Procesy, sygnały i usługi systemu Unix

Paweł Józiak

Wydział Matematyki i Informatyki Politechniki Warszawskiej

29 XI 2021

Plan zajęć

Plan zajęć

Procesy a programy

- `Proces` to środowisko wykonania, składające się z instrukcji, segmentów danych użytkownika i systemowych, a także zasobów uzyskiwanych w czasie cyklu trwania.
- `Program` to wykonywalny (uprawnienie `x`) plik regularny, składający się z instrukcji i danych używanych do uruchomienia procesu.

Tworzenie procesów – *fork*

- Jeśli program używa zapytania systemowego `fork()` , jądro tworzy nowy proces (*dziecko*) w imieniu procesu tworzącego (*rodzica*).
- Instrukcje, dane użytkowinka oraz systemowe, procesu-rodzica są kopiowane do procesu-dziecka; zmianie ulegają jedynie meta-parametry środowiska, więc efektywnie proces-dziecko jest klonem procesu-rodzica.
- `fork()` zwraca ID procesu-dziecka w procesie-rodzicu, zaś 0 w procesie-dziecku (albo -1 w wypadku błędu). Bazując na tym, osobne bloki insrukcji mogą być wywołane w procesie-rodzicu oraz procesie-dziecku.

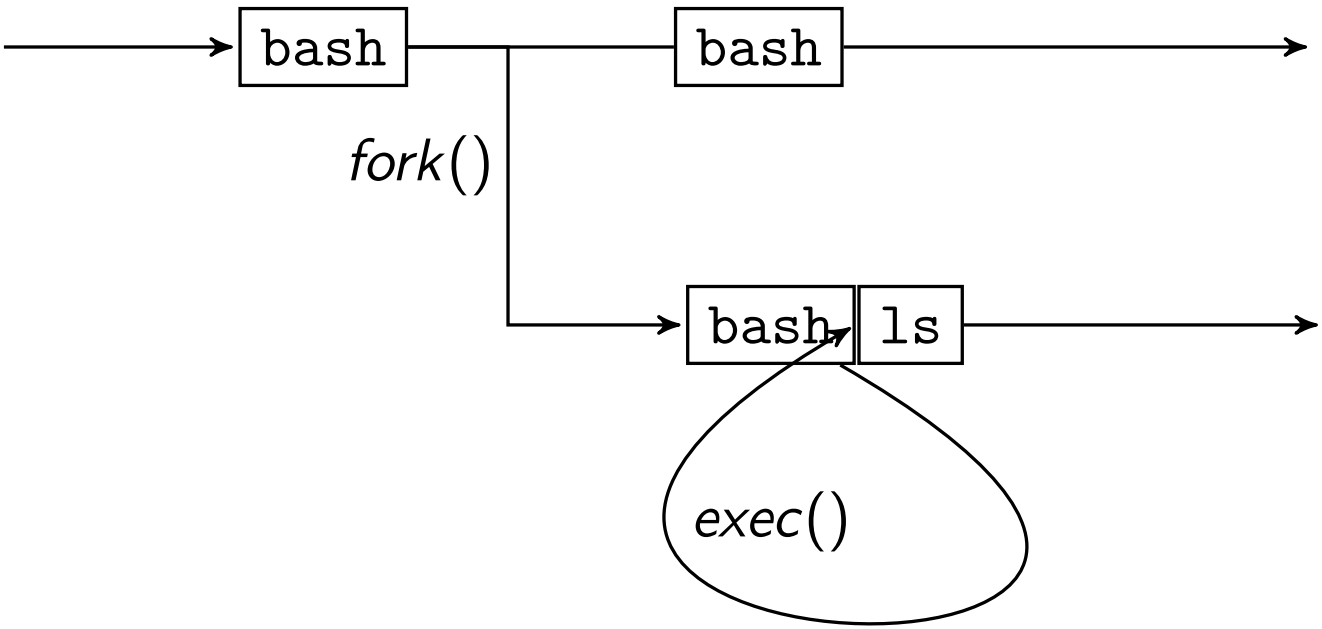
```
switch (id = fork()) {
  case 0: child_work(); exit(EXIT_STATUS);
  case -1: error_occured(); exit(EXIT_FAILURE);
}
parent_work(id);
```

Tworzenie procesów II – *exec*

- Zapytanie systemowe `exec()` reinstancjalizuje proces ze wskazanym plikiem programu.
- W wyniku wywołania `exec()` proces pozostaje ten sam, zaś program się zmienia.
- `fork()` oraz `exec()` zwykle są wykonywane w parze, by wystartować nowy program.

Tworzenie procesów – przykład

- Komenda `ls` wywoływana z poziomu `bash`a



init – proces założycielski

- `init` to pierwszy proces, zakładany przez system zaraz po uruchomieniu jądra.
- Jego *Process ID* (`PID`) wynosi zawsze `1` .
- Jeśli proces-rodzic zakończy działanie, a jego procesy-dzieci wciąż funkcjonują, zostają one adoptowane przez proces `init` .
- Uwaga: w Linuxach opartych o architekturę `systemd` proces `init` jest przemianowany na `systemd` .

Zombie oraz sieroty

- Każdy proces może poznać swój PID, oraz o PID swojego procesu-rodzica, wywołując zapytania systemowe (odpowiednio, `getpid()` oraz `getppid()`), zaś jedynym sposobem na zdobycie PID swojego procesu-dziecka jest zapamiętanie wartości zwróconej przez wywołanie `fork()`.
- Gdy proces kończy działanie, wysyła sygnał `SIGCHLD` (por. nast. sekcja) do swojego procesu-rodzica. Proces-rodzic powinien wówczas wywołać zapytanie `wait()` (albo `waitpid()`) by uzyskać *status wyjścia* procesu-dziecka. Jeśli tego nie zrobi, proces-dziecko wciąż widnieje w tabeli aktywnych procesów, choć nic nie robi. Taki proces nazywamy *zombie*.
- Rolą procesu `init` jest niezwłoczne wywołanie `wait()` na wszystkich zaadoptowanych zombie.

Grupy procesów

- Procesy wystartowane za pomocą pojedynczej komendy (np. procesy w *pipeline*'ie) są zorganizowane w tzw. *grupy procesów* (*job*).
- Co do zasady, procesy mogą migrować pomiędzy grupami procesów, a także zakładać nowe (używając `setpgid()` oraz `setpgrp()` – w zależności od systemu).
- Grupy procesów są używane do kontroli dystrybucji sygnałów. Na przykład: naciśnięcie <CTRL> + <C> wyśle sygnał zakończenia do wszystkich procesów działających w tle (por. nast. slajd).

Sesje

- Jeden bądź więcej grupy procesów tworzy `session` (*sesję*).
- Zwyczajowo, Unix tworzy jedną sesję per zalogowanie (wyłączając zalogowanie via GUI); Powłoka udostępniona po zalogowaniu staje się tzw. *liderem sesji*.
- Procesy nie mogą migrować do istniejących sesji, mogę jednak zakładać nowe – `setsid()` .
- Sesji może być przyposany *terminal kontrolujący* (otwarty przed lidera, który staje się *procesem kontrolującym*). Udostępnienie dostępu terminalowego do grup procesów (przenoszenie ich do *tła*/*czoła* – *background*/*foreground*) nazywamy *job control*.
- Jeśli nie powzięto adekwatnych kroków zapobiegawczych, zamknięcie procesu kontrolującego bądź zawieszenie terminala kontrolującego zamyka wszystkie procesy w grupie pracującej na czele.

Podsumowanie atrybutów procesów

- Atrybuty procesów (przechowywane jako pseudopliki w przestrzeni adresowej `/proc`) zawierają:
  - `PID – ID procesu` ;
  - `PPID – ID procesu-rodzica` ;
  - `real user ID` oraz `real group ID` – właściciel/grupa procesu;
  - `effective user ID` oraz `effective group ID` – te same, co wyżej, chyba, że użyto *setuid*/*setgid*;
  - `PGID` – ID grupy procesu;
  - `session leader ID` ;
  - `terminal` .

Drzewo procesów – *pstree*

- Komenda `pstree` reprezentuje działające procesy jako drzewo (krawędź od procesu-rodzica do procesu-dziecka, korzeniem proces `init`).
- Wraz z opcję `-p` wypisywane są również PID każdego procesu (w nawiasie).
- Wszystkie osierocone procesy są wizualizowane jako procesy-dzieci procesu `init`.
- Uwaga – liczby w nawiasach klamrowych reprezentują *wątki*, nie *procesy* – identyfikowane przez *Thread ID*.

Informacje o działających procesach – *ps*

- `ps` wyświetla statystyki o działających procesach. Istnieje wiele opcji agregacji informacji do wyświetlania; by uzyskać pełną informację często używa się `ps -ef` (pełny listing dla każdego procesu).
- Systemy BSD używają innych opcji i innego syntaksu (brak myślnika przez nazwą opcji). Pełny listing w systemach BSD uzyskiwany jest przez `ps aux` .

Jaki jest PPID tej instancji basha?

```
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
pawel      50701    1918  0 lis26 pts/4    00:00:00 /bin/bash
pawel      56393   50701  0 18:17 pts/4    00:00:00 ps -f
```

Informacje o działających procesach – *pgrep, pidof*

- Często używany wzorzec `ps -e | grep` *`nazwa_programu`* jest uproszczany przez komendę `pgrep` .
- Zbliżoną (lecz szybszą) jest alternatywą jest `pidof` .
- Oba programy biorą jako argument nazwę programu, i drukują na *stdout* PID procesu (-ów) o takiej nazwie programu.
- Typowe użycie sprowadza się do natychmiastowego wywołania w argumencie komendy, która opiera się o PID.

Komenda *kill* (por. następna sekcja) opiera się o PID

```
$ kill ID_procesu
$ kill $(pidof nazwa_programu)
```

Informacje o działających procesach – *top* i inne

- `top` wyświetla dynamiczne informacje o procesach, w tym zużycie CPU i pamięci, w czasie rzeczywistym.
- Informacje można posortować (domyślnie: od największego zużycia CPU) i wizualizować na różne sposoby.
- `top` sam z siebie zużywa znaczący ułamek CPU.
- `htop` jest nowszą, bardziej rozbudowaną wersją klienta `top` .
- Kilka innych, zbliżonych, programów, wzorowanych na `top` , jest dostępnych:
  - `iotop` – sumaryzujące (dyskowe) użycie I/O;
  - `iftop` – sumaryzujące połączenia sieciowe;
  - `ntop` – sumaryzujące stan połączeń sieciowych.
- Powyższe programy są nieustandaryzowane, zależne od platformy i zwykle nie należące do podstawowego pakietu aplikacji.

System plików */proc* – drugie spojrzenie

- Wszystkie informacje o procesach są dostępne z poziomu pseudo-systemu plików *procfs* – w formie plików pod ścieżką `/proc/<PID>` . Zestaw plików jest zależny od systemu, zajrzyj w `man 5 proc` swojego systemu dla ujednoznacznienia.
  - `/proc/<PID>/cmdline` – pełna treść komendy, która wystartowała proces,
  - `/proc/<PID>/cwd` – symlink do katalogu roboczego komendy,
  - `/proc/<PID>/environ` – zmienne środowiskowe procesu,
  - `/proc/<PID>/exe` – symlink do pliku wykonywalnego,
  - `/proc/<PID>/fd` oraz `/proc/<PID>/fdinfo` – deskryptory plików, pozycje i flagi,
  - `/proc/<PID>/status` – statystyki procesu.
- Rolą programów typu `ps`, `pstree`, `pgrep`, `pidof`, `top`, `htop` jest *jedynie* sprawne sparsowanie zawartości tych katalogów.

Śledzenie procesów

- Systemy Uniksowych dostarczają narzędzi do śledzenia i wyświetlania informacji o zapytaniach systemowych wykonywanych przez proces (Linux: `strace` , BSD: `ktrace` , AIX i Solaris: `truss`)
- Celem śledzenia procesu może być:
  - odnalezienie źródła błędnego funkcjonowania programu,
  - określenie źródła i kolejności konfiguracji czytanej przez proces,
  - inżynieria wstecznej (odkrycie logiki działania programu).

Śledzenie procesu – typowe użycie

```
$ strace [-f|-ff] [-o plik] [-e cel] [komenda|-p <PID>]
```

- Użyte opcje to:
  - `-f` oraz `-ff` – czy śledzić również procesy-dzieci,
  - `-o plik` – zapis logów do pliku (domyślnie: *stdout*). Z opcją `-ff` , osobny plik dla każdego procesu-dziecka, z opcją `-f` : jeden plik.
  - `-e` określa szczegóły śledzenia konkretnych wydarzeń,
  - `-p` podłącza do już działającego procesu (zadanego przez `PID`). Tylko `root` może śledzić cudze procesy.

Plan zajęć

Sygnały

- Najbardziej podstawowy system komunikacji między procesami. Żadne dane nie są przesyłane, jedynie typ sygnału (1-28).
- Notyfikują odnośnie wydarzeń systemowych (np. dzielenie przez zero generuje sygnał `SIGFPE`) bądź mogą być generowane przez proces.
- Jądro oraz procesy `roota` mogą wysyłać sygnały do każdego procesu.
- Procesy innych użytkowników mogą wysyłać sygnały jedynie do procesów tego samego użytkownika.
- Wśród (typowo) 28 sygnałów systemów Unix, jedynie 2: `SIGUSR1` oraz `SIGUSR2` , są konfigurowalne przez użytkownika.
- POSIX wprowadza dodatkowye *Real Time Signal*: od `SIGRTMIN` do `SIGRTMAX` , które są omawiane na *Systemach Operacyjnych*.

Obsługa sygnałów

- *Signal handler* to mała funkcja, której celem jest obsługa sytuacji związanej z konkretnym sygnałem.
- Jeśli dla danego sygnału żaden *signal handler* nie jest określony, wykonywana jest domyślna akcja (zazwyczaj: zakończenie procesu).
- Wszystkie sygnały z wyjątkiem `SIGKILL` oraz `SIGSTOP` można obsłużyć za pomocą *signal handler*a, zablokować bądź zignorować.
- Blokowanie sygnału wiąże się ze wstrzymaniem procesu. Sygnał taki jest odroczony do czasu odblokowania procesu.
- Sygnały ignorowane nie są dostarczane do procesu.

Rodzaje sygnałów

- Sygnały błędów:
  - `SIGFPE` – błąd obliczeń arytmetycznych,
  - `SIGILL` – nieprawidłowa instrukcja,
  - `SIGPIPE` – próba zapisu do *pipe* bez czytelników,
  - `SIGSEGV` – błąd segmentacji pamięci.
- Sygnały zamykania procesu:
  - `SIGTERM` – domyślny sygnał, prośba o bezpieczne zakończenie procesu; wysyłany do wszystkich procesów przy zamknięciu systemu;
  - `SIGKILL` – sygnał bezwarunkowego, natychmiastowego zamknięcia procesu; może skutkować utratą danych,
  - `SIGINT` – generowany przez <CTRL>+ <C> w terminalu,
  - `SIGQUIT` – generowany przez <CTRL>+ </> w terminalu; ten sam efekt, co `SIGINT` + zapis stanu rdzenia,
  - `SIGHUP` – wysyłany w razie zawieszenia terminala kontrolującego; typowo używany do restartowania *daemon*ów

Wysyłanie sygnałów

- Część sygnałów ma charakter systemowy, są emitowane przez `kernel`.
- Każdy sygnał można wygenerować sztucznie za pomocą zapytania systemowego `kill()` (zob. `man 2 kill`)
- Istnieje komenda `kill` (zob. `man 1 kill`), która jest interfejsem dla zapytania systemowego `kill()` .

Syntaks *kill*

```
$ kill [-SIGTYPE] <PID>
```

- Każdy sygnał (nawet `SIGCHLD` czy `SIGFPE`) mogą być wygenerowane przez `kill` .
- Sygnały mogą być określone nazwą (`-SIGKILL`) bądź odpowiadającym numerem (`-9`).
- `kill` wymaga określenia procesu za pomocą jego `PID`. Określamy go za pomocą innych narzędzi (zob. `ps`, `pidof`, `pgrep`). Ponadto, w niektórych systemach dostępne są komendy `killall` albo `pkill` .

Sekwencje kontrolne terminala

- Terminal definiuje szereg sekwencji kontrolnych, które pozwalają na emisję sygnału do grupy procesów pracujących na czele.
- Niektóre programy używają tych samych sekwencji kontrolnych do innych celów (np. `nano`). Odwzorowanie sekwencji kontrolnch terminala na nowe sekwencje jest indywidualne.
- Najpopularniejsze sekwencje kontrolne to:
  - <CTRL>+ <C> – emituje `SIGINT` , kończenie procesu.
  - <CTRL>+ </> – emituje `SIGQUIT` , kończenie procesu z zapisem rdzenia.
  - <CTRL>+ <Z> – emituje `SIGSTOP` , zastopowanie procesu (może być wznowiony, nie następuje zakończenie ani zwolnienie zasobów),
  - <CTRL>+ <D> – generuje znak końca pliku (`EOF`), nie emituje żadnego sygnału.

Zakończenie a zastopowanie procesu; procesy tła

- Zakończenie procesu wiąże się z wywołaniem `exit()` – zakończeniem wykonania, uwolenieniem zasobów i zmianą stanu procesu na *dead*.
- Zastopowanie procesu oznacza jego *zamrożenie*. Wykonanie procesu jest zawieszone, lecz pozostaje w stanie *alive*, zajmuje przydzielone zasoby. Zastopowane procesy można wznowić.
- `jobs` wyświetla stan grup procesów w bieżącej sesji.
- Komenda zakończona znakiem `&` będzie wykonana w tle (np. `ls &`).
- Alternatywnie, proces działający na czele może być zastopowany za pomocą <CTRL>+ <Z>, oraz przywrócony do działania za pomocą komendy `bg` .
- Komenda `fg` przywraca procesy z tła na czoło.
- Jeśli istnieje więcej niż 1 grupa procesów, `fg` oraz `bg` wymagają argumentu – numeru grupy procesów. Tę można uzyskać wywołując `jobs` (w razie braku – ostatnio modyfikowana grupa procesów będzie używa jako domyślna dla `fg` / `bg`).

Plan zajęć

Daemony

- *Disk And Execution MONitor*s, popularnie *serwis*y albo *server*y: ssh, name, web, mail, bazy danych itd.
- Sufiks "d": `sshd`, `named`, `httpd`, `mysqld` itd. symbolizuje daemon.
- Proces tła (bez terminalu kontrolującego).
- Przez większość czasu bezczynnie czekający na interakcje z użytkownikiem/systemem (np. ACPI).
- Komunikaty kontrolne pisze do swojego pliku logów bądź na `syslog` .
- Powinno startować się tylko jedną instancję naraz.
- Zwykle (adoptowane) procesy-dzieci `init` .
- Odpalane przez konta systemowe.
- Ze względów bezpieczeństwa, daemony wymagające ograniczonego dostępu do systemu plików, są uruchamiane w tzw. *chroot jail*.
- Mogą opierać się o dalsze daemony.
- Z powyższych powodów nie powinno się ich odpalać ręcznie. Istnieją dedykowane skrypty kontrolujące do ich uruchamiania.

Skrypty kontrolujące daemony

- Pisane w powłoce (zwykle `sh`) albo jej rozszerzeniu.
- Bezpośrednie, jawne, czytelne przez człowieka i konfigurowalne.
- Jego struktura reprezentuje ideologię systemu (krótkie i proste dla systemów KISS, rozbudowane dla kompleksowych systemów).
- Jeśli jakieś zmienne powinny być skonfigurowane, nie określa się ich w skrypcie kontrolującym, lecz w towarzyszącym pliku konfiguracyjnym.
- Hierarchia plików/katalogów powiązanych:
  - `/usr/sbin/` – programy daemonów,
  - `/etc/` – pliki/katalogi konfigurujące daemony,
  - `/etc/init.d/` albo `/etc/rc.d/` – skrypty kontrolujące,
  - `/etc/conf.d/` albo `/etc/sysconfig` –pliki konfiguracyjne dla skryptów kontrolujących,
  - `/var/run/*.pid` – pliki przechowujące PID uruchomionych daemonów, utworzone przez skrypty kontrolujące.

Wywoływanie daemonów

- Skrypty kontrolujące zwykle przyjmują jeden z poniższych argumentów pozycyjnych (brak wyświetla sposób użycia skryptu)
  - `start` ,
  - `stop` ,
  - `restart` – przy aktualizacji albo zmianie konfiguracji,
  - `reload` – (opcjonalnie) przy zmianie konfiguracji,
  - `status` – (opcjonalnie) wyświetla status.
- Ta lista nie musi być kompletna – zależy od systemu i/lub daemona.
- Dodatkowe komunikaty (np. `[ok]` bądź `[fail]`) mogą być wysyłane przy okazji wywołania skryptu kontrolującego z w/w argumentami.
- Przykład zestawu plików związanych z serwisem SSH:
  - `/usr/sbin/sshd` — program daemona,
  - `/etc/ssh/sshd_config` -– konfiguracja SSH,
  - `/etc/init.d/sshd` — skrypt kontrolujący,
  - `/etc/conf.d/sshd` -– plik konfiguracyjny skryptu kontrolującego,
  - `/var/run/sshd.pid` -– PID działającego procesu `sshd`.

Odpalanie serwisu SSH w Gentoo Linux

```
# /etc/init.d/sshd start
```

Plan zajęć

systemd – co to?

- Inspirowany *svchost.exe* z Windowsa, rozbudowany wariant procesu `init` , zarządzający przy okazji wszystkimi serwisami (daemonami).
- Niełatwy do skonfigurowania, monolityczny system, który próbuje dokonać wielu rzeczy automagicznie.
- Ze względu na agresywne zrównoleglanie, złożoność, niemodularność, łamiący podstawowe zasady ideologii Unix.
- Z tego względu nierekomendowany na maszynach serwerowych. W zastosowaniach domowych zyskujący przewagę nad *kanonicznym* podejściem do systemu (aka SysV) ze względu na lepszą wydajność.
- Konfigurację określają pliki `/usr/lib/systemd/system/*.service`
- Zbliżone do Windowsowych **.ini*; zależą od pomocniczych `*.mount` i `*.socket` – tak zwanych *jednostek*.
- Domyślną konfigurację można nadpisać, zapisując *jednostki* do `/etc/systemd/system/` . Znaczące zmiany w konfiguracji mogą prowadzić do niejasnych zachowań.

systemctl – zarządzanie systemd

- Uniksowe daemony można uruchamiać jako serwisy systemd.
- Dla niektórych serwisów istnieją alternatywne, ściśle zintegrowane serwisy obowiązkowe – na przykład `journald` , zastępstwo dla standardowego Uniksowego `syslog` .
- Niektóre standardowe daemonowy można zainstalować i duplikować działanie wbudowanych serwisów.
- `systemctl` jest narzędziem do zarządzania serwisami w systemd.
- Niestety, systemd potrafi uruchomić serwisy wbrew konfiguracji (jako niezbędną? zależność innego zadania).

systemctl – przykład

```
# systemctl reload network-manager.service
```

Plan zajęć

Cron – kluczowe cechy

- Cron to standardowy daemon Uniksa, odpowiedzialny za wykonywanie komend periodycznie w określonych dniach i godzinach.
- Wpisy *crontab* można określać zarówno ogólnie, jak i per-użytkownik:
  - `/etc/crontab` albo `/var/spool/cron/root` zawiera wpisy ogólnosystemowe,
  - `/var/spool/cron/<uzytkownik>` zawiera wpisy użytkownika.
- Cron nie musi być restartowany w razie zmian w *crontab*.
- Istnieje kilka implementacji serwisu (m.in. `vixie-cron` , `dcron` , `cronie` , `anacron`).
- Różnią się m.in. lokalizacją plików *crontab*, drobnymi różnicami w syntaksie, sposobem określania dostępów do usługi i rozszerzeniami.

Podstawowy syntaks crona

Syntaks pliku *crontab*

```
min hour dmonth month dweek  user  command
```

- Pola oznaczają:
  - *min* – minutę (0-59),
  - *hour* – godzinę (0-23),
  - *dmonth* – dzień miesiąca (1-31),
  - *month* – miesiąc (1-12),
  - *dweek* – dzień tygodnia (0 i 7 oznaczają niedzielę),
  - *user* – to pole nie występuje we wszystkich implementacjach crona, a jeśli – to wyłącznie w `/etc/crontab`
  - *command* – komendę do wykonania; warto podać pełną ścieżkę.
- W pierwszych 5 polach można używać znaków specjalnych, m.in.:
  - `*` – dowolna wartość,
  - `,` – kilka wartości w liście,
  - `-` – zakres,
  - `/` – krok wzrostu, np. `0/2` – każda parzysta wartość.
  - `@daily` , `@weekly` – zamiast tych 5 pól.

Cron – przykłady

Wyłącz komputer każdego dnia o 11:15 wieczorem

```
15 23 * * *  root  shutdown now
```

Utwórz kopię zapasową w każdy dzień roboczy o 10 wieczorem

```
00 22 * * 1-5  root  /root/make_backup.sh
```

Przygotowania do piątku 13.

```
1 0 13 * 5  root  /root/enhanced_fault_tolerance.sh
```

Comiesięczna synchronizacja

```
@monthly  root  /root/synchronize.sh
```

Cron – cechy

- Zestaw zmiennych środowiskowych dostępnych dla skryptów wykonywanych przez cron jest zminimalizownay
- Dla ułatwionego zarządzania, niektóre implementacje dopuszczają rozpisanie konfiguracji crona w kilku osobnych plikach *crontab* w `/etc/cron.d/`
- Inne zaglądają do następujących katalogów:
  - `/etc/cron.hourly/`
  - `/etc/cron.daily/`
  - `/etc/cron.weekly/`
  - `/etc/cron.monthly/`

  i wykonują pliki (zwykle skrypty, albo symlinki do skryptów) w nich zawarte z adekwatną częstotliwością.

crontab – komenda

- `/etc/crontab` można edytować ręcznie, acz jest to niewskazane.
- Można użyć komendy `crontab` :
  - `crontab -l` dla listowania *crontab*a użytkownika,
  - `crontab -e` dla edycji *crontab*a użytkownika.
- Aktualizacja wpisów przez `crontab -e` przechodzi wstępną walidację syntaksu, i tylko jeśli zakończy się sukcesem, crontab jest aktualizowany.
- W zależności od systemu, użytkownik może mieć dostęp do crona, jeśli:
  - należy do grupy `cron`,
  - jest wylistowany w `/etc/cron.allow`
  - spełnia inne wymagania.

Logowanie – syslog

- Jądro oraz procesy piszą komunikaty logów do daemona *syslog*.
- Każdy komunikat syslog powinien zawierać informację czasową, źródło (host, proces), poziom (*critical*, *error*, *warning*, *info* etc.) i opisową informację.
- Komunikaty syslog można przekierować na terminal (np. `tty12`), pliku (`/var/log/*`) albo zewnętrzną instancję syslog.
- Niektóre daemony zamiast pisać do syslog, mają własną hierarchię plików logów.
- Pisanie na syslog jest dostępne za pomocą interfejsu `logger` .

Logowanie – czytanie i logrotate

- `less` jest rekomendowanym sposobem czytania plików logów, ponieważ nie cache'uje całego pliku. Otwieranie kilku-gigabajtowego pliku z logami za pomocą edytora tekstu to zły pomysł!
- Filtry strumieni, jak `grep` , są szczególnie przydatne do przeglądania logów z pojedynczego pliku.
- Istnieją automatyczne analizatory logów. Zwykle oparte są o regułowe parsery wyrażeń regularnych, które wykonywane są (cronem) codziennie, i sumaryzowane mailem do administratora.
- Logrotate periodycznie (np. co tydzień) odcina plik logów i kompresuje go.
- Starsze pliki logów (np. starsze niż miesiąc) są regularnie usuwane.
- Logrotate operuje na logach sysloga, jak i innych w `/var/log/*` .

Efekt działania logrotate

```
$ ls /var/log | grep kern
kern.log kern.log.1 kern.log.2.gz kern.log.3.gz
kern.log.4.gz
```