Processes, Address Spaces, and Context Switches

Chester Rebeiro
IIT Madras
Executing Apps (Process)

- **Process**
  - A program in execution
  - Most important abstraction in an OS
  - Comprises of
    - Code
    - Data
    - Stack
    - Heap
    - State in the OS
    - Kernel stack
  - State contains: registers, list of open files, related processes, etc.

```c
#include <stdio.h>

int main()
{
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

$gcc hello.c

$./a.out
# Program ≠ Process

<table>
<thead>
<tr>
<th>Program</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>code + static and global data</td>
<td>Dynamic instantiation of code + data + heap + stack + process state</td>
</tr>
<tr>
<td>One program can create several processes</td>
<td>A process is unique isolated entity</td>
</tr>
</tbody>
</table>
Process Address Space

- **Virtual Address Map**
  - All memory a process can address
  - Large contiguous array of addresses from 0 to MAX_SIZE
Process Address Space

- Each process has a different address space
- This is achieved by the use of virtual memory
- I.e. 0 to MAX_SIZE are virtual memory addresses
Virtual Address Mapping

Virtual Memory

Physical Memory

Process A
- Stack
- Heap
- Data
- Text (instructions)

Page Table

Process B
- Stack
- Heap
- Data
- Text (instructions)

Page Table
Advantages of Virtual Address Map

• Isolation (private address space)
  – One process cannot access another process’ memory

• Relocatable
  – Data and code within the process is relocatable

• Size
  – Processes can be much larger than physical memory
Process Address Map in xv6

• Entire kernel mapped into every process address space
  – This allows easy switching from user code to kernel code (i.e. during system calls)
    • No change of page tables needed
  – Easy access of user data from kernel space
Process Stacks

• Each process has 2 stacks
  – User space stack
    • Used when executing user code
  – Kernel space stack
    • Used when executing kernel code (for eg. during system calls)
  – Advantage: Kernel can execute even if user stack is corrupted
    (Attacks that target the stack, such as buffer overflow attack, will not affect the kernel)
Process Management in xv6

• Each process has a PCB (process control block) defined by `struct proc` in xv6
• Holds important process specific information
• Why?
  – Allows process to resume execution after a while
  – Keep track of resources used
  – Track the process state

ref : proc.h (struct proc) (2353)
Summary of entries in PCB

• More entries

```c
struct proc {
    uint sz;
    pde_t* pgdir;
    char *kstack;
    enum procstate state;
    int pid;
    struct proc *parent;
    struct trapframe *tf;
    struct context *context;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd;
    char name[16];
};
```

- Size of process memory
- Page directory pointer for process
- Kernel stack pointer
- Files opened
- Current working directory
- Executable name
Entries in PCB

• PID
  – Process Identifier
  – Number incremented sequentially
    • When maximum is reached
    • Reset and continue to increment.
    • This time skip already allocated PID numbers
Process States

- Process State: specifies the state of the process

  - **EMBRYO** → The new process is currently being created
  - **RUNNABLE** → Ready to run
  - **RUNNING** → Currently executing
  - **SLEEPING** → Blocked for an I/O

Other states: ZOMBIE (later)

ref: proc.h (struct proc) 2350
Scheduling Runnable Processes

Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
Scheduler picks another process from the ready queue
Performs a context switch
Page Directory Pointer
Entries in PCB

- Pointer to trapframe
• **Context pointer**
  - Contains registers used for context switches.
  - Registers in context: `%edi, %esi, %ebx, %ebp, %eip`
  - Stored in the kernel stack space
Storing procs in xv6

- In a globally defined array present in ptable
- NPROC is the maximum number of processes that can be present in the system (\#define NPROC 64)
- Also present in ptable is a lock that serializes access to the array.

```c
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```
Creating a Process by Cloning

• Cloning
  – Child process is an exact replica of the parent
  – Fork system call
Creating a Process by Cloning (using fork system call)

- In parent
  - fork returns child pid

- In child process
  - fork returns 0

- Other system calls
  - Wait, returns pid of an exiting child

```c
int pid;

pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit(0);
}
```
Virtual Addressing Advantage (easy to make copies of a process)

- Making a copy of a process is called forking.
  - Parent (is the original)
  - child (is the new process)
- When fork is invoked,
  - child is an exact copy of parent
    - When fork is called all pages are shared between parent and child
    - Easily done by copying the parent’s page tables
Modifying Data in Parent or Child

```c
int i=0, pid;
pid = fork();
if (pid > 0){
    sleep(1);
    printf("parent : %d\n", i);
    wait();
} else{
    i = i + 1;
    printf("child : %d\n", i);
}
```

Output

parent : 0
child : 1
Copy on Write (COW)

• When data in any of the shared pages change, OS intercepts and makes a copy of the page.
• Thus, parent and child will have different copies of this page.
• Why?
  – A large portion of executables are not used.
  – Copying each page from parent and child would incur significant disk swapping.. huge performance penalties.
  – Postpone copying of pages as much as possible thus optimizing performance.

This page now is no longer shared.
How COW works

• When forking,
  – Kernel makes COW pages as read only
  – Any write to the pages would cause a page fault
  – The kernel detects that it is a COW page and duplicates the page
Executing a Program (exec system call)

- **exec system call**
  - Load into memory and then execute

- **COW big advantage for exec**
  - Time not wasted in copying pages.
  - Common code (for example shared libraries) would continue to be shared

```c
int pid;

pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execlp("ls", ",", NULL);
    exit(0);
}
```
Virtual Addressing Advantages (Shared libraries)

- Many common functions such as `printf` implemented in shared libraries
- Pages from shared libraries, shared between processes
The first process

• Unix: `/sbin/init` (xv6 initcode.S)
  – Unlike the others, this is created by the kernel during boot
  – **Super parent.**
    • Responsible for forking all other processes
    • Typically starts several scripts present in `/etc/init.d` in Linux
Process tree

Processes in the system arranged in the form of a tree.

pstree in Linux

Who creates the first process?
Process Termination

• Voluntary: `exit(status)`
  – OS passes exit status to parent via `wait(&status)`
  – OS frees process resources

• Involuntary: `kill(pid, signal)`
  – Signal can be sent by another process or by OS
  – `pid` is for the process to be killed
  – `signal` a signal that the process needs to be killed
    • Examples: SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP
Zombies

- When a process terminates it becomes a **zombie** (or **defunct** process)
  - PCB in OS still exists even though program no longer executing
  - Why? So that the parent process can read the child’s exit status (through **wait** system call)

- When parent reads status,
  - zombie entries removed from OS… **process reaped**!

- Suppose parent does’nt read status
  - Zombie will continue to exist infinitely … **a resource leak**
  - These are typically found by a reaper process
Orphans

• When a parent process terminates before its child
• Adopted by first process (/sbin/init)
Orphans contd.

- **Unintentional orphans**
  - When parent crashes

- **Intentional orphans**
  - Process becomes detached from user session and runs in the background
  - Called *daemons*, used to run background services
  - See *nohup*
The first process in xv6
The first process

- initcode.S

- Creating the first process
  - `main` (1239) invokes `userinit` (2503)
  - `userinit`
    - allocate a process id, kernel stack, fill in the proc entries
    - Setup kernel page tables
    - copy initcode.S to 0x0
    - create a user stack
    - set process to runnable
      - the scheduler would then execute the process
allocproc (2455)

1. Find an unused proc entry in the PCB table.
2. Set the state to EMBRYO (neither RUNNING nor UNUSED).
3. Set the pid (in real systems. Need to ensure that the pid is unused).

```c
static struct proc* allocproc(void) {
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;

    found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    release(&ptable.lock);

    // Allocate kernel stack.
    if((p->kstack = kalloc()) == 0){
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe*)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint*)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context*)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;

    return p;
}
```
allocate kernel stack of size 4KB.

We next need to allocate space on to kernel stack for
1. the trapframe
2. trapret
3. context

forkret: this is important, but we’ll look at it later
Setup pagetables

- Kernel page tables
  - Invoked by setupkvm(1837)

- User page tables
  - Setup in inituvm (1903)

Create PTEs in page directory
VA = 0 → PA (v2p(mem))
Size 1 page (4KB)
...do the rest

```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[], _binary_initcode_size[];
2506   p = allocproc();
2507   initproc = p;
2508   if((p->pgdir = setupkvm()) == 0)
2509     panic("userinit: out of memory");
2510   inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2511   p->sz = 4096;
2512   memset(p->tf, 0, sizeof(*p->tf));
2513   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2514   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2515   p->tf->es = p->tf->ds;
2516   p->tf->ss = p->tf->ds;
2517   p->tf->eflags = FL_IF;
2518   p->tf->esp = PGSIZE;
2519   p->tf->eip = 0; // beginning of initcode.S
2520   safestrcpy(p->name, "initcode", sizeof(p->name));
2521   p->cwd = namei("/");
2522   p->state = RUNNABLE;
2523 }
```
Executing User Code

• The kernel stack of the process has a trap frame and context
• The process is set as RUNNABLE
• The scheduler is then invoked from main
  
  \[
  \text{main} \rightarrow \text{mpmain (1241)} \rightarrow \text{scheduler (1257)}
  \]
  
  – The initcode process is selected
    (as it is the only process runnable)
  – …and is then executed
Scheduling the first process
Recall: the virtual memory map

Before userinit:

- eip → text
- esp → stack

0x80000000

0x0

After userinit:

- eip → text
- esp → stack
- Initcode
- Initcode kstack

0x80000000

0x0

The code and stack for Initcode has been setup.

But we are still executing kernel code with the kernel stack.

scheduler() changes this to get Initcode to execute
What we need!

before userinit

0x80000000

0x0

after userinit

0x0

Initcode

Initcode kstack

Need to get here
(stack starts at 4KB and grows downwards)
Scheduler ()

• main ➔ mpmain (1241) ➔ scheduler

```c
2708 scheduler(void)
2709 {
2710     struct proc *p;
2711     for (;;) {
2712         // Enable interrupts on this processor.
2713         sti();
2714         // Loop over process table looking for process to run.
2715         acquire(&ptable.lock);
2716         for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
2717             if (p->state != RUNNING)
2718                 continue;
2719             // Switch to chosen process. It is the process’s job
2720             // to release ptable.lock and then reacquire it
2721             // before jumping back to us.
2722             proc = p;
2723             switchvm(p);
2724             p->state = RUNNING;
2725             switch(&cpu->scheduler, proc->context);
2726             switchkvm();
2727             // Process is done running for now.
2728             // It should have changed its p->state before coming back.
2729             proc = 0;
2730         }
2731         release(&ptable.lock);
2732     }
2733     proc = 0;
2734 }
```

Find the process which is RUNNABLE. In this case initcode is selected.

```
extern struct proc *proc asm("%gs:4"); // cpus[cpunum()].proc
```
switchuvm

New TSS segment in GDT

Set the new stack (this is the kernel stack corresponding to initcode.S)

Set the new page tables (corresponding to initcode.S)

Load TSS offset

1873    switchuvm(struct proc *p)
1874    {
1875        pushcli();
1876        cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
1877        cpu->gdt[SEG_TSS].s = 0;
1878        cpu->ts.ss0 = SEG_KDATA << 3;
1879        cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
1880        ltr(SEG_TSS << 3);
1881        if(p->pgdir == 0)
1882            panic("switchuvm: no pgdir");
1883        lcr3(v2p(p->pgdir)); // switch to new address space
1884        popcli();
1885    }
swtch(cpu→scheduler, proc→context) (1)

```assembly
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977 ret
```
swtch(cpu → scheduler, proc → context) (2)

2957  .globl swtch
2958  swtch:
2959      movl 4(%esp), %eax
2960      movl 8(%esp), %edx
2961
2962      # Save old callee-save registers
2963      pushl %ebp
2964      pushl %ebx
2965      pushl %esi
2966      pushl %edi
2967
2968      # Switch stacks
2969      movl %esp, (%eax)
2970      movl %edx, %esp
2971
2972      # Load new callee-save registers
2973      popl %edi
2974      popl %esi
2975      popl %ebx
2976      popl %ebp
2977      ret
swtch(cpu → scheduler, proc → context) (3)

2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
swtch(cpu → scheduler, proc → context) (4)

```assembly
2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
```
swtch(\text{cpu} \rightarrow \text{scheduler}, \text{proc} \rightarrow \text{context}) (5)

So, swtch return corresponds to initcode’s eip. Where can that be?
return from switch

- recollect forkret (a couple of slide back)

\[
p \rightarrow \text{context} \rightarrow \text{eip} = (\text{uint}) \text{forkret};
\]

- So, switch on return executes forkret
forkret

- Does nothing much.
  - Initializes a log for the first process
- And then returns to trapret

```
2783  forkret(void)
2784  {
2785    static int first = 1;
2786    // Still holding ptable.lock from scheduler.
2787    release(&ptable.lock);
2788
2789    if (first) {
2790      // Some initialization functions must be run in the context
2791      // of a regular process (e.g., they call sleep), and thus cannot
2792      // be run from main().
2793      first = 0;
2794      initlog();
2795    }
2796
2797    // Return to "caller", actually trapret (see allocproc).
2798  }
```
recall the trapframe

- Allocated in allproc.
- Filled in userinit

```
2502 userinit(void)
2503 {
2504  struct proc *p;
2505  extern char _binary_initcode_start[], _binary_initcode_size[];
2506
2507  p = allocproc();
2508  initproc = p;
2509  if ((p->pgdir = setupkvm()) == 0)
2510      panic("userinit: out of memory?");
2511  initvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
2512  p->sz = PGSIZE;
2513  memset(p->tf, 0, sizeof(*p->tf));
2514  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2515  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2516  p->tf->es = p->tf->ds;
2517  p->tf->ss = p->tf->ds;
2518  p->tf->eflags = FL_IF;
2519  p->tf->esp = PGSIZE;
2520  p->tf->eip = 0; // beginning of initcode.S
2521  strcpy(p->name, "initcode", sizeof(p->name));
2522  p->cwd = namei("/");
2523
2524  p->state = RUNNABLE;
```

ref : struct trapframe in x86.h (0602 [06])
trapret

Initcode.S
Kernel stack

- SS
- ESP
- EFLAGS
- CS
- EIP
- Error Code
- Trap Number
  - ds
  - es
  - ...
  - eax
  - ecx
  - ...
  - esi
  - edi

```c
.globl trapret
trapret:
    popal
    popl gs
    popl fs
    popl es
    popl ds
    addl $0x8, %esp # trapno and errcode
    iret
```
Return from trapret (iret)

```
3254 alltraps:
3255  # Build trap frame.
3256  pushl %ds
3257  pushl %es
3258  pushl %fs
3259  pushl %gs
3260  pushal
3261
3262  # Set up data and per-cpu segments.
3263  movw $(SEG_KDATA<<3), %ax
3264  movw %ax, %ds
3265  movw %ax, %es
3266  movw $(SEG_KCPU<<3), %ax
3267  movw %ax, %fs
3268  movw %ax, %gs
3269
3270  # Call trap(tf), where tf=%esp
3271  pushl %esp
3272  call trap
3273  addl $4, %esp
3274
3275  # Return falls through to trapret...
3276  .globl trapret
3277  trapret:
3278  popal
3279  popl %gs
3280  popl %fs
3281  popl %es
3282  popl %ds
3283  addl $0x8, %esp # trapno and errcode
3284  iret
```

Initcode.S

Kernel stack

```
<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp→</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Loads the new

- `%cs = SEG_UCODE | DPL_USER`
- `%eip = 0`
- `eflags = 0`
- `%ss = SEG_UDATA | DPL_USER`
- `%esp = 4096 (PGSZE)`

... there by starting initcode.S
finally ... initcode.S 😊

- Invokes system call exec to invoke /init

exec('/init')
init.c

- forks and creates a shell (sh)
CPU Context Switching
Process States

NEW (in xv6 EMBRYO) → The new process is currently being created
READY (in xv6 RUNNABLE) → Ready to run
RUNNING → Currently executing
WAITING (in xv6 SLEEPING) → Blocked for an I/O
Context Switches

1. When a process switches from RUNNING to WAITING (eg. due to an I/O request)
2. When a process switches from RUNNING to READY (eg. when an interrupt occurs)
3. When a process switches from WAITING to READY (eg. Due to I/O completion)
4. When a process terminates
Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
Scheduler picks another process from the ready queue
Performs a context switch
Process Context

• The process context contains all information, which would allow the process to resume after a context switch
Process Contexts Revisited

• Segment registers not needed
  – Since they are constants across kernel contexts
• Caller has saved eax, ecx, edx
  – By x86 convention
• Context contain just 5 registers
  – edi, esi, ebx, ebp, eip
• Contexts always stored at the bottom of the process’ kernel stack
How to perform a context switch?

1. Save current process state
2. Load state of the next process
3. Continue execution of the next process

• Need to save current process registers without changing them
  – Not easy!! because saving state needs to execute code, which will modify registers
  – Solution: Use hardware + software ... architecture dependent
Context switch in xv6

1. Gets triggered when any interrupt is invoked
   - Save P1's user-mode CPU context and switch from user to kernel mode
2. Handle system call or interrupt
3. Save P1’s kernel CPU context and switch to scheduler CPU context
4. Select another process P2
5. Switch to P2’s address space
6. Save scheduler CPU context and switch to P2’s kernel CPU context
7. Switch from kernel to user mode and load P2’s user-mode CPU context
Tracing Context Switch (The Timer Interrupts)

• Programming the Timer interval
  – Single Processor Systems: PIT ([80], 8054)
  – Multi Processor Systems: LAPIC

• Programmed to interrupt processor every 10ms
Timer Interrupt Stack

vector.s
[32]

alltraps
(3254)

trap
(3351)

yield
(2272)

sched
(2753)

swtch
(2958)

kernel stack of process 1

SS
ESP
EFLAGS
CS
EIP
0 (error code)
32 (trap num)
ds
es
fs
gs
All registers
esp
eip (alltraps)
trap locals
eip (trap)
yield locals
eip (yield)
sched locals
cpu→scheduler
&proc→context
(eip) sched

only if stack changed
By hardware

trapframe (602)
trap, yield & sched

trap.c (3423)

```c
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if (proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

(2772)

```c
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock); //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

(2753)

```c
// Enter scheduler. Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;
  if (!holding(&ptable.lock))
    panic("sched ptable.lock");
  if (cpu->ncli != 1)
    panic("sched locks");
  if (proc->state == RUNNING)
    panic("sched running");
  if (readflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```
swtch(&proc\rightarrow context, cpu\rightarrow scheduler)

2957 .globl swtch
2958 swtch:
2959   movl 4(%esp), %eax
2960   movl 8(%esp), %edx
2961
2962   # Save old callee-save registers
2963   pushl %ebp
2964   pushl %ebx
2965   pushl %esi
2966   pushl %edi
2967
2968   # Switch stacks
2969   movl %esp, (%eax)
2970   movl %edx, %esp
2971
2972   # Load new callee-save registers
2973   popl %edi
2974   popl %esi
2975   popl %ebx
2976   popl %ebp
2977   ret
swtch(&proc\rightarrow context, cpu\rightarrow scheduler)

globl swtch
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx
# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi
# Switch stacks
movl %esp, (%eax)
movl %edx, %esp
# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret

trapframe
esp
eip (alltraps)
trap locals
eip (trap)
yield locals
eip (yield)
sched locals
cpu\rightarrow scheduler
&proc\rightarrow context
(eip) sched
ebp
ebx
esi
edi

Scheduler stack

proc\rightarrow context
cpu\rightarrow scheduler
eip (scheduler)
ebp
ebx
esi
edi

Kernel stack

Process 1

Scheduler stack

esp

eax edx	

&proc\rightarrow context
cpu\rightarrow scheduler

eip

stack

Process 1

Kernel stack

Scheduler stack

esp
Execution in Scheduler

eswtch returns to line 2729.

1. First switch to kvm pagetables
2. Then select new runnable process
3. Switch to user process page tables
4. swtch(&cpu->scheduler, proc->conetxt)
swtch(&cpu ⇒ scheduler, proc ⇒ context)

2957  .globl  swtch
2958  swtch:
2959       movl  4(%esp), %eax
2960       movl  8(%esp), %edx
2961
2962       # Save old callee-save registers
2963       pushl %ebp
2964       pushl %ebx
2965       pushl %esi
2966       pushl %edi
2967
2968       # Switch stacks
2969       movl  %esp, (%eax)
2970       movl  %edx, %esp
2971
2972       # Load new callee-save registers
2973       popl  %edi
2974       popl  %esi
2975       popl  %ebx
2976       popl  %ebp
2977       ret

Scheduler stack
- proc ⇒ context
- cpu ⇒ scheduler
- eip
- ebp
- ebx
- esi
- edi

Process 2 Kernel stack
- trapframe
  - esp
  - eip (alltraps)
  - trap locals
  - eip (trap)
  - yield locals
  - eip (yield)
  - sched locals
  - cpu ⇒ scheduler
  - &proc ⇒ context
    - (eip) sched
    - ebp
    - ebx
    - esi
    - edi

Swtch returns to sched
sched in Process 2’s context

```c
// Enter scheduler. Must hold only ptable.lock
// and have changed proc->state.
void sched(void)
{
    int intena;

    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(cpu->ncli != 1)
        panic("sched locks");
    if(proc->state == RUNNING)
        panic("sched running");
    if(readflags()&FL_IF)
        panic("sched interruptsible");
    intena = cpu->intena;
    swtch(&proc->context, cpu->scheduler);
    cpu->intena = intena;
}
```

swtch returns to line 2767.

1. Sched returns to yield
2. Yield returns to trap
3. Trap returns to alltraps
4. Alltraps restores user space registers of process 2 and invokes IRET
Context Switching Overheads

• **Direct Factors** affecting context switching time
  – Timer Interrupt latency
  – Saving/restoring contexts
  – Finding the next process to execute

• **Indirect factors**
  – TLB needs to be reloaded
  – Loss of cache locality (therefore more cache misses)
  – Processor pipeline flush
Context Switch Quantum

• A short quantum
  – **Good** because, processes need not wait long before they are scheduled in.
  – **Bad** because, context switch overhead increase

• A long quantum
  – Bad because processes no longer appear to execute concurrently
  – May degrade system performance

• Typically kept between 10ms to 100ms
  – xv6 programs timers to interrupt every 10ms.
System Calls for Process Management
fork system call

- In parent
  - fork returns child pid

- In child process
  - fork returns 0

- Other system calls
  - Wait, returns pid of an exiting child

```c
int pid;
pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit(0);
}
```
Pick an UNUSED proc. Set pid. Allocate kstack. fill kstack with (1) the trapframe pointer, (2) trapret and (3) context np is the proc pointer for the new process

Copy page directory from the parent process (proc→pgdir) to the child process (np→pgdir)
Set size of np same as that of parent
Set parent of np
Copy trapframe from parent to child
In child process, set eax register in trapframe to 0. This is what fork returns in the child process
Other things… copy file pointer from parent, cwd, executable name
Child process is finally made runnable
Parent process returns the pid of the child
Copying Page Tables of Parent

- copyuvm (in vm.c)
  - replicates parents memory pages
  - Constructs new table pointing to the new pages
  - Steps involved
    1. Call kalloc to allocate a page directory (pgdir)
    2. Set up kernel pages in pgdir
    3. For each virtual page of the parent (starting from 0 to its sz)
      i. Find its page table entry (function walkpgdir)
      ii. Use kalloc to allocate a page (mem) in memory for the child
      iii. Use memmove to copy the parent page to mem
      iv. Use mappages to add a page table entry for mem

ref : 2053
Register modifications w.r.t. parent

Registers modified in child process

- `%eax = 0` so that `pid = 0` in child process
- `%eip = forkret` so that child exclusively executes function `forkret`
Exit system call

```c
int pid;

pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit();
}
```
exit internals

- **init**, the first process, can never exit
- For all other processes on exit,
  1. Decrement the usage count of all open files
     - If usage count is 0, close file
  2. Drop reference to in-memory inode
  3. wakeup parent
     - If parent state is `sleeping`, make it `runnable`
     - Needed, cause parent may be sleeping due to a wait
  4. Make init adopt children of exited process
  5. Set process state to `ZOMBIE`
  6. Force context switch to scheduler

Note: page directory, kernel stack, not deallocated here

Ref: proc.c (exit) 2604
exit(void)
{
  struct proc *p;
  int fd;

  if (proc == initproc)
    panic("init exiting");

  // Close all open files.
  for (fd = 0; fd < NOFILE; fd++) {
    if (proc->ofile[fd]) {
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  begin_op();
  input(proc->cwd);
  end_op();
  proc->cwd = 0;
  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->parent == proc) {
      p->parent = initproc;
      if (p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
Wait system call

- Invoked in parent parent
- Parent ‘waits’ until child exits

```c
int pid;
pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
    printf("In child process");
    exit();
}
```
wait internals

Wait system call

process 'p' in ptable

If p is a child

If p is a zombie

Deallocation kernel stack
free page directory
Set p.state to UNUSED

return pid(p)

return -1 if there are no children
int wait(void)
{
    struct proc *p;
    int havekids, pid;

    acquire(&ptable.lock);
    for(;);
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->parent != proc)
            continue;
        havekids = 1;
        if(p->state == ZOMBIE){
            // Found one.
            pid = p->pid;
            kfree(p->kstack);
            p->kstack = 0;
            freevm(p->pgdir);
            p->state = UNUSED;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            release(&ptable.lock);
            return pid;
        }
    }

    // No point waiting if we don't have any children.
    if(!havekids || proc->killed){
        release(&ptable.lock);
        return -1;
    }

    // Wait for children to exit. (See wakeup1 call in proc_exit.)
    sleep(proc, &ptable.lock); //DOC: wait-sleep
}
Executing a Program (exec system call)

- **exec system call**
  - Load a program into memory and then execute it
  - Here ‘ls’ executed.

```c
int pid;
pid = fork();
if (pid > 0){
    pid = wait();
} else{
    execlp("ls", ", " , NULL);
    exit(0);
}
```
ELF Executables
(linker view)

/bin/ls

This is an ELF file

---

ELF format of executable

ELF Header
Section header table
Section 1
Section 2
Section 3
Section 4
---
---

ref : www.skyfree.org/linux/references/ELF_Format.pdf
ref : see man elf
ELF Header

- Identification
  - type
    - Can have values relocatable object, executable, shared object, core file
    - i386, X86_64, ARM, MIPS, etc.
  - virtual address where program begins execution

- Machine details
  - Entry
    - Ptr to program header
      - number of program headers
    - Ptr to section header
      - number of section headers

- Program header table
  - Segment 1
  - Segment 2
  - Segment 3
  - Segment 4
  - ---

- Section header table
Hello World’s ELF Header

```c
#include <stdio.h>

int main()
{
    char str[] = "Hello World\n";
    printf("%s", str);
}
```

```
$ gcc hello.c -c
$ readelf -h hello.o
```

**ELF Header:**
- Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
- Class: ELF64
- Data: 2's complement, little endian
- Version: 1 (current)
- OS/ABI: UNIX - System V
- ABI Version: 0
- Type: REL (Relocatable file)
- Machine: Advanced Micro Devices X86-64
- Version: 0x1
- Entry point address: 0x0
- Start of program headers: 0 (bytes into file)
- Start of section headers: 368 (bytes into file)
- Flags: 0x0
- Size of this header: 64 (bytes)
- Size of program headers: 0 (bytes)
- Number of program headers: 0
- Size of section headers: 64 (bytes)
- Number of section headers: 13
- Section header string table index: 10
### Section Headers

- Contains information about the various sections

```
$ readelf -S hello.o
```

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Type</th>
<th>Address</th>
<th>Offset Size</th>
<th>EntSize</th>
<th>Flags</th>
<th>Link</th>
<th>Info</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rela.text</td>
<td>RELA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.bss</td>
<td>NOBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rodata</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.comment</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.note.GNU-stack</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.eh_frame</td>
<td>PROGBITS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rela.elframe</td>
<td>RELA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.shstrtab</td>
<td>STRTAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.symtab</td>
<td>SYMTAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.strtab</td>
<td>STRTAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key to Flags:**
- W (write), A (alloc), X (execute), M (merge), S (strings), L (large)
- I (info), L (link order), G (group), T (TLS), E (exclude), X (unknown)
- 0 (extra OS processing required) o (OS specific), p (processor specific)

**Type of the section:**
- PROGBITS: information defined by program
- SYMTAB: symbol table
- NULL: inactive section
- NOBITS: Section that occupies no bits
- RELA: Relocation table

**Virtual address where the Section should be loaded:**
(* all 0s because this is a .o file)

**Offset and size of the section**

**Size of the table if present else 0**
Program Header (executable view)

- Contains information about each segment
- One program header for each segment
- A program header entry contains (among others)
  - Offset of segment in ELF file
  - Virtual address of segment
  - Segment size in file (filesz)
  - Segment size in memory (memsz)
  - Segment type
    - Loadable segment
    - Shared library
    - etc
Program Header Contents

- **type**: type of segment
- **offset**: Offset of segment in ELF file
- **vaddr offset**: Virtual address where the segment is to be loaded
- **paddr offset**: Physical address where the segment is to be loaded (ignored)
- **Size in file image**: Size in file image
- **Size in memory**: Size in memory
- **flags**:
Program headers for Hello World

- `readelf -l hello`

Mapping between segments and sections
Parameters are the path of executable and command line arguments

Get pointer to the inode for the executable

Virtual Memory Map

Executable files begin with a signature. All executables begin with a ELF Magic number string : “\x7fELF”

Set up kernel side of the page tables again!!!

Do we really need to do this?
exec contd.
(load segments into memory)

Parse through all the elf program headers.

Only load into memory segments of type LOAD

Add more page table entries to grow page tables from old size to new size (ph.vaddr + ph.memsz)

Copy program segment from disk to memory at location ph.vaddr. (3\textsuperscript{rd} param is inode pointer, 4\textsuperscript{th} param is offset of segment in file, 5\textsuperscript{th} param is the segment size in file)
exec contd.
(user stacks)

The first acts as a guard page protecting stack overflows

Virtual Memory Map

```c
// Allocate two pages at the next page boundary.
// Make the first inaccessible. Use the second as the user stack.
size = PGROUNDUP(sz);
if ((sz = allocuvm(pgd, sz, sz + 2*PGSIZE)) == 0)
goto bad;
clearpteu(pgd, (char*)(sz - 2*PGSIZE));
sp = sz;
```
exec contd.
(fill user stack)

```c
// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pdgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
} ustack[3+argc] = 0;

ustack[0] = 0xffffffff; // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4; // argv pointer
sp -= (3+argc+1) * 4;
if(copyout(pdgdir, sp, ustack, (3+argc+1)*4) < 0)
  goto bad;
```

```
---
arg N
0
ptr to arg N
...
ptr to arg 1
ptr to arg 0
ptr to 0
argv
0xxffffffff
}
command line args
NULL termination
pointer to command line args (argv)
argc
dummy return location from main
Unused
```
Set the executable file name in proc

```
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(proc->name, last, sizeof(proc->name));
```

these specify where execution should start for the new program.
Also specifies the stack pointer

```
// Commit to the user image.
oldpgdir = proc->pgdir;
proc->pgdir = pgdir;
proc->sz = sz;
proc->tf->eip = elf.entry; // main
proc->tf->esp = sp;
switchuvm(proc);
freevm(oldpgdir);
return 0;
```

Alter TSS segment’s sp and esp.
Switch cr3 to the new page tables.
Exercise

• How is the heap initialized in xv6?
  see sys_sbrk and growproc