Midterm - W7-Tu

February 16, 2017 4:35 PM

Thread

- thread_fork : new threads can be created
  - new and old thread 同时运行
- thread_exit: terminate thread
- thread yield: let other thread run first
- wchan_sleep: running thread blocks
- preemption

thread 同时运行方法：

1. PCM: P processor, C cores per processor, M multhreading degree per core
2. timesharing: 好几个thread takes turn on same hardware

Context Switch:

- 从一个thread 跳到另一个thread 的转换叫做 context witch

1. 选择下一个将要运行的thread
2. sw for current thread
3. lw for next thread

i. func会造成context switch → call thread_switch → switch frame: save 当前thread regs

Thread States:

- ready: thread_yield() or preemption
- running : dispatch
- block: wchan_sleep()

Interrupts:

1. caused by system devices (timer, disk controller, network interface)
2. 做trap frame → save all regs
3. hardware 自动把 control transfer 到 a fixed memory location
4. calls interrupt handler
5. 看quantum 的 upper bound 到没到 (preemptive scheduling) → 如果到了，preempts by calling thread_yield

Synchronization

critical section

- shared object (global var) 存在的地方
- volatile 使 compiler to lw and sw every use

mutual exclusion

- Mutual Exclusion: 确保只有一个 thread 在 critical section
- 方法： use lock

test and set

- atomic: indivisible operation
- ex. Xchg

  test: return old value of lock (true: locked, false: free)

set: set a new value to lock

Other synchronization instruction

- Load-linked / Store-conditional
  - ll: return current value
  - sc: store new value iff 别人没碰过memory location (lock) since ll
- SPARC:
  - cas addr, R1, R2
  - value in addr == R1, then swap value of addr & R2
- compare-and-swap:
  - = cas addr, expectedval, newval;
  - return old val of addr.

spinlocks

- it's a loop to test if look is free;
  - ex. while (Xchg (true, lock) == true);
- spinlock_acquire: calls (spinlock_data_testandset) in a loop until the lock is acquired

blocking and blocking locks

- lock_acquire: thread blocks until lock is free
- blocks: thread stop running
  - scheduler chooses new thread to run
  - context switch happens
  - blocking thread 在 wait queue 里面
  - blocking thread goes to ready if it's signaled and awakened by another
- wchan_lock:

semaphores

- cause mutual exclusion, & solve other synch problems
- an int , two operations:
  - initial int = 1;
  - P: if (int > 0) {int -- ;}
  - V: int++;
- atomic

condition variables

- cv only used within critical section that is protected by lock
- operations:
  - wait: cause thread to block → release lock →onece unblock→reacquires lock
  - signal: one of blocked threads unblocked
  - broadcast: unblocks all thread
- no matter it calls wait, or wait returns, it's in the critical section, but in between, if thread blocks, then out of critical section. other may enter.
- one calls signal (it's in critical sec) → waiting thread wait until signaller release thelock → unblock and return from wait call

deadlocks

- neither thread can make progress
  - ex. lock A B are free
    - thread1 acquire lock A → acquire lock B
    - thread2 acquire lock B → acquire lock A
- Prevention:
  - no hold and wait
    - if (A holds resources) {cannot ask for more}
    - if (A wanna holds several resources) {make single request for them}

        - resource ordering
            - order resource type
            - if (A holds i) {A cannot ask for j <= i}

Processes and System Calls

process

- an environment of user application
    - created & managed by kernel
    - each process isolates
- includes:
    - threads (in os161 only one)
    - virtual memory for user's program's code and data
    - others: file, etc.

system call

- interfaces between processes and kernel
- kernel privilege
    - allows kernel to keep processes isolated from (process & kernel)
- How System Calls work
    - interrupts -> interrupt handler (in kernel)
    - exceptions -> detected by CPU -> control transferred to fixed location -> calls exception handler (a part of kernel)
- syscall
    - cause exception on MIPS (EX_SYS)
    - steps
        - application calls lib wrapper for desired syscall
        - lib func perform syscall
            - trap frame
            - exception handler checks exception code (check exception type)
            - determine which syscall (application store the syscall func code in specified location, excep handler check this code)
            - restores -> return from exception
        - lib func finish and return
- syscall parameters
    - parameter values in kernel-specified location
    - looks for return values in specified location

stacks

- user stack
    - in virtual memory used when user code is running
    - kernel sets it when it sets up virtual memo
- kernel stack
    - used when thread is executing kernel code
    - holds trap frames and switch frames

processor exception

- step:
    - saves user stack point
    - switch stack pointer to kernel stack
    - saves (user state & addr of instruction that was interrupted) in trap frame
    - calls mips_trap(pointer)
        - determines types of exception
        - call different handler for different exception
    - restore

fork/execv

- fork
  - create child process (克隆parent)
  - both running copies of same program, have same virtual memories
  - return in parent (pid) & return in child (0)
- exit
  - process supply an exit code
  - kernel records the code (another process ask for the code via waitpid)
- waitpid lets a process wait for another to terminates, then get the code
- execv
  - changes the program that a process is running
  - process' virtual memo is destroyed, get new virtual memo
  - initialized with code and data of new program

multiprocessing

- multiple processes share available hardware resources
  - each's virtual memo implemented using physical memo
  - each process' threads in available CPUs
  - share access to disk, network devices etc.

Virtual Memory

virtual memory

- kernel provide virtual memo for each process
- it holds the code, data, and stack for program
- virtual addresses are V bits, virtual memory is 2^V
- user only see virtual addresses
  - program counter and stack pointer hold virtual addresses of next instruction

physical memory

- have P bits -> addressable physical memory is 2^p bytes

address translation

- each vm is mapped to a different part of pm
- when one try to access (lw,sw) a vm, it translates to pm
- addr translation is performed in hardware

MMU

- memo management unit, a hardware
- has relocation register & limit register
  - relocation: hold offset (R)
  - limit: holds the size (L)
  - it changes the value of R,L when context switch
  - translate v to p

    if $v \geq L$

        exception

    else

        $p = v + R$
- page table base register: points to page table
  - determine page # & offset of v-addr
  - looks up PTE- page entry

    if (PTE not valid)

        exception

    else

        p = frame # * frame size + offset
  - PTEs may contain other filds (write rpoelds (write protection bit etc.)

        - dirty bit: have contents been changed

TLB

- translation lookaside buffer in MMU, MMU handles miss
- each TLE entry stores a (page# -> frame#) mapping
- hardware-managed TLB
  if (p,f) entry exist
      return f
  else
      find p's frame number f from page table, add (p,f)
      return f
- software-managed TLB
    - if (p,f) not exist, exception
    - Kernel determine frame number, and if needs to evicting
    - after handled miss, instruction retried

relocation

- each v-addr translate to 连续的一块 p-addr
- kernel decide where mapping to

paging

- pm divide to fixed size chunks called frames
- vm divide to fixed size called pages
- each page maps to different frames use MMU
- page table
    - page table size = # of pages * size of PTE
    - page tables are kernel data structure

segmentation

- kernel maintains an offset and limit for each segment
- MMU has multiple offset and limit registers, one pair for each
- v-addr can be thought of two parts: k bit segment ID, offset within segment
    - 2^k segments
    - 2^{v-k} bytes per segment

Multi-Level Paging

- Two level paging
    - add page table directory
    - if all PTE in smaller table are invalid, we can avoid creating that table
    - each v-addr has three parts:
        - level one page number -> index of directory
        - L-2 page number -> index a page table
        - offset
- limits of L-2 paging
    - page table's size <= page size
- Properties
    - add more levels to deal with larger vm
    - page tables can remain small
    - TLB miss become more expensive

OS161 on mips: dumbvm

- 32-bit v & p addr
- software-managed TLB
- TLB exception is handled by vm_fault
    - vm_fault use addrspace struct to determine (p,f)

executable file

- program's code and data is described in it
- it's in ELF(executable and linking format)
    - contain addr space segment description
    - identifies the virtual address of first instruction