Examen Sisteme de Operare

Anul III – 18.01.2021

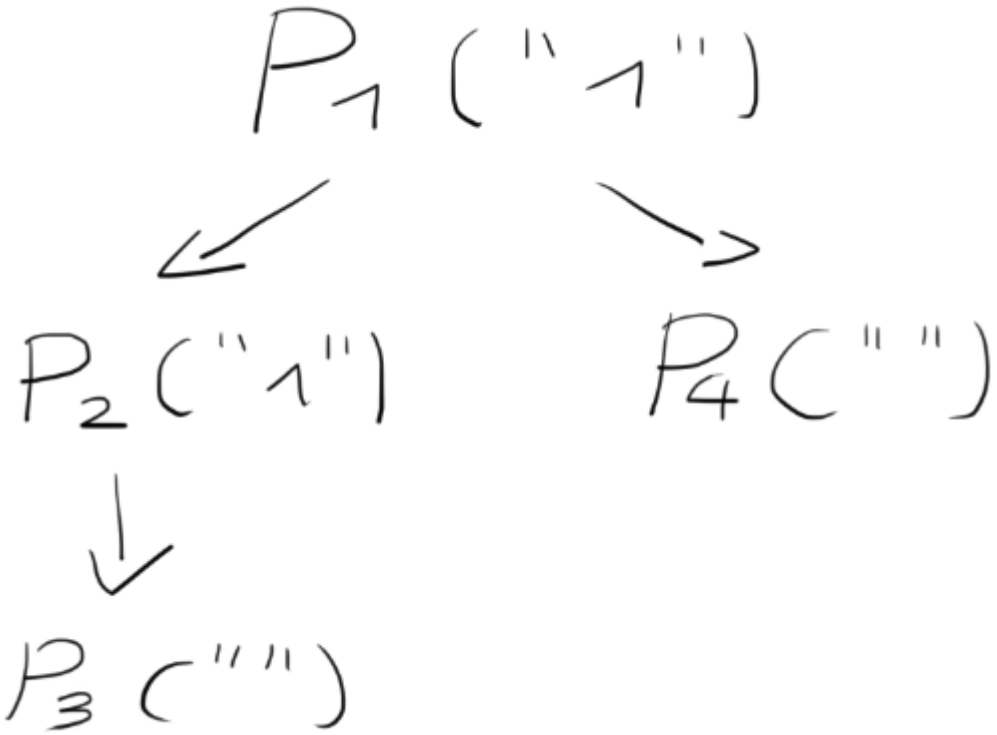
Albăstroiu Dragoș

Grupa 351

Subiectul 1:

În total se vor forma 4 procese (asta incluzând și procesul inițial). Pe terminal se vor afișa cele două caractere: 11

Dependența proceselor este după cum se poate observa:



Primul apel către fork va da naștere procesului P2 din P1 și va închide fd=0 (adică stdin pentru procesul P1) deoarece fork() o să intoarcă PID-ul procesului P2 pentru procesul P1 si 0 pentru P2 și se va intra în corpul lui if pentru P1. În continuare procesele P1 și P2 vor da naștere la procesele P3 și P4. Deoarece pentru P1 și P2 se vor întoarce PID-ul proceselor P3 respectiv P4, valoarea negată a PID-ului va rezulta că P1 și P2 nu vor intra în corpul if-ului pentru executa close(1), adică close de stdout.

În schimb, procesele P3 și P4 vor executa close de stdout. Se execută după în toate procesele menționate dup pe pipe-ul d. Astfel, când se va scrie in 1, în stdout (terminal) vor apărea doar caracterele scrise de procesele P1 și P2. În timp ce P3 și P4 vor scrie în d[0] respective d[1].

Codul modificat:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
   int d[2]; char c='1';
   pipe(d);
   if(fork())close(0);
   wait(NULL); // cod adaugat
   if(!fork())close(1);
   dup(d[0]); dup(d[1]);
   wait(NULL); // cod adaugat
   fprintf(stderr, "PID parinte: %d PID propriu: %d\n", getppid(), getpid()); // cod adaugat
   write(1,&c,sizeof(char));
   return 0;
}
```

Am adăugat două apeluri către wait() pentru a nu se închide părintele înaintea copilului, lucru care ar fi dus la întoarcerea din getppid() a valorii 1. Prin codul de mai sus P1 dă naștere lui P2, P2 lui P3, iar P1 lui P4.

Subiectul 2:

Blocuri de 1KB și adrese de disc de 4 octeți. 10 intrări directe = 10*1KB = 10KB

Indirectare simplă

Număr de blocuri în blocuri indirecte: 1 KB/4 = 256 blocuri

Deci un bloc indirect poate să pointeze la 256 * 1 KB = 256KB

Indirectare dublă

Dublu => 256 * 256 * 1 KB = 65536KB = 64MB

Indirectare triplă

Triplu => 256 * 256 * 256 * 1KB = 16777216KB = 16GB

Dimensiunea maximă a unui fișier = 16777216KB + 65536KB + 256KB + 10KB = 16843018KB ≅ 16.84GB

Subiectul 3:

Ambele procese se vor termina la ora 15:00. Între 12:00 și 12:30 primul process va petrece 50% in IO, adica 15 minute in IO și 15 min procesor. Deci mai are 45 de minute de procesare. După 12:30 vom avea și al doilea proces care va consuma și el 50% din IO. Cele 2 procese vor consuma 50% din IO, deci procesorul cu planificarea Round Robin (va comuta atunci când unul dintre procese este în IO pentru a fi mai eficient) va fi folosit în proces de 75%. (1 – 0.5^2 = 0.75 = 75%)

Pentru a termina primul proces, trebuie să calculăm timpul x necesar, ținem cont de faptul că sunt 2 procese care rulează (deci x se va executa doar 50% din timp):

x * 50% * 75% = 45 min (atât a mai rămas din primul proces)

⇨ x = 120

Rezultă că primul proces se va termina peste 30 + 120 minute de la inceputul lui (ora 12:00, rezultă 14:30)

În aceste 120 de minute, se vor executa tot 45 de minute din procesul 2. Deci mai rămân doar 15 minute din procesul 2.

De la 14:30, pentru a termina cele 15 minute din procesul 2 trebuie să asteptăm:

z * 50% (din IO) = 15 min

⇨ z = 30 min

Rezultă că trebuie să mai așteptăm 30 de minute după ora 14:30 pentru a termina și procesul al doilea.

În final, cele două procese vor fi terminate la 15:00 (primul process terminându-se la 14:30 iar al doilea la 15:00).

Subiectul 4:

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

int copiere(char *numefis, int n) {
 int fd = open(numefis, O_RDWR);
 if (fd<0) return -1;

 int size = lseek(fd, 0, SEEK_END);
 if (size < 0 || n > size) {
  close(fd);
```

```
  return -1;
 }

 char c;
 for (int i=n-1;i>=0;i--)
 {
  if (lseek(fd, i, SEEK_SET)<0 || read(fd, &c, 1) != 1 || lseek(fd, -
n+i, SEEK_END)<0 || write(fd, &c, 1) != 1) {
    close(fd);
    return -1;
  }
 }

 close(fd);
 return size;
}

int main(int argc, char *argv[]) {
 int k;
 if(argc != 3)
  {fprintf(stderr, "Utilizare: %s fisier numar\n", argv[0]); return 1;}
 if((k = copiere(argv[1], atoi(argv[2]))) == -1)
  {perror(argv[1]); return  1;}
 printf("%d\n", k);
 return 0;
}
```

Subiectul 5:

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>

int cauta(char *director) {
  struct passwd *pwd;
  char buffer[256];
  int count = 0;

  if ((pwd = getpwuid(getuid())) == NULL) return -1;

  struct dirent *pde;
  DIR *dir = opendir(director);

  if (dir == NULL) return -1;
```

```
while ((pde = readdir(dir)) != NULL)
{
  if (pde->d_type != DT_REG) continue;

  if (strncmp(pwd->pw_name, pde->d_name, strlen(pwd->pw_name)) != 0) continue;

  struct stat sb;

  strcpy(buffer, director);
  strcat(buffer, "/");
  strcat(buffer, pde->d_name);

  if (stat(buffer, &sb) == 0) {

    if ((sb.st_mode & S_IRUSR) == 0 || (sb.st_mode & S_IWUSR) == 0) continue;

    time_t sys_seconds = time(NULL);
    time_t acc_seconds = sb.st_atim.tv_sec;
    time_t mod_seconds = sb.st_mtim.tv_sec;

    struct tm *timeinfo_sys;
    timeinfo_sys = localtime(&sys_seconds);
    struct tm *timeinfo_acc;
    timeinfo_acc = localtime(&acc_seconds);
    struct tm *timeinfo_mod;
    timeinfo_mod = localtime(&mod_seconds);


    if (timeinfo_sys->tm_year == timeinfo_mod->tm_year) {

      int fd = open(buffer, O_WRONLY | O_APPEND);
      if (fd<0) {
        closedir(dir);
        return -1;
      }
      char *data = asctime(timeinfo_acc);
      if (write(fd, data, strlen(data)) != strlen(data)) {
        close(fd);
        closedir(dir);
        return -1;
      }
      close(fd);
      count++;
    }
  } else {
    closedir(dir);
    return -1;
  }
}
```

```
    closedir(dir);

    return count;
}

int main(int argc, char *argv[]){
  int a;
  if(argc != 2){fprintf(stderr, "Utilizare: %s director\n", argv[0]); return 1;}
  if((a = cauta(argv[1])) == -1) perror ("cauta()");
  else printf("%d\n", a);
  return 0;
}
```

Subiectul 6:

```
#include <signal.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char** argv)
{
    if (argc != 3) {
        fprintf(stderr, "Utilizare: %s prog_1 prog_2\n", argv[0]);
    }

    pid_t prog1, prog2;
    int tub1[2], tub2[2];

    pipe(tub1);
    pipe(tub2);

    prog1 = fork();

    if (prog1 == 0) {
        char* exec_file = argv[1];
        char* exec_argv[] = {exec_file, NULL};

        dup2(tub1[0], 0);
        dup2(tub1[1], 1);
        // close(2); // isi vor inchide descriptorii nefolositi pe tuburi (?), dar nu se va mai afisa n
imic

        execvp(exec_file, exec_argv);

        perror(argv[1]);
        return -1;
    } else {
        prog2 = fork();
```

```
    if (prog2 == 0) {
        char* exec_file = argv[2];
        char* exec_argv[] = {exec_file, NULL};

        dup2(tub2[0], 0);
        dup2(tub2[1], 1);
        // close(2); // isi vor inchide descriptorii nefolositi pe tuburi (?), dar nu se va mai afis
a nimic

        execvp(exec_file, exec_argv);

        perror(argv[2]);
        return -1;
    }
    else {
        int p1_exista = 1;
        int p2_exista = 1;
        int n1;
        int n2;
        while (1) {
            if (p1_exista) {
                waitpid(prog1, NULL, WUNTRACED);
                p1_exista = !kill(prog1, 0);
            }
            if (p2_exista) {
                waitpid(prog2, NULL, WUNTRACED);
                p2_exista = !kill(prog2, 0);
            }
            if (p1_exista && p2_exista) {
                read(tub1[1], &n1, sizeof(int));
                read(tub2[1], &n2, sizeof(int));
                write(tub1[0], &n2, sizeof(int));
                write(tub2[0], &n1, sizeof(int));
                kill(prog1, SIGCONT);
                kill(prog2, SIGCONT);
            } else if (p1_exista && !p2_exista) {
                kill(prog1, SIGCONT);
            } else if (!p1_exista && p2_exista) {
                kill(prog2, SIGCONT);
            }
            else break;
        }
    }
}

close(tub1[0]);
close(tub1[1]);
close(tub2[0]);
close(tub2[1]);
```

```
    return 0;
}
```