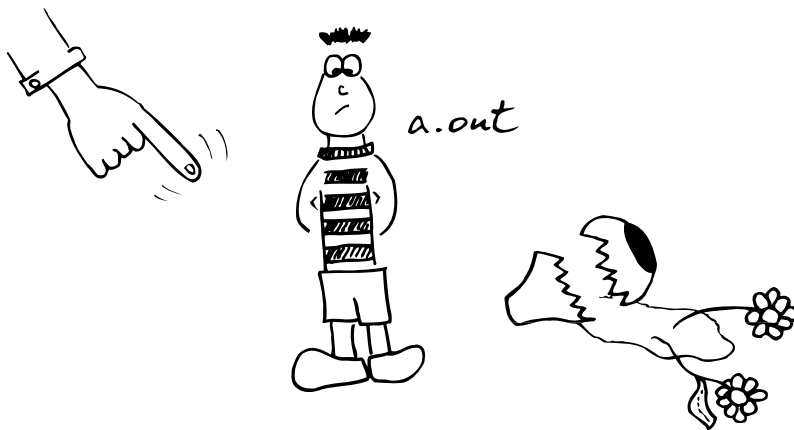


Řízení procesů



Proces je abstrakce, kterou Linux používá pro reprezentaci běžícího programu. Je to objekt, přes který lze řídit a sledovat využití paměti, procesoru a vstupně-výstupních prostředků pro daný program.

Součástí linuxové a unixové filozofie je myšlenka, že co nejvíce práce by se mělo provádět v kontextu procesů a nezpracovávat přímo jádrem. Systémové i uživatelské procesy se řídí stejnými pravidly, takže můžete pro jejich řízení použít stejnou sadu nástrojů.

Součásti procesu

Proces se skládá z adresového prostoru a ze sady datových struktur v jádře. Adresní prostor je sada paměťových stránek,* které jádro označilo pro využití daným procesem. Obsahuje kód a knihovny, které proces provozuje, proměnné procesu, jeho zásobníky a další dodatečné informace, které jádro potřebuje během doby, kdy proces běží. Linux podporuje virtuální paměť, a proto nemusí existovat vztah mezi umístěním stránky v adresovém prostoru a jejím umístěním ve fyzické paměti počítače nebo v odkládacím prostoru.

Interní datové struktury jádra zaznamenávají různé kusy informací o každém procesu. Mezi nejdůležitější informace patří:

- Mapa adresního prostoru procesu.
- Aktuální stav procesu (spí, zastaven, spustitelný atd.).

* Stránky jsou jednotky správy paměti; na PC je to obvykle 4 Kb.

- Provozní priorita procesu.
- Informace o prostředcích, které proces využívá.
- Informace o souborech a síťových portech, které proces otevírá.
- Signálová maska procesu (záznam o tom, které signály jsou blokovány).
- Majitel procesu.

Některé z těchto atributů může sdílet několik procesů, které tak vytvoří „skupinu vláken“, což je linuxová analogie vícevláknového procesu v Unixu. I když mohou sdílet adresový prostor, každý člen skupiny vláken má vlastní prioritu a stav. V praxi to znamená, že jen velmi málo procesů, o něž by se správce jinak měl zajímat, běží po několika vláknech, a dokonce ani ty, které tak činí (např. proces `BIND 9 named`), nevyžadují na této úrovni pozornost správce.

Mnohé parametry spojené s procesem přímo ovlivňují jeho provoz: objem času centrálního mikroprocesoru pro proces, soubory, ke kterým smí proces přistupovat, atd. V následující části rozebíráme význam a důležitost parametrů, které jsou z hlediska správce systému nejdůležitější. Tyto atributy jsou společné pro všechny verze Unixu a Linuxu.

PID: identifikační číslo procesu

Jádro přiřazuje každému procesu unikátní identifikační číslo. Většina příkazů a systémových volání ovládajících procesy od vás vyžaduje zadat PID cíle vaší operace. Čísla PID se přiřazují ve vzestupném pořadí vždy při vytvoření procesu.

PPID: rodičovský PID

Linux neposkytuje takové systémové volání, které by vytvořilo nový proces provozující konkrétní program. Místo toho musí existující proces vytvořit nový proces tak, že klonuje sám sebe. Tento klon pak smí vyměnit program, který právě provozuje, za jiný program.

Když je proces klonován, původní proces se nazývá rodič a kopie se nazývá potomek. Atribut PPID procesu je PID rodiče, ze kterého byl proces klonován.* Narazíte-li na nerozpoznaný (a nejspíš i nevycálaný) proces, rodičovský PID je užitečná informace. Když znáte původce procesu (ať už je to shell nebo jiný program), snáze pochopíte jeho účel a význam.

UID a EUID: reálná a provozní identita uživatele

Další informace o UID naleznete na str. 132.

UID procesu je identifikační číslo uživatele, který proces vytvořil, nebo přesněji řečeno je to kopie hodnoty EUID rodičovského procesu. S procesem smí obvykle manipulovat pouze tvůrce (neboli „majitel“) a superuživatel.

EUID je „provozní“ identifikační číslo uživatele; to je další UID, pomocí něhož se určí, jaké prostředky a soubory smí daný proces v kterémkoliv okamžiku používat. Pro většinu procesů jsou UID a EUID shodné a mezi obvyklé výjimky patří programy, které mají zapnutý stav `setuid`.

*Alespoň zpočátku. Když původní rodič zemře, novým rodičem se stává proces `init` (proces č. 1).

Proč má proces UID i EUID? Prostě proto, že je užitečné udržovat rozdíl mezi identitou a oprávněním a protože program s příznakem `setuid` třeba nechce pracovat s rozšířenými právy neustále a vždy. Provozní UID lze zapínat a vypínat a tím povolovat nebo zakazovat dodatečná oprávnění, která toto UID poskytuje.

Linux také sleduje „uložená UID“, což je kopie UID v okamžiku, kdy začne program běžet. Pokud proces nepodnikne kroky k likvidaci tohoto uloženého UID, zůstane dostupné pro využití jako reálné nebo provozní UID. Konzervativně napsaný `setuid` program se tedy může vzdát svých speciálních oprávnění pro většinu své činnosti a používat tato oprávnění jen v konkrétních okamžicích, kdy jsou potřebná dodatečná privilegia.

Linux rovněž definuje nestandardní procesní parametr FSUID, který řídí zjišťování oprávnění souborového systému. Mimo jádro se používá zřídka.

Důsledky systému vícenásobné identity mohou být poněkud choulostivé. Chcete-li se v tom trochu porýpat, podívejte se do online knihy *Secure Programming for Linux and Unix HOWTO*, je skvělá. Najdete ji na www.dwheeler.com.

GID a EGID: reálná a provozní identita skupiny

Další informace o skupinách naleznete na str. 133.

GID je skupinové identifikační číslo procesu. EGID má ke GID stejný vztah jako EUID k UID v tom smyslu, že se dá „vylepšit“ spuštěním programu s nastaveným příznakem `setgid`. Linux udržuje uložené GID ve stejném duchu jako uložené UID.

V linuxových systémech může být proces členem mnoha skupin najednou. Úplný seznam skupin je uložen odděleně od rozlišených GID a EGID. Určení přístupových oprávnění obvykle bere v úvahu EGID a dodatečný seznam skupin, ale nikoliv GID.

GID vyjde na scénu jen tehdy, když proces vytváří nové soubory. V závislosti na tom, jak jsou nastavena přístupová práva v souborovém systému, nové soubory mohou přijmout GID procesu, který je vytvořil.

Hodnota ohleduplnosti

Plánovací priorita procesu určuje, kolik času centrálního mikroprocesoru proces dostane. Pro výpočet priorit používá jádro dynamický algoritmus, který bere v úvahu množství času spotřebované procesem v nedávné minulosti a dobu, po kterou proces čekal na pokračování svého běhu. Jádro navíc přihlíží k hodnotě nastavené správcem, která se většinou nazývá „hodnota nice“, tedy česky „hodnota ohleduplnosti“; nazývá se tak proto, že říká, jak ohleduplný chcete být k ostatním uživatelům systému. Otázku ohleduplnosti procesů probíráme podrobně na straně 99.

Ve snaze poskytnout lepší podporu aplikacím, které nesnižují výkon systému, byl klasický unixový plánovací model doplněn v Linuxu o „plánovací třídy“. V současnosti existují tři plánovací třídy a každý proces je přiřazen jedné třídě. Bohužel, třídy v reálném čase se používají málokdy a ani nemají náležitou podporu z příkazového řádku. Všechny procesy v systému používají klasický plánovač založený na ohleduplnosti a v této knize budeme hovořit pouze o něm. Pokud by vás zajímaly další podrobnosti o plánování v reálném čase, navštivte www.realtimelinuxfoundation.org.

Řídicí terminál

Většina nedaemonovských procesů má přiřazený řídicí terminál, který určuje implicitní připojení na standardní vstup, standardní výstup a standardní chybové kanály. Když spustíte příkaz z shellu, zpravidla se řídicím terminálem procesu stane váš terminál. Koncepce řídicího terminálu má vliv i na distribuci signálů, které podrobněji rozebíráme na straně 95.

Životní cyklus procesu

Aby se vytvořil nový proces, musí proces kopírovat sám sebe systémovým voláním **fork**. Toto volání **fork** vytvoří kopii původního procesu, která je z větší části identická s rodičem. Nový proces má unikátní PID a své vlastní účetní informace.

Volání **fork** má unikátní vlastnost: vrací dvě odlišné hodnoty. Z hlediska potomka vrací nulu. Rodič na druhé straně dostává PID nově vytvořeného potomka. Oba procesy jsou v ostatních ohledech identické, a musejí tedy oba prozkoumat návratovou hodnotu, a tak zjistit, jakou roli mají hrát.

Po volání **fork** potomek často použije jedno z volání rodiny **exec**, aby se spustil nový program.* Tato volání totiž pouze změní text programu, který proces vykonává, a obnoví datové a zásobníkové segmenty do předem definovaného počátečního stavu. Různé formy volání **exec** se liší jen ve způsobu, kterým specifikují argumenty příkazového řádku a prostředí, které se předá novému programu.

V Linuxu je definován alternativní příkaz k **fork**, který se nazývá **clone**. Vytváří množinu procesů, které sdílejí paměť, prostor V/V nebo obojí. Tato funkce je analogická s funkcí na vytváření vláken, která existuje ve většině verzí Unixu, avšak každé vlákno představuje plnohodnotný proces, nikoli pouze specializovaný „vláknový“ objekt.

*Další informace o zavádění systému a daemonech **init** naleznete v kapitole 2.*

Když se systém spouští, jádro automaticky vytvoří a nainstaluje několik procesů. Nejvýznamnější z nich je proces **init**, který má vždy číslo procesu 1. Proces **init** odpovídá za spouštění systémových startovních skriptů. Všechny procesy kromě procesů vytvořených jádrem jsou potomky procesu **init**.

Proces **init** rovněž hraje důležitou roli v řízení procesů. Když proces dokončí svou činnost, zavolá rutinu nazvanou **_exit** a sděluje tak jádru, že je připraven zemřít. Dodá ukončovací kód (celočíselný), který sděluje, proč se proces ukončí. Konvenčně se používá číslo 0 pro normální neboli „úspěšné“ ukončení.

Než smí proces úplně zmizet, jádro vyžaduje, aby jeho smrt potvrdil jeho rodič, což tento provede voláním **wait**. Rodič obdrží kopii ukončovacího kódu (nebo indikaci důvodu zabití, když potomek neskončil dobrovolně), a když chce, může získat i souhrn informací o tom, jak potomek využíval prostředky.

Tento postup funguje dobře, pokud rodiče přežívají své děti a pokud vědí, že mají volat **wait**, aby se mohli mrtví potomci odstranit. Když ale rodič zemře jako první, jádro si uvědomí, že žádné volání **wait** nepřijde, a upraví sirotka tak, že se stane potomkem procesu **init**. Proces **init** adoptuje tyto osiřelé procesy a provádí volání **wait** potřebné pro jejich likvidaci, až zemřou.

* S výjimkou jediného systémového volání to ve skutečnosti jsou knihovní rutiny.

Signály

Signály jsou požadavky na přerušení na úrovni procesů. Celkem je definováno asi třicet různých druhů signálů a používají se různými způsoby:

- Mohou si je posílat procesy mezi sebou jako formu komunikace.
- Může je posílat řadič terminálu a ukončovat, přerušovat nebo odkládat procesy, když uživatel stiskne speciální kombinace kláves, např. <Control+C> nebo <Control+Z>.*
- Může je poslat správce (příkazem **kill**), který chce dosáhnout různých výsledků.
- Může je posílat jádro, když proces spáchá přestupek, jako např. dělení nulou.
- Může je posílat jádro, aby oznámilo procesu „zajímavou“ okolnost, např. smrt potomka nebo dostupnost dat na V/V kanále.

Výpis paměti je obraz paměti procesu, který lze využít při ladění.

Když je přijat nějaký signál, může se stát jedna ze dvou věcí. Když přijímající proces vytvořil rutinu pro zpracování daného signálu, zavolá se tato rutina s informacemi o kontextu, ve kterém byl signál dodán. Jinak kernel podnikne s procesem nějakou implicitní akci. Implicitní akce je pro každý signál jiná. Mnohé signály proces ukončují; některé také generují výpis paměti.

Specifikování rutiny pro zpracování daného signálu v programu se označuje jako „zachycení“ signálu. Když se tato rutina ukončí, pokračuje se v provádění původního programu tam, kde byl signál přijat.

Když se program nechce signály zabývat, může je buď ignorovat nebo zablokovat. Ignorované signály se prostě zahodí a nemají na proces žádný vliv. Blokováný signál je zařazen do fronty pro dodání, avšak jádro nepožaduje, aby ho proces zpracoval před tím, než ho explicitně odblokuje. Zpracovatelská rutina se pro nově odblokováný signál volá pouze jedenkrát, i když během blokování příjmu přišel několikrát.

V tabulce 4.1 je seznam signálů, které by měl znát každý správce. Konvence použití velkých písmen pro názvy signálů pochází z tradice jazyka C. Z podobných důvodů se můžete setkat s názvy signálů s předponou SIG (například SIGHUP).

Existují i další signály, v tabulce 4.1 neuvedené, přičemž většina z nich se používá pro hlášení zřídka se objevujících chyb, jako např. „nepovolené instrukce“. Implicitní zpracování takových signálů je ukončení procesu s výpisem paměti. Zachycení a blokování je zpravidla povoleno, protože některé programy mohou být tak chytré, že se samy pokusí odstranit problém, který chybu způsobil, a teprve pak budou pokračovat v práci.

Signály BUS a SEGV jsou zároveň chybové signály. V tabulce jsme je uvedli proto, že jsou tak běžné: v 99 % případů havárie programu je to jeden z těchto signálů, který je definitivně srazí na kolena. Samy o sobě nemají specifickou diagnostickou hodnotu. Oba indikují pokus o nesprávné použití paměti nebo nesprávný přístup do paměti.**

* Funkce klávesových kombinací mohou být příkazem stty přiřazeny jiným klávesám, ale to se v praxi dělá jen zřídka. V této kapitole je označujeme podle jejich konvenčních přiřazení.

** Přesně řečeno, chyby sběrnice vznikají kvůli porušení ochrany paměti nebo vinou nesmyslné adresy. Chyba stránkování je chybou ochrany paměti stejně jako pokus o zápis do paměti určené pouze ke čtení.

Signály nazvané KILL a STOP nelze zachytit, blokovat ani ignorovat. Signál KILL zničí přijímající proces a signál STOP ho pozastaví až do přijetí signálu CONT. CONT lze zachytit nebo ignorovat, ale nemůžete ho blokovat.

Tabulka 4.1 Signály, které by měl znát každý správce^{a)}

č.	Název	Popis	Implicitní akce	Smí zachytit?	Smí blokovat?	Výpis paměti?
1	HUP	Zavěsit	Ukončit	Ano	Ano	Ne
2	INT	Přerušit	Ukončit	Ano	Ano	Ne
3	QUIT	Ukončit	Ukončit	Ano	Ano	Ano
9	KILL	Zabít	Ukončit	Ne	Ne	Ne
b)	BUS	Chyba sběrnice	Ukončit	Ano	Ano	Ano
11	SEGV	Segmentační chyba	Ukončit	Ano	Ano	Ano
15	TERM	Softwarové ukončení	Ukončit	Ano	Ano	Ne
b)	STOP	Zastavit	Zastavit	Ne	Ne	Ne
b)	TSTP	Zastavit klávesnici	Zastavit	Ano	Ano	Ne
b)	CONT	Pokračovat po zastavení	Ignorovat	Ano	Ne	Ne
b)	WINCH	Změna okna	Ignorovat	Ano	Ano	Ne
b)	USR1	Definuje uživatel	Ukončit	Ano	Ano	Ne
b)	USR2	Definuje uživatel	Ukončit	Ano	Ano	Ne

a) Seznam jmen signálů a čísel je také k dispozici u vestavěného příkazu **kill -l** v **bash**.

b) Liší se podle hardwarové architektury; viz **man 7 signal**.

TSTP je „měkká“ verze signálu STOP, kterou bychom nejvýstižněji popsali jako žádost o zastavení. Tento signál generuje řadič terminálu, když na klávesnici stisknete <Control+Z>. Programy zachycující tento signál obvykle uklidí svůj vlastní stav a pak pošlou samy sobě signál STOP, kterým dokončí operaci zastavení. TSTP může být i ignorován, aby program nemohl být zastaven z klávesnice.

Emulátory terminálů posílají signál WINCH, když se změní jejich konfigurační parametry (např. počet řádků virtuálního terminálu). Tato konvence umožňuje programům pracujícím na emulovaných terminálech (např. textovým editorů) v případě změny automatickou rekonfiguraci. Pokud se vám nedaří správně změnit velikosti oken, přesvědčte se, zda je WINCH správně vygenerován a šířen.*

Signály KILL, INT, TERM, HUP a QUIT možná znějí, jako by znamenaly zhruba totéž, ale ve skutečnosti se používají docela odlišně. Škoda, že pro ně byla vybrána taková vágní terminologie. Zde je dešifrovací příručka:

* Což se snáze řekne, než udělá. Při šíření signálu SIGWINCH mohou hrát roli jak emulátor terminálu (např. xterm), tak i ovladač a příkazy na uživatelské úrovni. Problém většinou spočívá v tom, že se signál zasílá pouze procesu v popředí (tedy nikoli všem procesům běžícím na terminálu) a v chybně zasílané změně velikosti vzdálenému počítači po síti. Protokoly, např. TELNET a SSH, explicitně rozpoznávají změny ve velikosti lokálních terminálů a předávají tuto informaci vzdáleným počítačům. Jednodušší protokoly (např. pro sériové linky) to neumějí.

- KILL nelze blokovat; proces je ukončen na úrovni operačního systému a tento signál nikdy „neobdrží“.
- INT je signál, který pošle řadič terminálu, když stisknete <Control+C>. Je to žádost o ukončení aktuální operace. Jednoduché programy by se měly ukončit (když tento signál zachytí) nebo povolit své zabití, což je implicitní chování, když tento signál není zachycen. Programy, které mají příkazový řádek nebo vstupní režim, by měly zanechat činnosti, uklidit a znovu čekat na vstup uživatele.
- TERM je žádost o úplné ukončení běhu programu. Očekává se, že přijímající proces uklidí svůj stav a ukončí se.
- HUP má dvě rozdílné interpretace. Za prvé, mnohé daemony jej chápou jako žádost o vynulování. Když daemon umí znovu načíst svůj konfigurační soubor a přizpůsobit se změnám, aniž by se restartoval, můžete HUP použít k vyvolání tohoto chování.
- Za druhé, signál HUP občas generuje řadič terminálu, snažící se o uklizení (tj. zabít) procesů připojených ke konkrétnímu terminálu. To se může stát, například když je uzavřeno terminálové sezení nebo když se modemové spojení neúmyslně přeruší (proto název signálu „zavěsit“, angl. *hang up*). Podrobnosti jsou různé podle systému.
- Shelly z rodiny C (**tcs**h apod.) obvykle procesy na pozadí před signálem HUP chrání, aby mohly pokračovat i po odhlášení uživatele. Uživatelé shellů typu Bourne (**ksh**, **bash** atd.) mohou toto chování emulovat příkazem **nohup**.
- QUIT se podobá TERM, ale implicitně vytváří obraz jádra, když není zachycen. Některé programy tento signál zneužívají a interpretují ho, jako by znamenal něco jiného.

Signály USR1 a USR2 nemají žádný určitý význam. Programy je mohou využívat libovolným způsobem. Například webový server Apache interpretuje signál USR1 jako požadavek na korektní restart.

kill a killall: posílání signálů

Jak už jméno napovídá, příkaz **kill** se nejčastěji používá pro ukončování procesů. Příkaz **kill** může posílat jakýkoliv signál, ale implicitně posílá TERM. Příkaz **kill** mohou používat normální uživatelé pro své vlastní procesy nebo superuživatel pro jakýkoliv proces. Syntaxe je následující:

```
kill [-signál] pid
```

kde *signál* je číslo nebo symbolický název signálu, který se má odeslat (jak je uvedeno v tabulce 4.1), a *pid* je identifikační číslo cílového procesu; *pid* s číslem -1 vysílá signál všem procesům kromě procesu **init**.

Příkaz **kill** bez čísla signálu nezaručuje ukončení procesu, protože signál TERM může být zachycen, blokován nebo ignorován. Příkaz

```
kill -KILL pid
```

„zaručuje“ ukončení procesu, protože signál 9, čili KILL, nelze zachytit. Slovo „zaručuje“ jsme dali do uvozovek proto, že procesy jsou někdy v takovém stavu, že ani KILL je neovlivní (obvykle je to způsobeno nějakým degenerovaným zámkem V/V operace, jako například

čekání na disk, který se přestal otáčet). Jediný způsob, jak se zbavit takových otravných procesů, je většinou restart.

Většina shellů má svou vlastní implementaci příkazu **kill**, která se řídí výše popsanou syntaxí. Podle manuálové stránky pro samostatný příkaz **kill** by měl před názvem nebo číslem signálu stát příznak **-s** (např. **kill -s HUP pid**). Ne všechny shelly ale rozumějí této syntaktické verzi, a proto navrhuje držet se formy s **-HUP**, které rozumí i samostatný příkaz **kill**. Pak si nemusíte dělat starost, kterou verzi používáte.

Když neznáte PID procesu, kterému chcete poslat signál, obvykle ho vyhledáte příkazem **ps**, který popisujeme na straně 100. Další alternativou je použít příkaz **killall**, který provede toto vyhledání za vás. Když například chcete, aby daemon **xinetd** obnovil svoji konfiguraci, spusťte:

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

Všimněte si, že pokud zadáte několik procesů, **killall** pošle signály všem.

Normální příkaz **kill** má podobnou funkci, ale zdá se, že není při vyhledávání názvů příkazů tak chytrý jako příkaz **killall**. Držte se příkazu **killall**.

Stavy procesů

Proces nemá automaticky právo na čas základní jednotky (CPU) jen proto, že existuje. Existují čtyři stavy procesu; je potřeba je znát a jsou uvedeny v tabulce 4.2.

Tabulka 4.2 Stavy procesů

Stav	Význam
Spustitelný (Runnable)	Proces lze spustit
Spí (Sleeping)	Proces čeká na nějaký prostředek
Zombie	Proces se snaží zemřít
Zastaven (Stopped)	Proces je zastaven (nesmí běžet)

Spustitelný proces je připraven běžet, kdykoliv je k dispozici čas CPU. Získal všechny potřebné prostředky a jen čeká na čas CPU, který má zpracovat jeho data. Jakmile proces provede systémové volání, které nebude možné okamžitě dokončit (například žádost o přečtení části souboru), Linux ho uspí.

Spící procesy čekají, až nastane určitá událost. Interaktivní shelly a systémové daemony tráví většinu času spánkem, když čekají na vstup z terminálu nebo na síťové spojení. Spící proces je prakticky zablokovaný do doby, než budou uspokojeny jeho požadavky, a proto nedostane žádný čas CPU, pokud neobdrží signál. Zombie jsou procesy, které skončily svůj běh, ale jejich stav ještě nebyl zpracován.

Některé operace mohou způsobit, že se proces dostane do nepřerušitelného spánku. Tento stav je obvykle pomíjivý a není ve výstupu **ps** (ve sloupci **STAT** označovaný jako **D**; viz str. 62). V některých degenerovaných situacích však může přetrvávat. Nejčastější příčinou je problém se souborovým systémem NFS na serveru, který je připojen s volbou „hard“ (napevno). Vzhledem k tomu, že procesy v nepřerušitelném spánku nelze vzbudit ani signálem, nelze je

ani zabít. Dostanete se z toho jen tak, že odstraníte problém, který tento stav způsobil, anebo znovu zavedete systém.

Zombie jsou procesy, které skončily výpočet, avšak nevrátily se ještě do původního stavu. Pokud kolem sebe uvidíte zombie, zjistěte si pomocí **ps** jejich PPID, abyste zjistili, odkud přicházejí.

Zastavené procesy jsou procesy, jejichž běh byl administrativně zakázán. Procesy se zastaví, když přijmou signál STOP nebo TSTP, a znovu pokračují po přijetí signálu CONT. Zastavení se podobá spánku, ale proces se z něj nemůže dostat jinak než vzbuzením, které provede jiný proces (nebo zabitím).

nice a renice: prioritizace procesů při plánování

Ohleduplnost (*niceness*) procesu je číselné doporučení pro jádro, definující, jak by měl být daný proces obslužen ve vztahu k ostatním procesům, soutěžícím o CPU. Jeho zvláštní název je odvozen od skutečnosti, že tato hodnota určuje, jak ohleduplní budete k ostatním uživatelům systému. Vysoká hodnota **nice** znamená nízkou prioritu procesu: bude ohleduplný. Nízká nebo záporná hodnota znamená vysokou prioritu: proces není moc ohleduplný. Rozsah povolených hodnot pro **nice** je -20 až +19.

Nově vytvořené procesy dědí hodnotu ohleduplnosti od svého rodiče, pokud uživatel neprovede speciální úkon. Uživatel daného procesu smí zvýšit hodnotu ohleduplnosti, avšak nesmí ji snížit, ani kdyby tím jen vrátil ohleduplnost procesu na původní hodnotu. Účelem tohoto omezení je zabránit tomu, aby procesy s nízkou prioritou plodily děti s vysokou prioritou. Superuživatel může nastavit hodnotu ohleduplnosti libovolně.

V současné době se čím dále méně využívá ruční nastavení priority procesů. V sedmdesátých a osmdesátých letech minulého století, kdy Unix běžel na pomalých systémech, závisel celkový výkon počítače hlavně na rychlosti CPU. Dnes, kdy má většina stolních počítačů více než dostačující výkon CPU, odvádí plánovač při obsluze procesů zpravidla kvalitní práci. Díky zavedení plánovacích tříd se rozšířila možnost správy i na případy, kdy hrozí dlouhá doba odezvy.

Hlavní brzdou výkonu se u většiny systémů staly V/V operace, neboť rychlost diskových pamětí nedejme za krok se stále rychlejšími CPU. Hodnota ohleduplnosti procesu bohužel nemá žádný vliv na to, jak jádro spravuje svoji paměť a V/V operace; i procesy s vysokou ohleduplností mohou mít neúměrně vysoký podíl na využívání těchto prostředků.

Hodnotu ohleduplnosti pro proces můžete nastavit v okamžiku vytvoření procesu příkazem **nice** a během provádění procesu ji lze upravit příkazem **renice**. Příkaz **nice** přijímá příkazový řádek jako argument a **renice** očekává PID nebo jméno uživatele. Příkaz **renice** vyžaduje zadání priority v absolutním tvaru, kdežto příkaz **nice** očekává *inkrement* priority, který pak sečte nebo odečte od aktuální priority shellu, což je trochu zmatené.

Několik příkladů:

```
$ nice -n 5 ~/bin/longtask // Sníží prioritu (zvýší ohleduplnost) o 5
$ sudo renice -5 8829      // Nastaví ohleduplnost na -5
$ sudo renice 5 -u zeman   // Nastaví hodnotu ohleduplnosti zemanových procesů na 5
```

Aby to nebylo tak jednoduché, C shell a některé další příkazové interprety mají vestavěnou svou vlastní verzi příkazu **nice**, ale příkazový interpret **bash** nikoliv. Když nenapíšete plnou

cestu k příkazu **nice**, spustí se verze shellu místo verze operačního systému. To může být matoucí, protože příkaz **nice** z příkazového interpretu používá jinou syntaxi než samostatný příkaz **nice**: příkazový interpret chce, aby se inkrement vyjadřoval jako *+inkr* nebo *-inkr*, kdežto samostatný příkaz požaduje příznak **-n** následovaný inkrementem priority.*

Proces s nejčastěji upravovanou ohleduplností je v současnosti daemon **xntpd** pro synchronizaci hodin. Vzhledem k tomu, že rychlost CPU je pro jeho úkol kriticky důležitá, běží obvykle s hodnotou ohleduplnosti 12 pod implicitní hodnotou (tedy s vyšší než normální prioritou).

Když některý proces zešílí a zvedne hodnotu zátěže systému na 65, možná budete muset použít příkaz **nice** a spustit příkazový interpret s vysokou prioritou, než budete moci spustit příkazy potřebné k prozkoumání problému. Jinak vaše příkazy nemusí dostat šanci se spustit.

ps: sledování procesů

Příkaz **ps** je hlavním nástrojem správce systému pro sledování procesů. Můžete ho použít pro zobrazení PID, UID, priority a řídicího terminálu procesů. Také poskytuje informace o tom, kolik proces spotřebuje paměti, kolik času CPU, a jaký je jeho aktuální stav (běží, zastaven, spí atd.). Zombie se v seznamu **ps** zobrazují jako `<defunct>`.

Chování **ps** se v různých variantách Unixu liší a mnohé implementace se za posledních několik let staly dost složitými. Linux se snaží přizpůsobit lidem, kteří si zvykli na příkaz **ps** z jiných verzí, a poskytuje proto trisexuální, hermafroditickou verzi, která rozumí sadám příznaků z jiných implementací a která používá proměnnou prostředí definující, jakou osobnost má **ps** napodobovat.

Nelekejte se této složitosti: je to hlavně pro programátory jádra, ne pro správce systému. Ačkoliv budete příkaz **ps** používat často, budete potřebovat jen několik konkrétních zaklíadel.

Obecný přehled všech procesů běžících v systému získáte příkazem **ps aux**. Zde je příklad (odstranili jsme sloupec START, aby se nám tento příklad vešel na stránku, a vybrali jsme jen některé výstupní řádky):

\$ **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	[pdflush]
...									
root	196	0	0	0	0	?	S	0:00	[kjournald]
root	1050	0	0.1	2652	448	?	S<s	0:00	udev
root	1472	0	0.3	3048	1008	?	S<s	0:00	/sbin/dhclient -1
root	1646	0	0.3	3012	1012	?	S<s	0:00	/sbin/dhclient -1
root	1733	0	0	0	0	?	S	0:00	[kjournald]

* Ve skutečnosti je to ještě horší: samostatný příkaz **nice** interpretuje **nice -5** jako kladný inkrement 5, kdežto vestavěný příkaz **nice** chápe stejný formát jako záporný inkrement 5.

```

root    2124  0      0.3    3004    1008  ?      Ss      0:00   /sbin/dhclient -1
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: accepi
...

```

Jména příkazů v hranatých závorkách ve skutečnosti nejsou příkazy, nýbrž vlákna jádra plánovaná jako procesy. Vysvětlení významu jednotlivých polí naleznete v tabulce 4.3.

Další užitečnou sadou argumentů je **lax**, poskytující techničtější informace. Také je rychlejší, protože nemusí překládat každé UID na jméno uživatele – účinnost může být důležitá, pokud systém pracuje šnečí rychlostí kvůli nějakému jinému procesu.

Dále následuje zkrácený příklad **ps lax**, s poli jako např. rodičovské PID (PPID), hodnota ohleduplnosti (NI) a prostředek, na který proces čeká (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]
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
...

```

Tabulka 4.3 Vysvětlení výstupu ps aux

Pole	Obsah
USER	Uživatelské jméno majitele procesu
PID	Identifikace procesu
%CPU	Procento času CPU použité tímto procesem
%MEM	Procento skutečné paměti použité tímto procesem
VSZ	Virtuální velikost procesu
RSS	Rezidentní nastavená velikost (počet stránek paměti)
TTY	Identifikace řídicího terminálu
STAT	Aktuální stav procesu:

Pole	Obsah
	R = Běžící D = nepřerušitelný spánek
	S = spí (méně než 20 s) T = trasován nebo zastaven
	Z = zombie
	Další příznaky:
	w = proces je v odkládacím prostoru
	< = proces má vyšší než normální prioritu
	N = proces má nižší než normální prioritu
	L = některé stránky jsou uzamknuté v jádru
	s = hlavní proces sezení
START	Čas, kdy byl proces spuštěn
TIME	Čas CPU spotřebovaný procesem
COMMAND	Název příkazu a jeho argumenty ^{a)}

a) Programy mohou tyto informace upravovat, takže to nemusí být přesný obraz skutečného příkazového řádku.

top: ještě lepší sledování procesů

Příkaz **ps** nabízí pouze jednorázový snímek vašeho systému, a proto je někdy obtížné získat takto celkový přehled o tom, co se děje. Příkaz **top** poskytuje pravidelně aktualizovaný souhrn aktivních procesů a údaje o tom, jak využívají prostředky.

Například:

```
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: 256044k total, 254980k used, 1064k free, 15944k buffers
Swap: 524280k total, 0k used, 524280k free, 153192k cached
```

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

...

Zobrazení se implicitně aktualizuje každých 10 vteřin. Neaktivnější procesy jsou nahoře. Příkaz **top** také přijímá vstup z klávesnice a umožňuje posílat signály a používat na procesy **renice**, takže pak můžete sledovat, jak vaše akce ovlivňují celkový stav počítače.

Superuživatel může spouštět příkaz **top** s argumentem **q** a nažhavit ho tak na nejvyšší možnou prioritu. To může být velice užitečné, když se snažíte najít proces, který srazil systém na kolena.

Souborový systém /proc

Linuxové verze příkazů **ps** a **top** čtou stavové informace z adresáře **/proc**, což je souborový pseudosystém, v němž si jádro ukládá různé zajímavé informace o stavu systému. Navzdory jménu **/proc** (a typu výchozího souborového systému „proc“) nejsou informace omezeny pouze na informace o procesech – jsou zde uloženy nejen stavové informace, ale i všechny statistiky vedené jádrem. Některé parametry můžete dokonce zápisem do příslušného souboru **/proc** změnit – některé příklady viz str. 874.

I když některé údaje získáte snáze pomocí frontendových příkazů, např. **vmstat** nebo **ps**, jiné, méně oblíbené informace je nutno číst přímo z **/proc**. Určitě se vyplatí, když se v tomto adresáři trochu pošťouráte a seznámíte se s jeho obsahem. Užitečné tipy a triky naleznete také v **man proc**.

Vzhledem k tomu, že jádro vytváří obsah souborů v **/proc** za chodu (když se k nim dostane), zadáte-li příkaz **ls -l**, většinou se zdá, že jsou prázdné. Chcete-li zjistit, co soubory skutečně obsahují, musíte zadat příkazy **cat** nebo **more**. Musíte však být opatrní – určité soubory obsahují binární data, s nimiž si některé terminálové emulátory při přímém prohlížení možná neporadí.

Údaje o procesech jsou rozděleny do podadresářů označených identifikačními čísly procesů (PID). Například **/proc/1** je vždy adresář, který obsahuje údaje o **init**. V tabulce 4.4 jsou uvedeny některé užitečné soubory, které se vztahují k jednotlivým procesům.

Tabulka 4.4 Informační soubory o procesech (v číslováních podadresářích)

Soubor	Obsah
cmd	Příkaz nebo program, který spustil daný proces
cmdline ^{a)}	Úplný příkazový řádek procesu (oddělovačem je znak NULL)
cwd	Symbolický odkaz do aktuálního adresáře procesu
environ	Vnější proměnné procesu (oddělovačem je znak NULL)
exe	Symbolický odkaz do souboru, s nímž proces pracuje
fd	Podadresář, který obsahuje odkazy do všech deskriptorů otevřených souborů
maps	Údaje o mapované paměti (sdílené segmenty, knihovny atd.)
root	Symbolický odkaz do kořenového adresáře procesu (nastaveného příkazem chroot)
stat	Obecné stavové informace procesu (nejlépe je zobrazí příkaz ps)
statm	Údaje o využívání paměti

^{a)} Je-li proces odložen mimo paměť, soubor může být nedostupný.

Komponenty v souborech **cmdline** a **environ** jsou odděleny znaky NULL – nikoli tedy znaky „newline“. Když je přefiltrujete pomocí **tr „\000“ „\n“**, budou čitelnější.

Podadresář **fd** obsahuje odkazy na otevřené soubory. Deskriptory souborů spojené s rourami nebo sokety neobsahují jména souborů. Jádro místo toho poskytuje odkazy na jejich obecný popis.

Soubor **maps** se může hodit při určování, ke kterým knihovnám je program sestaven nebo na kterých je závislý.

strace: sledovací signály a systémová volání

V klasickém Unixu někdy bývá dost obtížné udělat si obrázek o tom, co vlastně program právě dělá. Můžete si vytvářet kvalifikované odhady na základě nepřímých dat ze souborových systémů nebo pomocí některých nástrojů, např. **ps**. V Linuxu naopak můžete procesy přímo sledovat pomocí příkazu **strace**, který ukáže všechna systémová volání vydaná procesem a všechny signály, které obdrží. Příkaz můžete dokonce k běžícímu procesu připojit, nechat ho chvíli čenichat, a pak jej od procesu odpojit, aniž byste proces jakkoli rušili v činnosti*.

I když se systémová volání vyskytují na relativně nízké úrovni abstrakce, většinou se z výstupu příkazu **strace** můžete o činnosti procesu ledacos dozvědět. Například následující výpis byl pořízen při současném běhu **strace** a aktivní kopie **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
...
```

Nejenže **strace** ukáže názvy všech systémových volání vydaných procesem, nýbrž také dekoduje argumenty a ukáže výsledný kód vrácený jádrem.

V tomto příkladu začne **top** tím, že zjistí aktuální denní čas. Pak otevře a zjistí stav adresáře **/proc** a přečte jeho obsah, odkud obdrží seznam běžících procesů. Pak pokračuje stavem adresáře procesu **init** a poté otevře **/proc/1/stat**, kde si přečte stavové informace procesu **init**.

Splašené procesy

Další informace o splašených procesech naleznete na str. 818.

„Splašené“ procesy se vyskytují ve dvou typech: uživatelské procesy využívající příliš mnoho systémových prostředků, jako např. čas CPU nebo diskový prostor, a systémové procesy, které najednou zešílí a chovají se divoce. První typ nemusí být nutně porouchaný; může to zkrátka být žrout prostředků. Systémové procesy se ale mají chovat vždy rozumně.

* Obvykle je to možné. Někdy ovšem může **strace** přerušit systémová volání. Pak musí být sledovaný proces připraven k jejich obnově. To je součástí unixové softwarové hygieny, ale není tomu tak vždy.

Procesy využívající příliš mnoho času CPU můžete najít ve výstupu příkazu **ps** a **top**. Když je jasné, že proces spotřebovává více času CPU, než by se dalo rozumně očekávat, prozkoumejte jej. Prvním krokem je kontaktovat majitele procesu a zeptat se ho, co se děje. Když majitele nemůžete najít, budete se muset porozhlédnout sami. I když byste se zpravidla neměli dívat do domácích adresářů uživatelů, je to přijatelný postup v případě, že se snažíte najít zdrojový kód spášeného procesu a zjistit, co dělá.

Jsou dva důvody, proč byste měli zjistit, co se proces snaží udělat dříve, než si s ním začnete pohrávat. Za prvé může být proces legitimní a důležitý pro daného uživatele. Je nerozumné jen tak náhodně zabíjet procesy jen proto, že spotřebovávají velké množství času CPU. Za druhé, proces může být zlovolný nebo ničivý. V takovém případě potřebujete vědět, co proces dělal (např. prolamoval hesla), abyste mohli škodu opravit.

Když nelze zjistit důvod existence procesu, pozastavte ho signálem STOP a pošlete jeho majiteli e-mail, ve kterém vysvětlíte, co se stalo. Proces může být později restartován signálem CONT. Uvědomte si, že některé procesy lze dlouhým spánkem zabít, takže tento postup není vždy neškodný. Proces se například může probudit a zjistit, že některá jeho síťová spojení byla přerušena.

Když proces spotřebovává příliš mnoho času CPU, ale zdá se, že dělá něco rozumného a pracuje správně, měli byste ho upravit pomocí příkazu **renice** na vyšší hodnotu ohleduplnosti (menší prioritu) a požádat jeho majitele, aby si příště nastavil ohleduplnost sám.

Procesy, které využívají nadměrné množství paměti vzhledem k fyzické velikosti RAM v systému, mohou vážně ohrozit výkon systému. Velikost paměti využívané procesy můžeme zjišťovat příkazem **top**. Sloupec VIRT ukazuje celkovou velikost virtuální paměti přidělené jednotlivým procesům, zatímco ve sloupci RES je část té paměti, která je aktuálně namapována k určitým stránkám paměti („rezidentní množina“).

Obě tato čísla mohou obsahovat i sdílené prostředky, např. knihovny, což může být zavádějící. Přímější zjišťování spotřeby paměti příslušné k danému procesu naleznete ve sloupci DATA, který se nevypisuje implicitně. Má-li se tento sloupec vypisovat, musíme v průběhu činnosti **top** stisknout klávesu **f** a ze seznamu vybrat DATA. Hodnota DATA udává velikost paměti potřebnou pro všechna data a zásobníky jednotlivých procesů, takže je poměrně přesná (modulo sdílené paměťové segmenty). Kromě její absolutní velikosti také sledujte, zda se časem zvětšuje.

Spášené procesy produkující výstup mohou zaplnit celý souborový systém a způsobit tak množství problémů. Když se souborový systém zaplní, bude se na konzole vypisovat velké množství zpráv a pokusy o zápis do souborového systému budou vyvolávat chybové zprávy.

V takové situaci byste nejdřív měli zastavit proces, který zaplňuje disk. Když na disku udržujete rozumné množství prostoru k dýchání, můžete si být v případě jeho náhlého zaplnění jisti, že se děje něco nekalého. Neexistuje příkaz analogický k **ps**, sdělující, kdo zabírá diskový prostor nejrychleji, avšak existuje několik nástrojů, které mohou identifikovat momentálně otevřené soubory a procesy, které je používají. Další informace najdete v popisu příkazů **fuser** a **lsof**, který začíná na straně 111.

Možná budete chtít zastavit všechny podezřelé procesy, dokud nenajdete proces, který je příčinou problémů, nezapomeňte ale na konci znovu spustit nevinné. Až najdete provinilý proces, smažte soubory, které vytvářel.

Starý a dobře známý kanadský žertík je vytvořit v shellu nekonečnou smyčku, která dělá toto:

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

Občas se stává, že tento program běží z nechráněného uživatelského účtu nebo z terminálu, který byl přihlášeným uživatelem ponechán bez dozoru. Nespotřebovává sice moc diskového prostoru, ale zaplňuje tabulku i-uzlů v souborovém systému a brání ostatním uživatelům vytvářet nové soubory. Nemůžete s tím dělat nic jiného, než že tento nepořádek uklidíte a varujete uživatele, aby chránili své účty. Adresářový strom vytvořený tímto malým klenotem je obvykle tak velký, že ho příkaz **rm -r** nezvládne, a proto možná budete muset napsat skript, který sestoupí na konec adresářového stromu a pak odstraňuje jeden adresář za druhým až nahoru.

Když k tomuto problému dojde v adresáři **/tmp** a máte adresář **/tmp** jako oddělený souborový systém, můžete místo mazání jednotlivých souborů znovu inicializovat adresář **/tmp** příkazem **mkfs**. Více informací o řízení souborových systémů najdete v kapitole 7.

Doporučená literatura

BOVET, DANIEL P. AND MARCO CESATI. *Understanding the Linux Kernel (3rd Edition)*. Sebastopol, CA: O'Reilly Media, 2006.

Cvičení

- E4.1 Vysvětlíte vztah mezi UID souboru a reálným UID a provozním UID procesu. Jaký je účel provozního UID procesu kromě řízení přístupu k souboru?
- E4.2 Dejme tomu, že uživatel ve vaší organizaci spustil dlouhodobý proces, který spotřebovává významný podíl prostředků.
 - a) Jak byste poznali proces, který spotřebovává prostředky?
 - b) Předpokládejme, že provinilý proces může být legitimní a nezaslouží si zemřít. Ukažte příkazy, které byste použili pro jeho uložení k ledu (přechodné zastavení kvůli prošetření).
 - c) Později objevíte, že tento proces patří vašemu šéfovi a musí pokračovat v běhu. Ukažte příkazy, které byste použili pro obnovení běhu tohoto procesu.
 - d) Předpokládejme naopak, že proces je třeba zabít. Jaký signál byste poslali a proč? Co když musíte zaručit, že proces zemřel?
- E4.3 Najděte proces, který během své činnosti zabírá stále více paměti (váš vlastní program nebo program **netscape**, když takový program nemáte po ruce). Sledujte paměť programu za běhu pomocí příkazů **ps** nebo **top**.
- ★ E4.4 Napište jednoduchý skript v Perlu, který zpracovává výstup **ps** a vytváří součet VSZ a RSS daného procesu. Jaký mají tato čísla vztah ke skutečné velikosti fyzické paměti a k prostoru pro odkládání procesů?