Operating System

Donglin Jia

May 6, 2020

Contents

1 Multi-Programming **3**
- 1.1 Thread . . . 3
- 1.2 Concurrent Program Execution . . . 4
- 1.3 Timesharing and Content Switches . . . 4
- 1.4 Thread States . . . 5
- 1.5 Thread Blocking . . . 6
- 1.6 Interrupt . . . 6
- 1.7 Preemptive Scheduling . . . 7
- 1.8 Synchronization . . . 7
- 1.9 Critical Section . . . 7
- 1.10 Race Condition . . . 8

2 Concurrency **9**
- 2.1 Enforcing Mutual Exclusion With Lock . . . 9
- 2.2 Condition Variables . . . 11
- 2.3 Deadlocks . . . 11

3 Processes and Kernel **12**
- 3.1 Processes . . . 12
  - 3.1.1 Four Important processes . . . 13
- 3.2 System Calls . . . 14
  - 3.2.1 Kernel Privilege (high-level privilege) . . . 14
  - 3.2.2 Interrupts . . . 15
  - 3.2.3 Exceptions . . . 15
  - 3.2.4 How Syscall works . . . 15
  - 3.2.5 Example of Syscalls . . . 16
  - 3.2.6 User and Kernel Stacks . . . 16
  - 3.2.7 Multiprocessing . . . 17
  - 3.2.8 Kernel Library (steps with example) . . . 17

4 Virtual Memeory **18**
- 4.1 Physical Memory . . . 18
- 4.2 Virtual Memory . . . 18

4.3 Address Translation 19
4.4 Virtual Memory implementation 19
4.4.1 Goals: 19
4.4.2 Dynamic Relocation: 19
4.4.3 Segmentation: 20
4.4.4 Paging: 20
4.4.5 Page Table: 21

Chapter 1

Multi-Programming

1.1 Thread

- What is a thread?
  - Thread is a sequence of instructions.
  - A normal sequential program consists of a single thread of execution.
  - Threads provide a way for programmers to express concurrency (i.e. multiple programs or sequences of instructions running, or appearing to run at the same time) in a program.
  - There are multiple threads of execution.
    - Threads may perform the same task.
    - Threads may perform different tasks.
- Properties of Thread:
  - Resource Utilization: blocked/waiting threads give up resource.
  - Parallelism: multiple threads exectuing simultaneously.
  - Priority: higher priority, more CPU time; lower priority, less CPU time.
  - Modularization: organization of execution tasks.
- Important Notes for Thread:
  - A thread can create new threads using thread_fork.
  - All threads share the program global variables and heap, but they have private stack.
  - Operating system control which thread to run, orignal thread and the new thread runs concurrenctly.
  - Once the threads start running, you cannot predict the behavior since you have no control.

1.2 Concurrent Program Execution

- One program has at least one thread and all threads share that program address space.
- Operating system job to prevent stacks overlapping each other → but also make sure to get the maximum stack.
- **Multithread program gives:**
  - The illusion that be able to runs multi-things at the same time.
  - Better organization of the program
  - Better performance of the program
- **Implementation Concurrent Thread**
  - Hardware support: P(processors) C(cores) M(multithreading per core) PCM threads can execute simultaneously.
  - Timesharing: Multiple threads take turns on the same hardware(share CPU); rapidly switching between threads so all make progress; they are all making progress, use CPU one by one.
  - Hardware support + Timesharing = PCM + timesharing.

1.3 Timesharing and Content Switches

- When timesharing, the switch from one thread to another is called a content switch.
- **During the timesharing:**
  - Decide which thread will run next (go to operating system schedulor).
  - Save the states(register value) of the current thread.
  - Load the states(register value) of the nwe thread.
  - Make sure that the thread that running CPU again at the exact position and exact states when it pick up.
  - Thread context must be saved/restored carefully.
  - switchframe_switch (callee): save all the callee-save registers.
  - thread_switch (caller): save all the caller-save registers.
- **What causes context switch?**
  - thread_yield: running thread voluntarily allows other threads to run; give up CPU voluntarily.

- **Four ways of context-switch:**
  - Go to Blocked state → wait something.
  - Preemption: involuntarily give up the CPU since the scheduling quantum count down expires → interrupts → context switch.
  - Thread_yield: voluntarily called thread yield.
  - Thread terminated;

1.4 Thread States

- **Thread states (3 states)**
  - Running: thread is executing on the CPU right now; if CPU only support one thread at a time, then only one thread has that state.
  - Ready: the thread is not waiting for anything except CPU; waiting to run instructions.
  - Blocked: any threads that is waiting for something that can continue. Waiting to receive the "order" to execute.
- **Transaction:**
  - Ready to running: called by contextswitch.
  - Running to ready: called by yield.
  - Running to blocked: need something so that need to go to blocked to wait for that.
  - Blocked to ready: when something becomes available, then some threads will take the blocked thread and push to ready state.
- **Thread Stack after Voluntary Context Switch**
  - Program calls thread_yield first to yield CPU.
  - thread_yield then calls thread_switch, to perform a context switch.
  - thread_switch chooses a new thread, calls switchframe_switch to perfrom low-level content switch.
- **Timesharing and Preemption:**
  - timesharing – scheduling quantum (the upper bound on how long a thread can run before must yield the CPU), it is involunteering stop.
  - preemption forces a running thread to stop
  - Since CPU can only run one thread and that one is not aware that it needs to stop; the only way to stop a thread is through hardware – using Interrupt!

1.5 Thread Blocking

- When a thread blocks, it stops running:
  - The scheduler chooses a new thread to run.
  - A context switch from the blocking thread to a new thread occurs.
  - The blocking thread is queued in a Wait Queue.
- Eventually, a blocked thread is signaled and awakened by another thread.

1.6 Interrupt

- An interrupt is an event that occurs during the execution of a program.
- Interrupts are caused by system devices.
- CPU know who causes the interrupt and OS handle the interrupts.
- CPU has a trap table where stores some addresses of some codes.
  - CPU → receive interrupts → stop executing user's program → execute the code stored in address of the trap table (Interrupt handler (operating system code))
- **Interrupt handler**
  1. create a trap frame; save every register in the system.
  2. called the actual handler → perform device-specific processing.
  3. Restores the saved thread context from the trap frame and resuems execution of the thread.
- Trap frame is always followed by the interrupt.
- Interrupt handler stack frame followed by the trap frame.
- **Difference between trap frame and switch functions**
  - Trap frame saves all the registers comes before (the previous running program's registers)
  - thread_switch save all the registers between trap frame and itself (i.e. callee-save registers)
  - switchframe save all the registers (i.e. caller-save registers)
  - Trap frame called by CPU not program(cannot be predict when to be called, so we need to save every register), while switch functions called by program!

1.7 Preemptive Scheduling

- Preemptive Scheduling uses the schduling quantum.
- Quantum cannot be too long → not effective timesharing; cannot be too short → spend more time on timesharing, not do so much work!
- Every time the new thread get a fresh quantum, so quantum does not carry over.
- Every computer has clock → run the count down → when the count down is expire → clock fire a interrupt → causes CPU to execute the interrupt handler so OS take control of the interrupts (i.e. execute content switch (yield → thread_switch → switchframe).
- **Two passes to implement**
  1. Every time there is a content switch, tell the clock to start a new count down → really slow.
  2. Clock count down in a short period and repeat, when there is a time interrupt, go check (when the thread start running (record it)) whether it exceed its quantum, if yes, kick out, otherwise keep running.

1.8 Synchronization

- All threads in a concurrent program share access to the program's global variables and the heap (read & write).
- Any block of codes in a concurrent program where we have multiple threads reading and writing the shared variable(either global or heap) is called critical section.

1.9 Critical Section

- The order of the instructions to be exectued in that section matters.
- No control during preemption, no control over which thread run next → OS have ability controlling.
- Context switch happens expected to assembling, instead of C code.
- There are multiple programs running background, therefore, interrupts happens every time.

1.10 Race Condition

- When the program result depends on the order of execution, it can cause: incorrect output, return error; memory error.
- Race conditions occurs when multiple threads are reading and writing the same memory at the same time.
- Source of race conditions:
  - implementation
  - compiler
  - CPU
- Compiler and CPU introduce the race conditions due to optimization – rearranging the code.
- By applying memory model, CPU and complier knows which optimization is safe/unsafe.
- In C99, volatile disable the optimization of CPU.
  - Register allocation optimization: it keeps the up-to-date value in a register for as long as possible can.
  - func a, func b, might use different register for register allocation → do not know which one has the up-to-date value.
  - volatile → turn register optimization off; it forces to load from memory for every use.
- Identify race conditions:
  - Inspect each variable; is it possible for multiple threads to read/write it at the same time?
  - Constants and memoery that all threads only read.

Chapter 2

Concurrency

2.1 Enforcing Mutual Exclusion With Lock

- Acquire/release must ensure that only one thread at a time can hold the lock, even if both attempt to Acquire at the same time.
- If a thread cannot Acquire the lock immediately, it must wait until the lock is available.
- Must ACQUIRE the lock before exeucting the shared variable.
- C code cannot stop the interrupts & context switch, we need hardware to help:
  - provide a way to implement atomic test-and-set (i.e. atomic means cannot be interrupted!) for synchoronization primitives like locks.
- Mips & Arm use pair of value to implement Test-and-Set, they use the pair of values to check only.
  - load-link: ll get the lock immediately;
  - store conditionally: sc get the value of the current lock;
  - return success if both value are the same; fail otherwise
    - if success: return tmp (which is the old value (immediate value of lock))
      - if tmp is false, then we get the lock.
      - if tmp is true, then we keep spinning.
    - if fail: return true, then we keep spinning.
  - Spinlock and lock:
    - They both have a owener!
      - spinlock is owned by CPU.
      - lock is owned by thread.
    - Spinlock disable the interrupts.

        - minimize the spinning.
        - preemption is disable on the CPU → because they are doing busy wait → take up CPU.
        - DO NOT use spin lock to protect large critical sections.
    - Some CPU can queue some interrupts during the spinlock is on.

Semaphores

- Keep tracking the number of resources that are available. Try to synchoronizate in terms of number of resources.
- Ownership! lock care about, but semaphores does not.
- Can release it without own it.
- Having a counter to keep track, the counter is non-negative.
    - If the counter is zero, we go to sleep and wait for the counter to be anything else.
    - If the counter is greater than zero, we take the resourse and decrement the counter.
    - P(take the resourse, space++, item–) vs V(return the resourse, space–, item++).
    - No way to check the value of the counter in C, the only thing to do is P & V.
- **Type of Semaphores:**
    - **Binary Semaphore:**
        - Similar to the lock, create with one initial resouce, when P it, take one resouce, V it, return one resource.
        - Do P first, then V. → offer mutual exclusion.
        - Initial resource and Maxmimun resource are 1.
    - **Counting semaphore:**
        - A semaphore with an arbitrary number of resouces, no maximum resources.
        - Do not care about the order of P and V.
    - **Barrier semaphore:**
        - Use semaphore to force one thread wait another thread to finish.
        - Start the semaphore with zero resouces.
        - Threads need to wait for the other thread needs to P sempahore once, for the threads are waiting on has to V the semaphore to access.
- Proceducer/Consumer Synchronization

```
struct semaphore* Item, Space
Item = sem_create("Buffer Item", 0);
Space = sem_create("Buffer Spaces", N)
Producer's Pseudo-code:
P(Spaces)
add item to the buffer
V(Items)
Consumer's Pseudo-code:
P(Items)
remove item from the buffer
V(spaces)

```

2.2 Condition Variables

- The safe to simultaneously fall asleep and give up the lock, so other threads can change the condition, then wake up by someone and own the lock.
- wait: give up the lock → go to sleep → acquire the lock.
- **Difference:**
  - lock: mutual exclusion.
  - sempahore: resources.
  - CV: conditions.

2.3 Deadlocks

- It forever stopped and never execute or make any progress.
- Program stops running.
- OS cannot detect the deadlock; since it cannot differentiate the deadlock or waiting for something.
- **Solutions:**
  - No Hold and Wait: prevent a thread from requesting resource while owning a resource (no wait, includes sleep and busy wait).
  - Resource Ordering: Order the resource types and require that each thread acquire resources in increasing resource type order. (acquire in increasing order).

Chapter 3

Processes and Kernel

3.1 Processes

- Processes never execute codes, only **thread** execute the code.
- Processes has at least one thread.
- Execution environment created by OS for the program to run in; A structure that contains every thing that a program needs to run.
- Every process has its own process space, has own array of threads.
  - each tag in chrome is a process → has its own address space.
- Lots of processes running will slow down the server.
- Using Top to check the current state of the system.
- Using PID (Process Identification) → every process has its own unique PID.
- OS needs to keep track of every process that in the system → maintain, search, edit, etc.

3.1.1 Four Important processes

- **Fork**
  - Create a new process that is identical clone of the caller (parent).
  - Although it has its own address space and own thread array, the program and the address space is an identical clone copy of the parent → the heap, stack, the global, the code, the program counter, the thread, the registers are the same.
  - Process considers the relationship → with fork, we need to maintain the inside process structure and our relationship to children.
  - The caller fork is parent and new process is child → needs to maintain this relationship.
  - Program counter is the same, so after executing, it means when they return from executing they both execute the same instruction → both of children and parents return from fork.
  - The return value for parent and children are different:
    - For the fork return, parent return the PID of the child.
    - Child fork returns 0.
    - Each process has its own PID → they do not share process space → but the content of that address space are the same.
- **_exit**
  - It terminates the calling processes.
  - Keep track of why the process is terminated -¿ leave a message (integer → exit status) → why the process died.
- **waitpid**
  - Let a process wait for another to terminate and retrieve its exist status code.
  - The current process will be blocked until the waited process terminates → after wake up → waitpid retrive the reason(message) the process die
  - Process only allow to wait on its children → only care about its children.
  - One process death does not affect its child process.
- **execv**
  - Changes the program that a process is running → does not create a new program.
  - Keep PID, process structure, child-parent relationship same.
  - Address space and thread changes.
  - Execv takes two params.

    - The second of params is a pointer to array of arguments.
    - If execv does work, it does not return.
    - Take the old address space and create a new address space → cannot return old address space.
    - Return if fail → try to load program → fail.
        - Wrong program.
        - Not enough memory for creating address space.
        - Not enough space for creating threads.

3.2 System Calls

- The calls that programmer use to interact with kernel.
- Everything is done by kernel. (open files, print, all the function calls).
- Make a system call (system functions) use the system calls library (it is NOT part of kernel) → ask kernel to do → return to the user program.
- User privileged VS kernel privileged:
    - User program → unpriviledged code.
    - Kernel priviledged → priviledged code.
- To perform a system call, the application needs to cause an exception to make the system call.
    - li v_0 0 → v0 put the syscall code.
    - The kernel checks v0 to determine which system call has been requested → using kernel ABI (Application Binary Interface).
    - ABI all the information to share with user (size of Int, type, syscall code).

3.2.1 Kernel Privilege (high-level privilege)

- Execution privilege controled by CPU.
- User program has least privilege → cannot execute/assess only permitted privileged code → throw exception.
- The only way to run kernel code is by interruptions(hardware) and exceptions(software).
- User program is totally isolated from implementation of Kernel.

3.2.2 Interrupts

- Interrupts are raised by hardward.
- CPU receive the interrupts → find the kernel(by using the stored address of interrupt handler) → CPU execuate the interrupt handler → flip to the kernel privilege mode → interrupt handler save the stage, trap frame → kernel do things.

3.2.3 Exceptions

- CPU raise the exceptions → flip privilege → call the predetermined location → exception handler.
- Exception handler is part of the kernel.
- MIPs treat exceptions and interrupts are the same.

3.2.4 How Syscall works

- System calls take parameters and return values.
- The application places parameter value in kernel-specified location before the syscall, and looks for return value in that location after the exception handler returns.
  - How do the application know the kernel-specified location?
    - The locations are part of the kernel ABI.
  - Parameter and return value placement is handled by the application system call library functions.
  - On MIPS, parameters go in registers a_0,a_1,a_2,a_3
    - result success/fail code is in a_3 on return.
    - return value or error code is in v_0 on return.
      - lw a_0 ... a_3
      - syscall → raise exception
    - If a_3 return success, then v_0 has return value.
    - If a_3 return fail, then v_0 has error code(from AVI)

- **System Call Software Stack**
  - Application calls library wrapper function for desired system call.
  - Library function performs syscall instruction.
  - Kernel exception handler runs.
    * create trap frame to save program state.
    * determine this is a syscall exception.
    * determine which system call is being requested.
    * call the kernel implementation of it.
    * set the return value.
    * flip the priviledge back to kernel.
    * return from the exception.
  - Library wrapper function finishes and returns from its call.
  - Application continues execution.

3.2.5 Example of Syscalls

- printf
- malloc → depends, if there is no enough memory space, malloc needs to ask OS for more memory.

3.2.6 User and Kernel Stacks

- Every OS process thread has two stacks, although it only uses one at a time.
- User Stack: while application code is executing
  - located in the application's virtual memory (address space).
- Kernel Stack: used while the thread is executing kernel code, after an exception or interrupt.
  - Lives in the kernel.
  - Stack holds the trap frames and switch frames (because kernel creates those two frames).
- In order to avoid user stack to modify kernel stack, so when flip to kernel mode → everything(kernel's) flip to the kernel stack.
- Kernerl stack can overflow the user stack, so we keep they are separated.

3.2.7 Multiprocessing

- Multiple processes existing at the same time.
- All process share the availabe hardware resources.
- Thread execute the program instead of process → we still have context switch, thread yield, thread_switch.
- The OS ensures that processes are isolated from one another.

3.2.8 Kernel Library (steps with example)

1. v_0 put the system call code, in a_0 - a_3 put the parameter to tell which system call
2. raise the exception
3. switch priviledge mode and disable interruption
4. execute the common exception handler
5. switch the user stack to the kernel stack
6. common exception handler save the trap frame on the kernel stack
7. common exception called mips trap
8. mis trap figure out which exception is it
9. turn the interruption on
10. called that specific exception handler (syscall → system call dispatcher)
11. inside dispatcher (get v_0 a_0-a_3)
12. for example (fork)
13. call sysfork
14. return a_3(succeed, fail) v_0 (succeed code, error code)
15. system call return to mips trap
16. return common exception
17. restore the trap frame to the CPU and flip back user stack
18. switch to unpriviledge to priviledge mode
19. user program running

Chapter 4

Virtual Memeory

- Every user program is using virtual memory → it is fake physical memory
  - Isolation from the physical memory
  - Security reason
- Translation from virtual address to physical address → help determine CPU which process is running
- Different process using different translation (parameters to that translation function is different based on the process) → CPU only translate current process → proccess is isolated.

4.1 Physical Memory

- If physical addresses are P bits, then the maximum amount of addressable physical memory is 2^p bytes.
- OS manage the memory
- Most compiler only do 32 bits → only 4GB RAM has been used → if want to use 8GB → switch to 64bits compile

4.2 Virtual Memory

- Kernel provides a separate, private virtual memory for each process.
- The virtual memory of a process holds the code, data, and stack for the program that is running in that process.
- It is possible to use more bits virtual than physical

  - store the address on disk
  - load part of the address when we need it
- If virtual memory are V bits, the maximum size of a virtual memory is 2^v bytes.
- Running applications see only virtual address.
- Everything in user program are using virtual memory. → never ever see physical address.
- Why VM?
  - isolate processes from each other
  - potential to support VM larger than physical memory
  - more process running

4.3 Address Translation

- Each virtual memory is mapped to a different part of physical memory.
- At least one translation required for every assembly instruction (i.e. program counter).
- Virtual address is translated to its corresponding physcial address.
- Address translation is performed in HARDWARE. (MMU → Memory Management Unit) provided by the kernel.

4.4 Virtual Memory implementation

4.4.1 Goals:

- Transparency: user program believes the address is real.
- Protection: offer isolation from different processes and kernel.
- Efficiency: need to be done for every single instruction.

4.4.2 Dynamic Relocation:

- For each process we need to store 2 integers → offset and size.
- if size ¡ limit, just add the offset
- if not → raise exception and terminate the program

- Although efficient, suffer fragmentation → external waste space
- Memory waste problem → 1KB used (stack start from the middle) → 2KB needed → internal waste space

4.4.3 Segmentation:

- Instead of allocating a continue chunk of memory, we only allocate the exact memory that going to use → allocate 3 times (data, code, stack) instead of once.
- Then we do dynamic allocation on the segment
- The kernel maintains an offset and limit for each segment → hardware tells OS number of segment to support
- K bits for the segment ID, we have up to 2^k segments with 2^{v-k} bytes per segment
- address: seg# | offset → hex.
- MMU has a relocation register and limit register for each segment.
  - split the VM into segment number and address within segment
  - if offset (i.e. the second part of VM separated from seg#) $\geq$ limit, then raise exception
  - else p = offset + relocation offset
  - Efficient on space and time → two value stored for segment
  - Disadvantage:
    - Fragmentation problem
    - MMU is large(multiple registers) → take physical space (too many MMU footprint) → take too much CPU space

4.4.4 Paging:

- Logically divide physical memory equal size → physical page ÷ frame; normally we have 4KB.
- To get fragmentation: let any page of virtual memory map to any frame of physical memory.
- We never allocate the chunk of memory, instead we only allocate several individual memory for one frame → no more external fragmentation
- But still have internal fragmentation → since the smallest page is 4KB, so might lose a little bit → do not care
- Any page → Any frame → Any memory location (more process running than the RAM support)

4.4.5 Page Table:

- It has frame number and valid bit.
- Valid bit → track whether the page is in use → through exception if not.
- Number of PTEs(Page Table Entry) = Maximum Virtual Memory Size / Page Size.
  - e.g. 32-bit virtual address → 2^{32} Memory Size
  - page size of 2^{12} bytes → we have 2^{20} PTES
- Determine the page number and offset → PG# | offset
- Number of Bits for PG# = $\log(number\ of\ PTES)$
- Number of Bits for Offset = $\log(Page\ Size)$