Contrôle des interactions

Thèmes traités

— schémas d'interaction : scrutation, synchronisation et publier/s'abonner
— API de contrôle des flots d'E/S : `fcntl, select`

1 Interaction entre processus

Situation (abstraite) : 2 processus (un émetteur, un récepteur) doivent échanger une information. Le récepteur ne contrôle pas l'émetteur. Le récepteur doit obtenir l'information produite par le récepteur « dès que possible ».

Deux schémas d'interaction sont possibles

Asynchrone : le récepteur s'exécute indépendamment de l'émetteur. Lorsque l'émission a lieu, le récepteur est (momentanément) interrompu pour traiter le message émis, avant de poursuivre son exécution.

Dans ce schéma d'interaction, le récepteur ne contrôle pas le point de son flot d'exécution où le message sera traité.

C'est le schéma des signaux, ou des interruptions matérielles. On parle de schéma *publier/s'abonner* : le récepteur s'*abonne* (primitive `sigaction(-)`) à la réception de messages *publiés* (primitive `kill(-)`) au rythme de l'émetteur.

Synchrone : le récepteur choisit le point de son exécution où la réception sera traitée. À ce point, il **attend** que le message soit émis et lui parvienne. Cette attente peut être réalisée de deux manières :

— la scrutation (E/S non bloquantes) : le récepteur exécute une boucle consistant à tester si le message a été reçu, jusqu'à la réception effective du message.
Dans le cas de la communication par lecture/écriture de flots d'octets (E/S Unix), ce type d'attente peut être réalisé en positionnant en mode non bloquant (`O_NONBLOCK`) le descripteur associé au flot, grâce à la primitive `fcntl`.

— le blocage (E/S bloquantes) : si le message n'est pas immédiatement disponible, le récepteur est mis en veille. C'est l'émetteur qui provoquera son réveil, au moment de l'émission.
Dans le contexte des entrées-sorties Unix, ce type d'attente est le mode usuel : par défaut les primitives d'accès aux fichiers (`read`, `write`, ...) sont bloquantes.

2 Contrôle des flots d'E/S

planches 19-20 du support « Fichiers »

3 Exercice

Compléter le programme fourni, afin de réaliser l'application suivante, en deux versions :

1. une version avec scrutation et E/S non bloquantes,
2. une version avec attente bloquante.

Dans ce programme, un processus père crée un tube puis un fils. Le **fils** transmet un texte (contenu dans un fichier) **via le tube**. Le père traite le texte (filtrer les voyelles, mettre en majuscules, ne rien afficher...) et affiche (sur la sortie standard) le résultat du traitement en fonction de **commandes reçues sur l'entrée standard** (clavier).

La saisie des commandes est aussi simple que possible : chaque caractère saisi correspondra à une commande, et est pris en compte sans attendre de retour chariot [1]. Les commandes implantées sont 'M' (majuscules), 'm' (minuscules), 'X' (ne rien afficher)', 'R' (rétablir un affichage sans filtre) et 'Q' (quitter).

1. L'appel `system("stty␣-icanon␣min␣1")` en ligne 98 permet de configurer le terminal dans ce mode de saisie.

Programme à compléter

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #include <string.h>
6
7  #include <ctype.h>
8  #include <fcntl.h>
9
10 #define BUFSIZE 512
11
12 void traiter(char tampon [], char cde, int nb) {
13
14   int i;
15
16   /* toupper, tolower, module string.h */
17   switch(cde) {
18     case 'X' :
19       break;
20     case 'Q' :
21       exit(0);
22       break;
23     case 'R' :
24       tampon[nb] = '\0';
25       printf("%s", tampon);
26       break;
27     case 'M' :
28       for (i=0; i < nb; i++) {
29         tampon[i] = toupper(tampon[i]);
30       }
31       tampon[nb] = '\0';
32       printf("%s", tampon);
33       break;
34     case 'm' :
35       for (i=0; i < nb; i++) {
36         tampon[i] = tolower(tampon[i]);
37       }
38       tampon[nb] = '\0';
39       printf("%s", tampon);
40       break;
41     default :
42       printf("????");
43   }
44   return;
45 }
46 int main (int argc, char *argv[]) {
47
48   int p[2];
49   pid_t pid;
50   int d, nlus;
51   char buf[BUFSIZE + 1];
52   char commande = 'R'; /* mode normal */
53
54   if (argc != 2) {
55     printf("utilisation␣:␣%s␣<fichier␣source>\n", argv[0]);
56     exit(1);
57   }
58
59   if (pipe(p) == -1) {
60     perror ("pipe");
61     exit(2);
62   }
63
64   pid = fork();
65   if (pid == -1) {
66     perror ("fork");
67     exit(3);
68   }
69
70
```

```
71    if (pid == 0) {   /* fils  */
72
73      d = open (argv[1], O_RDONLY);
74
75      if (d == -1) {
76        fprintf (stderr, "Impossible␣d'ouvrir␣le␣fichier␣");
77        perror (argv[1]);
78        exit (4);
79      }
80
81      close(p[0]); /* pour finir malgre tout, avec sigpipe */
82
83      while (1) {
84        while ((nlus = read (d, buf, BUFSIZE)) > 0) {
85          /* read peut lire moins que le nombre d'octets demandes, en
86           * particulier lorsque la fin du fichier est atteinte. */
87          write(p[1], buf, nlus);
88          sleep(5);
89        }
90        sleep(5);
91        printf("on␣recommence...\n");
92        lseek(d, (off_t) 0, SEEK_SET);
93      }
94
95    } else {   /* pere */
96      close(p[1]);
97
98      system("stty␣-icanon␣min␣1");
99
100         /* a completer */
101     while (commande != 'Q') {
102
103         /* a completer */
104
105       sleep(1);
106     }
107   }
108   return 0;
109 }
```

4 Testez vous

Vous devriez maintenant être en mesure de répondre clairement aux questions suivantes :

- Peut-on contrôler l'*attente* de données provenant d'une source ? Comment ?
- Peut-on contrôler l'attente de données provenant de *plusieurs* sources ? Comment ?
- En termes de parallélisme, l'interaction asynchrone est elle préférable à l'interaction synchrone ?
- En termes d'efficacité (de consommation de ressources), la scrutation est elle préférable au blocage ?