Stack Architecture and Flat Memory For Faster Syscalls

By: Eric LaForest

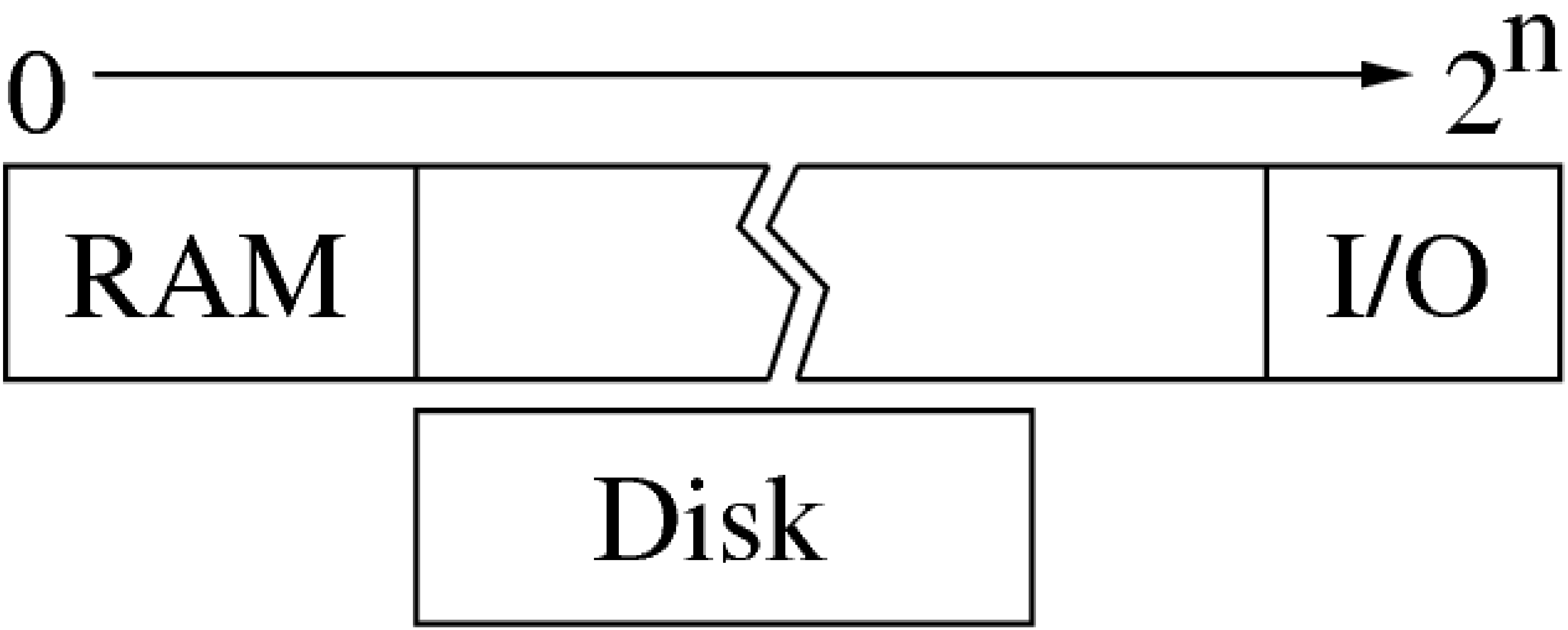
Background:

Switching between user and kernel mode can be expensive due to TLB flushes and saving processor state. This overhead negatively impacts fine-grained systems such as microkernel OSes.

Premise:

Use a simpler memory and processor architectures to improve the performance of mode switches.

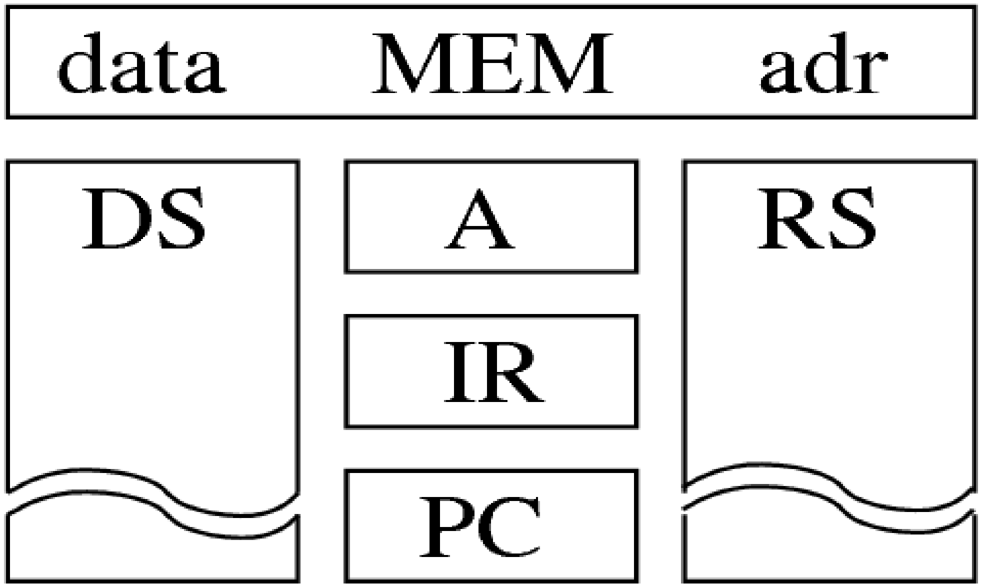
Address Space



Key Points:

- Flat address space: no virtual memory
- Memory-mapped I/O for disk, cycle counter, and console
- Address space after physical RAM is mapped to disk by kernel

Stack Architecture Summary

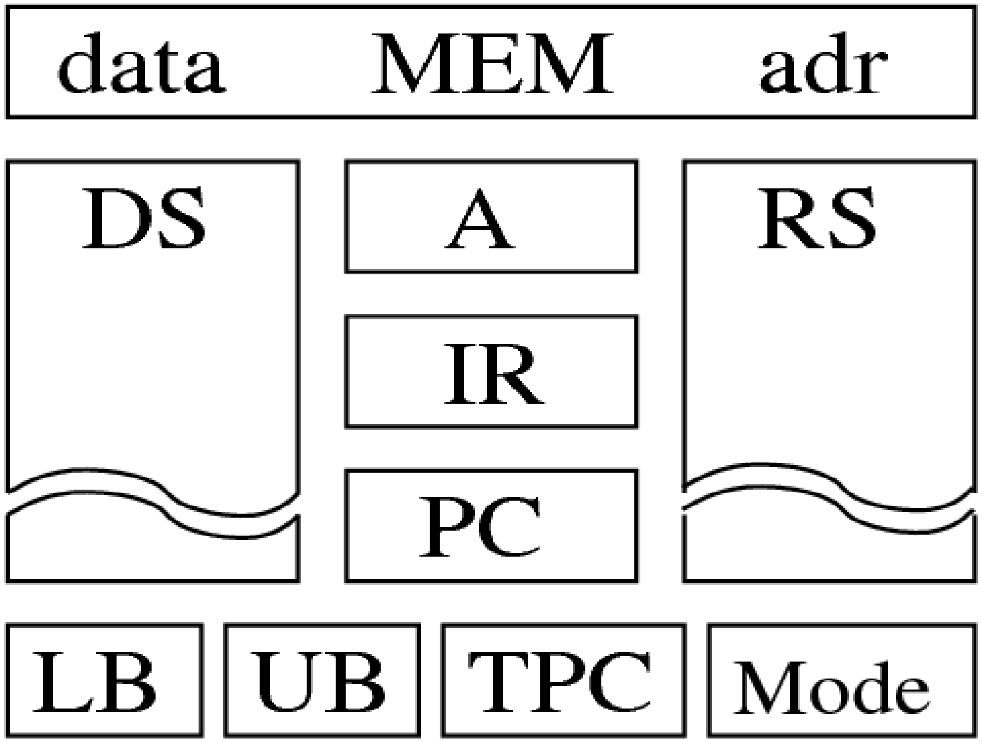



```
DS: Data Stack
RS: Return Stack
A:  Address Register
IR: Instruction Reg.
PC: Program Counter
MEM: Main Memory
```

Key Points:

- Stacks are non-addressable and on-chip
- All calculations done on top of Data Stack
- Memory load/store from top of Data Stack using Address Register
- Subroutine return addresses held in Return Stack
- Data can be moved between stacks
- Code is not position-independent (branches are absolute)

Virtualization

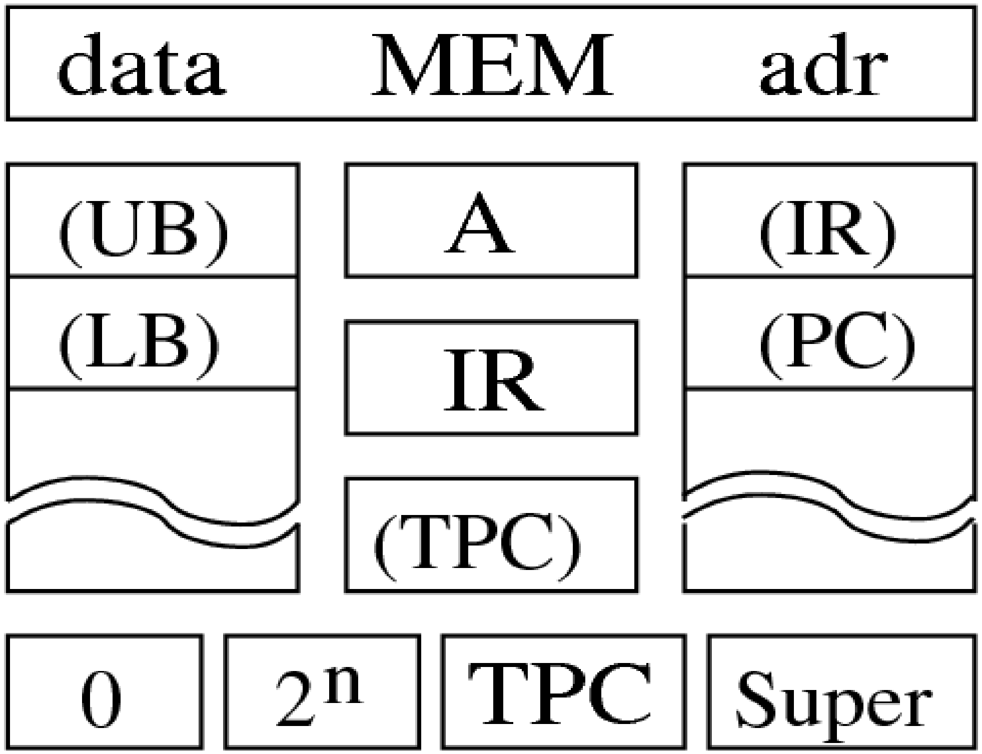



```
LB:   Lower Memory Bound
UB:   Upper Memory Bound
TPC:  Trap Program Counter
Mode: User/Super. Mode Bit
```

Key Points:

- Memory load/store outside of memory bounds will cause a trap
- Return To User (RTU) privileged instruction to enter User Mode
- Executing RTU in User Mode causes a trap, used for syscalls

State After A Trap




```
(UB):  User Upper Bound
(LB):  User Lower Bound
(IR):  User Instr. Reg.
(PC):  User Prog. Count.
(TPC): Trap Prog. Count.
Super: Supervisor Mode
```

Key Points:

- Trap to Supervisor Mode executes in **two cycles**
- Memory bounds set to maximum range to make traps impossible
- Return to User Mode is the exact reverse process
- No memory traffic other than an instruction fetch

Access To Memory

Two ways for a process to access data from outside its bounds:

Trap:
The process attempts to directly read/write the data, causing a trap to kernel which decides whether to complete the operation or deny access to memory.

Syscall:
The process places a syscall number on the Data Stack and executes a Return To User (RTU) instruction, causing a trap to kernel.

Tests

getpid(): have a process get its Process ID from its header

byte read: read one byte from a cached disk block
(Linux reads a byte, Stack reads an int)

Test Results

Linux results from lmbench 3.0-a7-1 on kernel 2.6.20.6
on 2.2GHz AMD Athlon™ 64 with warm cache.

Stack results from cycle-accurate simulator running a simple kernel.

Test	Linux (cycles)	Stack (cycles)	Speedup
getpid() trap:	N/A	98	3.22
getpid() syscall:	316	81	3.90
byte read trap:	N/A	105	5.87
byte read syscall:	616	N/A*	

*Stack syscall reads entire block, trap returns one buffered byte

Conclusions

- A stack architecture and flat memory can improve syscall performance.
- Performance speedup is not the expected order of magnitude as most of the cycles (~70) are spent saving/restoring state and checking permissions.
- However, Linux was tested in ideal conditions (no TLB misses)
- Finally: improved performance on much simpler hardware than x86.

Further Work

- Simplifying stack trap mechanism: don't copy LB/UB to stacks on trap, let the kernel remember it per process.
- Extend trap mechanism to subroutine calls.
- Alternatively, remove initial trap checks by reducing source of traps to one method only (call, RTU, or mem. trap).
- Managing flat, non-virtual memory by using cheap cross-domain calls to dynamically generated code (fast IPC).