Problem

- Escape the curse of blocking!
- A spreadsheet should be able to recompute the values while waiting for user input
- A file server should be able to serve other clients while waiting a disk read to complete
- ...

Solutions

- Multi-processing
- Multi-threading
- One process + event driven programming

What is a Process?

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- Code
- Data
- Stack

What is a Thread?

- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers

Process vs. Thread (1)

- Process: unit of allocation
  - Resources, privileges, etc
- Thread: unit of execution
  - PC, SP, registers
- Each process has one or more threads
- Each thread belong to one process
Process vs. Thread (2)

- Processes
  - Inter-process communication is expensive: need to context switch
  - Secure: one process cannot corrupt another process

![Process A and Process B](image)

S1: Switch from user space to kernel space
S2: Switch context from process A to B
S3: Switch from kernel space to user

User Level vs. Kernel Level Threads

- User level: use user-level thread package; totally transparent to OS
  - Light-weight
  - If a thread blocks, all threads in the process block
- Kernel level: threads are scheduled by OS
  - A thread blocking won’t affect other threads in the same process
  - Can take advantage of multi-processors
  - Still requires context switch, but cheaper than process context switching

Thread Creation Example (Java)

```java
final List list; // some sort unsorted list of objects
// A Thread class for sorting a List in the background
class Sorter extends Thread {
  List l;
  public Sorter(List l) { this.l = l; } // constructor
  public void run() { Collections.sort(l); } // Thread body
}

// Create a Sorter Thread
Thread sorter = new Sorter(list);
// Start running the thread; the new thread starts running the run method above
while the original thread continues with the next instructions
sorter.start();
System.out.println("I'm the original thread");
```

Thread Implementation

- Combining kernel-level lightweight processes and user-level threads
  - LWPs are transparent to applications
  - A thread package can be shared by multiple LWPs
  - A LWP looks constantly after runnable threads

![Thread state and Lightweight process](image)

User-level, Kernel-level and Combined

![Diagram of User-level, Kernel-level, and Combined threads](image)

Figure 4.6 - User-Level and Kernel-Level Threads

(Operating Systems, Stallings)
**Event Driven Programming**

- Organize program as a finite state automaton
- Never call blocking functions
- When a function returns
  - Need a callback mechanism to invoke process, or
  - Process can periodically pool for return values
- Very efficient; zero context switching!
- Hard to program

**Trade-offs**

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Event driven (Finite state machine)</td>
<td>Parallelism, nonblocking system calls</td>
</tr>
</tbody>
</table>

**The X-Window System**

- The basic organization of the X Window System

**Servers: General Design Issues**

a) Client-to-server binding using a daemon as in DCE  
b) Client-to-server binding using a superserver as in UNIX
Code Migration: Motivation

- Performance
  - Move code on a faster machine
  - Move code closer to data
- Flexibility
  - Allow to dynamically configure a distributed system

Dynamically Configuring a Client

- The client first fetches the necessary software, and then invokes the server

Code Migration Model

- Process model for code migration (Fugetta et al., 98)
  - Code segment: set of instructions that make up the program
  - Resource segment: references to external resources
  - Execution segment: store current execution state
- Type of mobility
  - Weak mobility: migrate only code segment
  - Strong mobility: migrate execution segment and resource segment

Models for Code Migration

- Mobility mechanism
  - Weak mobility
  - Receiver-initiated mobility
  - Sender-initiated mobility
  - Strong mobility

Migration and Local Resources

- Types of process-to-resource binding
  - Binding by identifier (e.g., URL, (IPaddr:Port))
  - Binding by value (e.g., standard libraries)
  - Binding by type (e.g., monitor, printer)
- Type of resources
  - Unattached resources: can be easily moved (e.g., data files)
  - Fastened resources: can be used but at a high cost (e.g., local databases, web sites)
  - Fixed resources: cannot be moved (e.g., local devices)

Migration and Local Resources

- Actions to be taken with respect to the references to local resources when migrating code
  - GR: establish a global system wide reference
  - MV: move the resource
  - CP: copy the value of resource
  - RB: rebind the process to locally available resource
Migration in Heterogeneous Systems

- Maintain a migration stack in an independent format
- Migrate only at certain points in the program (e.g., before/after calling a procedure)

Weak Mobility in D'Agents (1)

- A Tel agent in D'Agents submitting a script to a remote machine (adapted from [Gray '95])

```plaintext
proc factorial n { 
  if ($n <= 1) { return 1; } # fac(1) = 1
  expr $n * [factorial expr $n - 1] # fac(n) = n * fac(n - 1)
}

set number .... # tells which factorial to compute
set machine .... # identify the target machine
agent_submit $machine --procs factorial --vars number --script [factorial $number]
agent_receive ... # receive the results (left unspecified for simplicity)
```

Strong Mobility in D'Agents (2)

- A Tel agent in D'Agents migrating to different machines where it executes the UNIX `who` command (adapted from [Gray 95])

```plaintext
all_users $machines
proc all_users machines { 
  set list { } # Create an initially empty list
  set set machines [exec who]
  foreach $machine $set { # Consider all hosts in the set of given machines
    agent_jump $machine # Jump to each host
    exec who
    append list $users # Append the results to the list
  }
  return $list # Return the complete list when done
}

set machines .... # Initialize the set of machines to jump to
set this_machine # Set to the host that starts the agent

# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in $machines.
agent_submit $this_machine --procs all_users
  --vars machines
  --script {all_users $machines}
agent_receive ... # receive the results (left unspecified for simplicity)
```