Recap
- Memory Maps
- Programs
- Simple OS

We looked at a simple 16 bit processor with 64k addresses, and with progressively more complex ways of dividing processes and a program until we reached something that was very close to MSDOS, where you have program code in low memory, starting with 0000 and a heap expanding vertically above it and a stack growing down.

Compartmentalization
- A stack basically the state the program.
- You can take two processes.
- In one process, the Stack Pointer is pointing to the stack from program 1's code.
- The stack is shared, so each program just stores its own location in the same stack. <- No compartmentalization

One way to compartmentalize is to use base and bound registers.
Base and bounds registers
Registers that define which memory regions currently executing code can execute.

Problem: If the OS can set the registers, a malicious / badly made program can do it

We need **supervisor mode** and **user mode** to avoid this problem.

- An interrupt instruction can be used.
  - It acts as the gateway between the two modes.
  - Has a jump table of ____? Legal supervisor mode operations?

What this means...
- We now have a more complicated context switch
- Before, all info was stored on stack, but now we also have these two registers
- Before:
  - Switch from P1 to P2
    - P1.saved SP=SP
    - SP=P2.saved SP
    - RET
- New:
  - P1.saved SP=SP
  - base=p2.base
  - bounds=P2.bounds
  - SP=P2.saved SP

- You can also implement the hardware such that all memory accessing is relative to base and bounds.
  - This becomes UNIX
- How does stack allow you to trace computation??

How do you start a process?
**UNIX**
- Need a syscall because process can't write to another area of memory
- Call fork() which makes 2 copies of existing process, one becomes parent, and the other becomes child
  - Child calls *execute* and loads new code in itself

**Windows**
- You have a process that makes a call to the OS and tells it to spawn a new process
- Process is made and everybody is happy.

Handling multiple terminals

Bad:
• Read_terminal(#)
• Read_terminal_1()

• UNIX had io handles
  ○ Handle = open(terminal1)
  ○ Read(handle)
  ○ Write(--)

----------

Process: a running program ← typically protected
Thread: a _____ with a process

Race conditions

If...

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++;</td>
<td>y++;</td>
</tr>
</tbody>
</table>

No problem...

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x++;</td>
<td>x++;</td>
</tr>
</tbody>
</table>

Mutex

*Mutual Exclusion*

Lock that can be held by one thread at a time.

POSIX Threads. pthreads

pthread_mutex_t m;
pthread_mutex_lock(m);

Spinlock:

You have a location in memory that’s initialized to a certain value and each CPU has an instruction called SWAP which will exchange a register with that location in memory (atomically)