Definicje podstawowe i obowiązkowe:

- system operacyjny - System operacyjny jest to zbiór programów i procedur spełniających dwie podstawowe funkcje:

zarządzanie zasobami systemu komputerowego,
tworzenie maszyny wirtualnej.

- **zasób systemu** - Zasobem systemu jest każdy jego element sprzętowy lub programowy, który może być przydzielony danemu procesowi. Zasobami system zarządza w czasie i przestrzeni.
Przez zasoby sprzętowe rozumiemy:

czas procesora,
pamięć operacyjną,
urządzenia zewnętrzne,
inne komputery powiązane poprzez sieć teleinformatyczną.

Przez zasoby programowe rozumiemy:

pliki,
bufory,
semafory,
tablice systemowe.

- proces - Procesem nazywamy wykonujący się program wraz z jego środowiskiem obliczeniowym. Proces stanowi podstawowy obiekt dynamiczny w systemie operacyjnym.
- powłoka (interpreter poleceń) - Powłoka, interpreter poleceń ang. shell jest programem uruchamianym standardowo po otwarciu sesji użytkownika przez proces **login**. Powłoka jest aktywna aż do wystąpienia znaku <EOT>, który powoduje jej zatrzymanie i zgłoszenie tego faktu do jądra systemu. Każdy użytkownik otrzymuje własny i odrębny egzemplarz sh. Program sh wypisuje **monit $** na ekranie, dając znać o swojej gotowości do przyjęcia polecenia (komendy).
- sekcja krytyczna, - fragment programu, w którym występują instrukcje dostępu do zasobów dzielonych. Instrukcje tworzące sekcje krytyczne muszą być poprzedzone i zakończone operacjami realizującymi wzajemne wykluczanie.
- semafor - zmienna nazwana semaforem, inicjowana nieujemną wartością całkowitą i zdefiniowana poprzez definicje niepodzielnych operacji P(s) i V(s)

Wstęp

- tryby przetwarzania systemu komputerowego

tryb wsadowy, pośredni (ang. off-line, batch), autonomiczne wykorzystanie komputera bez konieczności obecności użytkownika
+ duża przepustowość systemu komputerowego,
– możliwy długi okres oczekiwania na wyniki, ograniczone możliwości szeregowania, niemożność bieżącej kontroli procesu wykonania.

tryb interaktywny, bezpośredni (ang. on-line, interactive), konwersacyjne współdziałanie użytkownika z systemem komputerowym z wykorzystaniem terminala komputera.
+ szybka reakcja systemu, możliwość kontroli przebiegu procesu wykonania,
– mniejsze wykorzystanie zasobów systemu komputerowego

tryb czasu rzeczywistego, system, którego użytkownikiem jest proces technologiczny narzucający pewne wymagania czasowe. Dwa podejścia:
– system jest zobowiązany do reagowania na zdarzenia zewnętrzne w ustalonym nieprzekraczalnym okresie.
– system bada okresowo stan procesu technologicznego

- rola przerwania w systemie komputerowym Umożliwia oddanie przez program użytkowy kontroli na rzecz systemu operacyjnego (wywołania systemowe) Umożliwia przełączanie kontekstu i wieloprogramowanie - przerwanie zegarowe Sygnalizuje zakończoną operację wejścia/wyjścia

- pojęcie blokady w systemie operacyjnym - Blokada (deadlock) występuje kiedy proces (lub wątek) oczekuje na dostęp do dzielonych zasobów, które są używane przez inny proces, który też czeka na dostęp do jakichś zajętych współdzielonych zasobów (i tworzą cykl zależności). Blokada oznacza że żaden z procesów nie może zmienić swojego stanu i będą czekać w nieskończonośc.

- przeznaczenie montowania systemu plików - Montowanie pozwala systemowi operacyjnemu na dostęp do plików i katalogów na urządzeniu przechowywania (takim jak dysk twardy, dysk SSD, pamięć USB) w sposób zorganizowany i kontrolowany

Programowanie w języku powłoki:

- użytkownicy w systemie Unix - Użytkownicy w systemie Unix

• superużytkownik (ang. superuser, root),
• pozostali użytkownicy.
Użytkownicy są opisani w pliku /etc/passwd
nazwa użytkownika
hasło (zaszyfrowane albo w ogóle ukryte)
uid - id użytkownika
gid - id grupy
informacje o użytkowniku
katalog domowy
domyślna powłoka
Grupy są opisane w pliku /etc/group
nazwa grupy
hasło grupy
numer grupy
lista użytkowników należących do grupy
W pliku /etc/shadow są przechowywane zaszyfrowane hasła użytkowników i dodatkowe **informacje**

- prawa dostępu do plików

Prawa dostępu do plików w systemach operacyjnych typu UNIX określają, kto i w jaki sposób może interagować z plikiem lub katalogiem. Są one podzielone na trzy główne kategorie:

- Właściciel (owner): Użytkownik, który ma kontrolę nad plikiem.
- Grupa (group): Użytkownicy należący do tej samej grupy, co plik.
- Inni (others): Wszyscy pozostali użytkownicy.

Prawa dostępu dla każdej z tych kategorii obejmują:

- Czytanie (r):
- Zapis (w):
- Wykonanie (x):

- znaczenie bitów SUID, SGID

Gdy bit SUID jest ustawiony na pliku wykonywalnym, proces uruchamiający ten plik otrzymuje uprawnienia właściciela pliku (zwykle roota) na czas jego wykonania.
SGID (Set Group ID): Podobnie, gdy bit SGID jest ustawiony na pliku wykonywalnym, proces uruchamiający ten plik otrzymuje uprawnienia grupy pliku.

- główne zmienne powłoki

- HOME - katalog domowy

- IFS - Internal Field Separator - znaki rozdzielające elementy składni w linii (np. domyślnie oddzielanie spacją)
- PATH - lista katalogów, w których szukane są pliki wywoływanych komend (np. /bin , /usr/bin)
- PS1 - pierwszy znak zachęty $
- PS2 - drugi znak zachęty > (kontynuacja linii)
- SHELL - domyślna podpowłoka
- TERM - rodzaj terminala

- parametry powłoki - Parametry powłoki

$0 nazwa wywołanej komendy (cmd)
$1 pierwszy argument (parametr) wywołania
$2 drugi argument (parametr) wywołania
$9 dziewiąty argument (parametr) wywołania
$* argumenty jako jeden łańcuch znaków "$*" = "$1 $2 .."
$@ argumenty jako osbne łańcuchy znaków "$@" = "$1" "$2" ..
$# liczba argumentów przekazanych przy wywołaniu lub przez set,
$? stan końcowy (ang. exit status) ostatnio wykonywanej komendy,
$$ numer procesu aktualnie wykonywanej powłoki,
$! numer procesu ostatnio wykonywanego procesu w tle.
$0-9 także: opcje przypisane powłoce przy wywołaniu lub przez set

- polecenia zewnętrzne a wbudowane – uzasadnienie rozróżnienia

Optymalizacja Wydajności: Używanie poleceń wbudowanych tam, gdzie to możliwe, zapewnia lepszą wydajność.
Zarządzanie Środowiskiem Powłoki: Polecenia wbudowane są niezbędne do zarządzania środowiskiem powłoki, co nie byłoby możliwe za pomocą poleceń zewnętrznych.
Elastyczność i Rozszerzalność: Polecenia zewnętrzne pozwalają na rozszerzenie funkcjonalności systemu bez konieczności modyfikacji samej powłoki.

- odczyt pliku /etc/passwd z wykorzystaniem read/set/IFS,

```
cat /etc/passwd | while IFS=: read username password uid other
do
    echo "user: $username"
    echo "uid: $uid"
done
```

```
cat /etc/passwd | while read line
do
    IFS=: set $line
    echo "username: $1"
    echo "uid: $3"
done
```

for wykonywana dla każdego słowa w liście słów

```
for i in /tmp /usr/tmp
do
        rm -rf $i/*
done
```

while - wykonywana tak długo jak spełniowy jest warunek

```
i=1
while [ $i -le 5 ];
do
        echo $i i=`expr $i + 1`
done
```

until - wykonywana tak długo jak warunek **nie** jest spełniony

```
i=1
until [ $i -ge 5 ];
do
    echo $i
    i=`expr $i + 1`
done
```

- obsługa argumentów wywołania skryptu i funkcji

Argumenty Skryptu: Dostępne poprzez $1, $2, ..., gdzie $1 to pierwszy argument, $2 to drugi itd. $# zawiera liczbę argumentów.

Argumenty Funkcji: Podobnie jak w skryptach, dostępne przez $1, $2, ..., w kontekście danej funkcji.

Procesy i wątki:

- **pojęcia procesów współbieżnych/równoległych/rozproszonych**

- procesy współbieżne - nie ma wymogu, żeby kolejny proces startował po zakończeniu poprzedniego (nie muszą być szeregowe)
- procesy równoległe - może istnieć chwila czasu, kiedy wykonują się dokładnie w tej samej chwili czasu (nie muszą ale nie można teog wykluczyć) musi być więcej niż jedno ALU (w praktyce wiele rdzeni, wiele procesorów) przereklamowana? (zysk zależy od poziomu ziarnistości dekompozycji problemu, są obliczenia które świetnie się nadają do zrównoleglania (grafika))
- procesy rozproszone - wykonują się na sprzęcie rozproszonym Nie mają współdzielonej pamięci Scentralizowany sprzęt nie wymaga komunikacji sieciowej Są z założenia równoległe

- graf przejść stanów procesów w systemie Unix - prosty i złożony (System V),

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

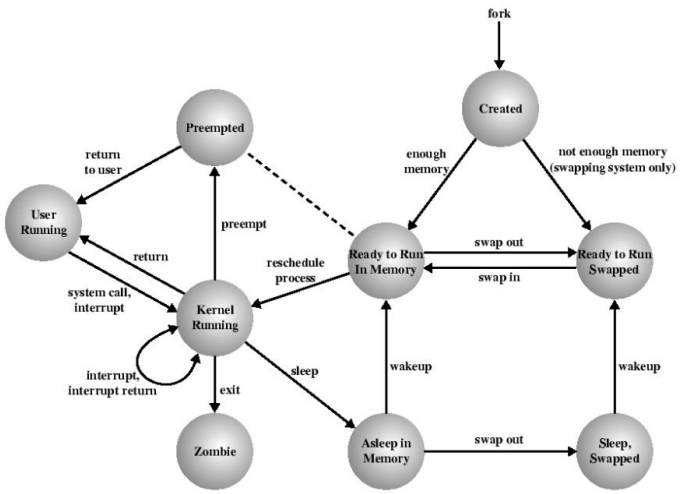

Figure 3.16 UNIX Process State Transition Diagram

- przeznaczenie i budowa wektora przerwań - wektor przerwania zawiera adres procedury dostarczonej przez system operacyjny, z każdym urządzeniem I/O skojarzona jest lokalizacja w pamięci z informacją, skąd ma nastąpić kontynuacja wykonywania podprogramu w przypadku wystąpienia przerwania

- porównanie własności procesów i wątków

Cecha	Procesy	Wątki
Izolacja	Wysoka (oddzielna przestrzeń pamięci)	Niska (współdzielona przestrzeń pamięci)
Zasoby	Własne zasoby (pamięć, pliki)	Współdzielone zasoby (pamięć, pliki)
Koszt tworzenia	Wyższy	Niższy
Zarządzanie	Bardziej kosztowne	Mniej kosztowne
Komunikacja	Poprzez IPC	Bezpośrednio przez współdzieloną pamięć
Odporność	Wyższa (błąd w jednym nie wpływa na inne)	Niższa (błąd może wpływać na inne wątki)

- **porównanie wątków poziomu jądra i wątków poziomu użytkownika,**

Realizacja

Kernel-level - tworzeniem, niszczeniem i szeregowaniem zajmuje się jądro systemu

User-level - realizowane przez biblioteki w językach programowania, niewidoczne dla jądra, mogą być zawsze zaimplementowane niezależnie czy system operacyjny to wspiera

Czas przełączenia

Przełączenie wątków poziomu jądra jest kosztowne, wymaga pełnego przełączenia kontekstu, przeładowania pamięci, cache staje się nieważny

Przełączenie wątków poziomu użytkownika jest bardzo szybkie - kilka instrukcji maszynowych w procedurze bibliotecznej

Blokujące operacje IO

User-level - blokuje cały proces (bo jądro nic nie wie o tym że proces ma wiele wątków)

Kernel-level - nie blokuje procesu, jądro może przełączyć się na inny wątek tego samego procesu

Szeregowanie

Dla wątków poziomu użytkownika można stosować algorytm szeregowania specyficzny dla aplikacji i zapewnić lepszą wydajność - można wykorzystać wiedzę co robi każdy z wątków (dispatcher, worker itd)

Dla wątków poziomu jądra, jądro nie ma takich informacji (można conajwyżej ustawiać odpowiednie priorytety)

- **architektura wielowątkowa w systemie Solaris -** Solaris wykorzystuje cztery rozłączne koncepcje:

• Procesy - standardowe procesy systemu Unix,

• Wątki poziomu użytkownika - zaimplementowane bibliotecznie, nierozróżnialne z punktu widzenia jądra, stanowią interfejs do współbieżności,

• Procesy lekkie (ang. Lightweight processes, LWP), LWP stanowi formę odwzorowania między wątkami poziomu użytkownika a wątkami jądra.

– każdy LWP obsługuje jeden bądź więcej wątków poziomu użytkownika odwzorowując w jeden wątek jądra,

– LWP rozróżniane i szeregowane przez jądro,

– LWP mogą być uruchomione równolegle w architekturze wieloprocesorowej,

• Wątki jądra podstawowe elementy szeregowane i rozmieszczane na procesorach.

- **metody konstrukcji serwerów** usług

Metody organizacji serwerów usług:

• serwery wielowątkowe - współbieżność, blokujące wywołania systemowe,

• procesy jednowątkowe - brak współbieżności, blokujące wywołania systemowe,

• automaty skończone - współbieżność, nieblokujące wywołania systemowe, przerwania.

- szeregowanie z wywłaszczaniem i bez wywłaszczania,

Szeregowanie bez wywłaszczania

Planista wybiera proces, proces wykonuje się dopóki nie się nie zablokuje (na IO, semaforze itd)

albo dobrowolnie odda kontrolę

Planista nie podejmuje decyzji szeregujących na przerwaniach zegarowych

Czas CPU nie jets dzielony na kwanty

Szeregowanie z wywłaszczaniem

Czas CPU jest podzielony na kwanty

Wybrany proces może wykonywać się conajwyżej przez dany kwant czasu

Po upłynięciu czasu przychodzi przerwanie zegarowe i planista może wybrać inny proces

- szeregowanie procesów - wskaźniki jakości szeregowania dla różnych trybów przetwarzania,

Wszystkie systemy

- sprawiedliwość - każdemu równą część czasu CPU,
- zgodność z polityką - praca wedle założeń,
- wyrównywanie - zbliżona zajętość wszystkich części systemu.

Systemy wsadowe

- przepustowość - maksymalizacja liczby zadań w czasie,
- czas w systemie - min. czasu między uruchomieniem a zakończeniem,
- wykorzystanie procesora - minimalizacja przerw w pracy procesora.

Systemy interaktywne

- czas odpowiedzi - możliwie szybka odpowiedź na żądanie,
- proporcjonalność - spełnianie oczekiwań użytkownika.

Systemy czasu rzeczywistego

- spełnianie wymagań - spełnianie ograniczeń czasowych,
- przewidywalność - np. unikanie spadku jakości w przekazie multimediów

Wzajemne wykluczanie i synchronizacja:

- wyścig i warunki wyścigu - nazywamy sytuację, w której dwa lub więcej procesów wykonuje operację na zasobach dzielonych, a ostateczny wynik tej operacji jest zależny od momentu jej realizacji.

- warunki konieczne implementacji sekcji krytycznej - Dla prawidłowej implementacji sekcji krytycznych muszą być spełnione następujące 3 warunki, przy czym nie czynimy żadnych założeń dotyczących szybkości działania procesów, czy też liczby procesorów:

1. wewnątrz SK może przebywać tylko jeden proces,
2. jakikolwiek proces znajdujący się poza SK, nie może zablokować innego procesu pragnącego wejść do SK,
3. każdy proces oczekujący na wejście do SK powinień otrzymać prawo dostępu w rozsądnym czasie.

- mechanizmy realizacji wzajemnego wykluczania z aktywnym oczekiwaniem – opis i porównanie

1. Blokowanie Przerwań (Disabling Interrupts)

Działanie: Proces wchodzący do sekcji krytycznej (SK) blokuje przerwania, a wychodząc odblokowuje.

Zalety: Zapobiega interwencji innych procesów podczas aktualizacji zasobów dzielonych.

Wady: Może prowadzić do upadku systemu, jeśli przerwania nie zostaną odblokowane; nieskuteczny w systemach wieloprocesorowych.

2. Zmienne Blokujące (Lock Variables)

Działanie: Użycie zmiennej lock, która wskazuje, czy SK jest zajęta.

Problem: Problem wyścigu, ponieważ sprawdzanie i ustawianie zmiennej lock nie jest atomowe.

3. Ścisłe Następstwo (Strict Alternation)
   Działanie: Procesy na zmianę wchodzą do SK, kontrolowane przez zmienną turn.
   Problem: Zagłodzenie – proces może zostać zablokowany nieskończenie długo, jeśli drugi proces nie chce korzystać ze SK.
4. Algorytm Petersona
   Działanie: Kombinacja ścisłego następstwa i zmiennych blokujących; każdy proces przed wejściem do SK wywołuje enter_region z własnym numerem.
   Zalety: Skutecznie rozwiązuje problem wzajemnego wykluczania bez użycia przerwań.
   Wady: Nadal wymaga aktywnego oczekiwania, co może być marnotrawstwem zasobów.
5. Instrukcja TSL (Test and Set Lock)
   Działanie: Sprzętowa instrukcja, która atomowo czyta i ustawia wartość blokady.
   Zalety: Atomowość działania zapobiega problemom wyścigu.
   Wady: Podobnie jak inne, wiąże się ze stratą czasu procesora na aktywne oczekiwanie.

Porównanie
   Efektywność: Algorytm Petersona i instrukcja TSL są bardziej efektywne niż blokowanie przerwań czy ścisłe następstwo, ponieważ lepiej radzą sobie z problemami wyścigu i zagłodzenia.
   Zastosowanie: Blokowanie przerwań jest ograniczone do środowiska jądra, podczas gdy inne techniki mogą być stosowane w przestrzeni użytkownika.
   Koszt zasobów: Wszystkie te techniki, oprócz instrukcji TSL, wiążą się z aktywnym oczekiwaniem, co może prowadzić do nieefektywnego wykorzystania procesora.
   Złożoność: Algorytm Petersona jest prostszy w implementacji niż ścisłe następstwo, ale bardziej skomplikowany niż proste zmienne blokujące.

- mechanizmy realizacji wzajemnego wykluczania z wstrzymywaniem procesu - opis i porównanie

1. Sleep and Wakeup
   Działanie: sleep() zawiesza proces do momentu obudzenia przez inny proces za pomocą wakeup().
   Zastosowanie: Przykładowo w problemie producent-konsument.
   Wady: Możliwość utraty sygnału wakeup, co prowadzi do blokady.
2. Semafory
   Działanie: Zmienna całkowita zliczająca sygnały wakeup. Operacje P (wait) i V (signal) zmieniają wartość semafora w sposób niepodzielny.
   Zastosowanie: Oszczędniejsze od sleep/wakeup, często używane w systemach operacyjnych.
   Wady: Potrzeba zarządzania kolejkami procesów czekających.
3. Monitory
   Działanie: Zbiór procedur, zmiennych i struktur danych w specjalnym module. Tylko jeden proces może przebywać w monitorze.
   Zastosowanie: Uproszczenie programowania współbieżnego, automatyzacja mechanizmu wzajemnego wykluczania.
   Wady: Ograniczona dostępność w językach programowania, trudność implementacji w środowiskach rozproszonych.
4. Komunikaty (Message Passing)
   Działanie: Wymiana informacji między procesami przez wysyłanie i odbieranie wiadomości.
   Zastosowanie: Skuteczne w systemach rozproszonych, gdzie procesy nie mają wspólnej pamięci.
   Wady: Złożoność zarządzania buforami wiadomości, potencjalne opóźnienia w komunikacji.

Porównanie

Efektywność: Semafory i monitory są zwykle bardziej efektywne niż sleep/wakeup, ponieważ lepiej zarządzają stanami oczekiwania procesów.
Złożoność: Monitory i komunikaty są bardziej skomplikowane w implementacji niż semafory czy sleep/wakeup, ale oferują większą kontrolę i bezpieczeństwo.
Uniwersalność: Komunikaty są najlepsze dla systemów rozproszonych, podczas gdy semafory i monitory są lepsze w środowiskach z wspólną pamięcią.
Ryzyko blokad: Sleep/wakeup ma ryzyko utraty sygnałów, podczas gdy semafory i monitory mają wbudowane mechanizmy zapobiegające takim sytuacjom.

- producent-konsument - poprawna realizacja z synchronizacją semaforami

```
void producer(void)
{
    while (TRUE)
    {
        produce_item();
        down( empty );
        down( mutex );
        enter_item();
        up( mutex );
        up( full );
    }
}
```

```
void consumer(void)
{
    while (TRUE)
    {
        down( full);
        down( mutex );
        remove_item();
        up( mutex );
        up( empty );
        consume_item();
    }
}
```

- producent-konsument - poprawna realizacja z synchronizacją monitorami,

```
monitor Buffer
    condition full, empty;
    integer count;

    procedure enter;
    begin
        if count = N
            then wait(full);
        enter_item;
        count := count + 1;
        if count = 1
            then signal(empty);
    end;

    count := 0;
end monitor;
```

```
procedure remove;
begin
    if count = 0
        then wait(empty);
    remove_item;
    count := count - 1;
    if count = N-1
        then signal(full);
end;
```

```
procedure producer;
begin
    while true do
    begin
            produce_item;
            Buffer.enter;
    end
end;
```

```
procedure consumer;
begin
    while true do
    begin
            Buffer.remove;
            consume_item;
    end
end;
```

Przykładowe zadania projektowe:

- proste skrypty sh (bez rozszerzeń bash):
- skrypt, który wysyła na wyjście linia po linii tekst ze standardowego wejścia, poprzedzając każdą linię numerem linii,

```bash
#!/bin/bash

# Initialize a counter
line_number=1

# Read from standard input
while IFS= read -r line
do
  # Output the line number followed by the line content
  echo "${line_number}: $line"

  # Increment the line number
  ((line_number++))
done
```

- skrypt, który przyjmuje jako argument id użytkownika i wypisuje na stdin nazwę użytkownika (korzysta z relacji w /etc/passwd),

```bash
#!/bin/sh

if [ "x$1" = "x" ];
then
    echo "Pass user id as the first argument" >&2
    exit 1
fi
```

```bash
uid=$1
while IFS=":" read username password userid _
do
    if [ "x$userid" = "x$uid" ];
    then
        echo $username
        exit 0
    fi
done < /etc/passwd

echo "User not found" >&2
exit 2
```

- skrypt wypisujący zadane argumenty w odwrotnej kolejności,

```bash
#!/bin/bash

# Loop through the arguments in reverse
for (( idx=$#; idx>0; idx-- )); do
    echo "${!idx}"
done
```

- opisz rezultat wykonania poleceń i grup poleceń:

x >/dev/null 2>&1 - zarówno standardowe wyjście, jak i standardowe wyjście błędów polecenia x są ignorowane. Nie zobaczysz żadnego wyniku ani komunikatów o błędach od x.
x 2>&1 1>/dev/null - Efektem tego jest, że standardowe wyjście błędów (stderr) zostanie wyświetlone, a standardowe wyjście (stdout) zostanie zignorowane.
x 2>/dev/null 1>/dev/null - zarówno standardowe wyjście, jak i wyjście błędów są ignorowane.
x < y > z - x przetwarza dane z pliku y i zapisuje wynik do pliku z.
x & y ; z - x działa w tle, podczas gdy y i następnie z są wykonywane kolejno.
x || y - jeśli polecenie x zakończy się niepowodzeniem (zwróci kod wyjścia różny od zero), zostanie wykonane polecenie y
- opisz precyzyjnie działanie poniższych komend, ile i dokładnie kiedy tworzonych jest procesów?
x > y
Uruchamia proces dla polecenia x.

Standardowe wyjście polecenia x jest przekierowywane do pliku y.
Tworzony jest 1 proces.

x | y

Uruchamia procesy dla poleceń x i y.
Standardowe wyjście x jest przekierowywane do standardowego wejścia polecenia y.
Tworzone są 2 procesy jednocześnie.

x < y

Uruchamia proces dla polecenia x.
Standardowe wejście polecenia x jest przekierowywane z pliku y.
Tworzony jest 1 proces.

cat y | x

Uruchamia procesy dla poleceń cat y i x.
Standardowe wyjście polecenia cat y (czyli zawartość pliku y) jest przekierowywane do standardowego wejścia polecenia x.
Tworzone są 2 procesy jednocześnie.

x & y

Uruchamia proces dla polecenia x w tle (asynchronicznie).
Następnie uruchamia proces dla polecenia y.
Tworzone są 2 procesy, ale nie jednocześnie (najpierw x, potem y).

x && y

Uruchamia proces dla polecenia x.
Jeśli x zakończy się sukcesem (kod wyjścia 0), następnie uruchamia proces dla polecenia y.
Tworzone są 1 lub 2 procesy, w zależności od powodzenia x

x || y

Uruchamia proces dla polecenia x.
Jeśli x zakończy się niepowodzeniem (kod wyjścia inny niż 0), następnie uruchamia proces dla polecenia y.
Tworzone są 1 lub 2 procesy, w zależności od powodzenia x

x ; y &

Uruchamia proces dla polecenia x.
Po zakończeniu x, uruchamia proces dla polecenia y w tle.
Tworzone są 2 procesy, ale nie jednocześnie (najpierw x, potem y).

cat x > y < z

Uruchamia proces dla polecenia cat x.
Standardowe wejście dla cat jest przekierowywane z pliku z, ale ponieważ cat x już używa pliku x jako wejścia, przekierowanie < z jest ignorowane.
Standardowe wyjście polecenia cat x jest przekierowywane do pliku y.
Tworzony jest 1 proces.

Zadanie

Załóżmy, że plik /etc/passwd miałby strukturę

login:id:gr0:shell:gr1,gr2,g3

tnowak:1001:tnowak:/bin/bash:admin,mail,www

przy czym gr0 występuje zawsze, a dodatkowe nazwy grup po ostatnim dwukropku
są opcjonalne.

Napisz skrypt 'gr', który jako argumenty pobiera nazwy użytkowników, a następnie

dla każdego użytkownika w osobnych liniach wypisuje "user: " i nazwę użytkownika, a potem po jednej nazwie grupy w linii, przykładowo:

$ ./gr tnowak
user: tnowak
tnowak
admin
mail
www
$

```
#!/bin/sh

process_user () {
    echo "user: $1"
    cat ./passwd | while IFS=: read login id group shell groups
    do
        if [ "x$login" = "x$1" ];
        then
            print_groups $group $groups
        fi
    done
}

print_groups () {
    echo $1
    if [ "x$2" = "x" ];
```

```
    then
        exit 0
    fi

    IFS=, set $2
    print_args $*
}

print_args () {
    while [ "x$1" != "x" ];
    do
        echo $1
        shift
    done
}


while [ "x$1" != "x" ];
do
    process_user $1
    shift
done
```