IOS Processes

IOS processes are essentially equivalent to a single thread in other operating systems—IOS processes have one and only one thread each. Each process has its own stack space, its own CPU context, and can control such resources as memory and a console device (more about that later). To minimize overhead, IOS does not employ virtual memory protection between processes. No memory management is performed during context switches. As a result, although each process receives its own memory allocation, other processes can freely access that same memory.

IOS uses a priority run-to-completion model for executing processes. Initially, it might appear that this non-preemptive model is a poor choice for an operating system that must process incoming packets quickly. In some ways, this is an accurate observation; IOS switching needs quickly outgrew the real-time response limitations of its process model, and in Chapter 2, "Packet Switching Architectures," you'll see how this apparent problem was solved. However, this model still holds some advantages that make it a good fit for support processes that remain outside the critical switching path. Some of these advantages are as follows:

- **Low overhead—**

  Cooperative multitasking generally results in fewer context switches between threads, reducing the total CPU overhead contributed by scheduling.

- **Less complexity for the programmer—**

  Because the programmer can control where a process is suspended, it's easy to limit context switches to places where shared data isn't being changed, reducing the possibility for side effects and deadlocks between threads.

Process Life Cycle

Processes can be created and terminated at any time while IOS is operating except during an *interrupt*. A process can be created or terminated by the kernel (during IOS initialization) or by another running process.

> **NOTE**
>
> The term *interrupt* used here refers to a hardware interrupt. When the CPU is interrupted, it temporarily suspends the current thread and begins running an interrupt handler function. New processes cannot be created while the CPU is running the interrupt handler.

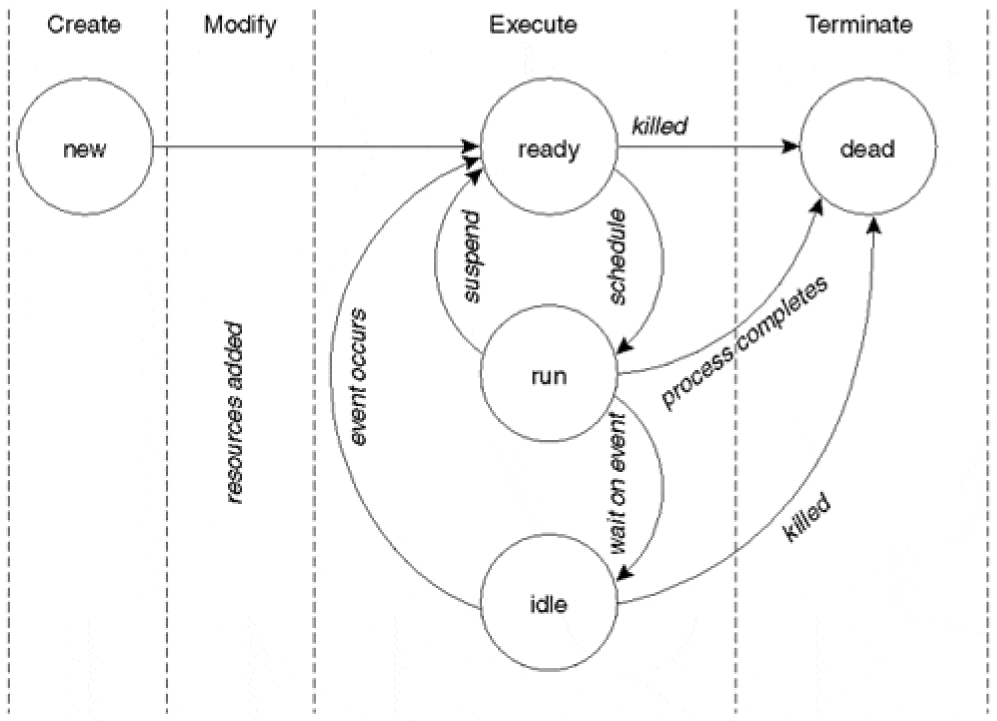
One component in particular is responsible for creating many of the processes in IOS: the *parser*. The parser is a set of functions that interprets IOS configuration and EXEC commands. The parser is invoked by the kernel during IOS initialization and EXEC processes that are providing a command-line interface (CLI) to the console and Telnet sessions.

Any time a command is entered by a user or a configuration line is read from a file, the parser interprets the text and takes immediate action. Some configuration commands result in the setting of a value, such as an IP address, while others turn on complicated functionality, such as routing or event monitoring.

Some commands result in the parser starting a new process. For example, when the configuration command **router eigrp** is entered via the CLI, the parser starts a new process, called *ipigrp* (if the *ipigrp* process hasn't already been started), to begin processing EIGRP IP packets. If the configuration command **no router eigrp** is entered, the parser terminates the ipigrp process and effectively disables any EIGRP IP routing functionality.

IOS processes actually go through several stages during their existence. Figure 1-4 shows these stages and their corresponding states.

Figure 1-4. Process Life Cycle



Creation Stage

When a new process is created, it receives its own stack area and enters the *new* state. The process can then move to the modification stage. If no modification is necessary, the process moves to the execution stage.

Modification Stage

Unlike most operating systems, IOS doesn't automatically pass startup parameters or assign a console to a new process when it is created, because it's assumed most processes don't need these resources. If a process does need either of these resources, the thread that created it can *modify* it to add them.

Execution Stage

After a new process is successfully created and modified, it transitions to the *ready* state and enters the execution stage. During this stage, a process can gain access to the CPU and run.

During the execution stage, a process can be in one of three states: ready, run, or idle. A process in the *ready* state is waiting its turn to access the CPU and to begin executing instructions. A process in the *run* state is in control of the CPU and is actively executing instructions. An *idle* process is asleep, waiting on external events to occur before it can be eligible to run.

A process transitions from the ready state to the run state when it's scheduled to run. With non-preemptive multitasking, a scheduled process continues to run on the CPU until it either suspends or terminates. A process can suspend in one of two ways. It can explicitly suspend itself by telling the kernel it wants to relinquish the CPU, transition to the ready state, and wait its next turn to run. A process can also suspend by waiting for an external event to occur. When a process begins waiting on an event, the kernel implicitly suspends it by transitioning it to the idle state, where it remains until the event occurs. After the event occurs, the kernel transitions the process back to the ready state to await its next turn to run.

Termination Stage

The final stage in the process life cycle is the termination stage. A process enters the termination stage when it completes its function and shuts down (called *self termination*) or when another process kills it. When a process is killed or self terminates, the process transitions to the *dead* state. A terminated process remains in the dead state, inactive, until the kernel reclaims all of its resources. The kernel might also record statistics about the process' stack when it terminates. After its resources are reclaimed, the terminated process transitions out of the dead state and is totally removed from the system.

IOS Process Priorities

IOS employs a priority scheme to schedule processes on the CPU. At creation time, every process is assigned one of four priorities based on the process' purpose. The priorities are static; that is, they're assigned when a process is created and never changed. The IOS process priorities are:

- **Critical—**

  Reserved for essential system processes that resolve resource allocation problems.

- **High—**

  Assigned to processes that provide a quick response time, such as a process that receives packets directly from a network interface.

- **Medium—**

  The default priority used by most IOS processes.

- **Low—**

  Assigned to processes providing periodic background tasks, such as logging messages.

Process priorities provide a mechanism to give some processes preferential access to the CPU based on their relative importance to the entire system. Remember though, IOS doesn't employ preemption, so a higher priority process can't interrupt a lower priority process. Instead, having a higher priority in IOS gives a process more *opportunities* to access the CPU, as you'll see later when you investigate the operation of the kernel's scheduler.

Process Examples

You can use the **show process** command to see a list of all the processes in a system along with some run-time data about each one, as demonstrated in Example 1-4.

Example 1-4. *show process* Command Output

```
router#show process CPU utilization for five seconds: 0%/0%; one minute: 0%; five minutes: 0% PID QTy
PC Runtime (ms) Invoked uSecs Stacks TTY Process 1 M* 0 400 55 727210020/12000 0 Exec 2 Lst
6024E528 201172 33945 5926 5752/6000 0 Check heaps 3 Cwe 602355E0 0 1 0 5672/6000 0 Pool
Manager 4 Mst 6027E128 0 2 0 5632/6000 0 Timers 5 Mwe 602F3E60 0 1 0 5656/6000 0 OIR Handler 6
Msi 602FA560 290744 1013776 286 5628/6000 0 EnvMon 7 Lwe 60302944 92 17588 5 5140/6000 0 ARP
Input 8 Mwe 6031C188 0 1 0 5680/6000 0 RARP Input 9 Mwe 60308FEC 15200 112763 13410748/12000 0
IP Input 10 Mwe 6033ADC4 420 202811 2 5384/6000 0 TCP Timer 11 Lwe 6033D1E0 0 1 011644/12000 0
TCP Protocols 12 Mwe 60389D6C 10204 135198 75 5392/6000 0 CDP Protocol 13 Mwe 6035BF28 66836
1030665 6411200/12000 0 IP Background 14 Lsi 60373950 0 16902 0 5748/6000 0 IP Cache Ager 15 Cwe
6023DD60 0 1 0 5692/6000 0 Critical Bkgnd 16 Mwe 6023DB80 0 13 0 4656/6000 0 Net Background 17
Lwe 6027456C 0 11 011512/12000 0 Logger 18 Msp 6026B0CC 100 1013812 0 5488/6000 0 TTY
Background 19 Msp 6023D8F8 8 1013813 0 5768/6000 0 Per-Second Jobs 20 Msp 6023D854 405700
1013812 400 4680/6000 0 Net Periodic 21 Hwe 6023D9BC 5016 101411 49 5672/6000 0 Net Input 22 Msp
6023D934 135232 16902 8000 5640/6000 0 Per-minute Jobs
```

The following list describes each of the **show process** command output fields found in Example 1-4.

- **PID—**

  Process identifier. Each process has a unique process identifier number to distinguish it from other processes.

- **Qty—**

  Process priority and process state. The first character represents the process' priority as follows:

  - **K—**

    No priority, process has been killed.

  - **D—**

    No priority, process has crashed.

  - **X—**

    No priority, process is corrupted.

  - **C—**

    Critical priority.

  - **H—**

    High priority.

  - **M—**

    Medium priority.

  - **L—**

    Low priority.

    The remaining two characters in this field represent the current state of the process as follows:

  - ***—**

    Process is currently running on the CPU.

  - **E—**

    Process is waiting for an event (event dismiss).

  - **S—**

    Process is suspended.

  - **rd—**

    Process is ready to run.

- **we—**

  Process is idle, waiting on an event.

- **sa—**

  Process is idle, waiting until a specific absolute time occurs.

- **si—**

  Process is idle, waiting for a specific time interval to elapse.

- **sp—**

  Process is idle, waiting for a specific time interval to elapse (periodic).

- **st—**

  Process is idle, waiting for a timer to expire.

- **hg—**

  Process is hung.

- **xx—**

  Process is dead.

- **PC—**

  Contents of the CPU program counter register when the process last relinquished the CPU. This field is a memory address that indicates where the process begins executing the next time it gets the CPU. A value of zero means the process is currently running.

- **Runtime—**

  Cumulative amount of time (in milliseconds) the process has used the CPU.

- **Invoked—**

  Total number of times the process has run on the CPU since it was created.

- **uSecs—**

  Average amount of CPU time (in microseconds) used each time the process is invoked.

- **Stacks—**

  Stack space usage statistic. The number on the right of the slash (/) shows the total size of the stack space. The number on the left indicates the low water mark for the amount of free stack space available.

- **TTY—**

  Console device associated with this process. Zero indicates the process does not own a console or communicates with the main system console.

- **Process—**

  Name of the process. Process names need not be unique (multiple copies of a process can be active simultaneously). However, process ids are always unique.

If you issue the **show process** command on several different IOS systems, you'll notice some processes appear on every one. Most of these are processes that perform housekeeping or provide services to other processes. Table 1-2 describes the most common of these processes and the tasks they perform.

Table 1-2. Common System Processes and Their Functions

System Process	Function
EXEC	Command-line interface (CLI) for the console and directly connected asynchronous TTY lines. The EXEC process accepts user input and provides an interface to the parser.
Pool manager	Manages buffer pools (more on this in the "Packet Buffer Management" section in this chapter).
Check heaps	Periodically validates the integrity of the runtime IOS code and the structure of the memory heap.
Per-minute jobs	Generic system process that runs every 60 seconds performing background maintenance, such as checking the integrity of process stacks.
Per-second jobs	Generic system process that performs tasks that need to be repeated every second.
Critical background	Critical priority process that performs essential system services, such as allocating additional IOS queue elements when they run out.
Net background	Sends interface keepalive packets, unthrottles interfaces, and processes interface state changes.
Logger	Looks for messages (debug, error, and informational) queued via the kernel by other processes and outputs them to the console and, optionally, to a remote syslog server.
TTY background	Monitors directly connected asynchronous TTY lines for activity and starts "EXEC" processes for them when they go active.

All the processes in Table 1-2, except EXEC, are created by the kernel during system initialization and normally persist until IOS is shut down.

< BACK   Make Note | Bookmark   CONTINUE >

Index terms contained in this section