Unix : concepts et outils avancés
TD no 6 : processus

Olivier Aumage
LIP, bureau 302
email : `oaumage@ens-lyon.fr`

8 novembre 2001

1 Pages de mémoire

- La commande `malloc` permet d'allouer de la mémoire à un processus avec une précision d'un octet. Le système est-il capable d'assurer le contrôle des accès mémoire avec une telle granularité ? Si non, pourquoi ?
- Écrivez un programme qui alloue une zone d'un caractère puis écrit un octet dans chacune des cases mémoire consécutives à cette zone en affichant à chaque itération l'adresse de la case mémoire testée. Déterminez à l'aide de ce programme la limite supérieure de la *page* dans laquelle la fonction `malloc` a alloué le bloc d'un caractère.
- Les adresses relevées sont elles les mêmes d'une exécution à l'autre ? Proposez une explication à votre observation.
- Essayez de déterminer la taille des pages mémoire du système que vous utilisez.

2 Création de processus

- L'appel système `fork` permet de répliquer un processus pour en créer un nouveau semblable au processus d'origine. Écrivez un programme qui appelle `fork` puis indique s'il est le père ou le fils.
- Essayez le programme ci-dessous :

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void error(char *msg)
{
  perror(msg);
  exit(1);
}

int main()
{
  pid_t process = -1;

  fprintf(stdout, "Hello, ");
```

```
  process = fork();

  if (process == -1)
    {
      error("fork");
    }
  else if (process)
    {
      fprintf(stdout, "Son!\n");
    }
  else
    {
      fprintf(stdout, "Father!\n");
    }

  return 0;
}
```

– Que se passe-t-il si on rajoute un *retour à la ligne* après l'affichage de `"Hello,_"` ? Interprétez les deux résultats obtenus.
– Utilisez la propriété précédente pour proposer un programme utilisant `fork` et tel que le *père* et le *fils* coopèrent pour créer un fichier contenant : `PFPFPFPFPFPFPFPFPFPF`. Le père n'a le droit d'écrire que des `'P'` et le fils ne peut écrire que des `'F'`.
– Le programme suivant divise le processus initial en un processus père et un processus fils puis les deux processus ainsi créés entrent dans une boucle infinie. Lancez l'exécution de ce programme, ouvrez un second terminal et lancez la commande `top` dans le nouveau terminal. Observez les tailles respectives des deux processus (la colonne `SIZE` indique la taille totale du processus, et la colonne `RES` indique la quantité de mémoire « effectivement consommée par le programme ») et proposez une interprétation de votre observation. **Note :** la signification des champs de la commande `top` diffère d'un Unix à l'autre ; sous LINUX, le champ intéressant dans ce contexte est le champ `SHARED`.

```
#include <unistd.h>

int main()
{
  fork();
  while (1);
}
```

– Essayez la version ci-dessous et observez l'évolution de la quantité de mémoire utilisée par le processus fils au cours du temps. **Notes :**
  – il sera peut-être nécessaire d'ajuster la boucle while si l'exécution est trop rapide ;
  – n'utilisez pas d'optimisations à la compilation.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

```

```
int main()
{
  char *ptr = NULL;
  int   i   =    0;

  ptr = malloc(16384);
  for(i = 0; i < 16384; i++)
    ptr[i] = 0;

  if (!fork())
    {
      i = 0;
      while (i < 0x1FFFFFFF) i++;

      for (i = 0; i < 16384; i++)
        ptr[i] = 1;
    }

  while (1);
}
```

3 Exercices complémentaires

Point de rupture Cette section a pour but de donner quelques précisions concernant le premier exercice sur les pages mémoire. La zone de l'espace mémoire d'un processus accessible par l'intermédiaire des fonctions d'allocation dynamique de mémoire (`([mc]|re)alloc`) est appelée le *tas*. La limite supérieure des adresses des cases mémoire du tas est appelée le *point de rupture* ou *breakpoint* en anglais (sans rapport avec les *breakpoints* d'un débogueur). Cette limite est modifiée dynamiquement — en cas de besoin — par `malloc` (et également par `free`). Il est également possible d'agir directement sur cette limite (et obtenir un programme non portable !) avec les appels système `brk` et `sbrk` (cf. pages du manuel). En particulier, le passage de la valeur `0` en paramètre à la fonction `sbrk` permet d'obtenir l'adresse du point de rupture actuel. Cette adresse est l'adresse immédiatement supérieure à la limite supérieure du tas (donc la première adresse non valide). Utilisez le programme ci-dessous pour vérifier les observations faites au cours du premier exercice. **Note :** n'oubliez pas d'adapter la constante `PAGE_SIZE` à la taille des pages de votre système.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* System-dependent page size */
#define PAGE_SIZE 4096ul

int
```

```
main()
{
  void *brk_ptr = NULL;

  /* Raw process breakpoint */
  brk_ptr = sbrk(0);
  printf("brk = %p\n", brk_ptr);

  printf("%p\n", brk_ptr - 1);
  *(char *)(brk_ptr - 1) = 0;
  printf("ok\n");

  printf("%p\n", brk_ptr);
  *(char *)brk_ptr = 0;
  printf("ok\n");

  /* Page-aligned process breakpoint */
  brk_ptr = (void *)(((long)brk_ptr + PAGE_SIZE) & ~(PAGE_SIZE - 1));
  printf("aligned brk = %p\n", brk_ptr);

  printf("%p\n", brk_ptr - 1);
  *(char *)(brk_ptr - 1) = 0;
  printf("ok\n");

  printf("%p\n", brk_ptr);
  *(char *)brk_ptr = 0;
  printf("ok\n");

  return 0;
}
```