Procesy

Systemy Operacyjne 2 — laboratorium

Mateusz Hołenko

9 października 2011

Plan zajęć

1. Procesy w systemie Linux
   - proces
   - procesy macierzyste i potomne
   - procesy zombie i sieroty
2. Funkcje systemowe
   - pobieranie identyfikatorów
   - tworzenie procesów potomnych
   - uruchamianie programów w ramach procesu
   - kończenie i zabijanie procesów
   - synchronizacja procesów macierzystych i potomnych
   - przekierowywanie strumieni wejściowych/wyjściowych
3. Zadania teoretyczne
4. Zadania praktyczne

Procesy w systemie Linux

Proces

- proces to nie program!
- pojęcie abstrakcyjne — program w trakcie wykonywania
- procesami zarządza system operacyjny
- procesom przydzielane są zasoby systemowe
  - procesor
  - pamięć operacyjna
  - urządzenia zewnętrzne

Proces

- proces to nie program!
- pojęcie abstrakcyjne — program w trakcie wykonywania
- procesami zarządza system operacyjny
- procesom przydzielane są zasoby systemowe
  - procesor
  - pamięć operacyjna
  - urządzenia zewnętrzne

Proces

- proces to nie program!
- pojęcie abstrakcyjne — program w trakcie wykonywania
- procesami zarządza system operacyjny
- procesom przydzielane są zasoby systemowe
  - procesor
  - pamięć operacyjna
  - urządzenia zewnętrzne

Proces

- proces to nie program!
- pojęcie abstrakcyjne — program w trakcie wykonywania
- procesami zarządza system operacyjny
- procesom przydzielane są zasoby systemowe
  - procesor
  - pamięć operacyjna
  - urządzenia zewnętrzne

Procesy macierzyste i potomne

- procesy tworzą hierarchię
  - proces główny nazywa się init
- proces tworzący inne procesy staje się dla nich rodzicem (procesem macierzystym)
- wynik polecenia `ps l a` (PID — identyfikator procesu, PPID — identyfikator rodzica procesu)

```
F   UID   PID  PPID  PRI  NI    VSZ   RSS WCHAN  STAT TTY      TIME COMMAND
...
0     0  1773     1   20   0   6196   644 n_tty_ Ss+  tty1     0:00 /sbin/getty ... tty1
0  1000  2090  2028   20   0  26188  6532 wait   Ss   pts/0    0:00 /bin/bash
0  1000 32096  2090   20   0  10660  1156 -      R+   pts/0    0:00 ps l a
```

Procesy macierzyste i potomne

- procesy tworzą hierarchię
  - proces główny nazywa się init
- proces tworzący inne procesy staje się dla nich rodzicem (procesem macierzystym)
- wynik polecenia `ps l a` (PID — identyfikator procesu, PPID — identyfikator rodzica procesu)

```
F   UID   PID  PPID  PRI  NI    VSZ   RSS WCHAN  STAT TTY      TIME COMMAND
...
0     0  1773     1   20   0   6196   644 n_tty_ Ss+  tty1     0:00 /sbin/getty ... tty1
0  1000  2090  2028   20   0  26188  6532 wait   Ss   pts/0    0:00 /bin/bash
0  1000 32096  2090   20   0  10660  1156 -      R+   pts/0    0:00 ps l a
```

Procesy macierzyste i potomne

- procesy tworzą hierarchię
  - proces główny nazywa się init
- proces tworzący inne procesy staje się dla nich rodzicem (procesem macierzystym)
- wynik polecenia `ps l a` (PID — identyfikator procesu, PPID — identyfikator rodzica procesu)

```
F   UID   PID  PPID  PRI  NI    VSZ   RSS WCHAN  STAT TTY      TIME COMMAND
...
0     0  1773     1   20   0   6196   644 n_tty_ Ss+  tty1     0:00 /sbin/getty ... tty1
0  1000  2090  2028   20   0  26188  6532 wait   Ss   pts/0    0:00 /bin/bash
0  1000 32096  2090   20   0  10660  1156 -      R+   pts/0    0:00 ps l a
```

Procesy zombie i sieroty

- procesy mogą kończyć swoje działanie w różnej kolejności
- procesy sieroty (ang. orphan) — proces macierzysty się zakończył
  - procesem macierzystym staje się init
- procesy zombie (ang. zombie) — proces zakończył się, ale czeka na przekazanie statusu zakończenia
  - nie zajmuje procesora ani innych zasobów
  - występuje w tablicy procesów (tam znajduje się bowiem status zakończenia procesu)

Procesy zombie i sieroty

- procesy mogą kończyć swoje działanie w różnej kolejności
- procesy sieroty (ang. orphan) — proces macierzysty się zakończył
  - procesem macierzystym staje się init
- procesy zombie (ang. zombie) — proces zakończył się, ale czeka na przekazanie statusu zakończenia
  - nie zajmuje procesora ani innych zasobów
  - występuje w tablicy procesów (tam znajduje się bowiem status zakończenia procesu)

Procesy zombie i sieroty

- procesy mogą kończyć swoje działanie w różnej kolejności
- procesy sieroty (ang. orphan) — proces macierzysty się zakończył
  - procesem macierzystym staje się init
- procesy zombie (ang. zombie) — proces zakończył się, ale czeka na przekazanie statusu zakończenia
  - nie zajmuje procesora ani innych zasobów
  - występuje w tablicy procesów (tam znajduje się bowiem status zakończenia procesu)

Funkcje systemowe

Pobieranie identyfikatorów

- identyfikator procesu PID

```
pid_t getpid(void)
```

- identyfikator rodzica procesu PPID

```
pid_t getppid(void)
```

Pobieranie identyfikatorów

- identyfikator procesu PID

```
pid_t getpid(void)
```

- identyfikator rodzica procesu PPID

```
pid_t getppid(void)
```

Tworzenie procesów potomnych

- tworzenie procesu potomnego

```
pid_t fork(void)
```

- przykład zastosowania

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("Początek programu\n");
  if (fork() == 0)
  {
    printf("Tutaj pisze proces potomny!\n");
  }
  else
  {
    printf("Tutaj pisze proces macierzysty!\n");
  }
  printf("Tutaj piszą oba procesy!\n");
}
```

Tworzenie procesów potomnych

- tworzenie procesu potomnego

```
pid_t fork(void)
```

- przykład zastosowania

```
#include <unistd.h>                                                    1
#include <stdio.h>                                                     2
int main(int argc, char* argv[])                                       3
{                                                                      4
  printf("Początek programu\n");                                       5
  if (fork() == 0)                                                     6
  {                                                                    7
    printf("Tutaj pisze proces potomny!\n");                           8
  }                                                                    9
  else                                                                 10
  {                                                                    11
    printf("Tutaj pisze proces macierzysty!\n");                       12
  }                                                                    13
  printf("Tutaj piszą oba procesy!\n");                                14
}                                                                      15
```

Uruchamianie programu w ramach procesu

- rodzina funkcji exec

```
int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
int execle(const char *path, const char *arg , ..., char *const envp[])

int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execve(const char *file, char *const argv[], char *const envp[])
```

- poprawne wykonanie funkcji exec powoduje bezpowrotną zmianę wykonywanego programu!
  - sugerowane wykorzystanie w połączeniu z funkcją fork

Uruchamianie programu w ramach procesu

- rodzina funkcji exec

```
int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
int execle(const char *path, const char *arg , ..., char *const envp[])

int execv(const char *path, char *const argv[])
int execvp(const char *file, char *const argv[])
int execve(const char *file, char *const argv[], char *const envp[])
```

- poprawne wykonanie funkcji exec powoduje bezpowrotną zmianę wykonywanego programu!
  - sugerowane wykorzystanie w połączeniu z funkcją fork

Kończenie i zabijanie procesu

- zakończenie działania procesu i zwrócenie statusu zakończenia
  - `status` — status zakończenia procesu
    - proces zakończony poprawnie: 0
    - błąd: inna wartość

```
void exit(int status)
```

- zabicie procesu potomnego
  - `pid` — identyfikator procesu, który ma zostać zabity (patrz `man 2 kill`!)
  - `sig` — numer sygnału wysyłany do procesu potomnego

```
int kill(pid_t pid, int sig)
```

Kończenie i zabijanie procesu

- zakończenie działania procesu i zwrócenie statusu zakończenia
  - `status` — status zakończenia procesu
    - proces zakończony poprawnie: 0
    - błąd: inna wartość

```
void exit(int status)
```

- zabicie procesu potomnego
  - `pid` — identyfikator procesu, który ma zostać zabity (patrz `man 2 kill`!)
  - `sig` — numer sygnału wysyłany do procesu potomnego

```
int kill(pid_t pid, int sig)
```

Synchronizacja procesów

- synchronizacja procesu macierzystego i potomnego
  - `status` — adres komórki pamięci, do którego zapisany zostanie status zakończenia procesu potomnego
  - `pid` — identyfikator procesu, na którego zakończenie funkcja ma oczekiwać (patrz `man 2 wait`!)
  - `options` — opcje określające sposób zakończenia funkcji

```
pid_t wait(int* status)
pid_t waitpid(pid_t pid, int* status, int options)
```

Przykład wykorzystania funkcji exec

```
#include <unistd.h>                                          1
#include <stdio.h>                                           2
                                                             3
int main(int argc, char* argv[])                             4
{                                                            5
  printf("Początek działania programu\n");                   6
  if (fork() == 0)                                           7
  {                                                          8
    execlp("ls", "ls", "-a", NULL);                          9
    perror("Błąd uruchamiania programu!");                   10
    exit(1);                                                 11
  }                                                          12
  wait(NULL);                                                13
  printf("Koniec działania programu\n");                     14
  return 0;                                                  15
}                                                            16
```

Przekierowywanie strumieni wejściowych/wyjściowych

- proces potomny dziedziczy tablicę otwartych deskryptorów
- przekierowania procesów ustawione przed wywołaniem funkcji `fork` obowiązują również w procesie potomnym!
- przykładowy program

```
#include <unistd.h>                                    1
#include <stdio.h>                                     2
#include <fcntl.h>                                     3
                                                       4
int main(int argc, char* argv[])                       5
{                                                      6
  dup2(creat("ls-out", 0600), 1);                      7
  execvp("ls", argv);                                  8
}                                                      9
```

Przekierowywanie strumieni wejściowych/wyjściowych

- proces potomny dziedziczy tablicę otwartych deskryptorów
- przekierowania procesów ustawione przed wywołaniem funkcji `fork` obowiązują również w procesie potomnym!
- przykładowy program

```
#include <unistd.h>                                    1
#include <stdio.h>                                     2
#include <fcntl.h>                                     3
                                                       4
int main(int argc, char* argv[])                       5
{                                                      6
  dup2(creat("ls-out", 0600), 1);                      7
  execvp("ls", argv);                                  8
}                                                      9
```

Przekierowywanie strumieni wejściowych/wyjściowych

- proces potomny dziedziczy tablicę otwartych deskryptorów
- przekierowania procesów ustawione przed wywołaniem funkcji `fork` obowiązują również w procesie potomnym!
- przykładowy program

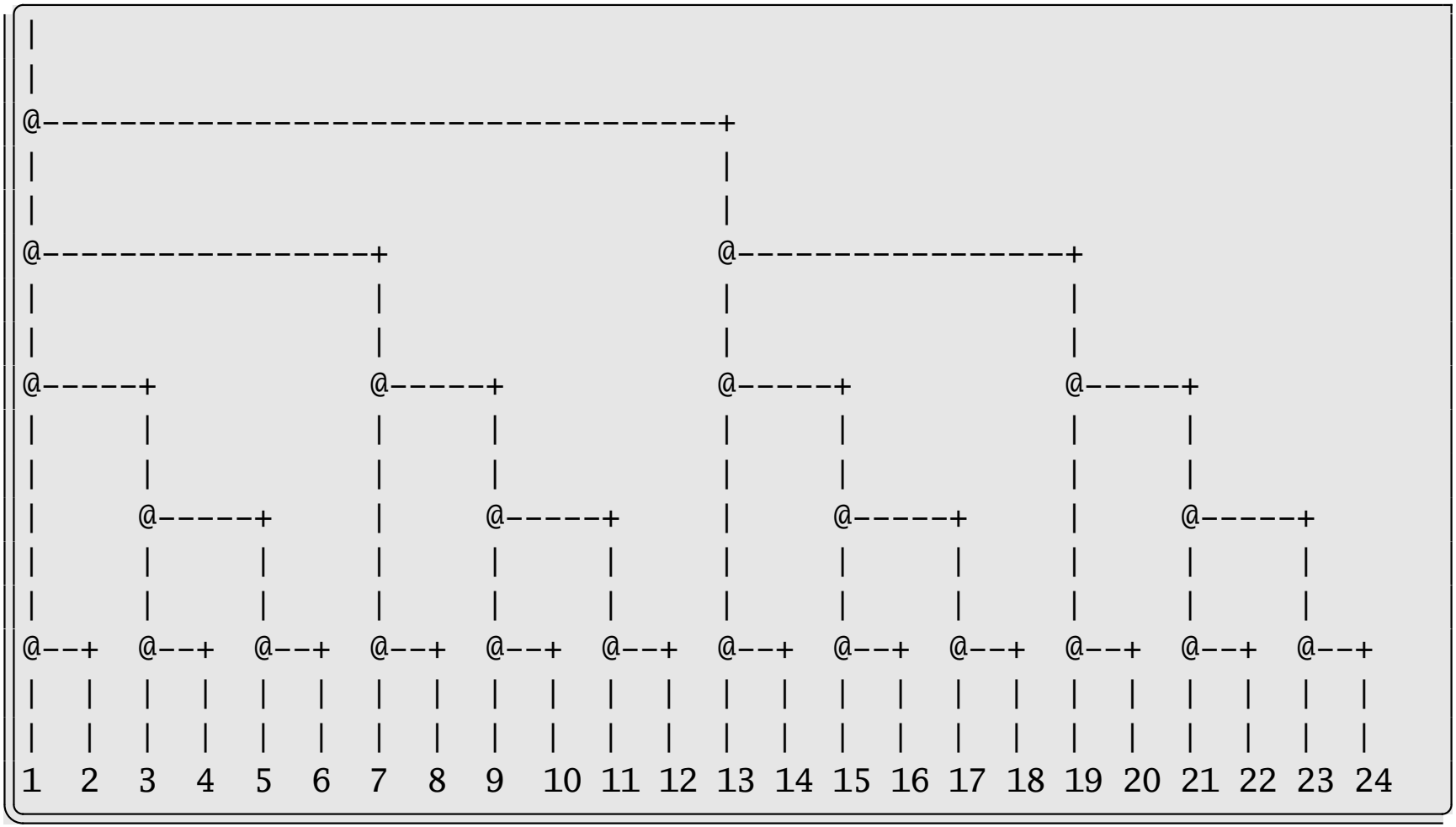
```
#include <unistd.h>                                    1
#include <stdio.h>                                     2
#include <fcntl.h>                                     3
                                                       4
int main(int argc, char* argv[])                       5
{                                                      6
  dup2(creat("ls-out", 0600), 1);                      7
  execvp("ls", argv);                                  8
}                                                      9
```

Zadania teoretyczne

Ile procesów zostanie stworzonych?

```
#include <unistd.h>                                    1
#include <stdio.h>                                     2
#include <fcntl.h>                                     3
                                                       4
int main(int argc, char* argv[])                       5
{                                                      6
  fork();                                              7
  fork();                                              8
  if(fork() == 0)                                      9
    fork();                                            10
  fork();                                              11
                                                       12
  return 0;                                            13
}                                                      14
```

Odpowiedź



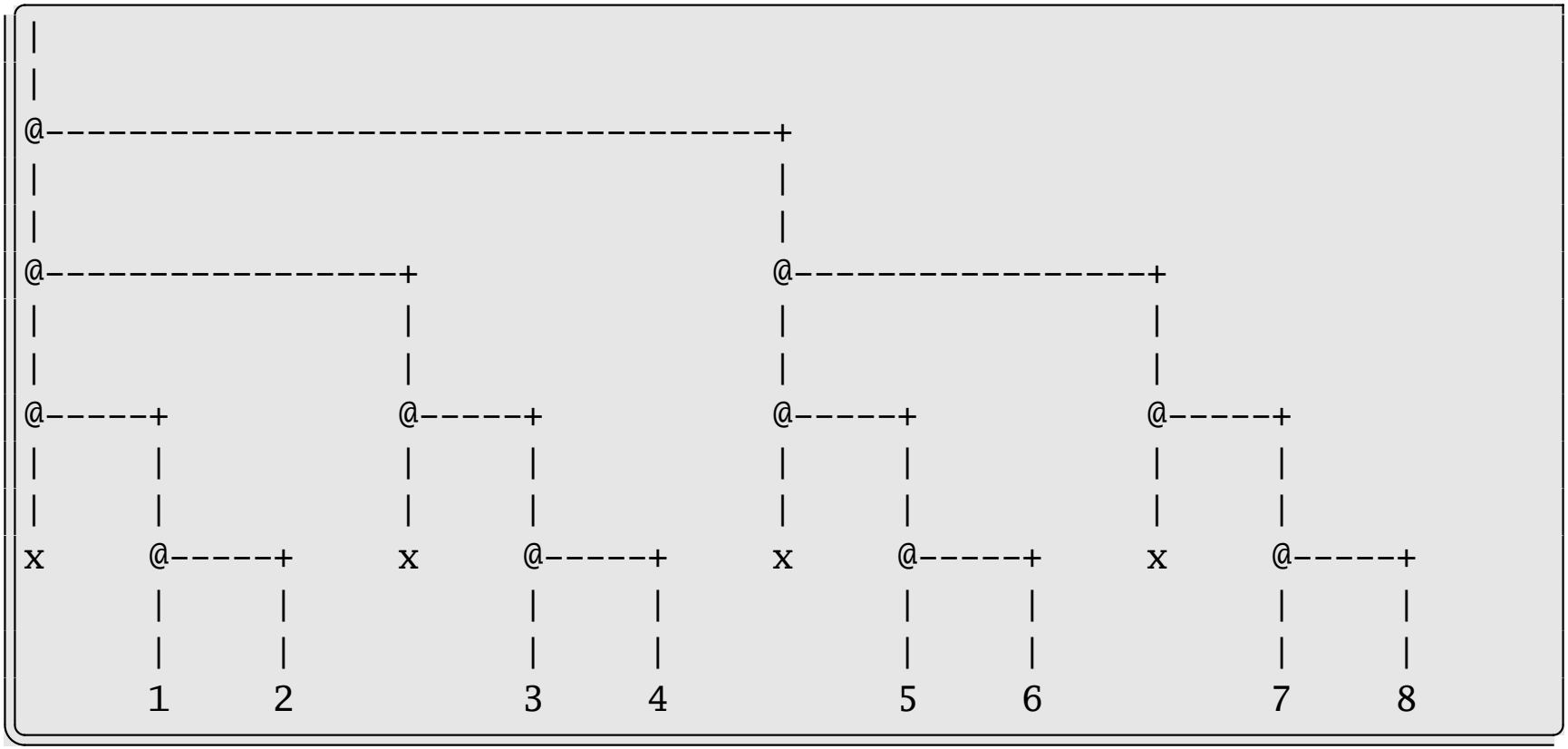
Ile procesów zostanie stworzonych?

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  fork();
  fork();
  if(fork() != 0)
    exit(0);
  fork();

  return 0;
}
```

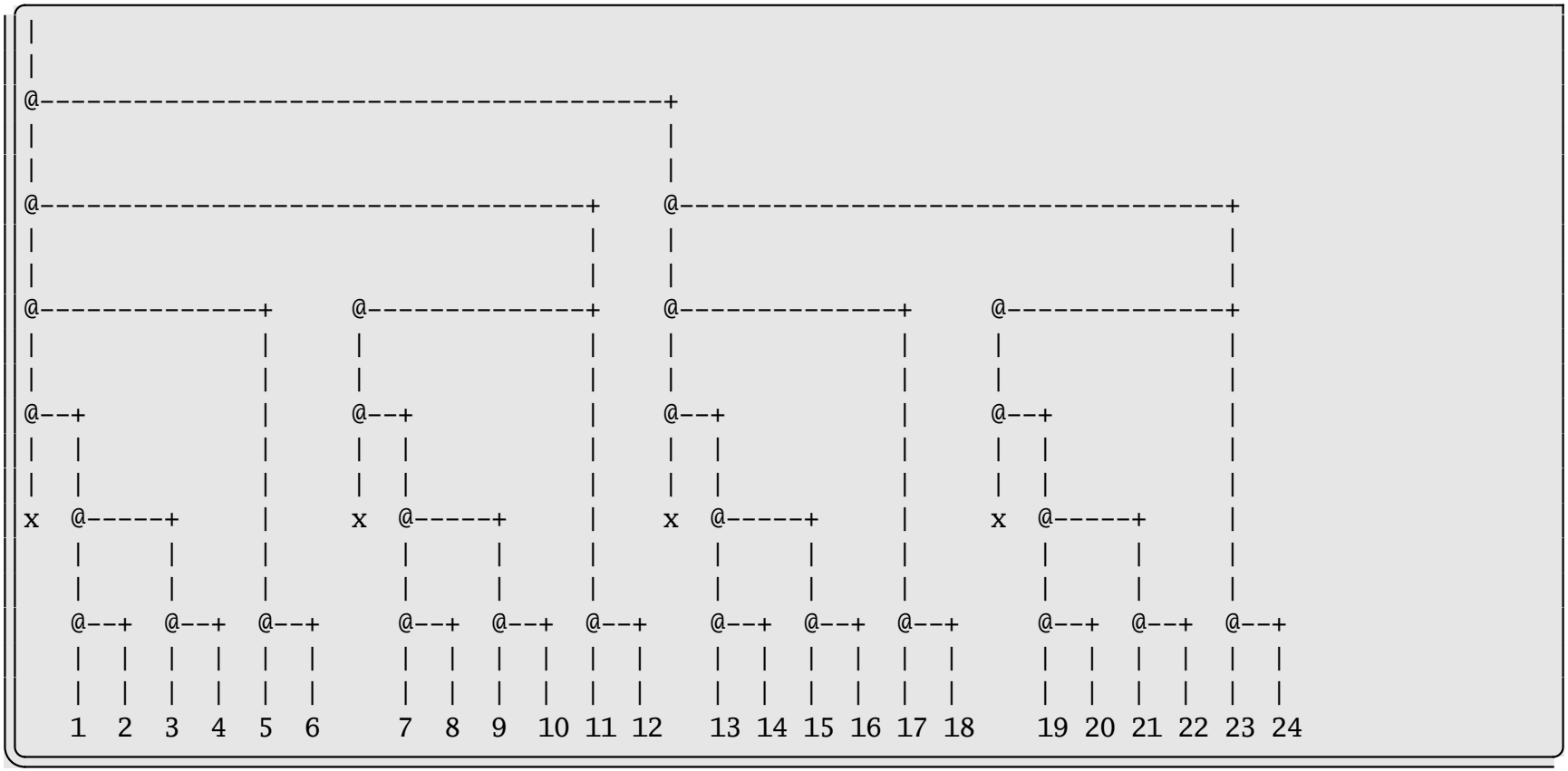
Odpowiedź



Ile procesów zostanie stworzonych?

```
#include <unistd.h>                                    1
#include <stdio.h>                                     2
#include <fcntl.h>                                     3
#include <stdlib.h>                                    4
                                                       5
int main(int argc, char* argv[])                       6
{                                                      7
  fork();                                              8
  fork();                                              9
    if(fork() != 0)                                    10
    if(fork() == 0)                                    11
      fork();                                          12
    else                                               13
      exit(0);                                         14
      fork();                                          15
                                                       16
  return 0;                                            17
}                                                      18
```

Odpowiedź



Zadania praktyczne

Zadania

Zadanie 1

Napisz program tworzący dwa procesy. Każdy ze stworzonych procesów powinien utworzyć proces – potomka. Należy wyświetlać identyfikatory procesów rodziców i potomków po każdym wywołaniu funkcji `fork`.

Zadanie 2

Napisz program wypisujący napis „Poczatek”, następnie wywołujący funkcję `fork` i wypisujący napis „Koniec”. Co jest efektem uruchomienia tego programu?

Zadania

Zadanie 1

Napisz program tworzący dwa procesy. Każdy ze stworzonych procesów powinien utworzyć proces – potomka. Należy wyświetlać identyfikatory procesów rodziców i potomków po każdym wywołaniu funkcji `fork`.

Zadanie 2

Napisz program wypisujący napis „Poczatek”, następnie wywołujący funkcję `fork` i wypisujący napis „Koniec”. Co jest efektem uruchomienia tego programu?

Zadania

Zadanie 3

Napisz program wypisujący napis „Poczatek”, następnie wywołujący funkcję exec powodującą wydruk zawartości bieżącego katalogu i następnie wypisujący napis „Koniec”. Co jest efektem uruchomienia tego programu?

Zadanie 4

Napisz program którego rezultatem będzie wydruk zawartości bieżącego katalogu poprzedzony napisem „Poczatek” a zakończony napisem „Koniec”.

Zadania

Zadanie 3

Napisz program wypisujący napis „Poczatek”, następnie wywołujący funkcję exec powodującą wydruk zawartości bieżącego katalogu i następnie wypisujący napis „Koniec”. Co jest efektem uruchomienia tego programu?

Zadanie 4

Napisz program którego rezultatem będzie wydruk zawartości bieżącego katalogu poprzedzony napisem „Poczatek” a zakończony napisem „Koniec”.

Zadania

Zadanie 5

Napisz program, którego wynikiem jest sformatowana lista procesów:
———poczatek listy————
proces 1
proces2
.....
———koniec listy————

Zadanie 6

Napisz program tworzący równocześnie trzy procesy zombi.

Zadania

Zadanie 5

Napisz program, którego wynikiem jest sformatowana lista procesów:

———poczatek listy————————

proces 1

proces2

.....

———koniec listy————————

Zadanie 6

Napisz program tworzący równocześnie trzy procesy zombi.