Les Processus 2

Programme principal et arguments

- Programation système : langage C
- Fonction principale : *main*

```
int main(int argc, char *argv[])
```

  - *argc* est le nombre d'arguments passés en paramètre par l'interpréteur de commande au programme appelé
  - *argv* est le tableau contenant les arguments au format texte

Exemple :

Powerbook :∼ thibaultbernard$./programme arg1 toto 428

argv[0] : ./programme

argv[1] : arg1

argv[2] : toto

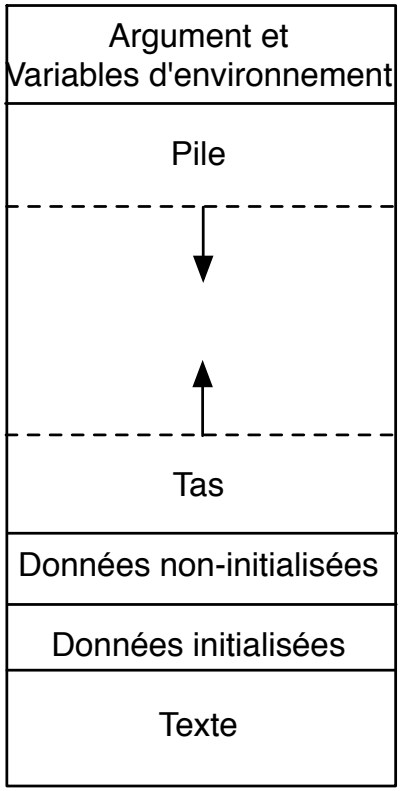
argv[3] : 428

Schéma mémoire d'un processus

La mémoire d'un processus est composé de :

- Le segment de texte : ce sont les différentes instructions exécutées par le processeur. Ce segment est généralement en lecture seule et peut être éventuellement partagé
- Segment de données initialisées : par exemple la déclaration `int` maxcount $= 99$;
- Segment de données non-initialisées : par exemple la déclararation `int` sum[1000] ; Les blocs de données sont alors initialisées à 0 ou *null*
- Pile : sert à la sauvegarde d'information en particulier lors de l'appel aux fonctions (adresse de retour) et au stockage des données temporaires durant l'exécution de fonction
- Tas : sert à l'allocation dynamique de mémoire

Schéma mémoire d'un processus



La commande *size* fournit les différentes tailles des segments.

Exemple : Powerbook :∼ thibaultbernard$ size /bin/ls

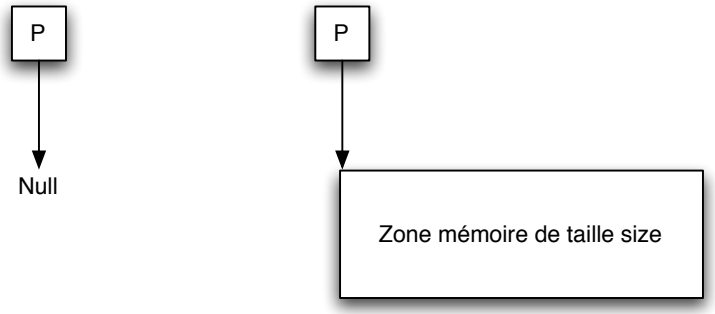
text	data	bss	others	dec	hex
24576	4096	0	7884	36556	8ecc

Primitives

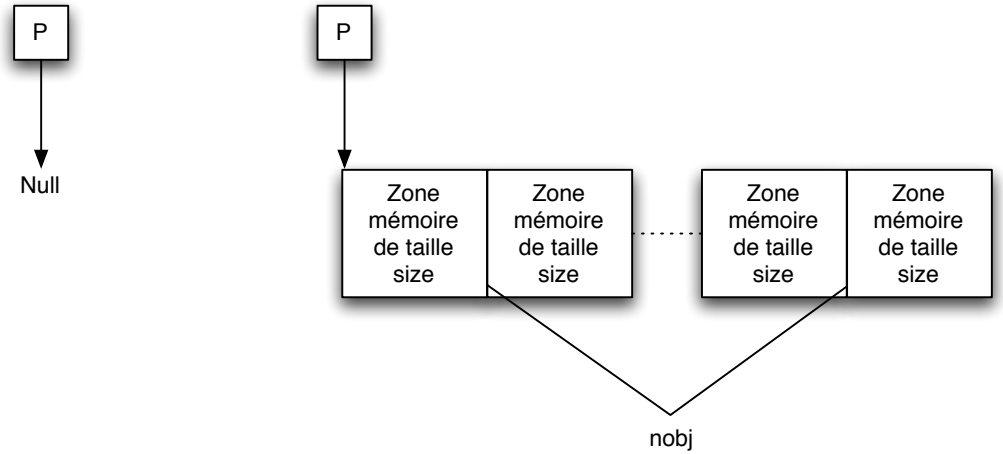
1. malloc : Alloue une zone mémoire de la taille spécifiée dans le tas. Le contenu de cette zone est arbitraire
2. calloc : Alloue une zone mémoire de la taille spécifiée dans le tas. Le contenu de cette zone est initialisé à 0
3. realloc change la taille d'une zone mémoire précédemment allouée (cette zone peut alors éventuellement être déplacée). La zone mémoire éventuellement ajoutée contient des valeurs arbitraires
4. free : libère la zone mémoire
5. alloca : Alloue une zone mémoire de la taille spécifiée dans la pile. Fonctionne comme malloc

Allocation

```
void * malloc(size_t size)
```

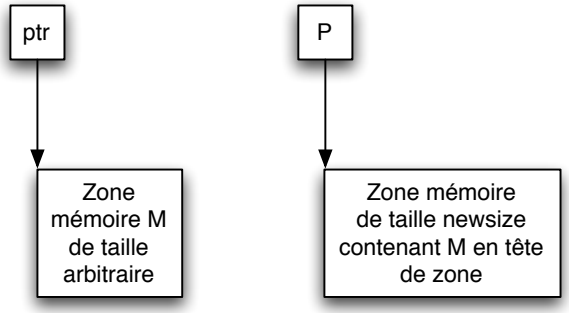



```
void * calloc(size_t nobj, size_t size)
```

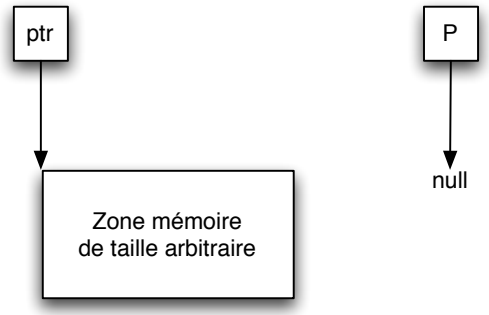


Réallocation et libération

```
void * realloc(void* ptr, size_t newsize)
```




```
void free(void* ptr)
```



Récupérer et modifier les variables d'environnement

- `char * getenv(const char * name)`
- `int putenv(const char * str)`
- `int setenv(const char * name, const char * value, int value)`
- `void unsetenv(const char * name)`

Variable	Description
HOME	Répertoire personnel
LOGNAME	Nom de login
PATH	Liste de prefixes pour rechercher les fichiers exécutables
TERM	Type de terminal
...	...

Récupérer et modifier les limitations de ressources

- `int getrlimit(int ressource, struct rlimit *rlptr)`
- `int setrlimit(int ressource, const struct rlimit *rlptr)`
-
```
struct rlimit{
    rlim_t rlim_cur /*soft limit : current limit*/
    rlim_t rlim_max /*hard limit : max of rlim_cur*/
}
```

Ressource	Description
RLIMIT_CORE	taille max du fichier core créé
RLIMIT_CPU	maximum du temps CPU en secondes
RLIMIT_DATA	taille max du segment de données (init et non init)
RLIMIT_FSIZE	taille maximale des fichiers créés
RLIMIT_NOFILE	nombre maximal de fichiers ouverts
RLIMIT_NPROC	nombre maximum de processus enfants
RLIMIT_STACK	taille maximal de la pile
...	...

Identifier un processus

- `pid_t getpid(void)` Fournit le pid du processus
- `pid_t getppid(void)` Fournit le pid du processus père
- `uid_t getuid(void)` Fournit l'uid du processus
- `uid_t geteuid(void)` Fournit l'uid effectif du processus
- `gid_t getgid(void)` Fournit le gid du processus
- `gid_t getegid(void)` Fournit le gid effectif du processus

Création de processus

`pid_t fork(void)` créé un nouveau processus. La valeur de retour est :

- 0 dans le processus fils
- le pid du processus fils dans le processus père

Exemple :

```
 int glob = 6 ;
int main(void)
{  int var ; /*variable dans la pile*/
   pidt_t pid ;
   var = 88 ;
   printf("avant fork") ;
   if(pid==0){ /*fils*/
       glob++ ; var++ ;
   } else
       sleep(2) ; /*parent*/
   printf("pid=%d, glob=%d, var=%d", getpid(),glob,var) ;
   exit(0) ;}
```

Exécution

```
Powerbook :~ thibaultbernard$./a.out
avant fork
pid=1024,glob=7, var=89
pid=1023,glob=6, var=88
```

Terminaison de processus

- `void exit(int status)` effectue un m̈énageët quitte le processus (fermeture des fichiers par exemple)
- `void _exit(int status)` quitte le processus et retourne directement au système
- `int atexit(void (*func) (void))` spécifie des fonctions à appeler lors de la terminaison du processus

Terminaison de processus

- `pid_t wait(int * statloc)` attend la fin d'un processus fils et récupère sa valeur de retour dans la variable `statloc`
- `pid_t waitpid(pid_t pid, int * statloc, int options)` attend la fin du processus fils `pid` et récupère sa valeur de retour dans la variable `statloc`. Les options servent éventuellement à avoir un appel non bloquant

Chargement d'un programme

- `int execl(const char * pathname, const char * arg0, ..., NULL)`
- `int execv(const char * pathname, char * const arg[])`
- `int execle(const char * pathname, const char * arg0, ..., NULL, char * const envp[])`
- `int execve(const char * pathname, char *const argv[], char * const envp[])`
- `int execlp(const char * filename, const char * arg0, ... NULL)`
- `int execvp(const char * filename, char *const arg[])`