1. Rappel du squelette actuel

- **main()** : contient la boucle readline(), gère Ctrl+D, a un shell_env passé dans la boucle.
- **signal_setup** : installe un handler qui met à jour g_received_signal.
- **create_shell_env / destroy_shell_env** : gèrent la copie de l'env et l'allocation dynamique.
- **Une seule variable globale** g_received_signal.

Tout le reste (parsing/exécution/builtins/expansions) est à faire.

Ctrl c

Cltl d

2. Étapes et répartition des tâches 27

Étape 1 : Préparer la base pour le parsing

Étudiant A (Parsing)

1. **Définir la structure de donnée pour stocker la commande**
    - Ex. t_cmd ou t_cmd_list (liste chaînée ou tableau).
    - Chaque élément contient :
        - char **argv (liste d'arguments de la commande),
        - Informations de redirection (fichier in/out, <<, >>, etc.),
        - Pointeur vers la commande suivante si un | existe.
2. **Créer une fonction parse_command_line()** qui :
    - Découpe la ligne en tokens (attention aux quotes).
    - Détecte et enregistre les redirections (<, >, <<, >>).
    - Gère le pipe | pour séparer les commandes.
    - Retourne une liste de t_cmd (chaînée ou un tableau).

Pendant ce temps...

Étudiant B (Exécution / Environnement)

1. **Préparer un module "execution"** avec :
    - Un prototype execute_commands(t_cmd *cmds, t_shell_env *shell_env).
    - À ce stade, ça peut juste afficher les commandes reçues (« Debug print ») pour valider le parsing.
    - Mettre en place une fonction setup_redirects() vide ou semi-fictive pour plus tard, quand A aura défini la structure.
2. **Gérer l'environnement en interne**
    - Mettre à disposition des fonctions style env_get(var), env_set(var, value), env_unset(var).
    - Ceci prépare le terrain pour les builtins export/unset.

Synchronisation sur l'étape 1

- **Étudiant A** donne à **B** un fichier d'en-tête (par ex. parser.h ou un simple minishell.h) décrivant la structure t_cmd.

- **Étudiant B** appelle parse_command_line() dans le main, récupère la structure de commandes, et fait un **simple debug** (print).

Objectif : **valider** que la structure de parsing est lisible et exploitable.

Étape 2 : Intégrer la gestion des builtins de base 3 fevrier

Étudiant B :

1. **Créer une table de dispatch** des builtins. Exemple :
2. typedef struct s_builtin {
3. char *name;
4. int (*func)(char **argv, t_shell_env *shell_env);
5. } t_builtin;
6. t_builtin g_builtins[] = {
7. {"cd", &builtin_cd},
8. {"exit", &builtin_exit},
9. {"pwd", &builtin_pwd},
10. {"echo", &builtin_echo},
11. {"env", &builtin_env},
12. {"export", &builtin_export},
13. {"unset", &builtin_unset},
14. {NULL, NULL}
15. };
16. **Implémenter** au moins quelques builtins (au choix) :
    - echo (avec option -n),
    - pwd,
    - exit (pour tester la sortie).
17. **Créer une fonction** int is_builtin(char *cmd_name) et int exec_builtin(t_cmd *cmd, t_shell_env *shell_env) qui :
    - Compare le cmd->argv[0] au tableau de builtins,
    - Si trouvé, appelle la fonction correspondante,
    - Sinon, retourne 0 (pas un builtin).

Étudiant A :

- Continue d'améliorer le parsing si besoin (gestion des quotes correctement, etc.).
- Gère la **découpe en argv** pour chaque commande.
- Peut s'occuper de la **partie expansions** (si déjà validé) ou au moins commencer à l'implémenter pour $VAR, $?.

Synchronisation sur l'étape 2

- **A** doit s'assurer que t_cmd->argv[0] contient bien la commande (ex : ls, echo, etc.).
- **B** peut appeler if (is_builtin(cmd->argv[0])) exec_builtin(...) ; else exec_external(...).

Objectif : vérifier que certains builtins (ex : echo, pwd) sont fonctionnels même sans redirection, sans pipe.

Étape 3 : Implémenter l'exécution des commandes externes 10 fev

Étudiant B (Exécution)

1. **Créer exec_external()** :
    - Gérer la recherche dans le PATH, par ex. :
    - char *resolve_path(char *cmd, char **env);
    - // Renvoie "/bin/ls" si cmd = "ls", ou cmd si c'est un chemin absolu
    - Faire un fork(), dans le fils → execve(path_resolved, cmd->argv, shell_env->env).
    - Récupérer le code retour via waitpid() et l'enregistrer dans shell_env->exit_status.
2. **Gérer les erreurs** (commande introuvable, permission denied, etc.).

Pendant ce temps...

Étudiant A (Parsing)

- Peut commencer à **gérer les expansions** ($VAR, $?) au moment où il découpe la ligne.
- **Ne pas** oublier que les single quotes ' ' **désactivent** l'expansion, alors que les double quotes \" \" l'autorisent.
- Ajouter éventuellement la gestion **basique** des redirections (repérer > file, < file, etc.) dans la structure t_cmd.

Synchronisation sur l'étape 3

- **B** aura besoin d'avoir cmd->infile, cmd->outfile (ou un tableau de redirections) pour plus tard.
- **A** peut livrer un t_cmd où infile et outfile sont initialisés.
- **B** pour l'instant peut ignorer les redirections (ou stub) et juste exécuter la commande.
- **Test** : taper ls, ./a.out, etc. dans votre shell, voir si ça fonctionne.

Étape 4 : Gérer redirections et pipes 17 fev

Étudiant A :

1. Dans parse_command_line(), détecter :
    - > = redirection sortie (truncate)
    - >> = redirection sortie (append)
    - < = redirection entrée (fichier)
    - << = here_doc (lecture jusqu'à un délimiteur)
2. Stocker ces infos dans t_cmd (ex. t_redir type; char *filename;).
3. Si plusieurs commandes séparées par |, créer autant de t_cmd chainés (ou un tableau).

Étudiant B :

1. **Implémenter setup_redirects(cmd)**
    - Selon le type, ouvrir le fichier, faire dup2() sur STDOUT_FILENO ou STDIN_FILENO.
    - Gérer le cas << (ici-doc) : lire la saisie jusqu'au délimiteur (ou la stocker dans un pipe, etc.).
2. **Implémenter execute_pipeline(cmd_list, shell_env)** pour enchaîner les fork() + pipe().

Synchronisation sur l'étape 4

- **A** doit donner à **B** une structure claire pour les redirections (type, filename).
- **B** appelle setup_redirects avant l'exec (dans le fils).
- Testez des commandes comme ls > out.txt, cat < file, cat file | grep hello.

Étape 5 : Finaliser les builtins et la gestion de l'environnement 24 fev

Étudiant B :

- Implémente tous les builtins restants :
    - env : affiche shell_env->env,
    - export : insère/modifie une variable,
    - unset : supprime la variable,
    - cd : gère le changement de répertoire,
    - exit : met shell_env->running = 0 et éventuellement exit_status.

Étudiant A :

- Continue à peaufiner le **parsing des expansions** (cas $VAR inexistant, $?, etc.).
- Vérifie que export, unset ont un effet si on retape une commande qui exploite la variable ajoutée/supprimée.

Synchronisation sur l'étape 5

- **A** et **B** se mettent d'accord sur la forme de shell_env->env. B en aura besoin pour manipuler/exporter.
- **Test** : export MYVAR=hello, puis echo $MYVAR. Ensuite unset MYVAR, echo $MYVAR.

Étape 6 : Gérer les signaux restants et le code de retour 3

Étudiant B :

- Vérifier SIGQUIT (Ctrl+) pour les programmes en cours d'exécution (ex: cat).
    - Si un process fils est en cat, Ctrl+\ doit lui envoyer SIGQUIT. Dans votre shell, vous ignorez peut-être ou vous gérez différemment.
- Vérifier que g_received_signal est remis à 0 à chaque tour, et ajuster exit_status comme bash (130 pour SIGINT, 131 pour SIGQUIT, etc.).

Étudiant A :

- Tester tout particulièrement le comportement dans le parsing quand on tape un Ctrl+C **pendant** qu'un here_doc attend une entrée, etc.
    - Il se peut que vous deviez interrompre la lecture du here_doc quand SIGINT arrive.

Synchronisation sur l'étape 6

- Communiquer sur la gestion de signaux en cours d'exécution vs. en attente de commande.
- Contrôler que le exit_status se met bien à jour dans shell_env.

Étape 7 : Nettoyage, tests et corrections 17 mars

- **Tous les deux** :
    1. **Tester systématiquement** toutes les fonctionnalités (pipelines complexes, redirections multiples, expansions, etc.).
    2. **Corriger** les fuites mémoire (tester avec valgrind).
    3. **Respect de la Norme** (nommage, fonctions < 25 lignes, etc.).
    4. Éventuellement, commencer les **bonus** (&&, ||, wildcards *, etc.) si le mandatory est **parfait**.

3. En résumé : tableau de répartition

Étape	Étudiant A	Étudiant B	Synchronisation clé
1. Parsing (base)	- Définir structure t_cmd - parse_command_line() minimal (split)	- Créer execute_commands() squelette - Fct. d'environnement (env_get/set)	A -> donne à B la structure de commande. B -> debug print.
2. Builtins (début)	- Améliorer parsing (quotes, expansions basiques)	- Table de dispatch g_builtins[] - Implémenter echo, pwd, exit	Vérifier que cmd->argv[0] arrive intact pour repérer la builtin.
3. Exécution externe	- Gérer expansions plus fines ($VAR, $?), single quote vs double quotes	- exec_external(): recherche PATH, fork(), execve() - Gérer erreurs	A -> expansions correctes B -> exécuter la commande si pas builtin.
4. Redirections / Pipes	- Parsing : détecter <, >, <<, >>, `	 - Stocker infos danst_cmd`	- setup_redirects() - execute_pipeline() (plusieurs fork et pipe())
5. Builtins avancées	- Affiner expansions / parsing (cas spéciaux)	- Implémenter env, export, unset, cd - Interaction avec shell_env->env	Tester export MYVAR=... et echo $MYVAR, etc.
6. Signaux et code retour	- Gérer Ctrl+C pendant un here_doc - Vérifier parsing interrompu	- Ajuster g_received_signal pour SIGINT, SIGQUIT - Mettre exit_status correct	Contrôler le comportement si on interrompt un programme ou un here_doc.
7. Finalisation / Tests	- Tester, debugger, corriger - Respect norme, no leaks	- Tester, debugger, corriger - Respect norme, no leaks	Validation conjointe du mandatory, envisager le bonus ensuite.

4. Conseils de collaboration au quotidien

1. **Pull Requests** ou **Merge Requests** :
    - Quand l'un finit une étape, l'autre la revoit avant de "merger".

2. **Points de synchro** :
    - Brief quotidien sur ce qui a été fait, ce qui reste à faire, blocages.
3. **Tests unitaires** :
    - Dès qu'A a fini un parsing, B fait un mini test.
    - Dès que B fait un builtin, A teste, etc.

En suivant ce plan, vous saurez **qui fait quoi** et **dans quel ordre**. L'important est de **valider à chaque étape** pour éviter d'avoir à tout refondre plus tard.

Bon courage pour la suite !

Bien sûr ! Voici quelques suggestions spécifiques pour améliorer le document et le rendre plus agréable à lire pour le lecteur :

1. **Utilisation de titres et sous-titres** :
    - Par exemple, au lieu de simplement écrire "1. Rappel du squelette actuel", vous pouvez ajouter un titre plus accrocheur comme "## 1. Aperçu du Squelette Actuel". Cela rendra la section plus visible et structurée.
2. **Ajout de listes à puces** :
    - Pour les sections détaillant les étapes, utilisez des listes à puces pour une meilleure lisibilité. Par exemple, dans la section "Étudiant A (Parsing)", vous pouvez structurer les informations comme suit :
    - - Définir la structure de donnée pour stocker la commande :
    - - Ex. `t_cmd` ou `t_cmd_list` (liste chaînée ou tableau).
    - - Chaque élément contient :
    - - `char **argv` (liste d'arguments de la commande),
    - - Informations de redirection (fichier in/out, `<<`, `>>`, etc.),
    - - Pointeur vers la commande suivante si un `|` existe.
3. **Utilisation de tableaux pour la répartition des tâches** :
    - Dans la section "En résumé : tableau de répartition", vous pouvez utiliser un tableau pour clarifier la répartition des tâches entre les étudiants. Par exemple :
    - | **Étape** | **Étudiant A** | **Étudiant B** | **Synchronisation clé** |
    - |-----------|----------------|----------------|-------------------------|
    - | 1. Parsing (base) | Définir structure `t_cmd`
 `parse_command_line()` minimal (split) | Créer `execute_commands()` squelette
 Fct. d'environnement (`env_get/set`) | A -> donne à B la structure de commande. B -> debug print. |
4. **Ajout de transitions et de connecteurs** :
    - Utilisez des phrases de transition pour relier les sections et les idées. Par exemple, après avoir décrit une étape, vous pouvez ajouter une phrase comme "Pendant ce temps, l'Étudiant B peut se concentrer sur...".
5. **Mise en évidence des points clés** :

- Utilisez du texte en gras pour mettre en évidence les points importants. Par exemple, "Objectif : **valider** que la structure de parsing est lisible et exploitable".

En appliquant ces suggestions, vous pouvez améliorer la lisibilité et l'attrait de votre document pour les lecteurs. Si vous avez besoin de plus de détails ou d'autres suggestions, n'hésitez pas à demander ! 😊

: Document actuel : Document actuel : Document actuel : Document actuel