Notes for ECE4820J

Tianzong Cheng

Contents

1. Operating Systems Overview

1.1. Computers and Operating Systems

- Job of an Operating System (OS):
  - Manage and assign the hardware resources
  - Hide complicated details to the end user
  - Provide abstractions to ease interactions with the hardware

1.2. Hardware

- CMOS: save time and date, BIOS parameters
  - powered by the CMOS battery

1.3. Basic concepts

- Five major components of an OS:
  - System calls: allows to interface with user-space
  - Processes: defines everything needed to run programs
  - File system: store persistent data
  - Input-Output (IO): allows to interface with hardware
  - Protection and Security: keep the system safe
- Monolithic kernel, micro kernel
  - Monolithic kernel
    - Everything in kernel space
    - Pros
      - High performance: communication between kernel components avoid overhead
    - Cons
      - Changes or bugs can affect the whole kernel
  - Micro kernel
    - Minimal kernel space
    - Pros
      - Higher stability
    - Cons
      - Performance overhead

1.4. Key Points

- What is the main job of an OS?
- Why are there so many types of OS?
- Why is hardware important when writing an OS?
- What are the main components of an OS?
- What are system calls?
  - switch to kernel mode to run privileged instruction
  - software interrupt

2. Processes and Threads

2.1. Processes

- **Process is the unit for resource management**
- Multiprogramming issue: rate of computation of a process is not uniform / reproducible
- Process hierarchies
  - UNIX
    - parent-child
    - “process group”
  - Windows
    - All processes are equal
    - A parent has a token to control its child
    - A token can be given to another process
- A simple model for processes:
  - A process is a data structure called process control block
  - The structure contains important information such as:
    - **State**
      - ready (input available)
      - running (picked by scheduler)
      - blocked (waiting for input)
    - Program counter
    - Stack pointer
    - Memory allocation
    - Open files
    - Scheduling information
  - All the processes are stored in an array called process table
- Upon an interrupt the running process must be paused:
  1. Push on the stack the user program counter, PSW (program status word), etc.
  2. Load information from interrupt vector
  3. Save registers (assembly)
  4. Setup new stack (assembly)
  5. Finish up the work for the interrupt
  6. Decides which process to run next
  7. Load the new process, i.e. memory map, registers, etc. (assembly)
- Difference between concurrency and parallelism
  - concurrency: multitasking on a single-core machine
  - parallelism: multicore processor

2.2. Threads

- **A thread is the basic unit of CPU utilisation**
- Each thread has its own
  - thread ID
  - program counter
  - **registers**
  - **stack**

- Threads within a process share
  - code
  - data
  - OS resources
- In C or C++, if one thread crashes, it can cause the whole thread to crash.
  - For example, if one thread crashes due to invalid memory access, it can corrupt the data in the shared memory, causing other threads to crash.

2.3. Implementation

- POSIX threads (pthread)
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
  - `void pthread_exit(void *retval)`
  - `int pthread_join(pthread_t thread, void **retval)`
  - Release CPU to let another thread run: `int pthread_yield(void)`
  - `int pthread_attr_init(pthread_attr_t *attr)`
  - `int pthread_attr_destroy(pthread_attr_t *attr)`
- Threading models
  - Threads in user space - N:1
    - Multiple user-level threads are mapped to a single kernel thread
    - Scheduling and management are handled at the user level
    - Pros
      - Fast because context switching in kernel level is reduced
    - Cons
      - If one thread is blocked, the entire process is blocked
      - Time slices are allocated to processes, not threads. So each thread gets a smaller time slice.
  - Threads in kernel space - 1:1
    - Each user-level thread is mapped to a separate kernel thread
    - Improves responsiveness and parallelism
    - Pros
      - Blocking one thread does not affect other threads
      - Processes with several threads get more CPU time
    - Cons
      - Bigger system cost
  - Hybrid threads - M:N
    - A threading library schedules user threads on available kernel threads
    - Pros
      - Combine the strengths of user-space threads and kenel-space threads
    - Cons
      - Complexity of implementation
  - Remarks
    - Hybrid threads are less common in modern OSes

2.4. Key Points

- What is a process?
- How can processes be created and terminated?
- What are the possible states of a process?
- What is the difference between single thread and multi-threads?
- What approaches can be taken to handle threads?

3. Interprocess Communication

3.1. Exhibiting the problem

Race conditions

Problems often occur when one thread does a “check-then-act”:

```
if (x == 5) {  // The "Check"
  // x is modified in another thread
  y = x * 2;  // The "Act"
}
```

A typical solution is:

```
// Obtain lock for x
if (x == 5) {
  y = x * 2;
}
// Release lock for x
```

3.2. Solving the Problem

Critical region: Part of the program where shared memory is accessed:

- No two processes can be in a critical region at a same time
- No assumption on the speed or number of CPUs
- No process outside a critical region can block other processes
- No process waits forever to enter a critical region

The lock should be obtained before checking and modifying the resource.

3.2.1. Peterson’s Algorithm

Peterson’s algorithm is symmetric for two processes.

```
#define TRUE 1
#define FALSE 0
int turn;
int interested[2];
void enter_region(int p) {
  int other;
  other = 1 - p;
  interested[p] = TRUE;
  turn = p;
  while (turn == p && interested[other] == TRUE)
}
void leave_region(int p) {
  interested[p] = FALSE;
}
```

Peterson’s idea only supports two processes and relies on busy waiting.

3.2.2. Mutual Exclusion at Hardware Level

- Disabling interrupts
  - Only available when there is only one CPU
- Use atomic operations
  - Test and set lock `TSL`

3.2.3. Semaphore

A semaphore is a positive integer and is only managed by two actions:

```
sem.down() {
  while(sem==0) sleep();
  sem--;
}
sem.up() {
  sem++;
}
```

An awaken sleeping process can complete its `down`.

Checking or changing the value and sleeping are done atomically:
- Single CPU: disable interrupts
- Multiple CPUs: use `TSL` to ensure only one CPU accesses the semaphore

A mutex is a semaphore taking values 0 (unlocked) or 1 (locked).

On the request of locking a mutex, if the mutex is already locked, put the calling thread to sleep.

The implementation of a mutex using `TSL`:

```
mutex-lock:
  TSL REGISTER, MUTEX
  CMP REGISTER, #0
  JZ ok
  CALL thread_yield
  JMP mutex-lock
ok:
  RET
mutex-unlock:
  MOVE MUTEX, #0
  RET
```

Using a mutex to solve the producer-consumer problem:

```
pthread_mutex_t m;
pthread_cond_t cc, cp;
int buf = 0;

void *prod() {
  for (int i = 1; i < MAX; i++) {
    pthread_mutex_lock(&m);
    while (buf != 0) {
      pthread_cond_wait(&cp, &m);
    }
    buf = 1;
    pthread_cond_signal(&cc);
    pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}

void *cons() {
  for (int i = 1; i < MAX; i++) {
    pthread_mutex_lock(&m);
    while (buf == 0) {
      pthread_cond_wait(&cc, &m);
    }
    buf = 0;
    pthread_cond_signal(&cp);
    pthread_mutex_unlock(&m);
  }
  pthread_exit(0);
}
```

The `pthread_cond_wait()` function atomically blocks the current thread waiting on the condition variable specified by `cond`, and releases the mutex specified by `mutex`. The waiting thread unblocks only after another thread calls `pthread_cond_signal`, or `pthread_cond_broadcast` with the same condition variable, and the current thread reacquires the lock on `mutex`.

Remarks: Mutexes and semaphores are different

- Purposes
  - Semaphore is designed for synchronization
- Usage
  - Mutex can only be release by the owner
  - Semaphore can be controlled by any process

3.3. More Solutions (TODO)

3.3.1. Monitors

Basic idea behind monitors:

- The mutual exclusion is not handled by the programmer
- Locking occurs automatically
- Only one process can be active within a monitor at a time
- A monitor can be seen as a “special type of class”
- Processes can be blocked and awaken **based on condition variables and wait and signal functions**

3.4. Key Points

- Why is thread communication essential?
- What is a critical region?
- Do software solutions exist?
- What is an atomic operation?
- What are the two best and most common solutions?

4. Scheduling

4.1. Requirements

- When to decide what process to run next
  - A new process is created
  - A process exits or blocks
  - IO interrupt from a device that has completed its task
- Compute bound v.s. input-output bound
- Two main strategies
  - Preemptive
    - A process is run for at most n ms
    - If it is not completed by the end of the period then it is suspended
  - Non-preemptive
    - A process runs until it blocks or voluntarily releases the CPU
    - It is resumed after an interrupt unless another process with higher priority is in the queue
- Goals when scheduling
  - All systems
    - Fairness: fair share of the CPU for each process
    - Balance: all parts of the system are busy
    - Policy enforcement: follow the defined policy
  - Interactive systems
    - Response time: quickly process requests
    - Proportionality: meet user's expectations
  - Batch systems
    - Throughput: maximize the number of jobs per hour
    - Turnaround time: minimize the time between submission and termination of a job
    - CPU utilization: keep the CPU as busy as possible
  - Real-time systems
    - Meet deadlines: avoid any data loss
    - Predictability: avoid quality degradation, e.g. for multimedia

4.2. Common Scheduling Algorithms

- Simplest algorithm but non-preemptive:
  - CPU is assigned in the order it is requested
  - Processes are not interrupted, they can run a long as they want
  - New jobs are put at the end of the queue
  - When a process blocks the next in line is run
  - Any blocked process becoming ready is pushed to the queue
- Shortest job first (SJF)
  - non-preemtive
  - Minimizes turnaround time
- Round-Robin scheduling
  - preemtive
  - A process runs until

    - Getting blocked
    - Its quantum has elapsed
    - Being completed
- Priority scheduling
  - Processes are more or less important, e.g. printing
  - Creates priority classes
  - Use Round-Robin within a class
  - Run higher priority processes first
- Lottery scheduling
  - preemptive
  - Processes get lottery tickets
  - When a scheduling decision is made a random ticket is chosen
  - Prize for the winner is to access resources
  - High priority processes get more tickets
- Earliest deadline first
  - preemptive
  - Process needs to announce (i) its presence and (ii) its deadline
  - Scheduler orders processes with respect to their deadline
  - First process in the list (earliest deadline) is run

4.3. Notes and Problems

- Limitations of the previous algorithms
  - They all assume that processes are competing
  - Parent could know which of its children is most important

Threads in user space is not able to run in the order of `A1 B1 A2 B2 A3 B3` (`A1 A2 A3` are threads of process `A`). Note that the kernel is not aware of the status of the threads in this case.

4.4. Key Points

- Why is scheduling the lowest part of the OS?
- What are the two main types of algorithm?
- What are the two most common scheduling algorithms?
  - First-come, first-served
  - Round Robin
- Give an example of theoretical problem related to scheduling

5. Deadlocks

5.1. Modelling the Problem

- Preemptable and non-preemptable
  - preemptable: resource can be taken away from a process without causing any negative impact

5.2. Dealing with the Problem

- Strategies to recover from a deadlock
  - Preemption
    - Take a resource from another process
  - Killing
    - Pick a process that can be re-run from the beginning
  - Rollback
    - Set periodical checkpoints on processes
    - Restart process at a checkpoint from before the deadlock

5.3. Avoiding the Problem

- States
  - Safe state: **there exists an order allowing all processes to complete**, even if they request their maximum number of resources when scheduled. It can guarantee that all processes can finish.
  - Unsafe state: the ability of the system not to deadlock depends on the order the resources are allocated and deallocated. There is no way to predict whether or not all the processes will finish.
  - An unsafe state does not necessarily imply a deadlock; the system can still run for a while, or even complete all processes if some release their resources before requesting some more.
- The Banker's Algorithm
  1. **Select a row in R whose resource request can be met.** If no such row exists there is a possibility for a deadlock
  2. When the process terminates it releases all its resources, and they can be added to the vector A
  3. If all the processes terminate when repeating steps 1. and 2. then the sate is safe. If step 1. fails at any stage (not all the processes being finished) then the state is unsafe and the request should be denied
- **Conditions**
  - **Mutual exclusion**
    - Use daemon that can handle specific output, e.g. SPOOL
    - Aside from carefully assigning resources not much can be done
  - **Hold and wait**
    - Require processes to claim all the resources at once
    - Not realistic, not optimal
    - Alternative strategy: process has to release its resources before getting new ones
  - **No preemption**: resources cannot be taken away by another process

    - Issue inherent to the hardware
    - Often impossible to do anything
  - **Circular wait**
    - Order the resources
    - **Processes have to request resources in increasing order**
    - **A process can only request a lower resource if it has released all the larger ones**
    - **Best solution** but not always possible

5.4. Key Points

- How to detect deadlocks?
  - Resource allocation graph: detect cycles
  - Timeout
- How to fix a deadlock in a clean way?
- What are the four conditions that characterize a deadlock?
- What is a common practice regarding deadlocks?
  - Preemption
  - Killing
  - Rollback

6. Memory Management

6.1. Handling Memory

- Simplest model (No memory abstraction)
  - Program sees the actual physical memory
  - Programmers can access the whole memory
  - Limitations when running more than one program
    - Have to copy the whole content of the memory into a file when switching program
    - No more than one program in the memory at a time
    - More than one program is possible if using special hardware
  - Problems
    - Protection: prevent program from accessing other's memory
    - Relocation: rewrite address to allocate personal memory
- Address space:
  - Set of addresses that a process can use
  - Independent from other processes' memory
- More memory than available might be needed
  - Processes are swapped in (out) from (to) the disk
  - OS has to manage dynamically assigned memory
- Ways to assign memory to processes
  - First fit: search for a hole big enough and use the first found
  - Best fit: search whole list and use smallest, big enough hole
  - Quick fit: maintain lists for common memory sizes, use the best
    - Fast allocation and deallocation
    - Internal fragmentation
- Virtual memory
  - Generalization of the base and limit registers
  - Each process has its own address space
  - The address space is split into chunks called pages
  - Each page corresponds to a range of addresses
  - Pages are mapped onto physical memory
  - Pages can be on different medium, e.g. RAM and swap
- Swap partition principles
  - When a process starts, its swap area is reserved and initialized
  - The new "origin" is computed
  - When a process terminates its swap area is freed
  - The swap is handled as a list of free chunks
- Two main strategies:
  - Copy the whole process image to the swap area
  - Allocate swap disk space on the fly

6.2. Paging

- Structure of a page entry:
  - Present|absent: 1|0; missing causes a page fault
  - Protection: 1 to 3 bits: reading/writing/executing

  - Modified: 1|0 = dirty|clean; page was modified and needs to be updated on the disk
  - Referenced: bit used to keep track of most used pages; useful in case of a page fault
  - Caching: important for pages that map to registers; do not want to use old copy so set caching to 0
- Page replacement: Least Recently Used
  - Hardware solution, for n page frames:
  - Initialize a binary $n \times n$ matrix to 0
  - When frame k is used, set row k to 1 and column k to 0
  - Replace the page with the smallest value
  - Example on slides, page 179
- Page replacement: Aging
  - Shift all the counters by 1 bit to the right
  - Add $2^{n-1} \cdot R$ to the counter
  - Difference from LRU: considers a period of time
- Thrashing: real storage resources are overcommitted, leading to a constant state of page fault
- Page replacement: WSClock
  - If reference bit is 1, set to 0
  - If reference bit is 0 and age is bigger than some constant
    - Clean: replace page
    - Dirty: schedule write, repeat algorithm
- Local v.s. global
  - Local: process only use the allocated portion
  - Global: dynamically allocate page frames to a process
- Adjust page frames allocation based on **page fault frequency**
  - If larger than A then allocate more page frames
  - If below B then free some page frames
- Page size
  - Optimal page size: $p = \sqrt{2se}$, where s is process size and e is average size for page entry
  - Common page frame sizes: 4KB, 8KB
- Page sharing
  - Pages containing the program can be shared
  - On a process switch do not remove all pages if required by another process: would generate many page fault
  - When a process terminates do not free all the memory if it is required by another process: would generate a crash
- Process on a page fault:
  1. Trap to the kernel is issued; program counter is saved on the stack; state of current instruction saved on some specific registers
  2. **Assembly code routine** started: save general registers and other volatile information
  3. OS search which page is requested
  4. Once the page is found: **check if the address is valid** and if process is allowed to access the page. If not kill the process; otherwise find a free page frame

5. If selected frame is dirty: have a context switch (faulting process is suspended) until disk transfer has completed. The page frame is marked as reserved such as not to be used by another process
6. When page frame is clean: schedule disk write to swap in the page. In the meantime the faulting process is suspended and other processes can be scheduled
7. When **receiving a disk interrupt to indicate copy is done**: page table is updated and frame is marked as being in a normal state
8. Rewind program to the faulting instruction, program counter reset to this value
9. Faulting process scheduled
10. Assembly code routine starts: reload registers and other volatile information
11. Process execution can continue

- Policy and mechanism
  - Low level MMU handler: architecture dependent
  - Page fault handler: kernel space
  - External handler: user space
- Page replacement algorithm in kernel space
  - Fault handler sends all information to external pager (which page was selected for removal)
  - External pager writes the page to the disk
  - No overhead, faster

6.3. Segmentation

- Segmentation: Divides memory into variable-sized segments based on the logical structure of the program, such as code, data, stack, etc.
- The size of each segment is variable
  - So segmentation only leads to external fragmentation
- External fragmentation v.s. internal fragmentation
  - Internal: Occurs if a page is not completely filled
  - External: Happens when free memory is divided into non-contiguous chunks.
- Modern systems combine paging and segmentation

6.4. Key Points

- What are the two main ways to model memory?
- What is the swap area?
- Cite two main page replacement algorithms
- Discuss the differences between paging and segmentation
  - Paging has higher swapping efficiency
  - Paging has no external fragmentation, segmentation has no internal fragmentation
- Explain external and internal fragmentation

7. Input-Output

7.1. Basic Hardware

- Categories
  - Block devices:
    - Stores information in blocks of fixed size
    - Can **directly access** any block independently of other ones
  - Character devices:
    - Delivers or accepts a stream of characters
    - **Not addressable**, no block structure
- Memory-mapped IO
  - Modern approach:
    - Map the buffer to a memory address
    - Map each register to a unique memory address or IO port
  - Strengths:
    - Access memory not hardware: no need for assembly
    - No special protection required: control register address space not included in the virtual address space
    - Flexible: a specific user can be given access to a particular device
    - Different drivers in different address spaces: reduces kernel size and does not incur any interference between drivers
  - Since memory words are cached what if the content of a control register is cached?
    - MMIO regions are typically mapped with specific flags indicating they are non-cacheable
- DMA

7.2. Hardware Interrupts

- **Precise interrupt**
  - Program counter represents the precise progress
  - All instructions before PC have been completed
  - No instruction after the one pointed by PC have be executed
  - Execution state of the instruction pointed by PC is known
- Difficult to figure out what happened and what has to happen:
  - Instructions near PC are in different stages of completion
  - General state can be recovered if given many details
  - Code to resume process is complex
  - The more details the more memory used
- Imprecise interrupts are slow to process, what is an optimal strategy? (TODO)

7.3. Software IO

- Three communications strategies:
  - Programmed IO:
    - Copy data into kernel space
    - Fill up device register and wait in tight loop until register is empty
    - Fill up, wait, fill up, wait, etc.

- Interrupt IO:
  - Copy data into kernel space, then fill up device register
  - The current process is blocked, the scheduler is called to let another process run
  - When the register is empty an interrupt is sent, the new current process is stopped
  - Filled up, block, resume, fill up, block, etc.
- DMA: similar to programmed IO, but DMA does all the work.

- Main goals on the design of IO software:
  - Device independence: whatever the support, files are handled the same way
  - Uniform naming: devices organized by type with a name composed of string and number
  - **Error handling: fix error at lowest level possible**
  - Synchronous vs. asynchronous: OS decides if interrupt driven operations look blocking to user programs
  - Buffer: need some temporary space to store data
- Basic plugin design strategy:
  - Similar devices have a common basic set of functionalities
  - OS defines which functionalities should be implemented
  - Use a table of function pointers to interface device driver with the rest of the OS
  - Uniform naming at user level

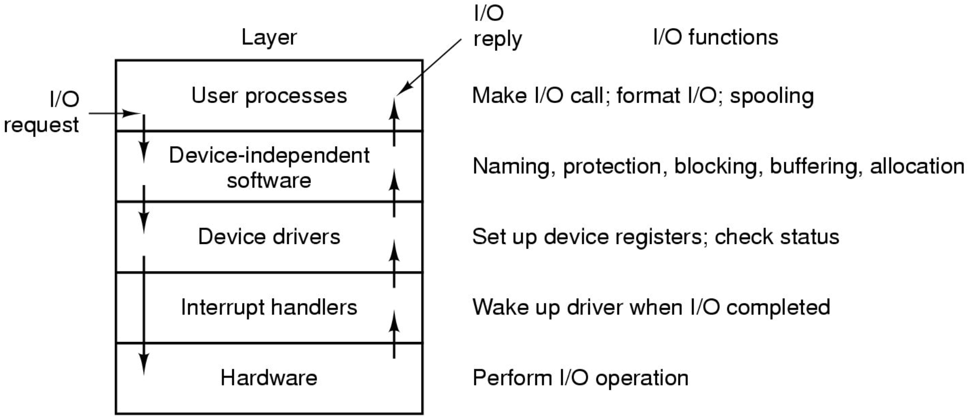
Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Figure 1: Software IO layers

7.4. Key Points

- What are the two main categories of devices?
- Explain the difference between precise and imprecise interrupts
- List three communication strategies
- Why should implementation be done using layers?
- What is a plugin architecture and why is it so important?

8. File Systems

8.1. Requirements

- Goals that need to be achieved:
  - Store large amount of data
  - Long term storage
  - Information shared among multiple processes

8.2. Design

- Contiguous allocation
  - Simple to implement
  - Fast: read a file using a single disk operation
  - Fragmentation when deleting files
- Linked list
  - No fragmentation
  - Slow random access
- File Allocation Table
  - Have a pointer for each disk block
  - Store all the pointers in a table
  - **Save the table in the RAM**
  - Advantage: fast random access
- **Inode**
  - Structure storing:
    - The file attributes
    - Pointers on the blocks where the file is written
  - Advantage: fast, requires little memory
  - What if a large file needs more blocks that can fit in an inode?
    - Point to the next inode
- Journal
  - Keep a journal of the operations:
    - Log the operation to be performed, run it, and erase the log
    - If a crash interrupts an operation, re-run it on next boot
  - Which of the following operations can be safely applied more than once?
    - Remove a file from a directory
    - Release a file inode
      - Released inode may have been allocated to other files
    - Add a file disk blocks to the list of free blocks
      - Double allocation

8.3. Management

- Block size
  - Small block size leads to a waste of time
  - Large block size leads to a waste of space
- Keeping track of the free blocks:
  - Using a linked list: free blocks addresses are stored in a block

    - Using a bitmap: one bit corresponds to one free block
    - Using consecutive free blocks: a starting block and the number of free block following it
- Common problems
    - Block related inconsistency:
        - List of free blocks is missing some blocks: add blocks to list
        - Free blocks appear more than once in list: remove duplicates
        - A block is present in several files: copy block and add it to the files
    - File related inconsistency:
        - Count in inode is higher: set link count to accurate value
        - Count in inode is lower: set link count to accurate value
- Caching
    - Useless to cache inode blocks
    - Dangerous to cache blocks essential to file system consistency
    - Cache partially full blocks that are being written
- Quota: limits the amount of disk space a user can use
- Defragment: Useful for HDDs to improve performance by reducing seek times.
- RAID

8.4. Key Points

- What are the three main goals of a file system?
- Describe a basic disk layout
- Explain the structure of an inode
- Mention three challenges in the design of a file system

9. Security

9.1. Basics on Security

- Categories:
  - Data stolen: confidentiality
  - Data changed: integrity
  - Intrusion: exclusion of outsiders
  - Denial of service: system availability

9.2. Basics on Cryptography

- Two basic encryption strategies:
  - Symmetric: same key used to encrypt an decrypt
  - Asymmetric: many can encrypt but only one can decrypt
- In an OS setup:
  - Symmetric protocols best fit confidentiality
  - Asymmetric protocols best fit authentication
- Hash to check whether data has been changed
- Prove that a user is really who he pretends to be
  - Secret: something the user knows
  - Token: something the user has
  - Challenge-response: the user's ability to correctly respond to a challenge posed by the system
  - Biometrics: something the user is

9.3. Basic Mechanisms

- Basic strategy:
  - Keep the system minimal
  - No new software versions
  - Regularly update the system
  - Install software only from trusted parties
  - Strong passwords or no password
- Advanced strategy:
  - Apply the basic strategy
  - Filter any outgoing network traffic
  - Block any incoming new connection
  - **Keep a checksum of all the files**
  - **Only use encrypted network traffic**
  - **Use containers or virtual machines to run sensitive services**
  - Associate with each program a profile that restricts its capabilities
- Paranoiac strategy:
  - Apply the advanced strategy
  - Encrypt all the disk, including the swap
  - Isolate the computer, no network connection
  - Keep an encrypted checksum of all the files
  - Physically block all the ports, no external device can be connected

9.4. Key Points

- What is security?
- What are the two main types of encryption?
- What are Access Control Lists?
- What is the safest authentication method?
- How safe can a computer be?

10. Multiple Processors Systems

10.1. Multiprocessor Setup

- Three main approaches:
  - Each CPU has its own OS: no sharing, all independent
  - Master-slave multiprocessors: one CPU handles all the requests
  - Symmetric Multi-Processor (SMP): one OS shared by all CPU
- SMP
  - Problems likely to occur:
    - More than one CPUs run the same process
    - A same free memory page is claimed at the same time
- Synchronization
  - A first idea:
    - Lock the memory bus by asserting a special line on the bus
    - The bus is only available to the processor which locked it
    - Perform the necessary memory accesses
    - Unlock the bus
    - New multiprocessor issue: slows down the whole system
      - Solution: do not lock the whole bus, but just the cache
- Multiprocessors cache implementation:
  - The requesting CPU reads the lock and copies it in its cache
  - As long as the lock is unchanged the cached value can be used
  - When the lock is released all the remote copies are invalidated
  - All other CPUs must fetch the updated value
  - Problems
    - A whole cache block needs to be recopied each time
    - Much traffic is generated just to check the lock

10.2. Multiprocessor Scheduling

10.3. Distributed Systems

A Labs

A.1 Lab 1

- `grep -rl` prints the filenames with a match, while `grep -r` also prints the line with the match.
- Use regular expression with `grep -E`.
- `find /etc -type f -name '*netw*'`
- File descriptors
  - 0: Standard input
  - 1: Standard output
  - 2: Standard error
- `>&1` redirects the output to standard output. `2>&1 >` first redirects standard error to standard output and then redirects to a file.
- `$`
  - `$0` is the name of the script.
  - `$1` is the first argument passed to the shell.
  - `$?` is the exit status of the last executed command. For example, 0 is success.
  - `$!` holds the process ID of the last background command.

A.2 Lab 2

- Compilation of Linux kernel
  - Copy old config
  - Tweak new config
  - `make` and `make install`

A.3 Lab 3

- `diff` and `patch`
  - `diff -u original_file modified_file > changes.patch`
  - `patch < changes.patch`
- `rsync`
  - `rsync -avz /source/directory/ user@remote:/destination/directory/`
    - `-a` archive mode
    - `-v` verbose
    - `-z` compress data during transfer
- Regular Expression (Note that some of these are non-POSIX)
  - `abc…` Letters
  - `123…` Digits
  - `\d` Any Digit
  - `\D` Any Non-digit character
  - `.` Any Character
  - `\.` Period
  - `[abc]` Only a, b, or c
  - `[^abc]` Not a, b, nor c
  - `[a-z]` Characters a to z
  - `[0-9]` Numbers 0 to 9

  - `\w` Any Alphanumeric character
  - `\W` Any Non-alphanumeric character
  - `{m}` m Repetitions
  - `{m,n}` m to n Repetitions
  - `*` Zero or more repetitions
  - `+` One or more repetitions
  - `?` Optional character
  - `\s` Any Whitespace
  - `\S` Any Non-whitespace character
  - `^…$` Starts and ends
  - `(…)` Capture Group
  - `(a(bc))` Capture Sub-group
  - `(.*)` Capture all
  - `(abc|def)` Matches abc or def
  - `echo "I like apple" | sed 's/apple/orange/'`
  - `echo -e "5 apple\n15 banana" | awk '$1 > 10'`

A.4 Lab 4

- `gcc -g -o sum sum.c`
- `gdb --args ./sum 1 2`
- `step` `s` can debug inside a function call
- `next` `n` skips over function calls without diving into their details
- `tui enable`
- `print` `p`
- `break` `b`

A.5 Lab 5

- Stages of compilation
  1. Pre-processing
  2. Compilation
  3. Assembly
  4. Linking
- Static libraries
  - `gcc -c list.c -o list.o`
    - `-c` to compile and assemble, but do not link
  - `ar rcs list.a list.o` : create an archive from the object
  - `gcc -o ex2_cli ex2_cli.c -L. -l:list.a -l:middleware.a`
    - `-L.` : Adds the current directory to the list of directories where GCC will look for libraries
    - `-l:` : Specify the exact name of the library file
- Dynamic libraries
  - `gcc -c *.c -fpic`
    - `-fpic` : Generates position-independent code (PIC).
  - `gcc -shared list.o -o list.so`
  - `gcc -o ex2_cli ex2_cli.o list.so middleware.so`

  - `set -x LD_LIBRARY_PATH ./ $LD_LIBRARY_PATH` : appends the current directory

A.6 Lab 6

A.7 Lab 7

A.8 Lab 8

A.8.1 eBPF (TODO)

Compile:

```
make NAME=helloworld obj  # Compile helloworld.o
make NAME=helloworld autoattach  # Compile and attach to tracepoint
```

Map struct:

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);      // Use a hash map
    __type(key, __u32);                   // Key is the user ID (UID)
    __type(value, __u64);                 // Value is the syscall count
    __uint(max_entries, 1024);            // Max entries, adjust as needed
} ecmap SEC(".maps");
```

- Monitor certain parameters of the kernel
- A simple page fault monitor with eBPF

A.8.2 Memory Management

- `mm` directory
  - `kmsan` : Kernel Memory Sanitizer, targets **uninitialized memory** accesses in the kernel.
  - `kasan` : Kernel Address Sanitizer, focuses on a broader range of memory errors like **out-of-bounds accesses and use-after-free**.
  - `kfence` : Kernel Fine-Grained Error Notification, provides a lightweight, low-overhead method to **detect use-after-free errors** with fine-grained error reporting.
  - Shadow memory: It works by **associating each byte of the application's memory with a corresponding entry in a "shadow" region of memory**, which stores metadata about the original memory location.
  - `kmemleak.c` provides a way of detecting possible kernel memory leaks, similar to Valgrind in user-space.
  - `nommu.c`
    - Lack of virtual memory
    - No memory isolation, processes have direct accesses to the entire physical memory
    - No swap
  - `oom_kill.c` : kills processes when the system is seriously out of memory
- MGLRU
  - multiple generations
    - Pages that are not being accessed gets demoted to lower generations
    - Evicted pages that suffer from page fault are being promoted to their original generation
  - both recency and frequency of access are taken into account
- Use `vmalloc` only for **large allocations that do not require contiguous memory** or when `kmalloc` fails to allocate sufficient memory.

- `/dev/shm` : shared memory, typically used for IPC between processes
- Page poisoning: **marking memory with a known value** to help detect memory corruption, such as use-after-free, buffer overflows, or uninitialized memory access.
- Architecture specific files under `arch/`
- `memblock` manages memory regions during early boot period when **the usual kernel memory allocators are not up and running**.

A.9 Lab 9

A.9.1 Memory Management

- Swap is only activated when there's no choice
- Watermarks
  - Below low watermark: swap, or even OOM killer
  - Below medium watermark: aggressively reclaim memory
  - Above high watermark: comfortable

A.9.2 Grandpa's Dice

- `fops`
- `put_user()` moves data to user-space

```
static int my_param = 10;
module_param(my_param, int, S_IRUGO);
```

```
sudo insmod my_module.ko my_param=42
```

A.10 Lab 10

- Copy a module to `/lib/modules/$(uname -r)` to make it known by the kernel
- `sudo depmod -a` to generate `modules.dep` and `map`
  - Scans /lib/modules
  - Generates dependency file
- Use `dbus` to listen to all system events: `dbus-monitor --system`
- `systemd`
  - Service manager
  - Services stored under `/etc/systemd/system/` or `~/.config/systemd/user/`
  - `[Unit]` `[Service]` `[Install]`
  - `sudo systemctl enable my-service.service` to autostart service

B Homework

B.1 Homework 1

- Booting
  1. The computer first runs a power-on self test which ensures the basic functions of the computer is running correctly.
  2. BIOS looks for a bootable device.
  3. BIOS hands over the booting process to the found bootloader.
  4. Bootloader loads the system kernel into memory.

```
sudo useradd username
lscpu && free -h
xxd file3  # hex dump
grep -rlw "nest_lock" --include-"*mutex"
```

B.2 Homework 2

- `man 2 open` : system calls are in section 2
- `man 3 printf` : library calls are in section 3
- `strace` `ltrace`
- `strace -p <pid>` helps pinpointing where the process is stuck in loops
- `printk()` prints to the kernel log
- `SYSCALL_DEFINE0` defines a system call with no arguments, while `SYSCALL_DEFINEx` defines a system call with x arguments.

B.3 Homework 3

- If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?
  - The single-thread process doesn't even have a chance to fork because it is blocked when waiting for input.
- POSIX: Portable Operating System Interface
  - Ensure that software developed for one POSIX-compliant system can run on others

B.4 Homework 4

- `semaphore.h`
  - `sem_init()`
  - `sem_wait()`
    - `sem_trywait()`
  - `sem_post()`
  - `sem_destroy()`

B.5 Homework 5

- A system has two processes and three identical resources. Each process needs a maximum of two resources. Can a deadlock occur?
  - No. A process can apply for the third resource.

B.6 Homework 6

- thrashing: CPU spends the majority of its time swapping data between RAM and disk
- Speculative Execution: Modern CPUs perform speculative execution, meaning they execute instructions before knowing if they are needed
  - Meltdown: Meltdown abuses this feature by triggering speculative execution of code that should not normally have access to privileged memory
  - Spectre: Spectre exploits CPU branch prediction mechanisms to access out-of-bounds memory
- Dirty COW
  - Copy on Write (COW): When a process forks a child process, it doesn't immediately duplicate the memory pages.

B.7 Homework 7

- Tracing subprocesses using eBPF (TODO)

B.8 Homework 8

- thin client: offloads the majority of the processing to a server

C Mid-Term

- What's the difference between user space and kernel space?
- What's the difference between stacks and heaps?
- What are the possible states of a process?
- What's an orphan process? What about zombie process?
  - Orphan Process: Parent has terminated; the system adopts it (init process in Linux).
  - Zombie Process: Finished execution but not yet removed from the process table.
- Can processes ignore signals?
  - Yes, for most signals. But some (like SIGKILL and SIGSTOP) cannot be ignored.
- List some common scheduling algorithms.
- Talk about context change.
- What are the primary means of inter-process communication (IPC)?
  - Pipes
  - Message Queues
  - Shared Memory
  - Semaphores
  - Sockets
- What are the differences between processes, threads and coroutines?
  - Processes: Independent units with their own memory space.
  - Threads: Share memory within a process but have independent execution paths.
  - Coroutines: Cooperative routines within a thread, controlled by the programmer.
- What are the prerequisites for a deadlock to happen?
- How to prevent deadlocks?

Index of Figures