4 Lecture: The Process Model

Outline:

4.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn2	LWP	Mon	Jan 26	23:59	
asgn3	dine	Wed	Feb 4	23:59	
lab03	problem set	Mon	Feb 9	23:59	
midterm	stuff	Wed	Feb 11		
lab04	scavenger hunt II	Wed	Feb 18	23:59	
asgn4	/dev/secret	Wed	Feb 25	23:59	
lab05	problem set	Mon	Mar 9	23:59	
asgn5	minget and minls	Wed	Mar 11	23:59	
asgn6	Yes, really	Fri	Mar 13	23:59	
final (sec01)		Fri	Mar 20	10:10	
final (sec03)		Fri	Mar 20	13:10	

  Use your own discretion with respect to timing/due dates.

- Asgn1 due Wednesday (reminder about late days)
- All assignments out
- Lab02: Why minix 3.1.8

4.2 From last time

- History
- Make(1) in AM
- System call mechanisms
- Booting
- Regaining control

4.3 The Process (Users' View)

A quick overview of the (human) users' view of the system:
Each running program becomes a process, isolated from all other processes:

- Each process has the illusion of being alone
- Has its own memory
- Has its own scheduling time
- Has its own interface to the outside world (file descriptors)
- All interprocess (and outside world) interaction takes place through the operating system.

Processes have identity:

- User ID
- Group ID
- Process ID

Processes have resources:

- memory (address space)
- time

4.4 The Operating System's view: The context

In the OS's view, a process consists of some resources:

registers	Each process gets its own copy.
address space	A region of memory, usually starting at address 0, corresponding to a particular process.
identity	uid, gid, pid, ppid, etc. All those things that determine a process's identity and privilege.
file descriptors	Connections to the global file descriptor table to allow for IO
signals/masks	Pending notifications

The OS's purpose is to keep these separate. How?

- Records kept in a **process table**
- Processes run for an allotted time(**quantum**), or until they yield (e.g. disk wait)
- **context switch**
  - Old process suspended (timer?)
  - Old process's registers saved (where?)
  - Old process's memory—text,stack,data—saved in a **core image**
  - Memory set up for new process (text,stack,data)
  - Registers set up for new process
  - New process "continued" as if nothing had happened.
    Imaging blinking and discovering that it was three hours later.

4.5 Example of a context switch

How does it really happen? (with pictures, and everything.)

- Process A is running (Figure 4.5a)
- An interrupt occurs (Figure 4.5b)
- Push registers to preserve them. (Figure 4.5c)
- Save SP in the process table and switch to the operating system (kernel) stack (Figure 4.5d)
- The OS is now running:
  - Preserve Process A's memory (if necessary)
  - Choose the next process to run.
  - Load Process B's memory (if necessary)
- Restore B's SP from process table. (Figure 4.5e)
- Pop B's registers (Figure 4.5f)
- Return from interrupt (Figure 4.5g)

4.6 System Calls Again

We said "An OS is defined by its system calls". What does that mean?

System Call
A system call is the means by which the kernel provides access to a particular operating system service, and the services available through system calls (e.g., reading and writing the disk, allocating memory, starting new processes, etc.) are reserved to the kernel; there is no other way of doing these things.

4.7 System Call Mechanisms

Last time we said

- It's all about privilege.

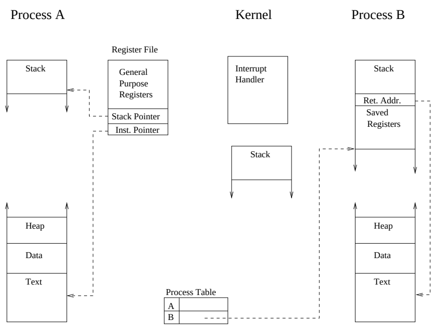
4.7.1 How to do it

How done? Machine dependent. usually in assembly, but hidden from the users' view.

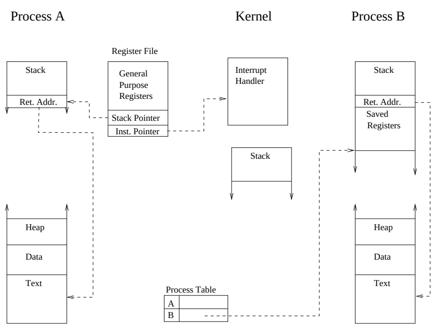
In any case, some sort of trap is involved to get supervisor privileges.

A *trap* is a software generated interrupt. Exactly what happens varies from architecture to architecture, but the result usually involves:

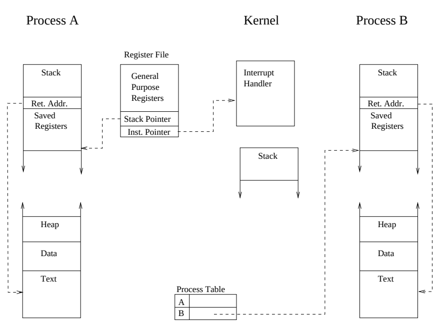
- saving the state of the currently executing program
- elevating of processor privilege level to "supervisor"



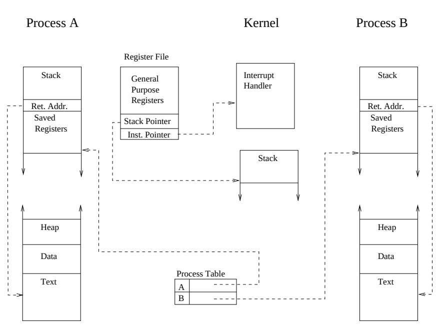
(a) While process A is running



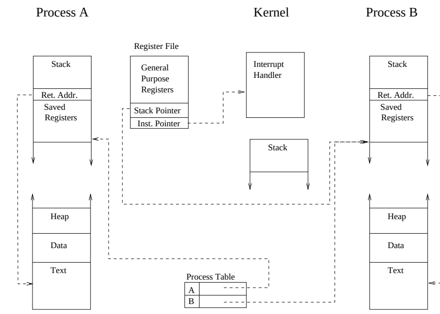
(b) After the interrupt



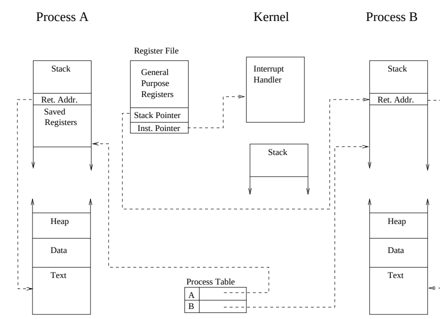
(c) Saving A's registers



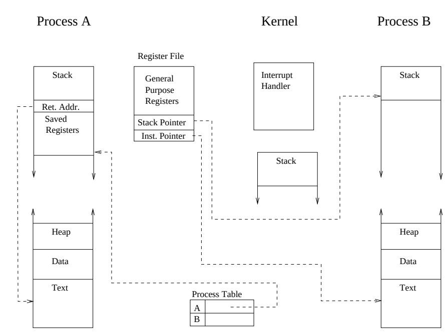
(d) switch to the operating system (kernel) stack



(e) Restore B's SP from process table.



(f) Pop B's registers



(g) After return from interrupt

Figure 5: The process of a context switch

- transfer of control to a pre-registered routine
- *The OS does something*
- Then, usually:
  - privilege level is reduced back to user
  - original state of executing program is restored

4.7.2 And onwards

- How does the OS get control back?

4.8 Onwards

Last time we talked about context switching and the concept of a Process. Now that we have the general idea, let's look at some general structures.

This week we are going to look at general OS architectures, then move into scheduling.

4.9 The Process Model: a little deeper

Now, we need to talk about some details of how it's actually done and the issues involved.

The **process** is the most important concept in understanding operating systems.

The operating system provides **multiprogramming**:

pseudoparallelism time-sliced parallelism on a uniprocessor

(yield?, pre-empt?)

true parallelism provided by a multiprocessor

Keeping track of parallel activities is difficult, so OS designers have developed the model of the **sequential process**:

Each process (**running program**) has its own:

- IO Access: open file descriptors
- Register file
- Memory
  - Text (code) segment
  - Data (bss and heap) segment
  - Stack segment
- Program Counter

4.10 Pseudoparallelism and nondeterminism

(Ir?)regular (random) context switches mean programs can make no assumptions about:

- order of execution relative to other programs
- timing (example: a clock, or game timing)

(If timing *is* important, you need to get a Real-Time O.S.)

4.11 Possible process states

Not all processes can run at the same time:

At any given time, any process can be in one of three possible states: Running, Ready, or Blocked. Possible transitions between these states are shown in Figure 6.

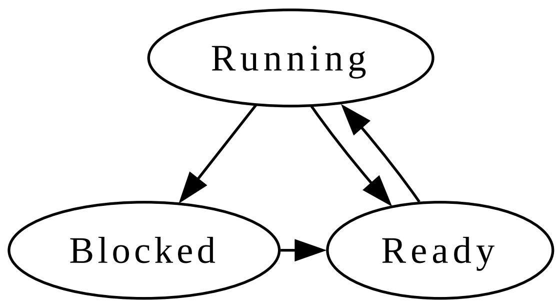


Figure 6: Possible states for a process

In more detail, a process can be:

Running The process is loaded in memory and is currently executing on the cpu.

Ready The process is runnable, but another process has the cpu.

Blocked The process is waiting for some external event to enable it to run. E.g. completion of some IO event, or an alarm, or availability of more memory.