



# NACHOS

## - Simulation d'un OS -

BARTHELEMY Romain  
MORISON Jake

EUDES Robin  
ROSSI Ombeline

# SOMMAIRE

- Présentation de NachOS
- Objectif 1 : Comprendre le fonctionnement d'un SYSCALL
- Objectif 2 : Comprendre les threads utilisateur
- Objectif 3 : Comprendre la pagination
- Objectif 4 : Avoir une gestion multi-processus
- Bilan



# NachOS... c'est quoi ?

- Une simulation d'OS
  - Une machine MIPS
  - « Nous » avons le contrôle d'un processeur MIPS, de la mémoire...
- *Pour résumer, on contrôle la machine, nous voilà OS, alors même que nachOS n'est qu'un processus linux sur notre machine Host !*



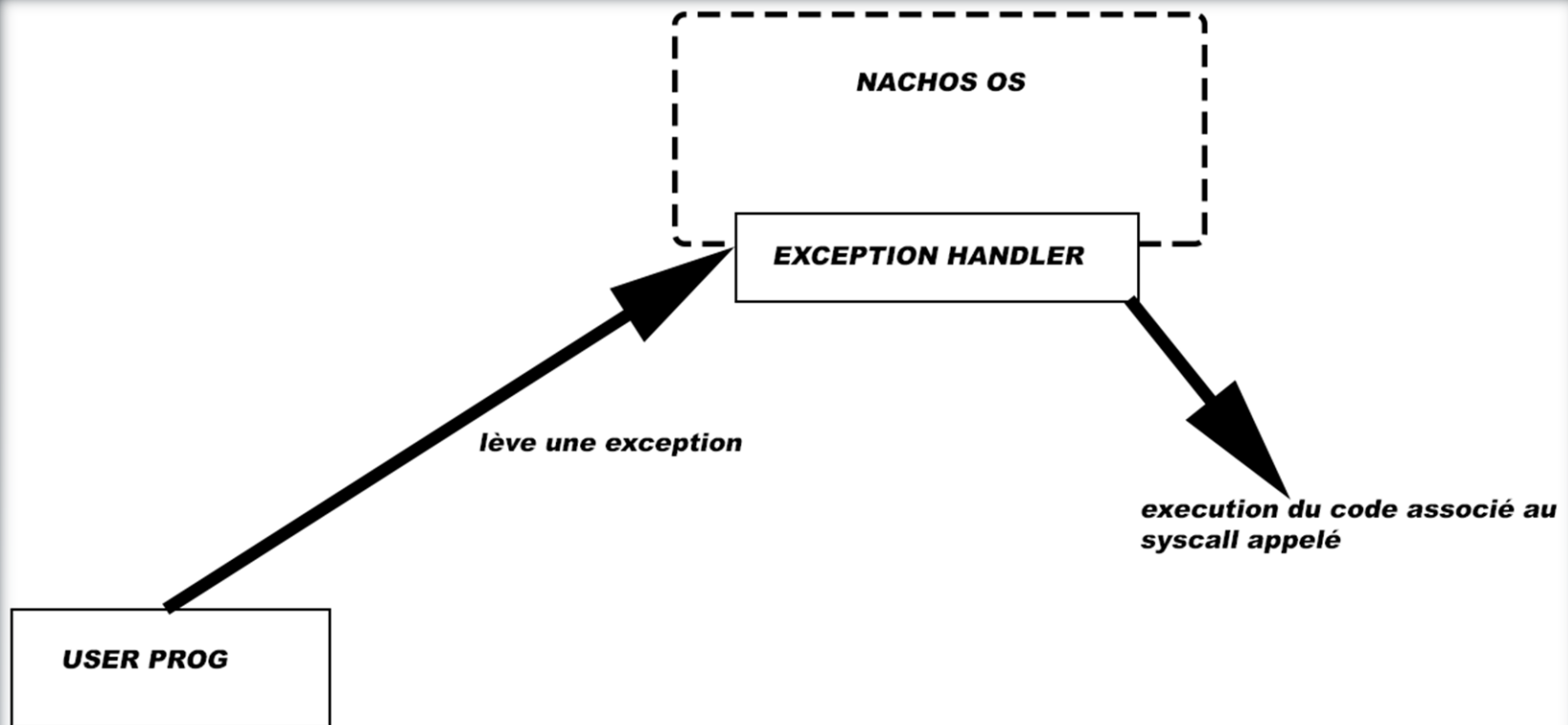
# Une simulation... simplifiée

- Bien entendu, de nombreux mécanismes présents dans les OS sont simplifiés :
  - *Gestion des fichiers*
  - *Allocation mémoire*
  - *Système de pagination « simple » et à un seul niveau...*
  - *Etc.*



# Objectif 1 : Les SYSCALL

# Un SYSCALL... c'est quoi ?

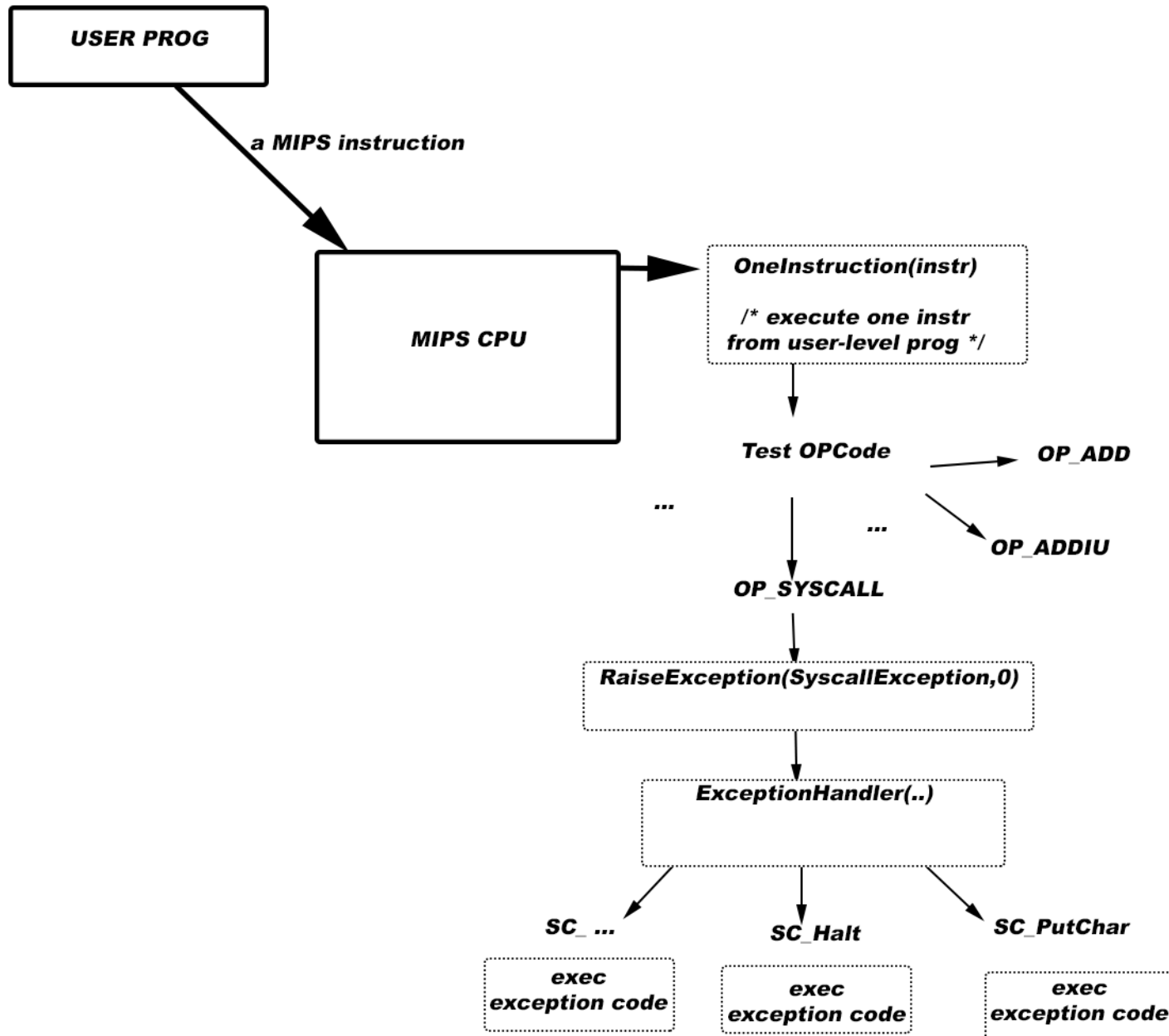




# Comment apparait un syscall ?

- Chaque instruction d'un programme utilisateur est compilé pour le processeur MIPS
  - Le processeur décode chaque instruction
  - Si le Code de l'instruction correspond à un syscall, une exception est levée !
  - Exception récupérée par un Handler, qui exécutera un flot d'instruction associé au SYSCALL.
  - Le SYSCALL terminé, retour au programme utilisateur, pc+4.

# Autrement dit...





# Etude de cas : le syscall PutString

```
#include "syscall.h"

int main(){
    SynchPutString("bon\127our\n");
    Exit(0);
}
```

# Etude de cas : le syscall PutString

```
case SC_SynchPutString:{
    char *buffer=new char[MAX_STRING_SIZE];
    int s = machine->ReadRegister (4);      //L'adresse du string à insérer est dans le registre 4
    copyStringFromMachine(s, buffer, MAX_STRING_SIZE);
    synchconsole->SynchPutString(buffer);    //On fait appel à SynchPutString de synchconsole
    delete buffer;
    break;
}
```

```
void SynchConsole::SynchPutString(const char s[])
{
    SemPut->P();
    int i;
    for (i=0;i<MAX_STRING_SIZE && s[i]!='\0';i++){
        if (s[i]=='\0')
            return;          // if end of string, quit
        console->PutChar ((char)s[i]);
        writeDone->P ();      // wait for write to finish
    }
    SemPut->V();
}
```

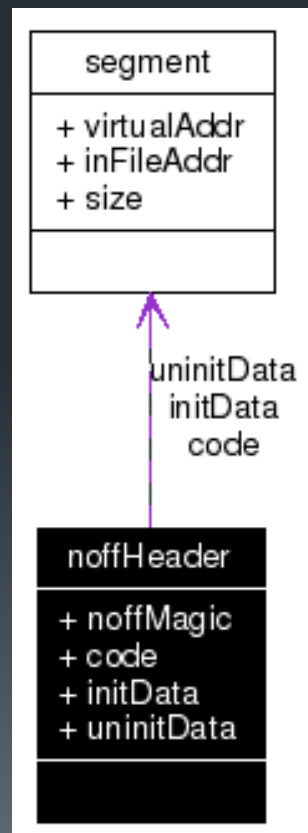
# Un petit détour par MIPS...

```
//-----  
// copyStringFromMachine : Used for SynchPutString  
//-----  
void copyStringFromMachine( int from, char *to, unsigned size){  
    unsigned int i;  
    int tmp;  
    for(i=0;i<size;i++){  
        if(machine->ReadMem(from+i,1,&tmp))  
            to[i]=tmp;  
    }  
    if(tmp!='\0'){  
        to[size-1]='\0';  
    }  
}
```



## Objectif 2 : Les Threads Utilisateurs

# Comment est chargé un programme user en mémoire ?





# Thread Utilisateur

- « Encadré » par un thread Nachos
- Paramètres du Fork : StartUserThread, params
- StartUserThread :
  - Initialise les registres
  - Alloue l'espace dans la pile
  - Assure la cohérence de la bitmap



# Utilisation d'une bitmap

- Gestion de l'allocation de la pile
  - Joue le rôle d'un index des pages utilisées par les threads du processus auquel est associé l'espace mémoire courant.

# Initialisation de la bitmap

## Bitmap initiale

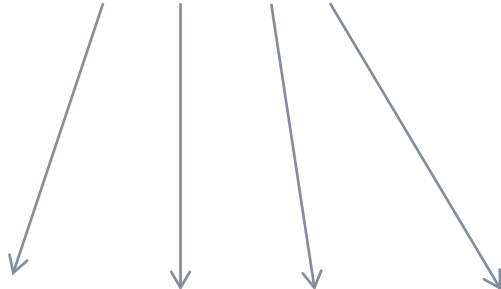
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
...	...	...	...	...	...	...

```
BitMap::BitMap (int nitems)
{
    numBits = nitems;
    numWords = divRoundUp (numBits, BitsInWord);
    map = new unsigned int[numWords];
    for (int i = 0; i < numBits; i++)
        clear (i);
}
```



# Arrivée du thread principal

```
//Main program's stack marked  
stack = new BitMap(divRoundUp(UserStackSize, PageSize));  
for(i=0; i<NbPagesThread; i++){  
    stack->Mark(i);  
}
```



1	1	1	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
...	...	...	...	...	...	...

# Allocation dans la pile

```
int
AddrSpace::AllocStack ()
{
    SemThread->P();
    //On veut allouer 4 pages pour chaque thread
    if((stack->NumClear())<=(NbPagesThread-1)){
        printf("Stack overflow\n");
        return -1;
    }
    int tmp = stack->Find();
    for(int i=1;i<NbPagesThread;i++){
        if(stack->Test(tmp+i)){
            printf("Pages block(%d) is not available\n",NbPagesThread);
            return -1;
        }
        stack->Mark(tmp+i);
    }
    nbThreads++;
    SemThread->V();
    return tmp;
}
```

# Attention au stack overflow !

- Plus on utilise un index grand dans la bitmap, plus on remonte la pile ! Attention à ne pas dépasser les limites de l'espace mémoire...

```
//-----  
// AddrSpace::StackValue  
//      Returns the stack value associated to the bitmap value.  
//-----  
  
int  
AddrSpace::StackValue(int BitmapValue)  
{  
    return PageSize*numPages - BitmapValue*PageSize;  
}
```

# Plusieurs syscall pour manipuler les threads users

```
case SC_UserThreadCreate:{
    do_UserThreadCreate(machine->ReadRegister (4), machine->ReadRegister (5));
    //Le résultat de la fonction est stocké dans le bon registre au coeur du code
    break;
}
case SC_UserThreadExit:{
    do_UserThreadExit();    //On quitte le thread
    break;
}
case SC_UserThreadJoin:{
    UserThreadJoin(machine->ReadRegister (4));    //UserJoin call
    break;
}
```

## THREAD PRINCIPAL

exécute à  $t_0$

exécute à  $t_1$

$t_1 > t_0$

`do_userThreadCreate(..)`

Le Thread principal continue  
son flot d'exécution après  
chaque Fork

`do_userThreadCreate(..)`

`Fork(StartUserThread(..),...)`

`Fork(StartUserThread(..),...)`

THREAD A

THREAD B

flot d'exécution A

`UserThreadJoin( ... )`

On veut attendre ce thread

ATTENTE

`do_UserThreadExit()`

On peut continuer notre exécution

flot d'exécution B

`do_UserThreadExit()`





## Pour que le « Thread join » fonctionne...

- Allocation dans la stack non concurrente
- Utilisation de sémaphores pour « mettre en attente »
- Libération des sémaphores lors de l'exit

# Syscall user join

```
int UserThreadJoin(int t){
    if(currentThread->dependance!=-1){
        printf("Le thread possède déjà une dépendance\n");
        return -1;
    }
    if(currentThread->initStackReg==t || t==0){
        printf("Tentative de dépendance vers un thread invalide\n");
        return -1;
    }
    CheckThreadExistence->P();
    if(!currentThread->space->Test(t)){
        printf("Tentative de dépendance vers un thread non existant\n");
        CheckThreadExistence->V();
        return -1;
    }
    CheckThreadExistence->V();
    currentThread->dependance=t;
    currentThread->space->TabSemJoin[t]->P();
    return 0;
}
```

# Syscall user exit

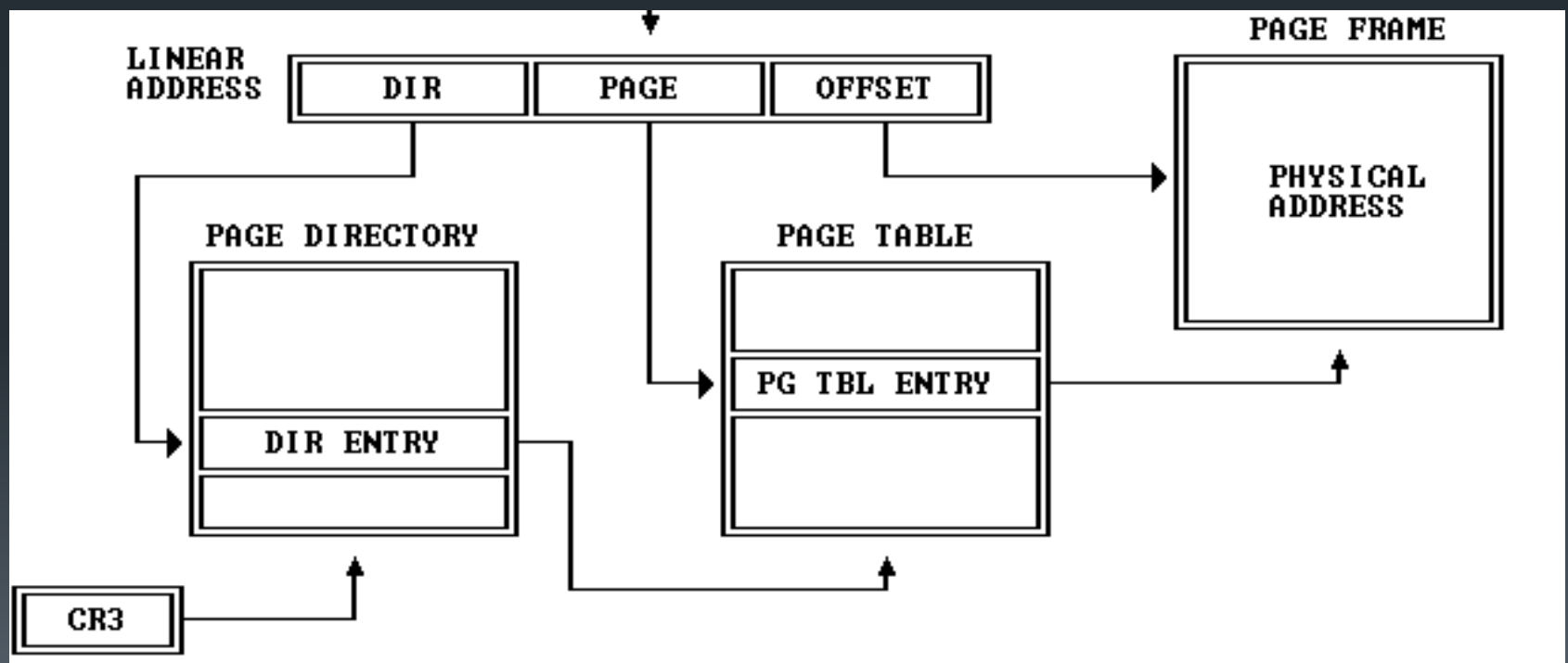
```
int do_UserThreadExit(){
    //On ne finit pas le thread avec un appel à cette fonction lorsque nous sommes dans l'appel principal
    if(currentThread->initStackReg==0){
        return 0;
    }
    currentThread->space->TabSemJoin[currentThread->initStackReg]->V();
    if(currentThread->dependance!=-1)
        currentThread->space->TabSemJoin[currentThread->dependance]->V();
    currentThread->space->FreeStack(currentThread->initStackReg);
    currentThread->Finish();
    return 0;
}
```





## Objectif 3 : La pagination

# Commencer passer d'une adresse virtuelle à une adresse physique ?



# Mise en place d'un FrameProvider

- Gestion de l'allocation
- Utilisation d'une bitmap pour savoir quelles frames sont libres
- Diverses autres fonctions...

```
void
FrameProvider::ReleaseFrame(int numFrame)
{
    if(!phyMemBitmap->Test(numFrame)){
        printf("Error, numFrame, %d\n", numFrame);
        return;
    }
    // liberation frame
    phyMemBitmap->Clear(numFrame);
}
```



## Objectif 4 :

# Une gestion Multi-processus



# Plusieurs processus...

- Un espace mémoire isolé pour chaque processus.
- Chaque processus peut lancer des threads dans son espace mémoire.
- NachOS s'arrêtera lorsque le dernier processus aura terminé.
  - -> un syscall ForkExec pour faire tout ça...

```
int ForkExec(char *s){
    OpenFile *executable = fileSystem->Open (s);
    AddrSpace *space;

    if (executable == NULL)
    {
        printf ("Unable to open file %s\n", s);
        return -1;
    }

    Thread *t = new Thread("UserProcess");

    if(t==NULL){
        printf("Error: Thread non created\n");
        return -1;
    }

    MajNbProcess(1);

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

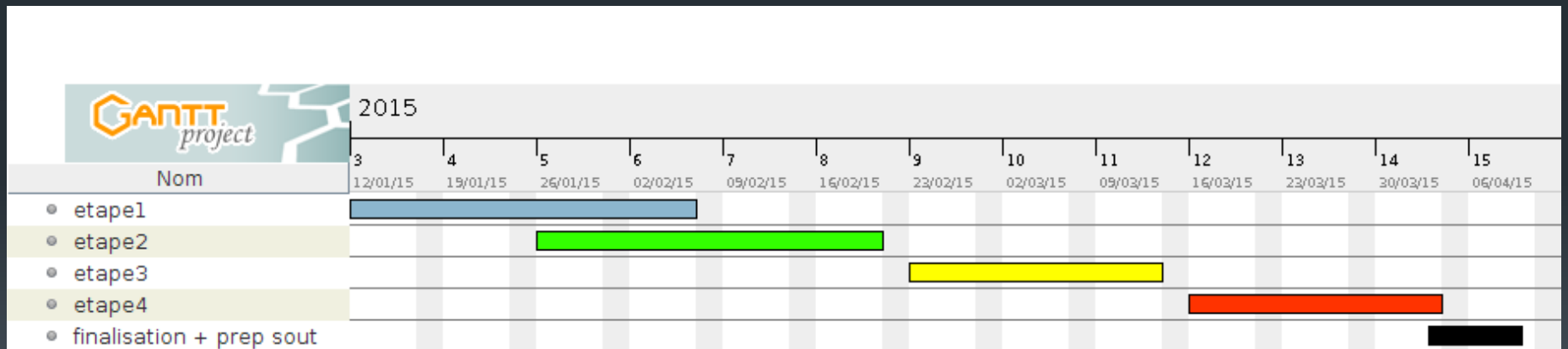
    delete executable;          // close file

    t->ForkExec(StartUserProcess,0);

    //Permet de démarrer le processus créé
    currentThread->Yield();

    return 0;
}
```

# Planning





# Bilan

- Une mise en pratique des notions vues au cours de l'année
- Un aperçut de la « complexité » d'un OS, même sur cette simulation « simplifiée »