User Space Multithreading
Thread – short for thread of execution/control
Threads

- Used to split a program into separate tasks, one per thread, that can execute concurrently.

- “Light weight process”: multiple threads exist within the context of a single process, sharing the process's code, global information, other resources.

- Threads usually communicate by processing shared global data values.

  - **Compare to the communication between processes**

  - **global shared space** – global data accessed from single global address space shared among the threads.

  - **local private space** – each thread also has its own local private data that is not shared.
Two key functions in C: `pthread_create` and `pthread_join`

```c
void * Calc(void *)

main() {
    int ret, param, tid;
    /*create a new thread*/
    ret=pthread_create(&tid, NULL, Calc, (void *) param);
    continue to do other tasks
    ret=pthread_join(tid, NULL); /*wait for the new thread to return*/
}

void *Calc(void *param) {
    int a, b;
    b=a+(int) param;
}

- **create** - create new thread of execution that runs specified procedure with specified arguments
- **join** - used to wait for the return from a specified thread
```
Scheduling User Space Threads

Unlike process switch, there is no time slice for each thread; a thread needs to call thread switch explicitly.

There is a kernel thread running in the background.

When a thread calls an explicit switch, the kernel thread jumps in and schedules the threads in the system.

A thread can hog the CPU so as to starve other threads.

User space threads usually switch fast.
Kernel Space (OS-supported)
Multithreading
OS-supported Multithreading

```c
void *Calc(void *)

main() {
    int ret, param;
    pthread_t tid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    ret=pthread_create(&tid, &attr, Calc, (void *) param);
    ret=pthread_join(tid, NULL);
}

Calc(void *param) {
    int a, b;
    b=a+(int) param;
}
```
Scheduling OS-supported Threads

Time slicing

System calls

trap

The switching overhead stands between processes and user space threads
Dealing with Multithreading
Problems with concurrency

- **Race conditions** – threads trying to update the same data structure at the same time
  
  \[ \begin{align*}
  \text{thread 1} & \quad \text{thread 2} \\
  A \text{ writes } X & \quad B \text{ writes } X
  \end{align*} \]

- **Deadlock** – to avoid race conditions threads lock data
  
  \[ \begin{align*}
  \text{thread 1} & \quad \text{thread 2} \\
  A \text{ locks } X & \quad B \text{ locks } Y \\
  A \text{ waits for } Y & \quad B \text{ waits for } X
  \end{align*} \]

- **Starvation** – low priority threads don’t get scheduled

- **Nondeterminism** – different executions of multi-threaded program yield different results
Coordination among threads

- asynchronous execution -- threads execute asynchronously, But,

- synchronize at certain points, for example accessing global shared memory locations.

- Two types of synchronisation: mutual exclusion and cooperation

- Techniques for synchronisation
  - Mutex (semaphore, lock) to address mutual exclusion
  - wait and notify to address cooperation
Mutual exclusion

- **Critical section** - section of code that access the global shared data and therefore should be accessed by one thread at a time

- **Mutex** (or *mutex* or *semaphore* or *lock*) - used to enforce *mutual exclusion* of threads in a critical section

  Mutex has two states: *locked* and *unlocked*. Initially unlocked, calling a certain statement locks the critical section (so that only the current thread can access it), executes the code, and then unlocks the mutex

- All shared data in a program, i.e. data used in critical sections, must be associated with an appropriate mutex, so locking the mutex locks access to the data

- If another thread attempts to access a locked mutex, it will block until the mutex is unlocked
Synchronisation in C

```c

void *Calc(void *);

main(){
  ...
  pthread_mutex_init(&my_mutex, NULL) /* before pthread_create*/
  for(ith=0; ith<NUM_THREADS; ith++) pthread_create(..., Calc, ...);
}

void *Calc(void *param){
  ...
  pthread_mutex_lock(&my_mutex);
  critical section;
  pthread_mutex_unlock(&my_mutex);
  ...
}

```

pthread_mutex_t my_mutex; /* declared in global area*/
Synchronisation in Java - Java Monitor

- A Java monitor can be associated with both data and code

- A section of code associated with a monitor is called monitor region.

- A monitor region can only be run by one thread at a time.
Java Monitor

Figure 20-1. A Java monitor.
Threads in Java

- most languages, e.g. C, need a separate library in order to use threads, e.g. POSIX threads

- two ways of implementing threads in Java – extend the `Threads` class or implement the associated `Runnable` interface

- Programmer must extend (implement) a `run` method that specifies what the thread should do

- Create a new instance of the class for each thread and invoke the `start` method

- `start` initializes a new thread of control that executes the `run` method
public class MyThread extends java.lang.Thread
{
    int threadNum;       // data

    public MyThread(int num)      // constructor
    {
        threadNum = num;
    }

    public void run()
    {
        for (int i = 0; i<20; i++)
        {
            System.out.println("Hello from thread "+ threadNum);
        }
    }
}
Example of using Thread Class

```java
public static void main(String[] args)
{
    System.out.println("Simple Thread Demonstration");
    System.out.println("Extending Thread");
    for (int i = 0; i < 4; i++)
    {
        MyThread newThread = new MyThread(i);
        newThread.start();
    }
}
```
Example of using Runnable interface

class RunBasicThread implements Runnable{
    char c;
    RunBasicThread(char c) {
        this.c = c;
    }
    // override run() method in interface
    public void run() {
        for(int i=0; i<100; i++) {
            System.out.print(c);
            try{
                Thread.sleep((int)(Math.random() * 10));
            } catch( InterruptedException e ) {
                System.out.println("Exception caught");
            }
        }
    }
    public static void main(String[] args) {
        RunBasicThread bt = new RunBasicThread('!');
        RunBasicThread bt1 = new RunBasicThread('*');
        // start RunBasicThread objects as threads
        new Thread(bt).start();
        new Thread(bt1).start();
    }
}