

Projet système Master 1

R. Blanc A.Castel C.Eymond Laritaz M.Garnier

UFR IM²AG
Université Grenoble-Alpes

Jeudi 17 Janvier 2019

Outline

- 1 Nachos Input-Output
 - Appels systèmes et choix d[Pleaseinsertintopreamble]implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 Mémoire virtuelle
 - Mémoire virtuelle
 - Multiprocessus
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

1 Nachos Input-Output

- Appels systèmes et choix d'implémentations

2 Multi-Threading

- Gestion de la pile
- Appels Système

3 Mémoire virtuelle

- Mémoire virtuelle
- Multiprocessus

4 Système de fichiers

- Arborescence de répertoires
- Page des fichiers ouverts

Appels systèmes

- `void PutChar(char c);`
- `void PutString(char *s);`
- `char GetChar();`
- `void GetString(char *s, int n);`
- `void PutInt(int n);`
- `void GetInt(int *n);`

Choix d'implémentation

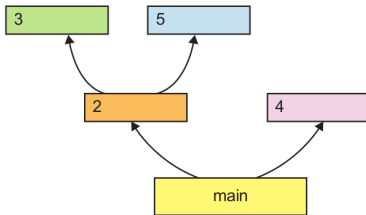
- Limite de chaîne de caractères traités de 200
- Les espaces sont lus comme les autres caractères

- 1 Nachos Input-Output
 - Appels systèmes et choix d[Please insert into preamble]implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 Mémoire virtuelle
 - Mémoire virtuelle
 - Multiprocessus
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

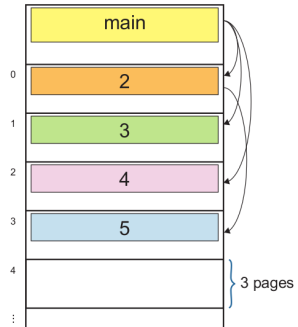
Choix d'implémentation

- L'emplacement dans la pile des Threads est géré avec une Bitmap
- 3 pages sont allouées à chaque Thread utilisateur
- La valeur maximale de SP que le main peut atteindre est fixée
- Le Thread Main n'apparaît pas dans la Bitmap

Gestion de la pile



Arborescence
de création des threads



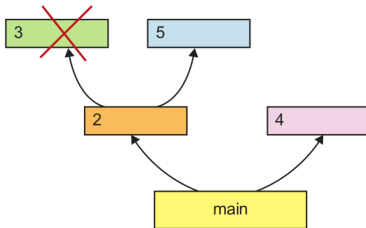
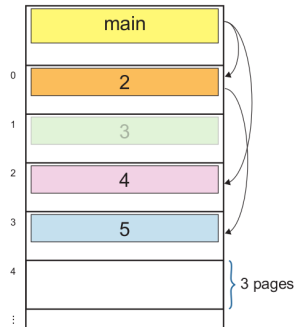
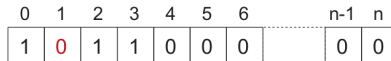
Organisation mémoire
des piles de threads

0	1	2	3	4	5	6	...	n-1	n
1	1	1	1	0	0	0		0	0

BitMap

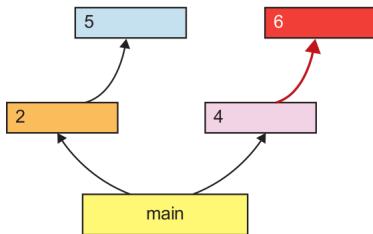
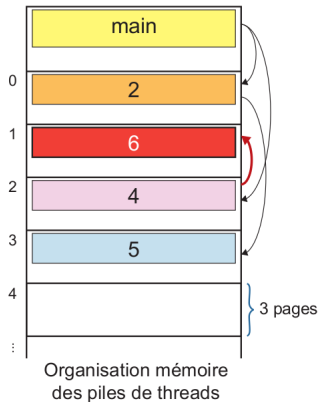
blocs mémoire occupés pour les piles de threads

Gestion de la pile

Arborescence
de création des threadsOrganisation mémoire
des piles de threads**BitMap**

blocs mémoire occupés pour les piles de threads

Gestion de la pile

Arborescence
de création des threads**BitMap**

blocs mémoire occupés pour les piles de threads

- 1 Nachos Input-Output
 - Appels systèmes et choix d'implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 Mémoire virtuelle
 - Mémoire virtuelle
 - Multiprocessus
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

Appels systèmes

- `int UserThreadCreate(void f(void *arg), void *arg);`
 - Créé un Thread utilisateur, qui exécute la fonction "f" avec les arguments "arg"
 - Retourne l'identifiant du Thread créé

Appels systèmes

- `int UserThreadCreate(void f(void *arg), void *arg);`
 - Créé un Thread utilisateur, qui exécute la fonction "f" avec les arguments "arg"
 - Retourne l'identifiant du Thread créé
- `void UserThreadExit();`
 - Termine un Thread utilisateur

Appels systèmes

- `int UserThreadCreate(void f(void *arg), void *arg);`
 - Créé un Thread utilisateur, qui exécute la fonction "f" avec les arguments "arg"
 - Retourne l'identifiant du Thread créé
- `void UserThreadExit();`
 - Termine un Thread utilisateur
- `void UserThreadJoin(int tid);`
 - Attend la terminaison du Thread utilisateur "tid"

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);
 - int Sem_P(semaphore s);

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);
 - int Sem_P(semaphore s);
 - int Sem_V(semaphore s);

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);
 - int Sem_P(semaphore s);
 - int Sem_V(semaphore s);
 - int Sem_GetValue(semaphore s);

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);
 - int Sem_P(semaphore s);
 - int Sem_V(semaphore s);
 - int Sem_GetValue(semaphore s);
 - int Sem_Destroy(semaphore s);

Autres ajouts

Synchronisation

- Nouveau type "semaphore" au niveau utilisateur, avec les appels systèmes qui permettent de le manipuler.
 - semaphore Sem_Init(int nbJetons);
 - int Sem_P(semaphore s);
 - int Sem_V(semaphore s);
 - int Sem_GetValue(semaphore s);
 - int Sem_Destroy(semaphore s);
- Fonctionnement avec une table de sémaphores noyau

Choix d'implémentation

- Le nombre de Threads est limité à 50

Choix d'implémentation

- Le nombre de Threads est limité à 50
- La gestion de la terminaison des Threads est laissée à l'utilisateur, sauf pour le main

Choix d'implémentation

- Le nombre de Threads est limité à 50
- La gestion de la terminaison des Threads est laissée à l'utilisateur, sauf pour le main
- Un Thread peut faire un `UserThreadJoin()` sur n'importe quel Thread créé lors de l'exécution du programme

Choix d'implémentation

- Le nombre de Threads est limité à 50
- La gestion de la terminaison des Threads est laissée à l'utilisateur, sauf pour le main
- Un Thread peut faire un `UserThreadJoin()` sur n'importe quel Thread créé lors de l'exécution du programme
- L'identifiant d'un Thread est unique, il n'est jamais réutilisé

Choix d'implémentation

- Le nombre de Threads est limité à 50
- La gestion de la terminaison des Threads est laissée à l'utilisateur, sauf pour le main
- Un Thread peut faire un `UserThreadJoin()` sur n'importe quel Thread créé lors de l'exécution du programme
- L'identifiant d'un Thread est unique, il n'est jamais réutilisé
- C'est à l'utilisateur d'utiliser correctement `UserThreadJoin()`

Choix d'implémentation

- Le nombre de Threads est limité à 50
- La gestion de la terminaison des Threads est laissée à l'utilisateur, sauf pour le main
- Un Thread peut faire un `UserThreadJoin()` sur n'importe quel Thread créé lors de l'exécution du programme
- L'identifiant d'un Thread est unique, il n'est jamais réutilisé
- C'est à l'utilisateur d'utiliser correctement `UserThreadJoin()`
 - Par exemple, l'utiliser sur un identifiant de Thread incorrect donnera lieu à un comportement indéfini.

- 1 Nachos Input-Output
 - Appels systèmes et choix d'implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 **Mémoire virtuelle**
 - **Mémoire virtuelle**
 - Multiprocessus
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

Pagination

Mémoire virtuelle

- Frame Provider

Pagination

Mémoire virtuelle

- Frame Provider
- Politiques d'allocation

Pagination

Mémoire virtuelle

- Frame Provider
- Politiques d'allocation
 - FIRST_FREE_FRAME

Pagination

Mémoire virtuelle

- Frame Provider
- Politiques d'allocation
 - FIRST_FREE_FRAME
 - RANDOM_FREE_FRAME

- 1 Nachos Input-Output
 - Appels systèmes et choix d[Please insert into preamble]implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 **Mémoire virtuelle**
 - Mémoire virtuelle
 - **Multiprocessus**
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

Multiprocessus

Appel système

- Création d'un nouveau type utilisateur pid_t

Multiprocessus

Appel système

- Création d'un nouveau type utilisateur `pid_t`
- `pid_t ForkExec(char* filename)`
 - Crée un nouveau processus qui exécute le programme contenu dans le fichier `filename`.
 - Renvoie le `pid_t` correspondant au nouveau processus

Multiprocessus

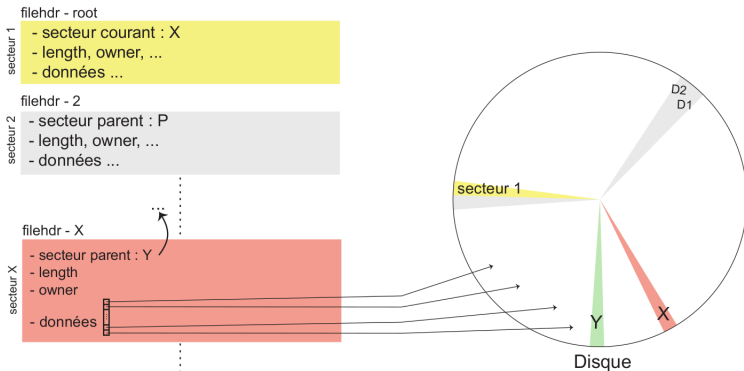
Appel système

- Création d'un nouveau type utilisateur `pid_t`
- `pid_t ForkExec(char* filename)`
 - Crée un nouveau processus qui exécute le programme contenu dans le fichier `filename`.
 - Renvoie le `pid_t` correspondant au nouveau processus
- `void waitpid(pid_t p)`
 - Attend la terminaison du processus `p`

- 1 Nachos Input-Output
 - Appels systèmes et choix d[Pleaseinsertintopreamble]implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 Mémoire virtuelle
 - Mémoire virtuelle
 - Multiprocessus
- 4 Système de fichiers
 - Arborescence de répertoires
 - Page des fichiers ouverts

Principe

- Le secteur contenant l'i-node du répertoire Root est le 1
- Un répertoire est représenté sur disque par un fichier Nachos classique



Fonctionnalités

- Création de répertoire (mkdir)

Fonctionnalités

- Création de répertoire (mkdir)
 - Le nouveau répertoire contiendra deux fichiers : "." et ".."

Fonctionnalités

- Création de répertoire (mkdir)
 - Le nouveau répertoire contiendra deux fichiers : "." et ".."
- Changement de répertoire courant (cd)

Fonctionnalités

- Création de répertoire (mkdir)
 - Le nouveau répertoire contiendra deux fichiers : "." et ".."
- Changement de répertoire courant (cd)
 - Parcours de l'arborescence grâce aux fichiers spéciaux "." et ".."

Fonctionnalités

- Création de répertoire (mkdir)
 - Le nouveau répertoire contiendra deux fichiers : "." et ".."
- Changement de répertoire courant (cd)
 - Parcours de l'arborescence grâce aux fichiers spéciaux "." et ".."
 - Le répertoire racine s'appelle "/"

- 1 Nachos Input-Output
 - Appels systèmes et choix d[Please insert into preamble]implémentations
- 2 Multi-Threading
 - Gestion de la pile
 - Appels Système
- 3 Mémoire virtuelle
 - Mémoire virtuelle
 - Multiprocessus
- 4 **Système de fichiers**
 - Arborescence de répertoires
 - **Page des fichiers ouverts**

Appels systèmes

- `int Create (char *name, int size);`
 - Retourne 1 si l'exécution s'est bien passé, 0 sinon
 - Crée un fichier "name" dans le répertoire courant, de taille "size"

Appels systèmes

- `int Create (char *name, int size);`
 - Retourne 1 si l'exécution s'est bien passé, 0 sinon
 - Crée un fichier "name" dans le répertoire courant, de taille "size"
- `OpenFileId Open (char *name);`
 - Ouvre le fichier "name", retourne un `OpenFileId`, manipulable par l'utilisateur
 - Renvoie -1 si l'ouverture est impossible

Appels systèmes

- `int Write (char *buffer, int size, OpenFileId id);`
 - Écrit la chaîne de taille "size" contenue dans "buffer" dans le fichier "id"
 - Renvoie le nombre de bytes effectivement écrits

Appels systèmes

- `int Write (char *buffer, int size, OpenFileId id);`
 - Écrit la chaîne de taille "size" contenue dans "buffer" dans le fichier "id"
 - Renvoie le nombre de bytes effectivement écrits
- `int Read (char *buffer, int size, OpenFileId id);`
 - Remplit "buffer" avec un nombre de caractères "size" lus depuis le fichier "id"
 - Renvoie le nombre de bytes lus

Appels systèmes

- `int Write (char *buffer, int size, OpenFileId id);`
 - Écrit la chaîne de taille "size" contenue dans "buffer" dans le fichier "id"
 - Renvoie le nombre de bytes effectivement écrits
- `int Read (char *buffer, int size, OpenFileId id);`
 - Remplit "buffer" avec un nombre de caractères "size" lus depuis le fichier "id"
 - Renvoie le nombre de bytes lus
- `void Close (OpenFileId id);`
 - Ferme le fichier "id"

Table des fichiers ouverts

- Interface entre le système de fichiers et le programme utilisateur.
- Nachos permet l'ouverture de 10 fichiers simultanément au total au niveau des programmes utilisateurs
- Les indices 0 et 1 se sont jamais utilisés, ils sont réservés.

