NACHOS
in a Nutshell

Raymond Namyst, Luc Bougé

Laboratoire de l'informatique du parallelisme
Ecole normale superieure de Lyon
69364 Lyon Cedex 07

NACHOS : Not Another Completely Heuristic Operating System

Plan

- Introduction
  . Vue générale : machine virtuelle, noyau du système
- Quelques éléments de C++
  . Classes, objets, encapsulation, constructeurs, allocation dynamique
- Emulation du matériel
  . Mémoire, MMU, processeur, interruptions, E/S
- Noyau du système
  . Environnement, processus, ordonnanceur, synchronisation
- Support pour l'exécution de programmes « utisateurs »
  . Appels systèmes, espaces d'adressage

NACHOS : un véritable OS dans un processus Unix

• Principe

- NACHOS = émulation matérielle (MIPS) + système d'exploitation
- Deux modes d'exécution (Kernel / User) :
  . Le processus s'exécute normalement en mode « réel »
  . Il est capable de charger et d'exécuter des binaires MIPS, auquel cas il simule un mode utilisateur parfaitement protégé

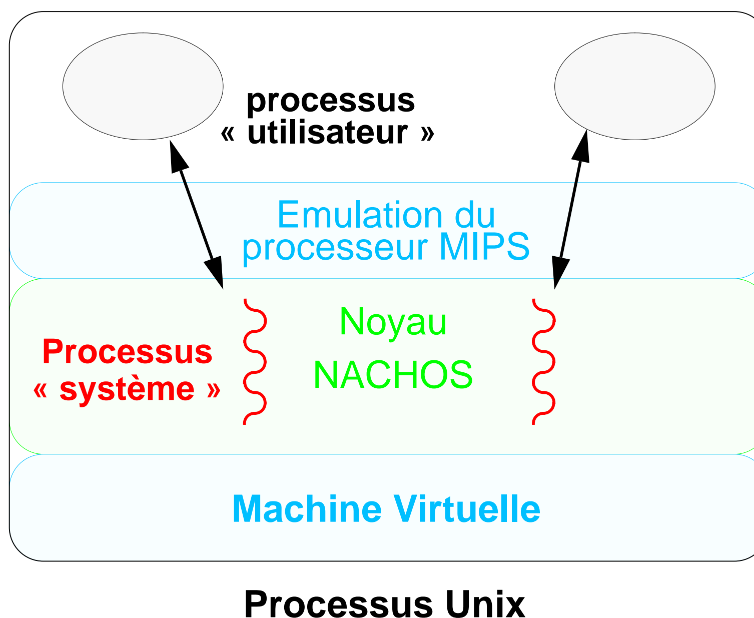
• Transitions Kernel <-> User

- K -> U : lancement de l'interprète, retour d'interruption
- U -> K : interruption matérielle, appel système

NACHOS : Not Another Completely Heuristic Operating System

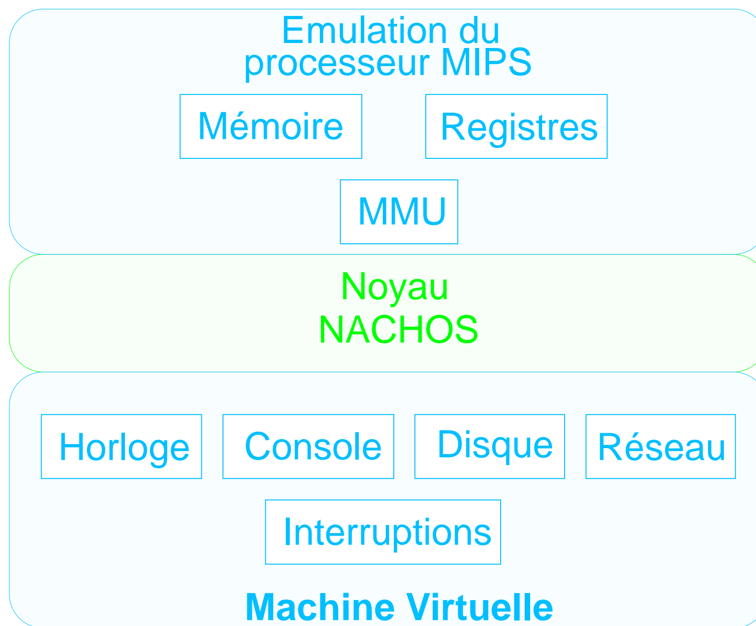
Schéma général

• Des processus « noyaux » supportent l'exécution de programmes utilisateurs



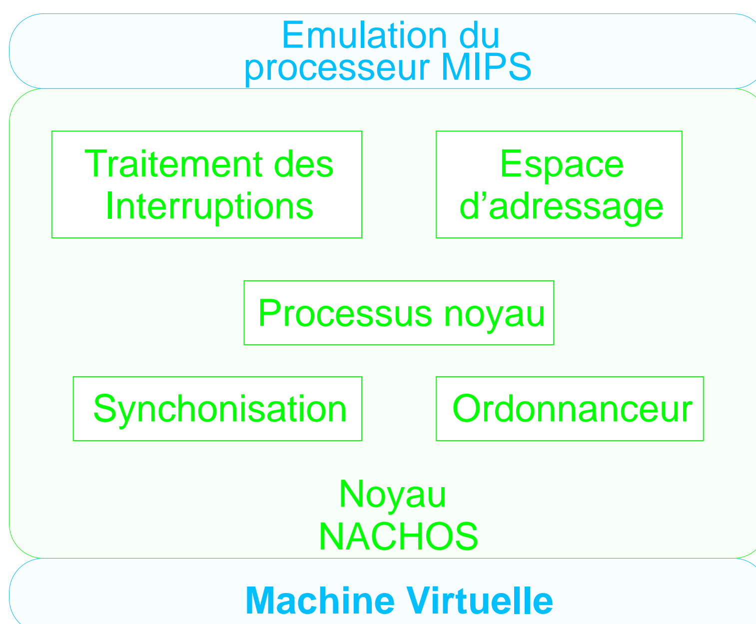
Machine virtuelle et émulation MIPS

- Composants principaux



Noyau NACHOS

- Composants principaux



Quelques éléments de C++

Historique

- Créateur : Bjarne Stroustrup (AT&T Laboratories)
- Historique :
  - 1980 : “C with classes”
  - 1983 : “C++”
- Objectifs :
  - Garder les avantages de C
    - Efficacité du code généré + parc logiciel énorme (UNIX)
  - Bénéficier d’une meilleure qualité de programmation

    => C++ = extension de C aux objets

Généralités

• C++ est un langage à objets à base de classes qui fournit :

- l'encapsulation «partielle»
- l'héritage multiple
- la généricité
- etc.

• Rappel : Classe = Données + Traitements (~ types en C++)

- les données sont propres à chaque instance (ou objet)
- les traitements sont communs à toutes les instances d'une classe

• Remarque : En C++, les classes n'ont d'existence qu'à la compilation

Spécification d'une classe

• Exemple : une classe "Compteur"

Compteur.h

```
class Compteur {
   public:
       void raz();
       void inc();
       int value() const;
   private:
       int val;
};
```

Implantation d'une classe

Compteur.cc

```
void Compteur::raz()
{
   val = 0;
}

void Compteur::inc()
{
   val++;
}

int Compteur::value() const
{
   return val;
}
```


Instanciation d'une classe

- Déclaration statique d'une instance de "Compteur"

```
{
   Compteur c;   // Creation d'un objet
                 // Compteur
   c.raz();
   c.inc(); c.inc();
   printf("%d", c.value());
   // Affiche: 2
}
```

- Remarque : L'instruction

```
printf("%d", c.val);
```

entraînerait une erreur à la compilation...

• Version dynamique

```
{
   Compteur *c;

   c = new Compteur;
   c->raz();
   c->inc(); c->inc();
   printf("%d", c->value());
   // Affiche: 2
   delete c;
}
```

• Opérateurs *new* et *delete*

- Mieux vaut les employer systématiquement en lieu et place de *malloc* / *free*
- Garantissent la gestion correcte des objets complexes

• En C, on écrivait :

```
int *zone;

   zone = (int *)malloc(sizeof(int)*5);
   // Alloue un tableau de 5 entiers

   free(zone);
   // libere l'espace occupe par zone
```

• En C++, on écrit :

```
int *zone;

   zone = new int[5];

   delete []zone;
```

(les crochets ne sont nécessaires que dans le cas des tableaux)

La pseudo-variable *this*

- Utilisable uniquement dans le corps d'une méthode d'objet
- Permet d'obtenir un pointeur sur l'objet courant

```
class C {
   public:
       void positionner_x(int x);
   private:
       int x;
};
```

Comment accéder à la variable *x* dans le corps de la fonction *positionner_x* ?

```
void C::positionner_x(int x)
{
   this->x = x;
}
```


Initialisation automatique : Les Constructeurs

- Initialisation automatique du Compteur :

```
class Compteur {
   public:
       Compteur();
       void raz();
       ...
};

Compteur::Compteur()
{
   raz();
}
```

• Les constructeurs peuvent avoir des paramètres :

```
class Compteur {
   public:
       Compteur(int v) { val = v; }
       ...
   private:
       int val;
};

Compteur c1(10);// ok
Compteur c2;     // interdit

Compteur *c3 = new Compteur(10);// ok
Compteur *c4 = new Compteur; // interdit
```


Libération automatique de mémoire : les Destructeurs

• Appelés automatiquement lorsque le bloc englobant l'objet se termine, ou lorsque l'objet est détruit par l'opérateur ***delete***

```
class String {
   public:
       String(char *s) { ... new ... }
       ~String();
       ...
   private:
       char *str;
};

String::~String()
{
   delete []str;
}
```

- Invocation implicite

```
String *ptr;

   ptr = new String("coucou");
   ...
   delete ptr;
// liberation de ptr->str, puis de ptr
```


Emulation du matériel

file://nachos/code/machine/

La classe Machine

- Décrit la machine physique, et ne possède donc qu'une seule instance
  - contient le code déclenchant l'interprète (méthode *Run()*)
  - communications Kernel/User s'effectuent via mémoire + registres
- Mémoire physique
  - variable *mainMemory*
  - par défaut, 32 pages de 128 octets !
- Registres
  - manipulés via *ReadRegister()* et *WriteRegister()*
- Memory Management Unit
  - table de conversion (variable *pageTable*) adresse physique/virtuelle

NACHOS : Not Another Completely Heuristic Operating System

- Utilisation transparente de la MMU
  - la mémoire est accédée via ReadMem() et WriteMem()
  - ces fonctions utilisent la méthode *Translate* :

```
Machine::Translate(int virtAddr, int* physAddr, int size, bool writing);
```

qui effectue la conversion d'adresse en utilisant pageTable

  - toutes ces fonctions vérifient l'alignement, les protections, etc.

```
bool Machine::WriteMem(int addr, int size, int value)
{
   int physicalAddress;

   Translate(addr, &physicalAddress, size, TRUE);
   switch (size) {
       case 1: ...
       case 2: ...
       case 4: *(unsigned int *)&machine->mainMemory[physicalAddress]
                      = WordToMachine((unsigned int) value);
...
```

La classe Interrupt

- Tâches
  - faire progresser une «heure logique» (méthode *OneTick()*)
  - maintenir les statistiques UserTime/KernelTime
  - déclencher des interruptions pré-programmées (à l'aide de *Schedule()*)

```
void Interrupt::Schedule(VoidFunctionPtr handler, int arg,
                         int fromNow, IntType type)
```

  - masquer/autoriser les interruptions (*SetLevel(IntOn / IntOff)*)
  - éteindre la machine (et donc NACHOS !) : *Halt()*

- Beaucoup d'objets sont « clients » de l'objet interrupt
  - Timer, Disk, Console, Network, etc.
  - Timer ré-arme une nouvelle interruption à chaque fois...

NACHOS : Not Another Completely Heuristic Operating System

Noyau du système

file://nachos/code/thread/

Initialisation du système d'exploitation

- Fichiers *main.cc* et *system.cc*
  - *main.cc* traite les arguments de la ligne de commande NACHOS
    . ex: nachos -x test
    lance le système en demandant l'exécution du programme «test»
  - *system.cc* initialise toutes les structures du noyau
- Le comportement de NACHOS dépend de l'endroit où il est compilé
  - code/machine/ : impossible
  - code/threads/ : NACHOS exécute un petit test interne de la multiprogrammation
  - code/userprog/ : NACHOS peut exécuter un programme utilisateur
  - code/vm/ : NACHOS peut exécuter plusieurs programmes !

Processus et ordonnancement

- La classe Thread
  - Gestion de processus indépendants
  - Fork(), Yield(), Sleep(), etc.
- La classe Scheduler : juste une liste de Threads prêts !

Synchronisation

- La classe Semaphore
  - Semaphore::Semaphore(nom, valeur)
  - Semaphore::P() et Semaphore::V()
- Verrous et variables de condition
  - Les prototypes sont écrits... mais pas le corps des fonctions !

- Corps de Semaphore::P()

```
void Semaphore::P()
{
   IntStatus oldLevel = interrupt->SetLevel(IntOff);
           // disable interrupts

   while (value == 0) { // semaphore not available
       queue->Append((void *)currentThread);// so go to sleep
       currentThread->Sleep();
   }
   value--;           // semaphore available, consume its value

   (void) interrupt->SetLevel(oldLevel);// re-enable interrupts
}
```


Support pour l'exécution de programmes

file://nachos/code/userprog/

Les espaces d'adressage

- Classe AddrSpace
  - Le constructeur prend un nom de fichier en paramètre
    -> binaire MIPS chargé en mémoire
  - Chaque espace contient une table de translation (= *pageTable*)
  - Une méthode *InitRegisters()* prépare le processeur pour l'exécution
  - Si un thread possède un espace d'adressage, il est sauvé / restauré à chaque changement de contexte

- Illustration : chargement et exécution d'un processus

```
void StartProcess(char *filename)
{
   OpenFile *executable = fileSystem->Open(filename);
   AddrSpace *space;

   space = new AddrSpace(executable);
   currentThread->space = space;

   delete executable;// close file

   space->InitRegisters();// set the initial register values
   space->RestoreState();// load page table register

   machine->Run();// jump to the user progam
    ASSERT(FALSE);// machine->Run never returns;
}
```