به نام خدا

گزارش کار پروژه پنجم آزمایشگاه سیستم عامل

"مدیریت حافظه در xv6"

گروه ۵ :

سینا طبسی۸۱۰۱۹۹۵۵۴ سید حامد میرامیرخانی ۸۱۰۱۹۹۵۰۰ فاطمه محمدی۸۱۰۱۹۹۴۸۹

Repository: https://github.com/HamedMiramirkhani/OS_Lab_CA5

سوالات

۱. **راجع به مفهوم ناحیه مجازی (VMA) در لینوکس به طور مختصر توضیح داده و آن را با xv6 مقایسه کنید.**

پاسخ:

در لینوکس هسته از نواحی virtual memory با پیگیری memory mapping های پردازه استفاده میکند. مثلا یک پردازه یک VMA برای کد، یک VMA برای هر نوع دیتا، یک VMA برای هر Memory mapping دارد. هر VMA شامل تعدادی page میباشد که هر کدام از این page ها یک entry به page table دارد. اما xv6 از آدرس های مجازی ۳۲ بیتی استفاده میکند که فضای آدرسی مجازی ۴ گیگابایتی ایجاد میکند. همینطور xv6 از جدول دوسطحی استفاده میکند و مفهومی از حافظه مجازی ندارد.

۲. **چرا ساختار سلسله مراتبی منجر به کاهش مصرف حافظه میگردد؟**

پاسخ:

مورد اول اینکه ما فقط بخشی از صفحه پردازه که پردازه میخواهد از اطلاعات آن استفاده کند را میخوانیم و قسمت هایی از آن که استفاده نمیکند را بیهوده لود نمیکنیم.

مورد دوم اینکه ما فقط آدرس شروع صفحه هر پردازه را نگه میداریم و قسمتی از صفحه پردازه را (که دوباره صفحه بندی کردیم) با یک offset انتخاب میکنیم و اینگونه مقدار اعداد ذخیره شده هم کمتر میشوند. بطور مثال فرض کنید یک حافظه ۳۲ بیتی با صفحه های ۴ کیلوبایت داریم، در این صورت باید 2^{20} سطر برای آدرس ابتدایی صفحات داشته باشیم در صورتی که میتوانیم این آدرسها را نیز به صورت ۱۰۲۴ صفحه، صفحه بندی کنیم که در این صورت به $2^{12} + 2^{10}$ فضای حافظه نیاز داریم برای آدرس دهی صفحات.

۳. **محتوای هر بیت یک مدخل (۳۲ بیتی) در هر سطح چیست؟ چه تفاوتی میان آنها وجود دارد؟**

پاسخ:

در هر دو سطح ۱۲ بیت برای سطح دسترسی نگهداری میشود.۲۰ بیت باقی میماند که در سطح table page این ۲۰ بیت برای آدرس صفحه فیزیکی استفاده میشود در حالی که در مدخل سطح directory page برای اشاره به سطح بعدی از آن استفاده میشود.

در بیت D یعنی بیت dirty با هم تفاوت دارند. در directory page این بیت به این معنا است که صفحه باید در دیسک نوشته شود تا تغییرات اعمال شود اما در table page این بیت معنایی ندارد.

۴. **تابع ()kalloc چه نوع حافظه ای تخصیص میدهد؟ (فیزیکی یا مجازی)**

پاسخ:

یک صفحه ۴۰۹۶ بایتی از حافظه فیزیکی را اختصاص می دهد. این تابع اشاره گر را برمیگرداند که هسته می تواند از آن استفاده کند. اگر حافظه قابل تخصیص نباشد ۰ برمی گرداند.

۵. **تابع ()mappages چه کاربردی دارد؟**

پاسخ:

از این تابع برای اتصال حافظه مجازی به حافظه فیزیکی و اضافه کردن صفحه جدید به pgdir استفاده میشود.

PTE هایی برای آدرس های مجازی که از va شروع می شوند ایجاد می کند که به آدرس های فیزیکی با شروع paاشاره می کنند va و اندازه ممکن است با صفحه تراز نباشند.

۶. **راجع به تابع ()walkpgdir توضیح دهید. این تابع چه عمل سخت افزاری را شبیه سازی میکند؟**

پاسخ:

این تابع آدرس PTE را در جدول صفحه pgdir که با آدرس مجازی va مطابقت دارد، برمی گرداند. اگر $alloc\ != 0$ هر صفحه جدول صفحه مورد نیاز را ایجاد می کند.

به طور خلاصه ()walkpgdir شبیه سازی عمل سخت افزاری ترجمه آدرس مجازی به فیزیکی را انجام می دهد.

۷. **توابع allocuvm و mappages که در ارتباط با حافظه ی مجازی هستند را توضیح دهید.**

پاسخ:

تابع allocuvm در سیستم عامل xv6 برای اختصاص دادن حافظه مجازی به یک process استفاده می‌شود. این تابع محدوده‌ی حافظه‌ای را که process نیاز دارد، به صورت پیوسته از آدرس شروع تا آدرس پایان مشخص می‌کند و فضای آن را برای process رزرو می‌کند.

تابع mappages هم همانطور که در سوال ۵ گفته شد برای نگاشت صفحات حافظه‌ی مجازی به فضای حافظه‌ی فیزیکی استفاده می‌شود. این تابع با دریافت آدرس صفحه‌ی مجازی، آدرس فیزیکی متناظر با آن را در دسترس قرار می‌دهد. به این ترتیب، process می‌تواند از حافظه‌ی مجازی خود استفاده کند و سیستم عامل با توجه به نیاز process، صفحات مجازی را به صفحات فیزیکی نگاشت می‌کند.

۸. **شیوه ی بارگذاری برنامه در حافظه توسط فراخوانی سیستمی exec را شرح دهید.**

پاسخ:

این فراخوانی سیستمی با دریافت مسیر فایل مورد نظر و پارامترهای مربوط به آن فرآیند جاری را با برنامه جدید جایگزین می کند. برای انجام این کار، سیستم ابتدا فایل مورد نظر را از دیسک بارگذاری کرده و سپس ساختارهای لازم برای آن برنامه (مانند ساختارهای پردازش، حافظه و ...) را در حافظه جدید فرآیند ایجاد می کند. در ادامه، این فرآیند جدید شروع به اجرای برنامه خود می کند.

شرح پروژه

بررسی ساختار اولیه حافظه xv6

درفایل exec.c دو اشاره گر مهم وجود دارد، یکی sz که انتهای حافظه ی مجازی را نشان میدهد و دیگری sp که اشاره گر stack است.

```
41    // Load program into memory.
42    sz = 0;
43    for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
44      if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
45        goto bad;
46      if(ph.type != ELF_PROG_LOAD)
47        continue;
48      if(ph.memsz < ph.filesz)
49        goto bad;
50      if(ph.vaddr + ph.memsz < ph.vaddr)
51        goto bad;
52      if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
53        goto bad;
54      if(ph.vaddr % PGSIZE != 0)
55        goto bad;
56      if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
57        goto bad;
```

حلقه for به تعداد تمام بخش های برنامه پیمایش میشود و هر بار در خط ۵۲ مقدار sz آپدیت میشود

آرگومان سوم تابع **allocuvm** مقدار جدید **sz** است که از حاصل جمع **ph.vaddr** با **ph.memsz** بدست می آید.

عبارت **"ph"** مخفف **"program header"** است و به بخشی هدر فایل اجرایی اشاره دارد که در آن اطلاعات مربوط به بخش‌های مختلف برنامه، مانند بخش‌های اجرایی، داده‌ها و غیره قرار میگیرند.

"ph.vaddr" یک فیلد در جدول برنامه های اجرایی (executable) در حافظه است که نشان دهنده‌ی آدرس حافظه‌ای است که بخش مشخص شده توسط ساختار **"ph"** در آن قرار دارد.**"ph.memsz"** نشان دهنده‌ی اندازه حافظه (به بایت) است که برای بخش مشخص شده توسط ساختار **"proghdr"** در حافظه‌ی سیستم در زمان اجرا اختصاص داده شده است.

```
24  // Program section header
25  struct proghdr {
26    uint type;
27    uint off;
28    uint vaddr;
29    uint paddr;
30    uint filesz;
31    uint memsz;
32    uint flags;
33    uint align;
34  };
```

پس از لود برنامه در مموری دو صفحه ساخته میشود یکی صفحه محافظ و دیگری صفحه ی stack که این دو صفحه اندازه ثابتی دارند. در ادامه اشاره گر دوم یعنی sz که همان stack pointer خواهد بود مقدار میگیرد.

```
63    // Allocate two pages at the next page boundary.
64    // Make the first inaccessible.
65    // Use the second as the user stack.
66    sz = PGROUNDUP(sz);
67    if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
68      goto bad;
69    clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
70    sp = sz;
```

```
71
72  // Push argument strings, prepare rest of stack in ustack.
73  for(argc = 0; argv[argc]; argc++) {
74    if(argc >= MAXARG)
75      goto bad;
76    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
77    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
78      goto bad;
79    ustack[3+argc] = sp;
80  }
81  ustack[3+argc] = 0;
82
83  ustack[0] = 0xffffffff;  // fake return PC
84  ustack[1] = argc;
85  ustack[2] = sp - (argc+1)*4;  // argv pointer
86
87  sp -= (3+argc+1) * 4;
88  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
89    goto bad;
90
```

با قرار گرفتن آرگومان های برنامه
در حافظه اشاره گر استک یا
همان sp آپدیت میشود.

منابع این بخش : منبع ۱ و منبع ۲

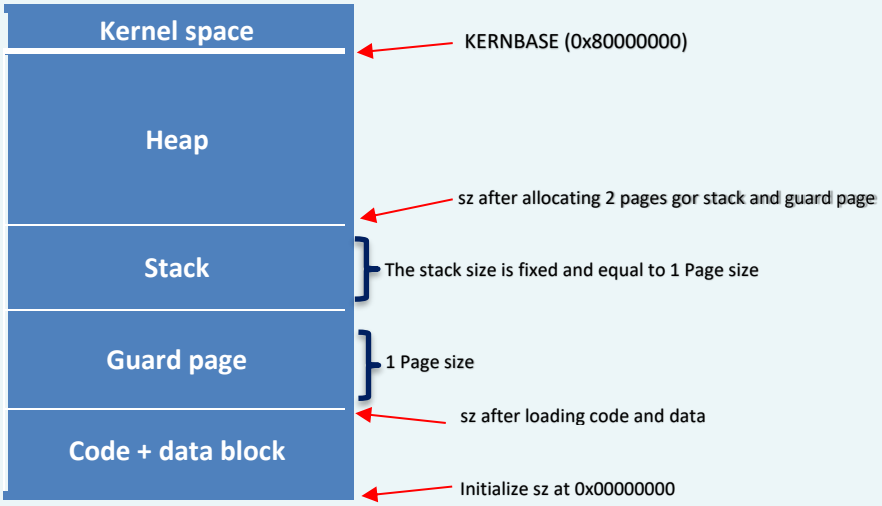
چه تغییری باید در این ساختار حافظه ایجاد کنیم؟

همانگونه که در بالا اشاره شد فضای stack در حافظه ثابت است. وظیفه ی ما این است که با تغییر در ساختار کد xv6 فضای stack را داینامیک کنیم تا در صورت نیاز stack رشد کند. اما چگونه این کار را انجام دهیم؟

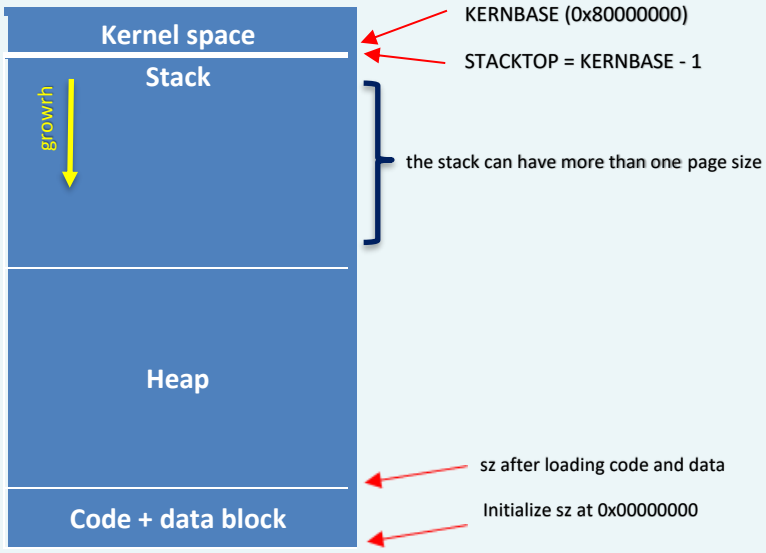
اگر دقت کرده باشید متوجه میشوید برای تعیین فضای هیپ اشاره گری نداریم و در حقیقت با تخصیص فضاهای دیگر حافظه، مقدار آن مشخص میشود. پس نمی توان محل دقیق heap را مشخص کنیم. حال برای رسیدن به هدفمان جای stack و heap را عوض میکنیم یعنی stack را به سمت فضای هسته منتقل میکنیم. مقدار sz در نقطه ای قرار میگیرد که code و data بارگذاری میشوند و sp به top.stack اشاره میکند.

در صفحه ی بعد حافظه برنامه در دو حالت به تصویر کشیده شده است.

حالت اول



حالت دوم



برای اعمال این تغییر در ساختار حافظه، نیاز است در کد xv6 تغییراتی ایجاد کنیم. در ادامه این تغییرات برای هر فایل بیان میشود.

a) proc.h

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  uint st_sz;                  // Size of stack // [ME]
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int status;                  // Exit status
  int stackPages;              // Num of stack pages
};
```

b) memlayout.h

```
// Memory layout

#define EXTMEM  0x100000            // Start of extended memory
#define PHYSTOP 0xE000000           // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
#define STACKTOP KERNBASE-1         // For realocate stack

#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) ((void *)(((char *) (a)) + KERNBASE))

#define V2P_WO(x) ((x) - KERNBASE)    // same as V2P, but without casts
#define P2V_WO(x) ((x) + KERNBASE)    // same as P2V, but without casts
```

c) exec.c

```
  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.
  // Use the second as the user stack.

  /* COMMENT following
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
  sp = sz;
  */

  uint stack_top = STACKTOP;
  if((stack_top =
      allocuvm(pgdir, stack_top-PGSIZE, stack_top)) == 0)
     goto bad;
  sp = stack_top;
```

زمانی که **data** و **code** داخل حافظه لود شدند و اشاره گر **sz** به مقدار جدید آپدیت شد، از **STACKTOP-PGSIZE** تا **STACKTOP** حافظه را برای **stack** جدید الوکیت میکنیم. سپس **STACKTOP** را به اشاره گر **sp** اساین میکنیم.

```
  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  curproc->st_sz = st_sz;

  curproc->stackPages = 1;
  cprintf("Initial number of pages by the process: %d\n", curproc-
>stackPages);
```

برای تعیین سایز stack

لاگ برای تعیین تعداد صفحات stack حین‌ران شدن برنامه

d) vm.c

هنگامی که یک پردازه fork میشود فرزند پردازه توسط تابع copyuvm() ساخته میشود و باید code و data و همینطور stack پردازه پدر در پردازه فرزند کپی شود. در حالت اول یعنی کد اصلی xv6 داخل تابع copyuvm() یک حلقه وجود داشت که بلاک code و data و همینطور stack پردازه پدر توسط آن کپی میشد اما در حالت جدید نیاز است یک حلقه دیگر اضافه شود چرا که حلقه اول در این شرایط تنها بلاک code و data را کپی میکند و باید توسط حلقه دوم stack پدر را در فرزند پردازه کپی کنیم.
همانطور که مشاهده میکنید تنها تفاوت حلقه اول و دوم فضایی است که کپی میکنند و بقیه دستورات کاملا مشابه اند.

```
if((d = setupkvm()) == 0)
  return 0;
for(i = 0; i < sz; i += PGSIZE){
  if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
    panic("copyuvm: pte should exist");
  if(!(*pte & PTE_P))
    panic("copyuvm: page not present");
  pa = PTE_ADDR(*pte);
  flags = PTE_FLAGS(*pte);
  if((mem = kalloc()) == 0)
    goto bad;
  memmove(mem, (char*)P2V(pa), PGSIZE);
  if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
    kfree(mem);
    goto bad;
  }
}
//Since stack is now moved to below KERNBASE, we need to copy all the
//pages of stack starting from KERNBASE - 1 upto the point where all the
//stack pages end. We keep the above loop as it is because we also
//wanted to copy Code+Data block in the new process.
for(i = 1; i <= st_sz; i++){
  if((pte = walkpgdir(pgdir, (void *)(STACKTOP - PGSIZE*i + 1) , 0)) == 0)
    panic("copyuvm: pte should exist");
  if(!(*pte & PTE_P))
    panic("copyuvm: page not present");
  pa = PTE_ADDR(*pte);
  flags = PTE_FLAGS(*pte);
  if((mem = kalloc()) == 0)
    goto bad;
  memmove(mem, (char*)P2V(pa), PGSIZE);
  if(mappages(d, (void*)(STACKTOP - PGSIZE*i + 1), PGSIZE, V2P(mem), flags) < 0) {
    kfree(mem);
    goto bad;
  }
}
```

حلقه for دوم به منظور کپی کردن stack پردزه پدر اضافه شد.

e) syscall.c

از آنجایی که محل stack تغییر پیدا کرد پس بالاترین نقطه stack همواره مقداری ثابت و برابر **STACKTOP** خواهد بود.
بنابراین در توابع fetchint و fetchstr و argptr باید تغییرات را اعمال کنیم.

```
-  if(addr >= curproc->sz || addr+4 > curproc->sz)
+  if(addr >= STACKTOP || addr+4 > STACKTOP)
```

```
-   if(addr >= curproc->sz)
+   if(addr >= STACKTOP)
```

```
-   if(size < 0 || (uint)i >= curproc->sz ||
  (uint)i+size > curproc->sz)
+   if(size < 0 || (uint)i >= STACKTOP ||
  (uint)i+size > STACKTOP)
```

f) proc.c

از آنجا که متغیر stackPages را به استراکت proc اضافه کردیم، در تابع fork() این مقدار را بعنوان یکی از فیلد های پردازه باید در نظر داشته باشیم.

```
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  np->stackPages = curproc->stackPages;
```

g) trap.c

رشد فضای حافظه

برای تحقق هدف دوم یعنی هندل کردن page fault کافی است یک case به تابع trap داخل فایل trap.c اضافه کنیم. rcr2() مقدار ذخیره شده در رجیستر CR2 (Control Register) را بازمی‌گرداند بنابراین با فراخوانی rcr2() آدرس خطای صفحه (**offendingAddr**) را دریافت می‌کنیم.

trap handler بررسی می‌کند که آیا خطای صفحه به دلیل دسترسی به فضای زیر stacktop فعلی رخ داده است یا نه. اگر چنین باشد یک صفحه allocate و map می‌کنیم و ۱ واحد به stackPage پردازه کنونی اضافه میکنیم. اما اگر page fault به دلیل آدرس دیگری رخ داده باشد به default handler و همانند xv6 یک kernel panic میکنیم.

```
//Added following case for handle page fault
case T_PGFLT:
  ; //Inserting empty statement because labels can only be followed by
  //statements, and declarations do not count as statements in C.
  //We are declaring offendingAddr below and hence in the absence of
  //empty statement, an error is thrown.
  uint offendingAddr = PGROUNDDOWN(rcr2());
  uint stackTop = STACKTOP - (myproc()->stackPages * PGSIZE);
  if(offendingAddr <= stackTop && offendingAddr >= (stackTop - PGSIZE)) {
    if(allocuvm(myproc()->pgdir, offendingAddr, stackTop) == 0) {
      cprintf("case T_PGFLT from trap.c: allocuvm failed. Num of current allocated pages: %d\n"\
      , myproc()->stackPages);
      exit(1);
    }
    //Successful allocuvm() while page handling, hence incrementing number
    //of stack pages.
    myproc()->stackPages += 1;
    cprintf("case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: %d\n", \
    myproc()->stackPages);
    break; //'break' inside 'if' because in the 'else' part, we want to handle the page fault using
    //default handler.
  }
```

```
default:
  if(myproc() == 0 || (tf->cs&3) == 0){
    // In kernel, it must be our mistake.
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",
            tf->trapno, cpuid(), tf->eip, rcr2());
    panic("trap");
  }
  // In user space, assume process misbehaved.
  cprintf("pid %d %s: trap %d err %d on cpu %d "
          "eip 0x%x addr 0x%x--kill proc\n",
          myproc()->pid, myproc()->name, tf->trapno,
          tf->err, cpuid(), tf->eip, rcr2());
```

h) test.c

در نهایت برای تست درستی عملکرد برنامه در شرایط جدید یک فایل جدید به نام test.c ایجاد میکنیم.

```c
#include "types.h"
#include "user.h"
// Prevent this function from being optimized, which might give it closed form
#pragma GCC push_options
#pragma GCC optimize ("O0")
static int
recurse(int n)
{
  if(n == 0)
    return 0;
  return n + recurse(n - 1);
}
#pragma GCC pop_options
int
main(int argc, char *argv[])
{
  int n, m;

  if(argc != 2){
    printf(1, "Usage: %s levels\n", argv[0]);
    exit(1);
  }
  n = atoi(argv[1]);
  printf(1, "--------------------------\n");
  printf(1, "Recursing %d levels\n", n);
  m = recurse(n);
  printf(1, "result = %d\n", m);
  printf(1, "--------------------------\n\n");
  exit(1);
}
```

همانطور که مشخص است این سیستم کال عدد n را به عنوان آرگومان دریافت میکند و مجموع اعداد ۱ تا n را بصورت بازگشتی محاسبه میکند.

i) Makefile

```
EXTRA=\
    test.c mkfs.c ulib.c user.h
cat.c echo.c forktest.c grep.c
kill.c\
    ln.c ls.c mkdir.c rm.c
stressfs.c usertests.c wc.c
zombie.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.*
```

```
UPROGS=\
    _test\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
```

صحت عملکرد xv6 در شرایط جدید

```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Initial number of pages by the process: 1
Group #5 Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
Initial number of pages by the process: 1
$ test 100
Initial number of pages by the process: 1
--------------------------
Recursing 100 levels
result = 5050
--------------------------

$ test 1000
Initial number of pages by the process: 1
--------------------------
Recursing 1000 levels
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 2
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 3
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 4
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 5
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 6
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 7
case T_PGFLT from trap.c: allocuvm succeeded. Num of pages allocated: 8
result = 500500
--------------------------

$
```

مطابق تصویر بالا اجرای برنامه را برای دو مقدار ۱۰۰ و ۱۰۰۰ مشاهده میکنید.

ابتدای برنامه برای stack تنها یک page الوکیت میشود، این فضا برای محاسبه مجموع اعداد 1 تا 100 کافی است اما هنگام اجرای test برای عدد 1000 دچار page fault میشویم چرا که عمق stack از حد کافی بیشتر میشود. در این صورت برای هندل کردن page fault باید یک page جدید برای stack الوکیت کنیم. در نهایت مشاهده میکنیم نتیجه ی نهایی به درستی محاسبه شده است.