Recap

Lottery Scheduling (Random, non-deterministic, long-run fairness)

Stride Scheduling (Deterministic, long-run and short-term fairness)

Linux CFS

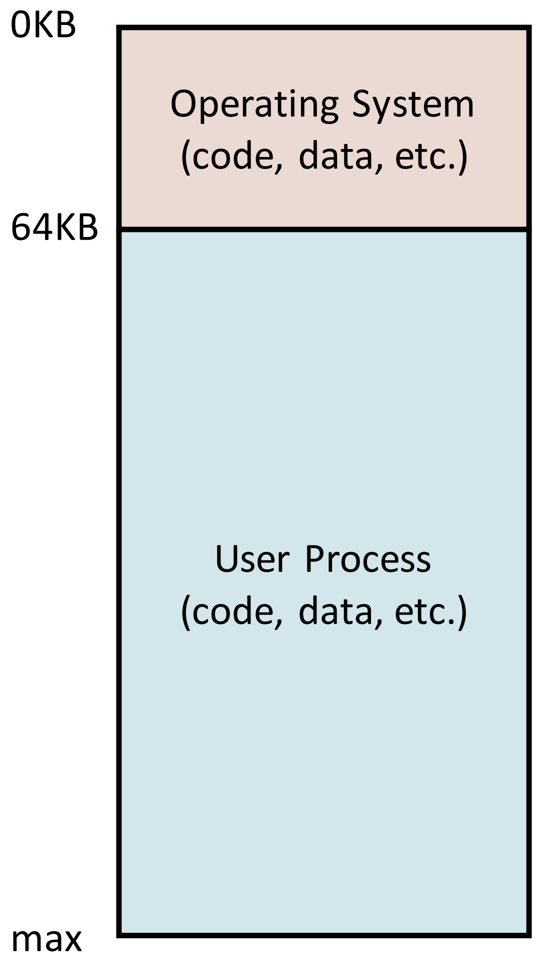
- Track vruntime for process; picks the shortest-vruntime to run using black-red tree
- Parameters: sched_latency & min_granularity
- Dynamic time_slice based on priority (weight)
- Dynamic vruntime based on priority (weight)

Address Spaces & Address Translation

(Based on Ch 13 & 15)

Slides revised from
Matthew Tancreti
Iowa State University

Single Programming (Mono Programming)



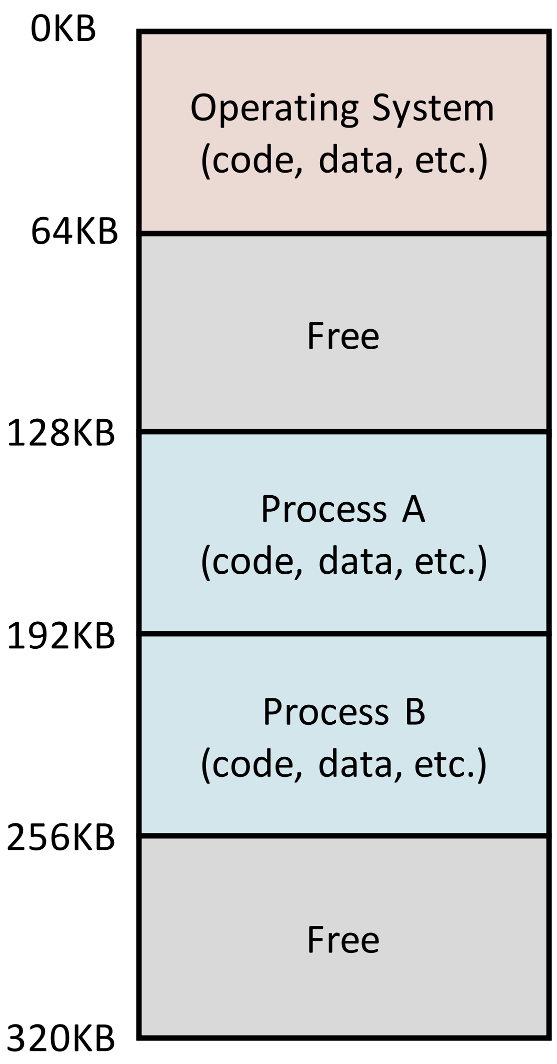
Process' state (playground): main memory (larger), registers (smaller).

Main memory divided between the OS and user process

Addresses assigned statically (at compile time)

Too simple for general-purpose computers; but still used in simple (single program) embedded systems

Multi-programming/Multi-tasking



Multiple processes run concurrently

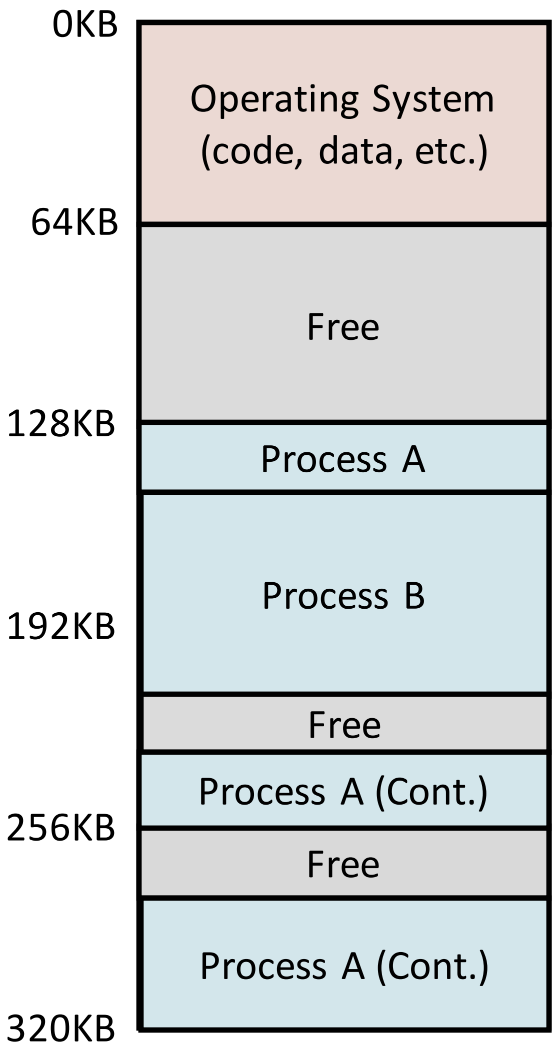
Processes have to share main memory

- When switch process, we can save the data at registers but can we save the data at main memory?

Simplest scheme is to assign contiguous regions of memory

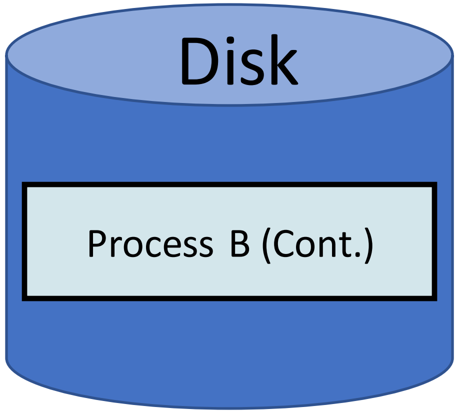
Becomes costly when a process needs to grow its memory

The Reality: OS Developer's View of Memory

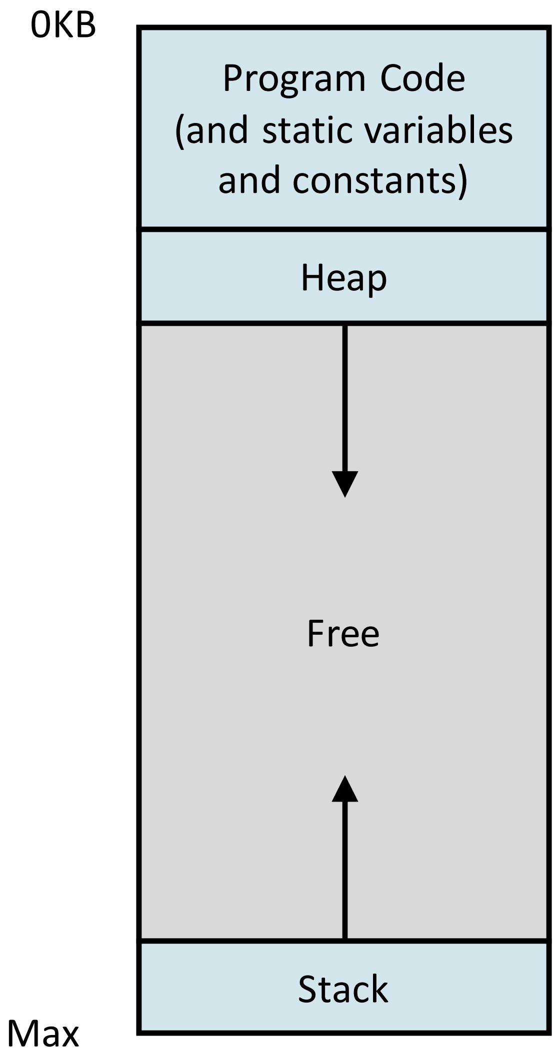


The reality is much more messy

- Memory can become fragmented
- Unused code/data doesn't need to be in physical memory
- Not every process will fit in memory, use disk for extra storage



The Ideal: Application Programmer's View of Memory



A process' view (An application programmer's view) of memory is called its **address space**

The address space is an abstraction of the physical memory, assumptions:

- Address space starts at 0
- Address space is contiguous
- All address available at any time

At the top are static items (e.g., code, global variables and constants)

Processes have **two forms of dynamic memory**: **heap** and the call **stack**

Because they both need to grow an unknown amount, it is logical to place them on opposite ends of the address space

Example (address.c)

```c
#include <stdio.h>
#include <stdlib.h>

int func(int y){
        int z=21;
        printf("location of y:  %p\n", &z);
        printf("location of z:  %p\n", &y);
        return y+z;
}

int main(int argc, char *argv[]){
        printf("location of main: %p\n", main);
        printf("location of func: %p\n", func);

        printf("location of heap: %p\n", malloc(1024));

        int x=10;
        int k=func(x);
        printf("location of argc: %p\n", &argv);
        printf("location of argv: %p\n", &argc);
        printf("location of k:  %p\n", &k);
        printf("location of x:  %p\n", &x);
}
```

Output:

```
> ./address
location of main: 0x40117e
location of func: 0x401136

location of heap: 0x239d6b0

location of y: 0x7ffcf7f83bec
location of z: 0x7ffcf7f83bdc

location of argc: 0x7ffcf7f83c00
location of argv: 0x7ffcf7f83c0c
location of k: 0x7ffcf7f83c18
location of x: 0x7ffcf7f83c1c
```

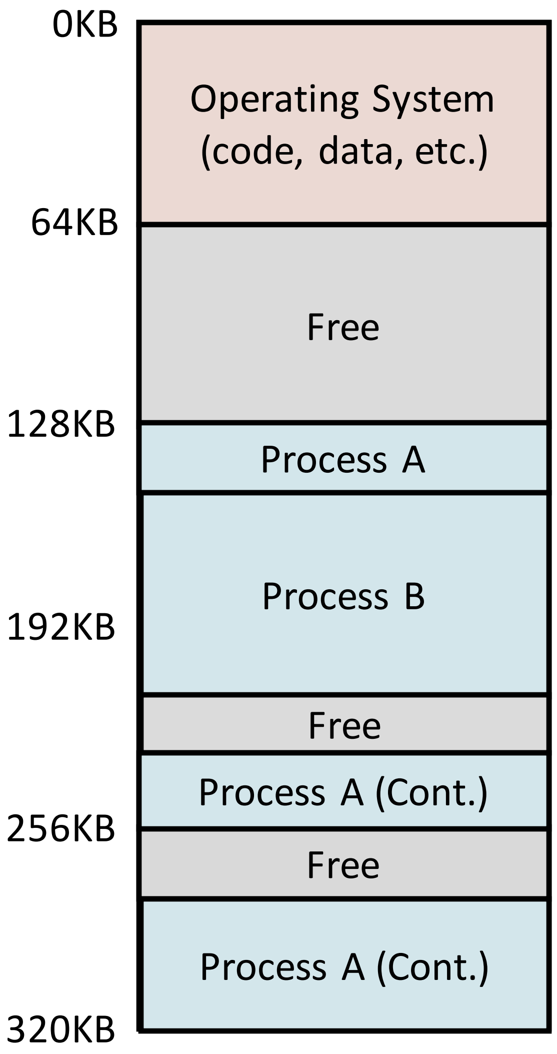
Code

Heap

Stack Frame for func

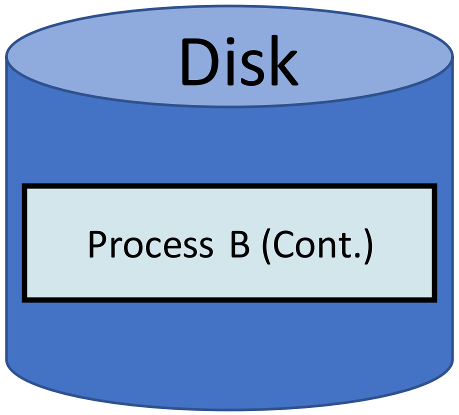
Stack Frame for main

From Messy Reality to Neat Illusion

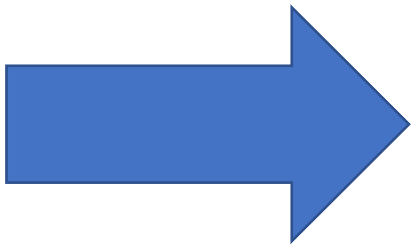


The reality is much more messy

- Memory can become fragmented
- Unused address space doesn't need to be mapped to physical memory
- Not every process will fit in memory, use disk for extra storage



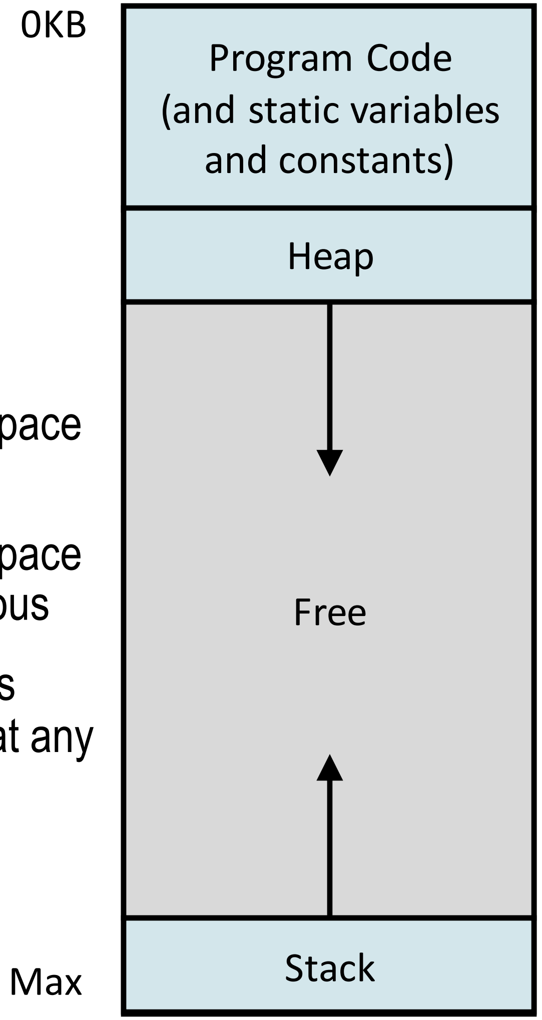
Memory Virtualization



Address space starts at 0

Address space is contiguous

All address available at any time



Word Size

Processor architecture determines word size

Registers, instructions, address bus and data bus are typically one word

Example: 32-bit machine can address 4GB of memory

8-bit	7 … 0	00000000	HEX: 0x00
16-bit	15 … 0	00000000 00000000	HEX: 0x00 00
32-bit	31 … 0	00000000 00000000 00000000 00000000	HEX: 0x00 00 00 00
64-bit	63 … 0	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000	HEX: 0x00 00 00 00 00 00 00 00

Address Translation
Key to Virtualize Memory

"All problems in computer science can be solved by another level of ***indirection****."*
- Butler Lampson

Memory virtualization – processes have abstract view of memory (their address space) but share single physical memory

Address translation – translate process address into physical address

How?

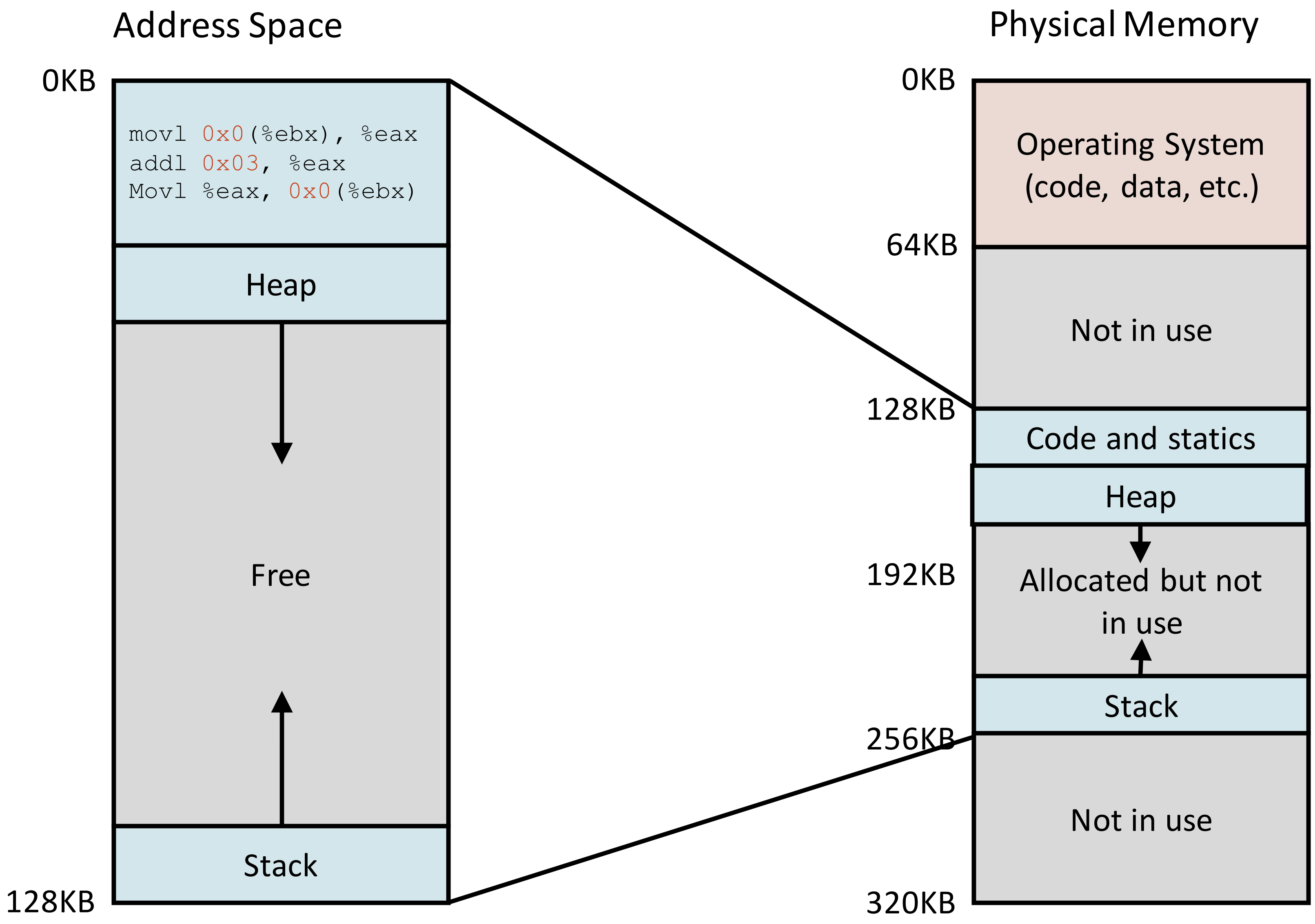
- To solve the problem of CPU virtualization we used **limited direct execution** – context switch to kernel mode, then returns to user mode
- Address translation similar conceptually, but memory accesses happen every instruction, not realistic to trap on every access
- Efficient address translation requires additional hardware support – **hardware-based address translation**

Assumptions (only for now)

- A process' address space must be placed contiguously in physical memory.
- The size of the address space is not too big; it is less than the size of physical memory.
- Each address space is exactly the same size.

(We will remove these assumptions in later lectures)

Example: Memory Relocation (i.e., VM->PM Mapping)



Software-Based Translation Method

Loader: the purpose is to load the binary program on disk into the process memory

Some early loaders also had the job of translating all addresses found in instructions from virtual to physical locations

Translation is performed once (**statically**) before the process begins execution

`movl(1000, %eax)` → `movl(4000, %eax)`

Disadvantages:

- the loader needs to be trusted code (or no memory protection)
- relocation after the process starts is costly

Hardware-based Translation (Dynamic Relocation) Method: Base and Bounds

The **base and bounds** method requires two CPU registers

- **base** – points to start of process in physical memory
- **bounds** – points to maximum legal address for process

When instruction is executed, all addresses translated by hardware

```
physical address = virtual address + base
```

Bounds used in one of two ways:

- Bounds specify the virtual bounds: If virtual address > bounds, access is illegal and so trap to kernel
- Bounds specify the physical bounds: If physical address > bounds, access is illegal and so trap to kernel

Allows **dynamic relocation** of process memory

MMU

The hardware responsible is the **Memory Management Unit (MMU)**

It is typically part of the CPU but sits between the core and the address buss

Translates all addresses between CPU and main memory

Example

Instruction in code:

```
128: mov 1000, %eax
```

0. Assume base: 32,768 and bounds: 128K
1. Program Counter (PC) is incremented to 128
2. CPU begins fetching instruction by reading from address 128
3. MMU checks 128 < bounds (valid virtual address); translates 128 to 32,896 = 32768+128 and memory is read
4. CPU decodes instruction and requests a read from address 1000
5. MMU checks 1000<bounds (valid virtual address); translates 1000 to 33,768 = 32768 + 1000 and memory is read
6. CPU finishes execution of instruction

Exception Handling

Memory access out of bounds results in a trap

OS typically terminates process

Hardware Requirements

Hardware Requirement	Common Implementation
Privilege mode	Kernel mode
Base and bounds registers	
Translate virtual address	**MMU** intercepts all addresses between CPU and bus
Privileged instructions to update base and bounds	Write to registers (kernel mode)
Privileged instructions to register exception handlers	Write to interrupt vector table (kernel mode)
Ability to raise exceptions	**Trap** when read out of bounds

Put together: Limited Direct Execution (Dynamic Relocation)

OS @ boot (kernel mode)	**Hardware**	**(No Program Yet)**
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

Limited Direct Execution (Dynamic Relocation)

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A: allocate entry in process table alloc memory for process set base/bound registers **return-from-trap** (into A)		
	restore registers of A move to **user mode** jump to A's (initial) PC	
		Process A runs Fetch instruction
	translate virtual address perform fetch	
		Execute instruction
	if explicit load/store: ensure address is legal translate virtual address perform load/store	
		(A runs...)
	Timer interrupt move to **kernel mode** jump to handler	

Limited Direct Execution (Dynamic Relocation)

Handle timer
decide: stop A, run B
call `switch()` routine
save regs(A)
to proc-struct(A)
(including base/bounds)
restore regs(B)
from proc-struct(B)
(including base/bounds)
return-from-trap (into B)

restore registers of B
move to **user mode**
jump to B's PC

Process B runs
Execute bad load

Load is out-of-bounds;
move to **kernel mode**
jump to trap handler

Handle the trap
decide to kill process B
deallocate B's memory
free B's entry
in process table