System calls

Programming interface to the services the OS provides:
- read input/write to screen
- create/read/write/delete files
- create new processes
- send/receive network packets
- get the time / set alarms
- terminate current process
- ...

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
  - "Narrow waist" makes it highly portable
  - Robust (small attack surface)
- Internet IP layer also offers a skinny interface!

- Much care spent in keeping interface secure
  - E.g., parameters first copied to kernel space, then checked
    - To prevent user program from changing them after they are checked!

System call interface

Portable OS Library

Portable OS Kernel

- Compilers
- Databases
- Web Browsers
- Email
- Word Processing
- Web Servers

- x86
- ARM
- PowerPC
- 10Mbps/100Mbps/1Gbps Ethernet
- 1802.11 a/b/g/n
- SCSI
- Graphics accelerators
- LCD Screens
Executing a System Call

Process:
- Calls system call function in library
- Places arguments in registers and/or pushes them onto user stack
- Places syscall type in a dedicated register
- Executes `syscall` machine instruction

Kernel
- Executes `syscall` interrupt handler
- Places result in dedicated register
- Executes `RETURN_FROM_INTERRUPT`

Process:
- Executes `RETURN_FROM_FUNCTION`
Executing read System Call

```c
int main(argc, argv){
    c = read(fd, buffer, nbytes)
    ...
}
```

**UPC:** user program counter
**USP:** user stack pointer
**KSP:** kernel stack pointer

**UPC:** user program counter
**KPC:** kernel program counter

**note:** kernel stack is empty while user process running
int main(argc, argv)
{
    ... 
    c = read(fd, buffer, nbytes) 
    ...
}

_executing read system call

_UPC_

user space

_kernel space

_UPC_: user program counter
_UPSP_: user stack pointer
_KSP_: kernel stack pointer

note: kernel stack is empty while user process running
Executing `read` System Call

```c
int main(int argc, char* argv)
{
    ...
    c = read(fd, buffer, nbytes)
    ...
}
```

_`read`_
- `mov R8, %R0`
- `syscall`
- `return`

**User stack frame for main()**

**Kernel stack frame for _read**

**User stack**

**Kernel stack**

**UPC:** user program counter  
**USP:** user stack pointer  
**KPC:** kernel program counter  
**KSP:** kernel stack pointer

Note: kernel stack is empty while user process running
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov R:READ, %R0
    syscall
    return

HandleIntrSyscall:
    push %Rn
    ...
    push %R1
    call _handleSyscall
    pop %R1
    ...
    pop %Rn
    return_from_interrupt

Executing read System Call

User space

Kernel space

User Stack

Kernel Stack

USP

UPC

KPC

KSP
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov RREAD, %R0
    syscall
    return

HandleIntrSyscall:
    push %Rn
    ...
    push %Rl
    call _handleSyscall
    pop %Rl
    ...
    pop %Rn
    return_from_interrupt
Executing `read` System Call

```c
int main(argc, argv){
    ...
    c = read(fd, buffer, nbytes)
    ...
}
```

```
_read:
    mov RREAD, %R0
    syscall
    return
```

```
HandleIntrSyscall:
    push %Rn
    ...
    push %R1
    call _handleSyscall
    pop %R1
    ...
    pop %Rn
    return_from_interrupt
```
Executing read System Call

```
int main(argc, argv) {
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov R:READ, %R0
    syscall
    return
```

```
int handleSyscall(int type) {
    switch (type) {
    case READ:
        ...
    }
}
```
int main(argc, argv) {
    ...
    c = read(fd, buffer, nbytes)
    ...
}

_read:
    mov READ, %R0
syscall
return

user space

kernel space

HandleIntrSyscall:
    push %Rn
    ...
    push %R1
    call _handleSyscall
    pop %R1
    ...
    pop %Rn
    return_from_interrupt

int handleSyscall(int type) {
    switch (type) {
    case READ: ...
    }
}
What if read needs to block?

- read may need to block if
  - It reads from a terminal
  - It reads from disk, and block is not in cache
  - It reads from a remote file server

We should run another process!
How to run multiple processes
The Problem

- Say (for simplicity) we have a single core CPU.
- A process physically runs on the CPU.
- Yet each process somehow has its own:
  - Registers
  - Memory
  - I/O Resources
- Need to multiplex/schedule to create virtual CPUs for each process.
Our friend, the Process Control Block

- A per-process data structure held by OS, with
  - location in memory (page table)
  - location of executable on disk
  - id of user executing this process (uid)
  - process identifier (pid)
  - process status (running, waiting, etc.)
  - scheduling info
  - kernel stack
  - saved kernel SP (when process is not running)
    - points into kernel stack
    - kernel stack contains saved registers (from user mode) and kernel call stack for this process
  - ...and more
Process Life Cycle

Init

Ready

Running

Waiting

Zombie
Process Life Cycle

- Init: PCB being created
- Registers: uninitialized
- Ready
- Running
- Waiting
- Zombie
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**
- **Running**
- **Zombie**

**PCB**: being created
**Registers**: uninitialized
Process Life Cycle

- **Init**: PCB on the Ready queue
- **Ready**: Admitted to the Ready queue
- **Running**: Registers pushed by kernel code onto kernel stack
- **Waiting**: Zombie

**PCB**: on the Ready queue
**Registers**: pushed by kernel code onto kernel stack
PCB: currently executing
Registers: popped from kernel stack into CPU
Process Life Cycle

- **Init**
  - Admitted to the Ready queue

- **Ready**
  - Dispatch
  - Yield

- **Running**

- **Waiting**
  - PCB: on Ready queue
  - Registers: pushed onto kernel stack (SP saved in PCB)

- **Zombie**
Process Life Cycle

- **Init**
  - Admitted to the Ready queue

- **Ready**
  - Dispatch
  - Yield

- **Running**
  - Yield

- **Waiting**

- **Zombie**

**PCB**: currently executing

**Registers**: SP restored from PCB; others restored from stack
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: blocking call e.g., read(), wait()
- **Zombie**

**PCB**: on specific waiting queue (I/O device, lock, etc.)

**Registers**: on kernel stack
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch
- **Running**: Yield
- **Waiting**: blocking call completion
- **Zombie**: blocking call e.g., read(), wait()

**PCB**: on Ready queue
**Registers**: on kernel stack
Process Life Cycle

- **Init**: Admitted to the Ready queue
- **Ready**: Dispatch, Blocking call e.g., read(), wait(), blocking call completion
- **Waiting**: Yield
- **Running**: Blocking call e.g., read(), wait()
- **Zombie**

**PCB**: currently executing
**Registers**: restored from PCB (SP) and kernel stack into CPU
Process Life Cycle

- **Init**
  - Admitted to the Ready queue
- **Ready**
  - Blocking call completion
  - Dispatch
- **Running**
  - Yield
  - Blocking call e.g., `read()`, `wait()`
  - Done `exit()`
- **Waiting**
- **Zombie**

**PCB**: on Finished queue, ultimately deleted

**Registers**: no longer needed
Invariants to keep in mind

- At most one process/core running at any time
- When CPU in user mode, current process is RUNNING and its kernel stack is empty
- If process is RUNNING
  - its PCB not on any queue
  - it is not necessarily in USER mode
- If process is READY or WAITING
  - its registers are saved at the top of its kernel/interrupt stack
  - its PCB is either
    - on the READY queue (if READY)
    - on some WAIT queue (if WAITING)
- If process is a ZOMBIE
  - its PCB is on FINISHED queue
Cleaning up Zombies

- Process cannot clean up itself
  - hard to clean up and switch without a stack!

- Process can be cleaned up
  - by some other process, checking for zombies before returning to RUNNING state
  - or by parent which waits for it
    - but what if parent turns into a zombie first?
  - or by a dedicated “reaper” process

- Linux uses a combination
  - if alive, parent cleans up child that it is waiting for
  - if parent is dead, child process is inherited by the initial process, which is continually waiting
Process Life Cycle

- **Init**
  - Admitted to the Ready queue

- **Ready**
  - Dispatch
  - Yield
  - blocking call completion

- **Running**
  - done exit()
  - blocking call e.g., read(), wait()

- **Waiting**

- **Zombie**
How to Yield/Wait?

- Must switch the “CPU state” (the context) captured in its registers and PSW
- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING → READY
  - Next process: READY → RUNNING

1. Save kernel registers of Current on its kernel stack
2. Save kernel SP of Current in its PCB
3. Restore kernel SP of Next from its PCB
4. Restore kernel registers of Next from its kernel stack