Processes 101

- The OS presents a running program with the illusion of a dedicated computer to run on...
  - private linear virtual address space for program code and data
  - architecturally supported mapping from virtual to physical addresses
  - a fair share of the CPU
  - virtualized I/O devices (files, windows, network connections)
- …and access to operating system services that let the process interact with the world outside.
  - procedural interface to I/O on virtual devices
  - ability to create child processes to run other programs, and monitor their progress

Processes and the Kernel
The Kernel

- The kernel program resides in a well-known executable file. The “machine” automatically loads the kernel into memory (boots) on power-on or reset.
- The kernel is (mostly) a library of service procedures shared by all user programs, but the kernel is protected:
  - User code cannot access internal kernel data.
  - User processes can execute within the kernel, but control transfers into the kernel only at well-defined points.
- Kernel code is just like user code, but the kernel is privileged:
  - The kernel has direct access to all hardware functions and the data structures in kernel memory.

The Virtual Address Space

A typical process VAS space includes:
- user regions in the lower half
  - V->P mappings specific to each process
  - mappings change on each process switch
  - accessible to user or kernel code
- kernel regions in upper half
  - shared by all processes
  - accessible only to kernel code
  - kernel VAS may include unmapped kseg

A VAS for a private address space system (e.g., Unix) executing on a typical 32-bit architecture.
The Stack

A Stack is used for
- passing parameters to procedures/methods
- storing local variables

`Stack ptr` register (SP) points at current stack top.

A stack frame (activation record)

<table>
<thead>
<tr>
<th>Return results</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old frame ptr</td>
<td>arg1</td>
</tr>
<tr>
<td></td>
<td>arg2</td>
</tr>
<tr>
<td>Local variables</td>
<td>stack ptr</td>
</tr>
</tbody>
</table>

First few return results and arguments are mapped to specific registers (calling conventions).

Kernel Mode

Modern CPUs support multiple execution modes.
- The current mode is determined by the value of a field in a special processor status register (e.g., `psw` or `psl`).
  - two modes (one bit) typical on RISCs today: `kernel` and `user`
- `Kernel mode` carries special privileges.
  - access to protected CPU facilities to control machine functions
    - protected registers
    - privileged instructions
  - read/write access through VAS mappings for kernel text/data

Transitions from user mode to kernel mode are caused by `interrupts` and `exceptions`.
Interrupts and Exceptions

- an “unnatural” change in control flow
- an interrupt is caused by an external event
device requests attention, timer expires, etc.
- an exception is caused by an executing instruction
CPU requires software intervention
- kernel handler routine for each event type

<table>
<thead>
<tr>
<th></th>
<th>unplanned</th>
<th>deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>async</td>
<td>interrupt</td>
<td>AST</td>
</tr>
</tbody>
</table>

Protecting Entry to the Kernel

Protected events and kernel mode are the architectural foundations of kernel-based OS (Unix, NT, etc).

- The machine defines what conditions will cause each event.
- Kernel installs handlers for each type of event at boot time.
  protected registers or a table in memory read by the machine

The machine transitions to kernel mode only on an event.
Therefore the kernel chooses what code will execute in kernel mode, and when.
System Call Traps

User code initiates system call traps to invoke kernel services.

- procedural interface for user code (through standard library)
  syscall stub or wrapper routine for each syscall
  executes a special trap instruction (e.g., chnk or callsys)
syscall arguments/results passed in registers or user stack

```
read() in libc.a (executes in user mode):
  move arg0...argn, a0...an
  move SYS_CALL_READ, v0
callsys
  move rl, _errno
  return
```

# syscall args in registers A0..AN
# syscall dispatch code in V0
# kernel trap
# (return unless high-order R0 bit set)
# errno = return status
**Handling a System Call Trap**

1. *Machine* saves return address and switches to kernel stack.
   - save user SP, global pointer, PC on kernel stack
   - *set kernel mode* and transfer to a syscall trap handler (*entSys*)

2. *Trap handler* saves software state, and dispatches.
   - save some/all registers/arguments on kernel stack
   - vector to syscall routine through *sysent[v0: dispatchcode]*

3. Trap handler returns to user mode.
   - restore software register state (for security and correctness)
   - execute privileged return-from-syscall instruction (e.g., *retsys*)
   - machine restores SP, GP, PC and sets user mode
   - emerges at user instruction following the *callsys*

**Handling a System Call**

1. Decode and validate by-value arguments.
2. Validate by-reference (pointer) IN arguments.
   - Validate user pointers and copy into kernel memory with *copyin*.
   - Kernel has access to user memory: syscall executes in process context with user space addresses mapped.

3. Call internal routines that implement the service.
   - Copy into user memory with *copyout*.

5. Set up registers with return value(s); return to trap handler.
Questions About System Call Handling

1. Why not just use `bcopy` instead of `copyin` and `copyout`?
2. What should `copyin` and `copyout` do?
3. What stack should the system call handler use?
   Why not use the user stack? A global kernel stack?
4. What would happen if the kernel did not save all registers?
5. Where should per-process kernel global variables reside?
   syscall arguments (consider size) and error code
   How should the system handle multiple threads per-process?
6. What if the kernel executes a `callsys` instruction? What if user code executes a `retsyst` instruction?