Chapter 1: Operating System Organization

— Oleh Kelompok 7

Anggota

1. Dewi Sugianti (1313619010)
2. Muhammad Anindyo Poetra Mufatyta (1313619004)
3. Putu Sanisa Pascaline (1313619023)
4. Savitri (1313619015)

1. Abstracting Physical Resources

System call	Description
fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return the current process's pid
sleep(n)	Sleep for n clock ticks
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Figure 0-2. Xv6 system calls

Pendekatan library memiliki kelemahan, yaitu aplikasi harus secara berkala melepas prosesor agar aplikasi lain dapat berjalan. Namun, hal tersebut bisa berjalan jika semua aplikasi saling percaya dan tidak ada bug. Maka, dibutuhkan isolasi yang lebih kuat.

Isolasi yang kuat melarang aplikasi mengakses langsung ke hardware resources, untuk itu dibutuhkan abstraksi resources.

Misalnya, aplikasi hanya berinteraksi dengan sistem file melalui system call buka, baca, tulis, dan tutup.

2. User Mode, Kernel Mode, and System Calls

Isolasi yang kuat butuh batasan yang tegas antara aplikasi dan sistem operasi. Untuk itu, prosesor mendukung hal tersebut dengan memiliki dua mode, yaitu mode kernel dan mode pengguna.

Di mode kernel, prosesor diizinkan menjalankan hak istimewanya. Sedangkan, jika aplikasi dalam mode pengguna mencoba mengeksekusi instruksi istimewa, maka prosesor tidak akan mengeksekusinya dan membersihkan aplikasi tersebut.

Prosesor menyediakan instruksi untuk pengalihan dari mode kernel ke mode pengguna. Misalnya, pada x86 menggunakan instruksi int.

Setelah prosesor beralih ke mode kernel. Kernel akan memvalidasi argumen system call untuk menjalankan atau menolak operasi yang diminta.

Perlu diketahui bahwa kernel yang menetapkan titik masuk ke mode kernel karena jika aplikasi yang melakukannya, aplikasi yang berbahaya dapat memasuki kernel pada saat validasi argumen dilewati.

3. Kernel Organization

Kernel Organization

- Seluruh sistem operasi berada di kernel, sehingga implementasi dari semua system call dijalankan dalam mode kernel, sehingga organisasi ini disebut kernel monolitik.
  - Kelebihan: Kernel organization ini nyaman karena perancang sistem operasi tidak harus memutuskan yang mana bagian dari sistem operasi tidak memerlukan hak istimewa perangkat keras penuh. Selain itu, mudah agar bagian berbeda dari sistem operasi dapat bekerja sama
  - Kelemahan: antarmuka antara berbeda bagian dari sistem operasi seringkali rumit,
- Untuk mengurangi risiko kesalahan pada kernel, perancang sistem operasi dapat meminimalkan file jumlah kode sistem operasi yang berjalan dalam mode kernel, dan menjalankan sebagian besar sistem operasi dalam mode pengguna. Kernel organization ini disebut kernel mikro.

Micro Kernel

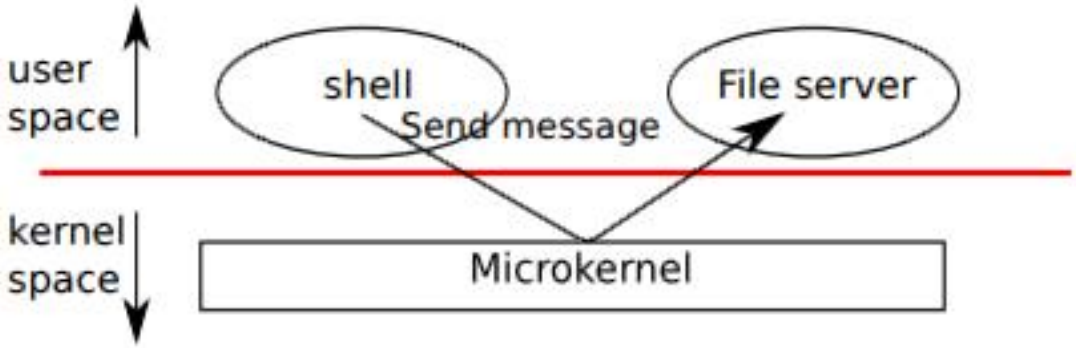

Figure 1-1. A mkernel with a file system server

Micro Kernel (Lanjutan)

- Pada gambar sistem file berjalan sebagai proses tingkat pengguna (user-level process).
- Layanan OS yang berjalan sebagai proses disebut server.
- Untuk memungkinkan aplikasi berinteraksi dengan file server, kernel menyediakan mekanisme komunikasi antar proses untuk mengirim pesan dari satu proses mode pengguna ke proses lainnya.
- Untuk misalnya, jika aplikasi seperti shell ingin membaca atau menulis file, aplikasi akan mengirim pesan ke server file dan menunggu respons.
- Dalam mikrokernel, antarmuka kernel terdiri dari beberapa fungsi tingkat rendah untuk memulai aplikasi, mengirim pesan, mengakses perangkat keras perangkat, dll. Struktur organisasi ini membuat kernel relatif sederhana, karena sebagian besar sistem operasi terletak di server tingkat pengguna.

Real World

- Di dunia nyata, kita dapat menemukan kernel monolitik dan mikro. Misalnya, Linux memiliki kernel monolitik, meskipun beberapa fungsi OS berjalan sebagai tingkat pengguna server (mis., sistem windowing). Xv6 diimplementasikan sebagai kernel monolitik, mengikuti kebanyakan sistem operasi Unix. Jadi, di xv6, antarmuka kernel sesuai dengan antarmuka sistem operasi, dan kernel menerapkan sistem operasi lengkap. Karena xv6 tidak menyediakan banyak layanan, kernelnya lebih kecil dari beberapa mikrokernel.

4. Process Overview

Process Overview

- Unit Isolasi di xv6 adalah sebuah proses
- Kernel harus melaksanakan Proses Abstraksi dengan hati-hati yang mencegah satu proses merusak memori proses lain, cpu, dan merusak kernel itu sendiri sehingga suatu proses tidak dapat menumbangkan mekanisme isolasi kernel.
- Mekanisme yang digunakan oleh kernel untuk mengimplementasikan proses ini mencakup flag user / kernel, ruang alamat, dan pemotongan waktu threads.
- Untuk membantu menegakkan isolasi, proses abstraksi memberikan ilusi ke program yang memiliki mesin pribadinya sendiri. Suatu proses menyediakan program dengan apa yang tampak seperti sistem memori privat, atau ruang alamat, yang tidak dapat dibaca atau ditulis oleh proses lain.

Process Overview

- xv6 menggunakan page tables untuk memberikan setiap proses ruang alamatnya sendiri. Page table x86 menerjemahkan virtual address ke physical address.
- xv6 mempertahankan page table terpisah untuk setiap proses yang menentukan proses itu adalah address space. Seperti yang diilustrasikan pada Gambar 1-2, address space mencakup memori pengguna dari proses memulai pada nol virtual address. Instruksi datang lebih dulu, diikuti oleh variabel global, lalu stack, dan akhirnya area "heap" yang prosesnya dapat berkembang sesuai kebutuhan.

Setiap address space berproses memetakan instruksi dan data kernel serta memori milik program pengguna. Ketika suatu proses memanggil system call, system call dijalankan di pemetaan kernel dari proses address space. Pengaturan ini ada supaya kode system call kernel dapat langsung merujuk ke memori pengguna. Untuk menyisakan ruang yang banyak bagi memori user, address space xv6 memetakan kernel ke address tinggi, mulai dari 0x80100000.

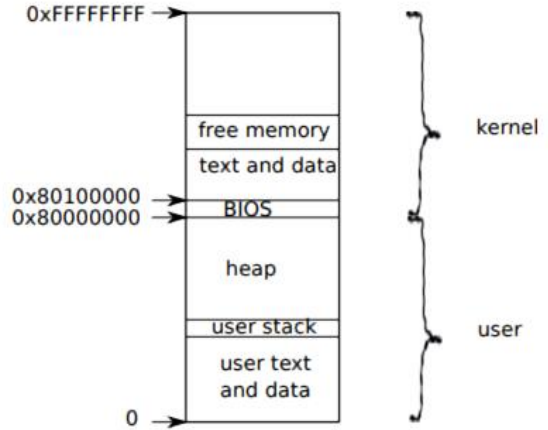

Figure 1-2. Layout of a virtual address space

Kernel xv6 mempertahankan banyak bagian untuk setiap proses, yang dikumpulkannya menjadi struct proc (2337):

Potongan kernel state terpenting milik sebuah proses adalah page table, kernel stack, dan run state miliknya. Di sini, notasi “p->xxx” digunkan untuk merujuk ke elemen dari proc structure.

```
2336 // Per-process state
2337 struct proc {
2338   uint sz;                     // Size of process memory (bytes)
2339   pde_t* pgdir;                // Page table
2340   char *kstack;                // Bottom of kernel stack for this process
2341   enum procstate state;        // Process state
2342   int pid;                     // Process ID
2343   struct proc *parent;         // Parent process
2344   struct trapframe *tf;        // Trap frame for current syscall
2345   struct context *context;     // swtch() here to run process
2346   void *chan;                  // If non-zero, sleeping on chan
2347   int killed;                  // If non-zero, have been killed
2348   struct file *ofile[NOFILE];  // Open files
2349   struct inode *cwd;           // Current directory
```

- Setiap proses memiliki thread eksekusi yang mengeksekusi file instruksi proses.
- Sebuah thread dapat ditangguhkan dan kemudian dilanjutkan.
- Untuk beralih secara transparan di antara proses, kernel menangguhkan thread yang sedang berjalan dan menghitung ulang thread proses lain.
- Sebagian besar status thread disimpan di stack milik thread.
- Setiap proses memiliki dua stacks: stack pengguna dan stack kernel.
- Ketika proses mengeksekusi pengguna dalam structure, hanya stack pengguna yang digunakan, dan stack kernel nya kosong.
- Ketika proses memasuki kernel, kode kernel dijalankan pada file proses kernel stack; ketika suatu proses ada di kernel, stack pengguna masih berisikan data, tetapi tidak digunakan secara aktif.

- Ketika suatu proses membuat system call, prosesor beralih ke stack kernel, menaikkan tingkat hak istimewa perangkat keras, dan mulai menjalankan instruksi kernel yang mengimplementasikan system call.
- Ketika system call selesai, kernel kembali ke user space: perangkat keras menurunkan tingkat hak istimewanya, beralih kembali ke stack pengguna, dan melanjutkan menjalankan instruksi pengguna tepat setelah instruksi system call.
- Sebuah proses thread dapat "memblokir" di kernel untuk menunggu I / O, dan melanjutkan kembali di tempat yang ditinggalkannya saat I / O telah selesai.

- p->state menunjukkan apakah proses dialokasikan, siap untuk dijalankan, berjalan, menunggu I / O, atau keluar.
- p->pgdir menyimpan proses page table, dalam format yang diharapkan perangkat keras x86. xv6 menyebabkan perangkat keras paging menggunakan proses p->pgdir saat menjalankan proses itu. Proses page table juga berfungsi sebagai catatan alamat dari halaman fisik yang dialokasikan untuk menyimpan memori proses.

5. Code: The First Address Space

Ketika PC menyala, ia akan menginisialisasi dengan sendirinya dan kemudian me-*load* sebuah boot loader dari disk ke memori dan menjalankannya.

Xv6's boot loader memuat kernel xv6 dari disk ke memori dan menjalankannya mulai dari **entry**

```
1044 entry:
1045   # Turn on page size extension for 4Mbyte pages
1046   movl    %cr4, %eax
1047   orl     $(CR4_PSE), %eax
1048   movl    %eax, %cr4
```

Boot loader me-*load* kernel xv6 ke memori di physical addess 0x100000.

Alasan kenapa tidak load kernel di 0x80100000, dimana kernel diharapkan untuk menemukan instruksi dan datanya adalah mungkin tidak ada physical memory di high address pada mesin kecil.

Dan alasan kenapa menempatkan kernel di 0x100000 daripada 0x0 adalah jarak address 0xa0000 : 0x100000 berisi perangkat I/O

Untuk memungkinkan sisa kernel berjalan, entry mengatur *page table* yang memetakan virtual addressnya mulai dari 0x80000000 (yang disebut KERNBASE) ke physical addressnya mulai dari 0x0.

```
0207 #define KERNBASE 0x80000000         // First kernel virtual address
```

Page table entry didefinisikan di main.c dimana entry 0 memetakan virtual address 0:0x400000 ke physical address 0:0x400000. Pemetaan ini diperlukan selama enrty sedang dijalankan pada low address, tetapi akhirnya akan dihapus

```
1306 pde_t entrypgdir[NPDENTRIES] = {
1307   // Map VA's [0, 4MB) to PA's [0, 4MB)
1308   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1309   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1310   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1311 };
```

Entry 512 memetakan virtual addresses KERNBASE:KERNBASE+0X400000 ke physical address 0:0x400000. Entry ini akan digunakan oleh kernel setelah entry selesai; itu memetakan high virtual address dimana kernel diharapkan untuk menemukan instruksinya dan data ke low physical memory dimana boot loader memuat mereka. Pemetaan ini membatasi intruksi kernel dan data di 4 Mbytes

```
1306 pde_t entrypgdir[NPDENTRIES] = {
1307   // Map VA's [0, 4MB) to PA's [0, 4MB)
1308   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1309   // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
1310   [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
1311 };
```

Kembali ke entry, ia memuat physical address dari entrypgdir ke kontrol register %cr3. Nilai %cr3 harus physical address.

Itu tidak masuk akal untuk %cr3 untuk menyimpan virtual address dari entrypgdir, karena paging hardware sampai saat ini tidak tahu bagaimana menterjemahkan virtual address.

Symbol entrypgdir mengacu ke sebuah address di high memory dan macro V2P_W0 mengurangi KERNBASE dengan tujuan mencari physical addess. Untuk mengizinkan paging hardware, xv6 mengatur flag CR0_PG di control register %cr0.

```
0213 #define V2P_WO(x) ((x) - KERNBASE)
```

Sekarang entry perlu dipindahkan ke code C kernel dan menjalankannnya di high memory.

Pertama buat stack pointer, %esp, point ke memory untuk digunakan sebagai stack.

```
1057    # Set up the stack pointer.
1058    movl $(stack + KSTACKSIZE), %esp
```

Semua symbol memiliki high address, termasuk stack, jadi stack masih akan valid bahkan ketika low mappings dihapuskan. Akhirnya entry melompat ke main, yang dimana juga high address.

6. Code: Creating the First Process

Setelah main menginisialisasi beberapa perangkat dan subsistem, ia membuat first process dengan memanggil userinit. Tindakan pertama userinit adalah call allocproc.

```
2518 // Set up first user process.
2519 void
2520 userinit(void)
2521 {
2522   struct proc *p;
2523   extern char _binary_initcode_start[], _binary_initcode_size[];
2524
2525   p = allocproc();
2526
```

Tugas dari allocproc adalah mengalokasikan sebuah slot (struct proc) di table proses dan menginisialisasi bagian bagian dari status proses yang diperlukan agar kernel threadnya bisa menjalankan.

Alloproc memindai proc table untuk slot dengan status UNUSED

```
2480    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481      if(p->state == UNUSED)
2482        goto found;
```

Ketika menemukan unused slot, allocproc menyetel status ke EMBRYO untuk menandainya sebagai used dan memberikan proses sebuah unique pid

Selanjutnya ia mencoba untuk mengalokasikan kernel stack untuk kernel thread proses. Jika memory alokasi kegagalan, alloproc mengubah state Kembali ke UNUSED dan mengembalikan ke 0 sebagai sinyal kegalalan.

```
2469 // If found, change state to EMBRYO and initialize
2470 // state required to run in the kernel.
2471 // Otherwise return 0.
2472 static struct proc*
2473 allocproc(void)
2474 {
2475   struct proc *p;
2476   char *sp;
2477
2478   acquire(&ptable.lock);
2479
2480   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2481     if(p->state == UNUSED)
2482       goto found;
2483
2484   release(&ptable.lock);
2485   return 0;
2486
2487 found:
2488   p->state = EMBRYO;
2489   p->pid = nextpid++;
```

Alloproc menyiapkan proses baru dengan kernel stack yang disiapkan khusus dan set of kernel register yang menyebabkan “return” ke user space ketika pertama kali dijalankan.

Alloproc memerlukan bagian dari pekerjaan ini dengan mengatur nilai perhitungan program kembali yang akan menyebabkan kernel thread proses baru untuk menjalankan pertama kali di forkret dan kemudian di trapret

Kernel thread akan mulai menjalankan dengan register contens yang disalin dari p->context. Dan pengaturan p->context->eip ke forkret akan menyebabkan kernel thread menjalankan di awal forkret

```
2504   // Set up new context to start executing at forkret,
2505   // which returns to trapret.
2506   sp -= 4;
2507   *(uint*)sp = (uint)trapret;
2508
2509   sp -= sizeof *p->context;
2510   p->context = (struct context*)sp;
2511   memset(p->context, 0, sizeof *p->context);
2512   p->context->eip = (uint)forkret;
2513
2514   return p;
2515 }
```

Fungsi ini akan kembali ke address manapun yang ada dibagian bawah stack. Context switch code mengatur stack pointer ke point tepat diakhir dari p->context.

Alloproc menempatkan p->context di stack, dan meletakkan pointer ke trapret di atas itu; dimana frokret akan kembali.

Trapret mengembalikan user register dari nilai yang disimpan di bagian atas kernel stack dan melompat ke dalam proses.

Setiap kali control transfer ke kernel saat proses sedang berjalan, hardware danxv6 trap code entry menyimpan user register pada proses kernel stack.

Userinit menulis nilai di bagian atas stack baru yang terlihat seperti yang akan ada di sana jika proses telah masuk kernel via interrupt, sehingga code biasa untuk returning dari kernel kembali ke proses user code akan berfungsi.

```
2533    p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2534    p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2535    p->tf->es = p->tf->ds;
2536    p->tf->ss = p->tf->ds;
2537    p->tf->eflags = FL_IF;
2538    p->tf->esp = PGSIZE;
2539    p->tf->eip = 0;  // beginning of initcode.S
```

Proses pertama akan mengalankan program kecil (initcode.S) Proses ini membutuhkan physical memory untuk menyimpan program ini, program perlu disalin ke memori tersebut dan proses perlu sebuah page table yang memetakan user-space address ke memory tersebut.

Awal dari memory proses pertama user-space adalah bentuk compiled dari initcode.S; sebagai bagian dari proses pembuatan kernel, tautan menyematkan biner di kernel dan mendefinisikan dua symbol special, _binary_initcode_start dan _binary_initcode_size, yang menunjukkan lokasi dan ukuran biner.

User init menyalin biner tersebut ke dalam proses baru memory dengan memanggil inituvm, yang mengalokasikan satu page dari physical memory, memetakan virtual address 0 ke memory itu dan menyalin biner ke page tersebut

```
1886 inituvm(pde_t *pgdir, char *init, uint sz)
1887 {
1888   char *mem;
1889
1890   if(sz >= PGSIZE)
1891     panic("inituvm: more than a page");
1892   mem = kalloc();
1893   memset(mem, 0, PGSIZE);
1894   mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
1895   memmove(mem, init, sz);
1896 }
```

Kemudian userinit mengatur trap frame dengan status user mode awal: register %cs berisi pemilih segmen untuk SEG_UCODE segemen yang berjalan pada privilege level DPL_USER (yaitu user mode bukan kernel mode), dan juga %ds, %es dan %ss menggunakan SEG_UDATA dengan provilage DPL_USER

Bit %eflags FL_IF diset untuk mengizinkan interupsi hardware

Stack pointer %esp diset ke proses virtual address yang terbesari, p->sz. Instruction pointer diset ke entry point untuk initcode, address 0.

Fungsi userinit sets p->name untuk initcode terutama untuk debugging. Setting p->cwd menetapkan proses dari current working directory

Setelah proses diinisialisasi, userinit menandainya tersedia untuk penjadwalan dengan mengatur p->state to RUNNABLE

7. Code: Running the First Process

Menjalankan Proses Pertama - Bagian 1 (Awal)

- Setelah dipersiapkan, process state pertama pun siap untuk dijalankan.
- Setelah main memanggil userinit, mpmain pun memanggil scheduler untuk memulai proses pada line 1257.

```
1250 // Common CPU setup code.
1251 static void
1252 mpmain(void)
1253 {
1254   cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
1255   idtinit();       // load idt register
1256   xchg(&(mycpu()->started), 1); // tell startothers() we're up
1257   scheduler();     // start running processes
1258 }
```

- Lalu, scheduler pada line 2758 mencari sebuah proses dengan p->state untuk diatur menjadi RUNNABLE (state dimana proses siap untuk dijalankan).
- Proses yang dicari tersebut hanya ada satu, yaitu initproc.

```
2758 scheduler(void)
2759 {
2760   struct proc *p;
2761   struct cpu *c = mycpu();
2762   c->proc = 0;
2763
2764   for(;;){
2765     // Enable interrupts on this processor.
2766     sti();
2767
2768     // Loop over process table looking for process to run.
2769     acquire(&ptable.lock);
2770     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771       if(p->state != RUNNABLE)
2772         continue;
2773
2774       // Switch to chosen process.  It is the process's job
2775       // to release ptable.lock and then reacquire it
2776       // before jumping back to us.
2777       c->proc = p;
2778       switchuvm(p);
2779       p->state = RUNNING;
2780
2781       swtch(&(c->scheduler), p->context);
2782       switchkvm();
2783
2784       // Process is done running for now.
2785       // It should have changed its p->state before coming back.
2786       c->proc = 0;
2787     }
2788     release(&ptable.lock);
2789
2790   }
2791 }
```

- Setelah variabel proc per-cpu diatur sampai proses itu ditemukan, selanjutnya proses memanggil switchuvm untuk memberitahu hardware untuk mulai menggunakan page table milik proses sasaran pada line 1879.
- Kita dapat mengganti page tables saat mengeksekusi dalam kernel karena stupkvm mengakibatkan semua proses pada page tables memiliki pemetaan yang identik untuk kode kernel dan data.

```
1860 switchuvm(struct proc *p)
1861 {
1862   if(p == 0)
1863     panic("switchuvm: no process");
1864   if(p->kstack == 0)
1865     panic("switchuvm: no kstack");
1866   if(p->pgdir == 0)
1867     panic("switchuvm: no pgdir");
1868
1869   pushcli();
1870   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
1871                                 sizeof(mycpu()->ts)-1, 0);
1872   mycpu()->gdt[SEG_TSS].s = 0;
1873   mycpu()->ts.ss0 = SEG_KDATA << 3;
1874   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
1875   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
1876   // forbids I/O instructions (e.g., inb and outb) from user space
1877   mycpu()->ts.iomb = (ushort) 0xFFFF;
1878   ltr(SEG_TSS << 3);
1879   lcr3(V2P(p->pgdir));  // switch to process's address space
1880   popcli();
1881 }
```

- switchuvm juga mempersiapkan sebuah task state segment SEG_TSS yang menginstruksikan hardware untuk menjalankan system calls dan menginterupsi proses milik kernel stack.

Menjalankan Proses Pertama - Bagian 2

- Sekarang, scheduler mengatur p->state menjadi RUNNING dan memanggil swtch pada line 3058 untuk melakukan peralihan (switch) konteks ke thread kernel milik proses sasaran.

```
3058 swtch:
3059   movl 4(%esp), %eax
3060   movl 8(%esp), %edx
3061
3062   # Save old callee-save registers
3063   pushl %ebp
3064   pushl %ebx
3065   pushl %esi
3066   pushl %edi
3067
3068   # Switch stacks
3069   movl %esp, (%eax)
3070   movl %edx, %esp
3071
3072   # Load new callee-save registers
3073   popl %edi
3074   popl %esi
3075   popl %ebx
3076   popl %ebp
3077   ret
```

- Pertama, swtch menyimpan current registers. scheduler memberitahu swtch untuk menyimpan current hardware registers di dalam penyimpanan per-cpu (cpu->scheduler).
- Kemudian, swtch memuat (load) registers dari kernel thread sasaran (p->context) yang telah disimpan ke dalam hardware registers x86 bersamaan dengan stack pointer dan instruction pointer.

- Instruksi ret terakhir pada line 3077 mengeluarkan (pop) %eip milik proses sasaran keluar dari stack, mengakhiri peralihan (switch) konteks.

```
3072    # Load new callee-save registers
3073    popl %edi
3074    popl %esi
3075    popl %ebx
3076    popl %ebp
3077    ret
```

- Sekarang, prosesor sedang berjalan pada stack kernel dari proses p.

Menjalankan Proses Pertama - Bagian 3

- Sebelumnya, Allocproc sudah mengatur p->context->eip milik initproc ke forkret sehingga ret pun memulai untuk mengeksekusi forkret.

```
Aug 29 15:52 2017  xv6/proc.c  Page 3


2500   // Leave room for trap frame.
2501   sp -= sizeof *p->tf;
2502   p->tf = (struct trapframe*)sp;
2503
2504   // Set up new context to start executing at forkret,
2505   // which returns to trapret.
2506   sp -= 4;
2507   *(uint*)sp = (uint)trapret;
2508
2509   sp -= sizeof *p->context;
2510   p->context = (struct context*)sp;
2511   memset(p->context, 0, sizeof *p->context);
2512   p->context->eip = (uint)forkret;
```

- Pada invocation (seruan) pertama, forkret pada line 2853 menjalankan inisialisasi fungsi-fungsi yang tidak bisa dijalankan dari main karena mereka harus dijalankan di konteks dari proses reguler dengan kernel stack miliknya sendiri.

```
2852 void
2853 forkret(void)
2854 {
2855   static int first = 1;
2856   // Still holding ptable.lock from scheduler.
2857   release(&ptable.lock);
2858
2859   if (first) {
2860     // Some initialization functions must be run in the context
2861     // of a regular process (e.g., they call sleep), and thus cannot
2862     // be run from main().
2863     first = 0;
2864     iinit(ROOTDEV);
2865     initlog(ROOTDEV);
2866   }
2867
2868   // Return to "caller", actually trapret (see allocproc).
2869 }
```

- Lalu, forkret kembali. Allocproc mengatur bahwa word teratas pada stack setelah p->context yang dikeluarkan (pop), akan di-trapret sehingga trapret pun mulai untuk mengeksekusi, dengan %esp diatur ke p->tf.
- Trapret pada line 3324 menggunakan pop instructions untuk memulihkan registers dari trapframe pada line 0602 seperti yang swtch lakukan dengan konteks kernel.

- Kemudian, trapret pada line 3324 menggunakan pop instructions untuk memulihkan registers dari trapframe pada line 0602 seperti yang swtch lakukan dengan konteks kernel, yaitu dengan popal memulihkan general registers, lalu instruksi popl memulihkan %gs, %fs, %es, dan %ds. Di sini, addl melewatkan dua fields: trapno dan errcode.

```
3322   # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325   popal
3326   popl %gs
3327   popl %fs
3328   popl %es
3329   popl %ds
3330   addl $0x8, %esp  # trapno and errcode
3331   iret
```

```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
```

- Akhirnya, instruksi iret pun mengeluarkan (pop) %cs, %eip, %flags, %esp, dan %ss dari stack.
- Isi dari trapframe sudah ditransfer ke CPU state sehingga prosesor melanjutkan pada %eip yang dispesifikan dalam trapframe.

Menjalankan Proses Pertama - Bagian 4 (Akhir)

- Pada saat ini, %eip menyimpan nol dan %esp menyimpan 4096 bytes. Inilah virtual addresses dalam address space dari proses.
- Paging hardware dari prosesor pun menerjemahkan virtual addresses sebelumnya menjadi physical addresses.
- Sebelumnnya, allocuvm sudah mengatur page table dari proses sehingga virtual address nol mengacu pada physical memory yang dialokasikan untuk proses ini.
- allocuvm juga mengatur flag (PTE_U) yang memberitahu paging table untuk mengizinkan kode user untuk mengakses physical memory.

- Fakta bahwa userinit pada line 2533 mengatur bits rendah dari %cs untuk menjalankan kode user dari proses di CPL = 3 berarti bahwa kode user hanya bisa menggunakan pages dengan set PTE_U, dan tidak bisa memodifikasi hardware register yang sensitif seperti %cr3.
- Jadi, proses ini dibatasi hanya dengan menggunakan memorinya sendiri

```
2518 // Set up first user process.
2519 void
2520 userinit(void)
2521 {
2522   struct proc *p;
2523   extern char _binary_initcode_start[], _binary_initcode_size[];
2524
2525   p = allocproc();
2526
2527   initproc = p;
2528   if((p->pgdir = setupkvm()) == 0)
2529     panic("userinit: out of memory?");
2530   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2531   p->sz = PGSIZE;
2532   memset(p->tf, 0, sizeof(*p->tf));
2533   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2534   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2535   p->tf->es = p->tf->ds;
2536   p->tf->ss = p->tf->ds;
2537   p->tf->eflags = FL_IF;
2538   p->tf->esp = PGSIZE;
2539   p->tf->eip = 0;  // beginning of initcode.S
```

8. The First System Call: exec

System Call Pertama: exec

- Langkah pertama dari file initciode.S (line 8409) adalah untuk meng-invoke syscall exec.

```
8407 # exec(init, argv)
8408 .globl start
8409 start:
8410   pushl $argv
8411   pushl $init
8412   pushl $0  // where caller pc would be
8413   movl $SYS_exec, %eax
8414   int $T_SYSCALL
```

- exec menggantikan memori dan registers dari proses terkini dengan sebuah program yang baru, tapi ia meninggalkan file descriptors, id proses, dan proses parent tanpa perubahan apapun.

- File Initcode.S (line 8409) mulai untuk memasukkan (push) 3 nilai pada stack ($argv, $init, dan $0) yang kemudian mengatur %eax ke SYS_exec dan menjalankan int T_SYSCALL: ia meminta kernel untuk menjalankan syscall exec. Jika semuanya berjalan dengan baik, exec tidak akan kembali: ia mulai untuk menjalankan program yang dinamai oleh $init, yang mana adalah pointer ke string / init NUL-terminated dari line 8422-8424.

```
8409 start:
8410   pushl $argv
8411   pushl $init
8412   pushl $0  // where caller pc would be
8413   movl $SYS_exec, %eax
8414   int $T_SYSCALL
```

```
8422 # char init[] = "/init\0";
8423 init:
8424   .string "/init\0"
```

- Argumen lainnya adalah array argv dari argumen-argumen command-line, yang mana nol pada akhir array (line 8430) menandai akhir dari array tersebut. Jika exec gagal dan melakukan pengembalian, initcode melakukan pemanggilan loops ke syscall exit, yang secara pasti tidak seharus kembali (dari line 8416-8420).

```
8426 # char *argv[] = { init, 0 };
8427 .p2align 2
8428 argv:
8429   .long init
8430   .long 0
```

```
8416 # for(;;) exit();
8417 exit:
8418   movl $SYS_exit, %eax
8419   int $T_SYSCALL
8420   jmp exit
```

- Kode ini (pada slide sebelumnya) secara manual membuat system call pertama untuk terlihat seperti system call seperti biasanya (lebih lengkap pada chapter 3).
- Seperti sebelumnya, setup / aturan ini menghindari special-casing pada proses pertama (pada kasus ini, system call pertama) dan sebagai gantinya menggunakan ulang kode yang xv6 harus sediakan untuk operasi standar.
- Pengimplementasian dari exec secara rinci akan dibahas di chapter 2.

- Sekarang proses initcode pada line 8400 selesai dilakukan, dan proses akan menjalankan /init (line 8500) sebagai gantinya.

```
Aug 29 15:52 2017  xv6/initcode.S  Page 1


8400 # Initial process execs /init.
8401 # This code runs in user space.
8402
8403 #include "syscall.h"
8404 #include "traps.h"
8405
8406
8407 # exec(init, argv)
8408 .globl start
8409 start:
8410   pushl $argv
8411   pushl $init
8412   pushl $0  // where caller pc would be
8413   movl $SYS_exec, %eax
8414   int $T_SYSCALL
```

```
Aug 29 15:52 2017  xv6/init.c  Page 1


8500 // init: The initial user-level program
8501
8502 #include "types.h"
8503 #include "stat.h"
8504 #include "user.h"
8505 #include "fcntl.h"
8506
8507 char *argv[] = { "sh", 0 };
8508
8509 int
8510 main(void)
8511 {
8512   int pid, wpid;
8513
8514   if(open("console", O_RDWR) < 0){
8515     mknod("console", 1, 1);
8516     open("console", O_RDWR);
8517   }
8518   dup(0);  // stdout
```

- Terakhir, init pada line 8510 membuat sebuah console device file baru jika diperlukan dan kemudian membukanya sebagai file descriptors 0, 1, dan 2. Lalu, ia akan melakukan loops, memulai console shell, menangani orphaned zombies sampai shell exit, dan berulang. Sistem telah selesai.

```
8510 main(void)
8511 {
8512   int pid, wpid;
8513
8514   if(open("console", O_RDWR) < 0){
8515     mknod("console", 1, 1);
8516     open("console", O_RDWR);
8517   }
8518   dup(0);  // stdout
8519   dup(0);  // stderr
8520
8521   for(;;){
8522     printf(1, "init: starting sh\n");
8523     pid = fork();
8524     if(pid < 0){
8525       printf(1, "init: fork failed\n");
8526       exit();
8527     }
8528     if(pid == 0){
8529       exec("sh", argv);
8530       printf(1, "init: exec sh failed\n");
8531       exit();
8532     }
8533     while((wpid=wait()) >= 0 && wpid != pid)
8534       printf(1, "zombie!\n");
8535   }
8536 }
```

9. Real World

Real World

- Kebanyakan sistem operasi sudah mengadopsi konsep proses, dan kebanyakan proses mirip seperti milik xv6.
- Sistem operasi yang sesungguhnya akan menemukan struktur-struktur yang bebas proc dengan list yang bebas eksplisit dalam waktu konstan, berbanding terbalik dengan linear-time search yang ada di allocproc. Perlu diketahui, xv6 menggunakan linear scan untuk simplicity / kemudahan saja.
- Tata letak / layout address space dari xv6 memiliki cacat, yang mana ia tidak dapat menggunakan lebih dari 2 GB physical RAM. Hal ini mungkin untuk diperbaiki, walaupun ada langkah terbaik yang bisa dilakukan yaitu dengan beralih kepada mesin yang dibekali dengan 64-bit addresses.

Sumber

- Cox, R., Kaashoek, F., & Morris, R. (2017). Chapter 1. In *xv6 a simple, Unix-like teaching operating system* (pp. 17–28). xv6-book@pdos.csail.mit.edu.

Pertanyaan

- Cicel: Perbedaan kernel monolitik dan mikro kernel?
- Pramudio: Bagaimana antara aplikasi dan hardware bisa diisolasi?
- Ridho: Apakah os lain menggunakan linear scan dan apakah hal tersebut masih relevan?

Terima Kasih.