Summary

Saturday, October 8, 2022

Thread

- A thread is a **sequence of instructions**
- A normal **sequential program** consists of a single thread of execution
- In threaded concurrent programs:
  - Multiple threads of execution, all occurring at the same time
  - Threads may perform the same task, or
  - Threads may perform different tasks
- Express **concurrency**
  - Multiple programs or sequences of instructions running, or appearing to run, at the same time

Why Thread?

- Resource utilization
  - Blocked threads give up resources to others
- Parallelism
  - Multiple threads executing simultaneously
- Responsiveness
  - Dedicate threads to UI
  - Other to loading/long tasks
- Priority
  - Higher priority - more CPU time
  - Lower priority - less CPU time
- Modulization
  - Organization of execution tasks

OS/161 Thread Concurrency

- A thread can create new threads using **thread_fork**
- All threads share access to the program's global variables and heap
- Each thread has its own stack
  - Private to that thread
- A thread is represented as a structure or object in the OS

OS/161 Thread Interface

- Create a new thread:

```
int thread_fork(
        const char *name,
        struct proc *proc,
        void (*func) (void *, unsigned long),
        void *data1,
        unsigned long data2,
)
```

- Terminate the calling thread:

```
void thread_exit()
```

- Voluntarily yield execution:

```
void thread_yield()
```

- Notes about thread_yield
  - Mainly used in kernel functions
  - Rarely used in user programs
  - No guarantee that CPU will be given up

Timesharing and Context Switches

- Timesharing
  - Each thread gets a small amount of time to execute on the CPU
  - When expired, a **context switch** occurs
  - Threads share the CPU, giving the user the illusion of multiple programs running at the same time
- Context switch
  - The switch from one thread to another is called a **context switch**
  - During context switch
    - Decide which thread runs next
    - Save register contents of the current thread
    - Load register contents of the next thread

thread_switch and switchframe_switch

- **switchframe_switch**: the low level assembly function called by thread_switch
  - It only needs to save the callee-save registers (s0-s7)
  - Steps:
    i. Allocate stack space for saving 10 registers
    ii. Save ra, gp, s8/fp and s0-s7
    iii. Save sp to a0
    iv. Load sp from a1, switching to the stack of the new thread
    v. Restore s0-s7, s8/fp, gp and ra, which stores the address of thread_switch of the new thread
    vi. Return
- **thread_switch**: the high level C function that calls switchframe_switch
  - It only needs to save the caller-save registers (t0-t7)

```
/* See kern/arch/mips/thread/switch.S */

switchframe_switch:
  /* a0: address of switchframe pointer of old thread. */
  /* a1: address of switchframe pointer of new thread. */

   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -40

   sw   ra, 36(sp)  /* Save the registers */
   sw   gp, 32(sp)
   sw   s8, 28(sp)
   sw   s6, 24(sp)
   sw   s5, 20(sp)
   sw   s4, 16(sp)
   sw   s3, 12(sp)
   sw   s2, 8(sp)
```

```
    sw   s4, 16(sp)
    sw   s3, 12(sp)
    sw   s2, 8(sp)
    sw   s1, 4(sp)
    sw   s0, 0(sp)

    /* Store the old stack pointer in the old thread */
    sw   sp, 0(a0)


  /* Get the new stack pointer from the new thread */
  lw   sp, 0(a1)
  nop           /* delay slot for load */

  /* Now, restore the registers */
  lw   s0, 0(sp)
  lw   s1, 4(sp)
  lw   s2, 8(sp)
  lw   s3, 12(sp)
  lw   s4, 16(sp)
  lw   s5, 20(sp)
  lw   s6, 24(sp)
  lw   s8, 28(sp)
  lw   gp, 32(sp)
  lw   ra, 36(sp)
  nop                   /* delay slot for load */

  /* and return. */
  j ra
  addi sp, sp, 40       /* in delay slot */
  .end switchframe_switch
```

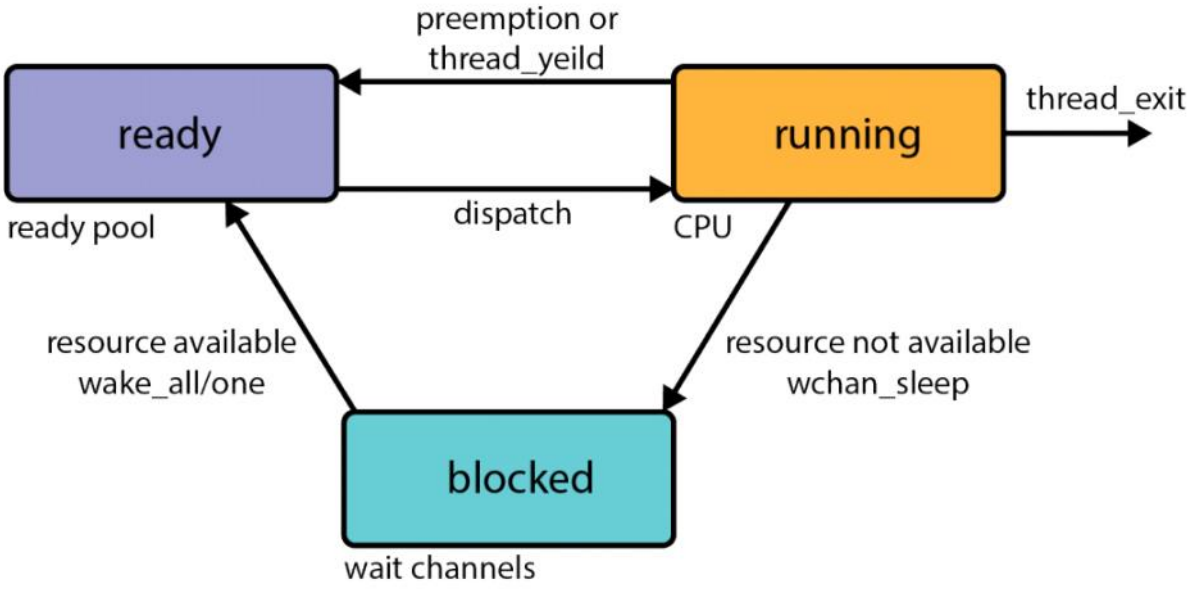
What causes context switches?

- thread_yield
  - Thread voluntarily allows other threads to run
- thread_exit
  - Thread terminates
- wchan_sleep
  - Thread blocks (sleeps)
- Preemption
  - Thread involuntarily stops running

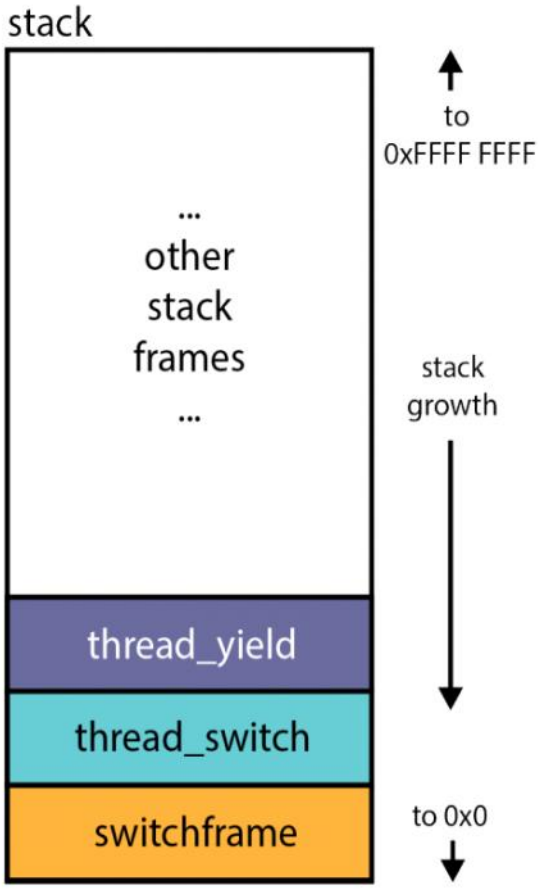
Thread states

- Running
  - Currently executing instructions
- Ready
  - Ready to execute instructions but is not
- Block

- Not ready
- Waiting for resource



OS/161 Thread Stack after Voluntary Context Switch



Timesharing and Preemption

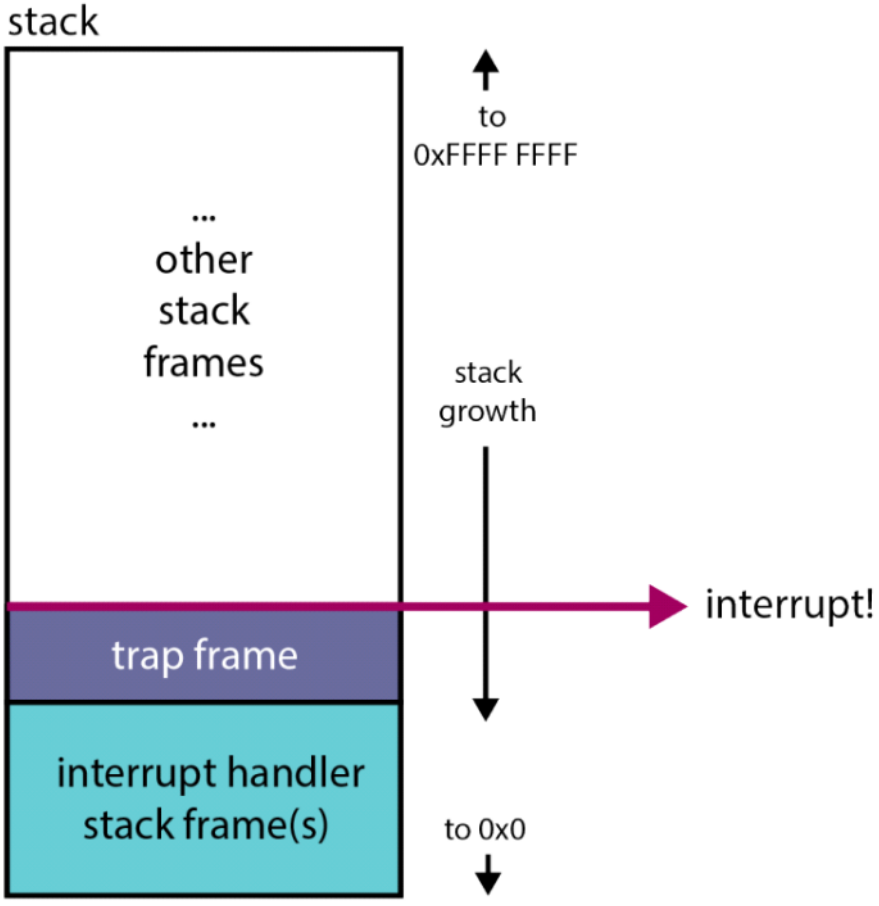
- Scheduling quantum

    - A limit on CPU time for each thread
    - An upper bound on how long a thread can run before it must yield the CPU
    - What if a running thread never yields, blocks or exits when the quantum expires?
- **Preemption**
    - Forces a running thread to stop running, so that another thread can have a chance to run
    - Normally accomplished using **interrupts**

Interrupts

- An event the occurs during the execution of a program
- Caused by system devices (hardware)
- Hardware automatically transfers control to a fixed location in memory, which contains a procedure called an **interrupt handler**
- The interrupt handler
    - Creates a **trap frame** to store the thread context at the time of the interrupt (every register is saved)
    - Determines which device caused the interrupt
    - Performs device-specific processing
    - Restores the saved thread context from the trap frame

OS/161 Thread stack after an Interrupt



Preemptive Scheduling

- Uses the **scheduling quantum** to impose a time limit on running threads

- If a thread has run too long
    - The timer interrupt handler preempts the thread by calling thread_yield
    - That thread changes state from running to **ready**, and is placed on the **ready queue**
    - The runtime starts at 0 when it is running again
- OS/161 threads use **preemptive round-robin scheduling**

Lecture 6 - Spinlock, Lock and Semaphore

Tuesday, September 27, 2022

Spinlock

- NEVER sleep when you own a spinlock
- Everyone else might try to acquire the spinlock but only you can release that spinlock
- Interrupts are disabled and deadlocks happen

ARM Synchronization Instructions

- LDREX and STREX should be used as close as possible to acquire context change
- If STREX failed return TRUE to keep running ARMTestAndSet
- If STREX succeeded, return the original value at addr, can be TRUE or FALSE
- ARMTestAndSet returns FALSE iff the lock was free and we acquired the lock, causing Acquire to break

MIPS Synchronization Instructions

- Similar to ARM
- Instructions used are called ll and sc

Spinlocks

- Shouldn't be used for a large amount of time
- Inefficient
- Use busy wait and disable interrupts
- The owner is the core itself, not a thread

Locks

- Locks have names, just for documentation / debug, not guaranteed to be unique
- If lock unavailable after calling lock_acquire, go to sleep instead of busy waiting
- Can be used to protect larger critical sections without being a burden on the CPU
- A type of mutex
- Have owners

Wait Channels

- wchan_lock uses a spinlock

Semaphore

- A synchronization primitive that keeps a count
- P: decrement the if it is greater than 0
- V: increment the value
- P and V are atomic

Lecture 7 - Condition Variable

Thursday, September 29, 2022

How do you make sure lock_acquire and lock_release are atomic?

- Using spinlocks

Semaphores

- No ownership
- Does not matter as long as you use it correctly
  - Only call V after having called P
- When used as a barrier semaphore
  - Similar to pthread_join, but different since you don't specify which thread to wait for

Condition Variable

- Wrapper for wchans
- It has no idea what condition it is keeping
- Programmers must have a lock when using condition variables
- CV assumes the lock has already been acquired, then release the lock, go to sleep, and when the condition is satisfied, it will be waken up and reacquire the lock
  - This is all achieved by cv_wait(cv, lock)
- How does it wake up?
  - Another thread calls cv_signal(cv, lock) when the condition has been satisfied
  - Can also call cv_broadcast(cv, lock) which wakes up **all threads** waking for this condition to be true

```
volatile int foo;
lock mutex;
cv foo10and50;

void bar (void *p1, unsigned long p2) {
        lock_acquire(mutex);
        while (!(foo > 10 && foo < 50)) {
                lock_release(mutex);
                thread_yield();
                lock_acquire(mutex);
        }
}
```

Instead of voluntarily yielding, it's probably better to go to sleep when you're waiting condition to be true.

Conditional Variable

- Why use a while loop?
  - When a thread wakes up. it does not start running immediately, but in a ready state. Other threads can mutate the global variable associated with the condition again

Volatile

- Write back a global variable immediately, instead of caching it in a register

Summary

Tuesday, October 11, 2022

Race Condition

- A **race condition** is when the program result depends on the order of execution
- Occurs when multiple threads are reading and writing the same memory at the same time
- Constants and memory that all threads only read, do not cause race conditions

Atomic test-and-set

- Atomic x86 (and x64) **xchg** instruction
  xchg src, addr
- src is a register, addr is a memory address
- Swaps the values stored in src and addr
- Logical behaviour:

```
Xchg(value,addr) {
  old = *addr;
  *addr = value;
  return(old);
}
```

X86 Lock Acquire and Release with xchg

```
Acquire(bool *lock) {
  while (Xchg(true,lock) == true) ;
}

Release(bool *lock) {
  *lock = false;            /* give up the lock */
}
```

X86 Lock Acquire and Release with xchg

- If xchg returns true, the lock was already set and we continue to loop
- If xchg returns false, then the lock was free, and we have required it
- This construct is known as a **spin lock**
  - A thread busy-waits in Acquire until the lock is free

ARM Synchronization Instructions

- **Exclusive load (LDREX)** and **exclusive store (STREX)**
- Act as a barrier
- Must be used together
- LDREX
  - Loads a value from address addr
- STREX
  - Attempt to store a value to address addr
  - Fail to store value at addr if addr was touched between LDREX and STREX
- It is recommended to place these instructions close together
  - Lower change for context switches

```
ARMTestAndSet(addr, value) {
        tmp = LDREX addr // load value
        result = STREX value, addr  // store new value
        if (result == SUCCEED) return tmp
        return TRUE
}

Acquire(bool *lock) {
        while( ARMTestAndSet(lock, true) == true ) {};
}
```

Spinlocks in OS/161

- A **spinlock** is a lock that repeatedly test lock availability in a loop until the lock is available
- Actively use the CPU while busy-waiting
- Interrupts are turned off

Locks in OS/161

- A thread that calls lock_acquire blocks until the lock can be acquire
- Used to protect larger critical sections without being a burden on the CPU
- A type of **mutex**
- Have owners

Lecture 8 - Processes and the Kernel

Tuesday, October 4, 2022

Process

- Execution environment for running programs
    - Address space
    - Threads
    - Virtualized resource - heap, stack
    - File and socket descriptors
- Isolate from other programs in other processes

Process Structure

- Name
    - For debugging purposes
    - Not a unique identifier
- Spinlock
    - For synchronization purposes
- Thread array
    - Array of all threads of this process
- Address space
    - Virtual memory for this process
- Current working directory
- Console

Process Management Calls

- fork
    - Create a new process with a separate copy of code and address space
- _exit
    - Terminate the calling process
- waitpid
    - Similar to pthread_join
    - Wait for another process to finish, given its pid
    - Get the return value (exit status) of that process
    - Get information about how that process is terminated

CPU Cycle

- Correct order of fetch/execute cycle
    - Fetch instruction
    - Increment PC
    - Execute instruction
- Note that the order is important

fork

- Creates a new process (the child) that is a clone of the original (the parent)
- Return both in the parent and child
- Different return values for parent and child
    - Parent gets child's pid

    - Child gets 0
- After fork, both parent and child execute copies of the same program
    - Exact copies but separate from each other (do not share the address space)
    - May diverge later (e.g. through an if-else statement)

execv

- Does not return

Lecture 9 - System Calls, Multiprocessing

Thursday, October 6, 2022 13:00

execv

- Destroys the current program's virtual memory
- Initialize with code and data of the new program and start running it
- Does not return to the previous process since it has been destroyed
    - Only return when it failed to start running the new program
- Does not change parent-child relationships

System Call

- The OS needs to protect critical parts from malicious programs
- System calls allow user programs to ask OS to perform certain tasks
    - Create, destroy, manage processes
    - Create, destroy, read, write files
    - Manage file system
    - Manage virtual memory

Unprivileged vs. Privileged code

- CPU has different modes (privilege levels or rings)
- Ring 0 - highest security
    - Kernel code

How System Calls Work

- There are two things that make kernel code run
    - Interrupts - caused by hardware devices
    - Exceptions - conditions that occur during the execution of a program instruction
- System calls are special instructions that produce exceptions
- In MIPS, interrupts treated as exceptions
    - The EX_SYS exception is the syscall

mips_traps

- Create a trapframe
- Determine which exception is raised
- If the exception is a syscall, invoke syscall, passing the system call code in the trap frame

syscall

- Get v0, which contains the system call code (callno)
- Check the callno
    - See which system call the user program wants to execute
- Update program counter in the trap frame
    - Syscall generated an exception so the PC would not update
    - Avoid restarting the syscall over and over again
- syscall.h contains which system call code is expected in v0 for each syscall
- In MIPS, parameters are expected in a0-a3

System Call Stack

- Each thread has 2 stacks: user stack and kernel stack

- User stack
    - Used while application code is executing
    - Located in the application's virtual memory
    - The kernel creates this stack when it sets up the virtual address memory for the process
- Kernel stack
    - Used while the thread is executing kernel code, after an exception or interrupt
    - A kernel structure
    - In OS/161, the pointer is called t_stack
    - Holds trap frames and switch frames

Inter-Process Communication (IPC)

- Processes do not share address spaces. How do they share data?
- Methods used to send data between processes
    - File
    - Socket
    - Pipe
    - Shared memory
    - Message passing/queue

Summary

Tuesday, October 11, 2022

Process

- A **process** is an environment in which an application program runs
- Includes virtualized resources that its program uses
    - Threads
    - Virtual memory
    - File and socket descriptors
- Created and managed by the kernel
- Isolates from other programs in other processes
    - Seems to have exclusive access to the processor, RAM and I/O devices when in fact they are shared

fork

- Creates a new process that is a clone of the parent (the original process)
- Makes a new thread for child
- The address spaces (code, global, heap, stack) are identical at the time of the fork
    - May diverge afterwards
- Returns in both the parent and child
- Different return values from fork
    - Returns 0 to the child process
    - Returns the child's pid to the parent process
    - Provides a way to distinguish the child from the parent

_exit

- Terminates the process that calls it
    - A process can supply an exit status code when it exits
    - The kernel records the exit status code in case another process waits for it (e.g. waitpid)

waitpid

- Waits for another to terminate
- Retrieves its exit status code

execv

- Changes the program that a process is running
- The calling process
    - Destroys the current virtual memory
    - Obtains a new virtual memory
    - Initializes it with the code and data of the new program to run
- These are not changed:
    - Process ID
    - Parent/child relationships
- Can pass arguments to the new program, if required

Example

```
//   sample code that uses fork, getpid, waitpid and _exit

 1.  main() {
 2.     int rc = fork();    // returns 0 to child, pid to parent
 3.     if (rc == 0) {      // child executes this code
 4.        my_pid = getpid();
 5.        x = child_code();
 6.        _exit(x);
 7.     } else {            // parent executes this code
 8.       child_pid = rc;
 9.       parent_pid = getpid();
10.       parent_code();
11.       p = waitpid(child_pid,&child_exit,0);
12.       if (WIFEXITED(child_exit))
13.         printf("child exit status was %d",
              WEXITSTATUS(child_exit))
14.     }
15.  }
```

- Line 12: WIFEXITED returns true if the child called _exit()
- Line 13: WEXITSTATUS extracts the exit code
- These two macros are defined in kern/include/kern/wait.h

getpid

- Returns the **process identifier** (**pid**) of the current process
- Each existing process has a unique pid

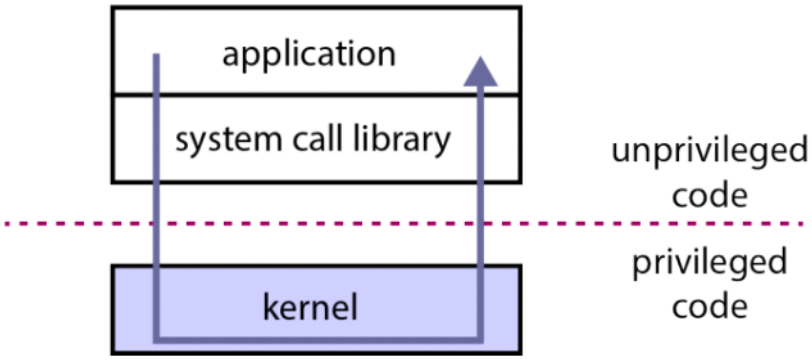
Example

```
//    sample code that uses execv to execute the command:
//    /testbin/argtest first second

      int main()  {
 1.     int status = 0;  // status of execv function call
 2.     char *args[4];   // argument vector

                         // prepare the arguments
 3.     args[0] = (char *) "/testbin/argtest";
 4.     args[1] = (char *) "first";
 5.     args[2] = (char *) "second";
 6.     args[3] = 0;     // end of args

 7.     status = execv("/testbin/argtest", args);
 8.     printf("If you see this output then execv failed");
 9.     printf("status = %d errno = %d", status, errno);
10.     exit(0);
      }
```

- Line 8:
  - If execv fails ⇒ the current program will continue executing
  - If execv succeeds ⇒ the current program will be replaced with argtest
    - The printf statement on line 8 will not be executed since the code section is also destroyed
- Line 9: errno is a global variable that holds the value of the last error number

System calls

- System calls are the interface between user processes and the kernel
- Called by user programs
- E.g. fork, execv, waitpid, getpid



Kernel privilege

- Different levels (or rings) of execution privilege
- Kernel code runs at the highest privilege level
  - Any processor instructions can be executed
- Application (user) code runs at a lower privilege level
  - User programs should not be permitted to perform certain tasks such as
    - Modifying the page tables used to implement virtual memories
    - Halting the CPU
  - Cannot directly call kernel functions or access kernel data

How system calls work

- Interrupts
  - Generated by devices when they need attention
- Exceptions
  - Caused by instruction execution when a running program needs attention

Interrupts

- Raised by devices
- Cause the processor to transfer control to a fixed location in memory
  - An **interrupt handler** is located here
- When an interrupt occurs, the processor
  - Switches to privileged mode
  - Transfers control to the **interrupt handler** - a part of the kernel
- This is how the kernel gets its execution privilege

Exceptions

- Conditions that occur during the execution of a program instruction
- Detected by the processor during instruction executions
- When an exception occurs, the processor
  - Switches to privileged mode
  - Transfers control to the **exception handler** - a part of the kernel
- In MIPS, everything (including an interrupt) is an exception

System call

- The kernel
  - Expects the application to place the appropriate system call code (in v0 for OS/161 on MIPS processor)
  - Checks this code to determine which system call
- The codes and code location are part of the kernel's **Application Binary Interface** (**ABI**)
- Arguments go in registers a0, a1, a2, a3
  - Result success/fail code is in a3 upon return
  - Return value or error code is in v0 upon return
- System calls are expensive
  - Call print 10000 times to print one character at a time v.s.
  - Call print once to print 10000 characters at a time

User and kernel stacks

- User (application) stack
  - Used while the thread is executing application code
  - Holds the stack frames for the application's functions
  - Created by the kernel when it sets up the virtual address space for the process
- Kernel stack
  - Used while the thread is executing kernel code
    - i.e. after an exception or interrupt
  - A kernel structure
  - Holds stack frames for the kernel functions
  - Holds trap frames and switch frames

Exception handling in OS/161

1. An application calls a library function
   - Puts the system call code in register v0
   - Raises an exception
2. common_exception
   - Saves the application's stack pointer
   - Switch the stack pointer to point to the kernel stack
   - Saves application state (including the pc) in a trap frame on the kernel stack
   - Calls mips_trap, passing a pointer to the trap frame as an argument
3. mips_trap
   - Determines the type of exception by looking at the exception code
     - Interrupt? Call mainbus_interrupt
     - System call? Call syscall (kernel function)
     - ...

4. Do the work to handle the exception
5. common_exception
    - Restore application state (including the application stack pointer) from the trap frame on the kernel stack
    - Go back to the application instruction that was interrupted
    - Switch back to unprivileged execution mode

Why have system calls?

- Ensure that the kernel is protected and have the highest privilege
- Keep the kernel isolated from applications
- Allow the application processes to use the services of the kernel

Multiprocessing

- Having multiple processes existing at the same time
- All processes must share the available hardware resources
    - Sharing is managed by the OS
- The OS ensures that processes are isolated from one another

Inter-Process Communication (IPC)

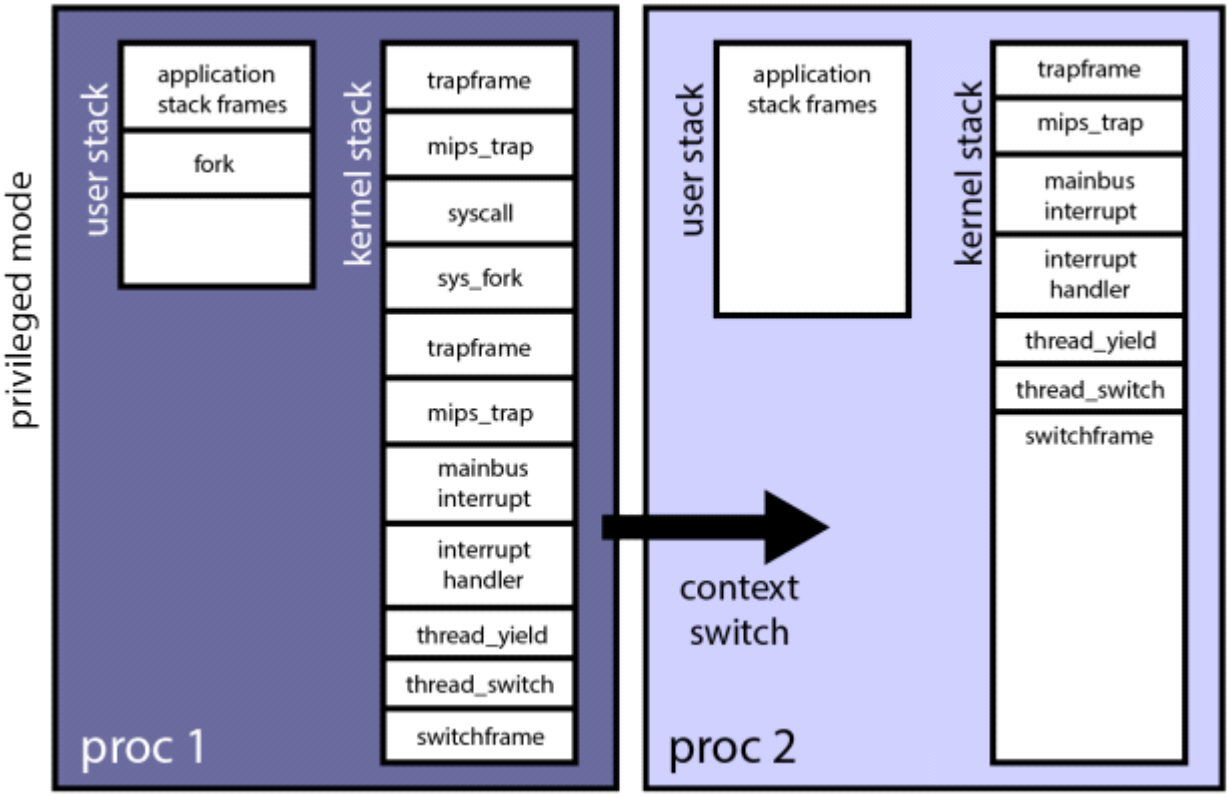
- A family of methods used to send data between processes
- File
    - Accessed by both processes
- Sockets
    - Network interface between processes
- Pipe
    - Unidirectional data transfer from one process to another via OS-managed data buffer
- Shared memory
    - Block of shared memory visible to both processes
- Message passing/queue
    - A queue/data stream provided by the OS

Summary: System Calls

- Implemented by putting a system call code in a particular register (in v0 for OS/161 on MIPS) then raising an exception with the assembly language instruction syscall
- The processor jumps to a fixed location that has the **exception handler**
- The exception handler
    1. Creates a trap frame to save application state
    2. Determines that this is a system call exception
    3. Determines which system call is requested
    4. Does the work of the system call
    5. Restores the application state from the trap frame
    6. Returns from the exception

Example diagram

- A system call (for fork) followed by a timer interrupt


privileged mode
user stack
application stack frames
fork
kernel stack
trapframe
mips_trap
syscall
sys_fork
trapframe
mips_trap
mainbus interrupt
interrupt handler
thread_yield
thread_switch
switchframe
proc 1
context switch
user stack
application stack frames
kernel stack
trapframe
mips_trap
mainbus interrupt
interrupt handler
thread_yield
thread_switch
switchframe
proc 2

Lecture 11 - Intro to Virtual Memory

Thursday, October 20, 2022

Dynamic Relocation

- Accessing a virtual address larger than the limit for the process
    - Raise an exception
- Otherwise add the relocation register to the virtual address asked for to get the physical address
- Efficient but suffers from *fragmentation*

Fragmentation

- External
    - Empty spaces between contiguous blocks memory
- Internal
    - Empty spaces within blocks of memory

Lecture 12 - Segments, Paging

Tuesday, October 25, 2022

Approach 1

- MMU has a relocation register and a limit register for each segment

Approach 2 - Maintain a segment table

- MMU will only have 2 registers
  - Segment table length register - stores the number of segments
  - Segment table base register - stores a pointer to the beginning of the segment table
- Pros
  - Store more information than Approach 1, e.g. read/write protection
  - More flexible than Approach 1
- Steps:
  1. Check given segment number
  2. Index into the segment table and find the physical address

Segment problems

- Still suffers from fragmentation

Paging: physical memory

- External fragmentation is impossible
- Minimize internal fragmentation
- Smaller "segments"
  - Also need a table

Page table

- Valid bit
  - Indicate whether a paged is actually allocated to a process
- Read/write bit
- MMU only needs one register to store a pointer to the beginning of the page table
- Example (page size = 2^{12})
  - Lower 12 bits for page offset
  - Higher k bits for page number (k = # virtual address bits - 12)

Page table address translation

- # of bits for offset = log(page size)
- # of PTEs = maximum virtual memory size / page size
- # bits for page number = log(number of PTEs)

Multi-level paging

- Split the page table into multiple levels
- A large, contiguous table is replaced with multiple smaller tables
  - Each fitting onto a single page

Lecture 13 - Multi-level Paging

Thursday, October 27, 2022

Multi-level paging

- The first page table becomes a page directory
  - Stores pointers to other page tables
  - E.g. first entry stores a pointer to a page table containing addresses 0…999
- Valid bits indicate whether *at least* one page in the range of pages are allocated
- We can have more than two levels

- A tree of page directories / tables
  - The root is the first page directory
  - The leaves are the actual page tables
- Accessing page table is $O(\log(n))$
- Time-space trade off
  - Larger time complexity
  - Less space usage

Multi-level paging address translation

- E.g. 0x502C on slide 26
  - 5 = 0101 in binary
  - Split 0101 into 01 and 01
  - The first 01 means access index 1 of the root directory, which points to Table 2
  - The second 01 means access index 1 of Page Table 2 to get the frame number 0x12
  - Result: 0x1202C
- E.g. 0x70A4 on slide 26
  - 7 = 0111 in binary
  - Split 0111 into 01 and 11
  - The 01 means access index 1 of the root directory, which points to Table 2
  - The 11 means access index 3 of the Page Table 2 to get the frame number 0x13
  - Result: 0x130A4
- Note: do not forget to check the valid bits

Multi-level paging (continued)

- MMU only needs one page table base register
  - Points to the root page table directory for the current process
- Size of page table = page size

In class problems

- V = 64, page size = 2^{20}, PTE = 2^4
- Each individual page table, at each level, must fit in a single frame
- How many bits of each virtual address are needed to represent the page offset?
  - log(page size) = 20
- What is the maximum number of entries in an individual page table?

$$\frac{\text{page table size}}{\text{PTE Size}} = \frac{2^{20}}{2^4} = 2^{16}$$

- How many pages can be at the bottom level?

- We needed 2^{44} pages ⇒ the number of page tables we need is

$$\frac{2^{44}}{2^{16}} = 2^{28}$$

- 64 - 20 = 44 remaining bits to be used for indices

$$\left\lceil \frac{\text{\# remaining bits for indices}}{\text{\# page table entries}} \right\rceil = \left\lceil \frac{44}{16} \right\rceil = 3$$

- Suppose that a particular process uses only 128 MB (2^27 bytes) of virtual memory, with a virtual address range from 0 to 2^27 - 1. How many individual page tables, at each level, will be required to translate this process' address space?
  - 2^7 pages used
  - One page table needed at the bottom level
  - $3 \cdot 2^{20}$= 3 MB used for page tables

Roles of the Kernel and the MMU

- Kernel
  - Manage MMU registers during context switch from a thread in one process to a thread in a different process
  - Create and manage page tables
  - Manage (allocate/deallocate) physical memory
  - Handle exceptions raised by the MMU
- MMU (hardware)
  - Translate virtual addresses to physical addresses
  - Check for and raise exceptions when necessary

Translation Lookaside Buffer (TLB)

- Motivation: want to cache the page tables
- TLB miss - access a page table that has not been loaded in to the cache
- Only store PTEs, not page tables

Midterm Range

- Everything up to single-level paging

Lecture 14 - TLB

Tuesday, November 8, 2022

TLBs

- Small, fast, dedicated cache of address translations
- The MMU now does not need to know the layout of the page tables since it only reads TLB entries
- The OS has more freedom to decide how to store the page tables
- Much faster than accessing memory
- Issue
    - TLB entries for one process need to be flushed before switching to another process
    - They do not need to be flushed if a new thread is added (since they share the same address space)

Software-managed TLB

- MMU only looks at the TLB; raises an exception upon TLB miss
- TLB hit and TLB miss
    - Similar as cache hit and cache miss

Paging conclusion

- Does not introduce external fragmentation since every allocation is the same size
- Multi-level paging reduces the amount of memory required to store page-to-frame mappings
- TLB misses are increasingly expensive with deeper page tables
    - A TLB miss for a three-level page tables requires three memory accesses

Lecture 15 - ELF, Kernel Memory

Thursday, November 10, 2022

Coremap

- Stores which frame is in use and which process is using it

OS/161 Memory

- The first 2GB of the virtual memory belongs to user
- The rest belongs to kernel
- MMU can
    - Check if a user is trying to access kernel memory when we're not in privileged mode
    - Use different translation methods based whether an address is a user address and kernel address
    - Use direct relocation for kernel addresses
- kseg0 and kseg1 use direct relocation; maps to the first 512 MB of memory
    - E.g. 0x8500 0000 - 0x8000 0000 = 0x500 0000 in physical memory
    - E.g. 0xA001 0A80 - 0xA000 0000 = 0x10A80 in physical memory
- Problem
    - kseg0, kseg1 and user can map to the same physical address

In-class problem

- Physical address: 0x0111 3FA0
    - kseg0: 0x8111 3FA0
    - kseg1: 0xA111 3FA0
    - User: 0x7FFF EFA0

Exploiting secondary storage

- On-demand paging
    - Some pages are not in memory, but in disk instead
- Resident set - the set of virtual pages present in physical memory
- Present bit - track which pages are in physical address
- We should not put pages that are not present in TLB

Lecture 16 - Page Replacement Algorithms

Tuesday, November 15, 2022

FIFO page replacement policy

- Replace the page that has been in memory the longest
- Simple policy, but not necessary the best one

Optimal page replacement

- MIN: replace the page that will not be referenced for the longest time
- Requires knowledge of the future

Locality

- Temporal locality: programs are more likely to access pages that they have accessed recently than pages that they have not accessed recently
- Spacial locality: programs are likely to access parts of memory that are close to parts of memory they have accessed recently

Least Recently Used page replacement policy

- Replies on temporal locality
- In practice much better than FIFO

Clock Replacement Algorith

- One of the simplest algorithms that exploits the use bit
- A victim pointer that cycles through the page frames
- Moves whenever a replace is necessary
- OS keeps the victim pointer
- MMU sets the use bit upon access

Summary

Saturday, October 29, 2022

Why have virtual address

- Provides each process with the illusion that it has a large amount of contiguous memory available exclusively to itself
    - An **address space**
- Isolate processes from each other and the kernel
- Potential to support virtual memory larger than physical memory
- The total size can be larger than physical memory
    - Greater support for multiprocessing

Goals

- Efficiently translate between virtual and physical addresses
- Protect the process's address space from other processes

Address translation

- Performed in hardware, on the **Memory Management Unit** (**MMU**)
- Use information provided by the kernel
- The program counter (PC) is also a virtual address
    - Each instruction requires at least one translation
    - Therefore the translation is done in hardware, faster than software

Virtual memory

- The OS provides a separate and private memory for each process
- The virtual memory holds the code, data, heap and stack for the running program in that process
- Programs only see virtual addresses
- Each process is isolated in its virtual memory
    - Cannot access another process's virtual memory

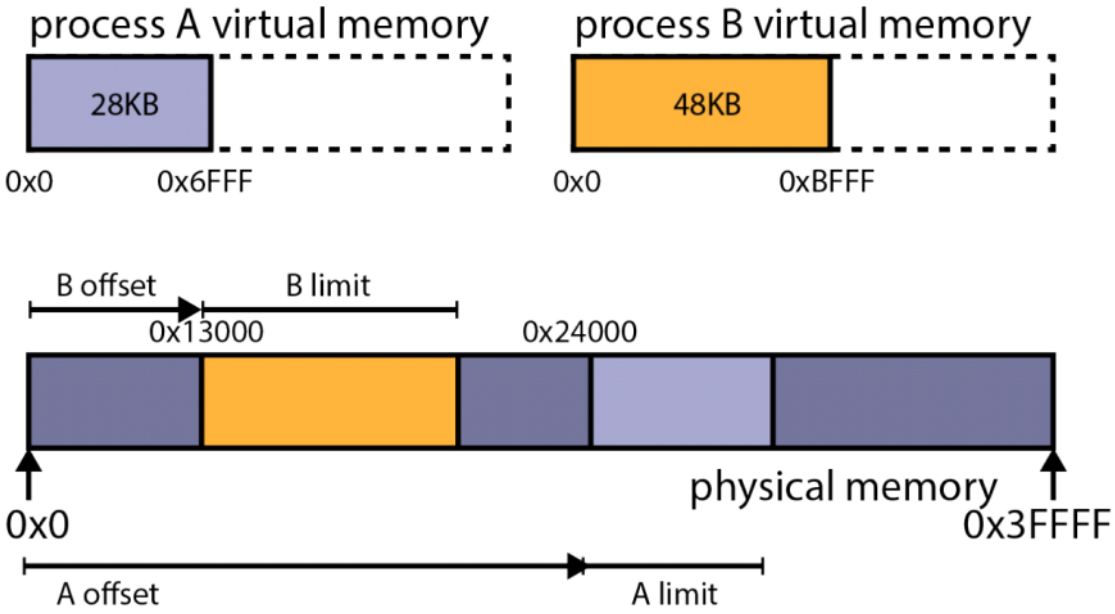
5 methods of address translation

1. Dynamic Relocation
2. Segmentation
3. Paging
4. Two-Level Paging
5. Multi-Level Paging

1. Dynamic Relocation

- The MMU has
    - A **relocation register** (offset) which holds the physical offset for the running process's virtual memory
    - A **limit register** which holds the size of the running process's virtual memory
- The kernel maintains separate offset and limit values for each process
- These values in the MMU registers are changed when there is a context switch between threads in different processes
- Translation from a virtual address v to a physical address p
    - Check if v is less than the limit

    - If this is satisfied, then set p to be v + offset
    - Otherwise raise an exception
- Pros
    - Efficient
- Cons
    - Fragmentation



2. Segmentation
- Map each segment of the address space separately
- The kernel maintains an offset and limit value for each segment
- A virtual address can be thought of as having two parts
    - Segment ID
    - Offset within segment
- Translation is similar to the dynamic relocation
    - Need to split the virtual address into two parts
    - The first bit specifies the segment
    - The next three bits specifies the first hex digit of the offset
- The kernel maintains a separate set of relocation offsets and limits for each process

3. Paging
- Divide physical memory into fixed-size chunks called **frames** (**physical pages**)
- Page size (virtual memory) must equal frame size (physical memory)
- Each page maps to a different frame
- Any page can map to any frame
- Page Table
    - Each row is a **page table entry** (**PTE**)
    - Indexed by a **page number**
    - A **valid bit** is used to indicate to indicate if the PTE is used or not
- Number of PTES = maximum virtual memory size / page size
- MMU includes a **page table register**

    - Stores a pointer to the page table for the current process

3. Page Table Size
- A page table has one PTE for each page in the virtual memory
- Page Table Size = number of pages × size of a PTE
- The size of a PTE is typically provided

3. Page Tables: where?
- Page tables are kernel data structures
- They live in the kernel's memory

Shrinking the Page Table: Multi-Level Paging
- Instead of having a single page table to map an entire virtual memory, we can organize it and split the page table into multiple levels
- A large, contiguous table is replaced with multiple smaller tables
    - Each fitting on to a single page
- If a table contains no valid PTEs, do not create that table
- The lowest-level page table contains the frame numbers
- All higher level tables contain pointers to tables on the next level
- The address translation is the same as single-level paging
    - Physical address = frame number × page size + offset
- Lookup is different

5. Multi-Level Paging: Address Translation
- The MMU's **page table base register** points to the page table directory for the current process
- Each virtual address v has $n + 1$ parts $(p_1, p_2, \dots, p_n, o)$
- To translate a virtual address
    1. Index into the page table directory using p_1 to get a pointer to a 2nd level page table
    2. If the directory entry is not valid, raise an exception
    3. Index into the 2nd level page table using p_2 to get a pointer to the 3rd level page table
    4. If the entry is not valid, raise an exception
    5. ...
    6. Index into the n-th level page table using p_n to find a PTE for the page being accessed
    7. If the PTE is not valid, raise an exception
    8. Otherwise, combine the frame number from the PTE with o to determine the physical address

Summary: Roles of the Kernel and the MMU
- Kernel (software)
    - Manages MMU registers on address space switches (context switch from one thread in one process)to thread in a different process)
    - Creates and manages page tables
    - Manages (allocates/deallocates) physical memory
    - Handles exceptions raised by the MMU
- MMU (hardware)
    - Translates virtual addresses to physical addresses
    - Checks for and raises exceptions when necessary

TLBs

- Each assembly language instruction requires at least one memory operation (to fetch the instruction)
- Address translation through a page table adds a minimum of one extra memory operation for each memory operation
- Translation Lookaside Buffer (TLB)
- Small, fast, dedicated cache of address translations in the MMU
- Each TLB entry stores a single page to frame mapping

Paging - Conclusion

- Benefits
    - Paging does not introduce external fragmentation
    - Multi-level paging reduces the amount of memory required to store page-to-frame mappings
- Costs
    - TLB misses are increasingly expensive with deeper page tables
        - To translate an address causing a TLB miss for a three-level page table requires three memory accesses

Locality

- There are two types of locality
- Temporal locality
    - Programs are more likely to access pages that they have accessed recently than pages that they have not accessed recently
- Spacial locality
    - Programs are likely to access parts of memory that are close to parts of memory the have accessed recently

Clock Replacement Algorithm

- Can be visualized as a victim pointer that cycles through the page frames
- Moves whenever a replacement is necessary

Lecture 16 - Scheduling

Tuesday, November 15, 2022 13:43

Simplest scheduling model

- Response time: time between the job's arrival and when the job starts to run
- Turnaround time: time between the job's arrival and when the job finishes running
- Must decide when each job should run, to achieve some goal

Round Robin

- Preemptive first-come-first-serve

CPU Scheduling

- Typically differs from the simple scheduling model
    - Runtime of threads are unknown
    - Threads can be in blocked state
    - Threads may have different priorities
- The objective of the scheduler is to achieve a balance between
    - Responsiveness - threads get to run regularly
    - Fairness - some threads are more prioritized and need more attention
    - Efficiency

Multi-level Feedback Queues

- Interactive threads are more frequently blocked

Lecture 17 - CPU Scheduling

Thursday, November 17, 2022

Objective of the scheduler

- Responsiveness
  - Interactive threads do not stay in ready/blocked state for too long
- Fairness
  - Threads with higher priority should have a larger share of the CPU time
  - Threads with lower priority should not starve
  - Multiple ready queues
    - Each queue will have a priority
    - Within each queue, we will use round-robin FCFS
    - Who gets put in what queue and why?
    - Higher priority for interactive threads
    - In order to interact with users, they use I/O devices and wait for user input, packets, etc.
    - If a thread is able to run without being blocked except when its scheduling quantum expired, it is not an interactive thread
      - We move it to a queue with lower priority
    - To prevent starvation, move the threads in the lower priority queues to the highest priority queue periodically
    - The higher the priority, the smaller the scheduling quantum
    - One small timer - shorter than the smallest scheduling quantum
    - May preempt lower priority thread when a thread wakes up

Linux Completely Fair Scheduler (CFS)

- Each thread has a **weight**
- Ensure that each thread gets a share of the CPU that's proportional to its weight
- Track the "virtual" runtime of each runnable thread
- Always run the thread with the lowest virtual runtime
- Virtual runtime
  - Suppose w_i is the weight of the ith thread
  - Actual runtime of the thread multiplied by $\frac{\sum_j w_j}{w_i}$

Scheduling on multi-core processors

- Per core ready queue vs. shared ready queue?

Load balancing

- An issue in per-core design
- Not an issue in shared queue design

Summary

Friday, December 2, 2022

Job Scheduling Problem

- For the ith job, there are two parameters that characterize it
  - An arrival time a_i, when the ith job becomes available to run
  - A run time r_i, the total length of time (total amount of processing time) required to complete the ith job
- A job scheduler decides which job should be running on the server at each point in time
- Two times for each job
  - A start time a_i - when the ith job starts running
  - A finish time f_i - when the ith job finishes running

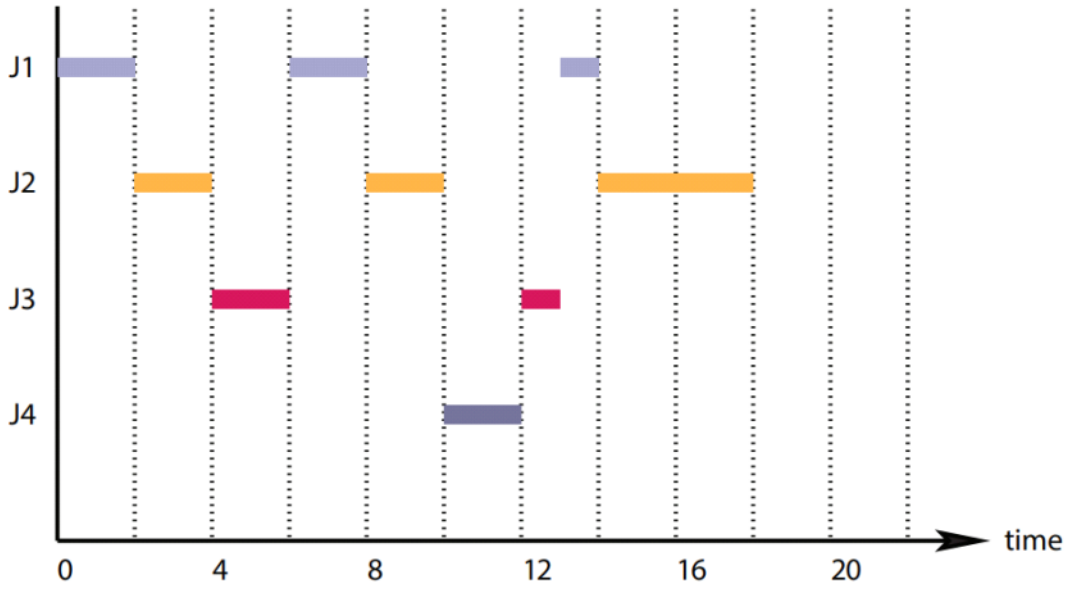
Performance metrics

- Characterized using two times
  - Response time: $s_i - a_i$
    - How long it takes from the arrival of the job until it starts running
  - Turnaround time: $f_i - a_i$
    - How long it takes from the arrival of the job until it finishes running

Four basic scheduler

- First Come First Serve (FCFS)
  - Runs jobs in arrival time order
  - Simple
  - Avoids starvation
  - Pre-emptive variant: Round-Robin (RR)
- Shortest Job First (SJF)
  - Runs jobs in increasing order of r_i
  - Minimizes average turnaround time
  - Long jobs may starve
  - Pre-emptive variant: Shortest Remaining Time First (SRTF)

Round Robin (Preemptive FCFS)



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	5
run time (r_i)	5	8	3	2

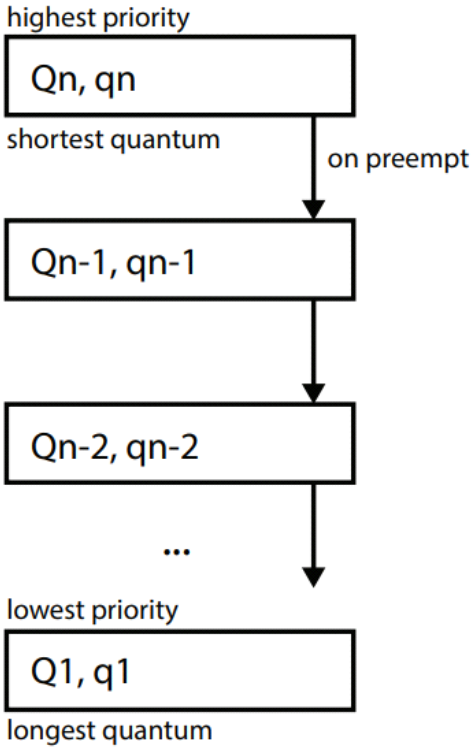
CPU Scheduling

- The jobs to be scheduled are the threads
- The runtime of threads are normally unknown
- Threads are sometimes not runnable
  - In a blocked state
- Threads may have different priorities
- The objective is normally to achieve a balance between
  - Responsiveness: ensure that threads get to run regularly
  - Fairness: sharing of the CPU
  - Efficiency: account for the fact that there is a cost in context switch
- Often expected to consider process and thread priorities
- Two approaches to scheduling with priorities
  1. Schedule the highest priority thread
  2. Weighted fair sharing
     - Give each thread a share of the CPU in proportion to its priority $\frac{p_i}{\sum_j P_j}$

Multi-level Feedback Queues (MLFQ)

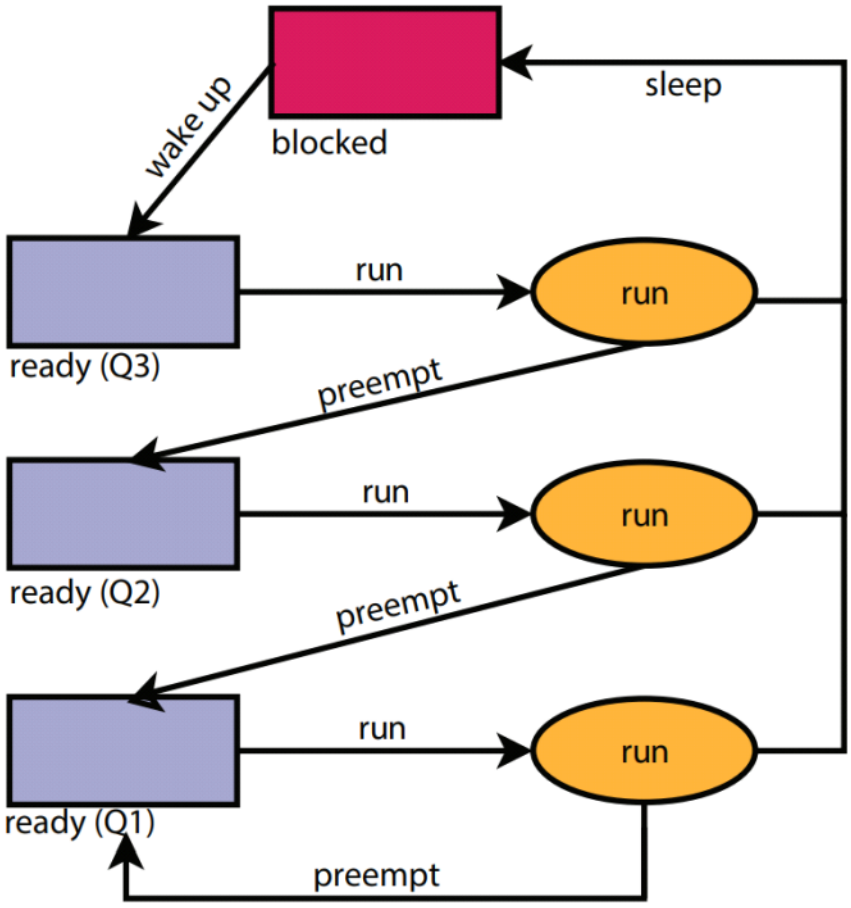
- The most commonly used scheduling algorithm
- Objective
  - Good responsiveness for interactive threads (interact with users via keyboard, mouse and display)
  - Non-interactive threads make as much progress as possible
- Key observation

    - Interactive threads are frequently blocked, waiting for user input, packets, etc.
- Approach: give higher priority to threads that block frequently



MLFQ

- Higher level ⇒ higher priority
- Higher level ⇒ smaller scheduling quantum
- The scheduler selects threads from the highest priority queue to run
    - i.e. threads in Q_{n-1} are only selected if Q_n is empty
- Preempted threads are put at the back of the next lower-priority queue
    - i.e. a thread from Q_n is preempted, it is pushed onto Q_{n-1}
- When a thread wakes after blocking, put it into the highest-priority queue
    - Interactive threads tend to block frequently, so they tend to stay in the higher-priority queues
    - Non-interactive threads will tend to sift down towards the bottom
- Each level has its own run queue and ready queue, but they all share the blocked queue
- To prevent starvation, all threads a periodically placed in the highest-priority queue
- Many variants
    - Preempt low-priority threads when a thread wakes to ensure a fast response to an event

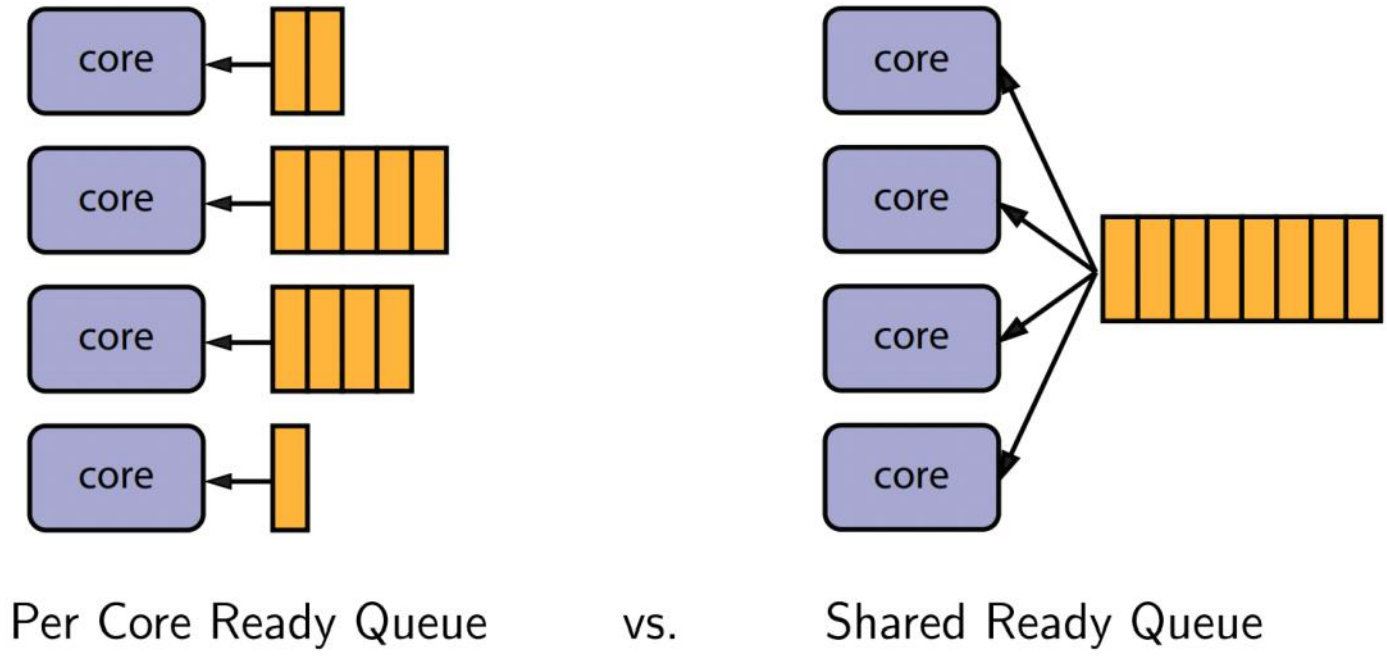


Completely Fair Scheduler (CFS)

- Key idea: each thread is assigned a **weight**
- Goal: ensure that each thread gets a share of the processor proportional to its weight
- The **virtual run time** of a runnable thread is the actual run time adjusted by the thread weights
  $a_i \frac{\sum_j w_j}{w_i}$
- Always run the thread with the lowest virtual run time
- The virtual runtime advances
  - Slowly for threads with high weights
  - Quickly for threads with low weights
- When a thread becomes runnable
  - Its virtual run time is initialized to some value between the min and max virtual run times of the threads that are already runnable

MLFQ vs CFS

- In MLFQ, the quantum depends on the thread priority
- In CFS, the quantum is the same for all threads and priorities



Per Core Ready Queue vs. Shared Ready Queue

Contention and scalability

- Access to shared ready queue is a critical section
  - Mutual exclusion required
- As the number of cores grows, contention for the ready queue becomes more of a problem
- Per core design scales to a larger number of cores

CPU cache affinity

- Typically each core will have some memory cache of its own
- Moving the thread to another core means data must be reloaded into that core's caches
- As thread runs, it acquires an **affinity** for one core
- Per core design benefits from affinity by keeping threads on the same core

Load Balancing

- In per-core design, queues often have different lengths
- This results in load imbalance
  - Some cores maybe idle while others are busy
  - Threads on lightly loaded cores get more CPU time than those on heavily loaded core
- Not an issue in shared queue design
- Per-core design needs some mechanism for thread migration
  - Moving threads from heavily loaded cores to lightly loaded cores

Lecture 17 - Devices

Thursday, November 17, 2022

Device registers

- Different from CPU registers
- Three primary types
    - **Status:** tells you something about the device's current state
    - **Command:** issue a command to the device by writing a particular value t o this register
    - **Data:** transfer larger blocks of data to/from the device
- Some device registers can be combination of these types

Sys/161 timer/clock

- Starts at Jan 1, 1970 midnight
- In general, we need an **epoch**
    - A particular time chosen for calculating the current time

Lecture 19 - I/O Scheduling

Thursday, November 24, 2022

Accessing Devices

- Port-mapped I/O
  - Own set of memory for I/O
  - Memories are broken into ports
  - Device registers are assigned port number
  - Special I/O instructions (such as in and out)
  - Very faster
  - Less generalized; restrictive
  - More expensive
  - Number of ports is small
- Memory-mapped I/O
  - Each device register has a physical memory address
  - kseg1 is for devices
  - Use regular load and store instructions
  - Memory used by an individual device is constant
  - Since number of devices an OS uses is not constant, the memory used by kseg1 is not constant

MIPS/OS161 Physical Address Space

- Each device is assigned to one of 32 64KB device "slots"
- If a device uses more than 64 KB, then it's unsupported

Persistent Storage Devices

- Any device where data persists even when the device is without power
- Physical memory is not persistent
- A hard disk is persistent
- Also referred to as **non-volatile**

Hard Disks

- Commonly used persistent storage device
- A number of spinning, ferromagnetic-coated platters
- Read/write head applies a magnetic field

Logical View of a Disk Drive

- Three component in a read
  - Seek of read/write head
  - Wait for the right sector to spin under the read/write head
  - Read the 512 byte block

Cost Model for Disk I/O

1. **Seek time**: move the read/write heads to the appropriate track
   - Depends on **seek distance**
2. **Rotational latency**: Wait until the desired sectors spin to the read/write heads
   - Value: 0 to cost of single rotation
3. **Transfer time:**

    - Depends on the rotational speed of the disk and the amount of data transferred

Performance Implications of Disk Characteristics

- Large transfers to/from a disk device are more efficient than smaller ones
- Sequential I/O is faster than non-sequential I/O
    - Eliminate the need for (most) seeks
- While sequential I/O is not always possible, we can group requests to try and reduce average request time
- Adding/deleting files introduce disk fragmentation
    - Needs to defragment the disk periodically

Disk Head Scheduling

- Goal: reduce seek times by controlling the order in which requests are serviced
- Can be performed by the OS or the hard disk, or both
- FCFS is fair and simple, but offers no optimization for seek times
    - Starvation is possible

Lecture 20 - Disks, File Systems

Tuesday, November 29, 2022

Disk controller

- Number of sectors
  - Read-only field
- Status
  - Read a status
  - Issue a command by setting a status (?)
- Sector number
  - Sector of disk we want to read into the transfer buffer (data)
- Transfer buffer
  - Read/write by block/sector

Solid State Drives (SSD)

- A higher voltage is 0
- A lower voltage is 1
- Flash memories are initialized with lo voltage

Writing and deleting from flash memory

- Read whole block into memory
- Re-initialize block (all page bits back to 1s)
- Mark this block as free/overwritten
- Write to an empty block
- When there is no usable empty block, flash the ones that have been marked as deleted/overwritten

Persistent RAM

- Values are persistent in the absence of power
- Improve the performance of secondary storage
  - Traditional CPU caches are small - not effective for caching many disk blocks
  - RAM can cache i-nodes and data blocks; but should be used for address spaces
  - Use persistant RAM instead

File

- Persistent, name data objects
- Data consists of a sequence of numbered bytes
- May change size over time
- Associated meta-data
  - Type
  - Timestamp
  - Access controls

File system

- The data structures and algorithms used to store, retrieve and access files
- Logical file system
  - High-level API
  - What a user sees

- Virtual file system
    - Abstraction of lower level file systems
    - Multiple different underlying files systems to the user as one
- Physical file system
    - How files are actually stored on physical media

File interface

- open
    - Returns a **file identifier** (or handle or descriptor)
        - Information about how you opened it (read, write, ...)
        - Where are we in the file (file index)
    - Used in subsequent operations to identify the file
    - Other operations (e.g., read, write) require file descriptor as a parameter)
- close
    - Kernel tracks while file descriptors are currently valid for each process
    - Close invalidates a valid file descriptor
- read
    - Copies data from a file into a virtual address space
- write
    - Copies data from a virtual address space into a file
- seek
    - Enables non-sequential reading/writing
- get/set file meta-data
    - fstat
    - chmod
    - ls
    - -la

File position and seeks

- Each file descriptor (open file) has an associated **file position**
    - Starts at byte 0 when the file is opened
- Read and write
    - Start from the current file position
    - Update the current file position as bytes are read/written
- seek are used for achieve non-sequential file I/O
    - lseek changes the file descriptor

Directories and file names

- A directory maps file names (strings representing the full path) to i-numbers
- An i-number is a unique identifier for a file or directory
- Given an i-number, the file system can find the data and meta-data for the file
- Since directories can be nested, a file system's directories can be viewed as a tree
    - Has a single **root** directory
    - Files are leaves
- Files may be identified by **pathname** through the directory tree from the root directory to the file

Summary

Saturday, December 3, 2022 16:36

Devices

- How a computer receives input from the outside world and produces output for the outside world
- Examples
    - Keyboard
    - Printer
    - Touch screen
    - Timer/clock
    - Disk drive
    - Serial console
    - Text screen
    - Network interface

Terminology

- Bus: a communication pathway between various devices in a computer
- Internal bus
    - Aka memory bus
    - Is for communication between the CPU and RAM
    - Fast and close to the CPU(s)
- Peripheral bus
    - Aka expansion bus
    - Allows devices in the computer to communicate
- Bridge
    - Connects two different buses

Device register

- Communications with devices carried out through **device registers**
- Three primary types
- Status
    - The device's current state
    - Typically, a status register is read
- Command
    - Issue a command to the device by writing a particular value to this register
- Data
    - Used to transfer larger blocks of data to/from the device
- Some device registers are combinations of primary types
    - A status and command register
        - Read to discover the device's state
        - Written to issue a command
    - A data buffer
        - Sometimes combined
        - Sometimes separated into data in and data out buffers

Offset	Size	Type	Description
0	4	status	current time (seconds)
4	4	status	current time (nanoseconds)
8	4	command	restart-on-expiry
12	4	status and command	interrupt (reading clears)
16	4	status and command	countdown time (microseconds)
20	4	command	speaker (causes beeps)

Sys161 timer/clock

- Used in preemptive scheduling

Offset	Size	Type	Description
0	4	command and data	character buffer
4	4	status	writeIRQ
8	4	status	readIRQ

Serial console

- Used to write outgoing characters and read incoming characters
- If a write is in progress, the device exhibits **undefined behaviour** if another write is attempted
- IRQ stands for interrupt request

Device drivers

- A **device driver** is a part of the kernel that interacts with a device
- Communication happens by reading from or writing to the command, status and data registers
- Two methods
- Polling
  - The kernel driver repeatedly checks the device status
- Interrupts
  - The kernel does not wait for the device to complete the command
  - Request completion is taken care of by the interrupt handler
  - The device updates a status register and then generates an interrupt
- Binary semaphores are usually used instead of lock
  - The thread that initiates communication with the device may have to block
  - The thread that acknowledges the interrupt is not necessarily the thread that initiates the communication with the device
  - Lock makes it slower since we have to switch to the thread that initiates communication
  - Binary semaphore can be released by any thread

Accessing devices

- **Port-mapped I/O**
  - Uses special assembly language I/O instructions

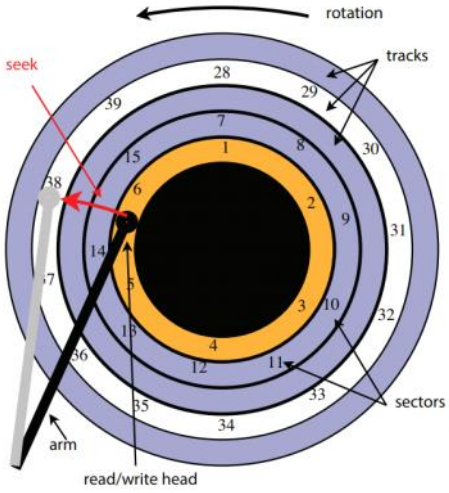
    - Device registers are assigned pot numbers - regions of memory in a separate, smaller address space
    - Special I/O instructions used to transfer data between a specified port and a CPU register
- Memory-mapped I/O
    - Each device register has a physical memory address
    - Device drivers can read or write to them using normal load and store instructions
- A system may use both port-mapped and memory-mapped I/O

Large data transfer to/from devices

- Large data blocks can be transferred using other strategies
- Program-controlled I/O (PIO)
    - The device driver moves the data between memory and a buffer on the device
    - Simple to implement
    - The CPU is used to transfer the data
- Direct memory access (DMA)
    - The CPU is used to initiate communication with the device
    - The device itself is responsible for moving data to/from memory
    - The CPU is not used to transfer the data, and is free to do something else
    - Used for block data transfers between devices (e.g. a disk controller and primary memory)

Persistant storage devices

- Persistant (aka non-volatile) storage is a device where data persists even when there is no power
- Primary memory (RAM) is not persistant
- Secondary storage (disk) is persistant



Logical view of a disk drive

- Logically a disk is an array of numbered blocks (sectors)
- Each block is the same size
- Blocks are the unit of transfer between the disk and memory

Physical view of a hard drive

- A small number of **platters** (disks) made of glass or porcelain
- Each platter has two **surfaces**
- A surface is broken up into a series of concentric circles called **tracks**
- **Cylinders** are tracks of the same radius
- Each track is broken up into a series of arcs called **sectors** or **blocks**
- Each surface has a **disk head** (also called **read/write head**) to read and write data from this surface
- All disk heads are put in position by a single **disk arm** (also called an **actuator arm**)

Cost model for disk I/O

- **Seek time** - the time it takes to move the read/write head to the appropriate track
  - Depends on seek distance
  - Value: 0 to cost of max seek distance
- **Rotational latency** - the time it takes for the desired sectors to spin to the read/write head
  - Depends on the rotational speed
  - Value: 0 to cost of single rotation
- **Transfer time** - the time it takes until the desired sectors spin past the read/write heads
  - Depends on the rotational speed of the disk and the amount of data accessed
- Request service time = seek time + rotational latency + transfer time

Performance implications of disk characteristics

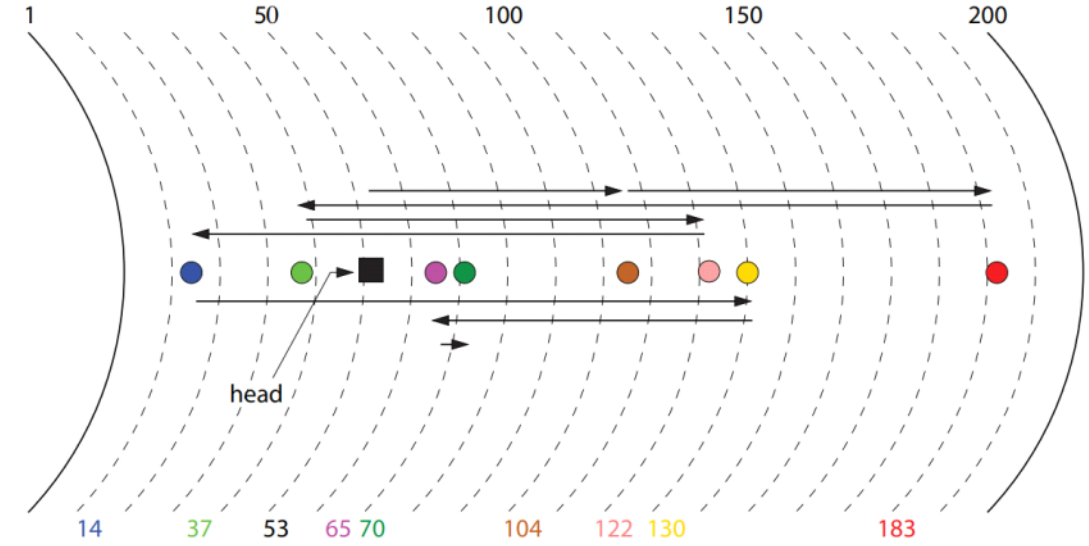
- Larger transfers are more efficient than smaller ones
  - The cost per byte is smaller for larger transfers
  - The time it takes to get there is amortized over many blocks of data
- Sequential I/O is faster than non-sequential I/O
  - Sequential I/O operations eliminate the need for (most) seeks

Disk head scheduling examples

- The disk request queue (in order of arrival) is:
  104 183 37 122 14 130 65 70

Disk head scheduling 1: First Come First Served (FCFS)

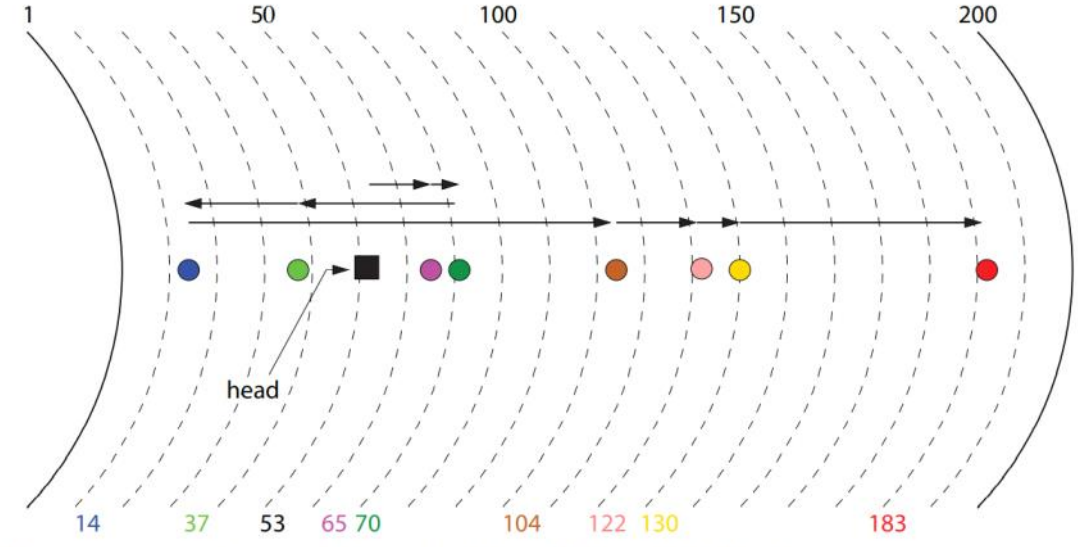
- Service the requests in the order they arrive
- Fair and simple
- Offers no optimization for seek time



The service order is: 104 183 37 122 14 130 65 70

Disk head scheduling 2: shortest seek time first (SSTF)

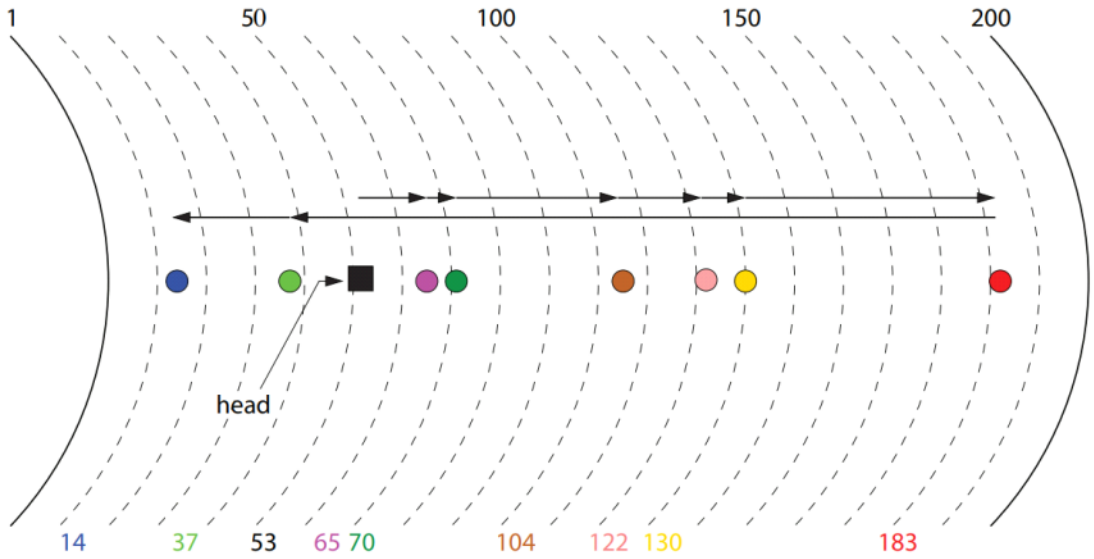
- Service the closest request first
- Seek times are reduced
- May cause starvation
    - For example, the request represented by the red dot



The service order is: 65 70 37 14 104 122 130 183

Disk head scheduling 3: elevator algorithm (SCAN)

- The disk head moves in one direction until there are no more requests in front of it, then reverses
- Reduce seek times (relative to FCFS)
- Avoids starvation



The service order is: 65 70 104 122 130 183 37 14

Lecture 21 - File Systems

Thursday, December 1, 2022

Links

- A **hard link** is an association between a name and an i-number
  - Each entry in a directory is a hard link
- Once a file is created, additional hard links can be made to it
- Linking to an existing file creates a new pathname for that file
  - Each file has a unique i-number, but may have multiple pathnames
- Not possible to link to a directory (to avoid cycles)
- Different from a shortcut, which link to the string to i-number mapping
- When a file is deleted, the number of hard links becomes 0
  - But it is still possible to restore the file
  - If the i-number has not been reused

Unlinking

- Hard links can be removed
  - unlink
  - Removes the link from the directory
- When the last hard link to a file is removed, the file is also removed

Multiple file systems

- DOS/Windows: use two-part file names
  - File system name
  - Pathname within file system
  - Example: C:\user\cs350\schedule.txt
- Unix: create single hierarchical namespace that combines the namespaces of two file systems
  - Unix mount system call does this

File system implementation

- What needs to be stored persistently?
  - File data
  - File meta-data
  - Directories and links
  - File system meta-data
- Non-persistent information
  - Per process open file descriptor table
    - File handle
    - File position
  - System wide
    - Open file table
    - Cached copies of persistent data

Lecture 22 - File Systems

Tuesday, December 6, 2022

i-nodes

- Direct data block pointers
  - Point at a single data block
- Single indirect pointer
  - E.g. single indirect pointer points at a data block, where we store direct pointers
  - $2^{12}/2^2 = 2^{10}$ pointers per block
  - $2^{10} \cdot 2^{12} = 2^{22}$ bytes
- Double indirect data block
  - $2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{32}$ bytes
- Triple indirect data block
  - $2^{10} \cdot 2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{42}$ byte

Chaining

- Directory table contains the name of the file, and each file's starting block
- Each block includes a pointer to the next block
- Random access is very bad
- Sequential access is better

External chaining

- Introduces a special file access table that specifies all of the file chains
- Read less blocks
  - Since the table just contains the links, it might fit on a single or two blocks

File system design

- What if we never delete or move a directory?
  - We can store the directories in the root's data

Summary

Sunday, December 4, 2022

File

- Files are persistent and named data objects
- Data consists of a sequence of numbered bytes
- No other assumptions are made about the data: it could be text, images, video, machine code, etc.
- Files may change size and content over time
- Files have associated meta data
    - Owner
    - File type
    - Date created
    - Date modified
    - Access controls

File Systems

- File systems are data structures and algorithms used to store, retrieve and access files
- Can be separated into three different layers
- Logical file system
    - High-level API
    - What a program sees (fopen, fscanf, fprintf, fclose)
    - Manages file system information
    - Abstract the details of how files are stored physically
- Virtual file system
    - Abstraction of lower level file systems
    - Presents multiple different underlying file systems to the user as one
- Physical file system

File Interface

- open
    - Find a file
    - Check permission (read, write, etc)
    - Returns a file descriptor (identifier or handle)
    - Other file operations (read, write, seek, close) for this process require this file descriptor as a parameter in order to identify the file
    - We do not need to use the pathname (string) in the future
    - The file descriptor is placed in the file descriptor table of the current process
- close
    - Invalidates a valid file descriptor
    - Tracks which file descriptors are currently valid for each process
- read, write, seek
    - read copies data from a file into a virtual address space
    - write copies data from a virtual address space into a file
    - read and write are sequential
        - They use the current file position
    - seek enables non-sequential reading/writing

        - Changes the file descriptor in the table
        - Allows a different starting position

File position and seeks

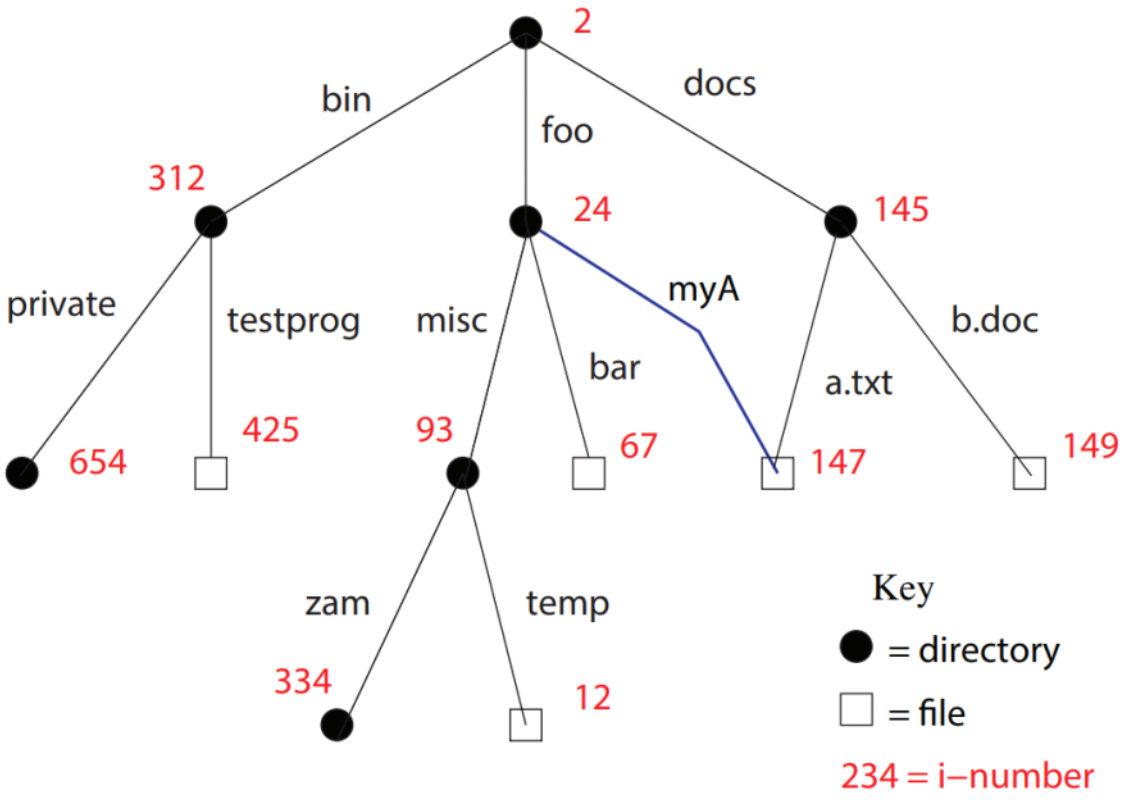
- Each open file (valid descriptor) has an associated file position
    - This position starts at byte 0 when the file is opened
- Read and write operations
    - Start from the current file position
    - Update the current file position as bytes are read/written
- Makes sequential file I/O easy
- seek (lseek) is used for non-sequential file I/O
    - Changes the file position associated with a descriptor
    - The next read or write from that descriptor will use the new position

Directories and i-numbers

- A directory maps file names (strings) to i-numbers
    - An i-number (index number) is a unique (within a file system) identifier for a file or directory
    - Given an i-number, the file system can find the data and meta-data for the i-number's corresponding file.
- Directories provide a way for applications to group related files
- Since directories can be nested, a file system's directories can be viewed as a tree
    - Has a single root directory
    - Files are always leaves
    - Directories can be interior nodes (if they are non-empty) or leaves (if they are empty)

File names and pathnames

- Files may be identified by pathnames, which describe a path through the directory tree from t he root directory to the file
- Directories also have pathnames
- Applications and humans refer to files using pathnames, not i-numbers
- Only the kernel is permitted to directly edit directories. Why?
    - In general, the kernel should not trust the user with direct access to the data structures that the kernel relies on



Links

- A hard link is an association between a name (string) and an i-number
- Each entry in a directory is a hard link
- When a file is created, a hard link to that file is also created
- E.g. open("/docs/a.txt", "O_CREAT|O_TRUNC")
  - This command opens the file and creates a hard link to that file in the directory
  - O_CREAT means if this file does not exist then create it
  - O_TRUNC means overwrite the existing contents (if the file is writable)
- Once a file is created, additional hard linkes can be made to it
- Example:
  - In C: link("/docs/a.txt", "/foo/myA")
  - In Linux: link /docs/a.txt /foo/myA
  - Creates a new hard link myA in directory /foo
  - The link refers to the i-number of file /docs/a.txt (147) which must exist
- Linking to an existing file creates a new pathname for that file
- Each file has
  - A unique i-number
  - May have multiple pathnames
- In order to avoid cycles, it is not possible to link to a directory

Unlinking

- Hard links can be removed:
- Example:
  - In C: unlink("docs/b.doc")
  - In Linux: unlink /docs/b.doc
  - This removes the link b.doc from the directory /docs
- Hard links have referential integrity

    - If the link exists, then the file it refers to also exists
    - When a hard link is created, it refers to an existing file
    - The kernel keeps the number of hard links to an existing file
    - A file is deleted when its last hard link is removed (count becomes zero)

Symbolic links

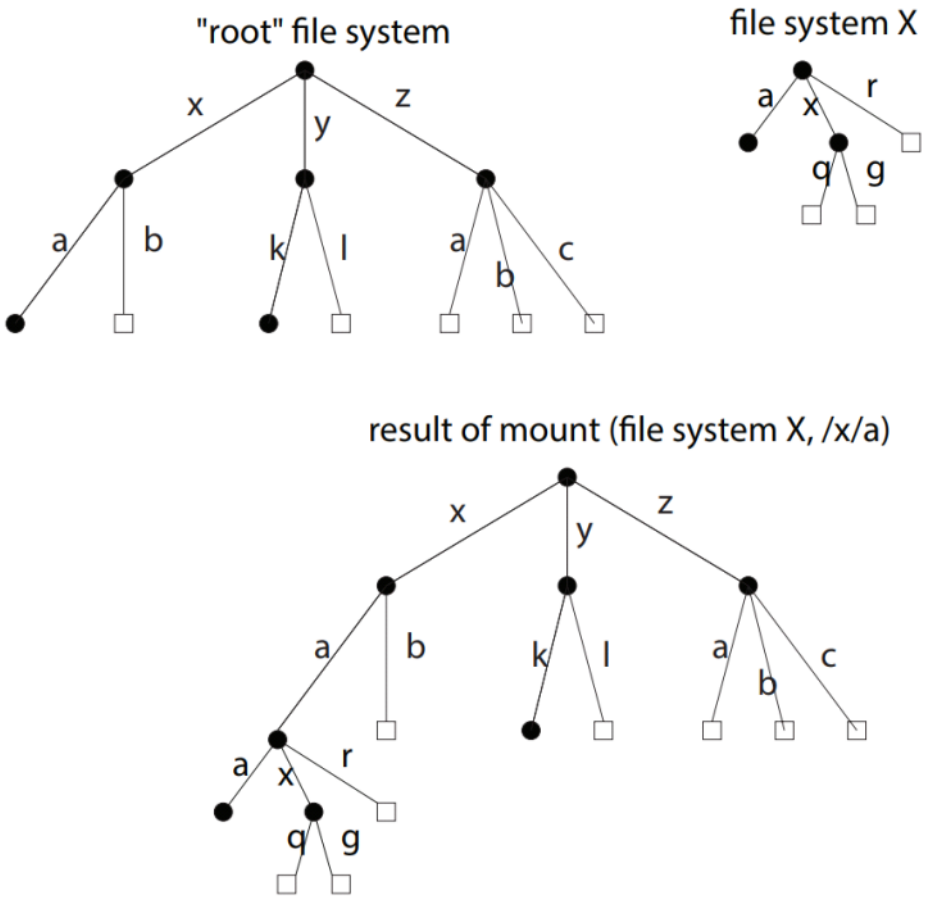
- A symbolic link, or soft link, is an association between a name(string) and a path name
- Command: symlink(/z/a, /y/k/m) - creates a symbolic link m in directory /y/k
- If an application attempts to open /y/k/m, the file system will
    - Recognize it as a symbolic link
    - Attempt to open /z/a instead
- Referential integrity is not preserved for symbolic links

Multiple File Systems

- Some kind of global file namespace is required
    - Uniform across all the file system
    - Independent of physical location
- Windows
    - Two-part file names
    - File system name and path name within file system
- Linux
    - Create a single hierarchical namespace that combines the namespaces of multiple file systems, e.g. the mount syscall

Mounting

- Does not make two file systems into one file system
- Merely creates a single, hierarchical namespace that combines the namespaces of two file systems
- Temporary - exists only until the file system is unmounted



File system implementation

- What needs to be stored persistently?
  - File data
  - File meta-data
  - Directories and links
  - File system meta-data
- What can be non-persistent?
  - File descriptor
  - File position for each open file
  - Open file table (info about files that are currently open)
  - Cached copies of persistent data

i-nodes

- An i-node is a fixed size index structure that holds the file meta-data and pointers to the data blocks
- i-node fields may include
  - File type
  - File permissions
  - File length
  - Time of last file access, last file update, last i-node update
  - Number of hard links to this file
  - Pointers to data blocks
- For small files, pointers in the i-node are sufficient to point to all data blocks (direct data block pointers)

- For larger files, we need another approach (single, double and triple indirect data block pointers)

Directories

- Implemented as a special type of file that contains directory entries
- Pairing up an i-number and a file name (the last component of a path name)
- Directory files can be read by application programs
- Directory files are only updated by the kernel
    - Create file
    - Create link
    - ...

i-num	name
5	.
2	..
12	foo
13	cs341
15	cs350

In-memory (non-persistent) structures

- In order to speed up file access, the kernel keeps some information in RAM which is updated as processes access files

Per process - descriptor table

- Tracks
    - The file descriptors this process has opened
    - The file each open descriptor refers to
    - The current file position for each descriptor

System wide

- Open file table
    - Files that are currently open by any process
- i-node cache
    - In-memory copies of recently-used i-nodes
- Block cache
    - In-memory copies of data blocks and indirect blocks

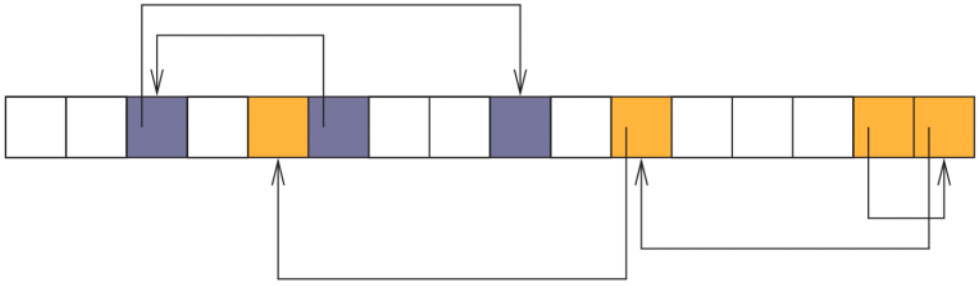
Chaining

- VSFS uses a per-file index (direct and indirect pointers) to access blocks
- Two alternative approaches
- Chaining

    - Each block includes a pointer to the next block
- External chaining
    - The chain is kept as an external structure

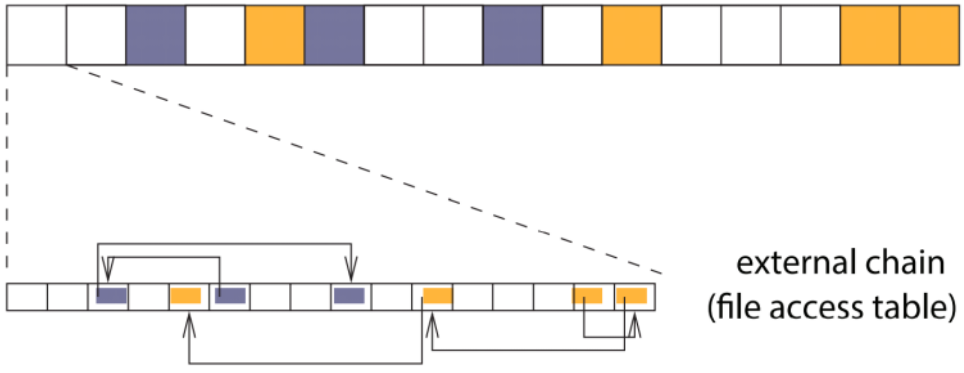
Chaining

- Implementation
    - The directory table contains the name of the file paired with its starting block (and possibly its end block to facilitate appending)
- Performance
    - Acceptable for sequential access
    - Very slow for random access



External chaining

- Idea
    - A file access table (FAT)
    - Specifies all of the file chains in one big table
- Key benefit
    - Reduces the number of blocks read by keeping this table in its own block
    - Rather than having one pointer per block
- This table can be cached to improve performance



File System Design

- Most files are small
- File systems contain many files
- Typically directories are small
- Even as disks grow in capacity, the average file system usage is 50%

Fault Tolerance

- Special-purpose consistency checkers
    - Runs after a crash but before normal operations resume
    - Finds and attempts to repair inconsistent file system data structures
        - Files with no directory entry
        - Free space that is not marked as free
- Journaling
    - Write-ahead logging
    - Record the file system meta-data changes in a journal
    - The sequences of changes can be written to disk in a single operations
    - After the changes have been journal, update thew disk data structures.

Midterm Solutions

Friday, December 2, 2022

1b

- If we do not use volatile our value is locally cached, instead of loading the actual global value
- Any time you use this variable, read it from memory.
- Any time you write to this variable, write it back to memory

1c

- A thread does not have its own address space
- A process is an environment for programs to execute. A thread is a sequence of instructions

1d

- Spinlock makes you busy-wait (still executing)

2a

- Virtual memory
- Ensure process that cannot access each other's memory

2b

- Not all tasks are can be run in parallel
- Context switch causes low efficiency
- Hardware support

2c

- If we have multiple threads, then we can gain efficiency if one of them blocks

2d

- If we omit page entries, we can have memory errors
- Because we are using indices to represent the page tables, we need to have an index for every page

3a

- Map individual pages to different physical fragmentation
- Internal fragmentation - much smaller
- By making all allocations unit sizes

4b

- X -> A -> B
- Node create locks A and B
- Node destroy locks X
- Node create tries to lock X, blocks
- Node destroy tries to lock A
- Deadlock
- Solution: force strict resource ordering
- All threads first lock parent, then lock current, and then lock child

Threads

Sunday, December 11, 2022

1. Which of the following are shared between different threads? Which are not?
    a. Global variables
    b. Heap
    c. Stack

    a and b are shared; c is not
2. Does thread_yield guarantee that the CPU will be given to another thread?
    No, the same thread can be scheduled
3. 1 processor, 10 cores, each core can run 2 threads. How many threads can be run simultaneously?
    20
4. List 4 ways that can cause a context switch.
    Thread yield, blocks, exit, preempted
5. Draw the thread state diagram.
    Running to ready: preempted, yield
    Running to blocked: waiting for a resource
    Blocked to ready: wake up from the wait channel
    Ready to running: dispatch
6. What is a difference between a switchframe and a trap frame?
    Switchframe only stores the s0-s7 registers. A trapframe stores all registers.

Synchronization

Sunday, December 11, 2022

1. Identify race condition in Slide 9.
2. Identify race condition in Slide 10.
3. What is the difference between exclusive LDREX/STREX and normal lw/sw?
   LDREX and STREX must be used together.
   STREX checks whether the value at an address has been touched since the last LDREX.
4. What is the difference between spinlock and lock in terms of their owners?
   Spinlock is owned by the CPU.
   Lock is owned by a thread.
5. Describe a semaphore's P and V operations.
   P: if the counter is less than or equal to 0, wait for it to become greater than 0, and decrement the counter. If it is already greater than 0, decrement the counter.
   V: increment the counter
6. What are the 3 types of semaphores?
   Counting, binary and barrier
7. List 3 differences between a lock and a semaphore.
   A lock has an owner. A semaphore does not.
   A lock only has a boolean value "hold". A semaphore has a counter with integer value.
   V does not have to follow P.
8. Identify race condition in Slide 33.
9. Why does the wait channel need to be locked before releasing the spinlock?
10. Describe the wait, signal and broadcast operations of a condition variable.
    cv_wait releases the lock associated with the cv, blocks, wakes up when the condition is satisfied (by cv_signal or cv_broadcast).
    cv_signal wakes up one thread in the wait channel.
    cv_broadcast wakes up all threads in the wait channel.
11. Write the pseudocode to implement cv_wait.
    Check owner
    Lock wait channel
    Release lock
    Sleep on wait channel
    Acquire the lock
12. T/F: When a blocked thread is unblocked by cv_signal or cv_broadcast, it needs to reacquire the lock explicitly.
    False
13. Why do we need to call cv_wait inside of a while loop?
    We need to check the condition again since another thread could have changed the variable before we check it.
14. What does the volatile keyword mean? Why is it needed for shared variables?
    Load and store into the memory instead of in registers; to prevent race conditions
15. How does no-hold-and-wait prevent deadlocks?
    No-hold-and-wait acquires all resources at the same time, or acquire none if any one cannot be acquired, so that a thread cannot block while it has resources.
16. How does resource ordering prevent deadlocks?
    Resource ordering assigns resources with numbers, and any threads can only acquire resources in order.

Process

Saturday, December 10, 2022

1. What is the difference between a process and a program?
   A process is an execution environment for programs.
   A program is an executable file.
2. Do processes share the same address space?
   No, they have separate address space and are isolated from each other.
3. If one process has an error, how will the other processes affected?
   They should not be affected.
4. What does fork do?
   Create a clone of the current process, such that the address spaces are identical at the time of clone.
5. True or false: two processes after fork have the identical address space for the rest of their lives.
   No, they have the same address space (not sharing)
6. What is the fork's return value to the parent? To the child?
   fork returns child's pid to the parent and returns 0 to the child
7. Why does the kernel records the exit status code when a process exits?
   Because its parent can call waitpid on it.
8. What changes in execv? What stays the same in execv?
   The program that a process executes is changed in execv, but pid and parent-child relationship stays the same.
9. Write the pseudocode for "/testbin/argtest first second" using execv
   char *[4] buffer
   buffer[1] = "/testbin/argtest"
   buffer[2] = "first"
   buffer[3] = "second"
   buffer[4] = "\0"
10. What is the difference between interrupts and exceptions?
    Interrupts are generated by devices.
    Exceptions are raised during program executation.
11. There is only one syscall exception. How does the kernel know which system call the application is requesting?
    The kernel puts a system call code in the register v0.
12. System calls take parameters and return values, like function calls. How does this work, since system calls are really just exceptions?
    Kernel-specified locations before the syscall, and looks for return values in kernle-specified locations after the exception handler returns.
    On MIPS, parameters go in registers a0, a1, a2, a3.
13. Briefly describe exception handling in OS/161.
    Creates a trapframe.
    Determines the type of exception.
    Finds out it is a system call exception.
    Determines the type of syscall.
    Does the work to handle exception.
    Restore state from the trapframe.
    Return from execution.

14. Where are trap frames and switch frames placed? Why?
    They are placed in the kernel stack to prevent users from accessing it
15. Why do we have system calls?
16. Draw the stack diagram for a fork system call followed by a timer interrupt when the syscall is executed by the kernel.

Virtual Memory

Friday, December 9, 2022

1. Why do we have virtual addresses?
   - To isolate the process from each other
   - To create an illusion of having a memory that is larger than the physical memory
2. What is the amount of addressable virtual memory in bytes if the virtual address is 32 bits?
   2^{32}
3. Since virtual memory is not real, how do processes access a virtual address?
   They need to translate the virtual addresses into physical addresses using via the TLB/page table
4. Why is address translation done in hardware, which is faster than software?
   Because each instruction execution involves a translation from a virtual address to a physical address.
5. In dynamic relocation, what two MMU registers are needed? What do these values represent? Are they the same for every process?
   We need a relocation and a limit register. The relocation register stores the offset (beginning of the virtual address for a process). The limit register stores the size of memory. They should not be the same for every process, otherwise the processes are not isolated from each other (i.e., they can access each other's data).
6. What is a disadvantage of dynamic relocation?
   External fragmentation - since the address spaces of different processes can have different size, so even if there is enough memory, it might not be contiguous.
   Internal fragmentation - the code, data and stack sections can have a lot of spaces between, causing wasted space.
7. What happens when we try to write to an invalid virtual address?
   The MMU raises an exception.
8.

Process A		Process B	
Limit Register: 0x0000 7000		Limit Register: 0x0000 C000	
Relocation Register: 0x0002 4000		Relocation Register: 0x0001 3000	
v = 0x102C	p = ?	v = 0x102C	p = ?
v = 0x8000	p = ?	v = 0x8000	p = ?
v = 0x0000	p = ?	v = 0x0000	p = ?

   Process A:
   0x0002 502C
   Exception
   0x0002 4000

   Process B:
   0x0001 402C
   0x0001 B000
   0x0001 3000

9. For segmentation, what two values do we need for each segment? What do these values represent?

We need a relocation and limit register for each segment. Relocation is the offset (beginning address of a segment), limit is the size of a segement.

10. With 3 bits of segment number, how many segments can we have? How many bytes per segment if the number of virtual address bits is 32?
    8; 2^{29}
11. How many bits of segment number do we need for
    a. 6 segments?
       3
    b. 12 segments?
       4
    c. 16 segments?
       4
12.

Process A

Segment	Limit Register	Relocation Register
0	0x2000	0x38000
1	0x5000	0x10000

Process B

Segment	Limit Register	Relocation Register
0	0x3000	0x15000
1	0xB000	0x22000

Translate the following for process A and B:

v	Segment	Offset	p
0x1240			
0xA0A0			
0x66AC			
0xE880			

Process A

V	Segment	Offset	P
0x1240 = 0001…	0	0x1240	0x39240
0xA0A0 = 1010…	1	0x20A0	0x120A0
0x66AC = 0110…	0	0x66AC	Exception
0xE880 = 1110…	1	0x6880	Exception

13. Virtual addr = 32 bits, Physical addr = 32 bits, Offset = 28 bits

```
Segment Table base reg   0x0010 0000
Segment Table len reg    0x0000 0004
```

Seg	Size	Prot	Start
0	0x6000	X-	0x7 0000
1	0x1000	-W	0x6 0000
2	0xC000	-W	0x5 0000
3	0xB000	-W	0x8 0000

```
Virtual address          0x0000 2004
Physical address  =      __________ ?
Virtual address          0x2000 31A4
Physical address  =      __________ ?
```

0x0007 2004
0x0005 31A4

14. Where do page tables live?
    Page tables are kernel data structures. They live in the kernel's memory.
15. Which are done by the kernel (software)? Which are done by the MMU (hardware)?
    a. Manages MMU registers during context switches
    b. Creates and manages page tables
    c. Manages physical memory
    d. Handles exceptions raised by the MMU
    e. Translates virtual addresses to physical addresses
    f. Checks for and raises exceptions

    Kernel: a, b, c, d
    MMU: e, f
16. What is stored in a TLB entry?
    Virtual address to physical address mappings
17. What does the kernel need to do if the MMU cannot distinguish TLB entries from different address spaces?
    It needs to clear the TLB before context switches
18. What are the advantages and disadvantages of multi-level paging?
19. Which kernel function handles TLB exceptions?
    vm_fault

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

20.

	Process 1	Process 2
Virtual addr	0x0040 0004	0x0040 0004
Physical addr =	---------- ?	---------- ?
Virtual addr	0x1000 91A4	0x1000 91A4
Physical addr =	---------- ?	---------- ?
Virtual addr	0x7FFF 41A4	0x7FFF 41A4
Physical addr =	---------- ?	---------- ?
Virtual addr	0x7FFF 32B0	0x2000 41BC
Physical addr =	---------- ?	---------- ?

21. What are contained in an ELF file?
    Address space segment descriptions.
    - The virtual address of the start of the segment
    - Length of the segment
    - Location of the segment in ELF
    - Length of the segment in ELF
    - Entry point
    - Other information
22. The ELF file does not describe the stack. Why?
    We don't know the content of the stack before hand
23. What does the kernel use to manage frame use?
    Coremap
24. How do we translate kseg0 addresses to physical ones? Is the TLB used?
    Subtract 0x8000 0000. The TLB is not used.
25. How do we translate kseg1 addresses to physical ones? Is the TLB used?
    Subtract 0xA000 0000. The TLB is not used.
26. What is the set of virtual pages present in physical memory called?
    Resident set of a process
27. How do we track which pages are in physical memory?
    We use a present bit in the page table. If present = 1 then the page is present in physical memory
28. In a software-managed TLB, when does the kernel detect the problem of a process trying to access a non-resident page? What is the event called?

When the MMU raises a TLB miss exception and the kernel finds that the present bit is 0. This event is called a page fault.

29. What is the optimal page replacement policy? Why is it not possible?
    The optimal policy is to replace the one that will be used farthest in the future. It is not possible because we cannot predict the future.

Scheduling

Friday, December 9, 2022

1. Describe and give an advantage and disadvantage of:
   a. First Come First Served
      Fair and simple; no optimization
   b. Round Robin
   c. Shortest Job First
      Reduce average turnaround time; starvation
   d. Shortest Remaining Time First
      Starvation still possible
2. In a MLFQ, which priority queue has the longest quantum and which priority queue has the shortest quantum? Why?
   The quantum is the shortest in the highest priority queue and the longest in the lowest priority queue
3. Where are preempted threads put in a MLFQ?
   Lower priority queue
4. Where are blocked threads put after they wake up in a MLFQ?
   Highest priority queue
5. How does a MLFQ prevent starvation of non-interactive threads?
   They are periodically put in the highest priority queue
6. In Linux Completely Fair Scheduler, how to calculate a thread's virtual runtime?
   Actual runtime times sum of weight divided by a thread's weight
7. What is the difference between MLFQ and CFS in terms of the quantum?
   MLFQ: the quantum is the shortest in the highest priority queue and the longest in the lowest priority queue
   CFS: the quantum is the same
8. Suppose the total weight of all threads in the system is 50 and the quantum is 5.

Time	Thread	Weight	Actual Runtime	Virtual Runtime
t	1	25	5	5·2=10
	2	20	5	5·2.5=12
	3	5	5	5·10=50
		50		
$t+5$	1	25	10	10·2=20
	2	20	5	5·2.5=12
	3	5	5	5·10=50

   Which thread is selected at t? 1 Which thread at $t+5$? 2
9. Per core ready queue vs. Shared ready queue
   a. Which one offers better performance?
      Per core (no need to use synchronization primitives)
   b. Which one scales better?
      Per core
   c. Which one has better cache affinity?

Per core

d. Which one has load balancing issues?
   Per core (need occasional rebalance - thread migration)

Devices and I/O

Friday, December 9, 2022

1. What is an internal bus? What is a peripheral bus? Which one is faster?
   Internal bus is for communication between RAM and CPU.
   Peripheral bus is for devices to communicate between each other.
   Internal bus is faster.
2. What are the three primary types of device registers?
   Status - showing what the device is doing
   Command - how the OS issues command for a device
   Data - large data transfer buffer
3. In the sys/161 timer/clock, why is the countdown time smaller than the scheduling quantum?
4. What does restart-on-expiry do?
   When set to 1, restart the countdown timer automatically when it expires
5. In the serial console, what will happen if a write is in progress but another write is attempted? How do we prevent this from happening?
   The behaviour is undefined. We need some kind of mutual exclusion.
6. What does a device driver do?
   It is a part of a kernel that communicates with a device.
7. Give the pseudocode for writing character to serial output device, using polling
   P(sem)
   Write to character buffer
   while (1) {
   If (writeIRQ) {
   break;
   }
   }
   Write writeIRQ register to clear the bit
   V(sem)
8. Why do we use binary semaphores instead of locks for device drivers?
   Because the thread that initiates the communication with the device might not be the same as the one that handles the interrupt and acknowledges the completion. Since semaphores have no owners, it is better than locks in this case.
9. Give the pseudocode for writing character to device data register using the device driver write handler and interrupt handler for serial device.
10. Describe differences between port-mapped I/O and memory-mapped I/O.
    Port mapped I/O: small, separate address space, each device is given a port number, fast, restrictive
    Memory mapped I/O: use the same address space, slow, each device needs to check every signal
11. Describe the difference between program-controlled I/O and direct memory access.
    Program-controlled I/O: the CPU does the job; it will be busy
    Direct memory access: the CPU issues a command to the disk to do the job, and it can do other things while waiting
12. A disk has a total capacity of 2^{32} bytes. The disk has a single platter with 2^{20} tracks. Each track has 2^{8} sectors. The disk operates at 10000 RPM and has a maximum seek time of 20 milliseconds.

a. How many bytes are in a track?
   2^{12}
b. How many bytes are in a sector?
   2^{4}
c. What is the maximum rotational latency?
   $6\ ms$
d. What is the average seek time and average rotational latency?
   Average seek time: 10 ms
   Average rotational latency: 3 ms
e. What is the cost to transfer 1 sector?
   $6\ ms \cdot \frac{1}{2^8} = \frac{6}{2^8}\ ms$
f. What is the expected cost to read 10 consecutive sectors from this disk?
   $10 + 3 + \frac{60}{2^8} = 13 + \frac{60}{2^8}\ ms$

13. Why is larger, sequential transfers to/from a di3sk device more efficient than smaller ones?
    Sequential transfers are more efficient per byte, since the seek time dominates, and sequential access means the seek time for the second to last transfer is 0 (or very small if we still need to move tracks)
14. What is a disadvantage of Shortest Seek Time First (choosing the closest request)?
    Starvation - there can be a never-ending stream of requests that are close to the current position of read/write head, which means a request at a track that is far away might never get processed.
15. What does writing to the status register and the sector number register of the sys/161 disk controller do?
    Issue a command with the sector number as a parameter
16. Give the pseudocode for writing to sys/161 disk using a write handler and an interrupt handler.
    Write handler:

    P(disk_sem)
    Write to the status register
    Write to the data buffer
    P(disk_completion_sem)
    V(disk_sem)

    Interrupt handler:

    Write to the interrupt register to ack completion
    V(disk_completion_sem)
17. Give the pseudocode for reading from sys/161 disk using a read handler and an interrupt handler.
    Read handler:

    P(disk_sem)
    Write to the status register
    Write to the data buffer
    P(disk_completion_sem)
    Get the data from the transfer buffer

V(disk_sem)

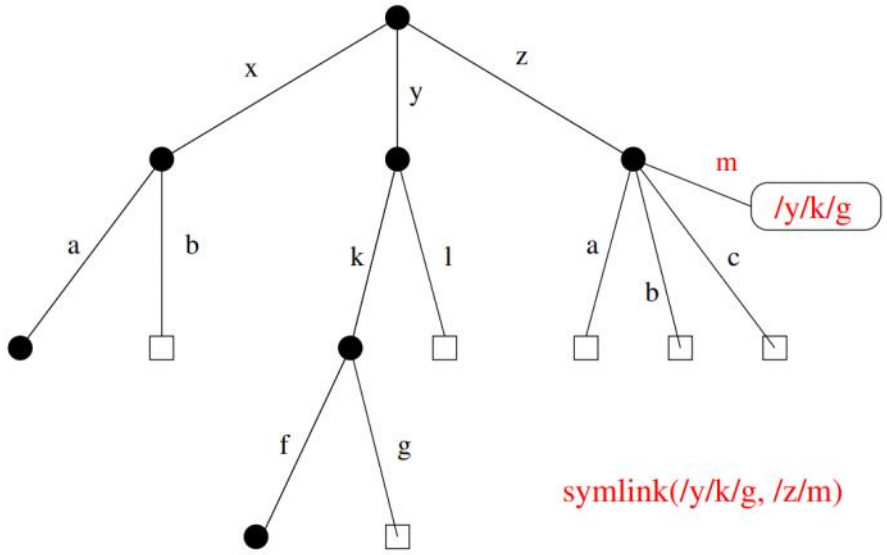
Interrupt handler:

Write to the interrupt register to ack completion
V(disk_completion_sem)

File System

Friday, December 9, 2022

1. What are logical file system, virtual file system and physical file system?
   Logical file system - provides high level API to users
   Virtual file system - abstract different file systems into one
   Physical file system - how files are actually stored on disk
2. What does open do to a file?
   Returns a file descriptor
3. Why do we need seek for read and write?
   To enable non-sequential read and write
4. Write the pseudocode for reading the first 64000 bytes of a file, 64 bytes at a time.
5. Write the pseudocode for reading the first 64000 bytes of a file, 64 bytes at a time, in reverse order.
6. What will happen if the file position for lseek is invalid?
   There will be no error but there can be errors in subsequent read/write
7. What is contained in a directory?
   Mapping from pathnames to i-numbers
8. Can directories be leaves?
   Empty directories are leaves
9. Why is only the kernel permitted to directly edit directories?
   To prevent user programs from changing the file system data structure
10. What is a hard link?
    An association between a name (string) and an i-number
11. Can hard links be made to a file once a file is created?
    Yes
12. Can a file has multiple i-numbers? Multiple pathnames?
    No; Yes (created through hard links)
13. Why is not possible to link to a directory? Give an example.
    To avoid cycles
14. What is the difference between a hard link and a soft link?
    Hard link has referential integrity. When the number of hard links go to 0, the file is deleted.
    Soft link does not have referential integrity. There can be dangling soft links
15. What will happen if /y/k/g is deleted and we try to open /z/m



There would be an error since /z/m is now a dangling soft link.

16. Describe what mounting does.
    Creates a single, hierarchical namespace that combines the namespaces of two file systems
17. What is one difference between RAM and disks related to read and write?
    RAM is byte addressable. Disk is sector addressable
18. Give 3 reasons why we group consecutive sectors into a block.
    1. better spatial locality
    2. reduces the number of block pointers
    3. 4 KB block is a convenient size for on demand paging
19. What is stored in an i-node? What is stored in a superblock?
    i-nodes contains file meta-data
    Superblock contains the metadata for the entire file system, e.g. Number of inodes, location of bitmaps, location of inode array.
20. If the disk blocks have 32-bit addresses, and each have size 4 KB, what is the maximum disk size?
    $2^{32} \cdot 4kB = 2^{32} \cdot 2^{12} = 2^{44}\ bytes = 16\ TB$
21. Suppose block size is 4 KB and pointer size is 4 bytes.
    a. What is the maximum file size using 12 direct and 1 indirect pointers?
       $(12 + 1024) \cdot 4\ KB$
    b. What is the maximum file size using 12 direct, 1 indirect and 1 double indirect pointers?
       $\left(12 + 1024 + 1024^2\right) \cdot 4\ KB$
    c. What is the maximum file size using 12 direct, 1 indirect, 1 double indirect and 1 triple indirect pointers?
       $\left(12 + 1024 + 1024^2 + 1024^3\right) \cdot 4\ KB$
22. How do we know the location of root i-node?
    It's usually in a specific location.
23. List the steps for reading from the file /foo/bar and reading three data blocks.
    Read root i-node
    Read root data to get foo's i-number
    Read foo i-node to get data block
    Read foo data to get bar's i-number
    Read bar i-node to check for permissions

Read bar i-node to get first data block
Read first data block
Write access time to bar i-node
Read bar i-node to get second data block
Read second data block
Write access time to bar i-node
Read bar i-node to get third data block
Read third data block
Write access time to bar i-node

24. List the steps for creating the file bar in the directory /foo and writing three data blocks.
    Read root i-node
    Read root data to get foo's i-number
    Read foo i-node to get foo data block
    Read foo data block to check whether the file bar already exists
    Read i-node bitmap to find a new i-node
    Write i-node bitmap to indicate a new i-node is being used
    Write foo data to store the mapping from /foo/bar to the i-number (create a hard link)
    Write foo data to create a new entry in the foo directory
    Read and write to initialize bar's i-nodes
    Write foo's i-node with a new value for the last time modified
    Read bar i-node
    Read data bitmap
    Write data bitmap
    Write bar data
    Write bar i-node
    Read bar i-node
    Read data bitmap
    Write data bitmap
    Write bar data
    Write bar i-node
    Read bar i-node
    Read data bitmap
    Write data bitmap
    Write bar data
    Write bar i-node
25. Describe one advantage and one disadvantage of chaining.
    Advantage: simple to implement, space efficient; ok for sequential access
    Disadvantage: very bad for random access
26. Describe one advantage and one disadvantage of external chaining.
    Advantage: better random access than chaining since we don't need read a lot of blocks (the table is smaller compared to the data)
    Disadvantage: need extra space
27. What does crash consistent mean?
    The file system data stays consistent after a system failure
28. What is write-ahead logging?
    Log the changes made to the metadata of a file system. Try to perform the changes, and apply the changes that have been logged afterwards.