Today's Lecture

Parallelism and Concurrency

1. Many Definitions
2. Parallel Architectures
3. Shared-Memory Primitives
   - Unix Primitives
   - A Sample Parallel Program

Definitions

- *Concurrent*

  separate tasks that *may* execute simultaneously;
  a program *structuring* mechanism

- *Parallel*

  separate tasks that *do* execute simultaneously;
  an *optimization* mechanism

- versus *Sequential*

Parallelism Everywhere

In computer systems:

•
•
•

Process	(User/Programmer)
Expression	(Programmer)
Instruction	(Hardware/Compiler)
Functional-Unit	(Hardware)

•
•
•

Many examples from nature

Definitions (cont'd)

- Synchronous
- Asynchronous

- Latency
  (how long)
- Bandwidth
  (how much)

Definitions (cont'd)

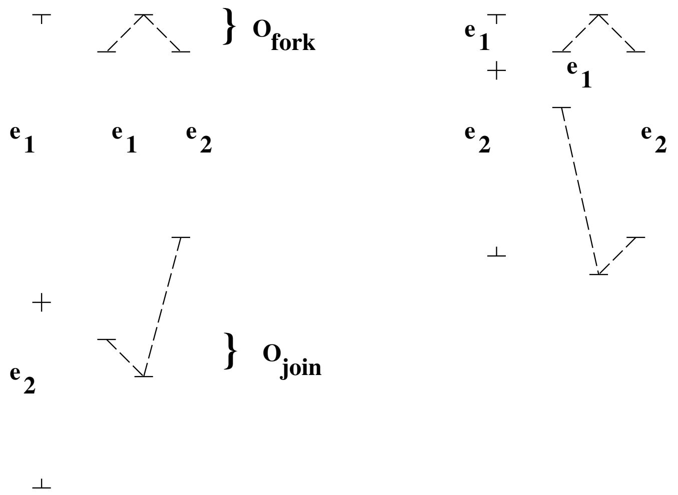
Granularity

Atomicity

Pipelines

Overhead

A Look at Overhead



Multiprocessor Terminology

- Node
- Interconnect
- Shared Memory
- Message Passing

Bus-Based MPs

Symmetric Multi-Processing (SMP)

HW (or SW) must must keep caches *coherent*

Distributed Memory MPs

Advantage over Bus-Based MPs: *Scalable*
Disadvantage: much harder to program

Research: *Shared Virtual Memory*

Networks of Workstations

Similar to distributed-memory MPs, except that parameters (latency, bandwidth, granularity, etc.) are quite different

Programming Bus-Based MPs

Parallel operating systems must supply a programmer interface to mechanisms that:

1. create parallel *threads*

2. *synchronize* such threads

Threads

Threads are a lot like processes, however:

- threads are lightweight (no kernel interaction)
- multiple threads share the same address space (and therefore the same variables, objects, *etc.*)
- ideally, threads should be viewed as an abundant resource

Synchronization

Since threads share the same address space, care must be taken to avoid conflicting accesses to variables

Threads are asynchronous; no assumptions should be made about their rate of execution

Consider two threads, both writing variable `x`:

```
x = 0;              x = 1;
x = x + 1;          x = x + 1;
```

What's the final value of `x`?

Synchronization coordinates execution

Synchronization (cont'd)

Two common synchronization primitives:

- spin locks
- barriers

Two common synchronization bugs:

- race conditions
- deadlock

Synchronization: Spin Locks

A lock is in one of two states: *held* or *free*

A thread trying to *acquire* a held lock, will spin (loop) until the thread holding the lock *releases* it.

A thread may directly acquire a free lock
(the lock then ceases to be free).

```
setlock(lock_for_x);      setlock(lock_for_x);
x = 0;                    x = 1;
x = x + 1;                x = x + 1;
unsetlock(lock_for_x);    unsetlock(lock_for_x);
```

Code delimited by a lock (as in this example) is often called a *critical region* or *monitor*

Synchronization: Barriers

- Barriers "corral" threads
- When a thread reaches a barrier, it waits until all other threads reach that barrier
- After all threads have *checked-in* at the barrier, all threads may proceed
- Barriers can be implemented using spin locks

```
barrier(a_barrier,2);
setlock(lock_for_x);      setlock(lock_for_x);
x = 0;                    x = 1;
x = x + 1;                x = x + 1;
unsetlock(lock_for_x);    unsetlock(lock_for_x);
                          barrier(a_barrier,2);
```

IRIX "threads"


```
NAME
     sproc - create a new share group process

C SYNOPSIS
     #include <sys/types.h>
     #include <sys/prctl.h>

     int sproc (void (*entry) (void *), unsigned inh, ...);

     Type of optional third argument:
     void *arg;

DESCRIPTION
     The sproc system call is a variant of the standard fork(2) call.
     Like fork, the sproc calls create a new process that is a clone of
     the calling process.  The difference is that after an sproc call,
     the new child process shares the virtual address space of the parent
     process (assuming that this sharing option is selected, as described
     below), rather than simply being a copy of the parent.  The parent
     and the child each have their own program counter value and stack
     pointer, but all the text and data space is visible to both processes.
     This provides one of the basic mechanisms upon which parallel programs
     can be built.

        .
        .
        .

     No scheduling synchronization is implied between shared processes:  they
     are free to run on any processor in any sequence.  Any required
     synchronization must be provided by the application using locks and
     semaphores [see usinit(3P)] or other mechanisms.

        .
        .
        .

 DIAGNOSTICS
     Upon successful completion, sproc returns the process id of the new
     process.  Otherwise, a value of -1 is returned to the calling process,
     and errno is set to indicate the error.
```


IRIX spin locks


```
NAME
     ussetlock, uscsetlock, uswsetlock, ustestlock, usunsetlock - spinlock
     routines

C SYNOPSIS
     #include <ulocks.h>

     int ussetlock (ulock_t lock);

     int uscsetlock (ulock_t lock, unsigned spins);

     int uswsetlock (ulock_t lock, unsigned spins);

     int ustestlock (ulock_t lock);

     int usunsetlock (ulock_t lock);

DESCRIPTION
     This set of routines provide a standard test and set facility.  If the
     lock is free then it is atomically locked, and control is returned to
     the caller.  If the lock is already locked, the caller either spins
     waiting for the lock or get queued.  The locks are based in user
     address space thus avoiding any system call overhead.

        .
        .
        .

     ussetlock doesn't return until the lock specified by lock is acquired.

     When invoked with a valid lock, ussetlock and uswsetlock do not return
     until the lock is acquired.  An invalid lock will yield unpredictable
     results.

        .
        .
        .

DIAGNOSTICS
     ussetlock, uswsetlock, uscsetlock, and ustestlock will return a 1 if
     the lock is acquired and a 0 if the lock is not acquired.  usunsetlock
     always returns 0 if successful.  On error, -1 is returned and errno
     is set to indicate the error.
```


IRIX barriers


```
NAME
     barrier, new_barrier, init_barrier, free_barrier - barrier functions

C SYNOPSIS
     #include <ulocks.h>

     barrier_t *new_barrier (usptr_t *handle);

     void free_barrier (barrier_t *b);

     void init_barrier (barrier_t *b);

     void barrier (barrier_t *b, unsigned n);

DESCRIPTION
     These routines provide a simple rendezvous mechanism for shared address
     processes.

     new_barrier takes a usptr_t as an argument to indicate the shared arena
     from which to allocate the barrier.  The usptr_t is a previously
     allocated handle obtained through a call to usinit(3P).

     The barrier function takes a pointer to a previously allocated and
     initialized barrier structure (as returned by new_barrier) and the
     number of processes/sub-tasks to wait for.  As each process enters
     the barrier, it spins (busy wait) until all n processes enter the
     barrier.  At that time all are released and continue executing.

     free_barrier releases all storage associated with b.

     init_barrier resets the barrier to its default state.

     new_barrier will fail if one or more of the following are true:

     [ENOMEM]    There is not enough space to allocate a barrier structure.

     [ENOMEM]    It is not possible to allocate a lock.

SEE ALSO
     sproc(2), usinit(3P), ussetlock(3P), usunsetlock(3P), usnewlock(3P).

DIAGNOSTICS
     Upon successful completion, new_barrier returns a pointer to a barrier
     struct.  Otherwise, a value of 0 is returned to the calling process.
```


Parallel Quicksort


```c
/* pqs.c */

/* this program sorts an array of integers using quicksort */
/* in particular, recursive calls to quicksort execute in parallel */

#include <stdio.h>

#include <parallel/microtask.h>
#include <parallel/parallel.h>

#include <math.h>

long    random();

#define TRUE    1
#define FALSE   0


#define ARRAY_SIZE      40000
#define QUEUE_SIZE      40000

/* declare the array to sort */
typedef long    array_type;
shared array_type       A[ARRAY_SIZE];

/* a queue holds tasks waiting to be done */
/* for quicksort, a task is a piece of the array (low,high) to sort */

typedef struct  {
        int     low, high;
} q_element;

/* shared variables; this stuff controlled via m_lock */
shared q_element queue[QUEUE_SIZE];
shared int head, tail;
shared int tasks;
```

```
/* partition A so that all A[l]..A[n] are <= original A[l]; */
/* and A[n+1]..A[h] are > original A[l] */
int part (A, l, h)
array_type A[];
int l, h;
{
        int     p;
        array_type      tmp;
        int     lp, hp;

        p = A[l];
        lp = l;
        hp = h;
        while (lp < hp)
        {
                while (lp < h && A[lp] <= p)
                        lp++;
                while (hp > l && A[hp] > p)
                        hp--;
                if (lp < hp) {
                        tmp = A[lp];
                        A[lp] = A[hp];
                        A[hp] = tmp;
                        lp++;
                        hp--;
                }
        }
        return hp;
}
```



```
/* a sequential quicksort */
void qs_sequential (A, l, h)
array_type      A[];
int l, h;
{
        array_type      tmp;
        int     mid;

        if (h <= l)
                return;
        if (h - l == 1) {
                if (A[l] > A[h]) {
                        tmp = A[h];
                        A[h] = A[l];
                        A[l] = tmp;
                }
        }
        else {
                mid = part (A, l, h);
                tmp = A[mid];
                A[mid] = A[l];
                A[l] = tmp;
                qs_sequential (A, l, mid-1);
                qs_sequential (A, mid+1, h);
        }
}
```



```
/* a parallel quicksort */
void qs_parallel (A, l, h, id)
array_type A[];
int l, h;
int     id;
{
        array_type      tmp;
        int     mid;

        if (h <= l) {
                /* nothing to sort; this task is done */
                m_lock ();
                tasks--;
                m_unlock ();
                return;
        }
        if (h - l < 50) {
                /* use sequential sort for small pieces pieces */
                qs_sequential (A, l, h);
                m_lock ();
                tasks--;
                m_unlock ();
        }
        else {
                mid = part (A, l, h);
                tmp = A[mid];
                A[mid] = A[l];
                A[l] = tmp;

                /* create a task in our local queue */
                m_lock ();
                tasks += 1;
                queue[tail].low = l;
                queue[tail].high = mid-1;
                tail = (tail+1) % QUEUE_SIZE;
                if (tail == head)
                        error ("queue full!!\n");
                queue[tail].low = mid+1;
                queue[tail].high = h;
                tail = (tail+1) % QUEUE_SIZE;
                if (tail == head)
                        error ("Queue full!!\n");
                m_unlock ();
        }
}
```



```
void do_tasks ()
{
        int     high, low;
        int     my_id;
        int     last_tasks = -1;

        my_id = m_get_myid ();
        while (TRUE) {
                m_lock();
                if (tasks == 0) {
                        m_unlock ();
                        return;
                }
                if (head != tail) {
                        low = queue[head].low;
                        high = queue[head++].high;
                        head %= QUEUE_SIZE;
                        m_unlock ();
                        qs_parallel (A, low, high, my_id);
                }
                else
                        m_unlock ();
        }
}

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

        /* start additional processes in this address space */
        procs = atoi(argv[1]);
        printf ("running with %d processors\n", procs);
        m_set_procs (procs);

        /* initialize the array to sort */
        for (i = 0; i < ARRAY_SIZE; i++)
                A[i] = random () % ARRAY_SIZE;

        /* initialize the queue of tasks */
        head = 0;
        queue[head].low = 0;
```

```
        queue[head].high = ARRAY_SIZE - 1;
        tail = 1;
        tasks = 1;

        m_fork (do_tasks);

        /* show sorted array */
        for (i = 0; i < ARRAY_SIZE; i++)
                printf ("%ld\n", A[i]);

        m_kill_procs ();
}

/* end of pqs.c */
```


Next Lecture

Guest: John Reppy, AT&T Bell Labs

Java Threads and Concurrency