Snake GAME

Section .text:

1. Partie init:

```
section .text
init:
    push rbp
    mov rdi, hidecursor
    call _printf
    xor rdi, rdi
    call _fflush
```

`push rbp` : Sauvegarde l'ancien cadre de pile (frame pointer).
`mov rdi, hidecursor` : Charge l'adresse de `hidecursor` (code pour cacher le curseur).
`call _printf` : Appelle `printf` pour exécuter cette commande.
`xor rdi, rdi` : Met `rdi` à 0 (équivalent à `mov rdi, 0` mais plus rapide).
`call _fflush` : Vide le buffer de sortie pour que la commande s'affiche immédiatement.

```
    ;switch to console mode, disable echo
    mov rdi, 0
    mov rsi, oldt
    call _tcgetattr
```

`_tcgetattr` est appelé pour obtenir les attributs du terminal actuel et les stocker dans `oldt`.

```
    mov rdi, newt
    mov rsi, oldt
    mov rcx, 64
    rep movsb
```

On copie `oldt` dans `newt` pour modifier les nouveaux attributs du terminal.
Cette instruction **copie `rcx` octets** de `oldt` à `newt` pour configurer le terminal.

```
  and word [newt + 3 *8], ~(0x0100 | 0x0008);
```

```
    mov rdi, 0
    mov rsi, 0
    mov rdx, newt
    call _tcsetattr
    pop rbp
    ret
```

- Modifie les paramètres de `newt` pour désactiver **l'echo des touches** et **le mode canonique** (nécessaire pour capturer immédiatement les touches pressées sans besoin d'appuyer sur "Entrée").
- **Applique les nouveaux paramètres** du terminal avec `_tcsetattr`, désactivant l'echo et permettant la lecture instantanée des touches.

2. Fonction exit:

```
exit:
    mov rdi , showcursor
    call _printf
    xor rdi, rdi
    call _fflush
```

Réaffiche le curseur lorsque le jeu se termine.

```
    ;restore terminal mode
    mov rdi, 0
    mov rsi, 0
    mov rdx, oldt
    call _tcsetattr
```

Rétablit les anciens paramètres du terminal (`oldt`) pour que l'affichage et l'input reviennent à la normale.

```
    mov rax, 60
    xor rdi, rdi
    syscall
```

Quitte le programme proprement (`syscall 60` est l'instruction de sortie sous Linux).

3. Fct render_table:

```
render_table:
    push rbp
    mov rdi, buf
    mov   rax, '┌'
    stosd
    dec rdi
    mov rcx, COLS
    mov   rax, '─'
```

Sauvegarde le registre de base (`rbp`) pour conserver le contexte d'exécution.
Charge `buf` (la mémoire où sera stocké l'affichage du jeu).
Place le caractère `┌` (coin supérieur gauche du terrain).
Décalage du pointeur `rdi`.
Charge le nombre de colonnes (`COLS = 60`).
Remplit la ligne du haut avec le caractère `─` (barre horizontale).

4. Dessin des bords :

- _r0:

```
_r0:
    stosd
    dec rdi
    dec rcx
    jnz _r0
    mov   rax, '┐'
    stosd
    mov byte [ rdi - 1], 10 ;new line

    ;mid line
    mov rdi, ROWS
```

Place **le coin supérieur droit (`┐`)**.
Ajoute un **saut de ligne** (`10` en ASCII).

- _r1:

```
_r1:
    mov   rax, '│'
    stosd
    dec rdi
    mov rcx, COLS
    mov ax, ''
    rep stosw
    mov   eax, '│'
    stosd
    mov byte [rdi - 1], 10
    dec rsi
    jnz _r1

    ;bottom line
    mov   rax, '└'
    stosd
    dec   rdi
    mov   rcx, COLS
    mov   rax, '─'
```

Dessin des bords verticaux (│).

Remplit l'intérieur avec des espaces.

Répète ce processus pour chaque ligne.

Ajoute **le coin inférieur gauche (└)**.

Remplit la dernière ligne avec ─ pour fermer le cadre.

- _r2:

```
_r2:
    stosd
    dec rdi
    dec rcx
    jnz _r2
    mov rax, '┘'
    mov   byte [rdi - 1], 10 ; new line


    mov rdi, buf
    call _printf

    mov rdi, cursortotop
    mov rsi, ROWS + 2
```

```
    call _printf



    pop rbp
    ret
```

Termine par **le coin inférieur droit (┘)**.

Affiche la grille en appelant `_printf`.

Remonte le curseur pour éviter d'écrire de nouvelles lignes à chaque frame.

5. Fct main:

```
_main:
    push rbp
    call init
```

Initialisation du jeu en appelant `init`.

- main_loop:

```
main_loop:
    call render_table
    mov   qword [tail], 0
    mov   qword [head], 0
    mov   qword [x], COLS / 2
    mov   qword [y], ROWS / 2
    mov   qword [xdir], 1
    mov   qword [ydir], 0
    mov   qword [applex], -1
```

Affiche la grille (`render_table`).
Initialise la **tête et la queue du serpent** (`head = 0`, `tail = 0`).
Place **le serpent au centre** (`x = COLS / 2`, `y = ROWS / 2`).
Initialise la direction de déplacement vers **la droite** (`xdir = 1`, `ydir = 0`).
Définit une valeur négative (`-1`) pour `applex` pour indiquer qu'aucune pomme n'existe encore.

- loop:

```
loop:
    lea   rbp, [data]
    cmp   qword [applex], 0
    jge   apple_exists

    ; Create new apple
    call  _rand
    xor   rdx, rdx
    mov   rbx, COLS
    div   rbx
    mov   [applex], rdx
    call  _rand
    xor   rdx, rdx
    mov   rbx, ROWS
    div   rbx
    mov   [appley], rdx

    ; New apple on the snake?
    mov   rdi, [head]
    mov   rax, [applex]
    mov   rbx, [appley]
    mov   rsi, [tail]
```

Charge l'adresse de `data` dans `rbp` pour faciliter l'accès aux variables.
Vérifie si une pomme existe déjà (`applex >= 0`).

Si oui, saute à `apple_exists` et évite de générer une nouvelle pomme.

Génère une position aléatoire pour `applex` (colonne) et `appley` (ligne).
Utilise `_rand` pour obtenir un nombre aléatoire.
Fait un **modulo** (`div rbx`) pour s'assurer que la pomme reste **dans les limites** du terrain.

Elle copie la valeur stockée à l'adresse mémoire `head` dans le registre `rdi`. On utilise souvent `rdi` comme premier argument lors d'un appel de fonction en convention de passage des arguments sous x86-64.

Elle charge la valeur de `applex` dans le registre `rax`. Cela pourrait représenter, par exemple, une coordonnée X d'un objet comme une pomme dans un jeu de serpent.

Elle copie la valeur de `appley` (la coordonnée Y de la pomme) dans `rbx`.

Elle charge dans `rsi` la valeur de `tail`, qui pourrait représenter, par exemple, la fin du corps du serpent dans un jeu.

6. Vérification de placement de la pomme:

```
q3:
    cmp   rsi, [head]
    jz    q5
    cmp   [rbp + (x - data) + rsi * 8], rax
    jnz   q4
    cmp   [rbp + (y - data) + rsi * 8], rbx
    jnz   q4
    mov   qword [applex], -1
q4:
    inc   rsi
    and   rsi, 1023
    jmp   q3
q5:

    ; Draw apple
    cmp   qword [applex], 0
    jl    apple_exists
    mov   rdi, applestr
    mov   rsi, [appley]
    mov   rdx, [applex]
    inc   rsi
    inc   rdx
    call  _printf
    mov   rdi, cursortotop2
    mov   rsi, [appley]
    inc   rsi
    call  _printf
```

Vérifie si **la pomme est placée sur le serpent** :
Compare chaque segment du serpent avec la position de la pomme.
Si la pomme est sur le serpent, on **réinitialise `applex = -1`**, ce qui force la génération d'une nouvelle pomme.
Affiche la pomme (❤) si elle a bien été générée.
Incrémente `rsi` et `rdx` car l'affichage utilise une base 1.
Remonte le curseur après avoir dessiné la pomme.

- La pomme existe:

```
apple_exists:

    ; Clear snake tail
    mov   rbx, [tail]
    mov   rdi, tailstr
    mov   rsi, [rbp + (y - data) + rbx * 8]
    mov   rdx, [rbp + (x - data) + rbx * 8]
    inc   rsi
    inc   rdx
    call  _printf

    mov   rbx, [tail]
    mov   rdi, cursortotop2
    mov   rsi, [rbp + (y - data) + rbx * 8]
    inc   rsi
    call  _printf

    ; Eat apple?
    mov   rbx, [head]
    mov   rax, [rbp + (x - data) + rbx * 8]
    cmp   eax, [applex]
    jnz   not_on_apple
    mov   rax, [rbp + (y - data) + rbx * 8]
    cmp   eax, [appley]
    jnz   not_on_apple

    mov   qword [applex], -1
    jmp   apple_is_eaten
```

Efface l’ancienne position de la queue en remplaçant l'affichage du dernier segment
Récupère **les coordonnées de la queue du serpent (`tail`)**.
Supprime visuellement le dernier segment du serpent en l’écrasant par un espace vide.
Ajuste l'affichage du curseur pour éviter des erreurs visuelles.
Récupère **les coordonnées de la tête** du serpent.
Vérifie si la tête du serpent **est à la même position que la pomme**.
Si les **coordonnées ne correspondent pas**, il saute à `not_on_apple`.
Si le serpent **mange la pomme**, on réinitialise `applex = -1` pour générer une nouvelle pomme.

- La pomme n’existe pas:

```
not_on_apple:
    ; Move tail
    mov   rbx, [tail]
    inc   rbx
    and   rbx, 1023
    mov   [tail], rbx
apple_is_eaten:

    ; Move head
    mov   rbx, [head]
    mov   rax, rbx
    inc   rbx
    and   rbx, 1023

    mov   rcx, [rbp + (x - data) + rax * 8]
    add   rcx, [xdir]
    cmp   rcx, COLS
    jb    ok0
    jge   o1
    add   rcx, COLS
    jmp   ok0
```

Déplace la queue en avançant l’indice de `tail`.
Applique un masque `& 1023` pour éviter de dépasser la mémoire allouée au serpent.
Incrémente l’indice de `head` pour avancer la tête du serpent.
Calcule la nouvelle position en X (`rcx = ancienne position X + direction X`).
Si le serpent **dépasse les limites**, il effectue un **wrap-around** (téléportation de l’autre côté de l’écran).
Elle charge dans `rdx` un élément à une position calculée à partir de `rbp`, d’un décalage et d’un indice (une coordonnée en X multipliée par 8 pour accéder à la bonne colonne).
Elle modifie la valeur de `rdx` en ajoutant la direction verticale (`ydir`), simulant un déplacement dans la direction Y.
Elle compare la nouvelle position Y (`rdx`) avec la limite supérieure de la grille (le nombre total de lignes). Cela sert à vérifier si le déplacement dépasse la frontière inférieure de la grille.
Si `rdx` est inférieur à `ROWS`, le programme saute à l’étiquette `ok2`. Cela signifie que le déplacement reste dans les limites de la grille.
Si `rdx` est supérieur ou égal à `ROWS`, il saute à l’étiquette `o2`. Cela gère les cas de dépassement des limites (débordement).
si la valeur dépasse la limite inférieure, elle "revient" en haut de la grille (comme dans certains jeux où le serpent passe du bas de l'écran vers le haut).

Après avoir corrigé un dépassement des limites, le programme saute directement à `ok2` pour continuer l'exécution normale.

- o1,ok0,o2,ok2:

```
o1:
    sub   rcx, COLS
ok0:
    mov   [rbp + (x - data) + rbx * 8], rcx

    mov   rdx, [rbp + (y - data) + rax * 8]
    add   rdx, [ydir]
    cmp   rdx, ROWS
    jb    ok2
    jge   o2
    add   rdx, ROWS
    jmp   ok2
o2:
    sub   rdx, ROWS
ok2:
    mov   [rbp + (y - data) + rbx * 8], rdx
    mov   [head], rbx

    ; Check gameover
    mov   rdi, [head]
    mov   rax, [rbp + (x - data) + rdi * 8]
    mov   rbx, [rbp + (y - data) + rdi * 8]
    mov   rsi, [tail]
```

Calcule la nouvelle position en Y (`rdx = ancienne position Y + direction Y`).
Applique également un **wrap-around vertical** si nécessaire.
Met à jour **la nouvelle position Y** de la tête du serpent.
Stocke **le nouvel indice de `head`** après le déplacement.

- r3,r4,r5:

```
r3:
    cmp   rsi, [head]
    jz    r5
    cmp   [rbp + (x - data) + rsi * 8], rax
    jnz   r4
    cmp   [rbp + (y - data) + rsi * 8], rbx
```

```
    jz    gameover
r4:
    inc   rsi
    and   rsi, 1023
    jmp   r3
r5:

    ; Draw head
    mov   rbx, [head]
    mov   rdi, headstr
    mov   rsi, [rbp + (y - data) + rbx * 8]
    mov   rdx, [rbp + (x - data) + rbx * 8]
    inc   rsi
    inc   rdx
    call  _printf
    mov   rdi, cursortotop2
    mov   rbx, [head]
    mov   rsi, [rbp + (y - data) + rbx * 8]
    inc   rsi
    call  _printf
    xor   rdi, rdi
    call  _fflush

    ; Delay
    mov   rdi, 5 * 1000000 / 60
    call  _usleep

    ; read keyboard
    mov   qword [fds], 1
    mov   rdi, 1
    mov   rsi, fds
    mov   rdx, 0
    mov   rcx, 0
    mov   qword [tv], 0
    mov   qword [tv + 8], 0
    mov   r8, tv
    call  _select
    test  rax, 1
    jz    nokey

    call  _getchar
    cmp   al, 27
```

```
    jz    exit
    cmp   al, 'q'
    jz    exit

    cmp   al, 'h'
    jnz   noth
    cmp   qword [xdir], 1
    jz    noth
    mov   qword [xdir], -1
    mov   qword [ydir], 0
```

Compare la tête du serpent avec toutes les parties de son corps (`tail → head`).
Si les coordonnées `x, y` de la tête correspondent à une **partie du corps**, il saute à `gameover` (collision détectée).

- Place **la nouvelle position de la tête (▒)** après le déplacement.
- Ajuste **l'affichage du curseur**.
- **Vide le buffer d'affichage** pour s'assurer que l'affichage est immédiat.
- Fait une **pause (`usleep`)** pour ralentir le jeu et éviter que le serpent ne bouge trop vite.

Vérifie **s'il y a une touche en attente** (`_select`).
Si aucune touche n'est pressée, il saute à `nokey`.
Lit une touche du clavier (`_getchar`).
Si la touche pressée est **`ESC (27)` ou `q`**, il appelle `exit` pour quitter le jeu.

Le jeu utilise les **touches `h`, `l`, `j`, `k`** pour changer la direction du serpent, en suivant la disposition **Vim** :

- `h` → Gauche
- `l` → Droite
- `j` → Bas
- `k` → Haut

Vérifie si `h` a été pressé.
Empêche le demi-tour immédiat (`xdir` ne peut pas être `1`).
Déplace le serpent vers la gauche (`xdir = -1`, `ydir = 0`).

- noth,notl,notj:

```
noth:
    cmp   al, 'l'
    jnz   notl
    cmp   qword [xdir], -1
    jz    notl
    mov   qword [xdir], 1
    mov   qword [ydir], 0
```

Vérifie si `l` a été pressé.
Empêche le demi-tour immédiat (`xdir` ne peut pas être `-1`).
Déplace le serpent vers la droite (`xdir = 1`, `ydir = 0`).

```
notl:
    cmp   al, 'j'
    jnz   notj
    cmp   qword [ydir], -1
    jz    notj
    mov   qword [xdir], 0
    mov   qword [ydir], 1
```

Vérifie si `j` a été pressé.
Empêche le demi-tour immédiat (`ydir` ne peut pas être `-1`).
Déplace le serpent vers le bas (`xdir = 0`, `ydir = 1`).

```
notj:
    cmp   al, 'k'
    jnz   notk
    cmp   qword [ydir], 1
    jz    notk
    mov   qword [xdir], 0
    mov   qword [ydir], -1
```

Vérifie si `k` a été pressé.
Empêche le demi-tour immédiat (`ydir` ne peut pas être `1`).
Déplace le serpent vers le haut (`xdir = 0`, `ydir = -1`).

```
nokey:
    jmp   loop
```

Si aucune touche directionnelle n'a été pressée, **le jeu continue** sans changer de direction.

7. Game over:

```
gameover:
    ; Show gameover
    mov   rdi, gameoverstr
    mov   rsi, ROWS / 2
    mov   rdx, COLS / 2 - 5
    call  _printf
    mov   rdi, cursortotop2
    mov   rsi, ROWS / 2
    call  _printf

    call  _getchar
    jmp main_loop
```

- **Affiche le message `Game Over`** au centre de l'écran.
- Ajuste **l'affichage du curseur** pour bien centrer le message.
- Attend **une entrée clavier** avant de **relancer une nouvelle partie** (`jmp main_loop`).

Résumé du Fonctionnement du Jeu

1. **Initialisation**
    - Configure le terminal (`init`).
    - Cache le curseur.
    - Initialise le serpent au centre.
2. **Boucle du jeu**
    - Efface la queue du serpent.
    - Vérifie si une pomme existe :
        - Sinon, en génère une nouvelle.
        - Vérifie qu'elle n'apparaît pas sur le serpent.
    - Vérifie si le serpent **mange la pomme**.
    - Déplace la **tête du serpent**.
    - Vérifie **les collisions** (Game Over si le serpent se mord).
    - Affiche la **nouvelle position** du serpent.
3. **Gestion du clavier**
    - Vérifie si une **touche est pressée** (`h, l, j, k` pour changer de direction).
    - `q` ou `ESC` → Quitte le jeu.
4. **Game Over**
    - Affiche `Game Over` et attend une entrée clavier avant de recommencer.