Développement d'un outil de monotoring système (comme htop/top)

Présenté par :
Amar HENNI
Salem Aghiles BRAI
Ali RAMOUL

Année universitaire : 2020/2021

List of Figures

List of Tables

Introduction :

Le rapport suivant met en évidence le coté théorique mais aussi pratique du travail réalisé, dans ce cadre notre projet consiste au "Développement d'un outil de monitoring système comme htop ou top". Notre tâche consiste plus précisément à réaliser un outils de monotoring avec un fonctionnement le plus proche possible de l'outil top ou htop (présenté dans le chapitre 1) et cela dans un cadre distribuée.

Avant de mettre en évidance les différentes fonctionalités réaliser par notre application, on doit présenté le contexte du travail demandé, le premier chapitre est ainsi dédier à cela.
Dans second chapitre on parlera des différents moyen mit en place afin d'assurer la réalisation de notre projet, ainsi q'une liste des différentes tâches à réalisées.

On terminera par un 3éme chapitre qui détaillera le fonctionnement de notre application ainsi que les différents tâches réalisé par rapport a celles demandées.

Chapter 1

Généralités

1.1 Définitions:

1.1.1 Processus et thread :

Un processus est un exécutable qu'on a mis en mémoire, qui a accès à l'intégralité de sa propre mémoire et qui exécute des instructions. Il a donc une partie de sa mémoire qui est du code, et une partie qui sont des données.
Le problème d'un processus c'est que c'est un peu lourd, cela implique des recopies de mémoire et surtout cela implique de passer par le système pour communiquer avec d'autre processus. Il existe donc un autre concept nommé thread.
Un thread, c'est un découpage d'un processus. Cela veut dire qu'il utilise exactement la même mémoire qu'un processus, ainsi que les mêmes descripteurs de fichiers. Ce qui diffère d'un thread à l'autre ce sont uniquement les pointeurs d'exécution et de pile. ce qui veut dire que 2 threads peuvent exécuter des bouts de code différents d'un même exécutable et avoir des variables locales différentes de celles des autres threads du processus.

comparaison

	Processus	**Thread**
La communication	La communication entre deux processus est coûteuse et limitée	La communication entre deux threads est moins coûteuse que celle du processus
Multitâche	Le multitâche basé sur les processus permet à un ordinateur d'exécuter deux ou plusieurs programmes simultanément	Le multitâche basé sur les threads permet à un programme unique d'exécuter deux threads ou plus simultanément
Espace d'adressage	Chaque processus a son espace d'adressage distinct	Tous les threads d'un processus partagent le même espace d'adressage que celui d'un processus
Tâche	Les processus sont des tâche lourde	Les threads sont des tâches légères
Exemple	Vous travaillez sur un éditeur de texte, il fait référence à l'exécution d'un processus	Vous imprimez un fichier à partir d'un éditeur de texte tout en travaillant dessus, ce qui ressemble à l'exécution d'un thread dans le processus

Table 1.1: Comparaison entre un processus et un thread

Sous Linux, il est possible d'avoir un aperçu sur l'état des différents processus en cours d'exécutions grâce à la commande **"top"** mais il existe aussi l'utilitaire **"htop"** qui permet de visualiser, et de gérer les processus de manière interactive.

1.1.2 htop/top :

htop est un moniteur système pour les systèmes d'exploitation type Unix très similaire à top, qui fonctionne comme lui en mode Terminal, mais qui dispose d'un environnement en mode texte plus convivial (et coloré) que ce dernier.

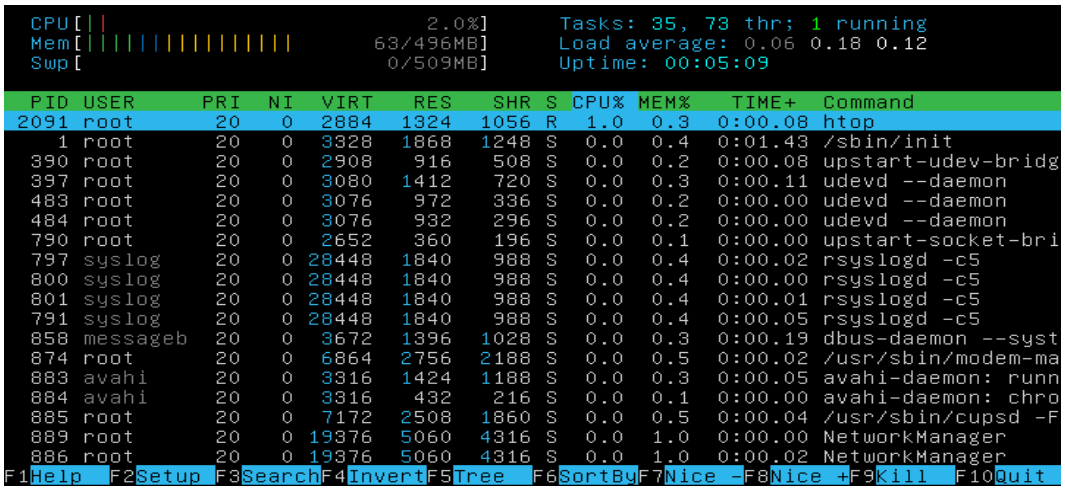

Figure 1.1: Rendu d'execution de la commande htop.

Dans la partie haute de l'interface on peut voir l'utilisation du CPU **(en pourcentage)**, l'utilisation de la mémoire **(en Mo)** par rapport à la totalité de mémoire disponible ainsi que la mémoire swap utilisée. Le temps depuis lequel le système est démarré est indiqué également. Ensuite, un "tableau" listant les processus est présenté où on peut y retrouver une multitude d'information comme l'utilisateur qu'il l'a exécuté, le pourcentage de mémoire et de CPU qu'il utilise ainsi que la commande qui sert à l'exécuter. Pour finir, dans la partie basse un menu explique les différentes actions qui sont disponibles.

Chapter 2

Organisation :

2.1 Outils utilisés :

Afin de mener à bien le travail demandé, tout en patiquand les différents aspect du modules Base du Genie Logiciel vu en Semestre 1, on a opter pour différentes ressources logiciel afin de mener a bien notre travail tout en aquérent une experience profesionnel qui nous sera que bénifique.

- **Git/Github :** Pour la gestion de versionnement.
- **Platforme : Overleaf :** Pour la rédaction du rapport en latex (rédaction simultanée et temps réel).
- **Platforme : Discord/facebook:** Pour la communication temps réel entre les 2 membres du groupe (communiction à distance dû à la crise sanitaire).

2.2 Tâches à réaliser :

Les différentes tâches prise en compte dans notre projets sont les suivante :

- Creation d'un makefile.
- Utilisation de Git pour le versionnement.
- Redaction du rapport.
- Affichage structuré de la sortie standard des relevés des sensors connectés.
- communication entre interfaces.
- Compatibilité en sortie avec un outils de monitoring externe (JSON).
- Monitoring du CPU global et de la mémoire physique.
- GUI enrichi qui donne une éxperience utilisateur plus amiliorée.
- l'interactivité et l'utilisation de plugins.

2.3 Socket :

2.3.1 Généralités:

Les sockets ont été mises au point en 1984, lors de la création des distributions BSD (Berkeley Software Distribution). Apparues pour la première fois dans les systèmes UNIX.

- les sockets sont des points de terminaison mis à l'écoute sur le réseau afin de faire transiter des données logicielles.

- Celles-ci sont associées à un numéro de port. Les ports sont des numéros allant de 0 à 2**16 - 1 inclus (soit 65535 :p). Chacun de ces ports est associé à une application (à savoir que les 1024 premiers ports sont réservés à des utilisations bien précises). Les sockets sont aussi associées à un protocole ; tels que UDP/IP TCP/IP. Dans notre cas nous utiliserons le protocole TCP/IP. Les sockets servent à établir une transmission de flux de données (octets de flux de données (octets) entre deux) entre deux machines ou applications. machines ou applications.

2.3.2 Utilisation:

En C, on a souvent besoin d'un moyen de communication entre deux programmes, dans ce cas on utilise parfois des fichiers qui servent de "passerelle" mais on passe souvent à côté des sockets qui peuvent le faire aussi bien ou même mieux encore.
Le principal atout des sockets est que les informations sont transmises directement au programme voulu en plus d'être plus sécurisées que les fichiers. Par exemple, le langage PHP illustre très bien les sockets, car il utilise ce principe "Client / Serveur".

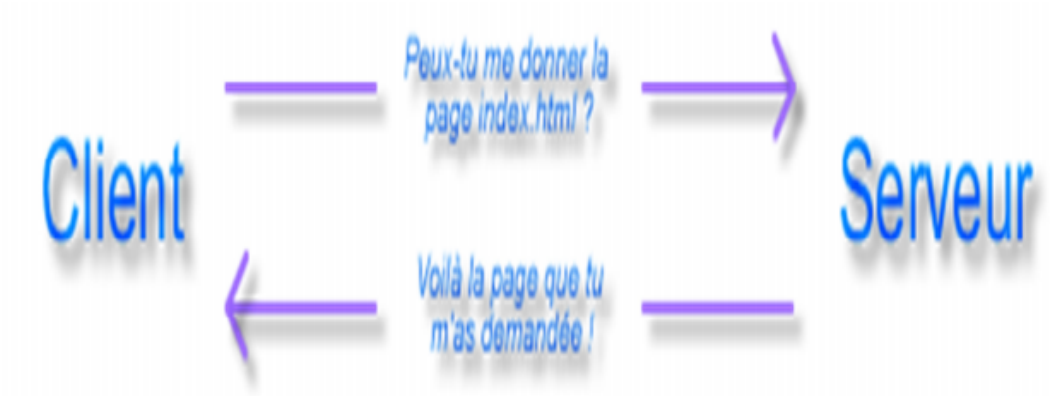

Figure 2.1: Exemple Socket.

Ce schéma est TRES simplifié ! Il n'est pas complet, mais permet d'avoir unevision simplifiée du principe. Elles servent aussi pour tout ce qui touche au réseau.

2.3.3 Leurs fonctionnements:

Les sockets ne s'utilisent pas de manière identique selon les différents systèmes d'exploitation :selon que l'on est sous windows ou sous linux.

Sur Windows :

Tout d'abord, il faut jamais oublier que dans chaque projet on crée , on doit ajouter le fichier "ws2_32.lib" (pour le compilateur Visual C++) ou "libws2_32.a" (pour les autres) dans notre éditeur de liens. on trouve ce fichier dans le dossier "lib" de notre IDE.

Il faut savoir que presque tout ce qui touche aux sockets Windows se trouve dans le fichier "winsock2.h", dans le dossier header de l'IDE. Celui-ci est un fichier standard de Windows, il n'y a pas besoin de le télécharger. Nous allons donc tout de suite l'inclure dans notre premier programme comme suit :

```
1 #include <winsock2.h> \\
```

En général, nous aurons besoin des fichiers standards "stdio.h" et "stdlib.h". Nous allons donc aussi les inclure :

```
\begin{lstlisting}[language=C, caption= Code qui affiche les informations des
    processus]
#include <winsock2.h>\\
#include <stdio.h>\\
#include <stdlib.h>\\
```

On peut remarquer que le type socklen_t qui existe sous Linux, n'est pas défini sous Windows. Ce type sert à stocker la taille d'une structure de type sockaddr_in. Il va donc falloir le définir nous même à l'aide du mot clef typedef comme il suit :

```
typedef int socklen\_t; \\
```

De plus, nous devrons ajouter, dans le début de votre fonction main, le code suivant pour pouvoir utiliser les sockets sous Windows :

```
WSADATA WSAData;\\
WSAStartup(MAKEWORD(2,2), &WSAData);\\
```

La fonction WSAStartup sert à initialiser la bibliothèque WinSock. La macro MAKEWORD transforme les deux entiers (d'un octet) qui lui sont passés en paramètres en un seul entier (de 2 octets) qu'elle retourne. Cet entier sert à renseigner la bibliothèque sur la version que l'utilisateur souhaite utiliser (ici la version 2,0). Elle retourne la valeur 0 si tout s'est bien passé. Puis à la fin, placez celui-ci :

```
WSACleanup(); \\
```

Cette fonction va simplement libérer les ressources allouées par la fonction:

```
WSAStartup();
```

Sur Linux:

Sur Linux, c'est un peu différent puisque les fichiers à inclure ne sont pas les mêmes
Pour combler l'écart entre Windows et Linux, nous utiliserons des définitions et des typedef.

Commençons par inclure les fichiers nécessaires :

```
#include <sys/types.h>\\
#include <sys/socket.h>\\
#include <netinet/in.h>\\
#include <arpa/inet.h> \\
#include <unistd.h>\\
#include <stdio.h>\\
#include <stdlib.h> \\
```

Un premier problème se pose :

Dans le fichier "socket.h" de Linux, la fonction qui sert à fermer une socket (que nous verrons par la suite) se nomme close alors que dans le fichier "winsock2.h" de Windows la fonction se nomme closesocket ... Pour éviter de faire deux codes sources pour deux OS différents, nous utiliserons une définition comme il suit :

```
#define closesocket(param) close(param)\\
```

Ainsi dans le code la fonction closesocket() sera remplacée par la fonction close() qui pourra ensuite être exécutée.
Le deuxième problème vient du fait qu'il "manque" deux définitions et trois typedef qui peuvent

nous être utile dans le fichier "socket.h" de Linux par rapport au fichier "winsock2.h" de Windows. Voila donc le contenu de notre fichier pour le moment :

```
#include <sys/types.h>\\
#include <sys/socket.h>\\
#include <netinet/in.h>\\
#include <arpa/inet.h>\\
#include <unistd.h>\\
#include <stdio.h>\\
#include <stdlib.h>\\
#define INVALID_SOCKET -1\\
#define SOCKET_ERROR -1\\
#define closesocket(param) close(param)\\
typedef int SOCKET;\\
typedef struct sockaddr_in SOCKADDR_IN;\\
typedef struct sockaddr SOCKADDR; \\
```

il y a beaucoup de fichiers à inclure par rapport à Windows mais qu'ils sont tous utiles.

Un code portable :

Pour pouvoir avoir un code un peu plus portable, nous utiliserons les définitions WIN32 et linux. Cette méthode indiquera à le compilateur le code à compiler en fonction de l'OS.

//Si nous sommes sous Windows

```
#if defined (WIN32) \\
#include <winsock2.h>\\
```

// typedef, qui nous serviront par la suite

```
typedef int socklen_t;\\
```

// Sinon, si nous sommes sous Linux

```
#elif defined (linux)\\
#include <sys/types.h>\\
#include <sys/socket.h>\\
#include <netinet/in.h>\\
#include <arpa/inet.h>\\
#include <unistd.h>\\

```

// Define, qui nous serviront par la suite

```
#define INVALID_SOCKET -1\\
#define SOCKET_ERROR -1\\
#define closesocket(s) close (s)\\

```

// De même

```
typedef int SOCKET;\\
typedef struct sockaddr_in SOCKADDR_IN;\\
typedef struct sockaddr SOCKADDR; \\
#endif\\

```

// On inclut les fichiers standards

```
#include <stdio.h>\\
#include <stdlib.h>\\
int main(void)\\
{\\

```

// Si la plateforme est Windows

```
#if defined (WIN32)\\
WSADATA WSAData;\\
WSAStartup(MAKEWORD(2,2), &WSAData);\\
#endif\\

```

// ICI on mettra notre code sur les sockets
// Si la plateforme est Windows

```
#if defined (WIN32)\\
WSACleanup();\\
#endif\\
return EXIT_SUCCESS;\\
} \\
```

2.4 Lecture des informations des processus en cours d'executions:

Pour en savoir un peu plus sur les processus sur un système Linux et pour essayer d'écrire du code en C, nous avons écrit une version de readproc adapté à nos besoins avec les différents flag nécessaires afin de récupérer les informations des différents processus.

Afin d'utiliser readproc on doit installer d'abord la bibliothèque **libproc**.

Sur Ubuntu:

- sudo apt-get update
- sudo apt-get install libproc-dev

Sur Manjaro:

- sudo pacman -Syyu
- sudo pacman install procps-ng

Code :

```
while(1){
   PROCTAB* proc = openproc(PROC_FILLARG | PROC_FILLSTAT | PROC_FILLMEM |
    PROC_FILLSTATUS | PROC_FILLUSR);

   proc_t proc_info;
   memset(&proc_info, 0, sizeof(proc_info));

   printf("PID \t PPID \t USER \t PR \t NI \t VIRT \t Etat \t\n\n");

  while (readproc(proc, &proc_info) != NULL) {

    printf("%-10d \t %-10d \t %s \t %ld \t  %ld \t %lu  \t %c \t", proc_info.tid,
    proc_info.ppid,proc_info.euser, proc_info.priority,proc_info.nice, proc_info.
    vm_size, proc_info.state);
    if (proc_info.cmdline != NULL) {

      printf("%s\n", *proc_info.cmdline);
      jsonout(proc_info.tid, proc_info.ppid, proc_info.priority,proc_info.nice,
    proc_info.vm_size, proc_info.state);

      } else {

       printf("[%s]\n", proc_info.cmd);
       jsonout(proc_info.tid, proc_info.ppid, proc_info.priority,proc_info.nice,
    proc_info.vm_size, proc_info.state);
```

```
21
22      }
23  }
24   closeproc(proc);
25   sleep(3);
26  }
```

Listing 2.1: Code qui affiche les informations des processus

La fonction **openproc** permet de specifier les différents flags qui permetterons d'obtenir les informations des processus. par exemple les flags :

- PROC_FILLARG et PROC_FILLSTAT permettent de définir le chemin d'executions de chaque processus.
- PROC_FILLMEM permet d'obtenir les information liée à la mémoire utilisée par chaque processus.
- PROC_FILLSTATUS permet d'obtenir des informations liée au status de chaque processus.
- PROC_FILLUSR permet d'obtenir des informations liée à l'entité qui à lancer les différents processus (root / autres utilisateurs).

2.5 Visualisation des données:

Afin de mieux visualisé les données lidée était de les mettre sous format JSON afin de construire des graphiques pour mieux visualisés l'état des processus.

On a donc crée une fonction jsonout qui crée un fichier JSON avec comme clé le PID des processus et leurs valeurs correspond au différentes valeurs liée à chaque processus, comme leurs PPID, quel utilisateur les a lancé ...

Code:

```
1  void jsonout (int pid, int ppid,long priority, long nice, unsigned long vm_size,char
       state){
2
3      FILE *fd = fopen("info.json","a+");
4      if(fd == NULL){
5        perror("fopen");
6      }
7
8      fprintf (fd, "{\n \t\"%ld\" : {\n",pid);    /* print header to file */
9      fprintf (fd, "\t \t\"Ppid\" : \" %ld \",\n",ppid);
10     fprintf (fd, "\t \t\"Priority\" : \" %ld \" \n",priority);
11     fprintf (fd, "\t \t\"Nice\" : \" %ld \" \n",nice);
12     fprintf (fd, "\t \t\"Vm_size\" : \" %lu \" \n",vm_size);
13     fprintf (fd, "\t \t\"State\" : \" %c \" \n",state);
14     fprintf (fd, "\t \t }\n}\n\n");     /* print closing tag */
15     fclose (fd);
16 }
```

Listing 2.2: Code qui structure les informations sous format JSON.

2.6 Compilation:

Création d'un makefile qui permet de complier le programme en appelant la commande "make".

```
CC = gcc

all: proc serveur client exeproc exeserveur

proc: proc.c
	$(CC) -o proc.exe proc.c -lprocps


exeproc:
	./proc.exe

exeserveur:
	./serveur.exe


serveur: serveur.c
	$(CC) -o serveur.exe serveur.c

client: client.c
	$(CC) -o client.exe client.c


clean:
	rm -f *.o client serveur proc

.PHONY: clean
```

Figure 2.2: Makefile de projet.