Topic 3: Run-Time Environment

- Memory Model
- Activation Record
- Call Convention
- Storage Allocation
- Runtime Stack and Heap
- Garbage Collection
ABET Outcome

• Understand the software conventions necessary to support various source languages, including data representation, storage allocation for the various storage classes of variables, visibility rules, call sequence, entry, exit, and return.

• Ability to apply the knowledge of run time support system to trace the program execution.
Reading List

- Slides
- Dragon book: chapter 7
- Muchnick’s book: Chapter 5
- Other readings as assigned in class or homeworks
What Does Runtime Environment Do

The execution environment provided by the system software services. The runtime environment dictates how executable code is loaded into memory, where data is stored, and how routines call other routines and system software routines.
Jobs of Runtime Environment

**Issues**

- Software conventions, such as data layout and allocation
- Mechanism to access variables
- Procedure calling
- Interface to OS, libraries, I/O devices, etc.

**Names versus data objects**

Same name can refer to different data objects during execution; the runtime environment provides the mapping

**Procedure activations**

- Each time a procedure is called, a new activation of that procedure occurs within an environment (who call it? where it was called from? What declarations are active at call site?)
- Recursion: a new activation of same procedure can start before an early activation has ended
Runtime Environment

• Architecture
  -- Instruction Set Architecture (ISA)
  -- Data Types
  -- Register Usage Convention
  -- Memory System
  -- Addressing Model (the organization of data in registers and memory)

• Operating system
  -- Memory organization and management
  -- Stack Usage Convention
  -- Special Register Initialization

• Compiler
  -- Data layout
  -- Activation Record
  -- Calling convention
Related Topics

- Memory Model
- Memory Layout
- Register usage
- Procedure calling convention
- Runtime Stack and Heap
MIPS Architecture Overview

• **Processor Features**
  Full 32-bit operation, efficient pipelining, on-chip TLB (Translation Lookaside Buffer) for virtual-to-physical memory mapping, cache control, coprocessor interface

• **Registers**
  32 general 32-bit registers, a 32-bit Program Counter

• **ISA (Instruction Set Architecture)**
  Each instruction is 32 bits long, three instruction formats (intermediate I-type, jump J-type and register R-type)

• **Memory Management System**
  Physical memory addressing range of 4 Gbytes (32 bits). The virtual address space is divided into 2 Gbytes for users and 2 Gbytes for the kernel.

• **Addressing Model**
  Defines a 64-bit doubleword, a 32-bit word, a 16-bit halfword and an 8-bit byte. Byte addressing. The byte ordering can be configured in either Big-endian or Little-endian format.
# MIPS Register Usage Convention

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Mnemonic Name</th>
<th>Conventional Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>zero</td>
<td>Permanently 0</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>Assembler Temporary (reserved)</td>
</tr>
<tr>
<td>$2, $3</td>
<td>$v0, $v1</td>
<td>Value returned by a subroutine</td>
</tr>
<tr>
<td>$4-$7</td>
<td>$a0-$a3</td>
<td>Arguments to a subroutine</td>
</tr>
<tr>
<td>$8-$15</td>
<td>$t0-$t7</td>
<td>Temporary (not preserved across a function call)</td>
</tr>
<tr>
<td>$16-$23</td>
<td>$s0-$s7</td>
<td>Saved registers (preserved across a function call)</td>
</tr>
<tr>
<td>$24, $25</td>
<td>$t8, $t9</td>
<td>Temporary</td>
</tr>
<tr>
<td>$26, $27</td>
<td>$k0, $k1</td>
<td>Kernel (reserved for OS)</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$30</td>
<td>$fp</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>Return Address (Automatically used in some instructions)</td>
</tr>
</tbody>
</table>
Memory Model

Modern computer systems nearly always use virtual memory. The purpose of virtual memory is to make it appear as if a program has the full address space available. So the programming model has the full address space.

• User memory space (Virtual memory)
• OS memory space (Physical memory)
Why Virtual Memory

• Why Virtual Memory
  ▪ Limited physical memory size
    ▪ 64MB to 2GB
  ▪ Unlimited virtual memory size
    ▪ Each process may have 4GB
    ▪ Many processes in the system
Virtual Memory/Physical Memory

- Physical memory and virtual memory broke into fixed size pages;
- Each physical page holds a virtual page (may come from different processes);
- Only the active pages of each process reside in physical memory, physical memory works as cache of virtual memory (disk);
- Other pages stay on disk.
Mapping Virtual Memory to Physical Memory

Physical Memory

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
<th>Page 5</th>
<th>Page 6</th>
<th>Page 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>OS</td>
<td>U1/P0</td>
<td>U2/P3</td>
<td>U1/P3</td>
<td>U1/P7</td>
<td>U1/P6</td>
<td>U2/P1</td>
</tr>
</tbody>
</table>

Process 1 VM

| U1/P0 | U1/P1 | U1/P2 | U1/P3 | U1/P4 | U1/P5 | U1/P6 | U1/P7 |

Process 2 VM

| U2/P0 | U2/P1 | U2/P2 | U2/P3 | U2/P4 | U2/P5 | U2/P6 | U2/P7 |

On Disk

2/29/2008
Virtual Memory Layout

Virtual address

Stack

High

Reserved for kernel

Stack segment

Data segment

Dynamic data

Static data

Text segment

Reserved

Low

VM Layout – (User View)

VM Layout – (OS View)

Stack

Heap

BSS

Global Data

Data

Text

Reserved for kernel

Virtual address

Stack

Heap

BSS

Global Data

Data

Text
A procedure is activated when it is called.
Activation Lifetime

- Lifetime of an activation of procedure $P$ is the sequence between first and last steps in execution of procedure body, including time spent executing any procedures called from $P$.

- Structure program languages allow only nested procedure lifetimes
  - Allows use of stack to define runtime environment
  - Can show relations in procedure activation tree
Activation Tree

A data structure that represents the activations during the running of an entire program.

Assumption

Each execution of a procedure starts at the beginning of the procedure body and eventually returns control to the point immediately following the place where the procedure was called.
Activation Tree (Con’t)

Features of activation tree

- Each node is a procedure activation, each edge represents opening an activation while the parent activation is still open. The root node is the main program activation.
- Flow of control in procedure calls corresponds to a preorder (depth-first) traversal of the activation tree.
- Flow of control in procedure returns corresponds to a postorder traversal of the activation tree.
- A node $A$ is to the left of node $B$ in the tree means the lifetime of $A$ occurs before the lifetime of $B$.
- We can use a control stack to keep track of live procedure activations.
Activation Tree: Example

\begin{verbatim}
fact(x)
{... fact(x-1);}
fib(y)
{int i; ... fib(y-1)+fib(y-2);}
main()
{fact(5);
fib(4);}
\end{verbatim}
Activation Record

- A data structure containing the variables belonging to one particular scope (e.g. a procedure body), as well as links to other activation records.
- Activation records are usually created (on the stack) on entry to a procedure and destroyed on exit.
Contents in Activation Record

- Static link to encompassing scope
- Local variables
- Return address to branch to caller’s code
- Temporaries
- Saved register contents
- Storage for outgoing arguments
- Stored in fixed order in frame. Frame pointer points to frame beginning. Fields are referenced using offset

The sizes of almost all the contents can be determined at compiler time – except for dynamic arrays or other variable length data objects
### Activation Record Layout

#### Layout form dragon book

<table>
<thead>
<tr>
<th>Actual parameters</th>
<th>Returned values</th>
<th>Optional control link</th>
<th>Optional access link</th>
<th>Save machine status</th>
<th>Local Data</th>
<th>Temporaries</th>
</tr>
</thead>
</table>

2/29/2008 
\course\cpeg421-08s\Topic-3.ppt

22
Frame pointer and stack pointer

- When a procedure starts running, the frame pointer and the stack pointer contain the same address.

- While the procedure is active, the frame pointer, points at the top of the stack (the last word that was pushed onto the stack, the top of the stack.)

- The stack pointer may be changed its value while a procedure is active but the frame pointer does not change its value while a procedure is active. The variables will always be the same distance from the unchanging frame pointer.
Activation Record Layout (Con’t)

- Incoming arguments
  - (frame pointer) fp
- Actual Parameters
- Return Values
- Control Link
- Access Link
- Saved machine status
- Local Data
- Temporaries

- Outgoing arguments
  - (stack pointer) sp

- Previous frame
- High addr.
- Current frame
- Low addr.
Stack of Activation Records

```plaintext
fact(x)
{... fact(x-1);}
fib(y)
{int i; ... fib(y-1)+fib(y-2);}
main()
{fact(5);
  fib(4);}
```

```
Main: fact(5) fib(4)
  fact(4) fib(3) fib(2)
  fact(3) fib(2) fib(1)
  fact(2)
  fact(1)
```

```
Top of stack: From OS
```

```
Top of stack: From main
```

```
Top of stack: From fib(4)
```

```
Top of stack: From fib(3)
```

```
Top of stack: From fib(2)
```

```
Top of stack: From fib(1)
```
Stack of Activation Records

fact(x)
{... fact(x-1);}
fib(y)
{int i; ... fib(y-1)+fib(y-2);}
main()
{fact(5);
 fact(4) fib(3) fib(2)
fact(3) fib(2) fib(1)
 fact(2)
 fact(1)