Université de Yaoundé 1

Master 1 – Option : Systèmes et Réseaux
INF 4097 : Conception des Systèmes d'Exploitation

Projet XV6 : Conceptionn d'un système d'exploitation

Extensions du noyau xv6-riscv
Gestion de la mémoire et ordonnancement

LANGOUL FADILA MARIAMA MOUNIRA 21T2528

Sous la supervision de Dr Adamou Hamza
Année académique 2025-2026

Table des matières

Option 1 : Appel système `getvmstats`

0.1 Spécification

Par défaut, xv6 ne fournit aucun mécanisme permettant à un programme utilisateur d'observer l'état de la mémoire du système ou de son propre processus.

L'objectif de l'appel système `getvmstats()` est de fournir une interface simple permettant à un processus utilisateur de récupérer des statistiques mémoire calculées dans le noyau, sans modifier la politique de gestion mémoire existante.

Les informations retournées sont :

— le nombre de pages physiques allouées ;
— le nombre de pages physiques libres ;
— le nombre de défauts de page du processus courant ;
— la taille du heap utilisateur ;
— une métrique personnelle.

L'appel système copie ces informations depuis le noyau vers l'espace utilisateur à l'aide d'une structure dédiée.

0.2 Modifications apportées au noyau

0.3 Structure `vmstats` (kernel/vmstats.h)

```
struct vmstats {
  int pages_allocated;
  int pages_free;
  int page_faults;
  uint64 heap_size;
  int mymetric;
};
```

0.4 Comptage des pages physiques (kernel/kalloc.c)

```
int freepages;
int allocatedpages;
```

Ces compteurs sont mis à jour lors de chaque allocation et libération de page physique.

0.5 Implémentation noyau (kernel/vm.c)

```
void getvmstats(struct vmstats *stats) {
  struct proc *p = myproc();
  stats->pages_allocated = allocatedpages;
  stats->pages_free = freepages;
  stats->page_faults = p->page_faults;
  stats->heap_size = p->sz;
  stats->mymetric = allocatedpages % 10;
}
```

0.6 Appel système (kernel/sysproc.c)

```
uint64 sys_getvmstats(void) {
  uint64 addr;
  argaddr(0, &addr);
  struct vmstats stats;
  getvmstats(&stats);
  if(copyout(myproc()->pagetable, addr,
      (char *)&stats, sizeof(stats)) < 0)
    return -1;
  return 0;
}
```

0.7 Programme utilisateur de test

Le programme utilisateur appelle `getvmstats()` avant et après plusieurs appels à `sbrk` afin d'observer l'évolution des statistiques mémoire.

```
struct vmstats s;
getvmstats(&s);
printf("Pages␣allouees␣:␣%d\n", s.pages_allocated);
```

0.8 Captures et logs réels

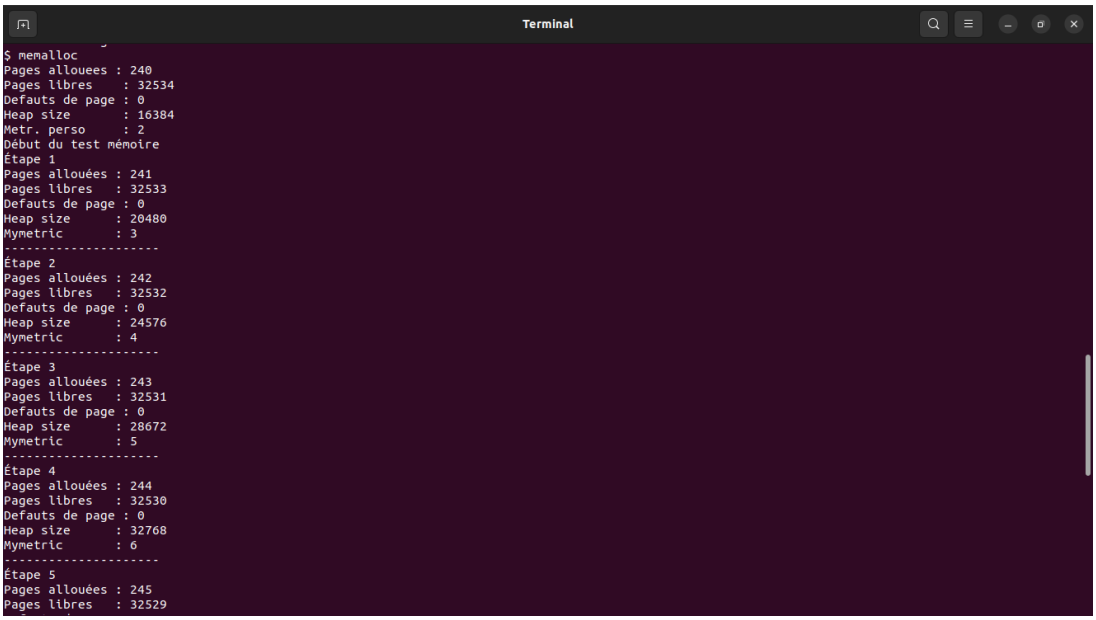

FIGURE 1 – Statistiques mémoire avant et après plusieurs allocations sbrk()

Chapitre 1

Option 2 : Lottery Scheduler

1.1 Spécification

L'ordonnanceur par défaut de xv6 repose sur une politique de type round-robin, dans laquelle tous les processus ont la même priorité.

Le Lottery Scheduler est un ordonnanceur probabiliste où chaque processus possède un nombre de tickets. À chaque décision d'ordonnancement, un tirage aléatoire désigne le processus gagnant.

La probabilité d'être sélectionné est proportionnelle au nombre de tickets détenus, ce qui permet de contrôler la part CPU attribuée à chaque processus.

1.2 Modifications apportées au noyau

1.3 Extension de `struct proc`

```
int tickets;
uint64 runtime;
```

1.4 Appel système `settickets`

```
uint64 sys_settickets(void) {
  int n;
  argint(0, &n);
  if(n < 1) n = 1;
  myproc()->tickets = n;
  return 0;
}
```

1.5 Nouvel algorithme du scheduler

Le scheduler calcule la somme des tickets des processus `RUNNABLE`, effectue un tirage aléatoire, puis sélectionne le processus gagnant.

À chaque sélection, le compteur `runtime` du processus est incrémenté.

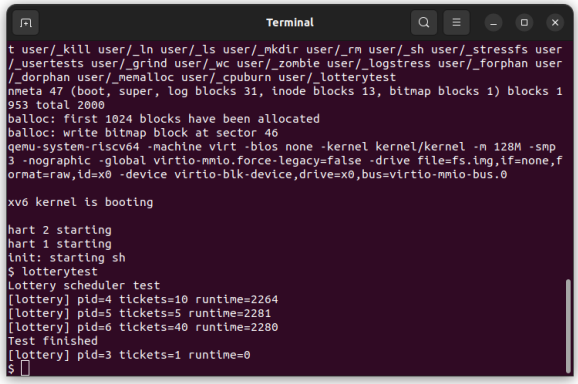

FIGURE 1.1 – Sortie de lotterytest montrant la répartition CPU

1.6 Programme utilisateur de test

Le programme `lotterytest` crée trois processus CPU-bound, leur attribue des nombres de tickets différents et affiche le nombre de fois où chacun est sélectionné.

```
int main(void){
  int pids[3];
  int tickets[3] = {10, 5, 40};
  printf("Lottery scheduler test\n");
  for(int i = 0; i < 3; i++){
    pids[i] = fork();
    if(pids[i] == 0){
      run(tickets[i]);
    }
  }
    // le parent attend les enfants
  for(int i = 0; i < 3; i++){
    wait(0);
  }
  printf("Test finished\n");
  exit(0);
}
```

1.6.1 Rôle de `cpuburn`

`cpuburn` est une boucle infinie de calcul permettant de maintenir le processus en état `RUNNABLE` pendant 30s, afin d'exposer uniquement le comportement de l'ordonnanceur.

```
int main(void){
  uint64 start = uptime();
  volatile uint64 i = 0;
  while(uptime() - start < 3000){ // ~30 secondes
    i++;                          // calcul pur
  }
  exit(0);
}
```

1.7 Captures et logs réels

Chapitre 2

Option 3 : Lazy Allocation

2.1 Spécification

Dans xv6, l'appel système `sbrk(n)` alloue immédiatement les pages physiques correspondant à l'extension demandée.

La Lazy Allocation consiste à différer l'allocation physique des pages jusqu'au premier accès mémoire, qui déclenche alors un défaut de page.

Cette approche permet d'optimiser l'utilisation de la mémoire physique et de rendre observable le mécanisme de défaut de page.

2.2 Modifications apportées au noyau

2.2.1 Modification de `sys_sbrk`

```
myproc()->sz += n;
```

2.2.2 Gestion du défaut de page

Lorsqu'un accès à une page non mappée se produit, le noyau :
— alloue une page physique ;
— initialise la page à zéro ;
— met à jour la table de pages ;
— incrémente le compteur de défauts de page.

2.3 Programme utilisateur de test

Le programme de test appelle `sbrk(PGSIZE)` puis accède à la page afin de provoquer explicitement un défaut de page.

2.4 Captures et logs réels

2.5 Conclusion

Ces trois extensions illustrent des mécanismes fondamentaux des systèmes d'exploitation modernes : observation du noyau, ordonnancement probabiliste et gestion différée de la mémoire.

Elles transforment xv6 en un environnement expérimental permettant d'analyser concrètement les décisions prises par le noyau.

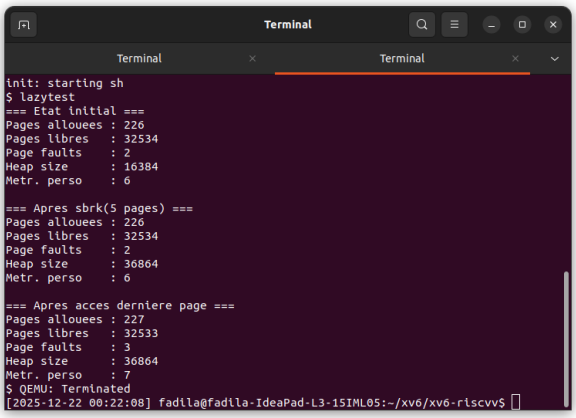

FIGURE 2.1 – défaut de page observé