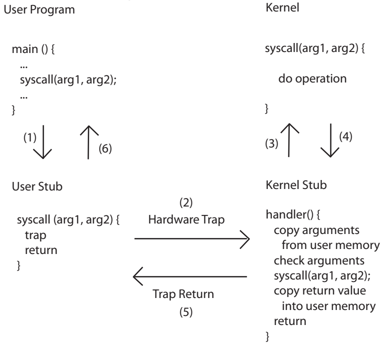
Threads (part 2)
(plus some loose ends)

Main Points

- Wrap up protection
  - – System calls and upcalls
- Wrap up threads
  - – Programming model
  - – Implementation
- Race conditions
  - – Motivation for synchronization

System Calls



User Program

```
main () {
  ...
  syscall(arg1, arg2);
  ...
}
```

(1) (6)

User Stub

```
 syscall (arg1, arg2) {
   trap
   return
 }
```

(2)
Hardware Trap

Trap Return
(5)

Kernel

```
syscall(arg1, arg2) {

    do operation

}
```

(3) (4)

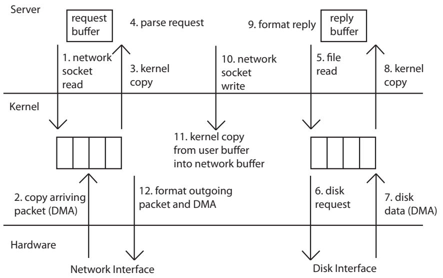
Kernel Stub

```
handler() {
  copy arguments
    from user memory
  check arguments
  syscall(arg1, arg2);
  copy return value
    into user memory
  return
}
```

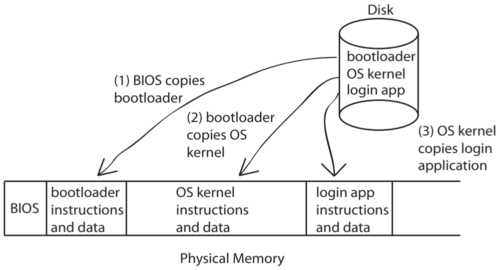
Kernel System Call Handler

- Locate arguments
  - – In registers or on user(!) stack
- Copy arguments
  - – From user memory into kernel memory
  - – Protect kernel from malicious code evading checks
- Validate arguments
  - – Protect kernel from errors in user code
- Copy results back
  - – into user memory

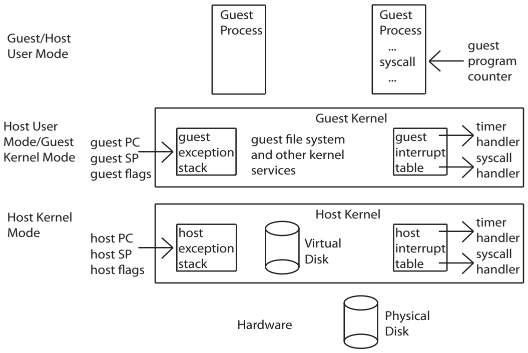
Web Server Example


Server
request buffer
4. parse request
9. format reply
reply buffer
1. network socket read
3. kernel copy
10. network socket write
5. file read
8. kernel copy
Kernel
11. kernel copy from user buffer into network buffer
2. copy arriving packet (DMA)
12. format outgoing packet and DMA
6. disk request
7. disk data (DMA)
Hardware
Network Interface
Disk Interface

Booting


Disk
bootloader
OS kernel
login app
(1) BIOS copies bootloader
(2) bootloader copies OS kernel
(3) OS kernel copies login application
BIOS
bootloader instructions and data
OS kernel instructions and data
login app instructions and data
Physical Memory

Virtual Machine


Guest/Host User Mode
Guest Process
Guest Process … syscall …
guest program counter
Host User Mode/Guest Kernel Mode
Guest Kernel
guest PC
guest SP
guest flags
guest exception stack
guest file system and other kernel services
guest interrupt table
timer handler
syscall handler
Host Kernel Mode
Host Kernel
host PC
host SP
host flags
host exception stack
Virtual Disk
host interrupt table
timer handler
syscall handler
Hardware
Physical Disk

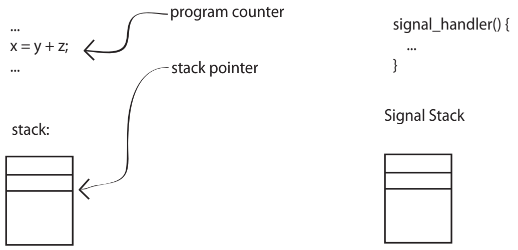
User-Level Virtual Machine

- How does VM Player work?
  - Runs as a user-level application
  - How does it catch privileged instructions, interrupts, device I/O, …
- Installs kernel driver, transparent to host kernel
  - Requires administrator privileges!
  - Modifies interrupt table to redirect to kernel VM code
  - If interrupt is for VM, upcall
  - If interrupt is for another process, reinstalls interrupt table and resumes kernel

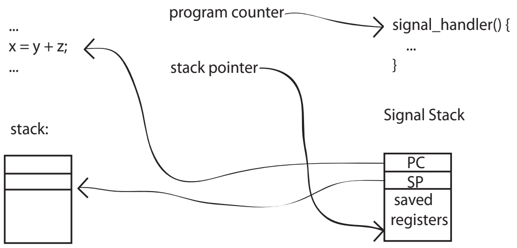
Upcall: User-level interrupt

- AKA UNIX signal
  - Notify user process of event that needs to be handled right away
    - Time-slice for user-level thread manager
    - Interrupt delivery for VM player
- Direct analogue of kernel interrupts
  - Signal handlers – fixed entry points
  - Separate signal stack
  - Automatic save/restore registers – transparent resume
  - Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: After



Last Time

- Thread use case
  - Operating systems need to be able to handle multiple things at once
    - processes, interrupts, background system maintenance
  - Servers need mtao
    - Multiple connections handled simultaneously
  - Parallel programs need mtao
    - To achieve better performance
  - Programs with user interfaces often need mtao
    - To achieve user responsiveness while doing computation
  - Network and disk bound programs need mtao
    - To hide network/disk latency

Last Time

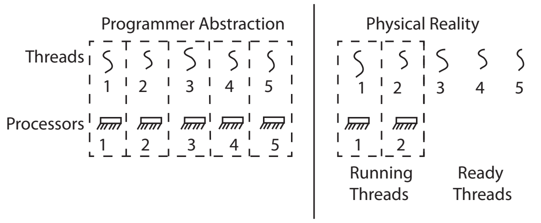
- Threads can be implemented in several ways
  - Multiple user-level threads, multiplexed onto a UNIX process (early Java)
  - Multiple single-threaded processes (early UNIX, Pintos)
  - Mixture of single and multi-threaded processes and kernel threads (Linux, MacOS, Windows)
    - To the kernel, a kernel thread and a single threaded user process look quite similar
  - Scheduler activations (Windows)

Last Time (continued)

- Thread state (thread control block)
  - Program counter
  - Stack
  - Registers
  - Priority
  - …

Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule



Thread Operations

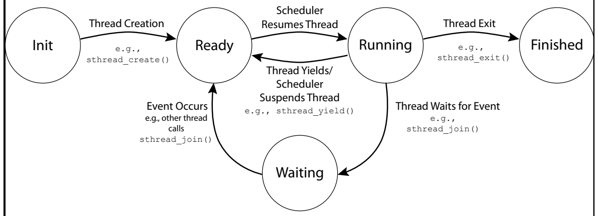
- sthread_fork(func, args)
  - Create a new thread to run func(args)
  - Pintos: thread_create
- sthread_yield()
  - Relinquish processor voluntarily
  - Pintos: thread_yield
- sthread_join(thread)
  - In parent, wait for forked thread to exit, then return
  - Pintos: tbd (see section)
- sthread_exit
  - Quit thread and clean up, wake up joiner if any
  - Pintos: thread_exit

Main: Fork 10 threads call join on them, then exit

- What other interleavings are possible?
- What is maximum # of threads running at same time?
- Minimum?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Thread States



Implementing threads

- Thread_fork(func, args)
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)
- stub(func, args)
  - Call (*func)(args)
  - Call sthread_exit()
  - Pintos: switch_entry
    - Switch_entry designed to work with switch_threads

Implementing (voluntary) thread context switch

- User-level threads in a single-threaded process
  - Save registers on old stack
  - Switch to new stack, new thread
  - Restore registers from new stack
  - Return
- Kernel threads
  - Exactly the same!
  - Pintos: thread switch always between kernel threads, not between user process and kernel thread

Pintos: switch_threads (oldT, nextT) (interrupts disabled!)

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
# This stack frame must match the
     one set up by thread_create()
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
     thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
```

```
# Change stack pointer to new
     thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (Pintos)
  - End of interrupt handler calls switch_threads()
  - When resumed, return from handler resumes kernel thread or user process
- Faster version (textbook)
  - Interrupt handler returns to saved state in TCB
  - Could be kernel thread or user process

Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library
  - Context switch in library
  - Kernel switches between processes, e.g., on system call I/O
- Option B (Linux, MacOS): use kernel threads
  - System calls for thread fork, join, exit
  - Kernel does context switching
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O