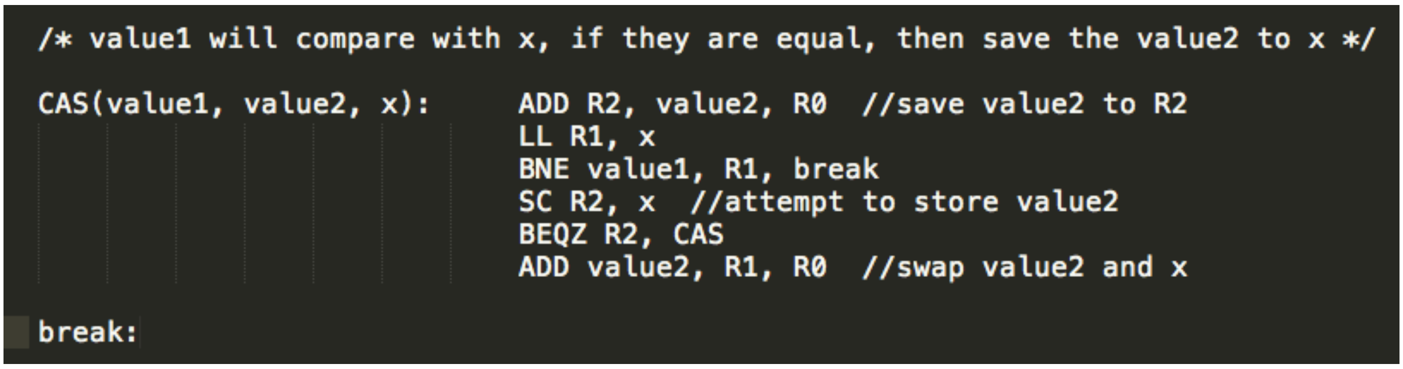
Write Assignment1

--Shaowei Su

1. Solution:

(a) Generally, load-linked will reserve one memory location on aligned words or cache lines, while the store-conditional will store the new value only if the memory location is still reserved, i.e. no intervening write has occurred to the memory location. If many processors perform the LL at the same time, only the first one that manages to put its SC on the bus will actually successfully store. In this way it can emulate all the read-modify-write atomic primitives.

For example, the CAS can be emulated as:



```
/* value1 will compare with x, if they are equal, then save the value2 to x */

CAS(value1, value2, x):     ADD R2, value2, R0  //save value2 to R2
                            LL R1, x
                            BNE value1, R1, break
                            SC R2, x  //attempt to store value2
                            BEQZ R2, CAS
                            ADD value2, R1, R0  //swap value2 and x

break:
```

(b) The ABA problem of CAS is caused between the time read original value and try to swap it with the new value, it could have been changed to something else and back to the original value. In this case the change will not be detected. Since LL/SC can detect the intervening write to reserved location, it does not suffer from the ABA problem.

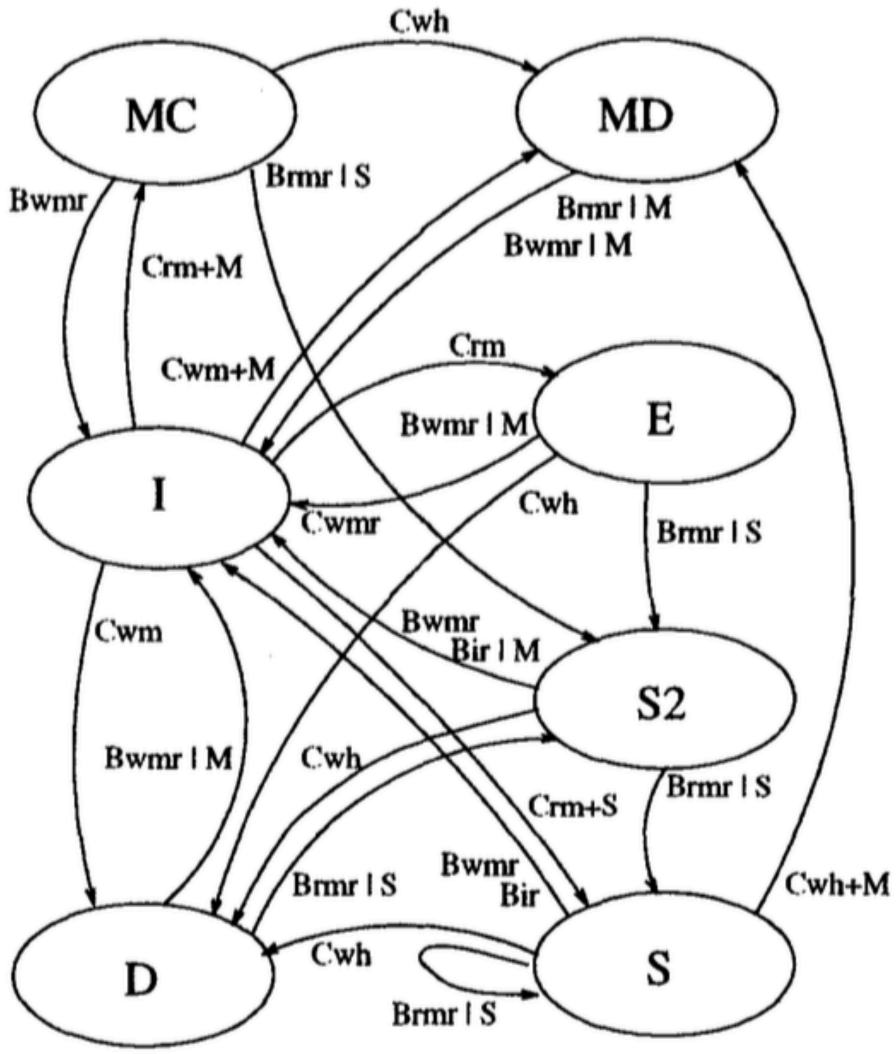
But there are possible drawbacks with LL/SC: in the first place, even a successful LL/SC pair does not guarantee the instructions between them are executed atomically and thus do not constitutes a critical section. Also, the simple LL/SC lock is not a fair lock and does not reduce traffic to a minimum.

2. Solution:

REFERENCE:

Cox, Alan L and Fowler, Robert J, Adaptive cache coherency for detecting migratory shared data, 1993

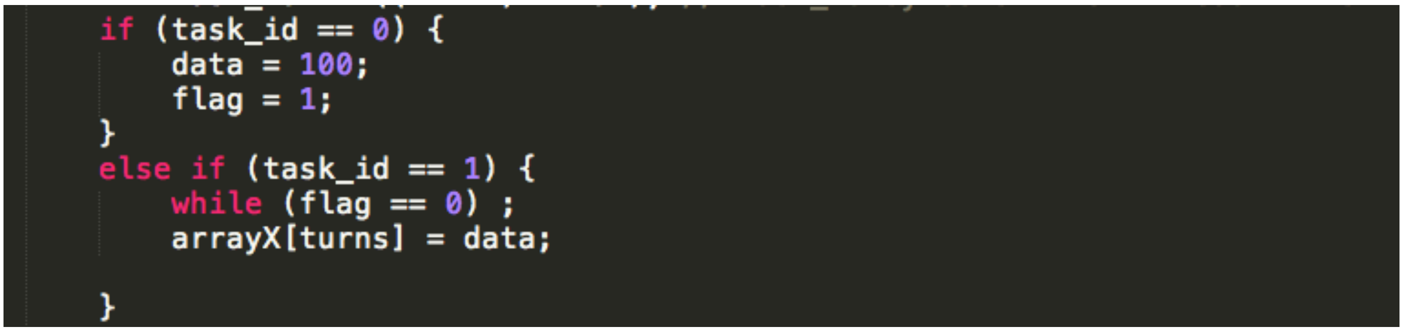
Indicated by the paper above, one an adaptive protocol on bus-based multiprocessor extends MESI protocol could be in the following way. It follows the “replicate-on-read-miss” policy, which means that if a migratory block read before it is written, there would be two separate inter-cache operations to move the blocks from one cache to another. In the adaptive protocols, once one block is classified as migratory, it will be expected as been modified at every processor it visits.



Where D state stands for the "Modified" state in the original MESI protocol; S2 stands for the state that are no more than two cached copies of the block; MC stands for "Migratory-Clean", meaning that has not been modified at its current location while MD vice verse.

More details:
Crm = read miss, Cwm = write miss,
Cwh = write hit, Brmr = read miss requests,
Bwmr = write mess requests, Bir = invalidation requests,

3. Solution:
Compared with PowerPC, which has no specific ordering, TSO supports for the R<->R, R<->W, W<->W ordering. Take W<->W ordering as example, PowerPC won't support for the following code:


```
    if (task_id == 0) {
        data = 100;
        flag = 1;
    }
    else if (task_id == 1) {
        while (flag == 0) ;
        arrayX[turns] = data;

    }
```

Also I use the same code to test on the X86 machine and it turns out that X86s are also TSO.

Codes:

```
/*
	this program aims at the consistency model of the testing machine

	by: shaowei su
*/

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <sys/time.h>

#define ITERATION 100000
int data;
int flag;
int turns = 0;
int arrayX[ITERATION];

int count1 = 0; // count for x = 1, y = 1;



void error_exit(const char *err_str) {
	fprintf(stderr, "%s\n", err_str);
	exit(1);
}

void *work_thread(void *thnum) {
	int task_id = *((int *) thnum); // task_id symbols which thread it is
	if (task_id == 0) {
		data = 100;
		flag = 1;
	}
	else if (task_id == 1) {
		while (flag == 0) ;
		arrayX[turns] = data;
	}
}

int main(int argc, char *argv[]){

	int i, j;
	pthread_t *tid;
	int *id;
	long r;

	int num_th = 2; // two threads are enough for this issue


	id = (int *) malloc (sizeof(int) *num_th);
	tid    = (pthread_t *) malloc (sizeof (pthread_t) *num_th);

	if(!id || !tid)
		error_exit("Out of shared memory");

	for (turns = 0; turns < ITERATION; turns++) {

		data = 0;
		flag = 0;

		for (i=0; i<num_th; i++) {
			id[i] = i;
			r = pthread_create(&tid[i], NULL, work_thread, &id[i]);
			if (r) {
			    fprintf(stderr, "=====!!!Thread %d creation failed... return code from pthread_create() is %ld ...\n", i, r);
				error_exit("fail to create thread");
			}
		}
		for (i=0; i<num_th; i++) {
			r = pthread_join(tid[i], NULL);
			if (r) {
				fprintf(stderr, "=====!!!Thread %d join failed... return code from pthread_create() is %ld ...\n", i, r);
				error_exit("fail to join thread");
			}
		}
	}

	for (turns = 0; turns < ITERATION; turns++) {
		if (arrayX[turns] == 100) {
			count1 ++;
		}
```

```
    }

    printf("RESULTS:\ncount1 = %d\n", count1);

    free(id);
    free(tid);
    return 0;
}
```