Operating Systems Concepts: Chapter 6: The System Nucleus

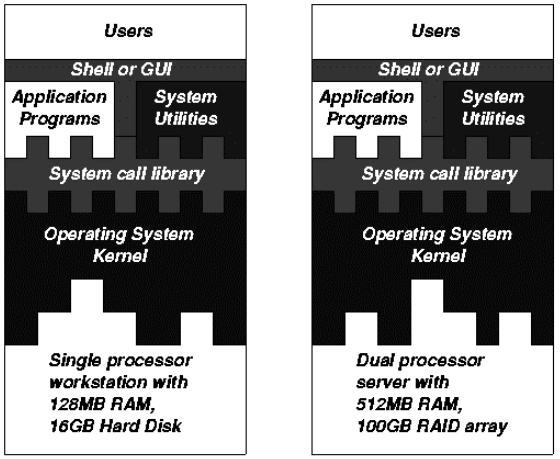
William Knottenbelt (wjk@doc.ic.ac.uk)
(after Jeff Magee, Kevin Twidle, Steve Vickers and Ariel Burton)

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.*)
- 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

The Operating System as Virtual Machine



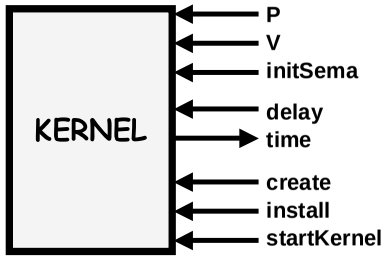
The OS presents the *same interface* and offers the *same services* to users and applications irrespective of the underlying hardware

The System Nucleus (Kernel)

- Shares processor among multiple processes
  - scheduling
  - context switching
- Process management
  - creation, deletion, initiation, termination
  - interprocess 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: see http://www.doc.ic.ac.uk/~wjk/OperatingSystemsConcepts
- Linux: http://www.tamacom.com/tour/linux or /usr/src/linux.

Simple Kernel Design



Simplifying assumptions:
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.

```
module "kernel.hpp"
/* process synchronisation: */
   opaque 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: */
   enum Priority {high, normal, low, idle};
   typedef void Proc();
   void create(Proc *P, int size, Priority pr);
   void install(Proc *P, int intno);
   void startKernel();
```

opaque means "black box" – internal structure is hidden

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.cpp"
#include "kernel.hpp"
// 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("*");
  }
}
```

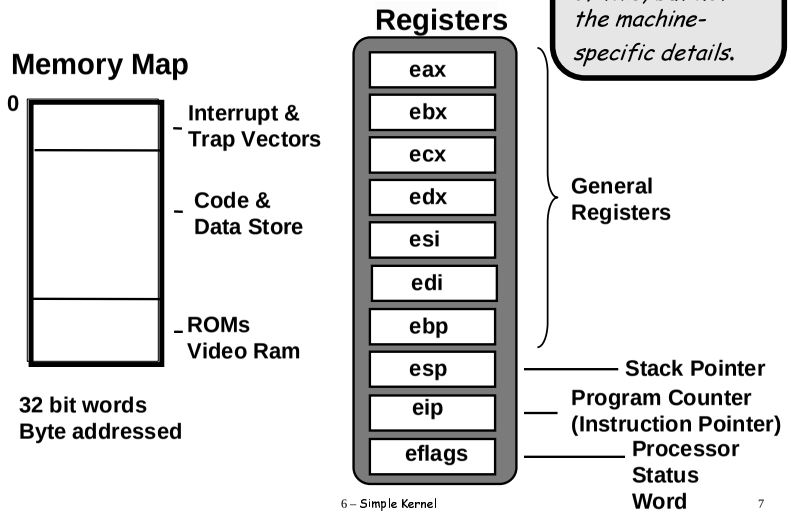
```
void main() {    /* main routine starts here */
  count = 0;
  initSema(sync, 0)
  install(tick, clockvector);
  create(print, 500, normal);   // normal is a priority level
  startKernel();
}
```

Example: Program to output an asterisk every 10 seconds

Two phases of execution –
1: Initialization (creates, installs)
2: StartKernel to run processes.

Target Architecture: Intel 386

You need to understand the general features of this, but not the machine-specific details.



Vectored Interrupts on the 386.

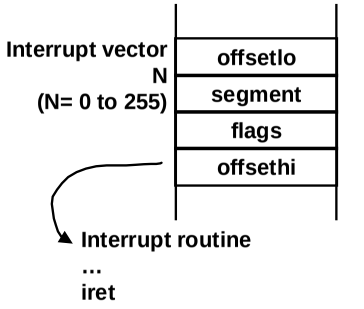
Flags Register

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.

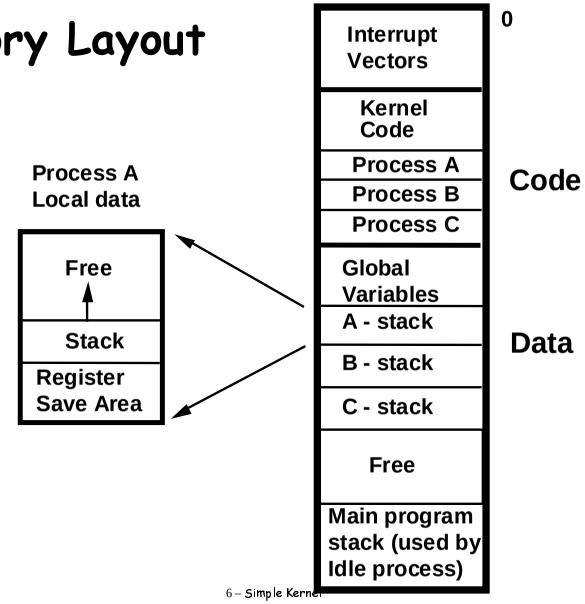
cli	clear IF (disable interrupts)
sti	set IF (enable interrupts)
pushf	put flags on stack
popf	set flags from top of stack

Vector configuration

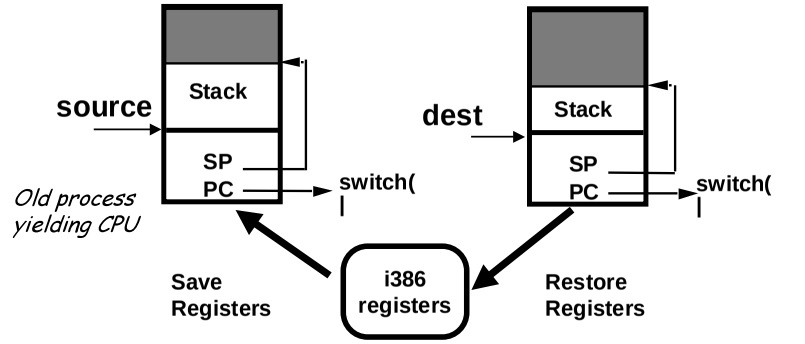


Interrupting device – signals interrupt, puts vector number N (8 bits) on CPU pins.
CPU hardware then – pushes flags and PC (eip) onto the stack;
– sets flag values from the vector;
– calculates new PC (eip) from the segment and offsets …
– … thus effecting a jump to interrupt routine.
At end of interrupt routine, iret instruction pops return PC and flags from stack.

Memory Layout



Context Switch




```
typedef struct Context {
   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 switch(Context *source, Context *dest);

Implementation of switch

```
.globl proc_switch
proc_switch:                # (it helps to first see stack layout on next slide)
movl %eax,-4(%esp)          # Save current eax
movl 4(%esp),%eax           # Get addr of 'source' ctxt
movl %ebx,4(%eax)           # Save ebx in 'source' ctxt
movl -4(%esp),%ebx          # Get orig. eax into ebx
movl %ebx,(%eax)            # Save orig. eax in 'source' ctxt
movl %ecx,8(%eax)           # Save other regs
movl %edx,12(%eax)          # (see offsets on previous slide)
movl %esi,16(%eax)
movl %edi,20(%eax)
movl %ebp,24(%eax)
pushf                       # Only way to get flags
popl %ebx                   # Get them into ebx
movl %ebx,28(%eax)          # Save them in 'source' ctxt
popl %ebx                   # Get return address off stack
movl %ebx,32(%eax)          # Save as 'source' IP
movl %esp,36(%eax)          # Save an esp that has return
                            # addr popped  to simulate a ret
```

You don't need to learn all this! But:

- In any kernel, some parts 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

iret
```

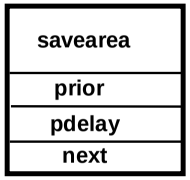
Process stack just after switch() has been called (i.e. at start of switch):

SP	return address
SP+4	address of source
SP+8	address of dest

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

Kernel Representation of Processes

Process Control Block



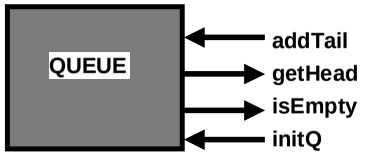
— Register Save Area (context)
— scheduling priority
— delay time
— pointer for pcb management

```
enum Priority {high, normal, low, idle};
typedef PCB *Process;
typedef struct PCB {
    Context savearea;
    Priority prior;
    int pdelay;
    Process next;
} PCB;
```

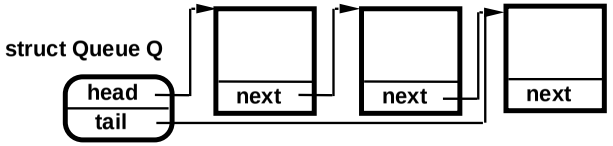
Processes are managed by forming "linked lists" or queues of process control blocks.

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?

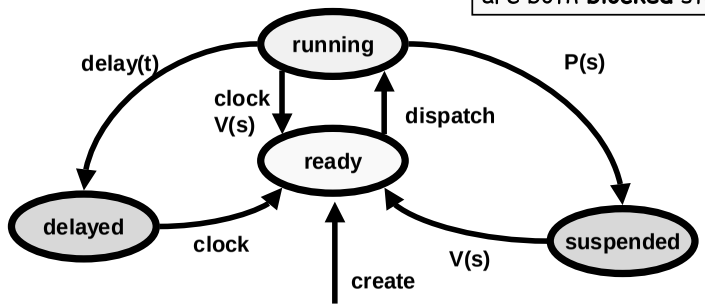
Queue Implementation

```
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:

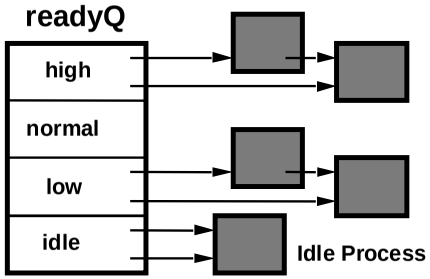
A *ready* process – is on one of the four ready queues, readyQ[priority] (see below)
A *delayed* process – is on delayQ
A *suspended* process – is on the queue associated with a semaphore S
The *running* process – is pointed to by the variable running
(Just one processor, so only one process at a time 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.

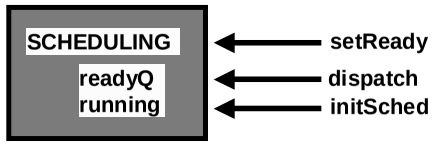



```
Process running;    //points to currently running process
Queue readyQ[4];    //one queue for each priority level
```

```
enum Priority {high,normal,low,idle};
```


Scheduling Operations

Selecting a process to run




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

void dispatch() {
  Process oldrunning = running;
  Priority pr = high;
  while (!isEmpty(readyQ[int(pr)])) {
    pr = Priority(int(pr) + 1);
      // gets next priority
  }
  running = getHead(readyQ[int(pr)]);
  switch(oldrunning->savearea,
                    running->savearea);
}
```

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

void initSched() {
  for (int pr=0; pr<4; pr++)
    InitQ(readyQ[int(pr)]);

  create(null, 0, idle);
  running = getHead(readyQ[int(idle)]);
}
```


Kernel Exit & Entry

To ensure mutual exclusion to the kernel data structures, kernel procedures must run with interrupts disabled. We will use the functions:

```
int int_mutexon()          disables interrupts, result is old value of flags reg.
void int_mutexoff()        restores flags register to value provided in w
```

Assembler:

```
.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 form:

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

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

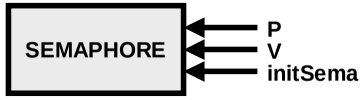
Kernel Exit & Entry (cont.)

For convenience we will use the following macros:

```
ENTERKERNEL
        int psw = int_mutexon();

EXITKERNEL
        int_mutexoff(psw);
```

Semaphore Implementation

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



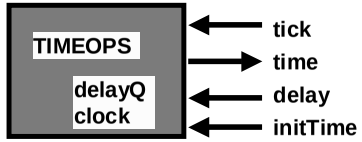

```
void P(Semaphore *s) {
  ENTERKERNEL
  if (s->count > 0)
    s->count--;
  else {
    addTail(s->waiting, running);
    dispatch();
  }
  EXITKERNEL
}
```

```
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

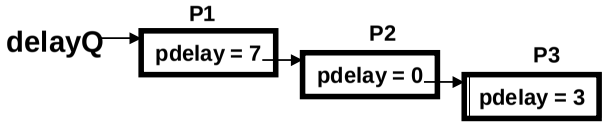
Process delayQ;
int clock;



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.

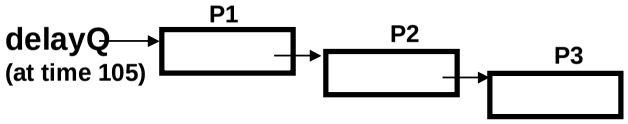
Example:

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



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 (see below).

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();
}
```

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.

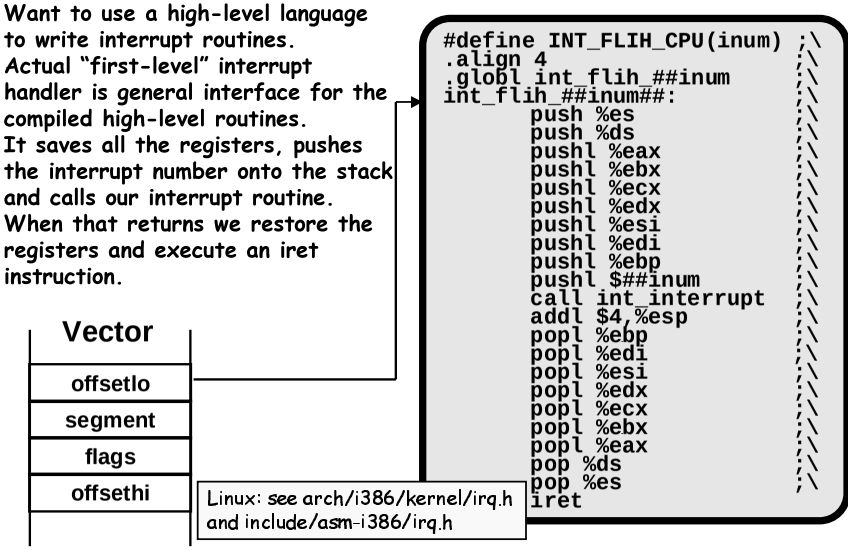
p0 is the current position in delayQ
p1 is next process in delayQ

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
}
```

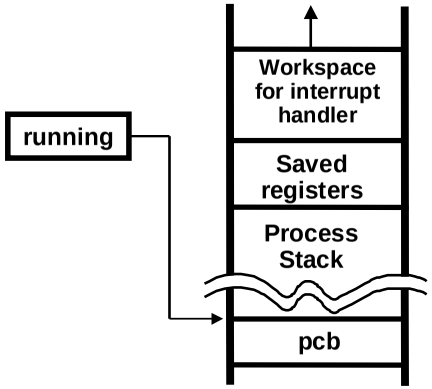

Interrupts First level Interrupt Handler, int_flih

- Want to use a high-level language to write interrupt routines.
- Actual "first-level" interrupt handler is 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.



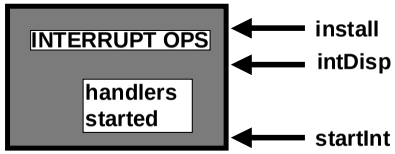
Interrupts (cont.)

- The assembly code routine int_flih also saves and restores the registers since these will be used by the interrupt handler procedure.
- *Interrupt handlers are not processes.* They use the stack of the currently running process.



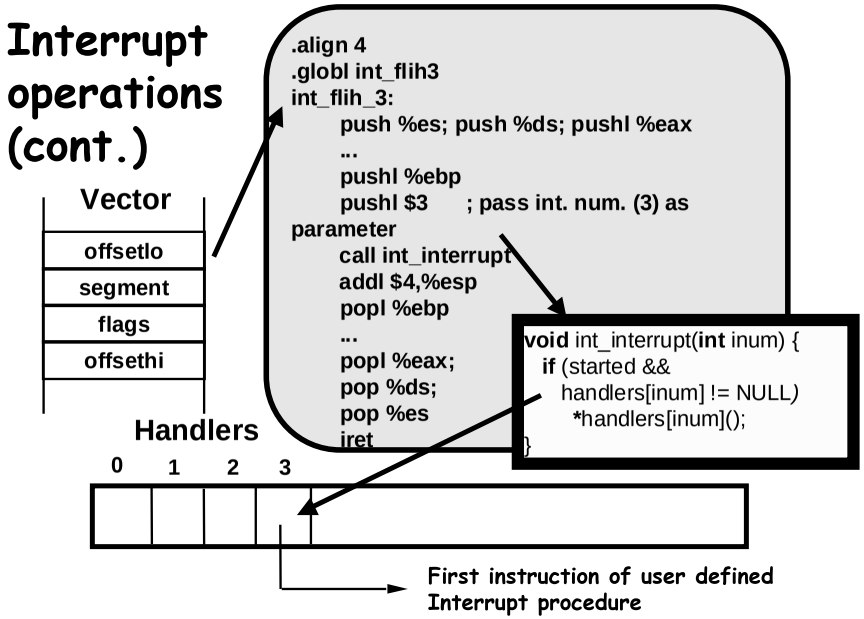
Stack of a process which has been interrupted (must be the currently running process)

Interrupt Operations



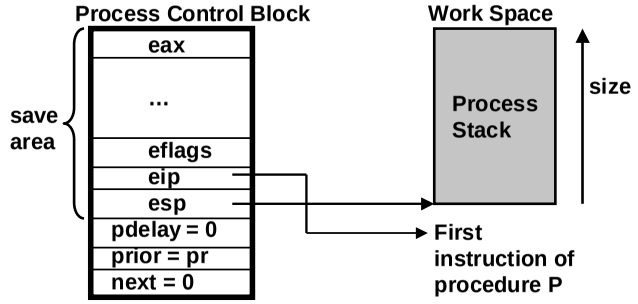

```
typedef void Proc();
Proc *handlers[256];          // 256 different interrupt vectors
bool started = false;

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
}
```

Interrupt operations (cont.)



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


```
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
}
```


Kernel Initialisation

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