Compte Rendu

Nom : Hugo Wendjaneh, Louis Maury, Fouad Id Gouahmane **Date** : 15-04-2025 **Sujet** : Systèmes d'exploitation **Contexte** : OS XV6 Risc V

1. Objectif

L'objectif du cours est de présenter les concepts et les techniques fondamentales utilisées dans les systèmes d'exploitation modernes. Ces TPs porteront en particulier sur : la gestion des processus (processus, thread, ordonnancement, synchronisation), la communication interprocessus (IPC), la gestion mémoire (segmentation, pagination et mémoire virtuelle) ainsi que le système de fichier.

2. Qu'est-ce que le XV6 ?

XV6 est un système d'exploitation éducatif basé sur Unix, conçu pour être simple et facile à comprendre. Il est écrit en C et assembleur, et il est utilisé pour enseigner les concepts fondamentaux des systèmes d'exploitation. XV6 est une version simplifiée de Unix Version 6 (V6), qui a été développé dans les années 1970.

3. Répartition des taches

Nom	Tâches
Louis Maury	Implémentation de la commande `pstree`
Hugo Wendjaneh	Implémentation des commandes `psync_tst` et `pmem_test`
Fouad Id Gouahmane	Implémentation de la commande `phierarchy_tst`

3. TP 2: Réalisation

3.1 Création d'une commande pstree

Implémenter la commande UNIX pstree. Cette commande doit permettre l'affichage d'un "pseudo" arbre qui permet de visualiser les liens de parentées entre les processus. Votre implémentation sera simplifiée afin de limiter les modifications au sein du kernel de XV6. Ainsi plusieurs branches pourront être redondantes.

- `user/Makefile` : Ajout de la commande pstree dans le Makefile pour compilation.

```
    $U/_pstree\
```

- `user/pstree.c` : Implémentation de la commande pstree.

```
#include "kernel/types.h"
#include "kernel/stat.h"
```

```c
#include "user/user.h"

int main(int argc, char *argv[])

{
  if(argc != 1){
    fprintf(2, "Usage: pstree\n");
    exit(1);
  }
  pstree();
  exit(0);
}
```

- `user/user.h` : Ajout de la déclaration de la fonction `pstree()`.

```c
int pstree(void);
```

- `user/usys.pl` : Ajout de la déclaration de la fonction `pstree()` dans le fichier usys.pl pour la génération des appels système.

```
entry("pstree");
```

- `kernel/proc.c` : Ajout de la fonction `proctree()` dans le fichier proc.c pour la gestion des processus.

```c
void
proctree(void)
{
  struct proc *p;
  printf("\n");
  for (p = proc; p < &proc[NPROC]; p++) {
    if (p->state == UNUSED)
      continue;
    if (p->parent == 0) {
      printf("PROCESS %s\tPID: %d", p->name, p->pid);
    } else {
      printf("PARENT %s\tPID: %d\n└── PROCESS %s\tPID: %d", p->parent->name, p-
>parent->pid, p->name, p->pid);
    }
    printf("\n");
  }
}
```

- `kernel/defs.h` : Ajout de la déclaration de la fonction `proctree()` dans le fichier defs.h pour la gestion des processus.

```
void proctree(void);
```

- `kernel/sysproc.c` : Ajout de la fonction `proctree()` dans le fichier sysproc.c pour la gestion des appels système.

```
uint64 sys_pstree(void)
{
  proctree();
  return 0;
}
```

- `kernel/syscall.c` : Ajout de la déclaration de la fonction `sys_pstree()` dans le fichier syscall.c pour la gestion des appels système.

```
[SYS_pstree]  sys_pstree
```

- `kernel/syscall.h` : Ajout de la déclaration de la fonction `SYS_pstree` dans le fichier syscall.h pour la gestion des appels système à l'aide des identifiants commandes (23).

```
#define SYS_pstree 23
```

3.2 Synchronisation des processus

On veut tester la synchronisation entre processus parent/enfant. On se base sur l'affichage d'un message partagé entre deux processus. Vous allez créer un programme qui contiendra le processus parent qui attendra que son processus fils ait affiché la première partie du message pour affciher la seconde. Il sera possible de ne pas respecter la synchronisation comme parametre du programme.

- `user/psync_tst.c`: Implémentation du programme de test de synchronisation entre processus.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  int wait_flag = 1; // Default wait for child

  if(argc > 2){
    fprintf(2, "Usage:\n\ttpsync_tst\n\tpsync_tst [1 or 2]\n\t0: do not wait
for child, 1: wait for child\n");
    exit(1);
```

```
  }

  // If a parameter is provided, check its value.
  if(argc == 2) {
    int param = atoi(argv[1]);
    if(param != 1 && param != 0){
      fprintf(2, "Usage:\n\ttpsync_tst\n\tpsync_tst [1 or 0]\n\t0: do not wait
for child, 1: wait for child\n");
      exit(1);
    }
    wait_flag = (param == 1);
  }

  printf("-- Test process synchronisation --\n");

  int pid = fork();

  if(pid < 0){
    fprintf(2, "Humm, something went wrong. Fork failed\n");
    exit(1);
  }

  if(pid == 0){
    // Child process
    printf("Hi, I am the child process with pid %d\n", getpid());
    printf("I will sleep for 2 seconds\n");
    sleep(2);
    printf("I end my life now\n");
    exit(0);
  } else {
    // Parent process
    if(wait_flag) {
      int child_pid = wait(0);
      if(child_pid < 0) {
        fprintf(2, "Humm, something went wrong. The parent has no child
process.\n");
        exit(1);
      }
    }
    printf("Hi, I am the parent process with pid %d\n", getpid());
    if(wait_flag)
      printf("I was waiting for my child process to finish.\n");
    else
      printf("I did not wait for my child process.\n");
  }
  exit(0);
}
```

- `Makefile`: Ajout de la commande psync_tst dans le Makefile pour compilation.

```
  $U/_psync_tst\
```

Ici nous avons implémenté un programme de test de synchronisation entre processus. Le programme prend un argument qui détermine si le processus parent doit attendre la fin du processus enfant avant de continuer son exécution. Si l'argument est 1, le parent attend la fin du fils, sinon il continue sans attendre. Le programme crée un processus enfant qui affiche un message, attend 2 secondes, puis se termine. Le processus parent affiche également un message et indique s'il a attendu la fin du processus enfant ou non.

Si l'argment de la fonction est 0, alors le parent ne doit pas attendre et peut continuer son exécution immédiatement. Cela crée des problème d'affichage, car le parent veux afficher sont message en même temp que le fils. Il est donc possible d'avoir un affichage comme celui-ci :

```
Hi, HiI am ,t heI p araent prm theo cess wcihth pidi l7d p
roI cdeid snots  wawiitht  pfiord  my8
cIh ilwdi lprl socleeseps.
for 2 second$ s
I end my life now
```

Nous pouvons voir que les deux processus essaient d'afficher en même temps, ce qui crée un affichage confus. Cela est dû à la nature concurrente des processus et à l'absence de synchronisation entre eux.

3.2.1 Diagrame UML représentant la concurrence entre les processus

Synchronisation processus

3.3 Parentalité des processus

L'objectif de cette partie est de vérifier la conformité de la parentalité des processus dans le système d'exploitation XV6. Nous allons créer un programme qui génère une hiérarchie de processus en fonction du nombre d'enfants et du nombre de générations spécifiés en arguments.

- `user/phierarchy.c`: Implémentation du programme de test de parentalité des processus.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

/**
  * run pstree command in child process
  * and wait for it to finish
  * @param nchilds number of child processes to create
  * @return void
*/
void test_child_processes(int nchilds) {
  printf("---Test process relationship with %d child\n\n", nchilds);
```

```
  for (int i = 0; i < nchilds; i++) {
    int pid = fork();
    if (pid < 0) {
      fprintf(2, "Error: fork failed\n");
      exit(1);
    } else if (pid == 0) {
      printf("child %d created with PID %d\n\n", i, getpid());
      printf("child %d execute pstree and die\n", i);

      char *args[] = {"pstree", 0};
      exec(args[0], args);
      fprintf(2, "exec pstree failed\n");
      exit(1);
    }

    wait(0);
  }
}
/**
  * test genealogy of processes
  * @param gen number of generations
  * @param max_gen maximum number of generations
  * @param current_gen current generation
  * @return void
*/
void test_genealogy(int gen, int max_gen, int current_gen) {
  if (current_gen >= max_gen) {
    return;
  }

  printf("---Test process genealogy on %d's'th generation\n", current_gen);

  int pid = fork();
  if (pid < 0) {
    fprintf(2, "Error: fork failed\n");
    exit(1);
  } else if (pid == 0) {
    printf("child %d created with PID %d from parent PID %d\n\n", current_gen,
getpid(), getppid());

    // If we're at the last generation, show the complete hierarchy
    if (current_gen == max_gen - 1) {
      printf("\nLet's see process hierarchy with pstree:\n");
      int ps_pid = fork();
      if (ps_pid < 0) {
        fprintf(2, "Error: fork failed\n");
        exit(1);
      } else if (ps_pid == 0) {
        char *args[] = {"pstree", 0};
        exec(args[0], args);
        fprintf(2, "exec pstree failed\n");
        exit(1);
      }
      wait(0);
```

```
    }

    test_genealogy(gen, max_gen, current_gen + 1);
    exit(0);
  }

  wait(0);
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
      fprintf(2, "Usage: phierarchy <number of childs> <number of
generation>\n");
      exit(1);
  }

  int nchilds = atoi(argv[1]);
  int ngen = atoi(argv[2]);
  if (nchilds < 1 || ngen < 1) {
      fprintf(2, "Error: number of childs and generation must be greater than
0\n");
      exit(1);
  }

  test_child_processes(nchilds);
  printf("All child must be terminated : see that with pstree\n");

  int pid = fork();
  if (pid < 0) {
    fprintf(2, "Error: fork failed\n");
    exit(1);
  } else if (pid == 0) {
    char *args[] = {"ps", 0};
    exec(args[0], args);
    fprintf(2, "exec ps failed\n");
    exit(1);
  }

  printf("\n");

  wait(0);
  test_genealogy(nchilds, ngen, 0);

  wait(0);
  exit(0);
}
```

- `Makefile`: Ajout de la commande phierarchy dans le Makefile pour compilation.

```
  $U/_phierarchy\
```

Le programme `phierarchy` permet de tester la conformité de la parentalité des processus dans le système d'exploitation XV6. Il génère une hiérarchie de processus en fonction du nombre d'enfants et du nombre de générations spécifiés en arguments. Le programme crée d'abord un certain nombre de processus enfants, puis il teste la généalogie des processus en créant des processus enfants.

Le programme affiche également la hiérarchie des processus à l'aide de la commande `pstree` pour vérifier la conformité de la parentalité. Il est important de noter que le programme utilise des appels système pour créer des processus et exécuter des commandes, ce qui est typique dans un système d'exploitation comme XV6.

3.3.1 Diagrame UML représentant la parentalité entre les processus

Parentalité processus

3.4 Cloisonnement des données entre processus

L'objectif est de tester le bon fonctionnement de XV6 pour le cloisonnement des données entre processus.

Ce programme va créer des données côté parent et vérifier le cloisonnement avec les données de l'enfant. Vérifiez aussi les valeurs des données issus du parent à partir de l'enfant.

- `user/pmem_test.c`: Implémentation du programme de test de cloisonnement des données entre processus.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
  int parent_int = 10;
  char parent_msg[50] = "parent message";

  int p[2];
  if(pipe(p) < 0) {
    fprintf(2, "pipe creation failed\n");
    exit(1);
  }

  printf("--- Test du cloisonnement de données entre processus\n\n");
  printf("Parent datas : int parent_int = %d | char *parent_msg = \"%s\"\n\n",
         parent_int, parent_msg);

  int pid = fork();
  if(pid < 0) {
    fprintf(2, "fork failed\n");
    exit(1);
  }

  if(pid == 0) { // Child process
    close(p[0]);
```

```
    printf("we are in child and we have all copies of data's parent\n");
    printf("Datas from parent:\n");
    printf("parent_int : %d\n", parent_int);
    printf("parent_msg : %s\n\n", parent_msg);

    // Modify the values
    parent_int += 10;
    printf("child add 10 to parent_int variable. It should be 20 now\n");
    printf("from child, parent_int : %d\n", parent_int);

    strcpy(parent_msg, "changed by child ;)");
    printf("child change parent_msg variable. It should be \"changed by child
;)\" now from child, parent_msg : %s\n\n", parent_msg);

    write(p[1], "done", 4);
    close(p[1]);
    exit(0);
  } else { // Parent process
    close(p[1]);

    char buf[10];
    read(p[0], buf, sizeof(buf));
    close(p[0]);

    printf("\nparent : child exited\n");
    printf("from parent : parent_int : %d\n", parent_int);
    printf("from parent : parent_msg : %s\n\n", parent_msg);

    wait(0);
  }

  return 0;
}
```

- `Makefile`: Ajout de la commande pmem_test dans le Makefile pour compilation.

```
  $U/_pmem_test\
```

Le programme `pmem_test` teste le cloisonnement des données entre processus dans le système d'exploitation XV6. Il crée un processus parent qui initialise des données, puis crée un processus enfant qui tente de modifier ces données. Le programme vérifie si les modifications apportées par l'enfant affectent les données du parent, ce qui permet de tester le cloisonnement des données entre les deux processus.

3.4.1 Diagrame UML représentant le cloisonnement des données entre les processus

Cloisonnement des données entre processus

8. Annexes et Références

8.1 Lancer le projet.

- Installer vscode (https://code.visualstudio.com/)
- Installer l'extention pack "Dev containers"
- Installer docker
- Ouvrer VSCode dans le dossier du projet
- Cliquer sur la notification: "Ouvrir le project dans dev container"
- Lancer le projet: `make qemu`

Conclusion

Au travers de ces travaux pratiques sur XV6, nous avons pu explorer et implémenter plusieurs concepts fondamentaux des systèmes d'exploitation. En développant des commandes comme `pstree`, `psync_tst`, `phierarchy_tst` et `pmem_test`, nous avons mis en œuvre des fonctionnalités liées à la gestion des processus, à leur synchronisation, à la parentalité, ainsi qu'au cloisonnement mémoire.

Ces exercices nous ont permis de mieux comprendre le comportement des processus dans un environnement système, notamment la façon dont ils interagissent entre eux, leur hiérarchie et leur indépendance en mémoire. En manipulant directement le code source du noyau de XV6, nous avons également pris conscience des enjeux liés à la conception d'un système d'exploitation, et de la complexité sous-jacente à des fonctionnalités qui paraissent simples à l'utilisateur final.

Ce projet fut donc une excellente opportunité de mettre en pratique nos connaissances théoriques, tout en nous familiarisant avec les mécanismes internes d’un OS minimaliste mais complet. Ces acquis constituent une base solide pour aborder des systèmes plus complexes dans le futur.