622 mmap et cie.

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Chargement (*paging*) à la demande

Au démarrage des programmes

- Il faut charger des fichiers (exécutables, bibliothèques, etc.)
- Qui sont possiblement gros
- Et possiblement pas nécessaire *tout de suite* (voire jamais)

$\rightarrow$ Charger **au besoin** les **morceaux** de fichiers **nécessaires**

Si on est très paresseux

- `execve(2)` ne charge rien du tout
- On attend les vrais accès à la mémoire

Stratégie niveau système d'exploitation

Une page virtuelle peut être

- Soit en mémoire physique (RAM)
- Soit sur le disque : en *swap*
- Soit sur le disque : morceau de fichier pas encore chargé

Lorsqu'on accède à une page qui est sur le disque

- On la charge en RAM (*page in*)
- Depuis la *swap* ou depuis le fichier

Lorsque la RAM est pleine

- Si morceau de fichier : facile, il est déjà sur le disque
- Si page anonyme (pas de fichier) : on utilise la *swap* (*page out*)
- Éventuellement on écrit les données sur disque si besoin

Projection de fichiers en mémoire

Idée : fournir aux processus le chargement à la demande

- `mmap(2)` associe une zone mémoire (virtuelle) à un morceau de fichier
- `msync(2)` demander l'écriture des changements dans le fichier
- `mprotect(2)` change les droits d'une zone mémoire
- `munmap(2)` libère la zone mémoire

Beaucoup d'options possibles

- Quel bout du fichier est demandé? `offset` et `length`
- Fichier lu en entier ou à la demande? `MAP_POPULATE`
- Quels droits appliquer? `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`
- Les modifications en mémoire sont-elles écrites dans le fichier? `MAP_SHARED`, `MAP_PRIVATE`
- La zone mémoire est-elle copiée ou partagée lors d'un `fork(2)`? `MAP_SHARED`, `MAP_PRIVATE`
- Etc.

mmap.c

```c
#include "machins.h"
int main(int argc, char **argv) {
  int fd = open("compteur", O_RDWR|O_CREAT, 0666);
  if (fd==-1) { perror(NULL); return 1; }
  struct stat stat;
  fstat(fd, &stat);
  if (stat.st_size < 11) // version initiale si besoin
    stat.st_size = write(fd, "0000000000\n", 11);

  char* buf = mmap(NULL, stat.st_size,
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd); // plus besoin du descripteur

  int num = atoi(buf) + 1; // on « lit » directement du fichier
  sprintf(buf, "%010d\n", num); // et écrit aussi
  printf("%d\n", num); // affichage

  if (argc>1) pause(); // pause si argument
  munmap(buf, stat.st_size);
  return 0;
}
```

Efficacité des fichiers projetés

Accès très efficace

- Simple accès processeur $\leftrightarrow$ mémoire
- Pas d'appel système `read(2)`, `write(2)`
- Pas de copie noyau $\leftrightarrow$ processus
- Possibilité d'accès direct au matériel:
  Mémoire vidéo, *ringbuffer*, etc.

Table des inodes en mémoire (le retour)

- Les zones mémoires projetées
- Les données des fichiers ouverts ou en caches
- Peuvent utiliser les mêmes pages physiques que les fichiers projetés

$\rightarrow$ Pas besoin de dupliquer ou de synchroniser des zones mémoire

Inconvénients des fichiers projetés POSIX

- Pas de gestion simple de la taille une fois un ficher projeté
- Pas de `ftruncate(2)` ou d'ajout à la fin du fichier
- Pas de gestion simple des erreurs d'entrée-sortie
- Plus lent que read/write classiques dans certains cas
- Pas pour certains fichiers non réguliers
  Ni pour certains systèmes de fichiers
- Problématiques si disque réseau (NFS)

Allocation de zones anonymes

Idée : *mapper* des zones mémoire sans fichier associé

- `mmap(2)` avec indicateur `MAP_ANONYMOUS`
- Abus de langage (et d'appel système)
  Projection de fichier faite sans fichier !
- Mais on profite de l'expressivité de `mmap(2)`

Communication par mémoire partagée anonyme

- Sera hérité et partagé via `fork(2)`
- Permet la communication interprocessus extrêmement efficace
- Attention, sera perdu via `execve(2)`
- Attention aux situations de compétition

Communication par mémoire partagée nommée

API POSIX

- `shm_open(3)` créer ou ouvrir un « objet » mémoire partagé
- `shm_unlink(3)` supprime l'objet mémoire partagée
- `shm_overview(7)` pour les détails
- On l'utilise ensuite comme un fichier ouvert
- On peut aussi bien évidemment le « mmaper »
- Persistant jusqu'à l'arrêt du système ou une libération explicite

API POSIX sous Linux

- C'est en fait des fichiers « normaux »
- Dans un système de fichier mémoire `tmpfs`
- Monté habituellement dans `/dev/shm`

API System V

- `shmget(2)`, `shmat(2)`, `shmdt(2)`, `shmctl(2)`
- Vieille API plus complexe et bas niveau

shm.c

```c
#include "machins.h"
int main(int argc, char **argv) {
  int fd = shm_open("autre_compteur", O_RDWR|O_CREAT, 0666);
  if (fd==-1) { perror(NULL); return 1; }
  struct stat stat;
  fstat(fd, &stat);
  if (stat.st_size < 11) // version initiale si besoin
    stat.st_size = write(fd, "0000000000\n", 11);

  char* buf = mmap(NULL, stat.st_size,
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd); // plus besoin du descripteur

  int num = atoi(buf) + 1; // on « lit » directement du fichier
  sprintf(buf, "%010d\n", num); // et écrit aussi
  printf("%d\n", num); // affichage

  if (argc>1) pause(); // pause si argument
  munmap(buf, stat.st_size);
  return 0;
}
```

Mémoire côté processus chez GNU/Linux

Chargement des bibliothèques

- Est fait par le processus (pas par le noyau)
- `ld.so` utilise `mmap(2)` pour charger les morceaux de bibliothèques

→ Les détails une autre fois

Allocation de mémoire

- `malloc(3)` gère et alloue la mémoire du processus
- `brk(2)` pour agrandir (ou réduire) la zone du tas
- `mmap(2)` anonyme pour de grosses allocations

→ Les détails une autre fois

Pile pthread

- `pthread_create(3)` gère et alloue le thread
- `mmap(2)` pour crée un pile neuve (qui grandit à l'envers)
- `clone(2)` pour créer la tache
- Le tas pour les structures internes

Appels système supplémentaires

- `mcore(2)` (Linux) indique quelles pages sont en mémoire
- `madvise(2)` (Linux) indique l'utilisation future de zones mémoires
  - Peut changer des comportements
  - Peut changer les stratégies internes d'optimisations
- `posix_madvise(2)` version POSIX du précédent
- `mlock(2)` force une zone mémoire à rester résidente