CS 537 - Lecture 9

1. Semaphores
2. Implement our own semaphore.
3. Deadlocks.

Concurrency

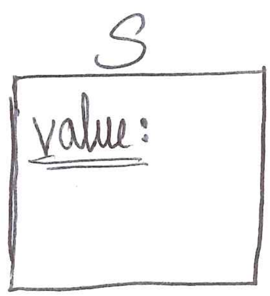
1) Locks
   - HW locks
   - S/W locks

2) CVs
   - ordering
   - wait (c, m)
   - signal (c)

3) PC problem.

```
//
// SEMAPHORE: PSEUDO-CODE
//
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}

sem_wait(sem_t *s) {
    s->value--;
    if (s->value < 0)
        put_self_to_sleep(); // put self to sleep
}

sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}

//
// IMPORTANT: each is done atomically
// (i.e., body of post() and wait() happen all at once)
//
```

sem_init(&s, v);

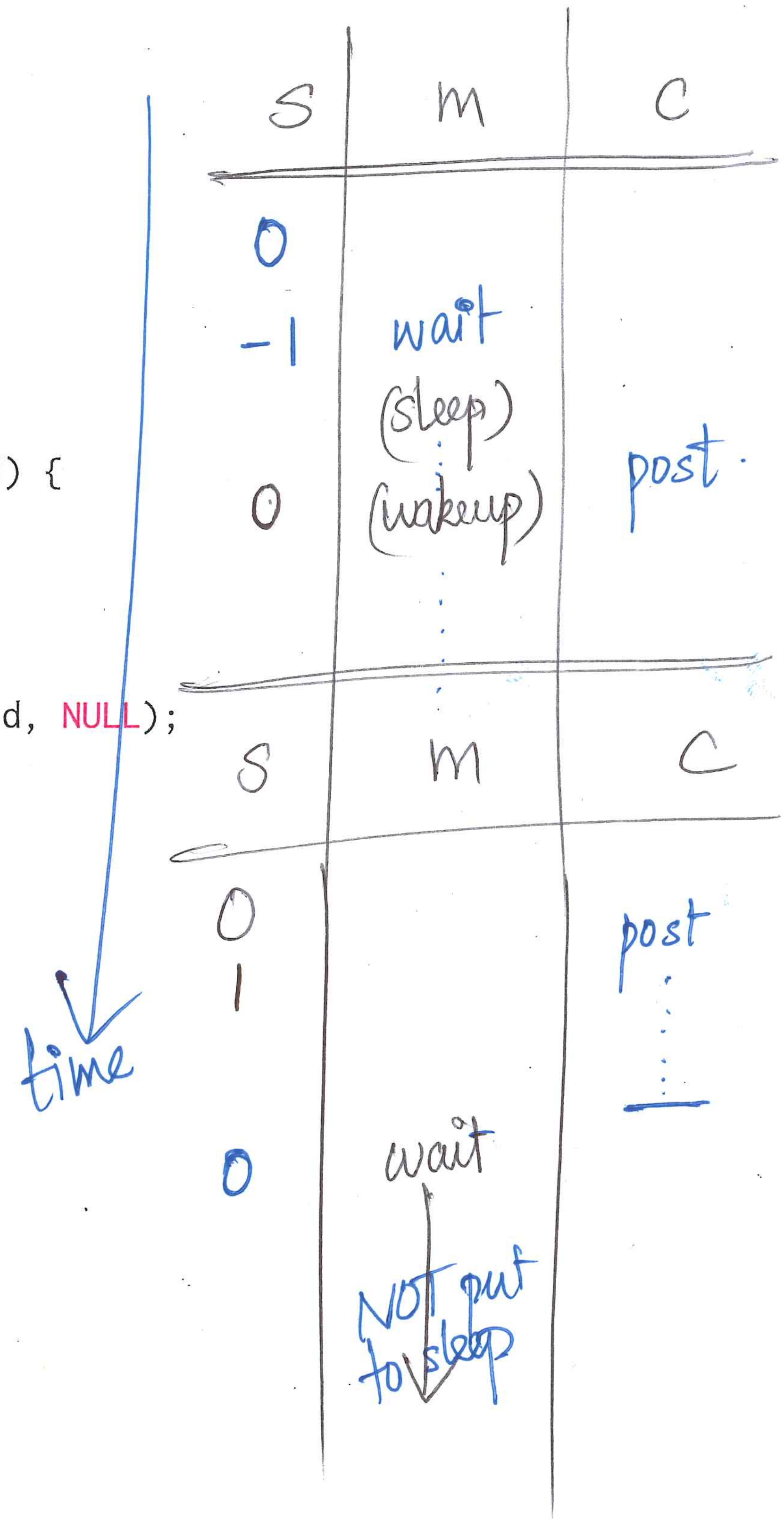


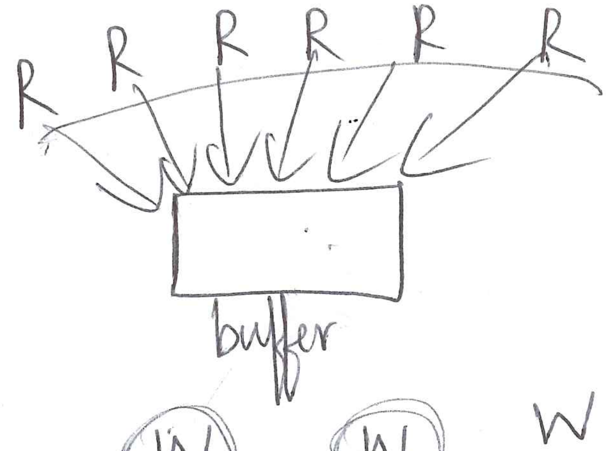
s→value	T_1	T_2	T_3	s→value (3 threads)
1				1
0	wait			0
		wait (sleep)		−1
−1			wait	−2
	CS		time ↓	
0	post →	wake up.		−1
		CS		
1		post.		0
			(wakeup)	
			CS	
			post	1

```c
// Using a semaphore as a mutex lock!
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mythreads.h"
#define PMAX (100)

volatile static int counter = 0;
sem_t lock;

void *worker(void *arg) {
  int i;
  // add something here to provide mutual exclusion
  sem_wait(&s);
  for (i = 0; i < 1e6; i++)
    counter++;
  // something here: end mutex
  sem_post(&s);
  return NULL;
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "usage: sem-lock <numthreads>\n");
    exit(1);
  }
  int threads = atoi(argv[1]);
  if (threads > PMAX) {
    fprintf(stderr, "%d threads is the max\n", PMAX);
    exit(1);
  }

  pthread_t pid[PMAX];
  Sem_init(&lock, ???); // what value should we initialize here?
  int i;

  for (i = 0; i < threads; i++)
    Pthread_create(&pid[i], NULL, worker, NULL);

  for (i = 0; i < threads; i++)
    Pthread_join(pid[i], NULL);

  printf("counter: %d\n", counter);
  return 0;
}
```

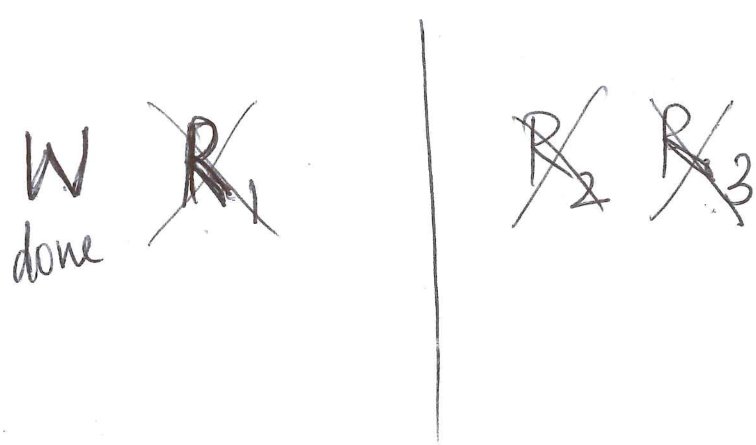
binary semaphores

```c
// Using a semaphore for ordering!
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include "mythreads.h"

sem_t s;  // global, shared

void *child(void *arg) {
  printf("child\n");
  // something here
  sem_post(&s);

  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  printf("parent: begin\n");
  // init here
  sem_init(&s, 0);

  Pthread_create(&p, NULL, child, NULL);
  // something here
  sem_wait(&s);

  printf("parent: end\n");
  return 0;
}
```





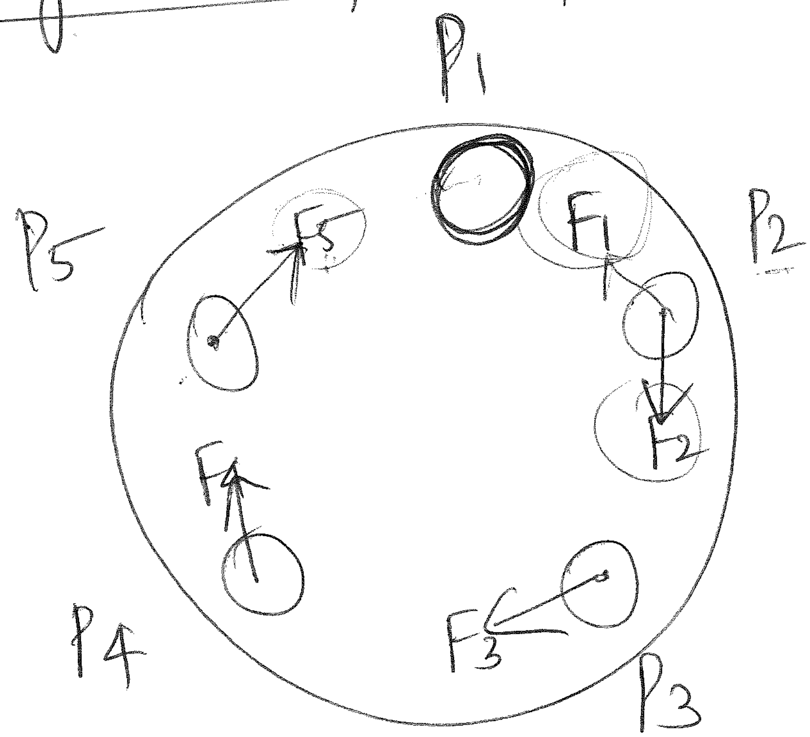

```c
//Reader-Writer Locks

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

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

void rwlock_acquire_readlock(rwlock_t *L) {
  sem_wait(&L->lock);              // a1
  L->readers++;                    // a2
  if (L->readers == 1)             // a3
    sem_wait(&L->writelock);       // a4
  sem_post(&L->lock);              // a5
}

void rwlock_release_readlock(rwlock_t *L) {
  sem_wait(&L->lock);              // r1
  L->readers--;                    // r2
  if (L->readers == 0)             // r3
    sem_post(&L->writelock);       // r4
  sem_post(&L->lock);              // r5
}

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

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



```
// Dining Philosopers Problem
// The basic setup for the problem is this.
// Assume there are five "philosophers" sitting around a table.
// Between each pair of philosophers is a single fork (and thus,
// five total). The philosophers each have times where they think,
// and don't need any forks, and times where they eat.
// In order to eat, a philosopher needs two forks, both the one
// on their left and the one on their right.

// Basic Loop for each philosopher
while (1) {
  think();
  getforks();
  eat();
  putforks();
}
sem_t forks[5];

// Helper Functions
int left(int p) {
  return p;
}

int right(int p) {
  return (p + 1) % 5;
}

// getforks() routine
void getforks() {
  sem_wait(forks[left(p)]);
  sem_wait(forks[right(p)]);
}

// putforks() routine
void putforks() {
  sem_post(forks[left(p)]);
  sem_post(forks[right(p)]);
}
```

$R_1 \; W_1 \; R_2 \; R_3 \; R_{1_{done}} \; \ldots \; R_{2_{done}} \; \ldots \; R_4$

Dining Philosophers problem

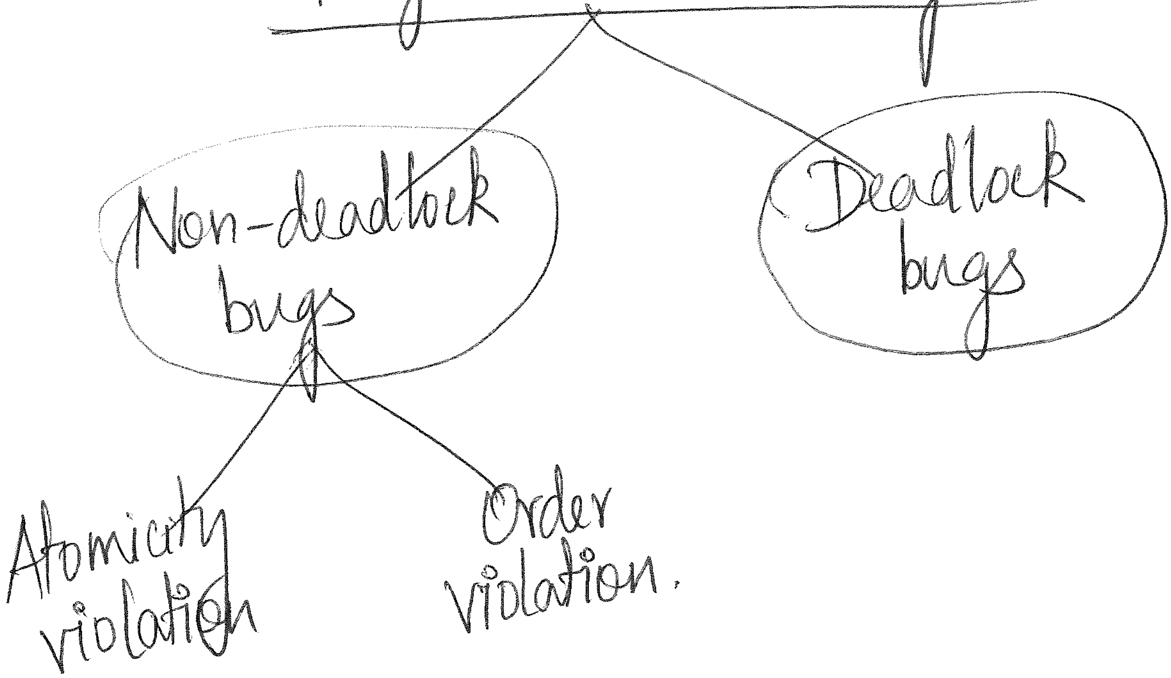


think();
getforks();
eat();
putforks();

$P_1 \rightarrow$ right, left

$P_2 - P_5 \rightarrow$ left, right

Bugs in Concurrency



Atomicity Violation

MySQL.

T1 :

```
if ( thd -> proc_info ) {
    fputs ( thd -> proc_info, ... ) ;
}
```

T2 :

```
thd -> proc_info = NULL;
```

Order violation bugs

T1:

```
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}
```

T2:

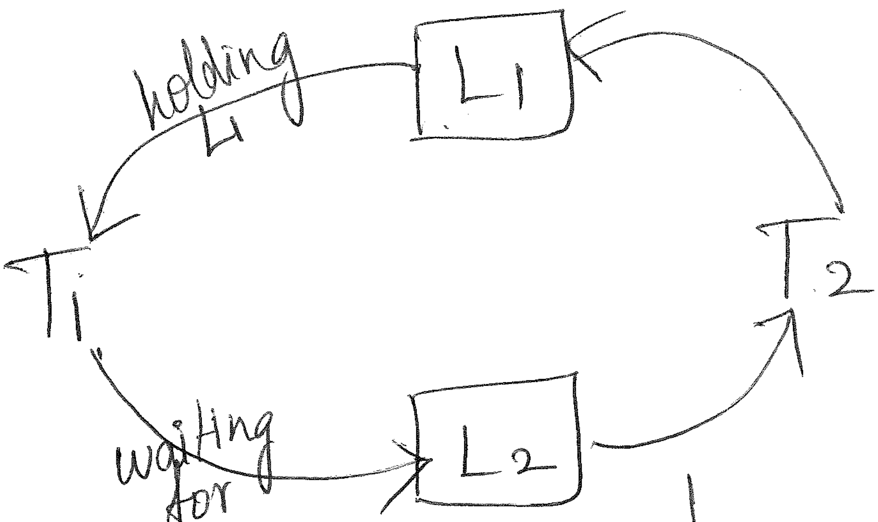
```
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

T2 ... T1

Deadlocks

T1: lock(L1)
lock(L2)

T2: lock(L2)
lock(L1)



Conditions for deadlock

1. Mutual exclusion
2. Hold-and-wait
3. No preemption.
4. Circular Wait.

V1. add(V2);

V2. add(V1);

Prevention

① Circular Wait

L_1, L_2, L_3

→ order

- → total ordering of lock acquisition
- → partial " " "

(L_1, L_2, L_3) (L_7, L_8, L_9)

② Hold-and-wait

T_1	T_2
lock(g);	lock(g);
lock(l_1);	lock(l_2);
lock(l_2);	lock(l_1);
unlock(g);	unlock(g);

③ · No preemption

T1

```
top:
a1   lock(L1);
a2   if(trylock(L2) != 0) {
a3       unlock(L1);
a4       goto top;
     }
```

T2

```
top:
b1   lock(L2);
b2   if(tryLock(L1) != 0) {
b3       unlock(L2);
b4       goto top;
     }
```

$a_1\ b_1\ a_2\ b_2\ a_3\ b_3\ a_4\ b_4 \rightarrow$

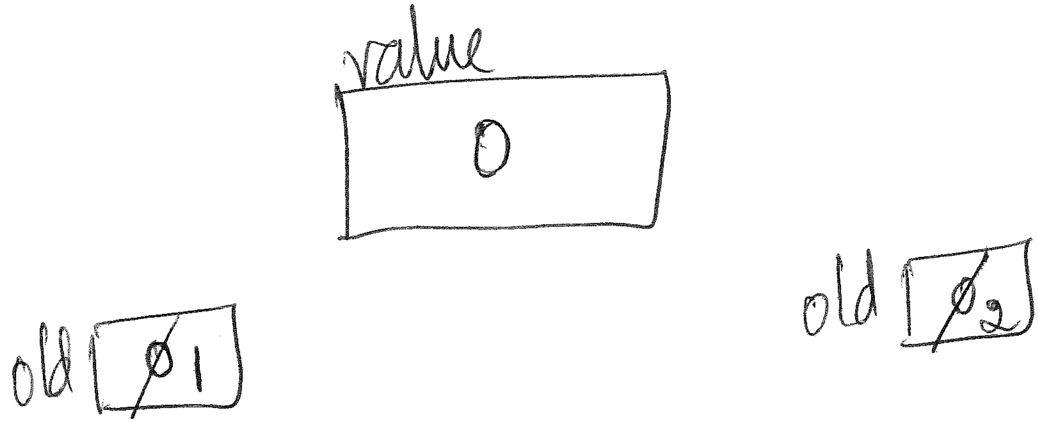
Live-lock

④ Mutual Exclusion

lock-free data structures
wait-free

```
int CompareAndSwap(int *add, int exp, int new)
{
    if (*addr == exp) {
        *addr = new;
        return 1;   // success
    }
    return 0;
}
```

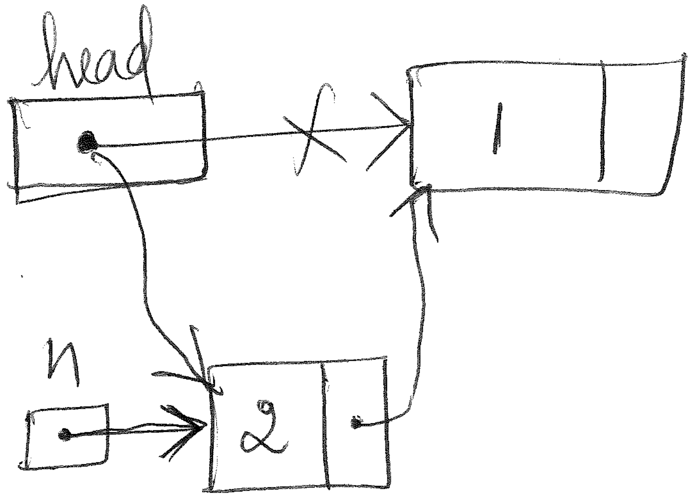
```
void AtomicIncrement(int *value, int amt){
                                    (1)
    do {
        int old = *value;
    } while (CAS(value, old, old+amt) == 0);

}
```

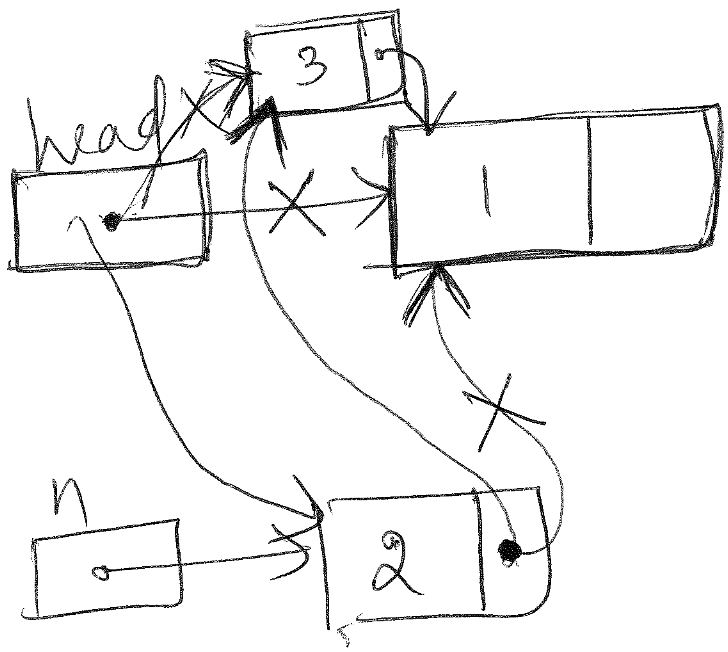


```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    n->next = head;
    head = n;
}
```

```
do {
    n->next = head;
} while (CAS(&head, n->next, n) == 0);
```

(expected: n->next)


head
1
n
2


3
head
1
n
2