Section 5: Condition Variables and Address Translation

Frank Austin Nothaft*

July 8th, 2015

* This work is a composite of the Spring 2015 section 5 handout by Jason Jia/Roger Chen and the Fall 2010 midterm 1.

1 Warmup

1.1 Hello World

Will this code compile/run?
Why or why not?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void print_hello() {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
}
```

2 Vocabulary

- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is associated with:
  - a lock (a condition variable + its lock are known together as a **monitor**)
  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

  In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:

  - **cv_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
  - **cv_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
  - **cv_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

When a **wait()**ing thread is notified and put back in the ready state, it also re-acquires the lock before the **wait()** function returns.
When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call notify() or broadcast() on the condition's CV, so waiting threads can be notified, and finally release the lock.
Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition C as false, then thread 2 makes condition C true and calls **cv.notify**, then 1 calls **cv.wait** and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.

- **Hoare Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.
- **Mesa Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true, with no guarantee that the thread will execute immediately. The newly woken thread simply gets put on the ready queue and is subject to the same scheduling mechanisms as any other thread. The implication of this is that **you must check the condition with a while loop instead of an if statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU**.
- **Spin Locks** - A type of lock where the implementation of **lock.acquire()** is to simply check if the lock is available in a loop ("spin"). Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting.

3 Problems

3.1 Hello Word Continued

Add in the necessary code to the warmup to make it work correctly.

3.2 Spot the Problem

What is wrong with this code?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int n = 3;
void counter() {
  pthread_mutex_lock(&lock);
  for (n = 3; n > 0; n--)
    printf("%d\n", n);
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
}
void announcer() {
  while (n != 0) {
    pthread_mutex_lock(&lock);
    pthread_cond_wait(&cv, &lock);
    pthread_mutex_unlock(&lock);
  }
  printf("BLAST OFF!\n");
}
```

3.3 Baking with Condition Variables

A number of people are trying to bake cakes. Unfortunately, they each know only one skill, so they need to all work together to bake cakes. Use independent threads (one person is one thread) which communicate through condition variables to solve the problem. A skeleton has been provided, fill in the blanks to make the implementation work.

A cake requires:

- 1 cake batter
- 2 eggs

Instructions:

1. Add ingredients to bowl
2. Heat bowl (it's oven-safe)
3. Eat the cake, clean out the bowl, and go back to step 1

Requirements:

- Don't start heating the cake in the oven unless there are exactly the right number of ingredients in the bowl.
- Don't add raw ingredients to a currently-baking cake or a finished cake.
- Don't eat the cake unless it's done baking.
- Given enough time, the code should bake an unbounded number of cakes, and should never stop.

```
int numBatterInBowl = 0;
int numEggInBowl = 0;
bool readyToEat = false;
pthread_mutex_t lock;
pthread_cond_t needIngredients;
pthread_cond_t readyToBake;
pthread_cond_t startEating;

void batterAdder()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    addBatter(); // Sets numBatterInBowl += 1

    ------------------------------
  }
}
```

```
void eggBreaker()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    addEgg(); // Sets numEggInBowl += 1

    Text
    ------------------------------
  }
}

void bowlHeater()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ------------------------------------------------------------ {

      ------------------------------
    }
    heatBowl(); // Sets readyToEat = true, numBatterInBowl = 0, numEggInBowl = 0

    ------------------------------
  }
}

void cakeEater()
{
  pthread_mutex_lock(&lock);
  while (1) {

    ---------------------------------------------- {

      ------------------------------
    }
    eatCake(); // Sets readyToEat = false and cleans the bowl for another cake

    ------------------------------
  }
}

int main(int argc,char *argv[])
{
  // Initialize mutex and condition variables
  // Start threads: 1 batterAdder, 2 eggBreakers, 1 bowlHeater, and 1 cakeEater
  // main() sleeps forever
```

Problem 5: Virtual Memory [18pts]

Consider a two-level memory management scheme on 24-bit virtual addresses using the following format for virtual addresses:

Virtual Page # (8 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Virtual addresses are translated into 16-bit physical addresses of the following form:

Physical Page # (8 bits)	Offset (8 bits)

Page table entries are 16 bits in the following format, *stored in big-endian form* in memory (i.e. the MSB is first byte in memory).

Page Table Entry (PTE)

Physical Page # (8 bits)	Kernel Only	Uncacheable	0	0	Dirty	Use	Write	Valid

Note that a virtual-physical translation can fail at any point if an incompatible PTE is encountered. Two types of errors can occur during translation: "invalid page" (page is not mapped at all) or "access violation" (page exists, but access was illegal).

Problem 5a[2pts]: How big is a page? Explain.

Problem 5b[2pts]: What is the largest size for a page table with this address space? We are asking for the total size of both levels of the page table. Explain.

Problem 5c[3pts]: What does "TLB" stand for and what is its function? How big would a TLB entry be for this system?

Problem 5d[3pts]: Sketch the format of the page-table for this multi-level virtual memory management scheme. Illustrate the process of resolving an address as well as possible.

Problem 5e[2pts]: What is "Copy on Write"? How would you perform Copy on Write with the Virtual Memory system discussed in this problem?

Problem 5f[6pts]: The contents of physical memory are given on the next page. *Assume that the page-table base pointer = 0x2000, and that the CPU is in user-mode.* Please return the results from the following load/store instructions. Addresses are virtual. The return value for load is an 8-bit data value or an error, while the return value for a store is either "**ok**" or an error. For errors, please specify which type of error (either "invalid page" or "access violation").

Instruction	Return Value
Load [0x700FE]	**0xEE**
Store [0x700FE]	**Access violation**
Load [0xC2345]	
Load [0x00115]	

Instruction	Return Value
Store [0x10310]	
Load [0x20102]	
Store [0x20731]	
Load [0x81015]	

Virtual Address Format

Virtual Page # (8 bits)	Virtual Page # (8 bits)	Offset (8 bits)

Page Table Entry (PTE)

Physical Page # (8 bits)	Kernel	Not Cacheable	0	0	Dirty	Use	Write	Valid

Physical Memory

Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x0000	E0	F0	01	11	21	31	41	51	61	71	81	91	A1	B1	C1	D1
0x0010	1E	1F	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D
….																
0x1010	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
0x1020	40	07	41	06	30	06	31	07	00	07	00	00	00	00	00	00
….																
0x2000	21	01	22	03	25	01	22	01	2F	03	28	03	30	03	22	03
0x2010	40	81	41	81	42	81	43	83	00	00	00	00	00	00	00	00
….																
0x2100	30	05	31	01	32	03	33	07	34	00	35	00	36	00	37	00
0x2110	38	00	39	00	3A	00	3B	00	3C	00	3D	00	3E	00	3F	00
….																
0x2200	30	01	31	83	00	01	00	0F	04	00	05	00	06	00	07	00
0x2210	08	00	09	00	0A	00	0B	00	0C	00	0D	00	0E	00	0F	00
….																
0x2500	10	01	00	03	12	85	13	05	14	05	15	05	16	05	17	05
0x2510	18	85	19	85	1A	85	1B	85	1C	85	1D	85	1E	85	00	00
….																
0x2800	50	01	51	03	00	00	00	00	00	00	00	00	00	00	00	00
….																
0x2F00	60	03	28	03	62	00	63	00	64	03	65	00	66	00	67	00
0x2F10	68	00	69	00	00	00	00	00	00	00	00	00	00	00	00	00
0x2F20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
….																
0x30F0	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF
0x3100	01	12	23	34	45	56	67	78	89	9A	AB	BC	CD	DE	EF	00
0x3110	02	13	24	35	46	57	68	79	8A	9B	AC	BD	CE	DF	F0	01
….																
0x4000	30	00	31	06	32	07	33	07	34	06	35	00	43	38	32	79
0x4010	50	28	84	19	71	69	39	93	75	10	58	20	97	49	44	59
0x4020	23	87	20	07	00	06	62	08	99	86	28	03	48	25	34	21