZE;
    q->count = q->count - 1;
    return x;
}