UE NSY 103
Programmation Système

EXAMEN 2007-2008 – Session Septembre

documents autorisés : Non

Une sonde WEB

L'équipe de développement dans laquelle vous travaillez est chargée de la réalisation d'une petite sonde autonome de surveillance. Cette sonde teste la disponibilité de l'accès aux serveurs, et peut soit émettre une alerte en direction des services centraux, soit répondre à des interrogations du système central. Il s'agit d'un petit boitier qui sera installé sur un site distant, et connecté au réseau local. Ce produit répond à un besoin des services informatiques centraux, quand ceux-ci offrent des solutions complètes d'accès pour des utilisateurs dans des locaux distants (cas de franchises, concessionnaires, succursales, etc : Des établissements très éloignés qui ont besoin d'échanger de l'information avec le système de la maison mère, que ce soit pour l'accès au stock, la remontée d'informations dans le cadre du suivi de la clientèle, de suivi de produit etc).

Très souvent, dans le cas d'établissements de petite taille, les problèmes d'accès (que ce soit un problème de réseau, ou du service) ne sont détectés que lorsque l'utilisateur se rend compte qu'il est « planté ». La panne n'est donc traitée que lorsqu'il est trop tard, à posteriori.

La sonde, placée au plus près de l'utilisateur, va simuler très régulièrement une utilisation standard des ressources offertes (interrogation, réservation puis annulation, etc). En cas de panne du réseau ou de problème d'accès au service, quel qu'il soit, la sonde permettra aux informaticiens du site central une réaction bien plus rapide, avant même que l'utilisateur distant ne se rende compte de l'interruption (Même en cas de rupture du réseau, on est alerté : si on sait que la sonde d'une succursale, prenons Nevers par exemple, doit simuler une interrogation de notre stock toutes les 10 minutes, l'absence de connexion provenant de Nevers sur le serveur de stock durant le dernier quart d'heure sera là encore une indication de rupture de connexion avec ce site).

Le site central pourra interroger régulièrement lui aussi chacune des sondes afin d'obtenir des statistiques sur les pannes et leurs durées, sur les temps de réponse du service, et d'autres statistiques. Elle est donc configurable afin de lui faire réaliser des tests, et offre aussi un accès aux statistiques.

Question 1 : Connexion réseau (8 pts)

-1.a) Sockets [2 points]

Comment est caractérisé un socket réseau ?

-1.b) Gestion de fichiers et sockets[2 points]

Décrire les primitives de base utilisées pour travailler avec les fichiers. Même chose avec les sockets.

-1.c) Explication du code [2 point]

Expliquer ce que fait le code aux endroits indiqués (ANNEXE 1)

-1.d) Écriture du code manquant [2 points]

Compléter le code (ANNEXE1) afin d'envoyer la requête (REQUEST) au serveur et de rendre compte dans le fichier de log (OK_RESPONSE ou ERROR_MESSAGE)

Question 2 : Matériel (4 pts)

-2.a) Choix de matériel [1 points]

La plate forme matérielle utilisée pour la sonde utilise un CPU little-endian (« petit-boutiste »). Expliquez cette notion. Quel peut être l'impact de cette caractéristique ?

-2.b) Explication [3 points]

Décrire et comparer la gestion de la mémoire par segmentation et par pagination.

Question 3 : Processus (8 pts)

Votre sonde tourne sous un système d'exploitation Linux. Ce système est multiprocessus.

-3.a) processus [2 points]

A quoi correspond un processus ? Comment l'identifier précisement ? Que contient-il ? Et comment sont organisés les processus dans un système d'exploitation ?

-3.b) Primitives [2 points]

Quel est le cycle de vie d'un processus ? Quelles sont les primitives systèmes qui permettent de gérer les processus ?

-3.c) Écriture d'un programme multi-tâche[4 points]

Écrivez un programme qui récupère les programmes dont les noms sont passés en arguments sur la ligne de commande, puis les exécute dans des processus fils.

Annexe 1 : Code du programme Test

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define SERVER_PORT 80              /* port ou se connecter            */
#define BUFFER_SIZE 15              /* taille du buffer pour la reponse */
#define OK_MESSAGE "Connexion active" /* message quand tout est okay */
#define ERROR_MESSAGE "Pas de connexion" /* message d'erreur */
#define REQUEST "GET / HTTP/1.0\nHost: pc\n\n" /* la requête HTTP */
#define OK_RESPONSE "HTTP/1.0 200 OK"       /* la réponse HTTP */


int main(int argc,char* argv[]) {
  int                soc;                /* no de socket          */
  struct sockaddr_in sock_in;            /* infos de connexion    */
  struct hostent   * host;               /* adresse IP du serveur */
  int                ncarlu;             /* nombre d'octets lus   */
  char               buffer[BUFFER_SIZE]; /* buffer               */
  int                fd;                 /* fichier en ecriture   */

  /* Creation de la socket (pas de connexion encore) */
  soc = socket(AF_INET, SOCK_STREAM, 0);

  if ( soc == -1 ) { COMMENTEZ ICI
    perror("socket");
    exit(-1);
  }
  /* Remplissage de la structure permettant d'indiquer ou se connecter */

  bzero(&sock_in, sizeof(sock_in)); /* mise a zero de la structure */
  sock_in.sin_family = AF_INET;     /* Type de connexion */
  sock_in.sin_port = htons(SERVER_PORT); /* COMMENTEZ ICI*/

  /* On recupere COMMENTEZ ICI */
  host = gethostbyname(argv[1]);
  if ( host == NULL ) {
    /* ici, on ne peut PAS utiliser perror car la variable contenant       */
    /* l'erreur n'est PAS errno. Donc il faut gerer les erreurs nous-meme. */
    switch(h_errno) {
    case HOST_NOT_FOUND:
      fprintf(stderr, "gethostbyname : Hote inconnu.\n");
      break;
    case NO_ADDRESS:
      fprintf(stderr, "gethostbyname : Pas d'adresse IP associee a ce nom.\n");
      break;
    case NO_RECOVERY:
      fprintf(stderr, "gethostbyname : Erreur fatale du serveur de noms.\n");
      break;
    case TRY_AGAIN:
      fprintf(stderr, "gethostbyname : Erreur temporaire du serveur de noms.\n");
      break;
    } /* switch */
    exit(-1);
  }
  /* On recopie l'adresse IP du serveur dans la structure */
  bcopy(host->h_addr, &sock_in.sin_addr.s_addr, host->h_length);
  /* Connexion */
  if ( connect(soc, (struct sockaddr *)(&sock_in), sizeof(sock_in)) == -1 ) {
    perror("connect");
    exit(-1);
  }

  /* ouverture du fichier de log*/
```

```
  fd=open("/var/log/message",O_WRONLY | O_TRUNC | O_CREAT,"0660");
  if (fd==-1) {
     perror("erreur création fichier");
     exit(-1);
  }

  /* on déclenche la lecture de la page web */
       ECRIRE LE CODE ICI
  return 0;
}
```

Corrigé

1-a)

Les deux principales options sont le mode connecté (TCP, avec mise en place de connexions fiables), et le mode non connecté (UDP, communication via datagrammes).

Un socket est caractérisé par 5 informations : les adresses ip de chaque coté du socket, les ports de chaque coté du socket, et enfin le type de socket (TCP ou UDP). Par exemple, l'ouverture d'une connexion réseau via le système de socket sur le serveur web www.cnam.fr nécessite le socket suivant : adresse ip du serveur du cnam (163.173.128.50), le port du service (80), mon adresse ip (88,254,XX.xxx), un port client (1075), et le type TCP (le protocole HTTP est de type connecté)

1-b)

La gestion des fichiers de bas niveau en C – système utilise principalement 4 primitives :

open(), close(), read() et write()

int open(char * pathname, int flags, mode_t mode)

le pathname, c'est le nom du fichier (trivial)

les flags permettent de décrire le mode d'ouverture (O_RDONLY, O_WRONLY ou O-RDWR). A combiner avec d'autres attributs tels que O_CREAT, O_APPEND ou O_TRUNC). On combine ces attributs avec des OU (|)

Enfin, les modes correspondent aux droits du fichiers.

L'int renvoyé est le file descripteur, un entier qui permet de se référer ensuite à ce fichier.

int close(int fd)

Ferme le fichier; et renvoie 0 ou -1 si erreur.

ssize_t read(int fd, void *buffer, size_t taille)

ssize_t write(int fd, void *buffer, size_t taille)

Un buffer est un espace d'une taille définie arbitrairement qu'on pourra manipuler... On peut par exemple utiliser un tableau de XXX char dans notre cas.

Char buffer[1024]; est un excellent candidat pour ce genre de choses.

Un tableau de char comme celui-ci est en fait un pointeur sur le début de la zone, donc, l'utilisation est assez simple.

Lecture : On indique le nombre de caractère qu'on veut lire (taille). La primitive read() déclenche la lecture et remplit le buffer. Elle renvoie le nombre de caractères lus ou -1 en cas de problème. Le nb de caractères peut être inférieur à la taille demandée, en cas de fin de fichier, ou si on lit des fichiers un peu particuliers. (on gardera bien la valeur de result, si on veut manipuler ensuite le buffer, pour éviter de sortir de l'espace utilisé)

Ecriture : idem, on donne le buffer à écrire, et la taille du buffer.

1c et 1d)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

```
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define SERVER_PORT 80              /* port ou se connecter            */
#define BUFFER_SIZE 15              /* taille du buffer pour la reponse */
#define OK_MESSAGE "Connexion active" /* message quand tout est okay */
#define ERROR_MESSAGE "Pas de connexion" /* message d'erreur */
#define REQUEST "GET / HTTP/1.0\nHost: pc\n\n" /* la requête HTTP */
#define OK_RESPONSE "HTTP/1.0 200 OK"       /* la réponse HTTP */


int main(int argc,char* argv[]) {
  int                soc;                /* no de socket          */
  struct sockaddr_in sock_in;            /* infos de connexion    */
  struct hostent   * host;               /* adresse IP du serveur */
  int                ncarlu;             /* nombre d'octets lus   */
  char               buffer[BUFFER_SIZE]; /* buffer               */
  int                fd;                 /* fichier en ecriture   */

  /* Creation de la socket (pas de connexion encore) */
  soc = socket(AF_INET, SOCK_STREAM, 0);

  if ( soc == -1 ) {
    perror("socket");
    exit(-1);
  }
  /* Remplissage de la structure permettant d'indiquer ou se connecter */

  bzero(&sock_in, sizeof(sock_in)); /* mise a zero de la structure */
  sock_in.sin_family = AF_INET;     /* Type de connexion */
  sock_in.sin_port = htons(SERVER_PORT);
  /* port auquel on veut se connecter */
  /* (avec mise des octets dans le bon ordre) */

  /* On recupere l'adresse du serveur auquel on veut se connecter */
  host = gethostbyname(argv[1]);
  if ( host == NULL ) {
    /* ici, on ne peut PAS utiliser perror car la variable contenant       */
    /* l'erreur n'est PAS errno. Donc il faut gerer les erreurs nous-meme. */
    switch(h_errno) {
    case HOST_NOT_FOUND:
      fprintf(stderr, "gethostbyname : Hote inconnu.\n");
      break;
    case NO_ADDRESS:
      fprintf(stderr, "gethostbyname : Pas d'adresse IP associee a ce nom.\n");
      break;
    case NO_RECOVERY:
      fprintf(stderr, "gethostbyname : Erreur fatale du serveur de noms.\n");
      break;
    case TRY_AGAIN:
      fprintf(stderr, "gethostbyname : Erreur temporaire du serveur de noms.\n");
      break;
    } /* switch */
    exit(-1);
  }
  /* On recopie l'adresse IP du serveur dans la structure */
  bcopy(host->h_addr, &sock_in.sin_addr.s_addr, host->h_length);
  /* Connexion */
  if ( connect(soc, (struct sockaddr *)(&sock_in), sizeof(sock_in)) == -1 ) {
    perror("connect");
    exit(-1);
  }

  /* ouverture du fichier */
  fd=open("/var/log/message",O_WRONLY | O_TRUNC | O_CREAT,"0660");
  if (fd==-1) {
     perror("erreur création fichier");
     exit(-1);
  }

  /* on déclenche la lecture de la page web */
  write(soc,REQUEST,strlen(REQUEST));
  /* lit et affiche la reponse */
  ncarlu = read(soc, buffer, BUFFER_SIZE);
  printf(buffer); /* affichage pour trace */
```

```
  if (strncmp(buffer,OK_RESPONSE,strlen(OK_RESPONSE))==0)
        write(fd,OK_MESSAGE,strlen(OK_MESSAGE));
  else
        write(fd,ERROR_MESSAGE,strlen(ERROR_MESSAGE));

  return 0;
}
```

2-a)

Les architectures des ordinateurs intervenant dans l'échange peuvent différer, notamment sur leurs façons de stocker les mots de plusieurs octets. Dans le cas d'un mot de 32 bits, ceux-ci stokent parfois l'octet de poids fort en fin du mot (on dit que le processeur est **gros-boutiste (big-endian)**), et dans le cas contraire (octet de poids fort stocké en debut), il est appelé **petit-boutiste (little-endian**). Si deux ordinateurs utilisant l'un un petit-boutiste l'autre un gros-boutiste communiquent, il y a fort à parier que l'échange sera altéré par cette différence d'organisation. Il faut donc normaliser l'échange des entiers. On utilisera donc toujours les fonctions *htonl*(), *htons*(), *ntohl*() et *ntohs*() afin de convertir les entiers dans la norme choisie par l'Internet (petit-boutiste).

2-b)

le processus dispose d'un espace d'adressage, protégé, qui lui est réservé. Il y accède via des adresses logiques, que la MMU traduit en adresses réelles. Cette indirection permet de réorganiser le contenu réel de la RAM sans impacter les processus (il suffit simplement de mettre à jour la table de translation d'adresses pour tenir compte des changements). La mémoire va être découpée en zones, afin d'être mieux gérée, et de pouvoir être réorganisée.

Le découpage peut être fait selon deux philosophies : la pagination ou la segmentation.

- La pagination découpe la mémoire en blocs de taille fixe, les pages. L'intérêt du système est que la réorganisation de la RAM est plus simple à réaliser avec des blocs de taille fixe. L'inconvénient est que ce découpage est arbitraire, et ne correspond pas aux besoins logiques du processus. D'où la fragmentation interne
- La segmentation est une division de la RAM en bloc de taille variable, adaptée aux besoins du processus. La problématique est alors la gestion de la réorganisation, puisqu'avec des blocs de taille variables, il est plus difficile d'optimiser l'occupation, des « trous » risquent d'apparaître, on parle alors de fragmentation externe.

3-a)

Un processus est l'image d'un programme en mémoire centrale. Le processus est identifié par un entier qui lui est propre, unique à un instant t. C'est le PID (le process Identifier). Ce PID est donné par le scheduler (l'ordonnanceur de tâches), et permet ensuite de se référer à ce processus spécifique.

- Le processus utilise un espace mémoire qui lui est réservé. Cet espace mémoire contient le code binaire correspondant au programme, et diverses autres zones. Parmi celles-ci, on peut remarquer : la pile (un espace dernier entrée – premier sorti qui contiendra notamment l'empilement des appels aux sous-programmes (que ce soient les primitives systèmes ou nos propres fonctions) avec l'adresse de retour et l'espace pour les valeurs de retour),
- le tas (espace dynamique pour stocker des informations, variables, etc),
- et enfin une zone contenant les variables d'environnement.

Les processus sont organisés selon un arborescence père-fils, c'est à dire un arbre n-aire. La racine de cet arbre est le processus initial (init, processus 1). Chaque processus connaît son père (via le PPID, le PID du Parent). En général, le processus fils hérite des principales caractéristiques du

processus père.

3-b)

Les primitives système offrent la gestion des processus (organisés en arborescence). Les primitives principales sont :

- fork() : pour dédoubler le processus, c'est la création proprement dite d'un fils, identique en tous points à son père (même contenu, même variables et valeurs, même étape).
- wait() : le processus se met en attente (très souvent, il attend un signal de son fils)
- exit() : fin du processus, permet d'envoyer un signal libérateur au père, qui est certainement en attente.
- exec() : recouvrement, le fils souvent doit faire quelque chose de différent du père. Or, après le fork(), le père et le fis sont absolument identique. Exec() et tous ces dérivés permettront au fils de se spécialiser, de se transformer en autre chose, de « recouvrir » son code binaire par un autre code binaire.

3c)

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main(int argc, char **argv) {

	int i; /* pour le suivi du comptage des arguments */

	for (i=1;i<argc;i++) { //Attention, on commence à 1 !!!
		char * cde=argv[i]; // on stocke la commande voulue
		pid_t p=fork();
		if(p==0) {
			//je suis le fils
			printf("je suis le fils chargé de faire %s\n",cde);
			execlp(cde,cde,NULL);
			exit(0); //inutile, mais on est jamais trop prudent
		} else {
			//je suis le père
			printf("%i se charge du travail\n",p);

		}
	}

	return 0;
}
```