SCUT 2019 Fall Operating System Review

qixuan

December 16, 2019

Abstract

Introduction (chapter 1)
Process and Threads (chapter 2)
Deadlocks (Chapter 6)
Memory Management (Chapter 3)
File Systems (Chapter 4)
Input/Output (Chapter 5)

problem types:

1. Explain the meaning of concepts.
Thread, Page Table, ...

2. Questions.
Each process has three states. Draw a diagram to show the transitions between these three states. If there are multiple processes in the ready state, the scheduler will use a scheduling strategy to select one process to run the CPU. Describe the basic ideas of the following process scheduling strategies: Round robin, Priority scheduling, and shortest job first. (15 points)

•Organize your answers and make the important points clear.
•Explain your answers

Introduction(Chapter 1)

A computer = Hardware + OS + User Programs

The distinct features of OS

Run in **kernel** mode
Has complete access to **all** hardware
Can execute **any** instruction the machine is capable of executing
Huge, complex, long-lived

Two major tasks of OS

1. provide abstractions

OS as an Extended Machine

- programs in machine language level are primitive and difficult to write and interpret
— reading data from floppy disk requires to know 16 commands, 13 input parameters, 23 return status and error fields ...
- OS hides the implementation details and provides user programs with nice and standard abstractions
- A standard "Virtual Machine" across different physical machines.

2. manage resources

OS as a Resource Manager

- A computer contains many resources
— CPU, memory, disk, keyboard, mouse, printer, speakers, microphone, etc.
- OS effectively allocates resources to support multi-users and multi-programming
— Process Manager
— Memory Manager
— File Manager
— Device Manager
- Time multiplexing: Different programs or users take turns using a resource
— Example: CPU; Issues: Who goes next and for how long?
- Space multiplexing: Each program or users gets part of a resource
— Example: memory and disk; Issues: fairness, protection, and so on.

History of OS

The First Generation(1945-55): Vacuum Tubes and Plugboards

- No programming language, all programming works
- No OS
- All problems were numerical calculations

The Second Generation(1955-65): Transistors and Batch Systems

- Languages
— Fortran, Assembly language
- Typical OS
— FMS(Fortran Monitor System), IBSYS
- Problems to be solved
— scientific and engineering calculations

The Third Generation(1965-1980) ICs and Multiprogramming

- The use of ICs(Integrated Circuits)

- IBM System/360

- Operating System: OS/360, MULTICS, UNIX

- **Multiprogramming:** Allow multiple programs run in memory simultaneously. **GOAL: Keep CPUs and I/O devices busy.**

- **Spooling(simultaneous peripheral operation on line)**

— In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output.

—— Example: Instead of writing directly to a printer, outputs are written to the disk. Programs can run to completion faster and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

- **Timesharing:** Allow multiple users to use a computer simultaneously

— A variant of multiprogramming technique

— Each user has an on-line terminal

— The computer system must respond quickly

- CTSS(Compatible Time Sharing System)

- MULTICS(UNIX $->$ MINIX $->$ LINUX)

The Fourth Generation(1980-Present): Personal Computers

- With the development of LSI(Large Scale Integration) circuits, chips, operating systems entered in the personal computer and the workstation age.

- Microprocessor technology evolved to the point that it becomes possible to build desktop computers as powerful as the mainframes of the 1970s

- Windows, Unix, Linux, GUI, MacOS, ...

Computer Hardware Review

The **CPU**, **Memory** and **I/O devices** are connected by a **bus** and communicate with one another over it.

Processor

Fetch-Decode-Execute Cycle

- Fetch: Fetch instruction from memory to **instruction register(IR)**, the **program counter(PC)** points to the next instruction.

- Decode: The instruction presented in the IR is interpreted by the decoder

- Execute: The function of the instruction is performed. If the instruction involves arithmetic or logic, the Arithmetic Logic Unit is utilized.

Registers in Processor

- Each CPU has a set of instructions

- General registers

- Program counter

- Stack pointer

- Program Status Word(PSW)

- When a program stops, the values in all registers are stored

Pipeline

Superscalar CPU

- Holding buffer

Multithreaded and Multicore Chips

Memory

A typical memory hierarchy

Registers Typical access time:1 nsec; Typical capacity: <1 KB

- Registers are a group of circuits used for memory addressing, data operation and processing
- They are made of the same material as the CPU, thus there is no delay in accessing them
- Their capacity is small: 32 x 32 bits for 32-bit CPU, 64 x 64 bits for 64-bit CPU
- Examples: General register, Accumulator register, PC, IR.

Cache Typical access time:2 nsec; Typical capacity: 1 MB

- Cache hit and Cache miss
- L1 cache and L2 cache
- Access to the L1 cache is faster than the L2 cache

Main Memory Typical access time:10 nsec; Typical capacity: 64-512 MB

- RAM(Random Access Memory)
- ROM(Read Only Memory)
- EEPROM(Electrically Erasable PROM)

Magnetic disk Typical access time:10 msec; Typical capacity: 5-50 GB

Magnetic tape Typical access time:100 sec; Typical capacity: 20-100 GB

- Disks: The structure of a disk drive. 512 bytes per sector
- The access speed of disk is low

I/O devices

Device driver

- A software that talks to a controller, giving it commands and accepting responses
- Each OS should has a version of device driver

Three ways to put a driver into kernel

- Relink the kernel and reboot the system(UNIX)
- Make entry file and reboot and load it(WINDOWS)
- Accept and install it without rebooting the system(USB)

Three ways to do input/output

- Busy waiting

— user program $->$ system call $->$ driver $->$ The driver starts the I/O and sits in a tight loop to see if it is done $->$ I/O done $->$ driver return $->$ return to caller

- Interrupt

— user program $->$ system call $->$ driver $->$ The driver starts the I/O and ask it to given an interrupt when it is finished $->$ return $->$ block the caller $->$ run another program

- DMA

— Allows certain hardware subsystems to access main memory independently of the CPU

The Operating System Zoo

Mainframe operating system: OS/360

Server operating systems: UNIX, Windows 2000, Linux

Multiprocessor operating systems: Windows and Linux

Personal computer operating systems: Windows98, 2000, XP, Macintosh, Linux

Real-time operating systems: Hard real-time system: the action absolutely must occur at a certain moment(eg. E-Cos)

Embedded operating systems: PalmOS, Pocket PC for PDA

Smart card operating systems: Java Virtual Machine(JVM)

Process and Threads(Chapter 2)

Process

There are a number of processes running in our computers, but what is a process?

Definition: A key concept in all operating systems is the **process**. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from 0 to some MAXIMUM, which is the process can read and write. The address space contains the **executable program**, the **program's data** and its **stack**. Also associated with each process is a set of resources, commonly including **registers**(including the **program counter** and **stack pointer**), **a list of open files**, **outstanding alarms**, **lists of related processes**, and **all the other information needed to run the program**. A process is fundamentally a container that holds all the information needed to run a program.

Essentially:

- a program in execution

- a container that holds all the information needed to run a program

Process Table

- Stores information about processes(UID, PID, GID)
- Process control block, PCB

Process Tree

- A created two child processes B and C
- B created three child processes D, E and F

The communication to cooperate and synchronize processes is called **inter-process communication(IPC)**

Address Space

- One process can't even see another's address space

- Same pointer address in different processes point to different memory

- Can change mapping dynamically

- Question: 32-bit processors have 2^32 possible addresses, while 64-bit processors have a 48-bit address space, why?

- Answer: 2^{32} bit = 4 GB, 2^64 bit = much more which we can't use all of them yet. At the moment 64-bits is quite a lot for addressing real physical memory. It won't make sense to use up precious motherboard real estate to run wires that will always have 0's.

Files

A **file system** provides users with a nice, clean abstract model of device-independent files

A **directory** is used to group files together

A **path name** can specify the location of a file

A **root directory** is the top of the directory hierarchy. In Unix/Linux, / is the root directory

Each process has a **current working directory(PWD)**

File Hierarchies

File Descriptor: a small integer(int = 4 bytes = 32 bit)

Mount a file system in removable media

- Before mounting, files on floppy are inaccessible. After mounting floppy on b, files on floppy are part of file hierarchy

Protection

Important information: emails, documents, etc.

Task of OS

- Ensure files are accessible to authorized users

Files in UNIX are protected by a 9-bit binary protection code

- The protection code consists of three 3-bit fields, one for owner, one for group and one for others

-Example: rwx r-x –x

Shell

A shell is a user interface for access to an operating system's services(sh, bash, zsh, ...)

Many personal computers use a GUI(KDE, Gnome, ...)

System Calls(#include<unistd.h>)

Definition: The interface between a running program and the OS.

System Calls For Process Management

- pid = fork(): Create a child process identical to the parent

- pid = waitpid(pid, &statloc, options): Wait for a child to terminate(to move the parent process off the ready queue until the termination of the child)

- s = execve(name, argv, environp): Replace a process' core image(to replace the process' memory space with a new program(used after fork))

- exit(status): Terminate process execution and return status(used when a process is finished)

System Calls For Directory Management

- s = mkdir(name, mode): Create a new directory
- s = rmdir(name): Remove an empty directory
- s = link(name1, name2): Create a new entry, name2, pointing to name1
- s = unlink(name): Remove a directory entry
- s = mount(special, name, flag): Mount a file system(mount("/dev/fd0", "/mnt", 0): mount "/dev/fd0" to "/mnt", flag is 0.)
- s = umount(special): Unmount a file system

System Calls For Miscellaneous Tasks

- s = chdir(dirname): Change the working directory
- s = chmod(name, mode): Change a file's protection bits
- s = kill(pid, signal): Send a signal to a process
- seconds = time(&seconds): Get the elapsed time since Jan. 1, 1970.

System Calls(Win32)

- A window program is usually **event-driven**
- There is almost a **one-to-one relationship** between the system calls of Unix and Windows
- Microsoft has defined **Win32 API(application program interface)** to get OS services
- The **number** of Win32 API calls **is** extremely **large**, **not all of them run in kernel mode**

Operating System Structure

Jigsaw reading

- Monolithic System: All operating system operations are put into a single file. The operating system is a collection of procedures, each of which can call any of the others. (e.g., Linux, windows)

- Layered System: The operating system is organized as a hierarchy of layers of processes.

Structure of the "THE" operating system, which was built by E. W. Dijkstra and his students.

——Layer 5: The operator
——Layer 4: User programs
——Layer 3: Input/Output management
——Layer 2: Operator-Process communication
——Layer 1: Memory and drum management
——Layer 0: Processor allocation and multi-programming

- Microkernels

Split the OS into modules, only one runs in kernel mode and the rest run in user mode; (a lot communication)

Put as little as possible into kernel model to improve the reliability of the system.

Examples: MINIX 3

- Client-Server Model

Contains two classes of processes, the servers and the clients. Communication between servers and clients is done by **message passing**. It is an abstraction that can be used for a single machine or a network of machines.(Clients obtain service by sending messages to server processes)

- Virtual Machine

The operating system is a timesharing system that provides multiprogramming;VM monitors are exact copies of the bare hardware, e.g., VM/370, JVM (Java Virtual Machine)

Problems

1. What are the major tasks of an OS?

- provide abstractions

- manage resources

2. What is multi-programming?

- Allow multiple programs run in memory simultaneously

3. A computer has a pipeline with 4 stages. Each stage takes the same time to do its work, namely 1 nsec. How many instructions per second can this machine execute?

- $10^9 - 3$

4. What is the purpose of a "system call" in an OS?

- To build communication between User Programs(User) and OS(Kernel)

5. What are monolithic system and microkernel?

Monolithic system puts all system calls into a single file; Microkernel system has multiple kernels but only one of them run in kernel mode, others run in user mode, the kernel mode kernel is minimal to enhance the stability of the system.

6. What is a process

A process is a program in execution and also a container holds all information needed to run a program.

Multi-programming

The CPU switchs from process to process quickly, running each of tens or hundreds of milliseconds

At anytime, the CPU is running only one process

Process Creation

Events that cause process creation

1. System initialization
2. Created by a running process
3. User request to create a new process
4. Initiation of a batch job

Foreground processes: processes that interact with users and perform work for them

Background processes that handle some incoming requests are called **daemons**

How to write a program to execute another program?

Unix/Linux: Process Creation

Use **fork()** system call to create a new process. The **execve()** system call can be used to load a new program.

Windows: Process Creation

Use **CreateProcess** to handles both creation and loading the correct program into the new process.

Process Termination

Conditions that cause process termination:
- Normal exit(voluntary)
— "Exit" in UNIX and "ExitProcess" in Windows
- Error exit(voluntary)
— Example: input file is not exist
- Fatal error(involuntary)
— Example: referencing nonexistent memory
- Killed by another process(involuntary)
— "Kill" in UNIX and "TerminateProcess" in Windows

Process States

Running: using the CPU at that instant.

Blocked: unable to run until some external event happens.

Ready: runnable, temporary stopped to let another process run.

Running $->$ **Blocked** $->$ **Ready**; **Running** $->$ **Ready**

OS: the lowest layer, handles interrupts, scheduling

Above that layer are sequential processes(eg. user processes, terminal processes)

Implementation of Processes

The OS maintains a **Process Table** with one entry(called a **process control block(PCB)**) for each process.

When a context switch occurs between processes P1 and P2, the current state of the RUNNING process, say P1, is saved in the PCB for process P1 and the state of a READY process, say P2, is restored from the PCB for process P2 to the CPU registers, etc. Then, process P2 begins RUNNING.

Pseudo- parallelism: the rapid switching between processes gives the illusion of true parallelism and is called pseudo-parallelism.

Modeling Multiprogramming

Suppose a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once. The CPU utilization can be obtained by:

$$\text{CPU utilization} = 1 - p^n$$

Thread

Definition: A thread of execution is the smallest sequence of programmed instructions that **can be managed independently by a scheduler**, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases **a thread is a component of a process**. **Multiple threads can exist within one process, executing concurrently and sharing resources** such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Why do we need threads?

Responsiveness: multiple activities can be done at the same time.

Resource Sharing: threads share the memory and the resources of the process.

Economy: threads are easy to create and destroy.

Utilization of MP (multiprocessor) Architectures: threads are useful on multiple CPU systems.

Thread Model

process: ((code, data, files), ((registers, stack), thread))

Some items are shared by all threads in a process, while some items are private to each thread

In Linux: pthread_create()

In Windows: CreateThread()

Thread v.s. Process

A thread – **lightweight process**, a basic unit of CPU utilization.

It comprises a thread ID, a program counter, a register set, and a stack.

A traditional(**heavyweight**) process has a single thread of control.

If the process has multiple threads of control, it can do more than one task at a time. This situation is called **multithreading**.

Process: used to group resources together;

Thread: the entity scheduled for execution on the CPU.

POSIX Threads

POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system.

The need for standardization arose because enterprises using computers wanted to be able to develop programs that could be moved among different manufacturer's computer systems directly.

- Pthread_create: Create a new thread
- Pthread_exit: Terminate the calling thread
- Pthread_join: Wait for a specific thread to exit
- Pthread_yield: Release the CPU to let another thread run
- Pthread_attr_init: Create and initialize a thread's attribute structure
- Pthread_attr_destory: Remove a thread's attribute structure

Implementation of threads: Threads Table

Jigsaw reading

Implementing threads in user space

- Advantages: fast, flexible, scalable

- Drawbacks: Blocking blocks all threads in a process; No thread-level parallelism on multiprocessor

Implementing threads in kernel

- Advantages: Blocking blocks only the appropriate thread in a process; Support thread-level parallelism on multiprocessor;

- Drawbacks: Slow thread management; large thread tables

Hybrid Implementation

Multiplex user-level threads on kernel threads. Kernel is only aware of the kernel-level threads and schedule those.

More flexible, but more complicated.

<u>Problems</u>

1. How many states can a process have?

Three; Running, Blocked, Ready.

2. How to create and terminate a process in Windows?

CreateProcess(); TerminateProcess()

3. What is PCB?

Process Control Block

4. What is Context Switch?

Switch between processes, store the current RUNNING process into corresponding PCB and set its process state as Ready, restore a READY process from PCB to CPU register and set its process state as RUNNING.

5. What are the differences between process and thread?

A thread is a component of a process(light-weighted process)

Process: used to group resources together.

Threads: the entity scheduled for execution on the CPU.

Inter Process Communication(IPC)

How to pass information among processes?

How to make sure two or more processes do not get into each other's way when engaging in critical activities.

Proper sequencing when dependencies are present.

Race Conditions

Race conditions: situations in which several processes access shared data and the final result depends on the order of operations.

With increasing parallelism due to increasing number of cores, race condition are becoming more common.

Critical Regions

Key idea to avoid race condition:**prohibit more than one process from reading and writing the shared data at the same time.**

Critical Region: part of the program where the share memory is accessed

Four conditions to support a good solution:

- No two processes may be simultaneously in critical region

- No assumption made about speeds or numbers of CPUs
- No process running outside its critical region may block another process
- No process must wait forever to enter its critical region

Mutual Exclusion Solution using Critical Regions

Disabling Interrupts

- The CPU is only switch from process to process when clock or other interrupts happen; Hence, by disabling all interrupts, the CPU will not be switched to another process.

- However, it is unwise to allow user processes to disable interrupts.

— One thread may never turn on interrupt;

— Problem still exist for multiprocessor systems.

Lock Variable

```
shared int lock = 0;
/* entry_code: execute before entering
critical section */
while (lock != 0) ; // do nothing
lock = 1;
- critical section -
/* exit_code: execute after leaving
critical section */
lock = 0;
```

Problem?: If a context switch occurs after one process executes the while statement, but before setting lock = 1, then two (or more) processes may be able to enter their critical sections at the same time.

Strict Alternation

Since the processes must strictly alternate entering their critical sections, a process wanting to enter its critical section twice will be blocked until the other process decides to enter (and leave) its critical section.

Peterson's Solution

This solution satisfies all 4 properties of a good solution. Unfortunately, this solution involves **busy waiting** in the while loop.

Hardware solution: Test-and-Set Locks (TSL)

The hardware must support a special instruction, TSL, which does two things in a single atomic action:

(a) copy a value in memory (flag) to a CPU register

(b) set flag to 1.

Mutual Exclusion with Busy Waiting

BUSY-WAITING: a process executing the entry code will sit in a tight loop using up CPU cycles, testing some condition over and over, until it becomes true.

Busy-waiting may lead to the **priority-inversion problem**.

Sleep and Wakeup

From busy waiting to blocking...

Sleep: is a system call that causes the caller to block, that is, be suspended until another process wakes it up.

Wakeup: has one parameter, the process to be awakened.

Producer-Consumer Problem: Consider a circular buffer that can hold N items. Producers add items to the buffer and Consumers remove items from the buffer. The Producer-Consumer Problem is to restrict access to the buffer.

Semaphores [E.W. Dijkstra, 1965]

A SEMAPHORE, S, is a structure consisting of two parts:
(a) an integer counter, COUNT
(b) a queue of pids of blocked processes, Q

That is,

```
struct sem_struct {
    int count;
    queue Q;
} semaphore;

semaphore S;
```

There are two operations on semaphores, **UP** and **DOWN (PV)**. These operations must be **executed atomically** (that is in mutual exclusion). Suppose that P is the process making the system call. The operations are defined as follows:

```
DOWN(S):
if (S.count > 0)
    S.count = S.count - 1;
else
    block(P); that is,
(a) enqueue the pid of P in S.Q,
(b) block process P (remove the pid from the ready queue)
(c) pass control to the scheduler.

UP(S):
if (S.Q is nonempty)
    wakeup(P) for some process P in S.Q; that is,
(a) remove a pid from S.Q (the pid of P),
(b) put the pid in the ready queue, and
(c) pass control to the scheduler.
else
    S.count = S.count + 1;
```

Mutual Exclusion Solution

Semaphores do not require busy-waiting, instead they involve BLOCKING.

```
semaphore mutex = 1; // set mutex.count = 1
DOWN(mutex);
    - critical section -
UP(mutex);
```

A **mutex** is a semaphore that can be in one of two states: unlocked (0) or locked (1).

Using Semaphores

Process Synchronization: Suppose we have 4 processes: A, B, C, and D. A must finish executing before B start. B and C must finish executing before D starts. How many semaphores should be used to achieve the above goal?

Answer: Three.

```
Process A:
- do work of A
UP(S1); /* Let B or C start */

Process B:
DOWN(S1); /* Block until A is finished */
- do work of B
UP(S2);

Process C:
- do work of C
UP(S3);

Process D:
DOWN(S2);
DOWN(S3);
- do work of D
```

<u>Problems:</u>

1. What is Race Condition?

Situations that multiple processes access shared date at the same time and the final result depends on the order of operations.

2. What is Critical Region?

Part of program where the shared memory is accessed.

3. What is Busy Waiting

A process executing the entry code will sit in a tight loop using up CPU cycles, testing some condition over and over, until it becomes true.

4. What is Semaphore?

A data structure consisting of 2 parts:
(a) an integer counter, COUNT
(b) an queue of pids of blocked processes, Q

5. What is Mutex?

A Semaphore whose COUNT only has two values: 0(unlocked) and 1(locked).

Process Behavior

Nearly all processes alternate bursts of computing with I/O requests.

CPU-bound: Spend most of the time computing.

IO-bound: Spend most of the time waiting for I/O.

Process Scheduling

Scheduler: A part of the operating system that decides which process is to be run next.

Scheduling Algorithm: a policy used by the scheduler to make that decision.

To make sure that no process runs too long, a clock is used to cause a periodic interrupt (usually around 50-60 Hz); that is, about every 20 msec.

Preemptive Scheduling: allows processes that are runnable to be temporarily suspended so that other processes can have a chance to use the CPU.

When to Schedule

- When a new process is created;
- When a process exist;
- When a process blocks on I/O;
- When an I/O interrupt occurs (e.g., clock interrupt).

Scheduling Algorithm Goals

- Fairness - each process gets its fair share of time with the CPU.

- Efficiency - keep the CPU busy doing productive work.

- Response Time - minimize the response time for interactive users.

- Turnaround Time - minimize the average time from a batch job being submitted until it is completed.

- Throughput - maximize the number of jobs processed per hour.

First-Come, First-Served (FCFS) Scheduling

- Suppose that the processes arrive in the order: P_1, P_2, P_3
- Waiting time for P_1, P_2, and P_3 = 0,24,27
- Average waiting time = (0+24+27)/3 = 17

Suppose that the processes arrive in the order P_2, P_3, P_1

- Waiting time:$P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.

Shortest-Job-First (SJF) Scheduling

Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

Drawback? The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

Two schemes:

• nonpreemptive – once CPU given to the process it cannot be preempted until it completes its CPU burst.

• **preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First (SRTF)**.

SJF is optimal – gives minimum average waiting time for a given set of processes.

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. **No process waits more than (n-1)q time units.**

- Performance

— q large $->$ FIFO

— q small $->$ q must be large with respect to context switch, otherwise overhead is too high.

(Example of RR with Time Quantum = 20, The **Gantt chart** is ..., Typically, higher average turnaround than SJF, but better response.)

Priority Scheduling

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority

(smallest integer == highest priority).

Preemptive

nonpreemptive

- SJF is a priority scheduling where priority is the predicted next CPU burst time.

- Problem: **Starvation – low priority processes may never execute.**

Solution: **Aging – as time progresses increase the priority of the process.**

Mean process turnaround time == Average process waiting time

Scheduling in Interactive Systems

More Scheduling

Shortest Process Next

- SJF can be used in an interactive environment by estimating the runtime based on past behavior. Aging is a method used to estimate runtime by taking a weighted average of the current runtime and the previous estimate.

Guaranteed Scheduling

- Suppose 1/n of the CPU cycles.

- Compute ratio = actual CPU time consumed / CPU time entitled

- Run the process with the lowest ratio

Lottery Scheduling

- Give processes lottery tickets for various system resources

- When a scheduling decision is made, a lottery ticket is chosen, and the process holding that ticket gets the resource.

Fair-Share Scheduling

- Take into account how many processes a user owns.

- Example: User 1 – A, B, C, D and Use 2 – E

- Round-robin: ABCDEABCDE...

- Fair-Share: if use 1 is entitled to the same CPU time as user 2

AEBECEDEAEBECEDE...

Thread Scheduling

- The process scheduling algorithms can also be used in thread scheduling. In practice, round-robin and priority scheduling are commonly used.
- User-level and kernel-level threads

— A major difference between user-level threads and kernel-level threads is the performance.

— User-level threads can employ an application-specific thread scheduler.

Problems:

1. What is IO-bound process?

Most of CPU time used on waiting for I/O.

2. When to schedule processes?

- When a new process created
- When a process exist
- When a process blocked on I/O
- When an I/O interrupt occurs(eg. clock interrupt)

3. What are the goals of Scheduling algorithms?

Fairness, Efficiency, Response Time, Turnaround Time, Throughput.

4. What is the drawback of the SJF algorithm?

Hard to know the burst time in advance.

5. What is RR scheduling?

attach each process with a quantum time q. Once q elapsed, add current process into the end of the Ready queue and context switch to the next process in the Ready queue.

Classical IPC Problems

Readers and Writers Problem

Sleeping Barber Problem

Dining Philosophers Problem

Solution: semaphore and mutex

Solution: Monitors

- A **monitor** is a collection of procedures, variables, and data structures that **can only be accessed by one process at a time** (for the purpose of mutual exclusion).
- To allow a process to wait within the monitor, a **condition variable** must be declared, as **condition x, y**;
- Condition variable can only be used with the operations **wait** and **signal** (for the purpose of synchronization).

— **x.wait();** means that the process invoking this operation is suspended until another process invokes

— **x.signal();** operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

Advantages: **Ease of programming.**

Disadvantages:

• Monitors are a programming language concept, so they are difficult to add to an existing language; e.g., how can a compiler know which procedures were in monitor?

• Monitors are **too expensive to implement**.

Comparing monitor with Semaphores

Differences:

1. When a process executes a P operation, it does not necessarily block that process because the counting semaphore may be greater than zero. In contrast, when a wait statement is executed, it always blocks the process.

2. When a task executes a V operation on a semaphore, it either unblocks a task waiting on that semaphore or increments the semaphore counter if there is no task to unlock. On the other hand, if a process executes a signal statement when there is no other process to unblock, there is no effect on the condition variable.

3. Another difference between semaphores and monitors is that users awaken by a V operation can resume execution without delay. Contrarily, users awaken by a signal operation are restarted only when the monitor is unlocked.

Message Passing

Assign each process a unique address such as addr. Then, send messages directly to the process, e.g., signals in UNIX.

- send(addr, msg)
- recv(addr, msg)

Message passing is commonly used in parallel programming systems. e.g., MPI (Message-Passing Interface).

Message passing is applicable for:

- processes inside the same computer.
- processes in a networked/distributed system.

Synchronization in message passing

Message passing may be blocking or non-blocking.

Blocking is considered **synchronous**

• **Blocking send** has the sender block until the message is received

• **Blocking receive** has the receiver block until a message is available

Non-blocking is considered **asynchronous**

• **Non-blocking send** has the sender send the message and continue

• **Non-blocking receive** has the receiver receive a valid message or null

Sender: it is more natural not to be blocked after issuing send:

• can send several messages to multiple destinations.

• but sender usually expect acknowledgment of message receipt (in case receiver fails).

Receiver: it is more natural to be blocked after issuing receive:

• the receiver usually needs the information before proceeding.

• but could be blocked indefinitely if sender process fails to send.

Other methods are offered, e.g., blocking send, blocking receive:

- both are blocked until the message is received.
- provides tight synchronization (**rendezvous**).

So, there are three combinations that make sense:
(1) Blocking send, Blocking receive;
(2) Nonblocking send, Nonblocking receive;
(3) **Nonblocking send, Blocking receive – most popular.**
Problems:
1. What is condition variable in monitor?
condition x, y: condition variable can only be used with operations signal and wait(for the purpose of synchronization).
2. What are the two operation in monitor?
signal, wait.

Computer Deadlock(Chapter 6)

What is deadlock? eg.
You can't get a job without experience;
You can't get experience without a job.

Definition: A set of processes is in a deadlock state when every process in the set is waiting for a resource that can only be released by another process in the set.

Resources
A **resource** is anything that can be acquired, used, and released over the course of time.
Two types of Resources:
Preemptable resources
- can be taken away from a process with no ill effects (e.g. memory)
Nonpreemptable resources
- will cause the process to fail if taken away (e.g. CD recorder)
Potential deadlocks that involve Preemptable resources can usually be resolved by reallocating resources from one process to another.

Resource Usage
Sequence of events required to use a resource
- request the resource
- use the resource
- release the resource
Must wait if request is denied
- requesting process may be blocked
- may fail with error code

Resource Acquisition
- Associate a semaphore with each resource

- Code with a potential deadlock

Conditions for Deadlocks
Mutual exclusion
- Resources are held by processes in a non-sharable (exclusive) mode.
Hold and Wait
- A process holds a resource while waiting for another resource.
No Preemption
- There is only voluntary release of a resource - nobody else can make a process give up a resource.
Circular Wait
- Process A waits for Process B waits for Process C .... waits for Process A.
ALL of these four conditions **must happen simultaneously** for a deadlock to occur.

Deadlock Modeling (Resource with single instance)
Modeled with **directed graphs**
(a) resource R assigned to process A
(b) process B is requesting/waiting for resource S
Resource-Allocation Graph
Deadlock Modeling (Resource with multiple instances)
- Basic Fact
— no cycles $->$ no deadlock
— if graph contains a cycle
—— if only one instance per resource type, then deadlock
—— if several instances per resource type, possibly of deadlock

The Ostrich Algorithm
Pretend that there is no problem
Reasonable if
1. deadlocks occur very rarely
2. cost of prevention is high

Detection with One Resource of Each Type
- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock
- Many algorithms for detecting cycles in directed graphs.
Detecting Cycles in Directed Graphs
Detection with Multiple Resources of Each Type
Data structures needed by deadlock detection algorithm:
Resources in existence, Resources available, Current allocation matrix, Request matrix.

$$\sum C_{ij} + A_j = E_j$$

Deadlock Detection Algorithm

1. Recovery look for an unmark process P_i s.t. the i-th row of R is less than A. (P_i 's request can be satisfied)

2. If such a process is found, add the i-th row of C to A, mark the process and go back to step 1. (When P_i completes, its resources will become available).

3. If no such process exist, the algorithm terminates. The unmarked process, if any, are deadlock.

Recovery from Deadlock

Recovery through preemption
- take a resource from some other processes
- depends on the nature of the resource

Recovery through rollback
- checkpoint a process periodically
- use this saved state
- restart the process if it is found deadlocked

Recovery through killing processes
- crudest but simplest way to break a deadlock
- kill one of the processes in the deadlock cycle
- the other processes get its resources
- choose process that can be rerun from the beginning

Problems:

1. What is deadlock?

A set of processes is in a deadlock state when every process on the set is waiting for a resource that can only be released by another process in the set.

2. what is resource in computer?

A resource is anything can be acquired, used and released over the course of time.

3. What are the four conditions for a deadlock to occur?

Mutual Exclusion
Hold and Wait
No preemption
Circular Wait

4. How to detect deadlock

Using direct graph to model deadlock, use cycle in the graph to detect deadlock(single resource instance, multiple resource instances)

5. How to recovery from deadlock?

Recovery from preemption
- Reallocate resources

Recovery from rollback
- checkpoint

Recovery from killing process

Can we avoid deadlock?

Motivation: In most systems, resources are requested one at a time. It is desirable that the system can decide whether granting a resource is safe or not.

Question: Is there an algorithm to make the correct decision?

Answer: Yes, we can avoid deadlock, but only if **certain information** is available in advance.

Resource Trajectories

Safe and Unsafe States

A state is **safe** if there is some scheduling order in which every process **can run to completion** even if all of them suddenly **request their maximum number of resources**.

The **difference** between a safe state and an unsafe state is that: **from a safe state the system can guarantee that all processes will finish**; But **from an unsafe state, no such guarantee can be given**.

The Banker's Algorithm for a Single Resource

Check to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out.

Banker's Algorithm

Step 1: Look for a new row in R which is smaller than A. If no such row exists the system will eventually deadlock ==> not safe.

Step 2: If such a row exists, the process may finish. mark that process (row) as terminate and add all of its resources to A.

Step 3: Repeat Steps 1 and 2 until all rows are marked ==> safe state or exist row not marked ==> not safe.

Banker's Algorithm for Multiple Resources

Banker's Algorithm Summary

In theory : the algorithm is wonderful.

In practice: it is essentially useless because processes rarely know in advance what their maximum resource needs will be.

Thus, in practice, **few** systems use the banker's algorithm for avoiding deadlocks.

Starvation

Definition: a process is perpetually denied necessary resources to process its work.

Example: A system always chooses the process with the shortest file to execute. If there is a constant stream of processes with short files, the process with long file will never be executed.

Starvation vs Deadlock

Starvation: thread waits indefinitely

Deadlock: circular waiting for resource.

What are the differences between them?

1. deadlock is a subset of starvation

2. Starvation can end , but deadlock can't end without external intervention.

Some notes of Deadlock

Deadlock is not always deterministic.

- Deadlock won't always happen with this code.

- Have to have exactly the right time;

Deadlock Detection, Deadlock Avoidance, and Deadlock Prevention

Deadlock Detection: To make sure whether there is a deadlock now.

Deadlock Avoidance: to ensure the system won't enter an unsafe state. The system dynamically considers every request and decides whether it is safe to grant it at this point.

Deadlock Prevention: to ensure that at least one of the necessary conditions for deadlock can never hold.

Attacking the Mutual Exclusion Condition

No resource were assigned exclusively to a single process.

Problem?: This method is generally impossible, because the mutual-exclusion condition must hold for non- sharable resources. e.g., a printer can not be shared by multiple processes simultaneously.

Attacking the Hold and Wait Condition

Require processes to request resources before starting

- a process never has to wait for what it needs e.g. In the dining philosophers' problem, each philosopher is required to pick up both forks at the same time. If he fails, he has to release the fork(s) (if any) he has acquired.

Problems?: It is difficult to know required resources at start of run

Attacking the No Preemption Condition

If a process is holding some resources and requests another resource that cannot be allocated to it, then all resources are released.

Problem?: The method can be applied to resources whose state can be save and restored later, e.g., memory. It cannot be applied to resources such as printers.

Attacking the Circular Wait Condition

A process is entitled only to a single resource at any moment.

Impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Other Issues: Two-Phase Locking

Phase One

- process tries to lock all records it needs, one at a time
- if needed record found locked, restart
- (no real work done in phase one)

If phase one succeeds, it starts second phase,

- performing updates
- releasing locks

Similar to requesting all resources at once

Non-resource Deadlocks

Communication Deadlock

- E.g., each is waiting for the other to do some task. Can happen with semaphores

Problems:

1. The banker's algorithm is being run in a system with m resource classes and n processes? In the limit of large m and n, the number of operations that must be performed to check a state for safety is proportional to $m^a n^b$. What are the values of a and b?

Comparing a row in the matrix to the vector of available resources takes m operations. This step must be repeated on the order of n times to find a process that can finish and be marked as done. Thus, marking a process takes on the order of mn steps. Repeating the algorithm for all n processes means that the number of steps is then mn^2

2. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

3. A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

n <= 5

4. Consider p processes each needing a maximum of m resources and a total of r resources available. What condition must hold to make the system deadlock free?

p - 1 + m <= r

5. Consider a system consisting of m resources of the same type being shared by n processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:

a) The maximum need of each process is between 1 resource and m resources.

b) The sum of all maximum needs is less than m + n.

Once a maximum need of a process reachs m, the others' maximum needs must be 1.

Memory Management(Chapter 3)

Memory manager handles the memory hierarchy

File System(Chapter 4)

Input/Output(Chapter 5)