

4 Controllo dei processi



In Linux un processo è l'astrazione usata per rappresentare un programma in esecuzione. È l'oggetto attraverso il quale è possibile gestire e monitorare l'uso della memoria, del tempo di elaborazione del processore e delle risorse di I/O da parte di un programma.

Nella filosofia di Linux e UNIX il lavoro deve essere svolto per quanto possibile nel contesto dei processi, e non gestito in modo particolare dal kernel. Processi di sistema e processi utente seguono le stesse regole, perciò è possibile utilizzare un unico insieme di strumenti per controllarli entrambi.

4.1 COMPONENTI DI UN PROCESSO

Un processo è costituito da uno spazio di indirizzamento e da un insieme di strutture dati nel kernel. Lo spazio di indirizzamento è costituito da un insieme di pagine di memoria che il kernel riserva all'uso del processo; contiene il codice e le librerie che il processo esegue, le variabili, gli stack e ulteriori informazioni necessarie al kernel durante l'esecuzione del processo in questione. Poiché Linux è un sistema a memoria virtuale, non vi è correlazione tra la posizione di una pagina in uno spazio di indirizzamento e la posizione all'interno della memoria fisica o dell'area di swap della macchina.

1. Le pagine sono le unità in cui viene gestita la memoria; hanno solitamente dimensione di 4K sui PC.

Le strutture dati interne del kernel memorizzano varie informazioni su ciascun processo; di seguito sono elencate alcune tra le più importanti:

- mappa dello spazio di indirizzamento del processo;
- stato corrente del processo (sleeping, stopped, runnable e così via);
- priorità di esecuzione del processo;
- informazioni sulle risorse usate dal processo
- informazioni su file e porte di rete aperti dal processo;
- maschera dei segnali del processo (un elenco dei segnali bloccati);
- proprietario del processo.

Alcuni di questi attributi possono essere condivisi tra più processi per creare un “gruppo di thread” che corrisponde in Linux a un processo multithreaded di UNIX. Benché possano condividere uno spazio di indirizzamento, i membri di un gruppo di thread hanno le proprie priorità e i propri stati di esecuzione. Nella pratica sono pochi i processi significativi per gli amministratori di sistema che usano più thread di esecuzione, e anche quelli che lo fanno (come ad esempio **named** di BIND 9) generalmente non richiedono attenzione da parte degli amministratori a questo livello di granularità.

Molti dei parametri associati a un processo influiscono direttamente sulla sua esecuzione: la quantità di tempo del processore ottenuta, i file a cui può accedere e così via. Nei paragrafi seguenti vengono discussi il significato e l'importanza dei parametri più interessanti per un amministratore di sistema. Questi attributi sono comuni a tutte le versioni di UNIX e Linux.

PID: numero identificativo del processo

Il kernel assegna un numero identificativo univoco a ciascun processo. La maggior parte dei comandi e delle chiamate di sistema che manipolano processi richiede di specificare un PID per identificare il target dell'operazione. I PID sono assegnati ai processi nel momento in cui questi sono creati.

PPID: PID del padre

Linux non fornisce una chiamata di sistema per creare un nuovo processo che esegua un particolare programma. Un processo esistente deve clonare se stesso per creare un nuovo processo; il clone può quindi sostituire il programma che esegue con un altro.

Quando un processo viene clonato, quello originale è indicato come padre, mentre la copia è indicata come figlio. L'attributo PPID di un processo è il PID del padre da cui è stato clonato.²

Il PID del padre rappresenta un'informazione utile quando si ha a che fare con un processo non riconosciuto (e che potrebbe causare problemi). Risalendo all'origine del processo (ad esempio la shell o un altro programma), si può capire meglio il suo scopo e la sua importanza.

2. Almeno inizialmente. Se il padre originale muore, **init** (processo 1) diventa il nuovo padre. Vedere il Paragrafo 4.2.

UID ed EUID: ID utente reale ed effettivo

L'UID di un processo è il numero identificativo utente della persona che lo ha creato, o per la precisione è una copia del valore UID del processo padre. Solitamente, soltanto il creatore (il "proprietario") e il superuser hanno il permesso di manipolare un processo.

L'EUID è l'ID utente "effettivo", un UID in più usato per determinare a quali risorse e file può accedere un processo in un momento dato. Per la maggior parte dei processi UID ed EUID coincidono, con l'eccezione dei programmi setuid.

La disponibilità di UID ed EUID separati è utile per mantenere una distinzione tra identità e permessi, e perché non sempre un programma setuid deve operare con permessi ampliati: l'EUID può essere impostato e reimpostato per abilitare o rimuovere i permessi aggiuntivi che assegna.

Linux tiene traccia anche di un "saved UID", o "UID salvato", una copia dell'EUID del processo nel momento in cui quest'ultimo inizia l'esecuzione. A meno che il processo non provveda appositamente a rimuoverlo, questo UID salvato rimane disponibile per l'uso come l'UID reale o effettivo. Un programma setuid ben scritto può quindi rinunciare ai propri privilegi speciali per la maggior parte dell'esecuzione, utilizzandoli soltanto nei momenti specifici in cui gli servono.

Linux definisce anche un parametro di processo FSUID non standard che controlla la determinazione dei permessi sul filesystem. Tale parametro non è utilizzato spesso al di fuori del kernel.

Questo sistema a più tipi di UID può avere sottili implicazioni. Per chi è interessato all'argomento, un'ottima risorsa è offerta dal libro online gratuito di David A. Wheeler, *Secure Programming for Linux and Unix HOWTO* (in inglese), disponibile presso il sito www.dwheeler.com.

GID ed EGID: ID di gruppo reale ed effettivo

Il GID è il numero che identifica il gruppo di un processo. L'EGID è correlato al GID esattamente come l'EUID è correlato all'UID, nel senso che può essere "aggiornato" dall'esecuzione di un programma setgid. Il kernel mantiene anche un "saved GID", o "GID salvato", analogo all'UID salvato.

L'attributo GID di un processo può essere considerato sostanzialmente figurativo. Per quanto riguarda la determinazione del tipo di accesso, un processo può essere membro di più gruppi contemporaneamente. L'elenco completo dei gruppi è memorizzato separatamente dai valori GID ed EGID. Normalmente, quando si determinano i permessi di accesso si tiene conto dell'EGID e dell'elenco di gruppi supplementare, ma non del GID.

L'unica occasione in cui il GID risulta utile è quando un processo crea nuovi file: a seconda dei permessi impostati sul filesystem, i nuovi file potrebbero adottare il GID del processo che li crea. Vedere il Paragrafo 5.5 per ulteriori informazioni.

Nice

La priorità di schedulazione di un processo determina quanto tempo di CPU è assegnato allo stesso. Il kernel utilizza un algoritmo dinamico per calcolare le priorità, tenendo conto della quantità di tempo di CPU che un processo ha consumato recentemente e della lunghezza del tempo di attesa prima dell'esecuzione. Inoltre il kernel considera un valore impostato a livello amministrativo chiamato "nice" o "niceness" perché indica la "cortesia" (dal termine inglese *nice*) della pianificazione rispetto agli altri utenti del sistema. Questo argomento è trattato in dettaglio nel Paragrafo 4.6.

Nel tentativo di fornire un miglior supporto per applicazioni a bassa latenza, Linux ha aggiunto le "classi di schedulazione" al modello di schedulazione tradizionale di UNIX. Attualmente esistono tre classi di schedulazione e ogni processo è assegnato a una di esse. Sfortunatamente le classi real-time non sono usate spesso, né sono ben supportate dalla riga di comando. I processi di sistema utilizzano tutti lo scheduler tradizionale basato su nice. In questo libro viene esaminato soltanto lo scheduler standard, si rimanda al sito www.realtimelinuxfoundation.org per ulteriori informazioni sugli aspetti legati alla schedulazione real-time.

Terminale di controllo

La maggior parte dei processi "non demoni" ha un terminale di controllo associato, che determina i collegamenti di default per i canali standard input, standard output e standard error. Quando si avvia un comando dalla shell, il terminale in uso normalmente diventa il terminale di controllo del processo. Il concetto di terminale di controllo influenza anche sulla distribuzione dei segnali, discussi nel paragrafo seguente.

4.2 IL CICLO DI VITA DI UN PROCESSO

Per creare un nuovo processo, tipicamente un processo copia se stesso con la chiamata di sistema **fork**. Tale chiamata crea una copia del processo originale praticamente identica al padre, ma con un PID distinto e informazioni di account proprie.

fork ha la peculiarità di restituire due valori: al figlio restituisce zero, mentre il padre riceve il PID del figlio appena creato. Poiché i due processi sono per il resto identici, entrambi devono esaminare il valore restituito dalla chiamata per determinare il proprio ruolo.

Dopo la **fork**, il processo figlio spesso usa una chiamata di sistema della famiglia **exec** per iniziare l'esecuzione di un nuovo programma.³ Queste chiamate cambiano il testo del programma che il processo esegue e riportano i segmenti di dati e stack a uno stato iniziale predefinito. Le varie forme delle chiamate **exec** differiscono soltanto per il modo in cui sono specificati gli argomenti della riga di comando e le variabili di ambiente da fornire al nuovo programma.

Linux definisce un'alternativa a **fork** denominata **clone**. Questa chiamata crea insiemi di processi che condividono memoria, spazi di I/O o entrambi. Si tratta di una funzionalità analoga al multithreading disponibile nella maggior parte delle versioni di UNIX, ma ogni thread di esecuzione è rappresentato come un processo completo, invece di un oggetto "thread" specializzato.

All'avvio del sistema, il kernel crea e installa in modo autonomo diversi processi. Il più importante di questi è **init**, che è sempre il processo numero 1 ed è responsabile dell'esecuzione degli script di avvio del sistema; da esso discendono tutti i processi tranne quelli creati direttamente dal kernel.

init ha anche un altro ruolo importante nella gestione dei processi. Quando un processo termina, richiama una routine denominata **_exit** per notificare al kernel un codice di uscita (un intero) che indica il motivo della terminazione. Per convenzione, 0 indica una terminazione normale o "con successo".

Prima che un processo possa scomparire del tutto, Linux richiede che la sua terminazione sia attestata dal suo genitore, mediante una chiamata a **wait**. Il genitore riceve una copia del codice di uscita del figlio (o un'indicazione del motivo per cui il figlio è stato terminato, se la terminazione non è avvenuta per volontà del figlio stesso) e può anche ottenere un riepilogo delle risorse usate da esso.

Questo schema funziona bene se i padri sopravvivono ai loro figli e provvedono sempre alla chiamata di **wait** per consentire l'eliminazione dei processi terminati. Tuttavia, se il padre termina per primo, il kernel si rende conto che non vi potranno essere chiamate **wait** e provvede a rendere il processo orfano figlio di **init**. Quest'ultimo accetta i processi orfani ed esegue la chiamata **wait** necessaria per l'eliminazione dei processi alla loro terminazione.

4.3

SEGNALI

I segnali sono richieste di interruzione a livello di processo. Ne esistono circa trenta tipi diversi, usati in vari modi:

- possono essere inviati da un processo a un altro come mezzi di comunicazione;
- possono essere inviati dal driver di terminale per terminare, interrompere o sospendere processi quando si premono tasti speciali come <Control-C> e <Control+Z>;⁴
- possono essere inviati dall'amministratore (con **kill**) per ottenere vari risultati;
- possono essere inviati dal kernel quando un processo commette un'infrazione, come una divisione per zero;
- possono essere inviati dal kernel per notificare al processo una condizione "interessante", come la terminazione di un processo figlio o la disponibilità di dati su un canale di I/O.

4. Le funzioni di <Control-Z> e <Control-C> possono essere riassegnate ad altri tasti con il comando **stty**, ma nella pratica ciò avviene raramente. In questo capitolo ci si riferisce alle normali convenzioni.

Un core dump è l'immagine della memoria di un processo. Può essere usato a scopo di debugging.

Quando viene ricevuto un segnale, possono accadere due cose: se il processo ricevente ha designato una routine di gestione per quel particolare segnale, tale routine viene richiamata con informazioni sul contesto in cui il segnale è stato recapitato. Altrimenti, il kernel intraprende delle azioni predefinite in base al comportamento del processo. Tali azioni variano da un segnale all'altro: molti segnali terminano un processo, alcuni generano anche un core dump.

Quando si specifica una routine di gestione per un segnale in un programma, si parla di “catturare” o “intercettare” il segnale. Al termine della routine di gestione, l'esecuzione riprende dal punto in cui il segnale è stato ricevuto.

Per evitare l'arrivo di segnali, i programmi possono richiedere che siano ignorati o bloccati. Un segnale ignorato viene semplicemente rimosso senza effetti sul processo. Un segnale bloccato è messo in coda per il recapito, ma il kernel non richiede al processo di agire su di esso finché non viene esplicitamente sbloccato. Il gestore per un segnale sbloccato viene richiamato una sola volta, anche se il segnale è stato ricevuto più volte mentre il recapito era bloccato.

Nella Tabella 4.1 sono elencati alcuni segnali che dovrebbero essere noti a tutti gli amministratori. L'uso delle maiuscole deriva dal linguaggio C. Talvolta si vedono nomi di segnali scritti con un prefisso SIG (come SIGHUP) per motivi simili.

Esistono altri segnali non riportati nella Tabella 4.1, che per la maggior parte sono usati per riportare errori particolari come “istruzione illegale”. Per default, i segnali come quelli terminano con un core dump. Cattura e blocco sono generalmente permessi, perché alcuni programmi potrebbero essere in grado di provare a rimuovere il problema che ha causato l'errore, prima di continuare.

Tabella 4.1 Segnali che tutti gli amministratori dovrebbero conoscere.

#	Nome	Descrizione	Default	Cattura possibile?	Blocco possibile?	Core dump?
1	HUP	Hangup (riavvio)	Termina	Sì	Sì	No
2	INT	Interrupt	Termina	Sì	Sì	No
3	QUIT	Quit (uscita)	Termina	Sì	Sì	Sì
9	KILL	Kill (terminazione)	Termina	No	No	No
a	BUS	Errore bus	Termina	Sì	Sì	Sì
11	SEG V	Errore segmentazione	Termina	Sì	Sì	Sì
15	TER M	Terminazione software	Termina	Sì	Sì	No
a	STOP	Stop	Stop	No	No	No
a	TSTP	Stop tastiera	Stop	Sì	Sì	No
a	CONT	Continua dopo stop	Ignora	Sì	No	No
a	WINCH	Finestra cambiata	Ignora	Sì	Sì	No
a	USR1	Definito dall'utente	Termina	Sì	Sì	No
a	USR2	Definito dall'utente	Termina	Sì	Sì	No

a. Varia in base all'architettura hardware; vedere **man 7 signal**.

~~TSTP → CTRL-Z~~

Anche BUS e SEGV sono segnali di errore. Sono inclusi nella tabella perché molto comuni: nel 99 per cento dei casi in cui un programma si blocca, alla fine si presenta uno di questi due segnali, che di per sé non hanno uno specifico valore diagnostico: entrambi indicano un tentativo di usare o accedere alla memoria in modo non corretto.⁵

I segnali denominati KILL e STOP non possono essere catturati, bloccati o ignorati. Il segnale KILL distrugge il processo che lo riceve, STOP sospende l'esecuzione fino alla ricezione di un segnale CONT, che a sua volta può essere catturato o ignorato, ma non bloccato.

TSTP è una versione "soft" di STOP che si potrebbe descrivere come una richiesta di sospensione. È il segnale generato dal driver di terminale quando si preme la combinazione di tasti <Control-Z>. I programmi che catturano questo segnale solitamente azzerano il proprio stato, poi si autoinviano un segnale STOP per completare l'operazione di sospensione. In alternativa, i programmi possono ignorare TSTP per evitare la possibilità di sospensione da tastiera.

Gli emulatori di terminale inviano un segnale WINCH quando cambiano i loro parametri di configurazione (come il numero di righe nel terminale virtuale). Questa convenzione consente ai programmi che supportano gli emulatori, come gli editor di testi, di riconfigurarsi automaticamente in risposta alle modifiche. Se non si riescono a ridimensionare le finestre nel modo opportuno, conviene assicurarsi che WINCH sia generato e propagato correttamente.⁶

I segnali KILL, INT, TERM, HUP e QUIT sembrano indicare tutti la stessa cosa, ma i loro impieghi sono piuttosto diversi. La scelta di nomi così vaghi purtroppo non aiuta, in questo caso, perciò è utile fornire qualche informazione in più in modo da chiarire le cose.

- KILL non può essere bloccato e termina un processo al livello del kernel. Un processo non può mai "ricevere" effettivamente questo segnale.
- INT è il segnale inviato dal driver di terminale quando si preme la combinazione di tasti <Control-C>. Indica la richiesta di terminare l'operazione corrente. I programmi semplici dovrebbero uscire (se riescono a catturare il segnale) o consentire la loro terminazione, comportamento di default nel caso in cui il segnale non sia catturato. I programmi dotati di una riga di comando dovrebbero sospendere le loro operazioni, cancellare lo stato e attendere nuovamente l'input dell'utente.

5. Più specificamente, gli errori di bus sono causati da violazioni di requisiti di allineamento, o dall'uso di indirizzi errati. Le violazioni di segmentazione rappresentano violazioni di protezione, come tentativi di scrivere in porzioni dello spazio di indirizzamento riservate alla sola lettura.
6. Cosa più facile a dirsi che a farsi. Emulatore di terminale (ad esempio xterm), driver di terminale e comandi a livello utente possono tutti avere un ruolo nella propagazione di SIGWINCH. Tra i problemi comuni vi sono l'invio del segnale al solo processo in primo piano del terminale (invece che a tutti i processi associati al terminale stesso) e il fallimento nel propagare la notifica di un cambio di dimensioni in rete a un computer remoto. I protocolli come TELNET e SSH riconoscono la modifica delle dimensioni del terminale locale e comunicano tale informazione all'host remoto; i protocolli più semplici (ad esempio le linee seriali dirette) non sono in grado di farlo.

- TERM è una richiesta di terminare completamente l'esecuzione. Il processo che lo riceve deve cancellare il proprio stato e uscire.
- HUP ha due interpretazioni comuni: la prima è come richiesta di reset da parte di molti demoni. Se un demone è in grado di rileggere il proprio file di configurazione e aggiornare le modifiche senza necessità di riavvio, generalmente si può usare un segnale HUP per avviare tale operazione. Con la seconda interpretazione, i segnali HUP vengono generati dal driver di terminale in un tentativo di "cancellare" (ovvero terminare) i processi collegati a un particolare terminale. Si tratta di un residuo dei tempi dei terminali via cavo e delle connessioni via modem, da cui il nome "hangup" ("riaggancia" in italiano).

Le shell della famiglia C (`tcsch` et al.) solitamente rendono i processi in background immuni ai segnali HUP, in modo che possano continuare l'esecuzione dopo la disconnessione dell'utente. Gli utenti delle shell Bourne-ish (`ksh`, `bash` e così via) possono emulare questo comportamento con il comando `nohup`.

- QUIT è simile a TERM, ma per default produce un core dump se non è catturato. Alcuni programmi danno a questo segnale un'interpretazione diversa.

I segnali USR1 e USR2 non hanno un significato assegnato; possono essere impostati dai programmi nel modo definito dall'utente. Ad esempio, il server web Apache interpreta il segnale USR1 come richiesta di riavvio "gentile".

4.4 KILL E KILLALL: INVIARE SEGNALI

Come indica il nome, il comando `kill` è utilizzato generalmente per terminare (`kill` in inglese significa "uccidere") un processo. Il comando può inviare qualsiasi segnale, ma per default invia TERM. `kill` può essere usato da utenti normali sui propri processi, o dal superuser su qualsiasi processo. La sintassi è:

`kill [-segnale] pid`

dove `segnaile` è il numero o il nome simbolico del segnale da inviare (come mostrato nella Tabella 4.1) e `pid` è il numero identificativo del processo target. Un `pid` di -1 invia il segnale a tutti i processi tranne `init`.

Un comando `kill` senza un numero di segnale non garantisce la terminazione del processo, perché il segnale TERM può essere catturato, bloccato o ignorato. Il comando:

`kill -KILL pid`

"garantisce" che il processo terminerà, perché il segnale 9, KILL, non può essere catturato. Si è scritto "garantisce" tra virgolette perché i processi talvolta si bloccano in modo tale che neppure KILL riesce a terminarli (solitamente a causa di blocchi in fase di I/O, come nell'attesa di un disco che ha interrotto la rotazione). Solitamente il riavvio è l'unico modo per eliminare tali processi.

La maggior parte delle shell ha una propria implementazione di **kill** che segue la sintassi descritta in precedenza. Secondo la man page per il comando **kill** autonomo, il nome o numero di segnale deve essere preceduto dall'opzione **-s** (come in **kill -s HUP pid**). Tuttavia, poiché alcune shell non comprendono questa versione della sintassi, si suggerisce di mantenere la forma **-HUP**, anch'essa riconosciuta da **kill** autonomo. In tal modo non occorre preoccuparsi della versione di **kill** usata.

Se non si conosce il PID del processo a cui inviare il segnale, normalmente lo si può cercare con il comando **ps** descritto nel Paragrafo 4.7. Un'altra possibilità è quella di utilizzare il comando **killall**, che esegue automaticamente la ricerca. Ad esempio, per fare in modo che il demone **xinetd** ricarichi la propria configurazione, si può utilizzare:

```
$ sudo killall -USR1 xinetd
```

Se più processi corrispondono alla stringa fornita, **killall** invia il segnale a tutti.

Il comando **kill** vanilla ha una funzionalità simile, ma non sembra essere intelligente come **killall** nell'interpretare i nomi dei comandi. Conviene usare **killall**.

4.5 STATI DI UN PROCESSO

Il fatto che un processo esista non significa che sia abilitato a ricevere tempo di CPU. Occorre tenere conto dei quattro stati di esecuzione elencati nella Tabella 4.2.

Un processo runnable è pronto all'esecuzione non appena è disponibile tempo di CPU. Ha acquisito tutte le risorse che gli servono e attende soltanto tempo di CPU per elaborare i dati. Non appena il processo effettua una chiamata di sistema che non può essere completata immediatamente (come la richiesta di leggere parte di un file), Linux lo pone nello stato sleeping.

I processi sleeping sono in attesa che si verifichi un evento specifico. Shell interattive e demoni di sistema passano la maggior parte del tempo in questo stato, in attesa di input del terminale o di connessioni di rete. Poiché un processo sleeping è effettivamente bloccato finché la sua richiesta non è stata soddisfatta, non gli viene assegnato tempo di CPU finché non riceve un segnale.

Alcune operazioni fanno entrare i processi in uno stato sleeping senza possibilità di interruzione. Questo stato solitamente è transitorio e non viene osservato nell'output di **ps** (è indicato da una D nella colonna STAT; vedere la Tabella 4.3 più avanti in questo capitolo). Tuttavia, alcune situazioni degeneri possono causa-

Tabella 4.2 Stati di un processo.

Stato	Significato
Runnable	Il processo può essere eseguito.
Sleeping	Il processo è in attesa di una risorsa.
Zombie	Il processo sta tentando di terminare.
Stopped	Il processo è sospeso (non è consentita l'esecuzione).

re la sua persistenza. La causa più comune è rappresentata da problemi di server su un filesystem NFS montato con l'opzione "hard". Poiché i processi nello stato sleeping senza possibilità di interruzione non possono essere risvegliati nemmeno per rispondere a un segnale, non possono essere terminati. Per eliminarli, occorre risolvere il problema che ha determinato la situazione o riavviare.

Gli zombie sono processi che hanno concluso l'esecuzione, ma il cui codice di uscita non è ancora stato raccolto. Quando si vedono in giro degli zombie, occorre verificare i rispettivi PPID con `ps` per determinare da dove provengono.

Per quanto riguarda i processi stopped, l'amministratore ha proibito la loro esecuzione. I processi entrano in questo stato alla ricezione di un segnale STOP o TSTP e sono riavviati con CONT. Questo stato è simile a sleeping, ma non vi è modo di uscirne se non grazie all'azione di un altro processo che effettua il risveglio (o la terminazione).

4.6 NICE E RENICE: INFLUENZARE LA PRIORITÀ DI SCHEDULAZIONE

Il "nice" di un processo rappresenta un suggerimento al kernel, in forma di numero, per indicare come trattare il processo in relazione ad altri che gli contendono la CPU. Questo strano nome deriva dal fatto che il numero determina la "cortesia" (nice in inglese) con cui ci si rapporta ad altri utenti del sistema. Un valore nice elevato significa una bassa priorità per il processo: la cortesia è alta. Un valore basso o negativo significa alta priorità: la cortesia è scarsa. L'intervallo dei valori utilizzabili per nice va da -20 a +19.

A meno che l'utente non intraprenda un'azione particolare, un processo appena creato eredita il valore nice del proprio genitore. Il proprietario del processo può aumentare il suo valore nice, ma non diminuirlo, nemmeno per farlo tornare al suo valore di default. Questa restrizione evita che processi con bassa priorità possano avere figli con alta priorità. Il superuser può impostare i valori nice in modo del tutto arbitrario.

Oggi non capita spesso di impostare le priorità manualmente. Nei sistemi degli anni '70 e '80 le prestazioni erano notevolmente influenzate dal processo a cui era assegnato il tempo di CPU, mentre oggi, con sistemi che dispongono quasi sempre di CPU con potenza più che adeguata, lo scheduler riesce solitamente a servire bene tutti i processi. L'aggiunta di classi di schedulazione fornisce agli sviluppatori un controllo in più nei casi in cui è essenziale una bassa latenza di risposta.

Le prestazioni dei sistemi di I/O non sono state al passo con l'aumento di velocità delle CPU, e il principale collo di bottiglia nella maggior parte dei sistemi è rappresentato dalle unità a disco. Sfortunatamente il valore nice di un processo non ha effetto sulla gestione da parte del kernel della memoria o dell'I/O; per questo, processi con valore nice elevato possono comunque monopolizzare una quota sproporzionata di tali risorse.

Il valore nice di un processo può essere impostato al momento della creazione con il comando `nice` e poi regolato con il comando `renice`. `nice` accetta come ar-

gomento una riga di comando, mentre **renice** richiede un PID o un nome utente. A confondere le cose vi è il fatto che **renice** richiede una priorità assoluta, mentre **nice** richiede un *incremento* di priorità che poi viene aggiunto o sottratto dalla priorità corrente della shell.

Alcuni esempi:

```
$ nice -n 5 ~/bin/longtask // Abbassa la priorità (alza il nice) di 5
$ sudo renice -5 8829    // Imposta il valore nice a -5
$ sudo renice 5 -u mario  // Imposta il valore nice dei processi di mario a 5
```

Per complicare le cose, una versione di **nice** è integrata nella shell C e in altre shell comuni (ma non in **bash**). Se non si digita il percorso completo per il comando **nice**, si utilizza la versione della shell e non quella del sistema operativo. La presenza di questo doppione può portare confusione, perché il **nice** della shell e quello del sistema utilizzano una sintassi diversa: in quella della shell l'incremento di priorità va espresso come **+incr** o **-incr**, mentre nel comando autonomo l'incremento di priorità si indica preceduto dall'opzione **-n**.⁷

Il processo per cui si utilizza più spesso **nice** oggi è **xntpd**, il demone di sincronizzazione del clock. Poiché la pronta risposta della CPU ha per esso importanza critica, il processo viene eseguito solitamente con un valore nice inferiore di circa 12 a quello di default (e quindi con una priorità più alta del normale).

Se un processo impazzisce e porta il carico medio di sistema a 65, potrebbe essere necessario usare **nice** per avviare una shell ad alta priorità prima di poter eseguire comandi che consentano di determinare le cause del problema; altrimenti, potrebbe risultare difficile eseguire perfino comandi molto semplici.

4.7 PS: MONITORARE I PROCESSI

ps è il principale strumento dell'amministratore di sistema per monitorare i processi. Lo si può usare per visualizzare PID, UID, priorità e terminale di controllo di un processo; fornisce anche informazioni sulla quantità di memoria che un processo sta usando, sulla quantità di tempo di CPU consumata e sullo stato corrente (running, stopped, sleeping e così via). I processi zombie vengono evidenziati in un listato **ps** come **<defunct>**.

Il comportamento di **ps** tende a variare notevolmente nelle varie versioni di UNIX e molte implementazioni sono divenute piuttosto complesse negli ultimi anni. Nello sforzo di accontentare persone abituate ai comandi **ps** di altri sistemi, Linux fornisce una versione multivariata che comprende le opzioni di molte altre implementazioni e usa una variabile d'ambiente per determinare la personalità che deve assumere.

7. In realtà la situazione è ancora peggiore: il **nice** autonomo interpreta **nice -5** come un incremento *positivo* di 5, mentre il **nice** integrato nella shell interpreta la stessa sintassi come incremento *negativo* di 5.

Non è il caso di allarmarsi per tutta questa complessità: riguarda principalmente gli sviluppatori del kernel, non gli amministratori di sistema. Questi ultimi usano spesso **ps**, ma possono accontentarsi di poche implementazioni specifiche.

Si può ottenere una panoramica generale di tutti i processi in esecuzione sul sistema con **ps aux**. Ecco un esempio (abbiamo rimosso la colonna START e selezionato soltanto alcune righe dell'output in modo da far stare l'esempio nella pagina):

```
$ ps aux
USER     PID %CPU %MEM   VSZ   RSS TTY STAT TIME COMMAND
root      1  0.1  0.2  3356   560 ? S 0:00 init [5]
root      2  0    0      0      0 ? SN 0:00 [ksoftirqd/0]
root      3  0    0      0      0 ? S< 0:00 [events/0]
root      4  0    0      0      0 ? S< 0:00 [khelper]
root      5  0    0      0      0 ? S< 0:00 [kacpid]
root     18  0    0      0      0 ? S< 0:00 [kblockd/0]
root     28  0    0      0      0 ? S 0:00 [pdfflush]
...
root    196  0    0      0      0 ? S 0:00 [kjournald]
root   1050  0  0.1  2652   448 ? S<s 0:00 udevd
root   1472  0  0.3  3048  1008 ? S<s 0:00 /sbin/dhclient -l
root   1646  0  0.3  3012  1012 ? S<s 0:00 /sbin/dhclient -l
root   1733  0    0      0      0 ? S 0:00 [kjournald]
root   2124  0  0.3  3004  1008 ? Ss 0:00 /sbin/dhclient -l
root   2182  0  0.2  2264   596 ? Ss 0:00 syslogd -m 0
root   2186  0  0.1  2952   484 ? Ss 0:00 klogd -x
rpc    2207  0  0.2  2824   580 ? Ss 0:00 portmap
rpcuser 2227  0  0.2  2100   760 ? Ss 0:00 rpc.statd
root   2260  0  0.4  5668  1084 ? Ss 0:00 rpc.idmapd
root   2336  0  0.2  3268   556 ? Ss 0:00 /usr/sbin/acpid
root   2348  0  0.8  9100  2108 ? Ss 0:00 cupsd
root   2384  0  0.6  4080  1660 ? Ss 0:00 /usr/sbin/sshd
root   2399  0  0.3  2780   828 ? Ss 0:00 xinetd -stayalive
root   2419  0  1.1  7776  3004 ? Ss 0:00 sendmail: accept
...
I nomi tra parentesi quadre non indicano comandi reali, ma thread del kernel schedulati come processi.
```

Il significato di ciascun campo è spiegato nella Tabella 4.3.

Un altro insieme di argomenti è fornito da **lax**, che offre informazioni più tecniche ed è anche leggermente più veloce nell'esecuzione perché non richiede la traduzione di ciascun UID in un nome utente. L'efficienza può essere importante se il sistema è già appesantito da altri processi.

Di seguito è riportato un esempio abbreviato; **ps lax** include campi come l'ID del processo padre (PPID), il valore nice (NI) e la risorsa che il processo sta aspettando (WCHAN).

```
$ ps lax
F  UID     PID  PPID  PRI  NI   VSZ   RSS  WCHAN  STAT  TIME  COMMAND
4    0       1    0   16   0  3356   560  select  S  0:00  init [5]
```

Tabella 4.3 Spiegazione dell'output di ps aux.

Campo	Contenuto
USER	Nome utente del proprietario del processo.
PID	ID del processo.
%CPU	Percentuale di CPU che il processo sta usando.
%MEM	Percentuale della memoria reale che il processo sta usando.
VSZ	Dimensione virtuale del processo.
RSS	Dimensione dell'insieme residente (numero di pagine in memoria).
TTY	ID del terminale di controllo.
STAT	Stato corrente del processo: R = Runnable D = In pausa non interrompibile S = Sleeping (< 20 sec) T = Tracciato o fermato Z = Zombie Parametri aggiuntivi: W = È stato eseguito lo swapping del processo. < = Il processo ha priorità più alta del normale. N = Il processo ha priorità più bassa del normale. L = Alcune pagine sono bloccate in core. s = Il processo è un leader di sessione.
START	Tempo a cui è stato avviato il processo.
TIME	Tempo di CPU consumato dal processo.
COMMAND	Nome del comando e argomenti. ^a

a. Questa informazione può essere modificata dai programmi, perciò non offre sempre una rappresentazione precisa della riga di comando effettiva.

1	0	2	1	34	19	0	0	ksofti	SN	0:00	[ksoftirqd/0]
1	0	3	1	5	-10	0	0	worker	S<	0:00	[events/0]
1	0	4	3	5	-10	0	0	worker	S<	0:00	[khelper]
5	0	2186	1	16	0	2952	484	syslog	Ss	0:00	klogd -x
5	32	2207	1	15	0	2824	580	-	Ss	0:00	portmap
5	29	2227	1	18	0	2100	760	select	Ss	0:00	rpc.statd
1	0	2260	1	16	0	5668	1084	-	Ss	0:00	rpc.idmapd
1	0	2336	1	21	0	3268	556	select	Ss	0:00	acpid
5	0	2384	1	17	0	4080	1660	select	Ss	0:00	sshd
1	0	2399	1	15	0	2780	828	select	Ss	0:00	xinetd -sta
5	0	2419	1	16	0	7776	3004	select	Ss	0:00	sendmail: a

4.8 TOP: MONITORARE I PROCESSI IN MODO ANCORA MIGLIORE

Poiché i comandi come **ps** forniscono soltanto un'istantanea del sistema in un momento specifico, spesso è difficile ottenere l'immagine complessiva di ciò che sta accadendo realmente. Il comando **top** fornisce un riepilogo aggiornato regolarmente con i processi attivi e l'uso delle risorse.

Per esempio:

```
top - 16:37:08 up 1:42, 2 users, load average: 0.01, 0.02, 0.06
Tasks: 76 total, 1 running, 74 sleeping, 1 stopped, 0 zombie
Cpu(s): 1.1% us, 6.3% sy, 0.6% ni, 88.6% id, 2.1% wa, 0.1% hi, 1.3% si
Mem: 256044ktotal, 254980kused, 1064kfree, 15944kbuffers
Swap: 524280ktotal, 0kused, 524280kfree, 153192kcached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3175 root 15 0 35436 12m 4896 S 4.0 5.2 01:41.9 X
3421 root 25 10 29916 15m 9808 S 2.0 6.2 01:10.5 rhn-applet-gui
  1 root 16 0 3356 560 480 S 0.0 0.2 00:00.9 init
  2 root 34 19 0 0 0 S 0.0 0 00:00.0 ksoftirqd/0
  3 root 5 -10 0 0 0 S 0.0 0 00:00.7 events/0
  4 root 5 -10 0 0 0 S 0.0 0 00:00.0 khelper
  5 root 15 -10 0 0 0 S 0.0 0 00:00.0 kacpid
  18 root 5 -10 0 0 0 S 0.0 0 00:00.0 kblockd/0
  28 root 15 0 0 0 0 S 0.0 0 00:00.0 pdflush
  29 root 15 0 0 0 0 S 0.0 0 00:00.3 pdflush
  31 root 13 -10 0 0 0 S 0.0 0 00:00.0 aio/0
  19 root 15 0 0 0 0 S 0.0 0 00:00.0 khubd
  30 root 15 0 0 0 0 S 0.0 0 00:00.2 kswapd0
  187 root 6 -10 0 0 0 S 0 0 00:00.0 kmirrord/0
  196 root 15 0 0 0 0 S 0 0 00:01.3 kjournald
...

```

Per default la visualizzazione è aggiornata ogni 10 secondi. I processi più attivi appaiono in alto. **top** accetta l'input dalla tastiera e consente di inviare segnali e di eseguire **renice** sui processi, in modo da poter osservare come le azioni dell'utente influiscono sulla condizione complessiva della macchina.

L'utente root può eseguire **top** con l'opzione **q** per assegnargli la massima priorità possibile; questo può essere molto utile quando si cerca di tracciare un processo che ha già messo in ginocchio il sistema.

4.9 IL FILESYSTEM /PROC

Le versioni Linux di **ps** e **top** leggono le proprie informazioni di stato sui processi dalla directory **/proc**, uno pseudo filesystem in cui il kernel espone una varietà di informazioni interessanti sullo stato del sistema. Nonostante il nome **/proc** (e il nome del tipo di filesystem sottostante, "proc"), le informazioni non si limitano ai processi, ma comprendono dati di stato e statistiche generate dal kernel. È anche possibile modificare alcuni parametri scrivendo nel file **/proc** appropriato; nel Paragrafo 28.4 sono presentati alcuni esempi.

Benché sia più facile accedere ad alcune informazioni tramite comandi front-end come **vmstat** e **ps**, alcune tra le informazioni meno comuni devono essere lette direttamente da **/proc**. È utile guardarsi attorno in questa directory per familiarizzare con le informazioni disponibili. **man proc** elenca alcuni trucchi e suggerimenti.

Poiché il kernel crea il contenuto dei file **/proc** al momento (quando sono letti), la maggior parte di questi file appare vuota quando si richiede un elenco con **ls -l**; oc-

corre utilizzare **cat** o **more** per vedere il loro contenuto effettivo. È necessario procedere con cautela: alcuni file contengono o sono collegati a dati binari che potrebbero causare problemi all'emulatore di terminale, se visualizzati direttamente.

Le informazioni specifiche dei processi sono suddivise in sottodirectory con nomi corrispondenti ai PID. Per esempio, **/proc/1** è sempre la directory che contiene informazioni su **init**. Nella Tabella 4.4 sono elencati i file più utili con informazioni sui processi.

Tabella 4.4

File con informazioni sui processi in /proc (sottodirectory numerate).

File	Contenuto
cmd	Comando o programma che esegue il processo.
cmdline^a	Riga di comando completa del processo (separata con null).
cwd	Collegamento simbolico alla directory corrente del processo.
environ	Variabili d'ambiente del processo (separate con null).
exe	Collegamento simbolico al file di esecuzione.
fd	Sottodirectory con collegamenti per ciascun descrittore di file aperto.
maps	Informazioni sulla mappatura della memoria (segmenti condivisi, librerie e così via).
root	Collegamento simbolico alla directory root del processo (impostata con chroot).
stat	Informazioni generali sullo stato del processo (si interpretano meglio con ps).
statm	Informazioni sull'uso della memoria.

a. Potrebbe non essere disponibile se è stato eseguito lo swapping del processo dalla memoria.

Le singole componenti contenute nei file **cmdline** ed **environ** sono separate da caratteri null e non da caratteri di nuova riga. È possibile filtrarne il contenuto con **tr “\000” “\n”** per renderlo più leggibile. La sottodirectory **fd** rappresenta i file aperti in forma di collegamenti simbolici. I descrittori di file connessi a pipe o socket di rete non hanno un nome di file associato; il kernel fornisce una descrizione generica come target del collegamento. Il file **maps** può essere utile per determinare le librerie a cui un programma è collegato o da cui dipende.

4.10 STRACE: TRACCIARE SEGNALI E CHIAMATE DI SISTEMA

Su un sistema UNIX tradizionale può risultare difficile determinare che cosa stia effettivamente facendo un processo. Potrebbe essere necessario fare delle ipotesi in base a dati indiretti ricavati dal filesystem e da strumenti come **ps**. Linux invece consente di osservare direttamente un processo con il comando **strace**, che mostra ogni chiamata di sistema effettuata dal processo e ogni segnale ricevuto. Si può anche agganciare **strace** a un processo in esecuzione, curiosare un po' e poi staccarsi dal processo senza disturbarlo.⁸

8. Non sempre ciò è possibile. In alcuni casi, **strace** può interrompere chiamate di sistema; il processo monitorato deve quindi essere preparato a riavivarle. Questa è una regola standard per l'igiene del software UNIX, ma non è sempre rispettata.

Anche se le chiamate di sistema avvengono a un livello di astrazione relativamente basso, solitamente è possibile trarre utili informazioni sull'attività di un processo dall'output di **strace**. Ad esempio, il log seguente è stato prodotto eseguendo **strace** su una copia attiva di **top**:

```
$ sudo strace -p 5810
gettimeofday({1116193814, 213881}, {300, 0})      = 0
open("/proc", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 7
fstat64(7, {st_mode=S_IFDIR|0555, st_size=0, ...})    = 0
fcntl64(7, F_SETFD, FD_CLOEXEC)                   = 0
getdents64(7, /* 36 entries */, 1024)   = 1016
getdents64(7, /* 39 entries */, 1024)   = 1016
stat64("/proc/1", {st_mode=S_IFDIR|0555, st_size=0, ...}) = 0
open("/proc/1/stat", O_RDONLY)                     = 8
read(8, "1 (init) S 0 0 0 0 -1 4194560 73"..., 1023) = 191
close(8) = 0
...
...
```

Oltre a visualizzare il nome di ciascuna chiamata di sistema effettuata dal processo, **strace** decodifica gli argomenti e mostra il codice di risultato restituito dal kernel.

In questo esempio, **top** per prima cosa verifica l'ora corrente, poi apre la directory **/proc**, ne richiede lo stato e ne legge il contenuto, ottenendo così un elenco di processi attualmente in esecuzione. Poi richiede lo stato della directory che rappresenta il processo **init** e quindi apre **/proc/1/stat** per leggere le informazioni di stato di **init**.

4.11

PROCESSI RUNAWAY

Per ulteriori informazioni sui processi runaway, vedere il Paragrafo 25.4.

Esistono due versioni di processi runaway: i processi utente che consumano quantità eccessive di risorse di sistema, come tempo di CPU o spazio su disco, e i processi di sistema che impazziscono improvvisamente. Quelli del primo tipo non corrispondono necessariamente a malfunzionamenti, potrebbero semplicemente richiedere molte risorse. Si suppone sempre che i processi di sistema si comportino in maniera ragionevole.

È possibile identificare i processi che utilizzano troppo tempo di CPU esaminando l'output di **ps** o **top**. Se appare evidente che un processo utente consuma più tempo di CPU rispetto al previsto, occorre prenderlo in esame. Il primo passo, su un server o un sistema condiviso, consiste nel contattare il proprietario del processo e chiedergli che cosa succede. Se non si riesce a trovare il proprietario, occorre guardare un po' in giro. Normalmente si dovrebbe evitare di spulciare nelle home directory degli utenti, ma lo si può fare quando si cerca di risalire al codice sorgente di un processo runaway per capire che cosa stia facendo.

Esistono due ragioni che possono spingere a determinare che cosa stia cercando di fare un processo, prima di interferire con esso: la prima è che il processo potrebbe essere legittimo e importante per l'utente; non ha senso terminare processi a caso soltanto perché usano parecchio tempo di CPU. La seconda è che il processo potrebbe essere maligno e distruttivo; in questo caso, occorre sapere che cosa stia facendo (ad esempio, potrebbe cercare di rubare delle password) per poter rimediare al danno.

Se non è possibile determinare la ragione dell'esistenza di un processo, conviene sosperderlo con un segnale STOP e inviare un messaggio email al suo proprietario spiegando che cosa è successo. In seguito è possibile riavviare il processo con un segnale CONT. Occorre tenere presente che alcuni processi potrebbero essere danneggiati da una sospensione lunga, perciò questa procedura non è del tutto priva di rischi. Ad esempio, al riavvio dopo la sospensione un processo potrebbe scoprire che alcune delle sue connessioni di rete sono state interrotte.

Se un processo usa troppo tempo di CPU, ma appare comportarsi in modo ragionevole e corretto, è possibile utilizzare **renice** per assegnargli un valore nice più alto (priorità più bassa) e chiedere al proprietario di usare lui stesso **nice** in futuro.

I processi che usano troppa memoria rispetto alla quantità di RAM fisica del sistema possono causare seri problemi di prestazioni. È possibile verificare la dimensione in memoria dei processi usando **top**. La colonna VIRT dell'output mostra la quantità totale di memoria virtuale allocata per ciascun processo, mentre la colonna RES mostra la porzione di tale memoria attualmente mappata a pagine di memoria specifiche (il cosiddetto "insieme residente"). Entrambi questi valori possono includere risorse condivise, come librerie, e questo può renderli inaffidabili. Una misura più diretta del consumo di memoria specifico di un processo si trova nella colonna DATA, che non è visualizzata per default. Per aggiungere questa colonna alla visualizzazione dell'output di **top**, occorre premere il tasto f mentre **top** è in esecuzione e selezionare DATA dall'elenco. Il valore DATA indica la quantità di memoria nei segmenti dati e stack di ciascun processo, perciò è relativamente specifico dei singoli processi (a parte i segmenti di memoria condivisa). Convieni cercare eventuali aumenti nel tempo e dimensioni che rimangono fisse.

I processi runaway che producono output possono riempire un intero filesystem, causando numerosi problemi. Quando un filesystem si riempie, molti messaggi vengono riportati sulla console e tentativi di scrittura sul filesystem in questione produrranno messaggi di errore. In questa situazione, la prima cosa da fare è sospendere il processo che stava riempiendo il disco. Se quest'ultimo disponeva di una quantità di spazio libero ragionevole, è presumibile che qualcosa sia andato storto, qualora si riempia improvvisamente. Non esiste un comando analogo a **ps** che indichi chi sta consumando spazio su disco alla massima velocità, ma diversi strumenti possono identificare i file attualmente aperti e i processi che li utilizzano. Vedere le informazioni su **fuser** e **lsof** nel Paragrafo 5.2.

Convieni sospendere tutti i processi sospetti finché non si trova quello che causa il problema, ricordando però di riavviare poi quelli non colpevoli. Quando si trova il processo dannoso, occorre rimuovere i file da esso creati. Un vecchio scherzo, ben noto, consiste nell'avviare un loop infinito dalla shell:

```
while 1
    mkdir adir
    cd adir
    touch afile
end
```

Talvolta capita di vedere questo programma eseguito da un sistema accessibile al pubblico che è stato abbandonato per errore senza disconnettersi. Il ciclo non consuma tanto spazio reale su disco, ma riempie la tabella di inode del filesystem e impedisce ad altri utenti di creare nuovi file. Non vi è molto da fare, a parte eliminare le conseguenze e avvisare gli utenti di proteggere i propri account. Poiché l'albero di directory lasciato da questo programma è solitamente troppo grande perché **rm -r** sia in grado di gestirlo, potrebbe essere necessario scrivere uno script che scenda in fondo all'albero e rimuova le directory risalendo.

Se il problema si verifica in **/tmp** e questa directory è stata impostata come filesystem separato, si può reinizializzare **/tmp** con **mkfs** invece di tentare di eliminare singoli file. Si rimanda al Capitolo 7 per ulteriori informazioni sulla gestione dei filesystem.

4.12 ESERCIZI

- E4.1 Spiegare la relazione tra l'UID di un file, l'UID reale di un processo in esecuzione e l'UID effettivo (EUID). Oltre al controllo di accesso sui file, qual è lo scopo dell'UID effettivo di un processo?
- E4.2 Un utente del sito ha avviato un processo in esecuzione da parecchio tempo e che consuma una significativa quantità delle risorse di sistema.
- Come si riconosce un processo che consuma molte risorse?
 - Supponendo che il processo sia legittimo e non debba essere terminato, indicare i comandi da utilizzare per sosporarlo temporaneamente mentre lo si esamina.
 - In seguito si scopre che il processo appartiene al capo e deve continuare l'esecuzione. Indicare i comandi da usare per riavviarlo.
 - In alternativa, supponendo che il processo debba essere terminato, quale segnale si deve inviare, e perché? E se occorre la garanzia che il processo termini davvero?
- E4.3 Trovare un processo con un memory leak (scrivere un programma appropriato, nel caso non se ne abbia uno a disposizione). Usare **ps** o **top** per monitorare l'uso della memoria da parte del programma durante l'esecuzione.
- ★ E4.4 Scrivere un semplice script Perl che elabori l'output di **ps** per determinare i valori totali di VSZ e RSS dei processi in esecuzione sul sistema. Come si rapportano questi numeri alla quantità effettiva di memoria fisica e all'area di swap del sistema?