Signaling and Hardware Support

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 12

Sept 26, 2014

Reading: A&D 5-5.6
HW 2 due
Proj 1 Design Reviews
Mid Term Monday

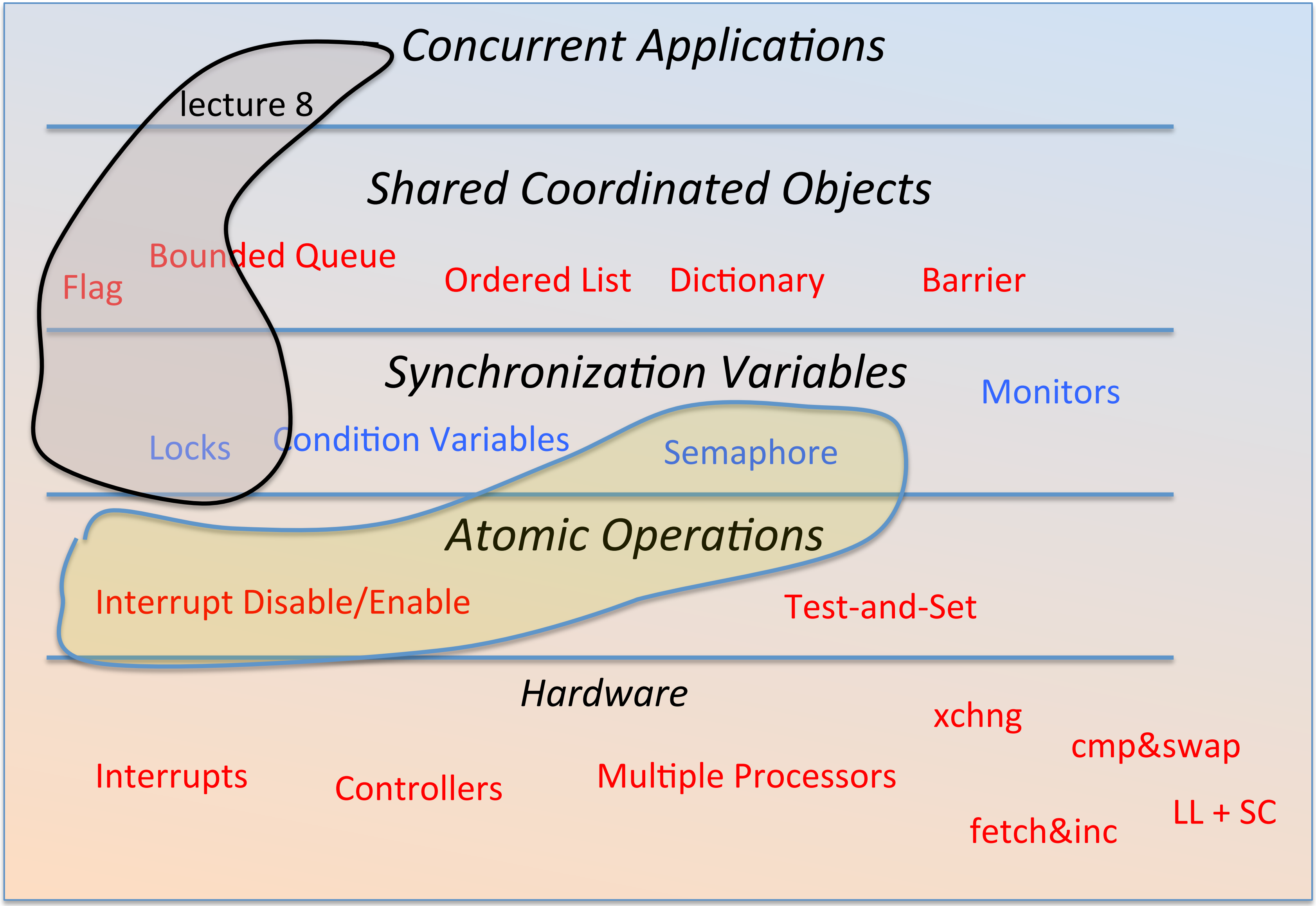
Synchronization Mechanisms

- flags
- semaphores
  - value, waiter*
  - unstructured combination of mutex and scheduling
- locks
  - state, waiter*, owner
  - coarse uniprocessor implementation

  => fine-grain multiprocessor implementation
- condition variables
  - means of conveying scheduling under lock regime

Two Key Roles

Concurrency Coordination Landscape



Concurrent Applications

lecture 8

Shared Coordinated Objects

Bounded Queue

Flag

Ordered List

Dictionary

Barrier

Synchronization Variables

Monitors

Locks

Condition Variables

Semaphore

Atomic Operations

Interrupt Disable/Enable

Test-and-Set

Hardware

xchng

cmp&swap

Interrupts

Controllers

Multiple Processors

fetch&inc

LL + SC

A Lock

- Value: FREE (0) or BUSY (1)
- A queue of waiters (threads*)
  - – attempting to acquire
- An owner (thread)

semaphore has these
- value is int

Incorporate Mutex into shared object

- Methods on the object provide the synchronization
  - Exactly one consumer will process the line

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val) {
    while (1) {
      pthread_mutex_lock(&so->solock);
      if (so->flag == val)
        return 1; /* rtn with object locked */
      pthread_mutex_unlock(&so->solock);
    }
}
int release(so_t *so) {
  return pthread_mutex_unlock(&so->solock);
}
```

Recall: Multi Consumer

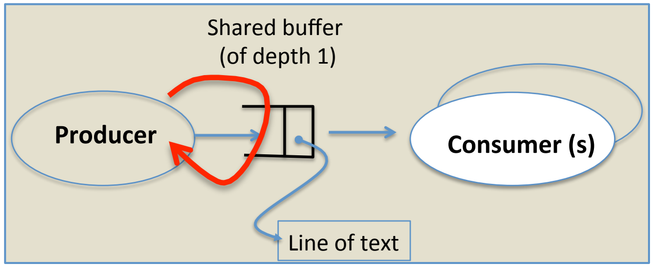
```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  int w = 0;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    waittill(so, 0);            /* grab lock when empty */
    so->linenum = i;            /* update the shared state */
    so->line = line;            /* share the line */
    so->flag = 1;               /* mark full */
    release(so);                /* release the loc */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  waittill(so, 0);              /* grab lock when empty */
  so->line = NULL;
  so->flag = 1;
  printf("Prod: %d lines\n", i);
  release(so);          /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

Coordination

Critical Section for consistency

Eliminate the busy-wait?

- Especially painful since looping on lock/unlock of contended resource



```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val) {
  while (1) {
    pthread_mutex_lock(&so->solock);
    if (so->flag == val)
        return 1; /* rtn with object locked */
    pthread_mutex_unlock(&so->solock);
  }
}
int release(so_t *so) {
  return pthread_mutex_unlock(&so->solock);
}
```

Condition Variables

- Wait: atomically release lock and relinquish processor until signaled
  - may have some spurious wakeups too
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

- *Called only when holding a lock !!!!*

In the object

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  pthread_cond_t flag_cv;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val, int tid) {
  pthread_mutex_lock(&so->solock);
  while (so->flag != val)
    pthread_cond_wait(&so->flag_cv, &so->solock);
return 1;
}

int release(so_t *so, int val, int tid) {
  so->flag = val;
  pthread_cond_signal(&so->flag_cv);
  return pthread_mutex_unlock(&so->solock);
}

int release_exit(so_t *so, int tid) {
  pthread_cond_signal(&so->flag_cv);
  return pthread_mutex_unlock(&so->solock);
}
```

in case of other wake ups (spurious)

release and regain atomically

Critical Section

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  int w = 0;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    waittill(so, 0, 0);         /* grab lock when empty */
    so->linenum = i;            /* update the shared state */
    so->line = line;            /* share the line */
    release(so, 1, 0);          /* release the loc */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  waittill(so, 0, 0);           /* grab lock when empty */
  so->line = NULL;
  release(so, 1, 0);            /* release it full and NULL */
  printf("Prod: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

Change in invariant on exit

```
void *consumer(void *arg) {
  targ_t *targ = (targ_t *) arg;
  long tid = targ->tid;
  so_t *so = targ->soptr;
  int *ret = malloc(sizeof(int));
  int i = 0;;
  int len;
  char *line;
  int w = 0;
  printf("Con %ld starting\n",tid);
  while (waittill(so, 1, tid) &&
         (line = so->line)) {
    len = strlen(line);
    printf("Cons %ld: [%d:%d] %s", tid, i, so->linenum, line);
    release(so, 0, tid);                /* release the loc */
    i++;
  }
  printf("Cons %ld: %d lines\n", tid, i);
  release_exit(so, tid);                /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release? What if release, then wait?

```
int waittill(so_t *so, int val, int tid) {
  pthread_mutex_lock(&so->solock);
  while (so->flag != val)
    pthread_cond_wait(&so->flag_cv, &so->solock);
  return 1;
}
```

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait())
    condition.Wait(lock);
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

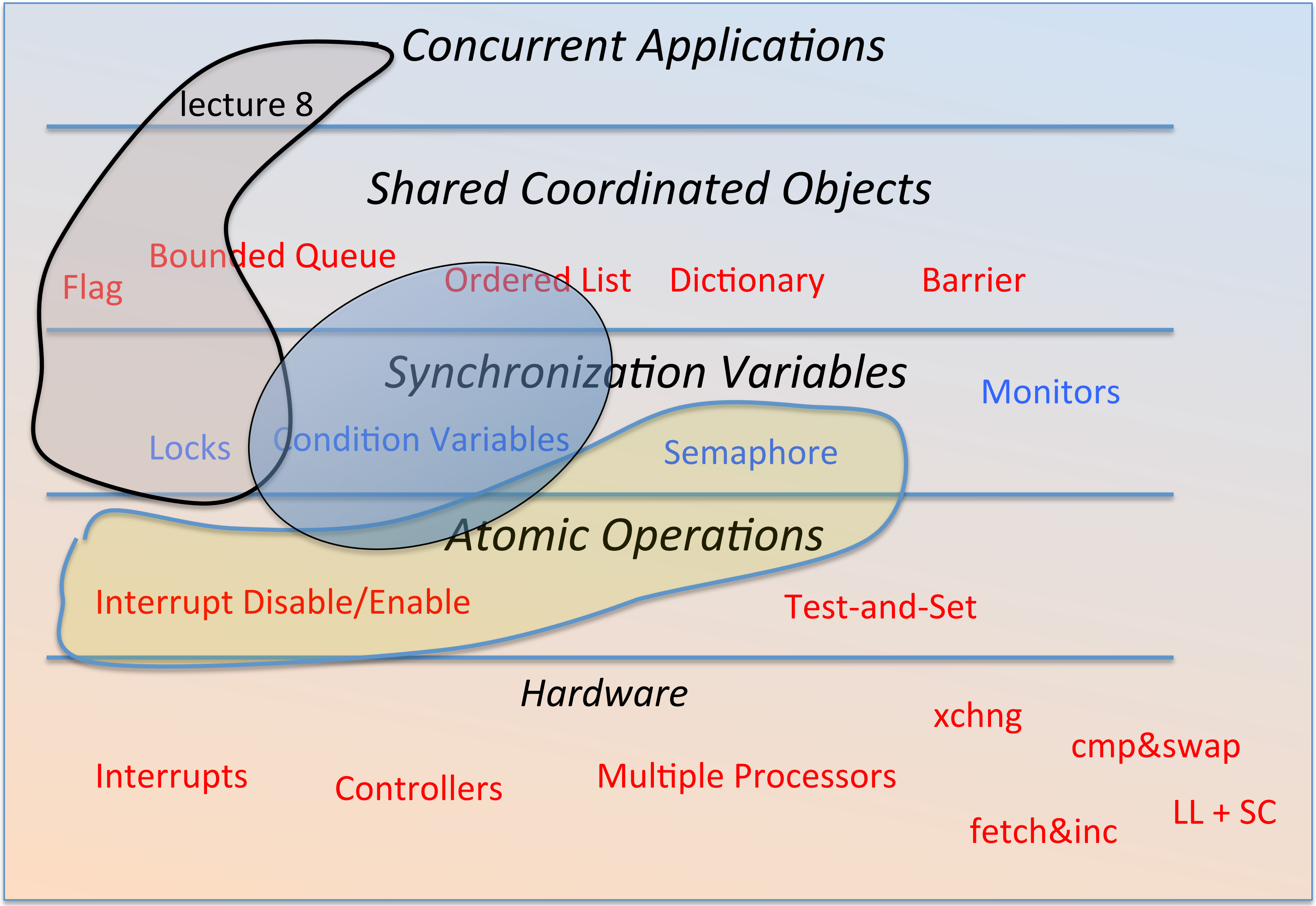
Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) condition.Wait(lock);
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Mesa vs. Hoare semantics

- Mesa (in textbook, Hansen)
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

Concurrency Coordination Landscape



Recall: OS Implementation of Locks

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

Checking and Setting are indivisible
- otherwise two thread could see !BUSY

```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Put at front of ready queue
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

Critical Section

Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
  - Does not work at User level (only system)
  - Doesn’t work well on multiprocessor
    - Disabling interrupts on all processors requires coordination and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory AND write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors & at User level

Examples of Read-Modify-Write

-
  ```
  test&set (&address) {     /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

-
  ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

-
  ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```

Implementing "Locks" with test&set

- Simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // wh
}
Release() {
   value = 0;
}
```

```
test&set (&address) {
  result = M[address];
  M[address] = 1;
  return result;
}
```

- Simple explanation:
  - If free:
    - test&set reads 0 and sets value=1, so now busy.
    - returns 0 so while exits
  - if busy:
    - test&set reads 1 and sets value=1 (no change). while loop continues
  - When we set value = 0, someone else can get "lock"

Why is this less than a Lock ?

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - Inefficient: busy-waiting thread consume cycles
  - Waiting thread takes cycles away from thread holding lock!
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
    - Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Even if OK for locks, definitely not ok for other primitives

What do we want?

- Grab free locks quickly

- otherwise we are going to sleep anyways…

Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
… owner, waitlist
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Locks using test&set vs. Interrupt Disable

```
int value = FREE;
… owner, waiters …
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

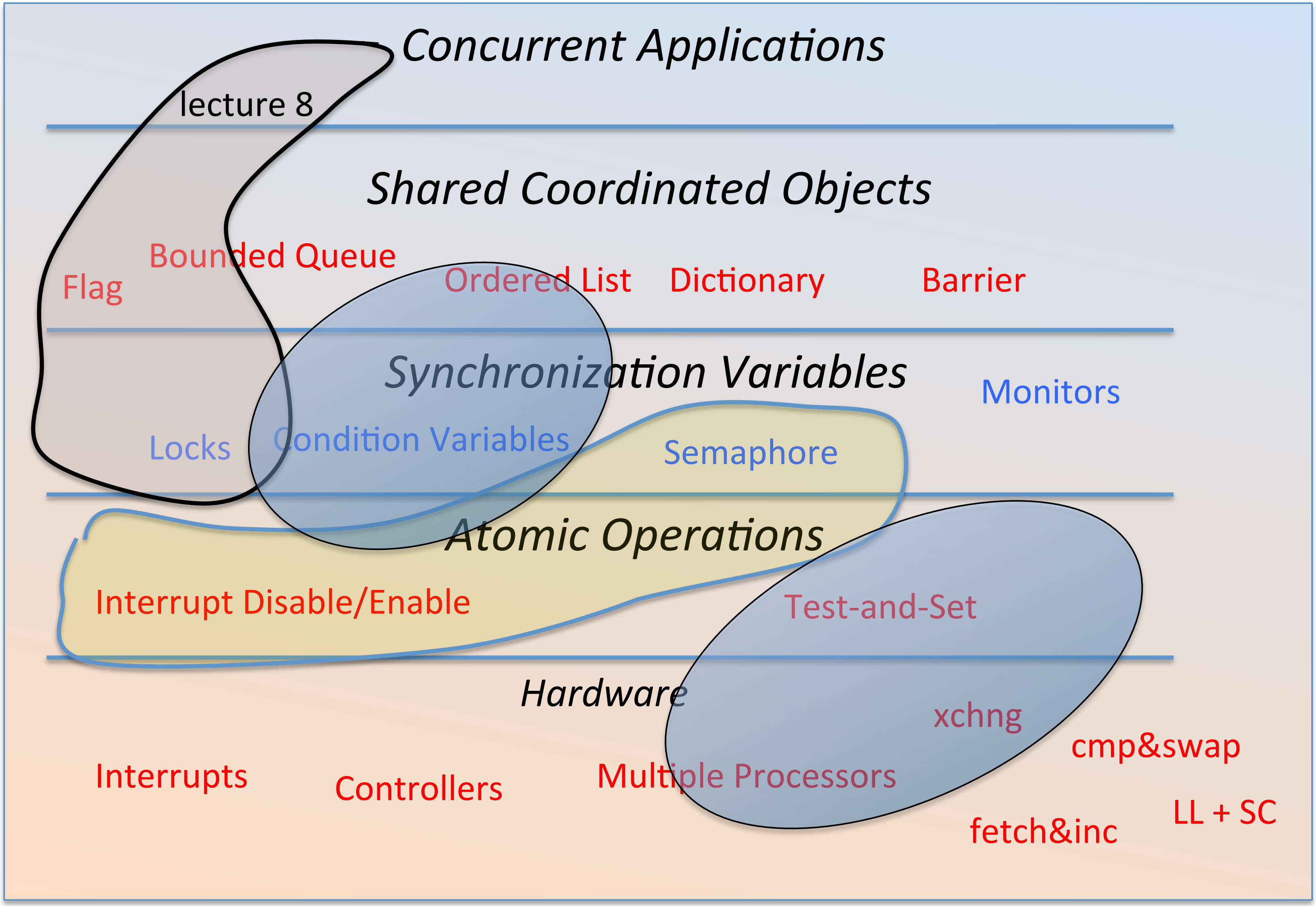
Locks using test&set vs. Interrupts

```
int value = FREE;
… owner, waiters …
```

```
Acquire() {
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // guard = 0;
   } else {
      value = BUSY;
   }
   guard = 0;
}
```

```
Release() {
   while (test&set(guard));
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
}
```

Concurrency Coordination Landscape



Concurrent Applications

lecture 8

Shared Coordinated Objects

Bounded Queue

Flag

Ordered List

Dictionary

Barrier

Synchronization Variables

Monitors

Locks

Condition Variables

Semaphore

Atomic Operations

Interrupt Disable/Enable

Test-and-Set

Hardware

xchng

cmp&swap

Interrupts

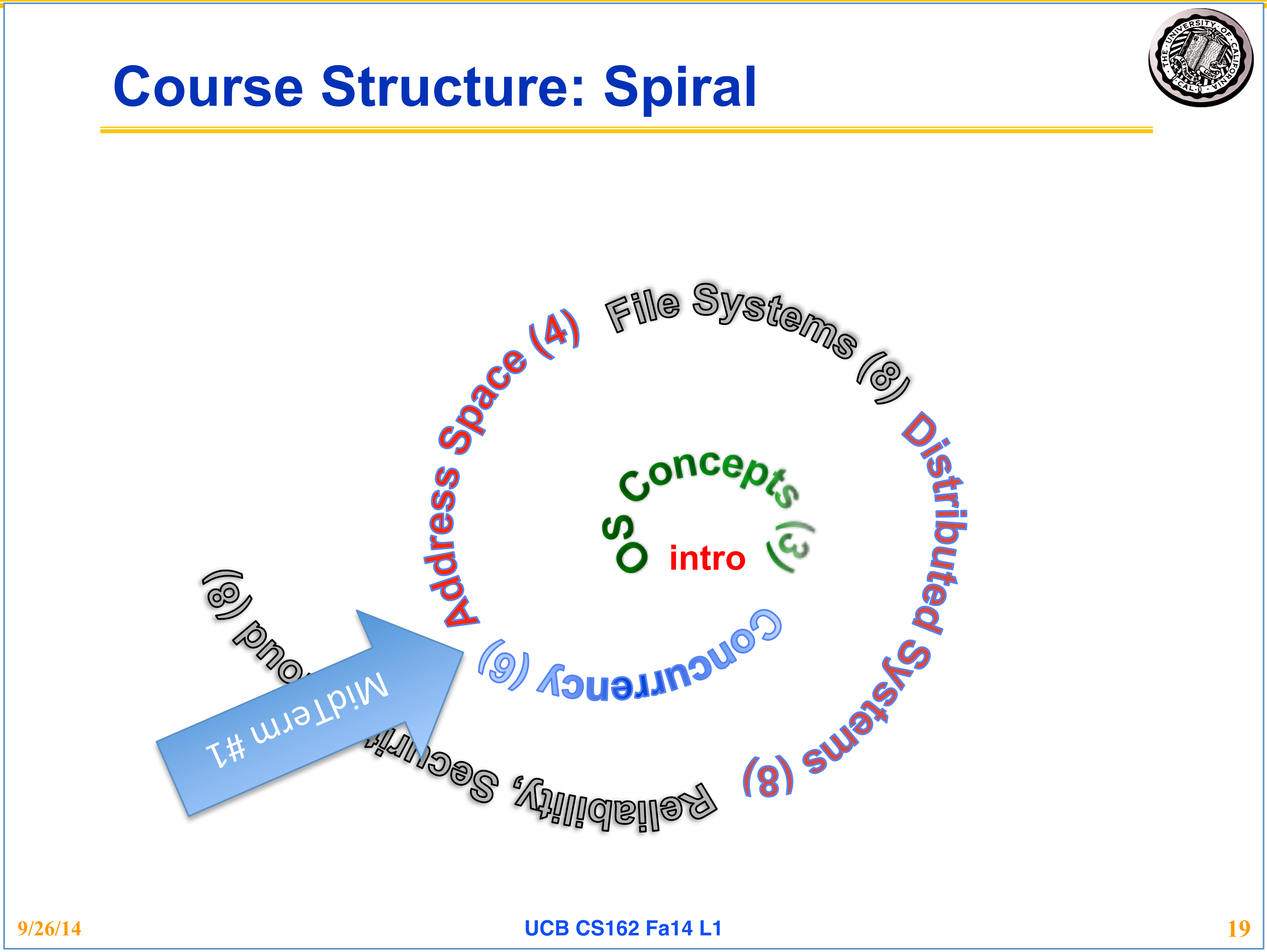
Controllers

Multiple Processors

LL + SC

fetch&inc

You are here …



Course Structure: Spiral

intro

OS Concepts (3)

Concurrency (6)

Address Space (4)

File Systems (8)

Distributed Systems (8)

Reliability, Securit… …oud (8)

MidTerm #1

9/26/14 UCB CS162 Fa14 L1 19