6. Praktika

Multiprogramazioa: sistema-deiak eta komandoak

Kontzeptuak

Prozesuen sorrera, amaiera eta sinkronizazioa (bukaerarengatik eta denborarengatik). Atzeko planokoak (konkurrenteak, *background* edo *spawn*) eta aurreko planokoak (sekuentzialak, *run* edo *foreground*).

Enuntziatua

Multiprogramazioaren aukerak ezagutzea eta zenbait ariketa egin eta probatzea makinaren gainean da praktikaren helburua.

Emandako ariketen gain, *jaurti1* programatik abiatuta sistema-deiekin honako ariketa hauek ere egin behar dira:

1.- *jaurti2*: sarrera estandarretik komandoak irakurri eta **konkurrentzian** (*SPAWN*) edo **sekuentzian** (*RUN*) exekutatzen dituen bertsio berri bat egin. Aukera karaktere batez zehaztuko da komandoaren lerro berean, hasierako karaktere bezala. Adibidez:

```
R kopiatu fitx1 fitx2

S kopiatu fitx1 fitx2
```

2.- *jaurti3*: aurreko bertsioa aldatu, ume-prozesu bakoitza hasten denean eta bukatzen denean bere pid-a azal dadin.

3.- *jaurti4*: aurreko bertsioa aldatu komando bakoitzaren lerroan zehazten den denbora -R edo S karakterearen ondoren- kontrola dadin, komandoaren exekuzioa amaiaraziz denbora hori gainditzen denean. Adibidez:

```
R 3 kopiatu fitx1 fitx2

S 4 kopiatu fitx1 fitx2
```

Dokumentazioa

- Klaseko gardenkiak. Liburuaren 5. kapitulua.
- Ariketen enuntziatua eta dokumentazioa.
- UNIX-eko laguntza: man –s2 *sistema-deia*

UNIXeko sistema-deiak (multiprogramaziorako)

Prozesuak sortu, amaitu eta kontrolatzeko:

```
int fork(); /* umearen pid gurasoari eta 0 umeari */

int execlp(char *path, char *arg0, …, char *argn, NULL);
int execvp(char *path, char *argv[]);
                /* execl, execv, execle, execve ere */

void exit(int egoera);

int wait(int *egoera);   /* umearen pid eta egoera jasotzen da.
                            -1 umerik ez badago */

int kill(int pid, int SIGKILL);   /* 2. param. konstantea */
                     /* bestelako erabilpenak (gertaerak) */
```

```
main()
{
   int pid;

   printf("fork deiaren proba\n");
   pid = fork();
   printf("kontrola itzuli zaio ");
   if (pid == 0) printf("umeari\n");
   else printf("gurasoari, %d izanik umearen pid-a\n", pid);
}
```

fork-aren proba

```
int exekutatu_programa(char *izena,
                       char *k0, char *k1, char *k2,
                       char *argv[])
{
   int pid;

   pid = fork();
   if (pid == 0) {
      if (k0 != NULL) {close(0); open(k0, O_RDONLY);}
      if (k1 != NULL) {close(1); open(k1, O_WRONLY);}
      if (k2 != NULL) {close(2); open(k2, O_WRONLY);}
      execvp(izena, argv);
      errore("exec");
   }
   else return(pid);
}
```

fork eta exec-en konbinaketa

Denboraren kontrola:

```
void pause();
unsigned alarm(unsigned segundo_kopurua);
int signal(int seinalea, void *fun());   /* seinaleak*/
unsigned long time(0);
            /* UNIXeko denbora, 1970eko urtarrilaren 1etik */
char *ctime(unsigned t_unix);
            /* liburutegikoa, denbora string bihurtzeko */
```

```
/* denbora.h fitxategia */

void itxaron_denbora(unsigned segundo_kopurua)
{
   signal(SIGALARM, fnulua);
   alarm(segundo_kopurua);
   pause();
}

void fnulua()
{
   return;
}
```

denboraren kontrola (*denbora.h* fitxategia)

Beste sistema-deiak multiprogramaziorako:

```
int getpid();           /* prozesuaren pid-a */
int nice(int balioa);   /* prozesuaren lehentasuna jaitsi */
```

Adibide-programa

Guraso-prozesu batek programa baten exekuzioa abiarazten du, honen izena eta argumentuak bigarren argumentutik pasatzen zaiolarik. Lehen argumentuak programaren exekuzio-denbora maximoa adieraziko du; denbora hori pasa eta programak bukatu ez badu, guraso-prozesuak umearen exekuzioa amaiaraziko du. Guraso-prozesuak bere identifikadorea eta abiarazten duen programarena ere idazten ditu. Programa berriak amaitzean, gurasoak bere iraupena idatzi eta itzulera-kodea itzultzen du, edo -1 balioa programa amaiarazi badu.

Beharrezkoa da, oraingoz, erloju-prozesu bat abiaraztea programaren exekuzio-denbora kontrolatzeko, ezin baita zuzenean *itxaron_denbora* erabili, honek, sinkronoa izatean, guraso-prozesua blokeatuta utziko bailuke umea bukatu ondoren ere.

```
/* gurasoa.c */

main(int argc, char *argv[])
{
   int umeid, erlojuid, id, t1, lag, lag2;

   id = getpid();
   printf("--guraso-prozesua: %d\n", id);
   if ((erlojuid = fork()) == 0)
      execlp("erlojua", "erlojua", argv[1], NULL);
   printf("--erloju-prozesua: %d\n", erlojuid);
   if ((umeid = fork()) == 0) execvp(argv[2], &(argv[2]));
   printf("--ume-prozesua: %d\n", umeid);
   t1 = time(0);
   if ((id = wait(&lag)) == umeid) {   /* amaiera normala */
      kill(erlojuid, SIGKILL);
      wait(&lag2);
   }
   else {   /* bertan behera utzi ume-prozesua*/
      kill(umeid, SIGKILL);
      wait(&lag2);
      printf("--denbora pasata\n");
      lag = -1;
   }
   t1 = time(0) - t1;
   printf("--umearen denbora: %d\n", t1);
   exit(lag);
}
```

```
/* erlojua.c */

#include <stdlib.h>
#include "denbora.h"

main(int argc, char *argv[])
{
   itxaron_denbora(atoi(argv[1]));
}
```

Jaurti1 (KI sinple bat: sarrera estandarretik programen izenak irakurri –lerroko bat- eta modu sekuentzialean exekutatzen ditu. EOF irakurtzen duenean amaitzen du.)

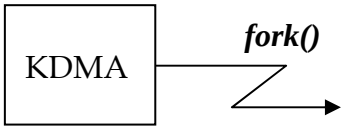
```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define errore(a) {perror(a); exit(1);};
#define BUFSIZE 512
#define MAXARG 10

int lortu_argumentuak(char *buf, int n, char *argk[], int ma);

main(int argc, char *argv[])
{
   int err, n, pid;
   char buf[BUFSIZE];
   char *arg[MAXARG];

   for (n = 0; n < BUFSIZE; n++) buf[n] = '\0';
   /* irakurketa */
   write(1, "Jaurti1> ", 9);
   while ((n = read(0, buf, BUFSIZE)) > 0){
      buf[n] = '\n'; n++;
      err = lortu_argumentuak(buf, n, arg, MAXARG);
      switch (pid = fork()){
         case -1: errore("fork");
                  break;
         case  0: /* umearen kodea */
                  execvp(arg[0], arg);
                  errore("exec");
                  break;
         default: /* aitaren kodea */
                  printf("%d (%s ..) prozesua sortua\n", pid, arg[0]);
                  /* umea bukatu arte itxoitea */
                  if (wait(NULL) != pid) errore("wait");
                  for (n = 0; n < BUFSIZE; n++) buf[n] = '\0';
                  write(1, "Jaurti1> ", 9);
                  break;
      }
   }
}

int lortu_argumentuak(char *buf, int n, char *argk[], int m)
{
   int i, j;

   for (i = 0, j = 0; (i < n) && (j < m); j++) {
      /* zuriuneak pasa */
      while (((buf[i] == ' ') || (buf[i] == '\n')) && (i < n)) i++;
      if (i == n) break;
      argk[j] = &buf[i];
      /* bilatu zuriune karakterea */
      while ((buf[i] != ' ') && (buf[i] != '\n')) i++;
      buf[i++] = '\0';
   }
   argk[j] = NULL;
}
```

ARIKETAK – MULTIPROGRAMAZIOA

1.- Prozesuen lehentasunak honakoak direla suposatu: PA < PB < PC < PD < KI.

Bete ezazu taula (hurrengo bi egoera-aldaketak soilik), honako gertaeren ondoren:

1. PA prozesuak ***fork()*** exekutatzen du, PC prozesua sortuz.
2. Diskoko (alegia, KDMA-ko) eten bat gertatzen da.



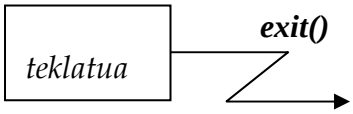
Gertaera	Exekutatzen (Run)	Prest (Ready)	Blokeatuta	Bukatuta
...	PA	PB - Nulua	KI(tekl.) - PD(Diskoa)	

2.- Prozesuen lehentasunak honakoak direla suposatu: PA < PB < PC < KI.

Oharra. PA, PB, eta PC prozesuen aita Komando Interpretatzaile prozesua da (KI).

Bete ezazu taula (hurrengo bi egoera-aldaketak soilik), honako gertaeren ondoren:

1. PA prozesuak bere lana amaitu egiten du, ***exit()*** exekutatuz.
2. Teklatuko eten bat gertatzen da.



Gertaera	Exekutatzen (Run)	Prest (Ready)	Blokeatuta	Bukatuta
...	PA	PB - Nulua	KI(tekl.) - PC(Diskoa)	

3.- Ondoko egoeren trantsizio-taula emanik, eta prozesuen lehentasunak $P_A < P_B < P_D < P_C$ izanik, bete ezazu taula ondoko gertaerak ematen direnean:

1. *run* egoeran dagoen prozesuak ***fork()*** exekutatzen du (P_C prozesua sortuz).
2. *run* egoeran dagoen prozesuak ***fork()*** exekutatzen du (P_D prozesua sortuz).
3. DISKOaren (edo KDMAren) etena heltzen da.

gertaera	exekutatzen (*run*)	prest (*ready*)	Blokeatuta	amaituta
	P_A	nulua	P_B(diskoa)	
fork()				
fork()				
KDMA →				

(bete itzazu bakarrik hiru trantsizioak)

4.- Ondoko bi programa zatietan, guraso prozesu batek eta bere ume prozesuak modu konkurrentean atzitzen dute `datuak` fitxategia:

```
        /* 1. adibidea */
f = open("datuak", O_RDONLY);
if (fork() == 0) { /* umea */
  read(f, buf, 80);
  close(f);
}
else { /* gurasoa */
  read(f, buf, 80);
  close(f);
}
```

```
        / 2. adibidea */
if (fork() == 0) { /* umea */
  f = open("datuak", O_RDONLY);
  read(f, buf, 80);
  close(f);
}
else { /* gurasoa */
  f = open("datuak", O_RDONLY);
  read(f, buf, 80);
  close(f);
```

Zein da bi adibideen arteko desberdintasuna fitxategitik irakurriko den informazioari dagokionez? Iradokizuna: gogoan izan prozesu bakoitzaren kanal-taularen eta UNIXeko Fitxategi Irekien Taularen (FIT) egoerak.

5.- `tty_erloju` izeneko komandoa programatu behar da, pantailak erloju baten portaera izan dezan; hau da, pantailan minuturo ordua (`oo:mm` formatuan) idatz dadin. Horrez gain, teklatuaren oihartzuna galarazi behar da eta sarrerak ez dira aintzakotzat hartuko. Programaren argumentuak, badaude, hasierako ordua eta minutuak izango dira (bestela SEaren ordua hartu ezazu hasierako ordua bezala). Adibidez:

```
$ tty_erloju 10 58
```

```
10:58
10:59
11:00
…
```

6.- Aurreko komandoa erabiliz inplementatu **`tty_erloju_q`** izeneko komando berri bat, lehengoak egiten zuenaz gain **`'q'`** tekla sakatzean programa amaitu eta kontsola modu arruntera itzul dadin.

7.- Unix programa bat denbora-tarte jakin batean exekutatzen den ala ez jakiteko programa bat idatzi behar da. Programaren izena *denbora_proba* da eta erabiltzeko modua honakoa:

`denbora_proba` `exekutagarria saio_kop denb_minimoa denb_maximoa`

Ikus daitekeenez argumentuak lau dira: probatu behar den programa exekutagarria, egin behar den saio-kopurua eta exekutatzeko behar lituzkeen denbora minimoa zein maximoa (segundotan).

Egin nahi dugun aipatutako saio-kopurua (*saio_kop*) modu sekuentzialean burutuko du exekutagarriarekin, eta bukaeran esan beharko du zenbat aldiz exekutatu den denbora-tartean, zenbat aldiz behar izan duen denbora minimo baino gutxiago eta zenbat maximoa baino gehiago. Horretan laguntzeko *erlojua* izeneko programa exekutagarria erabiliko da. Programa honek argumentuan aipatutako segundo-kopurua itxaron ondoren bukatu egiten du besterik gabe:

`erlojua` `denbora`

8.- Ingeniaritza-zentro batean oso programa konplexuak erabiltzen dira. Programa hauen ezaugarriak direla-eta, lasterketa-baldintzak gerta daitezkeela susmoa dagoenez gero, emaitzak ez dira guztiz fidagarriak izango programa batzuetan. Akatsak detektatzeko eta zuzendu ahal izateko utilitate berri bat sortu behar da. Normalean modu honetan exekutatuko litzatekeen komandoa:

```
programa <sarrera_fitx >irteera_fitx
```

utilitate berriarekin beste modu honetan exekutatuko dugu:

`modu_segurua` `programa sarrera_fitx irteera_fitx`

Utilitate hau erabiltzen denean argumentu gisa zehazten den programa bi aldiz eta konkurrentzian exekutatuko da, eta bietako emaitzak berdinak badira emaitza hauek `irteera_fitx` fitxategian idatziko dira. Emaitzak berdinak ez badira ez da fitxategirik sortuko, eta errore-mezu bat idatziko da errore-kanal estandarrean. Emaitzen berdintasuna egiaztatzeko honako programa erabiliko dugu:

`berdinak` `fitx1 fitx2`

fitx1 eta fitx2 konparatzen dituena, berdinak badira 0 bueltatuz amaitzen duelarik, eta -1 bestelakoan.

9.- Aurreko ariketan definitu den **`modu_segurua`** komandoa hobetu nahi da zera lortzeko:

(A) Prozesu baten exekuzioa amaitzen denetik 30 segundo pasatuz gero bestea bukatu ez bada, azken hau amaiarazi eta errorea eman dezan (irteera fitxategirik sortu gabe).

(B) Prozesu baten exekuzioa amaitzen denetik 30 segundo pasatuz gero bestea bukatu ez bada, azken hau amaiarazi eta bi prozesuak berriro abia daitezen (behar beste aldiz errepikatuz).