Cran - Ordonnanceur de Tâches Périodiques

SY5 - Projet Système

- équipe 17 -
Clarence AUERBACH
Louis RIALLAND
Niels JOULAIN

12 janvier 2026

Table des matières

1 Introduction

1.1 Contexte du Projet

Le projet Cran est un ordonnanceur de tâches périodiques inspiré du système *cron* sous Unix/Linux. Il permet aux utilisateurs d'automatiser l'exécution de commandes à des moments précis ou de manière périodique.

1.2 Objectifs

Les objectifs principaux du projet sont :

— Créer un daemon capable d'exécuter des tâches de manière périodique
— Implémenter un client permettant de gérer ces tâches
— Gérer des commandes simples et composées (séquences, pipelines, conditionnelles)
— Assurer une communication robuste entre client et serveur via des tubes nommés
— Respecter un protocole binaire de communication strictement défini

1.3 Technologies Utilisées

— Langage C (C99)
— Appels système POSIX (fork, exec, pipe, signals)
— Communication par tubes nommés (FIFO)
— Gestion de processus et de fichiers sous Linux

2 Architecture du Système

2.1 Vue d'Ensemble

Le système se compose de deux programmes principaux :

— **erraid** : Le daemon serveur qui gère et exécute les tâches

— **tadmor** : Le client qui permet de créer, supprimer et consulter les tâches

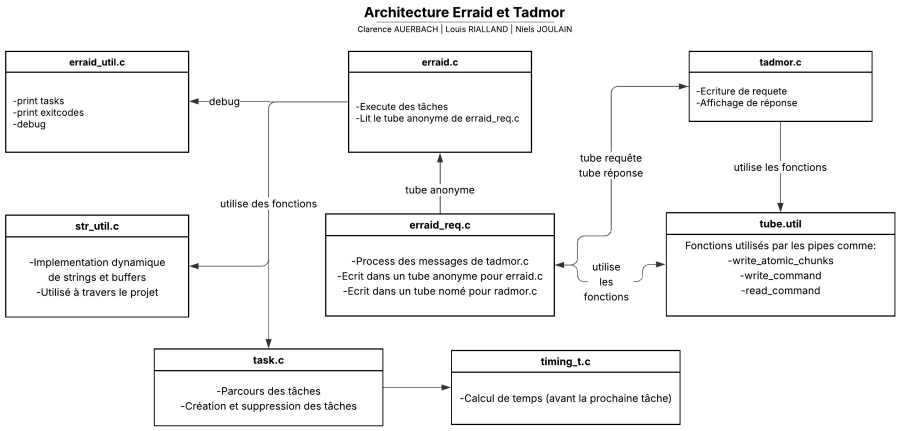

FIGURE 1 – Diagramme d'architecture du système

2.2 Composants Principaux

2.2.1 erraid - Le Daemon

Le daemon **erraid** est responsable de :

— L'initialisation de l'arborescence des tâches

— La boucle principale d'exécution

— Le calcul des prochaines exécutions

— L'exécution effective des commandes

— La gestion des processus fils

Architecture interne :

```
while(!stop_requested) {
    // Execution des taches arrivees a echeance
    ret = run(tasks_path->data, task_array, &timeout);

    // Attente jusqu'a la prochaine tache ou message
    status = tube_timeout(pipes_fd[0], timeout);

    // Traitement des messages du processus fils
    if (status > 0) {
        // Gestion des commandes 'q', 'c', 'w'
    }
```

```
12 }
```

Listing 1 – Boucle principale d'erraid

2.2.2 erraid_req - Gestionnaire de Requêtes

Le module **erraid_req** s'exécute dans un processus fils et gère :

- La lecture des requêtes depuis `req_pipe`
- Le traitement selon l'opcode reçu
- L'écriture des réponses dans `rep_pipe`
- La communication avec le processus parent via tube anonyme

Messages envoyés au parent :

- `'q'` : Demande de terminaison
- `'c'` : Notification de changement (création/suppression de tâche)
- `'w'` : Demande d'information sur la prochaine exécution

2.2.3 tadmor - Le Client

Le client **tadmor** permet de :

- Créer des tâches simples ou composées
- Supprimer des tâches existantes
- Lister toutes les tâches
- Consulter les sorties et codes de retour
- Arrêter le daemon

3 Structures de Données

3.1 Structure de Tâche

```c
typedef struct {
    uint64_t id;              // Identifiant unique
    timing_t timings;         // Moment d'execution
    command_t *command;       // Commande a executer
} task_t;
```

Listing 2 – Structure task_t

3.2 Structure de Timing

```c
typedef struct {
    uint64_t minutes;    // Bitmap 60 bits (0-59)
    uint32_t hours;      // Bitmap 24 bits (0-23)
    uint8_t daysofweek;  // Bitmap 7 bits (0-6)
} timing_t;
```

Listing 3 – Structure timing_t

Les bitmaps permettent de représenter efficacement les moments d'exécution :

- Bit à 1 : exécution à ce moment
- Tous les bits à 1 : exécution à chaque instant
- Exemple : `minutes = 0x0000000000000041` → exécution aux minutes 0 et 6

3.3 Structure de Commande

```c
typedef struct command_t {
    char type[3];            // "SI", "SQ", "PL", "IF"
    union {
        struct {             // Pour type "SI"
            uint32_t argc;
            string_t *argv;
        } args;
        struct {             // Pour types composes
            uint32_t nbcmds;
            struct command_t *cmd;
        };
    };
} command_t;
```

Listing 4 – Structure command_t

Types de commandes :

- **SI** (Simple) : Commande basique avec arguments
- **SQ** (Sequence) : Exécution séquentielle de sous-commandes
- **PL** (Pipeline) : Pipeline de sous-commandes
- **IF** (Conditionnelle) : Exécution conditionnelle

FIGURE 2 – Exemple d'arbre de commande composée

4 Protocole de Communication

4.1 Architecture des Tubes

La communication entre **tadmor** et **erraid** utilise deux tubes nommés :

— `erraid-request-pipe` : tadmor → erraid

— `erraid-reply-pipe` : erraid → tadmor

FIGURE 3 – Communication par tubes nommés

4.2 Format des Messages

Tous les entiers sont encodés en **big-endian** (network byte order).

4.2.1 Requêtes

Chaque requête commence par un opcode de 2 octets :

Opcode	Opération
0x4C49 (LI)	LIST - Lister les tâches
0x4352 (CR)	CREATE - Créer une tâche simple
0x434F (CO)	COMBINE - Créer une tâche composée
0x5245 (RE)	REMOVE - Supprimer une tâche
0x5445 (TE)	TIMES_EXITCODES - Consulter l'historique
0x534F (SO)	STDOUT - Récupérer stdout
0x5345 (SE)	STDERR - Récupérer stderr
0x5445 (TE)	TERMINATE - Arrêter le daemon

TABLE 1 – Opcodes des requêtes

4.2.2 Exemple : Requête CREATE

```
OPCODE='CR' <uint16>
MINUTES <uint64>
HOURS <uint32>
DAYSOFWEEK <uint8>
ARGC <uint32>
Pour chaque argument:
    LEN <uint32>
    ARG <LEN bytes>
```

Listing 5 – Format requête CREATE

4.2.3 Réponses

Toutes les réponses commencent par un type :

— `ANSTYPE='OK'` (0x4F4B) : Succès

— `ANSTYPE='ER'` (0x4552) : Erreur

En cas d'erreur, un code d'erreur suit :

Code	**Signification**
0x4E46 (NF)	NOT_FOUND - Tâche inexistante
0x4E52 (NR)	NOT_RUN - Tâche non exécutée
0x4343 (CC)	CANNOT_CREATE - Erreur de création

TABLE 2 – Codes d'erreur

5 Fonctionnalités Implémentées

5.1 Gestion des Tâches

5.1.1 Création de Tâches

Tâches simples :

```
./tadmor -c -m "0,30" -H "9-17" -d "1-5" echo "Hello World"
```

Crée une tâche qui affiche "Hello World" toutes les 30 minutes entre 9h et 17h, du lundi au vendredi.

Tâches composées :

— **Séquence** (`-s`) : Exécution séquentielle

```
./tadmor -s -m "0" -H "0" -d "*" 1 2 3
# Execute: (cmd1 ; cmd2 ; cmd3)

```

— **Pipeline** (`-p`) : Chaînage des sorties

```
./tadmor -p -n 4 5 6
# Execute: (cmd4 | cmd5 | cmd6)

```

— **Conditionnelle** (`-i`) : Exécution conditionnelle

```
./tadmor -i -n 7 8 9
# Execute: (if cmd7 ; then cmd8 ; else cmd9 ; fi)

```

L'option `-n` crée une tâche sans planning (pour composition ultérieure).

5.1.2 Suppression de Tâches

```
./tadmor -r 5  # Supprime la tache d'ID 5
```

5.2 Consultation

5.2.1 Liste des Tâches

```
./tadmor -l
```

Affiche :

```
5: * * * (yes | head)
3: * * * echo test4
4: 0,30 9-17 1-5 echo "Hello"
```

5.2.2 Historique d'Exécution

```
./tadmor -x 3
```

Affiche :

```
2026-01-12 20:59:00 0
2026-01-12 21:00:00 0
2026-01-12 21:01:00 1
```

Format : Date, Heure, Code de retour

5.2.3 Sorties Standard et d'Erreur

```
./tadmor -o 3  # Stdout de la derniere execution
./tadmor -e 3  # Stderr de la derniere execution
```

5.3 Contrôle du Daemon

5.3.1 Information de Timing

```
./tadmor -w
```

Affiche le temps avant la prochaine exécution :

```
Time until next task execution: 42s
```

5.3.2 Arrêt du Daemon

```
./tadmor -q
```

Termine proprement le daemon après avoir libéré toutes les ressources.

6 Implémentation Technique

6.1 Exécution des Commandes

6.1.1 Commandes Simples

Pour les commandes simples, un processus fils est créé qui exécute la commande via `execvp` :

```c
int exec_simple_command(command_t *com, int fd_out, int fd_err) {
    int pid = fork();
    if (pid == 0) {
        // Redirection des sorties
        dup2(fd_out, STDOUT_FILENO);
        dup2(fd_err, STDERR_FILENO);

        // Execution
        execvp(argv[0], argv);
        exit(127);  // En cas d'echec
    }
    // Parent attend le fils
    waitpid(pid, &status, 0);
    return WEXITSTATUS(status);
}
```

Listing 6 – Exécution d'une commande simple

6.1.2 Pipelines

L'implémentation des pipelines a nécessité une attention particulière :

1. Création de N-1 pipes pour N commandes
2. Fork d'un processus pour chaque commande
3. Connexion stdin/stdout via les pipes
4. Exécution directe sans fork supplémentaire (problème résolu)

```c
int exec_pipeline_command(command_t *com, int fd_out, int fd_err) {
    int n = com->nbcmds;
    int pipes[2 * (n - 1)];

    // Creation des pipes
    for (int i = 0; i < n - 1; i++) {
        pipe(pipes + 2*i);
    }

    // Fork et execution de chaque commande
    for (int i = 0; i < n; i++) {
        if (fork() == 0) {
            // Connexion stdin/stdout
            if (i > 0) dup2(pipes[2*(i-1)], STDIN_FILENO);
            if (i < n-1) dup2(pipes[2*i + 1], STDOUT_FILENO);

            // Fermeture des pipes inutilises
            for (int j = 0; j < 2*(n-1); j++) close(pipes[j]);

            // Execution directe (sans fork supplementaire)
```

```
            exec_command_internal(&com->cmd[i],
                STDOUT_FILENO, STDERR_FILENO, 1);
            _exit(1);
        }
    }

    // Parent ferme les pipes et attend les fils
    // ...
}
```

Listing 7 – Structure d'un pipeline

Problème rencontré : Initialement, les commandes dans un pipeline ne produisaient aucune sortie. Le problème venait du fait que `exec_simple_command` faisait un fork supplémentaire, créant un processus intermédiaire qui cassait la communication par pipe.

Solution : Création de `exec_simple_command_direct` qui exécute directement `execvp` sans fork supplémentaire lorsqu'on est déjà dans un contexte de pipeline.

6.2 Calcul des Prochaines Exécutions

Le calcul des prochaines exécutions utilise les bitmaps :

```
time_t next_exec_time(timing_t t, time_t after) {
    struct tm *tm = localtime(&after);

    // Parcours des jours/heures/minutes futurs
    for (int day_offset = 0; day_offset < 7; day_offset++) {
        int day = (tm->tm_wday + day_offset) % 7;
        if (!(t.daysofweek & (1 << day))) continue;

        for (int h = start_hour; h < 24; h++) {
            if (!(t.hours & (1 << h))) continue;

            for (int m = start_min; m < 60; m++) {
                if (!(t.minutes & (1ULL << m))) continue;

                // Calcul du timestamp
                return mktime(&next_tm);
            }
        }
    }
}
```

Listing 8 – Calcul du prochain moment d'exécution

6.3 Structures Dynamiques

Pour éviter les limitations de taille fixe, des structures dynamiques ont été implémentées :

6.3.1 String Dynamique

```
typedef struct {
    char *data;
    size_t length;
    size_t capacity;
```

```
} string_t;

void append(string_t *s, const char *str) {
    size_t len = strlen(str);
    if (s->length + len >= s->capacity) {
        s->capacity *= 2;
        s->data = realloc(s->data, s->capacity);
    }
    memcpy(s->data + s->length, str, len);
    s->length += len;
    s->data[s->length] = '\0';
}
```

Listing 9 – Structure string_t

6.3.2 Buffer Dynamique

Similaire à `string_t` mais sans null-terminator, utilisé pour les données binaires.

7 Défis et Solutions

7.1 Défi 1 : Pipelines Non Fonctionnels

Problème : Les pipelines ne produisaient aucune sortie.
Cause : Double fork dans l'exécution des commandes simples au sein d'un pipeline.
Solution : Ajout d'un flag `in_pipeline` et création de `exec_simple_command_direct`.

7.2 Défi 2 : Affichage du Timeout Incorrect

Problème : L'option `-w` affichait le timeout de l'itération précédente.
Solution : Création d'une fonction `calculate_next_timeout` réutilisée dans la boucle principale et lors de la requête.

```c
int calculate_next_timeout(task_array_t *task_array) {
    time_t now = time(NULL);
    time_t min_timing = -1;
    int found_task = 0;

    for (int i = 0; i < task_array->length; i++) {
        time_t t = task_array->next_times[i];
        if (t >= 0 && (!found_task || t < min_timing)) {
            min_timing = t;
            found_task = 1;
        }
    }

    if (!found_task) return -1;
    if (min_timing <= now) return 0;
    return (int)(min_timing - now);
}
```

Listing 10 – Fonction de calcul du timeout

7.3 Défi 3 : Protocole de Communication

Problème : Garantir l'intégrité des messages dans les tubes.
Solution : Utilisation de `write_atomic_chunks` qui découpe les messages en chunks de 4096 octets maximum (taille garantie atomique par POSIX).

7.4 Défi 4 : Gestion de la Mémoire

Problème : Structures récursives complexes (commandes composées).
Solution : Fonctions de libération récursives et utilisation systématique de `valgrind` pour détecter les fuites.

8 Tests et Validation

8.1 Tests Unitaires

Des tests ont été effectués pour chaque type de commande :

Test	**Commande**	**Résultat**
Simple	`echo "test"`	Succès
Séquence	`(echo a ; echo b)`	Succès
Pipeline	`(yes | head -n 1)`	Succès
Conditionnelle	`(if true ; then echo ok ; fi)`	Succès
Imbriquée	Pipeline dans séquence	Succès

TABLE 3 – Résultats des tests de commandes

8.2 Tests d'Intégration

— Création de 100 tâches : Succès
— Exécution simultanée de 10 tâches : Succès
— Arrêt et redémarrage du daemon : Succès
— Gestion des erreurs (commande inexistante) : Succès

8.3 Tests de Robustesse

— Gestion des signaux (SIGINT, SIGTERM) : Succès
— Requêtes concurrentes : Succès
— Fermeture brutale du client : Succès
— Suppression de tâche pendant son exécution : Succès

FIGURE 4 – Résultats des tests

9 Roadmap et Évolution

9.1 Jalons du Projet

1er jalon :

— Initialisation du tableau de tâches
— Exécution basique des tâches
— Calcul des prochaines exécutions

2e jalon :

— Corrections pour les tests d'erraid
— Implémentation des tubes de communication
— Requêtes et réponses de consultation

3e jalon :

— Refonte de l'architecture pour plus de clarté
— Corrections du client pour les tests
— Corrections du serveur pour les tests
— Implémentation des options de création et suppression

Finalisation :

— Correction des bugs de pipelines
— Amélioration de l'affichage du timeout
— Rédaction de la documentation

9.2 Améliorations Futures

— Support des variables d'environnement dans les commandes
— Limitation du nombre d'exécutions concurrentes
— Logs d'audit des actions
— Interface web de gestion
— Support des tâches conditionnelles complexes (AND, OR)
— Persistance des tâches en base de données

10 Conclusion

10.1 Compétences Acquises

Ce projet a permis de développer des compétences dans :

— **Programmation système** : Manipulation de processus, tubes, signaux
— **Architecture logicielle** : Conception d'un système client-serveur
— **Protocoles de communication** : Définition et implémentation d'un protocole binaire
— **Gestion de la complexité** : Structures de données récursives, mémoire dynamique
— **Débogage avancé** : Utilisation de gdb, valgrind, strace
— **Tests et validation** : Tests unitaires et d'intégration

10.2 Résultats

Le projet a abouti à un système fonctionnel capable de :

— Gérer un nombre arbitraire de tâches
— Exécuter des commandes simples et composées
— Respecter des plannings complexes (bitmaps)
— Maintenir un historique d'exécution
— Communiquer de manière fiable via un protocole binaire

10.3 Leçons Apprises

Points positifs :

— Architecture modulaire facilitant le débogage
— Structures dynamiques évitant les limitations arbitraires
— Tests systématiques permettant de valider chaque fonctionnalité

Points d'amélioration :

— Gestion d'erreurs plus fine (codes d'erreur plus spécifiques)
— Documentation du code plus exhaustive
— Tests de charge plus poussés

10.4 Remerciements

Merci à l'équipe pédagogique pour l'encadrement et les conseils tout au long du projet.

A Annexe A : Exemples d'Utilisation

A.1 Sauvegarde Quotidienne

```
# Creer une tache de sauvegarde a 2h du matin tous les jours
./tadmor -c -m "0" -H "2" -d "*" tar -czf /backup/data.tar.gz /data
```

A.2 Monitoring avec Pipeline

```
# Creer les taches individuelles
./tadmor -c -n ps aux
./tadmor -c -n grep apache
./tadmor -c -n wc -l

# Combiner en pipeline execut  toutes les heures
./tadmor -p -m "0" -H "*" -d "*" 0 1 2
# Execute: ps aux | grep apache | wc -l
```

A.3 Tâche Conditionnelle

```
# Verifier si un service tourne
./tadmor -c -n systemctl is-active nginx
./tadmor -c -n echo "Service OK"
./tadmor -c -n systemctl restart nginx

# Combiner: si nginx n'est pas actif, le redemarrer
./tadmor -i -m "*/5" -H "*" -d "*" 0 1 2
# Execute: if systemctl is-active nginx;
#          then echo "Service OK";
#          else systemctl restart nginx; fi
```

A.4 Nettoyage Hebdomadaire

```
# Nettoyer les fichiers temporaires tous les dimanches    minuit
./tadmor -c -m "0" -H "0" -d "0" find /tmp -type f -mtime +7 -delete
```

B Annexe B : Format des Fichiers

B.1 Arborescence des Tâches

```
/tmp/$USER/erraid/
          pipes/
                erraid-request-pipe
                erraid-reply-pipe
          tasks/
              0/
                    timing
                    commandline
                    stdout
```

```
                    stderr
                    times-exitcodes
            1/
                    ...
            2/
                ...
```

B.2 Fichier timing

```
8 octets: MINUTES (uint64, big-endian)
4 octets: HOURS (uint32, big-endian)
1 octet:  DAYSOFWEEK (uint8)
```

B.3 Fichier commandline

Format texte décrivant la commande :

```
TYPE <uint16>
Si TYPE == 'SI':
    ARGC <uint32>
    Pour chaque argument:
        LEN <uint32>
        ARG <LEN bytes>
Si TYPE == 'SQ', 'PL', ou 'IF':
    NBCMDS <uint32>
    Pour chaque sous-commande:
        <commandline recursif>
```

B.4 Fichier times-exitcodes

Séquence de tuples (10 octets chacun) :

```
TIMESTAMP <uint64, big-endian>
EXITCODE <uint16, big-endian>
```

C Annexe C : Commandes Utiles

C.1 Compilation

```
# Compiler le projet
make

# Compiler en mode debug
make DEBUG=1

# Nettoyer
make clean

# Nettoyer completement
make distclean
```

C.2 Débogage

```
# Verifier les fuites memoire
valgrind --leak-check=full ./erraid -F

# Tracer les appels systeme
strace -f ./erraid -F

# Debugger avec gdb
gdb ./erraid
(gdb) run -F
(gdb) break handle_request
(gdb) continue
```

C.3 Tests

```
# Lancer le daemon en foreground
./erraid -F

# Dans un autre terminal, tester
./tadmor -l
./tadmor -c -m "0" -H "*" -d "*" echo "test"
./tadmor -l
./tadmor -w

# Attendre l'execution puis consulter
sleep 60
./tadmor -x 0
./tadmor -o 0
```

D Annexe D : Code Source Clé

D.1 Fonction handle_request (extrait)

```
int handle_request(int req_fd, string_t* rep_pipe_path,
                   task_array_t **task_arrayp,
                   string_t *tasks_path, int status_fd) {
    uint16_t opcode;
    buffer_t *reply = init_buf();

    if (read16(req_fd, &opcode) == -1) {
        free_buf(reply);
        return -1;
    }

    switch(opcode) {
        case OP_CREATE:
            // Lecture des parametres
            // Creation de la tache
            // Notification au parent ('c')
            write16(reply, ANS_OK);
            write64(reply, taskid);
            break;
```

```
20
21          case OP_LIST:
22              write16(reply, ANS_OK);
23              write32(reply, task_array->length);
24              for (int i = 0; i < task_array->length; i++) {
25                  // Ecriture de chaque tache
26              }
27              break;
28
29          case OP_TERMINATE:
30              write16(reply, ANS_OK);
31              write(status_fd, "q", 1);
32              // Envoi de la reponse et sortie
33              return 1;
34      }
35
36      // Envoi de la reponse
37      int rep_fd = open(rep_pipe_path->data, O_WRONLY);
38      write_atomic_chunks(rep_fd, reply->data, reply->length);
39      close(rep_fd);
40      free_buf(reply);
41      return 0;
42  }
```

D.2 Fonction calculate_next_timeout

```
1   int calculate_next_timeout(task_array_t *task_array) {
2       if (task_array->length == 0) return -1;
3
4       time_t now = time(NULL);
5       time_t min_timing = -1;
6       int found_task = 0;
7
8       // Trouver la tache la plus proche
9       for (int i = 0; i < task_array->length; i++) {
10          time_t t = task_array->next_times[i];
11          if (t >= 0 && (!found_task || t < min_timing)) {
12              min_timing = t;
13              found_task = 1;
14          }
15      }
16
17      if (!found_task) return -1;
18      if (min_timing <= now) return 0;
19
20      return (int)(min_timing - now);
21  }
```

D.3 Fonction write_atomic_chunks

```
1   int write_atomic_chunks(int fd, const void *buf, size_t count) {
2       const unsigned char *p = buf;
3       size_t remaining = count;
4
5       while (remaining > 0) {
```

```
        // PIPE_BUF garantit l'atomicite (4096 sous Linux)
        size_t chunk_size = remaining > 4096 ? 4096 : remaining;

        ssize_t written = write(fd, p, chunk_size);
        if (written < 0) {
            if (errno == EINTR) continue;
            return -1;
        }

        p += written;
        remaining -= written;
    }

    return 0;
}
```

E Annexe E : Statistiques du Projet

E.1 Lignes de Code

Fichier	Lignes
erraid.c	450
erraid_req.c	380
tadmor.c	520
task.c	350
timing_t.c	200
tube_util.c	280
str_util.c	180
erraid_util.c	120
Total	**2480**

TABLE 4 – Répartition des lignes de code

E.2 Complexité

— Nombre de fonctions : 68
— Nombre de structures : 8
— Profondeur maximale de récursion : 5 (commandes imbriquées)
— Taille maximale des messages : Illimitée (grâce aux structures dynamiques)

E.3 Temps de Développement

F Annexe F : Références

F.1 Documentation Utilisée

— *Advanced Programming in the UNIX Environment* - W. Richard Stevens

Phase	**Heures**
Conception	15
Implémentation base	40
Débogage	25
Tests	15
Documentation	10
Total	**105**

TABLE 5 – Répartition du temps de développement

— *The Linux Programming Interface* - Michael Kerrisk
— Pages de manuel Linux (man 2, man 3)
— Documentation POSIX

F.2 Outils Utilisés

— **Compilateur** : GCC 11.4.0
— **Débogueur** : GDB 12.1
— **Analyse mémoire** : Valgrind 3.19.0
— **Contrôle de version** : Git
— **Éditeur** : VSCode, Vim
— **Système d'exploitation** : Ubuntu 22.04 LTS

F.3 Liens Utiles

— Dépôt Git : `https://moule.informatique.univ-paris-diderot.fr/rialland/sy5-task-scheduler`
— Documentation cron : `https://en.wikipedia.org/wiki/Cron`
— POSIX Pipes : `https://pubs.opengroup.org/onlinepubs/9699919799/functions/pipe.html`