Threads

One way to reduce context switch time is to run lightweight processes within the same address space. This abstraction is often called a *thread*. Look at Figure 2-6.

Global variables and resources are shared between threads within the same process. Each thread has its own stack and program counter (see Fig 2-7).

What system uses threads? Mach, Xinu, OSF/1 (Digital Unix), Java Virtual Machine, almost all new OSes.

Threads vs. Processes

- threads are cheaper to create
- switching between threads in same process is much cheaper
- easier sharing of resources (common memory) between threads
- lack of protection between threads within a process

Notes

- Relative to processes, they should be cheap to create, destroy, and context switch among.
- Good to use on multiprocessors.
- Good to use for applications that naturally have concurrent parts—performance monitor, user interface, data retrieval
- Often use with a multi-threaded server. Create a thread to handle an incoming request.
- Reentrant code issues. Must be careful of routines that return pointers to static storage in memory. Subsequent calls will overwrite this space.

Context Switching Among Threads

- no page or segment table manipulations
- no flushing of the associative memory cache (when switching among threads sharing an address space)
- no copying of data when exchanging messages among threads of the same address space

Synchronization primitives:

- mutex
- condition variables. Can unlock mutex, wait on a condition and then get it back when the condition occurs.

Scheduling

Can use either *preemptive* or *non-preemptive* scheduling. Why one over the other? Same issue as with process scheduling? Closer coordination. Preemptive scheduling does not seem as crucial.

Windows OSes use preemptive threads.

User vs. Kernel threads

Fig 2-13 (from perspective of user threads):

+ can implement on a system not supporting kernel threads
+ fast creation of threads (kernel not involved)
+ fast switching between threads (kernel not involved)
+ customized scheduling algorithm

− must have *jackets* around system calls that may block

− no clock interrupts for time slicing

− use threads when there are many system calls, not much more work to switch threads in the kernel.

− do not gain on a multiprocessor.

− for all threads, worry about non-reentrant code (`errno`).

Have pthreads package on our system. Run-time library.

Shared Memory Fork()

Linux supports a *clone()* system call that creates a new process like fork(), but causes the memory space of the two processes to be shared.

It can be used for separate processes, but its primary purpose is to implement kernel-level threads where Linux 2.4+ has the notion of thread groups.

Multiple processes are shown in the process table if the "H" option is used with "ps".

Mutex Example

```
/* mutexthr.C */
#include <iostream>
using namespace std;
#include <pthread.h>

/* g++ -o mutexthr mutexthr.C -lpthread */

void Deposit(int), BeginRegion(), EndRegion();
pthread_mutex_t mutex;  /* mutex id */
main()
{
    pthread_t idA, idB; /* ids of threads */
    void *MyThread(void *);

    if (pthread_mutex_init(&mutex, NULL) < 0) {
        perror("pthread_mutex_init");
        exit(1);
    }
    if (pthread_create(&idA, NULL, MyThread, (void *)"A") != 0) {
        perror("pthread_create");
        exit(1);
    }
    if (pthread_create(&idB, NULL, MyThread, (void *)"B") != 0) {
        perror("pthread_create");
        exit(1);
    }
    (void)pthread_join(idA, NULL);
    (void)pthread_join(idB, NULL);
    (void)pthread_mutex_destroy(&mutex);
}

int balance = 0;  /* global shared variable */

void *MyThread(void *arg)
{
    char *sbName;

    sbName = (char *)arg;
    Deposit(10);
    cout << "Balance = " << balance << " in Thread " << sbName << "\n";
}
```

```
void Deposit(int deposit)
{
    int newbalance; /* local variable */

    BeginRegion();   /* enter critical region */
    newbalance = balance + deposit;
    balance = newbalance;
    EndRegion();   /* exit critical region */
}

void BeginRegion()
{
    pthread_mutex_lock(&mutex);
}

void EndRegion()
{
    pthread_mutex_unlock(&mutex);
}
```

Synchronization Example

```
#include <iostream>
using namespace std;
#include <pthread.h>
#include <semaphore.h>

/* g++ -o pcthreads pcthreads.C -lpthread -lrt */

sem_t psem, csem; /* semaphores */
int n;
main()
{
    pthread_t idprod, idcons; /* ids of threads */
    void *producer(void *);
    void *consumer(void *);
    int loopcnt = 5;

    n = 0;
    if (sem_init(&csem, 0, 0) < 0) {
        perror("sem_init");
        exit(1);
    }
    if (sem_init(&psem, 0, 1) < 0) {
        perror("sem_init");
        exit(1);
    }
    if (pthread_create(&idprod, NULL, producer, (void *)loopcnt) != 0) {
        perror("pthread_create");
        exit(1);
    }
    if (pthread_create(&idcons, NULL, consumer, (void *)loopcnt) != 0) {
        perror("pthread_create");
        exit(1);
    }
    (void)pthread_join(idprod, NULL);
    (void)pthread_join(idcons, NULL);
    (void)sem_destroy(&psem);
    (void)sem_destroy(&csem);
}

void *producer(void *arg)
{
    int i, loopcnt;
```

```
    loopcnt = (int)arg;
    for (i=0; i<loopcnt; i++) {
        sem_wait(&psem);
        n++;  /* increment n by 1 */
        sem_post(&csem);
    }
}

void *consumer(void *arg)
{
    int i, loopcnt;
    loopcnt = (int)arg;
    for (i=0; i<loopcnt; i++) {
        sem_wait(&csem);
        cout << "n is " << n << "\n"; /* print value of n */
        sem_post(&psem);
    }
}
```

Java Threads

Java has a pre-defined thread object. Can write new threaded objects by extending this class in Java. Threads can be assigned different priorities.

Mutex Example (Java)

```
// in Mutex.java

// javac Mutex.java
// java Mutex

public class Mutex {
    public static void main(String[] args) {
        Account acct = new Account();  // create shared object
        AcctThread thrA = new AcctThread("A", acct); // create A thread
        AcctThread thrB = new AcctThread("B", acct); // create B thread

        thrA.start();                // start A thread
        thrB.start();                // start B thread
        try {
            thrA.join();                // wait for A to finish
            thrB.join();                // wait for B to finish
        } catch (InterruptedException e) {
            System.out.println("Join interrupted");
        }
    }
}


// in AcctThread.java

public class AcctThread extends Thread {
    private String myName;        // string identifier
    private Account acct;        // reference to shared object

    public AcctThread(String whoami, Account acctarg) {
        myName = whoami;
        acct = acctarg;
    }

    public void run() {
```

```
        acct.Deposit(10);
        System.out.println("Balance is " + acct.GetBalance() + " in Thread " + myName
    }
}


// in Account.java

public class Account {
    private int balance;

    public Account() {
        balance = 0;        // initialize balance to zero
    }

    // use synchronized to prohibit concurrent access of balance
    public synchronized void Deposit(int deposit) {
        int newbalance; // local variable

        newbalance = balance + deposit;
        balance = newbalance;
    }

    public synchronized int GetBalance() {
        return balance;          // return current balance
    }
}
```

Synchronization Example (Java)

```
// in PCExample.java

// javac PCExample.java
// java PCExample

public class PCExample {
    public static void main(String[] args) {
        SharedN nobj = new SharedN();  // create shared object
        PCThread thrP = new PCThread("P", nobj); // create producer thread
        PCThread thrC = new PCThread("C", nobj); // create consumer thread

        thrP.start();                // start producer thread
        thrC.start();                // start consumer thread
        try {
            thrP.join();                // wait for producer to finish
            thrC.join();                // wait for consumer to finish
        } catch (InterruptedException e) {
            System.out.println("Join interrupted");
        }
    }
}

// in PCThread.java

public class PCThread extends Thread {
    private String myName;         // string identifier
    private SharedN nobj;         // reference to shared object

    public PCThread(String whoami, SharedN nobjarg) {
        myName = whoami;
        nobj = nobjarg;
    }

    public void run() {
        int i;
        int loopcnt = 5;

        if (myName.equals("P")) {
            // producer thread
            for (i=0; i<loopcnt; i++) {
                nobj.IncrementN();
            }
```

```
        }
        else {
            // consumer thread
            for (i=0; i<loopcnt; i++) {
                System.out.println("n is " + nobj.ReadN());
            }
        }
    }
}


// in SharedN.java

public class SharedN {
    public int n;
    int cValueAvail = 1;        // initially use one value

    public SharedN() {
        n = 0;                        // initialize N to zero
    }

    // use synchronized and conditions to control access to N
    public synchronized void IncrementN() {
        while (cValueAvail > 0) {
            try {
                wait();   // wait for value to be consumed
            } catch (InterruptedException e) {
                System.out.println("Wait interrupted");
            }
        }
        n++;
        cValueAvail = 1;
        notifyAll();  // could also use notify() if only one is waiting
    }

    public synchronized int ReadN() {
        while (cValueAvail == 0) {
            try {
                wait();   // wait for value to be produced
            } catch (InterruptedException e) {
                System.out.println("Wait interrupted");
            }
        }
        cValueAvail = 0;
        notifyAll();  // could also use notify() if only one is waiting
```

```
        return n;  // return value of n (will continue after notify)
    }

}
```