Section 10
Barriers

An application

Graphical Display for a computer game

Problem

The processes require synchronization!

Solution

Organize computation in a sequence of phases, where no thread starts the ith phase until the others have finished the (i-1)th.

Sequential Code

```
while (TRUE) {
    prepare(frame);
    display(frame);
}
```

Code with n concurrent processes available

Code for p_i

```
while (TRUE) {
   prepare(frame[i]);
   display(frame[i]);
}
```

An Application

- A barrier is a way of forcing asynchronous threads to act almost as if they were synchronous.
- When a thread calls the barrier's await() method, it is blocked until all n threads have also "reached the barrier" (i.e., have also called the await() method).
- Barriers should be fast.
  - ❑ minimize the duration between when the last thread reaches the barrier and when the last thread leaves the barrier.
- A thread's notification time is the interval between when some thread has detected that all threads have reached the barrier, and when that specific thread leaves the barrier.

Code with n concurrent processes available
Code for p_i

```
while (TRUE) {
    prepare(frame[i]);
    wait(barrier);
    display(frame[i]);
}
```


Barrier Implementations

Is anything wrong with the implementation on the right?

A Simple Implementation

```
shared int Count = n;     // initial value = n
// a Fetch&Inc/Dec() object with initial value n;
// this object supports also read and write

void await(void) {
    int position = Get&Dec(Count);
    if (position == 1) Count = n;
    else
        while (Count != 0) noop;
}
```

Sense-Reversing Barrier

Main Ideas

A phase's sense is a Boolean value: TRUE for even-numbered phases and FALSE, otherwise. Each barrier has a sense field which indicates the sense of the currently executing phase.

Additionally, each thread has its own local variable keeping the sense of this thread.

Initially, the barrier's sense is the complement of the local sense of all the threads.

```
struct barrier {
    shared int count;
     // a Fetch&Inc/Dec() object with initial value n
     // this object supports also read and write
      boolean sense;
      // initially FALSE;
      boolean mysense[n];
      // initially, psense[i] = TRUE,
      // for each 1 ≤ i ≤ n.
};

void await(struct barrier *B) {
// code for process p_i
    int position = Get&Dec(B->count);
    if (position == 1) {
        B->count = n;
        B->sense = B->mysense[i];
    }
    else {
       while (B->sense!= B->mysense[i])
             noop;
    }
    B->mysense[i] = 1-B->mysense[i];
}
```


Combining Tree Barrier

Main Idea

Split a large barrier into a tree of smaller barriers, and have threads combine requests going up the tree and distribute notifications going down the tree.

A tree barrier is characterized by the number of processes, n, and by the radix r, which is each node's number of children.

We assume there are exactly $n = r^d$ processes, where d is the height of the tree.

Process p_i starts at leaf node $\lfloor i/r \rfloor$.

Contention
A tree-structured barrier reduces memory contention by spreading memory accesses across multiple barriers.

Latency
It is reduced if it is faster to visit a logarithmic number of barriers than decrement a single location.

```
#typedef r <radix>
typedef struct node {
    shared int count;
    boolean sense;
    struct node *parent;
} NODE;

typedef struct barrier {
      NODE *leaf[r^d];
} BARRIER;

int mysense = TRUE;
// persistent local variable of process p_i

void await(BARRIER *B) {
    NODE *nd = B->leaf[i/r];
    wait(nd);
    mysense = !mysense;
}

void wait(NODE *nd) {
    int position = Get&Dec(nd->count);
    if (position == 1) {
        if (nd->parent != NULL )
            wait(nd->parent);
        nd->count = n;
        nd->sense = mysense;
    }
    else {
      while (nd->sense !=mysense)
          noop;
    }
}
```

Combining Tree Barrier

```
void InitializeBarrier(BARRIER *B) {
    int height = 0;
    while (n > 1) {
            height++;  n = n/r;
    }
    root = newcell(NODE);
    root->count =r;   root->parent = NULL;
    root->sense = FALSE;
    Build(B, root, height-1);
}
```

```
void Build(Barrier *B, NODE *parent,
                                int height) {
      static int leaves = 0;

      if (height == 0) {
         B->leaf[leaves++] = parent;
      }
      else {
          for (j = 0; j < r; j++) {
              NODE child = newcell(NODE);
              child->count =r;
              child->parent = parent;
              child->sense = FALSE;
              Build(B,child, height-1);
          }
      }
}
```

Static Tree Barrier

- Each thread is assigned to a node in a tree.
- The thread at a node waits until all nodes below it in the tree have finished, and then informs its parent.
- It then spins waiting for the global sense bit to change.
- Once the root learns that its children are done, it toggles the global sense bit to notify the waiting threads that all threads are done.
- Completing the barrier requires log(n) steps.
- Notification simply requires changing the global sense bit.

Static Tree Barrier

```
typedef struct barrier {
        boolean sense;
        NODE *node[n];  // we assume n = r^d -1
} BARRIER;

void InitializeBarrier(BARRIER *B) {
    int height = 0;
    while (n > 1) { height++;  n = n/r; }
    Build(B, NULL, height);
    B->sense  = FALSE;
}
void Build(Barrier *B,NODE *parent, int height) {
     static int nodes = 0;
     NODE *nd = newcell(NODE);
     nd->parent = parent;
     if (height == 0) {
          nd->count = nd->children = 0;
          B->node[nodes++] = nd;
      }
      else {
          nd->count = nd->children = r;
          B->node[nodes++] = nd;
          for (j = 0; j < r; j++) {
               Build(B, nd, height-1);
          }
      }
}
```

```
#typedef r <radix>
typedef struct node {
     int children;
     int count;
     struct node *parent;
} NODE;

int mysense=TRUE; // persistent local variable of p_i

void await(BARRIER *B) {  // code for p_i
    NODE *nd = B->node[i];
    wait(nd);
}

void wait(NODE *nd) {  // code for p_i
    while (nd->count > 0) noop;
    nd->count = nd->children;
    if (nd->parent == NULL)
          B->sense = !B->sense;
    else {
          Get&Dec(nd->parent->count);
          while (B->sense != mysense) noop;
    }
    mysense = !mysense;
}
```

Termination Detection Barriers

- Work-stealing Schedulers
  - Each thread has its own pool of tasks and works on one of them.
  - If the pool of a thread becomes empty the thread tries to steal some task from the pool of some other processor.
- How can the processes determine termination?
- Each thread is either active or inactive.
- As long as some thread is active, other threads may become active (although they were inactive) by stealing work from this thread.
- Detecting that the computation as a whole has terminated is the problem of determining that at some instant in time there are no longer active threads.
- A termination detection barrier provides operations setActive(v) and isTerminated().
  - Each thread calls setActive(true) to notify the barrier when it becomes active, and setActive(false) to notify the barrier when it becomes inactive.
  - The isTerminated() operation returns TRUE if and only if all threads had become at some earlier instant.

Termination Detection Barriers

- The barrier encompasses a Fetch&Inc/Dec() object initialized to n.
- Each thread that becomes active performs Fetch&Dec() on this object, and each thread that becomes inactive performs Fetch&Inc().
- The computation is deemed to have terminated, when the object has the value 0.

Safety Property

- If isTerminated() returns TRUE, then the computation really has terminated.

Liveness Property

- If the computation terminates, then isTerminated() eventually returns TRUE.

```
shared int Count = n;
// a Fetch&Inc/Dec object

void setActive(boolean active) {
     if (active) Fetch&Dec(Count);
     else Fetch&Inc(Count);
}

boolean isTerminated(void) {
   return (Count == 0);
}
```

Termination Detection Barriers

```
void run() {
    setActive(true);
    task = popBottom(queue[i]);
    while (TRUE) {
        while (task != NULL) {
            run the task;
            task = popBottom(queue[i]);
        }
        setActive(false);
        while (task == NULL) {
            int victim = choose a random integer in range;
            if (! isEmpty(queue[victim])) {
                setActive(TRUE);
                task = popTop(queue[victim]);
                if (task == NULL) setActive(false);
            }
            if (isTerminated()) return;
        }
    }
}
```