Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Appels de fonctions et gestion de la pile d'exécution

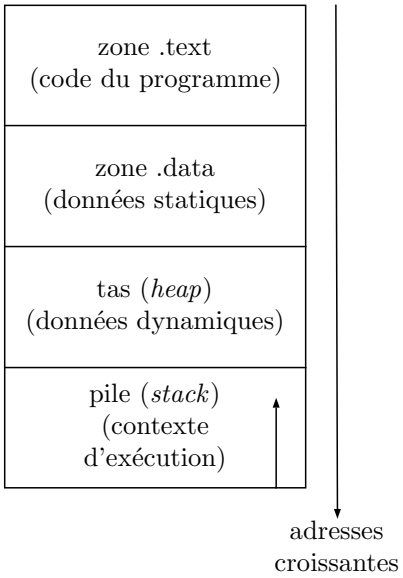
Introduction

Le but de cette séance est d'apprendre à programmer et appeler des fonctions en assembleur, et de comprendre les conventions de gestion des registres et de la mémoire imposées par l'ABI (*Application-Binary Interface* : ensemble de conventions permettant aux différentes parties du programme de communiquer, quelques-soient leurs langages de développement respectifs) et la bibliothèque C que l'on utilise pour écrire nos programmes en assembleur. En respectant ces conventions, on verra qu'il est possible de mélanger du code C et assembleur sans difficulté dans le même programme.
On se restreindra dans cette séance à l'architecture x86_64 que l'on va utiliser en TP. L'architecture x86_32 est basée sur des conventions très différentes : elles seront présentées avant le mini-projet constituant la fin du module. L'intérêt d'étudier les deux architectures est de mettre en parallèle deux philosophies très différentes : les conventions x86_32 sont typiques d'une architecture CISC alors que celles du x86_64 se rapprochent d'une architecture RISC.
L'ABI de l'architecture x86_64 est un document complexe à lire (que l'on met néanmoins à votre disposition sur la page principale du cours) qui a pour but de définir les règles que doivent respecter les compilateurs et assembleurs, dans toutes les situations possibles, y compris certaines dépassant largement le but de ce cours. Les conventions présentées ci-dessous sont donc un sous-ensemble volontairement simplifié des conventions détaillées dans l'ABI, mais elles ont été sélectionnées de façon à être compatibles avec l'ensemble complet.

Structure de la pile d'exécution d'un processus

Introduction

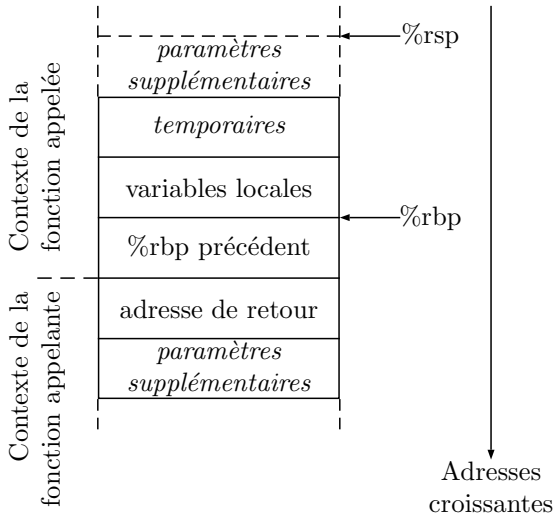
On a vu qu'un processus a une structure ressemblant au schéma suivant :



On a détaillé le contenu des zones `text` et `data` qui contiennent respectivement le code et les données statiques. Le tas est la zone dans laquelle les fonctions d'allocation dynamique de mémoire prennent l'espace mémoire à réserver : sa gestion interne sera détaillée dans le cours de système. On va étudier en détail dans cette séance la structuration de la dernière zone : la pile d'exécution.

Définition d'un cadre de pile

La pile est la zone mémoire contenant le contexte d'exécution des fonctions du programme. A chaque fois que le programme commence l'exécution d'une fonction, on doit mettre en place un cadre de pile (*stack frame* en anglais) qui contiendra notamment les variables locales de la fonction, les paramètres passés à la fonction, etc. Un cadre de pile a la structure suivante :



Dans ce schéma, les adresses sont croissantes vers le bas mais les valeurs sont empilées en haut car il s'agit d'une pile au sens algorithmique du terme. Les différentes zones (en partant du bas du schéma) sont détaillées ci-dessous :

- les paramètres supplémentaires sont ceux passés à la fonction en cours d'exécution : on parle de paramètres « supplémentaires » car les 6 premiers paramètres seront passés dans des registres ;
- l'adresse de retour est l'adresse de l'instruction suivant l'appel de la fonction en cours d'exécution dans la fonction appelante (i.e. c'est là qu'on revient quand on exécute `return` à la fin de la fonction) ;
- le `%rbp` « précédent » est la sauvegarde du pointeur de base de la fonction appelante (on détaillera son rôle plus bas) ;
- les variables locales de la fonction en cours d'exécution sont localisées dans son cadre de pile, à des adresses fixes par rapport à `%rbp` ;
- on peut éventuellement utiliser de la place dans le cadre de pile pour sauvegarder des registres ou des valeurs temporaires si besoin ;
- les contextes d'exécution s'empilent les uns au dessus des autres lors des appels de fonctions, donc on retrouve la même structure en haut du schéma si la fonction appelée appelle elle-même une autre fonction.

Les registres pointeurs

L'architecture x86_64 comprend 2 registres dont le rôle est lié à la pile d'exécution : `%rsp` et `%rbp`.
`%rsp` est le pointeur de pile : il contient en permanence l'adresse de la dernière case occupée dans la pile d'exécution. On ne le manipule pas directement en général pour éviter de risquer de déséquilibrer la pile.
`%rbp` est le pointeur de base : il contient l'adresse de la base du contexte d'exécution de la fonction en cours d'exécution. Dans notre cas, il pointera en permanence sur la case dans laquelle on a sauvegardé le `%rbp` précédent. On se sert de `%rbp` pour accéder aux variables locales et aux paramètres de la fonction en cours d'exécution.

Appels de fonctions

L'instruction `call` permet d'appeler une fonction : on l'utilise en précisant l'étiquette correspondant au début de la fonction à appeler, par exemple `call pgcd`. Cette instruction a le même comportement qu'un branchement inconditionnel (`jmp`) mais en plus, elle empile automatiquement l'adresse de retour dans la pile d'exécution. L'adresse de retour est simplement l'adresse de l'instruction suivant le `call` dans la fonction appelante.
L'instruction réciproque s'appelle `ret` : elle dépile l'adresse de retour et revient donc à l'instruction suivant le `call`. C'est l'équivalent de l'instruction `return` qu'on connait en C par exemple.
Comme on l'a déjà vu dans les exemples des séances précédentes, on renvoie par convention le résultat de la fonction dans le registre `%rax` ou une de ses fractions (i.e. `%eax`, `%ax` ou `%al`) si le résultat est sur 32, 16 ou 8 bits. On ne traitera pas dans le cadre de ce cours le cas particulier des fonctions renvoyant une valeur de taille supérieure à 64 bits.

Mise en place du cadre de pile

Le cadre de pile d'une fonction est mis en place en deux temps :

- dans la fonction appelante, qui copie les éventuels paramètres supplémentaires qu'elle va passer à la fonction appelée et empile l'adresse de retour en exécutant `call` ;
- dans la fonction appelée, qui réserve la place nécessaire à son contexte d'exécution.

Lorsqu'on écrit le code d'une fonction, on commence par l'instruction `enter` qui a pour rôle de réserver l'espace nécessaire au contexte d'exécution. Jusqu'à présent, on l'a toujours utilisée sous la forme `enter $0, $0` mais le premier paramètre sera en général différent de zéro, et précise justement le nombre d'octets à réserver pour le contexte (le deuxième paramètre sera toujours 0 dans notre cas). L'instruction `enter $N, $0` est en fait équivalente à la suite d'instructions suivante :

```
subq $8, %rsp
movq %rbp, (%rsp)
movq %rsp, %rbp
subq $N, %rsp
```

Comme `%rsp` pointe par convention sur la dernière case occupée, on doit commencer par le décrémenter de 8 avant de pouvoir sauvegarder `%rbp` dans la pile : c'est la sauvegarde du `%rbp` précédent dans le schéma du cadre de pile ci-dessus. Ensuite, on fait pointer `%rbp` sur la case où on a sauvegardé son ancienne valeur, et on décrémente `%rsp` de N octets : c'est cette opération qui réserve la place nécessaire au cadre de pile. On rappelle que la pile « descend » vers les adresses décroissantes, d'où le fait qu'on effectue des soustractions.
La partie délicate de la mise en place du cadre de pile est le calcul de la valeur de N, c'est à dire combien d'octets on doit réserver pour le contexte d'exécution de la fonction appelée. On doit donc :

1. calculer la place nécessaire pour toutes les variables locales ;
2. y ajouter éventuellement la place nécessaire pour les sauvegardes temporaires ;
3. y ajouter ensuite la place **maximale** nécessaire pour placer les paramètres de tous les appels de fonctions effectués dans la fonction courante ;
4. arrondir finalement la valeur au multiple de 16 immédiatement supérieur.

Pour le point 1, il suffit de faire la somme des tailles des variables locales. On note bien que l'ordre et le placement des variables à l'intérieur de la zone qui leur est reservée n'a pas d'importance : il suffit d'être cohérent lors des accès à ces variables.
Pour le point 2, il faut simplement calculer la taille nécessaire au cas où la fonction a besoin de sauvegarder temporairement des registres : en pratique, ça sera rarement le cas, donc cette valeur sera souvent nulle.
Pour le point 3, il faut calculer la taille maximale des paramètres supplémentaires pour chaque appel de fonction réalisé dans la fonction courante, c'est à dire :

- que si la fonction courante n'appelle aucune autre fonction, ou (cas très courant) si elle n'appelle que des fonctions avec moins de 6 paramètres, alors la taille calculée sera 0 ;
- si par contre la fonction courante appelle, par exemple, une fonction avec 7 paramètres sur 64 bits chacun, et une autre fonction avec 9 paramètres sur 64 bits chacun, la taille à réserver sera $max((7-6)\times 8, (9-6)\times 8) = 24$ octets.

Le point 4 enfin n'est strictement nécessaire que si on appelle des fonctions de la librairie C depuis la fonction courante. En pratique, il est plus simple de systèmatiquement faire l'alignement sans se poser de question.
L'instruction réciproque de `enter` s'appelle `leave`. Elle s'utilise sans paramètre et a pour effet de détruire le cadre de pile mis en place par `enter`. Elle est équivalente à la séquence d'instructions :

```
movq %rbp, %rsp
movq (%rsp), %rbp
addq $8, %rsp
```

Elle commence par faire pointer `%rsp` sur la case contenant l'ancienne valeur de `%rbp` : c'est comme cela qu'on détruit le cadre de pile (la mémoire n'est bien sûr pas « vidée » ; et contient toujours les valeurs précédentes, mais ces valeurs ne doivent plus être utilisées par le programme). Ensuite, on restaure l'ancienne valeur de `%rbp` en le dépilant.

Passage de paramètres

Une fonction qui souhaite en appeler une autre en lui passant des paramètres doit les copier dans des registres (ou sur la pile s'il y a plus de 6 paramètres) avant d'exécuter l'instruction `call`.
L'ABI de l'architecture x86_64 définie une association précise entre le numéro du paramètre (dans l'ordre de déclaration), sa taille et le registre à utiliser, qu'on détaille dans le tableau ci-dessous :

Numéro du paramètre	**Registre**			
	64 bits	**32 bits**	**16 bits**	**8 bits**
#1	%rdi	%edi	%di	%dil
#2	%rsi	%esi	%si	%sil
#3	%rdx	%edx	%dx	%dl
#4	%rcx	%ecx	%cx	%cl
#5	%r8	%r8d	%r8w	%r8b
#6	%r9	%r9d	%r9w	%r9b
#7 et +	pile			

A partir du septième paramètre, on doit utiliser la pile : ce sont les paramètres supplémentaires dont on a parlé plus haut. Ces paramètres seront donc copiés dans le contexte de la fonction appelante, avant d'exécuter l'instruction `call`. On pourrait calculer l'adresse de ces paramètres en se basant sur `%rbp` comme pour les variables locales, mais il est plus simple d'accéder à ces paramètres en se basant sur la valeur du pointeur de pile `%rsp`, car on n'a alors pas besoin de prendre en compte la taille des variables locales. Par convention, les paramètres supplémentaires sont copiés dans l'ordre des adresses croissantes, comme illustré dans le tableau ci-dessous :

Numéro du paramètre	**Adresse dans la pile** *(contexte de la fonction appelante)*
#7	%rsp
#8	%rsp + 8
#9	%rsp + 16
...	

La taille du paramètre n'influe pas le placement dans la pile, tant qu'elle reste inférieure ou égale à 64 bits (en pratique, on ne passera jamais de paramètre de taille supérieure à 8 octets : on préfèrera alors passer un pointeur sur les données de grande taille). Cela signifie que si on passe un caractère comme septième paramètre et un entier sur 32 bits comme huitième paramètre, ils seront bien respectivement aux adresses `%rsp` et `%rsp + 8`. Dis autrement, on gâche de la place dans la pile en alignant systématiquement les paramètres sur des adresses multiples de 8, mais c'est nécessaire à cause de conventions définies dans l'ABI.

Récupération des paramètres

Dans la fonction appelée, on peut directement utiliser les 6 premiers paramètres passés dans des registres. On rappelle qu'en C, **les paramètres sont toujours passés par copie** : cela signifie que la fonction appelante a dû recopier dans les 6 registres les valeurs qu'elle souhaitait passer en paramètre, et qu'on peut donc modifier ces registres librement car la fonction appelante n'a pas le droit de considérer que ces registres seront préservés au retour de la fonction appelé.
Cette procédure d'appel a une conséquence importante : **si la fonction appelée appelle elle-même une autre fonction , elle doit commencer par sauvegarder dans la pile tous les registres contenant des paramètres**. En effet, la fonction qu'elle va appeler peut parfaitement détruire les registres contenant les paramètres. Dans le cas d'une fonction feuille (c'est à dire qui n'appelle aucune autre fonction) ou bien si on est sûr qu'on n'aura plus besoin des paramètres dans la suite de la fonction, on peut se passer de cette sauvegarde.
Si la fonction a plus de 6 paramètres, l'accès aux paramètres supplémentaires est un peu plus compliqué, car on doit là encore calculer leur adresse dans la pile. Dans ce sens, on se base sur `%rbp` et on trouvera les paramètres aux adresses suivantes :

Numéro du paramètre	**Adresse dans la pile** *(contexte de la fonction appelée)*
#7	`%rbp + 16`
#8	`%rbp + 24`
#9	`%rbp + 32`
...	

Le paramètre numéro 7 est bien à l'adresse `%rbp + 16`, et pas `%rbp + 8`, car la case mémoire `%rbp + 8` contient l'adresse de retour à la fonction appelante.

Convention d'utilisation des registres

L'intérêt principal de respecter les conventions définies dans l'ABI est de pouvoir mélanger des fonctions écrites en C avec d'autres écrites en assembleur. Cela nous permet notamment d'écrire des programmes de tests en C, ou d'utiliser directement des fonctions de la bibliothèque C depuis une fonction écrite en assembleur.
Pour pouvoir interagir avec du code généré par le compilateur C, on doit (en plus de toutes les conventions de gestion de la pile vues plus haut) respecter certaines conventions d'utilisation des registres. Pour éviter d'avoir à différencier l'appel de fonctions écrites en C et de fonctions écrites en assembleur, on choisi de respecter les mêmes conventions dans toutes les fonctions que l'on écrira en assembleur.
Les registres généraux de l'architecture x86_64 sont en principes des registres généralistes avec lesquels on peut effectuer les calculs que l'on veut. En pratique, ils ont des rôles spécialisés définis par l'ABI et rappelé ci-dessous :

- `%rax` est le registre servant à retourner une valeur à la fin d'une fonction, il peut être utilisé librement pour des calculs ;
- `%r10` et `%r11` sont des registres qui ont un rôle défini dans des situations qui ne nous concernent pas dans le cadre de ce cours : on s'en servira donc librement comme des registres de calcul ;
- `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` sont les registres servant à passer les 6 premiers paramètres d'une fonction, ils peuvent être utilisé librement pour des calculs si on n'a plus besoin des valeurs des paramètres ;
- `%rsp` et `%rbp` sont les registres de gestion de la pile : on ne doit évidemment pas les utiliser pour faire des calculs, on ne les manipule en pratique que lors de la mise en place et de la destruction du cadre de pile ;
- `%rbx` et les registres `%r12`, `%r13`, `%r14` et `%r15` sont des registres dont la valeur doit être sauvegardées dans la pile avant de pouvoir utiliser le registre.

Les registres pouvant être utilisés librement dans une fonction écrite en assembleur sont couramment appelés « registres scratch ». On a jusqu'à présent utilisé uniquement `%rax`, `%r10` et `%r11` : on voit maintenant qu'on peut aussi utiliser les registres servant à passer des paramètres, si on est sûr qu'on n'aura plus besoin des valeurs passées (et bien sûr aussi si la fonction a moins de 6 paramètres).

Les autres registres contiennent à l'entrée de la fonction une valeur qui doit impérativement être préservée. Dans le cas des registres `%rsp` et `%rbp`, la règle est simple : on ne doit jamais utiliser ses registres pour faire des calculs. Pour les autres registres, on peut les utiliser si on en a besoin, mais on doit d'abord sauvegarder leur contenu dans la zone temporaire de la pile. En effet, la fonction appelante a le droit de copier des valeurs importantes dans ces registres avant l'appel à la fonction courante, et elle s'attend à retrouver ces valeurs au retour de la fonction.
En pratique, le fait que les opérations puissent travailler directement avec un paramètre en mémoire réduit significativement le nombre de registres nécessaires et on n'a quasiment jamais besoin d'utiliser les registres non-scratch pour faire des calculs.
La convention décrite ci-dessus doit bien sûr être respectée lorsque la fonction que l'on est en train d'écrire en assembleur appelle une autre fonction (C ou assembleur). Dans ce sens, cela signifie que la fonction courante ne peut laisser de valeurs importantes dans les registres scratch (qui risquent d'être écrasés par la fonction appelée) mais qu'elle peut en stocker dans la pile.

Exemples détaillés

Commencez par télécharger l'archive contenant les sources sur la page principale et décompressez-la dans un répertoire de votre choix.

PGCD

Soit le programme C ci-dessous que l'on souhaite traduire systématiquement en assembleur dans le fichier `fct_pgcd.s` :

```
uint32_t pgcd(uint32_t a, uint32_t b)
{
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}
```

Cette fonction est appelée par le programme de test écrit en C dans le fichier `pgcd.c`.
On note que :

- la fonction `pgcd` prend deux paramètres de taille 32 bits chacun : GCC va donc utiliser les registres `%edi` et `%esi` pour nous passer les paramètres `a` et `b` ;
- il n'y pas de variable locale dans cette fonction.

Le code assembleur sera donc (noter l'importance des commentaires pour préciser où sont situés les paramètres, variables locales, etc.) :

```
    .text
    .globl pgcd
    // uint32_t pgcd(uint32_t a, uint32_t b)
    // a : %edi
    // b : %esi
pgcd:
    enter $0, $0
    // while (a != b) {
while:
    cmpl %edi, %esi
    je fin_while
    // if (a > b) {
```

```
    cmpl %edi, %esi
    jnb else
    // a = a - b;
    subl %esi, %edi
    jmp fin_if
else:
    // b = b - a:
    subl %edi, %esi
fin_if:
    jmp while
fin_while:
    // return a;
    movl %edi, %eax
    leave
    ret
```

Vous pouvez exécuter pas à pas ce programme avec GDB, en affichant le contenu des registres `%eax`, `%edi` et `%esi` pour comprendre ce qui se passe (rappel : la commande GDB pour afficher un registre est `display $eax`).

Multiplication

On traduit maintenant la fonction de multiplication simple déjà vu (fichier `fct_mult.s`) :

```
uint64_t mult(uint64_t x, uint64_t y)
{
    uint64_t res = 0;
    while (y != 0) {
        res = res + x;
        y--;
    }
    return res;
}
```

Le programme principal de test est dans le fichier `mult.c`.
Cette fois-ci, on remarque :

- deux paramètres sur 64 bits chacun : GCC nous passera ces paramètres dans les registres `%rdi` et `%rsi` ;
- une variable locale `res` : cette variable a besoin de 8 octets de mémoire, mais comme on l'a vu plus haut, on doit arrondir la taille réservée au multiple de 16 immédiatement supérieur.

Cela donne donc le code assembleur suivant :

```
    .text
    // uint64_t mult(uint64_t x, uint64_t y)
    .globl mult
    // x : %rdi
    // y : %rsi
mult:
    // uint64_t res : %rbp - 8
    // on aligne sur un multiple de 16
    enter $16, $0
    // res = 0;
    movq $0, -8(%rbp)
    // while (y != 0) {
while:
    cmpq $0, %rsi
    je fin_while
```

```
    // res = res + x;
    addq %rdi, -8(%rbp)
    // y--;
    subq $1, %rsi
    jmp while
fin_while:
    // return res;
    movq -8(%rbp), %rax
    leave
    ret
```

Vous pouvez dérouler l'exécution de la fonction en utilisant GDB. Pour afficher le contenu de la variable locale stockée à l'adresse `%rbp - 8`, utiliser la commande `display *((long long*)($rbp-8))`.

Suite de Fibonacci

On va enfin étudier un exemple de fonction récursive. Une fonction récursive est une fonction qui s'appelle elle-même, ce qui ne change absolument rien par rapport au fait d'appeler une autre fonction.
Le code C à traduire dans le fichier `fct_fibo.s` est le suivant :

```
uint64_t fibo(uint64_t n);
{
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibo(n - 1) + fibo(n - 2);
    }
}
```

Le programme principal en C est dans `fibo.c`.
Le code assembleur produit est le suivant :

```
    .text
    .globl fibo
    // uint64_t fibo(uint64_t n)
    // n : %rdi
fibo:
    // on reserve de la place pour la sauvegarde de %rdi et %rax
    enter $16, $0
    // sauvegarde du parametre %rdi = n dans la pile
    movq %rdi, -16(%rbp)
    // if (n == 0)
    cmpq $0, %rdi
    jne elsif
    // return 0;
    movq $0, %rax
    jmp fin
elsif:
    // else if (n == 1)
    cmpq $1, %rdi
    jne else
    // return 1;
    movq $1, %rax
    jmp fin
else:
    // return fibo(n - 1) + fibo(n - 2);
```

```
    // on commence par appeler fibo(n - 1)
    subq $1, %rdi
    call fibo
    // restauration du %rdi = n initial
    movq -16(%rbp), %rdi
    // sauvegarde de %rax = fibo(n - 1) dans la pile
    movq %rax, -8(%rbp)
    // on appelle maintenant fibo(n - 2)
    subq $2, %rdi
    call fibo
    // et on calcule la somme finale
    addq -8(%rbp), %rax
fin:
    leave
    ret
```

Les points intéressants sont dans le cas où N est supérieur ou égal à 2, que l'on détaille ci-dessous.
On commence par calculer N - 1 dans `%rdi` pour réaliser l'appel `fibo(N - 1)`. On peut modifier le paramètre N passé dans `%rdi` car on a sauvegardé le `%rdi` initial dans la pile, on pourra donc le restaurer après l'appel :

```
else:
    subq $1, %rdi
    call fibo
```

Au retour de cet appel, le résultat du calcul de `Fibo(N - 1)` est dans `%rax`. Comme on va faire un autre appel de fonction plus bas, on doit sauvegarder cette valeur, `%rax` étant un registre scratch. Au passage, on restaure systématiquement après l'appel les paramètres initiaux qui ont pu être modifiés : ici, `%rdi` :

```
    movq -16(%rbp), %rdi
    movq %rax, -8(%rbp)
```

On peut maintenant faire simplement l'appel à `fibo(N - 2)` :

```
    subq $2, %rdi
    call fibo
```

Et on récupère le résultat `Fibo(N - 2)` dans `%rax` : on peut immédiatement l'additionner avec `Fibo(N - 1)` qu'on avait sauvegardé dans la pile :

```
    addq -8(%rbp), %rax
```

On finit donc bien avec `Fibo(N - 1) + Fibo(N - 2)` dans `%rax` à la fin de la fonction.
Comme on a sauvegardé deux registres dans la pile, il faut penser à réserver de la place au moment de la mise en place du cadre de pile, ce qu'on fait avec `enter`. On recommande de sauvegarder systématiquement les valeurs des paramètres dans la pile si la fonction qu'on écrit en appelle elle-même une autre (ce qui est le cas ici) :

```
fibo:
    enter $16, $0
    movq %rdi, -16(%rbp)
```

Vous pouvez dérouler l'exécution de cette fonction dans GDB et comparer le résultat avec la liste des nombres de Fibonacci sur wikipedia par exemple.
Vous pouvez afficher le contenu des cases mémoire où sont sauvegardés `%rdi` et `%rax` en utilisant la même syntaxe que pour des variables locales : `display *((long long*)($rbp-8))`.

Exercices

Dans tous les exercices ci-dessous, on vous demande de réaliser une traduction systématique du code C vers l'assembleur, c'est à dire **sans chercher à optimiser votre code**, par exemple en remplaçant des variables par des registres. On vous demande donc de manipuler les variables locales directement dans la pile d'exécution.

Première fonction

Ouvrez le fichier `fct_age.s` et complétez le en traduisant systématiquement la fonction C suivante :

```
uint16_t age(uint16_t annee_naissance)
{
    uint16_t age;
    age = 2022 - annee_naissance;
    return age;
}
```

Le programme `age.c` vous permettra de tester votre code en passant votre année de naissance en paramètre sur la ligne de commande.

Appels croisés C/assembleur

Complétez maintenant le fichier `fct_fact.s` pour implanter la fonction de calcul de factorielle ci-dessous :

```
uint64_t fact(uint64_t n)
{
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

Vous utiliserez pour cela l'instruction de multiplication déjà vue à la première séance.
Tester votre code en utilisant le programme principal contenu dans le fichier `fact.c`. Vous pouvez comparer le résultat obtenu avec la liste des valeurs de factorielle sur Wikipedia.
Que se passe-t'il lorsqu'on cherche à calculer 21! (qui vaut comme chacun le sait 51090942171709440000) ?
Ajouter à votre code assembleur ce qu'il faut pour appeler la fonction `erreur_fact` fournie dans le fichier `fact.c` dans ce cas-là.
Si vous avez fini en avance : remplacer l'utilisation de l'instruction `mul` par un appel à la fonction `mult` écrite plus haut. Vous aurez besoin de modifier le `Makefile` pour inclure le fichier `fct_mult.o` dans la liste des dépendances du binaire `fact`.

Utilisation de la bibliothèque C

Compléter le fichier `fct_palin.s` pour implanter la fonction ci-dessous qui teste si une chaine de caractère est un palindrome. On rappelle qu'un palindrome est un mot qui se lit aussi bien de gauche à droite que de droite à gauche et que par convention la chaine vide n'est pas un palindrome.

```
bool palin(char *chaine)
{
    uint64_t inf, sup;
    for (inf = 0, sup = strlen(chaine) - 1;
         (inf < sup) && (chaine[inf] == chaine[sup]);
         inf++, sup--);
    return inf >= sup;
}
```

La fonction `strlen` est une fonction de la bibliothèque C dont vous pouvez obtenir la spécification en tapant `man strlen`.
On rappelle qu'en C99, le type `bool` est implanté par un octet (8 bits) valant 0 ssi la condition est fausse et 1 sinon.
La classe d'intruction `setcc` (dont on fourni la documentation sur la page principale) vous permettra de simplifier le calcul de la valeur de retour (mais vous pouvez aussi implanter cela comme s'il s'agissait d'un `if`).

Vous pouvez tester votre code avec le programme principal fourni dans `palin.c`.
Si vous avez fini en avance, à la fin de la séance : modifier votre programme pour qu'il ignore les espaces, la ponctuation et les majuscules dans la chaine passée en paramètre (e.g. les chaines « Esope reste ici et se repose » et « Dammit, I'm mad ! » doivent être considérées comme des palindromes).

Passage de tableaux en paramètre

Complétez enfin le fichier `fct_tri_nain.s` pour implanter l'algorithme du tri du nain de jardin, dont on rappelle le principe ci-dessous.
Un nain de jardin souhaite trier des pots de fleurs par taille croissante en appliquant la stratégie suivante. Il regarde le pot devant lui :

- s'il est plus petit que le pot à sa droite, le nain avance d'un pas vers la droite (s'il n'est pas arrivé à la fin de la file de pots) ;
- si le pot devant lui est plus grand que le pot à sa droite, le nain échange les deux pots, et recule d'un pas vers la gauche (s'il n'est pas revenu au début de la file de pots).

En C, on pourrait écrire la fonction à implanter comme ci-dessous :

```
void tri_nain(int32_t tab[], uint64_t taille)
{
    for (uint64_t i = 0; i < taille - 1; ) {
        if (tab[i] > tab[i+1]) {
            int32_t tmp = tab[i];
            tab[i] = tab[i+1];
            tab[i + 1] = tmp;
            if (i > 0) {
                i = i - 1;
            }
        } else {
            i = i + 1;
        }
    }
}
```

On note que la fonction `tri_nain` prend en paramètre le tableau d'entiers à trier. On rappelle qu'en C, lorsqu'on passe un tableau en paramètre d'une fonction, c'est l'adresse du tableau qui est passée (le premier paramètre d'une fonction est stockée dans `%rdi` et on ne peut évidemment par recopier l'ensemble des éléments du tableau dans un seul registre !).
Vous pouvez tester votre programme grâce au fichier `tri_nain.c` fourni, qui calcule les temps d'exécution de votre tri et d'un tri de référence, et vérifie que votre tri est correct (un message d'erreur sera affiché si le tableau résultat est faux).
Si vous avez fini en avance : on remarque qu'on fait beaucoup d'accès mémoire redondants dans cette traduction systématique, et qu'il semble facile d'optimiser le tri en utilisant intelligemment les registres pour stocker des valeurs.
Implantez dans le fichier `tri_nain.s` une fonction `tri_nain_opt` en utilisant les registres `%r10` et `%r11d` pour éviter autant que possible les accès mémoires.
Notez bien que le tri de référence est un *quicksort*, c'est à dire un tri en $O(n \times log(n))$ alors que le tri du nain est un tri en $O(n^2)$: même en optimisant le code produit, cela ne compensera pas le fait que l'algorithme du nain est intrinsèquement peu efficace.

Fonction avec beaucoup de paramètres

Le programme principal `val_bin.c` prend en paramètre une chaine de caractères représentant un nombre binaire sur 8 bits, et appelle la fonction `val_binaire` à écrire en assembleur. Cette fonction prend chaque bits du nombre sous la forme d'un `uint8_t` passé en paramètre, et calcule la valeur décimale correspondante selon la formule ci-dessous :

```
uint8_t val_binaire(uint8_t b7, uint8_t b6, uint8_t b5, uint8_t b4,
```

```
                    uint8_t b3, uint8_t b2, uint8_t b1, uint8_t b0)
{
  return (b7 << 7) | (b6 << 6) | (b5 << 5) | (b4 << 4) |
      (b3 << 3) | (b2 << 2) | (b1 << 1) | b0;
}
```