



## Process Management 101

*Jim DeRoest has been involved (for better or worse) with IBM UNIX offerings from the IX/370 days, through PC/IX, AIX RT, AIX PS/2, AIX/370, PAIX, AIX/ESA and AIX V3. He is employed as an assistant director supporting academic and research computing at the University of Washington, and is the author of AIX for RS/6000—System and Administration Guide (McGraw-Hill). He plays a mean set of drums for the country gospel band Return. Email: deroest@cac.washington.edu.*

**T**his month, I thought I would step back from technology's bleeding edge and talk about some of the basics of the AIX operating system—specifically, AIX process architecture and management. My daughter, just beginning her foray into the world of computer science, reminds me that there are always new programmers and administrators out there looking for a hand up from us old-timers. Even those of us who have waged and won OS battles with one flavor of UNIX can always use a hint or two when it comes to another. With this in mind, what follows is a top-level description of AIX process architecture. You won't find a detailed description of threads, mutexes or multiprocessor scheduling; I'll tackle those topics another time. What you will find is enough information to understand and manage the process space from the command line. Basically, it's meat-and-potatoes AIX process administration.

### Process Basics

A process comprises an executing program and its associated resources. Processes are created in the system address space by invoking the `fork()` and `exec()` system calls. A parent process creates a new child

process by invoking `fork()`. The kernel reserves a vacant Process Identifier (PID) for the child and copies the attribute data associated with the parent into the child's process structures. The child is essentially a clone of the parent until either the child, the parent or a privileged authority modifies the child's attributes via a system call. The most common method of altering a child process is by invoking a new program image using the `exec()` call.

The PID is a pointer into the kernel process table. Process table entries point to per-process kernel data structures that represent process attributes and resources. These structures are represented in the `/usr/include/sys/proc.h` header file. Some process attributes are listed in Table 1.

Active process attributes can be interrogated from the command line using the `ps` command or by using SMIT (System Management Interface Tool). AIX supports two flavors of `ps`, `SYSV` and `BSD`. `SYSV` is used when the command-line arguments are preceded by a hyphen; otherwise, `BSD` is used.

`ps -elk`      `SYSV` process display format

`ps auxw`      `BSD` process display format

When displaying the active process table, you may notice a set of processes listed as `kproc` under the command column. These are a special set of kernel processes that collect accounting data for system overhead. One `kproc` process in particular usually displays a very high CPU utilization. No need to panic; this process collects system wait and idle time and represents it in the `ps` CPU fields (see Figure 1).

Along with the PID, each process records its Process Parent Identifier (PPID) and its group affiliation as the Process Group Identifier (PGID). Process groups are collections of one or more processes. The group leader has a PGID equal to its PID. Each group member has a PGID that matches the leader. Unless reset by a `setpgrp()` call, a process inherits the PGID of its parent.

Process groups provide a mechanism for signaling all processes within the group using the PGID. This eliminates the need to know each member's PID. The PID, PPID and PGID are the primary handles used by systems administrators to control process behavior. There's a nice public domain tool called `pstree` that will graphically map process relationships on the screen. It's available from `aixpdslib.seas.ucla.edu` via WWW or FTP.

Other basic handles that may be used to control process behavior include the owning user identifier (UID), group identifier (GID) and controlling terminal (TTY). UID and GID associations are mapped to accounts and groups listed in the `/etc/passwd` and `/etc/group` files. The process UID or GID represents two mappings designated as "real" and "effective": The real UID and GID identify the process owner; the effective UID and GID identify the privileges available to the process. The real and effective mappings don't have to be the same. The TTY identifies the default

## Table 1. Process Attributes

- Process Identifier
- Process Group Identifier
- Process Parent Identifier
- Process Owner
- Real/Effective User and Group Identifiers
- Priority
- Controlling Terminal
- Address Space
- Size in Pages
- Paging Statistics
- Resource Utilization
- Process State

device for standard input, output and error channels, and for sending signals.

In AIX Version 4, the process structure is further broken down into a set of execution structures called "threads." Threads provide the means for overlapping, multiplexing and parallelizing operations within a process. Threads are peer entities within the process and share global resources like the process address space. As such, they can usually be controlled via signals at the process level. This is not always true, however, because threads may be scheduled and executed independently on different processors in a multiprocessor environment.

I'm not going to say more about threads in this column. My intent here is to offer a description of basic AIX process architecture. Nevertheless, it is important to remember that AIX processes comprise thread structures. A full discussion of process threads, locks and signals must wait for another time.

## Figure 1. Process ps Output

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
303	A	0	0	0	80	16	--	808	4		-	0:00	swapper
200003	A	0	1	0	0	60	20	505	212		-	0:01	init
303	A	0	516	0	120	127	--	909	0		-	7:30	kproc
303	A	0	774	0	0	36	--	606	8		-	0:00	kproc
303	A	0	1032	0	0	37	--	c0c	40	*	-	0:00	kproc
40201	A	0	1874	0	0	60	20	1f5f	8		-	0:00	kproc
240001	A	0	2114	1	0	60	20	1736	64	7737158	-	0:00	syncd
40201	A	0	2368	1	0	60	20	13d3	8	19dbaa4	-	0:00	kproc
240001	A	0	2766	3588	0	60	20	1998	652		-	0:00	dtsession
40001	A	0	3052	1	0	60	20	f4e	296		-	0:00	dtlogin
240001	A	0	3270	11444	0	60	20	14b6	144		-	0:00	pppauthd
40201	A	0	3362	1	0	60	20	13b3	12	1959b60	-	0:00	kproc
40001	A	0	3588	3052	0	60	20	564	628		-	0:00	dtlogin
40201	A	0	3968	1	3	61	20	1270	8	7737898	-	0:02	kproc
240001	A	0	4594	3052	3	61	20	1b5a	3340	54a9e94	-	0:14	X
240001	A	0	4658	1	0	60	20	d2f	152	74adc7c	-	0:00	cron
40401	A	0	4940	1	0	60	20	fce	308	95318	-	0:00	errdemon
240001	A	0	5266	1	0	60	20	15d4	232		-	0:00	srcmstr
240001	A	0	5546	5266	0	60	20	12f3	124		-	0:00	syslogd
40001	A	0	5758	1	0	60	20	567	1600	*	-	0:00	pmd
240001	A	0	6068	5266	0	60	20	200	584		-	0:00	sendmail
240001	A	0	6334	5266	0	60	20	705	232		-	0:00	portmap

## Process Priorities

AIX uses a priority-based set of run queues to allocate CPU resources among the set of active processes as shown in Figure 1. Priority values range from 0 to 127, each of which is represented by a run queue. Low numbered queues are scheduled more often than high numbered queues. Processes within a run-queue level are scheduled in a round-robin fashion. Each process' queue priority is calculated from the sum of its short-term CPU usage, its nice value and the minimum user process level. The priority value increases for processes that execute frequently and decreases for those that are waiting for execution. Processes with a priority value exceeding 120 will execute only when no other process in the system requires CPU resources. Process short-term CPU usage, priority and nice value are displayed in the PRI, C and NI fields using the `SYSV ps -l` option.

The nice value is an integer that represents coarse priorities between processes. AIX supports both the BSD nice value range of -20 to 20 and the SYSV range of 0 to 39. The larger the number, the lower the scheduling priority. The two value ranges are mapped such that BSD -20 corresponds to SYSV 0 for highest priority, and BSD 20 to SYSV 39 for lowest priority.

New processes inherit the nice value of their parents. The nice value may be altered dynamically during the process lifetime. The owning UID for a process can lower a process' nice value. Only the superuser can improve nice priority. The nice value can be set from the command line using the `nice` command (think of the `nice` and `renice` commands as throttles for controlling process CPU utilization):

```
nice -n <value> <command>
```

Process owners and the superuser can modify existing process nice values by using the `renice` command:

```
renice <value> -p <PID>
```

Before you lower the hammer, however, be aware that the BSD %CPU field represents the percentage of CPU resources that a process has used in its lifetime. You may see short-lived processes shoot up to very high %CPU numbers. A better gauge for identifying CPU crunchers or runaway processes is the TIME column.

The scheduler parcels out CPU time slices at a frequency that makes it appear as if all processes are executing at the same time. In fact, they are being scheduled one at a time, except in the case of multiprocessor systems. When a process isn't executing on the CPU, it may be waiting on a resource or lock, sleeping on an event, suspended or moving through some dispatch or scheduler state. The process state is maintained as part of the `proc` structure information. The process state is displayed by `ps` in the STAT column when either the BSD `l` or SYSV `-l` flag is used. For processes that are flagged `w` (for waiting), the WCHAN column identifies the address of the event being waited on. Refer to Table 2 for a list of process state tags displayed by the `ps` command.

**Table 2. Process State Tags**

A	Active
O	Nonexistent
S	Sleeping
W	Waiting
R	Running
I	Intermediate
Z	Canceled
T	Stopped
K	Available kernel process
X	Growing

## Controlling Processes

We've already talked about lowering a process' priority using the `nice` and `renice` commands. What do you do when process management requires a heavier hand? You use `kill`! The `kill` command sounds much more ominous than it is. What `kill` does is send a specified signal to a process. The signal does not necessarily cause process termination. Note that `kill` is a built-in command for some shells, for example, `csh`. Be aware that the behavior of the shell version of `kill` and `/usr/bin/kill` may be quite different.

```
kill [-Signal] [PID PID PID ...]
```

If you want to send a signal to all your processes except the sending process, use the `killall` command:

```
killall [-signal]
```

To display the set of supported signals, use the `-l` argument to `kill`:

```
kill -l
```

AIX signals are based on the SYSV implementation. However, some BSD signals are mapped to their SYSV counterparts and BSD signal system calls are available. When writing or porting programs that use BSD signals and calls, note that signals are not automatically reset after being caught. They must be specifically reset to the required behavior in the signal handler routine.

## Rules of Thumb

It seems to be a common practice to use the `KILL (9)` signal to terminate a process. I recommend that you do this only as a last resort after first trying `HUP (1)` and `ABRT (6)`. The latter two signals allow a process to terminate gracefully. In the case of `ABRT`, a core file is produced that may be used for debugging. The `KILL` signal basically attempts to yank the process out of the process table without permitting any cleanup activity.

```
kill -1 <PID> First try HUP.
```

```
kill -6 <PID> Then try ABRT.
```

```
kill -9 <PID> Use KILL if all else fails.
```

Occasionally, a user may try out some ingenious bit of C code that contains a statement along the lines of:

```
while(1) fork();
```

I'm not insinuating that this is done on purpose, but it can be a pain in the neck to stop. New processes are created as fast as you can kill them. One little trick you can try is to kill them by PGID. Use the formatted output, `-F`, option with `SYSV ps` to display the PGID. Then, send a signal to the negative PGID:

```
ps -el -F pgid,runame=<procname>
```

```
kill -6 -<pgid>
```

## Scheduling Processes

One way of controlling overall consumption of resources is to schedule execution off shift. The UNIX `cron` utility provides a basic means for scheduling jobs to be run at a particular time of the day or on a periodic basis. `cron` can be used to take care of regular system housecleaning tasks such as synchronizing disk writes, cleaning out `/tmp` and running accounting programs. Such periodic tasks may be tailored using `crontabs`. A `crontab` is a list of commands and scripts with designated run times that will be invoked by `cron` under the effective UID of the owner. `cron` reports any errors or output information to the owning user after the commands are executed. `cron` logs errors to a log file, `/var/adm/cron/log`, and if AIX auditing is enabled, produces audit records.

To create a `crontab`, use your favorite editor and create a table with the following fields:

```
minutes  hours  day  month  weekday  command
```

Each of the time-associated fields may be represented as a comma-separated list. An asterisk (\*) may be used to represent all possible times. For example, if I wanted to display uptime statistics every 30 minutes on the system console, I would add the following line to my `crontab` file:

```
0,30 * * * * /bin/uptime > /dev/console
```

Once you have your `crontab` file tailored to your liking, hand it off to `cron` by invoking the `crontab` command:

```
crontab <YourCrontabFile>
```

The systems administrator can enforce access controls on who may use `cron` services by listing usernames, one per line, in the `/usr/adm/cron/{cron.allow,cron.deny}` files. `cron` checks these files' authorization before invoking a user's `crontab` file. The default is to allow access to all users.

Suppose you want to run a job off hours but don't want to create a `crontab` entry for it. It may be a one-time-only run. You can do this using the `at` and `batch` commands. Note that `batch` is just a script that invokes `at`. Execute `at` jobs by speci-

fying the time and the input stream of commands on the command line. The job stream is copied to the `/var/spool/cron/atjobs` directory, `cron` then executes the job stream at the specified time. As with `cron`, authorization is controlled by listing usernames in the `/usr/adm/cron/{at.allow,at.deny}` files. The default is to allow access to all users.

```
at <time> input <Ctrl-D>      Start a job at time.
```

```
at -r jobnumber              Remove a job.
```

```
atq <username>              List scheduled jobs.
```

## More Information

I hope I've presented enough background information to assist you in managing processes in AIX. If you're interested in more detail, then I would recommend you take a look at a text dedicated to AIX systems administration. It just so happens I have a new book on AIX Version 4 systems administration, appropriately titled *AIX Version 4: System and Administration Guide*, published by McGraw-Hill, ISBN 0-07-036688-8. Pick one up for all your friends and relatives. I'm shameless. There are a number of other good AIX texts out there. You'll find a list of AIX references in the AIX FAQ distributed in the `comp.unix.aix` newsgroup. Use your favorite Web search tool to track down a copy on the Web. ✍