

ating system kernel brings a program into memory and starts it running. To start a program, the operating system performs the following steps:

1. Reads the executable file's header to determine the size of the text and data segments.
2. Creates a new address space for the program. This address space is large enough to hold the text and data segments, along with a stack segment (see Section A.5).
3. Copies instructions and data from the executable file into the new address space.
4. Copies arguments passed to the program onto the stack.
5. Initializes the machine registers. In general, most registers are cleared, but the stack pointer must be assigned the address of the first free stack location (see Section A.5).
6. Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's `main` routine. If the `main` routine returns, the start-up routine terminates the program with the `exit` system call.

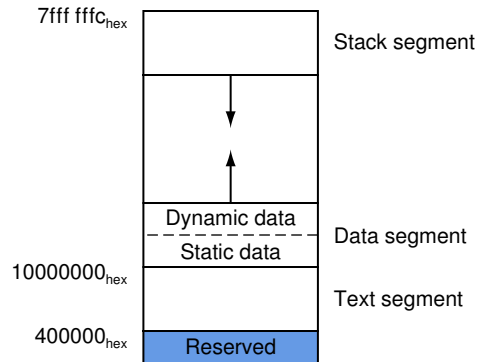
## A.5 Memory Usage

The next few sections elaborate the description of the MIPS architecture presented earlier in the book. Earlier chapters focused primarily on hardware and its relationship with low-level software. These sections focus primarily on how assembly language programmers use MIPS hardware. These sections describe a set of conventions followed on many MIPS systems. For the most part, the hardware does not impose these conventions. Instead, they represent an agreement among programmers to follow the same set of rules so that software written by different people can work together and make effective use of MIPS hardware.

Systems based on MIPS processors typically divide memory into three parts (see Figure A.5.1). The first part, near the bottom of the address space (starting at address `400000hex`), is the *text segment*, which holds the program's instructions.

The second part, above the text segment, is the *data segment*, which is further divided into two parts. **Static data** (starting at address `10000000hex`) contains objects whose size is known to the compiler and whose lifetime—the interval during which a program can access them—is the program's entire execution. For example, in C, global variables are statically allocated since they can be referenced

**static data** The portion of memory that contains data whose size is known to the compiler and whose lifetime is the program's entire execution.



**FIGURE A.5.1** Layout of memory.

Because the data segment begins far above the program at address  $10000000_{\text{hex}}$ , load and store instructions cannot directly reference data objects with their 16-bit offset fields (see Section 2.4 in Chapter 2). For example, to load the word in the data segment at address  $10010020_{\text{hex}}$  into register  $\$v0$  requires two instructions:

```
lui $s0, 0x1001 # 0x1001 means 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

(The *0x* before a number means that it is a hexadecimal value. For example,  $0x8000$  is  $8000_{\text{hex}}$  or  $32,768_{\text{ten}}$ .)

To avoid repeating the *lui* instruction at every load and store, MIPS systems typically dedicate a register ( $\$gp$ ) as a *global pointer* to the static data segment. This register contains address  $10008000_{\text{hex}}$ , so load and store instructions can use their signed 16-bit offset fields to access the first 64 KB of the static data segment. With this global pointer, we can rewrite the example as a single instruction:

```
lw $v0, 0x8020($gp)
```

Of course, a global pointer register makes addressing locations  $10000000_{\text{hex}}$ – $10010000_{\text{hex}}$  faster than other heap locations. The MIPS compiler usually stores *global variables* in this area because these variables have fixed locations and fit better than other global data, such as arrays.

anytime during a program's execution. The linker both assigns static objects to locations in the data segment and resolves references to these objects.

Immediately above static data is *dynamic data*. This data, as its name implies, is allocated by the program as it executes. In C programs, the `malloc` library routine finds and returns a new block of memory. Since a compiler cannot predict how much memory a program will allocate, the operating system expands the dynamic data area to meet demand. As the upward arrow in the figure indicates, `malloc` expands the dynamic area with the `sbrk` system call, which causes the operating system to add more pages to the program's virtual address space (see Section 7.4 in Chapter 7) immediately above the dynamic data segment.

The third part, the program **stack segment**, resides at the top of the virtual address space (starting at address `7fffffffhex`). Like dynamic data, the maximum size of a program's stack is not known in advance. As the program pushes values on the stack, the operating system expands the stack segment down, toward the data segment.

This three-part division of memory is not the only possible one. However, it has two important characteristics: the two dynamically expandable segments are as far apart as possible, and they can grow to use a program's entire address space.

**stack segment** The portion of memory used by a program to hold procedure call frames.

## A.6 Procedure Call Convention

Conventions governing the use of registers are necessary when procedures in a program are compiled separately. To compile a particular procedure, a compiler must know which registers it may use and which registers are reserved for other procedures. Rules for using registers are called **register use** or **procedure call conventions**. As the name implies, these rules are, for the most part, conventions followed by software rather than rules enforced by hardware. However, most compilers and programmers try very hard to follow these conventions because violating them causes insidious bugs.

The calling convention described in this section is the one used by the `gcc` compiler. The native MIPS compiler uses a more complex convention that is slightly faster.

The MIPS CPU contains 32 general-purpose registers that are numbered 0–31. Register `$0` always contains the hardwired value 0.

- Registers `$at` (1), `$k0` (26), and `$k1` (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
- Registers `$a0`–`$a3` (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers `$v0` and `$v1` (2, 3) are used to return values from functions.

**register-use convention** Also called **procedure call convention**. A software protocol governing the use of registers by procedures.