Rapport de projet de LP25

Imadeddine BETKA (TC3), Abel HALLER (TC3), Basile Kone-Boussemart (TC03)

Automne 2025

Contexte du projet

Le projet consiste à concevoir un **programme de gestion de processus pour systèmes Linux**, capable de fonctionner aussi bien **en local que sur des hôtes distants**. L'objectif est de fournir une interface interactive, inspirée de l'outil htop, permettant :

d'afficher dynamiquement la liste des processus actifs sur une ou plusieurs machines,
de visualiser des informations détaillées (PID, utilisateur, consommation CPU/mémoire, temps d'exécution, etc.),
et d'interagir directement avec les processus (mise en pause, arrêt, reprise, redémarrage). L'outil reposera sur des mécanismes standards de communication et d'administration à distance (SSH, etc.), compatibilité avec les principales distributions Linux.

État des lieux des compétences du groupe

Imadeddine BETKA :
Avant le module de LP25, j'avais des bases en C vu en IF2/3 l'année dernière, comme les définitions, des boucles, des fonctions et procédures. En revanche, j'ai eu du mal avec des fichiers, la partie mémoire avec les pointeurs et l'organisation de projet. Je m'attendais donc à des difficultés sur ces points.

Abel HALLER:
Avant LP25, j'ai également suivi l'UE IF2A-B dans laquelle j'ai pu consolider mes bases dans la compréhension du langage C. J'ai notamment travaillé sur un projet (mini-jeu) qui m'a permis de travailler la modularité entre différents fichiers, la compilation Makefile et toutes les autres notions du langage C (fonctions, pointeurs...).
Le langage C ne représentait donc pas une appréhension majeure dans ce projet car les TD,TP m'ont permis de me refamiliariser avec la programmation C. En revanche, j'avais plus de difficulté concernant l'environnement LINUX et l'aspect machine car ce sont des notions que j'ai eu du mal à comprendre en première partie d'UE.
Enfin, j'avais de l'appréhension pour la partie réseaux du projet car nous n'avons pas vraiment abordé ces notions durant l'UE (ssh, telnet).

Basile KONE-BOUSSEMART
Avant LP25, j'avais les connaissances de base des l'ue IF (IF2A, IF2B). Cependant en dehors de celà je n'avais aucune connaissance pouvant servir à ce projet. J'ai donc eu beaucoup d'appréhension quand aux parties sur l'interface graphique et la partie réseau

Répartition des tâches : planification

Dès le départ, nous avons procédé à une étude structurée du projet. Nous avons rapidement pu séparer les 5 étapes majeures de ce dernier: **manager** (le main du projet),**options** (la gestion des options entré par l'utilisateur) **process** (la partie de récupération, puis traitement des processus), **ui** (la gestion des interactions hommes/machines et l'interface) et enfin **network** (gestion de la connexion et de la manipulation des machines distantes).

Nous avons donc décidé de séparer les 4 modules principaux entre chaque membre du groupe puis de se coordonner ensemble sur la réalisation du main.

Imadeddine BETKA :
Responsable de la partie **Processus (**Récupération, recherche...).

Abel HALLER:
Responsable de la création du **GitHub**, de la création de **l'architecture du projet** et de la réalisation de **la partie Network** et de **la gestion des options** (création de la listes des machines distantes puis connexion à ces dernières).

Basile KONE-BOUSSEMART :
Responsable de la partie interactions hommes/machines et de l'interface.

Répartition des tâches : réalisation

Partie Process réalisé sur 10 jours par Imad (et Abel)

Lecture des processus: 1ère version avec /proc (3 jours)
L'objectif de ce module était de récupérer toutes les informations nécessaires sur les processus en cours d'exécution.
Pour cela j'ai utilisé le répertoire spécial /proc ou chaque dossier numérique correspond à un PID.

La fonction `read_processus()` parcours /proc, détecte uniquement les répertoires numériques (Vrais processus) , puis crée une structure Process pour chacun. Suivant ça, `read_status()` est appelé pour remplir les champs (nom,état,mémoire,utilisateur), et chaque processus est ensuite inséré dans une liste chaînée.
La fonction `read_status()` lit le fichier /proc/[pid]/status et extrait :
- Name -> commande de processus
- State –> état (R,S,Z...)
- VmRSS -> mémoire utilisé en KB
- UID -> utilisateur propriétaire (converti ensuite en nom avec `getpwid()`)

Cette étape m'a permis de comprendre concrètement comment Linux expose les informations système.

Lecture des processus: 2ère version avec la commande ps (1 jours) Par Abel

Après le début du développement de la partie network et quelques discussions avec l'équipe, il fallut changer la méthode d'extraction des processus car il n'était pas possible d'utiliser les fonctions `read_processus()` et `read_status()` sur les machines distantes.

Dans la version finale, nous avons donc remplacées ces fonctions par la fonction `read_proc()` qui n'utilise plus `/proc` mais le résultat de commande "`ps -eo pid,user,rss,pmem,state,comm --no-headers`" grâce à 3 fonctions: popen() pour le local , ssh_exec() ou telnet_exec() sinon.
La resultat de commande est analysé ligne par ligne chaqu'un passée avec `sscanf()` pour remplir la structure process.

Calcul de la consommation mémoire (%MEM)-(1 jour)

Pour afficher la mémoire en pourcentage, j'ai d'abord récupéré la RAM totale dans /proc/meminfo.
Ensuite , `update_mem_percentage()` calcule pour chaque processus !

$$\%MEM = (MemTotal / VmRSS) \times 100$$

Cela permet ensuite de classer les processus selon leur consommation réelle.

De même que pour la lecture des processus, dans le version finale de process.c, nous n'avions plus besoin de cette fonction car le changement d'approche avec "ps" permet de récupérer directement les consommations en pourcentage..

Tri des processus par mémoire(1 jour)

La fonction `sort_by_mem()` trie la liste chaînée par ordre décroissant de %MEM.
Le tri par insertion a demandé une bonne gestion des pointeurs qui a causé quelques problèmes pendant le codage avec les erreurs d'allocation et pour garder la liste cohérente.

Gestion des signaux (2 jours)

J'ai implémenté plusieurs fonctions pour contrôler les processus :
`kill_process_soft()` -> arréte propre (SIGTERM)
`kill_process_hard()` -> arréte forcé (SIGKILL)
`pause_process()` -> mise en pause (SIGSTOP)
`continue_process()` -> reprise (SIGCONT)

Une fonction interne send_signal_to_pid() factorise l'envoi de signal et gère les erreurs

Filtrage des processus (recherche) (2 jours)

La fonction `processus_recherche()` permet de filtrer selon une chaîne entrée par l'utilisateur:
-Nom du programme.
-Utilisateur
-PID
La recherche est insensible à la casse, ce qui rend l'outil plus pratique.

Partie UI réalisé sur 6 jours par Basile

1ere version de l'affichage (5 heures)

La première version de l'affichage était une version où les processus étaient affichés dans le terminal et ce dernier était rafraîchi toutes les 5 secondes. Cependant après discussion avec le reste du groupe nous avons décidé qu'il valait mieux réaliser un affichage dans une fenêtre alternative afin de faciliter la lecture

Récupération des processus (3h)

Afin de pouvoir interagir avec les processus il est nécessaire de récupérer celui que l'on sélectionne. pour celà j'ai utilisé une fonction,
"`get_nth_process()`"
, qui permet de parcourir la liste chaînée des processus, les compter et de retourner le processus dont l'index correspond

initialisation/arrêt de ncurses (2h)

j'ai simplement fait en sorte d'initialiser et de pouvoir arrêter ncurses avec plusieurs fonctions :

`ui_init()` qui contient :
`initscr()` -> démarre ncurses
`cbreak()` -> lecture immédiate du clavier
`noecho()` -> ne pas afficher les touches tapées
`keypad(stdscr, TRUE)` -> activer les touches spéciales (flèches)
`curs_set(0)` -> cacher le curseur

`ui_shutdown()` qui contient :
`endwin()` -> restaure le terminal proprement

Affichage des aides (2h)

Après consultation du groupe nous nous sommes dit qu'il serait intéressant d'avoir une fonction qui indique à l'utilisateur les différentes options du programme et comment les utiliser.
j'ai donc réalisé la fonction " `affiche_aide()`" qui crée une nouvelle fenêtre avec les aides.

Affichage des différents processus (5h)

L'une des grosses parties de mon travail a été l'affichage des différents processus notamment afin d'avoir une interface qui est complète mais reste claire et n'est pas surchargée d'informations afin de faciliter la lecture par l'utilisateur.
Pour cela j'ai réaliser une fonction "`affichePrinc(Process *list, int selected)`"
Cette dernière est découpée en plusieurs parties :

- Définition des codes de couleur de l'affichage :

```
init_pair(1, COLOR_CYAN, -1);    // titre
init_pair(2, COLOR_YELLOW, -1);  // noms de colonnes
init_pair(3, COLOR_GREEN, -1);   // première machine
init_pair(4, COLOR_MAGENTA, -1); // ligne d'aide
init_pair(5, COLOR_BLUE, -1);    // titres processus
```

- Affichage de l'entête du programme :
  - titre du programme
  - tableau de connexion
  - ligne d'aide clavier

- Liste des processus :
  - parcours de la liste des processus
  - mise en surbrillance du processus sélectionné
  - scroll vertical
  - adaptation du la largeur du terminal

L'affichage des différents éléments se fait avec la fonction `mvprintf()`

Gestion du clavier (4 heures)

Afin de gérer les actions de l'utilisateur j'ai créé une fonction "`ui_traite_event()`" qui effectue différentes actions en fonction des touches pressées par l'utilisateur :

- ↑↓ : déplacement
- **q** : quitter le programme
- **F1** : afficher la fenêtre d'aide
- **F2** : passage à la page précédente
- **F3** : passage à la page suivante

- **F4 :** rechercher un processus via son nom ou son pid
- **F5** : mettre en pause le processus (avec `SIGSTOP`)
- **F6** : kill le processus (de manière douce avec `SIGTERM`)
- **F7** : kill le processus (de manière dure avec `SIGKILL`)
- **F8** : reprise du processus après mise en pause (avec `SIGCONT`)

J'ai aussi introduit d'un état global de l'interface, avec "`ui_state`" qui permet de connaître l'état de l'interface (Normal/recherche). Grâce à cette fonction on peut bloquer bien différencier les entrées clavier qui ont pour but de se déplacer dans l'interface et celles qui sont destinées a recherché un processus.

Adaptation de l'ui au network (2 jour)

Après le développement de la partie Network, nous avons décidé de changer certains aspect du fonctionnement de l'interface::

- Changement de la gestion du temps dans le programme via l'ajout de `timeout()` afin de rafraîchir l'écran même si aucune touche n'est pressée (avant cela, le rafraichissement de la page était bloqué par `getch()`)
- Gestion du redimensionnement du terminal via "`handle_resize()`" qui permet d'adapter l'affichage du terminal quand sa taille change
- Gestion des machines distantes avec notamment une pagination (avec les touches **F2/F3**)
- Amélioration de la sécurité, via l'ajout de nouvelles vérification pour éviter des erreurs.

Gestion des Options réalisé sur 2 jours par Abel

Initialisation des options (1 heures)

Pour gérer les options de la ligne de commande entrée par l'utilisateur, nous avons commencé par la création d'une structure `program_options` qui contient pour chaque option, une variable associé (de type `bool`, `int` ou `char*`). Une fonction permet d'initialiser ces options en leur donnant des valeurs par défaut.

Analyse des arguments de la ligne de commande (3 heure)

Le traitement des arguments est réalisé dans la fonction `traiter_options()`, qui s'appuie sur la fonction standard `getopt_long()` que nous avons étudiée en cours. Ce choix nous permet de gérer les entrée d'options longues (--login...) et courtes (-l)
Ainsi, chaque options est reconnue et permet de mettre à jour la structure `program_options` initialisé plus tôt (ex: passer l'options help à true ou assigner une adresse IP a l'options serveur...).

Validation de la cohérence des options (5 heure)

Enfin, nous avons mis en place la fonction `valider_options()` qui permet de traiter la cohérence des différentes options. Par exemple, le fait que l'option `--all` nécessite que `--remote-config` ou `--remote-server` soit renseigné ou encore que l'options `--remote-server` nécessite que `--username` et `--password` soit renseignée.
cette tâche à mis plus de temps car il a fallu bien réfléchir à toute les possibilité que l'utilisateur pouvait entrer en option

Libération mémoire (30 min)

Pour éviter toute fuite mémoire, nous avons mis en place une fonction `free_options()` qui permet de libérer la mémoire alloué par la structure **program_options**

Partie Network réalisé sur 1 semaine par Abel

Gestion de la liste des machines (6heure)

Pour pouvoir avoir un visuel global sur l'ensemble des machines que le programme doit prendre en charge, nous avons réalisé une structure `remote_machine` qui contient tous les éléments caractéristique d'une machine (name, username, password, connexion type, port, server...).

Nous avons ensuite mis en place une série de fonctions qui permet de créer la liste des machines:

- `lire_config()`**:** Lit le fichier .config renseigné par l'utilisateur avec l'option `--remote-config` en séparant chaque lignes du fichier écrites sous la forme "", puis ajoute chaque machine à la liste des machines.
- `ajouter_machine_utilisateur()`**:** permet d'ajouter la machine distante renseignée par l'utilisateur avec les autres options (`--remote-server`,`--username`…)
- `ajouter_machine_local()`**:** Permet d'ajouter la machine sur laquelle le programme est lancée à la liste des machines. Cette fonction ne récupère pas toutes les informations de la machine car cela nous est inutile car nous n'avons pas besoin de connecter à la machine locale. Elle ne fait que créer une structure ou elle donne un nom "LocalHost" à la machine.

Quand une machine est ajoutée à la liste, une fonction `machine_existe()` vérifie que l'IP de la machine n'apparaît pas déjà dans la liste des machines (doublons).

Cette étape a durée ~ 6 heure car il a fallu réfléchir l'architecture du projet (comment est ce qu'on allait gérer les machines distantes ?) . L'objectif était d'avoir un accès facile à l'ensemble des données de chaque machine.

Connexion aux machines distantes (5 jours)

ssh: une fonction `connection_ssh()` permet de créer une session SSH grâce à la librairies **libssh** qu'il faut télécharger sur la machine pour pouvoir correctement utiliser le programme.

telnet: une fonction `connection_telnet()` permet de créer un socket pour établir une connection TCP vers le serveur distant et gérer l'authentification

deux fonctions – `ssh_exec() et telnet_exec()` permettent d'exécuter une commande sur une machine distante afin de récupérer les processus ("ps -eo pid,user,pcpu,pmem,state,comm --no-headers")

Une fonction complémentaire `send_signal_ssh()` permet d'envoyé les signaux aux processus des machines connecté avec ssh (SIGTERM, SIGSTOP...).

Cette étape a pris plusieurs jours car il a fallu d'abord apprendre les notions de connection ssh et telnet pour savoir comment les implémentée dans le code. Deuxièmement, une fois la connection établie, nous nous sommes rendu compte que la partie **Network** n'était pas compatible avec la partie **Process** (voir partie **Difficultés rencontrées**)nous avons donc dû corriger tous les conflits**.**

Gestion de mémoire (30min)

De même que pour les options, une fonction `free_machine_list()` sert à libérer la mémoire alloué par toutes les `remote_machine` créées.

Le Main réalisé sur toute la durée du projet par tous les membres

Initialisation des options et création de la liste des machines

Nous avons d'abord commencé par initialisé la variable opts qui est une structure `program_options` et qui contiendra toutes les options du programme. on applique ensuite les différentes fonctions du module options.c à cette structure pour correctement initialisé

Le main crée ensuite la liste des machines distantes à partir des fonctions du module network.c comme `lire_config()`, `ajouter_machine_utilisateur()` ou `ajouter_machine_local()`**.**

Une fois la liste créée, le programme vérifie si la première machine de la liste est une machine distante. Si c'est le cas, il se connecte à cette machine via le bon protocole (telnet ou ssh).

Boucle principale et affichage de l'interface

L'interface utilisateur est initialisée avec `ui_init()` et un signal `SIGWINCH` est capté pour gérer le redimensionnement de la fenêtre (car le programme s'arretait lorsqu'on redimensionnait la fenêtre).

La boucle principale du programme gère :

- La récupération des processus via `read_proc()` selon le type de machine et la session active.
- Le tri des processus par consommation mémoire avec `sort_by_mem()` et la mise à jour de pourcentages mémoire.
- L'affichage dynamique des processus sur la console avec `affichePrinc()`.
- La gestion des interactions clavier grâce à `ui_traite_event()`.

L'interface se rafraîchit toute les 150ms ou lorsque l'utilisateur appuie sur une touche.

Gestion de la mémoire et fermeture

À la fin de chaque itération de la boucle, le programme libère la liste des processus alloués pour éviter les pertes mémoire..

Une fois que la boucle se termine, la mémoire allouée par la liste des machines est aussi libérée et l'interface est coupée.

Difficultés rencontrées

Compréhension des objectifs

Imadeddine Betka:

Accès aux informations des processus sous Linux

- **Instruction incomplète:** le sujet demandait d'afficher des informations sur les processus , mais ne précisait pas clairement ou les récupérer. De même façon, le calcul de mémoire en pourcentage.

- **Solution adoptée :** Après exploration de la documentation et plusieures tests , j'ai trouvé les répertoires /proc/[pid]/status , et /proc/meminfo. Par la suite, adaptation en utilisant la commande `ps` (plus simple et utilisable pour les machines distantes)

Interaction avec les processus

- **Instruction incomplète:** Le sujet demandait de “pouvoir interagir avec les processus” (pause, arrêt, reprise...) , mais il ne précisait pas comment réagir en cas d’erreur ou duplication de code.

- **Solution adoptée :** Nous avons décidé de créer une fonction `send_signal_to_pid()` qui appelle `kill()` et vérifier si l’envoi a échoué.. (ou appelle une fonction `send_signal_ssh()` pour les machines distantes qui utilise la commande kill.

Tri et affichage des processus

- **Instruction incomplète:** le sujet demandait d’afficher la liste des processus triés , mais ne précisait pas quel critère prioritaire utiliser ni le type de tri attendu.

- **Solution adoptée :** Nous avons choisi de trier les processus par ordre décroissant de pourcentage mémoire similaire au Task Manager, ou les processus gourmands sont affichés au top.
- Pour cela, j’ai implémenté un tri par insertion vu en **LO21**.
- Côté affichage, j’ai limité volontairement le nombre de lignes affichées afin d'éviter une sortie illisible et faciliter le débogage.

Abel HALLER:

Gestion des conflits entre options d’identification

- **Instruction incomplète:** le sujet ne parlait pas des conflits entre `--remote-server`, `--username` et `--login`. Il était donc possible de renseigner deux utilisateurs et deux adresse IP

- **Solution adoptée :** Nous avons mis en place une hiérarchie : l’option `--login` prévaut sur `--remote-server` et `--username`. Avant de l’appliquer, nous informons l’utilisateur et demandons son accord via un prompt **[Y/o]**. Si la réponse est autre que **Y**, le programme s’arrête.

- Nous avons traiter ce problème avant de commencer à programmer

Absence de l’option --server-name

- **Instruction incomplète :** Le sujet indiquait que le fichier de configuration devait contenir `server_name`, mais aucune option `--server-name` n’était prévue pour renseigner le nom de serveur de la machine renseignée manuellement.
- **Solution adoptée :** Nous avons choisi de remplir ce champ avec la valeur par défaut `'machine utilisateur'`, correspondant au nom de la machine renseignée manuellement.

- Nous avons traitez ce problème pendant la réalisation de la partie network mais bien que le problème aurait pu être traîté plus tôt, il ne nous a pas posé de problème d'implémentation

Comportement en cas d'échec de connexion à une machine distante

- **Instruction incomplète :** Le sujet ne précisait pas comment le programme devait réagir si une machine distante n'était pas accessible.

- **Solution adoptée :** Le programme continue de fonctionner pour les autres machines. La page de la machine non accessible devient une page d'erreur. Si la première machine de la liste échoue, le programme ne se lance pas.

- **Amélioration possible :** La gestion de ce problème s'est faite tardivement et à généré plusieurs erreurs qui nous ont fait perdre du temps. Si nous avions pensé à cette fonctionnalité plus tôt nous aurions évité ces problèmes

Basile KONE-BOUSSEMART

Création d'une interface ergonomique

- **Instruction incomplète** : le sujet nous invitait à nous inspirer de htop afin de réaliser notre interface mais ne nous donnait pas clairement la façon dont elle devait se présenter

- **Solution adoptée** : Réalisation d'une interface largement inspirée de htop tout en prenant quelques libertés afin d'améliorer la lisibilité.

Réalisation des objectifs

Conflits entre la partie Network et la partie Process

Symptômes: Nous avons réalisé la partie **network** en dernier, et une fois celle-ci réalisée, nous nous sommes rendu compte qu'elle n'était pas compatible avec les fonctions du module **Process.**
En effet, lorsque nous avons tenté de récupérer les processus sur les machines distante avec `read_processes()`, nous nous sommes rendu compte que les processus affichés restaient tout le temps ceux de la machine local. La fonction `read_processes()` fonctionnait à ce moment là en lisant le fichier `/proc` du PC. Cependant cette méthode ne peut fonctionner qu'avec la machine locale.

Résolution du problème: Pour répondre à ce problème, nous avons alors fait fonctionner la fonction en exécutant la commande `ps -eo pid,user,rss,pmem,state,comm --no-headers` car il est tout à fait possible d'exécuter des commandes sur les machines distantes et de récupérer les résultats. ainsi, l'architecture de la fonction `read_proc(int sockfd, ssh session session)` est la suivante:

- Création d'une liste chaînée de processus avec un pointeur sur la tête et une sur la queue
- On exécute la commande **ps:**
    - Avec `telnet_exec()` si `sockfd >0` (ce qui signifie qu'il y a une connection ssh)
    - Avec `ssh_exec()` si `ssh_session != NULL`
    - avec `popen("ps …")` si aucune machine n'est connecté (lecture local)
- Le résultat de la commande est écrit dans la variable `output`
- Cette variable output est parsé ligne par ligne avec le délimiteur '\n' et avec la fonction `sscanf()` qui permet d'extraire des données comme `scanf()` mais dans une chaîne de caractère au lieu d'un flux de données.
- Chaque processus est inséré en queue de la liste chaînée de processus et le pointeur tail est mis à jour.

Anticipation: Pour mieux anticiper ce problème, nous aurions dû commencer la partie network en parallèle des autres parties mais étant donné qu'il fallait rendre la version locale en premier nous avons préféré d'abord mettre la partie network de coté et nous n'avions donc pas considéré les potentiels problèmes engendrés..

Inclusion de variable globale

Problème rencontré: Au cours de l'implémentation de la partie réseaux, nous nous sommes rendu compte que plusieurs variables était appelé beaucoup de fois dans beaucoup de fonctions et que cela rendait le code moins lisible.

Résolution du problème: pour parer ce problème, nous avons déclaré plusieurs variables globale dans un fichier global.h que nous avons ensuite définis dans le main.c. Ces variables globales sont pratiques car elles sont accéssibles. Nous avons donc les variables globale suivantes:

- `liste_machine` **:** la liste des machine avec leurs informations
- `nb_machines` : le nombre de machine au total traité par le programme (longueur de `liste_machine`)
- `current_page`: la page actuelle ou se trouve l'utilisateur dans l'interface. cette page sert aussi d'indice pour savoir quelle machine utilisé (ex: `liste_machine[current_page]`)
- `current_ssh_session` **:** cette variable permet d'enregistrer une session ssh
- `current_telnet_session` **:** cette variable permet d'enregistrer un socket de connexion Telnet.

Pour les deux dernière, il faut savoir qu'elles ne sont jamais définis en même temps:

- Si `liste_machine[current_page]` est la machine locale:
    - `current_ssh_session = NULL`
    - `current_telnet_session = -1`

- Si `liste_machine[current_page]` est une machine connecté via telnet:
  - `current_ssh_session = NULL`
  - `current_telnet_session = val` avec val >= 0

- Si `liste_machine[current_page]` est une machine connecté via ssh:
  - `current_ssh_session = session`
  - `current_telnet_session = -1`

Avec ces variables global, il est plus facile de se repérer qu'importe le module dans lequel on travaille.

Anticipation: Nous n'avions jamais utilisé de variable globale (ou très peu) avant ce projet. Nous n'avions donc pas pensé à cette option tout de suite, mais c'est un paramètre qui aurait pu être traité plus tôt.

Connection Telnet

Problème rencontré: nous n'avons pas réussi à établir de connexion Telnet concrète avec une machine distante malgré de nombreux essais, telnet ne voulait pas s'activer sur la machine et nous ne savons toujours pas comment faire pour l'activer.

Résolution du problème: nous avons réussi à établir une connexion **TCP** sur le port 23 grâce à la commande `nc -l 23`. la fonction telnet_connection() fonctionne bien et la fonction exec_telnet() écris bien la commande ps dans le terminal de la machine distante. Nous avons donc réussi à accéder à la machine distante via telnet mais nous n'avons pas réellement pu exécuter de commande dessus.

Anticipation: il était compliqué d'anticiper ce problème tôt dans le programme car nous n'avions aucune connaissance sur la connexion Telnet.

Problèmes de lecture dans /prc

Symptômes: Au début, certains informations lues dans /proc/[pid]/status étaient :

- Incorrects
- Vides
- un segmentation fault

Par exemple , certains champs n'étaient pas présents pour tous les processus, ce qui faisait planter sscanf().

Résolution du problème: Pour répondre à ce problème, nous avons ajouté des vérifications systématiques :

- vérifier que le fichier est bien ouvert.
- vérifier le retour de sscanf().
- initialiser les variables par défaut avant lecture.

Ainsi, même si une ligne n'est pas trouvée, le programme continue sans planter.

Anticipation: Oui, Lire plus tôt la documentation de /porc nous aurait permis de voir que certaines lignes ne sont pas toujours présentes.

Segfaults liés aux listes chaînées

Symptômes:
- Plantages aléatoires
- Accès mémoire invalides.
- Tri impossible

Cela venait surtout de pointeurs non initialisés ou mal mis à jour..

Résolution du problème: Pour répondre à ce problème :
- Initialiser systématiquement next = NULL
- vérifier avant tout accès : if (p != NULL)
- Isoler la création, l'insertion et libération dans des fonctions dédiées.

Cela a rendu le module plus stable..
Anticipation: Partiellement, plus de tests unitaires sur les listes aurait aidé.

Proposition d'une amélioration

Uniformisation de la partie réseau

Du côté réseau, Une piste d'amélioration consisterait à **introduire une abstraction plus claire des protocoles de communication** (SSH, Telnet) à l'aide d'une interface commune:

Principe:
Au lieu de gérer les spécificités de chaque protocole dans la partie network, il faudrait mettre en place une structure générique de connexion avec des fonctions comme `connect`, `send`, `receive`, `disconnect` et des informations communes (adresse, port, utilisateur, etc.).

Grâce à cela, nous n'aurions plus besoin de connaitre le protocole de connexion dans le module manager. Ainsi, tous les tests du type `if (ssh) ... else if (telnet) ...` pourrait être supprimé et remplacée par des appels commun.

Ajout de nouvelle fonctionnalité

L'objectif serait de proposer plus de possibilité de manipulation de l'interface de manière à se rapprocher encore plus de htop avec la possibilité par exemple de trier la liste des processus selon la caractéristique que l'on souhaite.

Analyse des résultats

Network

Les résultats que nous avons obtenus sur la partie network sont satisfaisants mais nous restons sur notre faim, la connexion ssh fonctionne parfaitement mais la connexion telnet n'as pas pu être testé.
Le programme fonctionne donc parfaitement tant que toute les machines distantes se connecte via ssh mais pas avec des machines telnet.

Perte de mémoire

Après avoir testé le code avec **Valgrind**, nous avons eu les résultats ci-dessous

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 201 bytes in 3 blocks
   still reachable: 73,929 bytes in 393 blocks
        suppressed: 0 bytes in 0 blocks
```

Nous constatons donc que notre projet n'as **aucune perte définitive où indirect**, ce qui est un très bon point car cela signifie qu'aucun pointeur d'est perdu et qu'aucune zone de mémoire allouée n'est définitivement perdue.
Nous avons cependant **quelques pertes possibles**. Cela signifie que **Valgrind n'est pas sûr qu'il s'agisse de perte**. Nous pensons que ces pertes possibles sont dues à l'utilisation des bibliothèques comme ncurses, signal...)

Retour d'expérience

Globalement, le projet a atteint ses objectifs fonctionnels, mais plusieurs difficultés ont impacté son déroulement.

Sous-estimation de la complexité de l'architecture globale :

Au début du projet nous avons abordé chaque module de manière relativement indépendante, en particulier la partie Process, qui a été développée principalement pour une exécution locale. Cette approche a conduit à une incompatibilité tardive avec la partie Network.
Il aurait été plus efficace de définir dès le départ une interface commune pour la récupération des processus, indépendante de la source (locale ou distant).

Apprentissage progressif de ncurses et de l'ergonomie

La première version de l'interface était fonctionnelle mais peu ergonomique. Au fil du projet est apparu nécessaire d'améliorer la lisibilité, la navigation et la gestion des entrées utilisateur, ce qui a conduit à plusieurs mises à jour.
Un prototype plus simple,accompagné de tests utilisateurs (au sein du groupe), aurait permis de converger plus rapidement vers une interface satisfaisante.

Communication et coordination au sein du groupe
Malgré une répartition claire des tâches, certaines décisions techniques (comme l'utilisation de /proc ou de ps) ont été prises de manière isolée, ce qui a parfois généré des incohérences entre les module
Des réunions de synchronisation plus fréquentes, même courtes, auraient permis d'anticiper ces divergences.

Conclusion

Ce projet LP25 a permis de concevoir un outil de supervision de processus fonctionnel, modulaire et relativement complet, inspiré d'outils professionnels existants comme *htop*.
Il a couvert de nombreux aspects fondamentaux de la programmation système en C : gestion de la mémoire, manipulation de listes chaînées, signaux Unix, lecture d'informations système, interface ncurses et communication réseau.

Au-delà de l'aspect technique, ce projet a mis en évidence l'importance de l'architecture logicielle et de la planification. Certaines difficultés, notamment les conflits entre les modules Process et Network, auraient pu être anticipés par une conception plus globale dès le début du projet. Cependant, leur résolution a permis de renforcer la compréhension du fonctionnement interne des systèmes Linux et des interactions entre les différentes couches du programme.

Le travail en équipe a également été un élément clé de la réussite du projet. La répartition claire des tâches, combinée à des phases de coordination régulières, a permis d'intégrer progressivement les différentes parties et d'aboutir à une version cohérente du programme.

Enfin, ce projet a constitué une expérience formatrice pour la suite du parcours d'ingénieur. Il a permis de consolider des compétences techniques en langage C et en environnement Linux, tout en développant des réflexes essentiels en gestion de projet, en résolution de problèmes et en analyse critique de son propre travail.