# TE Lab2 Cheatsheet

**SYE**

-

**Résumé du document**

Definition

## Table des matières

# 1. Laboratoire pipelining

Ce laboratoire est découpé en deux parties. La première consiste à compter le nombre de lettres dans un texte en utilisant du multi-threading. La seconde partie consiste à implémenter le tag `|` pour utliser des commandes chainées.

## 1.1. Multi-threading

Pour préparer les threads nous pouvons utiliser une boucle for permettant de séparer le fichier texte en plusieurs sous tableau et de les envoyer dans un thread.

```
// Charge le contenu d'un fichier spécifié par `filename` dans un tampon.
char *text = buffer_from_file(filename);

// Calcule la taille totale du texte (nombre de caractères).
size_t text_size = strlen(text);

// Détermine la taille de chaque segment (chunk) à traiter par un thread.
// Chaque thread traite une portion de texte de `chunk_size` caractères.
size_t chunk_size = text_size / thread_num;

// Calcule le reste des caractères qui ne peuvent pas être divisés uniformément entre les
threads.
// Ce reste sera ajouté au dernier thread.
size_t remaining_size = text_size % thread_num;

// Déclare un tableau pour stocker les identifiants de chaque thread.
pthread_t threads[thread_num];

// Déclare un tableau de structures pour passer des paramètres à chaque thread.
// Chaque élément contient les informations nécessaires pour que le thread traite sa portion de
texte.
count_param_t params[thread_num];

// Boucle pour créer `thread_num` threads.
for (int i = 0; i < thread_num; ++i) {
    // Calcule le pointeur de départ de la portion de texte à traiter par le thread.
    // Le pointeur est décalé de `i * chunk_size` caractères par rapport au début du texte.
    params[i].text_pointer = text + i * chunk_size;

    // Calcule la taille du segment à traiter par ce thread.
    // Si c'est le dernier thread, il prend les caractères restants.
    // Sinon, il traite exactement `chunk_size` caractères.
    params[i].size = (i == thread_num - 1) ? chunk_size + remaining_size : chunk_size;

    // Initialise le tableau `counters` (par exemple, pour compter les occurrences de lettres).
    // La fonction `memset` met tous les éléments de `counters` à 0.
    memset(params[i].counters, 0, sizeof(params[i].counters));

    // Crée un thread qui exécute la fonction `count_letters` avec les paramètres correspondants.
    // Si la création du thread échoue, affiche un message d'erreur, libère la mémoire allouée
    // et retourne une erreur.
    if (pthread_create(&threads[i], NULL, count_letters, &params[i]) != 0) {
        printf("Error: Failed to create thread %d\n", i);
        free(text); // Libère la mémoire utilisée pour le texte.
        return EXIT_FAILURE; // Quitte avec un code d'échec.
    }
}
```

Une fois que les threads sont créés, il faut attendre qu'ils terminent leur travail. Pour cela, on utilise la fonction `pthread_join` qui permet d'attendre la fin de l'exécution d'un thread.

```
for (int i = 0; i < thread_num; ++i) {

  // Attend la fin de l'exécution du thread `threads[i]`.
  pthread_join(threads[i], NULL);

  // Enregistre les valeurs des compteurs de chaque thread dans le tableau partagé
  for (int j = 0; j < LETTERS_NB; ++j) {
    result_counters[j] += params[i].counters[j];
  }
}
```

## 1.2. Pipelining

Pour implémenter le tag | pour les commandes chainées, il faut d'abord créer un pipe pour la communication entre les deux commandes. Ensuite, il faut rediriger la sortie standard de la première commande vers l'entrée standard de la seconde commande.

```
int in_pipe = 0;          // Indique si un pipe a été détecté
int arg2_pos = 0;         // Position des arguments de la commande après le pipe
int pid_pipe;             // PID pour le second processus
char *argv2[ARGS_MAX];    // Arguments pour la commande après le pipe
int pipe_fd[2];           // Descripteurs pour le pipe
int res;                  // Résultat des appels système

arg_pos = 0;

// Séparation des commandes avant et après le pipe
while (tokens[arg_pos][0] != 0) {
  if (!in_pipe && !strcmp(tokens[arg_pos], "|")) {
    in_pipe = 1;        // Pipe détecté
    argv[arg_pos] = NULL; // Terminer les arguments de la première commande
  } else {
    if (in_pipe) {
      argv2[arg2_pos] = tokens[arg_pos]; // Arguments après le pipe
      ++arg2_pos;
    } else {
      argv[arg_pos] = tokens[arg_pos]; // Arguments avant le pipe
    }
  }
  ++arg_pos;
}

// Terminer les listes d'arguments
if (in_pipe)
  argv2[arg2_pos] = NULL; // Terminer les arguments de la seconde commande
else
  argv[arg_pos] = NULL; // Terminer les arguments de la première commande

// Fork pour gérer les processus
pid_child = fork();
if (!pid_child) { // Processus enfant
  if (in_pipe) {
    // Création du pipe
    if (pipe(pipe_fd) == -1) {
      perror("pipe");
      exit(EXIT_FAILURE);
    }

    pid_pipe = fork();
    if (!pid_pipe) { // Premier enfant (commande avant le pipe)
      close(pipe_fd[0]);             // Ferme la lecture
      dup2(pipe_fd[1], STDOUT_FILENO); // Redirige la sortie standard vers le pipe
```

```
      close(pipe_fd[1]);

      // Construire le nom du fichier exécutable
      strcpy(filename, argv[0]);
      strcat(filename, ".elf");

      res = execv(filename, argv); // Exécute la commande
      if (res == -1) {
        perror("execv");
        exit(EXIT_FAILURE);
      }
    } else { // Deuxième enfant (commande après le pipe)
      close(pipe_fd[1]);             // Ferme l'écriture
      dup2(pipe_fd[0], STDIN_FILENO); // Redirige l'entrée standard depuis le pipe
      close(pipe_fd[0]);

      // Construire le nom du fichier exécutable
      strcpy(filename, argv2[0]);
      strcat(filename, ".elf");

      res = execv(filename, argv2); // Exécute la commande
      if (res == -1) {
        perror("execv");
        exit(EXIT_FAILURE);
      }
    }
  } else {
    // Pas de pipe, exécute une seule commande
    strcpy(filename, argv[0]);
    strcat(filename, ".elf");

    if (execv(filename, argv) == -1) {
      perror("execv");
      exit(EXIT_FAILURE);
    }
  }
} else { // Processus parent
  waitpid(pid_child, NULL, 0);
}
```

# 2. Laboratoire sockets

Le laboratoire concerne la création d'un jeu de bataille navale en traitant les sujets de **sockets** ainsi que la gestion des **signaux**.

## 2.1. Sockets

### 2.1.1. Création d'un serveur TCP

Voici le code permettant de créer la partie serveur

```c
int server_create(const int port)
{
  int listenfd = 0, connfd = 0;
  struct sockaddr_in serv_addr;

  // Création de la socket
  listenfd = socket(AF_INET, SOCK_STREAM, 0);
  if (listenfd < 0) {
    return -1;
  }

  // Initialisation de l'adresse du serveur
  memset(&serv_addr, 0, sizeof(serv_addr));
  serv_addr = (struct sockaddr_in) {
    .sin_family = AF_INET,
    .sin_addr.s_addr = htonl(INADDR_ANY),
    .sin_port = htons(port),
  };

  // Association de la socket à l'adresse
  if (bind(listenfd, (struct sockaddr *)
&serv_addr, sizeof(serv_addr)) < 0) {
    close(listenfd);
    return -1;
  }

  // Mise en écoute de la socket
  // listen(listenfd, 1) correspond a la
longueur de la file d'attente des connexions
  if (listen(listenfd, 1) < 0) {
    close(listenfd);
    return -1;
  }

  printf("%s %d...\n", strings[SERVER_LISTEN],
port);

  // Acceptation d'une connexion
  connfd = accept(listenfd, (struct sockaddr
*) NULL, NULL);
  if(connfd < 0) {
    close(listenfd);
    return -1;
  }

  close(listenfd);

  printf("%s\n",
strings[SERVER_CLIENT_CONNECT]);

  return connfd;
}
```

1. Créer des variables contentant le file descripteur pour lire les connexions entrantes `listenfd` et pour gérer la connexion **spécifique** avec un client `connfd`.

2. **SOCKET**: créer la socket sur le fd `listenfd` dans le but de lire les tentatives de connexions.

3. Permet de définir la structure de l'adresse du serveur grâce à la `struct sockaddr_in`.
   - `sin_family` : spécifie IPv4.
   - `sin_addr.s_addr` : accepte les connexions sur n'importe quelle interface réseau (adresse locale) grâce à `htonl(INADDR_ANY)`.
   - `sin_port` : convertit le numéro de port en ordre réseau (big-endian) via htons(port).

4. **BIND**: l'appel à `bind` associe la socket représentée par le descripteur `listenfd` à une **adresse IP** et un **port** spécifiés dans la structure `serv_addr`. Cela permet au système d'identifier sur quelle interface réseau et sur quel port le serveur doit écouter les connexions.

5. **LISTEN**: l'appel à listen place la socket en mode écoute pour les connexions entrantes. Elle prépare le serveur à accepter des connexions client avec accept.

6. **ACCEPT**: l'appel à `accept` permet au serveur d'accepter une connexion entrante d'un client. Il utilisera le descripteur de fichier (`connfd`) pour la communication avec ce client spécifique.

7. On ferme le file descriptor `listenfd` car notre connexion est désormais établie et celui-ci ne sert plus. Une fois les affichages fait, on retourne donc la variable `connfd` qui permettera au serveur de discuter avec le client connecté.

# 3. Laboratoire ordonnanceur

Dans ce laboratoire il était demandé de pouvoir modifier la priorité des processus en utilisant une commande `renice <pid> <priority>`. Pour cela, il faut d'abord récupérer le PID du processus à modifier, puis utiliser modifier la variable `prio` contenue dans le pcb.

```
int do_renice(uint32_t pid, uint32_t new_prio) {
  //TO COMPLETE

  pcb_t *pcb;

  pcb = find_proc_by_pid(pid);

  pcb->main_thread->prio = new_prio;

  printk("Process %d has been reniced to %d\n", pid, new_prio);

  return 0;

}
```

Lancer l'application « time_loop » en mode continu, en arrière-plan (avec le symbole &) depuis le shell. Essayer d'exécuter des commandes dans le shell.

**Que se passe-t-il et pourquoi ?**

1. Le processus time_loop reste en mode ready car a la même priorité que le shell. De ce fait il va attendre et ne rien faire. Une fois que nous lançons une commande sans être en waiting cela va engendrer un waitpid et démarrera toutes les processus en mode ready dans l'ordre d'arrivée FIFO.

Changer la priorité du shell pour que le processus « time_loop » devienne plus prioritaire. Essayer d'éxecuter quelques commandes dans le shell.

**Que se passe-t-il et pourquoi ?**

2. Toutes les commandes démarrent immédiatement et automatiquement car auront une priorité plus grande que le shell. Cela s'applique même pour celle étant lancée sans le &.

Refaire la manipulation en mode bloquant, toujours en lançant l'application en background.

**Comparer les deux modes et expliquer leurs différences de comportement.**

3. En mode bloquant, le processus time_loop utilise une attente passive via une fonction usleep. Cette approche ne place pas le processus en état Waiting mais entraîne un changement de contexte, permettant à l'ordonnanceur de le placer en état Ready.