

# Operating Systems Concepts: Chapter 6: The System Nucleus

Olav Beckmann

Huxley 449

<http://www.doc.ic.ac.uk/~ob3>

Acknowledgements: There are lots. See end of Chapter 1.

Home Page for the course:

<http://www.doc.ic.ac.uk/~ob3/Teaching/OperatingSystemsConcepts/>

This is only up-to-date after I have issued printed versions of the notes, tutorials, solutions etc.

# Operating Systems as Virtual Machines

An operating system:

- manages a system's resources so that they are used efficiently and safely, *e.g.*,
  - CPU(s)
  - memory
  - devices (modems, disks, network interfaces, video interfaces, *etc.*)
  - Q: Example where efficiency matters? Safety?
- presents a virtual machine that provides convenient abstractions, *e.g.*,
  - files rather than disk locations (device independence)
  - inter-process synchronisation and communication.

*Hardware + OS = Usable Virtual Machine*

# Example: "Virtualising" File Access

```
#include <stdio.h>
#include <assert.h>

#define length 100

int main( int argc, char *argv[] ) {
    FILE *file;
    char buffer[length];

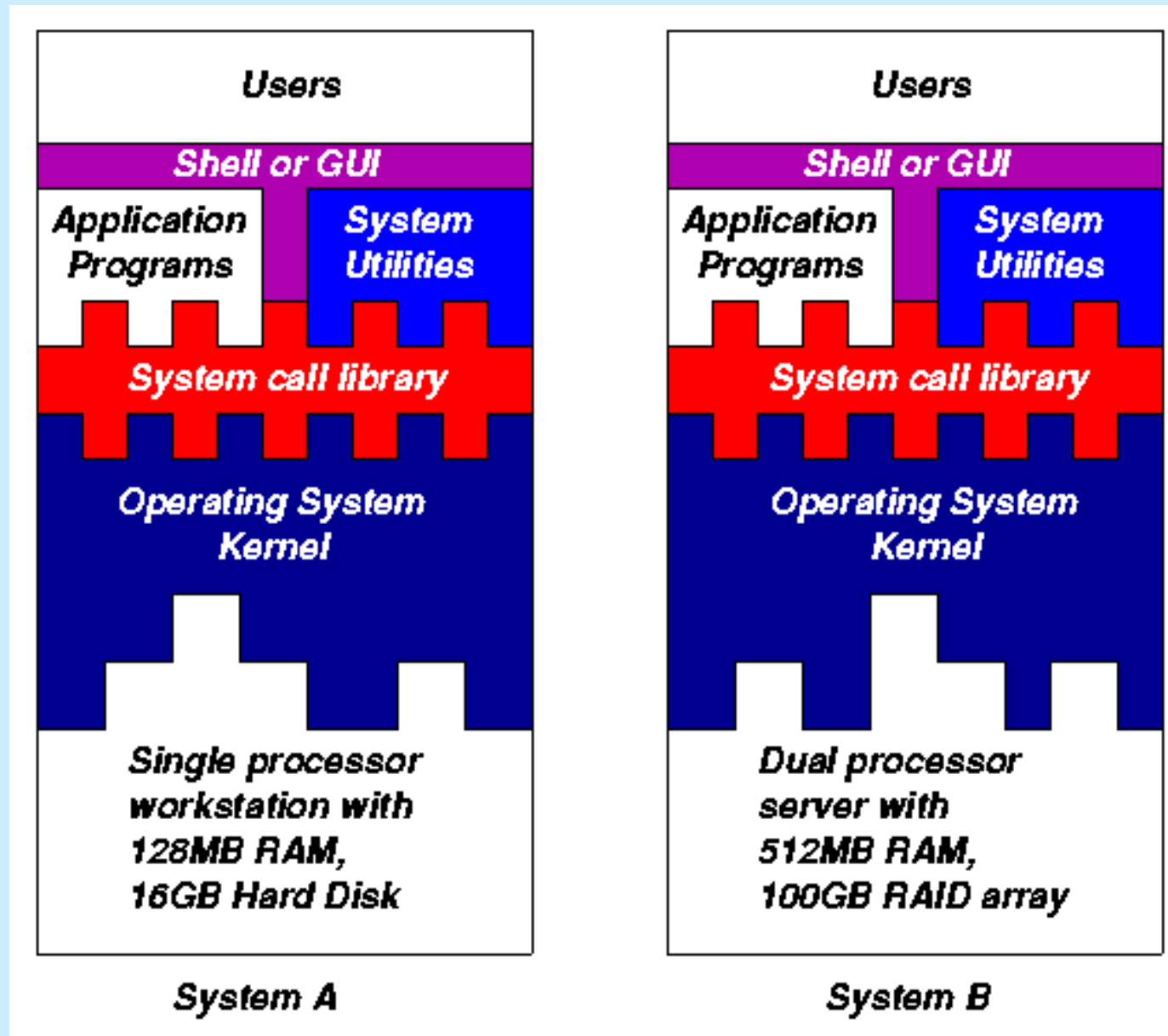
    assert( argc == 2 );
    file = fopen( argv[1], "r" );

    while( fgets(buffer, length, file) ) {
        fputs( buffer, stdout );
    }

    fclose( file );
    return 0;
}
```

- What does this program do?
- Run this program on 3 files:
  - H:\network\_drive.txt
  - C:\hard\_drive.txt
  - E:\usb.txt
- We have illustrated a transparent, uniform interface for accessing files
- *(How does the OS provide this uniform virtual machine?)*

# The Operating System as Virtual Machine



The OS presents the *same interface* and offers the *same services* to users and applications irrespective of the underlying hardware

# Example Revisited: C++ File Reading

```
#include <iostream>
#include <fstream>
#include <assert.h>
#define length 100

int main( int argc, char *argv[] ) {
    char buffer[length];
    assert( argc == 2 );
    ifstream toRead( argv[1], ios::in );

    while( toRead.getLine(buffer, length, '\n') ) {
        cout << buffer << endl;
    }

    toRead.close();
    return 0;
}
```

- **Illustration: Compile both the C and the C++ version under Linux and inspect system calls made with `strace`.**

- Same functionality as previous C program
- Standard libraries:
  - C
  - C++
  - Others
- The OS's virtual machine interface is below these standard libraries
- Linux `strace` allows seeing system calls made by a program

# OS Design Decisions



- **System Calls**
  - Define the interface between OS and application programs, including application program libraries
  - See Tanenbaum Section 1.6
- **Unix: The POSIX standard defines around 100 system calls**
  - More specifically, POSIX defines procedures (functions) which make system calls
  - Examples for POSIX: fork, exec, exit, open, close, read, write
- **Windows: Win32 API defines an interface of 1000s of functions, not all of which invoke OS system calls**
  - Win32 is the published API for writing portable Windows code; underlying system calls may change

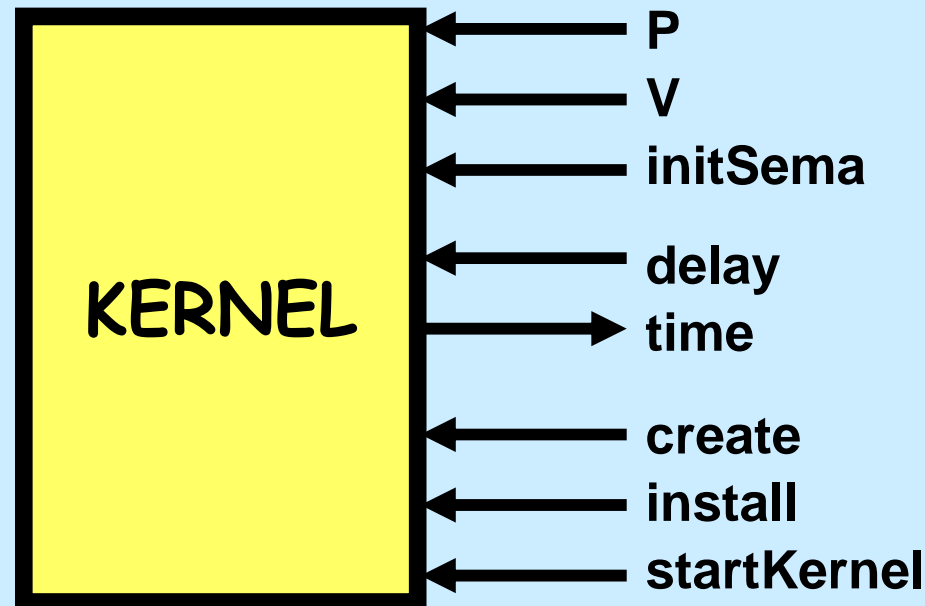
# The System Nucleus (Kernel)

- Shares processor among multiple processes
  - scheduling
  - context switching
- Process management
  - creation, deletion, initiation, termination
  - inter-process synchronisation e.g. P & V, signal & wait
- Synchronisation with real-time
- Interrupt handling
- *Monolithic* kernels additionally include higher-level services, e.g. memory management, I/O, filing system, networking etc.
- *Microkernel* systems delegate higher-level services to servers that run as applications in user mode

Resources (real source code of kernels):

- Simple Kernel: Source code distributed with lab exercises.
- Linux: <http://tamacom.com/tour/kernel/linux/> or /usr/src/linux.

# Simple Kernel Design



This is the system call interface for the Simple Kernel operating system.

Simplifying assumptions made for Simple Kernel:

- 1) Processes are assumed to run forever
- 2) The code for the kernel + user processes is loaded at start-up time

⇒ Suitable for embedded applications.

# Kernel Interface

*Today, kernels are usually written in a high-level programming language, with some low-level assembly code.*

**header “kernel.h” // not a real header**

*/\* process synchronisation: \*/*

**struct** semaphore;

**void** P(semaphore \*s);

**void** V(semaphore \*s);

**initSema**(semaphore \*s, **int** value);

*/\* synchronization with real time: \*/*

**int** time();

**void** delay(**int** ticks),

*/\* system set up: \*/*

**int** priority;

**typedef** void Proc();

**void** create(Proc \*P, **int** size, **int** priority);

**void** install(Proc \*P, **int** intno);

**void** startKernel();

**create(P, size, pr)** creates a new process executing function *P* with workspace *size* and priority *pr*.

**time()** returns clock ticks since kernel was started.

**delay(t)** suspends a process for *t* clock ticks.

**install(P, intno)** makes function *P* the handler for interrupt *intno*, so that *P* is invoked whenever that interrupt occurs.

**startKernel()** starts processes running under kernel control.

```

module "asterisk.c"
#include "kernel.h"
// clock interrupts every 20 ms
const int ticksper10s = 500;
const int clockvector = 32;
int count; // incremented every tick

```

```
Semaphore sync;
```

```

void tick() {
    count = (count+1) % ticksper10s;
    if (count == 0) {
        V(&sync);
    }
}

```

```

void print() {
    while (1) {
        P(&sync);
        put("*");
    }
}

```

```

void main() { /* main routine starts here */
    count = 0;
    int normal = 2;
    initSema(sync, 0);
    install(tick, clockvector);
    create(print, 500, normal); // normal is a priority level
    startKernel();
}

```

## Example:

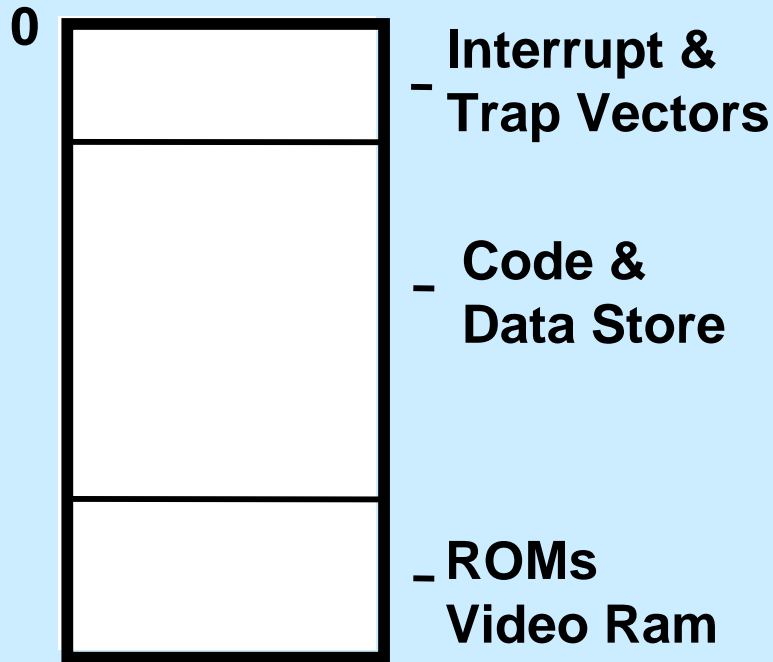
### Program to output an asterisk every 10 seconds

*Two phases of execution -*

- 1. Initialization (creates process, installs interrupt handler)*
- 2. startKernel to run process.*

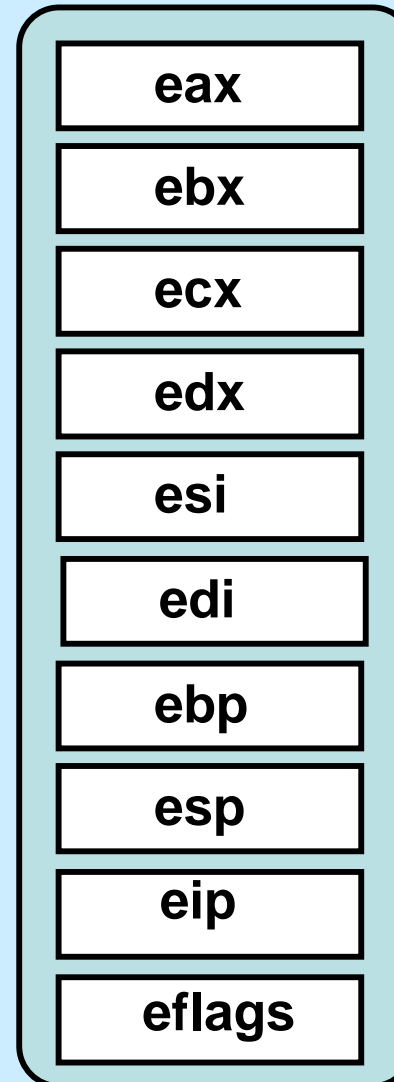
# Target Architecture: Intel 386

## Memory Map



32 bit words  
Byte addressed

## Registers



*We will come back to this later when we do interrupts!  
At this stage, note:*

- **extended 32-bit registers**
- **the eflags register**

General Registers

Frame Pointer

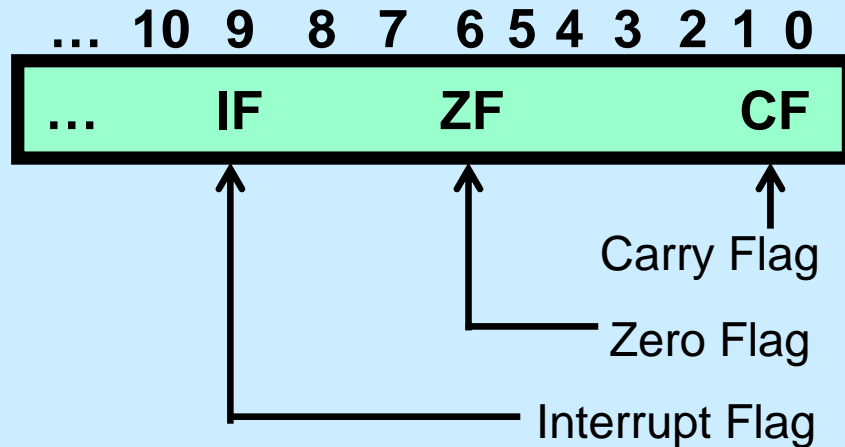
Stack Pointer

Program Counter  
(Instruction Pointer)

Processor  
Status Word

# Enabling and Disabling Interrupts on Intel i386

## Flags Register (Status Word)



**Interrupt Flag: when set (equal to 1) interrupts are enabled otherwise they are disabled.**

**The only way to change the IF flag is through the following instructions:**

**cli** clear IF (disable interrupts)  
**sti** set IF (enable interrupts)  
**pushf** put flags on stack  
**popf** set flags from top of stack

- The **eflags** register carries “state” (information) between instructions
  - Zero flag – whether the result of a comparison instruction was zero; can be used by **subsequent** branch.
  - Carry flag – can be used to do “add with carry”.
- The **eflags** register controls whether interrupts are enabled.
- For reasons of instruction set design, the **eflags** register cannot be directly addressed by assembler instructions.

# Kernel Exit & Entry

To ensure mutual exclusion to the kernel data structures, kernel procedures must run with interrupts disabled. Only one process can be in the kernel at a time.

```
int int_mutexon()  
void int_mutexoff( int psw )
```

disables interrupts, result is old value of eflags reg.  
restores eflags register to value provided in psw

## Assembler:

Assembler generated by gcc uses the accumulator register (or extended accumulator %eax) to return the result of some function calls. **DON'T DO THIS AT HOME ☺.**

```
.globl int_mutexon      .globl int_mutexoff  
int_mutexon:           int_mutexoff:  
    pushf                pushl 4(%esp)  
    cli                  popf  
    → popl %eax          ret  
    ret
```

Kernel access procedures will have the following form:

```
void ... (...){  
    int psw = int_mutexon();    // disable interrupts  
    :  
    :  
    int_mutexoff(psw);         // restore interrupts  
                                // to previous state  
}
```

Why can't int\_mutexoff simply enable interrupts?  
Hint: what happens if one kernel procedure calls another?

# Kernel Exit & Entry (continued)

For convenience we will use the following macros:

```
#define ENTERKERNEL
```

```
    int psw; psw = int_mutexon();
```

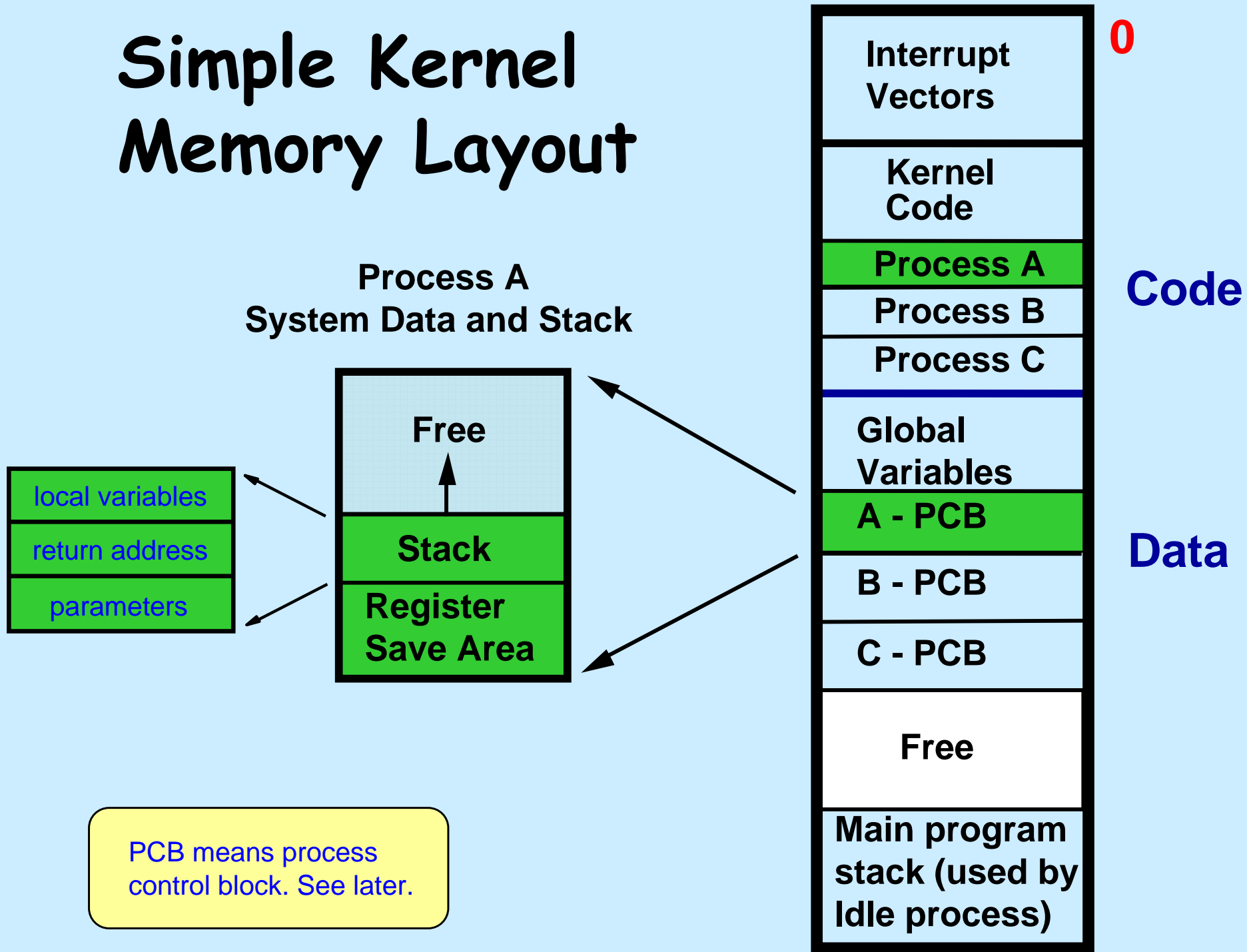
```
#define EXITKERNEL
```

```
    int_mutexoff(psw);
```

See `include/icos/intP.h` in Simple Kernel source code.

The assembler routines `int_mutexon` and `int_mutexoff` (see previous slide) can be found in `int_asm.S`.

# Simple Kernel Memory Layout

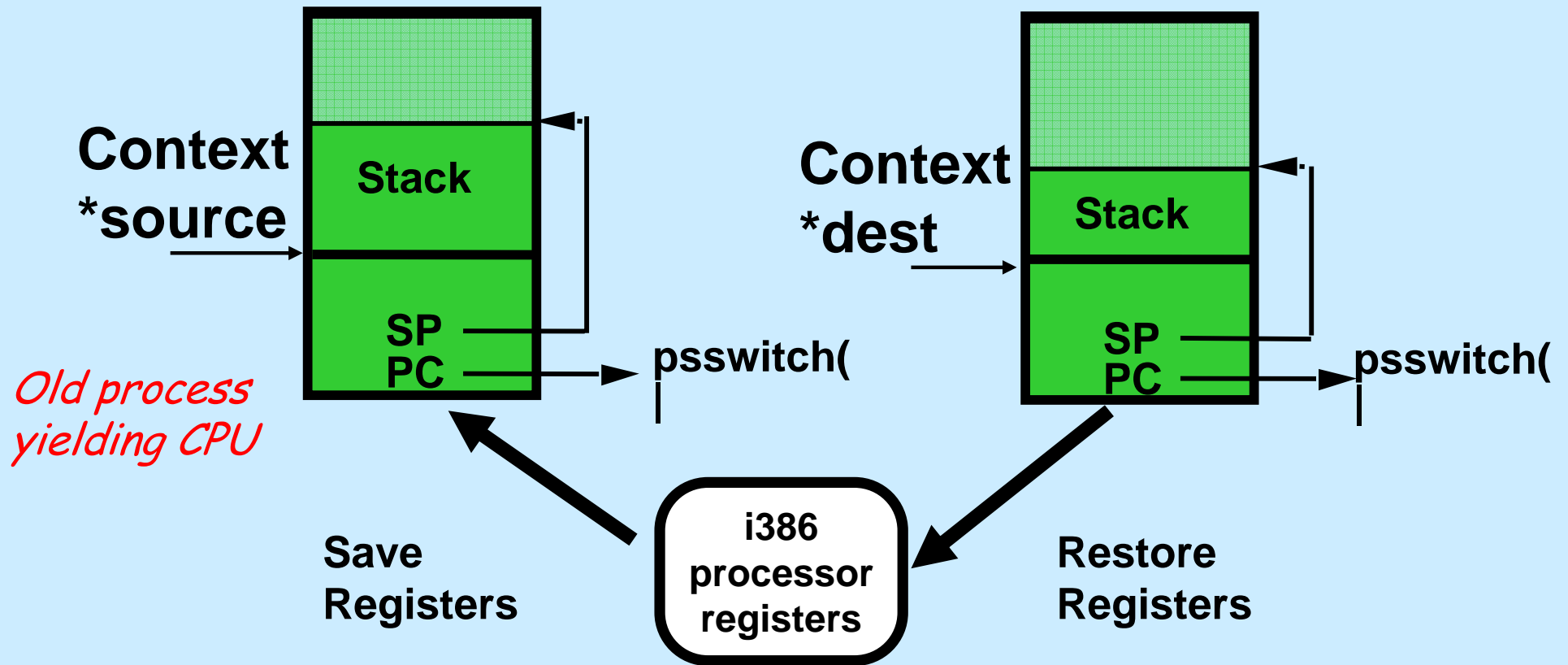


PCB means process control block. See later.

# Context Switch

We will come back to this in more detail.

*New process acquiring CPU*

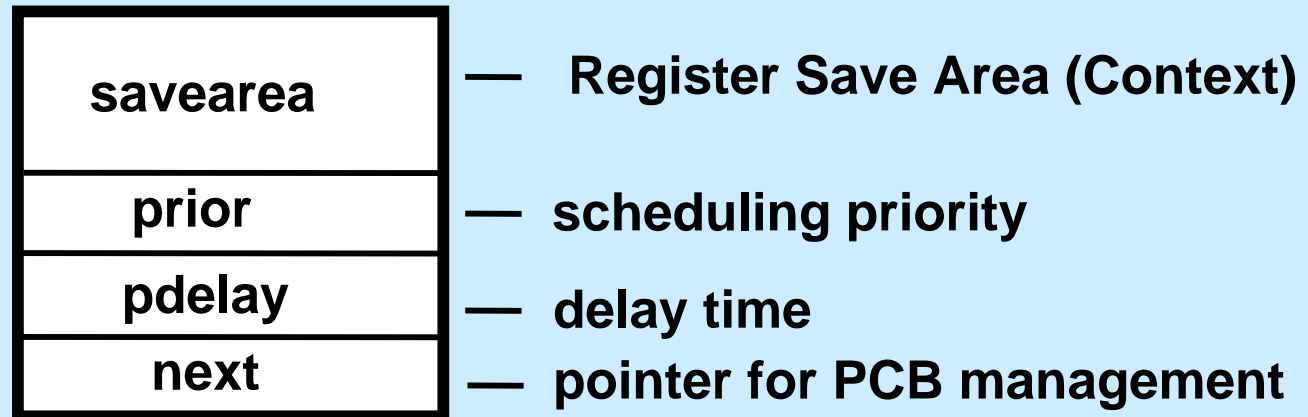


```
typedef struct Context { // Data structure for holding i386 registers in memory
    int eax, ebx, ecx, edx, esi, edi, ebp, eflags, eip, esp;
    /* 0 4 8 12 16 20 24 28 32 36 note these offsets*/
} Context;
```

```
void psswitch(Context *source, Context *dest);
```

# Kernel Representation of Processes

## Process Control Block

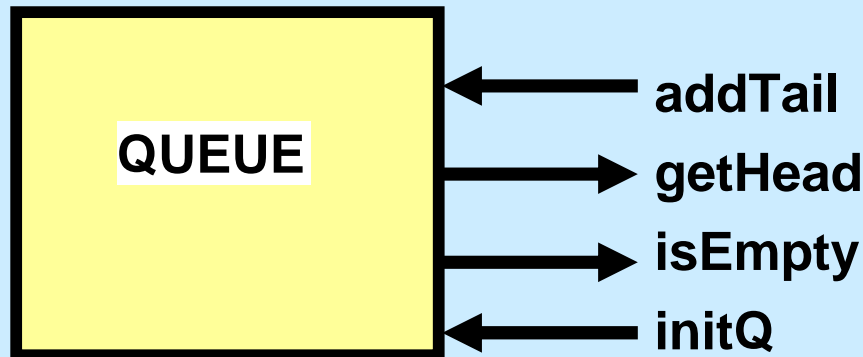


```
typedef int Priority;  
typedef PCB *Process;  
typedef struct PCB {  
    Context savearea;  
    Priority prior;  
    int pdelay;  
    Process next;  
} PCB;
```

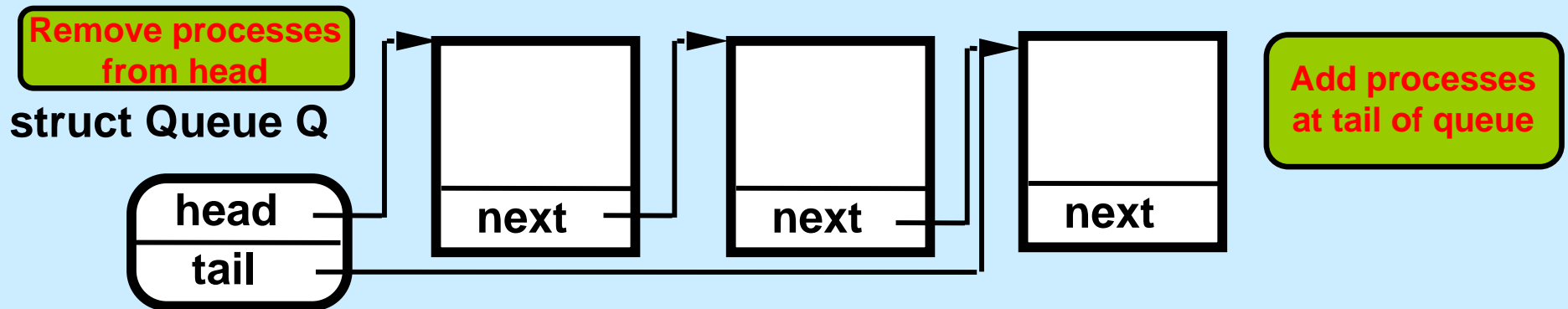
Processes are managed by forming "linked lists" or queues of process control blocks.

In linux, a process is represented by a *struct task\_struct*, defined in include/linux/sched.h

# Kernel FIFO Queue Operations



```
typedef struct Queue {  
    Process head, tail;  
} Queue;
```



Advantages of this data structure?

# Queue Implementation

```
bool isEmpty(Queue *Q) {  
    return (Q->head == NULL);  
}
```

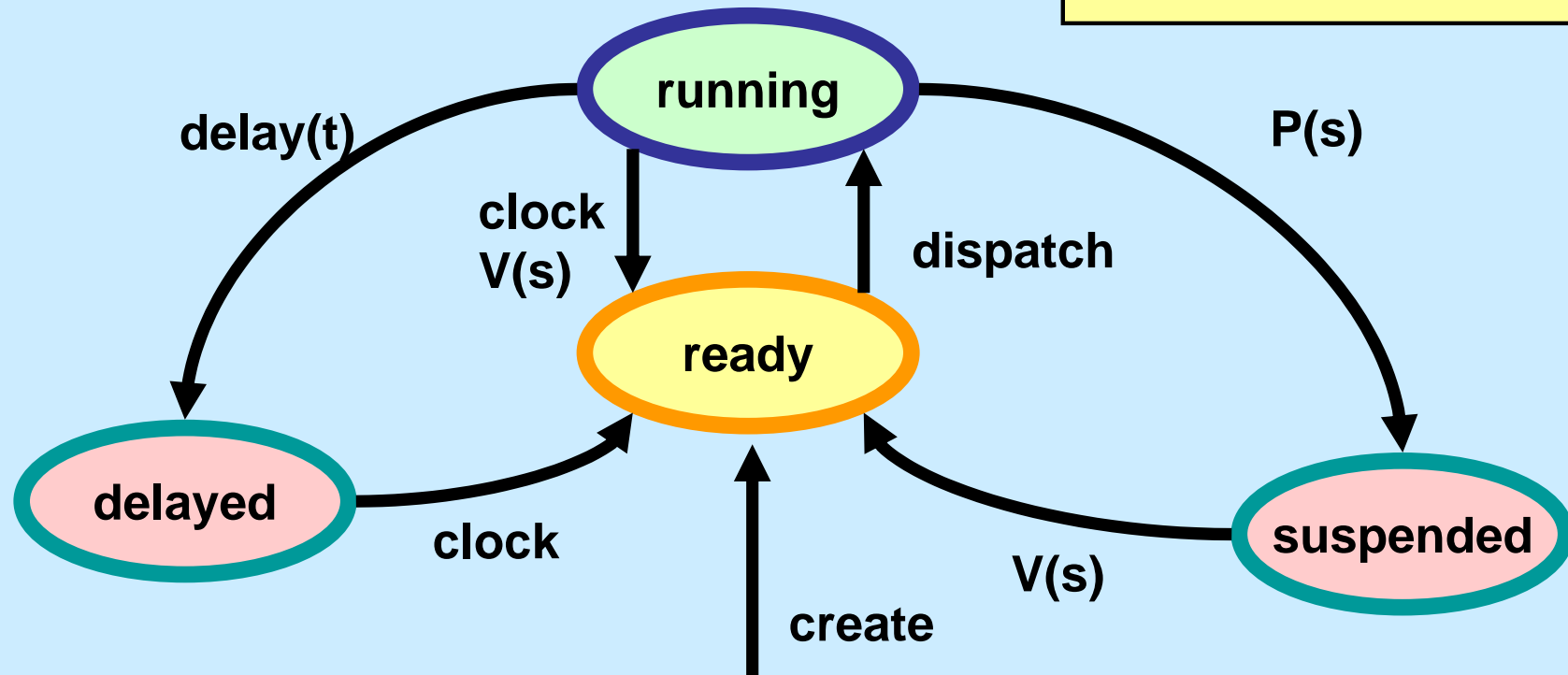
```
void addTail(Queue *Q, Process p) {  
    if (Q->head == NULL)  
        Q->head = p;  
    else  
        Q->tail->next = p;  
    Q->tail = p;  
    p->next = NULL;  
}
```

```
Process getHead(Queue *Q) {  
    Process p = Q->head;  
    if (Q->head != NULL)  
        Q->head = Q->head->next;  
    return p;  
}
```

```
void initQ(Queue *Q) {  
    Q->head = NULL;  
    Q->tail = NULL;  
}
```

# Process State Transitions

*Delayed and suspended are both blocked states.*



A process will be in exactly one of these states. The kernel maintains data structures to keep track of the processes in each state:

A **ready** process - is on one of the four ready queues, readyQ[priority] (see below)

A **delayed** process - is on delayQ

A **suspended** process - is on the queue associated with a semaphore S

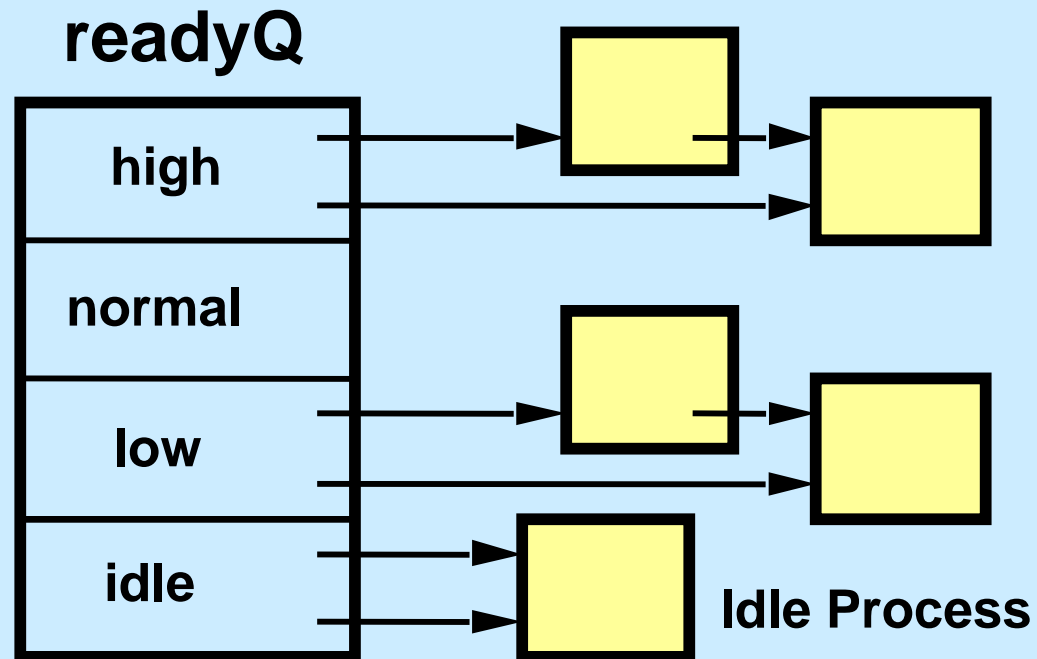
The **running** process - is pointed to by the variable running  
(Just one processor, so only one process at a time in the running state.)

# Ready Queue - Static multi-level priority

Actually *four* ready queues, one for each priority.

There must always be a process available to run.

To ensure this the kernel provides an **idle** process at the lowest priority level which can be run when there are no other ready processes.

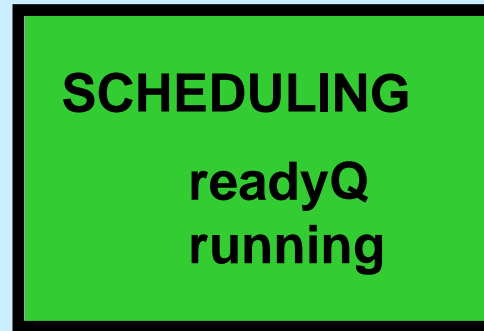


Process running; *//points to currently running process*  
Queue readyQ[4]; *//one queue for each priority level*

```
enum Priority {high = 3,normal = 2,low = 1,idle = 0};
```

# Scheduling Operations

Selecting a process to run



```
void setReady( Process p ) {  
    addTail( readyQ[p->prior], p );  
}
```

```
void dispatch() {  
    Process oldrunning = running;  
    int pr = 3; // high  
    while( isEmpty(readyQ[pr]) ) {  
        pr--; // gets next lower priority  
    }  
    running = getHead( readyQ[pr] );  
    psswitch(oldrunning->savearea,  
            running->savearea );  
}
```

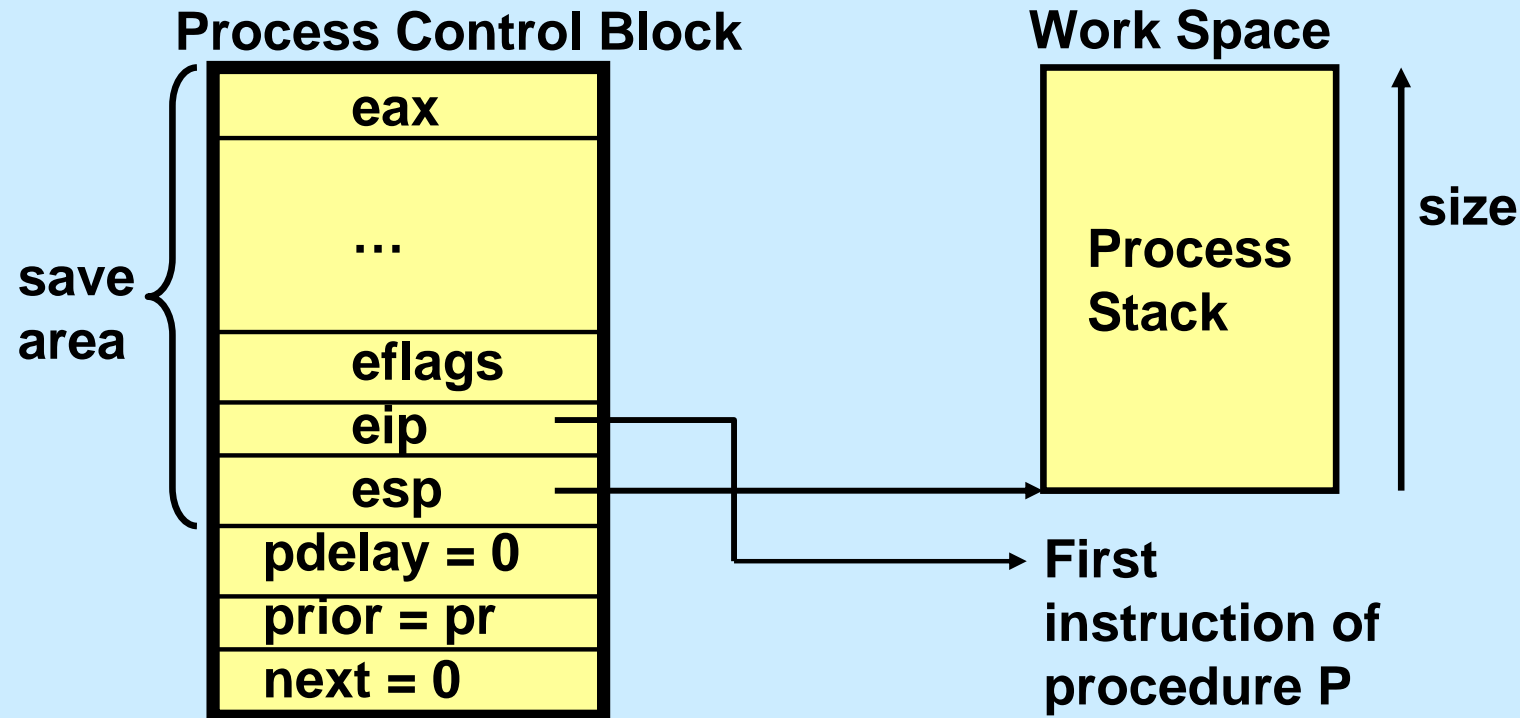
```
void idle_loop() {  
    while (1) {  
        // idling loop---loop forever  
        ; // Question: how does anything  
        // else ever run?  
    }  
}
```

*Functionality of  
initSched is done  
by proc\_init and  
startKernel in  
Simple Kernel*

```
void initSched() {  
    for (int pr=0; pr<4; pr++)  
        InitQ(readyQ[pr]);  
  
    create(idle_loop, 0, idle);  
    running = getHead(readyQ[idle]);  
}
```

# Process Creation

Processes are created in the following state:



The existence of a low-level memory manager is assumed which supplies the function

```
int alloc(int size);
```

*alloc(n)* returns the address of a chunk of free memory *n* bytes in size.

# Process Creation Implementation

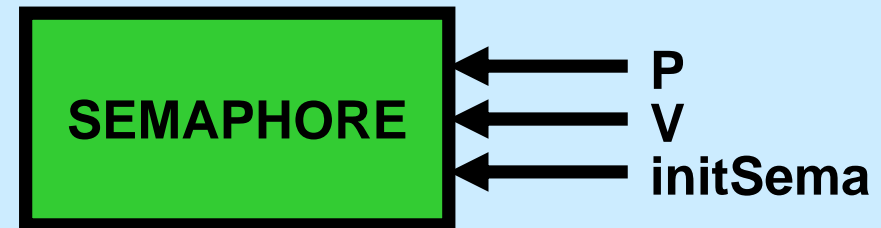


```
typedef void (*Proc) ( void );

void create(Proc *P, int size, Priority pr) {
    ENTERKERNEL
    Process newp = new PCB;
        // get a new Process Control Block for the new process
    newp->savearea.eip = P;                // set the initial value of the process's
                                           // PC to be the start of P
    newp->savearea.esp = alloc(size)+size; // set SP to end of new work space
    newp->savearea.eflags = 512;          // will run with interrupts enabled
    newp->prior = pr;
    newp->pdelay = 0;
    setReady(newp);
    EXITKERNEL
}
```

# Semaphore Implementation

```
typedef struct Semaphore {  
    int count;  
    Queue waiting;  
} Semaphore;
```



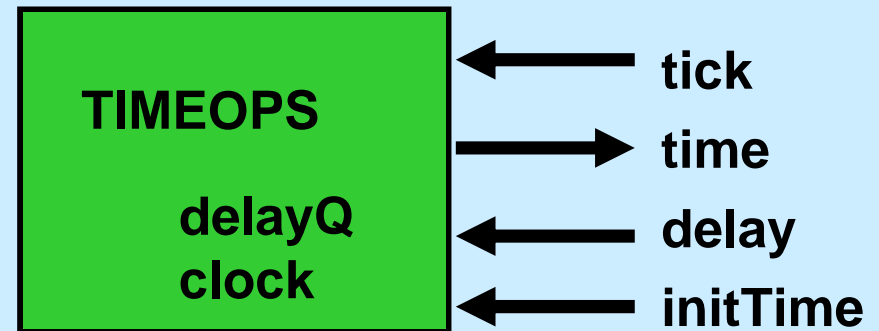
```
void P(Semaphore *s) {  
    ENTERKERNEL  
    if (s->count > 0)  
        s->count--;  
    else {  
        addTail(s->waiting, running);  
        dispatch();  
    }  
    EXITKERNEL  
}
```

```
void V(Semaphore *s) {  
    ENTERKERNEL  
    if (isEmpty(s->waiting))  
        s->count++;  
    else {  
        setReady(getHead(s->waiting));  
        setReady(running);  
        dispatch();  
    }  
    EXITKERNEL  
}
```

```
void initSema(Semaphore *s, int value) {  
    ENTERKERNEL  
    s->count = value;  
    initQ(s->waiting);  
    EXITKERNEL  
}
```

# Time Operations

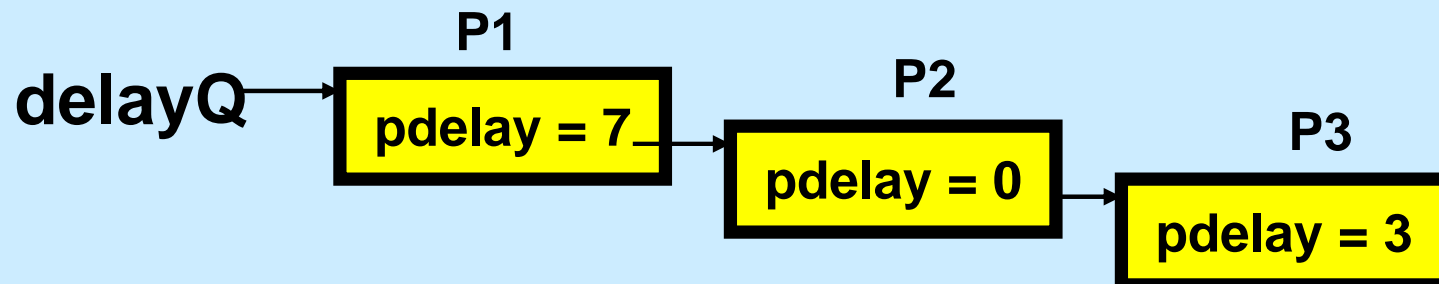
```
Process delayQ;  
int clock;
```



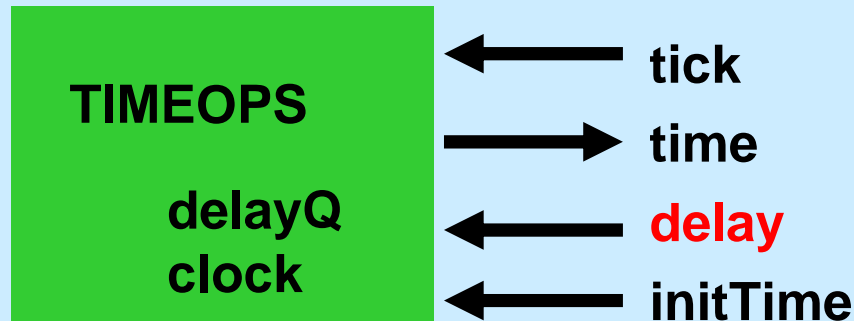
The delayQ is not a FIFO queue. We will use a “delta” queue where the delay time of a process in the queue is the sum of its *pdelay* field and the *pdelay* fields of those processes that precede it in the queue.

## Example:

At time 100 P1 executes delay(7);  
At time 100 P2 executes delay(7);  
At time 100 P3 executes delay(10);



# Delay

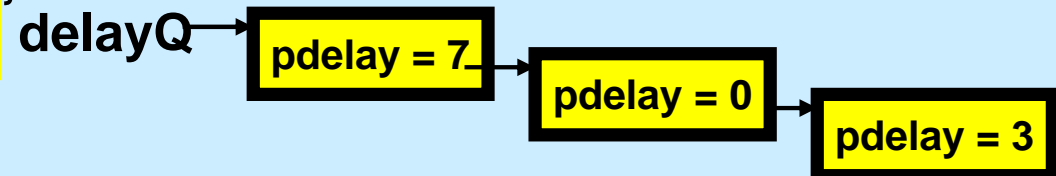


Loop over *delayQ* to find the correct place to add the process. As *delayQ* is traversed, *t1* is adjusted to be the time the process must wait after its predecessor is woken up.

*p0* is the current position in *delayQ*  
*p1* is next process in *delayQ*

If there is another process, *p1*, after the newly added process, then its *pdelay* must be adjusted.

```
void delay(int t) {  
    ENTERKERNEL  
    if (t > 0) {  
        int t1 = t;  
        Process p0 = delayQ, p1 = delayQ;  
        while (p1 != NULL && t1 >= p1->pdelay) {  
            t1 -= p1->pdelay;  
            p0 = p1;  
            p1 = p0->next;  
        }  
        if (p0 == p1) /* == delayQ */  
            delayQ = running;  
        else  
            p0->next = running;  
            running->next = p1;  
            running->pdelay = t1;  
            if (p1 != NULL) {  
                p1->pdelay -= t1;  
            }  
        dispatch();  
    }  
    EXITKERNEL  
}
```



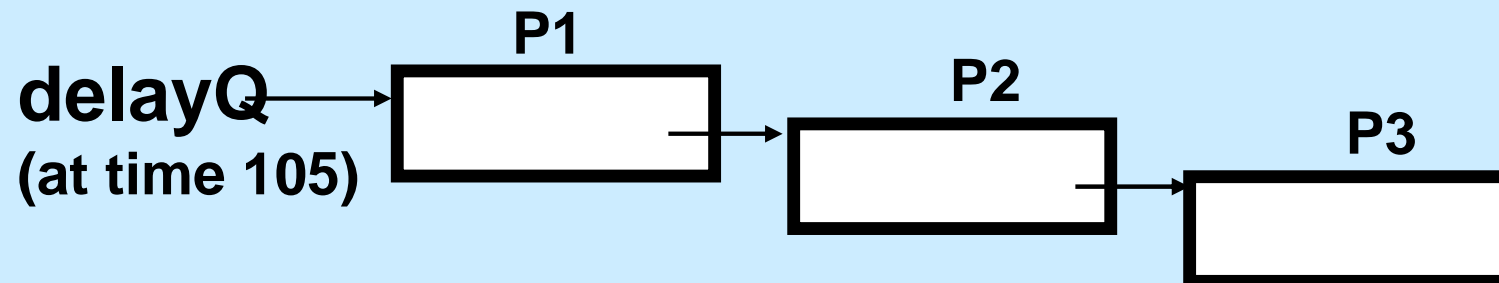
# Time Implementation

```
int time() {  
    // return current time  
    return clock;  
}  
  
void initTime() {  
    clock = 0;  
    delayQ = NULL;  
    // set up Tick to be called  
    // every clock interrupt.  
    // 32 is the clock interrupt  
    // vector on the Intel 386  
    install(tick, 32);  
}
```

```
void tick() {  
    // Update current time  
    clock++;  
    // wake up any processes whose delays have expired  
    if (delayQ != NULL) {  
        delayQ->pdelay--;  
        Process p = NULL;  
        while (delayQ != NULL && delayQ->pdelay == 0) {  
            // loop setting ready all processes with  
            // pdelay = 0, or until delayQ is empty  
            p = delayQ;  
            delayQ = p->next;  
            setReady(p);  
        }  
        // reschedule  
        setReady(running);  
        dispatch();  
    }  
}
```

# Exercise

At time 95 P1 executes delay(20);  
At time 100 P2 executes delay(30);  
At time 105 P3 executes delay(35);

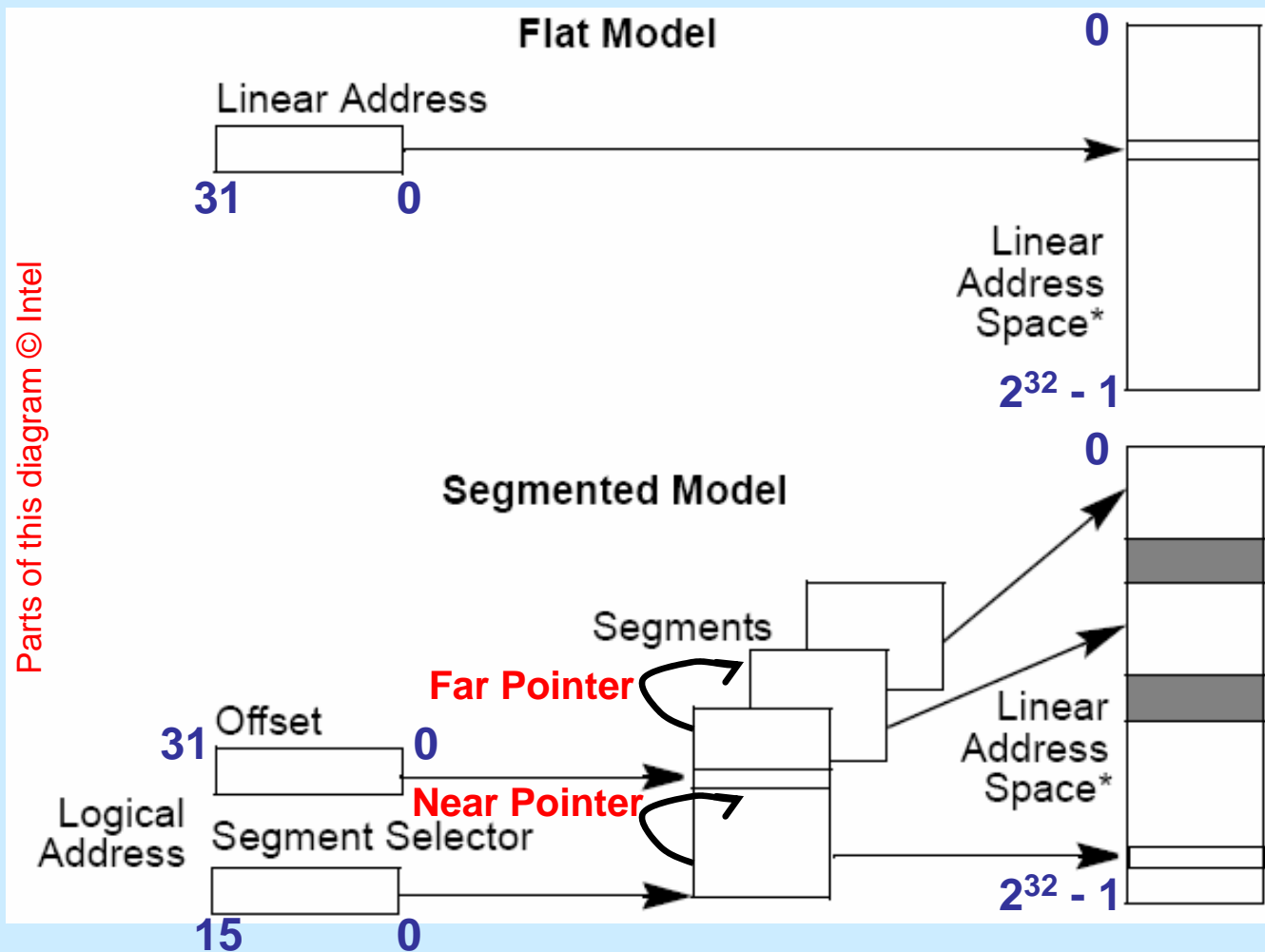


What is the advantage of this organisation?

Answer: efficiency---only the process at the head of the queue needs to be examined by the interrupt handler on each clock tick.

# Chapter 6b: Simple Kernel Interrupts

- Begin with a brief overview of memory addressing (more in Chapter 7)



- Intel i386 architecture uses byte-addressing
- Processor provides for different addressing modes
- **Linear: 32-bit address**
- **Segmented: 16-bit segment register plus 32-bit offset**
- **Segmented provides better protection (see Chapter 8).**

- Typically kernel code (including interrupt handlers) will be in a different segment from user code.

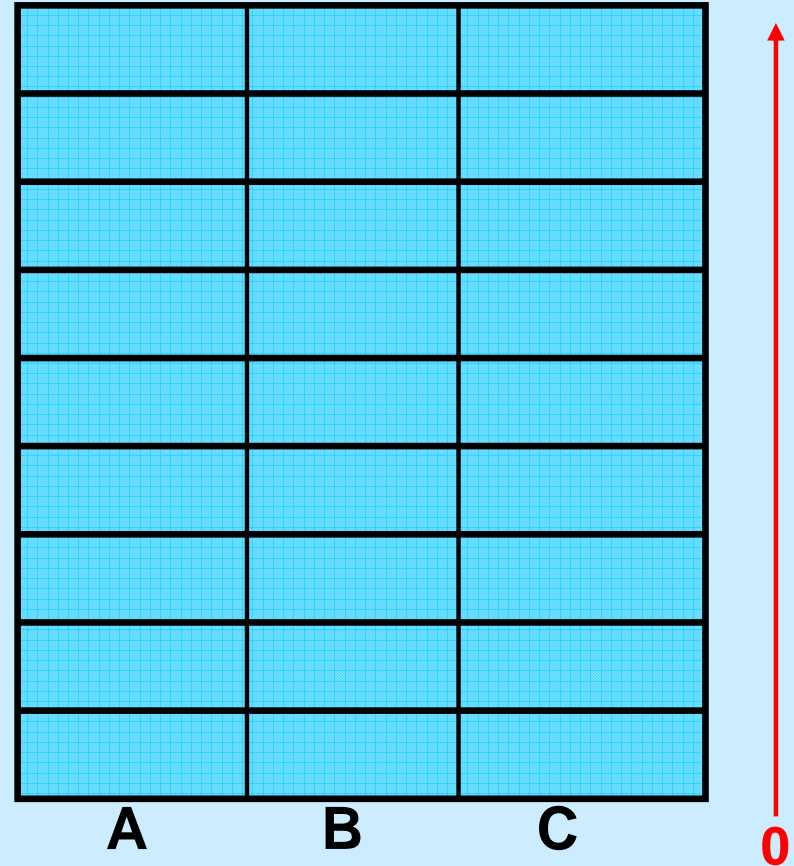
# Recap: Function Call and Return

- Key registers
  - `eip`: program counter (“instruction pointer”) Cannot be directly manipulated
  - `esp`: stack pointer
  - `ebp`: frame pointer (“base pointer”)
- `call f` instruction
  - passing parameters has to be done by caller **before** executing `call` instruction
  - pushes current value of `eip` onto stack (the address of the next instruction after the `call` instruction)
  - branches to address given by target operand to `call`
- `ret` instruction
  - transfers control back to the address at the top of the stack

# Example: Function Call and Return

```
extern int N[];
int f( const int a ) {
    int b[10];
    int i;
    for(i = 0; i < a; i++) {
        b[i] += i + N[i];
    }
    return b[i];
}
```

← before call f (A)  
 ← Here (B)  
 ← Here (C)



```
.text
.global f
.type f, @function
f:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $52, %esp
    movl 8(%ebp), %ebx
```

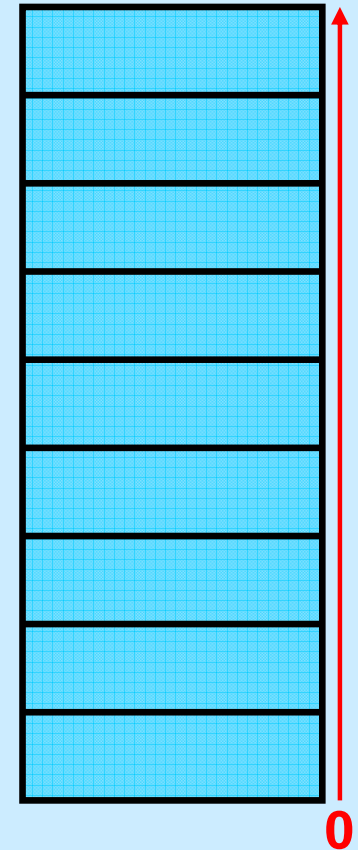
- Draw the state of the stack at the three points (A, B, C) in the execution of f.
- Assembler produced with Intel C compiler (icc) v 7.1.

# Another Example: Simple PIN Check

```
#define MAX_PIN 4
const int secret_pin[] = { 1, 2, 3, 4 };

int check_pin( const int position, const int value ) {
    int pin[MAX_PIN];
    pin[position-1] = value;
    return pin[position-1] == secret_pin[position-1];
}

int main( int argc, char *argv[] ) {
    printf( "Hello. Checking your pin now. \n" );
    if( ! check_pin( atoi( argv[1] ), atoi( argv[2] ) ) ) {
        printf( "You have entered a wrong digit. \n" );
        exit( -1 );
    }
    Success:
    printf( "You have got in. What do you want to do?\n" );
    printf( "Success = %d\n", (int) &&Success );
    return 0;
}
```



# Function Calls and Interrupts

- Function calls happens at known locations in the code, where the programmer intended. We generally call functions whose code resides in the same code segment.
- **Interrupts can occur at any point in the execution path.**
  - Must make sure that this is transparent to the executing code when the interrupt handler returns.
  - The **eflags** register is an “implicit result” of some other instructions – program state that carries information between instructions. Interrupt handlers need to preserve its state.
- We need to make sure that none of the general-purpose registers are overwritten either.
  - Need to save on stack and restore.
- Interrupts generally result in executing kernel code (i.e. likely to be code which is stored in a different code segment)

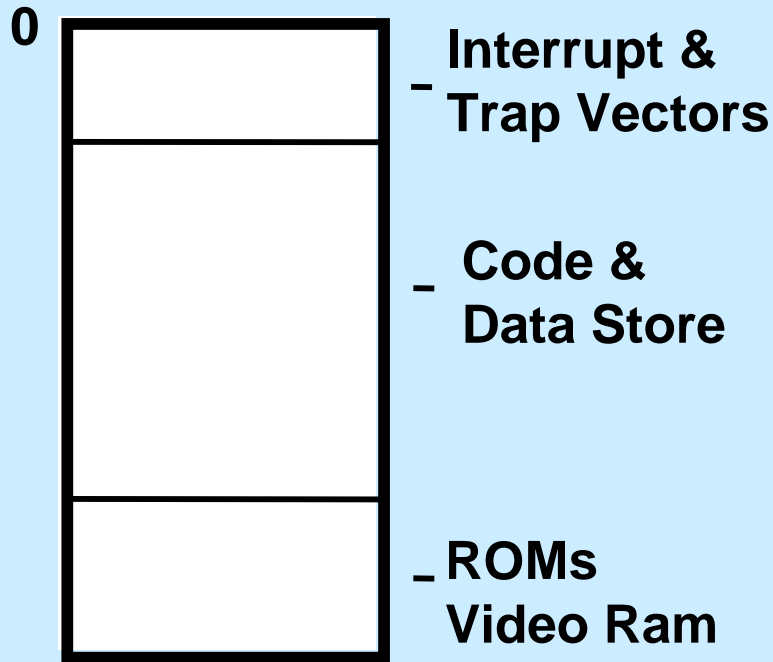
# Recap:

## Target Architecture: Intel 386

*Note:*

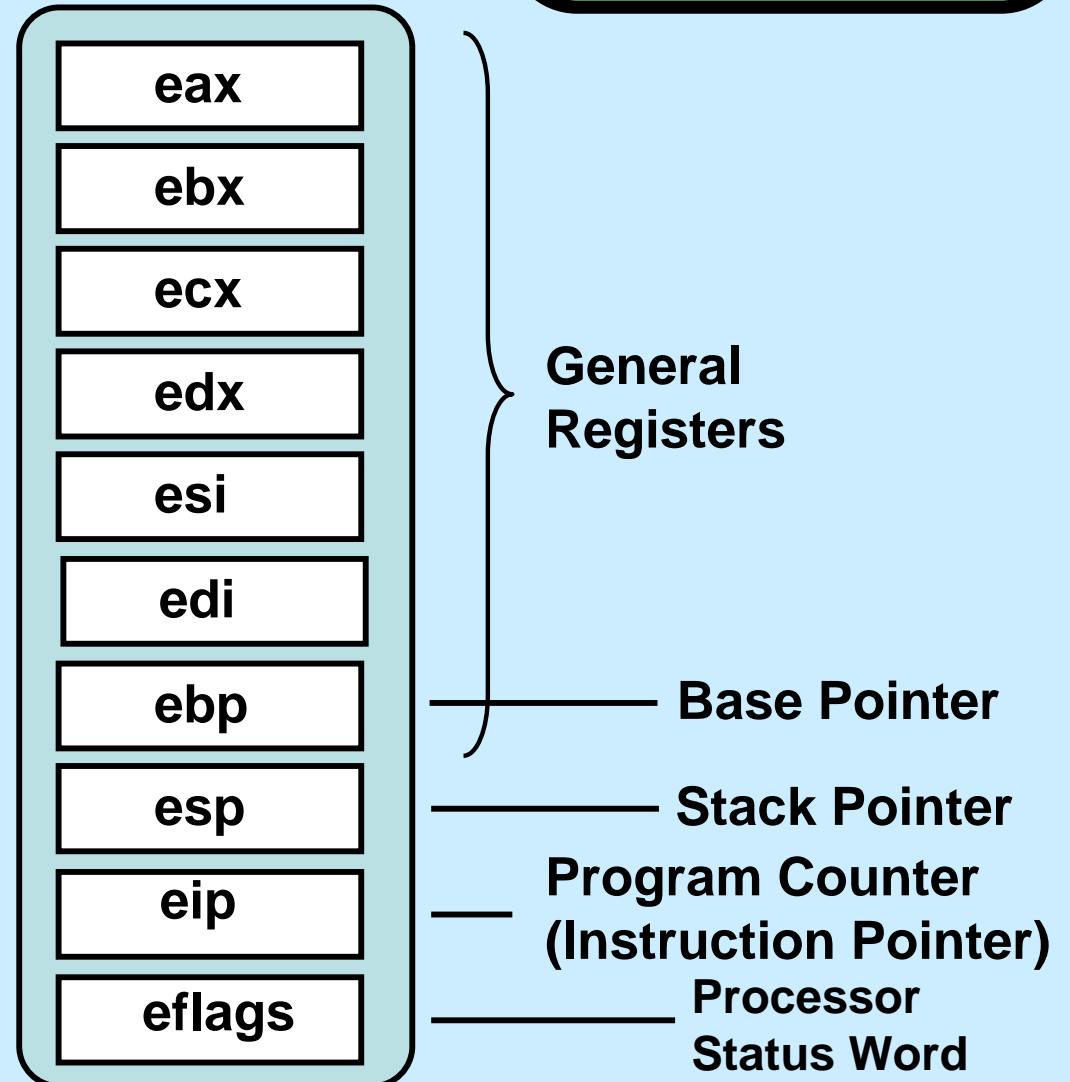
- extended 32-bit registers
- the eflags register

### Memory Map



32 bit words  
Byte addressed

### Registers



# Vectored Interrupts on the 386

## eflags Register (Status Word)

11 10 9 8 7 6 5 4 3 2 1 0

OF DF IF TF SF ZF AF PF CF

IF - Interrupt Flag. When set interrupts are enabled otherwise they are disabled.

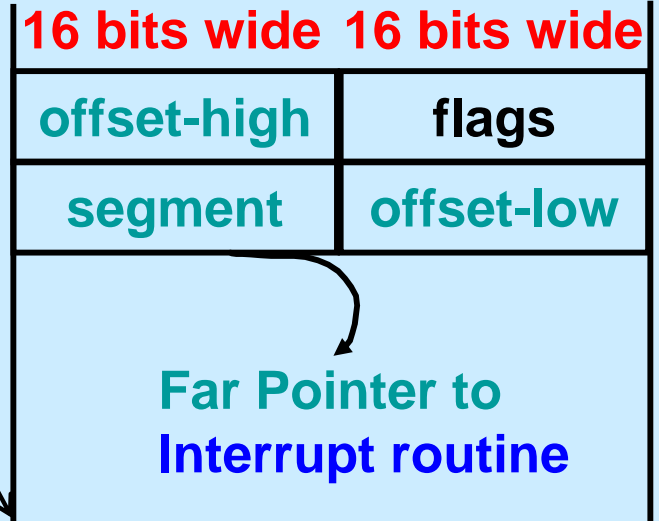
The only way to manipulate eflags:

**cli** clear IF (disable interrupts)  
**sti** set IF (enable interrupts)  
**pushf** put flags on stack  
**popf** set flags from top of stack

## Interrupt Descriptors

Interrupt descriptor N  
 (N= 0 to 255)  
 (8 bytes)

idtr register holds location of interrupt descriptor table  
 (init32.S)



- Interrupting device - signals interrupt, puts vector number N (8 bits) on CPU pins.  
 CPU hardware then - pushes eflags register, code segment and PC (eip) onto stack  
 - CPU also clears IF bit in eflags - disabling interrupts  
 - the new PC (eip) is given by a far pointer (segment and offset) in entry number N of the interrupt descriptor table.  
 - ("flags" in interrupt descriptor do NOT correspond to eflags)
- At end of interrupt routine, **iret instruction pops eip, CS and eflags** from stack.

# Interrupts First level Interrupt Handler, int\_flih

- We would like to use a high-level language to write interrupt routines.
- The actual “first-level” interrupt handler is a general interface for the compiled high-level routines.
- It saves all the registers, pushes the interrupt number onto the stack and calls our interrupt routine.
- When that returns we restore the registers and execute an **iret** instruction.

## Vector

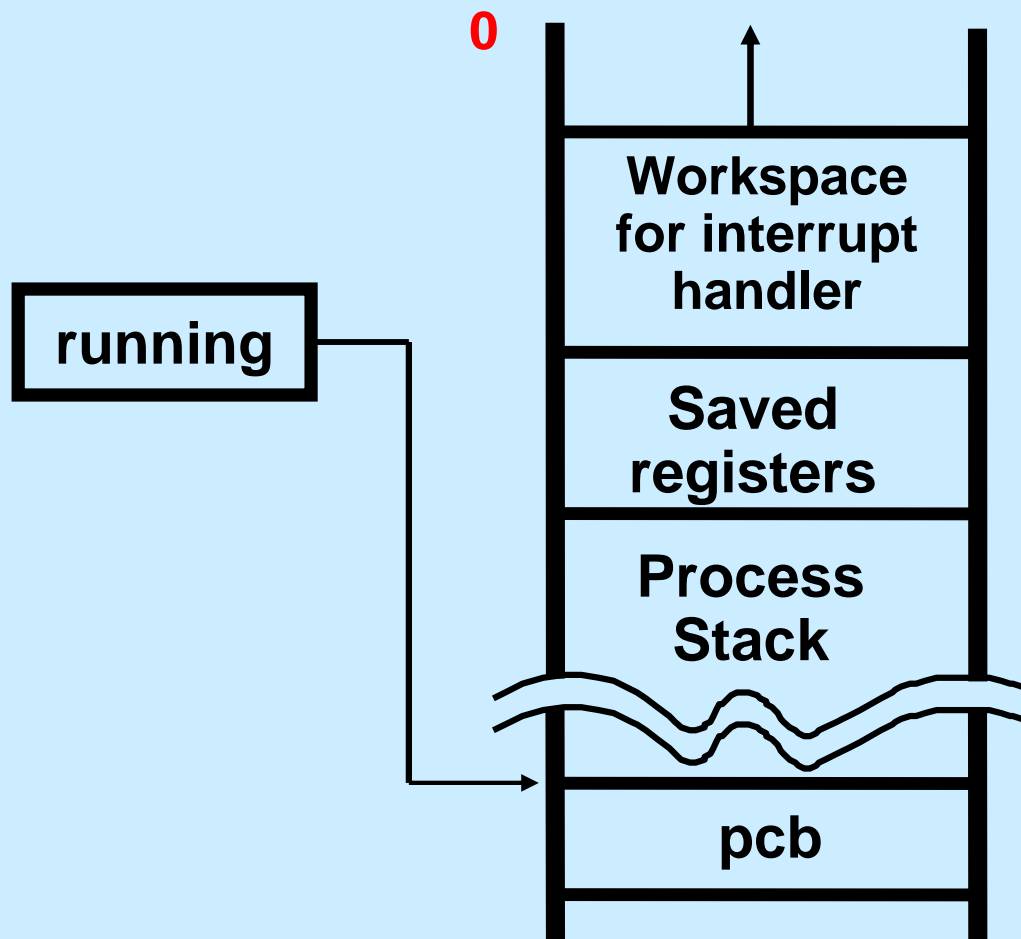
16 bits wide		16 bits wide	
offset-high	flags		
segment	offset-low		

Linux: see arch/i386/kernel/irq.h  
and include/asm-i386/irq.h

```
#define INT_FLIH_CPU(inum) ;\
.align 4 ;\
.globl int_flih_##inum ;\
int_flih_##inum##:\
    push %es ; Segment \
    push %ds ; Registers \
    pushl %eax ;\
    pushl %ebx ;\
    pushl %ecx ;\
    pushl %edx ;\
    pushl %esi ;\
    pushl %edi ;\
    pushl %ebp ;\
    pushl $$$inum ;\
    call int_interrupt ;\
    addl $4,%esp ;\
    popl %ebp ;\
    popl %edi ;\
    popl %esi ;\
    popl %edx ;\
    popl %ecx ;\
    popl %ebx ;\
    popl %eax ;\
    pop %ds ;\
    pop %es ;\
    iret ;\
```

# Interrupts (cont.)

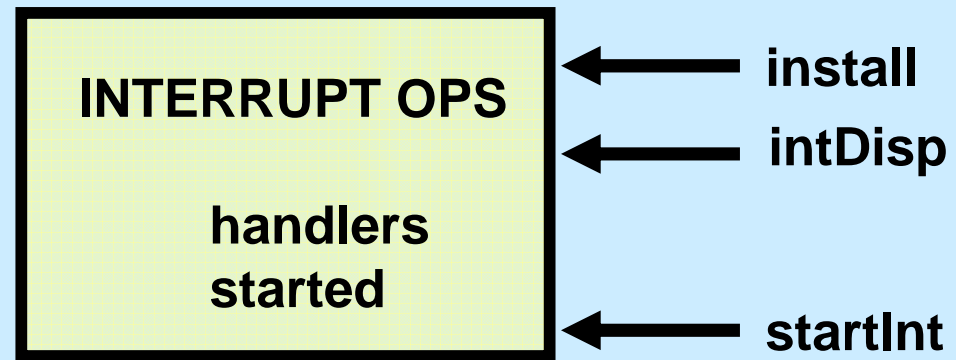
- The assembly code routine `int_flih_X` saves and restores the registers since these may be used by the interrupt handler procedure.
- *Interrupt handlers are not processes.* In Simple Kernel, they use the stack of the currently running process.



Stack of a process which has been interrupted (must be the currently running process) in **Simple Kernel**.

**Q: Any possible issues?**

# Interrupt Operations



```
typedef void Proc( void );  
Proc *handlers[256];      // Interrupt vectors are 8-bit, i.e. up to 256  
bool started = false;
```

```
void startInt() {  
    started = true;      // allow system to respond to interrupts  
}
```

```
void install(Proc *P, int vector) {  
    ENTERKERNEL  
    handlers[vector] = P;  
    // must also initialize interrupt vector (machine dependent)  
    EXITKERNEL  
}
```

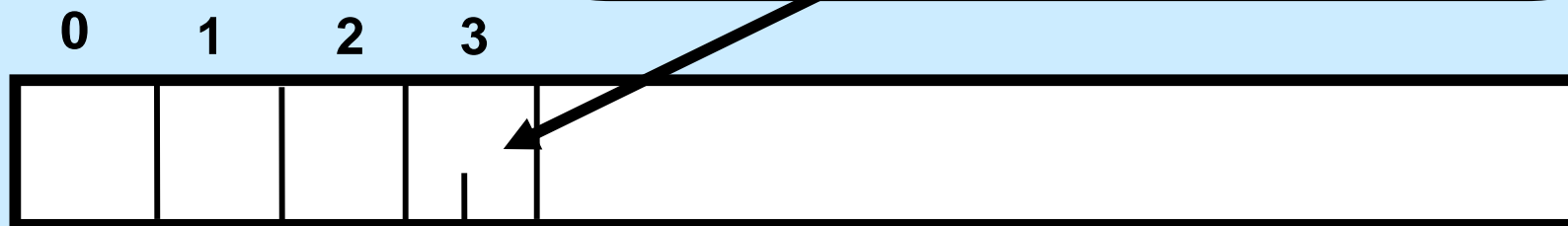
*The point of this architecture is to allow us to write interrupt handlers in C.*

# Interrupt operations (cont.)

## Descriptor



## handlers



```
.align 4
.globl int_flih3
int_flih_3:
    push %es; push %ds; pushl %eax
    ...
    pushl %ebp
    pushl $3 ; pass int. num. (3) as param
    call int_interrupt
    addl $4,%esp
    popl %ebp
    ...
    popl %eax;
    pop %ds;
    pop %es
    iret
```

```
void int_interrupt(int inum) {
    if (handlers[inum] != NULL)
        *handlers[inum]();
}
```

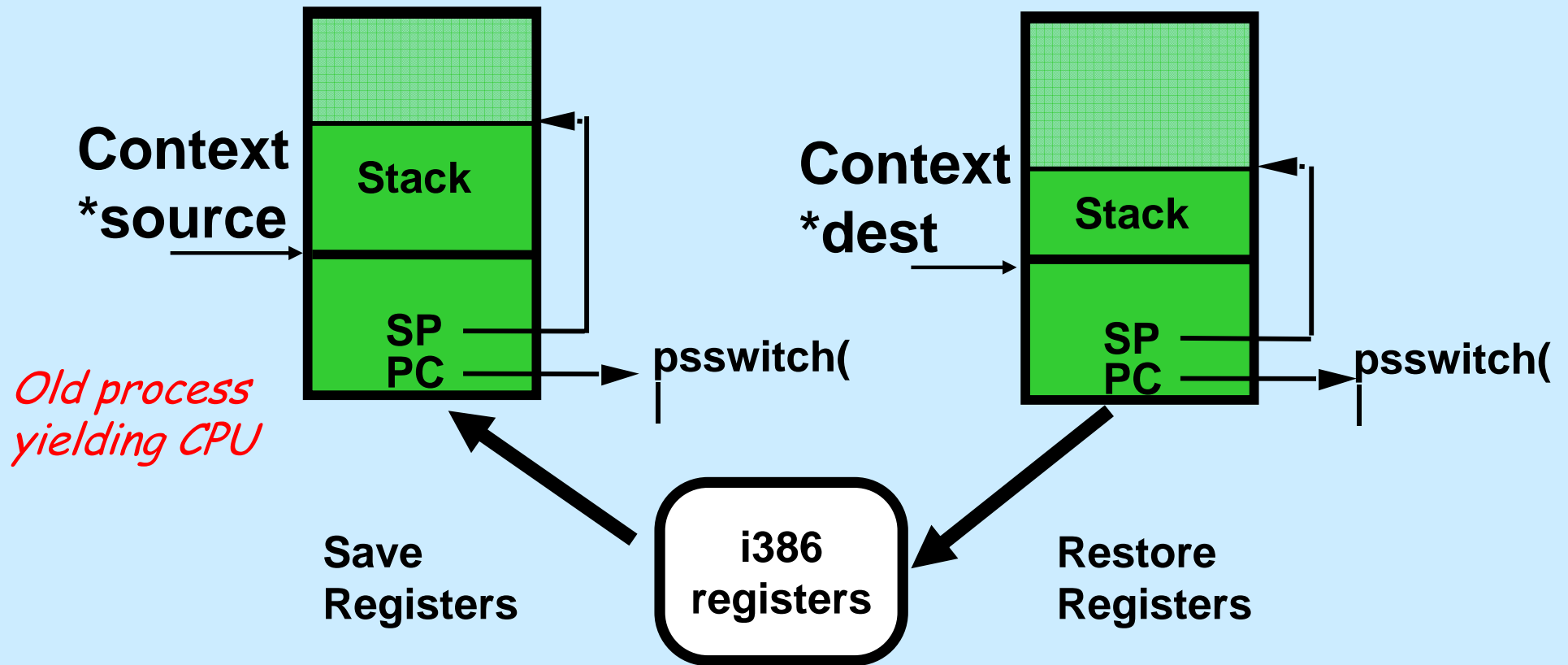
# Recap on Interrupts

- Interrupt handlers are not processes, they run (in Simple Kernel) using the current processes' stack.
- Unlike function calls, interrupts can occur anywhere – between any two assembler instructions. This means that we need to save the **eflags** register to preserve state.
- First level interrupt handler has to be written in assembler.
- The job of the second level interrupt handler is simply to call the third-level handler.
- The third level interrupt handler is the actual routine, written in C, which specifies what to do when this interrupt occurs. Example: **tick()**.

# Context Switch

We are now coming back to this.

*New process acquiring CPU*



```
typedef struct Context { // Data structure for holding i386 registers
    int eax, ebx, ecx, edx, esi, edi, ebp, eflags, eip, esp;
    /* 0 4 8 12 16 20 24 28 32 36 note these offsets*/
} Context;
```

```
void psswitch(Context *source, Context *dest);
```

# Implementation of switch

```
.globl pswitch
```

```
pswitch:
```

```
movl %eax,-4(%esp)
```

```
movl 4(%esp),%eax
```

```
movl %ebx,4(%eax)
```

```
movl -4(%esp),%ebx
```

```
movl %ebx,(%eax)
```

```
movl %ecx,8(%eax)
```

```
movl %edx,12(%eax)
```

```
movl %esi,16(%eax)
```

```
movl %edi,20(%eax)
```

```
movl %ebp,24(%eax)
```

```
pushf
```

```
popl %ebx
```

```
movl %ebx,28(%eax)
```

```
popl %ebx
```

```
movl %ebx,32(%eax)
```

```
movl %esp,36(%eax)
```

```
# (it helps to first see stack layout on next slide)
```

```
# Save current eax on Stack
```

```
# Get addr of 'source' ctxt
```

```
# Save ebx in 'source' ctxt
```

```
# Get orig. eax into ebx
```

```
# Save orig. eax in 'source' ctxt
```

```
# Save other regs
```

```
# (see offsets on previous slide)
```

```
# Only way to get flags
```

```
# Get them into ebx
```

```
# Save them in 'source' ctxt
```

```
# Get return address off stack
```

```
# Save as 'source' IP
```

```
# Save an esp that has return
```

```
# addr popped to simulate a ret
```

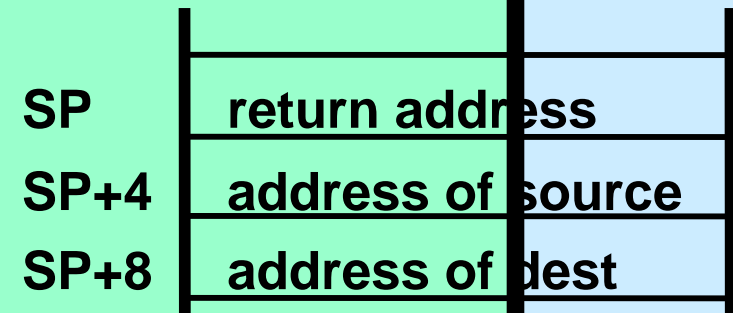
- In any kernel, some parts (that directly deal with registers) have to be written in machine code. (In Linux, 8,000 out of 470,000 lines are assembler).
- You can see the general pattern of saving all the current register values then loading all the new ones.

# (Switch cont.)

```
movl 4(%esp),%eax # Get addr of 'dest' ctxt
movl 36(%eax),%esp # Switch to the 'dest' ctxt stack frame
# Following 3 pushl's set up the stack
# so we can use an iret to return to
# the task (iret is equivalent to ret, popf)
```

```
movl 28(%eax),%ebx
pushl %ebx # Push eflags
movl $KERNEL_CS,%ebx
pushl %ebx # Push code segment of the 'dest' proc
movl 32(%eax),%ebx
pushl %ebx # Push execution addr of 'dest' proc
movl 24(%eax),%ebp # Now just load the rest of the regs
movl 20(%eax),%edi # from the 'dest' ctxt into the regs
movl 16(%eax),%esi
movl 12(%eax),%edx
movl 8(%eax),%ecx
movl 4(%eax),%ebx
movl (%eax),%eax
```

Process stack just after switch() has been called (i.e. at start of switch):



**iret**

In linux this routine is called *switch\_to*, and is implemented in `arch/i386/kernel/process.c` and `include/asm-i386/system.h`.

# Kernel Initialisation

```
void startKernel() {  
    ENTERKERNEL  
    initSched();  
    initTime();  
    startInt();  
    setReady(running);  
    dispatch();  
    int_enable(); // a routine to enable interrupts  
    null();  
    EXITKERNEL  
}
```