

TD 4 : MIPS

Microprocessor without Interlocked Pipeline Stages

Gestion de la mémoire

Tout processus chargé en mémoire possède son propre espace d'adressage *virtuel* comprenant les 4 zones décrites ci-dessous dans le cas d'une architecture MIPS:

- **kseg2**, 1024 MBytes d'espace d'adressage kernel traduit par la MMU.
- **kseg1**, 512 MBytes d'espace d'adressage kernel direct (pas de cache ni MMU). Généralement utilisé pour les adresses I/O et boot ROM.
- **kseg0**, 512 MBytes d'espace d'adressage kernel. Espace principalement accédé à travers le cache.
- **kuseg**, 2048 MBytes d'espace d'adressage utilisateur.

0xF000.0000 0xE000.0000 0xD000.0000 0xC000.0000	Mapped (cached) kseg2
0xB000.0000 0xA000.0000	Unmapped uncached kseg1
0x9000.0000 0x8000.0000	Unmapped cached kseg0
0x7000.0000 0x6000.0000 0x5000.0000 0x4000.0000 0x3000.0000 0x2000.0000 0x1000.0000 0x0000.0000	32-bit user space (Mapped cached) kuseg

Le composant en charge de traduire les adresses virtuelles en adresses physiques est la MMU (Memory Management Unit). Celle-ci accède au TLB (Translation Look-aside Buffer) sinon à la *table des pages* pour connaître l'adresse physique. A noter que le TLB sert de cache pour la table des pages. Le fonctionnement est le suivant:

- L'adresse virtuelle est connue du TLB (*Hit*): l'adresse physique est renvoyée.
- L'adresse virtuelle est inconnue du TLB (*Miss*): la table des pages est interrogée.
- L'adresse virtuelle est connue de la table des pages : l'entrée est écrite dans le TLB et l'adresse physique est renvoyée.
- L'adresse virtuelle est inconnue de la table des pages (*page fault exception*): les données absentes sont chargées du disque dur vers la mémoire, les entrées sont écrites dans le TLB et la table des pages.

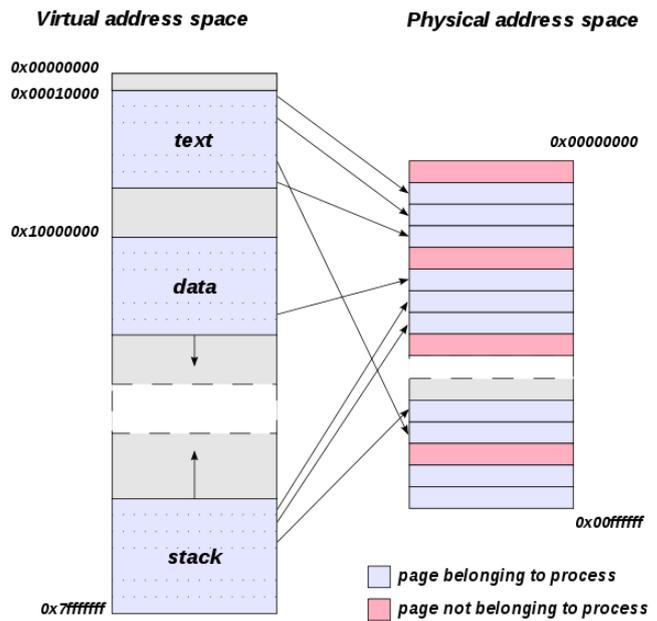
Le chargement de nouvelles pages mémoires peut nécessiter l'éviction d'anciennes pages vers le disque lors que la mémoire physique est pleine. L'algorithme de gestion des pages généralement implanté dans les systèmes d'exploitation est *Least Recently Used*.

L'espace utilisateur

L'espace utilisateur va donc (dans le cas de MIPS) de 0x0000.0000 à 0x7FFF.FFFF. Cet espace comprend principalement les parties ci-dessous :

- `.text`: contient le code du processus.
- `.data`: contient les données statiques du processus.
- `stack`: contient la pile du processus.
- `heap`: contient le tas du processus (inclus dans `data` dans le schéma ci-joint).

Le placement de ces zones dans l'espace utilisateur ainsi que le sens vers lequel croît la pile (et inversement le tas) sont des conventions dépendant de l'architecture. MIPS impose que la pile croisse vers les adresses plus petites.



Pile et tas

La mémoire allouée de manière statique peut être stockée dans les zones `text`. La pile et le tas servent tout deux à l'allocation dynamique de mémoire: on ne sait pas a priori de quelle quantité de mémoire on aura besoin à quel moment.

La pile sert pour les appels de fonction, le passage d'arguments et les variables automatiques (entier, char...). On sait généralement à l'avance la quantité de mémoire qui sera réservée par l'exécution d'une instruction.

Le tas sert pour des allocations de mémoire plus compliquées (tableaux, structures...). On ne sait pas à l'avance quelle quantité de mémoire va être réservée par une instruction (`int* p = (int*)malloc(n*sizeof(int));`). L'accès au tas n'est pas lié à la fonction dans laquelle le programme se situe.

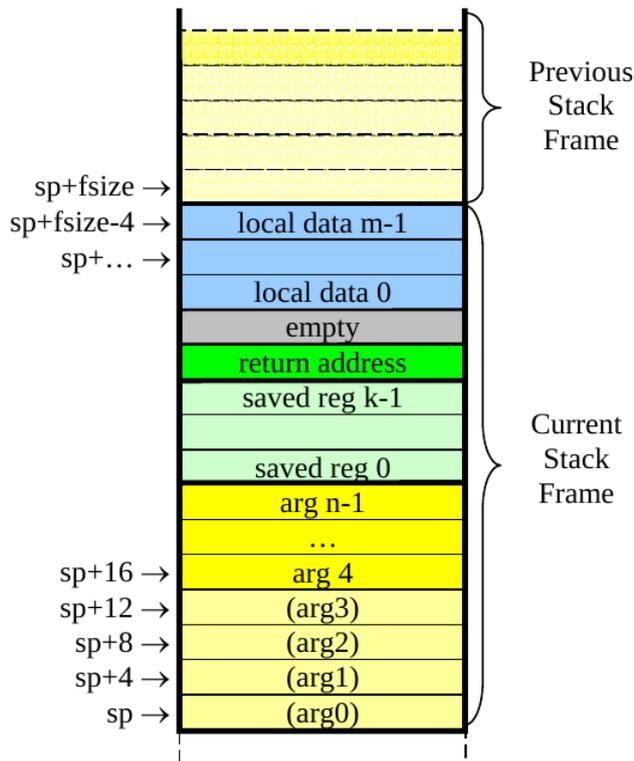
Registres et cadre de pile

Deux registres spécifiques, `sp` (Stack Pointer) et `fp` (Frame Pointer) pour MIPS (`bp` dans le cours), servent à mémoriser quel espace dans la pile est utilisé lors de l'appel d'une routine. Dans MIPS, la convention d'appel de routine impose que la pile soit alignée sur 8 octets sachant que chaque adresse correspondant à un octet (8 bits). On incrémente/décrémente donc le pointeur de pile par des multiples de 8.

On divise le cadre de pile en 5 zones:

1. La zone des arguments est la zone pour stocker les arguments passés à une sous-fonction appelée par la fonction courante. (i.e., la sous-fonction dont le cadre de pile sera en haut de la pile courante.) Les 4 premiers mots mémoires de cette section ne sont jamais utilisés par la fonction courante; ils sont réservés aux les sous-fonction appelées par la fonction courante. (4 arguments peuvent être passés par les registres arguments (`$a0` à `$a3`)). S'il y a plus de 4 arguments la fonction courante les place sur la pile aux adresses: `sp+16`, `sp+20`, `sp+24`, etc.
2. La zones des registres sauvegardés contient l'espace permettant de sauvegarder la valeur des registres sauvegardés (`$s0` à `$s7`) que la routine courante pourrait modifier au cours de son exécution. Elle empile donc les registres sauvegardés qu'elle risque de modifier, au début de son exécution, et les dépile, à la fin de son exécution, dans les registres correspondants. Cela permet donc à la fonction courante de disposer des registres sauvegardés librement, tout en assurant que la fonction appellante les récupérera dans l'état ou elle les avait laissés.

3. La zone d'adresse de retour est utilisée pour stocker la valeur du registre d'adresse retour \$ra. Cette valeur est empilée au début de l'exécution de la routine courante et dépilée juste avant qu'elle termine.
4. Le Pad est inséré dans le cadre de pile pour s'assurer que la taille du cadre de pile est un multiple de 8.
5. La zone de stockage des données locales est utilisée pour stocker les variables locales. La fonction courante doit réserver suffisamment d'espace dans cette zone pour toutes ses données locales, notamment de quoi stocker les registres temporaires (\$t0 à \$t9). La zone de stockage des données locales doit aussi être complétée pour que sa taille soit un multiple de 8.



- Empiler une donnée de 32bits contenue dans r3 :

```
addiu $sp, $sp, -4 # Decrement stack pointer by 4
sw    $r3, 0($sp) # Save r3 to stack
```

- Dépiler une donnée de 32bits vers r3 :

```
lw    $r3, 0($sp) # Copy stack to r3
addiu $sp, $sp, 4 # Increment stack pointer by 4
```

Comprendre la pile

Heapsters cannot dare using the stack only

1. Donner l'état de la pile et des registres lors de l'exécution du programme assembleur ci-dessous.
2. Que fait ce programme ?

```
.data
myArray:
    .space 40
    .text
main:
    li    $t6, 1
    li    $t7, 4
    sw    $t6, myArray($0)
    sw    $t6, myArray($t7)
    li    $t0, 8
loop:
    addi  $t3, $t0, -8
    addi  $t4, $t0, -4
    lw    $t1, myArray($t3)
    lw    $t2, myArray($t4)
    add   $t5, $t1, $t2
    sw    $t5, myArray($t0)
    addi  $t0, $t0, 4
    blt   $t0, 40, loop
    jr    $ra
```

La récursivité

Hope it will stop one day

1. Donner le code assembleur du programme C ci-dessous.

```
int fact( int n ) {
    if ( n == 0 )
        return 1 ;
    else
        return fact( n - 1 ) * n ;
}
```