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
- 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
- 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

6.3. Segmentation

7. Labs

7.1. 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.

7.2. Lab 2

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

7.3. 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'`

7.4. 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`

7.5. 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

7.6. Lab 6

7.7. Lab 7

8. Homework

8.1. 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"
```

8.2. 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.

8.3. 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

8.4. Homework 4

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

8.5. 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.

9. 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?