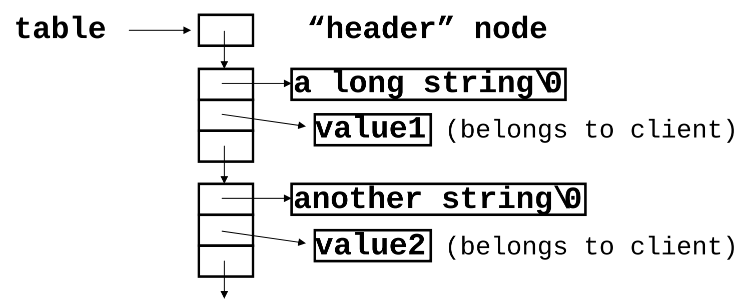
Memory Management in Program Design

CS 217

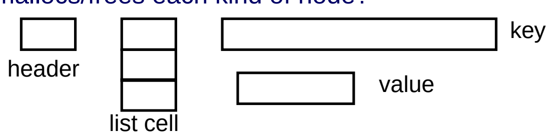
ADT Implementation

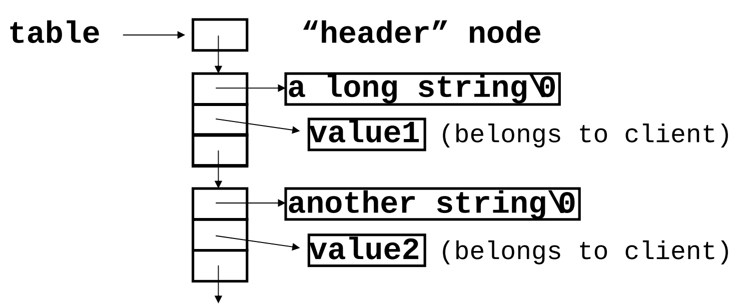
- Recall the simple implementation of the symtable ADT:



Memory Management Issues

- Does ADT or client "own" the data?
  - Who mallocs/frees each kind of node?





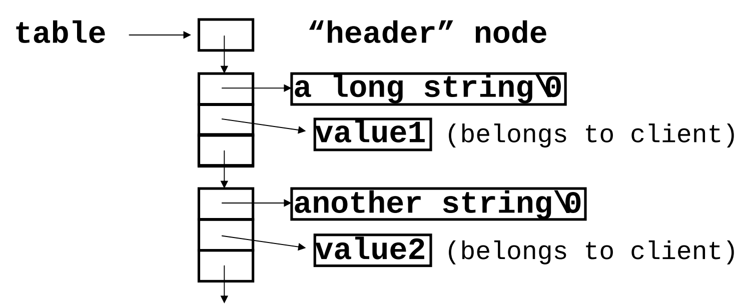
Who Free What

- What happens if,

`{SymTable_T table; . . . free(table);}`

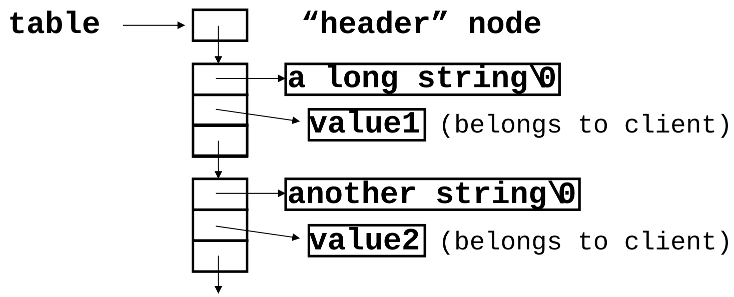
then the list cells don't get freed!

- So, ADT must "own" headers and list cells



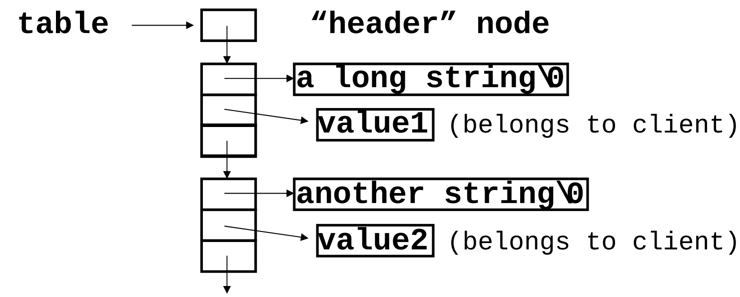
Who Free What, cont'd

- ADT just sees **`void *value;`**
- Value pointer might be root of big data structure, all the pieces need to be freed.
- Thus, client must "own" the value nodes.



Who Owns The Key?

- Both client and ADT "know" about **`char *key;`**
- Therefore, we are faced with a design choice
- Choice 1: client owns the key.
  - Consequence: must call SymTable_put only with a string that will last a long time. *(But our client didn't do that!)*

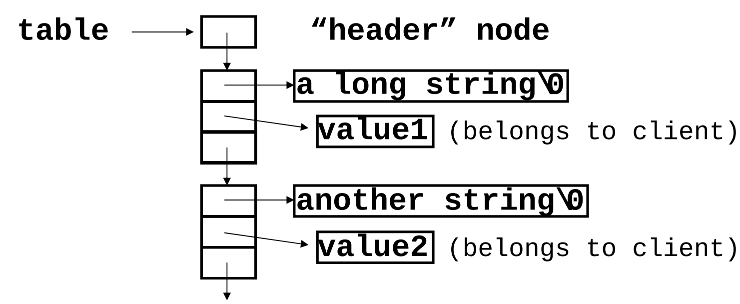


Previous Example Overwrites "line"

```
int main(int argc, char *argv[]) {
 char line[MAXLINE];
 SymTable_T table = SymTable_new();
 struct stats *v;
 while (fgets(line, MAXLINE, stdin)) {
   v = SymTable_get(table, line);
   if (!v) {
        v = makeStats(0);
        SymTable_put(table, line, v);
   }
 SymTable_map(table, maybeprint, NULL);
 return EXIT_SUCCESS;
}
```


Choice 2: ADT owns the key

- Consequence: **`SymTable_put`** must copy its **`key`** argument into a newly malloc'ed string object.



Put Away Your Toys…

- When client is done with a symbol table, it should give the memory back.
- But client can’t call **`free`** directly (as we already demonstrated)
- So there must be an interface function for client to say “I’m done with this”
- It should free the header, list cells, strings
  `SymTable_free(SymTable_T table);`
- Should it free the values?
  - Can’t do it by calling **`free`** directly (as we already demonstrated)
  - Another design choice!

Options to Free Values

- Option 1: Client frees all the values before calling `SymTable_free(table)`
  - Can do this using SymTable_map(table, free_it, NULL);
  - Minor bother: temporarily leaves dangling pointers in the table
  - Minor bother: it’s clumsy
- Option 2: `SymTable_free` calls client function

```
void SymTable_free(SymTable_T table,
   void (*f)(char *key, void *value, void *extra),
   void *extra);
```

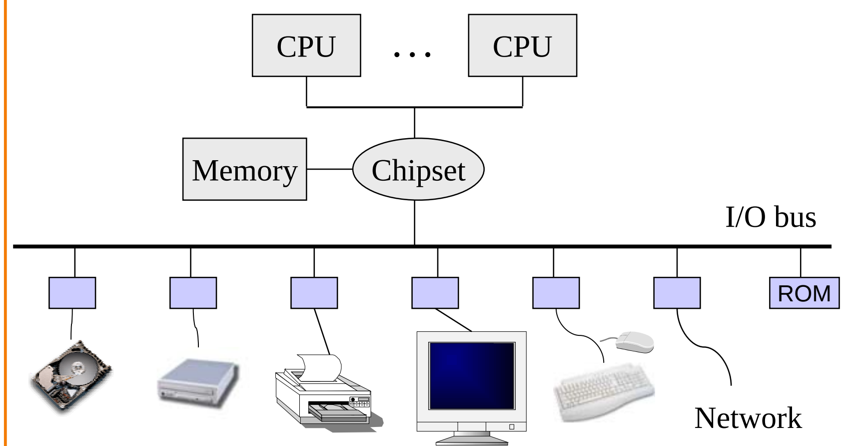
/* Free entire table. During this process, if f is not NULL, apply f to each binding in table. It is a checked runtime error for table to be NULL. */

- We will choose Option 1.

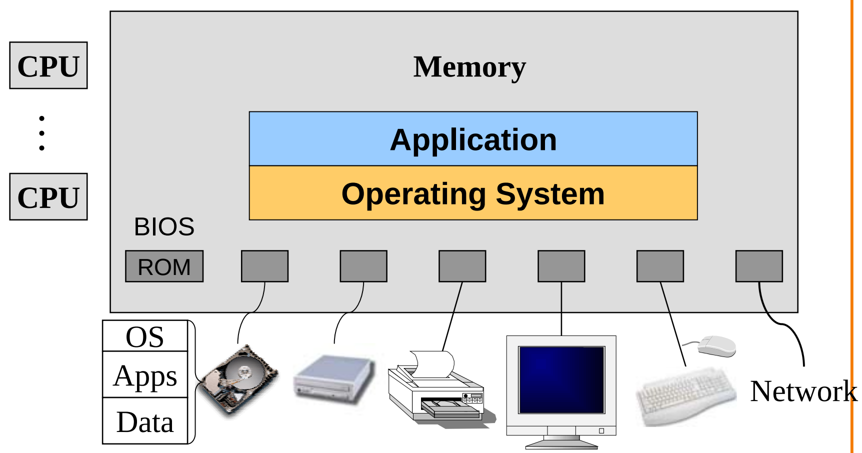
An Overview of Computer Architecture

CS 217

A Typical Computer



A Typical Computer System

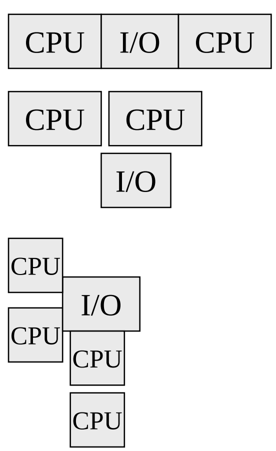


OS Service Examples

- Examples that are not provided at user level
  - System calls: file open, close, read and write
  - Control the CPU so that users won't stuck by running
    - while (1) ;
  - Protection:
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other
- Examples that can be provided at user level
  - Read time of the day
  - Protected user level stuff

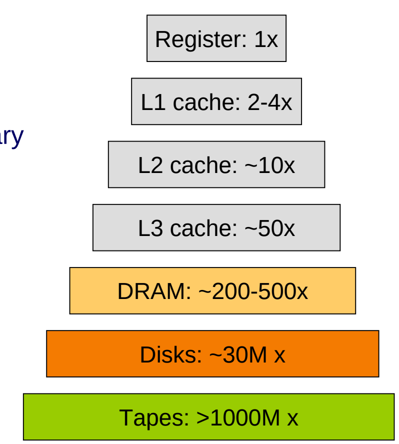
Processor Management

- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free



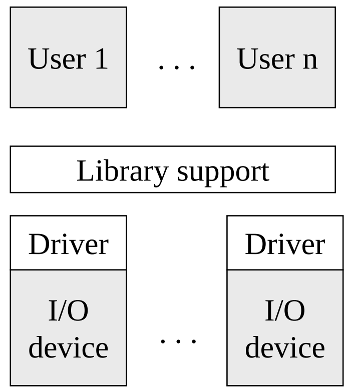
Memory Management

- Goals
  - Support programs to run
  - Allocation and management
  - Transfers from and to secondary storage
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection



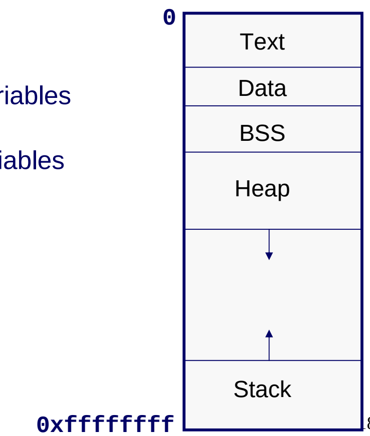
I/O Device Management

- Goals
  - Interactions between devices and applications
  - Ability to plug in new devices
- Issues
  - Efficiency
  - Fairness
  - Protection and sharing

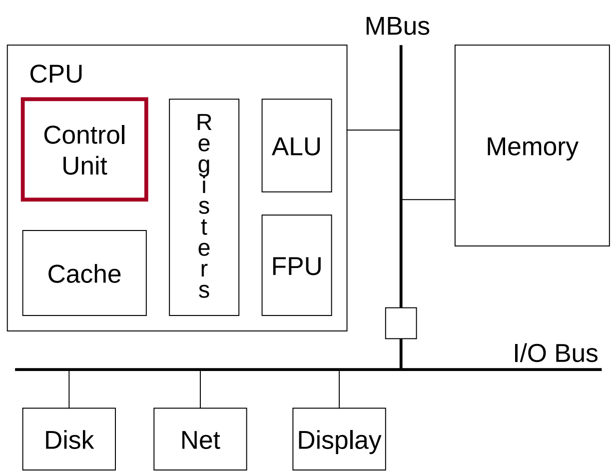


What Is An Application?

- An application has its "own" CPU, memory, and I/O
- "Own" CPU is virtual CPU
- "Own" memory is virtual memory
  - Text = code, constant data
  - Data = initialized global and static variables
  - BSS = (Block Started by Symbol) uninitialized (zero) global & static variables
  - Stack = local variables
  - Heap = dynamic memory
- "Own" I/O devices are virtual
- I/O and CPU may overlap



General Computer Architecture



General Instruction Execution

- CPU's control unit executes a program

  PC ← memory location of first instruction
  while (PC != last_instr_addr)
  execute(MEM[PC]);

- Multiple phases…
  - Fetch: instruction fetch; increment PC
  - Execute: arithmetic instructions, compute branch target address, compute memory addresses
  - Memory access: read/write memory
  - Store: write results to registers

Fetch	**Execute**	**Memory**	**Store**	**Fetch**	**Execute**	**Memory**	**Store**

Concept of Instruction Pipelining

- A simple pipeline

Fetch	Execute	Memory	Store		
	Fetch	Execute	Memory	Store	
		Fetch	Execute	Memory	Store

- What about branch instruction?
- Modern CPUs usually have deep pipelines
  - Pentium II has a 10-stage pipeline
  - Pentium 4 has a 20-stage pipeline
  - They all have sophisticated branch prediction mechanisms

Instructions

- High-level language

```
x = a + b;
```

- Assembly language

```
movl  12(%ebp), %eax
addl  8(%ebp), %eax
```

Symbolic Representation

- Machine code

```
00000011000011000100010l
110010010000100001000101
```

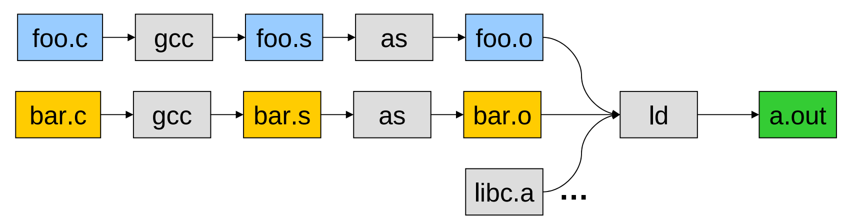
Bit-encoded Representation

Machine Code

- IA32 has variable-sized instructions
- Example:

```
push   %ebp              0x8B
mov    %esp,%ebp         0xE589
```

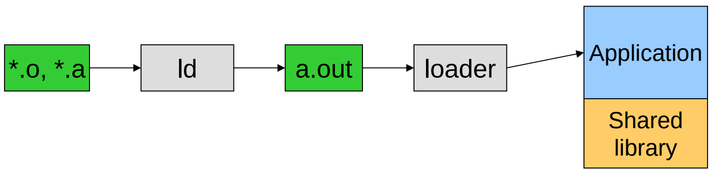

Pipeline of Creating An Executable File



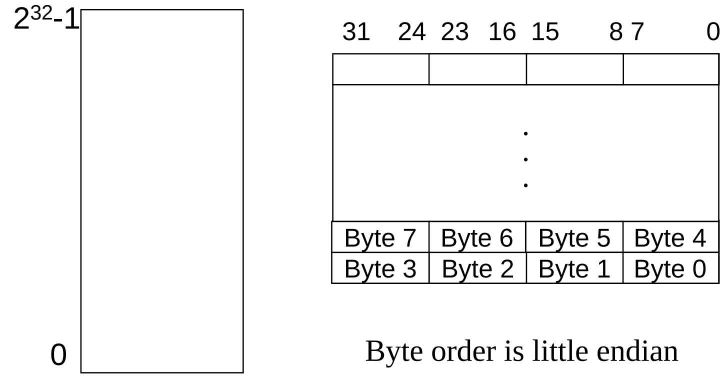
- gcc can compile, assemble, and link together
- Compiler part of gcc compiles a program into assembly
- Assembler compiles assembly code into relocatable object file
- Linker links object files into an executable

Execution (Run An Application)

- On Unix, "loader" does the job
  - Read an executable file
  - Layout the code, data, heap and stack
  - Dynamically link to shared libraries
  - Prepare for the OS kernel to run the application

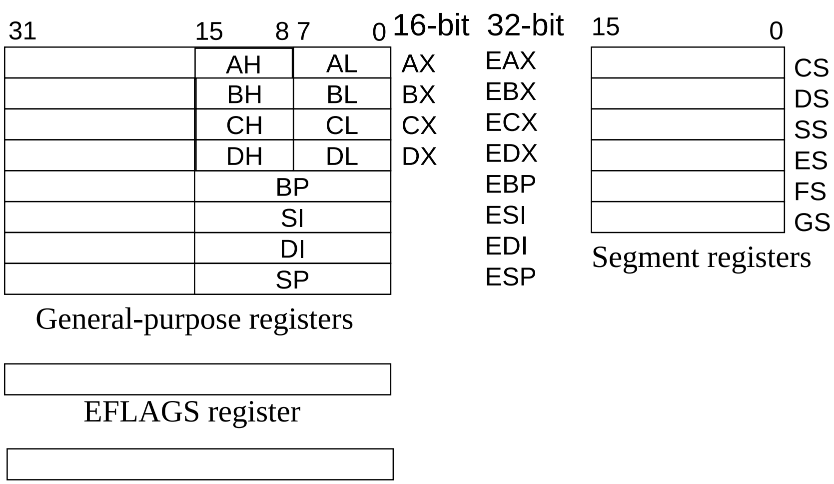


IA32 Memory



Byte order is little endian

IA32 Architecture Registers



EIP (Instruction Pointer register)

Upcoming Lectures ...

- Mode, registers and addressing
- Arithmetic and logic Instructions
- Control transfer instructions
- Assembly directives
- Assembler